Challenge URL: /dvwa/vulnerabilities/sqli/
Objective: There are 5 users in the database, with IDs from 1 to 5. Steal their passwords via SQLi.
Tools needed: john

Did you remember to read this section's README?

The Guide

Examining the Form

Okay, we see a form that takes user input in the form of a "User ID".

The challenge told us there are five users with user IDs 1, 2, 3, 4, and 5. Let's test this form by entering one of these IDs.

Okay, the form takes our input and returns the user's ID, first name, and last name. What happens if we test for a SQL injection (AKA a "SQLi") by inputting the time-honored classic, a single quote character (`'`)?

An error! You'll see something like this when user input isn't correctly sanitized before being used in a server-side SQL query. Not only that, we learn from the error that the web server is using MariaDB. That should be enough for us to get started.

Crafting a SQL Injection Exploit

Now that we know there's a potential SQLi, let's see what we can do with it. Reviewing the MariaDB documentation tells us that the special character `%` functions as a wildcard. Let's input that next and see what happens.

Interesting, nothing happens. We'll need to find away to break out of the query the code wants us to make. Let's tweak our query a little bit:

```
%' or '0'='0
```

The second part evaluates if zero is equal to zero. I really hope I don't have to explain why that's true! Since we combine our wildcard and this comparison with a logical OR, this query will always evaluate to "true", and we should get some form of output:

Note how our query replaces the ID field for all users, but it returns all users anyway. Honestly, we don't care much about the ID field. We know we can easily verify who has what ID by inputting user IDs the challenge gave us into the form field, like during our intial testing. More importantly, we just broke out of the intended query and now have an avenue to do whatever we want!

Let's start seeing where DVWA keeps the passwords. If we're lucky, DVWA will store them in the "password" column - seems logical, right? Let's build that query and try it out.

```
%' or '0'='0' union select * from password #
```

Note the addition of the `#` character at the end of the statement. That character starts a comment. It truncates the query and ensures we have full control over our future queries.

No luck on that specific column, but it does confirm the database name of "dvwa", which helps.

Reviewing the MariaDB documentation tells us more useful info. It appears that we can pull all the column names from a MariaDB built-in table called "information_schema.COLUMNS". If we can do that, we may be able to pull the correct name of the column where the app stores passwords!

Let's redo our query to account for this new information:

```
%' or '0'='0' union select COLUMN_NAME from information_schema.COLUMNS #
```

Hm, an odd error. If you look at the docs again, it's basically telling us the amount of columns we're trying to pull don't equal the amount of columns we're displaying. Remember, in our previous testing, we're overwriting the "ID" output with our query. We still display "First name" and "Surname", so maybe we need to try pulling two columns? Let's try again:

```
%' or '0'='0' union select TABLE_NAME, COLUMN_NAME from information_schema.COLUMNS #
```

Progress! We get output again, and by God we get all of the output. We definitely need to clean this up somehow. It's not as hard as you think. Based on the order of our `SELECT` clause above (`TABLE_NAME` , then `COLUMN_NAME`), we know that the info with a label of "First name" is our `TABLE_NAME` and "Surname" is our `COLUMN_NAME` . Let's filter on (what we logically assume is) our proper table name of "users".

```
%' or '0'='0' union select TABLE_NAME, COLUMN_NAME from information_schema.COLUMNS
where TABLE_NAME = 'users' #
```

Partway down this output, we see what we want:

We now know the usernames are stored in a column named "user", and passwords are stored in the column "password". Both exist in the table "users" in the "dvwa" database. Ha, turns out we weren't too far off in the beginning! Let's rebuild our query with this new info.

```
%' or '0'='0' union select user, password from dvwa.users #
```

We've got all the hashes! We only need to break them and we're done.

Cracking the Hashes

For offline password cracking, I prefer the tool "John the Ripper" (AKA "john"). We can start using it by running `john` in a Kali terminal.

The only useful parameters right now are:

- `--wordlist=[wordlist_file]` : We already had the "rockyou" wordlist from Challenge 1. We'll be using that again.
- `--format=[hash_format]` : There's a useful site available here that accepts a hash as input and outputs the algorithm believed to create it. john also has similar functionality, but I don't think it's reliable enough to use consistently. Plopping our hashes here tells us they're likely created with the MD5 algorithm. The john equivalent is "Raw-MD5".

I've stored my hashes in a text file called "hashes.txt" in my working directory. Let's give it a shot!

```
john --wordlist=rockyou.txt --format=Raw-MD5 hashes.txt
```

Perfect! If we needed help tying usernames and passwords together, we can add these cracked passwords to a wordlist. Adding the usernames we discovered in our last SQL query to a second wordlist, we can use Hydra to brute force them like we did in Challenge 1.

That was a little rougher than expected, but hey. Challenge complete!