**SAD Lab**

**EXPERIMENT NO. 7**

**Aim**: Data Validation Experiment.

**Theory**:

1. What is SQL Injection?

**SQL Injection** (SQLi) is a type of an injection attack that makes it possible to execute malicious SQL statements. These statements control a database server behind a web application. Attackers can use SQL Injection vulnerabilities to bypass application security measures. They can go around authentication and authorization of a web page or web application and retrieve the content of the entire SQL database. They can also use SQL Injection to add, modify, and delete records in the database.

An SQL Injection vulnerability may affect any website or web application that uses an SQL database such as MySQL, Oracle, SQL Server, or others. Criminals may use it to gain unauthorized access to your sensitive data: customer information, personal data, trade secrets, intellectual property, and more. SQL Injection attacks are one of the oldest, most prevalent, and most dangerous web application vulnerabilities.

1. How and Why is an SQL Injection Attack performed?

To make an SQL Injection attack, an attacker must first find vulnerable user inputs within the web page or web application. A web page or web application that has an SQL Injection vulnerability uses such user input directly in an SQL query. The attacker can create input content. Such content is often called a malicious payload and is the key part of the attack. After the attacker sends this content, malicious SQL commands are executed in the database.

SQL is a query language that was designed to manage data stored in relational databases. You can use it to access, modify, and delete data. Many web applications and websites store all the data in SQL databases. In some cases, you can also use SQL commands to run operating system commands. Therefore, a successful SQL Injection attack can have very serious consequences.

1. Attackers can use SQL Injections to find the credentials of other users in the database. They can then impersonate these users. The impersonated user may be a database administrator with all database privileges.
2. SQL lets you select and output data from the database. An SQL Injection vulnerability could allow the attacker to gain complete access to all data in a database server.
3. SQL also lets you alter data in a database and add new data. For example, in a financial application, an attacker could use SQL Injection to alter balances, void transactions, or transfer money to their account.
4. You can use SQL to delete records from a database, even drop tables. Even if the administrator makes database backups, deletion of data could affect application availability until the database is restored. Also, backups may not cover the most recent data.
5. In some database servers, you can access the operating system using the database server. This may be intentional or accidental. In such a case, an attacker could use an SQL Injection as the initial vector and then attack the internal network behind a firewall.

There are several types of SQL Injection attacks: **in-band SQLi** (using database errors or UNION commands), **blind SQLi**, and **out-of-band SQLi**.

1. How to prevent an SQL Injection?

Preventing SQL Injection vulnerabilities is not easy. Specific prevention techniques depend on the subtype of SQLi vulnerability, on the SQL database engine, and on the programming language. However, there are certain general strategic principles that you should follow to keep your web application safe.

Step 1: **Train and maintain awareness**

To keep your web application safe, everyone involved in building the web application must be aware of the risks associated with SQL Injections. You should provide suitable security training to all your developers, QA staff, DevOps, and SysAdmins. You can start by referring them to this page.

Step 2: **Don’t trust any user input**

Treat all user input as untrusted. Any user input that is used in an SQL query introduces a risk of an SQL Injection. Treat input from authenticated and/or internal users the same way that you treat public input.

Step 3: **Use whitelists, not blacklists**

Don’t filter user input based on blacklists. A clever attacker will almost always find a way to circumvent your blacklist. If possible, verify and filter user input using strict whitelists only.

Step 4: **Adopt the latest technologies**

Older web development technologies don’t have SQLi protection. Use the latest version of the development environment and language and the latest technologies associated with that environment/language. For example, PHP uses PDO instead of MySQLi.

Step 5: **Employ verified mechanisms**

Don’t try to build SQLi protection from scratch. Most modern development technologies can offer you mechanisms to protect against SQLi. Use such mechanisms instead of trying to reinvent the wheel. For example, use parameterized queries or stored procedures.

Step 6: **Scan regularly (with Acunetix)**

SQL Injections may be introduced by your developers or through external libraries/modules/software. You should regularly scan your web applications using a web vulnerability scanner such as Acunetix. If you use Jenkins, you should install the Acunetix plugin to automatically scan every build.

**SQL Injection Process**

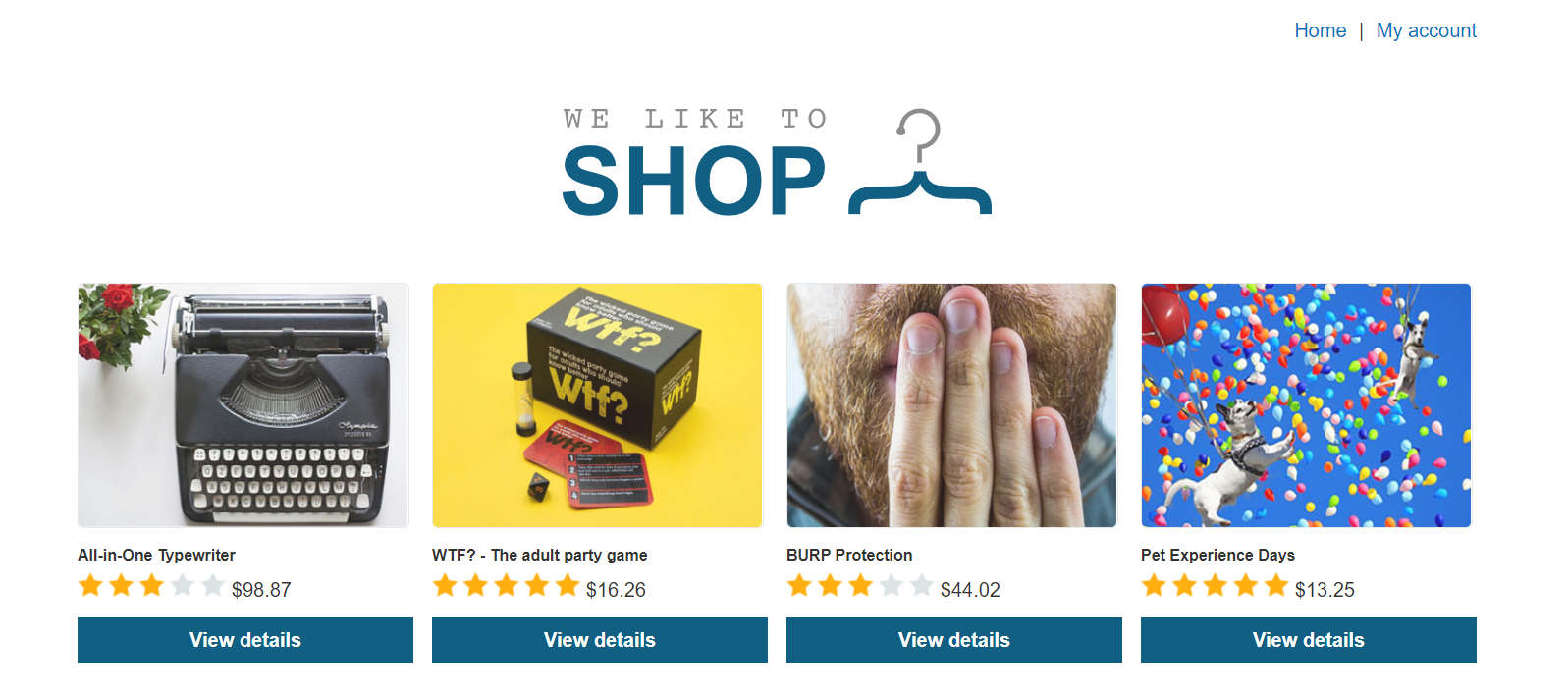
**SQL injection vulnerability allowing login bypass**

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator'-- and a blank password results in the following query:

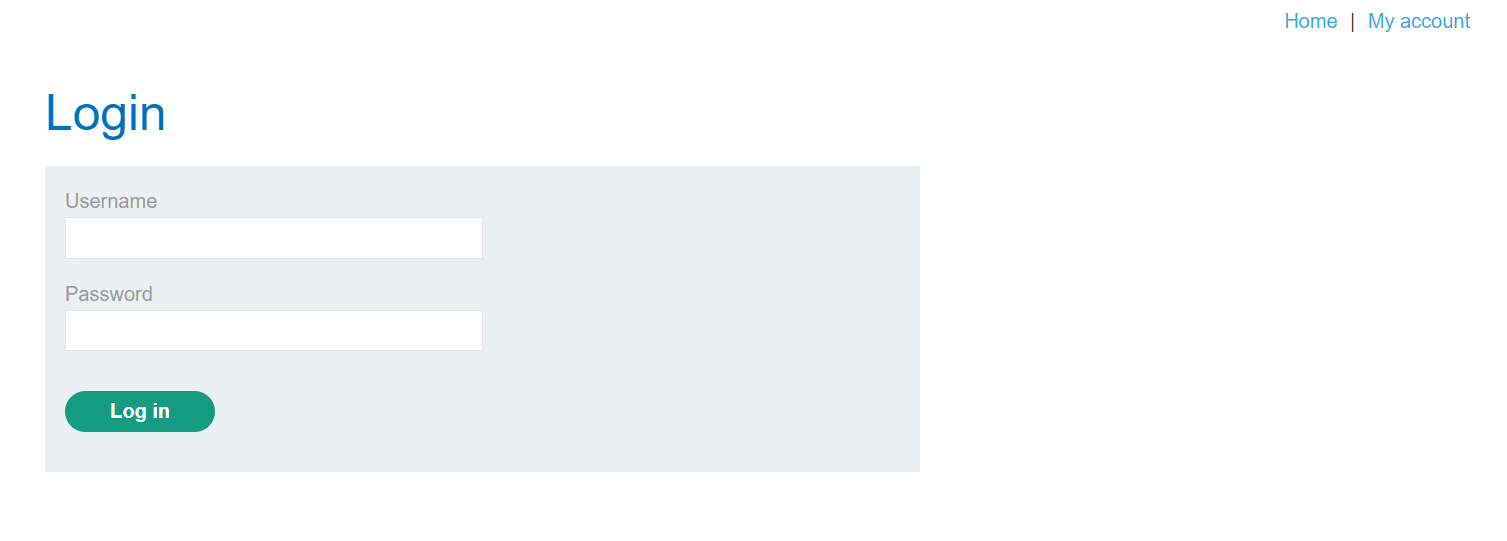
| SELECT \* FROM users WHERE username = 'administrator'*--' AND password = ''* |
| --- |

This query returns the user whose username is administrator and successfully logs the attacker in as that user. Consider a shopping application that displays products in different categories.

Step 1: Here we can see that there is a My Account button in which we will enter the username and password for login as admin.



Step 2: After clicking to My Account the screen appears for username and password.



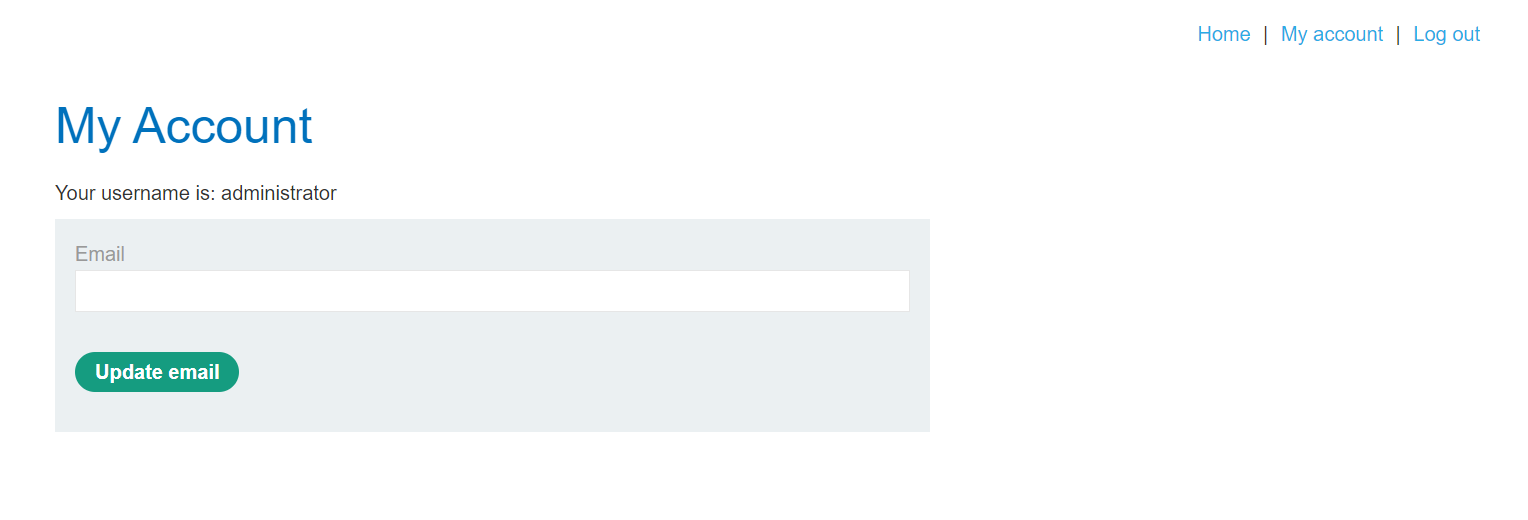
Step 3: After that we will enter username as **admin** and password as **admin** and try to get into the website. After clicking the LOGIN button an error will come and show as Invalid username and password.



Step 4: After this we will perform an SQL injection attack that logs in to the application as the administrator user.



Step 5: After then we can see that we got success in LOGIN as Administrator.



**SQL injection vulnerability in WHERE clause allowing retrieval of hidden data**

Consider a shopping application that displays products in different categories. This causes the application to make an SQL query to retrieve details of the relevant products from the database:

| SELECT \* FROM products WHERE category = 'Gifts' AND released = 1 |
| --- |

This SQL query asks the database to return all details (\*) from the products table where the category is Gifts and released is 1. The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably released = 0.

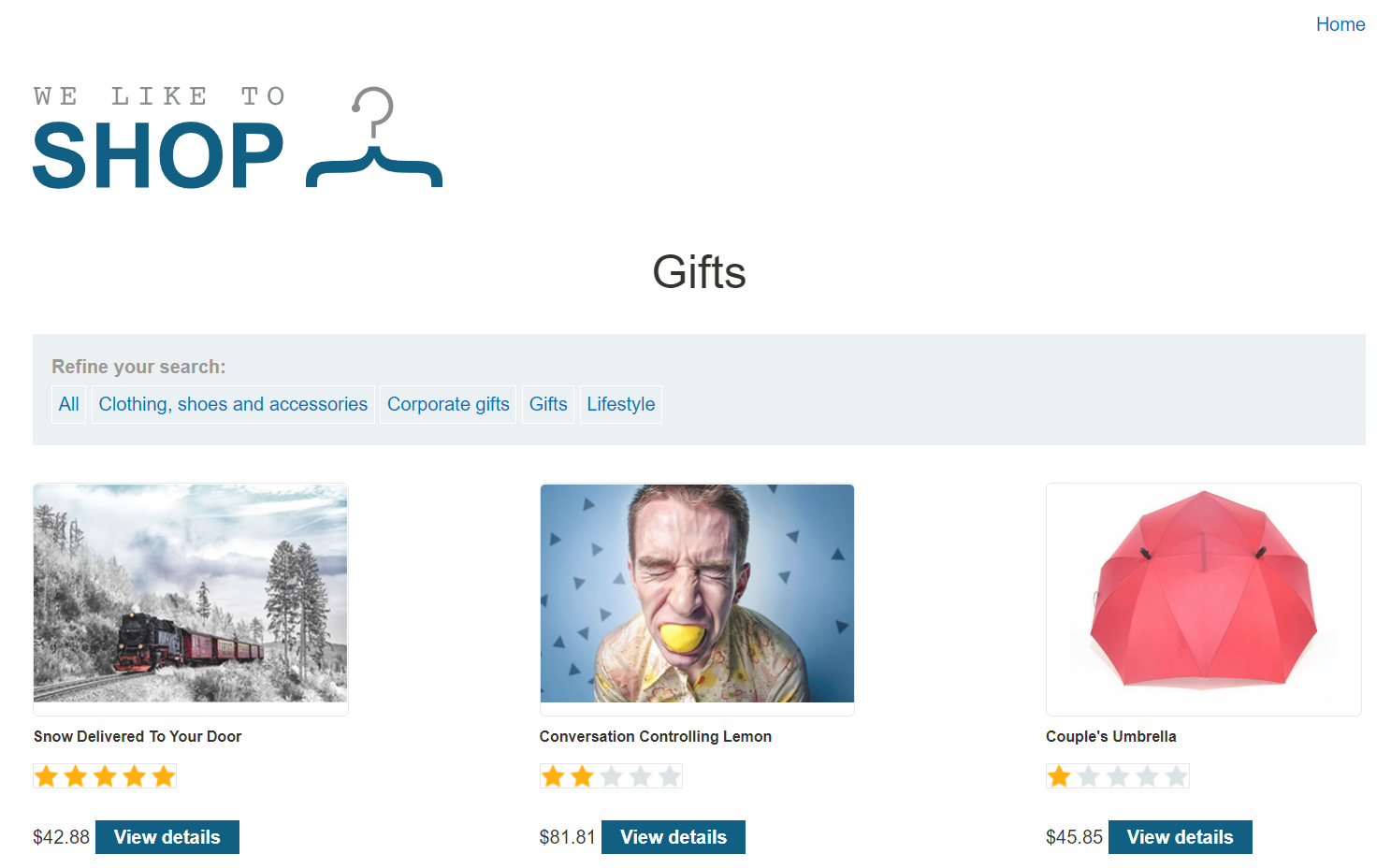
The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about. This results in the SQL query:

| SELECT \* FROM products WHERE category = 'Gifts' OR 1=1*--' AND released = 1* |
| --- |

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

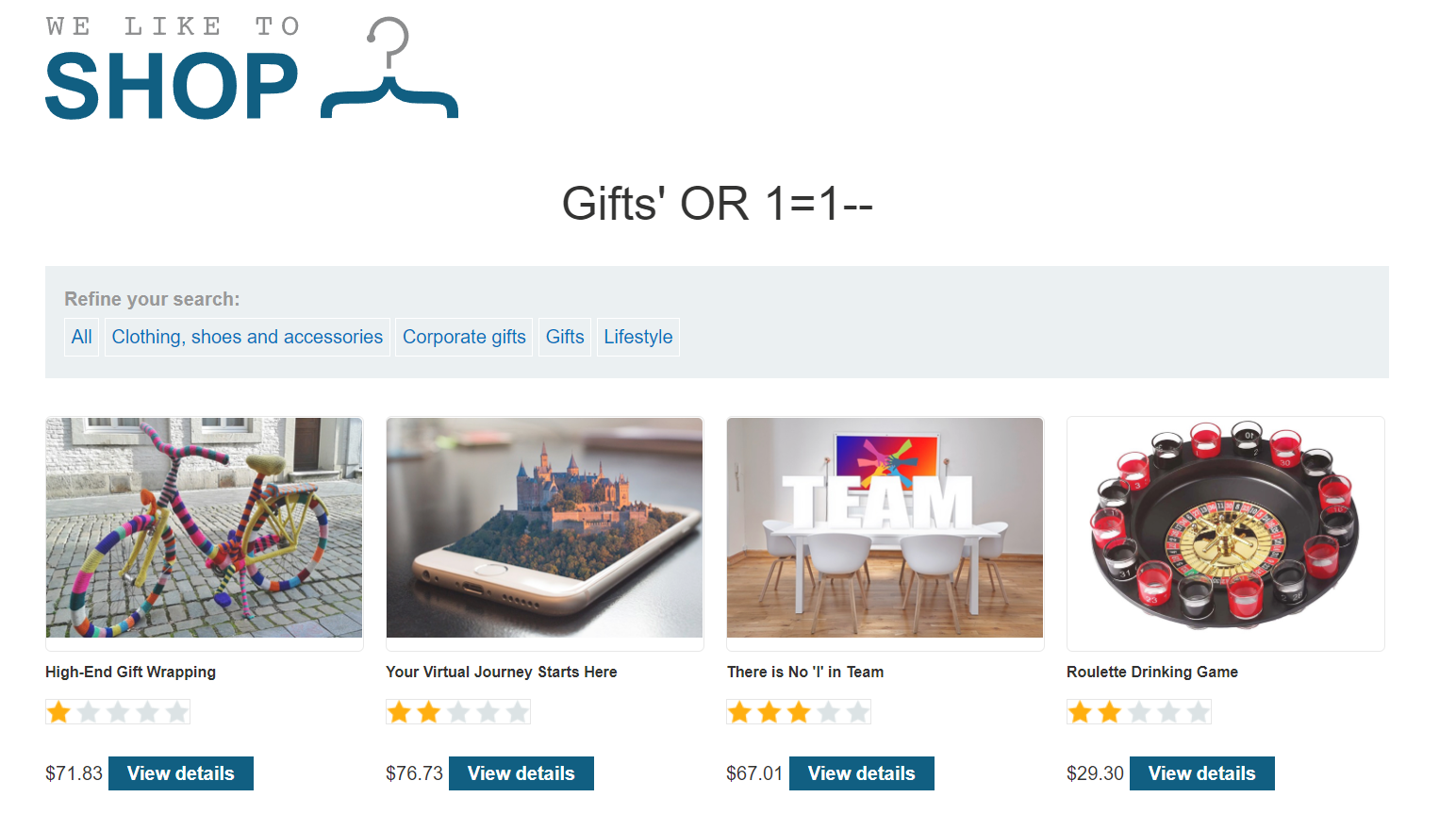
Step 1: Use Burp Suite to intercept and modify the request that sets the product category filter.



Step 2: Modify the category parameter, giving it the value '+OR+1=1–



Step 3: Submit the request, and verify that the response now contains additional items.



**SQL Injection Exploitation using DVWA**

Here, we will use the Damn Vulnerable Web Application (DVWA). It’s a web app developed in PHP and MySQL and intentionally made to be vulnerable.

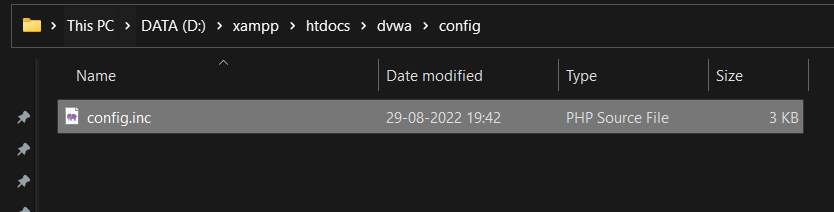
Step 1: Now [Download DVWA](https://github.com/digininja/DVWA) (Damn Vulnerable Web Application) from Github for Pen Testing purpose.



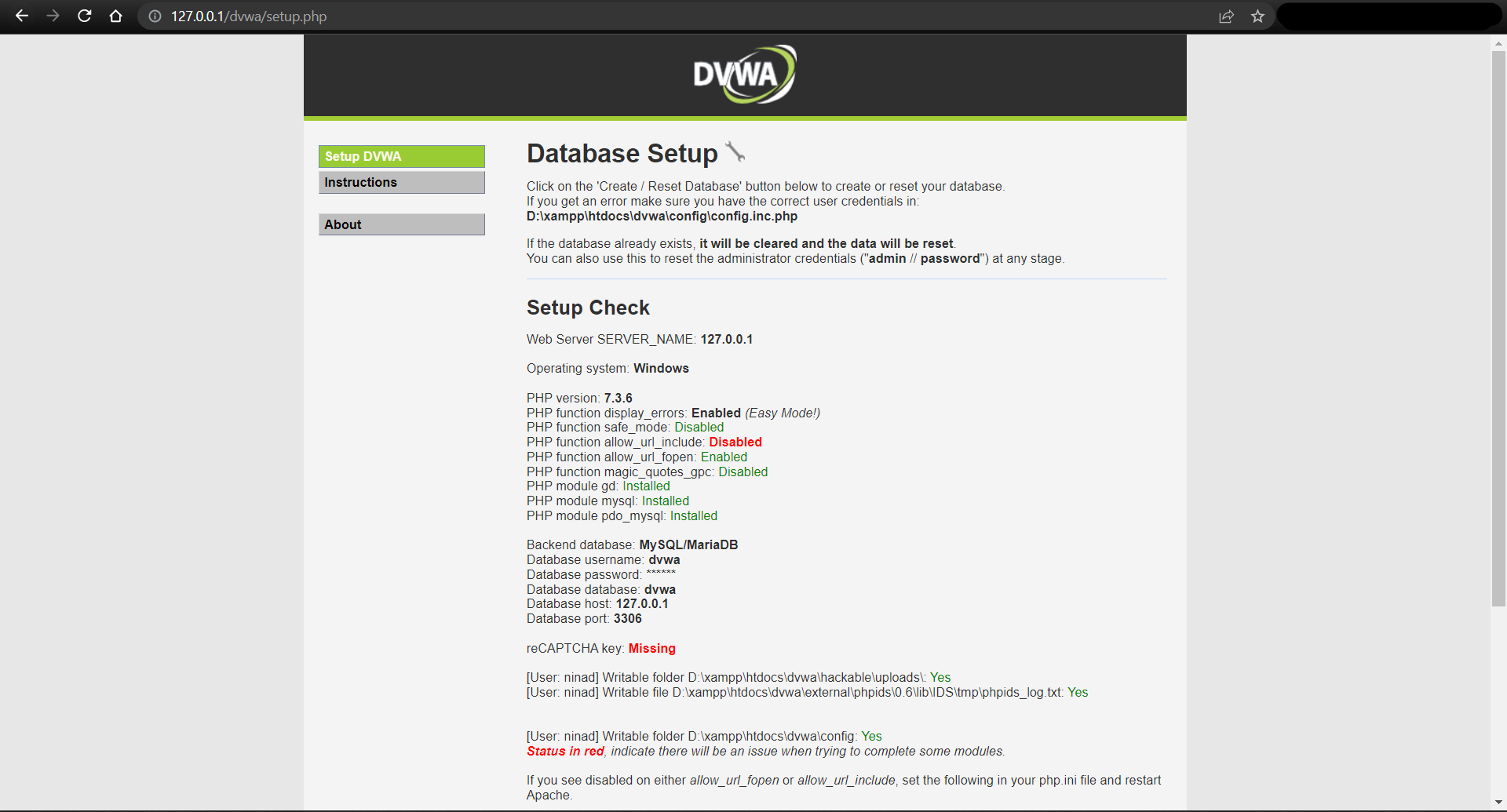
Step 2: Download Xampp and install. Reference for installation of Burp Suite, Xampp and DVWA.

Step 3: Move the DVWA folder to htdocs folder under Xampp:

* Goto config -> Change the **config.inc.php.dist** file name to **config.inc.php**



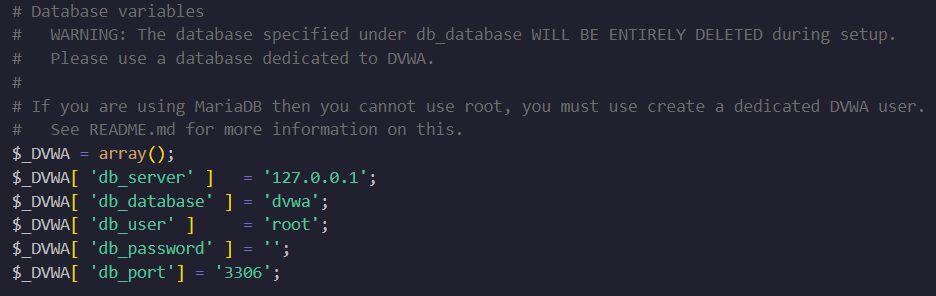
* Change the port number (8080) if required in Apache (httpd.conf) file
* Browse to <http://127.0.0.1/dvwa>

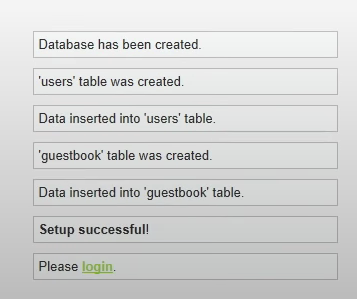


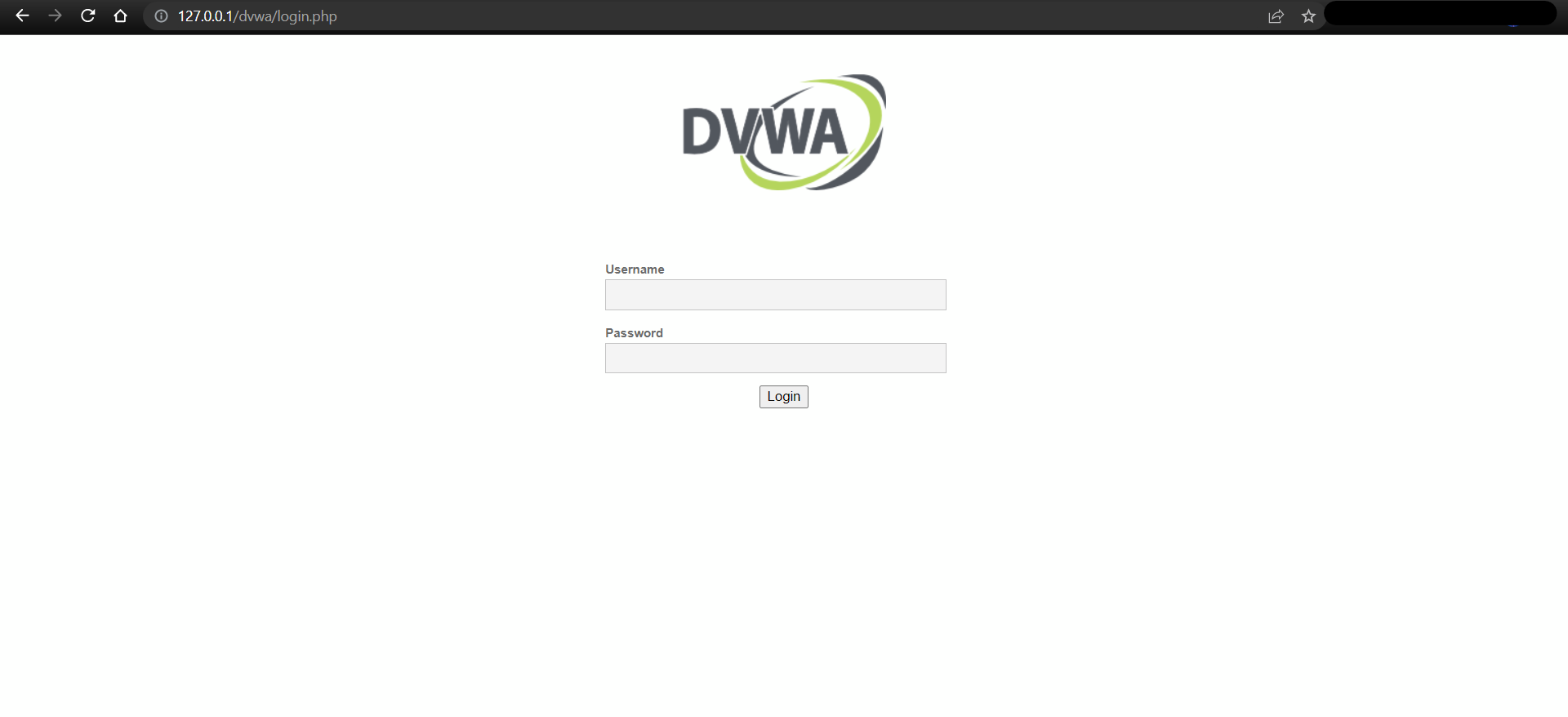
* Try to open [http://127.0.0.1:8012/dvwa/dvwa/setup.php](http://127.0.0.1:8012/DVWA-master/DVWA-master/setup.php)
* If it gives Sql Error , open Config.inc.php file and change the

*$\_DVWA[ 'db\_user' ] = 'root';*

*$\_DVWA[ 'db\_password' ] = '';*



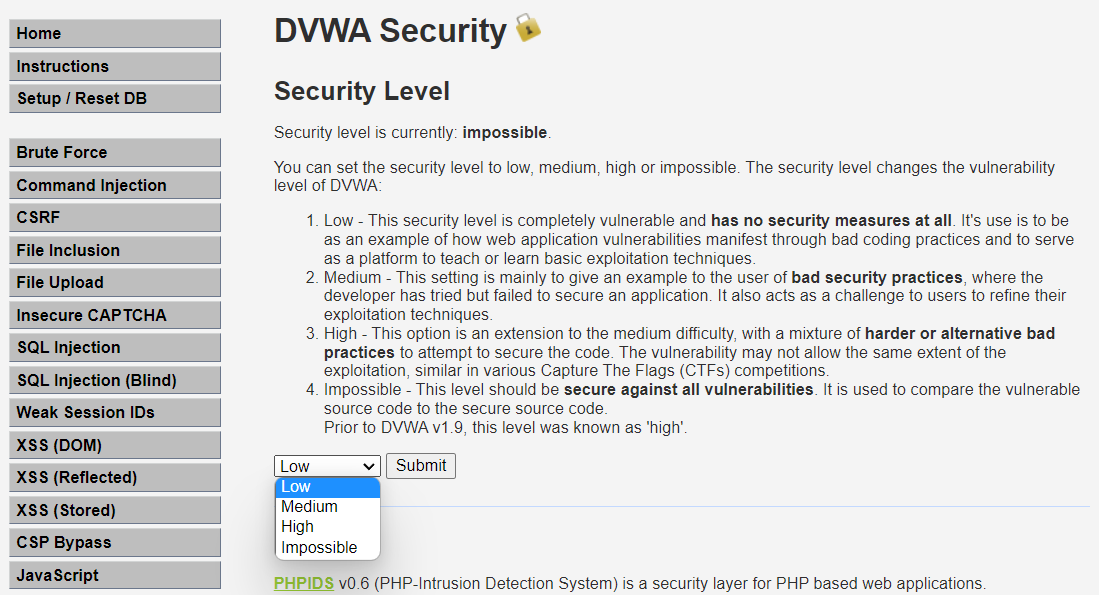




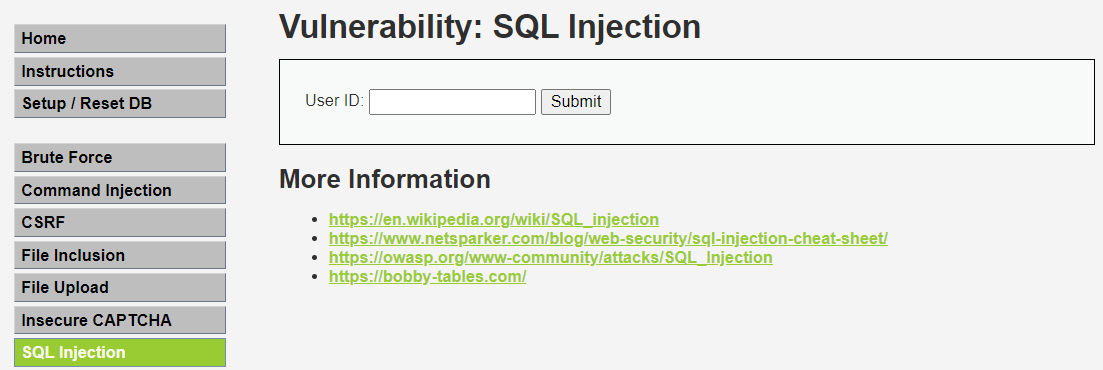
Step 4: Use the default credentials below:

* *Username:* ***admin***
* *Password:* ***password***

After a successful login, you will see the DVWA main page. First, click on the DVWA Security on the bottom left, set security to Low, and click Submit.



Step 5: On the left section of the page, you will see the various vulnerable pages to exploit. Click SQL Injection. You should see a page similar to this below.

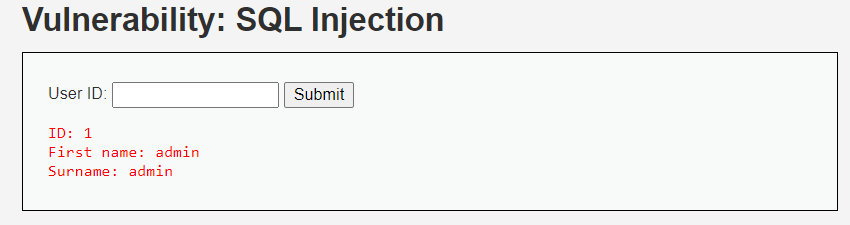


Step 6: On the SQL injection page, click the View Source button at the bottom right. That will open a page with the SQL Injection source code written in PHP. When you go through the code, you will see a line like:

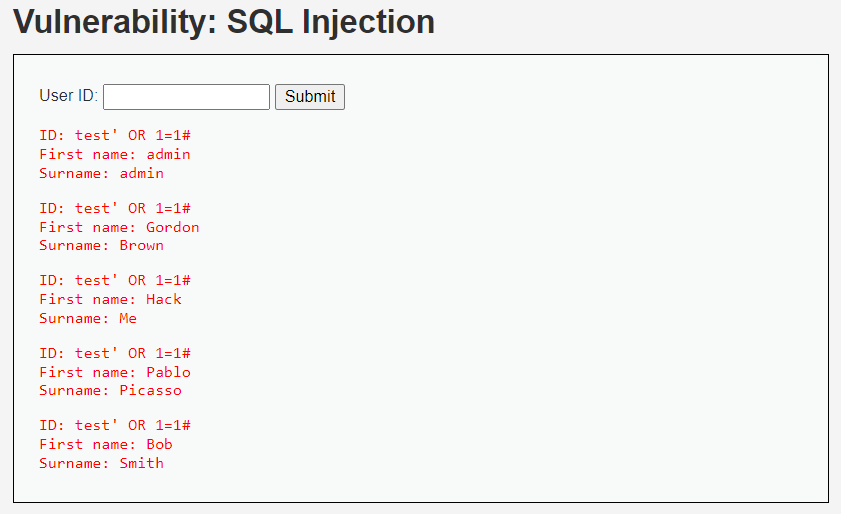
| $query = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id'"; |
| --- |

That is the vulnerable line of code. At the end of the line, you can see the user input is concatenated to the SQL query without being validated. That allows us to pass arbitrary commands into the database. Let’s get started.

On the SQL Injection page, we have a USER ID field. When we enter number 1, the application returns the Firstname and Surname of the user with ID 1.

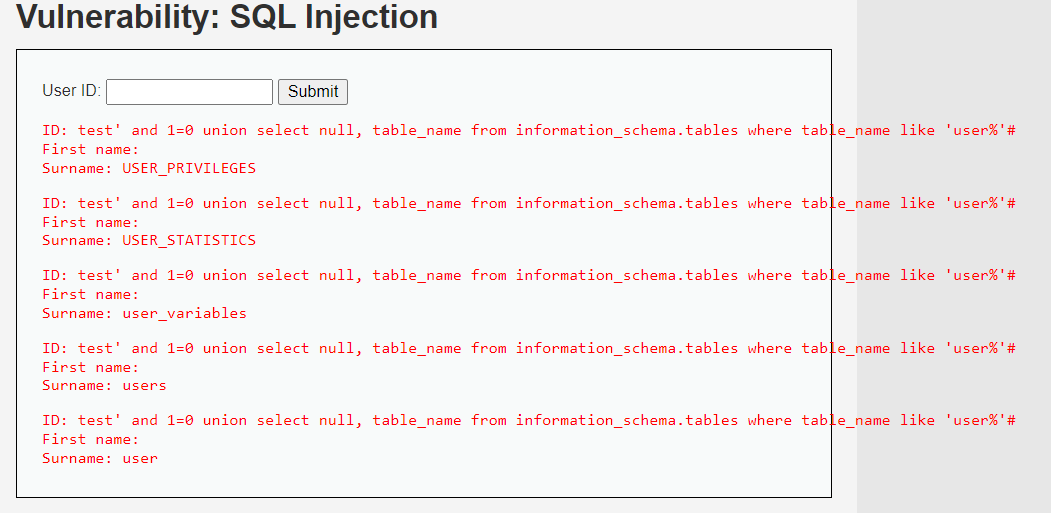


Step 7: We looked at this when talking about How an SQL Injection attack works. Enter an input like test' OR 1=1# and hit Enter. That will return the username and surname of all users in the database.



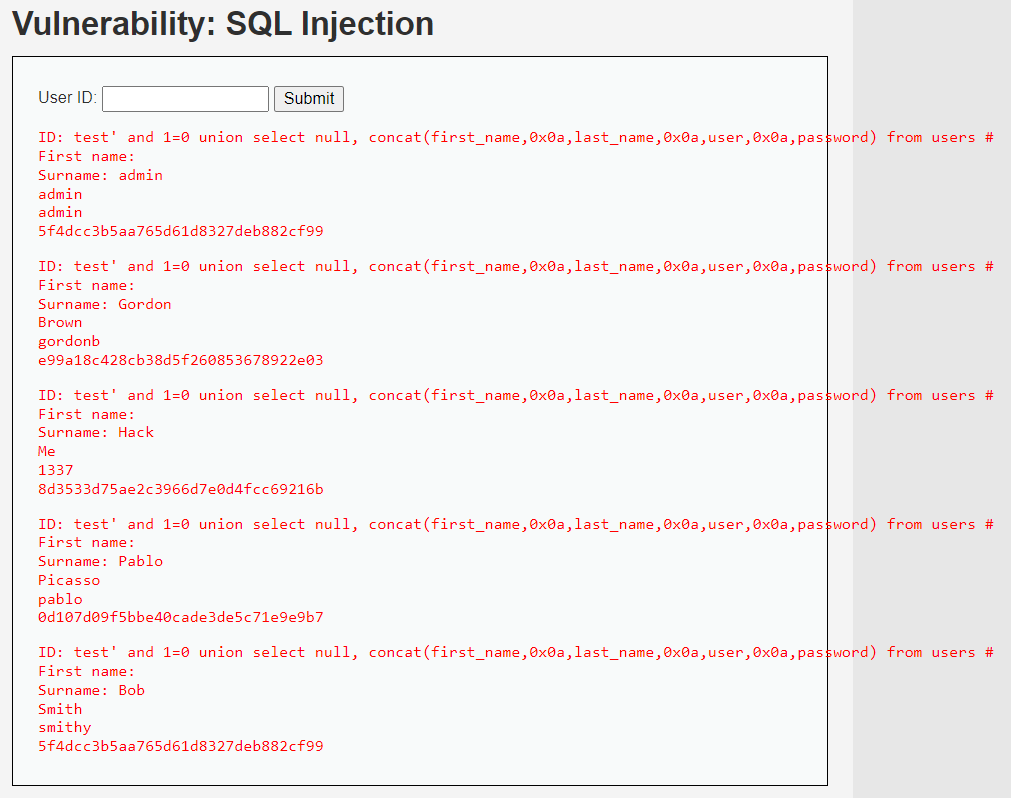
Step 8: List all tables in the information schema. The Information Schema is a record that holds information about all other databases maintained by MySQL RDBMS. Enter the query below in the USER ID field.

| test' and 1=0 union select null, table\_name from information\_schema.tables where table\_name like 'user%'# |
| --- |



Step 9: Display all the column contents in the information schema users table. This is much more interesting. We will display all the authentication information of all users in the database. That includes password hashes. Enter the query below.

| test' and 1=0 union select null, concat(first\_name,0x0a,last\_name,0x0a,user,0x0a,password) from users # |
| --- |



From the output above, you can see the hashed password. We can go ahead and crack the hash to reveal the actual password. Some of the password cracking tools that come in handy include John the Ripper and Medusa. There are also websites where you can paste the password hash to reveal the actual password.

**Conclusion**:

Thus we have studied how to validate the data and how to perform SQL injection attacks.