



DVWA Installation & SQL Injection

Saira S Elizabeth

Cyber Security Analyst
ICT Academy

CONTENTS

1. Installation of DVWA using Docker	02
a. Clone the Repository.....	02
b. Boot the Docker Container.....	02
c. Access to the DVWA Web Page.....	02
d. Login.....	03
e. Database Reset.....	04
f. Re-Login.....	04
g. Conclusion.....	05
2. Performing SQL Injection On DVWA.....	06
a. SQL Injection (Low Security Level)	06
i. First Injection	06
ii. SQL Payload.....	06
b. SQL Injection (Medium Security Level)	07
i. Using Burp Suite.....	07
ii. SQL Injection String.....	09
iii. Execution.....	09
c. SQL Injection (High Security Level)	10
i. Injection Point Identification.....	10
ii. Injection Payload.....	11
iii. Results.....	11
d. Conclusion.....	12

1. INSTALLATION OF DVWA USING DOCKER

For the hassle-free installation of Damn Vulnerable Web Application (DVWA), I used Docker. Below are the steps that I followed to complete the installation.

- CLONE THE REPOSITORY

First, I cloned the DVWA repository from [pentestlab.github.io](https://github.com/eystsen/pentestlab) using the following Command:

```
git clone https://github.com/eystsen/pentestlab.git
```

- BOOT THE DOCKER CONTAINER

After cloning the repository, I entered into the folder of DVWA and typed out some Docker commands to boot the web application. The steps which I followed were:

- a. Terminal opened, then cloned the pentestlab folder into the terminal

```
cd pentestlab
```

- b. Installed Docker container with the following command:

```
sudo apt install docker.io
```

- ACCESS TO THE DVWA WEB PAGE

After starting the Docker container I run this command to access the DVWA web page.

Command:

```
./pentestlab.sh start dvwa
```

```
(raw@data)-[~]
$ git clone https://github.com/eystsen/pentestlab.git
Cloning into 'pentestlab'...
remote: Enumerating objects: 153, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 153 (delta 7), reused 13 (delta 7), pack-reused 136 (from 1)
Receiving objects: 100% (153/153), 42.69 KiB | 227.00 KiB/s, done.
Resolving deltas: 100% (73/73), done.

(raw@data)-[~]
$ cd pentestlab

(raw@data)-[~/pentestlab]
$ sudo apt install docker.io
docker.io is already the newest version (26.1.5+dfsg1-2+b1).
The following packages were automatically installed and are no longer required:
  libdaxctl1 libjxl0.7 libpmm1 libre2-10 libsvtav1enc1d1 libx265-199
  libgeos3.12.1t64 libndctl6 librav1e0 libroc0.3 libu2f-udev openjdk-21-jre
  openjdk-21-jre-headless python3-mistune0 python3-pytzdata samba-dsdb-modules
  Use 'sudo apt autoremove' to remove them.

Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 959

(raw@data)-[~/pentestlab]
$ ./pentestlab.sh start dvwa
Starting Damn Vulnerable Web Application
Adding dvwa to your /etc/hosts
127.8.0.1 dvwa was added successfully to /etc/hosts
not set
Running command: docker run --name dvwa -d -p 127.8.0.1:80:80 vulnerables/web-dvwa
392a154c409f5b799ca3cf79e8fba4440a6e886264102784df8dd16a43d00b5e
DONE!

Docker mapped to http://dvwa or http://127.8.0.1

Default username/password: admin/password
Remember to click on the CREATE DATABASE Button before you start
```

Screenshot 1

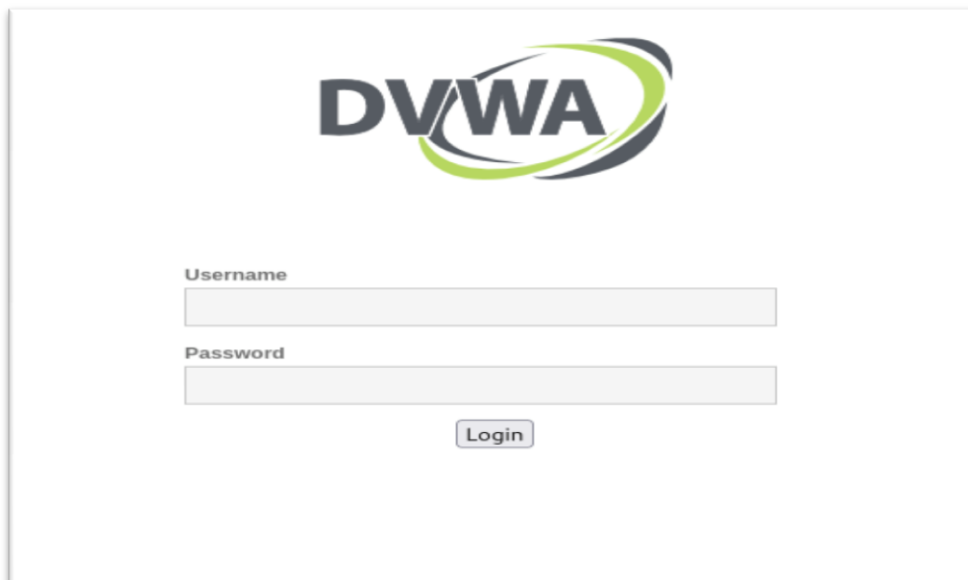
• LOGIN

On the login page, I used the following default information:

website: <http://dvwa>

username: admin

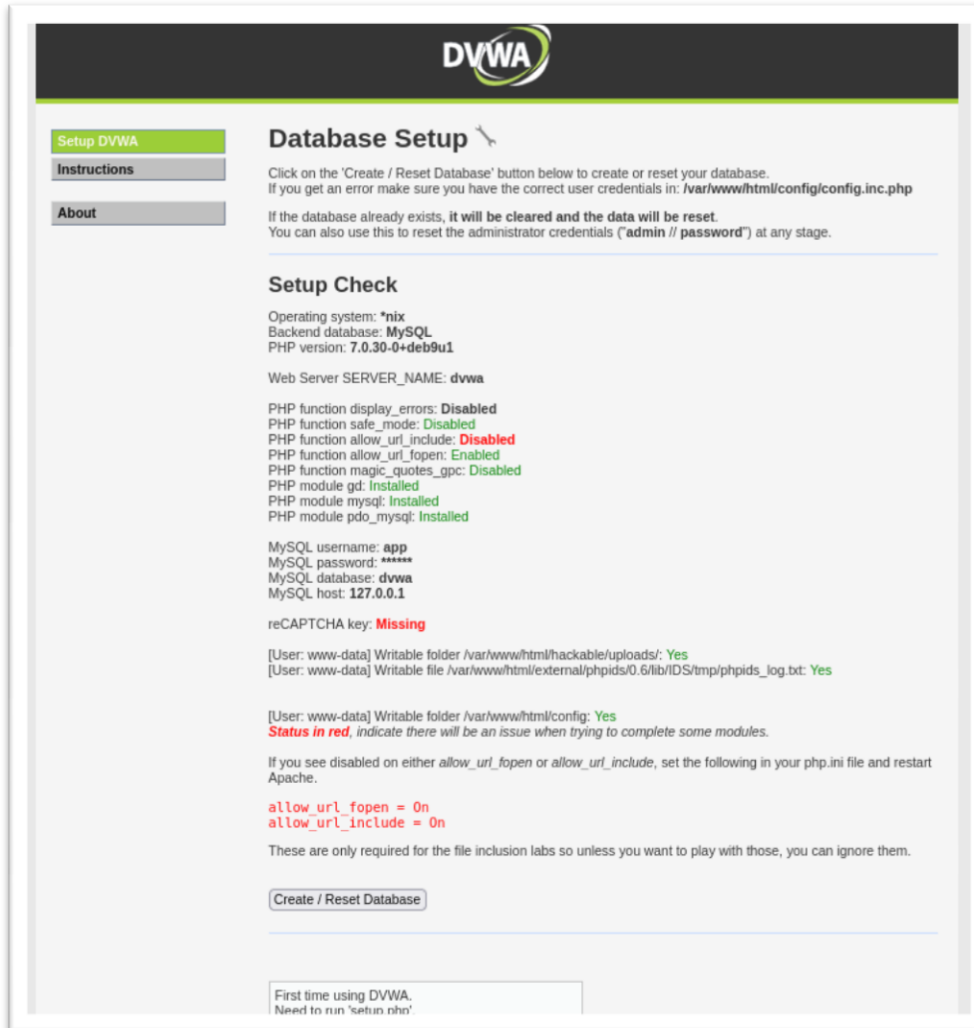
password: password



Screenshot 2

• DATABASE RESET

In the prompt, after logging in for the first time, I was automatically prompted to reset the database. I clicked the “Reset Database” button.



Screenshot 3

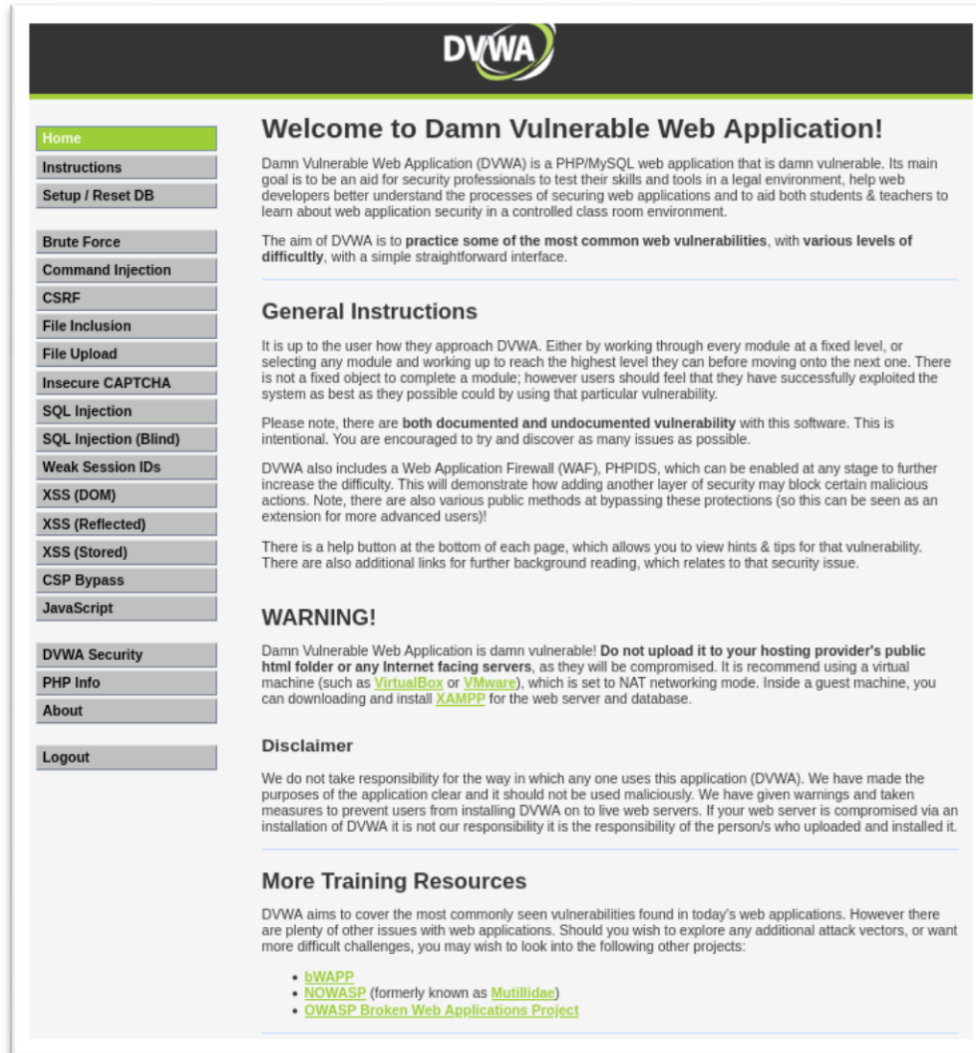
Following the reset of the database, the system took me back to the login page.

• RE-LOGIN

I then logged in again using the default credentials to access the DVWA dashboard.

• CONCLUSION

By this point, the DVWA installation was now done, and the environment was now set up for vulnerability testing.



Screenshot 4

2.PERFORMING SQL INJECTION ON DVWA

- SQL INJECTION (LOW SECURITY LEVEL)

I began by attempting the SQL injection at the Low security level.

FIRST INJECTION

After finding the SQL injection page, I identified the input field in which to inject my SQL code.

SQL PAYLOAD

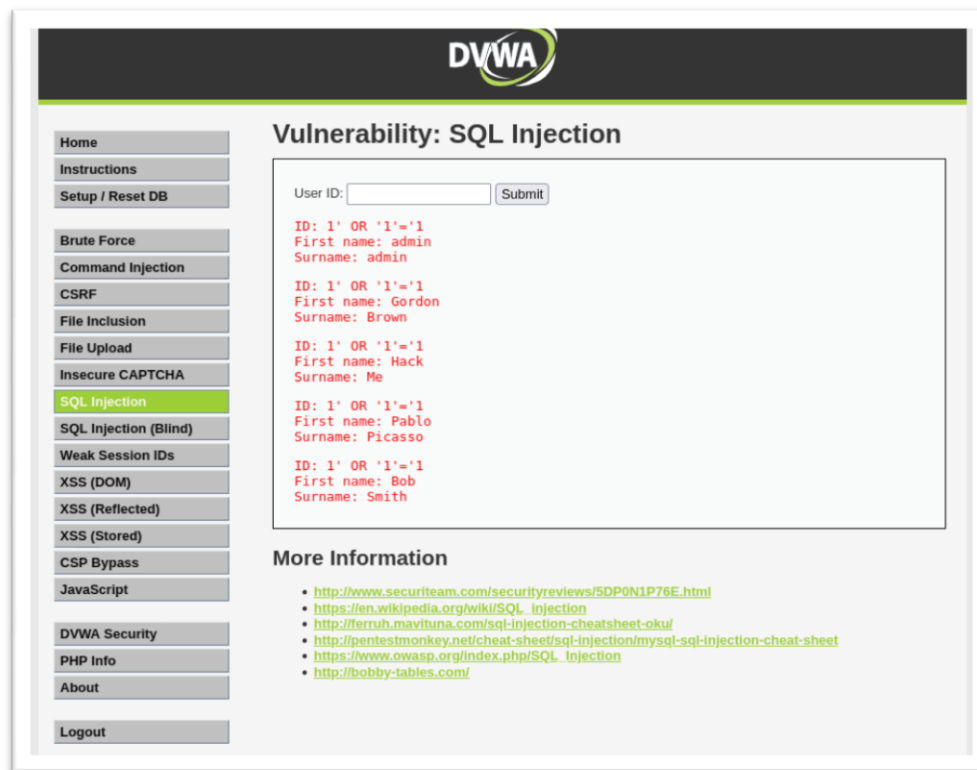
I used the following simple SQL injection string:

`1' OR '1'='1`



Screenshot 5

This payload circumvented the need for a valid input and printed out the first name and last name of all users.



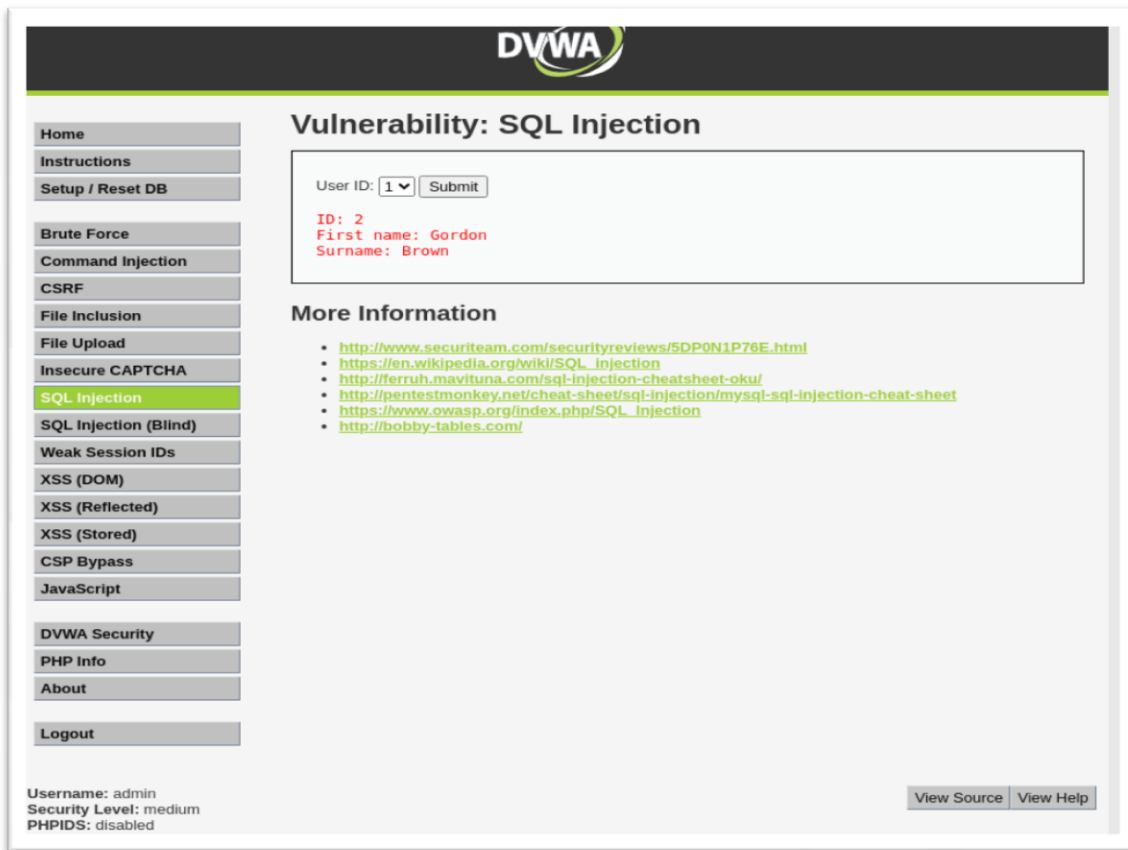
Screenshot 6

• SQL INJECTION (MEDIUM SECURITY LEVEL)

I set the security of DVWA to Medium and ran the test with a more complex payload.

USING BURP SUITE

I captured the HTTP request using Burp Suite. I modified the `id` parameter in the request to be a complex SQL injection string.



Screenshot 7

In Burp suite side request

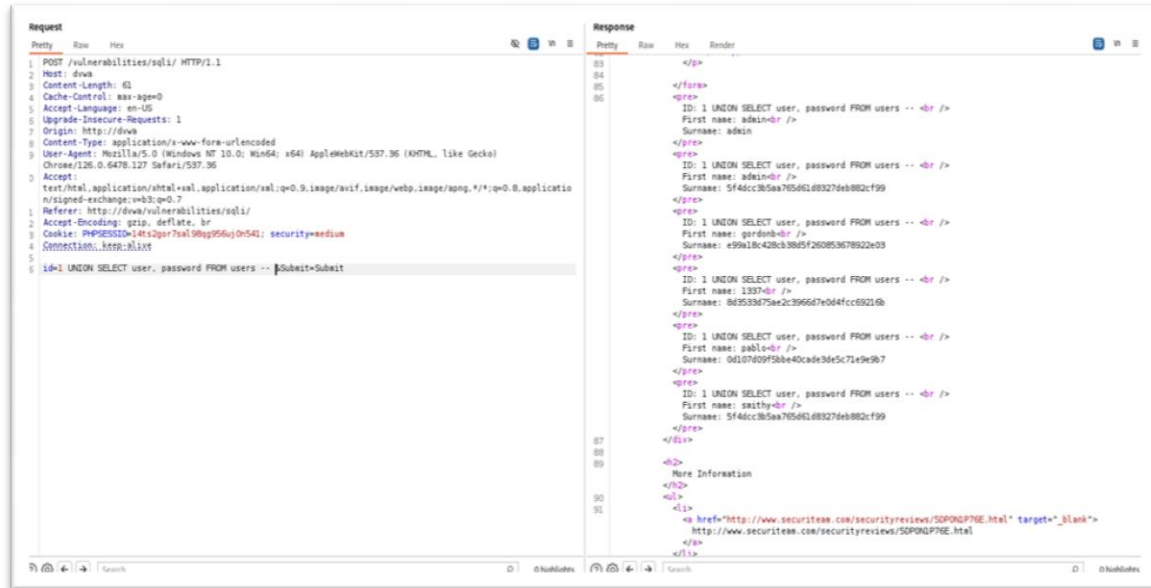


Screenshot 8

SQL INJECTION STRING

I inserted the following payload in the `id` field:

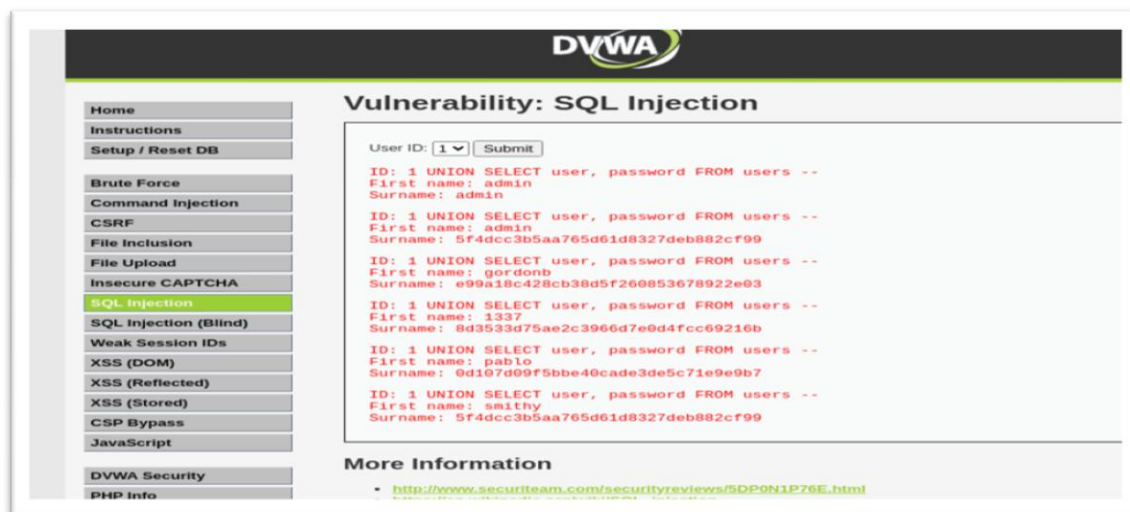
1 UNION SELECT user, password FROM users --



Screenshot 9

EXECUTION

After I had modified the request in Burp Suite, I sent it to the server. Because of this, I could see usernames and passwords fetched by the system's response (refer to screenshots 8 and 9).



Screenshot10

• SQL INJECTION (HIGH SECURITY LEVEL)

I attempted SQL injection at the High security level.

INJECTION POINT IDENTIFICATION

The interface is a bit different at the High security level. When I clicked the "[here to change your ID](#)" button



Screenshot 11

A new window appeared with which I could enter the SQL command.

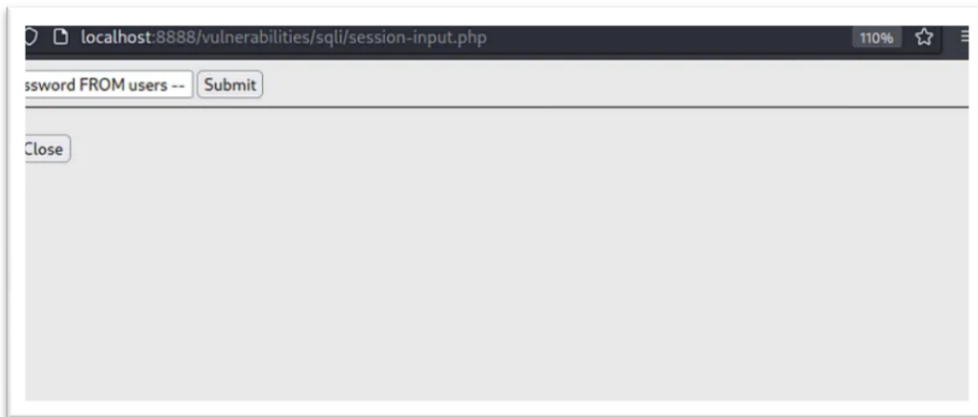


Screenshot 12

INJECTION PAYLOAD

I used the following SQL injection string:

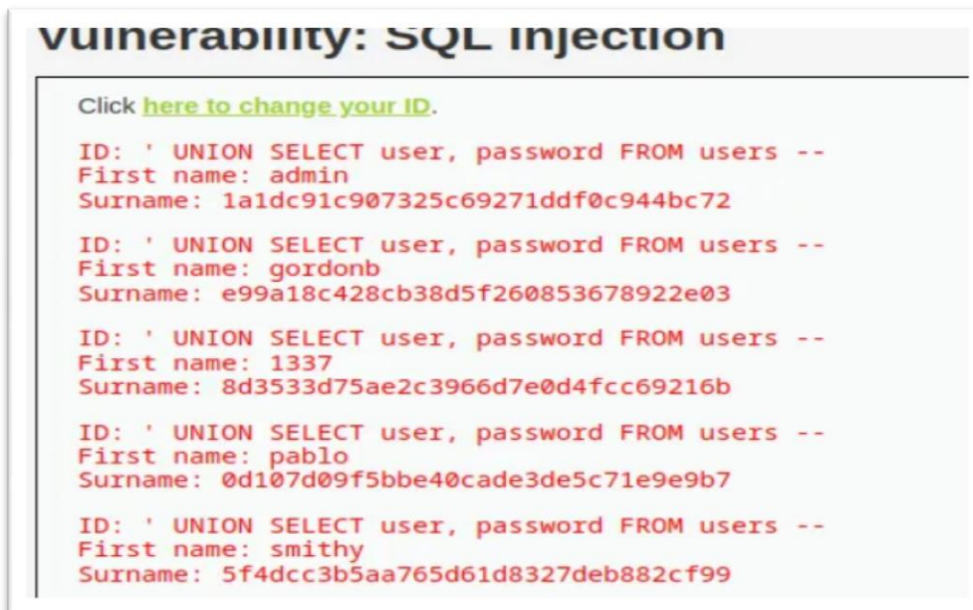
‘ UNION SELECT user, password FROM users --



Screenshot 13

RESULTS

As a sad reminder to the developers, the system responded to the malicious code with a list of usernames and passwords after it had been submitted, which successfully confirmed the vulnerability-even at the highest security setting.



Screenshot 14

- CONCLUSION

SQL Injection allows attackers to manipulate queries, potentially bypassing authentication and accessing sensitive data. In DVWA, the **low** and **medium** levels were vulnerable, while **high** security used parameterized queries, effectively preventing injections. The key takeaway is that **input validation** and **prepared statements** are crucial defenses to mitigate SQL Injection risks and protect web applications from attacks.