



**UNIVERSITY of INFORMATION
TECHNOLOGY and MANAGEMENT**
in Rzeszow, POLAND

Threat Analysis of SQL Injection Attacks

Teacher:

dr. Nataliia Poliakova

Class:

Cybersecurity Essentilas

Student:

Şevval Kayalı w71283

Field of study:

Information
Technology
ERASMUS

Rzeszów 2024

Contents

Introduction.....	3
1. Threat Analysis of SQL Injection Attacks	4
1.2.1 Injection Through Cookies	4
1.3 Classification of SQL Injection Threats	4
1.3.1 Classic SQL Injection	4
1.3.2 Blind SQL Injection.....	5
1.3.3 Boolean-Based SQL Injection	5
1.3.4 Time-Based SQL Injection	5
1.4Case Studies of SQL Injection Attacks	5
CVE-2024-1100	5
2 Planning countermeasures	7
Penetration Test: Practical Implementation	7
Objective.....	7
2.1 Environment Setup.....	7
2.2 Installing DVWA	8
2.3. SQL Injection Simulation.....	9
2.3.1Setting up the Vulnerable Environment For Low Security Level.....	9
2.3.2 Setting up the Vulnerable Environment For Medium Security Level	14
3 Preparing an experiment to verify the effectiveness of protection	17
Conclusions.....	20
List of Sources.....	21

Introduction

At the present times most business processes have incorporated the Web applications due to their convenient and dynamic services in the rapidly growing digital environment. But with this convenience has come the downside which is that the systems, applications and processes used in the business world have become much more susceptible to threat. Risk is the susceptibility of an organization and its systems to undue interference, malicious attacks or data loss. Weak authentication schemes inadequate input validation and bad error handling are among the main risks in web apps.

It was established that Structured Query Language (SQL) injection attacks are considered to be one of the most widespread and critical threats to Web applications. This is a very dangerous form of vulnerability since attackers can easily access all information that is stored in the application's database including usernames, names, address, phone numbers and credit card details etc. [1]

Sql injection attacks exploit interfaces, which are not sufficiently secure and allow the attacker to control SQL operations on Back end databases. In such attacks, vital details can be disclosed, records can be modified or removed, and worse, whole networks can be wiped out. These vulnerabilities are normally found as result of inadequate or incorrect input validation also inadequate secure coding.

SQL injection has been a critical and long-standing threat for web applications, or at least for its over two-decade existence, SQL injection was a constant member of the OWASP Top 10. Recent cyber attacks on high profile organizations like Sony Pictures and TalkTalk are a clear evidence that ignoring the SQL injection vulnerabilities is a very costly mistake.

On November 24, 2014, the hacker group "Guardians of Peace" leaked confidential data from the film studio Sony Pictures Entertainment (SPE). The data included employee emails, personal and family information, executive salaries, then-unreleased films, future film plans, screenplays, and other information.[2]

These events underscore the significance of strong defense mechanisms to safeguard the applications as well as the databases against such perils.

This report of mine, discusses the weaknesses of systems that allow SQL injection attacks together with their classification, operation modes and examples of application. Further, it has incorporated a virtual simulation to show how all these attacks are conducted in real life. In the last section of the report, the techniques in the prevention of SQL injection and the general fortification of web application security are provided.

Keywords: SQL Injection (SQLi), Web Application Vulnerabilities ,Database Security, Cyber Threats, Web Security

1. Threat Analysis of SQL Injection Attacks

SQL Injection is a very serious and critical vulnerability in data security, where malicious SQL statements are injected into an application to modify or gain unauthorized access to a database. According to OWASP, injection attacks, which include SQL Injections, have been rated as a significant threat to web application security. This type of attack was listed as the third most critical web application security risk for the year 2021.[3]

1.1 What is SQL and SQL Query?

Structured Query Language- programming language for manipulating and accessing databases. With SQL, one can create and manipulate the database. SQL Query: a written command in SQL performs certain actions in a database. SQL Queries follow a syntax that is defined and utilized for inserting, viewing, updating, or deleting the database. The standard SQL commands include specialized SELECT, INSERT, UPDATE, and DELETE.

1.2 Common Types of SQL Injection Attack Mechanisms

1.2.1 Injection Through Cookies

Cookies are small files created by web applications to store state information on a client's machine. These are sent back to the application upon the user revisiting the site, and this aids in restoring the user's previous session or settings. However, since the client has full access to these files, they can be manipulated by an adversarial user. If the web application constructs SQL queries based on data within the cookie without proper sanitization, a malicious actor could put evil SQL within the cookie to attack your database via an SQL injection [4].

1.2.2 Second-order injection

Second-order injection is a form of attack in which attackers inject malicious inputs into a system or database in order to potentially trigger an SQL injection attack at a later time. Unlike first-order injection attacks, the goal of second-order injections is not to execute the attack immediately when the input reaches the database. Instead, this type of attack is designed to be triggered at some later time, as the input is subsequently used or processed by the application[6].

1.2.3 Server variables injection

This occurs when a web application fails to properly filter data stored in server variables. Examples of server variables include HTTP headers, network headers, and environmental variables. These variables are commonly used for tasks such as logging access statistics and analyzing browsing behavior. When an application writes such variables directly into a database without proper sanitization, it could lead to a vulnerability in SQL injection [5].

1.3 Classification of SQL Injection Threats

1.3.1 Classic SQL Injection

In this type, the hackers are exploiting direct user input by sending some malicious query prompts.

1.3.2 Blind SQL Injection

Blind SQL Injection occurs when a web application is susceptible to SQL injection, but the attacker is not able to directly view the results of their injected queries.[7] This could mean that even if the application does not display error messages, it does not mean that it is immune from "blind" SQL injection attacks.

1.3.3 Boolean-Based SQL Injection

Boolean-based SQL injection constitutes a technique that entails transmitting SQL queries to the database, thereby altering the application to generate varying responses contingent upon the outcomes of the queries.[8]

1.3.4 Time-Based SQL Injection

Time-based SQL injection constitutes a specific form of blind SQL injection whereby an attacker performs SQL queries that induce the database to postpone its response for a predetermined duration. This induced delay provides the attacker with the opportunity to ascertain the existence of a vulnerability, even in instances where the application conceals error messages but inadequately addresses the fundamental risk associated with SQL injection.[9]

1.4Case Studies of SQL Injection Attacks

CVE-2024-1100

represents a high-risk SQL injection vulnerability identified in DIGIKENT GIS, which was developed by Vadi Corporate Information Systems. This vulnerability arises because of insufficient neutralization of certain elements in SQL commands and allows attackers to execute malicious SQL queries. This bug affects all versions of DIGIKENT GIS before 2.23.5.[10] This vulnerability appears due to insufficient neutralization of certain elements included within SQL commands. As a side effect, this allows an attacker to perform SQL Injection-type attacks using this vulnerability and might lead to unauthorized access or modification of the database.

Fig.1.CVE Record Information

Required CVE Record Information

CNA: TR-CERT (Computer Emergency Response Team Of The Republic Of Turkey)

Published: 2024-05-30 **Updated:** 2024-05-30

Title: SQLi In Vadi Corporate Information Systems' DIGIKENT GIS

Description

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') vulnerability in Vadi Corporate Information Systems DIGIKENT GIS allows SQL Injection.This issue affects DIGIKENT GIS: through 2.23.5.

CWE 1 Total

[Learn more](#)

- **CWE-89: CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

CVSS 1 Total

[Learn more](#)

Score	Severity	Version	Vector String
10.0	CRITICAL	4.0	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:H/SI:H/SA:H

Source: <https://www.cve.org/CVERecord?id=CVE-2024-1100> , as of 01.12.2024

2 Planning countermeasures

Penetration Test: Practical Implementation

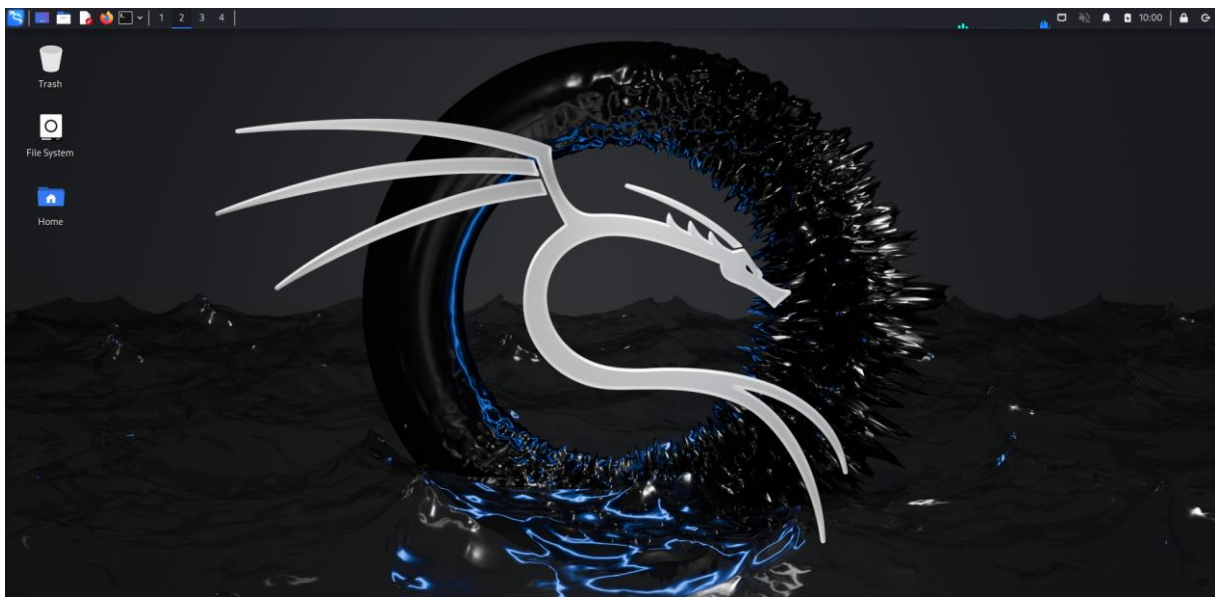
Objective

Perform a simulation of a basic SQL injection attack in a controlled environment to understand how it works and what the implications are.

2.1 Environment Setup

Installed **Kali Linux** as the guest OS on VirtualBox.

Fig.2 a screenshot of Kali Linux running in VirtualBox.

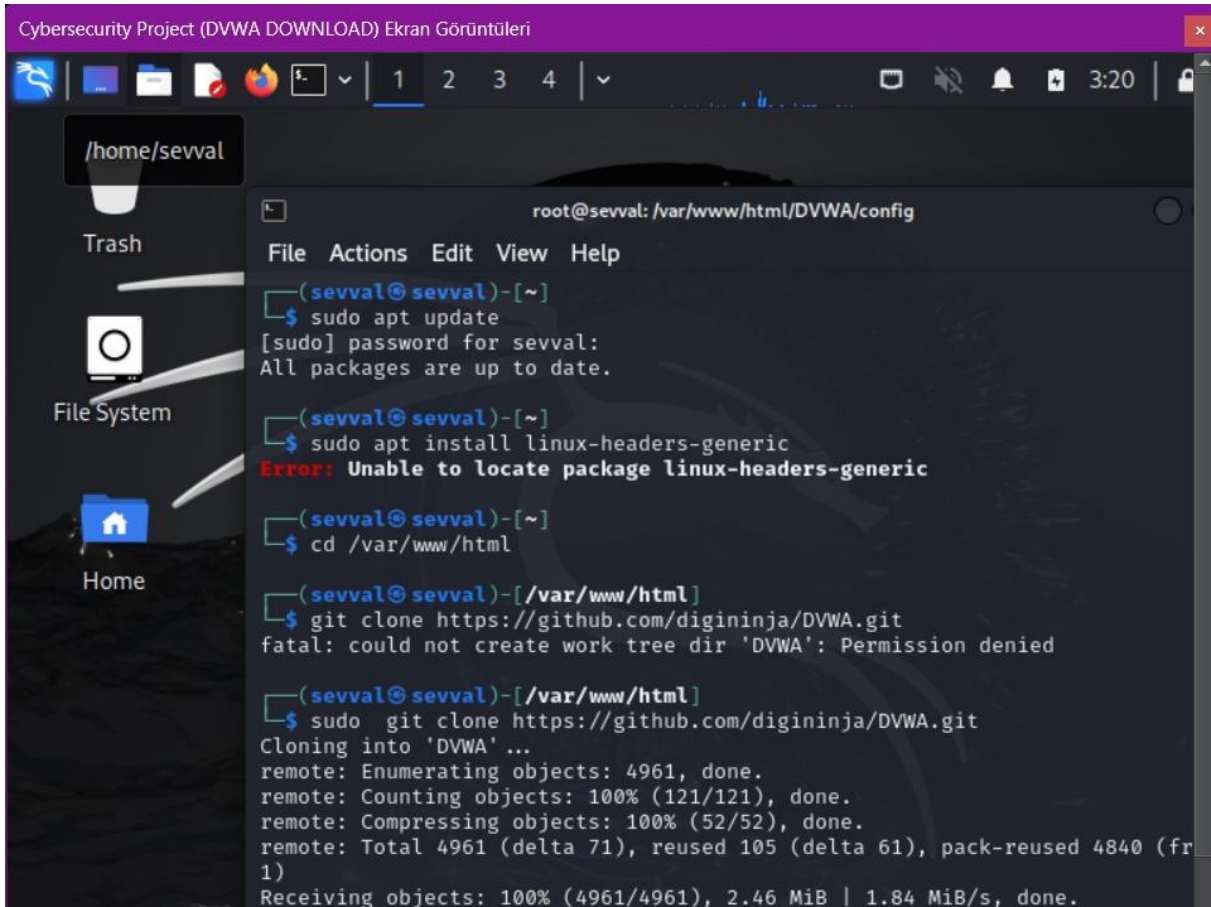


Source: Own Study, as of 14.12.2024

2.2 Installing DVWA

- Downloaded and installed DVWA in Kali Linux by using the GIT repository.

Fig.3 Cloning GIT repository



```
Cybersecurity Project (DVWA DOWNLOAD) Ekran Görüntüleri
root@sevval: /var/www/html/DVWA/config

File Actions Edit View Help

(sevval@sevval)-[~]
$ sudo apt update
[sudo] password for sevval:
All packages are up to date.

(sevval@sevval)-[~]
$ sudo apt install linux-headers-generic
Error: Unable to locate package linux-headers-generic

(sevval@sevval)-[~]
$ cd /var/www/html

(sevval@sevval)-[/var/www/html]
$ git clone https://github.com/digininja/DVWA.git
fatal: could not create work tree dir 'DVWA': Permission denied

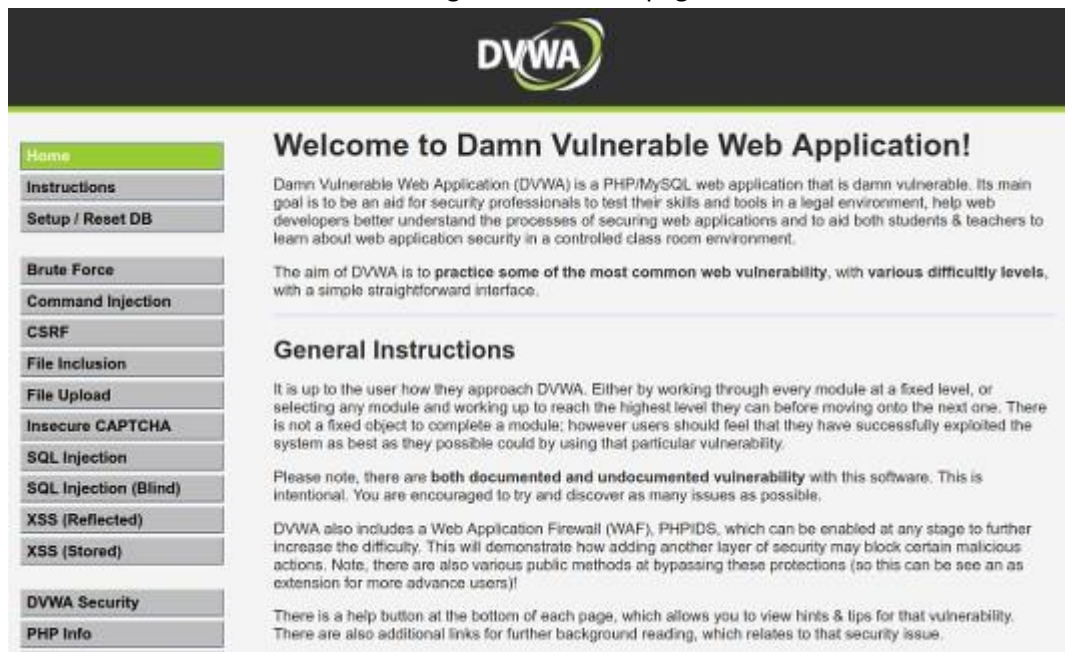
(sevval@sevval)-[/var/www/html]
$ sudo git clone https://github.com/digininja/DVWA.git
Cloning into 'DVWA' ...
remote: Enumerating objects: 4961, done.
remote: Counting objects: 100% (121/121), done.
remote: Compressing objects: 100% (52/52), done.
remote: Total 4961 (delta 71), reused 105 (delta 61), pack-reused 4840 (from 1)
Receiving objects: 100% (4961/4961), 2.46 MiB | 1.84 MiB/s, done.
```

Source: Own Study, as of 14.12.2024

- Configured config.inc.php for database connectivity. Key changes made:
\$_DVWA['db_user'] = 'root';
\$_DVWA['db_password'] = 'your_mysql_password';

- Verified installation by accessing the DVWA homepage:

Fig.5 DVWA Homepage



Source: Own Study, as of 14.12.2024

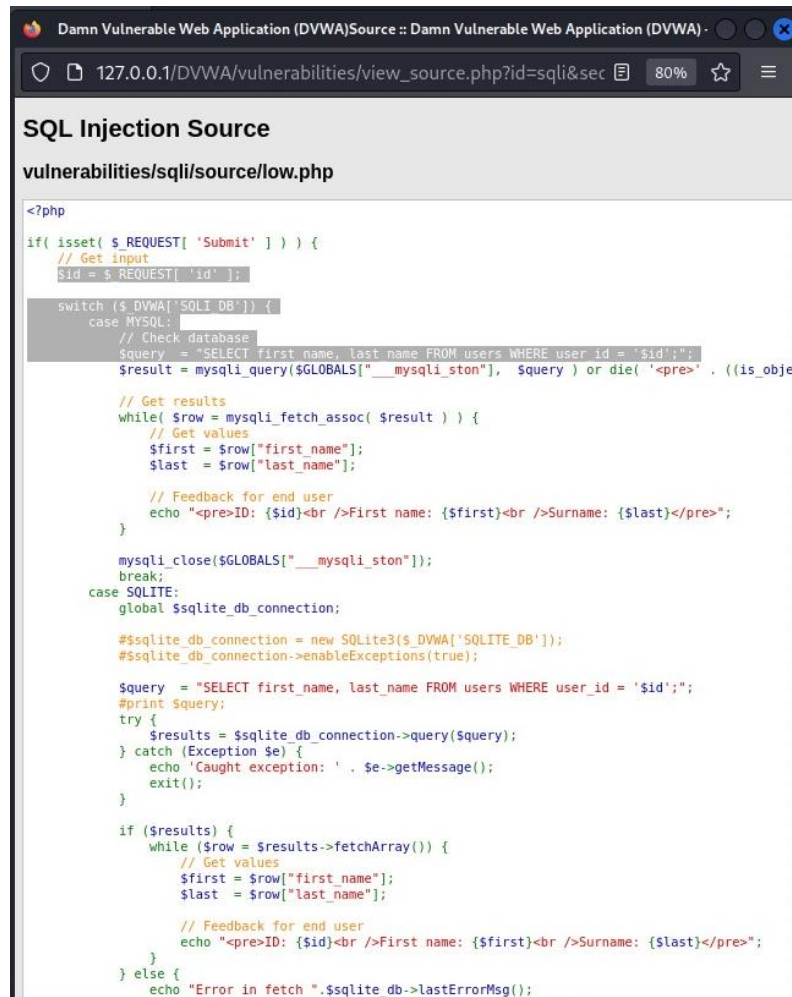
2.3. SQL Injection Simulation

2.3.1 Setting up the Vulnerable Environment For Low Security Level

- Logged in to DVWA using default credentials.
- Set the Security Level to "Low" from the **DVWA Security** settings page.

SQL Injection Source Code Vulnerability Analysis

Fig.6 Low Security Source Code



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

    switch ( $DVWA[ 'SQLI_DB' ] ) {
        case MySQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die( '' );

            // 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>";
            }

            mysqli_close($GLOBALS["__mysqli_ston"]);
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $sqlite_db_connection = new SQLite3($DVWA['SQLITE_DB']);
            $sqlite_db_connection->enableExceptions(true);

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            #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();
            }
        }
    }
}
```

Source:Own Study,as of 14.12.2024

In the sourcecode,we have an SQL Query "\$query = \"SELECT first_name, last_name FROM users WHERE user_id = '\$id';\"; \" So the issue here is there is no validation on the parameter we provide.No validation or sanitization is applied to \$id.

And that means if we're able to insert a quote here we can inject some code into this SQL Statement.

Performing SQL Injection

Basic SQL Injection Attempts

- ' OR '1'='1

Fig.7 Injection Attempt 1

DVWA

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

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://bobby-tables.com/>

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

Username: admin
Security Level: low
Locale: en

Source: Own Study, as of 14.12.2024

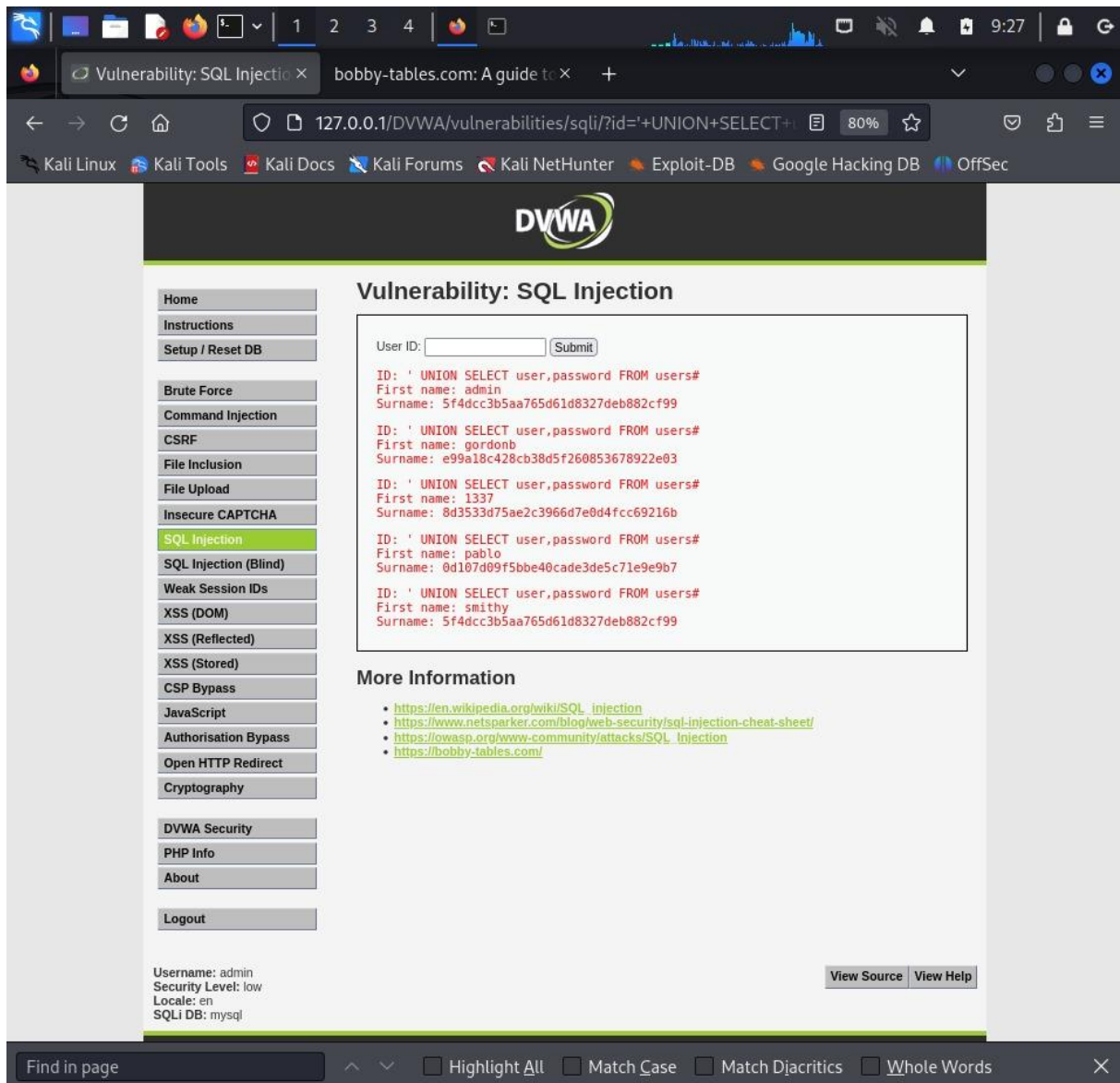
How This Payload Bypasses Authentication?

And now I just injected some code into the statement that says 'If the user ID equals 1 or 1 equals one (which is always true) then bring back the first name and last name.'

So essentially it does not matter whether the username equals 1 or not. It is gonna bring back everything because we have that one equals one.

And after submitting that we get back everything because the second condition always holds true.

Fig.8 Union-Based SQLi



Source: By me,as of 14.12.2024

From the information I got on the website; I tried using an UNION attack to reach to the password informations.

I injected the query "UNION SELECT user,password FROM users#"

Explanation of the Payload

- UNION =keyword enables you to execute one or more additional SELECT queries and append the results to the original query[11].
- user, password= Target columns containing usernames and passwords.
- #=Adding "#" at the end to comment out the rest of the Query to prevent syntax errors.

Finding the Correct Column Names

- The users table and its column names (user and password) were predictable.
- After a few attempts, I successfully identified the correct table and column names

Results

- It's still returning the first name and the surname but instead of surnames it's putting in the user and password from users.

Vulnerability Analysis

This vulnerability is caused by the direct concatenation of user input into the SQL query. Without input validation or using parameterized queries, an attacker can:

- Modify the structure of the query.
- Extract unauthorized data from the database.

Mitigation

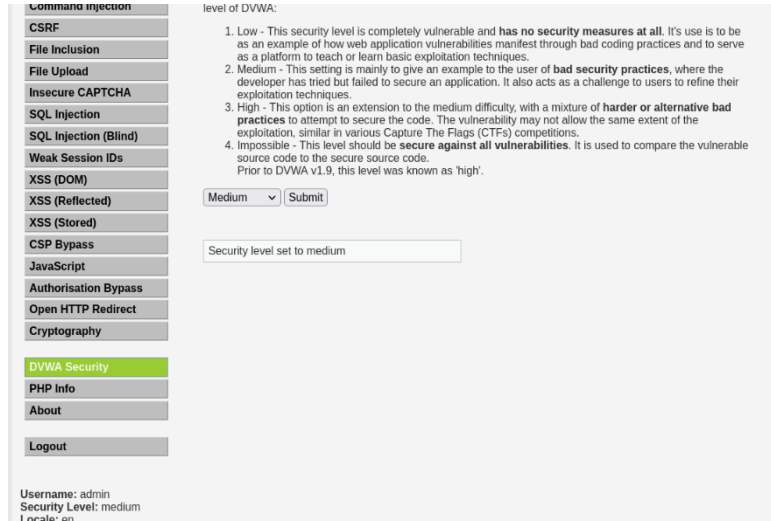
Things to do To avoid this weakness:

- Use prepared statements to separate SQL logic from user input.
- Implement proper input validation.
- Limit the application to return too many rows for authentication queries.

2.3.2 Setting up the Vulnerable Environment For Medium Security Level

- Logged in to DVWA using default credentials.
- Set the Security Level to "Low" from the **DVWA Security** settings page.

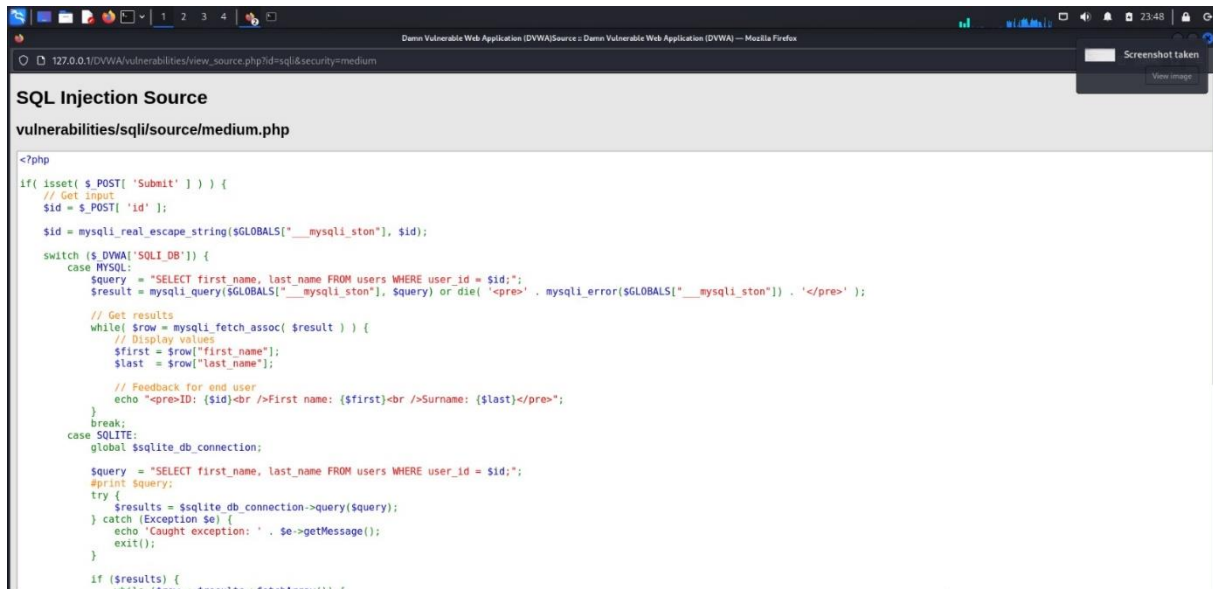
Fig.9 Dwva Medium Security Page



Source: Own Study, as of 15.12.2024

SQL Injection Source Code Vulnerability Analysis

Fig.10 Medium Page Sourcecode

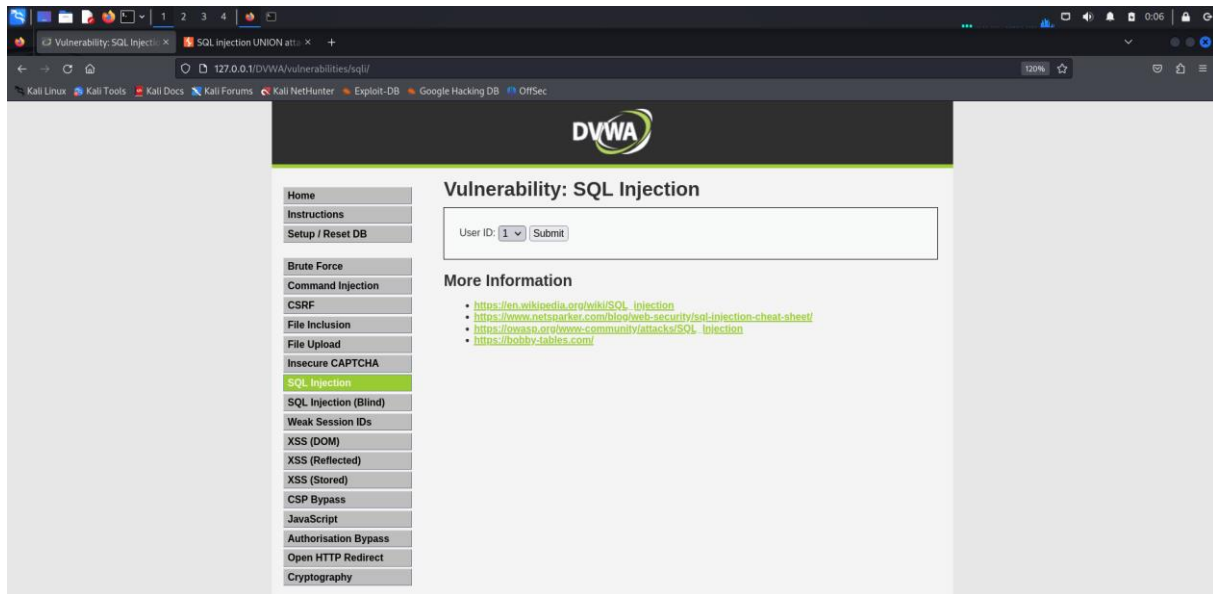


Source: Own Study, as of 15.12.2024

This time with the code we have this post parameter ID and we have the `mysql_real_escape_string()` which is going to escape the strings there. So if we put in a query it is gonna put in a backslash before so that will not be interpreted in the sql statement.

Performing SQL Injection

Fig.11 SQLi Page



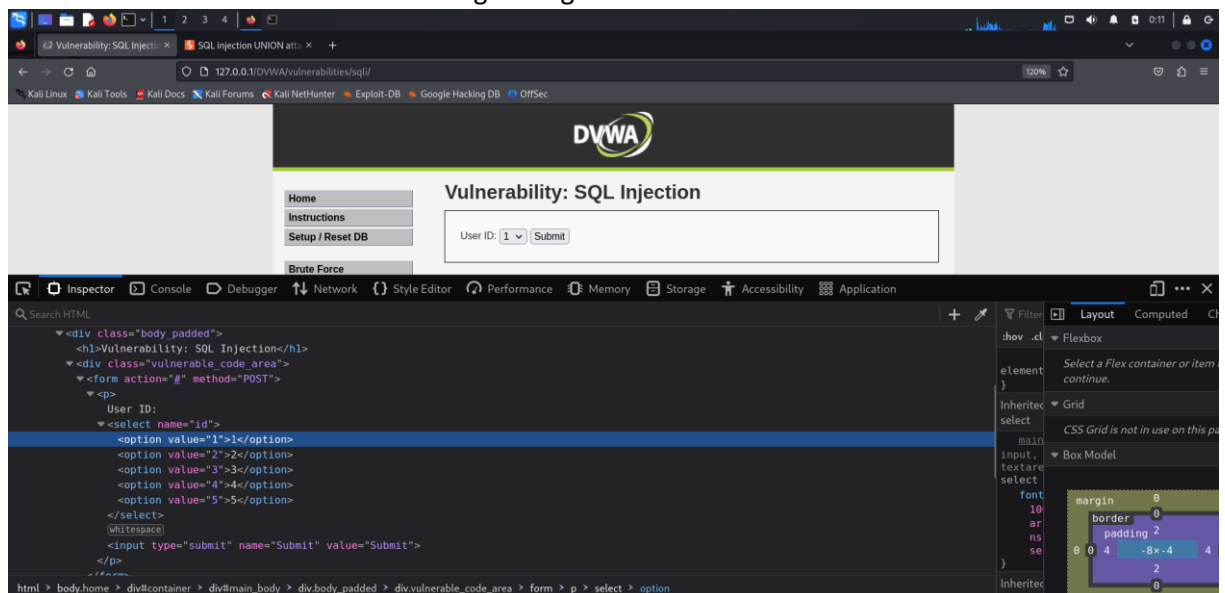
Source: Own Study ,as of 15.12.2024

The **User ID** input is a dropdown list, I won't be able to directly type an SQL injection payload.

How to Bypass Dropdown Restrictions?

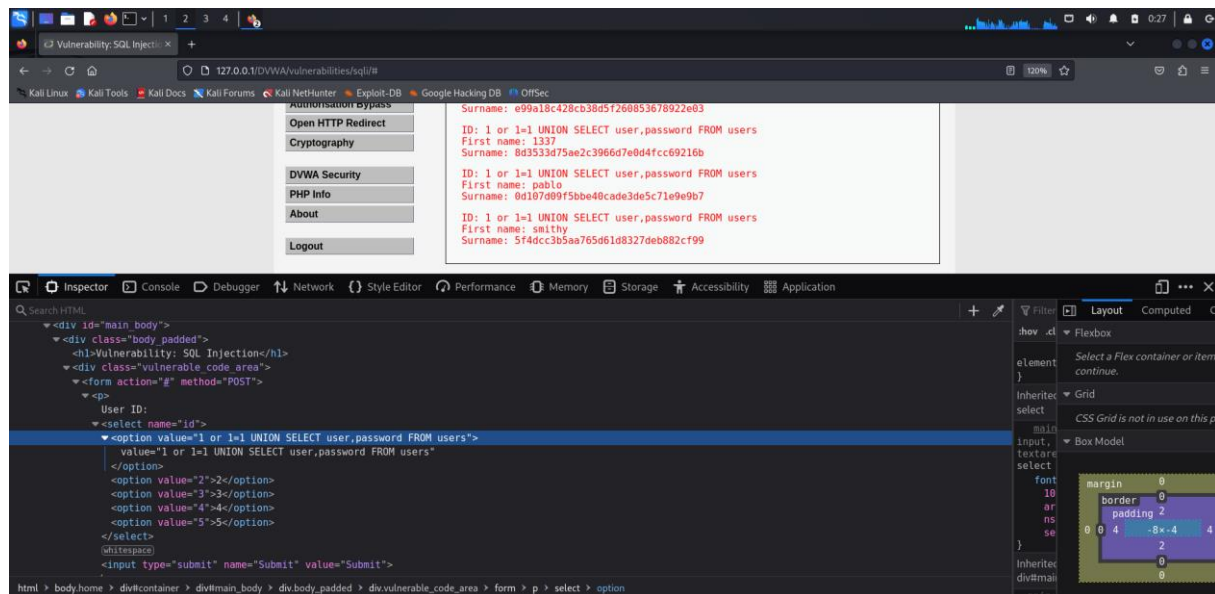
- Inspecting the Page Source

Fig.12 Page Source



Source: Own Study , as of 15.12.2024

Fig.12 Modified Source Page



Source: Own Study as of, 15.12.2024

From the screenshot, it is evident that a successful Union Based SQL Injection attack was executed in the context of the DVWA application. The vulnerable form allows injecting of SQL commands into a specified parameter 'User ID' through drop down list.

Explanation of the Payload

- **1 or 1=1:** This condition always holds true, the effect of which is that the query returns every row in the data set.
- **UNION SELECT user,password FROM users:** The UNION clause is used to join the results of two SELECT statements. In this case, it is used to join the user and password column from the users' table.

Outcome

The server executes the SQL commands that were inserted and returns data that is stored in the users table. In this instance the output revealed the usernames and the passwords which is evidence for the exploitation of SQL Injection. This demonstrates the high risk posed by raw user submissions without adequate cleansing in web applications.

Mitigation

Things to do To avoid this weakness:

- Make use of Prepared Statements (Parameterized Queries).
- Provide Mechanism for Filtering Bad Inputs.
- Apply Stored Procedures when Necessary.

3 Preparing an experiment to verify the effectiveness of protection

To verify the effectiveness of SQL Injection protection, the following experiment was conducted using **Burp Suite** and **DVWA**.

Tools Used

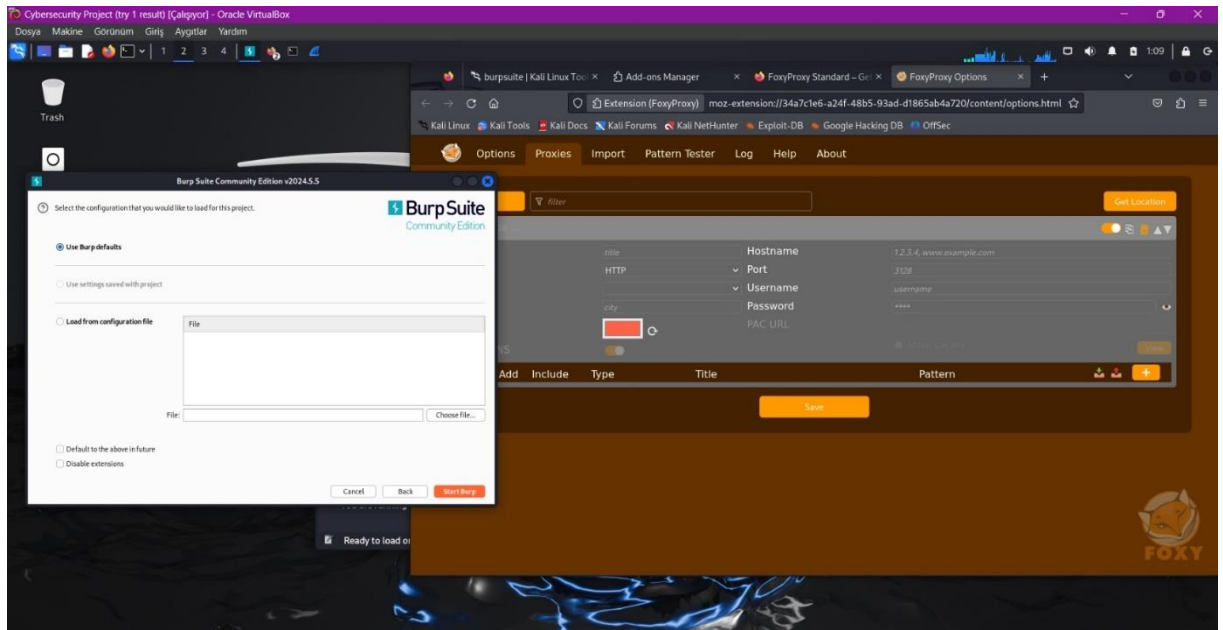
- **Burp Suite:** For intercepting, analyzing, and modifying HTTP requests.
- **FoxyProxy:** To direct browser traffic through Burp Suite.
- **DVWA (Damn Vulnerable Web Application):** The testing environment.

Steps

To verify the authenticity of the protection mechanisms that were applied regarding SQL Injection on different security levels in DVWA, the following steps were performed.

1 Configured Burp Suite to intercept traffic using FoxyProxy.

Fig.13 Configuring Burp Suite



Source: Own Study, as of 15.12.2024

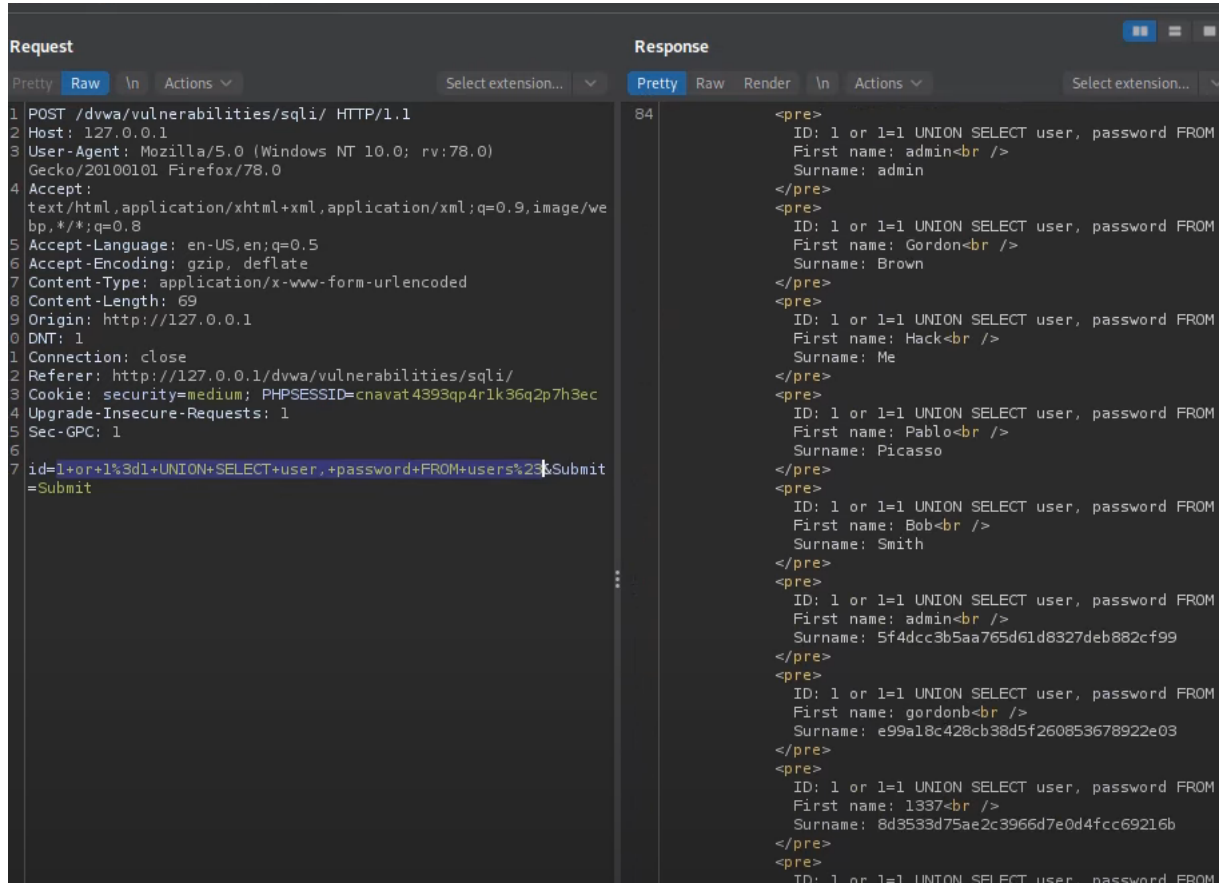
Burp Suite was launched to act as an intercepting proxy.

FoxyProxy was configured in the browser to forward all HTTP requests to Burp Suite (127.0.0.1:8080).

2. Tested SQL Injection Payloads

- Navigated to the **SQL Injection** page in DVWA.
- Used basic and advanced SQL Injection payloads, such as
 - „ 1' OR '1'='1 -- 1' UNION SELECT”
 - „ user,password FROM users --”

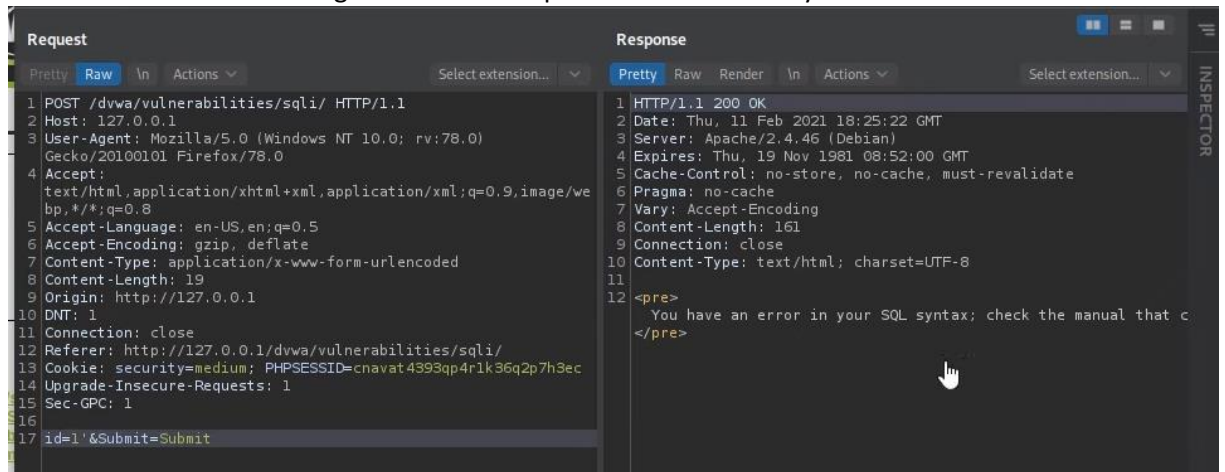
Fig.14 Responses in Burp Suite



Source: Own Study, as of 15.12.2024

- Tested the payloads at **Low**, **Medium**, and **High** security levels to observe the behavior.

Fig.15 Failed Attempt on Medium Security



Source: Own Study, as of 15.12.2024

2. Observed Server Responses for Evidence of Successful or Blocked Injections

Analyzed the intercepted HTTP requests and responses in Burp Suite:

Low Security: SQL Injection succeeded, sensitive data - usernames/passwords - retrieved.

Medium Security: Filtering was only partially implemented, and still possible with crafted payloads to bypass it.

High Security: SQL Injection attempts were blocked, and no data was leaked.

The protection effectiveness could be measured by server responses; for example, an error message or a successful query result.

Results

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.

This experiment demonstrated that as the security level in DVWA increases, the effectiveness of SQL Injection protection also improves. At **High Security**, proper input sanitization and protection mechanisms effectively prevent SQL Injection attacks.

Conclusions

SQL injection is still among the top vulnerabilities in web application security, and thus it creates serious threats for organizations and individuals alike. Here, the mechanism, classification, and real-life implications of SQL injection attacks have been discussed in detail, showing just how easily an attacker can play around with poorly secured systems. These vulnerabilities, ranging from very simple injections through cookies to advanced, time-based, and boolean-based attacks, have eventually resulted in unauthorized access to website information, data theft, and the compromise of crucial databases, as was in the case of Sony Pictures.

In fact, this report was able to show, through practical simulations using the DVWA platform, the real feasibility of such an attack under different security configurations. Indeed, at a low security level, basic SQL injection payloads succeed in bypassing authentication and extracting sensitive information. The introduction of partial countermeasures in the medium security settings can, however, be easily evaded by an advanced payload. Only at high-security levels, when enhanced input sanitization and parameterized queries are implemented, were the SQL injection attempts effectively foiled.

The findings point out the need to adhere to best practices that could help in countering such threats, including the following:

- 1. Implementation of Prepared Statement and Parameterized Query:** These techniques strictly separate SQL logic from user inputs, rendering malicious payloads innocuous.
- 2. Stringent Input Validation:** Filtering and validation of user inputs will help organizations get rid of entry points for SQL injection attacks.
- 3. Using Stored Procedures and Limiting Privileges:** These strategies minimize the attack surface and reduce the potential impact of successful intrusions.

As web applications continue to grow and evolve in complexity, developers and organizations alike must implement security measures to keep sensitive data secure and protect the integrity of the systems. This report makes it clear that SQL injection, though a powerful threat, is also one that can be prevented by adopting robust development practices, continuous security testing, and awareness of emerging attack techniques.

Understanding the vulnerabilities that SQL injection exploits, together with recommended countermeasures, forms a basis for strengthening organizational defenses and reducing the probability of becoming victims of such attacks.

List of Sources

- [1] Clarke-Salt, J. (2009). SQL injection attacks and defense. Elsevier. 01.12.2024
- [2] *Siboni, Gabi; Siman-Tov, David (December 23, 2014). [Cyberspace Extortion: North Korea versus the United States](#) (PDF) (Report). [INSS. Archived](#) (PDF) from the original on August 20, 2016. Retrieved March 23, 2023.* 01.12.2024
- [3] Kapoor, A. (2023). SQL-Injection Threat Analysis and Evaluation. Available at SSRN 4430812. 01.12.2024
- [4] M. Dornseif. Common Failures in Internet Applications, May 2005. <http://md.hudora.de/presentations/> 01.12.2024
- [5] T. M. D. Network. Request.servervariables collection. Technical report, Microsoft Corporation, 2005. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/9768ecfe-8280-4407-b9c0-844f75508752.asp> 01.12.2024
- [6] Halfond, W. G., Viegas, J., & Orso, A. (2006, March). A Classification of SQL Injection Attacks and Countermeasures. In *ISSSE*. 01.12.2024
- [7] Quatrini, S., & Rondini, M. (2011, December). *Blind Sql Injection with Regular Expressions Attack*. 01.12.2024
- [8] <https://beaglesecurity.com/blog/vulnerability/boolean-based-blind-sqli.html> 01.12.2024
- [9] <https://medium.com/@vikramroot/exploiting-time-based-sql-injections-data-exfiltration-791aa7f0ae87> 01.12.2024
- [10] <https://www.cvedetails.com/cve/CVE-2024-1100/> 01.12.2024
- [11] <https://portswigger.net/web-security/sql-injection/union-attacks> 15.12.2024