**Web Application Security Testing Report**

**Organization:** ApexPlanet Software Pvt. Ltd.

**Project Title:** Web Application Security – OWASP Top 10 (DVWA Lab)

**Test Environment:**

* Operating System: Kali Linux
* Target Application: Damn Vulnerable Web Application (DVWA)
* Web Server: Apache
* Database: MySQL / MariaDB
* Tools Used: Burp Suite Community, Firefox Browser

**1. Objective**

The objective of this project is to identify, exploit, and mitigate common web application vulnerabilities based on the OWASP Top 10 using a controlled lab environment (DVWA). The assessment demonstrates both attack techniques and secure coding practices.

**2. Scope**

The scope of testing is limited to DVWA hosted locally on Kali Linux. No external or real-world systems were tested.

**3. Vulnerability Assessment & Findings**

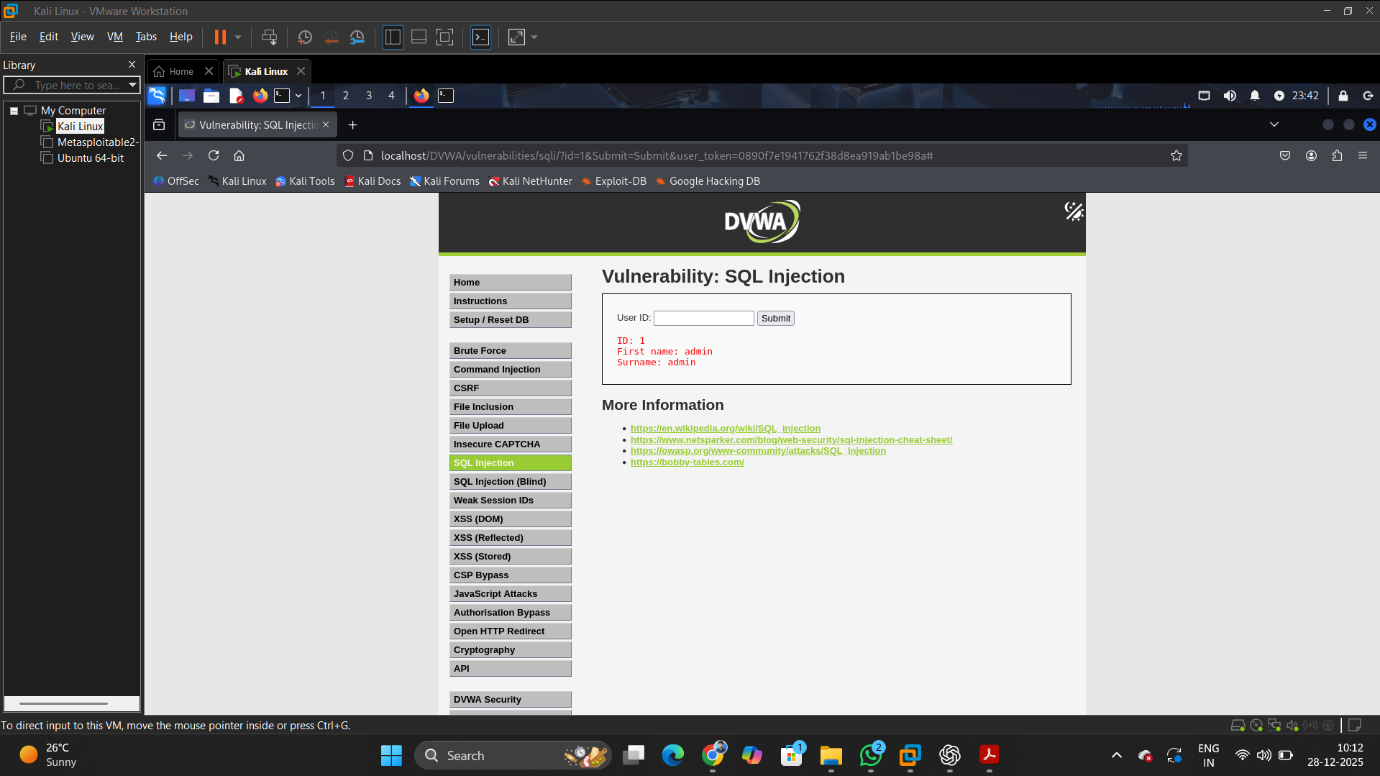
**3.1 SQL Injection (SQLi)**

**Description**

SQL Injection occurs when user input is improperly handled and directly embedded into SQL queries, allowing attackers to manipulate database queries.

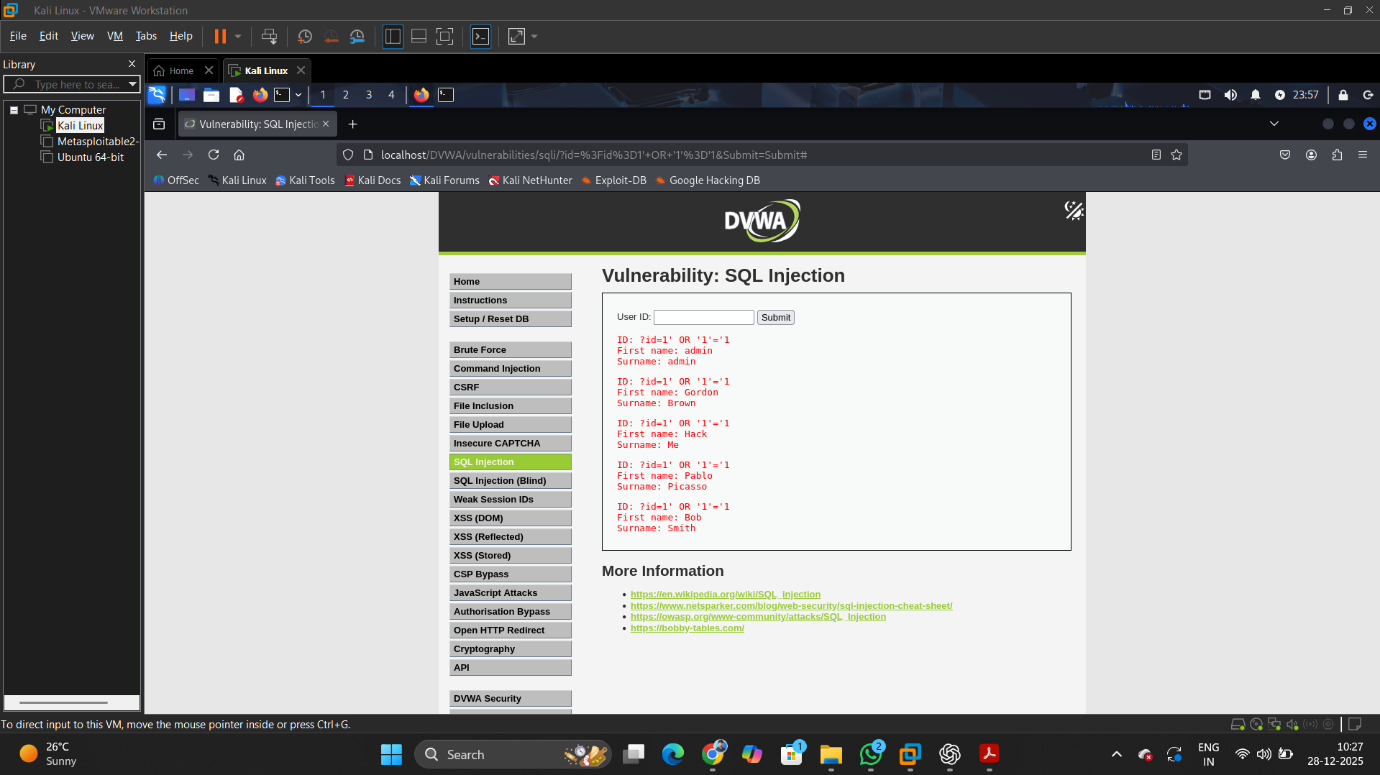
**Vulnerable Module**

DVWA → SQL Injection (Security Level: Low)

****

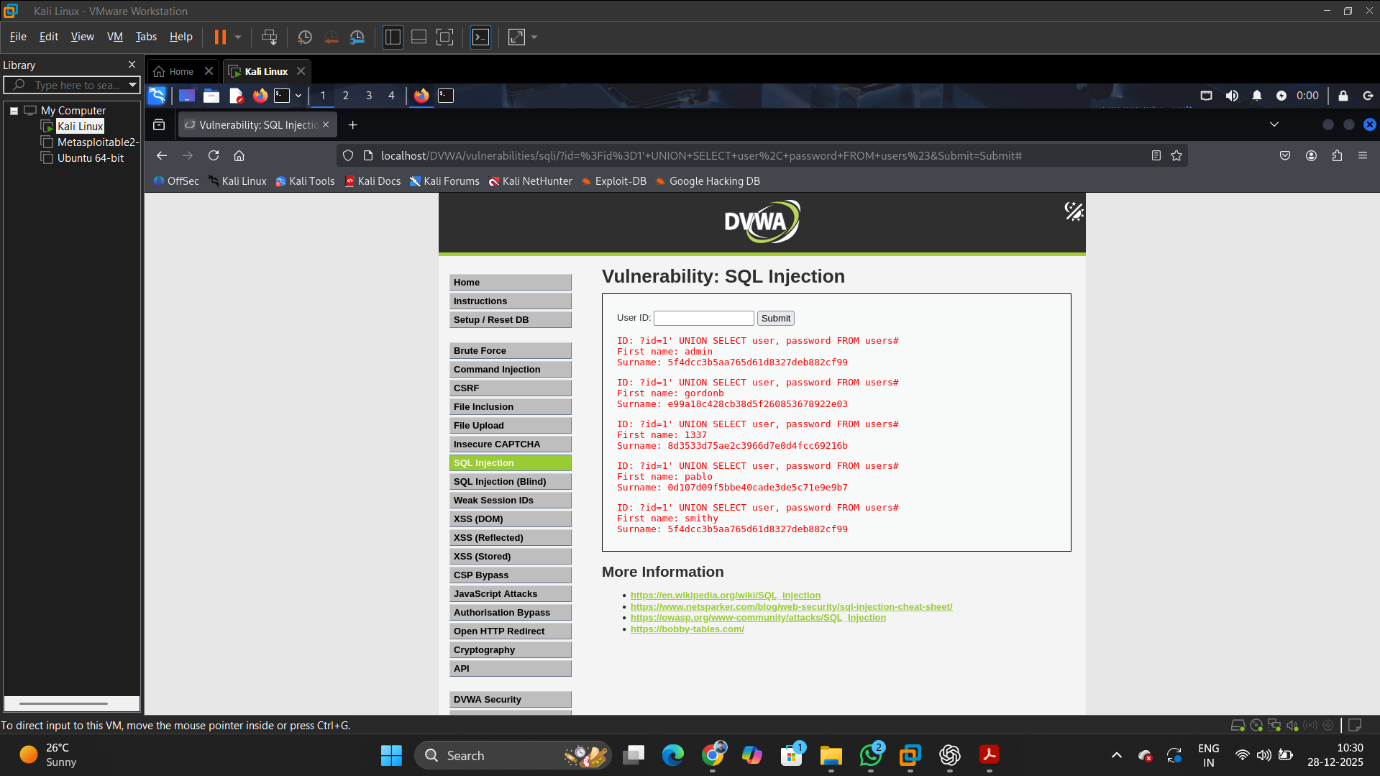
**Screenshot 1:** SQL Injection module with normal input (1)

* The application returns details of a single user (admin).
* This confirms expected behavior when valid input is provided.

****

**Screenshot 2:** SQL Injection using payload 1 OR 1=1

* Payload entered:
* 1 OR 1=1
* The condition always evaluates to TRUE.
* The database returns multiple user records instead of one.

****

**Screenshot 3:** SQL Injection using UNION-based payload

* Payload entered:
* 1' UNION SELECT user, password FROM users#
* Usernames and password hashes are successfully extracted from the database.
* Passwords are displayed in MD5 hashed format.

**Impact**

* Unauthorized access to database records
* Disclosure of usernames and password hashes
* Full compromise of application data confidentiality

**Vulnerable Code**

$query = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id'";

**Reason for Vulnerability:**

* User input is directly concatenated into the SQL query.
* No input validation or parameter binding is applied.

**Mitigation (Prepared Statements)**

$stmt = $conn->prepare("SELECT first\_name, last\_name FROM users WHERE user\_id = ?");

$stmt->bind\_param("i", $id);

$stmt->execute();

**Why This Fix Works:**

* Prepared statements separate SQL logic from user input.
* The database treats user input strictly as data.

**Result After Fix**

* SQL injection payloads no longer execute
* Malicious inputs such as OR 1=1 are ignored
* Only valid numeric User IDs return results

**3.2 Cross-Site Scripting (XSS)**

**Description**

Cross-Site Scripting (XSS) is a web application vulnerability that allows attackers to inject malicious client-side scripts into web pages viewed by other users. These scripts execute in the victim’s browser and can steal cookies, session tokens, or perform actions on behalf of the victim.

**DVWA demonstrates three major types of XSS vulnerabilities:**

* Reflected XSS
* Stored XSS
* DOM-based XSS

All tests were performed with DVWA Security Level: Low.

**3.2.1 Reflected Cross-Site Scripting (Reflected XSS)**

**Vulnerable Module**

DVWA → XSS (Reflected)

**Description**

Reflected XSS occurs when user input is immediately returned by the server in an HTTP response without proper validation or encoding. The malicious script is not stored permanently and executes only when the victim clicks a crafted URL or submits malicious input.

**Attack Steps**

1. Navigate to DVWA → XSS (Reflected).
2. Enter normal input to observe expected behavior.
3. Inject the following payload into the input field:

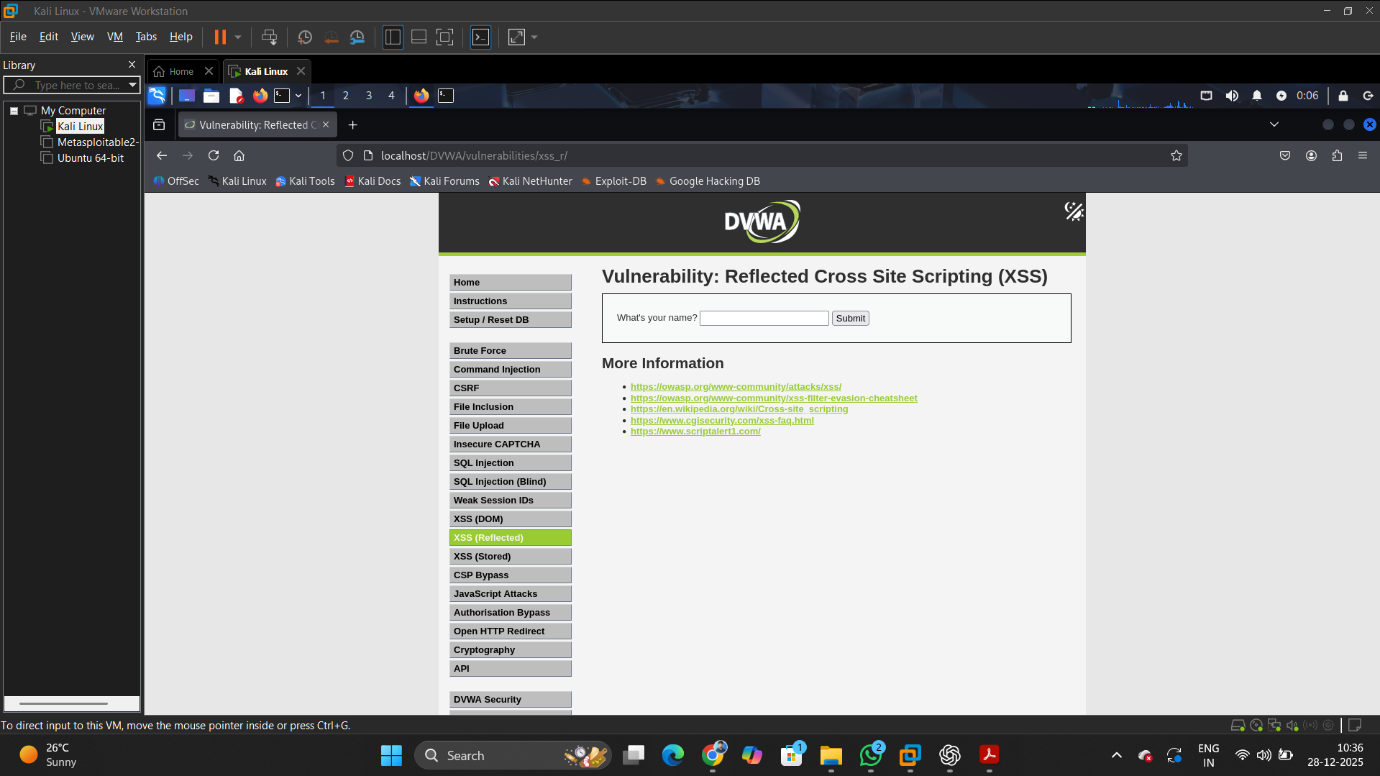
<script>alert('Reflected XSS')</script>

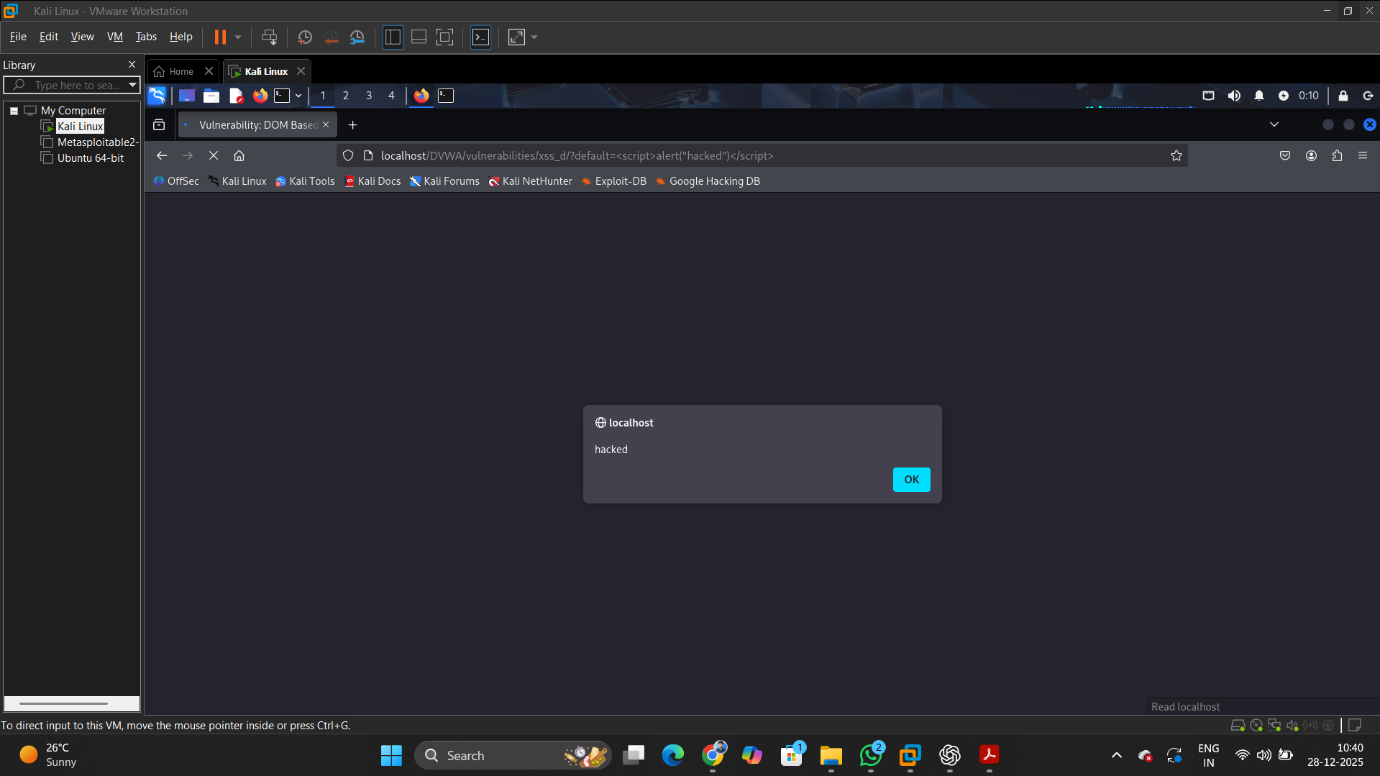
1. Submit the request.

**Observed Result**

* A JavaScript alert box appears in the browser.
* The script executes immediately, confirming reflected XSS.

**Screenshot Evidence**

****

****

**Impact**

* **Session hijacking**
* **Phishing attacks**
* **Malicious redirects**

**3.2.2 Stored Cross-Site Scripting (Stored XSS)**

**Vulnerable Module**

DVWA → XSS (Stored)

**Description**

Stored XSS occurs when malicious input is permanently stored on the server (e.g., in a database) and later displayed to users without sanitization. Every user who accesses the affected page executes the injected script.

**Attack Steps**

1. Navigate to DVWA → XSS (Stored).
2. Enter the following payload in the message or comment field:

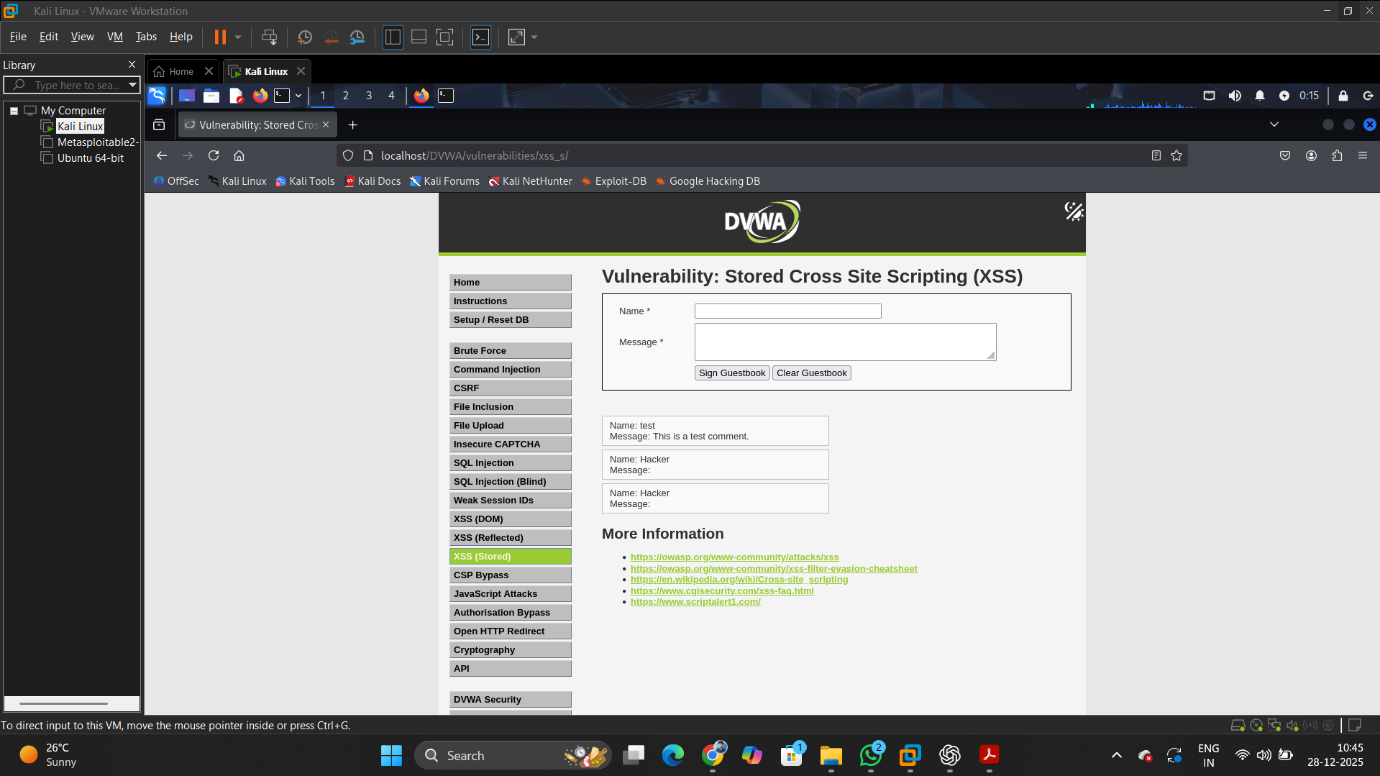
<script>alert('Stored XSS')</script>

1. Submit the form.
2. Refresh the page or access it from another browser session.

**Observed Result**

* The alert box appears every time the page is loaded.
* The script persists until the database is cleared.

**Screenshot Evidence**

****

**Impact**

* **Persistent compromise of users**
* **Cookie theft**
* **Full account takeover**

**3.2.3 DOM-Based Cross-Site Scripting (DOM XSS)**

**Vulnerable Module**

DVWA → XSS (DOM)

**Description**

DOM-based XSS occurs entirely on the client side. The vulnerability exists in JavaScript code that dynamically updates the DOM using untrusted input without proper sanitization. The server is not directly involved in rendering the malicious script.

**Attack Steps**

1. Navigate to DVWA → XSS (DOM).
2. Modify the URL parameter with the following payload:

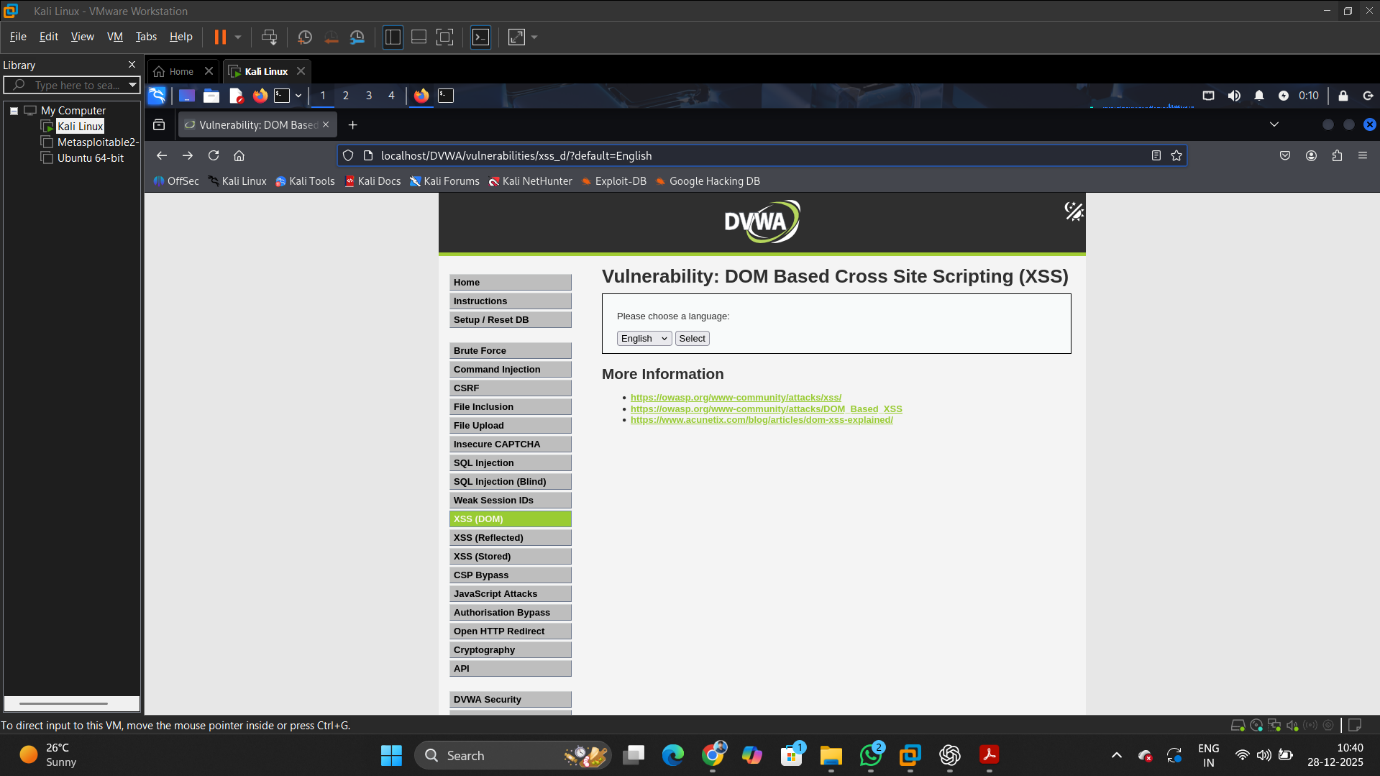
#<script>alert('DOM XSS')</script>

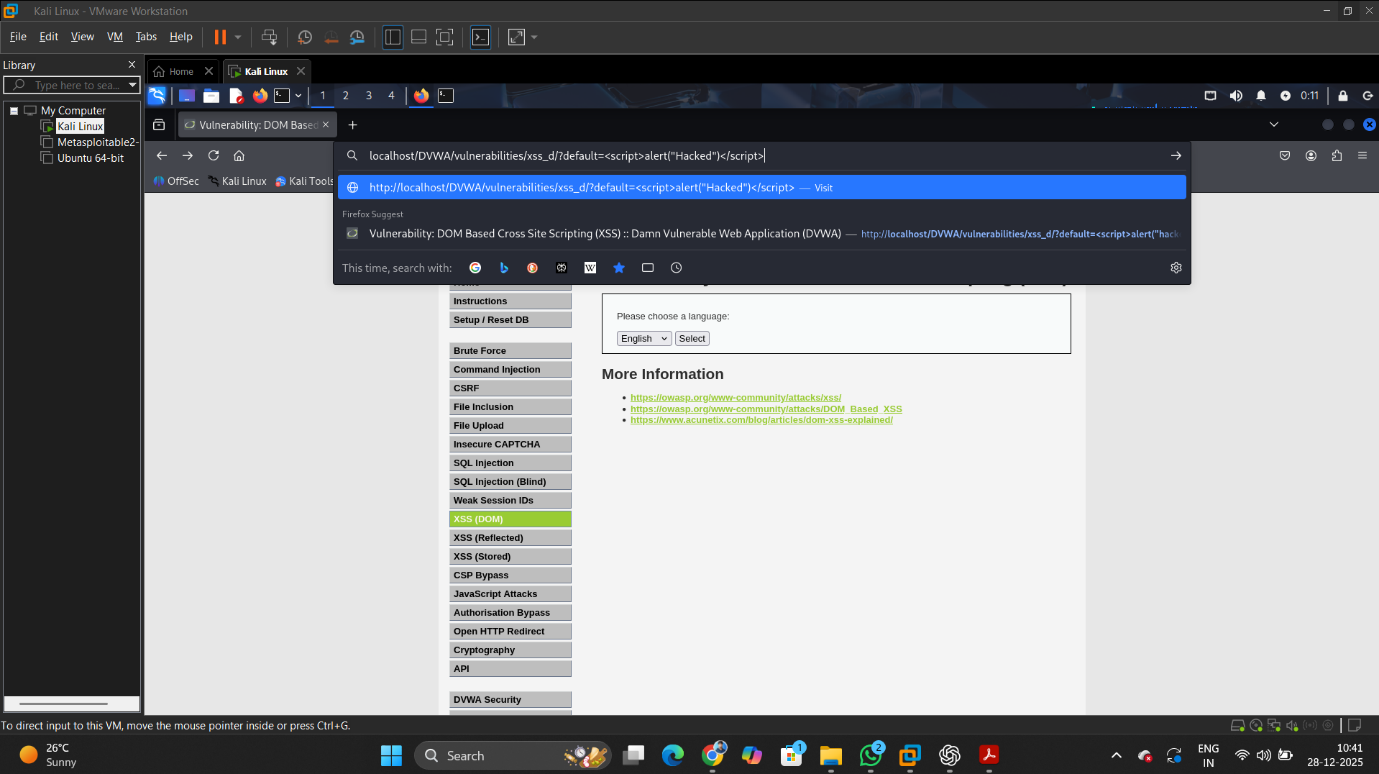
1. Press Enter to load the modified URL.

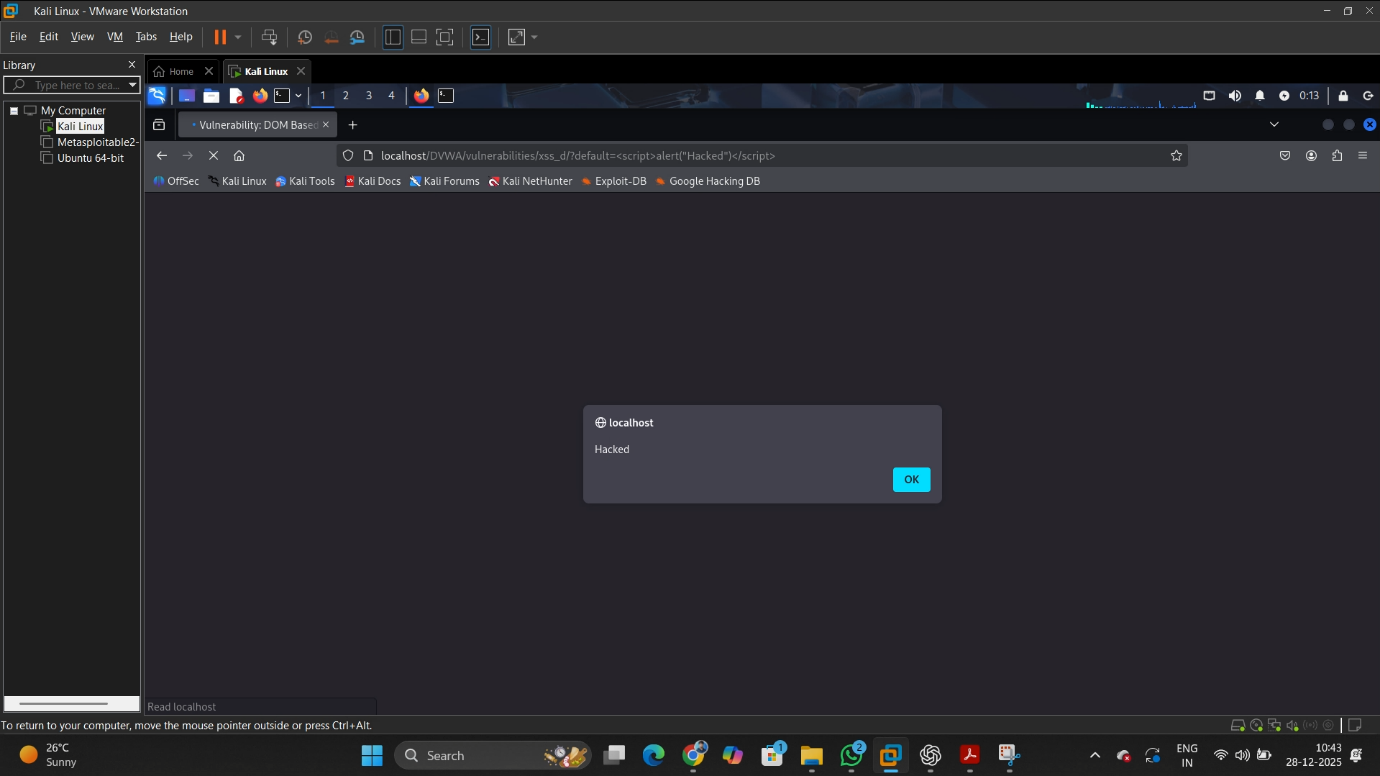
**Observed Result**

* JavaScript alert executes immediately.
* No request is sent to the server containing the payload.

**Screenshot Evidence**

****

****

****

**Impact**

* **Bypasses server-side security controls**
* **Executes malicious scripts directly in browser**

**Mitigation Techniques**

* **Input validation and sanitization**
* **Output encoding (HTML, JavaScript, URL encoding)**
* **Use of Content Security Policy (CSP)**
* **Avoid dangerous JavaScript functions such as document.write()**
* **Use secure frameworks and templating engines**

**Secure Code Example**

**echo htmlspecialchars($input, ENT\_QUOTES, 'UTF-8');**

**3.3 Cross-Site Request Forgery (CSRF)**

**Description**

Cross-Site Request Forgery (CSRF) is a web security vulnerability that forces an authenticated user to execute unwanted actions on a web application without their knowledge. It occurs when an application does not verify whether a request was intentionally made by the legitimate user.

In DVWA, the CSRF module demonstrates how an attacker can change a logged-in user’s password by tricking the browser into sending a crafted request.

**Vulnerable Module**

DVWA → CSRF (Security Level: Low)

**Attack Objective**

To change the administrator password without manually submitting the password change form, by reusing an authenticated request captured via Burp Suite.

**Tools Used**

* Kali Linux
* DVWA
* Burp Suite Community Edition
* Firefox Browser

**Attack Steps**

1. Log in to DVWA as admin.
2. Set DVWA security level to Low.
3. Navigate to DVWA → CSRF module.
4. Open Burp Suite and enable Intercept under the Proxy tab.
5. In the browser, submit the password change form with a new password.
6. Burp Suite intercepts the HTTP request containing password parameters.
7. Observe that the request is a simple GET request with parameters:
   * password\_new
   * password\_conf
   * Change=Change
8. Forward the request in Burp Suite.
9. The password is successfully changed without any CSRF token validation.

**Observed Request (Captured via Burp Suite)**

GET /DVWA/vulnerabilities/csrf/?password\_new=newpass&password\_conf=newpass&Change=Change HTTP/1.1

Host: localhost

Cookie: PHPSESSID=xxxxxxxx

This request proves that the application trusts any authenticated request without verifying its origin.

**Result**

* The admin password is changed successfully.
* No CSRF token is required.
* No origin or referrer validation is performed.

**Impact**

* Unauthorized account takeover
* Password changes without user consent
* Potential privilege escalation

**Vulnerable Code**

if( isset( $\_GET[ 'Change' ] ) ) {

$pass\_new = $\_GET[ 'password\_new' ];

$pass\_conf = $\_GET[ 'password\_conf' ];

if( $pass\_new == $pass\_conf ) {

$hashed = md5( $pass\_new );

$query = "UPDATE users SET password = '$hashed' WHERE user = 'admin'";

mysqli\_query( $conn, $query );

}

}

**Mitigation Techniques**

1. Use CSRF Tokens
   * Generate a unique token per session.
   * Validate the token on every state-changing request.
2. Use POST Requests instead of GET for sensitive operations.
3. Validate Origin and Referer Headers.
4. Enable SameSite Cookies to restrict cross-site requests.

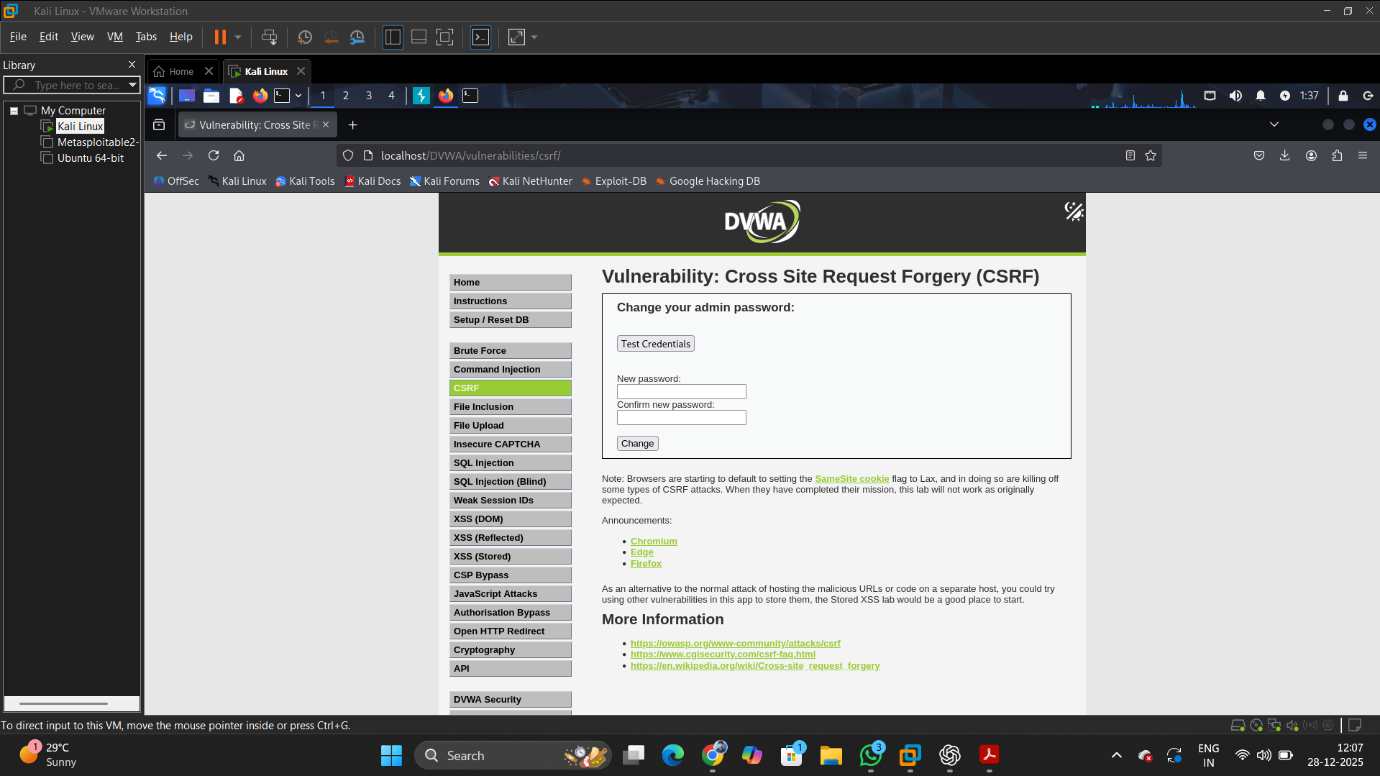
**Secure Code Example**

if ($\_POST['csrf\_token'] !== $\_SESSION['csrf\_token']) {

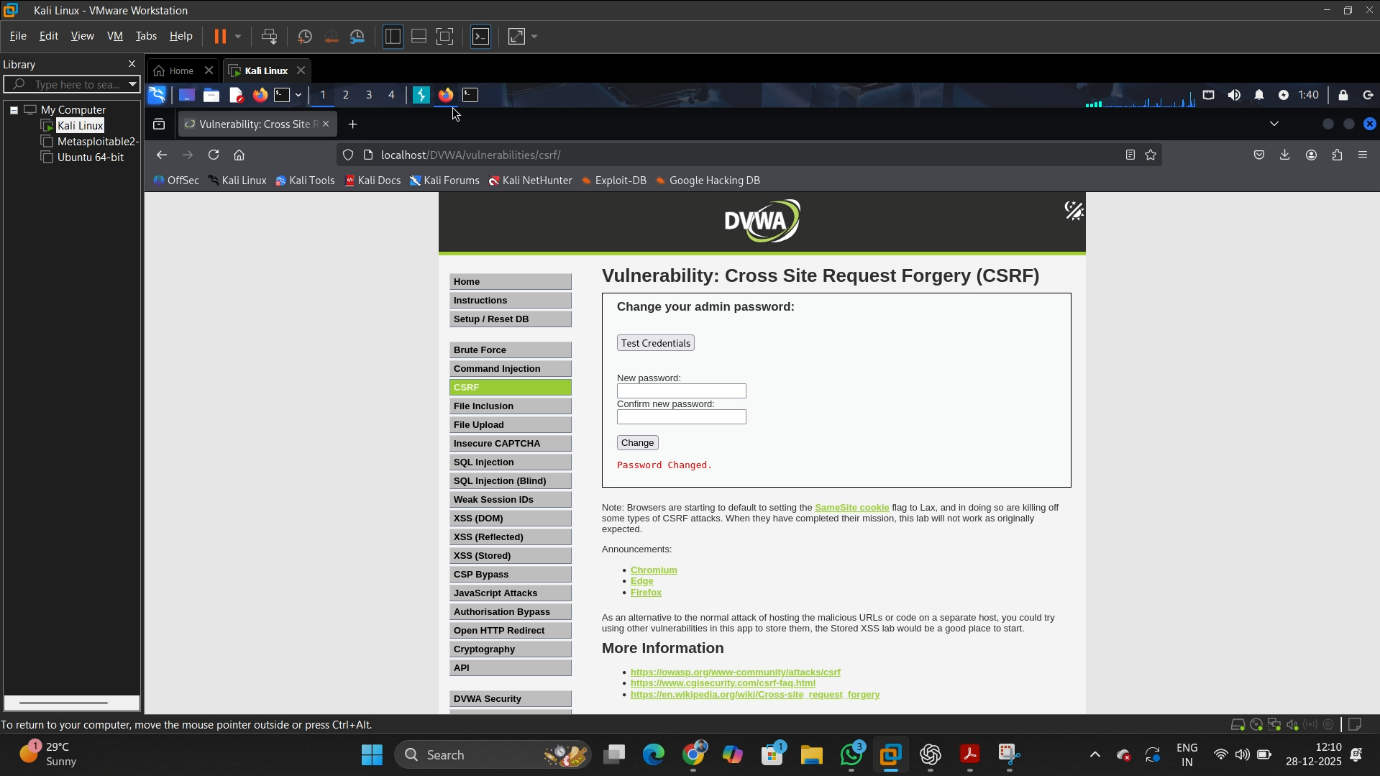
die("Invalid CSRF token");

}

**Screenshot Evidence**

****

****

****

**3.4 File Inclusion Attacks**

**Description**

File Inclusion vulnerabilities occur when an application dynamically includes files based on user-supplied input without proper validation. Attackers can manipulate file paths to include unintended local or remote resources.

In DVWA, the File Inclusion module demonstrates how improper handling of file parameters can lead to Local File Inclusion (LFI) and Remote File Inclusion (RFI).

**Vulnerable Module**

DVWA → File Inclusion (Security Level: Low)

**Attack Objective**

To manipulate the page parameter to include an unauthorized remote file, demonstrating how external content can be loaded and executed by the application.

**Attack Steps (RFI Demonstration)**

1. Log in to DVWA and set Security Level: Low.
2. Navigate to DVWA → File Inclusion module.
3. Observe the URL parameter:

page=include.php

1. Modify the parameter to include a remote resource:

page=www.google.com

1. Submit the request.
2. The external page content loads successfully inside the DVWA application, confirming Remote File Inclusion vulnerability.

**Observation**

The application fails to restrict external URLs in the file inclusion logic. As a result, remote resources can be loaded, proving that input is not validated or sanitized.

**Impact**

**•** Disclosure of sensitive local files (LFI)  
• Inclusion of malicious remote scripts (RFI)  
• Potential remote code execution  
• Full server compromise if attacker-controlled scripts are executed

**Vulnerable Code Example**

$page = $\_GET['page'];

include($page);

**Mitigation Techniques**

• Disable allow\_url\_include and allow\_url\_fopen in php.ini  
• Use a whitelist of allowed files  
• Avoid dynamic file inclusion using user input  
• Strictly validate and sanitize parameters  
• Use absolute paths instead of user-controlled values

**Secure Code Example**

$allowed\_pages = ['home.php', 'about.php'];

if (in\_array($page, $allowed\_pages)) {

include($page);

} else {

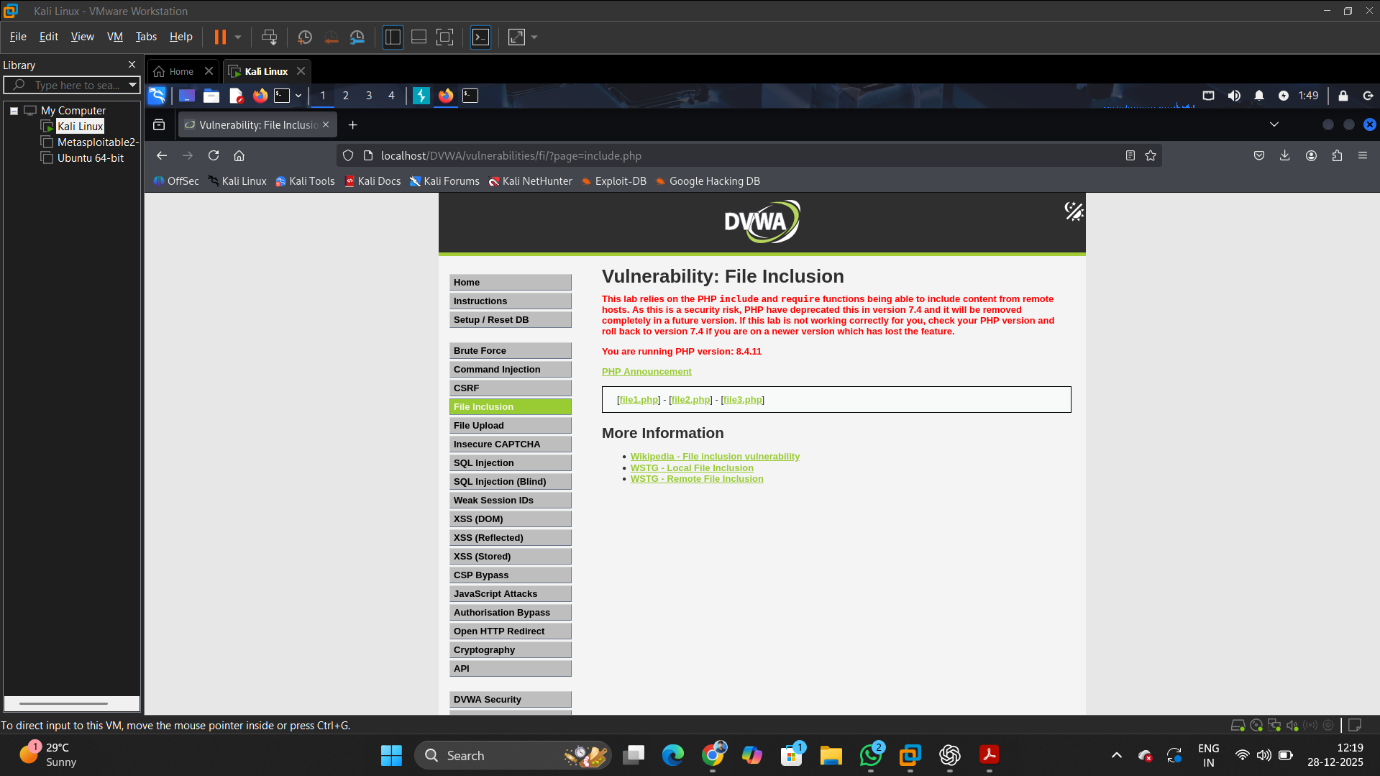
echo "Invalid page request";

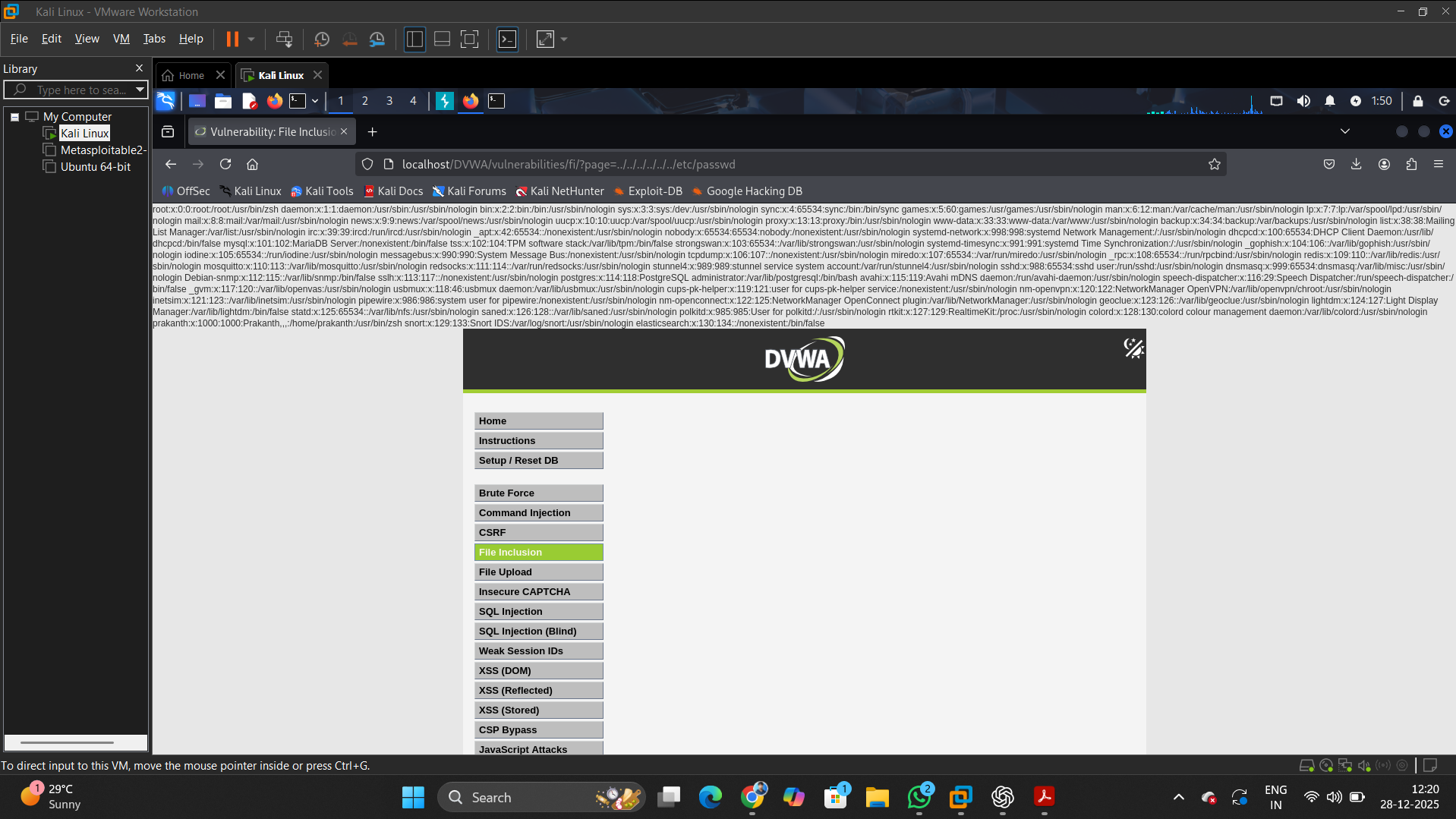
}

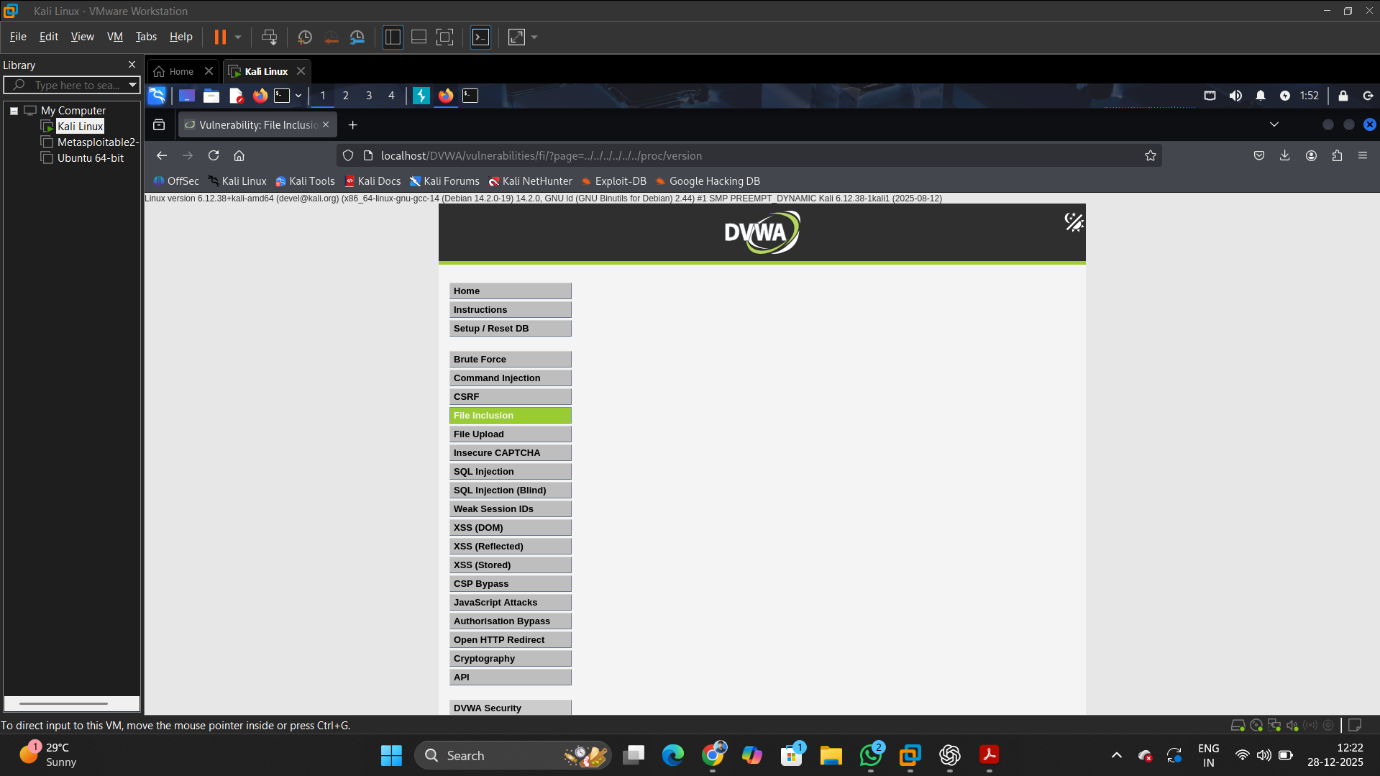
**Result After Fix**

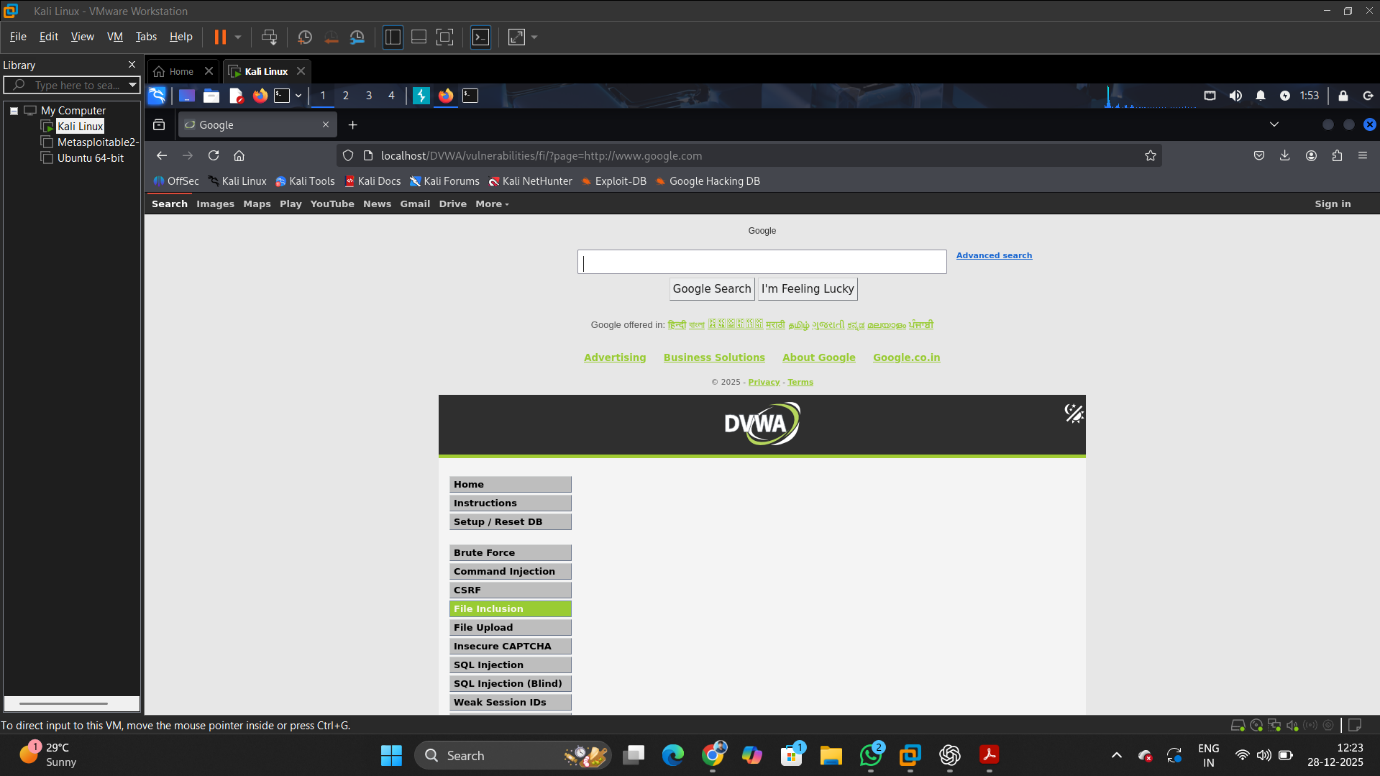
• Remote URLs are blocked  
• Only predefined files can be included  
• File Inclusion attacks no longer succeed

**Screenshot Evidence**

****

****

****

****

**3.5 Burp Suite Advanced Testing**

**Intercepting Requests**

* Burp Suite proxy was configured with Firefox.
* DVWA login requests were intercepted and modified.

**Intruder Attack**

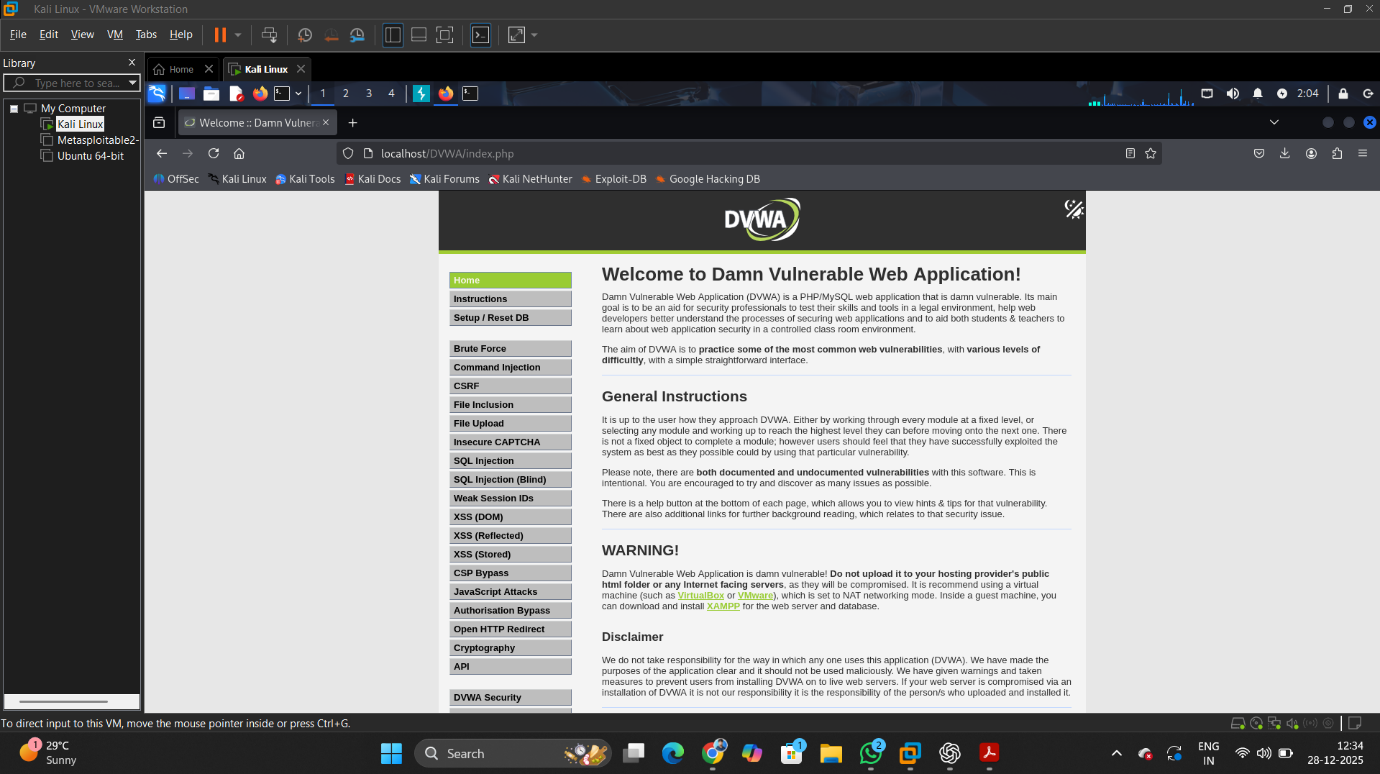
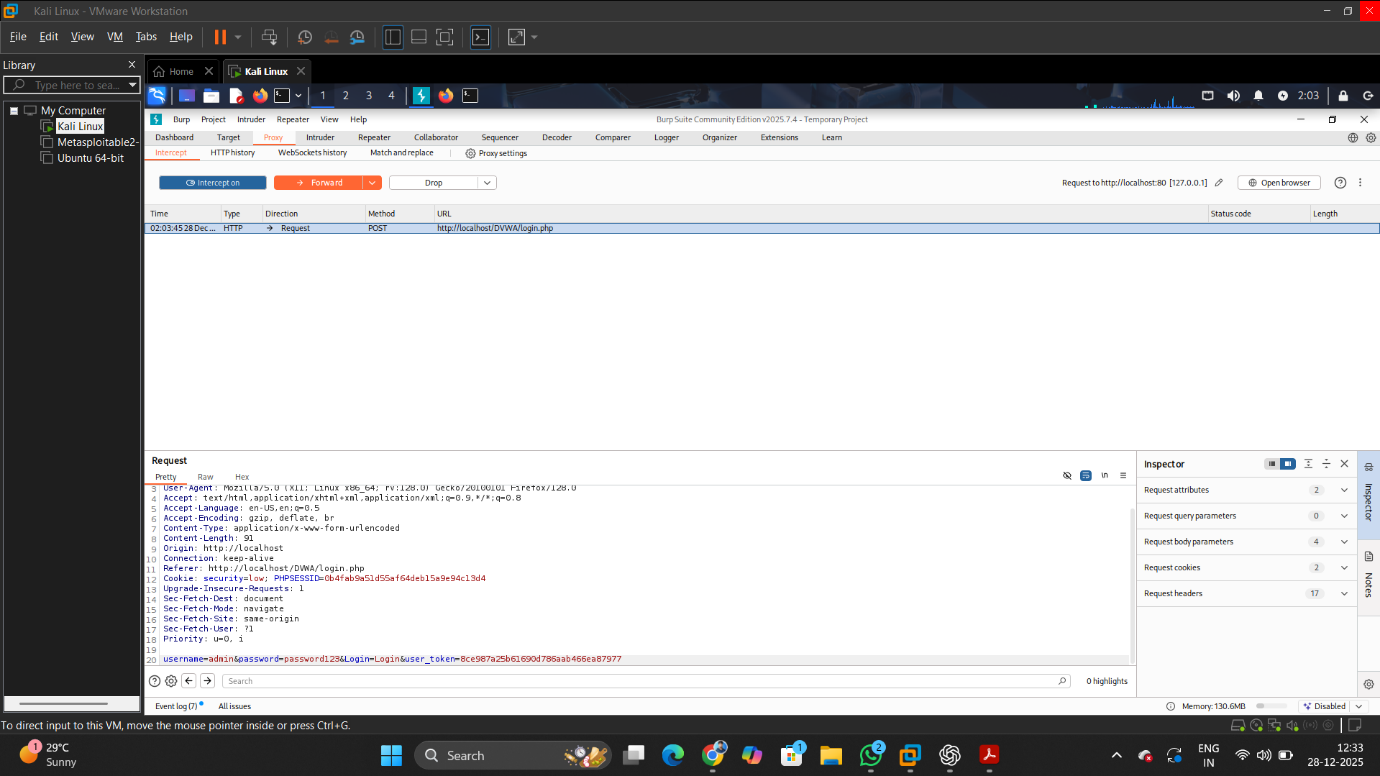
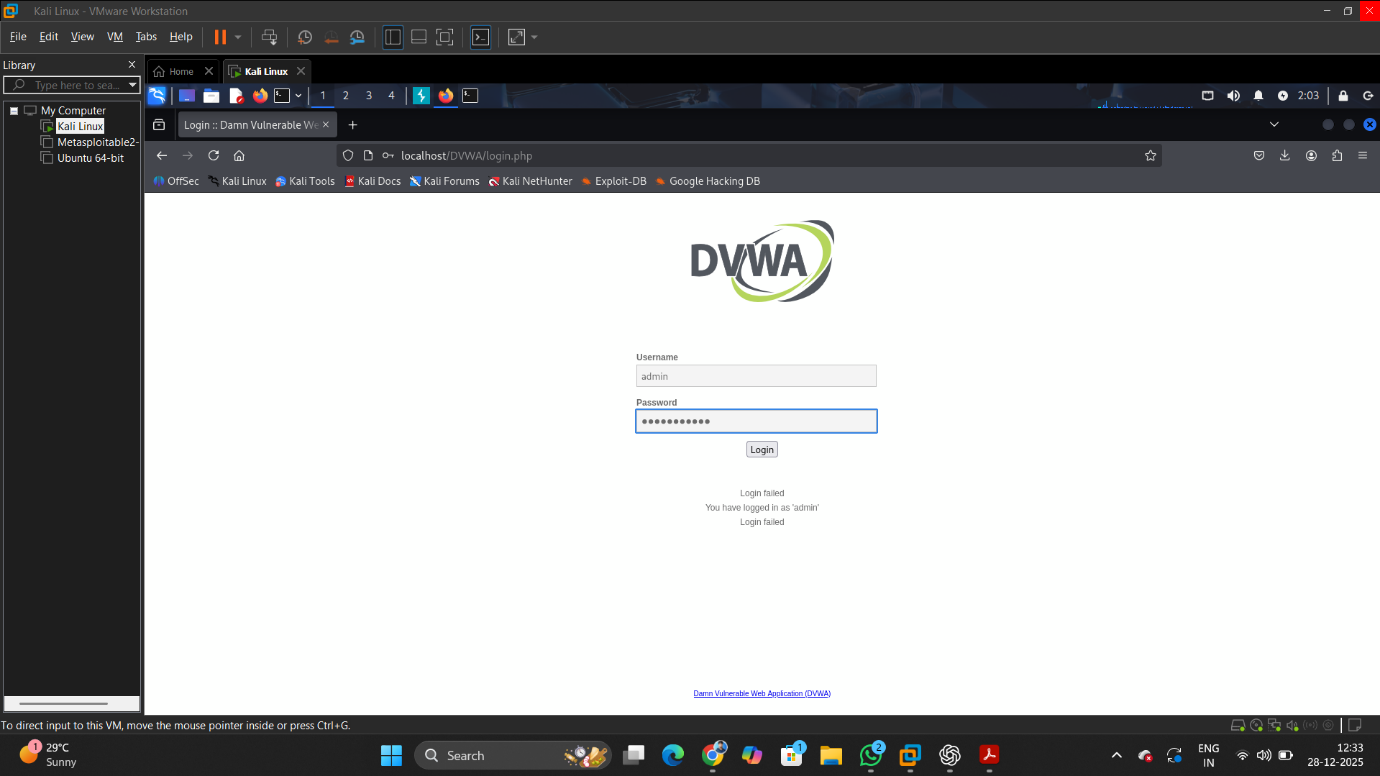
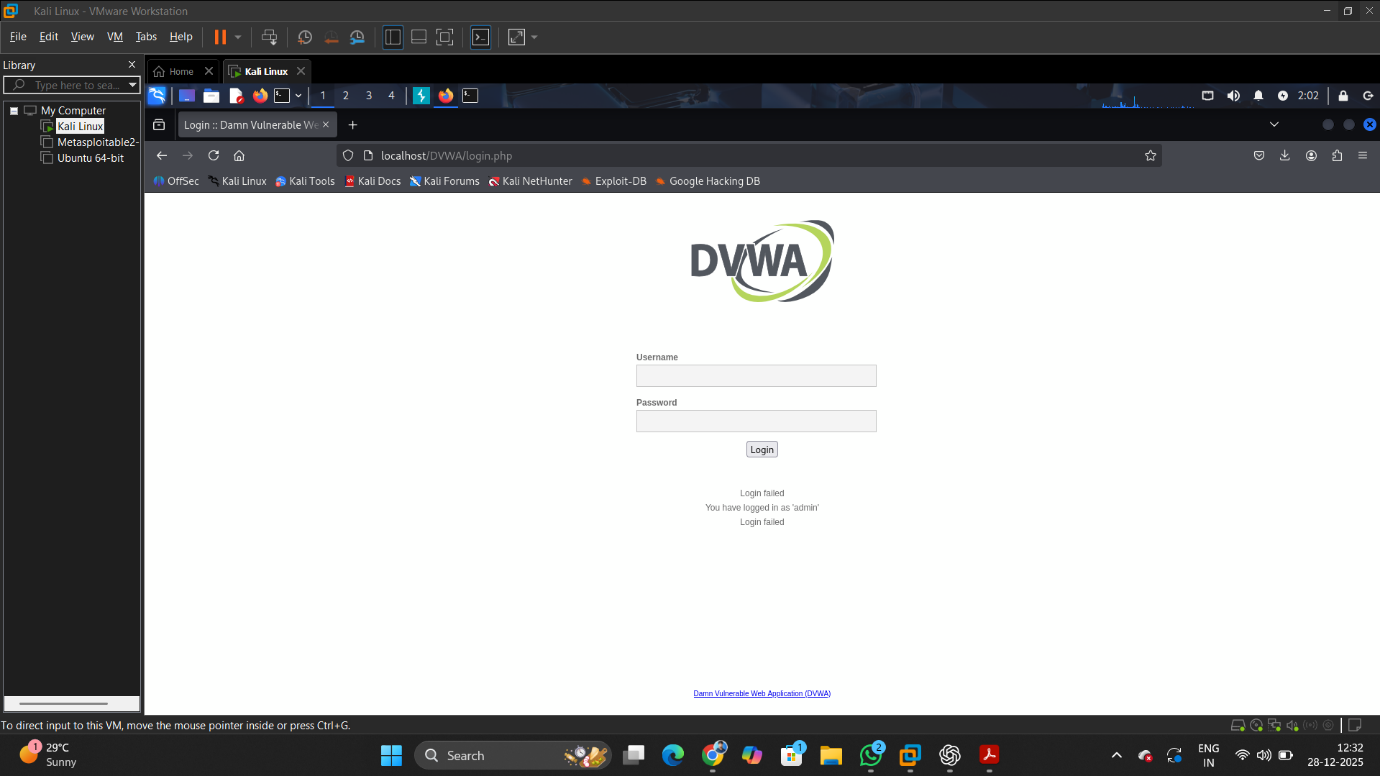
* Login request sent to Intruder.
* Password field selected as attack position.
* Password list used to simulate brute-force attack.

**Findings**

* Login requests can be replayed and modified.
* Weak passwords are vulnerable to brute-force attacks.

**Mitigation**

* Account lockout policy
* CAPTCHA
* Multi-factor authentication



**3.6 Web Security Headers**

**Analysis**

The application was tested using securityheaders.com.

**Missing Headers**

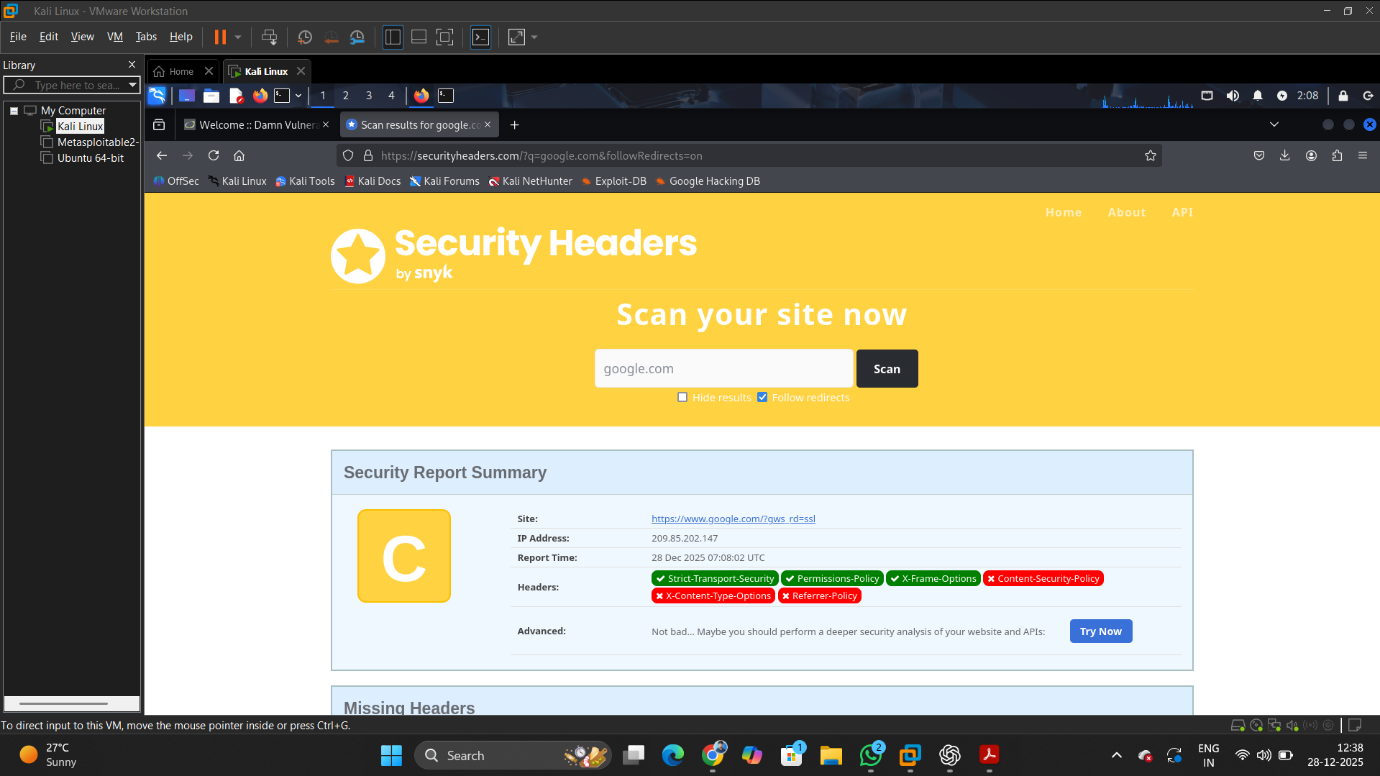
* X-Frame-Options
* X-Content-Type-Options
* Strict-Transport-Security

**Mitigation**

Header set X-Frame-Options "DENY"

Header set X-Content-Type-Options "nosniff"

Header set Strict-Transport-Security "max-age=31536000"



**4. Conclusion**

This assessment successfully demonstrated multiple OWASP Top 10 vulnerabilities using DVWA. Proper secure coding practices, input validation, and security controls significantly reduce attack surface and risk.

**5. Disclaimer**

This testing was performed strictly in a controlled lab environment for educational purposes only. No real systems were harmed or tested.