CYBER SECURITY TASK - 1
Web Application Security Testing Report

Name: Pinemula Arun Kumar
Organisation: Future Interns

Executive Summary

- This VAPT assessment was conducted on the DVWA (Damn Vulnerable Web Application) platform. The objective of this project was to identify critical web vulnerabilities aligned with the OWASP Top 10.
- Five major vulnerabilities were tested: SQL Injection, Command Injection, Reflected XSS, Stored XSS, and File Upload Bypass.

Tools & Methodology

1. Kali Linux
2. DVWA
3. Burp Suite
4. Manual Payload Testing
5. OWASP Testing Guide Methodology

OWASP Top 10 Mapping

SQL Injection A03: Injection Command
Injection A03: Injection
Reflected XSS A07: Identification & Authentication Failures
Stored XSS A07: Identification & Authentication Failures File
Upload A05: Security Misconfiguration

1. SQL Injection Vulnerability
Severity: High
OWASP Category: A03 – Injection

Finding Description:

The application's user input field fails to properly validate or sanitize data before using it in SQL queries. As a result, an attacker can inject malicious SQL payloads to manipulate the underlying database.During testing, the parameter id in the SQL Injection module was found vulnerable.

By submitting a crafted payload: "1' OR '1'='1" the application returned all user records, confirming that the backend query is being executed without parameterized statements or secure input handling. This indicates that the application concatenates user input directly into SQL queries, exposing it to SQL Injection attacks.

Technical Impact:

Exploiting SQL Injection allows an attacker to:

- Bypass authentication mechanisms
- Retrieve sensitive information (usernames, passwords, session data)
- Modify or delete database records
- Perform database enumeration and structure mapping
- Potentially escalate the attack to full server compromise
- Install a web shell using SQL functions (e.g., LOAD_FILE, INTO OUTFILE)
- Extract password hashes and crack them offline

This vulnerability poses a critical risk, potentially leading to complete loss of confidentiality, integrity, and availability of the application and database.

Business Impact:

If exploited in a real-world application, this vulnerability could result in:

- Leakage of confidential customer data
- Legal & compliance violations (GDPR, PCI-DSS, ISO 27001)
- Reputation damage
- Financial loss and customer trust issues
- Full database takeover

Recommended Fix:

1. Implement Parameterized Queries (Prepared Statements)
Use secure methods like:

- PDO with prepared statements (PHP)
- mysqli->prepare()
- ORM-level validation (Laravel, Django, Hibernate, Spring)

2. **Input Validation & Sanitization**

- Allow only expected characters and values
- Reject or escape dangerous characters
- Apply server-side validation

3. **Use Stored Procedures** (Where Applicable)
Avoid dynamic SQL statements.

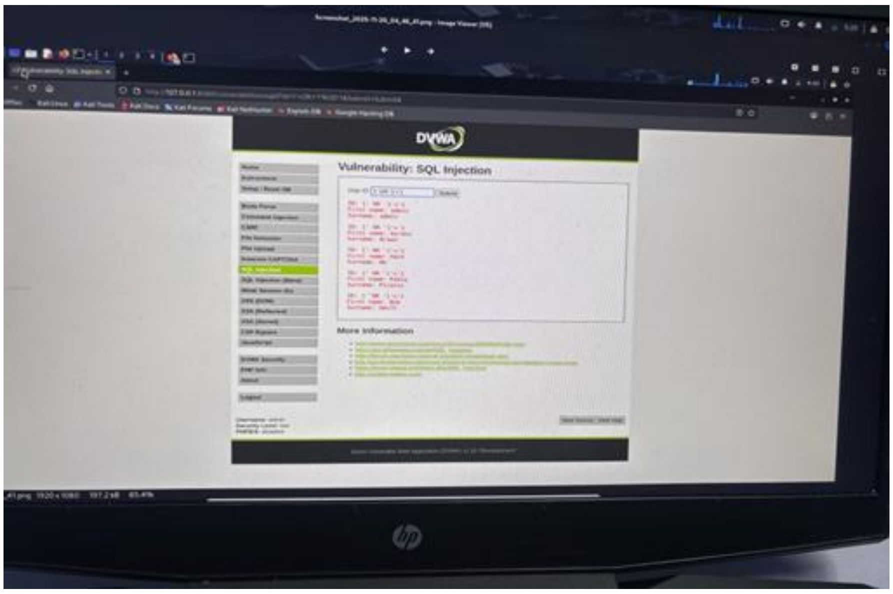
4. **Implement Web Application Firewall** (WAF)
Deploy a WAF to block basic SQLi attempts while the application is being patched.

5. **Enforce Least Privilege Database Permissions**
Ensure the application user has:

- No DROP, ALTER privileges
- Limited SELECT/INSERT/UPDATE access

6. **Conduct Secure Code Review**
Review database interaction code to eliminate unsafe query concatenation.

Evidence:



2. Command Injection

Severity: High

OWASP Category: A03 – Injection

Finding Description

During testing of the *Command Injection* module in DVWA, it was observed that user-supplied input is passed directly to the system shell without proper validation or sanitization. This allows an attacker to inject arbitrary operating system commands along with legitimate input.

Using the payload: "127.0.0.1; ls"the application executed the ls command on the underlying server and returned the directory contents (help, index.php, source) within the HTTP response. This confirms that the web application is vulnerable to OS Command Injection, allowing direct code execution on the host machine.

This vulnerability occurs because the application concatenates untrusted user input into OS commands executed through functions such as system(), exec(), or shell_exec() without applying any filtering or escaping.

Risk Impact

Severity: High

Command Injection is one of the most dangerous vulnerabilities because it gives an attacker the ability to execute arbitrary system-level commands. This can lead to:

Potential Impact:

- **Full server compromise**
  Attackers may execute commands to read sensitive files, modify system data, or establish persistence.
- **Privilege escalation**
  If the web server runs with elevated privileges, attackers can gain root-level control.
- **Data exfiltration**
  Database files, configuration files, and credentials can be stolen.
- **Remote code execution (RCE)**
  Attackers can upload backdoors, reverse shells, or malware.
- **Pivoting inside internal network**
  Once inside, the attacker can scan and attack other machines in the environment.
- **Denial of Service (DoS)**
  By executing destructive system commands (e.g., infinite loops, resource exhaustion), attackers can make the application unavailable.

Overall, command injection provides the attacker with unrestricted control over the application server.

Mitigation Recommendations

1. Strict Input Validation

Only allow valid, expected input types:

- Use allow-list validation (e.g., only numeric IP addresses).
- Reject or sanitize special characters like ;, |, &, ```, $, >.

2. Do NOT pass user input directly into shell commands

Avoid using functions such as:

- system()
- exec()
- shell_exec()
- popen()
- passthru()

If OS-level commands must be used, implement secure wrappers.

3. Use Built-in Language Functions Instead

Example:

Instead of executing ping via shell commands, use a server-side library to handle ICMP requests securely.

4. Escaping and Sanitization

If executing commands is unavoidable, escape input using:

- escapeshellcmd()
- escapeshellarg()

5. Implement Least Privilege

Ensure the web server runs as a restricted user with minimal permissions.

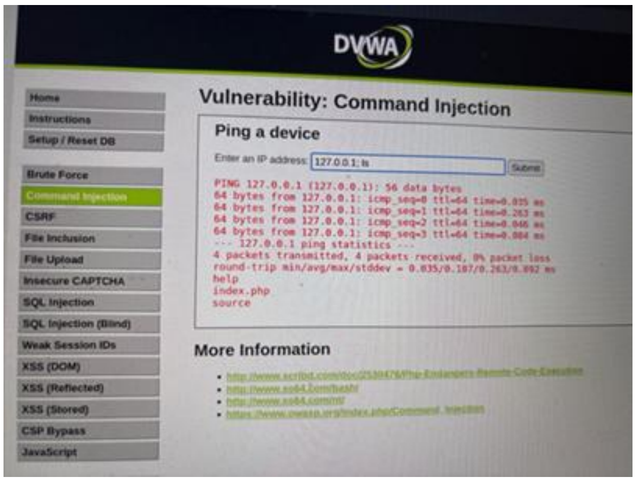
6. Web Application Firewall (WAF)

Deploy WAF rules to detect and block OS command injection payloads.

7. Conduct Regular Code Reviews

Developers should follow secure coding practices and adopt frameworks that prevent command execution vulnerabilities.

Evidence:



Vulnerability: Command Injection

Ping a device

Enter an IP address: 127.0.0.1; ls

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.035 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.263 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.084 ms
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.035/0.107/0.263/0.092 ms
help
index.php
source
```

More Information

- [illegible]
- [illegible]
- [illegible]
- [illegible]

3. Reflected Cross-Site Scripting (XSS)
Severity: High
OWASP Category: A07 – Identification & Authentication Failures

Finding:
A Reflected Cross-Site Scripting (XSS) vulnerability was identified within the application's input handling mechanism. The application fails to adequately sanitize user-supplied data before reflecting it back in the HTTP response. This allows an attacker to inject malicious JavaScript payloads into the application, which are executed in the victim's browser when they interact with a crafted link or compromised parameter.
This vulnerability demonstrates insufficient output encoding and a lack of proper validation of untrusted input, leading to the execution of arbitrary client-side scripts.

Impact:
The impact of Reflected XSS is High, as successful exploitation allows an attacker to execute arbitrary JavaScript in the context of the victim's session. Potential consequences include:

- Session Hijacking: Attacker can steal active session cookies or authentication tokens.
- Credential Theft: Keystroke logging or phishing forms can be injected.
- User Impersonation: Attacker can perform unauthorized actions on behalf of the victim.
- Defacement / Redirection: Application content can be modified or users can be redirected to malicious sites.
- Browser Exploitation: Attacker may exploit browser vulnerabilities using malicious scripts.

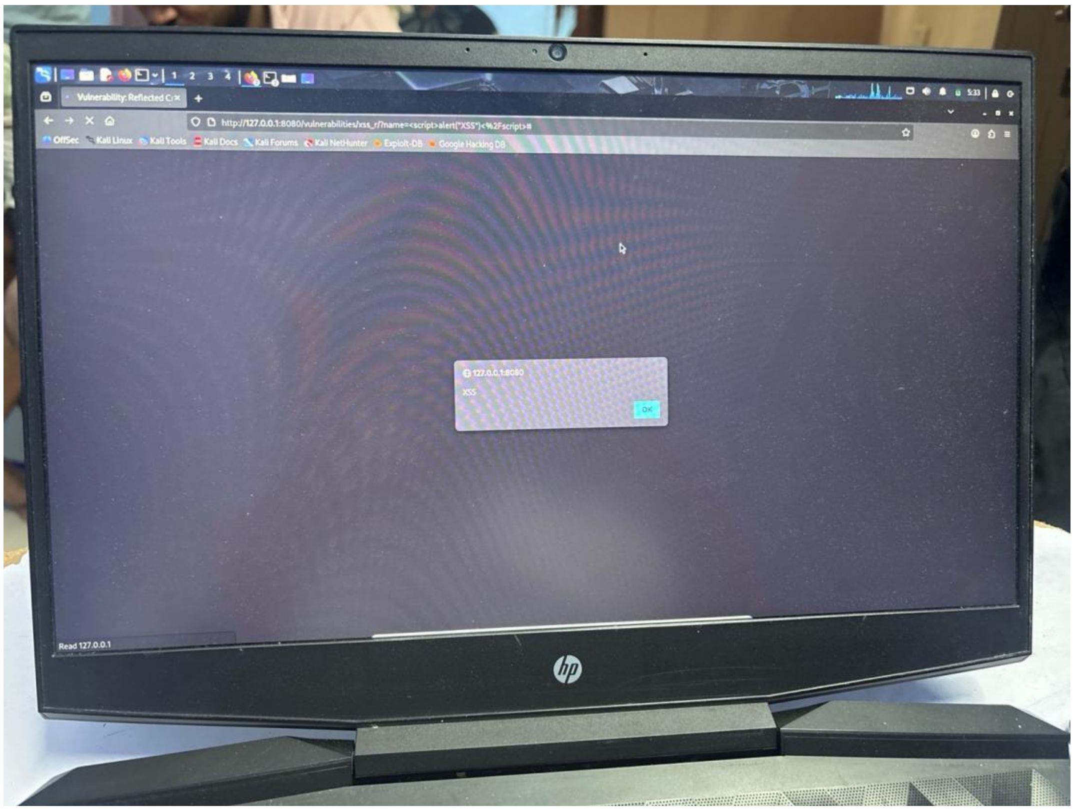
This significantly compromises the confidentiality, integrity, and availability of user data and can lead to full account takeover.

Mitigation:
To effectively mitigate this vulnerability, the following security measures are recommended:

1. **Input Validation:**
   - ➢ Reject or sanitize suspicious characters such as <, >, ", ', (,), and ;.
   - ➢ Implement server-side validation to ensure only expected input is processed.
2. **Output Encoding:**
   - ➢ Encode all user-supplied data before rendering it in the browser (e.g., HTML entity encoding).
   - ➢ Use a secure templating engine that auto-escapes output.
3. **Content Security Policy (CSP):**
   - ➢ Enforce a strict CSP to limit the execution of inline scripts and reduce XSS attack impact.

4. **HTTPOnly Cookies:**
   - Ensure session cookies are flagged as HttpOnly to reduce the risk of theft through JavaScript.
5. **Adopt Secure Frameworks:**
   - Use frameworks with built-in XSS protections such as React, Angular, or Django templates.

Evidence:



```
http://127.0.0.1:8080/vulnerabilities/xss_r/?name=<script>alert("XSS")<%2Fscript>#
```

```
127.0.0.1:8080
XSS
```

4. Stored Cross-Site Scripting (Stored XSS)
Severity: Critical
OWASP Category: A07 – Identification & Authentication Failures

Finding:

During the assessment, it was identified that the application is vulnerable to Stored Cross-Site Scripting (Stored XSS). The application fails to sanitize user-supplied input before storing it in the backend database. When this input is later retrieved and rendered on the page, it executes in the browser of any user visiting the affected page.

A payload such as: "<script>alert('Stored XSS')</script>" was successfully stored and executed. This behavior confirms that the application does not apply proper server-side validation, output encoding, or content sanitization before rendering user-generated content.

This vulnerability indicates a logical flaw in the way the application processes stored input fields (e.g., comment sections, feedback forms, guestbook entries).

Impact:
Severity: Critical

Stored XSS is significantly more dangerous than Reflected XSS because the malicious payload persists in the application until manually removed. It can impact every user, including administrators, who load the compromised page.

An attacker exploiting Stored XSS can:

1. Steal Account Sessions
   - Hijack user sessions using document.cookie
   - Gain unauthorized access to accounts, including admin users
2. Deface or Modify Application Content
   - Inject misleading, harmful, or phishing content
   - Display fake login forms to steal credentials
3. Deliver Automated Malware or Redirect Attacks
   - Redirect users to malicious websites
   - Load malicious scripts from remote servers
4. Create Self-Spreading Worms
   - Payloads can auto-post themselves to multiple pages
   - Spread across user accounts without interaction
5. Full Account Takeover & Privilege Escalation
   - If an admin loads the malicious page, attackers can take over the entire application
   - Modify settings, delete data, or escalate privileges

Stored XSS is considered a persistent, high-impact vulnerability capable of compromising the entire web application ecosystem.

Mitigation

To fully eliminate the Stored XSS risk, the organization should adopt a multi-layered defense strategy:

1. Implement Server-Side Input Validation
   - Reject unexpected characters (< > ' " / ; () {} [])
   - Use allow-lists based on expected input
   - Enforce strict length limits
2. Apply Contextual Output Encoding

Encode output depending on where it is placed:

- HTML encoding for web pages
- JavaScript encoding inside scripts
- URL encoding inside attributes

  Use libraries such as:
- OWASP Java Encoder
- PHP htmlspecialchars() or filter_var()

3. Deploy a Strong Content Security Policy (CSP)

A properly configured CSP:

- Blocks execution of inline JavaScript
- Restricts loading of external scripts
- Reduces the impact of XSS even if it occurs

Example CSP:

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

4. Use Web Frameworks with Built-In XSS Protection

Frameworks like React, Angular, Django, Laravel automatically escape content unless explicitly bypassed.

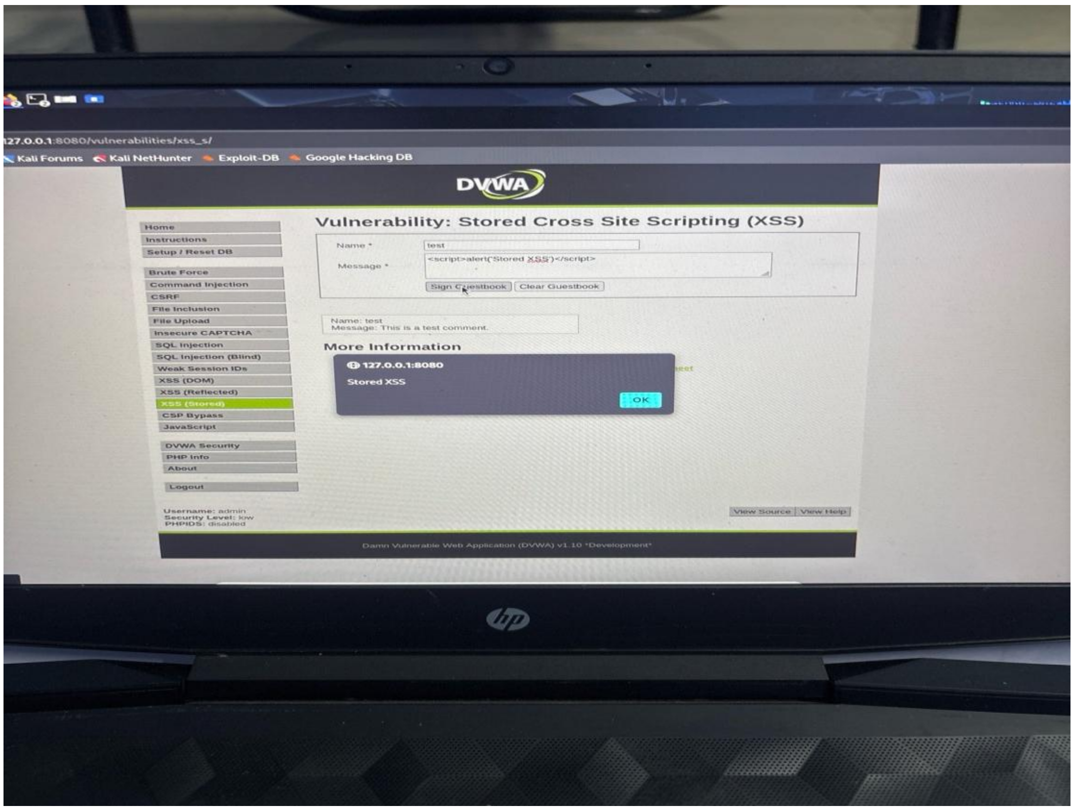
5. Sanitize User Inputs Before Storing
   - Use libraries like HTML Purifier or DOMPurify to remove harmful HTML
   - Strip or neutralize all potentially executable DOM elements
6. Conduct Developer Secure Coding Training

Ensure developers understand:

- Input sanitization
- Output encoding
- Secure JavaScript handling
- OWASP Top 10

7. Perform Regular Code Reviews & Automated Scans
   - Integrate SAST/DAST tools
   - Enforce secure code pipelines

Evidence:



Vulnerability: Stored Cross Site Scripting (XSS)

Name * test

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

Name: test
Message: This is a test comment.

More Information

127.0.0.1:8080

Stored XSS

Username: admin
Security Level: low
PHPIDS: disabled

Damn Vulnerable Web Application (DVWA) v1.10 *Development*

5.File Upload Vulnerability
Severity: Critical
OWASP Category: A05 - Security Misconfiguration

Finding:

The application allows users to upload files without enforcing proper validation or restrictions on file type, file extension, content-type, or server-side execution behavior. During testing, it was possible to upload a malicious PHP payload disguised as an image file (e.g., shell.php.jpg). The server stored the file inside a web-accessible directory: /hackable/uploads/ When the uploaded file was accessed via a browser, the embedded PHP code executed successfully on the server. This confirms insufficient validation of uploaded content and improper file handling on the backend.

This vulnerability can lead to remote code execution (RCE), enabling attackers to run arbitrary commands on the server.

Impact:

Critical — File Upload vulnerabilities are among the most dangerous security flaws, as they allow attackers to directly interact with the application server.
A successful exploit may lead to:

- **Remote Code Execution (RCE):** Attackers can upload a PHP web shell (<?php system($_GET['cmd']); ?>), enabling execution of OS-level commands.
- **Full Server Compromise:** Attackers can read, write, or delete files, modify application logic, or access configuration files.
- **Privilege Escalation:** Uploaded web shells may allow attackers to pivot inside the internal network.
- **Data Breach:** Access to sensitive files (credentials, logs, environment variables).
- **Website Defacement or Hijacking:** Attackers may upload malicious HTML/JS files.
- **Persistent Backdoor:** Attackers can maintain long-term access by uploading stealthy payloads.

Since uploaded files are stored in a publicly accessible directory, this vulnerability exposes the entire web server to compromise.

Mitigation:

To prevent malicious file uploads, the following security controls should be implemented:

1. Strict File Validation

- Validate file type using **MIME type checking** on the server.
- Restrict allowed file types to safe formats only (e.g., .jpg, .png, .pdf).
- Reject files with multiple extensions (e.g., file.php.jpg).

2. Store Files Outside the Web Root

Uploaded files should not be directly accessible from the web server.
Store them outside /var/www/html/ and serve them via controlled handlers.

3. Implement Content Scanning

- Use tools like ClamAV to scan all uploads for malware or scripts.
- Verify file content matches declared type (e.g., images contain valid image headers).

4. Rename Uploaded Files

The server should rename files using a safe random name instead of using user-provided filenames.

5. Enforce Output Encoding

Prevent uploaded content from executing scripts by ensuring all outputs are properly encoded.

6. Apply the Principle of Least Privilege

- File upload directories should not have execute permissions.
- The web server user should have minimal write permissions.

7. Disable Script Execution

Configure the upload directory (e.g., Apache .htaccess) to block script execution:

Options -Indexes
php_flag engine off

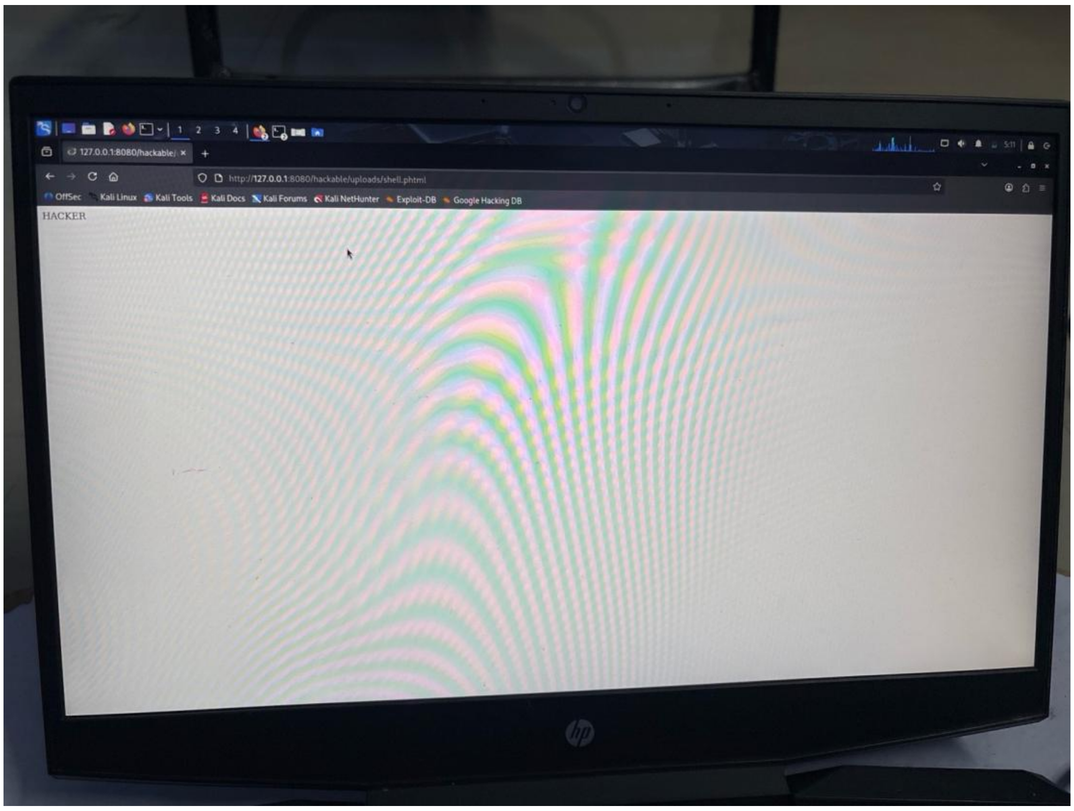
8. Implement Web Application Firewall (WAF)

Use rules to detect and block dangerous uploads based on signatures.

9. Secure Developer Practices

- Conduct code reviews for upload handling logic.
- Use secure frameworks that manage file uploads safely.

Evidence:



http://127.0.0.1:8080/hackable/uploads/shell.phtml

HACKER