**Security Testing Report — DVWA (Laboratory)**

**Task Number:** 3 – Web Application Security

**Name:** Jai Rane

**Intern ID:** ASPL2518045

**Date:** 20-10-2025

**Target:** DVWA (localhost)  
**Environment:** Kali Linux (attacker) → DVWA (target)  
**Tools:** Browser, sqlmap, Burp Suite (Community), OWASP ZAP, tcpdump, Apache, MariaDB/MySQL  
**Date:** *(fill date)*

**Executive summary**

This assessment demonstrates classic web-application vulnerabilities present in DVWA (Damn Vulnerable Web App) and practical mitigations. Tests performed: **SQL Injection, Cross-Site Scripting (stored & reflected), Cross-Site Request Forgery, Local & Remote File Inclusion**, plus **Burp Suite** interception and **security header** checks. For each issue we show exploitation steps, proof (screenshots/logs), and recommended fixes (code snippets and server configuration). All activity was performed in a controlled lab environment.

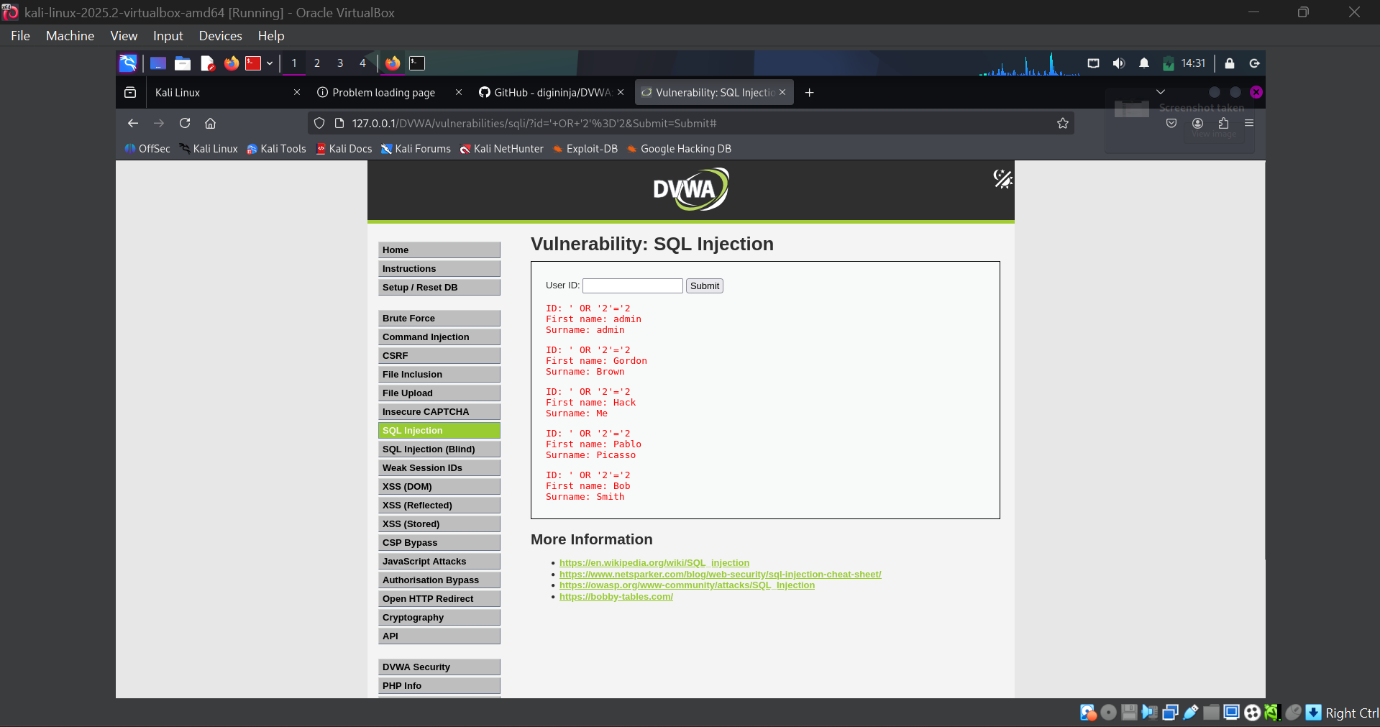
**1. SQL Injection (SQLi)**

**1.1 Summary**

SQL Injection allows an attacker to alter SQL queries by injecting input. In DVWA the id parameter on the SQLi page is vulnerable: payloads such as ' OR '1'='1 return all user rows.

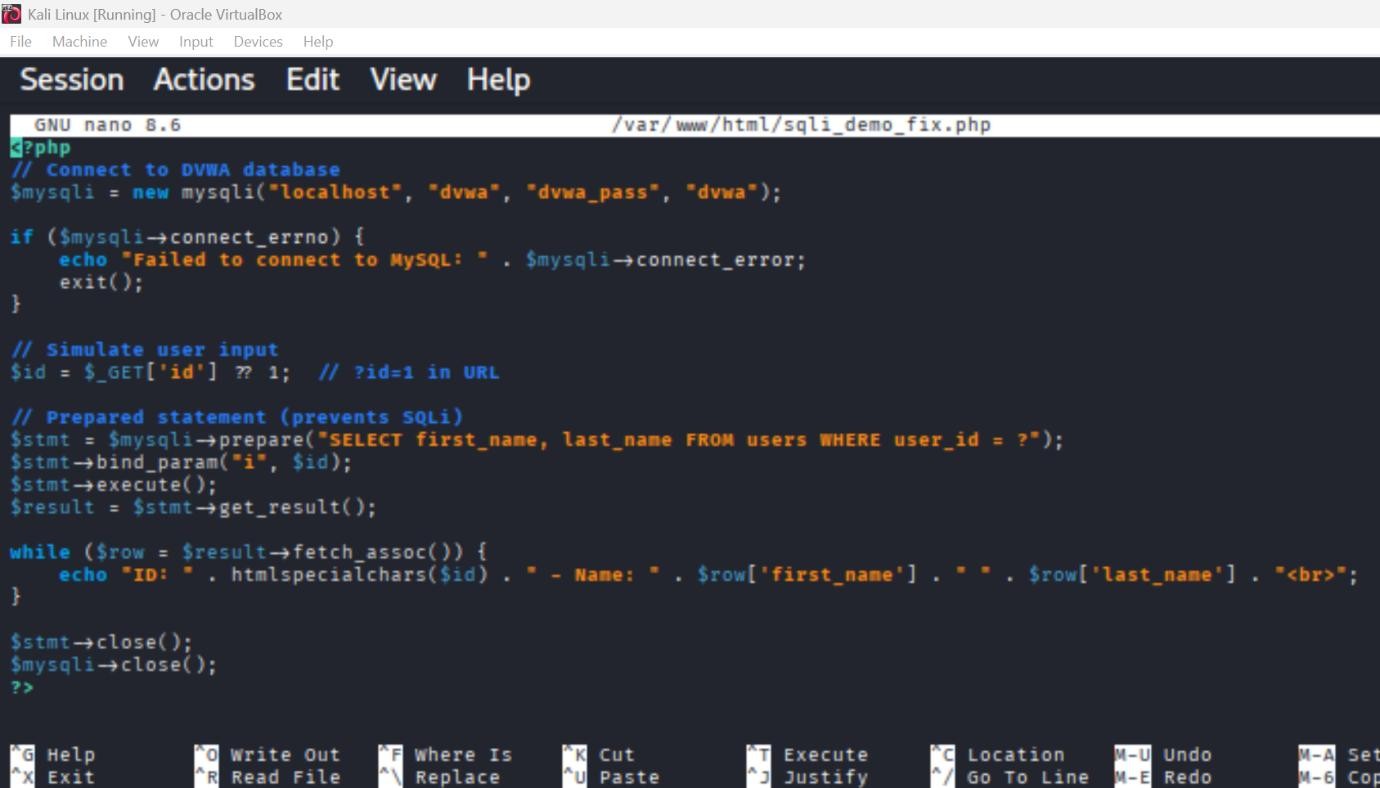
**1.2 Reproduction**

* Navigate: DVWA → SQL Injection.
* Manual payload: 1 OR 1=1 → DB returns many users.
* Automated extraction:



**1.3 Mitigation (Prepared Statements)**

Use parameterized queries. Example PHP (mysqli):



a. Vulnerable

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

$sql = "SELECT \* FROM users WHERE id = '$id'";

$result = mysqli\_query($conn, $sql);

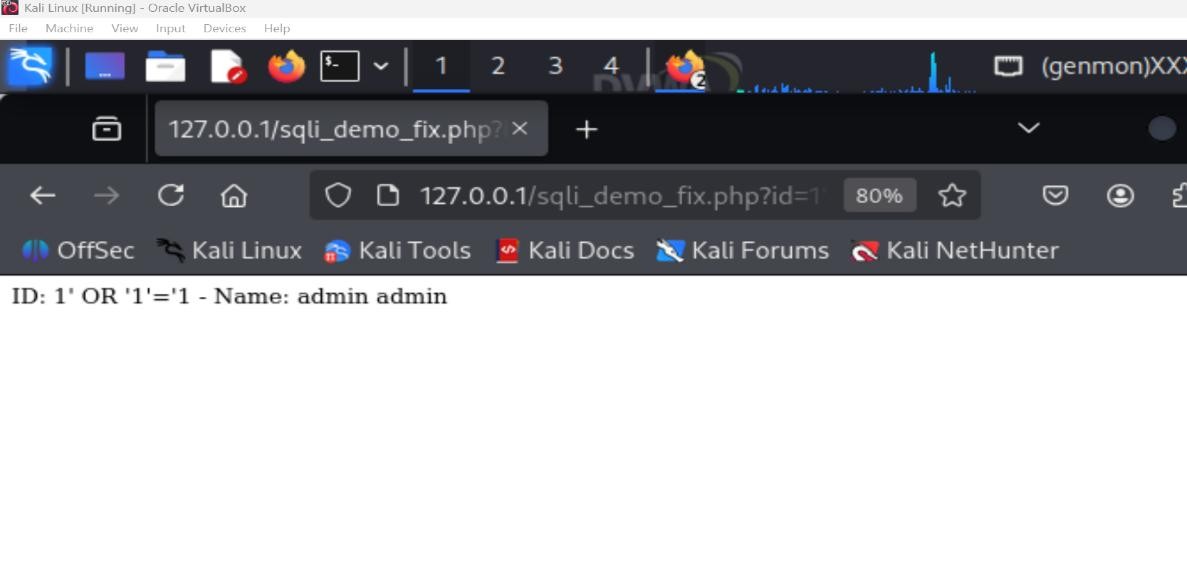
b. Fixed

$stmt = $conn->prepare("SELECT username, password FROM users WHERE id = ?");

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

$stmt->execute();

$result = $stmt->get\_result();

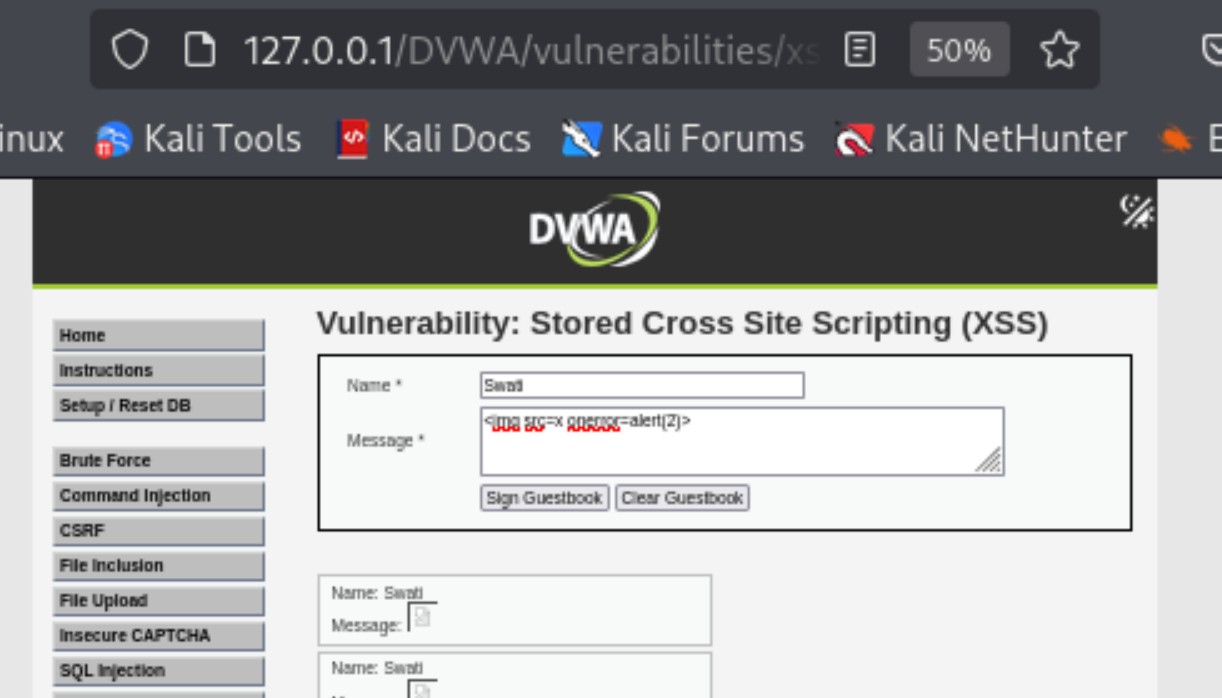
**Effect:** User input is treated as data — injection attempts like ' OR '1'='1 fail.

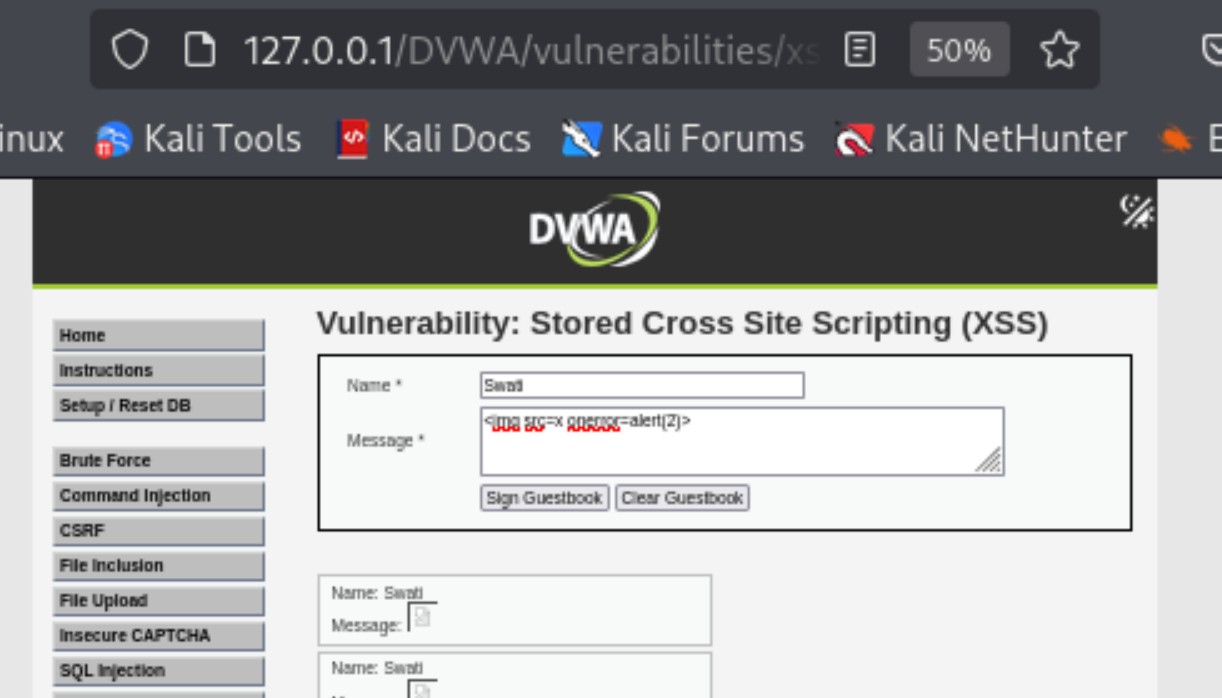
**2. Cross-Site Scripting (XSS)**

**2.1 Summary**

XSS lets attackers execute scripts in victims’ browsers. DVWA exhibits both **Stored** (persisted) and **Reflected** XSS.

**2.2 Reproduction**

* **Stored XSS:** post <img src=x onerror=alert(2)> in guestbook → popup on page load.
* **Reflected XSS:** craft URL with ?name=<script>alert('x')</script> → immediate execution.

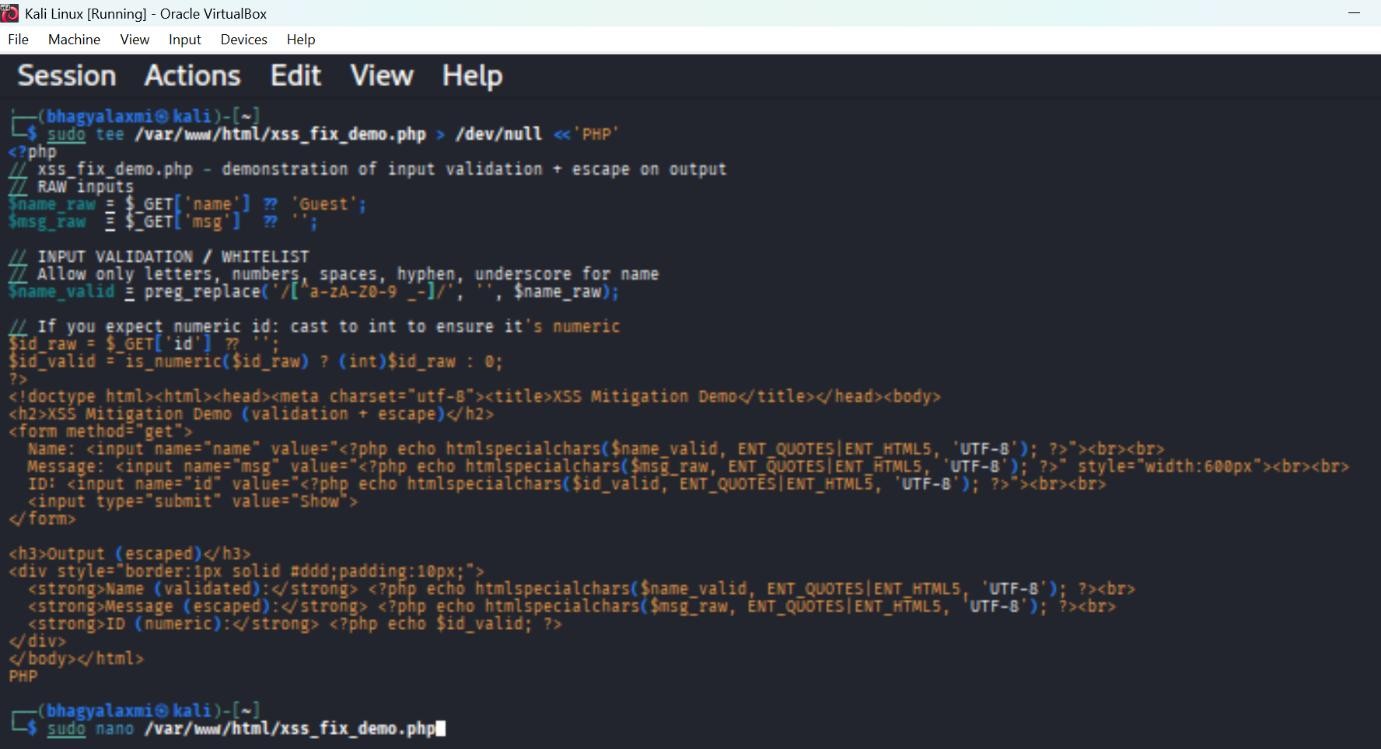


**C. Mitigation — Input validation + Output encoding + CSP**

Show examples:

**Output encoding (PHP)**

// Instead of echoing raw user input:

echo htmlspecialchars($user\_input, ENT\_QUOTES, 'UTF-8');

**Output:**

1. **Name will be stripped of <script> and likely become alert1 or empty (depending on characters).**



1. **Message will be escaped on output (so no alert).**

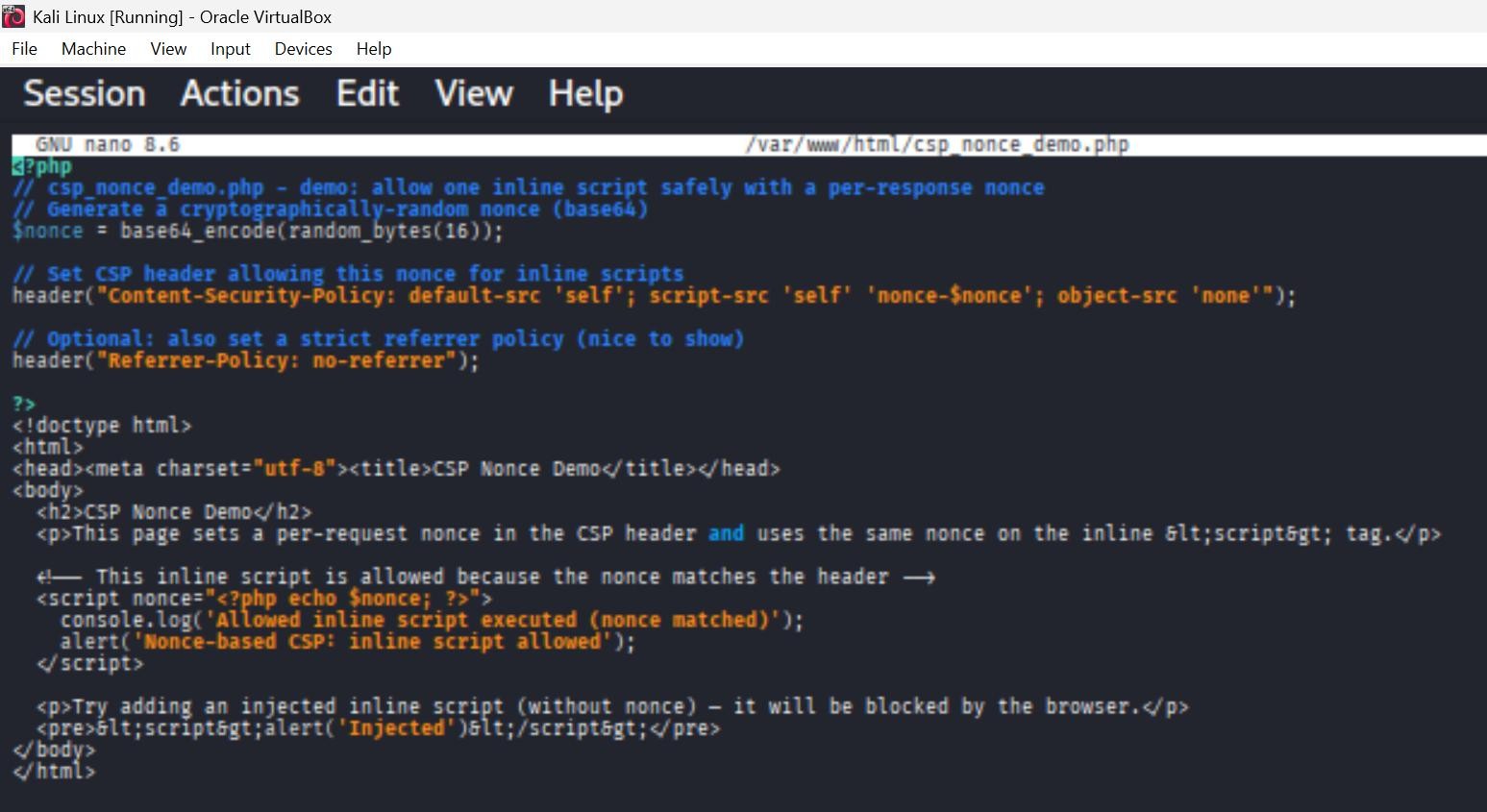


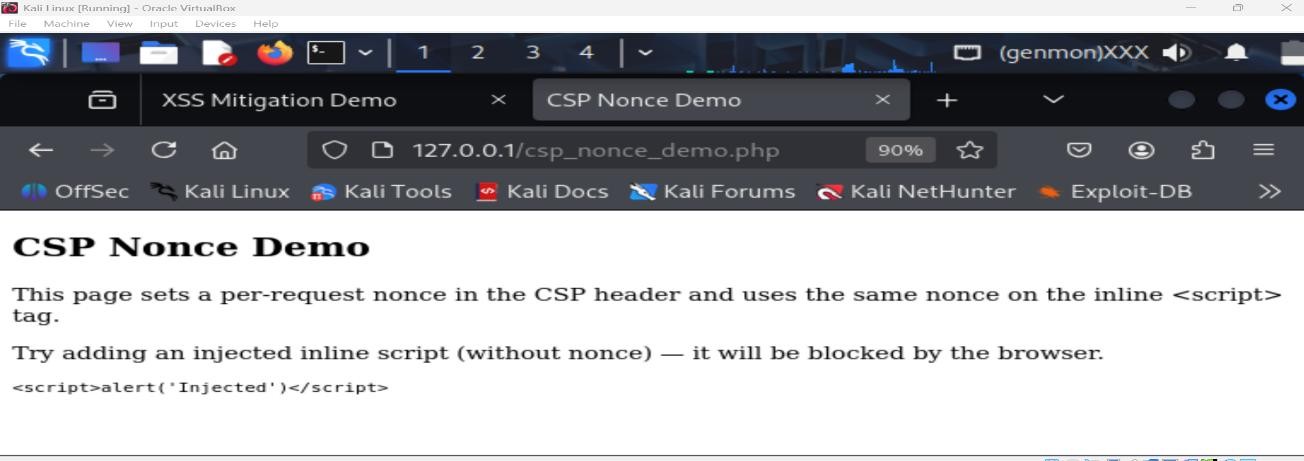
1. **Id will be parsed as 123 or 0 if not numeric.**



**CSP header (Apache)**  
Add to your site's Apache config or .htaccess (show how you add header):

* Header set Content-Security-Policy "default-src 'self'; script-src 'self'; object-src 'none';"
* Header set X-Content-Type-Options "nosniff"
* Header set X-Frame-Options "DENY"
* Header set Referrer-Policy "no-referrer"
* After adding, restart Apache:
* sudo a2enmod headers
* sudo systemctl restart apache2





**3) CSRF — exploit + token protection**

**A. Create exploit (demo)**

* Use DVWA **Change Password** or CSRF example page.
* Build a simple attacker HTML file (on your Kali desktop) that auto-submits a form to DVWA change-password endpoint:

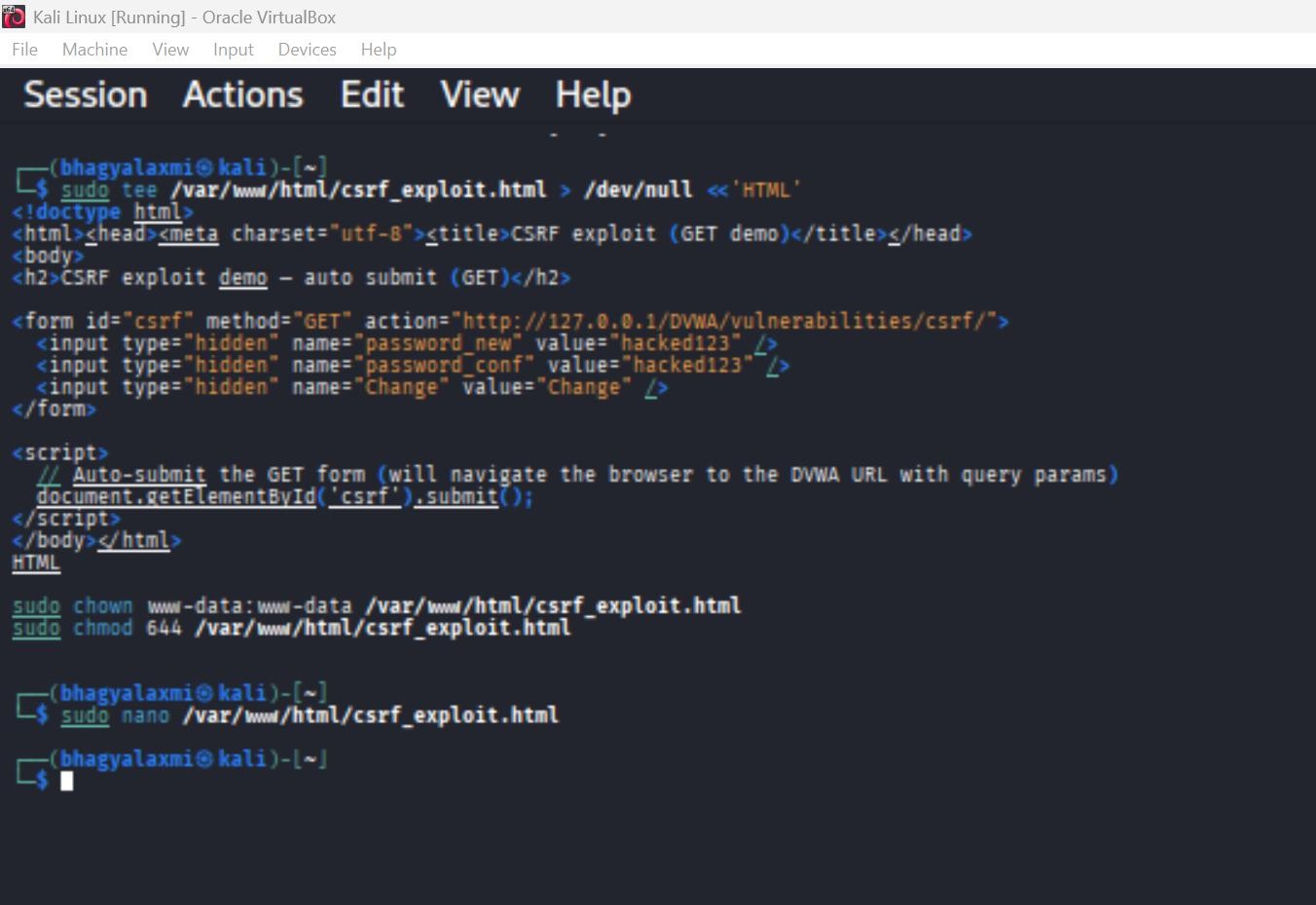
<html><body onload="document.forms[0].submit()">

<form action="http://127.0.0.1/dvwa/vulnerabilities/csrf/" method="POST">

<input type="hidden" name="password\_new" value="attacker123">

<input type="hidden" name="password\_conf" value="attacker123">

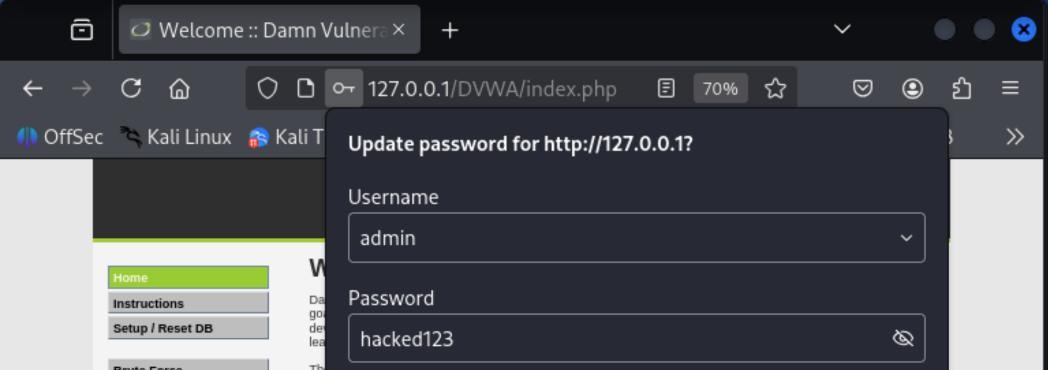
</form>

</body></html>

If you are still logged in, this request will change your password without you knowing.

1. Why is it Dangerous? No user interaction needed except clicking/opening attacker’s page.

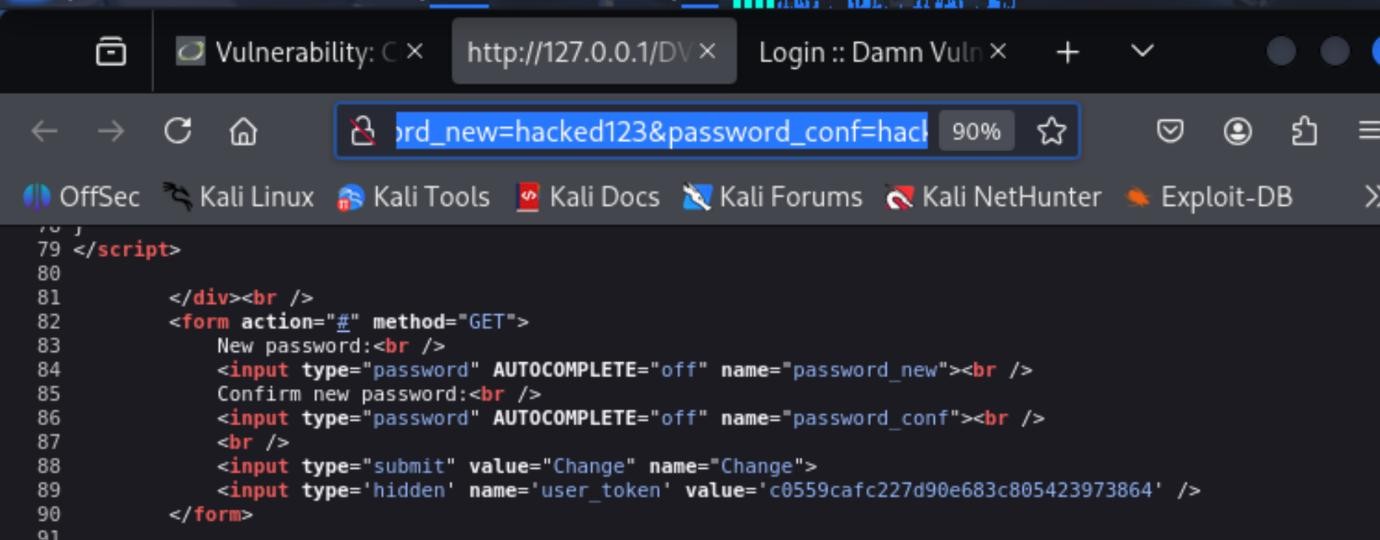
2. Works silently because cookies are sent automatically by the browser. - Can lead to account takeover.



Open that page in the same browser session (authenticated as victim) to demonstrate the change.

**B. Mitigation — Token-based CSRF protection (example)**

Show the minimal server-side approach (pseudo-PHP):



How to Prevent (Mitigation)

* The most common protection is CSRF Token (anti-CSRF token):
* Website generates a random unique token for each form request.
* The token is stored in the user’s session.
* When form is submitted, the token is validated.
* If token is missing/wrong → request is blocked.
* Switched DVWA Security From Low to High, which added a per-session hidden token to the CSRF form (e.g. <input type="hidden" name="user\_token" value="...">).

**# Conclusion & prioritized recommendations**

1. **Eliminate SQLi** — use prepared statements / ORM; apply least privilege DB accounts.
2. **Prevent XSS** — server-side output encoding (htmlspecialchars), apply CSP, and validate inputs.
3. **Protect against CSRF** — per-session, per-form CSRF tokens and proper SameSite cookie attributes.
4. **Mitigate file inclusion** — disable remote includes (allow\_url\_include = Off), apply whitelist patterns for file includes.
5. **Harden headers** — add X-Frame-Options, X-Content-Type-Options, CSP, Referrer-Policy.
6. **Logging & monitoring** — enable audit logging for suspicious requests and failed token validations.

These fixes are low to medium effort and dramatically reduce attack surface in real applications.