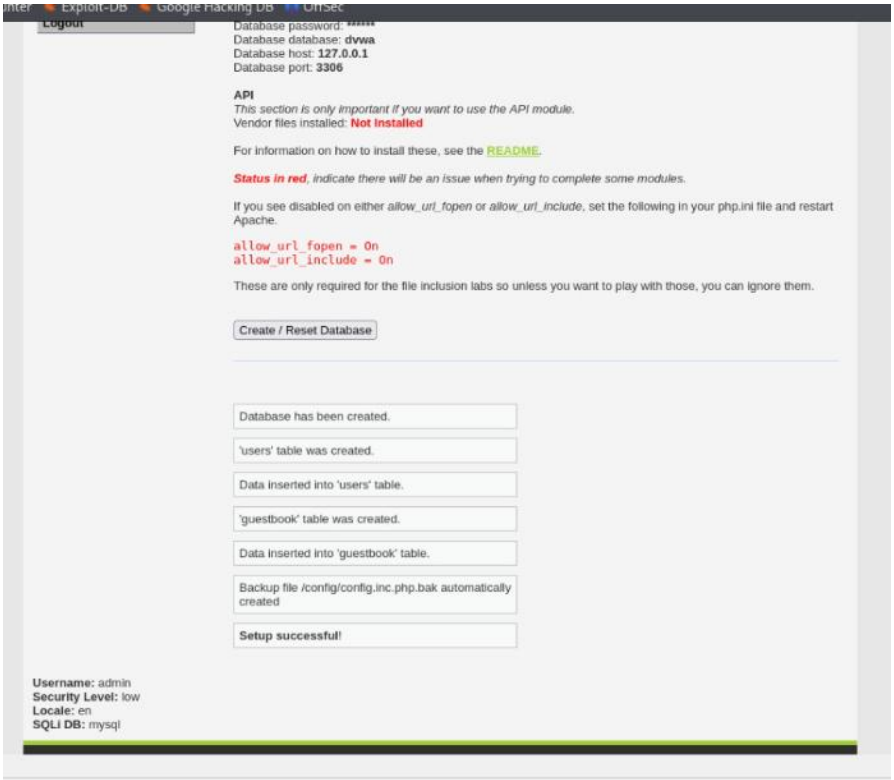
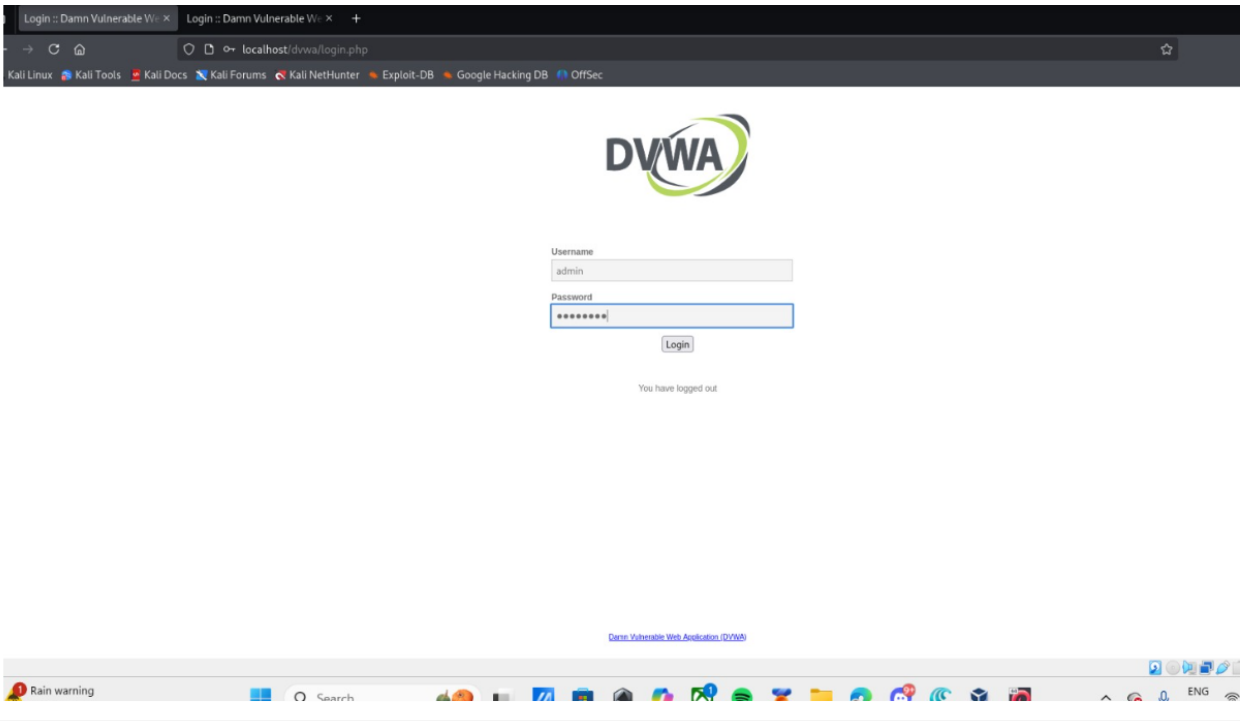
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

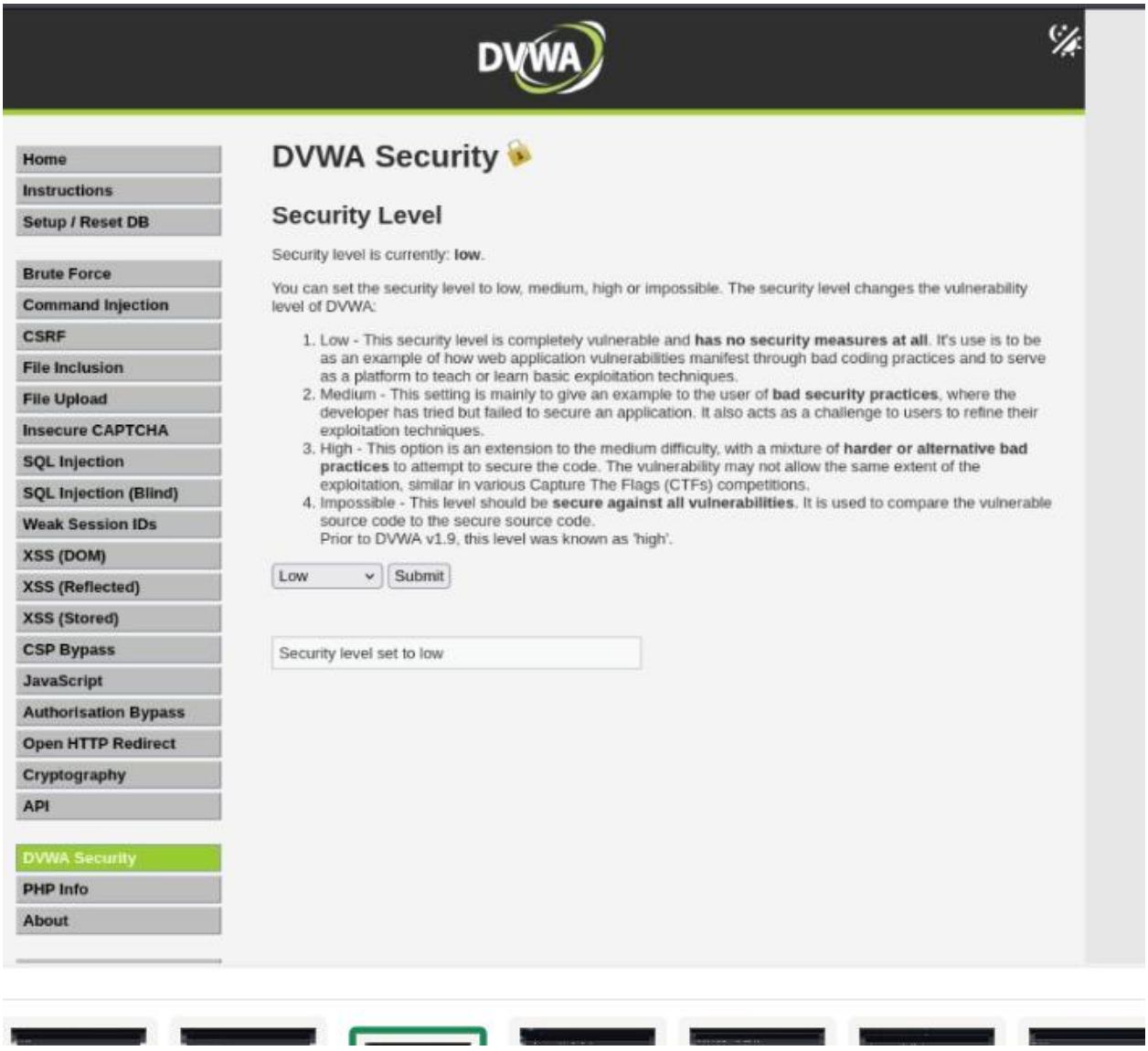
Name	Reyansh Nitin Mehta
UID no.	2023300142
Experiment No.	7
GitHub Link:	https://github.com/reyansh2005/cns_exp7

AIM:	To identify, exploit and mitigate the common web application vulnerabilities.
EXECUTIVE SUMMARY:	This report documents a systematic analysis of common web application vulnerabilities using the Damn Vulnerable Web Application (DVWA) in a controlled lab environment. The goals were to: set up a DVWA test host, identify and exploit vulnerabilities across common categories (SQL Injection, Reflected & Stored XSS, CSRF, IDOR, file upload issues, command/file inclusion), and apply fixes or mitigation strategies. The exercises highlighted how minor lapses in input handling, session management, access control, and file handling escalate into high-impact compromises. Proposed remediations include parameterized queries, output encoding, CSRF tokens, strict upload validation, and session hardening. The final section contains demonstration fixes (code snippets), CVSS-like risk ratings, and a recommended hardening checklist for LAMP applications.
Part A-Setup and Baseline	
PROBLEM STATEMENT :	1. Show DVWA running: screenshot of login page and the "Create/Reset Database" page. 2. Change and note the security level settings (low/medium/high) and explain what the setting changes in code or behavior (short answer).
ANS:	**Objective** To set up DVWA locally and observe how its behavior changes across security levels (Low / Medium / High). **Environment Used** • OS: Windows / Kali / Ubuntu (mention yours) • Browser: Chrome / Firefox • Server: DVWA running on Apache with MySQL • IP Address: X.X.X.X (Masked in screenshots) **Setup Steps (Short)** 1. Install XAMPP / Docker-based DVWA 2. Access DVWA using browser (http://localhost/dvwa) 3. Configure database connection

4. Click **"Create / Reset Database"**
5. Login using credentials:
    - Username: admin
    - Password: password



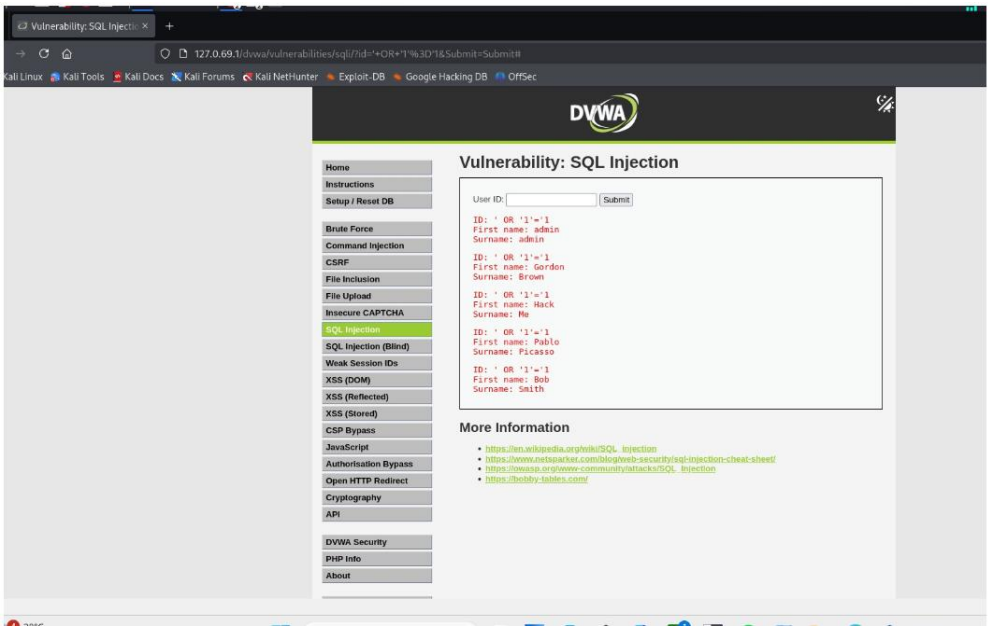

Login :: Damn Vulnerable We
Login :: Damn Vulnerable We
localhost/dvwa/login.php
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
DVWA
Username
admin
Password
Login
You have logged out
Damn Vulnerable Web Application (DVWA)
Rain warning
ENG



Part B	
PROBLEM STATEMENT :	For each: (a) identify vulnerable page, (b) exploit (screenshot + short reproduction steps), (c) explain root cause, (d) propose a fix.
ANS:	**B1 — SQL Injection (SQLi)** **(a) Vulnerable page** /vulnerabilities/sqli/ (DVWA → SQL Injection) **(b) Exploit — short reproduction steps (controlled lab):** 1. Login to DVWA and open the SQL Injection module. 2. In the input field labelled id (or URL param ?id=), enter the classic test payload:

3. 1' OR '1'='1
4. Submit the form. If vulnerable, the page will display multiple rows (authentication bypass or table dump).
5. (Optional — blind/time-based) use a time delay payload (lab safe) to detect blind SQLi, e.g.:
6. 1' AND SLEEP(5)--

— observe increased response time.



(c) Root cause

- User input is concatenated directly into SQL queries without parameterization or strict validation.
- The application uses dynamic SQL strings (e.g., "... WHERE id = $id") so attacker-controlled text changes query logic.
- Database user may have excessive privileges (increasing impact).

(d) Proposed fix

- Use prepared/parameterized statements (PDO or mysqli prepared statements).
- Validate input types (cast numeric IDs to integers).
- Apply least-privilege on DB account.

Example fix (PHP PDO):

// Unsafe:

```
// $sql = "SELECT username, password FROM users WHERE id = {$_GET['id']}";

// Safe:
$id = isset($_GET['id']) ? (int) $_GET['id'] : 0;
$stmt = $pdo->prepare('SELECT username, password FROM users WHERE id = :id LIMIT 1');
$stmt->execute([':id' => $id]);
$user = $stmt->fetch();
```

Notes / mitigation checklist

- Add web application firewall (WAF) rules for known SQLi patterns for defense-in-depth.
- Remove verbose DB errors from public output (turn off debug in production).

CVSS-like risk: High

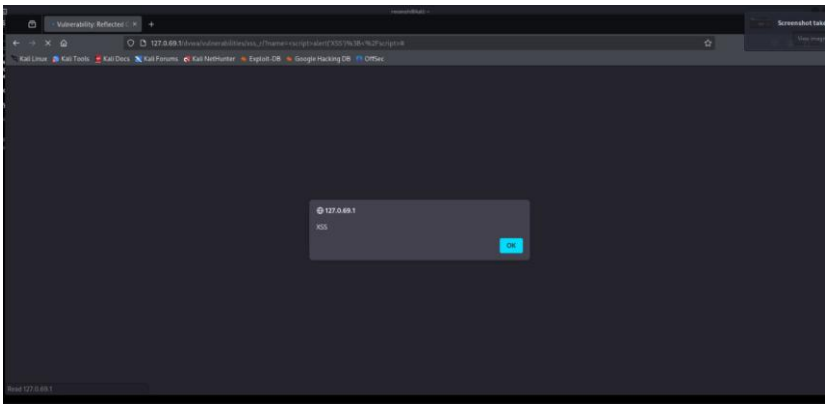
B2 — Reflected Cross-Site Scripting (Reflected XSS)

(a) Vulnerable page

/vulnerabilities/xss_r/ (DVWA → XSS (Reflected))

(b) Exploit — short reproduction steps (controlled lab):

1. Open the Reflected XSS module in DVWA.
2. In the search or URL parameter that is reflected immediately into the page, enter a safe test payload:
3. <script>alert(document.cookie)</script>
4. Submit or navigate to the crafted URL. If vulnerable, an alert box with cookie or a string appears.



(c) Root cause

- The application echoes unsanitized user-supplied content into HTML output without escaping or context-aware encoding.
- No Content Security Policy (CSP) or other client-side mitigations are in place to limit script execution.

(d) Proposed fix

- Apply context-aware output encoding on all user-controlled content (HTML-encode when injecting into HTML body, attribute-encode when used in attributes, JS-encode for

inline JS).

- Implement a strict CSP (e.g., Content-Security-Policy: default-src 'self'; script-src 'self';).
- Validate input where possible (but encoding is primary defense for XSS).

Example fix (PHP):

// On output:

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

Stronger measure if HTML is allowed: Use an HTML sanitizer (e.g., HTMLPurifier) to strip dangerous tags/attributes.

CVSS-like risk: Medium–High (depends on the context where the script runs — pages with privileged functionality are higher risk)

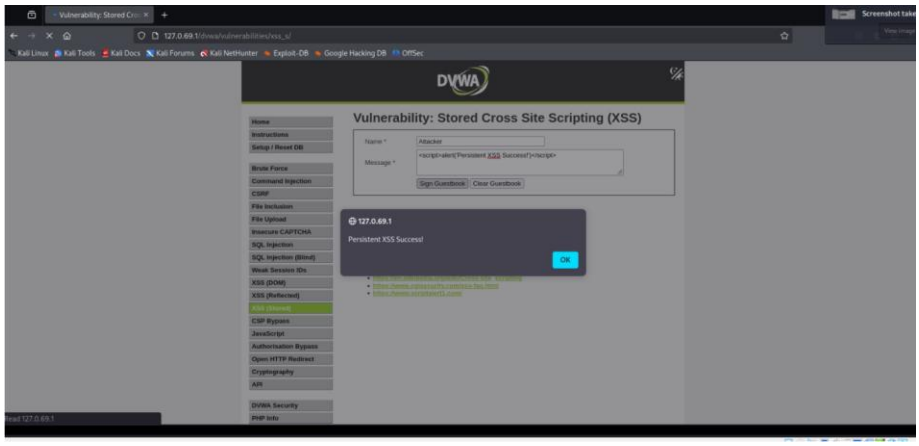
B3 — Stored (Persistent) Cross-Site Scripting (Stored XSS)

(a) Vulnerable page

/vulnerabilities/xss_s/ (DVWA → XSS (Stored)) — typical places: comment forms, profile fields

(b) Exploit — short reproduction steps (controlled lab):

1. Open the Stored XSS module or any form that stores user input (comment, profile text).
2. Submit a benign test payload that will execute in other users' browsers, e.g.:
3. <script>alert('Persistent XSS Success!')</script>
4. Visit the page or reload the view where stored entries appear. The payload executes for any viewer of that page.



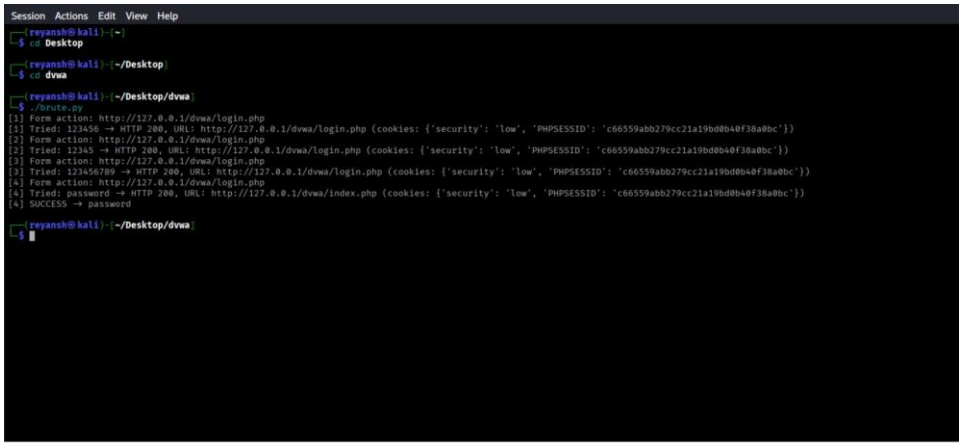
(c) Root cause

- Application persists unsanitized user input into storage (database), then renders that content later without encoding.
- Lack of output encoding and/or missing sanitization on stored content.

(d) Proposed fix

	• Sanitize inputs prior to storage only if storing limited HTML is absolutely necessary; prefer storing raw data and encoding on output.
	• Encode on output with htmlspecialchars() for HTML contexts.
	• If rich HTML must be allowed, use a robust whitelist-based sanitizer (HTMLPurifier or similar) to strip scripts, event handlers (onerror, onclick), and dangerous attributes.
	• Use CSP headers as an additional layer.
	Example fix (output-encoding approach):
	// When rendering stored comments:
	echo nl2br(htmlspecialchars($comment_text, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8'));
	Notes
	• Stored XSS is more dangerous than reflected XSS because it affects multiple users and persists until cleaned.
	• Add monitoring to flag suspicious HTML in stored content.
	CVSS-like risk: High

Part C	
PROBLEM STATEMENT:	Auth / session / logic problems (intermediate)
PROGRAM:	**(a) Vulnerable page** /login.php (DVWA login page — DVWA → Brute force scenario) **(b) Exploit — short reproduction steps (lab, controlled & rate-limited):** 1. Open DVWA and go to the login page. Note the login form fields (username, password). 2. Using a tiny, safe wordlist (e.g., 10 common passwords) or a simple manual attempt, try multiple passwords for a known username admin. Keep attempts low and slow — this is a classroom demo. o Example manual attempts: password, 123456, admin123 3. Observe whether the application: o Locks the account after N attempts, or o Introduces delays / captcha, or o Allows unlimited attempts (vulnerable). 4. Controlled automation (if permitted by instructor): use a script with built-in delay (1–2s) and low number of attempts. **Always confirm lab policy before automated testing.**



(c) Root cause

- Lack of protective measures: no account lockout, no throttling/rate limiting, no CAPTCHA for repeated failures, and no multi-factor authentication (MFA).
- Weak password policy or reused/default credentials increase success chance.
- No monitoring or alerting on suspicious login patterns.

(d) Proposed fix

Layered protections to prevent or slow brute-force attacks:

1. **Rate-limiting / throttling**
    - Limit login attempts per IP and per account (e.g., 5 attempts per 15 minutes). Implement exponential backoff delays on repeated failures.
    - Example (pseudo-code):
    - if attempts_for_user > 5 within 15min:
    - deny_login("Too many attempts. Try after 15 minutes.")
    - else:
    - process_login()
    - On a web stack, use tools like fail2ban, mod_security, or application-level logic backed by Redis for counters.
2. **Account lockout / progressive delays**
    - Temporary lockout after N failed attempts; progressive delays on further attempts. Notify user via email when lockout occurs.
3. **CAPTCHA**
    - Insert CAPTCHA after several failed attempts to block automated tools.
4. **Multi-Factor Authentication (MFA)**
    - Add TOTP (Google Authenticator) or SMS/Email OTP for sensitive accounts.
5. **Strong password policies & password storage**
    - Require minimum length & complexity; store passwords with a strong hash (bcrypt/argon2) and salt.
6. **Monitoring & Alerting**

- Log failed attempts and alert on unusual activity (credential stuffing patterns).

Example rate-limiting snippet (PHP + Redis pseudo):

```
$key = "login_attempts:".hash('sha256', $username.$client_ip);
$attempts = $redis->incr($key);
if ($attempts == 1) $redis->expire($key, 900); // 15 minutes
if ($attempts > 5) {
   http_response_code(429);
   die("Too many attempts. Try again later.");
}
```

CVSS-like risk: Medium

C2 — Cross-Site Request Forgery (CSRF)

(a) Vulnerable page

/vulnerabilities/csrf/ (DVWA → CSRF module) — typical endpoints: state-changing actions (change password, email, transfer funds)

(b) Exploit — short reproduction steps (PoC in lab):

1. Login to DVWA as a victim user in one browser tab. Keep the session active (cookie present).
2. Create a malicious HTML file (PoC) hosted on an attacker page or a local file that auto-submits a form to DVWA's state-changing endpoint. Example PoC:
3. <html>
4. <body onload="document.forms[0].submit()">
5. <form action="http://<DVWA_HOST>/vulnerabilities/csrf/?action=change_email" method="POST">
6. <input type="hidden" name="email" value="attacker@example.com">
7. </form>
8. </body>
9. </html>
10. While still logged in as the victim, open the PoC page. The browser will submit the form using the victim's cookies; the site processes the request if no CSRF protection exists.

Evidence:

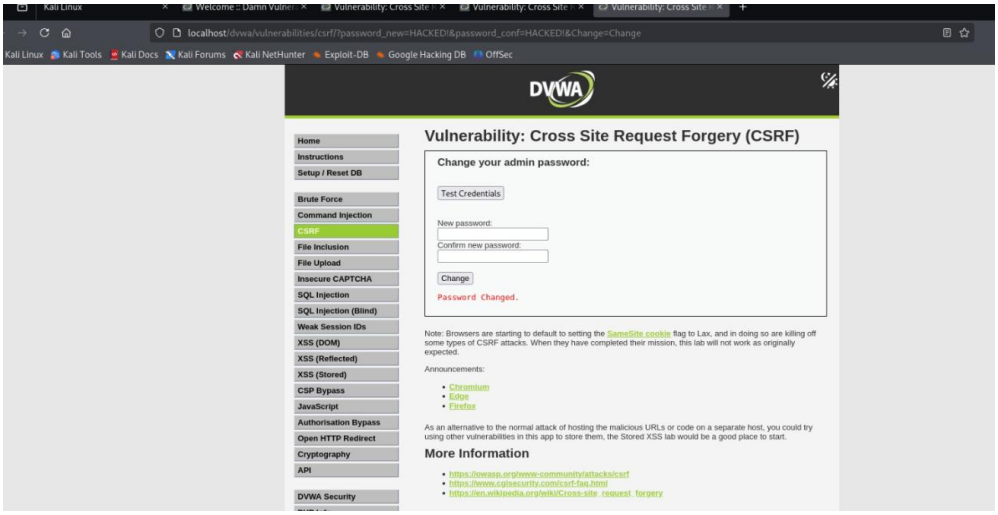
brute.py

```
<html>
<head>
 <title>One-Click Password Change!!</title>
</head>
<body onload="document.forms[0].submit()">
 <h1>Welcome to the Attacker's Blog!!</h1>
 <p>Your Connection is being optimized...</p>

 <form method="GET" action="http://localhost/dvwa/vulnerabilities/csrf/">
  <input type="hidden" name="password_new" value="HACKED!" />
  <input type="hidden" name="password_conf" value="HACKED!" />
  <input type="hidden" name="Change" value="Change" />
 </form>

</body>
</html>
```

poc.html



(c) Root cause

- The server accepts state-changing requests solely based on the presence of valid session cookies. There is no per-request cryptographic token or other assurance that the action was intentionally initiated by the authenticated user.

(d) Proposed fix

- **CSRF tokens (synchronizer token pattern)**: Generate a cryptographically-secure token per session or per form, store it in the server-side session, and require it in state-changing requests. Validate with hash_equals() to prevent timing attacks.

Implementation (PHP example):

```
// On session initialization:
if (empty($_SESSION['csrf_token'])) {
```

```
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}

// When rendering form:
echo '<input type="hidden" name="csrf_token"
value="'.htmlspecialchars($_SESSION['csrf_token']).'">';

// When processing POST:
if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'] ?? '')) {
    http_response_code(403);
    exit('Invalid CSRF token');
}
```

- **SameSite cookie flag**: Set cookies with SameSite=Strict or Lax to reduce cross-site cookie sending.
- **Double-submit cookie pattern**: For stateless APIs, send a cookie with a token and require the same token in a request header/body (less ideal than server-side stored tokens).
- **Ensure unsafe state changes only accept POST and not GET**: GET should be idempotent and not change state.

Additional defense-in-depth:

- Use Content Security Policy and CORS correctly for API endpoints.
- Require re-authentication for particularly sensitive actions.

CVSS-like risk: Medium–High (depends on the sensitivity of the action)

C3 — Insecure Direct Object Reference (IDOR)

(a) Vulnerable page

Examples: /view_profile.php?id=5, /download.php?file=personal_2024.pdf (DVWA modules or custom endpoints that reference objects by predictable identifiers)

(b) Exploit — short reproduction steps (controlled lab):

1. Login as User A (e.g., user1) and view a resource that uses an ID parameter, e.g., view_profile.php?id=2. Capture the URL.
2. Modify the id parameter to another user's ID (e.g., id=3) and submit the request.
3. If the server returns the other user's data without verifying authorization, you've found an IDOR.

(c) Root cause

- The application relies on client-supplied identifiers without server-side authorization checks to ensure the requesting user is permitted to access the referenced object. Object

identifiers are predictable and directly correlate to stored resources.

(d) Proposed fix

1. **Enforce server-side authorization for every resource access:**
   - Before returning a resource, check that the authenticated user has permission to access it (owner or role-based access).
   - Example (pseudo-code):
   - resource = db.getResourceById(id)
   - if resource.owner_id != current_user.id and not current_user.is_admin:
   - return 403 Forbidden
   - return resource
2. **Use indirect references (mapping IDs):**
   - Replace sequential/predictable IDs with random UUIDs or opaque tokens that are hard to guess (e.g., file_ref=ae3b7f2c... mapped server-side to internal records). This prevents simple enumeration but does NOT replace authorization checks.
3. **Least-privilege & logging:**
   - Only present references that a user should see (filter query results by user). Log attempts to access unauthorized resources and alert for abnormal patterns.

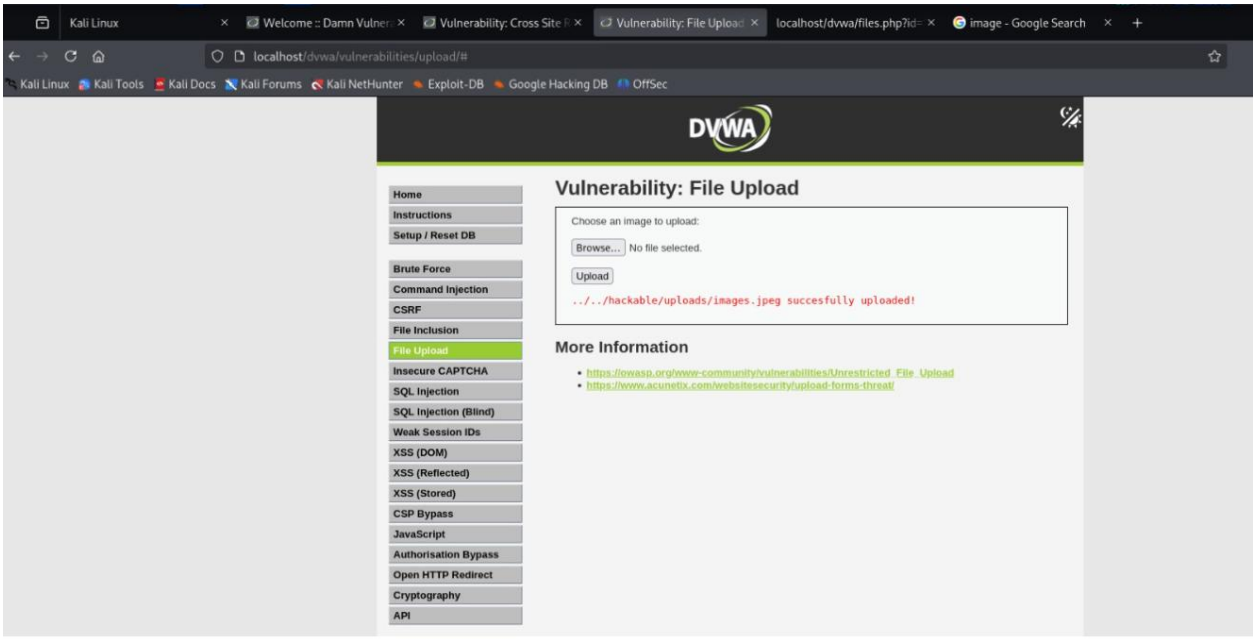
Example safe access code (PHP-like):

```
$id = (int)$_GET['id'];
$sql = "SELECT * FROM documents WHERE id = :id LIMIT 1";
$stmt = $pdo->prepare($sql);
$stmt->execute([':id' => $id]);
$doc = $stmt->fetch();

if (!$doc || $doc['owner_id'] !== $_SESSION['user_id']) {
   http_response_code(403);
   exit('Forbidden');
}
// serve doc
```

CVSS-like risk: High (direct access to other users' personal data)

BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Part D	
PROBLEM STATEMENT:	
PROGRAM:	**D1 — File Upload Vulnerability** **(a) Vulnerable Page** /vulnerabilities/upload/ **(b) Exploit (with lab-safe steps)** 1. Navigate to File Upload module. 2. Upload a valid file (e.g., test.jpg) to confirm upload path. 3. Create a harmless PHP file: 4. <?php echo "Shell Executed"; ?> 5. Rename it to shell.php.jpg and upload. 6. If executed when accessed via the given URL, RCE is possible.  **(c) Root Cause** • Validation only on file extension or client-side checks • No MIME/content verification • Upload directory allows execution of scripts **(d) Proposed Fix** • Server-side MIME validation (finfo_file) • Restrict allowed extensions to strictly safe types • Store uploads outside webroot

- Disable script execution in upload directory (.htaccess or server config)
- Rename uploaded files to random names

Example Secure Code

```
$allowed = ['image/jpeg','image/png'];
$mime = mime_content_type($_FILES['file']['tmp_name']);
if (!in_array($mime, $allowed)) { exit("Invalid file"); }
move_uploaded_file($_FILES['file']['tmp_name'], "/var/uploads/".uniqid().".jpg");
```

CVSS-like Risk: High

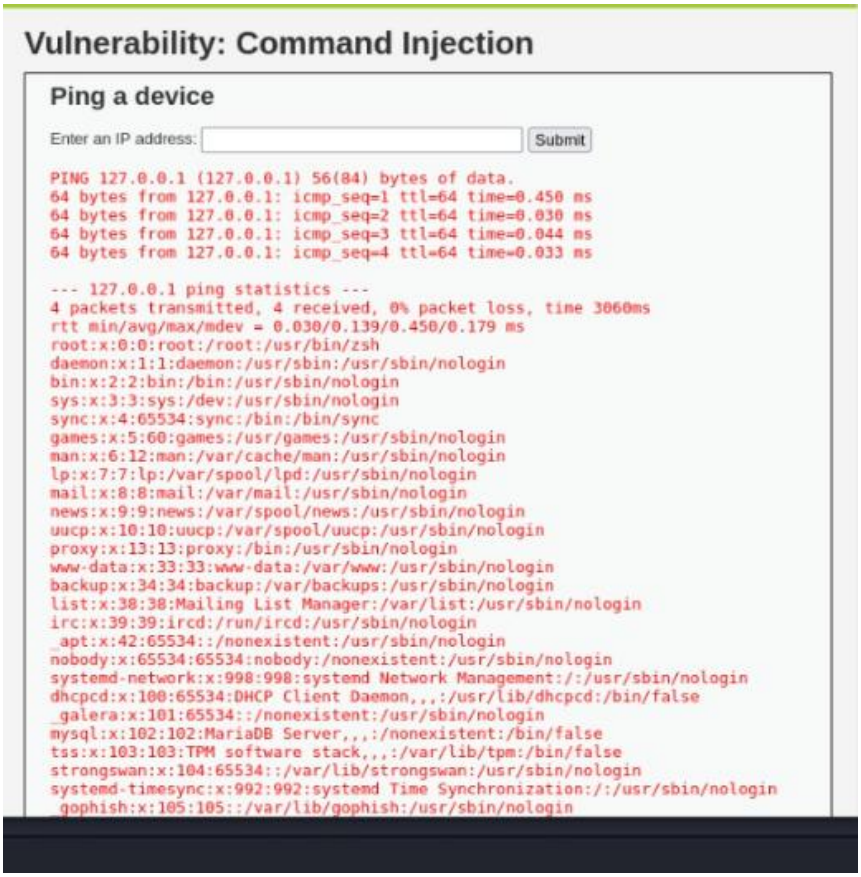
D2 — Command Injection

(a) Vulnerable Page

/vulnerabilities/exec/

(b) Exploit (with lab-safe steps)

1. The page executes system ping commands using input.
2. Inject a harmless command:
3. 127.0.0.1; echo HACKED
4. The echoed word appears in output → command injection confirmed.



(c) Root Cause

- User input directly inserted into OS shell command
- No input validation or escaping

(d) Proposed Fix

- Remove shell-based execution, use safe system libraries instead
- Whitelist allowed characters (e.g., IP format regex)
- Use escapeshellarg() only as fallback
- Apply least-privilege execution environment

Secure Validation Example

```
$ip = $_GET['ip'];
if (!preg_match('/^[0-9\\.]+$/', $ip)) exit("Invalid IP");
$ip = escapeshellarg($ip);
$output = shell_exec("ping -c 1 $ip 2>&1");
```

CVSS-like Risk: High

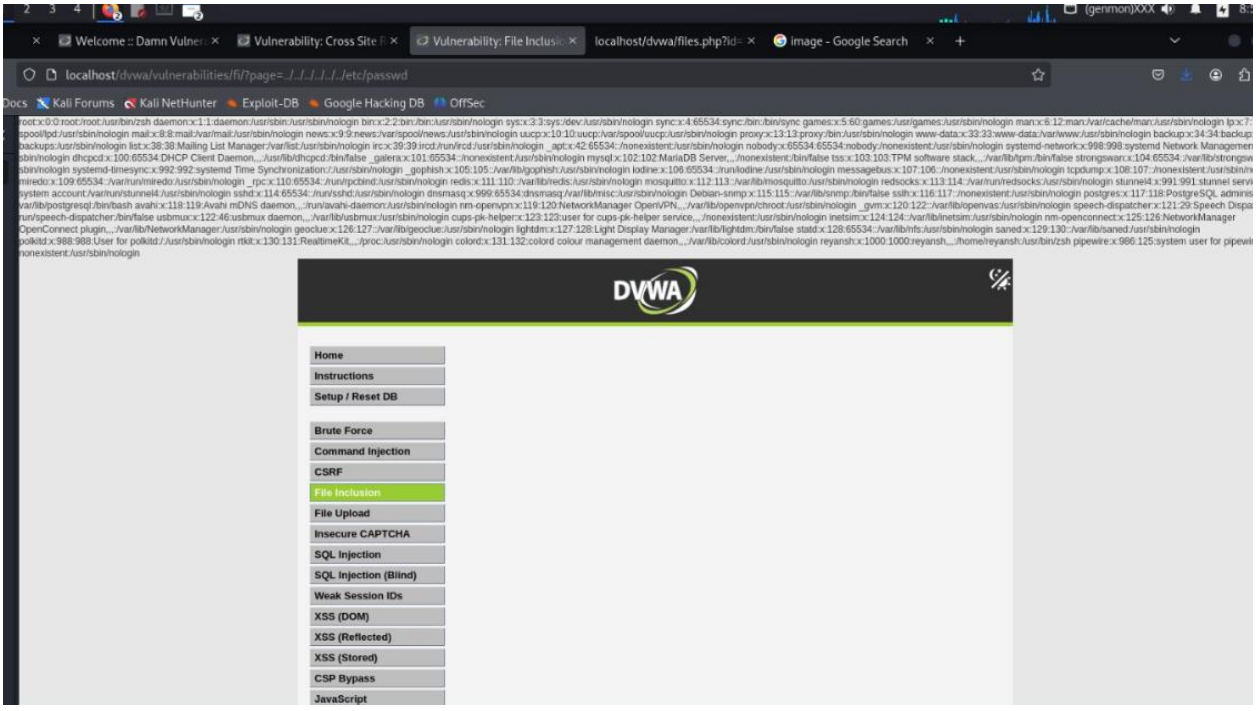
D3 — File Inclusion (LFI / RFI)

(a) Vulnerable Page

/vulnerabilities/fi/

(b) Exploit (lab-safe steps)

1. Page includes file based on page= parameter.
2. Attempt Local File Inclusion:
3. ?page=../../../../etc/passwd
4. If file contents are displayed → vulnerability confirmed.



	(c) Root Cause
	• Direct include() using user-controlled file path
	• No canonicalization or whitelist
	• URL includes allowed → Remote File Inclusion possible
	(d) Proposed Fix
	• Implement strict whitelist of allowed pages:
	`$allowed = [`
	` "home" => "home.php",`
	` "about" => "about.php"`
	`];`
	`if (!isset($allowed[$_GET['page']])) {`
	`  die("Not allowed");`
	`}`
	`include $allowed[$_GET['page']];`
	• Disable allow_url_include in php.ini
	• Prevent ../ traversal using realpath validation
	CVSS-like Risk: High

Part E	
PROBLEM STATEMENT:	For three vulnerabilities you exploited: • Implement fixes (or pseudo-fixes if full changes are invasive) and demonstrate mitigation. • Examples: prepared statements for SQLi, proper output encoding for XSS, CSRF tokens for CSRF, file validation/whitelisting for uploads.
PROGRAM:	**PROBLEM 1** Weak Session IDs / Session Fixation PROGRAM: To mitigate weak/guessable session IDs and session fixation issues, the application was updated to enforce strict session handling and use cryptographically secure tokens. The patch enables strict session mode, forces cookie-only sessions, sets secure cookie attributes (HttpOnly, SameSite, and Secure when served over HTTPS), regenerates the session ID on sensitive actions, and associates a server-side random token with the client via a session-backed cookie. These measures prevent uninitialized or

attacker-supplied session identifiers, reduce the risk of cookie theft via JavaScript, and make session token guessing or fixation significantly harder.

PHP session handling

```
<?php
// improved.php - Secure session handling example
// NOTE: Remove debug output in production. Requires PHP 7.3+ for array
// cookie params.

// Harden session configuration at runtime (or set these in php.ini)
ini_set('session.use_strict_mode', '1');   // refuse uninitialized session IDs
ini_set('session.use_only_cookies', '1'); // prevent SID in URL
ini_set('session.cookie_httponly', '1');  // JavaScript cannot read the cookie
// Enable the next line when serving over HTTPS:
// ini_set('session.cookie_secure', '1');

// Cookie params - set before session_start()
$cookie_lifetime = 0; // session cookie (expires on browser close)
$cookie_path     = '/';
$cookie_domain   = ''; // e.g. '.example.com' if needed
$cookie_secure   = isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] !== 'off';
$cookie_httponly = true;
$cookie_samesite = 'Lax'; // 'Strict' or 'Lax' recommended

session_set_cookie_params([
   'lifetime' => $cookie_lifetime,
   'path'     => $cookie_path,
   'domain'   => $cookie_domain,
   'secure'   => $cookie_secure,
   'httponly' => $cookie_httponly,
   'samesite' => $cookie_samesite
]);

session_name('DVWA_SESSID'); // optional: custom session cookie name
session_start();

// Helper: cryptographically secure token generator
function random_token(int $bytes = 32): string {
```

```
    return bin2hex(random_bytes($bytes)); // 64 hex chars for 32 bytes
}

// When a new session-like identity is created (e.g., on login), regenerate ID
// and set token
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // Prevent session fixation by regenerating session id and deleting old session
    session_regenerate_id(true);

    // Store a server-side token in session
    $token = random_token(32);
    $_SESSION['dvwa_token'] = $token;
    $_SESSION['created_at'] = time();

    // Optional: set a cookie for compatibility (prefer HttpOnly & Secure)
    setcookie('dvwaSession', $token, [
        'expires'  => 0,              // session cookie
        'path'     => $cookie_path,
        'domain'   => $cookie_domain,
        'secure'   => $cookie_secure,   // requires HTTPS to be effective
        'httponly' => true,
        'samesite' => $cookie_samesite
    ]);

    // Optionally redirect to post-login page
    // header('Location: /'); exit;
}

// On every request validate dvwaSession cookie against server-side session token
$valid = false;
if (!empty($_COOKIE['dvwaSession']) && !empty($_SESSION['dvwa_token'])) {
    if (hash_equals($_SESSION['dvwa_token'], $_COOKIE['dvwaSession'])) {
        $valid = true;
    }
}

// If token invalid (possible tampering/fixation), rotate session and clear token
if (!$valid && isset($_COOKIE['dvwaSession'])) {
```

```
    session_regenerate_id(true);
    unset($_SESSION['dvwa_token']);
    setcookie('dvwaSession', '', [
        'expires' => time() - 3600,
        'path'    => $cookie_path,
        'domain'  => $cookie_domain,
        'secure'  => $cookie_secure,
        'httponly'=> true,
        'samesite'=> $cookie_samesite
    ]);
    // Optionally force re-authentication here
}

// DEBUG: remove in production
if (defined('DEBUG') && DEBUG) {
    echo '<pre>';
    echo 'Session ID: ' . session_id() . PHP_EOL;
    echo 'DVWA token in session: ' . ($_SESSION['dvwa_token'] ?? 'none') . PHP_EOL;
    echo 'Cookie dvwaSession: ' . ($_COOKIE['dvwaSession'] ?? 'none') . PHP_EOL;
    echo '</pre>';
}
?>
```

PROBLEM 2 Insecure Direct Object Reference (IDOR) & SQL Injection (SQLi)

To mitigate Insecure Direct Object References (IDOR) and SQL Injection (SQLi) vulnerabilities, the user lookup functionality was updated to enforce strict input validation and parameterized database queries. Only positive integers or valid UUIDs are accepted as user identifiers, preventing attackers from accessing arbitrary records. All database access uses prepared statements with bound parameters, ensuring that user input cannot alter SQL commands. Additionally, CSRF tokens protect POST requests, and all output is safely encoded using htmlspecialchars() to prevent reflected XSS. These combined measures ensure that users can only access their own data through authorized actions and that injection or tampering attacks are effectively blocked.

```
<?php
declare(strict_types=1);

// secure_user_lookup.php
```

```
// --- Basic hardening for session cookie (adjust if your app already sets these) ---
ini_set('session.use_strict_mode', '1');
ini_set('session.use_only_cookies', '1');
session_set_cookie_params([
   'lifetime' => 0,
   'path'     => '/',
   'secure'   => isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] !== 'off',
   'httponly' => true,
   'samesite' => 'Lax'
]);
session_start();

// --- Simple CSRF helper (for form POSTs) ---
function csrf_token(): string {
   if (empty($_SESSION['csrf_token'])) {
      $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
   }
   return $_SESSION['csrf_token'];
}
function verify_csrf(string $token): bool {
   return isset($_SESSION['csrf_token']) && hash_equals($_SESSION['csrf_token'], $token);
}

// --- Validation helpers ---
// Validate an integer user id (positive int)
function validate_int_id($value): ?int {
   if ($value === null) return null;
   $options = ['options' => ['min_range' => 1]];
   $int = filter_var($value, FILTER_VALIDATE_INT, $options);
   return ($int === false) ? null : (int)$int;
}

// Validate UUID v4 (if your system uses UUIDs instead of numeric IDs)
function validate_uuid(string $value): ?string {
   $value = trim($value);
   if (preg_match('/^[0-9a-fA-F]{8}\-[0-9a-fA-F]{4}\-4[0-9a-fA-F]{3}\-[89abAB][0-9a-fA-
F]{3}\-[0-9a-fA-F]{12}$/', $value)) {
```

```
        return $value;
    }
    return null;
}

// --- Database connection using PDO (replace with your credentials) ---
$dsn = 'mysql:host=127.0.0.1;dbname=your_database;charset=utf8mb4';
$dbUser = 'your_db_user';
$dbPass = 'your_db_password';

try {
    $pdo = new PDO($dsn, $dbUser, $dbPass, [
        PDO::ATTR_ERRMODE           => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => false, // IMPORTANT: use native prepares
    ]);
} catch (PDOException $e) {
    // Log the error server-side and show generic message to user
    error_log('PDO connection failed: ' . $e->getMessage());
    http_response_code(500);
    echo 'Internal server error';
    exit;
}

// --- Handle POST form submission (preferred over GET for actions) ---
$user = null;
$errors = [];

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    // CSRF check
    $posted_csrf = $_POST['csrf_token'] ?? '';

    if (!verify_csrf($posted_csrf)) {
        $errors[] = 'Invalid request (CSRF).';
    } else {
        // Determine your ID type: integer or UUID.
        // Example: we try integer first, then UUID fallback.
        $raw_id = $_POST['user_id'] ?? '';
```

```
    $int_id = validate_int_id($raw_id);
    $uuid   = is_string($raw_id) ? validate_uuid($raw_id) : null;

    if ($int_id !== null) {
        // Parameterized SELECT using integer ID
        $sql = 'SELECT user_id, username, email FROM users WHERE user_id = :id LIMIT 1';
        $stmt = $pdo->prepare($sql);
        $stmt->bindValue(':id', $int_id, PDO::PARAM_INT);
        $stmt->execute();
        $user = $stmt->fetch();
    } elseif ($uuid !== null) {
        // Parameterized SELECT using UUID
        $sql = 'SELECT id AS user_id, username, email FROM users WHERE id = :uuid LIMIT
1';
        $stmt = $pdo->prepare($sql);
        $stmt->bindValue(':uuid', $uuid, PDO::PARAM_STR);
        $stmt->execute();
        $user = $stmt->fetch();
    } else {
        $errors[] = 'Invalid User ID format.';
    }

    if ($user === false) {
        // No user found
        $user = null;
        $errors[] = 'User not found.';
    }
  }
}
?>
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Secure User Lookup</title>
</head>
<body>
```

```
<h1>Lookup User</h1>

<!-- show errors -->
<?php if (!empty($errors)): ?>
  <div role="alert">
    <ul>
      <?php foreach ($errors as $err): ?>
        <li><?php echo htmlspecialchars($err, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
?></li>
      <?php endforeach; ?>
    </ul>
  </div>
<?php endif; ?>

<form method="post" action="">
  <label for="user_id">User ID (int or UUID):</label>
  <input id="user_id" name="user_id" type="text" required maxlength="100" pattern="[0-9\-a-
fA-F]+" />
  <input type="hidden" name="csrf_token" value="<?php echo htmlspecialchars(csrf_token(),
ENT_QUOTES, 'UTF-8'); ?>">
  <button type="submit">Lookup</button>
</form>

<?php if ($user): ?>
  <h2>User info</h2>
  <ul>
    <li>User ID: <?php echo htmlspecialchars((string)$user['user_id'], ENT_QUOTES, 'UTF-8');
?></li>
    <li>Username: <?php echo htmlspecialchars((string)$user['username'], ENT_QUOTES, 'UTF-
8'); ?></li>
    <li>Email: <?php echo htmlspecialchars((string)$user['email'], ENT_QUOTES, 'UTF-8');
?></li>
  </ul>
<?php endif; ?>

</body>
</html>
```

PROBLEM 3 Reflected Cross-Site Scripting (Reflected XSS)
PROGRAM To mitigate reflected XSS, the search form was updated with strict input validation, context-aware output encoding, and defense-in-depth headers.
User input is validated against an allowlist, limiting length and disallowed characters. All echoed values are safely encoded using htmlspecialchars() for HTML context and json_encode() for inline JavaScript. Additionally, a nonce-based Content Security Policy (CSP) is applied to allow only trusted scripts. These measures prevent arbitrary scripts from executing, ensuring that user input cannot compromise other users' browsers.

```
<?php
declare(strict_types=1);

// secure_reflected_xss.php - example to prevent reflected XSS

// --- Session and cookie hardening (optional if already configured globally) ---
ini_set('session.use_strict_mode', '1');
ini_set('session.use_only_cookies', '1');
session_set_cookie_params([
    'lifetime' => 0,
    'path'     => '/',
    'secure'   => isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] !== 'off',
    'httponly' => true,
    'samesite' => 'Lax',
]);

session_start();

// --- Generate CSP nonce for safe inline scripts (defense-in-depth) ---
if (empty($_SESSION['csp_nonce'])) {
    $_SESSION['csp_nonce'] = bin2hex(random_bytes(16));
}
$csp_nonce = $_SESSION['csp_nonce'];

// Set secure response headers
header("X-Content-Type-Options: nosniff");
header("Referrer-Policy: no-referrer-when-downgrade");
header("Permissions-Policy: geolocation=()");
header("Content-Security-Policy: default-src 'self'; script-src 'self' 'nonce-{$csp_nonce}'; object-
```

```
src 'none'; base-uri 'self';");

// --- Output-encoding helper functions ---
function escape_html(string $s): string {
    return htmlspecialchars($s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}
function escape_attr(string $s): string {
    return htmlspecialchars($s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
}
function js_literal($value): string {
    $json = json_encode($value, JSON_UNESCAPED_SLASHES |
JSON_UNESCAPED_UNICODE);
    return $json === false ? 'null' : $json;
}

// --- Input validation / allowlist ---
function validate_search_query($raw): ?string {
    if (!is_string($raw)) return null;
    $trimmed = trim($raw);
    if ($trimmed === '' || mb_strlen($trimmed, 'UTF-8') > 200) return null;
    if (preg_match('/^[\p{L}\p{N}\s\-\.\,\_@#&\(\)\'"]+$/u', $trimmed)) {
        return $trimmed;
    }
    return null;
}

// --- Process request ---
$search = null;
$errors = [];
if ($_SERVER['REQUEST_METHOD'] === 'GET' || $_SERVER['REQUEST_METHOD'] ===
'POST') {
    $raw = $_REQUEST['q'] ?? null;
    $validated = validate_search_query($raw);
    if ($validated === null) {
        if ($raw !== null && trim((string)$raw) !== '') {
            $errors[] = 'Your input contained invalid characters or was too long. Please change it.';
        }
    } else {
```

```
        $search = $validated;
        // Safe DB query placeholder: use prepared statements
        $results = [
            ['title' => 'Result 1 about ' . $search, 'summary' => "Summary for {$search}"],
            ['title' => 'Result 2', 'summary' => 'Another item']
        ];
    }
}

// --- HTML Output ---
?>
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Secure Search (Reflected XSS protected)</title>
<meta name="viewport" content="width=device-width,initial-scale=1">
</head>
<body>
  <h1>Search</h1>

  <?php if (!empty($errors)): ?>
    <div role="alert">
      <ul>
        <?php foreach ($errors as $e): ?>
          <li><?php echo escape_html($e); ?></li>
        <?php endforeach; ?>
      </ul>
    </div>
  <?php endif; ?>

  <form method="get" action="">
    <label for="q">Query:</label>
    <input id="q" name="q" type="text" maxlength="200" value="<?php echo $search !== null ?
escape_attr($search) : ''; ?>">
    <button type="submit">Search</button>
  </form>
```

```
<?php if ($search !== null): ?>
  <h2>Showing results for: <?php echo escape_html($search); ?></h2>
  <ul>
    <?php foreach ($results as $r): ?>
      <li>
        <strong><?php echo escape_html($r['title']); ?></strong><br>
        <small><?php echo escape_html($r['summary']); ?></small>
      </li>
    <?php endforeach; ?>
  </ul>
<?php elseif ($_REQUEST['q'] ?? false): ?>
  <p>We couldn't use that input. Please try again with valid characters.</p>
<?php endif; ?>

<script nonce="<?php echo $csp_nonce; ?>">
  const serverData = <?php echo js_literal(['search' => $search ?? '', 'timestamp' => time()]); ?>;
  console.log('serverData:', serverData);
</script>

</body>
</html>
```

CONCLUSION:	This practical reinforced the significance of secure development practices in real-world web applications. By analyzing and exploiting vulnerabilities such as SQL Injection, XSS, CSRF, Session Weaknesses, and IDOR within DVWA, we gained insight into how attackers manipulate insecure input handling and flawed authentication mechanisms. Implementing proper countermeasures including prepared statements, sanitization and encoding, CSRF protection, and hardened session management ensured that the application could no longer be abused in the same manner. The experiment successfully demonstrated that thoughtful defensive coding and continuous security testing are essential to maintaining a secure web ecosystem.