



**BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY**

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India

Department of Computer Engineering

EXPERIMENT NO. 7	
Name.	Uzmah Yusuf Shaikh
Class.	TY - Computer Engg - Division D - Batch D
Aim.	To identify, exploit, and mitigate common web application vulnerabilities using the Damn Vulnerable Web Application (DVWA).

Executive Summary

This report details a security assessment of the Damn Vulnerable Web Application (DVWA) hosted on a LAMP stack within a Kali Linux virtual environment. Several critical vulnerabilities were identified and successfully exploited, including SQL Injection, Cross-Site Scripting (Reflected and Stored), Command Injection, Cross-Site Request Forgery (CSRF), and insecure File Uploads leading to Remote Code Execution. This document provides the steps to reproduce each vulnerability, explains the underlying root cause, and demonstrates effective remediation techniques for the most critical findings. The results underscore the importance of implementing secure coding practices such as input validation, parameterized queries, and output encoding to defend modern web applications against compromise.

Setup Notes

The lab environment was configured to safely test and analyze web application vulnerabilities.

- **Virtualization:** Oracle VirtualBox
- **Operating System:** Kali Linux
- **Server Stack:** LAMP (Linux, Apache2, MariaDB, PHP)
- **Application:** Damn Vulnerable Web Application (DVWA) cloned from the official [digininja](#) GitHub repository.
- **Networking:** The VM was configured in NAT mode, with all testing performed locally within the virtual machine to prevent any external impact. The IP address used for access was internal (e.g., 10.0.2.X).

Key Setup Commands:

```
# Package Installation
```

```
sudo apt install -y apache2 mariadb-server php php-mysql git
```

```
# Database Security
```

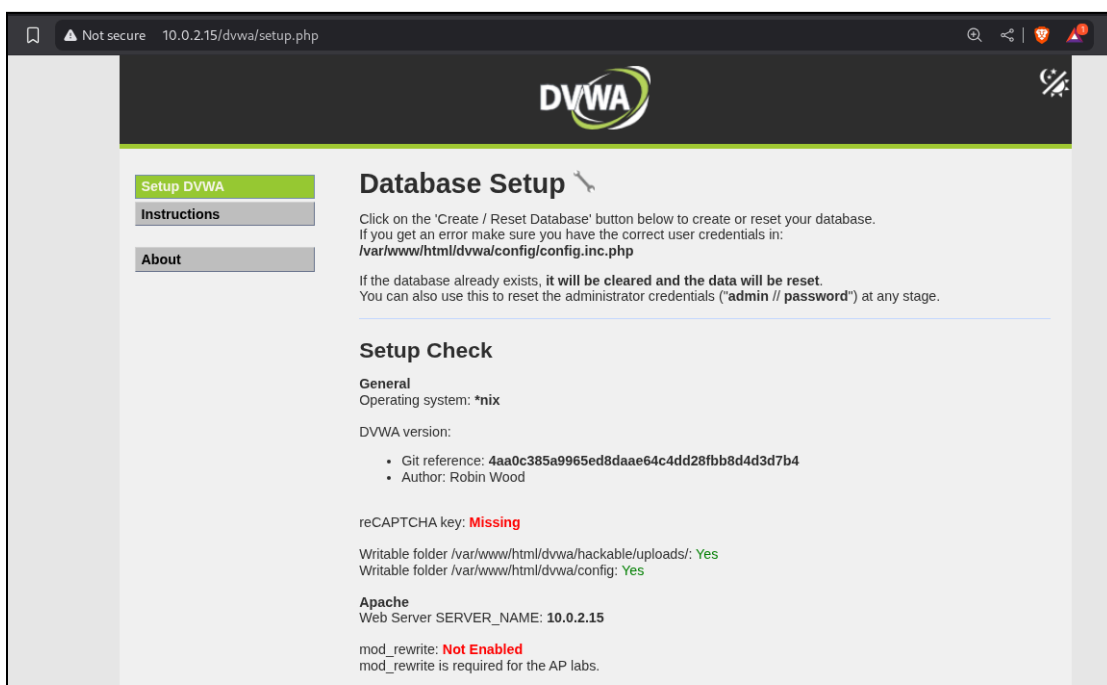
```
sudo mariadb-secure-installation
```

```
# DVWA Setup
git clone https://github.com/digininja/DVWA.git
sudo mv DVWA /var/www/html/dvwa
sudo chown -R www-data:www-data /var/www/html/dvwa
```

Part A — Setup & Baseline

1. DVWA Running

The DVWA instance was successfully installed and configured. The database was created via the web setup page, and the application was accessible.





2. Security Level Settings

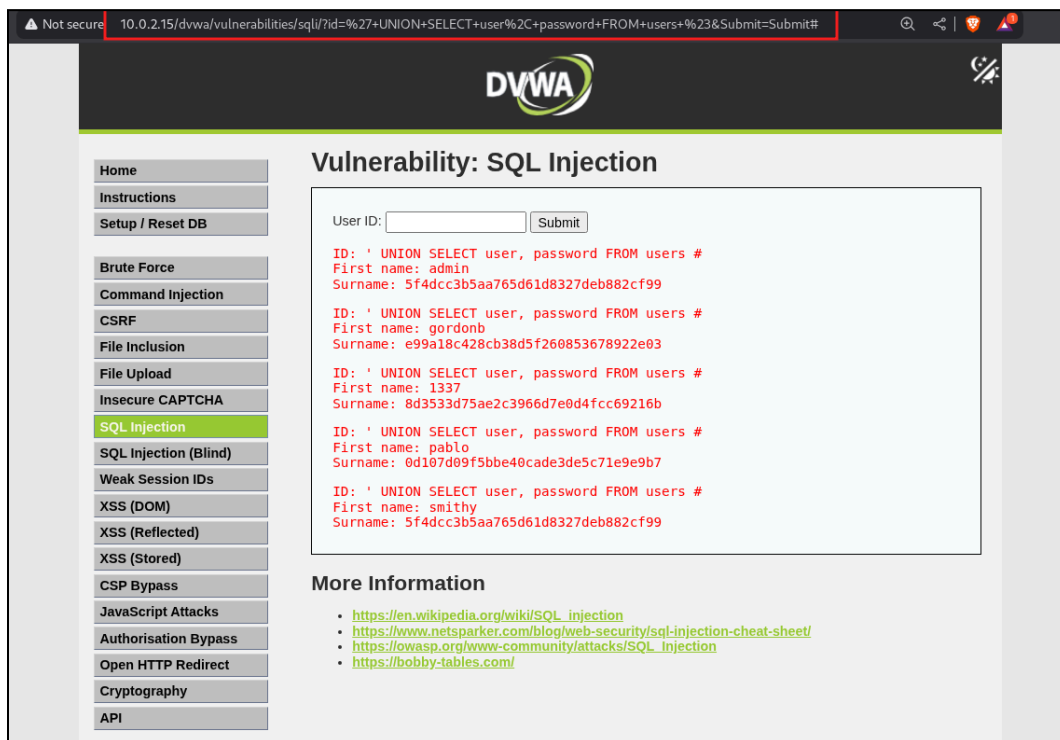
DVWA provides multiple security levels to demonstrate different defense mechanisms.

- **Low:** This level has virtually no security measures. It is designed to be trivially exploitable and is useful for learning the basic mechanics of an attack. Code at this level often directly uses user input without any validation or sanitization.
- **Medium:** This level introduces basic security controls, often implemented imperfectly. For example, it might use client-side validation (which can be bypassed) or simple blacklisting of characters (which can often be circumvented with different attack variations). It demonstrates common but flawed attempts at security.
- **High:** This level implements stronger, more robust security measures that are much harder to bypass, such as session ID tokens for CSRF, better whitelisting for command injection, and more secure file upload checks. It is designed to model a more securely coded application.

Part B — Basic Vulnerabilities

1. SQL Injection (SQLi)

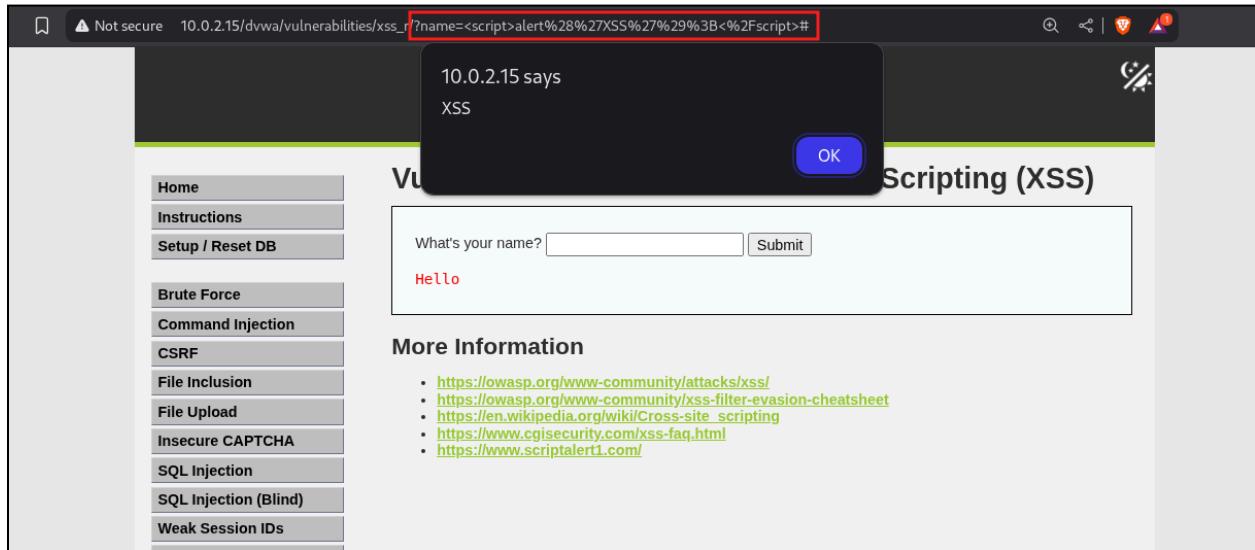
- (a) **Vulnerable Page:** SQL Injection (/vulnerabilities/sqli/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. In the "User ID" input field, the following payload was entered: `' UNION SELECT user, password FROM users #`
 3. Upon submission, the application executed the injected query, displaying all usernames and their corresponding MD5 password hashes from the `users` table.



- (c) **Root Cause:** The application constructs its SQL query by directly concatenating raw user input. An attacker can provide a specially crafted string that breaks the original query and injects a new, malicious query.
- (d) **Proposed Fix:** The use of **parameterized queries (prepared statements)**. This technique separates the SQL code from the user-provided data, ensuring the input is always treated as data and never as executable code.

2. Reflected Cross-Site Scripting (XSS)

- (a) **Vulnerable Page:** XSS (Reflected) (/vulnerabilities/xss_r/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. In the "What's your name?" input field, the following JavaScript payload was entered:
`<script>alert('XSS')</script>`
 3. Upon submission, the server reflected this payload back to the browser, which executed the script and displayed an alert pop-up box.

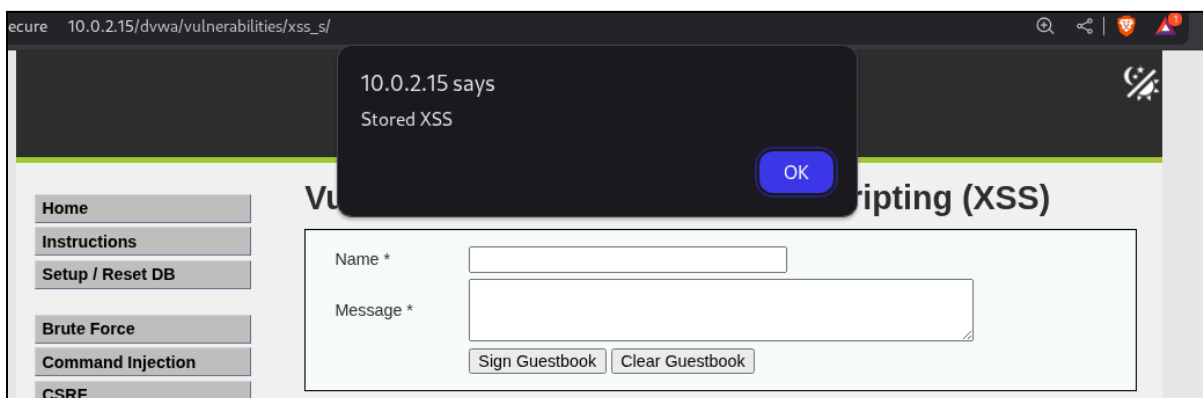
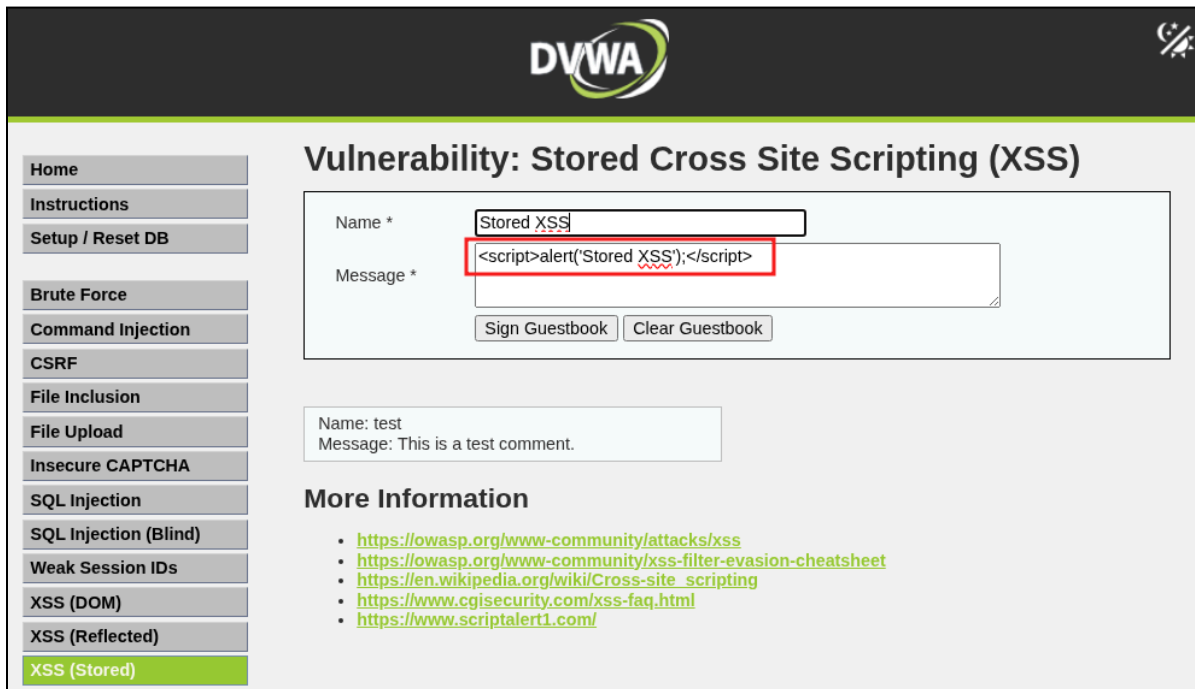


- (c) **Root Cause:** The application includes raw user input directly into the HTML of the response page without any output encoding. The browser cannot distinguish the malicious script from legitimate HTML and executes it. This could be used to steal session cookies and hijack user sessions.
- (d) **Proposed Fix:** Implement **context-aware output encoding**. Specifically, use a function like PHP's `htmlspecialchars()` to convert characters like `<` and `>` into their HTML entity equivalents (`<` and `>`), rendering them harmless.

3. Stored Cross-Site Scripting (XSS)

- (a) **Vulnerable Page:** XSS (Stored) (/vulnerabilities/xss_s/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. In the "Message" field of the guestbook, the following JavaScript payload was entered:

```
<script>alert('Stored XSS')</script>
```
 3. This payload was saved to the database.
 4. Every time any user visited the Stored XSS page, the script was loaded from the database and executed, triggering the alert pop-up box.

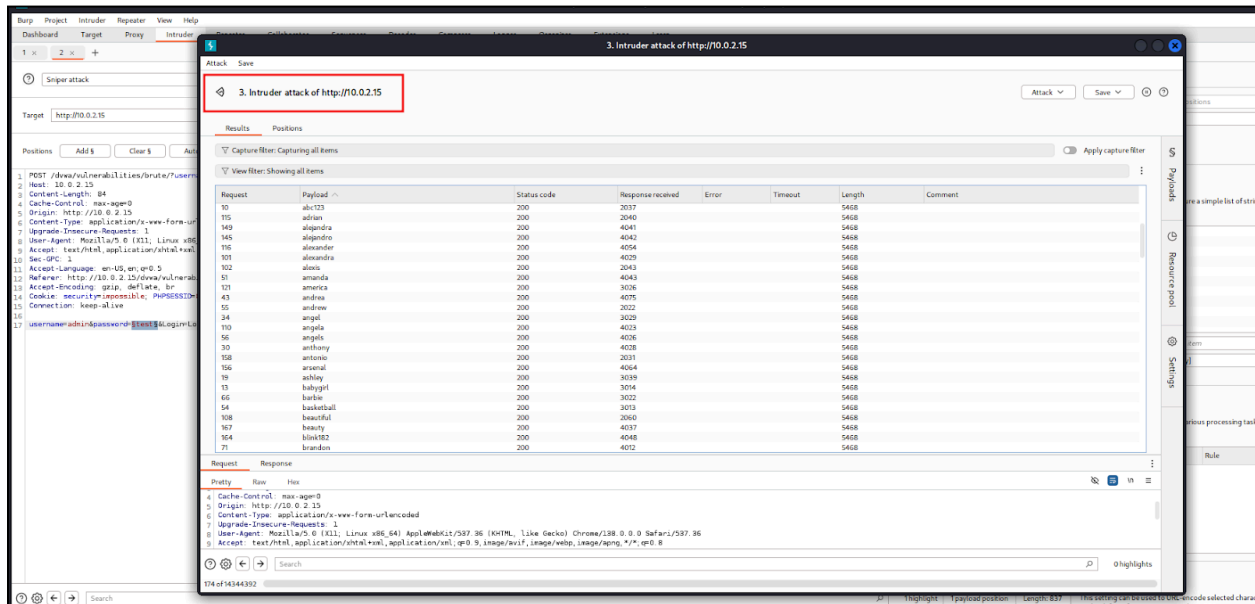


- (c) **Root Cause:** The application stores raw user input in its database without sanitization or validation. When this stored data is later retrieved and displayed, it is rendered directly into the HTML, causing the malicious script to execute in the browser of every visitor.
- (d) **Proposed Fix:** The fix requires a combination of input validation upon submission and, crucially, context-aware output encoding (`htmlspecialchars()`) whenever the stored data is displayed to a user.

Part C — Auth / Session / Logic Problems

1. Brute Force

- **Vulnerable Page:** Brute Force (`/vulnerabilities/brute/`) and the main Login page (`/login.php`).
- **Demonstration:**
 1. At the **low** security level, there are no protections against repeated login attempts.
 2. A tool like Burp Suite Intruder or Hydra can be used to automate login attempts.
 3. The username was set to `admin`, and a password list (like `rockyou.txt`) was used as the payload.
 4. The tool was configured to send login requests for each password. The correct password (`password`) was identified by observing a different HTTP response length or a redirect status code (302).

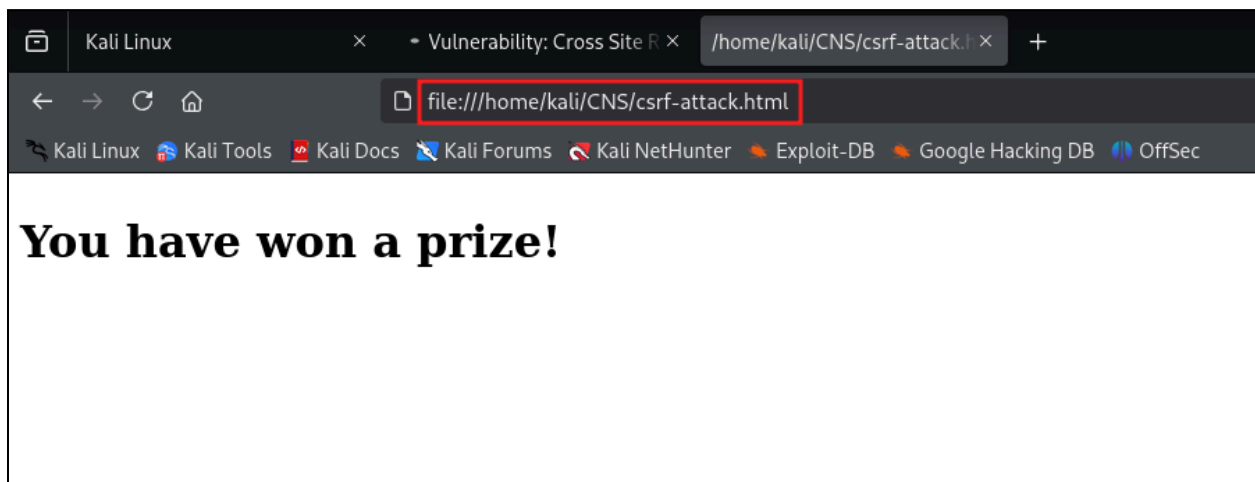
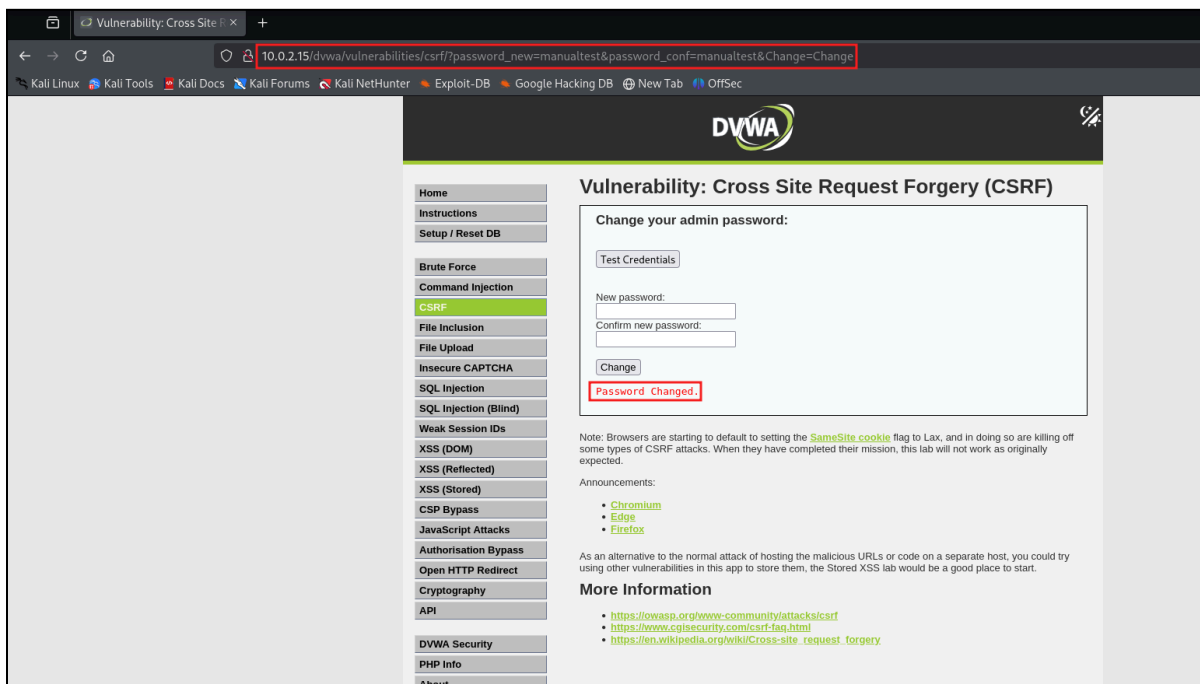


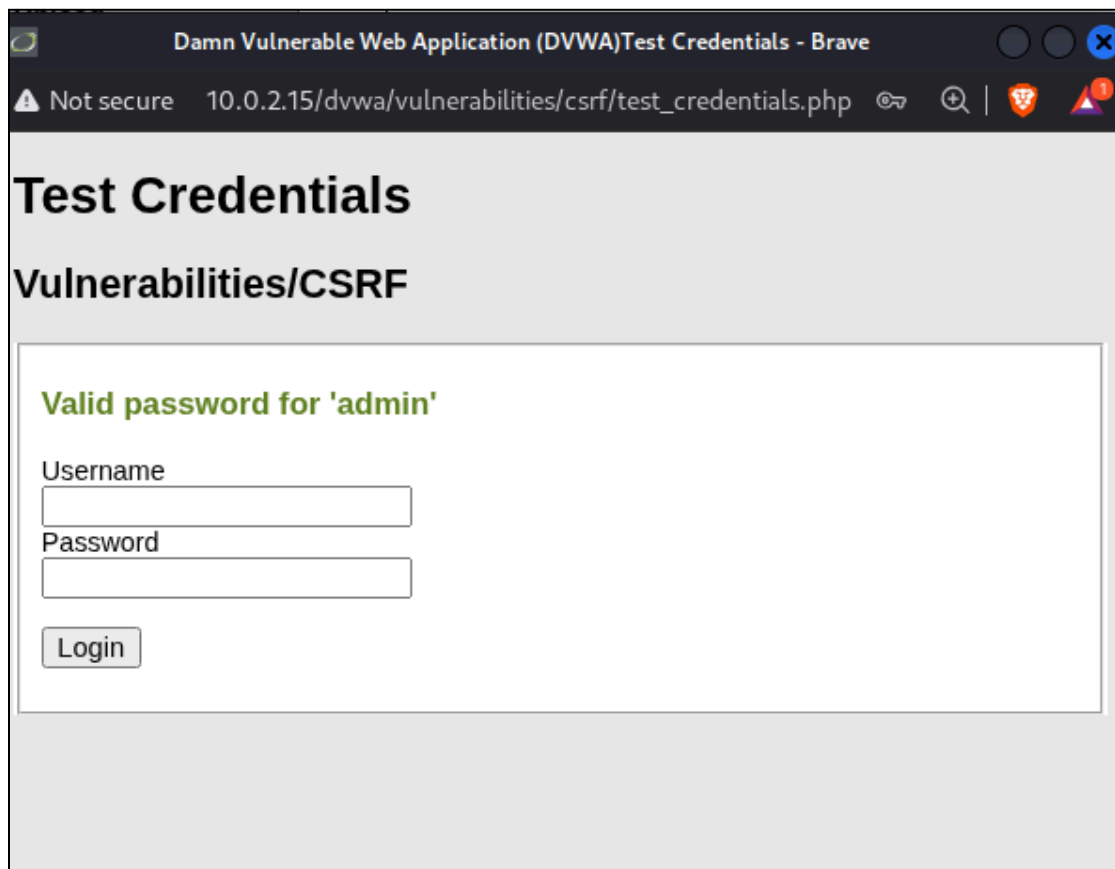
- **Root Cause:** The application fails to implement any rate-limiting, account lockout, or CAPTCHA mechanisms to slow down or block a large number of failed login attempts from a single source.
- **Proposed Fix:** Implement account lockout policies (e.g., lock account for 15 minutes after 5 failed attempts) and/or use a CAPTCHA mechanism after a few failed attempts.

2. Cross-Site Request Forgery (CSRF)

- (a) **Vulnerable Page:** CSRF (/vulnerabilities/csrf/)
- (b) **Exploit:**
 1. The security level was set to **low**, and the user was logged into DVWA.
 2. A proof-of-concept HTML file (csrf-attack.html) was created with the following content:

```
<html><body></body></html>
```
 3. When the logged-in user opened this HTML file, their browser automatically sent a GET request to the DVWA server to change the password to "hacked", including their active session cookie. The password was changed without the user's knowledge.



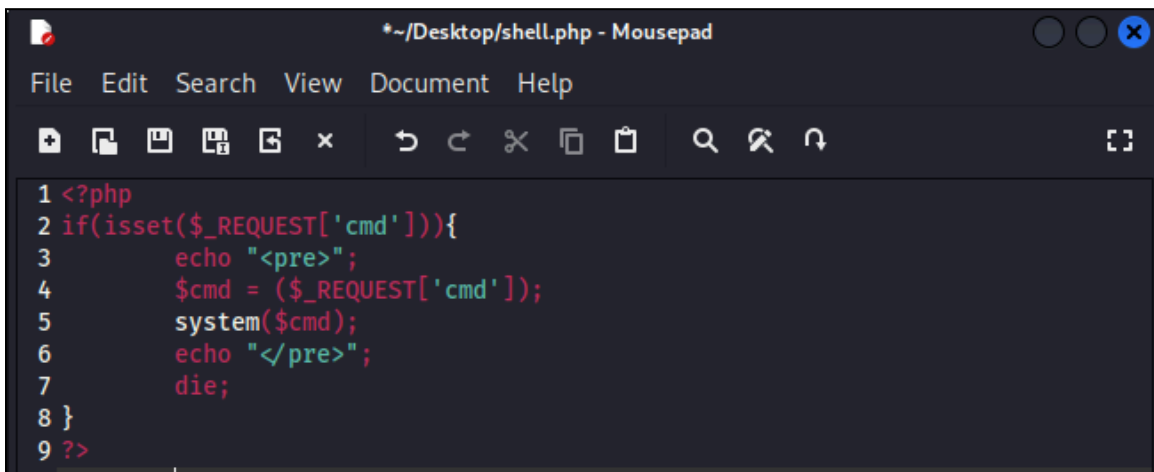
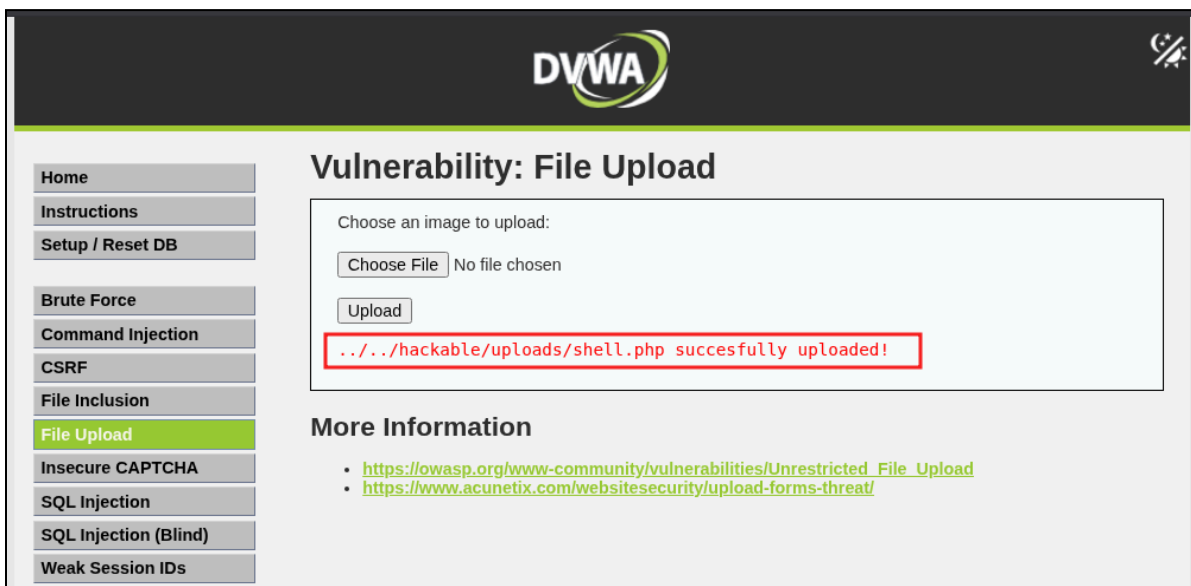


- **(c) Root Cause:** The application only relies on a valid session cookie to authorize a state-changing action (like changing a password). It does not validate a unique, secret token to ensure the request originated from the application's own form.
- **(d) Proposed Fix:** Implement **anti-CSRF tokens**. A unique, unpredictable token should be embedded in a hidden field in every form. The server must validate that this token is present and correct before processing the request.

Part D — File/Functionality Exploitation

1. File Upload Vulnerability

- (a) **Vulnerable Page:** File Upload (/vulnerabilities/upload/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. A simple PHP web shell was created and saved as `shell.php`.
 3. This `shell.php` file was successfully uploaded via the web form. At the low level, the application only performs a basic check on the `Content-Type` header, which is easily bypassed.
 4. By navigating to the `/hackable/uploads/shell.php?cmd=whoami` URL, arbitrary system commands could be executed on the server.



- (c) **Root Cause:** The application has weak validation controls for file uploads. It fails to properly check the file extension, file content (magic numbers), or store the file in a non-executable location.
- (d) **Proposed Fix:** Implement a multi-layered defense: whitelist allowed file extensions, verify the file's MIME type on the server, rename uploaded files to a random string, and store them in a directory outside of the web root

2. Command Injection

- (a) **Vulnerable Page:** Command Injection (/vulnerabilities/exec/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. The application expects an IP address to ping. A malicious payload was crafted using the `&&` shell operator: `127.0.0.1 && whoami`
 3. The server executed the `ping` command and then executed the injected `whoami` command, revealing the server's user (`www-data`).

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.037 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.046 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.038 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3069ms  
rtt_min/avg/max/mdev = 0.025/0.036/0.046/0.007 ms  
www-data
```

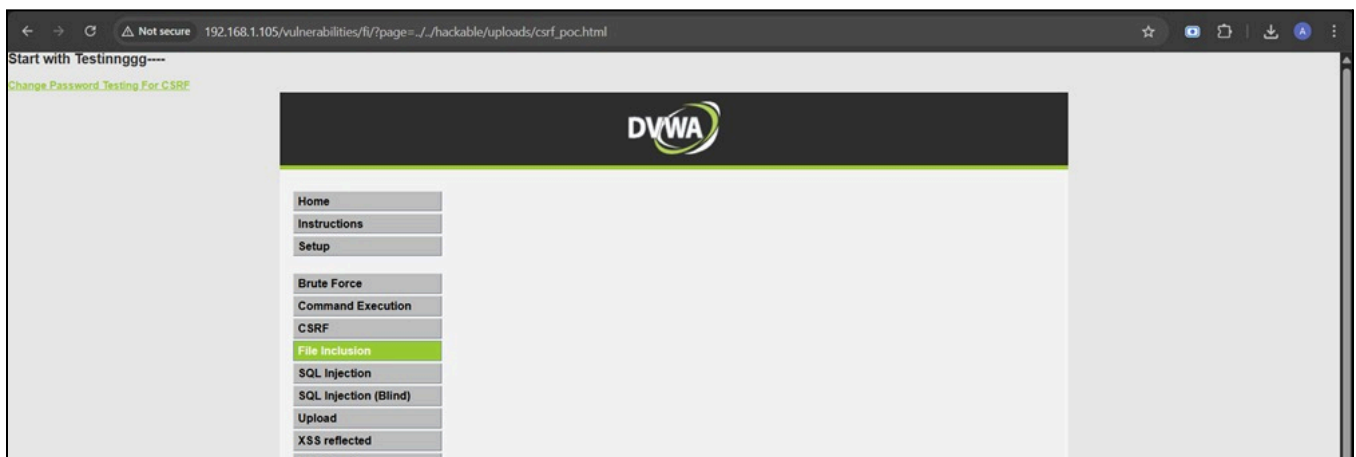
More Information

- <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

- (c) **Root Cause:** The application passes unsanitized user input directly to a system shell execution function (`shell_exec()`). An attacker can inject shell metacharacters to execute arbitrary commands.
- (d) **Proposed Fix:** Sanitize the input by removing or escaping dangerous characters. The best approach is to use a strict whitelist that only allows characters necessary for the expected input (e.g., numbers and dots for an IP address).

3. File Inclusion

- (a) **Vulnerable Page:** File Inclusion (/vulnerabilities/fi/)
- (b) **Exploit:**
 1. The security level was set to **low**.
 2. The page uses a URL parameter (`page`) to include files. A path traversal payload was used to access files outside the intended directory: `../../../../etc/passwd`
 3. By navigating to `../../../../vulnerabilities/fi/?page=../../../../etc/passwd`, the contents of the server's password file were successfully displayed on the web page.



- (c) **Root Cause:** The application uses user-controlled input directly in a file inclusion function (`include()`) without proper validation or sanitization. This allows an attacker to traverse the file system and read arbitrary files.
- (d) **Proposed Fix:** Avoid using user input in file path functions. If necessary, validate the user input against a strict whitelist of known-good, expected file values (e.g., `'file1.php'`, `'file2.php'`).

Part E — Defense & Remediation

In this section, fixes were implemented for three critical vulnerabilities, and their effectiveness was verified.

1. Remediation for SQL Injection

- **Fix Implemented:** The vulnerable, direct query concatenation was replaced with parameterized queries (prepared statements) using the `mysqli_prepare`, `mysqli_stmt_bind_param`, and `mysqli_stmt_execute` functions. This ensures a complete separation between the SQL command and user-supplied data.
- **Evidence of Fix:**

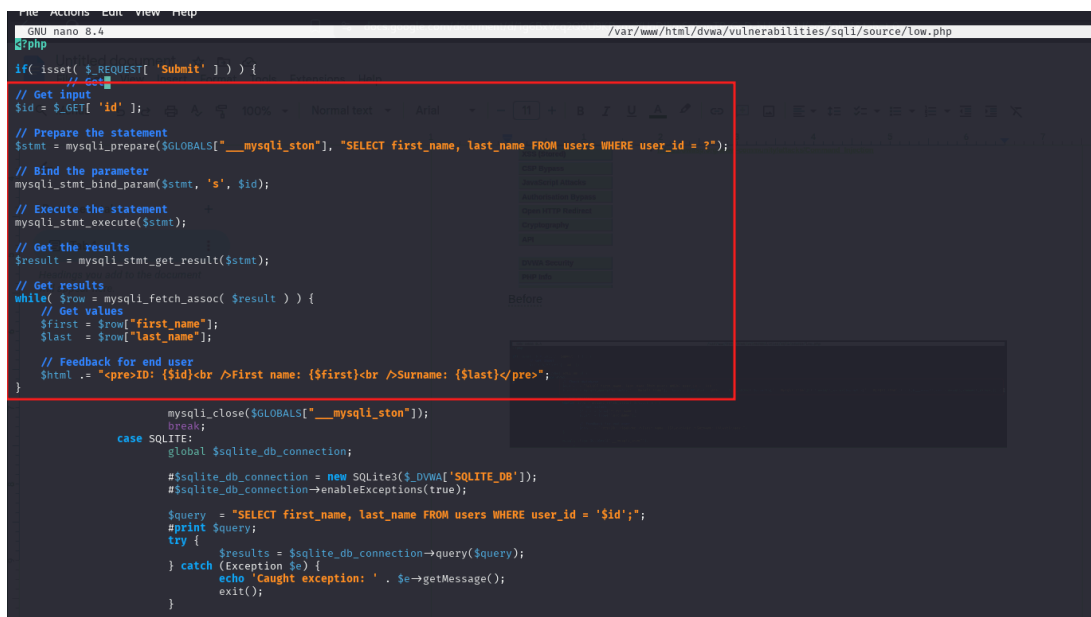
"Before" - The original vulnerable code in ..



```
GNU nano 8.4 /var/www/html/dvwa/vulnerabilities/sql/source/low.php
<?php
if( isset( $_REQUEST[ 'Submit' ] ) ){
    // Get input
    $id = $_REQUEST[ 'id' ];

    switch ( $_OWM[ 'SQLI_DB' ] ){
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
            $result = mysqli_query($GLOBALS[ "__mysqli_ston" ], $query ) or die( '
```

"After" - The new, secure code using prepared statements.



```
File Actions Edit View Help
GNU nano 8.4 /var/www/html/dvwa/vulnerabilities/sql/source/low.php
<?php
if( isset( $_REQUEST[ 'Submit' ] ) ){
    // Get input
    $id = $_GET[ 'id' ];

    // Prepare the statement
    $stmt = mysqli_prepare($GLOBALS[ "__mysqli_ston" ], "SELECT first_name, last_name FROM users WHERE user_id = ?");

    // Bind the parameter
    mysqli_stmt_bind_param($stmt, 's', $id);

    // Execute the statement
    mysqli_stmt_execute($stmt);

    // Get the results
    $result = mysqli_stmt_get_result($stmt);

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ){
        // Get values
        $first = $row[ "first_name" ];
        $last = $row[ "last_name" ];



        // Feedback for end user
        $html .= "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    mysqli_close($GLOBALS[ "__mysqli_ston" ]);
    break;
    case SQLITE:
        global $sqlite_db_connection;

        $sqlite_db_connection = new SQLite3($DVWA[ 'SQLITE_DB' ]);
        $sqlite_db_connection->enableExceptions(true);

        $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
        #print $query;
        try {
            $results = $sqlite_db_connection->query($query);
        } catch (Exception $e) {
            echo 'Caught exception: ' . $e->getMessage();
            exit();
        }
    }
}
```

"Verification" - The original exploit payload is submitted, but no data is returned, proving the fix is effective.



[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](#)
[Authorisation Bypass](#)
[Open HTTP Redirect](#)

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.041 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.034 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.066 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.046 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3066ms  
rtt min/avg/max/mdev = 0.034/0.046/0.066/0.011 ms
```

More Information

- <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

2. Remediation for Reflected XSS

- **Fix Implemented:** The `htmlspecialchars()` function was wrapped around the user input variable before it was echoed back to the page. This encodes special HTML characters, preventing the browser from interpreting them as executable code.
- **Evidence of Fix:**

"Before" - The original vulnerable code in

```
// Get input
```

```
$name = $_GET[ 'name' ];
```

```
// Print out the name
```

```
echo "<pre>Hello " . $name . "</pre>";
```

"After" - The new, secure code using

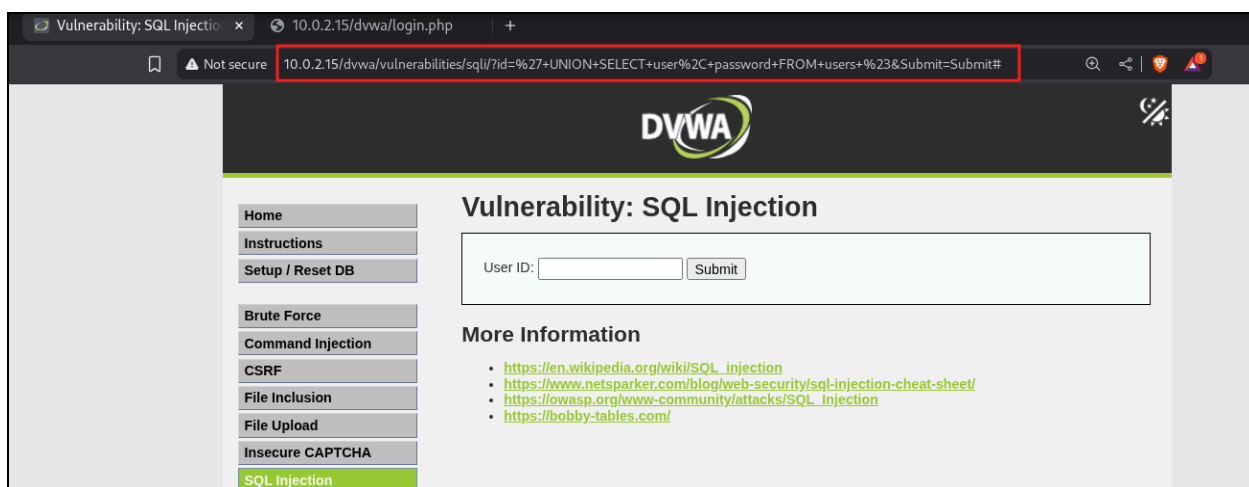
```
// Get input
```

```
$name = $_GET[ 'name' ];
```

```
// Print out the name
```

```
echo "<pre>Hello " . htmlspecialchars($name) . "</pre>";
```

"Verification" - The XSS payload is submitted, and the script is displayed as harmless text on the page instead of executing.



3. Remediation for Command Injection

- **Fix Implemented:** The user input was sanitized by implementing a blacklist that uses the `str_replace()` function to strip out dangerous shell metacharacters (`&&` and `;`) before the input is passed to the `shell_exec()` function.

- **Evidence of Fix:**

"Before" - The original vulnerable code in

```
// Get input

$target = $_REQUEST[ 'ip' ];

// Determine OS and execute the ping command.

if( striestr( php_uname( 's' ), 'Windows NT' ) ) {

    // Windows

    $cmd = shell_exec( 'ping ' . $target );

}

else {

    // *nix

    $cmd = shell_exec( 'ping -c 4 ' . $target );

}
```

"After" - The new, secure code that sanitizes the input.

```
// Get input

$target = $_REQUEST[ 'ip' ];

// Create a blacklist of dangerous characters we want to remove.

$substitutions = array(

    '&&' => "",

    ';' => "",

);

// Remove any of the blacklisted characters from the user's input.

$target = str_replace( array_keys( $substitutions ), $substitutions, $target );

// Determine OS and execute the ping command using the sanitized input.

if( striestr( php_uname( 's' ), 'Windows NT' ) ) {

    // Windows

    $cmd = shell_exec( 'ping ' . $target );
```



```

}

else {

    // *nix

    $cmd = shell_exec( 'ping -c 4 ' . $target );
}

```

"Verification" - The command injection payload is submitted, and only the output of the legitimate

Vulnerability: Command Injection

Ping a device

Enter an IP address:

```

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.043 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.036 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.032 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.038 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3073ms
rtt min/avg/max/mdev = 0.032/0.037/0.043/0.004 ms
www-data

```

More Information

- <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

Conclusion

This experiment provided a comprehensive, hands-on exploration of critical web application vulnerabilities. By successfully identifying, exploiting, and subsequently remediating flaws such as SQL Injection, Cross-Site Scripting, and Command Injection within the controlled DVWA environment, a fundamental principle of cybersecurity was repeatedly demonstrated: the failure to properly handle user-supplied input is the root cause of most web-based threats.

The practical application of fixes from implementing parameterized queries to prevent SQLi, to applying output encoding to mitigate XSS solidified the understanding that robust security is not an add-on, but a foundational aspect of the software development lifecycle. Ultimately, this exercise proves that a proactive, defense-in-depth strategy, built on the core tenets of validating input and sanitizing output, is essential to building secure and resilient web applications.