Secure Code Review Report

Role: Cybersecurity Intern

Task: Review the PHP code for security vulnerabilities and provide recommendations for secure coding practices.

# 1. Summary of Findings

The reviewed PHP script is part of a web application that interacts with both MySQL and SQLite databases to retrieve user details. The script lacks input sanitization and uses unsafe SQL query construction, which leaves it highly vulnerable to SQL Injection attacks. Error messages are also exposed directly to users, which can leak sensitive backend information.

# 2. Identified Vulnerabilities

1. 2.1 SQL Injection (Critical)

User input (`$\_REQUEST['id']`) is directly embedded into SQL queries without any validation or sanitization. This makes the application highly susceptible to SQL Injection attacks. An attacker can manipulate the input to extract, modify, or delete database records.

1. 2.2 No Input Validation or Sanitization

There are no checks on the input data to ensure it is numeric or safe for use in SQL queries. This amplifies the SQL Injection risk.

1. 2.3 Exposure of Detailed Error Messages

Detailed database error messages are displayed directly in the browser. These messages can provide attackers with valuable information about the database structure or vulnerabilities.

1. 2.4 Use of Deprecated or Global Variables

The use of global variables like `$GLOBALS["\_\_\_mysqli\_ston"]` and non-standard constants like `MYSQL` or `SQLITE` without definition or context introduces maintainability and security risks.

# 3. Recommendations for Secure Coding

1. 3.1 Use Prepared Statements

Replace dynamic SQL queries with prepared statements using parameterized queries. This completely mitigates the SQL Injection risk.

1. 3.2 Sanitize and Validate Input

Ensure all user inputs are properly validated (e.g., using `ctype\_digit()` for numeric IDs) and sanitized before use.

1. 3.3 Hide Internal Error Messages

Avoid exposing raw error messages to users. Use custom error messages and log detailed errors internally.

1. 3.4 Replace Deprecated Code Practices

Use modern database connection patterns (e.g., PDO for MySQL). Avoid using global variables and define all constants properly.

1. 3.5 Principle of Least Privilege

The database user should have limited privileges — only what is necessary for the application to function (e.g., SELECT only for read operations).

# CODE SNIPPER

<?php

if( isset( $\_REQUEST[ 'Submit' ] ) ) {

// Get input

$id = $\_REQUEST[ 'id' ];

switch ($\_DVWA['SQLI\_DB']) {

case MYSQL:

// Check database

$query = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id';";

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $query ) or die( '<pre>' . ((is\_object($GLOBALS["\_\_\_mysqli\_ston"])) ? mysqli\_error($GLOBALS["\_\_\_mysqli\_ston"]) : (($\_\_\_mysqli\_res = mysqli\_connect\_error()) ? $\_\_\_mysqli\_res : false)) . '</pre>' );

// Get results

while( $row = mysqli\_fetch\_assoc( $result ) ) {

// Get values

$first = $row["first\_name"];

$last = $row["last\_name"];

// Feedback for end user

echo "<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();

}

if ($results) {

while ($row = $results->fetchArray()) {

// Get values

$first = $row["first\_name"];

$last = $row["last\_name"];

// Feedback for end user

echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";

}

} else {

echo "Error in fetch ".$sqlite\_db->lastErrorMsg();

}

break;

}

}

?>