Secure Code Review | Report

Ruslan Amrahov

08.2.2025

Used laboratory: DVWA

Content:

1.Code analysis and vulnerability detection.

The code to be analyzed is as follows.

```
GNU nano 8.3                                                    sqli.php *
SQL Injection Source
vulnerabilities/sqli/source/medium.php
<?php

if( isset( $_POST[ 'Submit' ] ) ) {
    // Get input
    $id = $_POST[ 'id' ];

    $id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"], $id);

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
            $result = mysqli_query($GLOBALS["___mysqli_ston"], $query) or die( '<pre>' . mysqli_error($GLOBALS["___mysqli_ston"]) . '</pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Display 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>";
            }
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $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;
    }
}
```

```
// This is used later on in the index.php page
// Setting it here so we can close the database connection in here like in the rest of the source scripts
$query  = "SELECT COUNT(*) FROM users;";
$result = mysqli_query($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>' );
$number_of_rows = mysqli_fetch_row( $result )[0];

mysqli_close($GLOBALS["___mysqli_ston"]);
?>
```

First part (Getting data from the user)

The code receives the **`id`** parameter from the user via the **POST** method, then uses this value in an SQL query and displays the results on the screen.
First part (Getting data from the user):

```
if( isset( $_POST[ 'Submit' ] ) ) {
    // Get input
    $id = $_POST[ 'id' ];
```

- Here, the **`id`** value sent by the user through the form is retrieved. This value will later be used in the SQL query.
- The **`$_POST`** global variable holds the **POST** data sent by the user.

Securing the data (Attempt to secure):

```
$id = mysqli_real_escape_string($GLOBALS["___mysqli_ston"], $id);
```

- Here, the **`id`** value entered by the user is sanitized using the **`mysqli_real_escape_string()`** function. This function provides some protection against SQL injection. However, this method is weak against more advanced SQL injection attacks, as it only removes single quotes, repeated spaces, and special SQL characters. For example, attacks like **`UNION SELECT`** can bypass this function.

SQL Query Construction and Execution (SQL Query Vulnerability)

```
switch ($_DVWA['SQLI_DB']) {
   case MYSQL:
      $query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
      $result = mysqli_query($GLOBALS["___mysqli_ston"], $query) or die( '<pre>' . mysqli_error($GLOBALS["___mysqli_ston"]) . '</pre>' );
```

- Here, a **`switch`** is used to determine which database to use based on the **`$_DVWA['SQLI_DB']`** value.
  When **`MYSQL`** is selected, the SQL query is constructed as follows:
- **SELECT first_name, last_name FROM users WHERE user_id = $id;**

The `id` value is directly inserted into the SQL query. This is vulnerable to SQL injection. An attacker can input a value like **`1 OR 1=1`** for the `id`, potentially allowing them to retrieve data of all users.

Execution of Data and Display

```
while( $row = mysqli_fetch_assoc( $result ) ) {

   // Display 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>";

}
```

- The **`mysqli_fetch_assoc()`** function retrieves the query results and displays them on the screen. These results are shown to the user.
- Additionally, the query results are presented to the user with the **ID**, first name, and surname.

SQLite DB Usage

The code also includes a transition to using the SQLite database:

```
case SQLITE:
    global $sqlite_db_connection;


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

- Here, the SQL query is executed in the **SQLite** database. The SQL injection vulnerability is still present, as the **`id`** value is directly inserted into the query. This leaves the application open to SQL injection attacks.

Closing the Database Connection

```
$query  = "SELECT COUNT(*) FROM users;";

$result = mysqli_query($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>' );
```

$number_of_rows = mysqli_fetch_row($result)[0];

mysqli_close($GLOBALS["___mysqli_ston"]);

- In this section, a query is executed to retrieve the number of users in the database. If the query execution is successful, the number of users in the table is displayed. The connection is then closed using **`mysqli_close()`**.

2.Security Vulnerabilities and Recommendations

SQL Injection

The main security vulnerability is SQL injection. User-inputted data is directly inserted into queries, which allows various SQL injection attacks. For example, an attacker could add 1 OR 1=1 to the id value, potentially retrieving all data.

- **Recommendations:**Prepared Statements can be used: The best defense against SQL injection is using prepared statements. This allows SQL queries to work with parameters and prevents user input from being treated as executable code in the SQL query.

$query = "SELECT first_name, last_name FROM users WHERE user_id = ?";

$stmt = mysqli_prepare($GLOBALS["___mysqli_ston"], $query);

mysqli_stmt_bind_param($stmt, 'i', $id); // 'i' denotes integer type

mysqli_stmt_execute($stmt);

$result = mysqli_stmt_get_result($stmt);

- **Data Validation:**It is important to ensure that the data entered by the user is in the expected format. The `id` should only be numeric, and this can be checked using the `preg_match()` function:

```
if (!preg_match('/^\d+$/', $id)) {
   die('Invalid ID format');
}
```

Exposing Database Errors to Users

In the code, errors occurring after executing database queries are displayed directly to the user:

```
or die( '<pre>' . mysqli_error($GLOBALS["___mysqli_ston"]) . '</pre>' );
```

This provides attackers with information about the database and system, increasing their chances of gaining unauthorized access. Error details should not be shown to the user.

Recommendations:
Do not display error messages to the user. Instead, use a simple message: **die('An error occurred. Please try again);** later.

- Additionally, error details can be logged in a server log file.

Secure Data Validation

In some parts of the code, the **`mysqli_real_escape_string()`** function is used to secure data, but this method only sanitizes certain SQL characters. It does not provide complete protection against SQL injection. Attackers might try more sophisticated methods to bypass this security.

Recommendations:

- One of the best protections against SQL injection is the use of **`prepared statements`**. This approach provides more comprehensive security.
- **`mysqli_real_escape_string()`** only reduces certain threats and is weak against more complex SQL injection attacks.

Data Validation and Integrity

Validating user input is important. Here, only the `id` parameter is used for SQL query execution, but other dangerous variables may also be present.

Recommendations:

- Perform data validation: Each parameter should be validated before being used. Only numeric values should be accepted for `id`.
- Sanitize input: Clean all user inputs and replace dangerous characters appropriately.

Data Leakage by Attackers

The `echo` functions in the code display all data to the user. This could lead to security issues because users may be able to view sensitive information.

Recommendations:

- Do not display sensitive data through the screen: To ensure data security, the user should only see their own data, and no insecure information related to the database should be shown on the screen.

User Input and Sensitive Data Handling

The code shows that user data is inserted into the database without proper validation. This can also lead to the corruption of user data.

Recommendations:

- Store passwords and sensitive information securely: Passwords should never be stored in plain text in the database. They should be encrypted properly.
- Properly manage access rights: It is important to control user or system access to sensitive information.