SAST Penetration Testing Report

For Intern Intelligence

Project: LifestyleStore Web Application
Version: 1.0
Date: 15th March 2025
Author: AMAN GUPTA

Methodology

The review was conducted manually by:

- **Reviewing Key PHP Files:** Examining the source code for database queries, user input handling, file inclusion, and session management.
- **Identifying Vulnerabilities:** Matching code practices against the OWASP Top 10 vulnerabilities.
- **Code Comparison:** Documenting both the vulnerable (old) code and the secure (fixed) code along with explanations.
- **Risk Rating:** Assigning severity based on the potential impact and likelihood of exploitation.

1. SQL Injection

What it is:
SQL Injection happens when user input is inserted directly into SQL queries. An attacker can manipulate this input to change the query, possibly stealing or destroying data.

Affected Files and Code Areas:

a. connection.php

- **Problem Area:**

```
$con = mysqli_connect("localhost","username","password","database") or die(mysqli_error($con));
```

- **Logic:**
  This line connects to the database. Although the connection itself isn't doing injection, later code uses this connection unsafely.
- **Fix:** Use PDO and prepared statements:

Old Code:

```
php CopyEdit
<?php
$con = mysqli_connect("localhost","username","password","database") or die(mysqli_error($con)); ?>
```

Fixed Code:

```
php CopyEdit
<?php
try {
   $con = new PDO("mysql:host=localhost;dbname=database", "username", "password");
   $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
   error_log("Database Connection Error: " . $e->getMessage());
die("Database connection error.");
}
?>
```

Explanation:
The fixed code uses PDO, which supports prepared statements. This change prevents attackers from injecting malicious SQL.

b. **cart_add.php**

- **Problem Area (approx. line 4–8):**

Old Code:

```
php CopyEdit
$item_id = $_GET['id'];
$user_id = $_SESSION['id'];
$add_to_cart_query = "insert into users_items(user_id, item_id, status) values ('$user_id','$item_id','Added
to cart')";
$add_to_cart_result = mysqli_query($con, $add_to_cart_query) or die(mysqli_error($con));
```

- **Logic:** The code takes an id from the URL and directly inserts it into the SQL query without checking if it’s a valid number. This can allow SQL injection.
- **Fix:** Validate the input and use prepared statements.

Fixed Code:

```
php CopyEdit
<?php
require 'connection.php'; session_start();

if (!isset($_GET['id']) || !is_numeric($_GET['id'])) {
die("Invalid item ID");
}

$item_id = intval($_GET['id']);
$user_id = $_SESSION['id'];

$stmt = $con->prepare("INSERT INTO users_items (user_id, item_id, status) VALUES (?, ?, 'Added to
cart')");
$stmt->execute([$user_id, $item_id]);

header('location: products.php');
?>
```

Explanation:

- The input is checked to ensure it’s numeric.
- intval() converts it to an integer. ○ A prepared statement is used so that even if an attacker tries to inject SQL code, it will be treated as a plain value.

c. cart.php & cart_remove.php

- **Problem Area:**
  In both files, variables such as $user_id and $item_id are directly concatenated into SQL queries.
- **Logic:**
  For example, in cart_remove.php:

```
php CopyEdit
$delete_query = "delete from users_items where user_id='$user_id' and item_id='$item_id'";
```

This line lets an attacker modify the id parameter to delete items not belonging to them.

- **Fix:** Validate and use prepared statements.

Old Code:

```
php CopyEdit
<?php
$item_id = $_GET['id'];
$user_id = $_SESSION['id'];
$delete_query = "delete from users_items where user_id='$user_id' and item_id='$item_id'";
$delete_query_result = mysqli_query($con, $delete_query) or die(mysqli_error($con)); ?>
```

Fixed Code:

```
php CopyEdit
<?php
require 'connection.php'; session_start();

if (!isset($_GET['id']) || !is_numeric($_GET['id'])) {
die("Invalid request");
}

$item_id = intval($_GET['id']);
$user_id = $_SESSION['id'];

$stmt = $con->prepare("DELETE FROM users_items WHERE user_id = ? AND item_id = ?"); $stmt-
>execute([$user_id, $item_id]);

header('location: cart.php');
?>
```

Explanation:

This fix ensures that only numeric IDs are accepted and that the deletion is performed securely using a prepared statement.

2. Cross-Site Scripting (XSS)

What it is:
XSS occurs when an attacker can inject malicious scripts into webpages viewed by other users. This typically happens when output from the database (or user input) isn't properly sanitized.

Affected File: cart.php

- **Problem Area (approx. line where product names are displayed):**

php CopyEdit
```
<th><?php echo $row['name'] ?></th>
```

- **Logic:**
  This line outputs the product name directly. If the product name contains JavaScript, it could be executed by a user's browser.
- **Fix:** Escape output using htmlspecialchars().

Old Code:

php CopyEdit
```
<th><?php echo $row['name'] ?></th>
```

Fixed Code:

php CopyEdit

```
<th><?php echo htmlspecialchars($row['name'], ENT_QUOTES, 'UTF-8'); ?></th>
```

Explanation: htmlspecialchars() converts special characters (like < and >) into HTML entities, so any script code will not run.

3. Cross-Site Request Forgery (CSRF)

What it is:
CSRF tricks a logged-in user into unknowingly performing actions (like adding or removing items) on your site. This happens because state-changing requests (like adding to cart) are done via GET requests without a verification token.

Affected Files: cart_add.php & cart_remove.php

- **Problem Area:**
  Operations are performed simply by visiting a URL (e.g., cart_add.php?id=...), which can be exploited.
- **Logic:**
  There’s no check to ensure the request came from your site.
- **Fix:**
  - **Switch to POST requests:** Change the method of form submissions so that state-changing actions use POST.
  - **Implement CSRF Tokens:** Generate a token when the session starts and include it in forms.

Old Code:
The current code uses GET parameters:

```
php CopyEdit
$item_id = $_GET['id'];
```

Fixed Code (example for cart_add.php):

```
php CopyEdit
<?php
session_start();

// Generate CSRF token if not available if
(empty($_SESSION['csrf_token'])) {
   $_SESSION['csrf_token'] = bin2hex(random_bytes(32)); }

// When processing the form:
```

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!isset($_POST['csrf_token']) || $_POST['csrf_token'] !== $_SESSION['csrf_token']) {
die("CSRF token validation failed.");
    }

    if (!isset($_POST['id']) || !is_numeric($_POST['id'])) {
        die("Invalid item ID");
    }

    $item_id = intval($_POST['id']);
    $user_id = $_SESSION['id'];

    // Use prepared statements as shown previously...
    $stmt = $con->prepare("INSERT INTO users_items (user_id, item_id, status) VALUES (?, ?, 'Added to
cart')");
    $stmt->execute([$user_id, $item_id]);

    header('location: products.php');
}
?>
<!-- In the HTML form -->
<form method='POST' action='cart_add.php'>
    <input type='hidden' name='csrf_token' value='<?php echo $_SESSION["csrf_token"]; ?>'>
    <input type='hidden' name='id' value='<!-- item id here -->'>
    <input type='submit' value='Add to Cart'>
</form>
```

Explanation:

- The CSRF token ensures that the form submission came from your site.
- Changing from GET to POST means that actions aren't triggered just by visiting a URL.

4. Insecure File Inclusion

What it is:
Insecure file inclusion can allow attackers to load remote or unintended files if the file paths are not properly controlled.

Affected File: index.php (and similar includes)

- **Problem Area:**

```
php
CopyEdit
require 'header.php';
```

- **Logic:**
  This line includes a file. If the path is not fixed, an attacker might manipulate it to include another file.
- **Fix:** Use an absolute path and verify that the file exists.

Old Code:

```
php CopyEdit
<?php require
'header.php';
?>
```

Fixed Code:

```
php CopyEdit
<?php
$header_file = __DIR__ . '/header.php';
if (file_exists($header_file)) {
require $header_file;
} else {
   die("Error: File not found.");
}
?>
```

Explanation:
The fixed code uses __DIR__ (the current directory) to create an absolute path and checks if the file exists before including it.

5. Missing HTTP Security Headers

What it is:
Security headers tell the browser how to behave when handling your site. Missing headers leave the site open to attacks like clickjacking and MIME sniffing.

Affected Files:

All PHP files (when a response is sent to the browser).

- **Problem Area:**
  No HTTP headers are being set.
- **Logic:**
  Without these headers, attackers can more easily manipulate the browser's behavior.
- **Fix:** Add the following headers at the very beginning of each PHP file (or in a central configuration file):

Old Code:
(No security headers are present.)

Fixed Code:

```
php CopyEdit
<?php
header("X-Frame-Options: DENY"); // Prevent clickjacking header("X-Content-Type-Options:
nosniff"); // Prevent MIME type sniffing header("Referrer-Policy: no-referrer-when-
downgrade"); // Control referrer data
header("Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'"); // Restrict resource
loading
?>
```

Explanation:
These headers help the browser protect your site from various common attacks.

6. Error Handling and Information Disclosure

What it is:
Using statements like or die(mysqli_error($con)) shows detailed database errors to the user. This can reveal sensitive information about your system.

Affected Files:

Any file using error handling in queries (e.g., cart_add.php, cart_remove.php, etc.).

- **Problem Area:**

```
php CopyEdit
$result = mysqli_query($con, $query) or die(mysqli_error($con));
```

- **Logic:**
  This will display the actual database error, including query details, to the user.
- **Fix:**
  Log the error internally and show a generic error message.

Old Code:

```
php CopyEdit
$result = mysqli_query($con, $query) or die(mysqli_error($con));
```

Fixed Code:

```
php CopyEdit
$result = mysqli_query($con, $query); if
(!$result) {
    error_log("Database query error: " . mysqli_error($con));
die("An internal error occurred. Please try again later."); }
```

Explanation:
This fix ensures that detailed errors are logged for developers, while users only see a generic message.

7. Session Management

What it is:
Improper session management can allow attackers to hijack user sessions.

Affected Files:

All files that call session_start() (e.g., index.php, cart_add.php, cart.php, etc.).

- **Problem Area:**

```
php CopyEdit
<?php
session_start();
?>
```

- **Logic:**
  Simply starting a session is not enough. You need to protect session cookies and regenerate session IDs.

- **Fix:**

php CopyEdit
<?php
session_start();
session_regenerate_id(true); // Prevent session fixation attacks ini_set('session.cookie_httponly', 1); // Disallow JavaScript access to session cookie ini_set('session.cookie_secure', 1); // Ensure cookies are sent over HTTPS only
ini_set('session.use_only_cookies', 1); // Force sessions to use cookies, not URL parameters ?>

Explanation:
These settings secure the session against various attacks, including session hijacking and fixation.

Summary

Vulnerability	File (Example)	Old Code (Vulnerable)	Fixed Code (Secure)	Severity & Logic
SQL Injection	cart_add.php	Directly inserting $_GET['id'] into SQL query (lines ~4-8)	Validate with intval() and use prepared statements	**Critical** – Unchecked input can change database queries.
XSS	cart.php	Echoing product names without escaping (line with <th><?php echo $row['name'] ?></th>)	Use htmlspecialchars($row['name'], ENT_QUOTES, 'UTF-8')	**Medium** – Unescaped output can execute unwanted scripts.
CSRF	cart_add.php	GET request used for actions; no CSRF token	Use POST requests and validate a CSRF token	**High** – Unauthorized commands can be triggered via crafted requests.
Insecure File Inclusion	index.php	require 'header.php'; without path check	Use absolute paths and file_exists() check	**Medium** – Attackers may include unintended files.
Missing HTTP Headers	All PHP files	No security headers sent	Set headers like X-Frame-Options, X-Content-TypeOptions, etc.	**Medium** – Browser behaviors can be exploited.
Error Disclosure	Various (e.g., cart_add.php)	or die(mysqli_error($con)) exposes details	Log errors internally; show generic error messages	**Medium** – Reveals system details to attackers.

Session Management	All files with session_ start()	Just session_start();	Regenerate session IDs and set secure cookie options	**Medium** – Poor session handling can lead to hijacking.

8. Conclusion and Recommendations

The manual static analysis of the LifestyleStore project has identified several vulnerabilities that align with the OWASP Top 10. The most critical issues—SQL Injection and CSRF—must be addressed immediately by implementing prepared statements, proper input validation, and secure form handling. Other improvements include output escaping, secure file inclusion practices, proper error handling, adding security headers, and enhanced session management.

Recommendations:

- **Implement Secure Coding Practices:** Update all vulnerable code as detailed above.
- **Conduct Regular Security Reviews:** Perform periodic SAST and DAST to ensure new vulnerabilities are not introduced.
- **Adopt a Defense-in-Depth Strategy:** Use secure configurations, firewalls, and monitoring systems.
- **Training and Awareness:** Ensure developers understand secure coding techniques.

9.Refrence

Project link: https://github.com/sajalagrawal/LifestyleStore