FashionStore Security Audit Report

Project: FashionStore – Clothing Shop

Reviewer: Yuval Bailey Soloveichik

Course: Cyber2 – Final Project (Summary Report)

Repository: https://github.com/BaileySol/cyber2---final-project

# Introduction

This report provides a detailed security audit of the FashionStore web project. The goal is to identify vulnerabilities, explain their risks in professional security terms, and provide both insecure (vulnerable) and secure (fixed) code examples. This will help demonstrate common web application security flaws such as XSS, CSRF, improper input validation, and insecure password handling, and how to mitigate them according to cybersecurity best practices.

## Issue 1: Plaintext Password Handling

Observation: Passwords are captured and potentially transmitted in plaintext. This exposes credentials to interception and attacks.

Risk: Credentials can be stolen via Man-in-the-Middle attacks, XSS, or by accessing client-side storage. This may lead to account compromise.

Recommendation: Always transmit passwords over HTTPS. Store them only server-side as salted hashes (bcrypt/argon2). Never store passwords in client-side storage.

❌ Insecure Code Example:

<input type="password" id="password" placeholder="Password">  
  
// BAD: storing password in localStorage  
localStorage.setItem("password", password);

✅ Secure Code Example:

// GOOD: send password over HTTPS, hashing server-side  
fetch("/login", {  
 method: "POST",  
 body: JSON.stringify({ username, password }),  
 headers: { "Content-Type": "application/json" }  
});

## Issue 2: Client-Side Storage of Sensitive Data

Observation: Sensitive information such as usernames and passwords stored in localStorage.

Risk: localStorage is accessible by JavaScript and browser tools. Attackers exploiting XSS can steal credentials or tokens.

Recommendation: Never store passwords client-side. Store only temporary session tokens inside HttpOnly, Secure cookies.

❌ Insecure Code Example:

localStorage.setItem("user", username);  
localStorage.setItem("password", password);

✅ Secure Code Example:

// GOOD: store only session token (example)  
localStorage.setItem("sessionToken", sessionToken);

## Issue 3: Lack of Input Validation / XSS

Observation: User input is directly injected into the DOM without sanitization.

Risk: Enables Cross-Site Scripting (XSS), where attackers can inject malicious scripts to steal cookies, hijack sessions, or deface the site.

Recommendation: Sanitize and escape all inputs before inserting into DOM. Use textContent instead of innerHTML, or libraries like DOMPurify.

❌ Insecure Code Example:

messageList.innerHTML += `<li>${userMessage}</li>`;

✅ Secure Code Example:

const li = document.createElement("li");  
li.textContent = userMessage;  
messageList.appendChild(li);

## Issue 4: Client-Side Cart Manipulation

Observation: Prices and totals are computed on the client side, trusting user input.

Risk: Attackers can manipulate cart data, product prices, or quantities via DevTools. This can lead to financial fraud.

Recommendation: Perform all pricing and inventory calculations on the server. Treat client-submitted cart data as untrusted.

❌ Insecure Code Example:

function addToCart(name, price) {  
 cartTotal += price; // attacker can change "price" in console  
}

✅ Secure Code Example:

function addToCart(name, price, quantity) {  
 const total = price \* quantity;  
 document.getElementById("totalAmount").textContent = `Total: $${total}`;  
 // Server must validate correct price at checkout  
}

## Issue 5: No Rate Limiting

Observation: Forms (login, order, contact) accept unlimited submissions without restrictions.

Risk: Enables brute-force attacks, spamming, and denial-of-service (DoS).

Recommendation: Implement server-side rate limiting (per IP/account) and use CAPTCHA for login and order forms.

❌ Insecure Code Example:

<form id="loginForm">  
 <input type="text" id="username">  
 <input type="password" id="password">  
 <button type="submit">Login</button>  
</form>

✅ Secure Code Example:

let lastSubmit = 0;  
document.getElementById("orderForm").addEventListener("submit", e => {  
 const now = Date.now();  
 if (now - lastSubmit < 10000) {  
 e.preventDefault();  
 alert("Please wait before submitting again.");  
 } else {  
 lastSubmit = now;  
 }  
});

## Issue 6: Lack of CSRF Protection

Observation: Forms and requests lack anti-CSRF tokens.

Risk: Allows attackers to forge malicious requests (orders, account updates) on behalf of victims.

Recommendation: Use anti-CSRF tokens generated server-side and included with each form or request.

❌ Insecure Code Example:

<form id="orderForm">  
 <input type="text" name="product">  
 <button type="submit">Submit Order</button>  
</form>

✅ Secure Code Example:

fetch("/submitOrder", {  
 method: "POST",  
 headers: {  
 "Content-Type": "application/json",  
 "X-CSRF-Token": csrfToken  
 },  
 body: JSON.stringify(orderData)  
});

## Issue 7: No Server-Side Validation

Observation: All validation is done client-side, making it bypassable.

Risk: Attackers can send malformed or malicious data to the server directly.

Recommendation: Always validate and sanitize data server-side, even if client-side validation exists.

❌ Insecure Code Example:

const email = document.getElementById("orderEmail").value;  
// Only checked on client  
if (!email.includes("@")) {  
 alert("Invalid email");  
}

✅ Secure Code Example:

// Client check for UX only, must be validated on server  
const email = document.getElementById("orderEmail").value;  
if (!/^[^@]+@[^@]+\.[^@]+$/.test(email)) {  
 alert("Please enter a valid email.");  
}

## Issue 8: Missing Security Headers

Observation: No Content Security Policy (CSP), X-Frame-Options, or secure cookie attributes are configured.

Risk: Enables clickjacking, XSS, and session hijacking attacks.

Recommendation: Set proper HTTP security headers server-side: CSP, Strict-Transport-Security, HttpOnly cookies, SameSite flags.

❌ Insecure Code Example:

<!-- No CSP or X-Frame-Options headers -->

✅ Secure Code Example:

<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">

## Issue 9: File & Resource Exposure

Observation: Project files (images, scripts) are directly accessible without restrictions.

Risk: Attackers can enumerate files, discover sensitive assets, and exploit them.

Recommendation: Restrict public file access and implement proper access control.

❌ Insecure Code Example:

src="js/admin-config.js" <!-- exposed to public -->

✅ Secure Code Example:

// GOOD: restrict sensitive files on server (e.g., via .htaccess or nginx rules)

# Conclusion

The FashionStore project contains multiple common security vulnerabilities including plaintext password handling, client-side storage of sensitive data, lack of input sanitization (XSS), missing CSRF tokens, and reliance on client-side validation. These weaknesses expose the application to severe threats such as account compromise, data tampering, and financial fraud. By applying the recommended fixes and following best practices, the project can achieve a significantly higher security posture.