SECURITY FUNDAMENTALS AND ASSESSMENT REPORT

Toan, Sally, Marko

UNSECURE PWA Security Report

Document Details

Due Date	9/4/2025
Title	Progressive Web App (Unsecure PWA)
Purpose	Students will receive a Progressive Web Application's source code, which they must check out using the GitHub version control program and install it in the Visual Code Integrated Development Environment (IDE). Students must check out the source code using Visual Code Integrated Development Environment. Students are then required to complete the four (4) appendices attached to the booklet, which will guide them through the project's completion and break it into three (3) components as per the Programming for the Web Module.
Version	1.0

LOGBOOK

Version	**Date of completion**	**Description**
	25/2/2025 (date start)	Documentation Start
0.1	4/4/2025	Bcrypt Hashing Password
0.2	5/4/2025	CSRF protection
0.3	5/4/2025	Forwarding Redirect
0.4	5/4/2025	Bug Fixed
0.5	6/4/2025	Content Security Policy

Content

Table of Content

1. Executive Summary

1.1 Purpose of report

This report outlines the security evaluation and progressive enhancement of an intentionally vulnerable Progressive Web Application (PWA) provided to students as part of the Programming for the Web module. Through iterative updates, various security features have been implemented and tested. These include password hashing, CSRF protection, redirect sanitization, and a Content Security Policy (CSP). The report highlights key aspects of secure software development and presents strategies to manage application risks.

2.2 Highlight key findings and recommendations regarding security practices in software development.

Key Findings

1. **Lack of Secure Password Storage**
   The original code stored user passwords in plaintext, creating a high risk of credential leaks and identity theft in case of a data breach.

2. **Absence of CSRF Protection**
   Forms in the application were vulnerable to Cross-Site Request Forgery (CSRF), allowing malicious sites to trick users into making unintended requests.

3. **Insecure Redirect Mechanism**
   The application used unvalidated query parameters for redirects (?url=), making it susceptible to open redirect attacks and phishing exploits.

4. **Missing Input Validation and Sanitization**
   User input was not properly validated or sanitized, exposing the application to injection-based attacks such as SQL Injection and Cross-Site Scripting (XSS).

5. **No Content Security Policy (CSP)**
   The absence of a CSP allowed inline scripts and external resources to run freely, increasing the surface for XSS attacks.

6. **Minimal Session Handling Controls**
   Sessions lacked timeout controls, increasing the risk of session hijacking in shared or public environments.

2. Introduction

2.1 Overview of “The Unsecure PWA”

“The Unsecure PWA” is a deliberately vulnerable Progressive Web Application designed as an educational tool to help students identify, exploit, and fix common security flaws found in real-world web applications. The application simulates a basic user feedback platform with functionality for signing up, logging in, and submitting feedback.

Initially, the PWA lacked critical security features, including secure password storage, input validation, CSRF protection, and output sanitization. It also included intentionally misconfigured behaviors, such as open redirects and permissive Content Security Policies, which exposed it to common web-based attacks like SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).

As part of the module project, students cloned the PWA source code via GitHub and deployed it in the Visual Studio Code Integrated Development Environment (IDE). They then followed guided appendices to progressively identify security risks and implement protections in stages, using best practices in secure software development.

The final secured version of the PWA includes improvements such as:

- bcrypt password hashing
- CSRF token implementation
- removal of unsafe redirect parameters
- strict Content Security Policy (CSP)
- input validation and safe output encoding

2.2 Importance of secure software architecture in today's digital world

In today's hyper-connected and increasingly digital world, secure software architecture is more critical than ever. With cyber threats evolving in complexity and frequency, a weak or poorly designed software structure can expose sensitive data, damage business reputation, and even endanger user safety.

Secure software architecture ensures that security is not an afterthought but a foundational element of system design. It incorporates safeguards such as authentication, data encryption, access control, and secure data flow into the core structure of an application—making it resilient against known and emerging attack vectors.

Definition	**Characteristics**	**Example**
Data protection	Securing personal and business data	Encryption, backup systems
Cyber attacks	Malicious attempts to damage systems	Phishing, ransomware
Static application security testing (SAST)	Static analysis of code before execution	SonarQube, Bandit
Dynamic application security testing (DAST)	Testing the running application for issues	OWASP ZAP
Vulnerability assessment	Systematic identification of flaws	Nessus, Burp Suite
Penetration test	Simulated attack to find vulnerabilities	Ethical hacking
API Security	Protecting APIs from unauthorised access	Rate limiting, token checks
Cross site scripting	Injecting scripts into web apps	
Cross site request forgery (CSRF)	Exploiting user sessions for unauthorized actions	Forged form submissions

3. Benefits of Developing Secure Software

3.1 Data protection

3.1.1 Significance of protecting sensitive data.

Issue	Explanation
Confidentiality	Ensures only authorized access to sensitive information
Integrity	Prevents unauthorized changes to data
Availability	Ensures systems are up and running
Compliance	Meets standards like GDPR, HIPAA
Threat Protection	Prevents data breaches
Trust	Builds customer confidence
Business continuity	Ensures operations continue despite threats

3.1.2 Strategies for data encryption and Storage

- Hashing passwords with bcrypt
- Using HTTPS to encrypt data in transit
- Limiting stored data to only essential user info
- Regular database backups and access controls

3.2 Minimising Cyber Attacks and Vulnerabilities

3.2.1 Benefits

Issue	Explanation
Prevent data breaches	Stops sensitive leaks
Mitigate threats	Early prevention of attacks
Protect trust	Safeguards reputation
Comply with regulations	Avoids legal issues
Business continuity	Keeps services reliable
Cost savings	Avoids breach-related expenses
Competitive edge	Demonstrates responsibility

Preventative Measures:

- Input validation and sanitisation
- Secure authentication mechanisms
- CSRF token implementation
- Role-based access control

4. Fundamental software developer steps to develop security code

---

4.1 Requirements Definitions

Requirement	Description
Objective	Build a secure feedback system
Tools	Flask, SQLite, HTML/CSS/JS
Functional	Login, submit/view feedback
Non-Functional	Secure, responsive, user-friendly
Testing	Manual testing, simulated attacks
Documentation	Inline code docs, final report
Timeline	6 weeks of iterative dev and testing

4.2 Determining Specifications

Specifications	Description
Login	Secure login form with bcrypt
Password hashing	bcrypt with salt
Input sanitation	Regex filters and escaping
2FA	(Optional) via email/token
Penetration testing	Used Burp Suite, OWASP ZAP
Docs	GitHub README and internal docs
Tools	Flask-WTF, bcrypt, SQLite, pytest
Milestones	Weekly sprints with progress logs

Scalability	Simple but extendable architecture
Compliance	Followed OWASP Top 10 guidelines

4.3 Design

- Secure session handling using cookies
- Clear separation of concerns (MVC-like structure)
- Use of templates with {{ safe }} to prevent XSS

4.4 Development

Phases	**Description**
Requirements	Interview client (fictional)
Design	Secure architecture principles
Development	Python/Flask secure code
Testing	Manual input testing
Security validation	XSS, CSRF, SQLi simulations
Deployment	Localhost environment
Documentation	Markdown-based guidebook

4.5 Integration

Step	Description
System assessment	Identify insecure practices
Alignment	Security-first updates
Modular Dev	Reusable Flask routes
API integration	Secure endpoints added
DB integration	SQLAlchemy for queries
UI Integration	Input validation via JS
Testing	Manual + tool-based
Monitoring	Print logs for dev debug

4.6 Testing and Debugging

- CSRF and SQL injection tests
- Linting try/except error catching
- Unit tests for logic and edge cases

Installation

- Use virtual environment
- Install via pip install -r requirement.txt
- Enforce secure defaults (e.g HTTPS only)

Maintenance

- Weekly security review
- Apply updates and patches
- Monitor and feedback

5. Influence of end users on secure design features

5.1 User capabilities and Experience

Issue	Explanation
Friendly Auth	Simple login UX
RBAC	Limit access per role
Awareness	Show tooltips and help
Feedback	Test with real users
Simplified Controls	Minimal UI clutter

5.2 Importance of User Feedback

User feedback helped redesign form validation UX, error messaging, and clarified security indicators (e.g., locked icons for protected content).

6. Developing Secure code

Concept	Explanation
Confidentiality	Encrypt and limit access
Integrity	Prevent unauthorized data changes
Availability	Ensure app doesn’t crash
Authentication	Verified logins
Authorisation	Role-checks before actions
Accountability	Log actions with usernames

7. Security features incorporated into software

Feature	Application
Data protection	bcrypt, HTTPS, input sanitisation
Security measures	CSRF tokens, route checking
Privacy	Minimal user data stored
Compliance	OWASP-aligned dev
Auth and Authorisation	bcrypt + sessions, role-based actions

8. Security by design approach

8.1 Cryptography and Sandboxing

- **Cryptography** secures login data using bcrypt.
- **Sandboxing** limits form input via sanitisation and server-side checks.

8.2 Privacy by Design

- Collect minimal data
- Opt-in policies
- Transparent privacy messages on UI

9. Testing and Evaluating security and resilience

Method	Description
Hardening	Disabled debug mode, set secure cookies
Handling breaches	Error pages, try/except blocks
Disaster recovery	Backup DB scripts, error logging

10. Strategies to managing the security of programming code

Strategy	Explanation
Code Review	Peer-reviewed via GitHub
SAST	Used Bandit for Python linting
DAST	OWASP ZAP to simulate attacks
Vul assessment	Manual input checks
Pen testing	Performed with Burp Suite/ZAP

11. Defensive data input handling

- Used escaping to prevent XSS
- Limited input length
- Checked for script tags

12. Safe API development

- APIs require session authentication
- JSON responses validated on both server and client
- Tokens and session validation prevent misuse

13. Efficient execution for the user

Strategy	Explanation
Memory mgmt	Avoided memory leaks with short-lived sessions
Session mgmt	Secure cookies, expiry time set
Exception mgmt	try/except + error logging

14. Secure code for user action controls

Vulnerability	Code Sample
Broken auth	bcrypt.check_password_hash()
XSS	escape(request.form['input'])
CSRF	{{ csrf_token() }}
Invalid redirect	Avoided `?next=` logic
Race Conditions	Locked database write ops

15. Protective against file attacks

- Validates file names
- Stores outside static dir
- Rejects large/malicious files

16. Conclusion

This report demonstrates the progressive hardening of a vulnerable PWA. By applying secure coding practices, validating input, and testing for known attack vectors, the group was able to significantly improve the resilience of the application. Security is an ongoing process, and future work may involve session management improvements and implementing Two-Factor Authentication.

17. Reference

Student Resources from PWA Github
W3School
OWASP Top 10

18. Security report appendix