College of Computer and
Information Sciences
Computer Science Department

Building a Secure Web Application

Detection and Mitigation of Security Vulnerabilities

CSC429 – Project

Prepared by:

Name
Rimas Albahli
Maymona Alotaibi
Lama Alotibie
Atheer alasiri

Table of Contents

1. Introduction

Our project is a **secure web application** that implements a **basic user management system** with features such as user registration, login, dashboard access, and admin-only content via role-based access control. The main objective is to demonstrate an understanding of **common web vulnerabilities** and how to mitigate them using secure coding practices.

To achieve this, we developed **two versions** of the application:

- A **vulnerable version**, where common security flaws were deliberately introduced.
- A **secure version**, where those vulnerabilities were identified and properly mitigated.

Features Implemented:

- **User Registration**: Users can sign up with a username and password.
- **Login System**: Secure authentication for registered users.
- **Dashboard**: Personalized user dashboard after login.
- **Admin Page**: Accessible only to users with the admin role.

Technologies Used:

- **Frontend**: HTML, CSS
- **Backend:** Python (Flask framework)
- **Database:** In-memory dictionary
- **Security Tools:** Werkzeug's generate_password_hash() and check_password_hash() for password hashing, Flask sessions for authentication, HTTPS for encrypted communication

Purpose of the Project

The goal of this project is to design and implement **a secure web application** while gaining hands-on experience with common web vulnerabilities and their mitigations. To achieve this, we first developed a **deliberately vulnerable version** of the application, allowing us to simulate real-world attacks such as SQL injection, weak password storage, cross-site scripting (XSS), improper access control, and unencrypted data transmission. We then applied **secure coding practices** to detect and fix these issues, reinforcing our understanding of how to build more secure web systems.

2. Vulnerabilities and Mitigation

2.1 SQL Injection

- **Vulnerability:** SQL injection allows attackers to manipulate SQL queries by injecting harmful input.
- **Intentional Vulnerability:** In the vulnerable version, user input was directly embedded into a raw SQL query using an f-string.
- **Mitigation:** In the secure version, instead of using raw SQL queries, user credentials are checked safely using a Python dictionary and secure password checking**.**
- **How it works :** Avoiding raw SQL altogether or using parameterized queries prevents attackers from injecting harmful SQL commands. Here, input is never parsed as code, eliminating injection risk.

2.2 Weak Password Storage

- **Vulnerability:** Storing user passwords in plain text means that anyone who accesses the database can see and misuse those passwords.
- **Intentional Vulnerability :** In the vulnerable version, passwords were stored directly without hashing.
- **Mitigation :** In the secure version, we used generate_password_hash() function for hashing passwords.
- **How it works :** generate_password_hash() applies a strong hash, making brute-force attacks difficult, even if the database is leaked, original passwords cannot be easily recovered or used.

2.3 Cross-Site Scripting (XSS)

- **Vulnerability :** Cross-Site Scripting (XSS) occurs when user input is rendered on a web page without being escaped, allowing attackers to inject and run malicious JavaScript.
- **Intentional Vulnerability :** In the vulnerable version, user-submitted comments were stored and rendered in templates without sanitization
- **Mitigation :** In the secure version, the same content is rendered using auto-escaping.
- **How it works :** By default, we escapes all HTML characters unless explicitly marked as safe. This prevents any embedded scripts from executing, neutralizing XSS attempts.

2.4 Access Control (RBAC)

- **Vulnerability :** Without proper access control, any logged-in user can access privileged admin pages.
- **Intentional Vulnerability :** In the vulnerable version, the /admin route had no role-based restriction
- **Mitigation :** In the secure version, the route now checks the user's role before granting access
- **How it works :** By storing the user's role in the session and validating it before serving restricted pages, only admins can access protected content.

2.5 Encryption (Data &Communication)

- **Vulnerability :** Sensitive data like passwords and session information may be exposed if not encrypted, either in storage or during transmission.
- **Intentional Vulnerability :** passwords were stored in plain text and the app was served over HTTP.
- **Mitigation :** passwords are hashed and the Flask app is configured to run over HTTPS
- **How it works :**
  - Hashed passwords protect data at rest.
  - HTTPS (with SSL/TLS) encrypts communication between client and server, preventing attackers from intercepting login data and other sensitive information.

3. Challenges

1. Creating Vulnerable Code on Purpose

It was tricky to intentionally write insecure code (like for SQL injection) without breaking the app.

Solution: We carefully tested and added small, controlled vulnerabilities like using f-strings for SQL and skipping input validation in comments.

2. Handling Password Hashing

Switching from plain-text passwords to hashed ones caused login issues at first.

Solution: We updated both the registration and login logic to use generate_password_hash() and check_password_hash() consistently.

3. Learning a New Framework (Flask)

At the start, we were not familiar with Flask or how routing, templates, and session management work.

Solution: We followed Flask documentation and tutorials to understand how to build pages, handle form data, and use sessions securely.

4. Running the App with HTTPS

Setting up HTTPS locally using Flask's ssl_context was confusing and gave browser warnings.

Solution: We created self-signed SSL certificates and tested HTTPS locally to simulate secure communication.

4. key Security Implementation Decision

- **Password Hashing with generate_password_hash() and check_password_hash():**
  We chose to use Flask's built-in generate_password_hash() function to hash passwords before storing them. This decision was made to ensure that even if the database is compromised, attackers cannot retrieve actual passwords.
  During login, check_password_hash() is used to compare the entered password with the stored hash without revealing sensitive information.
  This method provides a secure and reliable way to authenticate users without exposing sensitive information.
- **Session Management for Authentication:**
  Flask sessions are used to manage user authentication. When a user logs in, their username, role, and user ID are stored in the session. These session values are then used to restrict access to certain routes like the dashboard and admin panel.
  We chose this method because it offers a lightweight and secure way to maintain stateful authentication across multiple requests.
  Sessions help maintain the user's login state across pages, allowing secure and controlled access to user-specific features without requiring repeated logins.
- **HTTPS for Secure Communication:**
  To protect user data during transmission, the app runs with HTTPS enabled using Flask's ssl_context=('cert.pem', 'key.pem') configuration. This ensures all data exchanged between the client and server is encrypted.
  HTTPS encrypts data exchanged between the client and server, protecting sensitive information such as login credentials and session cookies from being intercepted or modified. Using HTTPS is a key step in preventing man-in-the-middle attacks and ensuring data privacy.
- **Use of Prepared Statements (Recommended for Real Databases):**
  While the current setup uses an in-memory dictionary (user_db) for simplicity, a real-world application should connect to a database using prepared statements. This decision is based on the need to prevent SQL injection, one of the most common and dangerous web vulnerabilities.
  Technologies like PHP's PDO or Python's parameterized queries prevent SQL injection attacks by keeping user input separate from the SQL command structure. This is essential for securing any application that stores user data persistently.

Conclusion

This project allowed us to explore common web application vulnerabilities and apply secure development practices using Flask. By implementing both a vulnerable and a secure version of the app. The result is a functional, secure web application that follows modern security standards.