**PROJECT TOPIC: Secure Coding Review :Choose a programming language and application. Review the code for security vulnerabilities and provide recommendations for secure coding practices. Use tools like static code analyzers or manual code review.**

**1. Project Overview**

My project is a simple web application built with **Flask**, which includes a registration and login system. The application allows users to create accounts, log in, and manage sessions. The focus of this report is on reviewing the security aspects of my code and implementing secure coding practices to prevent common security vulnerabilities.

**2. Security Vulnerabilities in the Code**

**A. Password Storage**

**Vulnerability**: In my  original code, passwords were stored in plaintext. Storing passwords in plaintext is a serious security risk, as attackers can easily retrieve passwords if they gain access to the database.

**Fix Implemented**: I used **password hashing** to store user passwords securely. By using generate\_password\_hash from the werkzeug.security library, passwords are hashed before being stored in the database. This ensures that even if the database is compromised, attackers cannot easily retrieve the original passwords.

* **Hashing Algorithm**: I used the sha256 method, which is good for general purposes but could be strengthened with a salt to make attacks (e.g., rainbow table attacks) more difficult.

**Recommendation**:

* Use a stronger hashing algorithm like **bcrypt** or **argon2**, which automatically handles salting and is specifically designed for password storage. Libraries like **passlib** make it easy to implement bcrypt or argon2.

**B. Cross-Site Scripting (XSS)**

**Vulnerability**: My application includes user inputs (e.g., username, password) in HTML templates without escaping them, which could lead to **Cross-Site Scripting (XSS)** attacks.

**Fix Implemented**:

* Flask automatically escapes data in templates to prevent XSS by default. When you use {{ variable }} in Jinja2 templates, it ensures that the content is sanitized before being rendered in the HTML.

**Recommendation**:

* Continue using Flask’s automatic escaping mechanism for user inputs, as it helps prevent malicious scripts from being injected into your pages.

**C. Cross-Site Request Forgery (CSRF)**

**Vulnerability**: Without CSRF protection, My app is vulnerable to attacks where an attacker can make unauthorized requests on behalf of authenticated users (e.g., submitting the registration or login form without the user's consent).

**Fix Implemented**:

* **Flask-WTF** is used to implement CSRF protection by automatically adding a CSRF token to all forms. This prevents malicious requests from being processed.

**Recommendation**:

* Continue using **Flask-WTF** for CSRF protection, as it is a simple and effective way to secure forms against CSRF attacks.

**D. SQL Injection**

**Vulnerability**: SQL injection attacks can occur if user inputs are directly used in database queries without proper sanitization. For example, if you used raw SQL queries to query the database, an attacker might inject harmful SQL code.

**Fix Implemented**:

* I  used **SQLAlchemy**, an Object-Relational Mapping (ORM) tool, which prevents SQL injection by using prepared statements and automatically escaping input data. This ensures that malicious SQL cannot be executed even if an attacker tries to inject it.

**Recommendation**:

* Continue using SQLAlchemy or other ORMs to interact with the database, as they help prevent SQL injection attacks by using safe query methods.

**E. Session Management**

**Vulnerability**: If session cookies are not handled securely, they can be hijacked by attackers, leading to session fixation or session hijacking attacks.

**Fix Implemented**:

* I configured **secure cookies** for session management using SESSION\_COOKIE\_SECURE and SESSION\_COOKIE\_HTTPONLY. These settings ensure that session cookies are transmitted over HTTPS and cannot be accessed by JavaScript, protecting against **session hijacking**.

**Recommendation**:

* Keep using secure cookies and **session expiration** to automatically log out users after a period of inactivity. This reduces the risk of session fixation attacks.

**F. Brute-Force Attacks**

**Vulnerability**: My application does not currently limit the number of login attempts, which makes it vulnerable to brute-force attacks. Attackers can attempt to guess a user's password by repeatedly trying different combinations.

**Fix Implemented**:

* **Rate limiting** can be implemented using **Flask-Limiter** to restrict the number of login attempts a user can make in a short period.

**Recommendation**:

* Implement rate-limiting for the login route using **Flask-Limiter** or a similar package. This will protect against brute-force login attempts and help prevent attackers from guessing passwords through trial and error.

**3. Tools Used for Secure Coding Review**

**A. Static Code Analysis with Bandit**

* **Bandit** was used to analyze the code for security vulnerabilities automatically. It helped identify potential risks and confirmed that certain issues (e.g., password handling) were already addressed.

**Steps Taken**:

* Installed Bandit (pip install bandit)

Ran Bandit on the project folder:  
bash  
Copy code  
bandit -r my\_flask\_project

* Bandit reported no major vulnerabilities but served as a useful tool to ensure no additional security risks were overlooked.

**B. Manual Code Review**

* You manually reviewed the code for security best practices, such as ensuring password hashing, using CSRF protection, and using safe methods for database queries.

**4. Additional Recommendations for Secure Coding**

**A. Password Strength Validation**

* To ensure strong user passwords, implement password strength validation. A strong password should contain a mix of uppercase, lowercase, digits, and special characters.
* **Recommendation**: Use libraries like passlib to enforce password strength rules during registration.

**B. HTTPS (SSL/TLS)**

* Ensure that your application is served over HTTPS to encrypt data in transit and prevent man-in-the-middle (MITM) attacks.
* **Recommendation**: Obtain an SSL certificate and configure your web server (e.g., Nginx, Apache) to use HTTPS.

**C. Implement Logging and Monitoring**

* Implement logging of user actions (e.g., login attempts, registration) to monitor and detect unusual or suspicious behavior.
* **Recommendation**: Use **Flask-Logging** or Python’s built-in logging module to log critical actions, including failed login attempts, user creation, and changes to account data.

**D. Use Multi-Factor Authentication (MFA)**

* Enhance security by adding **Multi-Factor Authentication (MFA)** to your login system.
* **Recommendation**: Use third-party libraries like **Flask-Security** or **Flask-User** to integrate MFA and enhance user account security.

**5. Conclusion**

The current login and registration system has implemented several key security practices such as password hashing, CSRF protection, and session management. However, there are additional improvements that can be made to enhance security further, such as:

* Enforcing strong password policies
* Rate-limiting login attempts to prevent brute-force attacks
* Implementing logging and monitoring
* Enabling HTTPS for secure communication

By incorporating these additional security measures and using tools like Bandit for static code analysis, you can create a more secure and resilient web application.

**6.Steps Taken to Build the Project in Visual Studio Code**

**Step 1: Setting Up the Environment**

1. **Install Python**:

I ensured that Python was installed on my system by checking the version with the command:  
bash  
 code  
python --version

* If Python was not installed, I downloaded and installed it from the official Python website.

1. **Install Visual Studio Code**:
   * I installed **Visual Studio Code (VS Code)** from the official website and opened it to start the project.
2. **Create a Virtual Environment**:

I created a **virtual environment** to manage dependencies for the project and avoid conflicts with global Python packages:  
bash  
Copy code  
python -m venv venv

I activated the virtual environment:

On **Windows**:  
code  
.\venv\Scripts\activate

On **MacOS/Linux**:  
bash  
Copy code  
source venv/bin/activate

1. **Install Required Packages**:

I installed the necessary packages for the Flask application and security:  
bash  
Copy code  
pip install flask flask\_sqlalchemy flask\_migrate werkzeug

* If I was using any other packages like **Flask-WTF** or **Flask-Limiter** for additional features, I installed them as well.

1. **Set Up the Project Structure**:
   * I created a folder for my project, named it my\_flask\_project, and opened it in Visual Studio Code:

Inside the folder, I created the following structure:  
arduino  
Copy code  
my\_flask\_project/

    ├── app.py

    ├── templates/

        ├── login.html

        ├── register.html

    ├── static/

        ├── style.css

**Step 2: Writing the Flask Application Code**

1. **Set Up Flask Application (app.py)**:

I started by creating the main file app.py and wrote the initial Flask app setup:  
python  
Copy code  
from flask import Flask, render\_template, redirect, request, flash, url\_for

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

import os

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'your\_secret\_key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///users.db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

migrate = Migrate(app, db)

**2.Define the User Model**:

I defined the User model to represent the user data, including username and password:  
python  
Copy code  
class User(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    username = db.Column(db.String(80), unique=True, nullable=False)

    password = db.Column(db.String(200), nullable=False)

1. **Set Up Routes for Login and Registration**:

I created the routes for the **login** and **registration** pages, handling the form submissions and user authentication:  
python  
Copy code  
@app.route('/login', methods=['GET', 'POST'])

def login():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password']

        user = User.query.filter\_by(username=username).first()

        if user and user.password == password:

            flash("Login successful!", "success")

            return redirect(url\_for('index'))

        else:

            flash("Invalid username or password", "danger")

    return render\_template('login.html')

@app.route('/register', methods=['GET', 'POST'])

def register():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password']

        hashed\_password = generate\_password\_hash(password, method='sha256')

        new\_user = User(username=username, password=hashed\_password)

        db.session.add(new\_user)

        db.session.commit()

        flash('Registration successful! Please log in.', 'success')

        return redirect(url\_for('login'))

    return render\_template('register.html')

1. **Setting Up Database**:

I created the database when the app started if it didn't exist using:  
python  
Copy code  
if not os.path.exists('users.db'):

    with app.app\_context():

        db.create\_all()

        print("Database created.")

1. **Start Flask Application**:

I added the main execution block to start the Flask development server:  
python  
Copy code  
if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

**Step 3: Creating the HTML Templates**

1. **Login Template (login.html)**:

I created the login form inside the templates folder:  
html  
Copy code  
<form method="POST">

    <input type="text" name="username" placeholder="Username" required>

    <input type="password" name="password" placeholder="Password" required>

    <button type="submit">Login</button>

</form>

1. **Registration Template (register.html)**:

I created the registration form to capture the username and password:  
html  
Copy code  
<form method="POST">

    <input type="text" name="username" placeholder="Username" required>

    <input type="password" name="password" placeholder="Password" required>

    <button type="submit">Register</button>

</form>

**Step 4: Testing and Debugging in Visual Studio Code**

1. **Running the Application**:

I ran the application using the built-in terminal in **VS Code**:  
 code  
python app.py

* I visited http://127.0.0.1:5000 in my browser to test the application.

1. **Debugging**:
   * If there were any issues (e.g., 404 errors or unexpected behavior), I used **VS Code’s built-in debugger** to step through the code and inspect variables, inputs, and outputs.

**Step 5: Static Code Analysis Using Bandit**

1. **Installing Bandit**:

I installed Bandit in the virtual environment to perform static code analysis for security vulnerabilities:  
  
 code  
pip install bandit

1. **Running Bandit**:

I ran Bandit on my project folder to check for security issues:  
 code  
bandit -r my\_flask\_project/

* Bandit scanned the code and flagged any potential security risks, which I addressed (e.g., password handling).

**7. Conclusion**

The steps outlined above describe how I developed the login and registration functionality in Flask using **Visual Studio Code**. The process included setting up a virtual environment, installing dependencies, writing the Flask application code, implementing secure practices (like password hashing), creating HTML templates, testing the application, and running static code analysis with **Bandit** to ensure security.

By following this process, I ensured that the project adhered to secure coding practices, such as:

* Using password hashing (with generate\_password\_hash)
* Implementing secure database queries (via **SQLAlchemy**)
* Preventing Cross-Site Request Forgery (CSRF) by using **Flask-WTF**
* Securing session cookies to prevent session hijacking

These steps were executed and tested using **VS Code**, which provided an efficient and streamlined development experience for creating and securing the Flask web application.