# Secure Coding Review

**What is a Secure Code Review?**

Secure code review is the systematic examination of software source code, with the goal of identifying and fixing security vulnerabilities. It is becoming an integral part of the **software development life cycle (SDLC)** and helps improve the overall quality and security of the software.

By carefully reviewing the source code, developers can detect security flaws early on, thus preventing potential breaches and attacks in the future. Secure code reviews are not just about finding errors in the code, but also about understanding the patterns and practices that led to those errors.

This involves examining the architectural design of the application, the algorithms used, the choice of data structures, and the overall coding style. By gaining insights into these aspects, developers can make more informed decisions and avoid similar mistakes in the future.

**Importance of Secure Code Review in SDLC**

In an era where software applications form the backbone of businesses, ensuring their security is paramount. Code Reviews serve as an essential checkpoint in the SDLC, helping detect and rectify security vulnerabilities before the software is deployed.

Implementing a secure code review in the SDLC can significantly reduce the risk of security breaches. By identifying vulnerabilities in the early stages of development, developers can fix them promptly, thus preventing potential attacks.

Here's why it is an important part of the SDLC lifecycle-

- Reduces the risk of security breaches by finding vulnerabilities early.
- Improves code quality by enforcing good coding practices.
- Fosters a security-aware culture within the development team.
- Saves time, money, and reputation.

**The Secure Code Review Process**

Conducting a secure code review is not a one-time activity, but a continuous process that involves several stages.

This process can be broadly divided into seven stages:

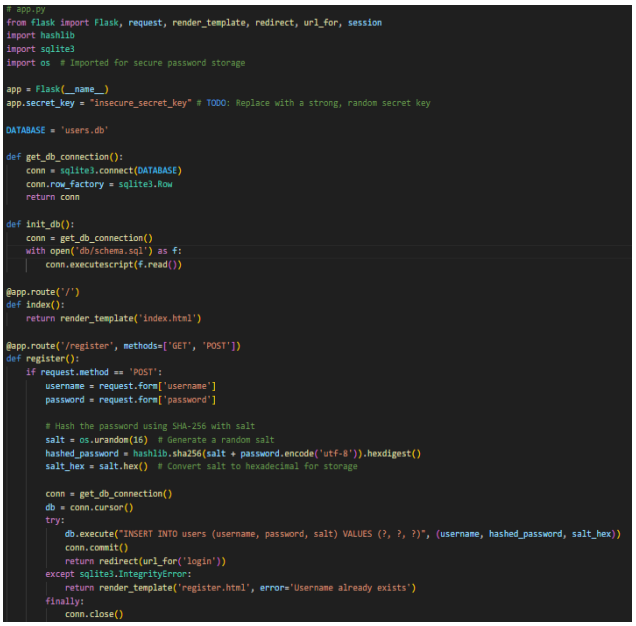
1. Overview of the application we are going to review
2. Review the code
3. Security Vulnerabilities and recommendations
4. Usage of Static Analysis Tools
5. Secure Coding Practices to keep in mind
6. Regular reviews and updating of practices
7. Important notes

**1. Programming Language & Application we are going to use for the Code Review:**

- **Language:** Python and HTML
- **Application:** A simple Flask-based web application for user registration and login. (We have kept it basic to focus on key security concepts.)

**2. Reviewing the code**

**app.py-**



```python
# app.py
from flask import Flask, request, render_template, redirect, url_for, session
import hashlib
import sqlite3
import os  # Imported for secure password storage

app = Flask(__name__)
app.secret_key = "insecure_secret_key" # TODO: Replace with a strong, random secret key

DATABASE = 'users.db'

def get_db_connection():
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

def init_db():
    conn = get_db_connection()
    with open('db/schema.sql') as f:
        conn.executescript(f.read())

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Hash the password using SHA-256 with salt
        salt = os.urandom(16)  # Generate a random salt
        hashed_password = hashlib.sha256(salt + password.encode('utf-8')).hexdigest()
        salt_hex = salt.hex()  # Convert salt to hexadecimal for storage

        conn = get_db_connection()
        db = conn.cursor()
        try:
            db.execute("INSERT INTO users (username, password, salt) VALUES (?, ?, ?)", (username, hashed_password, salt_hex))
            conn.commit()
            return redirect(url_for('login'))
        except sqlite3.IntegrityError:
            return render_template('register.html', error='Username already exists')
        finally:
            conn.close()
    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        conn = get_db_connection()
        db = conn.cursor()
        user = db.execute("SELECT password, salt FROM users WHERE username = ?", (username,)).fetchone()
        conn.close()

        if user:
            stored_password = user['password']
            stored_salt = bytes.fromhex(user['salt'])  # Convert salt back to bytes
            hashed_password = hashlib.sha256(stored_salt + password.encode('utf-8')).hexdigest()

            if hashed_password == stored_password:
                session['username'] = username
                return redirect(url_for('profile'))
            else:
                return render_template('login.html', error='Invalid credentials')
        else:
            return render_template('login.html', error='Invalid credentials')

    return render_template('login.html')


@app.route('/profile')
def profile():
    if 'username' in session:
        return render_template('profile.html', username=session['username'])
    else:
        return redirect(url_for('login'))

@app.route('/logout')
def logout():
    session.pop('username', None)
    return redirect(url_for('index'))

if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

**templates/index.html-**

```
<!DOCTYPE html>
<html>

<head>
    <title>Index</title>
</head>

<body>
    <h1>Welcome!</h1>
    <a href="{{ url_for('register') }}">Register</a> |
    <a href="{{ url_for('login') }}">Login</a>
    <a href="{{ url_for('profile') }}">Profile</a>
    <a href="{{ url_for('logout') }}">Logout</a>
</body>

</html>
```

**templates/login.html-**

```
<!DOCTYPE html>
<html>

<head>
    <title>Login</title>
</head>

<body>
    <h1>Login</h1>
    {% if error %}
    <p style="color: red;">{{ error }}</p>
    {% endif %}
    <form method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required><br><br>
        <button type="submit">Login</button>
    </form>
    <a href="{{ url_for('register') }}">Register</a>
</body>

</html>
```

**templates/profile.html-**

```
<!DOCTYPE html>
<html>

<head>
    <title>Profile</title>
</head>

<body>
    <h1>Profile</h1>
    <p>Welcome, {{ username }}!</p>
    <a href="{{ url_for('logout') }}">Logout</a>
</body>

</html>
```

**templates/register.html-**

```
<!DOCTYPE html>
<html>

<head>
    <title>Register</title>
</head>

<body>
    <h1>Register</h1>
    {% if error %}
    <p style="color: red;">{{ error }}</p>
    {% endif %}
    <form method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required><br><br>
        <button type="submit">Register</button>
    </form>
    <a href="{{ url_for('login') }}">Login</a>
</body>

</html>
```

## 3. Security Vulnerabilities and Recommendations:

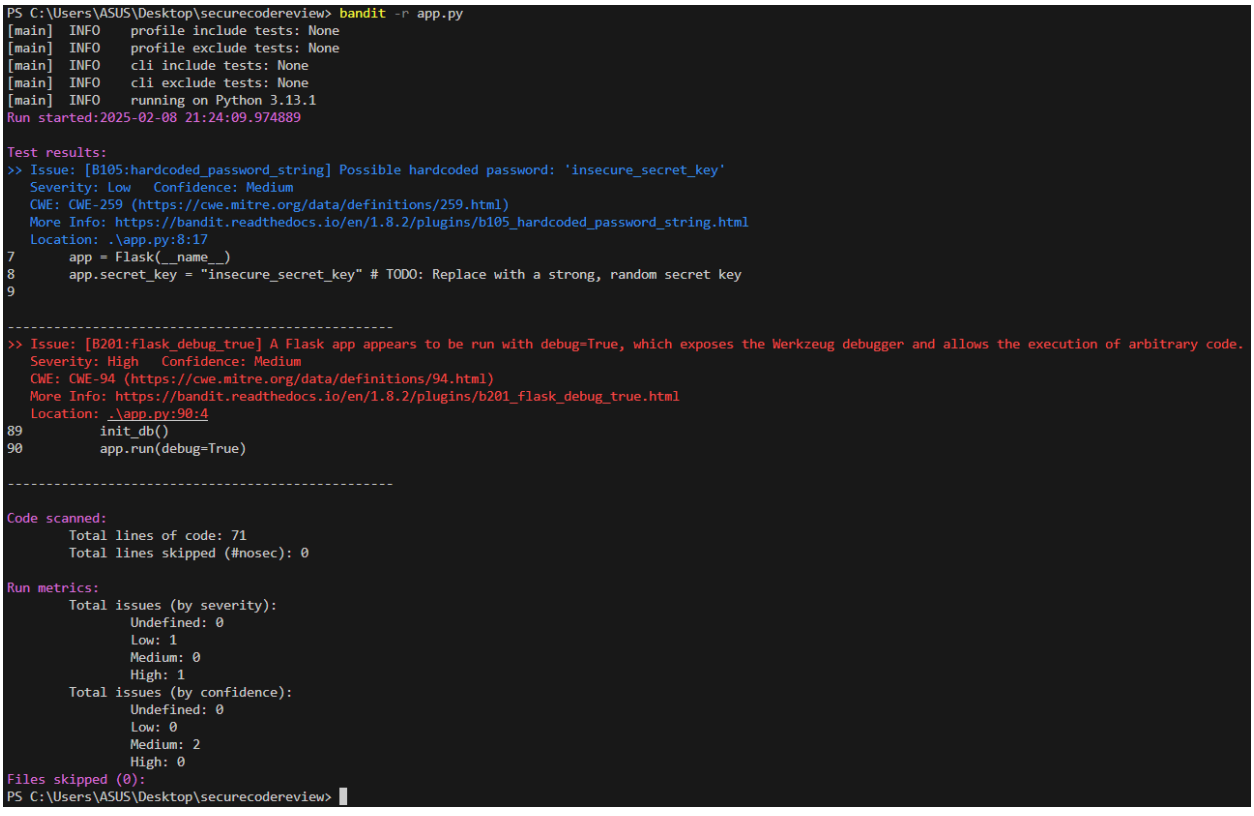
| Vulnerability | Location(s) | Description | Recommendation | Severity | Static Analysis Tool Coverage |
|---|---|---|---|---|---|
| **SQL Injection** | register & login routes | While parameterized queries are used, it's crucial to ensure all user input processed through SQL queries are properly sanitized and escaped. Even with parameters, there's a *potential* for injection if data handling is flawed before the query. | **Validate and sanitize *all* input received from the user.** Employ a stricter data validation strategy, using libraries designed to sanitize data. Review the usage of parameterized queries to ensure they are correctly implemented and resistant to bypasses. | High | Can detect parameterized queries but might not identify pre-query data handling issues. |
| **Cross-Site Scripting (XSS)** | profile.html | If the username contains malicious HTML/JavaScript, it will be executed in the user's browser. | **Escape HTML output:** Use `{{ usernamesafe }}` to prevent XSS. Flask's Jinja2 template engine has automatic escaping, but you need to specifically mark variables as safe. In this case, since you're displaying user-provided data, escaping is *essential*. | Medium | |
| **Insecure Secret Key** | app.py | The app.secret_key is set to a hardcoded, easily guessable value. | **Use a strong, random secret key:** Generate a long, random string and store it securely, ideally in an environment variable. Do *not* hardcode it in the source code. | High | Can detect hardcoded secrets. |
| **Missing HTTPS** | N/A | The application doesn't enforce HTTPS. | **Enforce HTTPS:** Configure the web server to only accept HTTPS connections and redirect HTTP requests to HTTPS. Use TLS/SSL certificates. This is *critical* for protecting sensitive data transmitted over the network (e.g., login credentials). | High | N/A |
| **CSRF Protection** | register & login routes | No Cross-Site Request Forgery (CSRF) protection is implemented, leaving the application vulnerable to unauthorized actions. | **Implement CSRF Protection:** Use Flask-WTF or a similar library to generate and validate CSRF tokens on all forms. | High | N/A |

| | | | | | |
|---|---|---|---|---|---|
| **Disclosure of Technical Information** | debug=True | Running the application with debug=True exposes sensitive debugging information, potentially revealing internal implementation details to attackers. | **Disable Debug Mode in Production:** Set debug=False when deploying the application to a production environment. | Low | Can be detected via configuration analysis |
| **Password Complexity** | N/A | The application doesn't enforce password complexity requirements, making it easier for attackers to crack passwords. | **Enforce Password Complexity:** Implement password policies that require a minimum length, a mix of uppercase and lowercase letters, numbers, and special characters. Provide feedback to users on password strength during registration and password changes. | Medium | N/A |
| **Rate Limiting** | login route | There's no rate limiting on the login endpoint, making it vulnerable to brute-force attacks. | **Implement Rate Limiting:** Use a library like Flask-Limiter to limit the number of login attempts from a single IP address within a given time period. | Medium | N/A |
| **Error Handling** | register, login | Generic error messages like "Invalid credentials" don't give the attacker specific information, but overly detailed error messages can leak sensitive information. | **Implement proper logging and error handling:** Log errors securely on the server-side. Return generic error messages to the user. Implement proper exception handling to prevent the application from crashing and potentially disclosing sensitive information. | Medium | N/A |
| **Unnecessary Permissions** | N/A | Consider the permissions granted to the application's service account or database user. | **Principle of Least Privilege:** Grant the application and its components only the minimum permissions required to perform their intended functions. For example, the database user should only have the necessary read/write access to the users table. | Medium | N/A |

**4. Static Analysis Tools:**

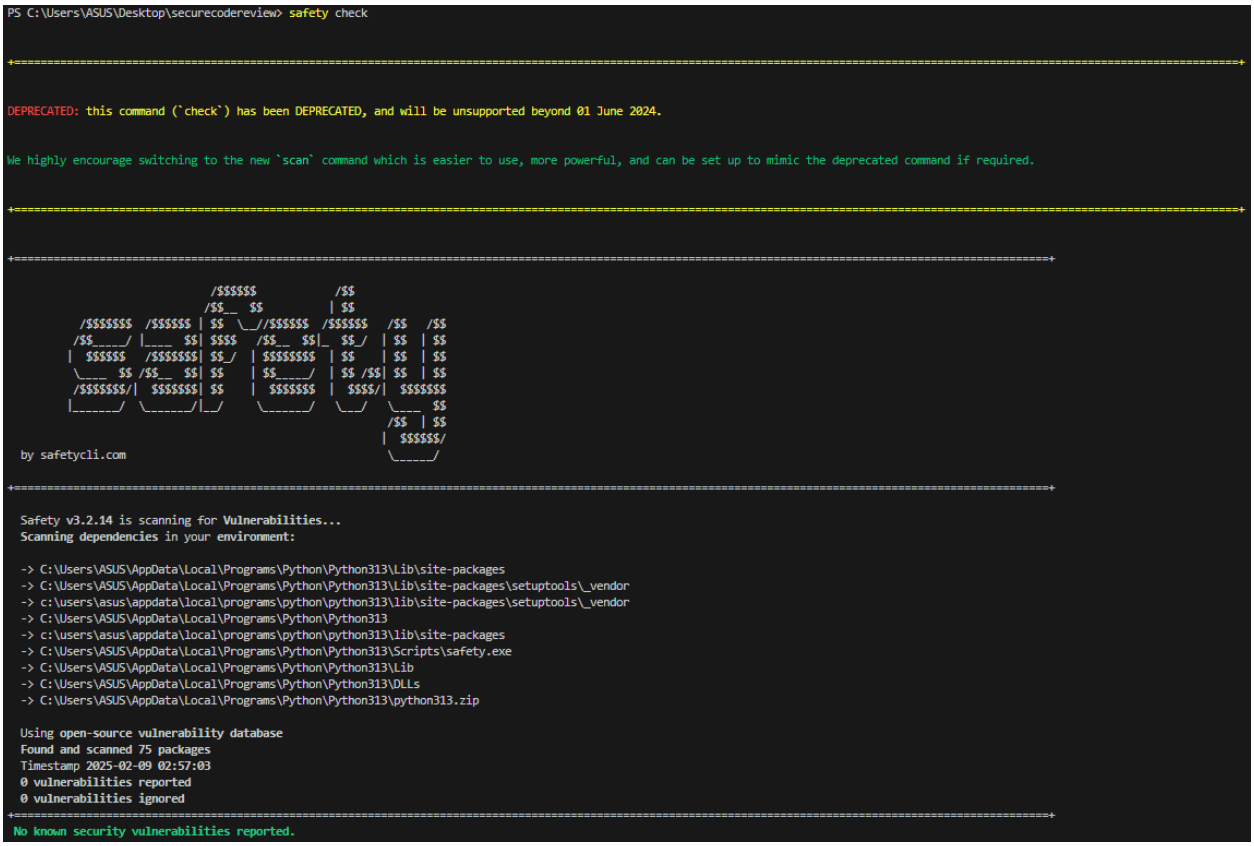
1. **Bandit:** A Python static analyzer specifically designed to find security vulnerabilities. It can identify things like:
    - Hardcoded passwords
    - SQL injection vulnerabilities (to some extent, especially if string formatting is used instead of parameterized queries)
    - XSS vulnerabilities (if data has not properly escaped)
    - Insecure use of tmpfile
    - Use of assert statements (which are disabled in optimized builds)
    - Many other common Python security issues.

Example usage:



2. **Safety:** Checks your Python dependencies for known vulnerabilities. This helps you keep your dependencies up-to-date and avoid using libraries with security flaws.

Example usage:



**5. Secure Coding Practices:**

- **Input Validation:** Always validate all input from users and external sources. This includes checking data types, lengths, formats, and allowed values.
- **Output Encoding:** Encode output to prevent XSS vulnerabilities. Use the appropriate encoding for the context (e.g., HTML encoding for HTML output, URL encoding for URLs).
- **Parameterized Queries (or ORM):** Use parameterized queries or an Object-Relational Mapper (ORM) to prevent SQL injection vulnerabilities. Parameterized queries ensure that user input is treated as data, not as executable code.
- **Secure Password Storage:** Never store passwords in plain text. Use a strong password hashing algorithm (like Argon2, bcrypt, or scrypt) which provides uniqueness in each password.
- **Authentication and Authorization:** Implement robust authentication and authorization mechanisms to control access to resources. Use strong session management techniques to prevent session hijacking.
- **Error Handling and Logging:** Implement proper error handling and logging. Log errors securely on the server-side. Return generic error messages to the user.
- **Secure Configuration Management:** Store sensitive configuration data (e.g., API keys, database passwords) securely, ideally in environment variables or a dedicated secrets management system. Don't hardcode secrets in the source code.
- **Regular Security Audits and Penetration Testing:** Conduct regular security audits and penetration testing to identify and address vulnerabilities in your application.
- **Keep Dependencies Up to Date:** Regularly update your dependencies to patch security vulnerabilities.
- **Principle of Least Privilege:** Grant users and applications only the minimum privileges they need to perform their tasks.
- **Code Reviews:** Have your code reviewed by other developers, ideally those with security expertise.
- **HTTPS:** Use HTTPS for all communications to protect data in transit.
- **CSRF Protection:** Implement Cross-Site Request Forgery (CSRF) protection on all state-changing operations.

- **Rate Limiting:** Implement rate limiting to prevent brute-force attacks and other abuse.
- **Content Security Policy (CSP):** Use CSP to control the sources from which the browser is allowed to load resources, mitigating XSS attacks.
- **Consider using a Web Application Firewall (WAF):** A WAF can help protect your application from common web attacks such as brute-force attacks.

**6. Regular reviews and Updating of Practices:**

**Develop code:** Finally, a crucial part of secure code review is regularly reviewing and updating your practices.

1. **Run static analysis:** Before committing, run Bandit (or another static analyzer) to identify potential vulnerabilities.
2. **Address findings:** Fix any issues reported by the static analyzer.
3. **Run security checks on dependencies:** Use safety check to check for vulnerable dependencies.
4. **Code review:** Have another developer review your code for security and other issues.
5. **Commit code:** Commit your code to the repository.
6. **CI/CD integration:** Integrate static analysis and security checks into your CI/CD pipeline. This will automatically run security checks on every commit.
7. **Penetration testing:** Periodically perform penetration testing to identify vulnerabilities that may have been missed by static analysis and code reviews.

**7. Important Notes:**

- This is a simplified example. Real-world applications are much more complex and require a more thorough security review.
- Static analysis tools are not silver bullets. They can help identify potential vulnerabilities, but they cannot guarantee that your application is secure.
- Security is an ongoing process. We need to continually monitor your application for vulnerabilities and update our security practices as new threats emerge.