Task - 3

Secure Coding Review Report

1. Task Overview

In this task, the objective is to review the source code of a web application and identify potential security vulnerabilities. The focus is to analyze the application for common security flaws, suggest secure coding practices, and recommend appropriate security measures to mitigate risks.

Application Overview

- **Programming Language:** Python
- **Application:** User authentication system (Login and Registration functionality)

2. Code Review Process

A detailed review of the source code was performed manually and using static code analysis tools (if applicable). The application was assessed for common security vulnerabilities such as weak password storage, SQL injection, improper session handling, and insufficient input validation.

Code Review Methodology

1. **Manual Code Review:** Analyzed the code for potential vulnerabilities based on known security issues and best practices in secure coding.
2. **Static Code Analysis Tools:** Used tools (such as Bandit, PyLint, or others) to automatically detect security vulnerabilities, such as weak cryptography, insecure SQL queries, and coding mistakes that can lead to security flaws.
3. **Security Standards & Best Practices:** Followed OWASP guidelines and other established security frameworks to identify security risks in the application.

3. Identified Vulnerabilities

The following vulnerabilities were identified during the code review process:

3.1 Weak Password Hashing

- **Issue:** The code uses SHA-256, which is not recommended for password storage due to its fast execution and vulnerability to brute-force attacks.
- **Impact:** A determined attacker can easily break SHA-256 hashes through brute-force methods.
- **Severity:** High

3.2 SQL Injection

- **Issue:** The application uses raw SQL queries with user inputs without proper sanitization in some cases.
- **Impact:** This leaves the application open to SQL injection attacks, where attackers can manipulate the SQL query to execute malicious commands on the database.
- **Severity:** Medium to High

3.3 Lack of Brute Force Protection

- **Issue:** There is no rate-limiting or account lockout mechanism to prevent brute-force attacks on the login functionality.
- **Impact:** Attackers can attempt an unlimited number of login attempts, making it easier to guess user passwords.
- **Severity:** Medium

3.4 Sensitive Information Exposure

- **Issue:** The database connection does not use encryption, and sensitive information like passwords could be accessed if the database is compromised.
- **Impact:** Exposure of user data, including passwords, poses a serious security risk.
- **Severity:** High

3.5 Improper Session Management

- **Issue:** The application does not implement secure session management practices, such as session tokens after login.
- **Impact:** Without proper session management, an attacker could impersonate a user or hijack the session.
- **Severity:** High

3.6 Error Handling and Logging

- **Issue:** The application exposes internal error details in responses, which could be exploited by attackers to gather information about the system.

- **Impact:** Detailed error messages can assist attackers in crafting successful attacks.
- **Severity:** Medium

4. Recommendations for Secure Coding Practices

4.1 Password Hashing

- **Recommendation:** Use a cryptographically secure password hashing algorithm such as **bcrypt**, **scrypt**, or **Argon2**, which are designed to be slow and computationally expensive, making them resistant to brute-force attacks.

Implementation Example:

```
 import bcrypt
password_hash = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
```

-

4.2 SQL Injection Prevention

- **Recommendation:** Always use **parameterized queries** or **prepared statements** to interact with the database to avoid SQL injection.

Implementation Example:

```
 cursor.execute("SELECT password_hash FROM users WHERE username=?",
(username,))
```

-

4.3 Brute Force Protection

- **Recommendation:** Implement rate limiting or account lockout mechanisms after a certain number of failed login attempts. This will help mitigate brute-force attacks.

Implementation Example:

```
 failed_attempts = {}
if failed_attempts[username] > 3:
    return "Too many failed attempts."
```

-

4.4 Sensitive Data Encryption

- **Recommendation:** Use **encrypted connections (SSL/TLS)** for database communication, and ensure sensitive information is stored securely (e.g., environment variables for database credentials).

- **Implementation Example:** Use SSL/TLS for database connections and store credentials in secure storage mechanisms.

4.5 Session Management

- **Recommendation:** Implement secure session management by using **JWT tokens** or **secure cookies** to track user sessions after authentication.
- **Implementation Example:** Use a library like **Flask-Session** or **Django sessions** for session handling.

4.6 Error Handling

- **Recommendation:** Avoid exposing detailed error messages to end users. Instead, log the detailed errors on the server side, and return generic messages to the user.
- **Implementation Example:** Use Python's **logging** library for server-side error logging.

5. Actions for Secure Coding Improvement

To ensure the security of the application, the following actions should be taken:

1. **Implement Secure Password Hashing:**
    - Update the password hashing mechanism to use bcrypt or another secure algorithm.
    - Ensure that the hashing process involves a salt for additional security.
2. **Refactor SQL Queries to Use Parameterized Queries:**
    - Review all SQL queries and ensure that they use prepared statements to prevent SQL injection vulnerabilities.
3. **Introduce Brute Force Protection:**
    - Add rate-limiting mechanisms or account lockouts after a certain number of failed login attempts.
4. **Implement Secure Session Management:**
    - Ensure secure session handling by using tokens or secure cookies to track authenticated users.
5. **Use SSL/TLS for Database Connections:**
    - Ensure that database connections are encrypted using SSL/TLS to prevent data interception.
6. **Review Error Handling and Logging:**
    - Configure the application to log errors securely without exposing sensitive information to end users.

6. Conclusion

This secure coding review has identified several critical vulnerabilities in the authentication system, including weak password hashing, SQL injection risks, lack of brute-force protection, and improper session management. The recommended actions will significantly improve the security posture of the application and reduce its exposure to common threats. By adopting these secure coding practices, the development team can better safeguard sensitive user data and enhance the overall security of the application.

Additional Actions:

1. **Use Static Code Analysis Tools:** For future code reviews, incorporate automated tools like **Bandit** (for Python) or **SonarQube** for continuous static code analysis.
2. **Regular Security Audits:** Schedule periodic security audits and penetration testing to identify and fix vulnerabilities.