**Secure Coding Review Report**

**Application: Flask-based Login System**

**Review Type: Manual Secure Code Review**

**Overview**

A security review was conducted on the initial login system implemented using **Flask** and **SQLite3**. The goal was to identify potential security vulnerabilities and recommend best practices to improve the system's security.

**Identified Security Risks**

1. **SQL Injection Vulnerability**
   * The initial code directly incorporated user input into an SQL query using **string formatting** (f"SELECT \* FROM users WHERE username = '{username}' AND password = '{password}'").
   * This allows **SQL injection attacks**, where an attacker can manipulate the query to gain unauthorized access.
2. **Plaintext Password Storage**
   * The database stored passwords in **plain text**, making them readable in case of a data breach.
   * This violates best security practices and makes user credentials highly vulnerable.
3. **Exposure of Credentials in URL (GET Request Issue)**
   * The login endpoint retrieved user credentials using a **GET request** (request.args.get()), exposing passwords in URLs, logs, and browser history.
   * Attackers monitoring network traffic (MITM attacks) can steal credentials easily.
4. **Lack of Response Security**
   * The system returned simple text messages ("Login successful" or "Login failed"), which could be **misused in brute-force attacks**.
   * The response structure also lacked proper error handling.

**Recommendations for Secure Coding Practices**

✅ **Use Parameterized Queries**

* Prevents SQL injection by safely handling user inputs.
* Corrected code example:

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

**def login():**

**username = request.args.get('username')**

**password = request.args.get('password')**

**conn = sqlite3.connect('example.db')**

**c = conn.cursor()**

**query = "SELECT \* FROM users WHERE username = ? AND password = ?"**

**print("Executing query with parameters:", query, (username, password))**

**c.execute(query, (username, password)) # Secure parameterized query**

**result = c.fetchone()**

**conn.close()**

**if result:**

**return "Login successful"**

**else:**

**return "Login failed"**

✅ **Implement Password Hashing**

* Store passwords securely using **bcrypt** or **PBKDF2** instead of plain text.
* Example (hashing a password before storage):

python

CopyEdit

import bcrypt

hashed\_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

✅ **Use POST Requests for Login**

* Prevents credential exposure in URLs by using request.form.get('username') instead of request.args.get().
* Example:

python

CopyEdit

username = request.form.get('username')

password = request.form.get('password')

✅ **Enhance Response Security**

* Return **JSON responses** instead of plain text for better security and structured data handling.
* Example:

python

CopyEdit

from flask import jsonify

return jsonify({"message": "Login successful"})

✅ **Rate Limiting & Logging**

* Implement **rate limiting** to prevent brute-force attacks.
* Use **logging & monitoring** to detect unusual login attempts.

**Conclusion**

The initial code contained critical security vulnerabilities, including **SQL injection risks, plaintext password storage, and exposure of credentials in URLs**. By implementing **parameterized queries, password hashing, POST requests, and structured responses**, the security of the system can be significantly improved. These changes align with industry best practices for secure web application development.