# **Secure Coding Review: Python Flask Web App**

### **1. Introduction and Scope**

This document details a secure code review performed on a simple user management application built with Python and the Flask framework. The purpose of this audit is to identify potential security vulnerabilities, provide concrete examples of the insecure code, and offer clear recommendations for remediation. The review utilized both manual code inspection and a conceptual static analysis.

### **2. Application Description**

The application is a basic web service that allows users to register, log in, and view their profile. It interacts with a hypothetical SQL database to store and retrieve user data.

### **3. Methodology**

The audit followed a two-phase approach:

1. **Manual Code Inspection:** A line-by-line review of the source code to identify common security anti-patterns, such as improper handling of user input, weak authentication mechanisms, and insecure data storage.
2. **Vulnerability Identification:** Based on the inspection, specific vulnerabilities were documented with a severity rating, a description of the risk, and a reference to the vulnerable code.

### **4. Findings and Vulnerabilities**

#### **Vulnerability 1: SQL Injection (High Severity)**

* **Description:** The user login function constructs an SQL query by directly concatenating user-provided input (username and password) into the SQL string. An attacker could enter malicious input (e.g., ' OR '1'='1' --) to bypass authentication and gain unauthorized access to the application.
* **Vulnerable Code:** See vulnerable\_app.py, lines 18-20.
* **Recommendation:** Use parameterized queries or prepared statements. These methods separate the SQL query structure from the user-provided data, preventing malicious input from being executed as part of the query itself.

#### **Vulnerability 2: Cross-Site Scripting (XSS) (Medium Severity)**

* **Description:** The user profile page directly renders a user's bio field without any sanitization or escaping. If a user stores malicious JavaScript code in this field (e.g., <script>alert('XSS!');</script>), it will be executed in the browsers of other users who view their profile, leading to potential session hijacking, data theft, or defacement.
* **Vulnerable Code:** See vulnerable\_app.py, lines 31-33.
* **Recommendation:** All user-provided input that will be displayed on a web page must be properly sanitized and escaped. Flask's Jinja2 template engine automatically escapes HTML, but it's crucial to be aware of this and avoid using the |safe filter unless absolutely necessary and after careful sanitization.

#### **Vulnerability 3: Hardcoded Secret Key (Medium Severity)**

* **Description:** The Flask application's secret key is hardcoded directly into the source code. This key is used for cryptographic operations, such as signing session cookies. If an attacker gains access to the source code, they can easily retrieve the key and potentially forge session cookies to impersonate users.
* **Vulnerable Code:** See vulnerable\_app.py, line 8.
* **Recommendation:** The secret key should be treated as a sensitive credential. It should be loaded from a secure source, such as an environment variable or a dedicated secret management system (e.g., AWS Secrets Manager, HashiCorp Vault).

### **5. Remediation and Best Practices**

To secure the application, the following remediation steps are recommended. The remediated\_app.py file demonstrates these fixes.

* **Implement Parameterized Queries:** The login query is refactored to use a prepared statement with placeholders (? or %s), preventing SQL injection. This is the single most effective way to prevent this class of attacks.
* **Sanitize User Input:** While Jinja2 provides built-in escaping, a more robust approach is to explicitly sanitize and validate user input before it is stored or used. For the bio field, ensure no malicious scripts can be stored.
* **Use Environment Variables for Secrets:** The Flask secret key is now loaded from the FLASK\_SECRET\_KEY environment variable. This ensures the secret is not exposed in the source code.
* **Principle of Least Privilege:** Ensure the database user account has only the minimum necessary permissions required by the application.
* **Error Handling:** Implement robust error handling to prevent the application from revealing sensitive information (e.g., database connection errors, stack traces) to the user.

By applying these changes, the application's security posture will be significantly improved, mitigating the most critical and common web vulnerabilities.

**Secure Code Review: Python Flask Web Application1. Introduction & Scope**

This document presents a secure code review of a basic user management application developed using Python and the Flask framework. The primary goal of this audit was to identify potential security vulnerabilities, provide concrete examples of the insecure code, and offer clear remediation recommendations. The review involved both manual code inspection and a conceptual static analysis.**2. Application Overview**

The application is a fundamental web service enabling users to register, log in, and view their profiles. It interacts with a hypothetical SQL database for user data storage and retrieval.**3. Methodology**

The audit was conducted in two phases:

1. **Manual Code Inspection:** A line-by-line examination of the source code was performed to pinpoint common security anti-patterns, including improper user input handling, weak authentication mechanisms, and insecure data storage.
2. **Vulnerability Identification:** Based on the inspection, specific vulnerabilities were documented with a severity rating, a description of the risk, and references to the vulnerable code.

**4. Findings & VulnerabilitiesVulnerability 1: SQL Injection (High Severity)**

* **Description:** The user login function constructs SQL queries by directly concatenating user-provided input (username and password). This allows an attacker to inject malicious input (e.g., ' OR '1'='1' --) to bypass authentication and gain unauthorized access.
* **Vulnerable Code:** See vulnerable\_app.py, lines 18-20.
* **Recommendation:** Utilize parameterized queries or prepared statements. These methods separate the SQL query structure from user-supplied data, preventing malicious input from being executed as part of the query.

**Vulnerability 2: Cross-Site Scripting (XSS) (Medium Severity)**

* **Description:** The user profile page directly renders a user's bio field without proper sanitization or escaping. Malicious JavaScript stored in this field (e.g., <script>alert('XSS!');</script>) would execute in the browsers of other users viewing the profile, potentially leading to session hijacking, data theft, or defacement.
* **Vulnerable Code:** See vulnerable\_app.py, lines 31-33.
* **Recommendation:** All user-provided input displayed on a web page must be thoroughly sanitized and escaped. Flask's Jinja2 template engine automatically escapes HTML; however, it's vital to be aware of this and avoid using the |safe filter unless absolutely necessary and after meticulous sanitization.

**Vulnerability 3: Hardcoded Secret Key (Medium Severity)**

* **Description:** The Flask application's secret key, used for cryptographic operations like signing session cookies, is hardcoded directly into the source code. If an attacker accesses the source code, they can easily retrieve this key and potentially forge session cookies to impersonate users.
* **Vulnerable Code:** See vulnerable\_app.py, line 8.
* **Recommendation:** The secret key is a sensitive credential and should be loaded from a secure source, such as an environment variable or a dedicated secret management system (e.g., AWS Secrets Manager, HashiCorp Vault).

**5. Remediation & Best Practices**

To enhance the application's security posture, the following remediation steps are recommended, as demonstrated in the remediated\_app.py file:

* **Implement Parameterized Queries:** The login query has been refactored to use a prepared statement with placeholders (? or %s), effectively preventing SQL injection. This is the most effective defense against this type of attack.
* **Sanitize User Input:** While Jinja2 provides built-in escaping, a more robust approach involves explicit sanitization and validation of user input before storage or use. For the bio field, this ensures no malicious scripts can be stored.
* **Use Environment Variables for Secrets:** The Flask secret key is now loaded from the FLASK\_SECRET\_KEY environment variable, preventing its exposure in the source code.
* **Principle of Least Privilege:** Ensure the database user account possesses only the minimum necessary permissions required by the application.
* **Error Handling:** Implement robust error handling to prevent the application from revealing sensitive information (e.g., database connection errors, stack traces) to the user.

By applying these changes, the application's security will be significantly improved, mitigating the most critical and common web vulnerabilities.