Secure Code Review Report

1. Introduction

This document provides a detailed secure code review analysis of a Python-based user login system. The review focuses on identifying common vulnerabilities, assessing their impact, and recommending secure coding practices to mitigate risks.

Code:

```
import sqlite3

ADMIN_USERNAME = 'admin'
ADMIN_PASSWORD = 'password123'

def connect_db():
    conn = sqlite3.connect('users.db')
    return conn

def create_table():
    conn = connect_db()
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                      id INTEGER PRIMARY KEY,
                      username TEXT,
                      password TEXT)''')
    conn.commit()
    conn.close()

def add_user(username, password):
    conn = connect_db()
    cursor = conn.cursor()
    cursor.execute(f"INSERT INTO users (username, password) VALUES
('{username}', '{password}')")
    conn.commit()
    conn.close()

def authenticate_user(username, password):
    conn = connect_db()
    cursor = conn.cursor()
    cursor.execute(f"SELECT * FROM users WHERE username='{username}' AND
password='{password}'")
    user = cursor.fetchone()
    conn.close()
    return user

def login():
    print("Welcome to the insecure login system!")
    username = input("Enter your username: ")
    password = input("Enter your password: ")

    if username == ADMIN_USERNAME and password == ADMIN_PASSWORD:
        print("Admin login successful!")
    elif authenticate_user(username, password):
        print("User login successful!")
```

```
    else:
        print("Login failed! Invalid credentials.")

def main():
    create_table()
    add_user('test_user', 'test_pass')
    login()

if __name__ == '__main__':
    main()
```

2. Tools Used

- **pylint**: General Python code analysis
- **bandit**: Security-focused Python code scanner
- **flake8**: Python style and minor vulnerability analysis
- **Manual Code Review**: Logical and security flow analysis

3. Vulnerability Findings

3.1 Hardcoded Credentials

- **Location:** Lines 6–7
- **Description:** Admin credentials are hardcoded into the source code.
- **Impact:** Exposure of plaintext credentials can lead to unauthorized admin access.
- **Recommendation:** Store credentials securely using environment variables or secret management tools.

Secure Example:

```
import os

ADMIN_USERNAME = os.getenv('ADMIN_USERNAME')

ADMIN_PASSWORD = os.getenv('ADMIN_PASSWORD')
```

3.2 SQL Injection

- **Location:** Lines 22, 30
- **Description:** User input is directly concatenated into SQL queries, allowing malicious query manipulation.
- **Impact:** Attackers can manipulate SQL queries to extract or modify sensitive data.

- **Recommendation:** Use parameterized queries.

Secure Example:

```
cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
(username, password))

cursor.execute("SELECT * FROM users WHERE username=? AND password=?",
(username, password))
```

3.3 Lack of Input Validation

- **Location:** Lines 45–46
- **Description:** User input is not validated or sanitized before processing.
- **Impact:** Malicious input can lead to injection attacks or system crashes.
- **Recommendation:** Validate user inputs using regex or validation libraries.

Secure Example:

```
import re

if not re.match("^[a-zA-Z0-9_]+$", username):

    print("Invalid username format!")
```

3.4 Insecure Database Connection

- **Location:** Line 10
- **Description:** The database connection lacks encryption and secure access controls.
- **Impact:** Data transmission can be intercepted.
- **Recommendation:** Use encrypted connections and secure database URIs.

Secure Example:

```
conn = sqlite3.connect('file:users.db?mode=ro', uri=True)
```

3.5 Lack of Exception Handling

- **Location:** Throughout the script
- **Description:** Database operations lack proper exception handling.

- **Impact:** Unhandled errors may expose sensitive information or crash the application.
- **Recommendation:** Implement proper try-except blocks.

Secure Example:

```
try:
    cursor.execute("SELECT * FROM users")
except sqlite3.Error as e:
    print(f"Database error: {e}")
finally:
    conn.close()
```

4. Summary Table of Findings

Vulnerability	Location	Risk Level	Recommendation
Hardcoded Credentials	Lines 6–7	High	Use environment variables.
SQL Injection	Lines 22, 30	Critical	Use parameterized queries.
Lack of Validation	Lines 45–46	Medium	Sanitize and validate inputs.
Insecure DB Connection	Line 10	Medium	Use secure connection options.
Exception Handling	Throughout	Medium	Add proper error handling.

5. Conclusion

The code review identified several critical vulnerabilities, including hardcoded credentials, SQL injection risks, and insufficient input validation. Addressing these issues through secure coding practices will significantly enhance the system's security posture.

6. Recommendations

I. Avoid hardcoding credentials.

II. Always use parameterized queries to prevent SQL injection.

III. Validate all user inputs.

IV. Implement secure database connections.

V. Add comprehensive exception handling.

7. References

- OWASP Top Ten Vulnerabilities
- Python Documentation
- SQLite Best Practices

Report Prepared By: Khandaker Shahariar

Date: 01/01/2025