# CodeAlpha Cyber Security Internship

## Project 3: Secure Coding Review

Submitted by: [Suraj Mishra]

## 1. Introduction

The Secure Coding Review project involves auditing application code for potential security vulnerabilities. The goal is to identify insecure practices, common flaws, and provide recommendations to ensure safer and more resilient code. This project demonstrates an understanding of secure coding standards, vulnerability analysis, and remediation strategies.

## 2. Methodology

Step 1: Select a programming language and application for review (Python application).

Step 2: Conduct static analysis using tools such as Bandit and Flake8 to detect vulnerabilities.

Step 3: Perform manual inspection to identify insecure coding practices including:

- Hardcoded credentials

- Weak input validation

- Insecure use of system commands

- Missing error handling

- Poor cryptography practices

Step 4: Document findings and classify them as High, Medium, or Low severity.

Step 5: Provide remediation steps and secure coding best practices.

## 3. Findings

During the review of a sample Python web application, the following vulnerabilities were identified:

1. Hardcoded Database Passwords (High Risk)

- The application stored DB credentials directly in the source code.

2. Insecure Input Validation (Medium Risk)

- User inputs were not sanitized, making the app vulnerable to SQL Injection and XSS.

3. Weak Cryptography (Medium Risk)

- MD5 hashing was used for password storage instead of modern algorithms.

4. Missing Exception Handling (Low Risk)

- Lack of proper try/except blocks caused application crashes on invalid inputs.

## 4. Recommendations & Best Practices

Based on the findings, the following remediation steps are recommended:

1. Remove hardcoded credentials. Use environment variables or secure vaults.

2. Implement strict input validation and sanitization to prevent injection attacks.

3. Use secure cryptographic algorithms like bcrypt or Argon2 for password hashing.

4. Implement structured error handling to ensure application stability.

5. Follow OWASP Secure Coding Guidelines for ongoing development.

## 5. Code Examples

### Vulnerable Code (Before Fix)

```
import hashlib
import sqlite3

# Hardcoded database credentials
DB_USER = "admin"
DB_PASS = "password123"

def login(username, password):
    conn = sqlite3.connect("users.db")
```

```
    cursor = conn.cursor()
    # Insecure query (SQL Injection vulnerability)
    cursor.execute(f"SELECT * FROM users WHERE username='{username}' AND
password='{password}'")
    result = cursor.fetchone()
    if result:
        print("Login successful")
    else:
        print("Login failed")

# Insecure password hashing
def store_password(password):
    hashed = hashlib.md5(password.encode()).hexdigest()
    print("Stored (weak hash):", hashed)
```

## Secure Code (After Fix)

```
import bcrypt
import sqlite3
import os

def login(username, password):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    # Secure parameterized query
    cursor.execute("SELECT password FROM users WHERE username=?", (username,))
    result = cursor.fetchone()
    if result and bcrypt.checkpw(password.encode(), result[0]):
        print("Login successful")
    else:
        print("Login failed")

# Secure password storage
def store_password(password):
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode(), salt)
    print("Stored (secure hash):", hashed)
```