Task (Secure Coding Review) Report

Project Introduction

This report provides an overview of the security assessment conducted on the Flask application app.py, with a specific focus on identifying SQL injection vulnerabilities. It also includes the findings from a static code analysis. The main objective was to evaluate the application's security posture and ensure it is protected against common coding vulnerabilities.

Setup Instructions

- Create a Virtual Environment

  Python - m venv .venv

- Activate the Virtual Environment

  .venv\Scripts\activate

- Install Required Packages (Ensure you have Flask and testing libraries installed)

  pip install Flask unittest mock

Code Snippets (SQL Injection Testing with Mocks)

The code below represents the unit test created to evaluate the application for SQL injection security flaws:

- import importlib.util
- import sys
- import unittest
- from unittest.mock import patch, MagicMock

#Import the application module

- spec=importlib.util.spec_from_file_location("app","C:/Users/MC/Intership/codeAlpha_TASK#3(Secure Coding Review)/.venv/app.py.py")
- app_module = importlib.util.module_from_spec(spec)
- sys.modules["app"] = app_module
- spec.loader.exec_module(app_module)
- app = app_module.app

SQL Injection Test Class

class SQLInjectionTest(unittest.TestCase)

    @patch('sqlite3.connect')

```
    def test_sql_injection(self, mock_connect):
        mock_cursor = MagicMock()
        mock_connect.return_value.cursor.return_value = mock_cursor
    # Simulate a login attempt with SQL injection
    response = app.test_client().post('/login', data={'username': "' OR '1'='1", 'password': "' OR '1'='1"})
    # Check that the expected response is returned
    session = response.environ['werkzeug.session']
    self.assertIn('Invalid credentials!', session['_flashes'][0][1])
    # Check the first flash message
if __name__ == '__main__':
    unittest.main()
```

Commands Used

- Run the Application

  python app.py

- Static Code Analysis

  bandit -r app.py

  pylint app.py

  pyflakes app.py

Run Unit Tests

- Run the Application in a Test Environment
  python -m unittest test_app.py
- Check for SQL Injection Vulnerability Without a Database
  python -m unittest test_sql_injection.py

Tools & Methodologies Used

1. **Bandit**

- **Purpose**

Bandit is a specialized security linter designed to identify common security issues in Python code.

- **Usage**

We utilized Bandit to detect vulnerabilities such as hardcoded credentials, improper use of APIs, and potential SQL injection points. This proactive scanning process aids in identifying risks early in the development cycle, significantly reducing the chances of exploitation.

2. **Pylint**

- **Purpose**

Pylint is a robust static analysis tool that enforces coding standards and detects errors and code smells.

- **Usage**

Through Pylint, we ensured adherence to best practices, improving code clarity, maintainability, and reliability. It provided valuable feedback on naming conventions, structural issues, and unused imports, promoting cleaner and more consistent code.

3. **Pyflakes**

- **Purpose**

Pyflakes is a lightweight static analyzer focused on identifying coding errors in Python scripts.

- **Usage**

We employed Pyflakes to uncover issues like unused variables and syntax errors, without enforcing stylistic conventions. It effectively complemented Pylint by concentrating strictly on functional errors, enhancing the overall debugging process.

4. **Unit Testing with test_app.py**

- **Purpose**

Unit testing plays a critical role in validating the behavior of individual components within the application.

- **Usage**

The test_app.py script was used to simulate SQL injection attacks using mock objects to test input validation and response. This ensured that the application could resist such vulnerabilities. Automated testing also established a safety net for future updates, preserving the security and stability of the codebase.

Results

- The SQL injection test passed, indicating that the application properly handles potentially harmful input.
- Static analysis tools did not report any critical issues, ensuring the code quality is maintained.

Conclusion

The evaluation for SQL injection vulnerabilities was effectively conducted through **unit testing** using mock objects. Results from static code analysis further validated the application's **security and code integrity**.

To enhance long-term protection, it is recommended to:

- Incorporate **routine code reviews**
- Extend **automated security testing measures**