Task Completion Report

Task Overview

This report summarizes the work completed on checking for SQL injection vulnerabilities in the Flask application app.py.py and the results of static code analysis. The primary goal was to ensure the application is secure and free from common vulnerabilities.

Setup Instructions

1. Create a Virtual Environment

   python -m venv .venv

2. Activate the Virtual Environment
   .venv\Scripts\activate

3. Install Required Packages: Ensure you have Flask and testing libraries installed:

   pip install Flask unittest mock

4. Code Snippets

   SQL Injection Testing with Mocks

   Here is the unit test code for checking SQL injection vulnerabilities:

```
import importlib.util

import sys

import unittest

from unittest.mock import patch, MagicMock


# Load the app module

spec = importlib.util.spec_from_file_location("app",
"C:\Users\Saipavan\Desktop\CodeAlpha_CyberSecurity\Task-3\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


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()
```

5. Commands Used

Run the Application:

```
python app.py.py
```

Static Code Analysis

```
bandit -r app.py.py
pylint app.py.py
pyflakes app.py.py
```

6. **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
```

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 task of checking for SQL injection vulnerabilities was successfully completed using unit tests with mocks. The static code analysis confirmed the overall security and quality of the application. Further improvements could include regular code reviews and additional security testing to ensure ongoing safety.

In this project, we utilized several tools and methods to ensure the security and quality of our Flask application, app.py.py. Each tool and approach played a critical role:

1. **Bandit**:
    - **Purpose**: Bandit is a security linter specifically designed to find common security issues in Python code.
    - **Use**: By running Bandit, we were able to identify potential vulnerabilities such as hardcoded passwords, insecure use of APIs, and SQL injection risks. This proactive approach helps in mitigating security threats before they can be exploited.
2. **Pylint**:
    - **Purpose**: Pylint is a comprehensive linting tool that checks for coding standards, errors, and code smells in Python programs.
    - **Use**: Using Pylint allowed us to enforce best practices in our code, ensuring it is clean, readable, and maintainable. It provides feedback on variable naming, code structure, and unused imports, which contributes to overall code quality.
3. **Pyflakes**:
    - **Purpose**: Pyflakes is a lightweight tool that detects errors in Python source files.
    - **Use**: Pyflakes helped us catch potential errors such as unused variables and syntax issues without imposing coding style rules. It complements the functionality of Pylint by focusing purely on error detection.
4. **Unit Testing with test_app.py**:
    - **Purpose**: Unit testing is essential for verifying that individual components of the application function correctly.
    - **Use**: The test_app.py file specifically tests for SQL injection vulnerabilities using mock objects. This ensures that our application is resilient against attacks, and the tests provide a safety net for future code changes. Automated testing helps in maintaining application integrity over time.

By incorporating these tools and testing methodologies, we not only improved the security posture of our application but also enhanced its overall quality and reliability. This comprehensive approach is essential in modern software development, where security and maintainability are paramount.