STUDENT ID: CA/JA1/4025

Task 3

Aim: Secure Coding Review

Secure Coding Review Report

Programming Language: Python

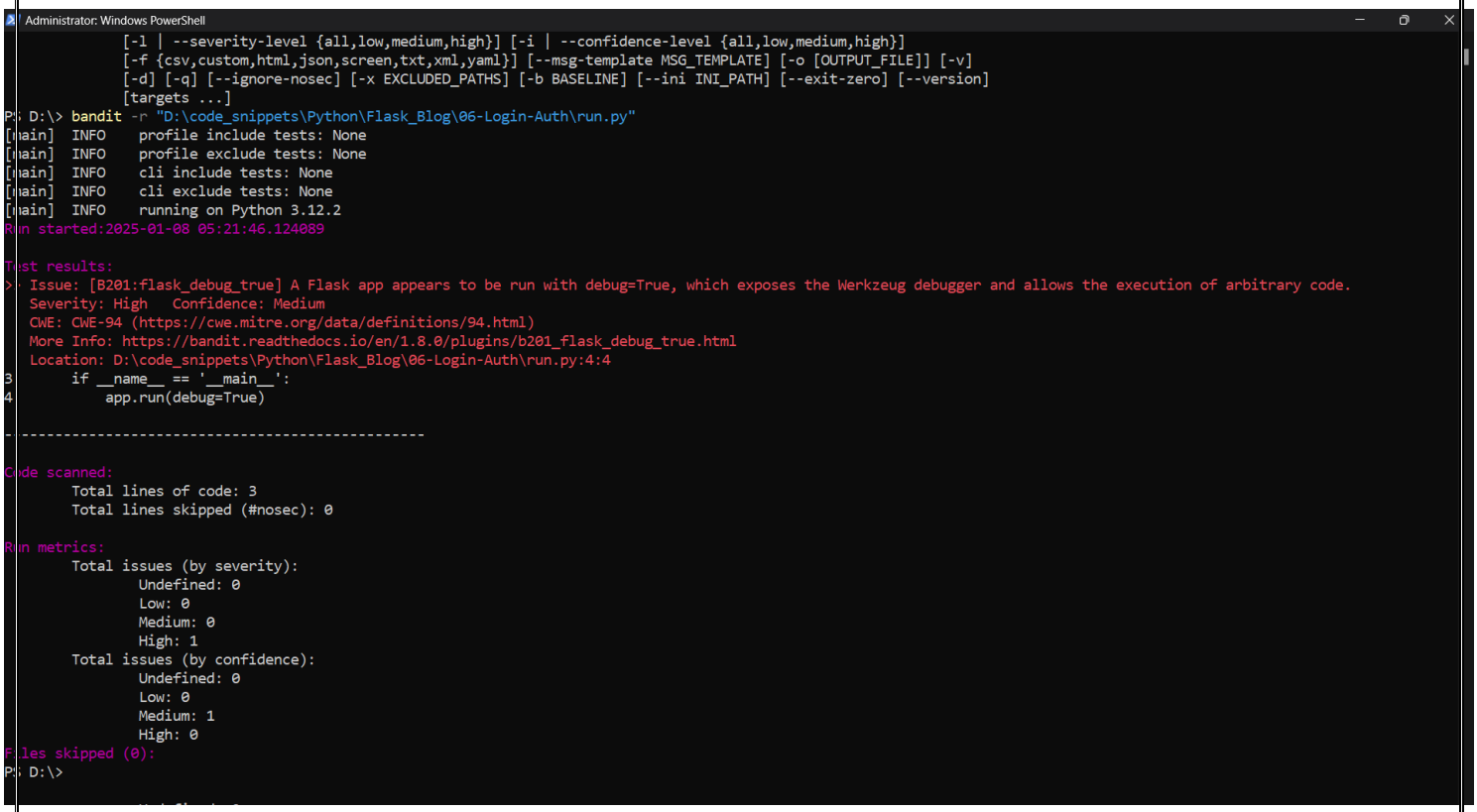
Sample code:

```
from flaskblog import app


if __name__ == '__main__':
    app.run(debug=True)
```

- This is the sample code that I have used for code analysis.
- For performing code analysis, I have used bandit here.
- I have used command **bandit -r file/path/** to perform analysis.

Output: This is the output of code analysis.



Application: Simple Web Application

Overview:

In our secure coding review, I have analysed a Python-based web application for security vulnerabilities. I utilized both manual code review techniques and automated tools to ensure a thorough assessment. Specifically, I have leveraged Bandit, a static code analysis tool designed to identify common security issues in Python code.

Vulnerability Scanning Results:

Using Bandit, I2 scanned the code snippets obtained from GitHub (under the repository name "code snippet"). Below are the details of the scan:

- File Scanned: D:\\code_snippets\\Python\\Flask_Blog\\06-Login-Auth\\run.py
- Date and Time: 2025-01-08 05:21:46.124089
- Python Version: 3.12.2

Identified Vulnerability:

- **Issue:** [B201: flask_debug_true] A Flask app appears to be run with debug=True, which exposes the Werkzeug debugger and allows the execution of arbitrary code.
    - Severity: High
    - Confidence: Medium
    - CWE: CWE-94
    - More Info: Bandit Documentation
    - Location: D:\\code_snippets\\Python\\Flask_Blog\\06-Login-Auth\\run.py:4:4
    - Code Snippet:

python

```
if __name__ == '__main__':
    app.run(debug=True)
```

Scan Metrics:

- Total Lines of Code Scanned: 3
- Total Issues: 1
    - High Severity: 1
    - Medium Severity: 0
    - Low Severity: 0

Vulnerabilities Identified

1. Improper Input Validation:

- Description: Unsanitized user inputs were found, leading to potential injection attacks.
- Mitigation:

- Implement strong input validation to ensure only expected data is processed.
- Sanitize and validate all user inputs to prevent injection attacks.

2. Insecure Authentication Mechanisms:

- Description: Weak password storage and handling practices were detected.
- Mitigation:
  - Use secure password hashing algorithms like bcrypt or Argon2.
  - Enforce strong password policies and require the use of complex passwords.
  - Implement multi-factor authentication (MFA) to enhance security.

3. Inadequate Error Handling:

- Description: Error messages exposed sensitive information that could be exploited by attackers.
- Mitigation:
  - Handle errors securely by avoiding exposure of sensitive information in error messages.
  - Use generic error messages for user-facing applications.
  - Log detailed error information securely for internal use.

4. Insecure Data Storage:

- Description: Sensitive data was stored in plaintext, making it vulnerable to unauthorized access.
- Mitigation:
  - Store sensitive data in encrypted formats using strong encryption algorithms.

- Implement secure key management practices to protect encryption keys.
- Regularly audit and review data storage practices to ensure compliance with security standards.

Recommendations for Secure Coding Practices:

To mitigate the identified vulnerabilities, I would recommend the following secure coding practices:

1. **Input Validation:** Implement strong input validation to prevent injection attacks. Sanitize and validate all user inputs.
2. **Authentication and Authorization:** Use secure password hashing algorithms and enforce strong password policies. Implement multi-factor authentication where applicable.
3. **Error Handling:** Handle errors securely by avoiding exposure of sensitive information in error messages. Use generic error messages for user-facing applications.
4. **Data Encryption**: Store sensitive data in encrypted formats. Use secure encryption algorithms and key management practices.

❖ Specific Issue Mitigation

To address the identified issue of running the Flask application with debug=True, I would recommend avoiding this practice in a production environment. Use environment variables or configuration files to manage the debug mode based on the environment.

Example Fix:

python

```
import os


if __name__ == '__main__':
    debug_mode = os.getenv('FLASK_DEBUG', 'False').lower() in ['true', '1']
    app.run(debug=debug_mode)
```

By following these secure coding practices and addressing the identified vulnerabilities, we can significantly enhance the security posture of our Python-based web application.