Secure Coding Review 20240917/vikramjat

Application Overview

We'll review a basic Python Flask web application that has a simple login feature. The app allows users to enter their username and password, which are then validated.

Example Code

Here's a basic example of a Flask web application with a login feature:

```
from flask import Flask, request, render_template_string, redirect, url_for


app = Flask(__name__)


# In-memory user store for demonstration purposes

users = {'admin': 'password123'}


@app.route('/')

def home():

    return 'Welcome to the secure web application!'


@app.route('/login', methods=['GET', 'POST'])

def login():
```

```
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        if username in users and users[username] == password:
            return redirect(url_for('home'))
        else:
            return 'Invalid username or password'

    return '''
        <form method="post">
            Username: <input type="text" name="username">
            Password: <input type="password" name="password">
            <input type="submit" value="Login">
        </form>
    '''

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

Review for Security Vulnerabilities

1. Hard-Coded Credentials

Issue:

- Storing credentials in code (`users` dictionary) is insecure. If someone gains access to the source code, they can see the passwords.

Recommendation:

- Store credentials securely using environment variables or a dedicated secrets management service. Use hashed passwords with a strong hashing algorithm like bcrypt.

Example:

```
from werkzeug.security import generate_password_hash, check_password_hash

import os


# Store hashed passwords instead of plain text

users = {'admin': generate_password_hash('password123')}


@app.route('/login', methods=['GET', 'POST'])

def login():

    if request.method == 'POST':
```

```
        username = request.form['username']
        password = request.form['password']

        if username in users and
check_password_hash(users[username], password):
            return redirect(url_for('home'))
        else:
            return 'Invalid username or password'

    return '''
        <form method="post">
            Username: <input type="text" name="username">
            Password: <input type="password"
name="password">
            <input type="submit" value="Login">
        </form>
    '''
```

2. Lack of Input Validation

Issue:

- User inputs are not validated or sanitized, which could lead to various attacks like SQL Injection if extended to database queries or other issues.

Recommendation:

- Validate and sanitize inputs to prevent injection attacks and other malicious inputs.

Example:

- For simple forms like this, basic validation can be done on the client side, but server-side validation is also essential.

3. Using `debug=True` in Production

Issue:

- Running the Flask application with `debug=True` in production can expose detailed error messages and stack traces, which can be useful to an attacker.

Recommendation:

- Ensure `debug` is set to `False` in production environments.

Example:

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

4. No Rate Limiting

Issue:

- The application does not implement any rate limiting, which makes it vulnerable to brute force attacks.

Recommendation:

- Implement rate limiting to mitigate brute force attacks.

Example:

- Use Flask-Limiter or similar libraries to add rate limiting.

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address


limiter = Limiter(
    get_remote_address,
    app=app,
    default_limits=["200 per day", "50 per hour"]
)


@app.route('/login', methods=['GET', 'POST'])
@limiter.limit("5 per minute")
def login():
    ...
```

Static Code Analysis Tools

1. **Bandit**: A static analysis tool designed to find common security issues in Python code.
    - **Install Bandit:**

      ```
      pip install bandit
      ```

    - **Run Bandit:**

      ```
      bandit -r path/to/your/code
      ```

2. **PyLint**: Although not solely focused on security, it can help identify code quality issues that may lead to vulnerabilities.
    - **Install PyLint:**

      pip install pylint

    - **Run PyLint:**

      `pylint path/to/your/code`

Manual Code Review Checklist

1. **Authentication and Authorization:**
    - Check for secure password storage.
    - Ensure proper access controls are in place.
2. **Input Validation:**
    - Verify that all inputs are validated and sanitized.
3. **Error Handling:**
    - Ensure error messages do not expose sensitive information.
4. **Configuration Management:**
    - Check that sensitive configurations (e.g., debug settings) are appropriately managed for different environments.
5. **Dependencies:**
    - Review and update dependencies regularly to avoid known vulnerabilities