Web application: Python using the Flask framework

Introduction

A simple web application for user authentication written in Python using the Flask framework. Here's an example code snippet for user registration and login:

```
from flask import Flask, request, jsonify

from werkzeug.security import generate_password_hash, check_password_hash

import sqlite3

app = Flask(__name__)

def create_user_table():
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users
```

```
            (id INTEGER PRIMARY KEY, username TEXT,
password TEXT)''')
    conn.commit()
    conn.close()
create_user_table()
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data['username']
    password = data['password']
    hashed_password = generate_password_hash(password,
method='sha256')
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('INSERT INTO users (username, password)
VALUES (?, ?)', (username, hashed_password))
    conn.commit()
    conn.close()
```

```
    return jsonify({'message': 'Registered successfully'}), 201


@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data['username']
    password = data['password']
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM users WHERE username = ?',
(username,))
    user = cursor.fetchone()
    conn.close()
    if user and check_password_hash(user[2], password):
        return jsonify({'message': 'Login successful'}), 200
    else:
        return jsonify({'message': 'Invalid credentials'}), 401
```

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

Review of the Code for Security Vulnerabilities

1) SQL Injection:

The code uses string interpolation with cursor.execute for database queries, which is vulnerable to SQL injection attacks.

2) Plaintext Password Transmission:

The passwords are transmitted in plaintext over the network. Without HTTPS, this can be intercepted by attackers.

3) Debug Mode:

Running the Flask app in debug mode (app.run(debug=True)) can expose sensitive information and should not be used in production.

4) Weak Password Hashing:

While generate_password_hash with sha256 is used, it is better to use a more secure hashing algorithm like bcrypt**.**

Recommendations for Secure Coding Practices

1) Use Parameterized Queries:

Prevent SQL injection by using parameterized queries with placeholders.

2) Enforce HTTPS:

Ensure the application uses HTTPS to protect data in transit.

3) Disable Debug Mode in Production:

Always set debug=False in production environments.

4) Use a Stronger Password Hashing Algorithm:

Use bcrypt for hashing passwords to improve security.

5) Implement Input Validation and Sanitization:

Validate and sanitize all user inputs to prevent various types of injection attacks**.**

Manual Code Review

1) SQL Injection:

The code correctly uses parameterized queries for both the INSERT and SELECT statements, which helps prevent SQL injection attacks.

2) Password Hashing:

The code uses generate_password_hash from werkzeug.security, which is good for hashing passwords. However, using bcrypt would provide a stronger hashing algorithm.

3) Plaintext Password Transmission:

The application transmits passwords in plaintext over the network. Without HTTPS, these can be intercepted.

4) Error Handling:

Detailed error messages are sent to the client, which can expose sensitive information.

5) Debug Mode:

Running the Flask app in debug mode (app.run(debug=True)) exposes detailed error information and should be avoided in production.

6) Input Validation:

There is no validation or sanitization of user input (e.g., checking for empty strings).

Static Code Analysis Tools

Using static code analysis tools can further enhance the security and quality of your code. Here are some recommended tools:

1) Bandit:

- Bandit is a tool designed to find common security issues in Python code.
- Bandit GitHub

Example usage:

- pip install bandit
- bandit -r your_flask_app.py

2) Pylint:

- Pylint is a static code analyzer for Python. It looks for programming errors, helps enforce a coding standard, and looks for code smells.
- Pylint GitHub

Example usage:

- pip install pylint
- pylint your_flask_app.py

3) Flake8:

- Flake8 is a tool for enforcing coding style (PEP 8), checking for programming errors, and linting.
- Flake8 GitHub

Example usage:

- pip install flake8
- flake8 your_flask_app.py

4) Safety:

- Safety checks your dependencies for known security vulnerabilities.
- Safety GitHub

Example usage:

- pip install safety
- safety check

Revised Code

from flask import Flask, request, jsonify

from werkzeug.security import generate_password_hash, check_password_hash

import sqlite3

```
import bcrypt

app = Flask(__name__)

def create_user_table():
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users
                 (id INTEGER PRIMARY KEY, username TEXT,
password TEXT)''')
    conn.commit()
    conn.close()
create_user_table()
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data['username']
    password = data['password']
    if not username or not password:
```

```
        return jsonify({'message': 'Username and password are required'}), 400

    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

    conn = sqlite3.connect('users.db')

    cursor = conn.cursor()

    cursor.execute('INSERT INTO users (username, password) VALUES (?, ?)', (username, hashed_password))

    conn.commit()

    conn.close()

    return jsonify({'message': 'Registered successfully'}), 201


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

def login():

    data = request.get_json()

    username = data['username']

    password = data['password']

    if not username or not password:
```

```
        return jsonify({'message': 'Username and password are required'}), 400
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM users WHERE username = ?', (username,))
    user = cursor.fetchone()
    conn.close()

    if user and bcrypt.checkpw(password.encode('utf-8'), user[2]):
        return jsonify({'message': 'Login successful'}), 200
    else:
        return jsonify({'message': 'Invalid credentials'}), 401
if __name__ == '__main__':
    app.run(debug=False)  # Ensure debug mode is off
```

Summary of Changes

1) Parameterized Queries:

Used '?' placeholders for SQL queries to prevent SQL injection.

2) Stronger Password Hashing:

'Replaced generate_password_hash' with 'bcrypt.hashpw' for stronger password hashing.

3) Input Validation:

Added checks for empty username and password.

4) Disabled Debug Mode:

Set debug=False to ensure sensitive information is not exposed.

Conclusion

By implementing these secure coding practices and using tools like static code analyzers (e.g., Bandit for Python), significantly enhance the security of Flask application. Regularly reviewing and updating security measures is crucial to maintaining a secure application.