Task 3-Secure Code Review

Full Name: Eric Thimi Alappatt

Program: CodeAlpha Cybersecurity Internship

Date: 15-03-2024

Language Chosen: Python [FLASK Framework]

Description of Application:

We have a Python Flask application that demonstrates a simple user authentication system. The system allows users to log in, access a dashboard page, and log out. It uses Flask for web development and session management to maintain user login status.

Code:

```
from flask import Flask, render_template, request, redirect, session, url_for

app = Flask(__name__)

app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

# Dummy user database (replace with actual database implementation)

users = {

    'john': 'password123',

    'mary': 'abcxyz',

    'admin': 'admin123'

}



# Login route

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

```
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Check if username and password are valid
        if username in users and users[username] == password:
            # Store username in session to indicate user is logged in
            session['username'] = username
            return redirect(url_for('dashboard'))
        else:
            return render_template('login.html', error='Invalid username or password')

    return render_template('login.html')

# Dashboard route (protected)
@app.route('/dashboard')
def dashboard():
    # Check if user is logged in
    if 'username' in session:
        username = session['username']
        return render_template('dashboard.html', username=username)
    else:
        return redirect(url_for('login'))
```

```
# Logout route

@app.route('/logout')

def logout():

    # Remove username from session to log out user

    session.pop('username', None)

    return redirect(url_for('login'))


if __name__ == '__main__':

    app.run(debug=True)
```

Security Vulnerabilities and Recommendations:-

1. **Insecure Password Storage**:

➔ Vulnerability: Passwords are stored in plaintext in the `users` dictionary, which poses a significant security risk if the database is compromised.

➔ Recommendation: Use secure password hashing techniques like bcrypt to hash passwords before storing them in the database. Libraries like Flask-Bcrypt can be used for this purpose.

2. Session Management Issues:

➔ Vulnerability: Session management is implemented using Flask's `session` object, which relies on the `app.secret_key` for encryption. However, the secret key is not securely generated.

➔ Recommendation: Generate a cryptographically secure secret key and store it in a secure location. Additionally, consider using more advanced session management techniques or libraries like Flask-Session for better security.

3. **Cross-Site Scripting (XSS)**:

➔ Vulnerability: User input is not sanitized before being rendered in templates, which could lead to XSS attacks if an attacker injects malicious scripts.

➔ Recommendation: Sanitize user input using functions like `escape()` or `safe` in Jinja templates to prevent XSS attacks. Additionally, consider using Flask-WTF for form handling, which automatically provides CSRF protection and input validation.

4. Brute Force Attacks:

➔ Vulnerability: There is no protection against brute force attacks on the login endpoint, allowing attackers to repeatedly guess passwords.

➔ Recommendation: Implement rate limiting or CAPTCHA challenges to mitigate brute force attacks. Tools like Flask-Limiter can be used for rate limiting.

5. Sensitive Data Exposure:

➔ Vulnerability: User information such as email addresses is stored in plaintext in the `users` dictionary, which could be exposed in case of a data breach.

➔ Recommendation: Encrypt sensitive user data at rest using appropriate encryption techniques. Consider using a secure database solution with built-in encryption features.

6. Missing Authentication and Authorization Checks:

➔ Vulnerability: There are no checks for strong authentication or authorization. Any user can access the dashboard page by simply navigating to `/dashboard`.

➔ Recommendation: Implement proper authentication mechanisms like password hashing and session management. Additionally, enforce authorization checks to ensure that only authenticated users have access to sensitive resources.

7. Error Handling Issues:

➔ Vulnerability: Error messages might reveal sensitive information about the application's internals, potentially aiding attackers in exploiting vulnerabilities.

➔ Recommendation: Implement custom error handling to provide generic error messages to users while logging detailed errors securely on the server side.

8. Security Headers:

- ➔ Vulnerability: Lack of security headers like:- Content Security Policy (CSP), Strict-Transport-Security (HSTS), or X-Frame-Options leaves the application vulnerable to various attacks like clickjacking, XSS, etc.

- ➔ Recommendation: Set appropriate security headers in the HTTP responses to enhance the security posture of the application. Flask extensions like Flask-Talisman can help in easily configuring security headers.

Hence, By addressing these vulnerabilities and implementing the recommended security practices, the overall security of the Flask application can be significantly improved, reducing the risk of exploitation by malicious actors.