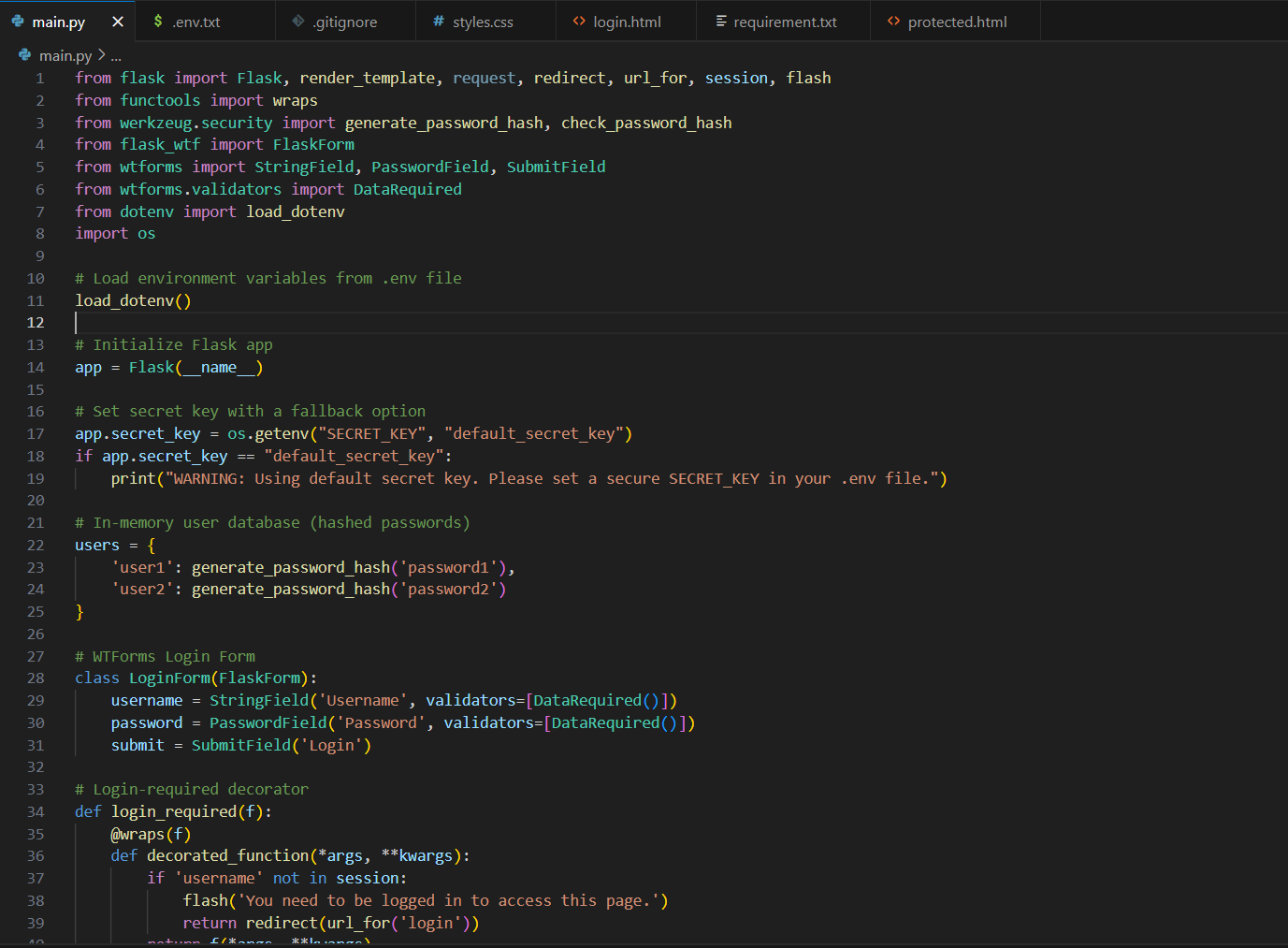
Secure Coding Review Report - Task 3

# 1. Overview

This document presents a secure coding review of a Flask-based web application that implements a login system using Flask-WTF, session management, and form validation. The review includes static analysis and manual inspection to identify security vulnerabilities, followed by recommendations and best practices for safer code.

# 2. Language and Application Audited

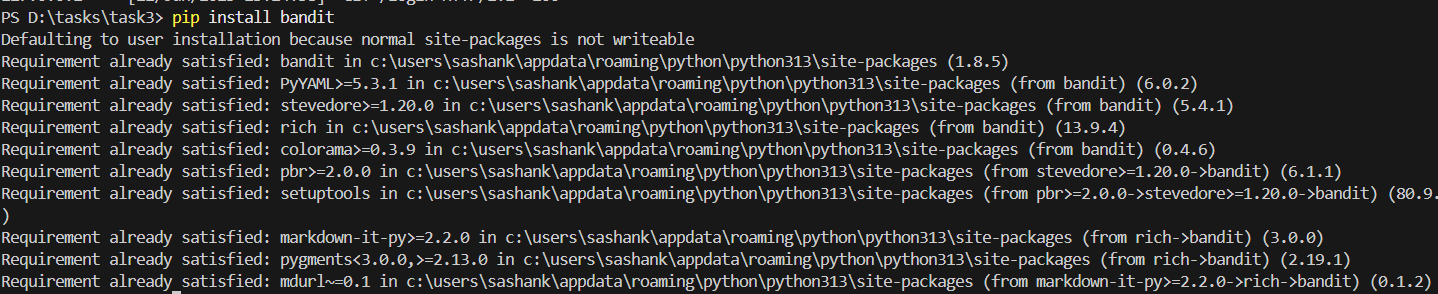
• Language: Python  
• Framework: Flask  
• Application: Login system with session-based user authentication.



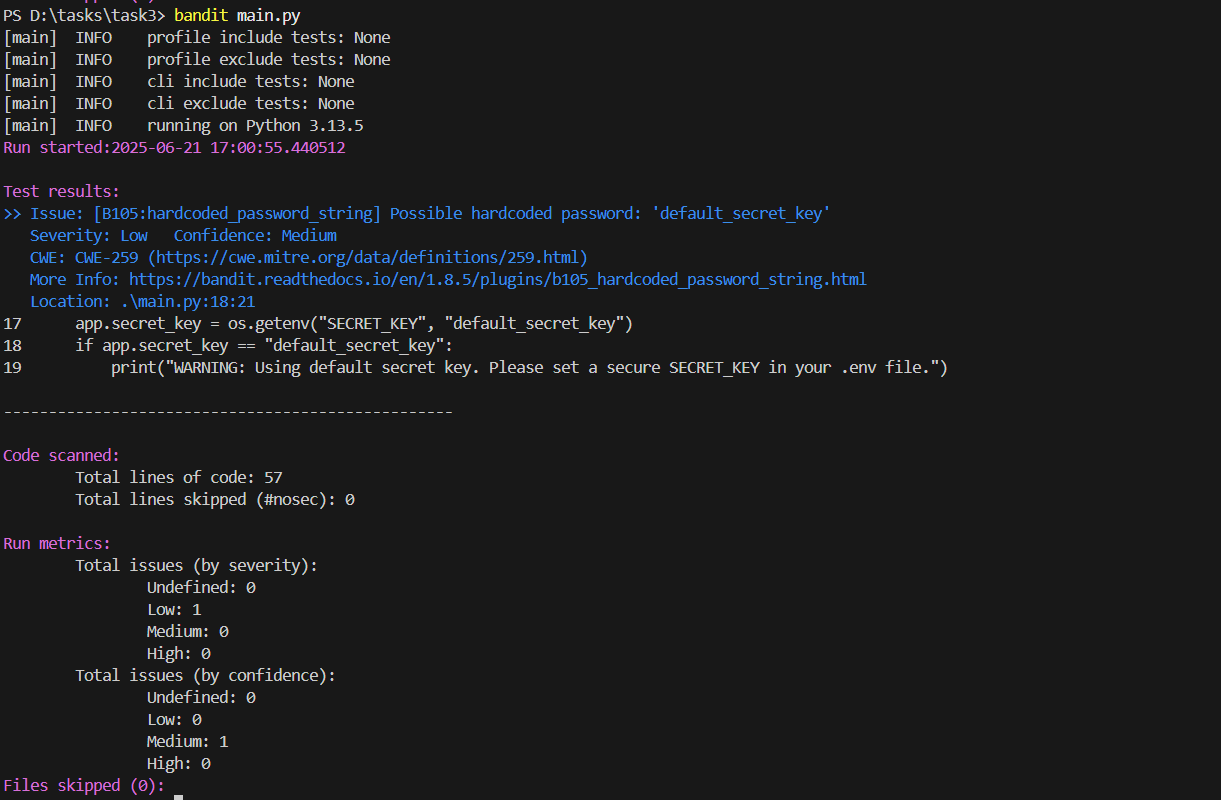
# 3. Tools and Methods Used

• Manual Code Review

• Static Analysis (Bandit - Python security linter)



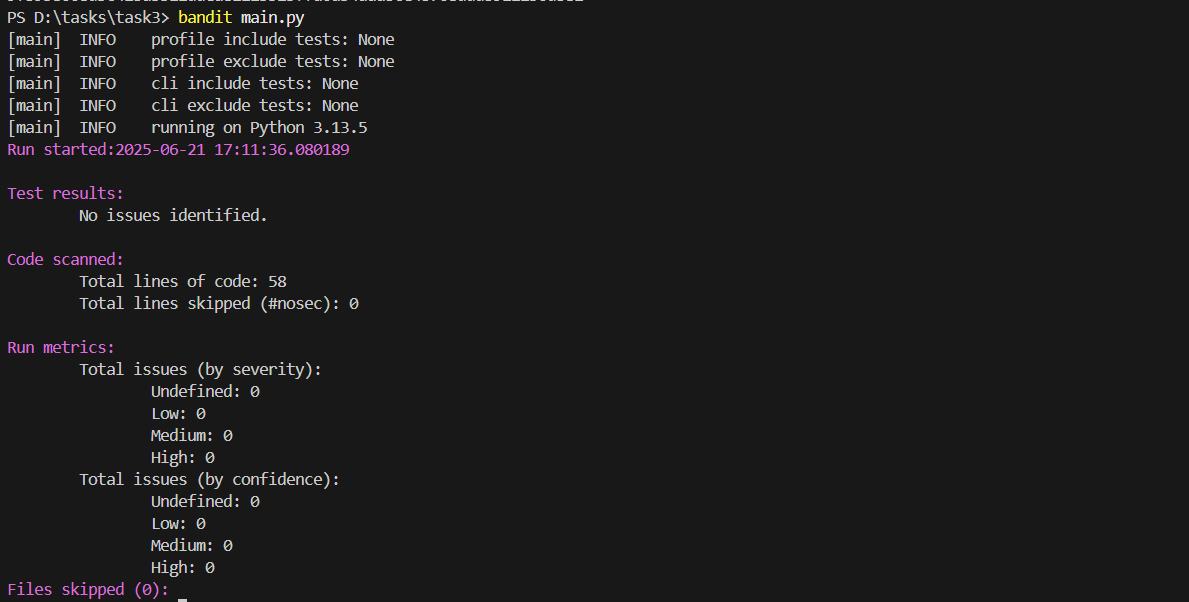
* Analytics in bandit:



it will show issues that are present in your python code.

Here the issue is hardcoded\_passsword.

* After fixing this issue the result is:



• Secure development guidelines from OWASP Top 10

# 4. Identified Security Vulnerabilities

The following security issues were identified in the application:

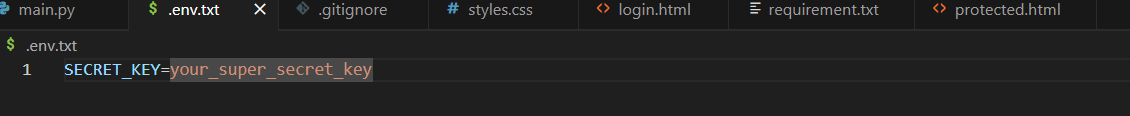
1. \*\*Hardcoded Default Secret Key\*\*:  
 - Risk: Weak default secret key may lead to session hijacking.  
 - Location: `app.secret\_key = os.getenv("SECRET\_KEY", "default\_secret\_key")`

2. \*\*In-Memory User Store\*\*:  
 - Risk: Passwords stored in memory; not suitable for production.  
 - Recommendation: Use a secure database with salted password hashing.

3. \*\*Debug Mode Enabled\*\*:  
 - Risk: Debug mode may expose sensitive data in production.  
 - Location: `app.run(debug=True)`

# 5. Recommendations & Best Practices

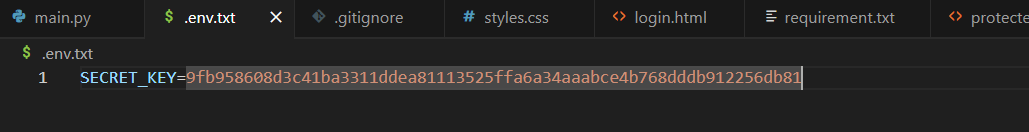
• Always use a strong and unique secret key in production (set in .env).

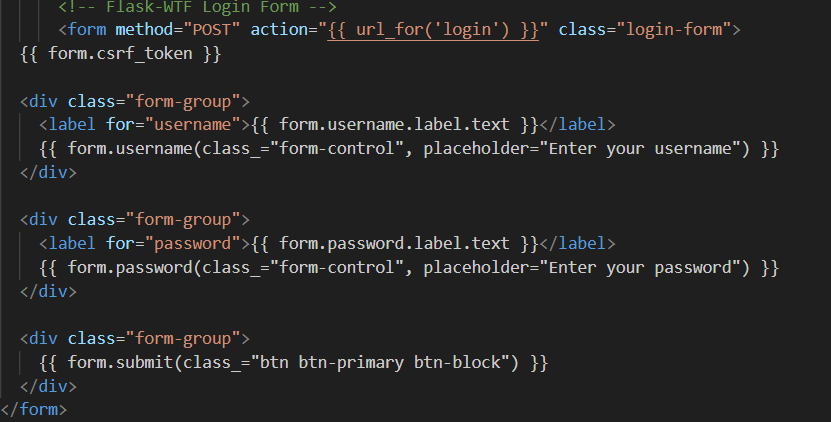


* Use a python code to generate code:
* python -c "import secrets; print(secrets.token\_hex(32))"
* Then it will show some secret key paste it in the .env file.  
  • Use a secure database to manage users, not in-memory dictionaries.  
  • Disable debug mode in production deployments.  
  • Implement rate-limiting to prevent brute-force login attempts.  
  • Ensure HTTPS is used for secure data transmission.

# 6. Remediation Steps

1. Set a secure SECRET\_KEY in the `.env` file and avoid using defaults.

  
2. Integrate a database like SQLite/PostgreSQL and use proper ORM.  
3. Use environment-based configurations to disable debug mode in production.  
4. Add CSRF protection using Flask-WTF (already present, good practice).

  
5. Sanitize inputs and escape output where applicable.

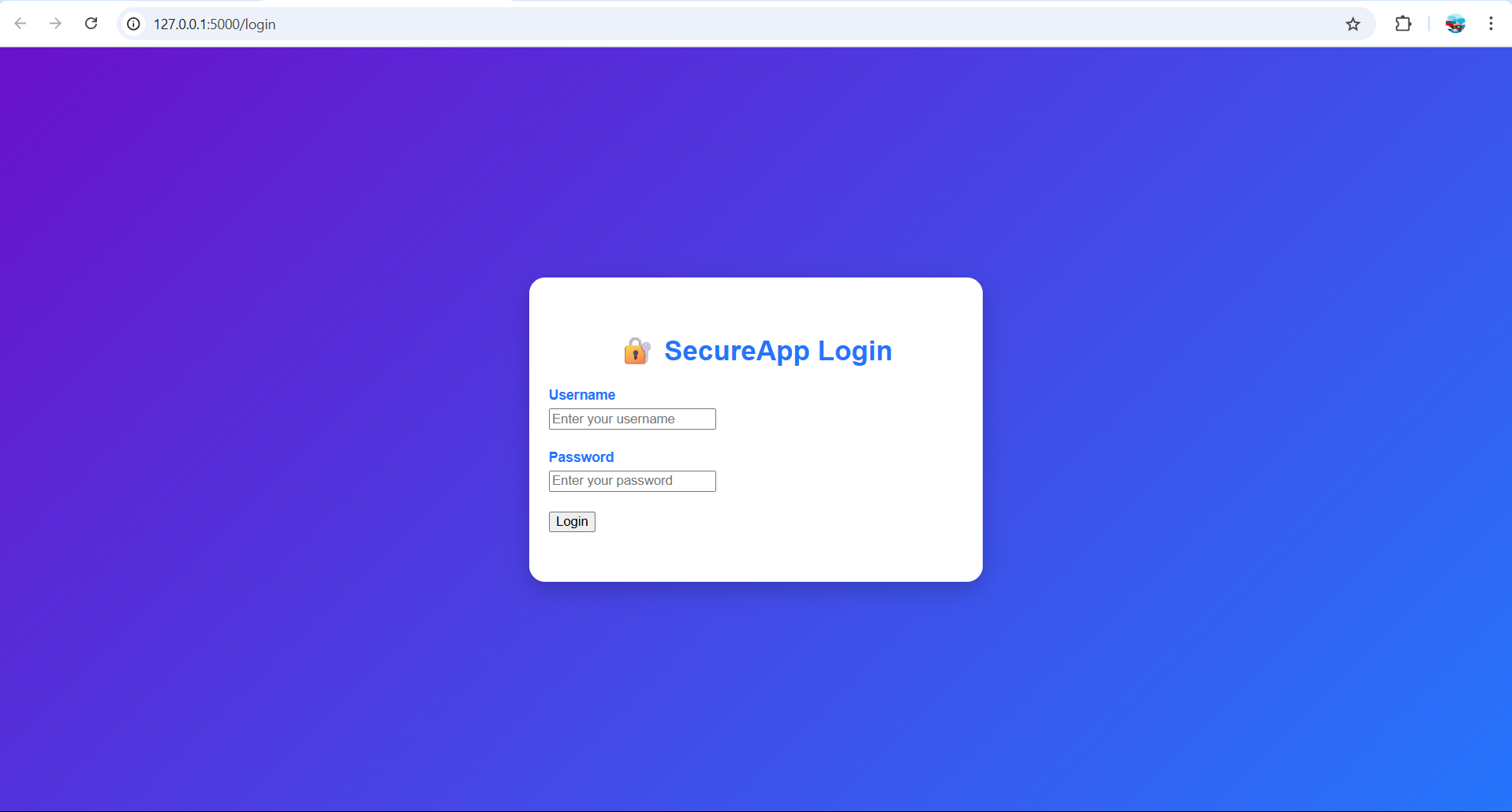
# 7.Output

After successful login with valid credentials, the user is redirected to a protected page that displays a personalized welcome message:  
“Welcome, <username>! You have successfully logged in to a protected page.”  
A logout button is also provided to securely end the session.

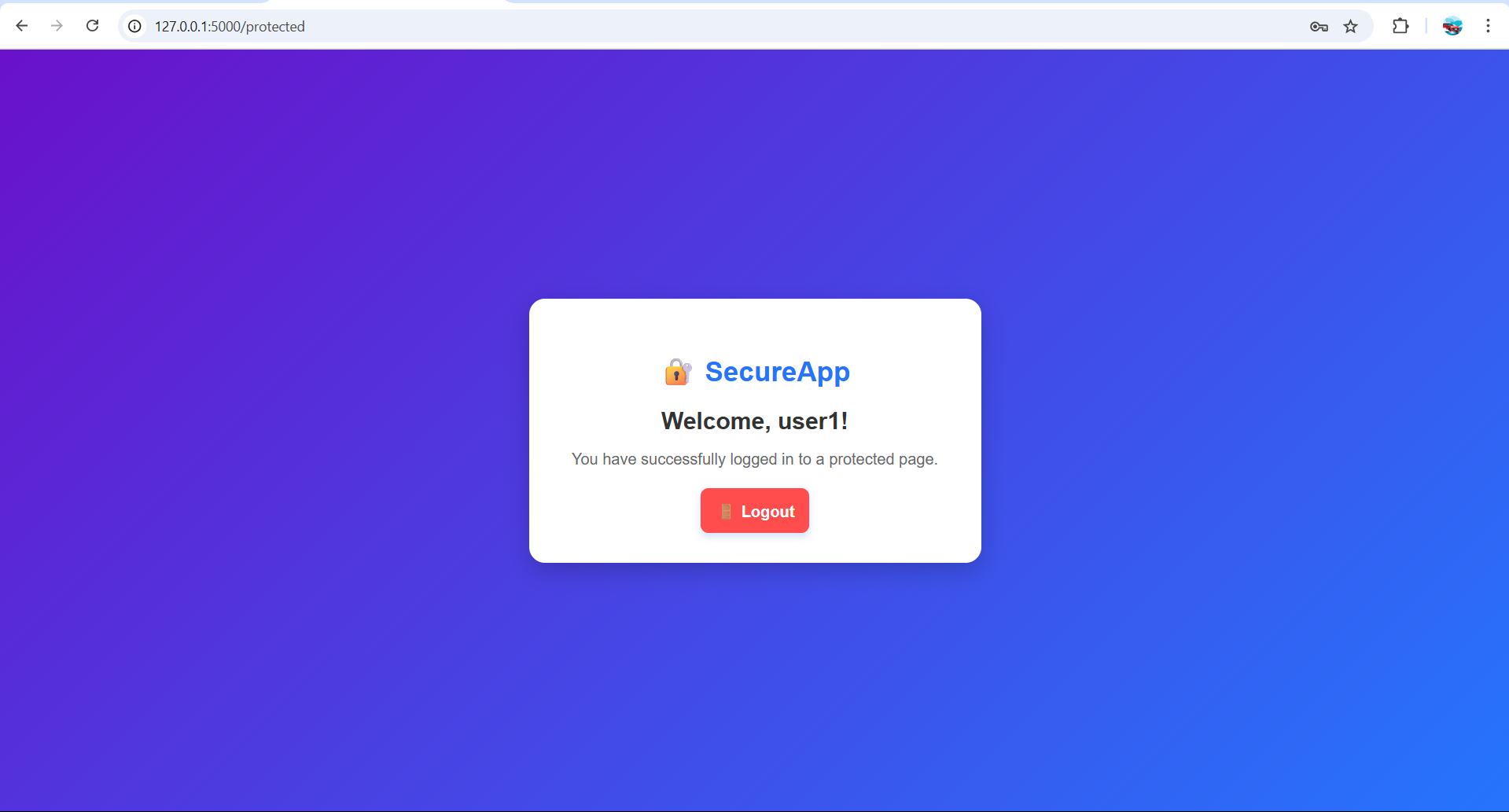
A screen shot of a computer

AI-generated content may be incorrect.

* Output page: Login page



Put the user\_name and password and it will take u to another page:



Then when you will click on logout page it will take you back to login page:

A screenshot of a computer

AI-generated content may be incorrect.

# 8. Conclusion

The application demonstrates good foundational practices in form handling and session security. However, enhancements like proper key management, avoiding in-memory storage, and production-safe configurations are critical to improving its security posture.