Secure Coding Review Report

Summary of Findings

1. SQL Injection - High Risk - Fixed with parameterized queries.

2. Plaintext Passwords - High Risk - Fixed using hashed passwords.

3. Debug Mode Enabled - Medium Risk - Fixed by setting debug=False.

4. No CSRF Protection - Medium Risk - Pending - use Flask-WTF.

5. No Input Validation - Medium Risk - Pending - add input validation.

6. No Rate Limiting - Medium Risk - Pending - use Flask-Limiter.

Recommendations and Best Practices

Authentication & Passwords:

- Use strong password hashing (bcrypt, werkzeug.security).
- Never store passwords in plaintext.
- Enforce password complexity.
- Add rate limiting or CAPTCHA to login forms.

Input Handling:

- Validate and sanitize all user inputs.
- Avoid insecure functions like eval(), exec().
- Use parameterized queries or ORMs like SQLAlchemy.

CSRF & XSS Protection:

- Add CSRF protection using Flask-WTF.
- Escape all template output.
- Set secure HTTP headers (Content-Security-Policy).

Deployment Best Practices:

- Never run in debug mode in production.
- Use environment variables for secrets.
- Regularly update dependencies.

Monitoring & Logging:

- Log failed logins and flag anomalies.
- Avoid logging sensitive data.

Suggested Tools

- Bandit: Python static code analyzer.
- Flake8: Linter for style/code issues.
- Safety: Checks for insecure Python dependencies.
- PyUp: Monitors dependency updates.

Remediation Steps Summary

SQL: Use parameterized queries or SQLAlchemy.

Passwords: Hash and verify using werkzeug.security.

CSRF: Implement Flask-WTF CSRF tokens.

Debug Mode: Set debug=False in production.

Rate Limiting: Use Flask-Limiter.

Validation: Use WTForms or regex checks.

Sample Commands

```
pip install bandit safety flask-limiter flask-wtf

bandit -r app.py

safety check
```

Final Note

Security is not a one-time fix - it's a mindset.

Like Tony Stark said, "You're not just building something. You're building something to last."