Secure Code Review Report

1. Application Reviewed

Flask Web Application with Login Functionality

Language: Python

Tools Used: Bandit (Static Analyzer), Manual Inspection

2. Objective

- Detect vulnerabilities like SQL Injection, XSS, hardcoded credentials, etc.
- Use static analysis and manual methods.
- Recommend secure coding practices.

3. Tools Used

Tool	Purpose
Bandit	Static analysis of Python code
Manual Inspection	Human review of logic and structure

4. Summary of Findings

No	Vulnerability	File/Line	Description	Risk	Recommendation
1	SQL Injection	login.py:25	Raw SQL query with unsanitized user input	High	Use parameterized queries or ORM
2	Hardcoded Secret	config.py:5	API key hardcoded in config	High	Move to environment variables
3	Missing Input Validation	register.py:18	No validation on user input	Medium	Add input checks and sanitization
4	Detailed Error Exposure	app.py:55	Internal errors returned in API responses	Medium	Use generic error messages

5. Detailed Findings

SQL Injection

File: login.py

Issue: Query built using string formatting: query = f"SELECT * FROM users WHERE username = '{username}'"

Risk: High

Fix: Use SQLAlchemy with parameterized queries.

Hardcoded Secrets

File: config.py

Issue: SECRET_KEY = 'mysecret'

Risk: High

Fix: Use a .env file or secret manager.

Missing Input Validation

File: register.py

Issue: User inputs directly used without checks

Risk: Medium

Fix: Add input validation using WTForms or Marshmallow.

Detailed Error Exposure

File: app.py

Issue: return str(e) returns full exception trace

Risk: Medium

Fix: Log error internally and return a generic message.

6. Best Practices for Secure Coding

- Validate all inputs (length, format, type)
- Use ORM for database queries (avoid raw SQL)
- Store secrets securely using environment variables
- Use HTTPS for all communication
- Implement secure authentication (bcrypt, Flask-Login)

- Set HTTP security headers
- Sanitize outputs to prevent XSS
- Keep dependencies up to date

7. Recommendations and Remediation

- Refactor code to remove all hardcoded secrets
- Apply parameterized queries throughout the app
- Implement input validation framework-wide
- Configure proper logging and error handling
- Run Bandit regularly as part of CI pipeline

8. Conclusion

This review uncovered multiple critical and medium-severity security issues.
Remediation steps have been outlined to guide the development team toward a more secure codebase.
Following secure coding practices consistently will greatly reduce the application's attack surface.

9. References

- https://owasp.org/www-project-secure-coding-practices/
- https://bandit.readthedocs.io/en/latest/
- https://flask.palletsprojects.com/en/latest/security/