Secure Code Review

1. Selected Language and Application

- **Programming Language:** Python
- **Application Type:** Web Application (Flask-based REST API)
- **Use Case:** User authentication and data management system
- **Reason for Selection:** Python web apps are widely used and frequently targeted due to common issues like injection flaws, insecure authentication, and misconfigurations.

2. Review Methodology

2.1 Review Techniques Used

- **Manual Code Inspection**
    - Reviewed authentication logic, database access, input handling, and error management.
- **Static Code Analysis**
    - Used tools such as:
        - Bandit (Python security linter)
        - Flake8 (code quality & linting)
- **Threat Modeling**
    - Identified possible attack vectors: SQL injection, authentication bypass, privilege escalation.

3. Identified Security Vulnerabilities

3.1 SQL Injection

Issue:
User input is directly concatenated into SQL queries.

query = f"SELECT * FROM users WHERE username = '{username}'"

Risk:
Attackers can manipulate queries to access or modify unauthorized data.

3.2 Hardcoded Credentials

Issue:
Database credentials are stored directly in the source code.

DB_PASSWORD = "admin123"

Risk:
If the repository is exposed, attackers gain full database access.

3.3 Weak Password Storage

Issue:
Passwords are stored using plain text or weak hashing (e.g., MD5).

hashlib.md5(password.encode()).hexdigest()

Risk:
Easily cracked if the database is compromised.

3.4 Lack of Input Validation

Issue:
User inputs are not validated or sanitized.

email = request.form['email']

Risk:
Enables XSS, injection attacks, and malformed input exploitation.

3.5 Insufficient Error Handling

Issue:
Detailed error messages are returned to the client.

return str(e)

Risk:
Reveals internal logic, file paths, or database structure.

3.6 Missing Authentication and Authorization Controls

Issue:
Sensitive endpoints lack access control.

```
@app.route('/admin')
def admin_panel():
    return "Admin Area"
```

Risk:
Unauthorized users can access privileged functionality.

4. Security Tools Findings (Static Analysis Summary)

Tool	Finding	Severity
Bandit	Hardcoded password detected	High
Bandit	SQL injection vector	High
Bandit	Use of weak cryptographic hash	Medium
Flake8	Poor exception handling	Low

5. Recommendations and Best Practices

5.1 Secure Database Access

- Use **parameterized queries / ORM**

```
cursor.execute("SELECT * FROM users WHERE username = %s", (username,))
```

5.2 Secure Credential Management

- Store secrets in:
    - Environment variables
    - Secret managers (Vault, AWS Secrets Manager)

5.3 Strong Password Hashing

- Use modern algorithms:

- bcrypt
- Argon2

bcrypt.hashpw(password.encode(), bcrypt.gensalt())

5.4 Input Validation and Sanitization

- Enforce:
  - Type checking
  - Length limits
  - Whitelisting
- Use libraries like `WTForms` or `pydantic`.

5.5 Proper Error Handling

- Log detailed errors internally.
- Show generic messages to users:

return "An unexpected error occurred", 500

5.6 Authentication and Authorization

- Use:
  - Role-Based Access Control (RBAC)
  - Token-based authentication (JWT)
- Protect sensitive routes with decorators.

6. Remediation Plan

Vulnerability	Remediation Action
SQL Injection	Use parameterized queries
Hardcoded Credentials	Move secrets to environment variables
Weak Password Hashing	Implement bcrypt or Argon2
Input Validation	Add centralized validation logic
Error Information Leakage	Implement secure error handling
Missing Access Controls	Enforce authentication & authorization

7. Conclusion

This secure coding review identified multiple **high-risk vulnerabilities** commonly found in Python web applications. By applying secure coding best practices, using modern cryptographic standards, and integrating automated security tools into the development lifecycle, the application's security posture can be significantly improved.

Key takeaway: Secure coding is not a one-time task—it must be integrated into development, testing, and deployment phases.