TASK 5 – Secure Code Review

Here’s an example of a Python code snippet containing multiple security vulnerabilities. I'll break down the vulnerabilities and provide recommendations for mitigating each one.

Vulnerable Code Example

```
import sqlite3
from flask import Flask, request, jsonify
import hashlib
import os

app = Flask(__name__)

# Hardcoded database credentials (Insecure)
DATABASE = 'app.db'

# Vulnerable login function
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')

    # Password hashing with MD5 (Insecure)
    hashed_password = hashlib.md5(password.encode()).hexdigest()

    # SQL Injection vulnerability due to unparameterized query
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{hashed_password}'"
    cursor.execute(query)

    user = cursor.fetchone()
    conn.close()

    if user:
        return jsonify({"message": "Login successful!"}), 200
    else:
        return jsonify({"message": "Invalid credentials"}), 401

# Exposed API for deleting user account
@app.route('/delete_user', methods=['POST'])
def delete_user():
    user_id = request.form.get('user_id')

    # No authorization check, allowing unauthorized account deletion
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute(f"DELETE FROM users WHERE id = {user_id}")
    conn.commit()
    conn.close()

    return jsonify({"message": "User deleted"}), 200

if __name__ == "__main__":
    app.run(debug=True)
```

Identified Vulnerabilities and Recommendations

1. Hardcoded Database Credentials

Issue: The DATABASE variable is hardcoded in the code, which is insecure, especially for production environments where database configurations should not be exposed in source code.

Recommendation: Use environment variables to store sensitive information such as database credentials. In Python, you can use the os module to fetch environment variables, e.g.,

```
DATABASE = os.getenv('DATABASE')
```

2. Insecure Password Hashing (MD5)

Issue: The application uses MD5 to hash passwords, which is insecure due to MD5's vulnerability to brute-force and collision attacks.

Recommendation: Use a stronger hashing algorithm, such as bcrypt or Argon2, designed for password hashing. Libraries like bcrypt make it easy to implement this securely:

```
import bcrypt

hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
```

3. SQL Injection (Unparameterized SQL Query)

Issue: The login function uses unparameterized SQL queries with direct input from the user, making it vulnerable to SQL injection.

Recommendation: Use parameterized queries (prepared statements) to prevent SQL injection. Here’s a safer way to execute the query:

```
query = "SELECT * FROM users WHERE username = ? AND password = ?"

cursor.execute(query, (username, hashed_password))
```

4. Missing Authentication/Authorization Check for Sensitive Action

Issue: The /delete_user endpoint allows user account deletion without any authentication or authorization checks, meaning any user could delete any account.

Recommendation: Implement authentication and role-based access control (RBAC) to ensure only authorized users can delete accounts. Use session tokens or JWTs for authentication and verify the user's permissions:

```
if not current_user.is_admin:

    return jsonify({"error": "Unauthorized"}), 403
```

5. Debug Mode Enabled in Production

Issue: Running the application in debug mode (app.run(debug=True)) is insecure for production as it can expose sensitive information if an error occurs.

Recommendation: Disable debug mode in production by setting debug=False or by using an environment variable to control the debug setting based on the environment.

Revised Code (With Security Fixes)

I would like to show here a more secure version of the initial code snippet with the recommended mitigations applied:

```
import sqlite3
from flask import Flask, request, jsonify
import bcrypt
import os

app = Flask(__name__)

# Securely load database credentials from environment variables
DATABASE = os.getenv('DATABASE')

# Secure login function
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')

    # Password hashing with bcrypt
    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    query = "SELECT password FROM users WHERE username = ?"
    cursor.execute(query, (username,))
    stored_password_hash = cursor.fetchone()

    conn.close()

    if stored_password_hash and bcrypt.checkpw(password.encode(), stored_password_hash[0].encode()):
        return jsonify({"message": "Login successful!"}), 200
    else:
        return jsonify({"message": "Invalid credentials"}), 401

# Secure user deletion endpoint with authorization check
@app.route('/delete_user', methods=['POST'])
def delete_user():
    if not request.user.is_authenticated or not request.user.is_admin:
        return jsonify({"error": "Unauthorized"}), 403

    user_id = request.form.get('user_id')

    conn = sqlite3.connect(DATABASE)
    cursor = conn.cursor()
    cursor.execute("DELETE FROM users WHERE id = ?", (user_id,))
    conn.commit()
    conn.close()

    return jsonify({"message": "User deleted"}), 200

if __name__ == "__main__":
```

```
# Disable debug mode for production

app.run(debug=False)
```

This version mitigates the identified vulnerabilities by:

- ✓ Using environment variables for sensitive configurations.
- ✓ Implementing bcrypt for secure password hashing.
- ✓ Securing SQL queries with parameterized statements.
- ✓ Adding an authorization check for sensitive operations.
- ✓ Disabling debug mode for production readiness.

These changes provide a more secure foundation and significantly reduce the risk of common vulnerabilities.