# Secure Code Review

To review a provided codebase for security issues, we need to follow a systematic approach. Here's a small sample codebase to analyze. It contains a basic web application in Python using Flask:

```python
# app.py
from flask import Flask, request, render_template, redirect, url_for, session
import sqlite3

app = Flask(__name__)
app.secret_key = 'supersecretkey'

# Database connection
def get_db_connection():
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

# Home Route
@app.route('/')
def home():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return render_template('login.html')

# Login Route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        conn = get_db_connection()
        user = conn.execute('SELECT * FROM users WHERE username = ? AND password = ?',
(username, password)).fetchone()
        conn.close()
        if user:
            session['username'] = username
            return redirect(url_for('home'))
        else:
            return 'Invalid credentials!'
    return render_template('login.html')

# Sign Up Route
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        conn = get_db_connection()
        conn.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, password))
        conn.commit()
        conn.close()
        return redirect(url_for('login'))
    return render_template('signup.html')

# Logout Route
@app.route('/logout')
def logout():
    session.pop('username', None)
    return redirect(url_for('home'))

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

The provided Flask application (`app.py`) contains several security issues that need to be addressed to enhance the overall security posture of the application. Below, I will outline the security vulnerabilities present in the code and provide recommendations for remediation.

Security Issues and Recommendations

1. Password Storage Issue: The application stores passwords in plaintext in the database. If an attacker gains access to the database, they can easily read user passwords.

Recommendation: Use a secure hashing library like `bcrypt` to hash passwords before storing them. This way, even if the database is compromised, the passwords remain protected.

```
from flask import Flask, request, render_template, redirect, url_for, session
import sqlite3
import bcrypt

# Sign Up Route
@app.route('/signup', methods=['GET', 'POST'])
def signup():
if request.method == 'POST':
username = request.form['username']
password = request.form['password']
```

```
hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
conn = get_db_connection()
conn.execute('INSERT INTO users (username, password) VALUES (?, ?)', (username,
hashed_password))
conn.commit()
conn.close()
return redirect(url_for('login'))
return render_template('signup.html')

# Login Route
@app.route('/login', methods=['GET', 'POST'])
def login():
if request.method == 'POST':
username = request.form['username']
password = request.form['password']
conn = get_db_connection()
user = conn.execute('SELECT * FROM users WHERE username = ?',
(username,)).fetchone()
conn.close()
if user and bcrypt.checkpw(password.encode('utf-8'), user['password'].encode('utf-
8')):
session['username'] = username
return redirect(url_for('home'))
else:
return 'Invalid credentials!'
return render_template('login.html')
```

2. SQL Injection Issue: Although the current code uses parameterized queries, the login process is susceptible to SQL injection due to the way the query is structured.

Recommendation: Always use parameterized queries for all SQL statements, and never concatenate user input into SQL commands. The current implementation is good in this aspect but ensure it is consistently applied everywhere, especially during sign-up.

3.Session Management Issue: The session secret key is hardcoded and not secure. If someone were to discover this key, they could hijack user sessions.

Recommendation: Use a securely generated key and store it in an environment variable instead of hardcoding it:

```
import os

app.secret_key = os.environ.get('SECRET_KEY', 'defaultsecretkey') # Use a strong
default and prioritize environment variable
```

4. Cross-Site Request Forgery (CSRF) Issue: The application does not have any protection against CSRF attacks.

Recommendation: Use Flask-WTF or similar libraries that provide CSRF protection. You can add CSRF tokens to your forms as follows: 2

```
bash
pip install Flask-WTF
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect(app)

# In your HTML forms
<form method="POST">
{{ csrf_token() }} <!-- Add CSRF token -->
<input type="text" name="username">
<input type="password" name="password">
<input type="submit">
</form>
```

5. Error Handling Issue: The application return plain text messages for invalid credentials and does not handle exceptions gracefully. This could potentially reveal information about the system.

Recommendation: Create a custom error handler that returns generic error messages. Avoid revealing whether the username or password is incorrect.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
if request.method == 'POST':
username = request.form['username']
password = request.form['password']
conn = get_db_connection()
user = conn.execute('SELECT * FROM users WHERE username = ?',
(username,)).fetchone()
conn.close()
```

```
if user and bcrypt.checkpw(password.encode('utf-8'), user['password'].encode('utf-
8')):
session['username'] = username
return redirect(url_for('home'))
else:
return 'Invalid credentials!', 401 # Use status code for unauthorized
return render_template('login.html')
```

6. Debug Mode Issue: The application is running in debug mode, which can expose sensitive information and stack traces in the event of an error.

Recommendation: Set `debug=False` in production environments.

```
if __name__ == '__main__':
app.run(debug=False) # Set to False in production
```

7. Database Connection Management Issue: The current implementation opens a new database connection for every request but does not handle exceptions or close connections properly.

Recommendation: Utilize context managers or implement error handling to ensure connections are closed properly. Consider using a connection pool for better performance.

Conclusion:

By addressing the above issues, we can significantly enhance the security of our Flask application. Implementing these changes will help protect user data, secure sessions, and prevent common web vulnerabilities.