For this task, we'll choose Python as the programming language and a simple web application as the application type. We'll conduct a secure coding review using both manual inspection and a static code analysis tool. We'll look at common vulnerabilities such as injection flaws, insecure handling of user input, improper error handling, and insufficient logging.

Example Python Web Application

Let's review a small Python Flask web application for security vulnerabilities. Here is a sample code snippet:

```
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)

# Route to handle user login
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # SQL Injection vulnerability
        conn = sqlite3.connect('database.db')
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM users WHERE username='{username}' AND
password='{password}'")
        user = cursor.fetchone()

        if user:
            return 'Logged in successfully'
        else:
            return 'Invalid credentials'

    return render_template_string('''<form method="POST">
                                        Username: <input type="text"
name="username"><br>
                                        Password: <input type="password"
name="password"><br>
                                        <input type="submit" value="Login">
                                     </form>''')

# Route to display user profile
@app.route('/profile/<username>')
def profile(username):
    return f'Welcome, {username}!'

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

Manual Code Review

Identified Security Issues:

1. **SQL Injection**: The `login` route constructs an SQL query by directly embedding user inputs:

- `cursor.execute(f"SELECT * FROM users WHERE username='{username}' AND password='{password}'")`

This is vulnerable to SQL injection attacks.

- **Cross-Site Scripting (XSS)**: The `profile` route directly includes user input in the response:
- `return f'Welcome, {username}!'`

This is vulnerable to XSS attacks.

- **Sensitive Data Exposure**: The application runs in debug mode, which can expose sensitive information:

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

4. **Improper Error Handling**: There is no proper error handling in the database connection and query execution, which can lead to information leakage.

Recommendations for Secure Coding Practices

1. **Prevent SQL Injection**: Use parameterized queries or an ORM (Object Relational Mapping) tool:

- `cursor.execute("SELECT * FROM users WHERE username=? AND password=?", (username, password))`

- **Prevent XSS**: Use Flask's built-in escaping or a template engine that escapes content by default:

```
return render_template('profile.html', username=username)
```

In `profile.html`:

- `<p>Welcome, {{ username }}!</p>`

- **Disable Debug Mode in Production**: Ensure that the application does not run in debug mode in a production environment:

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

- **Proper Error Handling**: Add try-except blocks to handle database connection and query execution errors:

```
4. try:
       conn = sqlite3.connect('database.db')
       cursor = conn.cursor()
       cursor.execute("SELECT * FROM users WHERE username=? AND password=?",
   (username, password))
       user = cursor.fetchone()
   except sqlite3.Error as e:
       return 'Database error: ' + str(e)
   finally:
       conn.close()
```

Static Code Analysis

Use a tool like Bandit to perform static code analysis. Bandit is designed to find common security issues in Python code.

Installation and Usage:

1. **Install Bandit**:

- `pip install bandit`

- **Run Bandit on the application code**:

2. `bandit -r /path/to/your/application`

Sample Bandit Report:

```
[main]  INFO    running on Python 3.8.10
Run started:2024-05-26 12:34:56.789012

Test results:
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call with
shell=True identified, security issue.
   Severity: High   Confidence: High
   Location: /path/to/your/application.py:34
   More Info:
https://bandit.readthedocs.io/en/latest/plugins/b602_subprocess_popen_with_shell_eq
uals_true.html
33          # subprocess vulnerability example
34          subprocess.Popen('ls -l', shell=True)

--------------------------------------------------

>> Issue: [B101:assert_used] Use of assert detected. The enclosed code will be
removed when compiling to optimised byte code.
   Severity: Medium   Confidence: High
   Location: /path/to/your/application.py:40
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b101_assert_used.html
39          # insecure assert statement example
40          assert user == 'admin'

--------------------------------------------------

Code scanned:
```

```
        Total lines of code: 100
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
                Medium: 1
                High: 1
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 2

Files skipped (0):
```

Conclusion

By conducting a manual code review and using a static code analysis tool like Bandit, we identified and mitigated several security vulnerabilities in the Python Flask application. Adopting secure coding practices such as using parameterized queries, disabling debug mode in production, escaping user inputs, and handling errors properly helps enhance the security of the application.