**SECURE CODING REVIEW**

**What is secure code review?**

Secure Code Review is the process of manually or automatically examining source code to identify and fix security vulnerabilities before the software is deployed. It's an essential practice in software development that helps ensure the application is not just functional but also secure against cyber threats.

**Purpose of Secure Code Review:**

Detect coding flaws that may lead to security issues like SQL Injection, Cross-Site Scripting (XSS), buffer overflows, etc.

Ensure the code follows secure coding standards and best practices.

Reduce the risk of exploitable vulnerabilities in production.

**Types of Secure Code Review:**

**Manual Code Review** – Performed by experienced developers/security experts line-by-line.

**Automated Code Review** – Uses tools like SonarQube, Fortify, or Checkmarx to scan for common vulnerabilities.

**Common Vulnerabilities Looked For:**

* Input validation issues
* Authentication and authorization flaws
* Insecure data storage
* Improper error handling
* Use of weak cryptography

**Benefits:**

* Improves overall code quality
* Reduces cost of fixing bugs (cheaper to fix in development than in production)
* Increases software reliability and user trust

Let’s go with Python and review a simple Flask web application (a common micro web framework). I’ll manually review the code for security vulnerabilities and then provide recommendations and best practices. Here’s a small sample application:

**Sample Flask Application (Code to Review):**

from flask import Flask, request, render\_template\_string

import sqlite3

app = Flask(\_\_name\_\_)

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

username = request.form['username']

password = request.form['password']

conn = sqlite3.connect('users.db')

cursor = conn.cursor()

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

cursor.execute(query)

user = cursor.fetchone()

if user:

return f"Welcome {username}!"

else:

return "Invalid credentials!"

return '''

<form method="post">

Username: <input name="username"><br>

Password: <input name="password" type="password"><br>

<input type="submit" value="Login">

</form>

'''

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Security Vulnerabilities Identified:**

**SQL Injection**

The query string directly inserts user input without sanitization.

A malicious user can input: ' OR '1'='1 to bypass login.

No Password Hashing

Storing and comparing plaintext passwords is dangerous.

**XSS Vulnerability**

render\_template\_string() is not used safely, though not in this sample, it's often a risk if untrusted input is rendered.

No Input Validation/Sanitization

Inputs are used as-is.

Debug Mode in Production

app.run(debug=True) should not be used in production.

**Recommendations for Secure Coding Practices:**

Use Parameterized Queries (Prepared Statements):

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

Hash Passwords Using a Library like bcrypt:

import bcrypt

hashed\_pw = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

And compare using:

bcrypt.checkpw(entered\_pw.encode(), stored\_hashed\_pw)

Validate User Input: Use libraries like WTForms or manually check for valid input format.

Avoid Debug Mode in Production:

app.run(debug=False)

Use a Template Engine Safely: Avoid using render\_template\_string() unless necessary. Use render\_template() with sanitized context.

Static Code Analysis Tools (Recommended):

Bandit – Security linter for Python.

SonarQube – Comprehensive static code analysis.

Flake8 + Plugins – Can help catch insecure code patterns.

**PHP example**

Let's go through a PHP web application example with common vulnerabilities. I'll review the code, highlight the security issues, and provide secure coding recommendations.

**Sample PHP Login Script (Vulnerable Code)**

<?php

$conn = new mysqli("localhost", "root", "", "users\_db");

if ($\_SERVER["REQUEST\_METHOD"] == "POST") {

$username = $\_POST["username"];

$password = $\_POST["password"];

$sql = "SELECT \* FROM users WHERE username = '$username' AND password = '$password'";

$result = $conn->query($sql);

if ($result->num\_rows == 1) {

echo "Welcome, $username!";

} else {

echo "Invalid credentials!";

}

}

?>

<form method="post">

Username: <input name="username"><br>

Password: <input name="password" type="password"><br>

<input type="submit" value="Login">

</form>

**Security Vulnerabilities Identified:**

**SQL Injection**

User input is directly inserted into SQL query.

A malicious input like ' OR '1'='1 can bypass login.

Plaintext Passwords

Passwords are stored and compared as plain text.

No Input Validation or Sanitization

Inputs are blindly trusted.

Cross-Site Scripting (XSS)

Echoing user input directly (e.g., echo "Welcome, $username!") can allow XSS if username contains script.

Secure Version with Fixes and Best Practices

<?php

$conn = new mysqli("localhost", "root", "", "users\_db");

if ($\_SERVER["REQUEST\_METHOD"] == "POST") {

$username = trim($\_POST["username"]);

$password = $\_POST["password"];

// Use prepared statements to prevent SQL injection

$stmt = $conn->prepare("SELECT password FROM users WHERE username = ?");

$stmt->bind\_param("s", $username);

$stmt->execute();

$stmt->store\_result();

if ($stmt->num\_rows == 1) {

$stmt->bind\_result($hashedPassword);

$stmt->fetch();

// Verify password using password\_verify()

if (password\_verify($password, $hashedPassword)) {

echo "Welcome, " . htmlspecialchars($username);

} else {

echo "Invalid credentials!";

}

} else {

echo "Invalid credentials!";

}

}

?>

<form method="post">

Username: <input name="username" required><br>

Password: <input name="password" type="password" required><br>

<input type="submit" value="Login">

</form>

**Secure Coding Recommendations:**

Always use Prepared Statements (mysqli or PDO)

**Prevents SQL Injection.**

1. Use password\_hash() and password\_verify() for Passwords
2. Never store passwords in plaintext.
3. Validate and Sanitize Input
4. Use filter\_input(), trim(), and regex where appropriate.
5. Use htmlspecialchars() when echoing user input to prevent XSS.
6. Enable HTTPS
7. Protect data in transit.
8. Use Error Logging Instead of Displaying Errors to Users
9. Never show raw error messages on a live app.

**Static Code Analysis Tools for PHP:**

PHPStan – Static analysis for detecting code issues.

Psalm – Finds security and type-related issues.

RIPS (now ShiftLeft) – PHP-specific security scanner.