# CS 405 Project Two Script Template

Gabriel Walls

6/20/2025

YouTube Video Link: https://youtu.be/Lh114qmvKjY

Complete this template by replacing the bracketed text with the relevant information.

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone. My name is Gabriel Walls, and this is my security policy presentation for Green Pace. Today I’ll be walking you through our newly standardized secure development policy. This presentation is designed to align our team around best practices for coding, architecture, and vulnerability management. |
| **2** | Let’s start with the big picture. Our new security policy formalizes the coding practices we’ve already been using and ensures they’re repeatable and enforceable across the team. This layered approach gives us multiple safety nets. |
| **3** | Here we refer to a threat matrix that evaluates our 10 core coding standards. Each was scored on severity, likelihood, remediation cost, and overall priority. This helps us quickly identify which vulnerabilities pose the greatest threat so we can prioritize mitigation efforts. For example, memory leaks and SQL injection are both high-risk and should be handled early. |
| **4** | Our policy is grounded in ten core principles. Each one reinforces secure behavior in development. For example, the principle of "Validate Input Data" connects directly to several standards like bounds checking and safe file access. Others, like "Least Privilege" and "Defense in Depth," help ensure that we limit exposure and add redundancy where it counts. By tying our coding standards to these principles, we keep our practices aligned with widely accepted security philosophy. |
| **5** | We’ve organized our 10 standards based on risk and impact. At the top of our list are SQL injections and string handling because they’re both highly exploitable and relatively easy to overlook. Lower-risk practices like exception handling are still important, but they’re less likely to be the source of a major breach. This prioritization helps developers focus efforts where they’ll have the biggest impact. |
| **6** | Encryption plays a major role in our policy. We divide it into three categories: at rest, in flight, and in use. At rest, we use AES-256 for database and file system protection. In flight, all traffic uses TLS 1.2 or higher. And for data in use — such as in memory — we apply secure enclaves for our most sensitive processes. Each type protects data in different contexts, forming a full coverage approach. |
| **7** | The Triple-A framework includes authentication, authorization, and accounting. We require multi-factor authentication and SSO integration for access control. RBAC ensures users only access what they need, and logs are tracked centrally through our SIEM. This isn’t just about access — it’s about knowing who did what and when. |
| **8** | For this section, we focused on four critical C++ coding vulnerabilities: type misuse, unchecked input, unsafe string handling, and SQL injection.  For each one, we built unit tests using Visual Studio’s framework to simulate both safe and unsafe inputs. These tests were designed to trigger common vulnerabilities—like buffer overflows or improper data access—and validate whether our safeguards were effective.  Positive tests confirmed the code behaved securely under normal conditions, while negative tests intentionally introduced unsafe scenarios to verify our protections would catch them.  This process not only confirmed that our standards are working as intended but also demonstrated how unit testing can play a powerful role in preventing vulnerabilities before deployment. |
| **9-12** | **STD-001-CPP: Use Explicit and Appropriate Data Types:** This unit test checks whether implicit data types are avoided in favor of explicitly defined types like uint64\_t. It simulates integer overflow scenarios and validates that the appropriate type is used to store large values safely.  **STD-002-CPP: Validate and Constrain Data Values:** This unit test introduces invalid index values to arrays and expects the application to deny access outside the array bounds. The test passes only if index validation logic correctly prevents out-of-bounds access.  **STD-003-CPP: Ensure Safe and Bounded String Operations:** This test intentionally copies long strings into small buffers. The application should use strncpy or std::string. If a buffer overflow occurs, the test fails, signaling unsafe string handling.  **STD-004-CPP: Prevent SQL Injection Using Parameterized Queries:** This test injects malicious strings into SQL query construction. If the system uses parameterized queries, it should treat them as data. If the database executes injected commands, the test fails. |
| **13** | In our DevSecOps pipeline, security automation happens at multiple points. Static analysis tools run after each code commit and again during nightly builds. These catch policy violations early. We’ve designed the process so that a build can’t proceed unless critical vulnerabilities are addressed — ensuring that security is baked into the workflow. |
| **14** | In our DevSecOps pipeline, we use tools like Cppcheck and Clang-Tidy during the coding and build phases to catch issues like unsafe data types and memory errors early. Fortify SCA is used during testing to detect deeper vulnerabilities through static analysis. As code moves toward deployment, SonarQube provides ongoing quality and security checks. All of these tools are integrated into our CI/CD pipeline to automatically scan with each commit or pull request, ensuring that no insecure code is promoted to production. |
| **15** | Implementing this policy has both short- and long-term benefits. On the short side, we reduce our exposure to well-known vulnerabilities like SQL injection and memory corruption. On the long side, we establish a scalable, maintainable foundation for secure growth. Some fixes are simple, like input validation, and offer a huge payoff. Others, like refactoring memory management, are more complex — but still worth it to protect against difficult-to-debug issues. |
| **16** | Looking ahead, our team should focus on closing three key gaps: secret scanning, memory safety, and ongoing developer training. I recommend integrating GitHub’s secret scanning into our CI workflow to catch exposed credentials early. We should also perform monthly audits on memory usage and encourage the adoption of modern C++ features like smart pointers and RAII. Finally, quarterly security refresh sessions can help ensure developers stay aligned with our evolving policies and practices. |
| **17** | To strengthen our defenses long term, we should formally adopt a secure coding standard such as SEI CERT for C++. This ensures consistent best practices across the team. We should also require static analysis integration in CI/CD pipelines, enforce role-based access for source control and environments, and document a clear exceptions process. These measures will help maintain a proactive security posture and reduce long-term risks. |
| **18** | These references support the policy you’ve seen today, from the SEI CERT C++ coding standards to documentation from SonarQube, Fortify, and other security tools we use. |