# CS 405 Project Two Script Template

Corey Sampson

Southern New Hampshire University

CS 405: Secure Coding

Professor Yi Wang

April 17, 2025

<https://youtu.be/dZr0SfHrQl8>

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | **Title Page**  Hello, my name is Corey Sampson. This presentation introduces the Green Pace Security Policy Guide. This guide is designed to ensure consistent application of modern security best practices in both coding and systems architecture. |
| **2** | **Overview: Defense in Depth**  As a development team expands, standardized security policies become more essential. These policies are grounded in industry-accepted principles and coding standards, enforced through automation in continuous integration and continuous delivery pipelines (CI/CD). This supports the overarching defense-in-depth strategy, promoting consistency, compliance, and reduced risk of vulnerabilities. |
| **3** | **Threat Matrix**  The Threat Matrix categorizes each identified vulnerability by likelihood and priority for mitigation. For example, data type and exception handling are classified as “Likely,” while SQL injection and string correctness are labeled “Priority.” Lower-priority risks, such as pointer safety and cryptography, remain important but are less urgent. Automation tools such as static analyzers flag these issues based on severity to enable prioritized remediation. |
| **4** | **10 Principles**  The Green Pace Security Policy is based on ten core principles:   1. Validating Data Input – Ensuring all user input is properly checked to prevent injection attacks and buffer overflows. 2. Heeding Compiler Warnings – Treating compiler warnings as potential vulnerabilities and addressing them proactively. 3. Architecture and Design for Security – Incorporating security requirements into the design phase to enforce a stronger architecture. 4. Keeping It Simple – Reducing complexity, making the system easier to secure and maintain. 5. Default Denying – Granting access only when explicitly authorized to reduce unauthorized use. 6. Principle of Least Privilege – Limiting user and process permissions to the minimum necessary. 7. Sanitizing External Data – Cleaning data shared across systems to prevent injection attacks. 8. Practicing Defense-in-Depth – Implementing multiple layers of security for redundancy and resilience. 9. Using Effective Quality Assurance Techniques – Employing rigorous testing, including code reviews, and penetration testing. 10. Adopting Secure Coding Standards – Applying industry best practices such as SEI CERT C++ to reduce vulnerabilities.   Each principle aligns with specific coding standards to enforce them in practice. |
| **5** | **Coding Standards**  The ten coding standards that are prioritized include:   1. Data Type Safety – Ensures correct use of data types to prevent confusion and memory corruption. 2. Data Value Initialization – Prevents use of uninitialized or incorrect data values. 3. String Correctness – Prevents buffer overflows and improper string handling. 4. SQL Injection Protection – Prevents SQL Injection attacks by using parameterized queries. 5. Memory Protection – Prevents memory leaks and unauthorized access. 6. Assertions – Ensures code correctness by validating assumptions. 7. Exception Handling – Improves error handling by using exceptions instead of error codes. 8. Thread Safety – Prevents race conditions and deadlocks in multithreaded applications. 9. Pointer Safety – Avoids dangling pointers and memory corruption. 10. Cryptography – Ensures secure data encryption and hashing.   Prioritization is based on exploitability and impact. Top concerns, such as data integrity and injection attacks, receive the highest attention. |
| **6** | **Encryption Policies**  To protect sensitive data:   * In Transit: Transport Layer Security (TLS) version 1.3 is used for secure communication. * At Rest: Advanced Encryption Standard (AES-256) 256-bit is used for all data storage. * In Use: Intel SGX and AMD SEV provide hardware-based memory protection.   This layered approach prevents data theft and ensures regulatory compliance. |
| **7** | **Triple-A Policies**  Authentication, authorization, and accounting policies consist of:   1. Multi-Factor Authentication (MFA) and Single Sign-On (SSO) to secure access points. 2. Role-Based Access Control (RBAC) to limit permissions based on user roles. 3. Centralized logging using Security Information and Event Management (SIEM) tools to capture all critical activity.   These measures enhance security and accountability. |
| **8** | **Unit Testing STD-001-CPP: Data Type Safety**  This unit test validates dynamic memory allocation using the malloc function. It ensures that the allocated memory is not null. The basic test confirms successful memory allocation. An advanced version tests different types of memory allocation, such as the new operator and the standard vector container. The Microsoft C++ Unit Testing Framework (CppUnitTest) uses Assert::IsNotNull to confirm allocation success. |
| **9** | **Unit Testing STD-002-CPP: Data Value Initialization**  This unit test checks that variables are properly initialized. The basic test verifies that an integer is set to zero. Additional tests can assess initialization for other data types such as floating-point numbers and character strings. Assert::AreEqual is used in the Microsoft C++ Unit Testing Framework to compare expected and actual values. |
| **10** | **Unit Testing STD-003-CPP: String Correctness**  This unit test verifies the safe copying of character strings using the strncpy\_s function. The basic test confirms that a string has been safely copied. Expanded testing includes handling buffer overflows and invalid input values. Assert::AreEqual is used to compare the original and copied strings. |
| **11** | **Unit Testing STD-004-CPP: SQL Injection Protection**  This unit test sanitizes input to prevent SQL injection. The basic test checks that potentially malicious characters are removed or neutralized. Further testing can involve additional filtering for other dangerous characters and injection patterns. Assert::IsFalse ensures that no injected code appears in the sanitized output. |
| **12** | **Unit Testing STD-005-CPP: Memory Protection**  This unit test uses smart pointers, such as standard unique pointers, to ensure memory protection. The basic test verifies that the smart pointer is not null after allocation. More advanced tests include the use of standard shared pointers. Assert::IsNotNull confirms that memory has been allocated safely. |
| **13** | **Unit Testing STD-006-CPP: Assertions**  This unit test demonstrates runtime checks through assertions. The basic test checks that an exception is thrown when invalid input is processed. Additional tests can involve various types of exceptions and custom exception classes. The Microsoft C++ Unit Testing Framework uses Assert::ExpectException for these tests. |
| **14** | **Unit Testing STD-007-CPP: Exception Handling**  This unit test ensures proper handling of runtime exceptions. The basic test confirms that the correct exception is thrown. Additional coverage may include testing specific exception types and validating associated messages. Assert::ExpectException is used to verify exception handling. |
| **15** | **Unit Testing STD-008-CPP: Thread Safety**  This unit test validates thread safety by using standard mutex and lock guard mechanisms. The basic test checks that a shared variable is accessed safely under a mutex. Expanded tests can evaluate protection under concurrent execution scenarios. Assert::IsTrue confirms that thread safety measures are effective. |
| **16** | **Unit Testing STD-009-CPP: Pointer Safety**  This unit test ensures that smart pointers are dereferenced safely. The basic test verifies that dereferencing yields the expected value. Additional testing may explore different smart pointer types and edge case behaviors. Assert::AreEqual checks that dereferencing works as intended. |
| **17** | **Unit Testing STD-010-CPP: Cryptography (Simulated)**  This unit test simulates a hash check to ensure that it is not empty. The basic test verifies that the resulting hash value is not empty. Advanced testing could implement actual cryptographic functions to verify hash integrity. Assert::IsFalse confirms that the simulated hash is not empty. |
| **18** | **Automation Summary**  Automated testing and analysis occur at multiple points:   1. Commit via Git hooks 2. CI/CD delivery, build, and test stages 3. Staging with scheduled scans   These checkpoints help detect vulnerabilities early and reduce reliance on manual review. Tools like SonarQube and Coverity enforce SEI CERT and CWE rulesets automatically. Any violations are logged and halt the CI/CD pipeline until resolution. |
| **19** | **Tools**  Tools such as SonarQube and Cppcheck are part of the DevSecOps pipeline. Security checks are embedded into all key stages to prevent vulnerabilities from progressing to production. Early notifications help improve overall code quality. |
| **20** | **Problems and Solutions**  Some of the problems that arise include:   1. Inconsistent use of secure coding practices. 2. No centralized enforcement of SEI CERT standards. 3. High-risk vulnerabilities such as memory corruption and injection attacks. 4. Increased breach risk due to scaling.   Some solutions to these problems could include:   1. Immediate integration of SEI CERT coding standards. 2. Automated security checks in CI/CD. 3. Required peer reviews and unit testing for critical code. |
| **21** | **Risks and Benefits**  Some of the benefits of immediate action are:   1. Reduced attack surface 2. Early vulnerability detection 3. Strong security-first culture 4. Simplified onboarding with standardized practices   On the other hand, some of the risks of delay are:   1. Hidden vulnerabilities could reach production 2. Higher remediation costs later in the Software Development Lifecycle (SDLC) 3. Greater chance of breaches and compliance violations 4. Damage to brand reputation |
| **22** | **Recommendations**  Some of the current gaps faced include:   1. Incomplete session handling policies 2. Inadequate dependency management 3. Absence of incident response documentation 4. Limited developer training resources   To cover these gaps, I recommend these actions:   1. Expand authentication and session expiration mechanisms 2. Integrate software composition analysis tools 3. Define formal incident response workflows 4. Develop onboarding and secure coding training materials |
| **23** | **Conclusions**  Keys for the future:   1. Perform regular audits and update policies 2. Automate vulnerability scans and include them in pull request checks 3. Promote developer security ownership through peer reviews and training 4. Maintain alignment with evolving SEI CERT and industry standards   The Green Pace policy is built on solid principles and a proactive approach. When consistently applied, the defense-in-depth strategy can effectively protect users and organizational assets. |
| **24** | **References**   * SonarSource. (2024). SonarQube Documentation. https://docs.sonarsource.com * Software Engineering Institute Computer Emergency Response Team. (2024). SEI CERT C++ Coding Standard. https://wiki.sei.cmu.edu * Veracode. (2024). Common Vulnerabilities. https://www.veracode.com   Thank you for listening to my presentation on the Green Pace Security Policy. |