# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, everyone. My name is Colin Kwasnik, and today I’ll be presenting our new security policy for Green Pace development teams. This policy outlines our defense-in-depth strategy using secure coding principles, best practices, and systems architecture enhancements to ensure a secure software development lifecycle. |
| **2** | This policy was created to formalize secure development practices across Green Pace. With increasing threats and regulatory demands, this policy defines consistent security principles, coding standards, and risk mitigation strategies. It supports our defense-in-depth framework by integrating secure methods at every stage of the software lifecycle. |
| **3** | This matrix visualizes the likelihood and criticality of each identified vulnerability. High-priority, likely threats such as SQL injection and buffer overflows appear in the upper-right, guiding where we focus our controls and automation. Lower-priority or less likely issues, such as incorrect data types or assertion misuse, still warrant attention but represent reduced risk to system stability. This matrix ensures our policy remains threat-informed and risk-prioritized. |
| **4** | Our ten core principles guide all secure development. Each standard maps back to at least one principle, ensuring consistency and completeness. For instance, STD-002 supports principles 1, 7, and 8 by validating input, sanitizing output, and layering protection. |
| **5** | In this slide, I’ve prioritized our ten C++ coding standards based on a composite threat ranking system that factors in four criteria: severity of potential impact, likelihood of exploitation, remediation cost, and overall risk level to system integrity.  Topping the list is STD-004: SQL Injection, due to its critical severity, high likelihood, and extremely low remediation cost. It’s a common and well understood vulnerability, and failing to prevent it can lead to full database compromise. Tools like SonarQube can detect it easily, making it a top priority.  Next is STD-002: Input Validation, which directly affects multiple vectors like buffer overflows, injection flaws, and data corruption. Its widespread applicability and high risk give it a strong ranking.  STD-003: Buffer Overflows comes next. These can lead to remote code execution and system crashes. While similar in severity to input validation, their remediation cost is slightly higher, and automated tools like Cppcheck and Clang-Tidy are essential for detection.  STD-009: Path Traversal ranks fourth. It has a medium likelihood but a high severity if exploited, especially when used against file systems with weak sanitization.  At fifth is STD-007: Exception Handling, which ensures stability under failure conditions. Uncaught exceptions can expose data or cause crashes, so safe handling practices are vital.  STD-005: Memory Leaks follows, due to the operational instability and denial of service potential caused by resource exhaustion. Though not always security critical, memory issues undermine system availability.  STD-008: Concurrency Issues, including race conditions, rank mid-level due to their high remediation cost and lower likelihood, but they can result in unpredictable security behavior in multithreaded applications.  Lower in priority but still important, STD-010: Resource Cleanup ensures files, sockets, and memory are not left dangling. It supports broader system hygiene and resilience.  Finally, STD-001: Incorrect Data Types and STD-006: Unsafe Assertions round out the list. While not immediately exploitable, they often cause undefined behavior and logic flaws. These have low likelihood and are typically easy to fix, so they're still enforced but with a lower priority.  This ranking system allows Green Pace to focus developer attention and automation tools where they will have the most security impact, maximizing return on investment in both effort and tooling. |
| **6** | Encryption at rest: Data stored in databases, files, or storage devices must be encrypted using AES-256 to protect against unauthorized access or theft. This policy applies to all sensitive data, including customer PII and proprietary data, stored on Green Pace servers or third-party cloud platforms. Encryption at rest ensures that even if physical or logical access is gained, the data remains unreadable without decryption keys, which are managed securely via a key management system (KMS). This is critical for compliance with data protection regulations and to mitigate breach risks.  Encryption in flight: Data transmitted over networks, including internal APIs and external communications, must use TLS 1.3 to encrypt data in transit. This applies to all network traffic containing sensitive information, such as user credentials or financial data, to prevent man-in-the-middle attacks. The policy ensures confidentiality and integrity during transmission and is enforced by configuring servers and clients to reject unencrypted connections. Regular audits verify TLS configuration compliance.  Encryption in use: Data processed in memory should be protected where feasible, using techniques like secure enclaves (e.g., Intel SGX) or homomorphic encryption for sensitive computations. This policy applies to applications handling cryptographic keys or highly sensitive data during runtime, such as financial transactions. It aims to protect data from memory-based attacks like buffer dumps and is implemented where supported by hardware and software capabilities, enhancing defense-in-depth. |
| **7** | Authentication: All users and systems must authenticate using multi-factor authentication (MFA) before accessing Green Pace systems or applications. This includes user logins and API interactions, verified through identity providers like Okta. The policy applies to all access points to ensure only verified identities can interact with systems, preventing unauthorized access. New users are provisioned with MFA during onboarding, and authentication logs are audited to detect anomalies.  Authorization: Access to resources, including files and databases, is restricted based on user roles and the principle of least privilege, enforced via role-based access control (RBAC). User access levels are defined during onboarding or role changes, and access to sensitive data (e.g., customer records) requires explicit approval. This policy applies to all systems to limit exposure and is audited to track access attempts, ensuring compliance and security.  Accounting: All user actions, including logins, database changes, new user additions, and file accesses, are logged and audited in real-time using a centralized SIEM (Security Information and Event Management, e.g., Splunk). Logs include timestamps, user IDs, and action details for traceability. This policy applies to all system interactions to enable forensic analysis and compliance reporting, ensuring accountability and detecting unauthorized or suspicious activities. |
| **8** | For STD-001, we focus on validating correct use of data types. Tests include edge cases like max and min integer values, negative-to-unsigned assignments, and type casting scenarios. GoogleTest is used to compare output values against expected types. This ensures predictable behavior and helps prevent underflow, overflow, or type mismatches. |
| **9** | For STD-002, we implement tests that validate incoming data for length, format, and character encoding. Using Catch2, we simulate various unsafe inputs, including null values and boundary-overflow attempts, to ensure our handlers reject them appropriately. These tests confirm that unsafe input is sanitized or blocked before processing. |
| **10** | Buffer overflow testing for STD-003 involves sending oversized input strings to functions and asserting that buffer sizes are not exceeded. We use both Catch2 and AddressSanitizer to test in CI. Tests also confirm that strings are null-terminated properly and memory boundaries are respected. |
| **11** | For STD-004, we mock database layers and feed them malicious input strings to test whether input is handled securely. Our tests ensure SQL queries are built using parameterized methods, not concatenation. If unsafe string assembly is detected, the test fails. This gives confidence that injection is not possible. |
| **12** | In STD-005, we test that all dynamically allocated memory is properly freed. GoogleTest integrates with Valgrind to run memory leak tests in CI. We also verify that smart pointers replace raw allocation wherever possible. The goal is to ensure that every new is matched with a delete or, ideally, avoided altogether. |
| **13** | Unit tests for STD-006 verify that runtime failures are handled using structured exception handling or error returns, rather than assert statements, which may be disabled in production. We simulate invalid states and confirm that they trigger runtime errors, not compile-time assertions, preserving error handling in deployed builds. |
| **14** | For STD-007, we write tests that intentionally throw exceptions, such as invalid inputs to std::stoi(), to ensure they are safely caught and logged. Our test framework confirms the system doesn’t crash and handles errors gracefully. These tests simulate real-world fault conditions and prove system resilience. |
| **15** | Concurrency unit testing for STD-008 involves running threads that increment shared variables to test for data races. We first test without synchronization to simulate failures, then introduce std::mutex or std::lock\_guard to confirm safety. ThreadSanitizer helps us detect unsafe memory access between threads automatically. |
| **16** | For STD-009, we pass malicious file paths like traversal sequences or absolute paths into our file-opening functions. Unit tests assert that these paths are rejected or normalized to a secure location. This confirms our file handling logic is safe from unauthorized file access. |
| **17** | Testing for STD-010 involves creating test scenarios that open files, allocate memory, or acquire locks. We then confirm that these are released in every control path, including on exceptions. Valgrind helps detect leaked descriptors. We also test that RAII patterns are implemented, ensuring cleanup happens automatically. |
| **18** | (DevSecOps Diagram) |
| **19** | Automation is fully embedded into Green Pace’s DevSecOps lifecycle to ensure continuous enforcement of our C/C++ coding standards.  In the pre-production phase, SonarQube is used during planning to flag high-risk issues like SQL injection. During design, Clang-Tidy helps detect problems such as improper data types. In the build phase, Cppcheck scans for vulnerabilities like buffer overflows and memory leaks and halts builds if violations are found. During testing, tools like Clang Static Analyzer, SonarQube, and ThreadSanitizer validate input validation, file path safety, and multithreading behavior.  In the production phase, Valgrind confirms resource cleanup before deployment. SIEM systems like Splunk continuously monitor logs to detect issues such as assertion misuse. Automated scripts can block deployments or trigger rollbacks if exceptions are mishandled. Finally, automated audits ensure long-term compliance and system stability.  By integrating tools directly into the CI/CD pipeline and aligning efforts across development, security, and operations, we maintain a strong, automated defense-in-depth posture from start to finish |
| **20** | This slide summarizes the key risks and benefits of implementing or delaying our security policy. The core problem we addressed was the lack of a unified approach to secure coding across teams, which created inconsistencies and exposed our systems to preventable vulnerabilities.  The solution is our centralized policy that enforces secure coding practices, supported by automated testing and monitoring throughout development and deployment.  If we delay adoption, we remain vulnerable to threats like SQL injection, memory leaks, or race conditions, which could lead to data loss or system compromise. On the other hand, by acting now, we strengthen system resilience, reduce long-term risk, improve compliance, and detect issues earlier.  For example, a critical buffer overflow was caught using Cppcheck in CI, preventing what could’ve become a production exploit. That’s the tangible value of proactive security enforcement. |
| **21** | While we've addressed most vulnerabilities, some services still lack runtime data protection. Future efforts should expand concurrency standards and introduce ML-based anomaly detection during reviews. |
| **22** | This policy reflects our commitment to secure development. It's comprehensive, actionable, and future facing. As threats evolve, so will we; continuously updating tools, standards, and procedures to keep our systems safe. |