# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, my name is Edwin Jones. This presentation introduces the security development policy I created for Green Pace. I’ll walk you through the coding standards, encryption policies, access controls, testing, and automation strategies that will help our team write secure, compliant, and maintainable software. |
| **2** | As Green Pace continues to scale, secure development needs to be built into every stage of the lifecycle. This policy formalizes our best practices and aligns with a defense-in-depth model—ensuring that security isn’t just a final step but something embedded from design through deployment.  Static analysis tools like clang-tidy and cppcheck help catch issues early, offering instant feedback as developers write code. By automating these checks, we prevent risky code from ever making it into version control. |
| **3** | This matrix groups each secure coding standard by its risk level.  "Likely" threats are the most frequent and dangerous—issues like unsafe memory handling and string misuse.  "Priority" risks are less common but still serious, such as unhandled exceptions or iterator misuse.  "Low priority" includes issues that are unlikely to be exploited but can still cause bugs, like buffer overreads or improper destructor behavior.  And finally, "Unlikely" threats, like unsafe integer conversions, pose low risk but are monitored.  These rankings help focus enforcement, and all are checked automatically using static and dynamic analysis tools during build and test phases. |
| **4** | The Green Pace policy is built around ten core security principles that guide all decisions made during development.  For instance, input validation plays a major role—it supports six of our ten standards and helps block attacks like buffer overflows and injection. Compiler warnings and secure coding standards apply to everything across the board, making them our most universal protections.  We also emphasize simple design, secure architecture, and layered defenses. Principles like least privilege and default deny show up in our access controls, while quality assurance and testing help us catch issues early.  Every standard in the policy is backed by one or more of these principles, ensuring we stay aligned with industry best practices. |
| **5** | The ten secure coding standards in this policy are ranked based on risk. At the top are issues like null pointer dereferencing, format string misuse, and improper memory cleanup. These are common, easy to exploit, and can cause serious damage.  In the middle, we’ve got things like exceptions, iterator safety, and the One-Definition Rule. These are still important, but less likely to be exploited directly. Finally, at the lower end, we’ve included violations like throwing from destructors and improper string access—still worth addressing, but less severe in practice.  Each ranking was based on how likely the issue is to occur, how hard it is to fix, and the potential impact if it’s exploited. |
| **6** | Our encryption policy protects sensitive data across three stages: at rest, in flight, and in use.  For data at rest, AES-256 encryption must be used for all disks, backups, and database storage.  Data in flight is protected with TLS 1.3, which secures both internal and external communication.  For data in use, we enforce memory isolation and recommend secure enclave support for processing sensitive information.  This layered encryption approach ensures we’re protecting data wherever it lives—whether it's stored, moving, or actively being used. |
| **7** | The Triple-A framework is a key part of our security policy.  For authentication, we require multi-factor authentication for all users to reduce the risk of unauthorized access.  Authorization is handled through role-based access control, so users only have access to what they need—nothing more.  Accounting ensures that every login, permission change, and data access event is logged and auditable. These logs are immutable, kept for at least one year, and reviewed on a regular basis.  Together, these three layers help us protect user identity, enforce access boundaries, and maintain accountability. |
| **8** | In this section, I’ll demonstrate how unit testing supports secure coding by identifying common vulnerabilities in container operations. Each I'll show was implemented using the GoogleTest framework and is mapped to a specific coding concern or principle.  This first test checks whether erasing a valid iterator range—from begin() to end()—successfully clears the container. When iterator ranges aren’t validated, you risk undefined behavior or memory corruption. This test ensures that the operation is safe and that the container ends up completely empty. |
| **9** | This test checks for proper bounds checking by calling .at(0) on an empty vector. It confirms that the code throws an std::out\_of\_range exception, as required by the standard. This ensures the program handles invalid access securely. |
| **10** | Here, I test whether resizing a filled vector to zero works correctly. The test verifies that memory is released safely, and the collection ends up empty with no errors. This supports good memory management practices in dynamic containers. |
| **11** | This test confirms that swapping two vectors properly exchanges their contents. After swapping, both containers hold the correct data from the other. This helps ensure state consistency and reliable behavior in operations that affect multiple data structures. |
| **12** | Finally, this test attempts to access index 10 in a vector that only has five elements. It confirms that an std::out\_of\_range exception is thrown, which protects the application from accessing invalid memory and enforces boundary safety. |
| **13** | Security is built into every stage of the development pipeline—not added at the end.  During coding and builds, static analysis tools and compiler flags enforce policy compliance automatically. The compiler plays a key role by flagging unsafe operations and enforcing secure configuration settings at build time.  Later stages include unit testing, vulnerability scanning, and runtime monitoring to validate behavior and catch violations early.  With automation integrated across the pipeline, security becomes part of the process—not a separate step. |
| **14** | Green Pace uses a DevSecOps pipeline where automated tools enforce policy compliance at every stage of development. During the design and build phases, tools like Clang Static Analyzer, CppCheck, and Parasoft C/C++test run static analysis to catch issues such as unsafe memory access, poor exception handling, or format string vulnerabilities. These are integrated directly into CI pipelines and development environments, blocking builds if violations are detected.  During the verify and test stages, we use Valgrind to detect memory leaks, invalid frees, and access errors. Polyspace Bug Finder and SonarQube are then used to perform additional quality checks and enforce long-term policy conformance.  This layered toolchain ensures that secure coding practices are enforced automatically — from planning through post-deployment — as part of our defense-in-depth strategy. |
| **15** | If security tools and coding policies are not implemented early, vulnerabilities may slip through undetected and become harder to fix later. This leads to delayed remediation, increased costs, and greater risk exposure.  The solution is to integrate secure coding checks and automation as soon as possible. Adding tools to the pipeline now helps detect issues early, enforce policies consistently, and reduce manual effort.  Waiting only allows potential flaws to grow into real problems. But acting now gives the team faster feedback, better coverage, and stronger code from the start. |
| **16** | Based on the current security policy and tool usage, several improvements are recommended. First, test coverage should be expanded to include more boundary and negative cases, ensuring that edge behaviors are fully validated.  Policy checks should be enforced consistently across all CI/CD pipelines so that no project bypasses security gates. Post-deployment, runtime monitoring should be added to detect violations that static and dynamic analysis might miss.  Regular reviews of the policy—at least quarterly—will help adapt standards to new threats and tools. Finally, providing secure coding training for new developers will keep the entire team aligned on expectations and best practices. |
| **17** | To strengthen the security policy, runtime monitoring should be implemented to catch violations after deployment. Static analysis must be consistently enforced across all CI/CD pipelines to ensure uniform compliance.  Testing should be expanded to include not only boundary conditions and negative cases, but also fuzz testing. Fuzz testing introduces randomized or malformed input to help uncover subtle vulnerabilities, such as those that contributed to the Heartbleed exploit. This technique offers a valuable layer of defense beyond traditional testing methods.  Regular quarterly audits are also recommended to assess policy effectiveness and keep pace with emerging threats. Adopting recognized standards like CERT C++ for memory safety, MISRA C++ for safety-critical reliability, and OWASP ASVS for application-layer security provides a strong foundation for continued policy maturity.  With these improvements in place, Green Pace can ensure that its secure coding practices are not only consistent and enforceable, but also adaptable to future threats. Thank you for reviewing this policy. |