# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone, I hope everyone is having a good day. My name is Nicholas Wyrwas and today I will be presenting the Green Pace Security Policy. This presentation will go over and explain in detail our policies, coding standards, security principles. While also outlining how we would maintain a strong defense-in-depth strategy.  If there are any questions after the video please leave a comment, as I always love having feedback and giving the chance to answer any questions. |
| **2** | In talking about the overview and defense in depth, we would like to keep the same standard secure development practices, good defense-in-depth strategy as well as standardize most coding principles.  In talking about these points, we would like to reiterate and discuss how this security policy was written to help maintain and ensure all Green Pace developers follow a standard secure development practice. We want to make sure that as this development team expands and we get more traction with more teams working on this project, that we hold this document close as it keeps all parties aligned with its standards. Meaning, it supports a defense-in depth strategy. Primarily by layering multiple protections across our software and systems. |
| **3** | This matrix lists the most severe and most likely coding risks, ranked by severity, probability, and cost. Items highlighted as HINT31-C and STR32-C (silent logic corruption and buffer over-read) and MEM54-CPP (memory corruption) have the potential to escalate any given bug to an exploitable condition. MSC32-C and MSC50-CPP guard against weak randomness or seeding that can lead to predictable tokens. FIO42-C is used to eliminate resource leaks and ensure system reliability and data integrity. EXP05-C is used to protect against fragile assertions by using real input validation instead. ERR51-CPP is used to require catch-all handling, to prevent catastrophic termination. CON37-C is used to eliminate unsafe use of signals in multi-threaded programs. DCL51-CPP is used to eliminate reserved identifier collisions. The coding checks will be enforced by warnings from compilers, static analysis tools (Coverity, CodeSonar, Astree), and unit tests for boundary conditions and failure paths. |
| **4** | These ten core principles form the basis of this policy. A given principle may be reflected by one or more coding standards (for instance, validating input data is reflected in INT31-C and STR32-C, defense in depth is reflected in each of them). This will allow us to demonstrate that each and every decision in our policy can be related back to one of the best-accepted principles. |
| **5** | We order the standards by impact/exploitability using the risk values from our Threat Matrix. Tier 1 - INT31-C, STR32-C, and MEM54-CPP - will harden the most catastrophic classes: integer corruption, string over-reads and memory misuse. Tier 2 targets MSC32-C, EXP05-C, and FIO42-C to eliminate predictable randomness, replace brittle assertions with actual validation, and end resource leaks. Tier 3 - ERR51-CPP, CON37-C, DCL51-CPP, and MSC50-CPP - will help to harden reliability and portability. This prioritization scheme allows us to prevent the highest probability, highest severity failures first while continually increasing our baseline security posture. |
| **6** | Our encryption policy protects data at rest, in flight, and in use. We use AES-256 for data at rest. For data in flight, we employ TLS or IPSec for encrypted communications. To protect data in use, we use secure enclaves and new encryption-in-use solutions to protect sensitive information while it's being processed. When layered together, these solutions can close gaps throughout the data lifecycle. |
| **7** | Access control can be enforced with the Three A's model. Authentication limits access to the system to trusted users only. Authorization enforces least privilege access by only giving users the minimum level of access they require. Accounting provides tamper-proof logs for audit compliance, investigations and system monitoring. |
| **8** | [Insert text.] |
| **9** | Automation: one of the key enablers of our security policy  To enforce our security policy, we have automated many security checks and integrated them into the DevSecOps pipeline. This ensures vulnerabilities are caught as early as possible and with maximum coverage across all builds, creating a repeatable and scalable way to ensure compliance without adding friction.  At the compiler level, Clang is configured to emit strict warning flags to detect unsafe language features or practices, such as reserved identifiers and implicit type conversions. Further along the pipeline, static analysis tools like CodeSonar, Coverity, Axivion, and Cppcheck scan the codebase for memory corruption, buffer overflows, and common string handling issues (STR32-C and MEM54-CPP). For a more in-depth analysis, we use formal verification tools like Astree and ÉCLAIR to perform path-sensitive checks and assist in validating rules like INT31-C and EXP05-C, where logical errors may not manifest during normal testing scenarios.  All of the above tools are integrated directly into the CI/CD process, so every code commit is automatically scanned, tested, and validated before it can be merged. Automation is also used post-compilation: unit tests are triggered, and dependency scans are run on third-party libraries and modules. Build artifacts are then validated before being deployed to target environments.  By automating at various stages of the SDLC – from coding, compilation, and unit tests to deployment – we not only reduce the possibility of human error but also make security enforcement continuous |
| **10** | Compiler level: Clang warnings are used to detect issues with reserved identifiers and unsafe macros; Static analysis tools (CodeSonar, Coverity, Axivion) can identify problems like buffer overflows, use of uninitialized memory, or erroneous integer casts; Formal analysis tools (Astree, ÉCLAIR) provide the ability to check more deeply into the code for rules like INT31-C or EXP05-C in a path-sensitive manner.  Tools are automatically executed during CI/CD builds so that these vulnerabilities can be found and addressed before code is merged. Unit tests and regression tests are executed as part of the pipeline that also check many of the standards such as string termination and integer handling.  Automating these tools and checks means Green Pace can provide consistent application of the security policy without burdening developers — security is part of the normal build process. |
| **11** | INT31-C - Integer Conversion Error, STR32-C - String Termination and MEM54-CPP - Memory Protection Failure. The benefit is that we avoid serious issues like data corruption, buffer over-read, memory exploits, which can be immediately weaponized by an attacker. The risk if not addressed is that the software can crash and burn during execution, and it can corrupt or leak data, or expose entry points for attackers.  MSC32-C - Predictable Random Seeds, EXP05-C - Assertions used Incorrectly or Redundantly. The benefit of fixing them in the near term is that they can cause system failures, but more often, they are used in conjunction with other lower- and medium-level flaws to mount a more serious attack. The risk or cost of not doing so is that medium and low level issues often do not get exposed by static analysis tools and go undetected until exploited. Thus, response time/cost can be high.  ERR51-CPP - Unhandled Exception, CON37-C - Unsafe Signal Handling and DCL51-CPP - Reserved Identifiers. These are generally not issues that lead to a system break-in, but do impact system reliability, maintainability, and scalability. The benefit is that by fixing the low-level ones, we ensure that our code is portable and will continue to scale as the number of people working on the code increase. The risk or cost of not doing so is that in a cost-benefit analysis, the work involved in fixing them is deferred to later, and becomes an increasing liability over time aka technical debt. Thus, the benefit-cost to fix is high.  For example:  I would order these in a descending risk priority.  1) High risk vulnerabilities are solved immediately 2) Medium vulnerabilities are fixed in the near term 3) Low vulnerabilities are handled via refactor. |
| **12** | My suggestions are centered around four core principles.  Firstly, dependencies should be under constant surveillance. Given that a majority of the security flaws can be traced back to libraries in use by a system, it is recommended that there are automated dependency scanners in place which can identify packages which are out-of-date or vulnerable, prior to their deployment in production.  Secondly, there needs to be improvements with respect to how Green Pace manages the vulnerabilities in their software stack. This includes a focus on core coding practices such as integer safety, string safety and memory safety, as well as putting a remediation framework in place, which would help guarantee that all the vulnerabilities which have been identified and assigned, have been patched by a specific deadline.  Thirdly, better encryption methodologies should be put in place. Although encryption in rest, in flight and in use are well-covered aspects, we can include more stringent key management and data loss prevention strategies, which can alert in the event of malicious access to sensitive data being attempted.  Lastly, implementation of least privilege can significantly help mitigate insider threats as well as accidental abuse. Application of this principle on both users as well as system processes ensure that permissions are only granted at a level that is required for each, thus limiting the damage that can be caused by loss of control.  The above suggestions aim to provide both short-term and long-term approaches, which should help Green Pace in its journey to maturity in the domain of cybersecurity, without losing sight of adaptivity to new threats as they emerge.. |
| **13** | Green Pace Security Policy  Sections that could benefit from enhancement or additional clarity:  Concurrency management in software projects: While the Green Pace policy provides guidelines on secure coding practices, including handling data in multi-threaded environments, there is a need for more explicit directives on concurrency management. This includes ensuring thread safety and managing concurrent processes effectively.  Exception and error handling: The policy emphasizes robust exception and error handling to prevent security vulnerabilities. However, specific requirements or standards for implementing these practices could be more detailed to ensure comprehensive coverage.  Monitoring and controlling third-party dependencies: The Green Pace policy addresses the risks associated with third-party dependencies and the need for careful monitoring and control. Additional guidance on implementing these controls effectively, including the use of automated scanning tools, would strengthen this aspect.  Recommendations for proactive measures to prevent issues:  Implementing additional concurrency control standards: Adopt more stringent concurrency control measures, such as the specific standard CON37-C, which mandates the elimination of data races in multi-threaded environments. This will enhance the thread safety of Green Pace applications.  Enhancing secure memory management practices: Strengthen secure memory management by incorporating practices such as those specified in MEM54-CPP. These practices include using the most current secure memory handling functions available in the C++ Standard Library to mitigate vulnerabilities.  Adopting automated dependency scanning tools: Implement automated scanning tools that can identify known vulnerabilities in third-party libraries and components before they are integrated into the production environment. This proactive measure will help in maintaining the security integrity of Green Pace software. |
| **14** | References supporting our security policy include: CERT C and C++ Secure Coding Standards, which influenced our coding rules; NIST 800-53, which outlines our control baselines; OWASP's Secure Coding Practices for input/output and data validation principles; Microsoft's DevSecOps documentation, which demonstrates the role of automation in our processes; and ISO/IEC 27001, the standard that frames our information security management approach. |