# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Welcome, everyone. My name is Hannah Hendrix, and I’m pleased to present the Green Pace Secure Development Policy Guide. This policy outlines the core principles, coding standards, and best practices that ensure all Green Pace applications are developed securely and consistently.  Throughout this presentation, I’ll cover the 10 core security principles, the C/C++ coding standards, and demonstrate how these align with risk mitigation strategies, automated testing tools, and secure development processes. Let’s get started! |
| **2** | Defense in Depth is a cybersecurity strategy that combines multiple layers of security to protect systems, data, and infrastructure from vulnerabilities. It was introduced to address growing risks from coding vulnerabilities, external attacks, and architectural flaws in development teams.  By standardizing existing security measures, it ensures consistent best practices and reduces the surface area of attacks.  Green Pace's development lifecycle will use this policy to support a comprehensive security framework, aligning with principles like code security, encryption, authentication, and automation testing. Vulnerabilities will be continuously identified, tested, and addressed using automation tools and unit testing frameworks to detect weaknesses early in the development process. |
| **3** | This slide presents the Threats Matrix, which categorizes potential security risks based on their likelihood and priority. The matrix helps us prioritize vulnerabilities so we can address the most critical issues first while keeping an eye on lower-priority risks that could escalate over time.  The two dimensions are: likelihood, or how likely it is for a vulnerability to be exploited and priority, or the impact of the vulnerability on the system if exploited.  Let’s walk through each quadrant:  Likely: This includes risks that are highly probable to occur, such as SQL injection attacks, input validation failures, and hardcoded credentials. These vulnerabilities often result from improper coding practices and must be addressed as a top priority.  Priority: This quadrant highlights the high-priority issues that would cause the most significant damage if exploited, even if they aren’t as likely. Examples include poor encryption strategies and insufficient authentication controls.  Low Priority: Here we have lower-impact risks, such as misleading error messages or minor performance bottlenecks that don’t immediately affect security but could be exploited indirectly.  Unlikely: This includes vulnerabilities that are less likely to occur due to proper controls or niche use cases, like buffer overflow in specific modules or rare system misconfigurations. While these are low likelihood, they are monitored periodically. |
| **4** | Here are the 10 guiding security principles that we use to ensure secure software development and systems architecture. Each principle is designed to address key vulnerabilities and promote consistent security practices across all development efforts:  Validate Input Data: Ensuring all input data is validated protects against injection attacks, buffer overflows, and other vulnerabilities. Validation includes verifying data type, size, format, and acceptable value ranges.  Heed Compiler Warnings: Compiler warnings often point to potential security flaws or functional errors. Developers must address these warnings proactively to preserve code quality and reduce risks.  Architect and Design for Security Policies: Security must be incorporated from the start. This includes applying risk assessment methods, using secure frameworks, and ensuring that all design decisions align with security policies.  Keep It Simple: Complexity increases the likelihood of errors and vulnerabilities. By designing simple systems, we ensure they are easier to test, manage, and protect.  Default Deny: Systems should deny access by default unless it is explicitly permitted. This principle minimizes the attack surface and prevents unauthorized access.  Adhere to the Principle of Least Privilege: Users, applications, and processes should only be granted the permissions they need to perform their tasks. This reduces potential harm from malicious or accidental activity.  Sanitize Data Sent to Other Systems: To prevent malicious inputs like SQL injection or cross-site scripting (XSS), all data sent to external systems must be sanitized and validated.  Practice Defense in Depth: Security should not rely on a single control. By layering multiple security measures, such as firewalls, encryption, and application-level controls, we ensure the system remains secure even if one layer fails.  Use Effective Quality Assurance Techniques: Security vulnerabilities can be identified and addressed through peer reviews, penetration testing, and static code analysis. These techniques ensure the software operates securely and as intended.  Adopt a Secure Coding Standard: A consistent and secure coding standard, such as CERT C/C++ or MISRA C, ensures that developers follow best practices, reducing the likelihood of security vulnerabilities. |
| **5** | This slide presents the 10 coding standards that form the foundation of the Green Pace Secure Development Policy. Each standard addresses specific vulnerabilities that, if left unresolved, could compromise system security.  Here is a brief overview of the coding standards:  Data Type Usage: Ensures proper data types are used to prevent precision loss or overflows.  Data Value Validation: Validates input ranges to avoid undefined behaviors.  String Correctness: Enforces safe string handling to prevent buffer overflows.  SQL Injection Prevention: Uses parameterized queries to sanitize SQL inputs.  Memory Protection: Ensures memory is properly managed to avoid leaks or undefined behavior.  Assertions: Verifies critical assumptions at runtime to prevent program crashes.  Exception Handling: Catches and handles errors gracefully, ensuring stability.  Logging and Auditing: Logs critical events for accountability and monitoring.  Thread Safety and Locking: Synchronizes multi-threaded operations to avoid race conditions.  Encryption Standards: Requires encryption for data at rest, in flight, and in use.  Next, I will explain how these standards are prioritized using a custom vulnerability ranking system.  The ranking system considers three key criteria:  Severity: How critical the vulnerability is to the security of the system.  Likelihood: How likely it is that the vulnerability will be exploited.  Remediation Cost: The effort and resources required to address the issue.  Each standard is assigned a priority level based on these factors. For example:  STD-003-STR (String Correctness) and STD-004-SQL (SQL Injection Prevention) are rated as Priority 5 because they are very likely to be exploited and require minimal effort to resolve.  On the other hand, STD-008-LOG (Logging and Auditing) is rated as a Priority 3 because while it’s important, the severity and impact are lower.  This ranking system allows us to focus on addressing the most critical vulnerabilities first, ensuring our applications are secure and compliant. |
| **6** | This slide outlines Green Pace's encryption policies to protect sensitive data throughout its lifecycle.  Encryption In Flight: Ensures all data transmitted between systems, such as API communications or web traffic, is encrypted using secure protocols like TLS 1.2 or higher. This protects data from interception during transmission.  Encryption At Rest: Requires sensitive data stored in databases, backups, or file systems to be encrypted using strong algorithms like AES-256. This safeguards data even if physical storage devices are compromised.  Encryption In Use: Addresses sensitive data processed in memory during runtime. By encrypting or masking this data, we minimize the risk of unauthorized access in memory or cloud-based environments. |
| **7** | This slide explains the policies supporting the Triple-A Framework: Authentication, Authorization, and Accounting.  Authentication:  All users and systems must authenticate using strong mechanisms like Multi-Factor Authentication (MFA). This ensures only authorized individuals or systems can access Green Pace resources. Examples include enforcing MFA for accounts, secure password policies, and using OAuth 2.0 for API authentication.  Authorization:  Access to resources is controlled through the Principle of Least Privilege (PoLP), meaning users or systems only have access to what is necessary for their roles. Examples include role-based access control, regular audits of permissions, and implementing access control lists for sensitive files and systems.  Accounting:  All activities, such as user logins and file modifications, are logged and monitored for accountability. This ensures compliance and provides an audit trail for investigations. Examples include logging login attempts, monitoring file access, and using centralized logging tools like Splunk or the ELK Stack.  Together, these policies ensure a secure, monitored, and accountable environment for all Green Pace systems. |
| **8** | This test validates the input handling for potential buffer overflow vulnerabilities.  The positive test case ensures that an input string of acceptable length is stored safely in the buffer. By verifying the string's size and null-terminating it, the system prevents memory overflows.  The negative test case tests the behavior when an oversized input is passed. The expectation is that the application will either throw an exception or prevent overflow by safely truncating the input.  The results confirm that:  The positive test successfully handled valid input without errors.  The negative test correctly identified and prevented buffer overflow, demonstrating robust input validation in our implementation |
|  | This demonstration highlights the risks of SQL injection and the importance of proper input handling in SQL queries.  The goal is to contrast a vulnerable SQL query implementation, executeQuery, with a secure implementation, executeQuerySafe.  Positive Test (Safe Implementation):  The executeQuerySafe function simulates secure query execution by validating user input properly.  When provided valid input like "admin", it correctly returns "User found: admin".  The test passed, demonstrating secure behavior.  Negative Test (Vulnerable Implementation):  The executeQuery function concatenates user input directly into SQL statements, creating a vulnerability.  When malicious input like "admin' --" is provided, the system incorrectly returns "Vulnerable: SQL Injection successful".  This test intentionally fails, exposing the vulnerability and emphasizing the risks of unsafe query execution. |
|  | This demonstration highlights the risks of command injection and the importance of input validation when executing shell commands.  The goal is to contrast a vulnerable command execution implementation, executeCommandInsecure, with a secure implementation, executeCommandSecure.  Positive Test (Secure Implementation):  The executeCommandSecure function simulates secure command execution by validating user input properly.  When provided valid input like "Hello World", it correctly returns "Hello World\n".  Additionally, it safely rejects invalid inputs like "Hello; rm -rf /", throwing an exception instead of executing dangerous commands.  The test passed, demonstrating secure behavior and proper input validation.  Negative Test (Vulnerable Implementation):  The executeCommandInsecure function directly concatenates user input into shell commands, creating a vulnerability.  When malicious input like "Hello; rm -rf /" is provided, the system incorrectly processes the input, confirming the injection vulnerability.  This test intentionally fails, exposing the risks of unsafe input handling and emphasizing the need for secure practices. |
|  | This demonstration highlights the risks of division by zero and the importance of input validation when performing mathematical operations.  The goal is to contrast a vulnerable division implementation, divideInsecure, with a secure implementation, divideSecure.  Positive Test (Secure Implementation):  The divideSecure function validates the denominator before performing the division.  For valid inputs, such as 10 / 2, it correctly returns 5.0.  For invalid inputs, like dividing by zero, the function throws a std::runtime\_error, ensuring that the system avoids undefined behavior or crashes.  The test passed, demonstrating proper error handling and secure behavior.  Negative Test (Vulnerable Implementation):  The divideInsecure function does not validate the denominator before performing division.  When attempting to divide by zero, the function allows undefined behavior, which can lead to crashes or other system failures.  This test intentionally fails, exposing the vulnerability and emphasizing the importance of validating inputs in mathematical operations. |
| **9** | This illustration outlines where security tools integrate into the software development lifecycle to ensure comprehensive protection. |
| **10** | The DevSecOps pipeline integrates security into every stage of the development and deployment lifecycle. Here's how external tools are used in each phase, as shown in the diagram:  In the Pre-production phase:  During the Assess and Plan stage, threat modeling tools like OWASP Threat Dragon identify potential risks and ensure security is considered early.  In the Design stage, tools like SonarQube enforce secure coding practices and catch vulnerabilities during development.  During the Build phase, dependency management tools like Snyk scan for vulnerabilities in libraries and ensure only secure components are used.  Finally, in the Verify and Test stage, static analysis tools like Cppcheck validate code compliance with security and functional requirements.  In the Production phase:  During the Transition and Health Check stage, penetration testing tools like OWASP ZAP validate deployment configurations.  In Monitor and Detect, SIEM tools like Splunk provide real-time monitoring, intrusion detection, and alerting.  If an issue arises, the Respond stage utilizes incident response platforms like Splunk Phantom to block attacks and remediate quickly.  Lastly, the Maintain and Stabilize stage ensures systems are reassessed, restored to baseline security, and stabilized after any incident. |
| **11** | This slide highlights the challenges, solutions, and risks of our current security strategy.  Problems: Security integration often happens late, leaving vulnerabilities unnoticed. Reactive measures are costly, and limited automation increases inconsistencies.  Solutions: We should integrate security early using tools like OWASP Threat Dragon, automate testing with Cppcheck and OWASP ZAP, and implement real-time monitoring with SIEM systems.  Risks and Benefits: Acting now reduces vulnerabilities and improves system integrity but requires upfront investment. Waiting may save costs temporarily but increases the risk of breaches and reputational damage.  Next Steps: To address these gaps, we should integrate security early, automate testing, and implement a robust incident response plan to ensure rapid recovery and system stability. |
| **12** | This slide identifies the key gaps in our current security policy and provides recommendations for improvement.  Identified Gaps:  First, there’s a lack of focus on runtime security, leaving production systems vulnerable to real-time threats. Second, we don’t have a comprehensive incident response plan to address breaches or recover quickly. Third, automation is not consistently applied across the CI/CD pipeline, increasing the risk of missed vulnerabilities. Lastly, there’s insufficient monitoring of vulnerabilities in third-party dependencies, which are often critical to our systems.  Recommendations:  To address these gaps, we should implement tools like SIEM and Runtime Application Self-Protection to monitor and block threats in real time. Developing and testing a robust incident response plan will ensure faster recovery from breaches. We also need to fully automate security testing during the build and verification phases, and use tools like Snyk to continuously monitor third-party dependencies for vulnerabilities.  By addressing these gaps, we can strengthen our security posture, reduce risks, and ensure the integrity of our systems. |
| **13** | This slide outlines the key standards we should adopt to prevent future security issues and improve our overall posture.  Standards to Adopt:  First, implementing secure coding standards like CERT C/C++ and MISRA C will enforce best practices and help minimize vulnerabilities during development. Next, we should adopt the OWASP Application Security Verification Standard, which provides a clear framework for application-level security. Additionally, following the NIST Cybersecurity Framework will allow us to take a comprehensive approach to identifying, protecting, detecting, responding, and recovering from threats. Lastly, we should embrace continuous security testing, using tools like SAST and DAST, to ensure security validation is ongoing and proactive.  Key Takeaways:  By adopting these standards and emphasizing continuous improvement, we can minimize vulnerabilities, maintain compliance, and establish a robust security foundation to meet future challenges. Automation and adherence to these frameworks will be critical as security threats continue to evolve. |
| **14** | No dialogue |