# CS 405 Project Two Script

**Dennis James Stelmach**

**Project Two: Security Policy Presentation**

[**https://youtu.be/Z-qQrl-QWrQ**](https://youtu.be/Z-qQrl-QWrQ)

| **Slide Number** | **Narrative** |
| --- | --- |
| **1**  **[Introduction]** | Hi everyone. I am Dennis James Stelmach, and this presentation will review the new Green Pace Development Security Policy. |
| **2**  **[Overview: Defense in Depth]** | This policy outlines coding practices that strengthen the security posture of the applications we develop. The need for secure development is growing due to several factors:   * The rise of cyber threats and software vulnerabilities. * The importance of protecting sensitive data within applications. * The need to comply with industry regulations (if applicable).   Green Pace's security policy follows the principle of "defense in depth," which creates a layered approach to application security. |
| **3**  **[Threats Matrix]** | This matrix categorizes threats based on their likelihood and priority. We prioritize addressing high-likelihood, high-priority threats, such as SQL injection and buffer overflow vulnerabilities. We still address likely-likelihood, medium-priority threats like memory leaks and cross-site scripting (XSS). Even low-priority threats deserve attention, though they are less concerning. |
| **4**  **[10 Principles]** | Secure coding practices are essential in our defense-in-depth strategy. Here are 10 key principles:   * Validate User Input: This prevents malicious attacks by verifying all external data. * Heed Compiler Warnings: Pay attention to warnings from the compiler, as they can indicate potential security issues. * Architect and Design for Security Policies: Integrate security considerations into the design phase of development. * Keep It Simple: Write clear, concise code that is easier to understand and maintain, reducing the risk of errors. * Default Deny: Follow the principle of least privilege, granting only the minimum permissions necessary for users and systems. * Adhere to the Principle of Least Privilege: Limit access to minimize the impact of potential breaches. * Sanitize Data Sent to Other Systems: Cleanse data before sending it to prevent vulnerabilities on receiving systems. * Practice Defense in Depth: Implement multiple security layers to enhance overall protection. * Use Effective Quality Assurance Techniques: Utilize tools and processes to identify and fix errors early in development. * Adopt a Secure Coding Standard: Follow Green Pace's Secure Development Policy, which incorporates these principles. |
| **5**  **[Coding Standards]** | Green Pace has established specific coding standards (STDs) that enforce secure coding practices. These standards address data types, data values, string correctness, SQL injection, memory protection, assertions, exceptions, input/output operations, object-oriented design, and concurrency. |
| **6**  **[Encryption Policies]** | Encryption plays a vital role in protecting data:   * At Rest: Protects data on storage devices (e.g., hard drives) even if accessed by unauthorized users. Critical for sensitive data and regulatory compliance. * In Flight: Secures data traveling over networks (emails, online banking) from eavesdroppers. Essential for secure online transactions and public Wi-Fi. * In Use: Considers encrypting highly sensitive data within the application for an extra layer of security. Particularly important for financial data or strict privacy regulations. |
| **7**  **[Triple-A Policies]** | Green Pace utilizes a Triple-A (Authentication, Authorization, Accounting) approach to security:   * Authentication (Pre-Application) verifies user identities before granting access to the application. Imagine checking IDs before entering a building that houses the application. Common methods include usernames/passwords, multi-factor authentication (MFA), or digital certificates. * Authorization (Within and Beyond the Application) determines what actions a verified user can perform after authentication. Think of granting access to specific floors (resources) within the building based on your ID role. Secure coding practices (like data validation) can indirectly support authorization by preventing unauthorized data access even within the application. * Accounting (Beyond the Application) tracks user activity on the network (not just within the application). Imagine keeping logs of who entered which areas (resources) within the building and when. Application logs focused on user actions within the application's scope can contribute data for broader accounting systems. |
| **8**  **[AccessingNegativeIndexThrowsException Unit Test]** | Now, we will demonstrate coding standard STD-005: Memory Protection  Here we see a unit test ensuring attempts to access invalid memory locations throw exceptions. This unit test also demonstrates the Exceptions coding standard by using a well-described exception to block non-safe code execution. |
| **9**  **[CanFindAddedValue Unit Test]** | This test verifies data can be found after adding it to a collection, enforcing that the correct memory operation occurred. |
| **10**  **[EraseBeginEndErasesCollection Unit Test]** | This test ensures that erasing a collection functions correctly, similarly enforcing that the correct memory operation occurred. |
| **11**  **[ReserveIncreasesCapacity Unit Test]** | This test confirms that reserving memory capacity doesn't alter collection size. Again, as memory operations are occurring, we want to ensure that they are occurring correctly and as expected. |
| **12**  **[Automation Summary]** | The DevSecOps pipeline automates various tasks throughout the development lifecycle to deliver secure software faster. |
| **13**  **[Tools]** | Here's a breakdown of the pipeline stages and how security tools integrate:   * Coding and Committing: Developers write code and commit it to a version control system (e.g., Git). * Static Analysis: Tools like PVS-Studio, CPP Lint, and Clang-Tidy analyze code for vulnerabilities, coding errors, and style violations early in development. * Building and Unit Testing: The code is compiled and undergoes unit testing to ensure individual units function correctly. * Security Testing: Tools like SQLMap and Acunetix scan the code for vulnerabilities like SQL injection and cross-site scripting (XSS). * Sanitizer Testing: Tools like AddressSanitizer (ASan) identify memory errors like leaks and data races, improving code stability and security. * Code Coverage Analysis: Tools like SonarQube analyze how much of the code is covered by unit tests, ensuring comprehensive testing. * Integration and Deployment (Optional): Depending on the pipeline configuration, this stage might involve automated integration testing and further security testing before deployment. * Monitoring and Feedback: The pipeline monitors the application after deployment for security incidents or errors. This feedback loop allows for continuous improvement of security practices. |
| **14**  **[Risks and Benefit]** | Proactive security measures are essential to avoid the consequences of security breaches. There are two major areas of thought:   * Act Now (Patch/Mitigate):   + Prevents attacks by patching vulnerabilities before attackers exploit them.   + Minimizes damage by taking early action to reduce data loss and financial impact.   + Builds a robust security posture through a proactive approach. * Wait:   + Increases the risk of breaches by leaving vulnerabilities open for attackers.   + Can lead to compliance issues and fines for neglected security.   + Makes it harder to contain a security incident if you delay action.   In order to fully practice Defense in Depth, action should be taken as soon as possible before breaches or attacks even occur. |
| **15**  **[Recommendations]** | Here are some recommendations for improving Green Pace's security standards:   * Provide more specific guidance on data validation types for different data formats. (e.g., numeric ranges, expected characters in strings) * Develop a broader standard for error handling that defines different severity levels and appropriate handling mechanisms. * Establish a standard outlining best practices for key management, encryption algorithms, and secure storage of sensitive data. * Create a standard for evaluating the security of external libraries and potential vulnerabilities. * Integrate threat modeling into the early stages of development to identify potential attack vectors and inform secure coding practices. * Implement secure logging practices that minimize data, control access, and employ tamper detection mechanisms. * Develop a standard for secure coding practices specific to developing APIs, including authentication, authorization, input validation, and error handling. |
| **16**  **[Conclusions]** | Green Pace's security policy emphasizes these key secure coding principles:   * Don't trust external data, verify it to prevent attacks. * Grant minimal permissions to users and systems. * Use smart pointers for safe memory management. * Cleanse data before processing to remove harmful elements. * Prevent SQL injection with secure query construction. * Patch vulnerabilities by keeping software up-to-date. * Integrate security testing throughout development. * Train developers on secure coding practices. |
| **17**  **[References]** | [End of Presentation] |