# CS 405 Project Two Script

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Introduction   * Security Policy Presentation for Green Pace |
| **2** | Defense in Depth   * This policy will governor all employees at Green Pace, to ensure best practices are continuous to be used. Defense in Depth is best described as a castle. A castle will have many layers of defense, a mote, outer and in walls, and gates with guards. This policy follows the same principle of ensuring our software has many layers of security, focusing on the network security, host security, endpoint security, and app security. |
| **3** | Threat Matrix   * On this slide, we are covering the Threat Matrix, a tool we have used to categorize potential security vulnerabilities based on their likelihood of occurrence and the priority they should be given. * The matrix is divided into four quadrants: Likely, Priority, Low Priority, and Unlikely. Each quadrant helps us prioritize where to focus our efforts on mitigating security risks. * In the "Likely" quadrant, we have Assertions, Exceptions, and Input/Output vulnerabilities. These are issues that we anticipate encountering frequently * The "Priority" quadrant includes Data Value, Memory Protection, and String Correctness vulnerabilities. These issues are not only likely but also critical to address due to their potential impact on the system's security and stability. * In the "Low Priority" quadrant, we have SQL Injection and Object-Oriented Programming concerns. While these are important, they are considered less likely to occur in our context based on current assessments. * Finally, in the "Unlikely" quadrant, we have Data Type and Concurrency vulnerabilities. These are issues that we expect to encounter less frequently, but they can still lead to significant problems if they occur. |
| **4** | 10 Principles   * Here we have our ten Security Principles that help guide the company during development. Starting with: * Validate Input Data - Validation data is important to prevent security vulnerabilities, such as cross-site scripting, SQL Injection, or buffer overflow attacks. * Heed Compiler Warnings - A complier warning can indicate potential flaws in the code that could lead to security vulnerabilities. Warning can result in software that is more susceptible to attacks. * Architect and Design for Security Policies - In all great system designs, security is an integral part of the architecture and design. Implementing security policies such as access controls, authentication, and encryption from the start. * Keep It Simple - By keeping the system simple, it helps reduce the risk of introducing security vulnerabilities. * Default Deny - By design, deny principle involves configuring systems to deny access first, only allowing access when explicitly granted. * Adhere to the Principle of Least Privilege - The principle of least privilege dictates that users and processes should have minimum level of access necessary to perform their functions. * Sanitize Data Sent to Other Systems - All data needs to be sanitized when sent to an external system to prevent injection attacks and data leakage. * Practice Defense in Depth - Defense in depth is a layered security approach that involves implementing multiple security measures at different levels. * Use Effective Quality Assurance Techniques - Quality assurance techniques such as code reviews, automated testing, and static analysis help in identifying and mitigating security vulnerabilities. * Adopt a Secure Coding Standard - Secure coding standards provide guidelines for writing code that is resistant to security vulnerabilities. |
| **5** | Coding Standards  Never qualify a reference type with const or volatile   * Qualifying a reference type with const or volatile can lead to confusing and undefined behavior, as it suggests that the reference itself is immutable or volatile rather than the object it references.   Never hard code sensitive information   * Hardcoding sensitive information such as passwords, cryptographic keys, or personal data directly in the source code exposes it to potential attackers who can easily extract it from the binary or source files.   Do not attempt to create a std::string from a null pointer   * Attempting to create a string from a null pointer can lead to undefined behavior, including crashes or security vulnerabilities.   Do not specify the bound of a character array initialized with a string literal   * Specifying the bound of a character array when initializing it with a string literal can lead to errors if the bound is incorrect, potentially causing buffer overflows or truncation of the string.   Do not access freed memory   * Accessing freed memory can lead to undefined behavior, including crashes, data corruption, and security vulnerabilities such as use-after-free attacks.   Choose an appropriate termination strategy   * Choosing an appropriate termination strategy is essential for handling unexpected conditions and errors in a controlled manner.   Honor exception specifications   * Honoring exception specifications ensures that functions only throw exceptions that are declared, promoting code reliability and predictability.   Do not slice derived objects   * Object slicing occurs when a derived class object is assigned to a base class object, resulting in the loss of the derived part of the object.   Wrap functions that can spuriously wake up in a loop   * Functions that wait for conditions, such as condition variable wait, can sometimes wake up without the condition being met due to spurious wakeups.   Do not copy a FILE object   * Copying a FILE object can lead to undefined behavior and resource management issues, such as multiple file handles pointing to the same file stream. |
| **6** | Encryption Policies   * Green Pace encryption policies, aiming to protect sensitive data throughout its lifecycle. These policies ensure that data remains secure whether it is stored, transmitted, or processed, reducing the risk of unauthorized access and data breaches. * Encryption at rest is critical for protecting data stored on physical storage devices, such as disks or other media. The goal is to ensure that if physical storage is compromised, the data remains unreadable without the appropriate decryption keys. * Encryption in flight secures data as it moves across networks, protecting it from interception or tampering during transmission. This policy covers all sensitive data communications, both internal and external, ensuring that data remains confidential and unaltered during its journey. * Encryption in use protects data while it is being actively processed in memory or during computation, particularly in environments such as cloud computing. This policy is crucial for scenarios where sensitive data is manipulated in real-time, ensuring that even during processing, the data remains secure and inaccessible to unauthorized parties. * Implementing robust encryption practices helps mitigate risks associated with data breaches and unauthorized access, thereby maintaining the trust and integrity of Green Pace's systems. |
| **7** | Triple-A Policies   * This slide covers the Triple-A framework, which stands for Authentication, Authorization, and Accounting. These policies are essential for ensuring that only the right users can access sensitive resources, that they have appropriate permissions, and that all critical activities are tracked and logged. * Authentication is the first line of defense, verifying the identity of users or systems before granting access. Green Pace mandates the use of strong, multi-factor authentication (MFA) for all systems to ensure that only authorized individuals can access sensitive resources. * Once authenticated, authorization controls what these users or systems can do within the environment. The policy enforces the principle of least privilege, meaning users and systems have only the access necessary for their roles, minimizing the potential damage from compromised accounts. * Accounting is about tracking and logging all critical activities within the system. This policy requires detailed logging of user activities, such as logins, database changes, and file accesses. It also tracks the addition of new users and changes to user access levels. |
| **8** | Unit Testing   * Unit testing is requirement for secure coding. Unit testing is a fundamental practice for verifying that individual components of the code work as expected before they are integrated into larger systems. * The primary goal of unit testing is to ensure that the smallest, most isolated pieces of code—often referred to as "units"—function correctly. By performing both positive and negative tests, we can confirm that the code behaves as expected in various scenarios, including edge cases. * On this slide, we see an example of unit tests performed in this course, specifically related to adding values to a collection. The tests verify different aspects, such as whether the collection can add values to an empty vector, whether it can handle multiple or single values, and whether it ensures that collections are empty when created. The tests also check properties like whether the capacity of the collection is greater than or equal to its size after resizing and whether resizing increases the collection size as expected. The results shown here include both passing and failing tests, which is normal during development. * By automating these tests and integrating them into the development workflow, we ensure that our codebase remains robust and that new changes do not negatively impact existing functionality. |
| **9** | Automation Summary   * On this slide, I want to highlight how we will apply the principles and best practices into the DevSecOps lifecycle to enhance security and efficiency in software development. The diagram illustrates the different stages of the software development lifecycle, dividing them into pre-production and production phases, with security integrated into each stage. * Pre-Production Phase * Assess and planning - In the planning stage, automation tools help in assessing the threat landscape, evaluating regulatory changes, and performing impact analysis. * Design - During the design phase, automation enforces security best practices like Open Web Application Security Project guidelines. * Build - In the build phase, automation ensures that secure builds are created using trusted repositories and secure source usage. * Verify and Test - This stage involves vulnerability scanning, automated functional testing, and compliance checks. * Production Phase * Transition and Health Check - Automation ensures that security settings are configured and deployed consistently across environments. * Monitor and Detect - In production, automated monitoring tools like Splunk or SIEM systems collect and analyze logs for real-time threat detection. * Response - Automated response tools like Cortex XSOAR can execute predefined playbooks to respond to detected threats, such as blocking attacks or rolling back compromised systems. * Maintain and Stabilize - Automation plays a key role in maintaining security baselines and ensuring that systems remain stable and secure after any changes or incidents. * By automating key security tasks, Green Pace can improve efficiency, reduce the risk of human error, and maintain a strong security posture throughout the software development lifecycle. |
| **10** | Automation Tools   * ThreatModeler - Used for automated threat modeling, helping to identify potential threats in the planning phase. * SonarQube - Can be used to enforce security best practices during the code design phase by analyzing code quality and security issues. * Jenkins – Automates the build process, ensuring that security policies are integrated into the build pipeline. * OWASP - An automated tool for vulnerability scanning and security testing of web applications. * Ansible - Automates the configuration and deployment process, ensuring that security settings are consistently applied across environments. * Splunk - Used for automated log collection, SIEM (Security Information and Event Management), and real-time analytics to detect security threats. * Cortex XSOAR - Automates the response to incidents by orchestrating security playbooks to block attacks and remediate threats. * Chef InSpec - An automated tool for continuous security assessment, ensuring that systems remain compliant with security baselines and policies. |
| **11** | Risks and Benefits   * Risk and Benefits * Don’t leave security until the end. This slide we will discuss the risk and benefits of when including security early in the SDLC vs later. * The first point emphasizes the importance of incorporating security from the very beginning of the SDLC. By integrating security early, it becomes a continuous part of the development process, ensuring that each stage is secure. Potential security issues are identified and addressed sooner, preventing them from becoming larger problems later. Proactive security measures significantly reduce the chances of security breaches, protecting both the organization and its customers. * The risks associated with delaying security integration until later stages of development are increased cost, delays in product release, and loss of private data. Delaying security often leads to higher costs due to the need to refactoring code to address security flaws that could have been prevented. Addressing security late in the process can cause significant delays in getting the product to market, as fixes and rework take time. The most sever risks is the potential loss of customer data, which can lead to a significant loss of trust and damage to the organization’s reputation. |
| **12** | Recommendations   * Although this document provides excellent recommendations for further improving Green Pace’s security practices, there is room for improvement. Cybersecurity is a forever growing and evolving field. * Comprehensive and detailed documentation ensures that all stakeholders understand the logging and monitoring processes, making it easier to maintain, audit, and improve these systems over time. * Regular documentation of patching processes ensures that patches are applied consistently and in a timely manner, reducing the risk of vulnerabilities being exploited. This also includes setting up a schedule for routine maintenance and ensuring that all systems remain up to date with the latest security patches. * A real-world example of a security breach is Equifax Breach in 2017. Attackers exploited an unpatched vulnerability in Apache Struts, a widely used software component, leading to one of the most significant data breaches in history. This incident underscores the need for a strong patch management process and the risks associated with neglecting timely updates. |
| **13** | Conclusions   * By adopting these standards, Green Pace can proactively address potential security issues before they become problems. Continuous improvement and adherence to these best practices will enhance the organization's overall security posture, protecting both the company and its customers from emerging threats. * Thank you for your attention, and I am open to any questions or further discussions on these topics. |
| **14** | * Please see the following references. |