**Date: 10/20/2024**

**Video Link:** <https://www.youtube.com/watch?v=zJaulcvGqBg>

**Slide 1: Introduction**

Hello team! My name is Jonathan, and today I will be presenting my new security policy, which contains core principles, secure coding standards, and best practices that help to prevent potential security vulnerabilities in both Green Pace’s code development and systems architecture.

**Slide 2: Defense-in-Depth Overview**

This presentation begins with the highlighting of Green Pace’s core defense strategy, Defense in Depth. This approach involves utilizing multiple layers of defense to ensure if one fails, others are still in place to mitigate potential threats. The adoption of this comprehensive strategy lowers the attack surface in systems and reduces threat potential.

**Slide 3: Threats Matrix**

On this slide, I have populated a threats matrix utilizing potential security vulnerabilities that are individually addressed with their own preventative coding standards in a future slide. The vulnerabilities in the low and likely category mostly relate to simple mistakes or oversights, such as attempting to modify a string literal in read-only memory or allowing an attempt to divide by zero. In contrast, the vulnerabilities in the likely and high category deal with the passing of data to complex subsystems and allow for the possibility of a buffer overflow attack. It is important to note that many of these vulnerabilities can be easily identified through the use of automated static analysis tools, such as cppcheck or Klocwork, which ensures these are caught early during the development phase prior to them becoming an issue in production.   
  
**Slide 4: Ten Core Security Principles**  
Now, let’s go ahead and review the ten core security principles that form the foundation of my security policy for Green Pace. These principles help guide every decision we make as developers when it comes to development.

1. **Validate Input Data –** We validate all input data so we can prevent the possibility of SQL injection attacks, buffer overflows, or other input-related vulnerabilities.
2. **Heed Compiler Warnings –** Compiler warnings should never be ignored. These often are an indication of an underlying issue, like type mismatches or uninitialized variables, that could lead to further vulnerabilities.
3. **Architect and Design for Security Policies –** Security must be part of every architecture developed by Green Pace from its inception to proactively address threats.
4. **Keep It Simple –** Simpler code is much easier to maintain, review, and secure. Complex and difficult to read code can hide vulnerabilities which would otherwise be obvious.
5. **Default Deny –** All access should be denied by default unless granted explicitly. This lowers unauthorized access and the attack of our systems.
6. **Adhere to the Principle of Least Privilege –** Both processes and users should have their access limited to only what is required to perform their functions effectively. This limits any potential damage caused in the event of a security breach.
7. **Sanitize Data Sent to Other Systems –** We must ensure that any and all data leaving our systems is sanitized to prevent the propagation of potential vulnerabilities.
8. **Practice Defense in Depth –** The utilization of multiple security layers ensures that if one fails, others still remain in place to keep the system secure.
9. **Use Effective Quality Assurance Techniques** – The team should conduct regular code reviews and implement automated testing to help detect security vulnerabilities early.
10. **Adopt a Secure Coding Standard –** Following recognized standards, such as the CERT C++ Coding Standard (CERT, 2024), ensures a consistent approach to security across all teams within Green Pace.

These principles help to ensure security is considered and integrated at every step of development, from design to deployment.

**Slide 5: Coding Standards**  
Here, I have organized the ten coding standards that my security policy has adopted first by their severity. If their severity is equal, the next priority to look at is likelihood. When a standard is both high in severity and likely to happen, the final comparison has to do with the remediation cost of fixing the code and making it adhere to the secure coding standards endorsed by this policy.

**Slide 6: Encryption Policies**  
Green Pace strives to ensure all user data is protected, and these are the encryption policies we utilize. Encryption at rest refers to the idea of encrypting data while it is in storage, such as on a database or system backup. This policy helps to ensure sensitive user data is secured with a strong algorithm to protect data even if a physical device is compromised. Encryption in flight is the concept of encrypting data as it is being transmitted across networks. This can be between the devices of clients or even between different internal components of an application. Green Pace’s adoption of this policy requires us to employ a secure protocol that can maintain data confidentiality and integrity while in transit. Encryption in use is the practice of encrypting data while it is being actively processed. In order to apply this policy, techniques like homomorphic encryption need to be implemented, which allows mathematical operations to be performed on data without needing to decrypt it. Perhaps the most important policy, encryption in use, applies to all critical operations that involve loading sensitive data into memory. Now, let’s take a look at the triple-A policies.

**Slide 7: Triple-A Policies**Here at Green Pace, we employ the triple-a policies, which are authentication, authorization, and accounting. Authentication revolves around verifying the identity of users, systems, or services that are attempting to access data or something else within our system. Ensuring the authenticity of users is crucial to prevent unauthorized access. Once authenticated, authorization takes over. Through the use of something like role-based access control, it is crucial that we define specific roles and grant them permissions in line with the least privilege principle. Accounting tracks and records everything that happens within the system, which allows us to review the logs and investigate any suspicious behavior. By tracking every action, accounting provides a high level of accountability for anyone interacting with the system.

**Slide 9: Unit Testing Example 1**

Hands-on testing of code is something we do here at Green Pace, so let’s take a look at a few examples. I have chosen to include two examples of a positive test and two examples of a negative test. A positive test is expecting the result of the function being tested to be correct, while a negative test is expecting to have some sort of error or exception thrown. My first example is a positive test, which verifies that resizing a collection to be zero empties the collection. This is checked by first making the collection greater than zero, resizing the collection to zero, and then utilizing the empty() function to return a bool that indicates whether or not the collection is empty.

**Slide 10: Unit Testing Example 2**

Here we have another test that is testing to see if entries can be added to a collection that is currently empty. First, the collection is checked to ensure it is empty. Next, an entry is added to the collection, and the empty() function is checked, with the expectation of it returning as false. Finally, the actual size of the collection is checked to be 1, which is the expected size.

**Slide 11: Unit Testing Example 3**This is my first example using a negative test. This function verifies that an out\_of\_range exception is thrown when an index that is out of bounds is attempted to be accessed. I do this by first adding five entries to the collection, verifying the collection is a size of 5, and then attempting to access index 8, which does not exist. In this case, the exception is thrown and the test is passed.

**Slide 12: Unit Testing Example 4**My fourth and final example of unit testing is my second negative test. This one checks to see if a length\_error exception is thrown when attempting to assign a number of elements to the collection that is greater than the max\_size variable. This is done simply by creating a variable that is the max\_size variable plus one. These elements are then assigned to the collection, which causes the length\_error exception to be thrown and the test to pass. Unit testing is an important part of Green Pace’s development cycle, which utilizes the DevSecOps pipeline.

**Slide 13: Automation Summary**This diagram is a visual representation of the DevSecOps pipeline and identifies the various phases of development a project goes through across both pre-production and production. The unit testing showed earlier would usually occur in the verify and test phase of pre-production. Automation plays a key role in ensuring security is integrated throughout each stage of the DevSecOps pipeline. During pre-production, the team will take advantage of automated tools to analyze the evolving threat landscape, incorporate automated input validation and security checks, and employ automated validation tools to test dependencies. Moving into production, automated deployment tools will help us configure security settings and validate the security of the system through penetration testing. We will also use continuous monitoring tools to detect and respond to potential threats in real time. All of this together ensures the robust protection needed for Green Pace systems. 

**Slide 14: Automation Tools**I listed a few examples of automated analysis tools that you may or may not already have experience utilizing in the past. Cppcheck, Helix QAC, and TrustInSoftAnalyzer are all static analysis tools created for C and C++. Some of these tools even directly address some of the coding standards discussed earlier. These are a small selection of the many different tools out there to pick from, so it is best to find the one that works best for your situation or project.

**Slide 15: Risks and Benefits**So far, I provided you all with Green Pace’s strategy, core security principles, and secure coding standards, but what are the risks and benefits of waiting to implement these practices versus acting now? The incorporation of new security tools and practices could be a costly initial investment, which requires the team to climb a steep learning curve to become comfortable with the tools. Despite this, acting now will allow us as a team to prepare for potential threats that could come in the future. Implementing these practices now will also streamline future development thanks to their early incorporation. If we wait to take action, we increase the risk that unaddressed security vulnerabilities are exploited. This is especially problematic if the vulnerability is created early in development, as the cost and effort needed to fix the issue down the line would be much higher. While waiting to act has no upfront cost or learning curve, the risk of a security breach occurring is a cost that has the potential to be much higher than any security tools or team training. 

**Slide 16: Recommendations**

Let’s move on to the recommendations I have to improve Green Pace’s security policy. Our current measures are robust, but there are still some key gaps that I have identified. Developer training is the first thing that needs to be addressed. Without a proper and consistent training program for our team, our developers could unknowingly introduce vulnerabilities into our systems. The next area that needs improvement is our policy with regard to third-party integration. We cannot rely on the security of third parties and must ensure that our end is secured at all times whenever these interactions occur. An example of this being done poorly can be found in the Target cyberattack of 2013, which saw a third-party HVAC company’s system being compromised and having that propagate to Target’s poorly protected servers (Steinberg et al., 2021). It is also clear to me that Green Pace’s current security policy lacks a concrete standard for the continuous monitoring of our system. It is crucial we implement enhanced and detailed logging to monitor and ensure suspicious activity can be detected and tracked.

**Slide 17: Conclusions:**In conclusion, to prevent future security breaches, it is essential that Green Pace adopt and adhere to widely accepted and recognized industry standards. These standards help to provide a structured approach to securing systems and mitigating vulnerabilities before they become severe threats. OWASP, or the Open Web Application Security Project, has a set of 10 best practices that help developers identify and avoid common web application vulnerabilities (OWASP, 2024). The CERT C++ Coding Standards offer specific guidance on secure coding practices in C++ that help prevent issues like buffer overflows or improper exception handling. All of the coding standards picked for my security policy were sourced from the CERT C++ Coding Standards. Finally, NIST, or the National Institute of Standards and Technology, provides a comprehensive framework for improving cybersecurity across companies. The frame emphasizes risk assessment, continuous monitoring, and incident response planning. By learning and adopting these critical industry standards, the team can ensure Green Pace has a proactive security posture and our systems stay safe and secure.