Sean Glass

October 20, 2024

Project Two: Security Policy Presentation

Presentation Link: <https://youtu.be/h2l1wWEslgY>

# CS 405 Project Two Script

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone! My name is Sean and today I will be presenting the Green Pace security policy guide and will be providing implementation guidelines and recommendations for maintaining it in the future. |
| **2** | The purpose of this security policy is to address potential vulnerabilities in code development and in system architecture. By standardizing secure coding practices, we can greatly reduce the risk of vulnerabilities such as SQL injection, buffer overflows, and other critical risks. To support our defense-in-depth best practice, this policy introduces multiple layers of protection. From code-level security, such as input validation and encryption, to system-level protections like network and endpoint security, each layer adds to the overall strength of our defense. The diagram shows how different security layers interconnect to provide comprehensive protection. |
| **3** | This Threats Matrix categorizes the risks based on likelihood and priority. As you can see, some issues like buffer overflows and SQL injection are both likely and high priority due to their severe consequences. On the other hand, resource management and error handling pose a lower risk to overall system security. This matrix allows us to prioritize our efforts based on where vulnerabilities are most likely to occur and cause damage. |
| **4** | These are the 10 core principles that guide our security policy. Each principle has been aligned with specific coding standards to ensure that we follow best practices in our development process. For example, the first principle of 'Validate Input Data' aligns with standards that address input validation like preventing buffer overflows and SQL injections. Similarly, 'Heed Compiler Warnings' ensures we catch vulnerabilities that have been flagged by the compiler early in the development process. |
| **5** | These coding standards are prioritized based on the severity of the vulnerability and the potential impact. High-priority issues, such as SQL injection and buffer overflow, are addressed first because they pose the greatest security risk. Medium-priority standards focus on system stability and resource management, while low-priority items like string correctness pose less immediate risk. |
| **6** | Our encryption strategy covers three key areas: encryption in flight, encryption at rest, and encryption in use. Encryption in flight secures data during transmission using protocols like TLS or SSL, encryption at rest protects stored data by using strong encryption algorithms, and encryption in use secures data while it's being processed in memory. Together, these measures ensure the confidentiality and integrity of data throughout its lifecycle. |
| **7** | The Triple-A framework consists of three key policies: authentication, authorization, and accounting. Authentication verifies user identities, requiring strong methods such as multi-factor authentication. Authorization controls access to resources based on user roles, ensuring adherence to the principle of least privilege. Accounting tracks user activities and system access to provide a trail for security analysis. Together, these policies ensure secure access control and accountability within our systems. |
| **8** | Now, we are going to look at some of the unit tests we have created to ensure the system maintains security. In this unit test, we are testing whether the smart pointer for a collection is initialized properly when the collection is created. The test uses the GoogleTest framework. The ASSERT\_TRUE macro checks whether the collection object was successfully created, ensuring that it's not null. The ASSERT\_NE macro is used to verify that the smart pointer inside the collection is not equal to nullptr, confirming that the pointer is correctly pointing to the collection.  If this test passes, it indicates that our collection is initialized properly and doesn't have null references when the collection is empty. By ensuring that smart pointers are valid upon initialization, we prevent potential null pointer dereference errors, which could otherwise lead to crashes or undefined behavior in our program. |
| **9** | In this unit test, we are verifying that we can successfully add an element to an initially empty collection. First, we check that the collection is empty using the ASSERT\_TRUE macro to confirm that it returns true for the empty() method and the size of the collection is 0. Next, we simulate adding a single element to the collection using the add\_entries(1) method. After adding the entry, we confirm that the collection is no longer empty using ASSERT\_FALSE, verifying that the empty() method now returns false. We also use ASSERT\_EQ to check that the size of the collection has increased to 1, ensuring the addition of the element was successful. This test confirms that the collection’s behavior for adding elements is functioning as expected. |
| **10** | This unit test checks whether the collection can successfully store and maintain multiple unique values. In this test, we use the push\_back method to add three distinct values of 10, 20, and 30 to the collection. After adding the values, we use the ASSERT\_EQ macro to verify that the values are correctly stored at the expected indices. Specifically, we check that:  The value at index 0 is 10,  The value at index 1 is 20, and  The value at index 2 is 30.  This test ensures that the collection can not only hold multiple unique values but also that it maintains the correct order of these values. |
| **11** | This unit test checks whether the collection correctly throws an exception when attempting to access an invalid negative index. This is an important negative test case to ensure that invalid operations are properly handled. The ASSERT\_THROW macro is used here to verify that when the method at(-1) is called, an out\_of\_range exception is thrown. This ensures that the collection protects against accessing elements using invalid negative indices, which could lead to undefined behavior or crashes. By confirming that the collection throws the appropriate exception, we reinforce robust error handling and prevent invalid accesses to the data structure. |
| **12** | This diagram illustrates how automation is integrated throughout the DevSecOps pipeline. The next slide discusses this integration. |
| **13** | In pre-production, automation starts with threat modeling and static analysis tools during the design and build phases. These tools catch vulnerabilities early. In testing, automated dynamic testing is used to simulate attacks and verify application security before moving to production. In production, continuous monitoring tools automatically detect and log potential threats, keeping the system secure in real time. In post-production, automated incident response tools isolate threats and restore the system to a secure state, ensuring minimal downtime. Automation ensures that security is continuously integrated at every stage, keeping the system secure throughout its lifecycle. |
| **14** | The main problem with delaying the implementation of security automation is that vulnerabilities may go undetected, increasing the risk of breaches. By acting now and integrating automated security tools, we can greatly reduce manual testing effort, improve security, and catch vulnerabilities early in the process. Waiting to implement these measures not only increases the likelihood of a breach but also raises recovery costs in the event of an attack. Acting now provides a stronger security posture, lower long-term costs, and a more efficient development process with proactive threat mitigation. |
| **15** | Despite the strong security measures, there are still gaps that need to be addressed. It is important to implement security tests in the continuous integration pipeline to ensure vulnerabilities are automatically detected with every code change. Input validation coverage should also be expanded, and fuzz testing should be included to catch unexpected inputs. It is also necessary to automate our incident response processes even more to ensure quicker containment and mitigation of potential threats. |
| **16** | In the future, adopting more comprehensive secure coding standards will ensure that best practices are applied consistently throughout development. Enhancing threat modeling and risk assessment will make the system more adaptive to new threats. Continuous education for the team on the latest security strategies and vulnerabilities will also help Green Pace to stay ahead of emerging risks. |