**Project Two: Security Policy Presentation**

**Justin Hancock**

**CS-405 Secure Coding**

**10.15.2024**

**https://youtu.be/WfDkkh6n9DI**

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1**  **Title** | Hello, my name is Justin Hancock, and today I'll be presenting the new security policy for Green Pace. In this presentation, we’ll discuss our approach to securing our software development lifecycle, how we address potential security risks, and the steps we’ve taken to create a more resilient and protected environment. We’ll also dive into key principles and strategies that help prevent vulnerabilities at each stage of development and system operation. |
| **2**  **Overview – DiD** | Our Defense in Depth strategy is all about creating multiple layers of security. This means that if one part of our system is compromised, there are several additional protections in place to minimize the impact of the breach. For example, in addition to using firewalls and encryption, we also regularly test our systems for vulnerabilities to detect any weaknesses early. By layering these defenses, we don’t rely on any single security measure—each layer works together to create a more comprehensive shield. This approach is inspired by real-world security measures like those used in a bank. There’s not just one lock on the vault—there are security guards, cameras, alarms, and backup systems in place to protect the money. Similarly, in our system, we combine several security techniques, such as access controls, intrusion detection, and encryption, to secure sensitive data. We also focus on the idea of redundancy. This means that if one defense fails, another one can take its place. For example, if an attacker manages to bypass a firewall, encryption certifies that the data they access is unreadable without the proper keys. This kind of planning ahead allows us to reduce the damage even if a breach occurs. |
| **3**  **Threats Matrix** | Our threats matrix highlights the most critical risks to our system. The top concerns include SQLinjection, where attackers manipulate databases by injecting malicious queries. This is particularly dangerous because it can expose confidential data or even allow attackers to control the system. In response, we focus on input validation and using secure coding practices to prevent these types of attacks. Another significant vulnerability is memorymanagement, which involves improper handling of system memory. If there’s a memory leak or buffer overflow, it could cause the system to crash or be exploited by attackers to gain unauthorized access. Tools like Valgrind help us identify these memory issues during testing so we can fix them before they affect the live system. Lower on the priority scale are issues like deadcodeelimination. Although it doesn’t pose a direct security threat, removing unused code helps reduce system complexity and improves performance, making it easier to manage and secure the system in the long term. Each of these vulnerabilities is addressed with different techniques and tools to make certain the system remains both secure and efficient. |
| **4**  **10 Principles** | We’ve built our security policy around 10 core principles. One of the most important is Validating Input Data. This principle is critical because many security vulnerabilities, like SQL injection mentioned previously, happen when systems fail to properly validate user inputs. By rigorously checking the data that enters our system, we prevent attackers from manipulating it in ways that could compromise our security. Another key principle is Heeding Compiler Warnings. Modern compilers are designed to catch potential errors or vulnerabilities in code before the software even runs. By addressing these warnings early on, we can avoid many common security flaws, like buffer overflows, that might be missed during manual code reviews. Each principle works together to support our overall security strategy. For example, with Default Deny, access to any system resource is blocked unless explicitly allowed. This reduces the chances of unauthorized access, especially when combined with other principles like Adhering to Least Privilege, which limits user permissions to only what is necessary for their role. |
| **5 Coding Standards** | Our coding standards are prioritized based on the severity of the threats they address. For instance, SQL Injection Prevention is at the top because it directly addresses one of the most common and dangerous types of cyberattacks. By validating inputs and using parameterized queries, we make it much harder for attackers to exploit the system. Memory Management is another high-priority standard because poor memory handling can lead to serious issues, from data leaks to system crashes. Promising that memory is properly allocated and freed helps avoid vulnerabilities, which are often exploited in attacks. Lower-priority standards, like Assertions and Dead Code Elimination, focus more on improving code quality and system efficiency. While they don’t directly prevent security breaches, they help create cleaner, more maintainable code, which makes the system easier to secure and monitor. |
| **6**  **Encryption Policies** | Encryption plays a vital role in protecting sensitive information. Encryption in flight refers to securing data as it’s transmitted between systems, such as when users access a website or transfer files. By using protocols like TLS (Transport Layer Security), we make sure that the data cannot be intercepted or read by unauthorized parties during transmission. Encryption at rest secures stored data, such as in databases or backups. Even if an attacker gains physical access to the storage device, the data remains encrypted and inaccessible without the proper decryption key. We use strong encryption algorithms like AES-256 to keep the data secure. Encryption in use is especially important for cloud environments, where data is processed by multiple users and systems. This method secures the data while it’s being actively worked on, which is critical for protecting sensitive information in shared or multi-tenant environments. All three types of encryption work together to create a full lifecycle of protection for our data. |
| **7**  **Triple-A Policies** | The Triple-A Framework—Authentication, Authorization, and Accounting—helps manage access to our systems. Authentication verifies a user’s identity, whether through a password, biometric scan, or multi-factor authentication. This step is essential to guarantee that only legitimate users can access our systems. Authorization comes next, determining what resources each authenticated user can access. By implementing Role-Based Access Control (RBAC), we limit users to only the permissions they need for their job. This reduces the risk of unauthorized access to sensitive areas of the system. Accounting tracks and logs all user activities, from login attempts to changes made within the system. This creates an audit trail that helps us detect suspicious behavior and provides accountability. In the event of a security incident, these logs allow us to trace the source of the problem and take appropriate action. |
| **8**  **Unit Test 1** | This test checks whether we can safely add elements to an empty collection, like a vector. We start by adding valid elements to confirm that the system behaves as expected. Then, we push the system further by testing edge cases, such as adding too many elements or invalid inputs. These tests help us identify any potential weaknesses in how the system handles different types of data. By testing these scenarios, we not only verify that the system works as intended under normal conditions, but we also explore how it handles unexpected or unusual situations. These edge cases are crucial because they often expose vulnerabilities that wouldn’t appear during regular use. |
| **9**  **Unit Test 2** | Resizing a collection is a fundamental operation, but it can cause problems if not done correctly. In this test, we check how well the system handles resizing, making sure that no data is lost and that the system doesn’t run into memory issues. For example, we test cases where the collection is resized to a larger or smaller capacity, checking that all elements are retained and memory is properly allocated. We also test scenarios where resizing might exceed system limits, such as trying to allocate more memory than is available. This helps us confirm that the system remains stable, even when pushed to its limits. |
| **10**  **Unit Test 3** | Clearing a collection is a common operation, but it’s important to verify that all data is properly removed and that the system doesn’t leave any memory or data behind. In this test, we make sure that after calling the clear() function, the collection is empty and ready for new data. We also test edge cases, such as attempting to clear an already empty collection. This helps us confirm that the system handles these situations gracefully, without throwing errors or crashing. |
| **11**  **Unit Test 4** | Handling invalid operations is a key part of keeping the system stable and secure. In this test, we verify that the system throws the correct exceptions when users attempt invalid operations, such as accessing out-of-bounds elements. Properly handling these errors prevents the system from crashing and makes sure that users are alerted when something goes wrong. We also check how well the system logs these exceptions, which is important for troubleshooting and auditing purposes. If an error occurs, having detailed logs allows us to trace the issue back to its source and make necessary adjustments. |
| **12**  **Unit Test 5** | Boundary cases often expose system vulnerabilities that might not show up during normal operations. In this test, we assess how the system handles operations at its capacity limits, such as adding elements right at or beyond the collection’s maximum size. By testing these boundary conditions, we make sure the system reacts appropriately under stress and doesn't crash or produce unexpected behavior. It’s important to test both positive and negative boundary scenarios. For example, in the positive case, we expect the system to handle adding elements up to its limit. In the negative case, where we try to exceed that limit, the system should throw an error or prevent the operation without crashing. Handling boundary cases effectively is key to building robust and reliable systems. |
| **13 & 14**  **Automation Summary** | Pre-Production: The Assess andPlan phase involves identifying potential risks and mapping out a security strategy. This is where we assess what threats might impact our system and decide how to mitigate those risks. During the Design phase, security is integrated directly into the system architecture. This proactive approach helps us avoid vulnerabilities later on by addressing security concerns from the beginning. In the Build stage, we use tools like Cppcheck and SonarQube to scan the code for vulnerabilities before it's deployed. These tools help us catch issues such as memory leaks, uninitialized variables, or potential security weaknesses during the development phase. The Verify and Test stage then focuses on dynamic testing, such as using Google Test for unit tests and OWASP ZAP for dynamic vulnerability analysis, to make sure the system behaves securely under various conditions.  Production: Once the system moves into production, the Monitor and Detect stage involves constant observation of system performance and security. Tools like Prometheus and the ELK Stack help us track logs and system metrics in real time, giving us the ability to detect anomalies or potential security breaches as soon as they occur. In the Respond stage, our team acts quickly to address detected vulnerabilities or issues. This involves reviewing logs, identifying the source of the problem, and applying a fix. The final phase, Maintain and Stabilize, is all about keeping the system up to date with security patches and monitoring its health to prevent new vulnerabilities from emerging over time. By integrating security into each phase of the DevOps lifecycle, we create a continuous loop of improvement and protection. |
| **15**  **Problems & Solutions** | One challenge we face is the inconsistent application of security practices across different teams. Without a unified approach, some vulnerabilities may go unnoticed. To address this, we’ve integrated automated security tools into every phase of the development process, allowing us to identify and mitigate issues earlier. This also means that security checks happen consistently, reducing the chance of something slipping through. Another issue is the lack of continuous real-time monitoring. Even after the system is deployed, it’s important to keep an eye on it for any potential breaches. Using tools like ELK Stack, we can monitor the system’s logs for any unusual activity and respond immediately if something looks off. This proactive approach helps us prevent small issues from becoming major breaches. |
| **16**  **Risks & Benefits** | Taking action now to implement these security measures will require an initial investment in tools, training, and time. However, this investment will greatly reduce the chances of a costly breach in the future. Proactively securing the system now helps avoid the long-term costs associated with data breaches, system downtime, and potential legal consequences. On the other hand, delaying these improvements increases our exposure to vulnerabilities, leaving the system at risk for attacks. If a breach occurs, the cost of recovery can be significant, both in terms of financial resources and the damage to our reputation. By acting now, we not only strengthen our security but also improve compliance with industry standards and ensure quicker response times when vulnerabilities are detected. |
| **17**  **Recommendations** | One gap in our current policy is the lack of ongoing security training for developers. Security is a constantly evolving field, and without regular training, our team may not be equipped to handle new threats. To address this, we recommend implementing regular workshops or training sessions to keep the team updated on the latest security techniques. Another gap is the absence of a formalized incident response plan. While we have monitoring tools in place, we need a clearly defined process for responding to security incidents. This includes assigning roles and responsibilities, setting up communication channels, and establishing response timelines. Having a response plan allows us to act quickly and effectively when an issue arises. Finally, third-party dependency management is an area that needs more focus. Many systems rely on external libraries and frameworks, which can introduce vulnerabilities if they are not regularly updated. We recommend integrating tools to automatically check for vulnerabilities in third-party dependencies and alert us when updates are needed. |
| **17**  **Conclusion** | In conclusion, our recommendations for improving the security policy focus on three key areas: continuous security training, a detailed incident response plan, and better third-party dependency management. These improvements will close the gaps in our current strategy and strengthen the overall security of our systems. By addressing these areas, we reduce the likelihood of vulnerabilities slipping through the cracks, improve our ability to respond to security incidents, and verify that we are always up to date with the latest security practices. This proactive approach will not only protect our systems but also safeguard the organization’s data and reputation in the long term. |