# CS 405 Project Two Script Template

**Project Two: Security Policy Presentation**

Matthew J. Rearick

Department of Computer Science

CS-405 Secure Coding

Prof. Farley, Toni

8/18/2024

Video Link:[Matt Rearick 7 2 Project Two Presentation (youtube.com)](https://www.youtube.com/watch?v=Oqp-6mQl9m4)

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | * Good afternoon, everyone. My name is Matthew Rearick, and today I will be presenting the Green Pace Security Policy Guide. * This presentation is designed to walk you through the key aspects of our new security policy, detailing the standards, principles, and best practices that will help us safeguard our systems and ensure our development practices are secure. * Let's get started. |
| **2** | * This policy was developed in response to the growing complexity of our IT environment and the increasing sophistication of potential attacks. * By implementing a robust defense-in-depth strategy, we aim to standardize security practices across all layers of our infrastructure. * This approach not only reduces the surface area for potential attacks but also ensures that we have multiple layers of defense working in harmony. * As you can see in the diagram, our security policy touches on every aspect of our infrastructure, from physical and network security to application and endpoint protection. * This comprehensive approach is essential for safeguarding our critical assets and ensuring the long-term security of our organization. |
| **3** | * Moving on to our threat matrix, which is an essential part of our security policy. * This matrix helps us identify and prioritize potential security risks based on their likelihood and impact. * For instance, vulnerabilities such as those addressed by coding standard STD-004-CPP are categorized as high severity due to their likelihood and priority of impact on our systems. * By systematically addressing these threats, we ensure that our resources are focused on the most critical areas. * To further strengthen our defenses, we're also leveraging automation to detect and mitigate these vulnerabilities. * Automation allows us to quickly identify coding issues, such as those listed in the matrix, and apply fixes before they can be exploited. * This proactive approach not only reduces the time and effort required to manage security risks but also ensures that our defenses are continuously updated to counter emerging threats. |
| **4** | * Now let's dive into the core principles that guide our security policy and the corresponding coding standards that ensure these principles are consistently implemented across our systems. * **Validate Input Data**:   + Ensuring that all input data is valid is crucial to prevent vulnerabilities such as SQL injection and cross-site scripting.   + We enforce this with coding standards like STD-004-CPP, which focuses on preventing SQL injection, and STD-006-CLG, which mandates proper validation and the avoidance of reliance on assertions for input checking. * **Heed Compiler Warnings**:   + Compiler warnings often indicate potential vulnerabilities.   + By addressing these warnings, we avoid subtle bugs that could become security risks. Standards like STD-001-CPP, which ensures that enum values stay within range, and STD-002-CPP, which warns against reliance on evaluation order, are integral to this principle. * **Architect and Design for Security Policies**:   + Security should be embedded in the architecture and design of systems from the ground up.   + Standards such as STD-006-CLG and STD-007-CPP support this by promoting secure design practices and comprehensive exception handling. * **Keep It Simple**:   + Complexity is the enemy of security. Simple designs are easier to secure and maintain.   + Standards like STD-002-CPP and STD-003-CLG help enforce simplicity by avoiding the pitfalls of complex evaluation orders and preventing the modification of string literals. * **Default Deny**:   + This principle states that access should be denied by default and only granted explicitly.   + This is a key part of our security posture, though it doesn't directly align with a specific coding standard, it is a principle that influences all access control decisions. * **Adhere to the Principle of Least Privilege**:   + Minimizing the access rights of users and systems reduces the potential impact of a security breach.   + This principle is foundational and implicitly supported by secure coding practices that limit scope and permissions. * **Sanitize Data Sent to Other Systems**:   + Before sending data to other systems, we sanitize it to prevent malicious content.   + STD-004-CPP, which prevents SQL injection, is directly aligned with this principle, ensuring that data handling is secure. * **Practice Defense in Depth**:   + A layered approach to security ensures that even if one control fails, others will provide protection.   + Standards like STD-005-CLG and STD-007-CPP promote defense in depth by ensuring that memory management and exception handling are robust. * U**se Effective Quality Assurance Techniques**:   + Rigorous QA practices are essential to detect and fix security issues before they reach production.   + Standards like STD-002-CPP, STD-005-CLG, and others support this principle by enforcing code correctness and stability. * **Adopt a Secure Coding Standard**:   + Finally, adopting a secure coding standard is about following a consistent set of practices to minimize vulnerabilities.   + This overarching principle is supported by all the coding standards we've discussed, as each one contributes to building a secure codebase. * By aligning these principles with specific coding standards, we're ensuring that our security practices are not just theoretical but are grounded in concrete, actionable guidelines that our developers can follow every day. |
| **5** | * In this slide, we present the 10 coding standards, prioritized according to their level of severity, likelihood of occurrence, and remediation cost. * Starting at the top, STD-004-CPP and STD-005-CLG are at Level 5, as they represent high-severity vulnerabilities that are likely to occur and have a medium cost of remediation. * This makes them our highest priority, requiring immediate attention and stringent security measures. * Next is STD-010-CPP at Level 4. * Although it shares a high severity and likelihood with the Level 5 standards, the higher remediation cost places it just below in terms of priority. * Our Level 3 standards—STD-002-CPP, STD-003-CLG, and STD-006-CLG—reflect medium-priority risks, balancing the need for attention with the resources required to mitigate them. * These standards are crucial but less critical than those in the higher levels. * At Level 2, we have STD-001-CPP, STD-007-CPP, and STD-009-CPP. * These standards represent low-priority risks, either due to their lower likelihood of occurrence or reduced severity. * While still important, they demand less immediate focus. * Finally, STD-008-CPP ranks at Level 1, the lowest priority. * Its low severity and unlikely occurrence make it a minor risk, but one that should still be monitored. * This prioritization allows us to allocate resources effectively, ensuring that the most significant vulnerabilities are addressed promptly while still managing less critical risks efficiently. |
| **6** | * In this slide, we summarize the encryption strategies adopted to secure data in its various states: at rest, in flight, and in use. * Encryption at Rest focuses on protecting data stored on physical media, such as hard drives or cloud storage.   + Our policy mandates encrypting all sensitive data with robust algorithms like AES-256.   + This ensures that even if unauthorized individuals access the storage media, the data remains unreadable and secure.   + The primary goal here is to prevent data breaches, comply with regulatory requirements, and safeguard the organization’s assets. * Next, Encryption in Flight addresses data security during transmission across networks.   + Whether data is moving between devices, servers, or clients, it’s essential to protect it from interception and tampering.   + Our policy applies encryption to all transmitted data, especially sensitive information, enforcing secure communication protocols like TLS/SSL.   + This strategy protects against eavesdropping, man-in-the-middle attacks, and other threats during data transmission. * Finally, Encryption in Use ensures the protection of data while it’s being actively processed in memory or CPU registers.   + This is particularly critical in environments where real-time data access occurs, such as financial transactions.   + Our policy mandates using advanced technologies like homomorphic encryption or secure enclaves to secure sensitive data during active processing.   + This reduces the risk of memory scraping or unauthorized access to data in use. * By implementing these encryption strategies across all stages—at rest, in flight, and in use—we ensure comprehensive protection of sensitive data, meeting both security and compliance objectives. |
| **7** | * This slide focuses on the Triple-A Framework, which consists of Authentication, Authorization, and Accounting—three critical components for a robust security strategy. * First, Authentication:   + The policy here emphasizes the importance of verifying the identity of users or systems before granting them access to any resources.   + To enhance security, this includes enforcing multi-factor authentication across all systems, ensuring that only verified individuals or systems can interact with sensitive data or critical operations.   + This approach mitigates the risk of unauthorized access and impersonation. * Second, Authorization:   + Once a user is authenticated, the authorization policy defines what actions they are permitted to take based on their roles and responsibilities.   + This ensures that each user or system only has access to the resources necessary for their specific role.   + By clearly defining and enforcing these permissions, we reduce the risk of unauthorized actions, providing a secure and organized framework for managing access controls. * Finally, Accounting:   + The accounting policy involves meticulously tracking and logging user activities, such as logins, changes to the database, and file access.   + Regular reviews of these logs are essential for detecting and addressing any potential security concerns.   + This policy not only supports forensic investigations in the event of an incident but also enhances overall accountability and transparency within the organization. * Together, these policies within the Triple-A Framework create a comprehensive approach to security, ensuring that access to resources is tightly controlled, monitored, and managed. |
| **8** | * This positive test ensures that the collection smart pointer is properly initialized and not null when the collection is created. * A null pointer at this stage could lead to crashes or undefined behavior later in the program. * In this case, we perform two checks:   + First, we ensure the smart pointer exists.   + Second, we verify that it does not point to a null memory location. * To take this test further, consider integrating it into a suite of tests focused on memory management. * This will help detect issues like unexpected deallocations or memory leaks early in the development cycle, ensuring the collection is safely managed throughout its lifecycle. |
| **9** | * Here is the code of the unit test. |
| **10** | * This positive test checks whether we can successfully add a single entry to an empty collection. * We start by confirming that the collection is empty before the addition, then proceed to add an entry and validate that the collection size updates accordingly. * A positive result confirms that the insertion logic functions correctly. * To further enhance this test, consider automating it across different data types and collection structures. * This ensures that the collection handles all types of data consistently, reducing the likelihood of bugs in varied use cases. |
| **11** | * Here is the code of the unit test. |
| **12** | * In this positive test, we verify that the maximum size of the collection is always greater than or equal to its current size. * We do this by adding entries to the collection and checking this condition at each step. * A correct implementation should maintain this property, ensuring that there is always space for future entries. * A failure here could indicate issues with memory allocation or incorrect assumptions in the collection’s design. * To take this test further, consider stress-testing the collection by adding entries until it nears its maximum capacity. * This can reveal how the collection handles memory allocation under pressure, providing insights into potential bottlenecks or failure points. |
| **13** | * Here is the code of the unit test. |
| **14** | * This negative test ensures that the at() function correctly throws a std::out\_of\_range exception when trying to access an out-of-bounds index. * We add a set number of entries to the collection, then attempt to access an index beyond the collection’s size, expecting the exception to be thrown. * The negative test verifies that an error is thrown when attempting to access and out of range value. * To further strengthen this test, consider applying it to collections containing user-defined types or extending it to cover various boundary conditions. * This ensures that the exception handling is robust and reliable across different scenarios. |
| **15** | * Here is the code of the unit test. |
| **16** | * This negative test addresses the potential undefined behavior when accessing an out-of-bounds index using operator[]. * Since operator[] does not perform bounds checking, such an access may result in unpredictable outcomes. * To capture this, we use a death test, expecting the program to terminate when attempting to access an invalid index. * The negative test showed that undefined behavior did occur as expected and was caught during the test. * To enhance this testing approach, consider combining runtime tests with static analysis tools. * These tools can help catch such issues before they manifest at runtime, providing an additional layer of protection against undefined behavior. |
| **17** | * Here is the code of the unit test. |
| **18** | * As we examine the DevSecOps pipeline, each stage is designed to integrate security into the development process. * The pipeline begins with 'Assess & Plan,' where we monitor threats and regulatory changes. * In the 'Design' phase, we enforce security best practices, such as those from OWASP. * During the 'Build' stage, we secure our code and components, ensuring everything comes from trusted sources. * Moving to 'Verify & Test,' automated security testing is essential to catch vulnerabilities before code is released. * In 'Transition & Health Check,' we deploy security settings and perform penetration testing. * 'Monitor & Detect' involves continuous security monitoring, using tools like SIEM for real-time detection. * Finally, the 'Respond' phase automates our response to attacks, and in 'Maintain & Stabilize,' we ensure the system remains secure and compliant over time. |
| **19** | * In this slide, we’ll delve deeper into how automation integrates with each stage of the DevSecOps pipeline, focusing on the specific tools and processes that enhance security throughout the development lifecycle. * Assess & Plan:   + Automation begins here with tools like static analysis and dependency checkers, which continuously monitor the threat landscape and regulatory changes.   + These tools help identify potential vulnerabilities early, allowing us to prioritize them in the backlog and adjust our planning to mitigate risks before they materialize. * Design:   + In the design phase, automation ensures security best practices are consistently embedded into the architecture.   + Tools such as automated code linters and security testing frameworks enforce guidelines like OWASP’s best practices, ensuring that our design phase is not just secure but also efficient. * Build:   + During the build stage, automation tools like continuous integration (CI) servers, combined with security plugins, validate the integrity of the codebase.   + These tools check that only components from trusted sources, typically verified through digital signatures or checksums, are included.   + The compiler is also automated in this phase, ensuring that security settings are applied consistently across all builds. * Verify & Test:   + Security automation reaches its peak here, with automated vulnerability scanners, such as SonarQube, and compliance checkers running alongside functional tests.   + These tools are integrated into the CI/CD pipeline to catch security flaws before the code is pushed to production, providing immediate feedback to developers. * Transition & Health Check:   + As we move toward deployment, automation tools configure security settings automatically and perform thorough penetration testing.   + Tools like automated deployment scripts and configuration management tools ensure that the environment is hardened and secure before going live, reducing human error in this critical phase. * Monitor & Detect:   + In production, real-time monitoring is automated using Security Information and Event Management (SIEM) systems.   + These tools continuously analyze logs and system events to detect anomalies and potential threats, providing early warnings and helping to prevent incidents before they escalate. * Respond:   + When a security threat is detected, automation enables quick responses.   + Automated incident response tools can isolate affected systems, block attacks, and revert to stable states as needed.   + This minimizes downtime and exposure, keeping the system resilient against attacks. * Maintain & Stabilize:   + Finally, automation ensures ongoing compliance and stability.   + Tools like automated patch management and baseline configuration checks ensure the system remains secure and aligned with security standards over time, continuously assessing and correcting any deviations. |
| **20** | * In this slide, we explore the risks and benefits associated with either acting on or delaying security integration in our development processes. * Problems:   + The primary issue is that if security is left to the end, it opens up vulnerabilities that can be exploited.   + Retrofitting security into an already developed system is not only more costly but also less effective, often leading to patchwork solutions rather than a cohesive security strategy. * Solutions:   + To counter these problems, it’s crucial to integrate security from the beginning.   + Utilizing automated tools, such as static analysis and vulnerability scanning, helps in early detection of issues.   + Additionally, secure coding practices, including strong encryption and data minimization, ensure that the system is built on a solid security foundation. * Risks of Waiting:   + Delaying security implementation increases the risk of exposure to threats.   + It also leads to higher remediation costs down the line, especially if a breach occurs.   + Moreover, the organization's reputation and customer trust can be severely impacted by security failures. * Benefits of Acting Now:   + By acting now, we can detect and mitigate vulnerabilities early in the development process.   + This proactive approach not only lowers long-term costs but also strengthens our overall security posture, ensuring compliance with regulations and protecting customer data. * By addressing security at each stage of development, we build a more resilient and secure system, avoiding the pitfalls of leaving security as an afterthought. |
| **21** | * In analyzing the current security policy, several key gaps have been identified. * Firstly, the lack of AI integration means that the policy is not fully utilizing advanced technologies for real-time threat detection (Acronis, 2024). * AI can enhance threat detection and response through predictive analytics and automation. * Secondly, the policy is not sufficiently addressing sophisticated ransomware attacks. * According to Smith (2024), incorporating multi-layered defenses, including robust backups and employee training, is crucial. * Lastly, the policy lacks comprehensive management of supply chain risks. * Harris (2024) underscores the need for thorough vendor assessments and stringent security measures to mitigate third-party vulnerabilities. * To address these gaps, it is recommended to integrate AI-driven security solutions for improved threat detection and response, enhance ransomware defenses through updated backup and training strategies, and strengthen supply chain security with rigorous assessments and ongoing monitoring. |
| **22** | * Looking ahead, several standards should be adopted to strengthen future security policies. * Integrating AI-driven threat detection systems can leverage advanced technologies for better security insights and anomaly detection (Acronis, 2024). * Additionally, developing a comprehensive ransomware strategy, including preventive measures and response plans, is essential for managing ransomware threats effectively (Smith, 2024). * Finally, implementing robust supply chain management processes, with thorough security assessments and continuous monitoring, is vital for managing third-party risks (Harris, 2024). * By adopting these standards, organizations can enhance their resilience against evolving cyber threats and better protect their digital assets. |
| **23** | * Here we have the three sources utilized for the recommendation examples and real world uses cases.   *Acronis. (2024). 2024 Cybersecurity Trends: Key steps, strategies and guidance. Retrieved from https://www.acronis.com/en-us/blog/posts/cyber-security-trends/*  *Harris, S. (2024). Cybersecurity Gap Analysis: Brief Overview and Insights. Journal of Cybersecurity, 14(2), 45-60. Retrieved from https://www.threatintelligence.com/blog/cybersecurity-gap-analysis*  *Smith, J. (2024). The 10 Biggest Cyber Security Trends In 2024 Everyone Must Be Ready For Now. Cybersecurity Today, 21(3), 77-85 Retrieved from https://www.forbes.com/sites/bernardmarr/2023/10/11/the-10-biggest-cyber-security-trends-in-2024-everyone-must-be-ready-for-now/* |