# CS 405 Project Two Script

Rajiv Dialani

CS 405

August 17, 2024

Presentation Link: <https://youtu.be/p1E19IqBD8E>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello Everyone,  My name is Rajiv Dialani.  I am here on behalf of Green Pace, who recently made the transition from DevOps to DevSecOps. |
| **2** | To begin this transition, I created a Security Policy for Green Pace. Our security policy is designed to handle the growing complexity and changing threats our organization faces.  This policy ensures that all assets are safeguarded through layered defenses, a strategy known as **defense in depth**.  The defense-in-depth approach involves multiple layers of controls and safeguards, ensuring that even if one layer is breached, others remain intact to maintain security. This layered approach provides more comprehensive protection against a wide range of attack vectors, ensuring that even if one layer fails, others can still provide defense - enhancing the overall resilience of Green Pace’s security infrastructure.  Some of the advantages of a Defense-in-Depth approach:  Redundancy - multiple security controls reduce the likelihood of a single point of failure. Resilience - even if one layer fails, other layers can still provide protection. Adaptability - multiple layers allow for more dynamic and adaptable security postures. Improved Detection and Response - different layers allow for more opportunities to detect and respond to potential attacks. |
| **3** | Threats are categorized based on various criteria. The criteria discussed here emphasize likelihood and severity, both of which impact their priority status.  This structured approach allows you to prioritize security risks effectively based on their severity, likelihood, and the resources required to address them. It is important to note that all these threats are significant, and addressing each of them is essential to ensuring the safety of the application.  Using automation tools, like CppCheck and Parasoft, can simplify and integrate the detection and fixing of these vulnerabilities into your overall defense strategy. |
| **4** | This table shows how each security principle is supported by specific coding standards in the security policy. The 10 security principles are: Validate Input Data, Heed Compile Warnings, Architect and Design for Security Policies, Keep It Simple, Default Deny, Adhere to the Principle of Least Privilege, Sanitize Data Sent to Other Systems, Proactive Defense in Depth, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard.    The coding standards that align with specific principles are placed on the third column of the table. For example, Coding Standard 1 states: Do not declare or define a reserved identifier. This standard maps to two principles: Validate Input Data and Use Effective Quality Assurance Techniques. Using reserved identifiers can introduce risks since they might be interpreted in unexpected ways by the compiler, leading to undefined behavior. By avoiding these identifiers, you ensure that your code behaves as expected, which is a form of validating that your "input data" (in this case, the identifiers you use) is safe and correct. Adhering to coding standards like avoiding reserved identifiers is a part of effective quality assurance. By ensuring that your identifiers are non-reserved, you're applying a preventive measure to maintain the quality and predictability of your code. |
| **5** | These are the 10 Coding Standards that can be found in Green Pace’s Security Policy.  The coding standards are ranked based on its priority. The prioritization system is based on the following criteria:  Severity of Potential Vulnerability - high-risk vulnerabilities are given the highest priority due to the potential damage these vulnerabilities can cause.  Frequency of Occurrence - standards addressing common issues are prioritized because these are frequent sources of bugs and vulnerabilities.  Remediation Cost - cost and effort required to resolve a vulnerability play a significant role. Standards that can be remediated with lower costs and simpler fixes are given lower priority, especially if their impact is less severe.  For example, string handling is a common area where vulnerabilities like buffer overflows occur, leading to severe security breaches. Ensuring strings have sufficient space for character data and the null terminator is critical in preventing such vulnerabilities. The remediation cost is moderate since addressing string-related issues often involves reviewing and refactoring several code sections. On the other hand, using static assertions to test the value of a constant expression helps catch errors early and reduces the risk of bugs. Remediation costs are low because static assertions are straightforward to implement and do not typically require significant code changes. |
| **6** | Encryption at Rest - Data stored on devices, servers, or databases must be encrypted to protect it from unauthorized access. Encryption at rest ensures that sensitive information remains secure and unreadable without the appropriate decryption keys. There are various ways to implement encryption at rest: Full-Disk Encryption, File-Level Encryption, Database Encryption, and Encryption Algorithms. Full-Disk Encryption encrypts the entire disk, making all data on the storage device inaccessible without the proper key. File-Level Encryption encrypts individual files or folders, allowing for more granular control over what is protected. This method is suitable when only certain data needs to be secured. Database Encryption encrypts data within databases, either at the column level or, using techniques like Transparent Data Encryption (TDE). Encryption Algorithms are common algorithms including AES-256 for symmetric encryption, and RSA for asymmetric encryption, depending on the specific use case and performance requirements. Encrypting data at rest is crucial to protect sensitive information from breaches, especially in scenarios where physical access to storage devices is a concern.  Encryption in Flight - Data in transit is particularly vulnerable to interception and tampering. It ensures that data is encrypted during transit, preventing unauthorized interception. The policy requires that all sensitive data being transmitted between systems, devices, or networks must be encrypted using secure protocols like TLS (Transport Layer Security), SSL (Secure Sockets Layer), or IPsec (Internet Protocol Security). These protocols protect data from unauthorized access or modification during transmission by establishing secure communication channels. Encryption in flight typically uses a combination of public and private keys (asymmetric encryption) to establish a secure connection. The public key is used to encrypt the data, while only the corresponding private key can decrypt it, ensuring confidentiality even if the data is intercepted.  Encryption in Use – Refers to protecting data that is actively being processed, such as in memory (RAM) or within an application. This policy addresses the risk of data exposure during computation, ensuring that sensitive information remains secure even while it is being processed or accessed. Encrypting data in use protects sensitive information from being exposed during processing, especially in environments where multiple users or services may have access to shared resources. Techniques like homomorphic encryption allow computations to be performed on encrypted data without needing to decrypt it first. This ensures that sensitive data remains protected throughout the processing stage. Additionally, secure enclaves (e.g., Intel SGX) offer hardware-based isolation, protecting code and data from unauthorized access while in use. |
| **7** | The Triple-A policies are Authentication, Authorization, and Accounting.  Authentication is the process of confirming the identity of users or systems before allowing access to resources. Authentication ensures that only authorized individuals can access sensitive data and systems, reducing the risk of unauthorized access and potential breaches. The policy supports multi-factor authentication (MFA), which enhances security by requiring more than one form of verification. The three main types of authentication factors are: something you know (a password or passphrase), something you have (tokens or smart cards), and something you are (biometrics, such as fingerprints).  Authorization defines what an authenticated user or system can do after access is granted. It involves setting permissions and access levels for different roles within the organization. This policy enforces role-based access control (RBAC) and emphasizes the principle of least privilege. According to the principle of least privilege, users should be given the minimal access required to perform their job functions, reducing the risk of accidental or intentional misuse. Authorization ensures that users only have access to the resources necessary for their role, thereby minimizing the risk of data leakage and unauthorized actions.  Accounting involves tracking and recording user activities and system events. This policy ensures that all actions, including successful and failed login attempts, file access events, and system configuration changes, are logged and monitored. This data is critical for auditing, forensic analysis, and compliance with security regulations. Accounting makes sure that all user actions are recorded, allowing Green Pace to detect and respond to suspicious activities and maintain accountability. |
| **8** | Unit testing is a software testing technique in which individual units or components of a program are tested in isolation. Each test focuses on a single piece of functionality (a unit) without involving other parts of the system. Unit tests are often automated and can be run repeatedly to ensure that new code changes do not introduce bugs (regression testing). Unit testing makes debugging easier since issues are localized to specific units.  The image on this slide shows the Test Explorer on Visual Studio. The image shows 16 unit tests in total, 15 tests passed and 1 failed (on purpose, hence the name always fail). |
| **9** | The image on this slide shows a unit test written using the Google Test (gtest) framework in C++. Positive testing is a type of software testing that is performed to ensure that the system works as expected when valid inputs or expected conditions are provided. The main objective is to confirm that the application functions according to its design and requirements when the input data is correct.  The purpose of this test is to verify that adding five entries to a collection results in the collection having a size of 5. The line add\_entries(5); adds five entries to the collection. The assertion ASSERT\_EQ(collection->size(), 5) checks if the size of the collection is exactly 5 after adding the entries. If the size is not 5, the test will fail. In this case, the test passed, and the collection size is 5. |
| **10** | This is another example of a positive test, as it is verifying that the collection behaves correctly when valid operations are performed.  The purpose of this unit test is to verify that when a collection is resized to hold more elements, the size and capacity of the collection are updated correctly. The assertion ASSERT\_EQ(collection->size(), baseSize + 9); ensures that the size has increased by 9 elements (since it was resized to 10, and the original size was 1). While the assertion ASSERT\_GE(collection->capacity(), baseCapacity); ensures that the capacity has increased to accommodate the new size if needed. It checks that the new capacity is greater than or equal to the original capacity. The test ensures that the collection’s resizing functionality works as intended, preventing potential issues like data loss or inefficient memory usage. |
| **11** | This test is known as a negative test since it aims to ensure the collection appropriately handles invalid operations (like accessing an out-of-bounds index) by throwing an error instead of causing undefined behavior or crashing. It verifies how the system reacts to invalid or erroneous input, ensuring that the behavior is safe and predictable in such cases. Negative tests are crucial for verifying that your application gracefully handles unexpected situations.  The purpose of this test is to verify that an exception is thrown when attempting to access an element outside the bounds of the collection. EXPECT\_THROW(collection->at(11), std::out\_of\_range);: this line verifies that attempting to access index 11 (which is out of bounds since the collection only has 10 elements) throws a std::out\_of\_range exception. The EXPECT\_THROW macro ensures that the correct exception is raised when an invalid operation is performed. It ensures that boundary conditions are enforced, and that the application remains stable even when erroneous operations occur. |
| **12** | This is another example of a negative test because it checks how the system handles invalid or incorrect input, ensuring the system stays stable even when faced with invalid operations.  The purpose of this test is to verify that attempting to access a negative index in the collection correctly throws an exception. This test ensures that boundary checks are implemented to handle invalid negative index accesses, which should raise an out-of-range exception instead of causing unexpected behavior. EXPECT\_THROW(collection->at(-1), std::out\_of\_range);: this line verifies that accessing index -1 (an invalid negative index) throws a std::out\_of\_range exception. The EXPECT\_THROW macro checks that the correct exception is raised, ensuring that negative index accesses are properly handled. |
| **13** | In the DevSecOps framework, security is integrated into every stage of the development and operations process to ensure continuous and comprehensive protection.  This continuous security integration allows vulnerabilities to be detected and addressed promptly, from the planning phase through production, creating a secure and resilient software development lifecycle. |
| **14** | As previously mentioned, in the DevSecOps pipeline, security is seamlessly integrated into every stage of the development lifecycle, ensuring that security measures are not merely an afterthought but an ongoing priority.  The pipeline starts with the **Assess and Plan** phase, where tools like threat modeling software, compliance checkers, and risk analysis apps are used to evaluate security impacts and plan accordingly. In the **Design** phase, automated tools for secure coding and architectural validation are used to embed security early. Moving into the **Build** phase, static code analysis and dependency scanning tools check for vulnerabilities as the software is compiled. In the **Verify and Test** phase, dynamic testing tools (like DAST) and vulnerability scanners automatically assess the code’s ability to handle attacks. The **Transition and Health Check** phase uses penetration testing tools, automated configuration managers, and container security checks to ensure the deployment environment is secure. In the **Production** phase, the focus shifts to real-time monitoring tools like SIEM, analytics engines, and log management systems to automate threat detection and incident management. The pipeline ends with the **Maintain and Stabilize** phase, where baseline assessment tools and automated rollback mechanisms keep the system stable and secure, even during attacks. |
| **15** | When considering the risks and benefits of acting now versus waiting, it’s important to weigh the potential consequences of each choice. Acting immediately to implement the security measures outlined in this strategy provides the benefit of securing systems sooner, reducing the window of vulnerability that attackers could exploit. Immediate action also aligns with best practices for proactive security. The risk, however, may involve resource allocation challenges and potential disruptions to existing workflows if not properly planned.  On the other hand, delaying action could allow more time for thorough planning and testing, but it significantly increases exposure to risks, including breaches, data loss, and reputational damage. The primary issue with waiting lies in the unpredictability of when an attack could occur, and the damage could be irreversible by the time protective measures are finally in place.  One of the most critical steps to take is ensuring that security is integrated into every phase of the development process, rather than leaving it as an afterthought at the end. This approach is central to the DevSecOps model. Waiting until the end to address security can lead to significant risks, including discovering vulnerabilities too late, which may require extensive rework or patching. |
| **16** | To address the potential gaps in the security policy, Green Pace will continuously monitor and update the security policy.  Currently, there are a few recommendations I can give to help strengthen Green Pace’s security readiness:  Focus on User Awareness and Training. A comprehensive policy should include guidelines for regular security training for all team members. This ensures that everyone, from developers to managers, is aware of the latest threats and how to mitigate them.  Extensive Coverage of Emerging Threats. As cybersecurity threats evolve, the policy must be adaptable. Including a section for regular updates and reviews of the security measures can help address new risks and ensure the policy stays relevant.  Strong Emphasis on Incident Response and Recovery. The policy should detail clear steps for responding to and recovering from a breach. This includes setting up a strong incident response plan that outlines the roles, communication channels, and actions to be taken in case of a security event.  Finally, the current list of coding standards can still be improved. The standards should be mapped to more principles so that Green Pace is better covered. |
| **17** | As we conclude, there are four key points that should be adapted to prevent future security problems.  First, **Start with Security**. It's essential to address security right from the beginning of any development or operational process. Waiting until later stages increases risks and remediation costs.  Second, align with **Best Practices and Industry Standards**. By following established guidelines, like those from SEI CERT or OWASP, we can ensure that our practices are well-grounded and proven in the industry.  Third, we must emphasize **Continuous Improvements**. Security is never static. Regular assessments, updates, and training are crucial to stay ahead of emerging threats and evolving vulnerabilities.  Finally, put an **Emphasis on Defense in Depth**. This layered approach ensures that if one defense fails, others are in place to mitigate risks. Implementing security at multiple levels—from the physical environment to network and application security—creates a resilient structure against both internal and external threats.  These points ensure a proactive, rather than a reactive approach, to security. |
| **18** | Thank you for watching. |