**CS 405 Project Two Script Template**

**Colin Aheron 12/11/24**

[**https://www.youtube.com/watch?v=\_-Bfkk7mh2U**](https://www.youtube.com/watch?v=_-Bfkk7mh2U)

|  |  |
| --- | --- |
| **Slide Number** | **Narrative** |
| **1** | Hello, my name is colin aheron and today I will be outlining Green Pace's security policy. |
| **2** | Our security policy focuses on implementing the Defense in Depth principle to safeguard systems through multiple layers of protection. This approach is essential because relying on a single security mechanism can leave systems vulnerable to unforeseen attacks or failures. By incorporating robust input validation, access controls, and secure coding practices at every layer, the policy ensures that even if one layer is compromised, others remain effective. This policy will be used as a guiding framework for designing, implementing, and maintaining secure systems, providing redundancy and resilience against threats while protecting sensitive data and critical resources. |
| **3** | Security standards requiring high attention are those addressing areas with a high risk of exploitation or significant impact, such as input validation, memory management, and authentication protocols. These areas are often targeted by attackers and can lead to severe vulnerabilities like SQL injection, buffer overflows, or unauthorized access if not properly secured.  In contrast, standards with lower attention priority typically focus on less critical aspects, such as minor stylistic coding conventions or guidelines that do not directly impact system security. While still valuable for consistency and maintainability, they pose minimal risk to security and can be deprioritized in resource-constrained scenarios. |
| **4** | Validate Input Data  STR02-C: Sanitize string data passed to complex subsystems to prevent injections.  MEM52-CPP: Handle memory allocation failures properly.  Heed Compiler Warnings  MEM30-C: Avoid dangling pointers to prevent undefined behavior.  ERR55-CPP: Avoid mismatched exception specifications to ensure safe error handling.  Architect and Design for Security Policies  CON54-CPP: Wrap functions like wait() in loops to handle spurious wakeups safely.  OOP50-CPP: Avoid invoking virtual functions in constructors or destructors to prevent resource issues.  Keep It Simple  MEM30-C: Write clear, maintainable code for handling memory safely.  MEM52-CPP: Use straightforward techniques for error handling in memory allocation.  Default Deny  STR02-C: Use whitelists to accept only safe input data.  ERR55-CPP: Ensure exception handling aligns with restrictive exception specifications.  Adhere to the Principle of Least Privilege  OOP50-CPP: Restrict functions to only handle their own resources during construction or destruction.  Sanitize Data Sent to Other Systems  STR02-C: Sanitize string data to ensure it doesn’t carry harmful commands.  MEM52-CPP: Validate memory pointers to ensure safe operations.  Practice Defense in Depth  CON54-CPP: Use proper condition variable handling to prevent race conditions.  MEM30-C: Ensure memory handling is robust to avoid exploitation.  Use Effective Quality Assurance Techniques  MEM30-C: Test pointer safety extensively.  STR02-C: Validate input with automated tools to catch injection vulnerabilities.  Adopt a Secure Coding Standard  Follow all listed standards (e.g., STR02-C, MEM30-C, MEM52-CPP) consistently to prevent known vulnerabilities. |
| **5** | STR02-C: Sanitize string data passed to complex subsystems to prevent injection attacks.  MEM30-C: Avoid dangling pointers to prevent undefined behavior and vulnerabilities.  MEM52-CPP: Handle memory allocation failures properly to prevent crashes and unsafe operations.  CON54-CPP: Wrap functions like wait() in loops to handle spurious wakeups safely.  OOP50-CPP: Avoid invoking virtual functions in constructors or destructors to prevent resource issues.  ERR55-CPP: Ensure exception handling matches specifications to avoid abnormal program termination.  DCL03-C: Use assertions and static checks to validate assumptions at compile time.  MEM30-C: Write clear, maintainable code for safe memory handling.  MEM52-CPP: Use straightforward techniques for error handling in memory allocation.  CON54-CPP: Use proper condition variable handling to prevent race conditions. |
| **6** | Encryption at rest: All stored data, including backups, must be encrypted using industry-standard encryption algorithms to protect against unauthorized access. Data on storage devices (e.g., databases, file systems, backups) must be encrypted at the hardware or software level. Access should be controlled with encryption keys securely managed and rotated regularly. Encrypting stored data protects it from unauthorized access, particularly if physical devices or storage systems are lost, stolen, or compromised. This is especially important for sensitive or regulated data like personal information and financial records.  Encryption in flight: All data transmitted over networks must be encrypted using secure protocols such as HTTPS, TLS, or equivalents. Encryption protocols must be implemented for communication between clients and servers, APIs, or internal systems. This ensures the confidentiality and integrity of sensitive data while in transit, preventing interception or tampering. Data in transit is vulnerable to interception (e.g., man-in-the-middle attacks). Encryption ensures that even if data is intercepted, it cannot be read or altered, protecting confidentiality and reducing the risk of data breaches.  Encryption in use: Sensitive data processed in memory must be encrypted or protected through secure computation methods, such as trusted execution environments (TEEs) or homomorphic encryption. Secure computation techniques should be used in sensitive operations like analytics, computation, or machine learning on encrypted datasets. TEEs or virtualized encryption may isolate data to ensure confidentiality during active processing. Data is at risk while being actively used in memory, as it can be accessed by malware or unauthorized processes. In-use encryption ensures sensitive data is protected during operations, minimizing the risk of compromise in highly sensitive applications like financial or healthcare systems. |
| **7** | Authentication: Ensure all users, devices, and systems are authenticated using strong methods like multi-factor authentication (MFA) or certificates. Implement secure authentication mechanisms for user logins, API calls, and system access. MFA adds a second layer of security, while certificates authenticate systems and devices communicating within the network. Authentication prevents unauthorized access by verifying the identity of users or systems. Strong authentication methods reduce the risk of compromised credentials and protect sensitive resources from unauthorized individuals or devices.  Authorization: Use role-based access control (RBAC) or attribute-based access control (ABAC) to enforce least privilege principles. Define and enforce access rules based on users' roles or attributes. Access policies should be reviewed regularly to ensure users or systems only have access to the resources necessary to perform their tasks. Authorization ensures users or systems can only perform actions or access resources they are explicitly allowed to, minimizing the risk of insider threats, privilege abuse, or accidental misuse.  Accounting: Implement logging and monitoring systems to track user and system activity, and regularly review logs for anomalies or violations. Use centralized logging systems to record actions performed by users or systems. These logs should include timestamps, event details, and user identifiers and should be reviewed periodically or automatically analyzed for suspicious activities. Accounting provides an audit trail to track and analyze user and system behavior, ensuring accountability. This is crucial for detecting security incidents, identifying misconfigurations, and meeting compliance requirements for regulated industries. |
| **8** | Unit testing is crucial for developing secure code. Identifying Coding Vulnerabilities in your Code, Test vulnerabilities and ensure proper behavior in dynamic collections using Google Test for Cpp. |
| **9** | The first test example's goal is to figure out Is the Collection Empty When Created?  Test Description: Verifies that the collection is initialized as empty.  Code snippet is provided. |
| **10** | Can the Collection Handle Out-of-Bounds Access?  Test Description: Verifies exception handling for out-of-range access.  Code snippet is provided. |
| **11** | Does Resizing Work Correctly?  Test Description: Tests resizing both increasing and decreasing collection size.  Code snippet is provided. |
| **12** | Does Clearing Erase the Collection?  Test Description: Verifies that clearing the collection removes all elements.  Code snippet is shown. |
| **13** | Here you can see the output from running the tests, and they pass as expected. |
| **14** | Shown is the devsecops pipeline in a figure 8 pattern, indicating that security is an ongoing process. |
| **15** | Automation should be integrated at key points in the DevSecOps lifecycle to ensure security standards are enforced. In the Pre-production phase, automation can help assess security risks and ensure the backlog includes security tasks. During the Design phase, automated checks can ensure security is part of the design process. In the Build phase, automation can scan code and dependencies for security vulnerabilities and enforce security configurations.  For Verify and Test, automated testing tools can be used to scan for vulnerabilities in the code before it's deployed. In the Production phase, automation continues during the Transition and Health Check stage to validate security settings and test for weaknesses. In the Monitor and Detect stage, automated systems should be in place to track security threats in real-time and alert teams to any issues.  When a threat is detected, automation should be used in the Respond phase to quickly block attacks and recover systems. Finally, in the Maintain and Stabilize phase, automation ensures systems meet security standards and can recover from attacks or failures.  Throughout the process, the entire pipeline should be automated to enforce security standards and provide continuous monitoring, allowing teams to quickly identify and address security risks. This ensures a strong security posture while maintaining operational efficiency. |
| **16** | Risks and Benefits: Acting now to implement solutions provides immediate benefits, such as improved system stability, reduced risk of runtime failures, and increased confidence in the codebase for future development. However, acting now requires time and resources and may introduce temporary regressions if changes are not carefully managed. Waiting may save time in the short term but increases the risk of encountering issues later and accumulating technical debt, making future fixes more complex and costly. |
| **17** | Recommendations: Regular Updates and Reviews: Establish a process for regularly reviewing and updating the security policy to reflect new threats, technologies, and regulatory changes.  Incorporate Emerging Threats: Ensure the policy includes coverage of emerging threats, such as zero-day vulnerabilities, cloud security risks, and IoT security considerations.  Detailed Implementation Guidelines: Provide more detailed guidance for implementing specific security controls, including best practices for access control, data encryption, and incident response.  Role Clarity and Accountability: Clearly define roles and responsibilities for enforcing security measures across different teams and departments to avoid ambiguity.  Training and Awareness Programs: Include provisions for ongoing security awareness training to keep employees informed about best practices and recent security developments.  Integration with Incident Response: Enhance the policy with a well-defined incident response framework that outlines roles, steps for investigation, and communication procedures.  Audit and Compliance Checks: Introduce regular audits and compliance checks to ensure adherence to security policies and standards.  Documentation and Communication: Ensure that the security policy is easily accessible and communicated effectively across the organization, fostering understanding and adherence.  Risk Assessment and Mitigation: Continuously assess and mitigate risks associated with security vulnerabilities, ensuring proactive rather than reactive responses. |
| **18** | Conclusion: Regular security testing, including static and dynamic analysis, ensures code is secure. An incident response plan helps handle breaches effectively. Additionally, managing risks from third-party vendors and adhering to legal and industry-specific security standards are crucial. Continuous security awareness training keeps employees informed about the latest threats, and privacy and data governance standards ensure data protection and compliance. |
| **19** | References. |