# CS 405 Project Two Script Template

Courtney Warner

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Introduction Welcome, everyone. My name is Courtney Warner, and today I will be going over the Security policy built for the company Green Pace |
| **2** | The Green Pace Security Policy ensures all software development follows secure, standardized practices. It defines coding standards, encryption, and access controls for all staff involved in creating or supporting applications. Aligned with the defense-in-depth model, the policy layers protection across systems, from governance to endpoint security, helping safeguard critical assets even if one layer is compromised. |
| **3** | The threat matrix ranks vulnerabilities by likelihood and priority to help us focus efforts on the most critical issues first.  High-priority and likely threats pose the greatest risk to system integrity and must be addressed first. Low-priority and unlikely threats still require mitigation but pose less immediate danger. This matrix helps prioritize secure coding efforts to align with our defense-in-depth strategy. |
| **4** | Here are the 10 core secure coding principles we follow:  Validate input data  Heed compiler warnings  Architect and design for security policies  Keep it simple  Default deny  Adhere to least privilege  Sanitize data sent to other systems  Practice defense in depth  Use effective quality assurance techniques  Adopt a secure coding standard |
| **5** | 1. Do not declare more than one variable per declaration 2. Do not read uninitialized memory 3. Guarantee that storage for strings has sufficient space for character data and the null terminator 4. Exclude user input from format strings 5. Properly deallocate dynamically allocated resources 6. Detect and handle undefined behavior due to violations of program invariants 7. Honor exception specifications 8. Do not access freed memory 9. Ensure that all memory is initialized before use 10. Overload allocation and deallocation functions as a pair in the same scope |
| **6** | Encryption policies cover data at rest, in flight, and in use, ensuring protection no matter where or how data is accessed:  Encryption at rest  This pertains to safeguarding stored data, including files or databases, by employing robust encryption algorithms. This guarantees that sensitive information remains unreachable if the physical or digital storage is compromised.  Encryption in flight  Data transmitted over networks must be encrypted using protocols like TLS or SSL. This protects data from interception, tampering, and eavesdropping while it is moving between systems.  Encryption in use  Data should be protected while being processed, such as by using secure memory allocation or hardware-based secure enclaves. This policy defends against runtime memory inspection and data leaks during operation. |
| **7** | Triple-A policies define who can access what and log every action—enforcing accountability and access control  Authentication  All users must verify their identity through secure authentication methods. This ensures that system access is restricted to verified individuals and minimizes the risk of unauthorized use.  Authorization  Users must be granted access rights based on their role or responsibility. Access should be restricted to the minimum necessary to perform assigned duties, following the principle of least privilege.  Accounting  It is essential to log and monitor all user and system activities. These audit logs are critical for detecting incidents, ensuring accountability, and supporting forensic investigations in the event of a security breach. |
| **8** | This unit test ensures our collection handles empty states safely, validating correct initialization and memory handling  Vulnerability Tested: Data validation and safe memory access  Explanation: This test ensures that your vector correctly allows data insertion when it starts empty. It confirms that your code properly handles initialization and can safely add values without crashing.  Why it matters: Adding to an empty collection is a common scenario where improper memory handling (like null dereferencing or capacity issues) could lead to crashes or undefined behavior.  Security Risk If Not Tested: A failure here could mean memory corruption or denial-of-service (DoS) from crashes when a collection operation is improperly handled. |
| **9** | This test confirms that resizing a collection to zero clears its contents and memory, preventing data leakage  Vulnerability Tested: Data validation and safe memory access  Explanation: This test ensures that your vector correctly allows data insertion when it starts empty. It confirms that your code properly handles initialization and can safely add values without crashing.  Why it matters: Adding to an empty collection is a common scenario where improper memory handling (like null dereferencing or capacity issues) could lead to crashes or undefined behavior.  Security Risk If Not Tested: A failure here could mean memory corruption or denial-of-service (DoS) from crashes when a collection operation is improperly handled. |
| **10** | This bounds-check test verifies that invalid access attempts are safely rejected, protecting against buffer overflows  Vulnerability Tested: Bounds checking (memory protection) Explanation: This test confirms that attempting to access an element outside the valid index range throws an appropriate exception.  Why it matters: This defends against buffer overflow and out-of-bounds access, which are serious vulnerabilities often exploited in cyberattacks.  Security Risk If Not Tested: Without this test, an attacker could deliberately access invalid memory, possibly leading to crashes, data corruption, or arbitrary code execution. |
| **11** | What it Tests  This test is designed to intentionally fail, usually for one of two purposes:  Confirm that the test framework reports failures correctly  Simulate a code path that would violate an expected condition, such as incorrect logic, failed assertions, or unsafe state. |
| **12** | Automation embeds security into every phase of the DevSecOps lifecycle.  In pre-production, it supports secure architecture, vulnerability scanning, and verifying trusted builds.  In production, it enables continuous monitoring, automated threat response, and scheduled penetration testing, helping teams stay secure without slowing development. |
| **13** | We use Parasoft, Cppcheck, and Clang to automate design reviews, static analysis, and compliance testing |
| **14** | Risks of waiting:  Data breaches and loss  Damage to customer trust and brand reputation  Increased financial and legal liability  Difficulty recovering from multiple or persistent threats  Benefits of early action:  Reduce vulnerability exposure  Strengthen infrastructure and defenses  Save on future remediation costs  Streamline security and testing through automation  What steps should be taken:  While early investment in security may incur initial costs and planning, it is far less expensive than the fallout from data breaches. Proactive security reduces long-term risk, improves resilience, and builds customer trust. |
| **15** | Regularly review and update the policy to close emerging gaps.  Schedule annual assessments by an external security firm.  Apply security measures early in development to reduce risks.  Define concrete actions for the “Respond” phase of post-production.  Specify which tools and repositories should be used in the build phase.  Clarify how policy enforcement will be handled beyond automation. |
| **16** | In conclusion: By integrating security throughout the development lifecycle, this policy reinforces a proactive DevSecOps strategy. Secure coding standards, static analysis tools, and automated enforcement measures help detect vulnerabilities early, reduce risk, and ensure compliance. Through continuous testing, monitoring, and incident response readiness, our team can maintain a strong security posture and deliver resilient software with confidence. |
| **17** | These are my references and thank you very much for listening. |