# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | This presentation covers the Green Pace Security Policy. I am Bruce Gaudet, the developer of this security policy and presentation. The purpose is to outline the critical security measures and standards necessary to protect Green Pace’s data, systems, and operational integrity. This will serve as a detailed overview for stakeholders to understand both the policy framework and its implementation. |
| **2** | This slide introduces the concept of “Defense in Depth” as the core principle behind the Green Pace security policy. It explains that the goal is to align all developers under a single, consistent set of secure coding practices so that as the team grows, everyone maintains the same focus on limiting risks. The narrative emphasizes that Defense in Depth works by layering multiple security measures across different parts of the codebase and system architecture, so if one control fails, others will still protect the system. The slide also highlights that the policy sets clear, proven security standards and encourages proactive identification and mitigation of weaknesses before they are exploited, ensuring that applications remain resilient against potential threats. |
| **3** | The threats faced by the Green Pace security environment are organized into four categories based on likelihood and severity. Likely threats include unvalidated or unsanitized inputs, which remain a common pathway for vulnerabilities such as SQL injection, cross-site scripting, and command injection. These risks require strict input validation and server-side sanitation aligned with established coding standards. Unsafe C-style string handling and missing bounds checks can result in buffer overflows or denial-of-service conditions, making the use of safe APIs and enforced length checks essential. Additional likely issues include mixing signed and unsigned types, missing error handling that allows silent failures, and sensitive data in logs that must be masked and access restricted.  Priority threats represent immediate and high impact risks that must be remediated without delay. These include broken authentication or authorization, least-privilege gaps, and insecure query building practices leading to account takeover, lateral movement, or remote code execution. Exposure of secrets in code, configuration, or logs can lead to full environment compromise, making secret vault storage and key rotation critical. Other priority risks involve insecure file I/O and path traversal vulnerabilities, as well as weak or missing transport security, which must be mitigated with strict TLS configurations and enforcement of secure cipher suites.  Low Priority threats are not immediately dangerous but can degrade code quality, maintainability, and overall security posture if ignored. These include style inconsistencies, dead code, outdated comments, verbose logging of non-sensitive data, and non-critical assertions in production builds. While these issues may not lead directly to exploitation, addressing them during refactoring and UX validation helps reduce potential long-term exposure and operational inefficiencies.  Unlikely threats require specialized conditions or capabilities, making them rare but still worth documenting. Examples include physical attacks on encrypted devices, advanced side channel attacks, tampering with supply chains, and collusion between multiple insiders to bypass layered security controls. These risks are typically mitigated through encryption, isolation, provenance checks, code signing, and continuous monitoring.  Overall, the threats matrix provides a prioritized view of risks, mapping each to relevant standards and remediation strategies, ensuring that the team focuses first on the most probable and damaging issues while maintaining long-term resilience against less likely but potentially severe threats. |
| **4** | This slide presents the “10 Principles” that form the foundation of secure software development and system protection. Each principle is tied to a corresponding coding standard, ensuring that best practices are not only understood conceptually but are also implemented in a measurable, actionable way. The narrative begins with Validate Input, emphasizing that all user supplied data must undergo thorough validation before processing, which prevents common vulnerabilities like SQL injection or cross site scripting. Least Privilege follows, ensuring that users and processes operate with only the permissions necessary to perform their tasks, reducing the risk of privilege abuse.  The third principle, Fail Securely, focuses on maintaining security controls during system failures, preventing sensitive information leaks and avoiding insecure fallback states. Separation of Duties ensures critical responsibilities are distributed across multiple roles, preventing insider threats or unintentional abuse. Defense in Depth advocates for layered security controls, ensuring multiple barriers exist to stop or slow down an attacker.  Secure Defaults prioritizes configuring systems with the strongest security settings from the start, minimizing reliance on administrators or developers to apply protections later. Keep Security Simple reminds developers to avoid overcomplicated code and architectures that might inadvertently create vulnerabilities. Minimize Attack Surface targets reducing exposed components, limiting what potential attackers can interact with.  The ninth principle, Establish Secure Failsafes, ensures fallback mechanisms preserve confidentiality, integrity, and availability even in abnormal conditions. Finally, Continuous Monitoring and Updating highlights the importance of regularly scanning for vulnerabilities, addressing new threats, and keeping all systems patched to maintain resilience over time. |
| **5** | Moving on to our next slide, we come to the coding standards that form the backbone of our secure development process. This slide outlines ten specific rules that take the broad security principles we discussed earlier and translate them into concrete actions in our code. For example, SQL Injection Prevention and Memory Protection are right at the top because they address two of the most critical and common vulnerabilities. By using parameterized queries instead of string concatenation and ensuring memory is properly handled, we eliminate major attack vectors before they have a chance to cause damage.  I then explain that Input Validation and String Correctness are equally important, as they stop unsafe or malformed data from entering our system in the first place. These standards require us to check every piece of external input for the right format, range, and length, and to manage strings in a way that avoids memory and logic issues. Medium-priority items like Secure File Access and Logging and Auditing focus on keeping sensitive information safe, whether it is stored in files or recorded in logs, and making sure that no unauthorized party can gain access to them.  Finally, I note that even the lower-priority standards, such as Data Values, still play a role in keeping our applications predictable and stable. By validating operands and avoiding risky operations like divide-by-zero, we ensure the application behaves consistently under all conditions. The severity, likelihood, and priority ratings on this slide give us a clear roadmap for which risks to address first, while making sure none are overlooked. Together, these standards ensure every part of the codebase actively contributes to the overall security and reliability of the system. |
| **6** | here we are looking at our encryption policies which are essential for protecting data throughout its entire lifecycle. When we talk about encryption in flight, we mean that all data traveling between clients, servers, and third-party services must use secure communication protocols like TLS 1.2 or higher. The use of valid and regularly renewed certificates ensures that our data is protected against eavesdropping or man-in-the-middle attacks during transmission.  Encryption at rest is all about securing data stored in databases, file systems, or backups. Here, we rely on strong algorithms like AES-256. The keys for this encryption are stored securely in a dedicated key management system and are rotated regularly, which means even if physical storage is compromised, the data remains protected.  Encryption in use covers the time when sensitive data is actively being processed in memory. Techniques such as in-memory encryption or secure enclaves help maintain protection. Applications must limit the time decrypted data lives in memory and wipe it securely after use to prevent unauthorized access or memory scraping.  Finally, policy enforcement ensures that our encryption methods remain up to date with industry standards. Keys are managed under strict access controls, and configurations are tested during security audits. This ongoing process is critical for ensuring the effectiveness of our encryption strategy. |
| **7** | Moving on to our next slide, I will explain our Triple A policies which stand for Authentication, Authorization, and Accounting. For Authentication, all systems must require secure user authentication before granting access. Passwords must meet strong complexity requirements and be stored using salted hashing algorithms such as bcrypt. Multi factor authentication is required for privileged accounts and remote access. These mechanisms are tested regularly to prevent vulnerabilities like brute-force attacks. For Authorization, we follow the principle of least privilege using Role Based Access Control, ensuring users only have the permissions necessary for their job functions. Any changes to access require documented approval, and permissions are reviewed quarterly to prevent privilege escalation. For Accounting, all authentication and authorization events must be logged securely, including login attempts and data access actions. Logs must be tamper proof, meet compliance standards, and be monitored for suspicious activity. Finally, for Policy Enforcement, we review and update these policies regularly, and non-compliance results in immediate access revocation and investigation. |
| **8** | This test verifies that when a valid username and password combination is entered, the system authenticates successfully and grants access to the user dashboard without any SQL errors. This confirms that the authentication system is functioning correctly for legitimate credentials. The result for this test is passed. |
| **9** | Next, we have our second unit test, which ensures the system properly denies access when an incorrect password is entered with a valid username. The system should display an error message, confirming that authentication is working as intended and that no sensitive database information is exposed during the process. This test result is also passed. |
| **10** | We then have our third unit test, which checks protection against SQL injection attacks. In this case, we input a malicious string such as “admin' --” in the username field. The system should use parameterized queries to reject the attempt, returning an authentication failure without revealing any database details. This test confirms our defense against injection attacks, and the result is passed. |
| **11** | For Test 4, we simulate a SQL injection attempt in the password field by entering a malicious string like anything' OR '1'='1 along with a valid username. The expectation is that the login process will fail, demonstrating that SQL injection prevention is not only applied to the username field but also to the password parameter. The system successfully rejected the attempt without exposing sensitive details, confirming that both parameters are adequately protected. |
| **12** | In Test 5, we leave the username and password fields blank to confirm that the system enforces required field validation. This negative test should result in an error message and a refusal to authenticate. The system behaved as expected by preventing access when no credentials were provided, which helps block accidental submissions and basic automated attacks. |
| **13** | For Test 6, we check how the system handles special characters in the username field by entering a value like user!@#. This positive test ensures that valid usernames containing special characters are processed correctly without triggering SQL injection or syntax errors. The system accepted the valid credentials and authenticated the user successfully, demonstrating robust input handling. |
| **14** | The Automation Summary slide outlines our continuous integration and deployment security process. We begin in the pre production phase with assessment and planning, ensuring security is factored into design, build, and verification. We conduct vulnerability scanning, secure configuration checks, and penetration testing before moving into production. Once deployed, the system undergoes continuous monitoring and detection for anomalies, rapid response to threats, and maintenance to restore security baselines after incidents. This cycle ensures ongoing security and compliance throughout the software lifecycle. |
| **15** | Moving on to our next slide, here we will focus on the tools that support our DevSecOps pipeline. The DevSecOps pipeline brings security into every stage of the software development lifecycle. From planning to deployment and monitoring, security is built into the process rather than being treated as something to add later. This begins with secure design and coding practices, followed by continuous integration where code is compiled, tested, and scanned for vulnerabilities. Automated tools help detect issues early so they can be addressed before moving forward. During deployment, we use controlled environments with secure configurations, and once the system is live, monitoring tools help identify and respond to threats in real time.  Alongside this pipeline, we also make use of a variety of external tools that enhance both efficiency and security at different stages. For example, static application security testing tools help spot insecure coding patterns during the build process. Dynamic testing tools simulate attacks during integration and testing to reveal vulnerabilities that only appear at runtime. Dependency scanners check third-party libraries for known issues, while monitoring platforms track logs, detect anomalies, and trigger alerts during operations. While this slide does not display the visual diagram, the explanation here reflects the full range of tools and how they integrate with the pipeline to provide thorough security coverage. |
| **16** | This slide addresses the current challenges in the security approach, highlighting gaps in automation and monitoring that leave windows of exposure. It outlines solutions such as fully automating security scans, enforcing standard remediation timelines, and integrating advanced monitoring. Acting now is shown to reduce immediate risk, strengthen compliance, and prevent recurring vulnerabilities, while delaying action can increase the chance of successful attacks and costly consequences. The strategy’s weaknesses include inconsistent runtime threat detection and lack of alignment between teams. Risks of doing nothing include enabling zero day exploits and missing compliance requirements. The recommended steps are to automate testing, formalize incident response, and establish clear KPIs for continuous improvement. |
| **17** | The current security policy leaves several vulnerabilities unaddressed, especially in runtime protection and anomaly detection during active operations. While encryption and coding standards are defined, there is no formalized process for dependency management or patch prioritization, and multi factor authentication enforcement is incomplete. Incident response protocols lack detailed timelines and role clarity, which can slow containment during breaches. Security audits are not conducted on a fixed schedule, and centralized monitoring is not fully implemented, reducing visibility into coordinated attacks. To close these gaps, the policy should be expanded to include operational monitoring, strict authentication enforcement, structured audit schedules, vendor security evaluations, and supply chain risk assessments. |
| **18** | To strengthen the security posture, the organization should adopt established frameworks and standards. The OWASP Application Security Verification Standard can ensure consistent and measurable security controls. The NIST Cybersecurity Framework provides a structured approach for identifying, protecting, detecting, responding, and recovering from incidents. Enforcing ISO/IEC 27001 ensures a formal security management system, and the CIS Critical Security Controls help prioritize actions against common attack vectors. Standardizing secure coding practices will further reduce vulnerabilities and improve long term resilience. |
| **19** | Here are my references used within my presentation. |