# CS 405 Project Two Script

**Please see the attached link for the video “https://youtu.be/itCy0VbQtug”**

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Security Policy Presentation  Developer: Merrik Wright    Hello everyone, and thank you for joining me. My name is Merrik Wright, and today I’ll be presenting the Green Pace Security Policy. This presentation outlines the key standards, practices, and principles we aim to use to protect our systems and codebase from evolving threats. By implementing these policies, we aim to keep all developers aligned under a unified, secure development strategy. Let’s dive in. |
| **2** | The Green Pace Security Policy was designed to formalize the secure development practices our team already uses every day. As our team grows, it’s critical to stay aligned through a structured, and layered approach to security. This is the core idea behind the defense-in-depth strategy. It means applying security controls at every stage of development, not just at the end. From coding and unit testing to deployment and monitoring, each step includes protections to reduce vulnerabilities and strengthen our overall resilience. |
| **3** | This slide uses a risk matrix to evaluate our most significant vulnerabilities. On the top left, we see SQL injection, a critical vulnerability due to its high likelihood and severe impact if left unpatched. Buffer overflow is less likely today due to modern memory protections but still carries a high impact. Numeric overflow, on the other hand, is both low likelihood and lower impact, making it a lower priority. This matrix helps us visualize which issues need immediate attention and guides our secure coding priorities moving forward. |
| **4** | These are the 10 fundamental security principles that form the foundation of our policy. Each principle is backed by at least one coding standard we’ve implemented. For example, minimizing the attack surface is supported by our buffer overflow and input validation standards. Least privilege and secure defaults guide how we handle user access and authentication. These principles ensure our policies are proactive, thorough, and aligned with proven security strategies. |
| **5** | Our 10 coding standards are ranked in priority based on the level of risk they pose and how easily they can be exploited. At the top of the list are buffer overflows and SQL injection, both known for enabling serious attacks. Next, we address input validation and hardcoded credentials, which can expose systems if overlooked. Lower on the list are standards like error handling and code readability. While they may seem less urgent, they play a key role in long-term security and debugging. Prioritizing this way ensures we tackle the most dangerous vulnerabilities first without ignoring foundational quality. |
| **6** | Green Pace enforces encryption across all stages of data handling. For encryption in flight, we rely on TLS to protect information as it moves across networks. For data at rest, such as files or database entries, we use strong encryption standards like AES-256 to ensure protection even if physical access is compromised. Finally, for encryption in use, data is decrypted only when absolutely necessary and kept in memory for as little time as possible. This three-part strategy reduces the risk of data leaks at every point in the system. |
| **7** | The Triple-A framework helps ensure that only the right people can access the right resources, and that we know exactly what they’re doing when they do. Authentication verifies identity, we use strong password policies and enforce multi-factor authentication for added protection. Authorization determines what each user can access, using clearly defined roles. Finally, accounting logs all access and system activity so we can audit usage and detect unusual behavior. This framework provides a strong foundation for secure access control across our systems. |
| **8** | Our first unit test simulates an SQL injection attack using a common payload: ' OR '1'='1' --. If the application fails to sanitize input, this could trick the system into returning all user records. In our secure implementation, the test confirms that parameterized queries reject this input and the unauthorized data is not retrieved. This is one of the most important tests for web applications, especially those with login forms or search fields that query a database. |
| **8.a** | This test simulates a malicious SQL injection using a classic ' OR '1'='1' -- payload. If the application fails to properly sanitize input, this could expose or manipulate database results. The test confirms that injected input does not return unauthorized data, helping validate the security of query construction. |
| **8.b** | This test uses a safe, expected username input: "john\_doe". It verifies that the system handles regular user data properly and returns a clean response. This confirms that legitimate users receive appropriate results while preventing unintended access or errors. |
| **9** | This diagram represents our DevSecOps pipeline, showing how security is integrated throughout the software development lifecycle. Automation plays a key role in each phase. In the pre-production stage, tools like static analyzers and unit test frameworks automatically scan for vulnerabilities during coding and building. During testing, we apply automated test suites like Google Test to validate that known attack vectors are handled securely. Once in production, automated monitoring tools detect unusual behavior, while alert systems and intrusion detection platforms provide real-time awareness. Automation ensures that security is continuous, repeatable, and not left to chance. |
| **10** | The DevSecOps pipeline integrates security practices directly into the development workflow. At the coding stage, we use static code analysis tools like Cppcheck to catch issues such as buffer overflows and unsafe input handling. During the build stage, secure repositories and dependency scanners ensure we’re not introducing vulnerabilities from third-party packages. When testing, we automate vulnerability checks using tools like Google Test for logic and unit validation. In production, log monitoring and alert systems like SIEM tools help detect real-time threats. By embedding these tools throughout the pipeline, we ensure security isn’t an afterthought, it’s a continuous, automated process. |
| **11** | There are clear benefits to implementing security policies and testing practices now. We reduce the attack surface, harden critical systems, and catch flaws before they can be exploited. But the strategy isn’t perfect. One weakness is the potential over-reliance on tools. Automated testing may not catch all logical vulnerabilities or poor design decisions. If we wait too long to address these gaps, small issues can evolve into serious security incidents. To avoid that, we need to continue combining automation with manual reviews, ongoing training, and regular threat assessments. These extra steps ensure our defenses are both proactive and adaptable. |
| **12** | Although our current policy addresses many major vulnerabilities, there are still important areas we need to improve. For example, there’s no formal process for managing third-party libraries, which can pose hidden risks if they’re outdated or compromised. We also rely heavily on automated testing, but without regular manual code reviews or penetration testing, we may miss critical design flaws. Another gap is developer training, not all team members have received consistent guidance on secure coding principles. To close these gaps, I recommend adding dependency scanning to our build pipeline, scheduling periodic security audits, and offering secure coding workshops to keep our team sharp and up to date. |
| **13** | Looking ahead, there are several key standards Green Pace should adopt to stay ahead of security risks. First, incorporating OWASP Secure Coding Guidelines will ensure developers are following widely accepted best practices. Second, aligning with the NIST 800-53 framework can help guide system-wide security measures and compliance. We should also adopt software composition analysis tools to automatically monitor third-party libraries for known vulnerabilities. Finally, we should standardize threat modeling and schedule routine penetration tests. These steps will help us stay proactive, consistent, and resilient as our applications evolve. |
| **14** | These are the resources I used to support the tools, standards, and testing methods throughout this project. They include official documentation for Google Test, Cppcheck, OWASP secure coding practices, NIST standards, and other DevSecOps tools referenced in the presentation. |