# CS 405 Project Two Script

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | **Title Slide**  Hello, my name is Darrell Walker, and this is my security policy presentation for Green Pace. Today, I’ll be walking you through our team’s coding policies, external testing methods, and recommendations to strengthen security for both development and architecture moving forward. |
| **2** | **Overview – Defense in Depth**  Our security policy follows a Defense in Depth strategy.  Instead of relying on one barrier like a firewall, we layer protections across every level — physical, cloud, network, applications, and user access. This way, even if one control fails, others are still in place to protect our systems and data.  We also enforce strict standards, encryption, automation, and secure coding to reduce vulnerabilities as our systems grow.  By taking this layered approach, we support long-term resilience, compliance, and safe development across all of Green Pace. |
| **3** | **Threats Matrix**  Our Threats Matrix identifies the most serious vulnerabilities by how likely they are to happen and how much damage they could cause.  High-priority issues like buffer overflows and improper input validation need to be fixed as quickly as possible.  Less-urgent threats, like unsafe logging or missing comments, are still tracked — but with lower priority.  This approach keeps our team focused on the most critical risks first, while still watching all potential threats. |
| **4** | **10 Principles**  Our security policy is built on ten guiding principles that support secure software development.  The first is Defense in Depth, where we use multiple layers of security to protect our systems.  Fail-Safe Defaults means systems deny access by default unless explicitly allowed.  With Least Privilege, every user and process only gets the permissions they need — nothing more.  Economy of Mechanism reminds us to keep designs and code as simple as possible so they’re easier to secure.  Complete Mediation requires that every access attempt is checked against the security policy.  Open Design tells us not to hide our security; we use strong, well-tested mechanisms.  Separation of Duties means critical tasks are split up to prevent errors and abuse.  Least Common Mechanism limits shared components to reduce cross-contamination between processes.  Psychological Acceptability reminds us that security must also be easy to use so people follow the rules.  And finally, Work Factor is about making it as difficult as possible for an attacker to succeed.  Together, these principles help us make sure that all of our coding standards support both safety and long-term resilience. |
| **5** | **Coding Standards**  Our coding standards highlight the top 10 secure coding practices that we follow.  These standards are ranked by risk level — the most dangerous vulnerabilities come first.  For example, buffer overflows are at the top because they’re both high-risk and easy for attackers to exploit.  Each standard also includes real examples of good and bad code, along with risk scores and tools we use to catch these issues early.  By using these standards, we focus on the most serious threats first and write safer, more secure code. |
| **6** | **Encryption Policies**  Our encryption strategy protects data at every stage — in flight, at rest, and in use.  For data in transit, we use HTTPS with TLS 1.3 to keep data safe as it moves between clients and servers.  For data at rest, we encrypt files, databases, and backups with AES-256, a strong, industry-standard method.  And for data in use, we protect data in memory too, making sure it stays secure while it's being processed.  These practices help us meet data protection requirements and defend against threats, both inside and outside our systems. |
| **7** | **Triple-A Policies**  Our security policy follows the Triple‑A model — Authentication, Authorization, and Accounting.  Authentication is where we verify who a user is, using strong methods like hashed passwords and multi‑factor authentication.  Authorization controls what each user can do — making sure they only have access to the data and systems they actually need.  Accounting tracks every action in the system, creating logs that help us spot suspicious behavior and maintain compliance.  Together, these three elements protect our data and ensure that all access is properly controlled and fully traceable. |
| **8** | **Unit Testing Vulnerability – Improper Input Validation**  For this unit test, we’re focusing on Improper Input Validation, which happens when an application doesn’t properly check or sanitize user input before using it.  This is dangerous because attackers can inject harmful commands — for example, SQL injection.  **Test #1: Valid Input Should Pass**  In this test, we pass a string like "; DROP TABLE users;--" into a validateInput() function.  The goal is for validateInput() to return false so we know the dangerous input is blocked and logged.  By testing this way, we make sure our system catches dangerous input and protects itself — stopping potential attacks before they do any harm. |
| **9** | **Test #2 – Special Characters Should Be Rejected**  In Test #2, we check whether the system rejects numeric-only input in a field that should contain text.  For example, we pass the string "123456" into our validation function, where we’d normally expect a user’s name.  The goal is for validateInput() to return false, ensuring the system doesn’t accept numbers in a text-only field.  By enforcing these validation rules, we make sure that the application only accepts proper data and stays secure against unexpected input. |
| **10** | **Test #3 – Numeric-Only Input Should Be Rejected**  In Test #3, we check whether the system rejects empty input properly.  Empty inputs can cause unpredictable behavior — like skipping validation or even crashing the program — so we need to catch them early.  Our test passes an empty string into the validation function  The function is expected to return false, which means blank input is successfully blocked.  This kind of test ensures that our validation logic prevents incomplete or empty data from entering the system — keeping the application stable and safe. |
| **11** | **Test #4 – Empty Input Should Trigger an Error**  In Test #4, we’re verifying that the system correctly accepts valid input.  This is a positive test — we use a properly formatted name like "JohnSmith" and pass it into our validateInput() function.  The function should return true — which confirms that safe, properly formatted data passes validation without issue.  By including this test, we make sure our validation logic allows good input to flow through the system as expected. |
| **12** | **Automation Summary**  Our automation strategy integrates security into every part of our DevSecOps pipeline.  During development, tools like Cppcheck and Visual Studio scan our code to catch bugs, buffer overflows, and unsafe input before they become problems.  In the build stage, we run automated unit tests that verify expected behavior and catch any broken or vulnerable code.  Finally, at deployment, vulnerability scanners and quality checks make sure our code is compliant and ready for a secure release.  By automating these steps, we catch issues earlier, reduce manual errors, and maintain strong security practices without slowing down our process. |
| **13** | **Tools**  This slide highlights the security tools we use in our DevSecOps process to catch problems as early as possible.  Cppcheck is a lightweight static analysis tool for C++ that catches issues like buffer overflows, memory leaks, unused variables, and syntax errors as we write code. It integrates into our development environment so we get real-time feedback.  Visual Studio Static Analyzer is built right into our IDE and identifies risky coding patterns — like null-pointer dereferences and uninitialized memory — before they can cause serious bugs.  Finally, all these tools tie into our CI/CD pipeline, so every commit is scanned automatically before it’s merged. If something doesn’t pass, the build is blocked until the issue is fixed.  By using this combination of static analysis and automated checks, we reduce human error and ensure secure, consistent coding practices across the entire team. |
| **14** | **Risk and Benefits**  Addressing vulnerabilities early has clear benefits — it’s faster, cheaper, and much safer than waiting until after a breach or crash.  By applying input validation, encryption, and static analysis throughout the development process, we reduce the chance of serious exploits and improve the overall stability of the system.  The key benefits include preventing critical issues like injection and buffer overflows, boosting confidence during audits, and cutting long-term maintenance and repair costs.  On the other hand, there are risks to consider — if we delay or ignore these protections, vulnerabilities may be discovered and exploited in production. That can make fixing issues much more expensive and time-consuming and could lead to serious legal or reputational damage.  By implementing a layered security strategy, supported by automation and thorough testing, we minimize these risks and build a more stable, secure, and sustainable system. |
| **15** | **Recommendations**  Finally, let’s look at our key recommendations moving forward.  First, we need secure logging — this means making sure we never store passwords or other sensitive data in log files.  Next, peer reviews for high-risk code will help catch security flaws before they go live.  It’s also important to improve training and documentation so all developers follow secure coding practices.  Finally, we’ll expand automated tests to catch edge cases, unsafe inputs, and other conditions tools might miss.  These improvements will help shift us from reacting to issues after the fact to preventing them up front — making Green Pace’s software safer, more stable, and easier to maintain long term. |
| **16** | **Conclusion**  To wrap up, our security policy lays a strong foundation by addressing core vulnerabilities, enforcing safe coding practices, and integrating automation across the development pipeline.  But security is never “done.” To stay ahead of evolving threats, we need to continuously improve.  That means regularly reviewing and updating our standards as new risks emerge, investing in ongoing developer training, and leveraging new tools for real-time detection and anomaly analysis.  Ultimately, security is an ongoing process — by staying proactive with prevention, education, and automation, we can better protect our systems, our data, and our people well into the future. |
| **17** | **References**   * Seacord, R. C. (2013). Secure Coding in C and C++. Addison-Wesley. * OWASP Foundation. (2024). Top 10 Web Application Security Risks. https://owasp.org * Microsoft Docs. (2024). Cppcheck Documentation. https://learn.microsoft.com |