# CS 405 Project Two Script Template

**Cameron Kinney**

**6/22/25**

**Project Two: Security policy presentation**

[**https://www.youtube.com/watch?v=5VVzI-O2AYo**](https://www.youtube.com/watch?v=5VVzI-O2AYo)

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone, my name is Cameron and today I will be walking you through the Green Pace security policy guide. This policy formalizes the secure coding practices that the team has already applied and expands them into a consistent, scalable strategy. Today I will walk you through the policy’s structure, the threat landscape that was addressing, and how we can ensure that security is embedded into every stage of our development cycle |
| **2** | The diagram on this slide shows what defense-in-depth looks like. This diagram goes beyond firewalls and antiviruses, as its about security built into every part of the workflow from research and development to how we store and protect data. It is important to consistently write clean code, limit access, or thinking about where data goes. But as the team starts growing, it is essential to maintain good habits official. This policy lays out the clear layers of defense so everyone can be on the same page regardless of where they are working. |
| **3** | For this slide we have the major threat categories that we consider while building the security policy. Each box shows how likely a threat is and how much priority we give it when designing secure systems. Staring with priority, these are the areas where we have the most control over things such as writing secure code, having solid software design, doing proper threat modeling, and reviewing each other’s work. When looking at the likely section, we are dealing with common stuff that includes malware, phishing, outdated software, and mistakes that users make. These can fall outside our control, but it is still important to protect against them through layered security. Low priority threats include physical theft or weak passwords. They are still important but less of a focus in the moment especially since we can address some of them with access controls and training. Lastly unlikely threats are things such as day-zero attacks or infrastructure failure. We can’t predict these, but our defense-in-depth helps to keep damage to a minimum if something extreme were to happen. This matrix is important as it can help figure out what parts of the code and systems need the most attention and shapes every standard within the policy. |
| **4** | This next slide shows the ten principles that we value for our security policy. First, we focus on input validation which involves making sure that we block SQL injections, cross site scripting and anything else that tries to sneak in. Then for the basics, like fixing compiler warnings early on so they don’t become exposed in production. Principle three is about designing for security and the beginning of the project. If you wait until the end to consider something, it becomes harder to fix. As for number four keep it simple as complex code can hide complex problems. The fifth principal default deny is thought of as if something isn’t explicitly allowed, it shouldn’t be happening in the first place. Combined with the next principle least privilege where no user or process has more access then its required and also shrinks attack surface. Number seven is a reminder that data sent out can be as dangerous as what you send in. This is why we sanitize outbound data. After this comes defense-in-depth which is all about layering security so there is always a backup if one thing fails. Number nine focuses on quality including testing, peer reviews, and other stuff that can help Catch early vulnerabilities. The last principle includes secure standards like SEI and CERT to avoid the biggest vulnerabilities. |
| **5** | On this slide I have ranked the ten coding standards from most to least critical based on their threat severity, exploitability, and real-world impact. As number one is SQL injection. This ranks highest because injections are severe and high-probability threat that leads to full database compromise. Exploiting SQL injection is often easy if attackers know what to look for which makes it the highest priority to prevent. Next is string correctness as unsafe string operations can cause buffer overflows which can lead to arbitrary code execution, system crashes or complete system compromise. Third is memory protection as poor memory management can expose programs to crashes or malicious code execution. Following this is input sanitization as unsanitized input is a key path to cross-site scripting, injections and data corruption. These are considered common vectors for attackers in user-facing system. Structured exception handling comes next. Without proper exception handing applications can crash or behave unpredictably which can result in denial-of-service attacks. File handling security is also critical as improper file access handling can expose sensitive data and enable path traversal which is dangerous in system-level applications. Next is threat safety as race conditions and synchronization issues can lead to unpredictable behavior, data corruption, or deadlocks. These are harder to detect but pose serious reliability and security concerns. Number eight is input range validation as this prevents divide-by-zero errors and numerical overflows. This can be a serious issue but are typically less catastrophic than injections or memory flaws. Type safety comes in at number 9 as violations can lead to subtle bugs or data loss. While not always directly exploitable, they can become dangerous when combined with other vulnerabilities. Lastly, safe assertions are unsafe assertions in production code that can create crashes, but their impact is usually limited in scope and less dangerous overall. |
| **6** | This next slide focuses on encryption policies which cover the three key states of data: in flight, at rest, and in use. First, encryption in flight protects data as in moves between systems, especially over the internet. It is important to use TLS 1.2 or higher for any HTTP or HTTPS communication. It is also important to mandate the use of strong cipher suites, disables outdated protocols such as SSL or older TLS versions and to use secure email gateways and VPNs for transmitting sensitive information. Next is encryption as rest as this secures data that is stored on a physical or virtual media This policy enforces encryption of all sensitive data using AES-256 or newer and should also require full-disk encryption on servers and endpoints, and database-level encryption. Encryption. Keys must be securely stored using hardware security modules (or HSMs for short) and key management systems (KMS), and key rotation should be handled consistently to reduce risk. Lastly encryption in use protects data while its being processed in memory. By limiting sensitive data exposure through secure coding practices, it can avoid writing sensitive values to logs or memory dumps. Using memory-safe languages and tools that restrict memory access, consider confidential computing in high-security environments that encrypts memory within secure enclaves during processing. |
| **7** | This slide covers triple-A security policies which stand for authentication, authorization, and accounting which are all essential for managing access to systems and data securely. First, authentication is the process which verifies the identity of users or systems before granting access to sensitive information. The goal of this is to ensure that only legitimate users can log in. This policy mandates that the use of strong authentication methods such as multi-factor-authentication, biometric verification, or facial recognition alongside with secure login credentials. The goal of this it to prevent unauthorized access from compromised accounts or external threats. Next is authorization which defines what authenticated users are allowed to do. This policy enforces role-based access control where permissions are assigned strictly based on the role of the user within the organization. This reinforces the principle of least privilege and ensures users have access only to the resources necessary for job necessities. Access control lists are used to clearly define and restrict access to specific resources and maintain granular security boundaries. Lastly comes accounting or auditing where logging and monitoring user activity ensures accountability and compliance. This includes keeping detailed audit logs, tracking changes to critical databases, and using monitoring tools to detect unauthorized behaviors. Effective accounting provides a clear trail of user actions, audits, and more in the event of a security incident. Altogether, these triple-A-policies create a secure, traceable, and role-aware access control strategy. |
| **8** | In this next slide specifically dealing with SQL injections, in this test I created we use normal login credentials such as the username ‘admin’ and a secure password. The login succeeds and shows that the system correctly handles safe, expected input. This confirms basic functionality is working as intended. |
| **9** | On our next slide is an example of a test SQL injection attempt using ‘ OR ‘1’=’1 as the username. This is a classic example of a bypass technique. The system does not sanitize the input, and the login succeeds which is a failure in terms of security. An attacker could use this to gain unauthorized access. |
| **10** | This next test uses the same injection attempt, but the backend uses a prepared statement. The input is treated as plain text. The login fails, which shows the injection didn’t work. This shows the system is protected when using prepared statements properly. |
| **11** | Lastly, we try a destructive SQL injection with admin’: DROP TABLE user; --. If this is not blocked, this command can delete critical data. In this example, the table is deleted and proves vulnerability is real and highlights the need for strict input validation and error handling |
| **12** | This diagram outlines the DevSecOps process lifecycle that divides into pre-production and production phases with clear stages such as planning, building, verifying, transitioning, monitoring, and maintaining. Each phase includes specific tasks aimed to ensuring continuous integration, security, and operational resilience throughout software development and deployment. |
| **13** | Going further into this, we start with access and plan where we identify threats, understand regulatory changes, and prioritize security fixes early. In the design phase we apply secure design principles such as OWAPS Top 10 and adopt test-driven development to build secure architecture from the beginning. During the build stage we use trusted repositories and verified open-source components to reduce supply chain risks. In verify and test we run vulnerability scans, compliance checks, and automated security tests to detect issues before deployment. In transition and health check, we securely configure systems and conduct penetration testing before release. The monitor and detect stage focus on collecting logs, using SIEM tools, and detecting threats in real-time. We then hit maintain and stabilize by comparing systems against baselines and having recovery plans ready for any future incidents. Lastly, we respond by blocking attacks, rolling back harmful changes, and isolating threats to minimize impact. Together these stages make security a continuous, automated process and not just an afterthought. |
| **14** | It is important to highlight where the current strategy still has gaps. Firstly, there’s no centralized system for tracking or detecting threats which can lead the organization blind to emerging attacks. Second, inconsistent enforcement of secure coding standards undermines overall security posture. There is also a lack of clear rollback or incident recovery plan, which increases response time when a breach occurs. To strengthen defense, we must implement automated policy enforcement tools within the CI/CD pipeline, mandating training for all developers on threat modeling, and prioritizing patching high-risk issues, especially SQL injections. Lastly we must establish well-defined response plan that includes rollback procedures to ensure faster recovery.  Slide 15: several gaps have been identified, first there are no runtime protections in place, second the policy lacks formal threat modeling tied to specific features or modules, third there’s no defined process for vetting third-party dependencies such as open-source libraries or external APIs. Additionally, the current incident response plan is weak with no documentation on rollback strategy or clear response steps in the event of a breach occurring. There is also an issue with admin tools as not all enforce multi-factor authentication. Lastly, logging and monitoring are inconsistent with no centralized log management or SIEM system in place.  Slide 16: Looking forward here are some standards that should be considered to prevent recurring security issues. First, aligning development practices with the OWASP Top 10 will directly address the most critical and common vulnerabilities. NIST’S secure software development framework should be followed to embed security into all phases of the software development lifecycle. The CWE/SANS Tops 25 shares continuously updated list of the most dangerous coding errors to watch out for. Enforcing static and dynamic analysis standards using automated tools ensures vulnerabilities are caught early in CI/CD. In closing we have explored how secure coding practices, DevSecOps integration, and strong authentication and logging policies are essential in today’s threat landscape. By identifying current gaps, performing hands on SQL injection testing, and aligning with industry standards, we are building a culture of proactive security. Thank you for your time and have a great day |