# CS 405 Project Two Script Template

Jade Pineda

6/20/2025

Project Two – Security Policy Presentation

<https://youtu.be/r3e-QYR04qg?si=c8jOr8rLhT0n_iiS>

| Slide Number | Narrative |
| --- | --- |
| 1 | Good morning, everyone. I’m Jade Pineda, and today I’ll walk you through Green Pace’s new Secure-Coding and DevSecOps Policy. Think of it as the playbook that lets us release features quickly and stay secure all the way from the first commit to production. |
| 2 | Let me start with why this policy exists. Our last audit showed each team solving the same security problems in different ways, sometimes missing them entirely. Moving to DevSecOps means we can’t bolt on security at the end anymore.  The wheel you see here is the classic defense-in-depth model. Network, host, and application layers are important, but the code layer at the center is the last line of defense. Our policy locks down that inner ring by standardizing ten CERT-based rules, mandatory encryption, and a Triple-A framework, all enforced automatically in the pipeline. In short, it turns secure coding into the default path rather than an after-thought. |
| 3 | Here’s the quick-scan view of those ten rules.  Priority in the upper-right are the nightmares, SQL injection and a trio of type-safety bugs. They’re easy to exploit and cause the most damage, so the pipeline refuses to merge code that triggers any of them.  In the Likely box we have memory-safety issues: double-free and resource leaks, problems that show up often and take services down when they do.  Low-priority items, like stripped assertions, rarely lead to breaches but can hide subtle logic errors, so we still keep an eye on them.  Finally, Unlikely risks, deadlocks and secret-logging, are severe yet hard to hit. ThreadSanitizer and log-scrubbing rules catch those before they ever reach production. This ranking tells every team where to focus first if time is tight. |
| 4 | All ten rules map back to ten larger principles. I won’t read every line, but notice two things:  First, each principle shows up more than once, ‘Validate Input Data’ alone drives three separate standards. Second, no rule stands on its own. For example, our resource-management standard is there because of ‘Keep It Simple’ and ‘Adopt Secure Coding Standards.’  Laying it out this way makes it clear that these rules aren’t random, they’re specific, measurable ways to live out best practices we’ve already agreed on |
| 5 | Here’s the top-ten list we’ll live and die by.  Number one is SQL injection, still the fastest route from public input to database compromise, so any trace of string-built SQL blocks a merge.  Next come three memory-safety rules: signed/unsigned mix, out-of-bounds indices, and unsafe string APIs. They’re easy to introduce and can escalate to remote-code execution.  Standards five and six tackle classic C++ pitfalls, double-free and resource leaks, that crash services and burn uptime.  Exception hygiene and concurrency follow; they’re serious, but they show up less often and are caught by sanitizers.  Assertions stripped in release and sensitive-data logging round out the list, lower impact but still worth fixing early.  The order isn’t arbitrary: we scored every rule on severity, likelihood, and remediation cost. Higher combined scores bubble to the top so teams always know what to tackle first. |
| 6 | We encrypt data in all three states, storage, transit, and memory.  At rest, every database file, backup, and log archive is sealed with AES-256-GCM. Keys stay in AWS KMS, rotate every ninety days, and never leave an HSM.  In flight, we accept only TLS 1.3 connections. External endpoints force HSTS, and internal microservices authenticate each other with mutual TLS, closing any gap that a man-in-the-middle could exploit.  In use, we go a step further: passwords, tokens, and private keys remain encrypted even in RAM using libsodium’s crypto\_secretbox. The moment we’re finished, we zero that memory to protect against cold-boot or crash-dump attacks.  Together, these controls make sure data is unreadable to anyone who isn’t explicitly authorised, no matter where it happens to live. |
| 7 | Our Triple-A stack covers who you are, what you’re allowed to do, and how we prove it later.  Authentication is handled by OAuth 2.1 with PKCE. Every admin role also requires multi-factor authentication. We issue ES-512-signed JWTs that expire after fifteen minutes, so stolen tokens have a very short shelf life.  Authorization relies on strict role-based scopes. Any new scope must be documented in a pull request; if code references an unknown scope, the build fails. The policy is default-deny, no role, no access.  Accounting ties it all together. Every login, configuration change, and database mutation is recorded as an immutable JSON event. We hash-chain those events to prevent tampering, index them in Elastic for real-time search, and ship daily digests to Glacier so we have seven years of provable history.  With these three layers in place, we always know who did what, when, and with which rights, exactly what auditors and incident responders need. |
| 8 | Our first test targets INT30-C, signed vs unsigned length checks. We send a normal value of 128 to prove the function works, then two negative values to prove it fails safely. GoogleTest’s EXPECT\_DEATH confirms the guard clause fires, and clang-tidy now reports zero signed-unsigned warnings. That closes the door on this entire class of overflow bugs. |
| 9 | Next, we attack our data layer. A valid ID of forty-two returns a single record. Passing the classic one-equals-one payload or a full DROP TABLE command now triggers a bind-error before the SQL ever reaches the server. SonarQube’s injection rule and a ZAP active scan both come back clean, proving the binder is doing its job. |
| 10 | This test proves our RAII wrapper stops double-free bugs. The positive run frees memory once; Valgrind is silent. When we intentionally free twice or skip the wrapper after an exception, EXPECT\_DEATH fires and Valgrind flags a double-free. With RAII in place both tools report clean, so MEM31-C is satisfied. |
| 11 | For concurrency, we test our lock-ordering rule. When both threads acquire locks in the same order, the test passes. When we reverse the order in one thread, ThreadSanitizer instantly reports a potential deadlock. By enforcing a consistent order we eliminate the issue and CON31-C passes in CI. |
| 12 | Under the hood we use GoogleTest for core logic, Catch2 for parameterized data checks, and libFuzzer to hammer new code with random inputs. Nightly builds run ThreadSanitizer and Valgrind to catch races and leaks we might miss during the day.  We’re already at eighty-two percent line coverage and ninety-four on critical paths. Next quarter we’ll add property-based tests for numeric edge cases, mutation testing with Mull, and an expanded SQLMap payload set. That pushes us toward ninety-percent coverage and full protection of every critical path. |
| 13 | Now here is the DevSecOps Diagram which i will get to in a moment |
| 14 | Here’s a quick tour of our DevSecOps toolchain and where each piece lives.  Design: As developers type, IDE plug-ins surface any CERT violations  immediately, so most issues never leave the workstation.  Build: When code hits a pull request, clang-tidy and Cppcheck run in seconds. If they spot a high-priority rule break, the merge stops right there.  Verify: A successful build triggers our test suite, GoogleTest for logic, libFuzzer for edge-case chaos, and a SonarQube quality gate to catch coverage drops or critical findings.  Pre-production: We stress the deployed stack with Chaos Monkey, then sweep it with OWASP ZAP to confirm no web vulnerabilities slipped through.  Production: Runtime Application Self-Protection sensors feed logs into Elastic SIEM; if we see an attack pattern, the WAF blocks it and the SIEM opens a ticket automatically.  This layered tooling means every stage, design, build, verify, pre-prod, and prod, has at least one automated control watching for security drift, giving us continuous confidence in every release. |
| 15 | Here’s the cost-benefit snapshot.  Problems: Right now we have inconsistent coding controls and a heavy reliance on manual reviews. That means repeat audit findings and expensive rework.  If we act now and roll out the policy, every critical flaw is blocked before it reaches main, we pass the next audit confidently, and we fix bugs when it’s ten times cheaper, during the pull-request stage.  If we wait six months, the risk curve flips. Breach probability rises, retro-fitting security controls costs about ten times more developer hours, and we risk failing the audit or delaying product launches.  One gap remains: our supply-chain bill of materials is still a manual checklist. We’ll close that in Q4 by generating SPDX SBOMs automatically and adding a software-composition-analysis gate.  The takeaway is simple: implementing now delivers measurable benefits and avoids exponentially higher costs later. |
| 16 | Based on everything we’ve reviewed, here are the clear recommendations to close the gaps.  First, our current SBOM process is still manual, which makes it easy to miss something. Automating SPDX generation fixes that.  Second, we need software composition analysis, SCA , to flag known vulnerabilities in open-source libraries early.  Third, secure coding isn’t formally part of onboarding yet. We should bake that into our dev team’s first week.  Fourth, our CI pipeline should actively enforce security rules, not just report them. That means builds should fail when a high-severity flaw slips in.  And last, we should expand our audit logs to catch failed login attempts. That’ll help with forensic analysis if anything goes sideways.  Each of these steps is focused, actionable, and realistically doable within the next quarter. |
| 17 | To wrap things up, here’s where we need to go moving forward.  First, the OWASP Top 10 isn’t just a list , it should be our baseline for every code review and design discussion.  Second, the NIST Secure Software Development Framework gives us a structured way to build security into every phase of development , not just tack it on at the end.  Third, we need to standardize how we assess vulnerabilities. Using CVSS scoring gives us a shared language for severity , whether we’re fixing it, reporting it, or tracking it.  Fourth, automation should be our default. Static analysis tools and dependency checkers aren’t just helpful , they’re essential.  And finally, we should document everything with SPDX and generate SBOMs by default. That gives us visibility and traceability if anything ever goes wrong.  By locking in these standards, we’re not just playing defense , we’re setting ourselves up to grow without compromise.  Thank you! |