Kain Mason

Southern New Hampshire University

CS 405: Secure Coding

October 12, 2025

**Critical Reflection**

“Don’t leave security to the end” means treating security as a design constraint from the first requirement, not as a cleanup step before release. When teams wait until the end, they tend to patch symptoms (like sanitizing one risky field) instead of removing root causes (like weak data models, missing authorization, or unsafe defaults). I have made that mistake before: I focused on getting a feature to work, then tried to harden it later. By then, the architecture baked in assumptions that were expensive to unwind. Shifting left changes the default behaviors. If I pick safe types up front (std::size\_t for sizes, std::string instead of raw buffers), validate inputs where data enters the system, and require parameterized database access, I am removing entire classes of bugs—not just fixing instances. It also improves maintainability: simple types and predictable error handling make reviews better and testing easier. Finally, building security into the pipeline gives me evidence for audits because it runs on every commit.

**Secure Coding Best Practices**

To prevent threats, I will align my work with the DevSecOps flow and map concrete actions to each stage. In Plan, I will write security acceptance criteria with each story and sketch quick STRIDE-style threats for high-risk features. In Create, I will default to safer language features—fixed-width integers where size matters, std::string or string\_view instead of C strings, and RAII/smart pointers for lifetime safety. In Verify, the CI pipeline will run static analysis (SAST) and dependency checks (SCA) for every commit and block merges on High/Critical issues; unit tests will include negative cases for malformed inputs, missing auth, and boundary values. Before release, I will add sanitizers (ASan/UBSan), fuzz key parsers, and run integration tests with least-privilege identities against a real database to catch permission problems. At Release, I will sign artifacts, publish an SBOM, and enforce infrastructure-as-code policy checks to start services with TLS 1.3 and restricted network access. For Prevent/Detect, I will use short-lived service tokens, centralized structured logs (no secrets), and alerts for suspicious events such as repeated 403s or blocked SQL patterns. In Respond/Adapt, I will track vulnerabilities, expire waivers on a timer, and revise standards when incidents or tests expose gaps.

**Example for Project Two**

For Project Two, I will include “Security Unit and Integration Tests as CI Gates.” The idea is to make a feature done only when its security behaviors are proven by tests, not when the happy path works. For a REST endpoint, I will add input validation tests that send out-of-range, oversized, and malformed payloads and assert 400 Bad Request with a safe error body (no stack traces). I will add authorization tests that call the endpoint without a token, with the wrong scope, and with the correct least-privilege scope, asserting 401/403 for the first two and success only for the last. I will include a guard against SQL injection by verifying the repository layer uses parameterized queries, and I will assert that structured logs capture who/what/when without secrets. These tests run automatically on every pull request; if any fail, the merge is blocked. That makes security intrinsic rather than something bolted on at the end.