**CS 405 – Secure Coding  
Module 8  
8-2 Journal: Portfolio Reflection  
Marissa Lanza  
Southern New Hampshire University  
Instructor: Wasim Alim  
Date: August 13, 2025**

Adopting a Secure Coding Standard (and Not Leaving Security to the End)  
At the start of this course, I treated security like a final checklist—something to worry about after the code “worked.” That changed once I actually coded to a standard. Using SEI CERT C/C++ rules in my exceptions and resource-management work gave me practical guardrails: more consistent error handling, safer memory habits, and calmer behavior under stress. Shifting left paid off fast: catching format-string issues with static analysis or swapping string concatenation for parameterized queries in the SQL injection activity was cheap early and painful if postponed. Writing to a standard from day one also cooled off code reviews—disagreements turned into “what does the rule say?” instead of opinions *(Seacord et al., n.d.; National Institute of Standards and Technology [NIST], 2022)*.

**Evaluating Risk and the Cost–Benefit of Mitigation**:

Thinking in terms of risk helped me pick my battles. I kept asking: What could go wrong? How likely is it? How big is the blast radius? The injection was high likelihood and high impact, and the fixed statements were low cost with a huge payoff. Input validation and tight error messages (no stack traces) were similar “high-value, low-effort” wins. Encryption took more judgment: TLS in transit is non-negotiable; at-rest encryption adds overhead but is worth it for anything sensitive. Static analysis and dependency checks are my low-cost, always-on safety net; fuzzing and DAST are the deeper dives I save for higher-risk spots.

That lines up with “likelihood × impact × cost,” not “do everything” *(NIST, 2012; OWASP Foundation, 2021)*. Day to day, that looks like strong identity *(MFA, short-lived tokens),* locking down which services can talk to which (least-privilege APIs, role-scoped DB users), ongoing health checks (device posture, anomaly alerts), and solid logs so I can show exactly what happened *(NIST, 2020).* Even in small class projects, I could apply it: separate app and DB roles, use the principle of least privilege in connection strings, parameterize all queries, and choose defensive config defaults. Zero Trust isn’t a product—it’s a design constraint that quietly shapes every choice.  
  
**Implementing and Recommending Security Policies:**

Write it down and ship with it. Keep a short secure coding standard mapped to SEI CERT and OWASP basics (input validation, output encoding, auth, session management, error handling, logging). Make it part of the definition of done.

**Shift left in CI/CD**. Run SAST, secret scanning, and dependency/SBOM checks on every PR; block merges on critical findings; add a brief security checklist to reviews.

**Threat-model early (lightweight).** Before building, list assets, sketch trust boundaries, brainstorm abuse cases, and record the decisions.

**Zero Trust by default**. Use least-privilege service accounts, per-environment secrets, short-lived credentials, and network/service segmentation; prefer secure defaults in config.

**Protect the data**. Use TLS everywhere; encrypt sensitive data at rest; store less when you can; set clear retention and deletion paths.

**Observability and response**. Emit security-aware, secret-free logs to tamper-resistant storage; alert on auth anomalies; keep a simple triage runbook.

**People and practice**. Offer short, recurring training tied to real findings; run blameless post-mortems; track a few leading indicators (time to fix high-risk vulns, percent prepared-statement coverage, percent of PRs with a security review).  
  
**What Changed for Me**:

I came in thinking security was mostly tools. I’m leaving seeing it as habits and constraints you apply from the first line of code to the last commit. The assignments—especially SQL injection prevention, exception handling, encryption, and static analysis—nudged me to make safer choices earlier, not after the fact. That mindset shift is the real portfolio piece for me.

Looking ahead. I still want to answer a couple of practical questions: How do we measure the ROI of added controls without slowing delivery (which leading indicators matter most)? And where’s the sweet spot between Zero Trust friction and developer velocity (like short-lived creds and rotation) on a small team? I’d also like to try automated CI guardrails that block raw SQL and insecure crypto by default and then measure how they affect PR cycle time.  
  
**References**

National Institute of Standards and Technology. (2012). Guide for conducting risk assessments (SP 800-30 Rev. 1). <https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final>

National Institute of Standards and Technology. (2020). Zero trust architecture (SP 800-207). <https://csrc.nist.gov/publications/detail/sp/800-207/final>

National Institute of Standards and Technology. (2022). Secure software development framework (SSDF): Recommendations for mitigating the risk of software vulnerabilities (SP 800-218). <https://csrc.nist.gov/publications/detail/sp/800-218/final>

OWASP Foundation. (2021). OWASP Top Ten: The ten most critical web application security risks. <https://owasp.org/www-project-top-ten/>

Seacord, R. C., et al. (n.d.). SEI CERT C and C++ coding standards. Software Engineering Institute. Retrieved August 13, 2025, from <https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>