Richard Seabridge
CS 405 Secure Coding
Southern New Hampshire University
06/29/2025

Portfolio Reflection

In traditional software lifecycles, security is often an afterthought—bolted on at the end of development when there is little time or budget to address vulnerabilities properly. However, our course readings (for example, McGraw's *Software Security: Building Security In*, and the SEI CERT C++ Coding Standard) consistently argue for shifting security left, embedding secure coding standards from day one. By adopting guidelines such as the CERT rules or the OWASP Top 10 mitigation strategies at the outset, teams establish clear expectations for input validation, error handling, and resource management. Early adoption not only prevents costly rework but also cultivates a security-minded culture: developers learn to write resilient code, security tasks become part of each sprint, and fewer defects slip into production.

Risk Assessment and Cost–Benefit Analysis

Effective security decisions hinge on evaluating risk and weighing the cost-benefit of mitigation. NIST SP 800-30 lays out a formal risk assessment process: identifying threats, gauging likelihood, and estimating impact. For example, adding encryption to data at rest may incur performance overhead; thus, teams perform a cost-benefit analysis to decide if the protected asset's value justifies that overhead. In our Module Three readings, we practiced threat modeling on a sample web service and saw how "over-protecting" low-risk endpoints can waste resources, whereas under-protecting critical assets invites exploitation. By classifying assets (e.g., customer PII vs. logging metadata) and mapping corresponding controls, organizations can allocate security budget where it yields the greatest reduction in overall risk.

Embracing Zero Trust

Richard Seabridge
CS 405 Secure Coding
Southern New Hampshire University
06/29/2025

The zero-trust paradigm—"never trust, always verify"—represents a radical departure from perimeter-based defenses. Readings from Week 6 (Kindervag's original white paper) emphasize that in modern, cloud-native environments, attackers often bypass traditional firewalls; internal nodes must authenticate and authorize every request. Zero trust mandates microsegmentation, mutual TLS authentication, and continuous monitoring. From a developer's perspective, it means embedding identity-aware SDKs (e.g., AWS IAM roles or OAuth2 flows) directly into code, and instrumenting services to validate tokens and permissions on each call. Zero trust not only reduces lateral movement in case of breach but also raises the bar for attackers, since trust is never implicit.

Implementing and Recommending Security Policies

Security policies translate high-level goals into concrete rules and procedures. Our CS 405 readings suggested a layered policy framework:

1. Coding Standards Policy: Mandate use of a vetted standard (e.g., SEI CERT C++), specify static analysis tools (like Coverity), and require code reviews focused on security.
2. Encryption Policy: Define when and how to use TLS (e.g., enforce TLS 1.2+ for all external communications) and store keys per industry best practices (e.g., hardware security modules).
3. Credential Management Policy: Prohibit hardcoded secrets, require vaulting (e.g., HashiCorp Vault), and enforce periodic rotation.

Richard Seabridge
CS 405 Secure Coding
Southern New Hampshire University
06/29/2025

4. Incident Response Policy: Outline roles, escalation paths, and communication plans for responding to detected vulnerabilities or breaches.

As a recommendation, each policy should be reviewed semi-annually, tied into the organization's change management process, and accompanied by developer training sessions. Automating policy adherence via CI/CD pipelines—blocking merges when critical findings appear—ensures ongoing compliance and fosters accountability.

Conclusion

Embedding secure coding standards early, rigorously assessing risk, embracing a zero-trust mindset, and translating strategy into enforceable policies are not disparate tasks but interconnected pillars of a robust security posture. Together, they shift security from a "check the box" activity at the end of development to an intrinsic part of how software is conceived, built, and maintained—ultimately reducing defects, lowering long-term costs, and protecting both users and organizations from evolving threats.