Elyssa Creed

8-2 Journal: Portfolio Reflection

Throughout this course, I have come to appreciate the importance of adopting a secure coding standard from the very start of development rather than treating security as an afterthought. Secure coding standards—such as the SEI CERT C/C++ guidelines—provide developers with concrete rules for writing safer code, for example validating all inputs, avoiding use-after-free errors, and preventing SQL injection with parameterized queries. By embedding these standards into coding practices and automated pipelines (e.g., static analysis with Cppcheck or SonarQube), teams ensure that vulnerabilities are caught early and consistently. This “shift-left” approach reduces the time and cost of fixing security flaws compared to discovering them in later testing or production stages.

A disciplined risk assessment process complements secure coding by helping teams prioritize which vulnerabilities to address first. For each identified issue, we must evaluate its severity (impact if exploited), likelihood (probability of occurrence), and remediation cost. High-severity flaws that are easy to exploit and inexpensive to fix—such as unchecked buffer copies—should be remediated immediately, while lower-risk issues might be scheduled into the regular release cycle. Conducting periodic threat modeling sessions and cost-benefit analyses ensures that limited resources are spent where they will yield the greatest reduction in overall risk.

The zero trust philosophy fundamentally reshapes how I think about security by assuming that no component—whether inside or outside the network perimeter—can be trusted by default. In practice, this means enforcing strong authentication and authorization checks at every boundary, using principles like least privilege, micro-segmentation, and continuous verification. Adopting zero trust at the application layer involves techniques such as Just-In-Time access, strict API gateways, and real-time anomaly detection, so that even a compromised user or service cannot move laterally through the system unchallenged.

Finally, robust security policy implementation ties these elements together by defining clear, enforceable rules for development and operations. A comprehensive policy should mandate the use of secure coding standards in every code review, require automated security testing in the CI/CD pipeline, enforce zero trust controls in the infrastructure, and specify regular risk assessments. Policies must also assign accountability—identifying who is responsible for triaging findings, remediating vulnerabilities, and updating standards as threats evolve. By institutionalizing these practices in our DevSecOps workflow, we create a defense-in-depth posture that continuously hardens our applications against emerging threats without sacrificing agility.