**Module 8: Journal**

Zane Milo Deso

Southern New Hampshire University

CS-405: Secure Coding

**Journal**

**Introduction**

In the evolving threat landscape of modern software development, secure coding is no longer an afterthought; it’s a baseline expectation. Throughout this course, I’ve developed a broader understanding of how adopting proactive security strategies like zero trust, cost-benefit risk mitigation, and codified security policies shape robust and resilient applications. This journal reflects each of these core ideas, integrating them into how I now approach development work both conceptually and practically.  
**Adopting a Secure Coding Standard Early**

One of the most fundamental lessons this course emphasized is that security must be baked in early, not sprinkled on at the end. Waiting until the final stages of development to implement security features is both inefficient and dangerous. Vulnerabilities identified late in the pipeline require costly refactoring and may have already allowed unsafe patterns to become deeply embedded in the codebase. I’ve come to view secure coding standards not as constraints, but as scaffolding; tools that help ensure consistency, minimize risk, and make security automatic rather than reactive. In my recent full stack project, this approach guided decisions such as centralizing token storage using Angular’s BROWSER\_STORAGE and structuring the API to consistently require authentication headers. A secure coding standard forces me to think intentionally at every layer; from form validation to error messaging to session handling.

**Evaluating and Assessing Risk and the Cost-Benefit of Mitigation**

Risk assessment in software isn’t about achieving perfect security; it’s about prioritizing resources based on realistic impact and likelihood. I’ve learned to apply cost-benefit thinking when deciding what security mechanisms to implement. For instance, not every endpoint needs multi-factor authentication, but protecting admin routes with strong credential validation and JWT expiry enforcement has a high return on investment. By mapping assets, potential threats, and attack vectors, I can weigh whether a mitigation is worth its performance cost or development time. This mindset also plays into budgeting security for future projects; balancing usability with defense and making intentional compromises when needed.

**Zero Trust**

The zero-trust model has fundamentally changed how I view systems architecture. The idea that no user, device, or component is implicitly trustworthy, even if inside the perimeter, is powerful. In our final project, this meant ensuring that even authenticated users only had access to routes and features explicitly granted their role. It also meant validating every request server-side, even if the client seemed “safe.” Zero trust reinforces a defensive programming posture: assume breach, validate constantly, and limit exposure. The practical side of this showed how we validated tokens, sanitized inputs, and avoided reliance on client-side logic for security decisions. Zero trust is more than a buzzword; it’s a mental model that drives architectural discipline.

## **Conclusion**

In summary, this course challenged me to adopt a security-first mindset, not just technically, but strategically. I now see secure development as an ecosystem of early standards, risk-aware decisions, zero trust architecture, and enforceable policies. Each part reinforces the others. As I continue building applications, these practices won’t just sit in the background; they’ll inform every decision I make about data, users, and code behavior. Ultimately, adopting this mindset makes me not only a more capable developer but also a more accountable one.