CS-405 Secure Coding

Justin Phillips

Southern New Hampshire University

Portfolio Reflection

Throughout CS-405, I strengthened my understanding of secure coding often by getting to troubleshoot technical issues, but also by reading through and understanding various vulnerabilities and writing my own security policy using my research into common vulnerabilities. Prior to this course, I was making an active effort to make my software as crash resilient as possible, but wasn’t aware of the utilities of certain crashes or that I could produce custom exceptions and use asserts to validate assumptions, which are helpful tools in building actually robust software. The difference between these two approaches – Crash resilience as opposed to robust and high quality code – Is that I test my assumptions, and do more research into the underlying issues of why my programs fail where they do, and what to do to actually handle these failures in a healthy way. What this means is that rather than less meaningful approaches such as catching and trying again or assuming a placeholder value to continue to run with invalid data, which is tempting to a less security-minded developer, I know to test my functions, deconstruct failed to initialize objects, and add more security to reading and writing files.

I’ve also adopted the industry standard DevSecOps workflow, which should certainly be in employment as a foundational philosophy for all development projects. The DevSecOps cycle involves integrating security mindedness pre-emptively during planning, resource research, and then throughout the development process and into the long-term support. There are a number of ways this workflow impacts my understanding of my career as a developer. Having completed two courses on full-stack development, I can see how security ties in. Ports should be closed with whitelists for approved processes, deny by default and least privilege should be adopted, and in webapps MFA should be required to access technical workings or user data. Ultimately, secure coding properly will prevent the data breaches that void peoples’ trust in your software, such as AT&T’s repeated breaches of user data. It will also protect you and your company from costly fines by not complying to industry standards in security.

My portfolio is full of investigations into security best practices. Overflow protection involves a level of foresight into whether an overflow or underflow is possible for different data types. Resisting these involves anticipating exceeding the maximum or minimum value of the type and handling this gracefully, either by throwing an error or coming up with a superior solution (i.e. switching a char array for a dynamically allocated string object). SQL injection is a complicated animal to me, where I was tempted to add ‘\’ literals before the elements of command-interpretables that were included in the received strings to be queried. I discovered that while a number of specific counters to an injection can be considered, the easiest option is to parameterize a query for the SQLite engine, causing an anticipation of the valid query. I also interfaced with the robust Google Unit Testing framework, which allows for various tests to be automated in a similar fashion to JUnit testing in Java (though I much prefer C++ for syntax, in my eyes perceived efficiency improvements, and for not being Java). I also interfaced with C++Check and got familiar with static code analysis. This reminded me of reverse software engineering to a degree, since that involved analyzing executables and their assembly commands and the stored data values in hex and string / char types, and was also a bit of an insight into how a program actually performs on a computer. However, the C++Check analyzer was smarter than I am, or at least more knowledgeable on certain bugs and better at predicting them. It showed me a list of vulnerabilities in the code, empowering me to resolve them and build a better program.

My research into vulnerabilities led to discovering bugs, misassumptions, and skill-errors to watch out for as a professional and also taught me how to target and resolve these issues. They can be read about in greater detail in the security policy I’ve drafted, but that document is not all-conclusive. Finding a vulnerability database in NVD.NIST.GOV felt a lot like discovering OWASP for dependency checking. A large library made itself available for investigating the things that can go wrong, and how to prevent those things. Reading through vulnerabilities on that platform was like reading through a catalogue of sometimes obvious, sometimes less obvious errors. Ultimately, NVD.NIST.GOV is another tool to add to my kit of competent developer research techniques.