University of Southern Mississippi

Gulf Park Campus

Software Security Paper

Jackie Diep

CSC 424 G001

Professor Michaela Elliott

17 April 2021

Software security is a well acknowledged but not entirely understood concept in a student developer's path. Secure software is software with limited vulnerabilities in such a way that malicious attacks or random accidents do not cause system failure or expose sensitive data. In today's online world, insecure software has caused massive issues all the way up to the government level, and according to McGraw [4], applications that enable internet access are the biggest security risk of the modern day. Especially when considered in conjunction with the amount of users completely unaware of the types of risks that come with the internet, the problem has continued to grow over time. Software developers can take steps to better the quality of their code, which also tends to benefit the strength of their code's security.

To go briefly off topic, I would like to talk about application security. There are a few interpretations of how to make software secure, and one interpretation is through application security. Application security is the act of creating safeguards around code that has already been written. Two common ways to do this are to "sandbox" the code in a way that it is isolated from the causing huge effects, such as the way Java's VM handles code or to instead respond to issues with better defenses such that a program is strengthened over time.

In the case of the "sandbox" method, new code must be isolated similarly to the JVM such that if vulnerabilities are exploited, effects to the overall system should not include huge data loss or have effect on parts of the system that should not be accessed by the particular instead of the JVM. This method is often used in conjunction with code obfuscation to prevent direct access to the source code, measures to prevent malicious code from interacting with the system around the JVM, and creating executable white-lists to prevent unsafe applications from starting [4].

In the case of the response method, extensive data collection must happen at all times, and particularly when attacks or errors happen [4]. All programs or applications have to be closely monitored and proper procedures must always be followed such that when anything happens, it is easy to see exactly where and how a vulnerability caused issues. By collecting this data, rapid response updates to ensure that problems are not recurring are released as quickly and as efficiently as possible.

The obvious weakness of application security is that it either prohibits the effectiveness of the software by requiring strict protocol-based usage, or it requires that problems have effect at least once before better defenses can be created against them. It also tends to lead to constant back and forths between finding ways passed an application's defenses and finding ways to block the new vulnerabilities that arise.

More on topic, there are many steps software developers can take towards stronger, less vulnerable code. The first step that I would like to mention involves external tools to manually or automatically check the vulnerabilities apparent in their code [3]. There are two tools that I would like to highlight.

The first is a tool that is often used in conjunction with Git. DeepSource is a tool that will automatically suggest fixes for possible issues and respond with pulls in Git with recommendations. Like every tool that analyzes code, DeepSource does have false positives, but

it is so effortless to use that it is hard to recommend *not* using it, Other than the price, of course. Specifically, DeepSource has a massive list of common issues that is constantly being updated. It will scan and report on similar issues found in your code. Not only will it automatically improve code, it will also provide detailed listings of the reasons the code is flawed such that expert and novice programmers can learn from the problems. Many large companies also have used DeepSource, such as NASA and Uber [1].

The second is a relatively simple open source scanner for C/C++ programs that does similar things to DeepSource on a smaller scale. Flawfinder is an intentionally simple tool that uses a built in database to find and produce a list of potential inefficiencies or security flaws and mention them to the user [6]. While it prioritizes based on the riskiest hits, it does not entirely understand semantics and, once again, can have many false positives or miss some security vulnerabilities. The magic of Flawfinder is that it is free, simple, and easily accessible. There is no reason not to use Flawfinder on smaller bits of code as a simple "just in case" option, and I currently use it on every assignment that I can. Where DeepSource feels like a professional, paid, and bulky tool, Flawfinder is just a friend that helps out.

Besides tools for analyzing code, there is a very obvious way to write secure code. This is to just write good code. Obviously, this is an extremely broad and simplified sentence that means almost nothing until it is broken down. Instead, I would like to focus a particular practice in coding and one of its subsets. This practice is known as defensive programming and the subset is secure coding. While secure coding is covered in defensive programming, its importance is high enough that it deserves its own spotlight.

Defensive programming is coding in such a way that software can continue to operate in the intended manner despite unforeseen circumstances [2]. Some techniques of defensive programming include proper re-utilization of old software and canonicalization of libraries. If there is source code used in other software that is proven to be free of vulnerabilities, then it should be re-used where applicable. The problem that may arise from this is that re-using software means that any issues present in that software are magnified over time. While it is a large part of defensive programming to re-use existing software, it is up to the developer to understand where it is applicable. Canonicalization of libraries is a much smaller part of defensive programming, but it is still very important. It refers to creating one standard form of representation for data such that paths such as file names cannot be abused to work backwards and affect other parts of the system. There are many other aspects of defensive programming that should be utilized but these are the two that I felt would be most easily understood in a short amount of explanation.

Secure coding focuses on the aspect of defensive programming that guards against introducing vulnerable code into the source code to begin with. Over time, security professionals have concluded that the majority of vulnerabilities that appear in code come from the same small common software coding errors [5]. This discovery allows steps to be taken proactively against a significant number of vulnerabilities before they can even be coded. one example of secure coding involves using strncpy instead of strcpy or malloc to allocate memory to avoid the common buffer overflow problems. Another example is by checking for overflow errors before returning values from functions. As a reminder, secure coding is a subset of defensive

programming. Subsequently, the given examples of secure coding are also parts of defensive programming. I decided to separate the two in this paper because while defensive programming involves proper thinking and analysis of the situation, the majority of secure coding steps should always be taken.

To conclude, I included both application and software security examples in this paper because they should always be used in conjunction. Though application security may not be in the hands of the developer who codes the software, it is still important to know how your code may be handled by others. Also, the developer should code using good standards and tools to help prevent the amount of possible vulnerabilities that may appear. Software is crazy, weird, confusing, and just plain does not work sometimes, but with good quality code, sometimes we can be happy with what we write. So, yes, to paraphrase in a probably irresponsibly simplified way, just write good code.

Works Cited

[1] “Automate code reviews with static analysis.” *deepsource*, https://deepsource.io/. Accessed 17 April 2021.

[2] “Defensive programming.” Wikipedia, Wikimedia Foundation, 8 February 2021, https://en.wikipedia.org/wiki/Defensive_programming. Accessed 17 April 2021.

[3] Edmiston, Taylor D. “What is Software Security?” *The Framework*, 28 Aug. 2019, https://medium.com/the-framework-by-tangram-flex/what-is-software-security-e03a5ee7a6b5. Accessed 17 April 2021.

[4] McGraw, Gary “Software security.” *Software Integrity Blog*, 31 Mar. 2004, https://www.synopsys.com/blogs/software-security/software-security/. Accessed 17 April 2021.

[5] “Secure coding.” Wikipedia, Wikimedia Foundation, 2 April 2021, https://en.wikipedia.org/wiki/Secure_coding. Accessed 17 April 2021.

[6] Wheeler, David A. “Flawfinder.” *David A. Wheeler’s Blog*, https://dwheeler.com/flawfinder/. Accessed 17 April 2021.