Edward Johnson

Some of the most important lessons that I have learned in programming have been over the course of this semester as we began the process of learning to reverse engineer software applications. This past week our assignment had us go through the process of converting legacy code to assembly, analyzing each line of instruction, and then rebuilding it in C++. We were to then take that C++ code and refactor it to remove security vulnerabilities that it contained.

- Define: What is a security vulnerability?

Security vulnerabilities, as the name implies, are deficits within code that expose the system running the program to exploitation. There are many ways and forms that security vulnerabilities can be created, and each programming language / paradigm brings with it their own set of vulnerabilities. Many times these vulnerabilities are simply a manner in which a software system can be forced into an error, failure, or at times undefined behavior. These can be especially troublesome when behaviors within the software application that is failing have not properly accounted for the conditions causing the failure. When this happens, it can lead to situations in which a normally secure system ends up exposing protected memory. This memory access is what attackers are after, and this is because it is what will allow them to execute their own instructions on the compromised machine. The attacker would then have been given a way to bypass normal authentication based access to the machine(s) or system(s).

- Identify: What kinds of vulnerabilities would be identifiable in C++ code?

The most common avenues responsible for creating security vulnerabilities within C++ code include the following, improper management of memory, improper error handling, not accounting for concurrent access to shared memory (known as race conditions), failure to properly validate and

sanitize user inputs, and not following established best, secure practices for your the language. The most common vulnerabilities that are exploited by hackers are ones that cause a buffer overflow, or code injections such as with cross-site scripting / SQL injection attacks. These two methods are also some of the most common ways in which a hacker can remotely execute code on a machine.

In C++ specifically, our practices must be in line with industry standard as it is very easy to mismanage memory if you are manually handling pointers. Proper memory management is by-far one of the most important factors to securely programming in languages such as C++. There are a number of ways in which pointers and memory access can be used by an attacker. Memory leaks are a very common cause for instability in an application, and they occur in C++ when memory that was allocated to the program is not properly released when the program is done using it. This can lead to more and more of the system memory being consumed and locked by the application. These can be identified by ensuring that you accurately cover any pointer creation with a corresponding call to the object's destructor / use the delete keyword when the application is finished with the object. There are also vulnerabilities that involve hijacking the process of freeing memory, such as with a double-free attack. These are generally paired with a buffer overflow to allow the attacker access to system memory. Many of these issues are preventable, but require a programmer to use industry standard, best practices.

A common vulnerability related to not using best practices in C++ is failing to initializing your variables. This can cause an issue when the variable has not been properly given a value during operations, but the program tries to access its value anyway. In C++ if the value is truly not initialized, you may be returned whatever data is stored at that memory location. The data stored at that location could be something sensitive, not related to normal operations. Attempting to then write to this location could corrupt the memory in that location, or cause other unintended consequences.

Another major area of importance when it comes to design and coding practices for C++, is with error handling. One of the other most common ways in which a program can be used to compromise a system is by not properly implementing error handling. In a common scenario, this may be not properly implementing validation and error checks into a program's input. It is very important to use well designed implementations for input processing and sanitization, or else you can end up handing bad actors the ability to directly execute code through your input system. Very commonly this is seen in forms such as an SQL (or other database) injection attack. This allows a remote user to execute commands to your system database without the need to go through proper authentication channels. They could for example, send the command to drop your database, destroying all of the data within it.

When refactoring a C++ calculation program this past week, I noticed a number of ways in which an integer overflow or underflow could be created due to lack of error handling or input sanitization. In this specific scenario it was not necessarily serious, but if the integers being forced into an over/underflow are responsible for indexes, or are core to the loop control of an application this can at times cause serious unintended consequences. An example of these consequences would be in situation where an integer is used in determining memory allocation for an application. If the value being stored exceeds the space in memory allocated for the value then we end up in a situation where the buffer overflows, and memory that is not intended for use in the operation can be written to. An integer overflow alone is not as dangerous in the same ways as a buffer overflow, but without careful design considerations they can be used to create a buffer overflow.

- Purpose: Why would you be looking for vulnerabilities during legacy to C++ conversion rather than during testing?

Looking for these types of vulnerabilities during the process of converting and updating legacy code is much easier than doing so during the process of testing. This is especially true with large code bases, as it can become overly burdensome to trace potential sources of errors or failures within the software, especially when the source causing it does so in a way that isn't well defined. It is much easier for the developers in charge of implementing the code for the legacy conversion to follow best coding practices as they are redefining and redesigning the application's implementation. There are always points in which we will make mistakes, but part of the process of becoming better at writing software is to learn from and improve on these mistakes. It is important that as we work through the process of converting legacy code, that we also do our best to adhere to the very best of known secure coding standards for the language. Not only is the process of trying to find errors during the testing process more troublesome, but then you run into the added complication of needing to rework the implementation that was causing the error. Even when best practices are followed, reworking the functionality of an area of the application can cause unintended consequences elsewhere within the code.

- Solutions: How do you determine the appropriate fix to a security vulnerability?

The steps that must be taken to fix a security vulnerability begin with identifying the implementation that is at the center of the vulnerability. Ensuring that your system has proper logging setup can greatly help in the process of detecting vulnerabilities early on. Understanding and locating the root of the vulnerability is not always as easy as one might think. This is true because sometimes an area of the application can appear to be affected, but ends up being unrelated directly to responsible vulnerability.

A vulnerability with undefined behavior can at times effect seemingly random areas of the application in an undesired manner. Once the responsible vulnerability has been properly located though, it is important to determine all areas which might be affected by the vulnerability, and to determine what the best known practice is for fixing a vulnerability of the type discovered. This approach entirely depends on the type or form of vulnerability discovered, but it is necessary to properly account for edge-cases when designing your fix / patch. In circumstances where error handling needs to be updated, it is important to provide the application with a means by which to "fail with grace". We want our programs to have all possible scenarios accounted for. Defining the ways in which your application can fail helps to prevent more serious vulnerabilities from causing the program to fail in a way that can cause more damage to the system, the data contained within, or expose data that is private, and sensitive. When combined with secure coding practices, and a robust logging system, you are able to better create a strong deterrent to black hat (malicious) hackers.