Preventing Buffer Overflows

Addison Babcock

# Code Based Protection

## Programming Language Selection

Sometimes the easiest way to guard against buffer overflows is to use a programming language which is difficult to exploit. The basic C programming language is well known for being easy to exploit due to its unguarded and trusting nature. The dangers in C are quite numerous:

1. Arrays are just pointers and are not bounds checked.
2. Libraries were designed before security was taken seriously in information technology. The string manipulation functions in particular are full of non-obvious sharp edges.
3. Exploit prevention techniques are often not part of the standard and will vary from platform to platform and compiler to compiler. Code which is safe on one platform might be dangerous on another.
4. C developers tend to place buffers on the stack in order to avoid dynamic memory allocation. While this allows for faster execution, it results in the mixing of data and the addresses of code next to each other in memory.

Nonetheless, safe code can be written in C. It just takes more effort and persistence than it does in other languages. More importantly, it takes more experience and knowledge about how hacking works than other languages do.

C++ can be flawed in the same ways as C because it inherits the same memory model and the “the programmer is always right” attitude in its language design. Even all the C libraries are available in C++ to ensure backwards compatibility. The creators of C++ did include some very useful libraries to help avoid some of the pitfalls of C, but it’s the programmer’s responsibility to use those instead of the old libraries.

Newer managed languages (Java, the .Net languages, and more) have the advantage of being run inside of a protected environment that hides the computer’s memory from the programmer. By keeping pointers and the call stack away from the programmer, most buffer overflows are effectively prevented by design.

## Bounds Checking

Bounds checking can be an effective means of preventing buffer overflows but is far from a fool proof method. This form of protections primary downfall is that it sadly relies on programmers doing their jobs perfectly. If that was possible, this paper would be redundant. Another problem is that buffer overflows can sometimes happen in surprising ways, as I will demonstrate in the next section.

The principle behind bounds checking is fairly simple. Know the size of your buffers, be aware of what data comes from the user, and make sure the user is not able to provide data that exceeds the buffer length.

It is also important to ensure that C-style null terminated strings are in fact null-terminated. It’s easy to read in some character data that isn’t null-terminated and assume that it is. For example:

FILE\* file = fopen ("E:\\hello.txt", "r");

char hello\_string[6]; //Unintialized!

fread (hello\_string, sizeof (char), 5, file);

printf ("%s", hello\_string);

The content of hello.txt is simply “Hello”. The problem with this example is that the null terminator is missing from the end of hello\_string.

## Safe Libraries

### Format String Vulnerabilities

Bounds checking will usually only get you so far. Your lead developers can preach bounds checking until they are blue in the face and still not cover all possible buffer overflows.

Consider the C function printf. Printf takes an arbitrary number of arguments, the first being a format string and every argument after that is parsed using the format string. The printf function will then take those arguments, format them into a string in the manner specified by the programmer and output them to the console. However it is all too easy for a programmer to make the following mistake:

printf (somestring); //Danger zone!

printf ("%s", somestring); //Working fine

The first line in the above code can become dangerous if somestring is provided by the user. The user will be able to provide any formatting string they desire, including the %n format specifier. The %n format specifier causes the printf function to write the number of characters written to the console to a pointer that it expects in the argument list. For example:

int characters;

printf ("Hello%n", &characters);

if (characters == 5) {

printf (" world\n");

}

If a sufficiently clever user is allowed to construct a format string (like in the danger zone code), they can use the %n specifier to write arbitrary values to arbitrary memory locations by varying the length of the format string. The printf function will interpret the stack as containing one or more pointers to an int somewhere else in memory and will dutifully write the number of characters printed to that location. Just like that, the user is writing arbitrary data to an arbitrary memory location.

Thankfully the danger zone code is fairly easy to spot in code review. But the developer doing the code review needs to be aware of the issue in order to catch it, and this one is well hidden. I’ve personally ran into developers with decades of experience making this simple mistake.

It’s not just printf that can be exploited. sprintf is another function in the C library that be exploited in almost the same way. The syslog function in Linux exposes an interface that’s similar to printf and uses the same format string.

Avoiding the sprintf and printf functions is easy for C++ developers, use the stringstream and iostream libraries respectively. The stream libraries do not expose format strings and as such as much harder to exploit:

printf (“Some value: %d\n”, value);

std::cout << “Some value: “ << value << std::endl;

### String Manipulation

## Static Code Analysis

## Canaries

## /GS Compiler Option

# System Level Protection

## No Execute Bit

## ASLR