PREVENTING BUFFER OVERFLOWS

Preventing Buffer Overflows

Addison Babcock

Table of Contents

Introduction

Buffer overflow exploits can occur in numerous ways. While they are not always exploitable, buffer overflows usually do result in undefined behavior. Meaning that the code might crash instantly, it might be broken in some obvious way. A buffer overflow bug might not reveal itself until years later when it causes some subtle data corruption. Data corruption bugs like this tend to be very difficult to track down.

Worst of all, buffer overflows can sometimes allow a sufficiently clever hacker to exploit your systems resulting in significant damage. The SQL Slammer worm exploited a buffer overflow in Microsoft SQL Server to infect 75000 machines in 10 minutes. Although this particular worm did not cause any significant damage and was easily defeated, the potential for data theft or destruction was very high (Wikipedia). Consider how much damage could be caused if 75000 SQL Server instances were to suddenly fall into the wrong hands.

Thankfully many methods have been developed over the years to prevent buffer overflows from occurring and to prevent them being exploitable. Prevention methods fall into two main categories, code level and system level. Code level protection is built into the program and includes obvious things like checking buffer sizes, using a less exploitable language, using safe coding practice. Code level protection can also include some less obvious things like using automated analysis tools. By contrast, system level analysis includes features built in to the operating system and hardware that make buffer overflows more difficult to exploit.

Code Level Protection

There are several ways to prevent buffer overflows without relying on the operating system. These are usually preferred since the protection will be built into the program, instead of relying on the rest of the system for 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:

- Arrays are just pointers and are not bounds checked.
- Some libraries were designed before security was taken seriously in information technology. The C library functions in particular are full of non-obvious sharp edges.
- 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 compiler and operating system might be dangerous on another.

- 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. In recent years the C++ standards committee has been adding features that make it easy to write safer code, but existing legacy code might not benefit.

Newer managed languages (Java, the .Net languages, and more) have the advantage of being run inside of a protected environment that abstracts 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. Because exploits in C/C++ programs are so much more common, the majority of this paper will focus on those languages.

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 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];       //Uninitialized!
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`. The `printf` function will not know when to stop and could read far past the intended end of the string. Because this is only a problem due to the uninitialized value, the

bug will only show up intermittently. The bug might not show up until years later when someone makes a seemingly unrelated change and the compiler tweaks the layout of the resulting executable just enough. These are the worst kinds of bugs to find.

Safe Libraries

Rewriting existing code to use safe libraries can be a daunting task, but the results are worth it. The ways in which old libraries can be exploited are numerous and often not obvious. The newer libraries have been designed with buffer security in mind, and are much harder to exploit.

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 syntax.

Avoiding the `sprintf` and `printf` functions is generally easy for C++ developers; use the `<iostream>` and `<sstream>` libraries. The stream libraries do not expose format string functions and as such are much harder to exploit:

```
std::cout << "Some value: " << value << std::endl;
```

Examples of Format String Vulnerabilities

A blogger recently (Implementing a web server in a single printf() call)implemented a web server in a single call to the `printf` function. Although this example is not immediately exploitable, it does show how `printf` can be manipulated into executing arbitrary code. The in depth details are available on the website so I will spare you the full description here. The basic idea is that the author put the assembly code for his function into a long string, and then used the %n format specifier to place the code into memory and overwrite the stack.

String Manipulation

The string manipulation libraries built in to libc are very difficult to use correctly. Even experienced developers will make mistakes using them, and often those mistakes can appear to work correctly.

```
void getStuff (char* someBuffer)
{
	strcat (someBuffer, "lengthy stuff.");		//dangerous
}
```

The above code should be considered dangerous because it's unclear to the caller how big of a buffer needs to be passed in. In addition, there is no way for the `getStuff` function to validate the size of the buffer. This can make it difficult for developers to spot problems because it can be hard to track down all the places where `getStuff` is called, especially if it is a common name in the application. Because of reasons like this, developers should prefer to use safer string manipulation libraries.

Safe string manipulation libraries are common for C++. Two of the most popular options are the built-in `<string>` header and the Boost String Algorithms Library. The Boost library can be viewed as an extension of the `<string>` header, which fills in some gaps for missing functionality. Both libraries provide type safe, flexible and overrun safe classes and functions when used within the spec. Consider how the above code could be rewritten using the C++ `<string>` library.

```
void getStuff (std::string& someString)
{
	someString += "lengthy stuff.";			//safe
}
```

The `std::string` class contains all the information it needs to know when it needs to reallocate its buffer. The buffer reallocation will occur automatically when the length of the string would exceed the buffer size. Because of this, there is no potential for the buffer to overflow and no potential for the buffer to be exploited. It should be noted that there can be a minor performance hit if the string happens to be quite long and needs to reallocate. This performance hit won't be noticed in the vast majority of applications and developers should not worry about it unless the performance becomes a problem.

> ***We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. (Knuth)***

Secure Function Overloads

Visual C++ 2005 and newer ship with non-standard replacements for the libc string manipulation functions. The functions prevent exploits by also taking the size of the buffer being modified as an argument. For example, the `strcpy` function is replaced by `strcpy_s`. There are many other functions that the information in this section applies to but for the sake of simplicity I am only going to refer to `strcpy`.

The problem is that these functions are not portable to other platforms; they simply don't exist on Linux. So if a developer were to replace all the `strcpy` calls with `strcpy_s` calls in their code base they would find that they have broken cross platform compilation. This places developers of cross platform applications in an awkward position. The developer can choose to do one of the following:

- Refactoring working code to use more secure libraries (incurring the wrath of the testing team for adding to their workload).
- Replacing insecure function calls with a wrapper functions to call either `strcpy` or `strcpy_s` depending on the platform (tedious and error prone).
- Ignoring the problem and hoping the code never gets exploited (I shouldn't need to explain why this is a bad idea!).

Thankfully the Visual C++ team saw that this would be a problem and added a little bit of functionality to their compiler to help developers. Developers using Visual C++ 2005 and newer can have the compiler transform the `strcpy` (and similar) function calls to use `strcpy_s` instead just by adding the following line to their source code:

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
```

Code that uses the `strcpy` function will automatically be transformed to use the `strcpy_s` function wherever the compiler can deduce the size of the buffer. That code will continue to be potentially vulnerable on other platforms, so the code should still be fixed properly. The above macro should be

considered an easy temporary fix. See the cited MSDN article for more details on what code is and isn't protected. (Secure Template Overloads)

Static Code Analysis

Static code analysis tools are a lot like a compiler that only produces errors and warnings instead of code. This may not sound useful, but code analysis tools generally do a much more thorough job of analyzing code than a typical compiler will. A compiler is developed for compilation speed and will only produce common warnings that can be quickly detected.

These days it is becoming more and more common for static code analysis tools to be built into integrated development environments like Visual Studio. The code analysis tools are usually presented as separate options in the IDE since code analysis tools only need to be run occasionally. Visual C++ 2012 includes a quite good code analysis tool for C++ that can catch a number of potential buffer overflows, among other problems.

A list of problems detected by the Visual C++ 2012 compiler can be found on (MSDN). Of particular interest to the topic of buffer overflows are C6029, C6054, C6059, C6200, C6201, C6292, C6293, C6383 and C6386. I recommend at least reading a few of them in order to understand how easy it is to write subtly flawed code and to understand the need for automated tools.

Eclipse is another IDE which includes a C++ static analyzer. The static code analysis tools in Eclipse work in a similar fashion to Visual Studio. Eclipse also supports adding third party analysis tools through plugins. The (cppcheclipse) project does just that by integrating cppcheck with an Eclipse project.

The downside to running tools outside of the compilation process is that they can take a significant amount of time to run for large projects, often several hours. This can be an issue for teams that want to run code analysis on every build and still want to release frequently. Projects of this size should consider setting their compiler warnings to their highest level. For most compilers, warnings can be turned to their highest reasonable level with the command line switch `/wall`. This will catch a decent portion of the problems while leaving the build time reasonable.

Another potential downside to code analysis tools is that they can sometimes produce false positives. A false positive is when the tool reports a potential problem but a more thorough manual analysis proves that the code is not flawed. These are rare, although egos can sometimes cause developers to ignore valid warnings they don't understand. It is usually a good policy to insist on a clean build with no warnings, even if your tools produce a lot of false positives.

Canaries

A canary is a portion of memory owned by the application that is being used to store pre-determined data for the purposes of detecting a buffer overflow exploit. The name canary references the canaries that were placed in coal mines to provide early warning of poisonous gases for the miners.

For example: if the code places a number of buffers on the stack the compiler might pad the buffers with 16 bytes (or some quantity) of a value that is unlikely to be used by the application. Common values for a canary include 0xDEADBEEF, 0x0BADF00D and 0x55555555. The application's run time environment will then monitor the canaries for changes by the application. Because the canary is not memory that should be manipulated by user code a change in the canary will trigger the runtime to halt the program.

The /GS compiler switch controls the implementation of canaries in the Visual Studio compiler (/GS (Buffer Security Check)). Turning the switch on will cause the Visual C++ compiler to use a canary (called a security token in the documentation) to detect when a buffer has exceeded its allocated size.

This can sometimes help programmers discover bad code before it becomes a problem. Not every buffer overflow is caused by a hacker, sometimes they are caused by sloppy coding. Because of this the /GS option should be turned on whenever possible.

However the /GS option is not without limitations. Objects which are placed on the stack are not protected, which can allow a malicious user to manipulate data. This sounds pretty scary and it is; but consider what happens when a C++ virtual object is placed on the stack. Every virtual object in C++ keeps a vtable next to its data members. The vtable is responsible for holding the location of the virtual methods that can be used with an object. Because the vtable is not protected a malicious user can manipulate the vtable to point to arbitrary code. At that point it's back to standard exploit techniques to get the hackers code to execute. The /GS compiler switch does not guarantee immunity against insecure code.

System Level Protection

System level protection is not the preferred way to fight buffer overflow exploits. Because the protection comes from the operating system or hardware and not from the program, system level protection cannot be relied on. A user can turn the protection off, or the system might not support a protection mechanism. In some cases system level protection cannot be used because of legacy code that does not support or expect the protection mechanism. They are very useful as an added layer of security however.

No Execute Bit

The no execute bit (often referred to as the NX bit) is a feature present on most desktop CPUs built in the last decade or so and supported by most platforms. It allows the operating system to mark an area of memory as not being non-executable. This effectively segregates data and code in memory, which goes a long way to preventing exploits.

Think of the NX bit like removing the execute privilege from the users documents folder. The user can still trash the documents folder, but they won't be able to bring in arbitrary executable files. Similarly, a

hacker will be able to trash the stack and overwrite memory but the hacker will not be able to execute arbitrary code.

The NX bit is enabled by default on every version of Windows since Windows XP SP2 (NX Bit). Because the NX bit is checked by dedicated hardware there is virtually no performance impact associated with having the protection enabled.

Address Space Layout Randomization

Address Space Layout Randomization, or ASLR for short, is a technique that is used by modern operating systems to make the address of critical areas of memory random. This will not absolutely prevent an exploit from occurring on its own, but the difficulty is increased to the point that an exploit might no longer be feasible.

For example, an attacker who wants to execute code in libc would first have to find where libc is located in memory. The location of the libc code will no longer be the same every time the program is executed. Buffers allocated by the program will also change address with every execution.

However ASLR should not be relied on as an absolute protection against exploits. Hackers can potentially use a `printf` exploit (described earlier) to leak the address of the `printf` function. Once a hacker has this address they can use the address to offset into the libc library and ASLR has effectively been defeated.

Although Windows Vista and higher have support for ASLR, the programs that are executed by Windows quite often do not. A program must be compiled with ASLR enabled (via the /DYNAMICBASE linker option) and all the DLLs that the program loads must have the switch enabled. If only one DLL is loaded without ASLR enabled the ASLR mechanism will not be enabled for that entire program. Because /DYNAMICBASE was not enabled by default until Visual Studio 2008 (Fan), a program loading older DLLs will not be protected by ASLR.

That ASLR is frequently disabled can sometimes come as a surprise to developers. Even popular internet facing applications can be shipped with ASLR disabled. In 2013, blogger Graham Sutherland discovered that ASLR is disabled on a Firefox plugin shipped by Dropbox. Because the plugin DLL did not have ASLR enabled, in effect Firefox was slightly easier to exploit. Thankfully the issue has since been resolved. (Sutherland)

Summary

The number of ways that unguarded buffers can be exploited is surprising. The easiest way to avoid an exploit is to simply write secure code. Check your buffer lengths, use sane modern libraries, run static code analysis regularly, use your tools and platform to help protect you. An understanding of where your security ends is required to fully understand whether or not your code is protected.

Works Cited

/GS (Buffer Security Check). 2014. 10 April 2014. <http://msdn.microsoft.com/en-us/library/8dbf701c.aspx>.

cppcheclipse. 29 February 2014. 18 March 2014. <https://code.google.com/a/eclipselabs.org/p/cppcheclipse/?redir=1>.

Fan, Xiang. */DYNAMICBASE and /NXCOMPAT*. 21 May 2009. 1 April 2014. <http://blogs.msdn.com/b/vcblog/archive/2009/05/21/dynamicbase-and-nxcompat.aspx>.

Implementing a web server in a single printf() call. 12 March 2014. 13 March 2014. <http://tinyhack.com/2014/03/12/implementing-a-web-server-in-a-single-printf-call/>.

Knuth, Donald. *Program optimization*. 2 February 2014. 1 April 2014. <http://en.wikipedia.org/w/index.php?title=Program_optimization&oldid=593530416>.

MSDN. *Code Analysis for C/C++ Warnings*. 18 March 2014. 18 March 2014. <http://msdn.microsoft.com/en-us/library/vstudio/a5b9aa09(v=vs.110).aspx>.

NX Bit. 24 March 2014. 1 April 2014. <http://en.wikipedia.org/w/index.php?title=NX_bit&oldid=601000671#Microsoft_Windows>.

Secure Template Overloads. Vers. Visual Studio 2013. 2013. 1 April 2014. <http://msdn.microsoft.com/en-us/library/ms175759.aspx>.

Sutherland, Graham. *Installing Dropbox? Prepare to lose ASLR*. 9 September 2013. 1 April 2014. <http://codeinsecurity.wordpress.com/2013/09/09/installing-dropbox-prepare-to-lose-aslr/>.

Vieque. *Dynamic Program Analysis*. 2 February 2014. 27 February 2014. <http://en.wikipedia.org/wiki/Dynamic_code_analysis>.

Wikipedia. *SQL Slammer*. 21 January 2014. 18 March 2014. <http://en.wikipedia.org/w/index.php?title=SQL_Slammer&oldid=591662922>.