GUARDENT℠
secure digital infrastructure

Format String Attacks

Tim Newsham
Guardent, Inc.
September 2000

GUARDENT℠
secure digital infrastructure

CONTENTS

ABSTRACT

The cause and implications of format string vulnerabilities are discussed. Practical examples are given to illustrate the principles presented.

GUARDENT℠
secure digital infrastructure

INTRODUCTION

I know it has happened to you. It has happened to all of us, at one point or another. You're at a trendy dinner party, and amidst the frenzied voices of your companions you hear the words "format string attack." "Format string attack? What is a format string attack?" you ask. Afraid of having your ignorance exposed among your peers you decide instead to break an uncomfortable smile and nod in hopes of appearing to be in the-know. If all goes well, a few cocktails will pass and the conversation will move on, and no one will be the wiser. Well fear no more! This paper will cover everything you wanted to know about format string attacks but were afraid to ask!

WHAT IS A FORMAT STRING ATTACK?

Format string bugs come from the same dark corner as many other security holes: The laziness of programmers. Somewhere out there right now, as this document is being read, there is a programmer writing code. His task: to print out a string or copy it to some buffer. What he means to write is something like:

```
printf("%s", str);
```

but instead he decides that he can save time, effort and 6 bytes of source code by typing:

```
printf(str);
```

Why not? Why bother with the extra **`printf`** argument and the time it takes to parse through that silly format? The first argument to **`printf`** is a string to be printed anyway! Because the programmer has just unknowingly opened a security hole that allows an attacker to control the execution of the program, that's why!

What did the programmer do that was so wrong? He passed in a string that he wanted printed verbatim. Instead, the string is interpreted by the **`printf`** function as a format string. It is scanned for special format characters such as **`"%d"`**. As formats are encountered, a variable number of argument values are retrieved from the stack. At the least, it should be obvious that an attacker can peek into the memory of the program by printing out these values stored on the stack. What may not be as obvious is that this simple mistake gives away enough control to allow an arbitrary value to be written into the memory of the running program.

PRINTF - WHAT THEY FORGOT TO TELL YOU IN SCHOOL

Before getting into the details of how to abuse **`printf`** for our own purposes, we should have a firm grasp of the features **`printf`** provides. It is assumed that the reader has used **`printf`** functions before and knows about its normal formatting features, such as how to print integers and strings, and how to specify minimum and maximum string widths. In addition to these more mundane features, there are a few esoteric and little-known features. Of these features, the following are of particular relevance to us:

- It is possible to get a count of the number of characters output at any point in the format string. When the **"%n"** format is encountered in the format string, the number of characters output before the **%n** field was encountered is stored at the address passed in the next argument. As an example, to receive the offset to the space between two formatted numbers:

```
int pos, x = 235, y = 93;

printf("%d %n%d\n", x, &pos, y);
printf("The offset was %d\n", pos);
```

- The **"%n"** format returns the number of characters that should have been output, not the actual count of characters that were output. When formatting a string into a fixed-size buffer, the output string may be truncated. Despite this truncation, the offset returned by the **"%n"** format will reflect what the offset would have been if the string was not truncated. To illustrate this point, the following code will output the value **"100"** and not **"20"**:

```
char buf[20];
int pos, x = 0;

snprintf(buf, sizeof buf, "%.100d%n", x, &pos);
printf("position: %d\n", pos);
```

A SIMPLE EXAMPLE

Rather than talking in vagaries and abstractions, we will use a concrete example to illustrate the principles as they are discussed. The following simple program will suffice for this purpose:

```
/*
 * fmtme.c
 *       Format a value into a fixed-size buffer
 */

#include <stdio.h>

int
main(int argc, char **argv)
{
    char buf[100];
    int x;

    if(argc != 2)
        exit(1);

    x = 1;

    snprintf(buf, sizeof buf, argv[1]);
    buf[sizeof buf - 1] = 0;
    printf("buffer (%d): %s\n", strlen(buf), buf);
    printf("x is %d/%#x (@ %p)\n", x, x, &x);
    return 0;
}
```

A few notes about this program are in order. First, the general purpose is quite simple: A value passed on the command line is formatted into a fixed-length buffer. Care is taken to make sure the buffer limits are not exceeded. After the buffer is formatted, it is output. In addition to formatting the argument, a second integer value is set and later output. This variable will be used as the target of attacks later. For now, it should be noted that its value should always be one.

All examples in this document were actually performed on an x86 BSD/OS 4.1 box. If you have been on a mission to Mozambique for the last 20 years and are unfamiliar with the x86, it is a little-endian machine. This will be reflected in the examples when multi-precision numbers are expressed as a series of byte values. The actual numbers used here will vary from system to system with differences in architecture, operating system, environment and even command line length. The examples should be easily adjusted to work on other x86 machines. With some effort and thought, they may be made to work on other architectures as well.

FORMAT ME!

It is now time to put on our black hats and start thinking like attackers. We have in our hands a test program. We know that it has a vulnerability and we know where the programmer made his mistake. We are also armed with a thorough knowledge of the **`printf`** function and what it can do for us. Let's get to work by tinkering with our program.

Starting off simple, we invoke the program with normal arguments. Let's begin with this:

```
% ./fmtme "hello world"
buffer (11): hello world
x is 1/0x1 (@ 0x804745c)
```

There's nothing special going on here. The program formatted our string into the buffer and then printed its length and the value out. It also told us that the variable **`x`** has the value one (shown in decimal and hex) and that it was stored at the address **`0x804745c`**.

Next lets try providing some format directives. In this example we'll print out the integers on the stack above the format string:

```
% ./fmtme "%x %x %x %x"
buffer (15): 1 f31 1031 3133
x is 1/0x1 (@ 0x804745c)
```

A quick analysis of the program will reveal that the stack layout of the program when the **snprintf** function is called is:

Address	Contents	Description
fp+8	Buffer pointer	4-byte address
fp+12	Buffer length	4-byte integer
fp+16	Format string	4-byte address
fp+20	Variable **x**	4-byte integer
fp+24	Variable **buf**	100 characters

The four values output in the previous test were the next four arguments on the stack after the format string: the variable **x,** then three 4-byte integers taken from the uninitialized **buf** variable.

Now it is time for an epiphany. As an attacker, we control the values stored in the buffer. These values are also used as arguments to the **snprintf** call! Let's verify this with a quick test:

```
% ./fmtme "aaaa %x %x"
buffer (15): aaaa 1 61616161
x is 1/0x1 (@ 0x804745c)
```

Yup! The four **'a'** characters we provided were copied to the start of the buffer and then interpreted by **snprintf** as an integer argument with the value **0x61616161** (**'a'** is **0x61** in ASCII).

X MARKS THE SPOT

All the pieces are falling into place! It is time to step up our attack from passive probes to actively altering the state of the program. Remember that variable **"x"**? Let's try to change its value. To do this, we will have to enter its address into one of **snprintf's** arguments. We will then have to skip over the first argument to **snprintf**, which is the variable **x**, and finally, use a **"%n"** format to write to the address we specified. This sounds more complicated than it actually is. An example should clarify things. [Note: We're using PERL here to execute the program which allows us to easily place arbitrary characters in the command line arguments]:

```
% perl -e ‘system "./fmtme", "\x58\x74\x04\x08%d%n"’
buffer (5): X1
x is 5/x05 (@ 0x8047458)
```

The value of **x** changed, but exactly what is going on here? The arguments to **snprintf** look something like this:

```
snprintf(buf, sizeof buf, "\x58\x74\x04\x08%d%n", x, 4 bytes from buf)
```

At first **snprintf** copies the first four bytes into **buf**. Next it scans the **"%d"** format and prints out the value of **x.** Finally it reaches the **"%n"** directive. This pulls the next value off the stack, which comes from the first four bytes of **buf**. These four bytes have just been filled with **"\x58\x74\x04\x08"**, or, interpreted as an integer, **0x08047458**. **Snprintf** then writes the amount of bytes output so far, five, into this address. As it turns out, that address is the address of the variable **x**. This is no coincidence. We carefully chose the value **0x08047458** by previous examination of the program. In this case, the program was helpful in printing out the address we were interested in. More typically, this value would have to be discovered with the aid of a debugger.

Well, great! We can pick an arbitrary address (well, almost arbitrary; as long as the address contains no NUL characters) and write a value into it. But can we write a useful value into it? **Snprintf** will only write out the number of characters output so far. If we want to write out a small value greater than four then the solution is quite simple: Pad out the format string until we get the right value. But what about larger values? Here is where we take advantage of the fact that **"%n"** will count the number of characters that should have been output if there was no truncation:

```
% perl -e 'system "./fmtme", "\x54\x74\x04\x08%.500d%n"
buffer (99): %0000000 ... 0000
x is 504/x1f8 (@ 0x8047454)
```

The value that **"%n"** wrote to **x** was **504**, much larger than the 99 characters actually emitted to **buf**. We can provide arbitrarily large values by just specifying a large field width[1]. And what about small values? We can construct arbitrary values (even the value zero), by piecing together several writes. If we write out four numbers at one-byte offsets, we can construct an arbitrary integer out of the four least-significant bytes. To illustrate this, consider the following four writes:

Address	**A**	**A+1**	**A+2**	**A+3**	**A+4**	**A+5**	**A+6**
Write to A:	0x11	0x11	0x11	0x11			
Write to A+1:		0x22	0x22	0x22	0x22		
Write to A+2:			0x33	0x33	0x33	0x33	
Write to A+3:				0x44	0x44	0x44	0x44
Memory:	0x11	0x22	0x33	0x44	0x44	0x44	0x44

After the four writes are completed, the integer value **0x44332211** is left in memory at address A, composed of the least-significant byte of the four writes. This technique gives us flexibility in choosing values to write, but it does have some drawbacks: It takes four times as many writes to set the value. It overwrites three bytes neighboring the target address. It also performs three unaligned write operations. Since some architectures do not support unaligned writes, this technique is not universally applicable.

[1] There is an implementation flaw in printf in certain versions of **glibc**. When large field widths are specified, **printf** will underflow an internal buffer and cause the program to crash. Because of this, it is not possible to use field widths larger than several thousand when attacking programs on certain versions of Linux. As an example, the following code will cause a segmentation fault on systems with this flaw: **printf("%.9999d", 1);**

SO WHAT?

So what? So what!? *SO WHAT*!#@?? So you can write arbitrary values to (almost any) arbitrary addresses in memory!!! Surely you can think of a good use for this. Let's see:

- Overwrite a stored UID for a program that drops and elevates privleges.
- Overwrite an executed command.
- Overwrite a return address to point to some buffer with shell code in it.

Put into simpler terms: you *OWN* the program.

Ok, so what have we learned today?

- **`printf`** is more powerful than you previously thought.
- Cutting corners never pays off.
- An innocent looking omission can provide an attacker with just enough leverage to ruin your day.
- With enough free time, effort, and an input string that looks like the winning entry in last year's obfuscated-C contest, you can turn someone's simple mistake into a nationally syndicated news story.