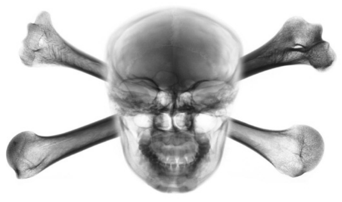

SIN 2

FORMAT STRING PROBLEMS

OVERVIEW OF THE SIN

Format string problems are one of the few truly new attacks to surface in recent years. One of the first mentions of format string bugs was on June 23, 2000, in a post by Lamagra Argamal (www.securityfocus.com/archive/1/66842); Pascal Bouchareine more clearly explained them almost a month later (www.securityfocus.com/archive/1/70552). An earlier post by Mark Slemko (www.securityfocus.com/archive/1/10383) noted the basics of the problem, but missed the ability of format string bugs to write memory.

As with many security problems, the root cause of format string bugs is trusting user-supplied input without validation. In C/C++, format string bugs can be used to write to arbitrary memory locations, and the most dangerous aspect is that this can happen without tampering with adjoining memory blocks. This fine-grained capability allows an attacker to bypass stack protections, and even modify very small portions of memory. The problem can also occur when the format strings are read from an untrusted location the attacker controls. This latter aspect of the problem tends to be more prevalent on UNIX and Linux systems. On Windows systems, application string tables are generally kept within the program executable, or resource Dynamic Link Libraries (DLLs). If an attacker can rewrite the main executable or the resource DLLs, the attacker can perform many more straightforward attacks than format string bugs.

Even if you're not dealing with C/C++, format string attacks can still lead to considerable problems. The most obvious is that users can be misled, but under some conditions, an attacker might also launch cross-site scripting or SQL injection attacks. These can be used to corrupt or transform data as well.

AFFECTED LANGUAGES

The most strongly affected language is C/C++. A successful attack can lead immediately to the execution of arbitrary code, and to information disclosure. Other languages won't typically allow the execution of arbitrary code, but other types of attacks are possible as we previously note. Perl isn't directly vulnerable to specifiers being given by user input, but it could be vulnerable if the format strings are read in from tampered data.

THE SIN EXPLAINED

Formatting data for display or storage can be a somewhat difficult task. Thus, many computer languages include routines to easily reformat data. In most languages, the formatting information is described using some sort of a string, called the *format string*. The format string is actually defined using limited data processing language that's designed to make it easy to describe output formats. But many developers make an easy mistake—they use data from untrusted users as the format string. As a result, attackers can write strings in the data processing language to cause many problems.

The design of C/C++ makes this especially dangerous: C/C++'s design makes it harder to detect format string problems, and format strings include some especially dangerous commands (particularly %n) that do not exist in some other languages' format string languages.

In C/C++, a function can be declared to take a variable number of arguments by specifying an ellipsis (…) as the last (or only) argument. The problem is that the function being called has no way to know just how many arguments are being passed in. The most common set of functions to take variable length arguments is the printf family: printf, sprintf, snprintf, fprintf, vprintf, and so on. Wide character functions that perform the same function have the same problem. Let's take a look at an illustration:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  if(argc > 1)
    printf(argv[1]);

  return 0;
}
```

Fairly simple stuff. Now let's look at what can go wrong. The programmer is expecting the user to enter something benign, such as **Hello World**. If you give it a try, you'll get back `Hello World`. Now let's change the input a little—try `%x %x`. On a Windows XP system using the default command line (cmd.exe), you'll now get the following:

```
E:\projects\19_sins\format_bug>format_bug.exe "%x %x"
12ffc0 4011e5
```

Note that if you're running a different operating system, or are using a different command line interpreter, you may need to make some changes to get this exact string fed into your program, and the results will likely be different. For ease of use, you could put the arguments into a shell script or batch file.

What happened? The printf function took an input string that caused it to expect two arguments to be pushed onto the stack prior to calling the function. The %x specifiers enabled you to read the stack, four bytes at a time, as far as you'd like. It isn't hard to imagine that if you had a more complex function that stored a secret in a stack variable, the attacker would then be able to read the secret. The output here is the address of the stack location (0x12ffc0), followed by the code location that the main() function will return into. As you can imagine, both of these are extremely important pieces of information that are being leaked to an attacker.

You may now be wondering just how the attacker uses a format string bug to write memory. One of the least used format specifiers is %n, which writes the number of characters that should have been written so far into the address of the variable you gave as the corresponding argument. Here's how it should be used:

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Your input was %d characters long\n, bytes");
```

The output would be:

```
E:\projects\19_sins\format_bug>format_bug2.exe "Some random input"
Some random input
Your input was 17 characters long
```

On a platform with four-byte integers, the %n specifier will write four bytes at once, and %hn will write two bytes. Now attackers only have to figure out how to get the address they'd like in the appropriate position in the stack, and tweak the field width specifiers until the number of bytes written is what they'd like.

NOTE You can find a more complete demonstration of the steps needed to conduct an exploit in Chapter 5 of *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), or in *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* by Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan "noir" Eren, Neel Mehta, and Riley Hassell (Wiley, 2004).

For now, let's just assume that if you allow attackers to control the format string in a C/C++ program, it is a matter of time before they figure out how to make you run their code. An especially nasty aspect of this type of attack is that before launching the attack, they can probe the stack and correct the attack on the fly. In fact, the first time the author demonstrated this attack in public, he used a different command line interpreter than he'd used to create the demonstration, and it didn't work. Due to the unique flexibility of this attack, it was possible to correct the problem and exploit the sample application with the audience watching.

Most other languages don't support the equivalent of a %n format specifier, and they aren't directly vulnerable to easy execution of attacker-supplied code, but you can still run into problems. There are other, more complex variants on this attack that other languages are vulnerable to. If attackers can specify a format string for output to a log file or database, they can cause incorrect or misleading logs. Additionally, the application reading the logs may consider them trusted input, and once this assumption is violated, weaknesses in that application's parser may lead to execution of arbitrary code. A related problem is embedding control characters in log files—backspaces can be used to erase things; line terminators can obfuscate or even eliminate the attacker's traces.

This should go without saying, but if an attacker can specify the format string fed to scanf or similar functions, disaster is on the way.

Sinful C/C++

Unlike many other flaws we'll examine, this one is fairly easy to spot as a code defect. It's very simple:

```
printf(user_input);
```

is wrong, and

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

is correct.

One variant on the problem that many programmers neglect is that it is not sufficient to do this correctly only once. There are a number of common code constructs where you might use sprintf to place a formatted string into a buffer, and then slip up and do this:

```
fprintf(STDOUT, err_msg);
```

The attacker then only has to craft the input so that the format specifiers are escaped, and in most cases, this is a much more easily exploited version because the err_msg buffer frequently will be allocated on the stack. Once attackers manage to walk back up the stack, they'll be able to control the location that is written using user input.

Related Sins

Although the most obvious attack is related to a code defect, it is a common practice to put application strings in external files for internationalization purposes. If your application has sinned by failing to protect the file properly, then an attacker can supply format strings because of a lack of proper file access.

Another related sin is failing to properly validate user input. On some systems, an environment variable specifies the locale information, and the locale, in turn, determines the directory where language-specific files will be found. On some systems, the attacker might even cause the application to look in arbitrary directories.

SPOTTING THE SIN PATTERN

Any application that takes user input and passes it to a formatting function is potentially at risk. One very common instance of this sin happens in conjunction with applications that log user input. Additionally, some functions may implement formatting internally.

SPOTTING THE SIN DURING CODE REVIEW

In C/C++, look for functions from the printf family. Problems to look for are

```
printf(user_input);
fprintf(STDOUT, user_input);
```

If you see a function that looks like this:

```
fprintf(STDOUT, msg_format, arg1, arg2);
```

then you need to verify where the string referenced by msg_format is stored and how well it is protected.

There are many other system calls and APIs that are also vulnerable—syslog is one example. Any time you see a function definition that includes … in the argument list, you're looking at something that is likely to be a problem.

Many source code scanners, even the lexical ones like RATS and flawfinder, can detect this. There's even PScan (www.striker.ottawa.on.ca/~aland/pscan/), which was designed specifically for this.

There are also countering tools that can be built into the compilation process. For example, there's Crispin Cowan's FormatGuard: http://lists.nas.nasa.gov/archives/ext/linux-security-audit/2001/05/msg00030.html.

TESTING TECHNIQUES TO FIND THE SIN

Pass formatting specifiers into the application and see if hexadecimal values are returned. For example, if you have an application that expects a file name and returns an error message containing the input when the file cannot be found, then try giving it file names like `NotLikely%x%x.txt`. If you get an error message along the lines of "NotLikely12fd234104587.txt cannot be found," then you have just found a format string vulnerability.

This is obviously somewhat language-dependent; you should pass in the formatting specifiers that are used by the implementation language you're using at least. However, since many language run times are implemented in C/C++, you'd be wise to *also* send in C/C++ formatting string commands to detect cases where your underlying library has a dangerous vulnerability.

Note that if the application is web based and echoes your user input back to you, another concern would be cross-site scripting attacks.

EXAMPLE SINS

The following entries in Common Vulnerabilities and Exposures (CVE) at http://cve.mitre.org are examples of SQL injection. Out of the 188 CVE entries that reference format strings, this is just a sampling.

CVE-2000-0573

From the CVE description: "The lreply function in wu-ftpd 2.6.0 and earlier does not properly cleanse an untrusted format string, which allows remote attackers to execute arbitrary commands via the SITE EXEC command."

This is the first publicly known exploit for a format string bug. The title of the BUGTRAQ post underscores the severity of the problem: "Providing *remote* root since at least 1994."

CVE-2000-0844

From the CVE description: "Some functions that implement the locale subsystem on UNIX do not properly cleanse user-injected format strings, which allows local attackers to execute arbitrary commands via functions such as gettext and catopen."

The full text of the original advisory can be found at www.securityfocus.com/archive/1/80154, and this problem is especially interesting because it affects core system APIs for most UNIX variants (including Linux), except for BSD variants due to the fact that the NLSPATH variable is ignored for privileged suid application in BSD. This advisory, like many CORE SDI advisories, is especially well written and informative and gives a very thorough explanation of the overall problem.

REDEMPTION STEPS

The first step is never pass user input directly to a formatting function, and also be sure to do this at every level of handling formatted output. As an additional note, the formatting functions have significant overhead. Look at the source for _output if you're interested—it might be convenient to write:

```
fprintf(STDOUT, buf);
```

The preceding line of code isn't just dangerous, but it also consumes a lot of extra CPU cycles.

The second step to take is to ensure that the format strings your application uses are only read from trusted places, and that the paths to the strings cannot be controlled by the attacker. If you're writing code for UNIX and Linux, following the example of the BSD variants and ignoring the NLSPATH variable, which can be used to specify the file used for localized messages, may provide some defense in depth.

C/C++ Redemption

There isn't much more to it than this:

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

EXTRA DEFENSIVE MEASURES

Check and limit the locale to valid values. (For more information, see David Wheeler's "Write It Secure: Format Strings and Locale Filtering" listed in the "Other Resources" section below). Don't use the printf-family of functions if you can avoid it. For example, if you're using C++, use stream operators instead:

```
#include <iostream>
//...
std::cout << user_input
//...
```

OTHER RESOURCES

- "format bugs, in addition to the wuftpd bug" by Lamagra Agramal: www.securityfocus.com/archive/1/66842
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, "Public Enemy #1: Buffer Overruns"
- "UNIX locale format string vulnerability, CORE SDI" by Iván Arce: www.securityfocus.com/archive/1/80154
- "Format String Attacks" by Tim Newsham: www.securityfocus.com/archive/1/81565
- "Windows 2000 Format String Vulnerabilities" by David Litchfield: www.nextgenss.com/papers/win32format.doc
- "Write It Secure: Format Strings and Locale Filtering" by David A. Wheeler: www.dwheeler.com/essays/write_it_secure_1.html

SUMMARY

- **Do** use fixed format strings, or format strings from a trusted source.
- **Do** check and limit locale requests to valid values.
- **Do not** pass user input directly as the format string to formatting functions.
- **Consider** using higher-level languages that tend to be less vulnerable to this issue.