

Variadic Functions

How they contribute to security vulnerabilities and how to fix them



BY ROBERT C. SEACORD

C/C++ language variadic functions are functions that accept a variable number of arguments. Variadic functions are implemented using either the ANSI C `stdarg` approach or, historically, the UNIX System V `vararg` approach. Both approaches require that the contract between the developer and user of the variadic function not be violated by the user.

Many of the formatted I/O functions in the ISO/IEC 9899:1999 C language standard (C99) such as `printf()` and `scanf()` are defined as variadic functions (including formatted output functions that operate on a multibyte characters [e.g., ASCII] and wide characters [e.g., UNICODE]).

These functions accept a fixed format string argument that specifies, among other things, the number and type of arguments that are expected. If the contents of the format string are incorrect (by error or by malicious intent), the resulting behavior of the function is undefined.

Incautious use of formatted I/O functions have led to numerous, exploitable vulnerabilities. The majority of these vulnerabilities occur when a potentially malicious user is able to control all or some portion of the format specification string as shown in the following program:

```
1. #include <stdio.h>
2. #include <string.h>
3. void usage(char *pname) {
4.     char usageStr[1024];
5.     sprintf(usageStr, 1024,
6.             "Usage: %s <target>\n", pname);
7.     printf(usageStr);
8. }
9. int main(int argc, char * argv[]) {
10.    if (argc < 2) {
```



```
10.    usage(argv[0]);
11.    exit(-1);
12. }
13. }
```

These vulnerabilities are often referred to as “format string” vulnerabilities. Exploits take a variety of forms, the most dangerous of which involves using the `%n` conversion specifier to overwrite memory and transfer control to arbitrary code of the attacker’s choosing. The easiest way to prevent format string vulnerabilities is to ensure that the format string does not include characters from untrusted sources. Because of internationalization, however, format strings and message text are often moved into external catalogs or files that the program opens at runtime. An attacker can alter the values of the formats and strings in the program by modifying the contents of these files. The entire topic of formatted output is covered in detail in my book on Secure Coding in C/C++.

Format string vulnerabilities have been discovered in a variety of deployed C language programs, including:

- The Washington University FTP daemon `wu-ftpd` that is shipped with many distributions of Linux and other UNIX operating systems (CA-2000-13).
- The common desktop environment (CDE), an integrated graphical user

interface that runs on UNIX and Linux operating systems (CA-2001-27).

- Helix Player, and media players based on the Helix Player, including Real Player for Linux systems (VU#361181).

The following is an example of a variadic function implementation using ANSI `stdarg`:

```
1. int average(int first, ...) {
2.     int count = 0, sum = 0, i = first;
3.     va_list marker;
4.     va_start(marker, first);
5.     while (i != -1) {
6.         sum += i;
7.         count++;
8.         i = va_arg(marker, int);
9.     }
10.    va_end(marker);
11.    return(sum ? (sum / count) : 0);
12. }
```

Variadic functions are declared using a partial parameter list followed by the ellipsis notation. The variadic `average()` function accepts a single, fixed integer argument followed by a variable argument list. Like other functions, the arguments to the variadic function are pushed on the calling stack.

Variadic functions are problematic for a number of reasons. The first and foremost is that the implementation has no real way of knowing how many arguments were passed (even though this information is available at compile time). The termination condition for the argument list is a contract between the programmers who implement the library function and the programmers who use the function in an application. In this implementation of the `average()` function, termination of the variable argument list is indicated by an argument whose value is `-1`. This means, for example, that `average(5, -1, 2, -1)` is 5, not 2, as the programmer might expect. Also, if the programmer calling the function neglects to provide this argument, the `average()` function will continue to process the next argument indefinitely until a `-1` value is encountered or an exception occurs.

A second problem with variadic functions is a complete lack of type checking. In the case of formatted output functions, the

ABOUT THE AUTHOR

Robert C. Seacord is a senior vulnerability analyst at the CERT/Coordination Center (CERT/CC) at the Software Engineering Institute (SEI) in Pittsburgh, PA, and author of *Secure Coding in C and C++* (Addison-Wesley, 2005). An eclectic technologist, Robert is coauthor of two previous books, *Building Systems from Commercial Components* (Addison-Wesley, 2002) and *Modernizing Legacy Systems* (Addison-Wesley, 2003).
rcs@cert.org

type of the arguments is determined by the corresponding conversion specifier in the format string. For example, if a %d conversion specifier is encountered, the formatted output function assumes that the corresponding argument is an integer. If a %s is found, the corresponding argument is interpreted as a pointer to a string. This could result in a program fault, for example, if the corresponding argument was actually a small integer value.

Every time a variadic function consumes an argument, an internal argument pointer is incremented to reference the next argument on the stack. If there is some type confusion, it is possible that the argument pointer is incorrectly incremented. This happens less than you might imagine on a 32-bit architecture such as the 32-bit Intel Architecture (IA-32) because almost all arguments (including addresses, char, short, int, and long int) use four bytes. However, conversion specifiers such as a, A, e, E, f, F, g, or G are used to output a 64-bit floating-point number, thereby incrementing the argument pointer by 8.

The standard C formatted output functions need modifications to print 64-bit integer and pointer values in hexadecimal. The %x modifier will only print out the first 32 bits of the value that is passed to it and increment the internal argument pointer by 4 bytes. To print out a 64-bit pointer, the ANSI C %p directive needs to be used rather than %x or %u. To print 64-bit integers, you need to use the one size specifier.

Solutions

One property of format string exploits is that the number of arguments referenced by the attacker's format string is greater than the arguments in the call to the formatted output function. Unfortunately, there is currently no mechanism by which a variadic function implementation can determine the number of arguments (or preferably the number of bytes) passed, so it is impossible to determine when this limit has been exceeded. If such a mechanism existed, variadic functions (such as printf()) could be implemented in such a way as to prevent most format string vulnerabilities.

One solution that is supported by existing C language standards is for the C language compiler to pass a byte count. The VAX standard

calling sequence (partially implemented in its hardware instructions) did pass a count of the number of long words making up the argument list. This was carried over into Alpha, and HP VMS for Alpha still does this.

If byte count were passed, the va_arg() macro (which currently returns the next argument and increments the argument pointer based on the size of the argument) could also decrement the count and force a runtime-constraint violation when a variadic function attempts to access more arguments than have actually been provided.

While the C Standard allows compiler implementations to pass a byte count for variadic functions and not for normal functions, most implementations do not provide a different calling sequence for variadic functions. A common reason to do so is to preserve compatibility between normal and variadic calls.

Unfortunately, it's unreasonable to modify the C language specification to require a byte count, as this change would break binary compatibility between existing applications and libraries. However, it might be possible to introduce a new syntax that could be used to enable the compiler to pass a byte count.

So, for example, instead of:

```
int printf(const char *format, ...) { }
```

we might have:

```
int safe_printf(const char *format, argc+...); { }
```

or some other, similar syntax.

Type Safety

Knowing the number of arguments does not eliminate the possibility of format string vulnerabilities. For example, the types of those arguments would still not be known, possibly causing confusion if an integer is interpreted as, say, a pointer. However, this information is useful in decreasing the number of such vulnerabilities, as well as increasing the complexity of exploiting those that do exist.

It may be possible to add type safety to variadic functions by placing argument list signatures into symbol tables, for example. It is well within the state of the art to generate code that creates a list of argument

types and to generate versions of variadic functions that examine the expected argument type and the actual argument type and generate a runtime error if it finds an unsafe or insecure mismatch. The biggest drawback of this approach is that it might introduce considerable overhead in processing variadic function calls.

Summary and Conclusion

The current implementation of variadic functions in the C programming language is error prone and a major factor in format string vulnerabilities in C and C++. Changes are possible (but in some cases unlikely) within the current constraints of the C language specification. Requiring a stdarg's variant that requires a compiler implementation to provide a byte count is a possible mitigation for format string exploits, but it does not address type safety concerns. A more comprehensive solution that addresses type safety concerns should be researched. In the meantime, programmers should take care that untrusted user input is not incorporated into format specifications for formatted I/O functions and that other uses of variadic functions cannot be used to compromise system security. Better implementations for the average() function, for example, include:

1. Giving the number of arguments followed by the values average(3, 5, -1, 2)
2. Giving the number of arguments followed by an array pointer average(3, a)

The first of these implementations is the "poor man's" equivalent to having the compiler automatically pass the argument count (but requires additional programming that may also be erroneous).

Acknowledgments

I would like to acknowledge the contributions of my coworkers, in particular Corey Cohen and Hal Burch, who originally suggested the alternative vararg syntax, and Pamela Curtis, Ken MacInnis, Art Manion, and Jeff Havrilla for their review comments. I would also like to acknowledge the contributions of my fellow members of the SC22 WG14 C standard language committee, including Randy Meyers, John Levine, Martyn Lovell, and Dave Prosser. 🍌

LINUXWORLD MAGAZINE WWW.LINUXWORLD.COM