Format Strings

G. Lettieri

18 October 2023

1 Introduction

We now introduce a class of vulnerabilities and attack vectors involving *format strings*.

"Format strings" are the control strings that are passed to the `printf()` family of functions and contain the output template for the functions. These functions are vulnerable whenever the attacker can control the format string itself.

These vulnerabilities can be very powerful in the hands of a skilled attacker. In the worst case, the attacker will be able to perform *arbitrary memory reads* and even *arbitrary memory writes*. That is, the attacker can be able read words from memory addresses chosen by the attacker, or overwrite memory locations chosen by the attacker with values chosen by the attacker.

It should be clear how these powers allow an attacker to completely defeat stack canaries, e.g., by reading the canary from memory, or by overwriting the global canary, or by overwriting a return address without touching the canary.

2 Format string bugs

The attack vectors come from the way *variadic* functions are implemented in C. Variadic functions are declared by ending the list of their arguments with "`...`". For example, `printf()` can be declared as

```
int printf(const char *fmt, ...);
```

Basically, the C compiler handles variadic functions by simply *not* checking the number and types of the arguments that are passed to the function in the "`...`" position. All the arguments found in the call site are put in their place in the registers or on the stack. If the called function needs one of these arguments, it reads the expected location for that argument. The function has no way of knowing if the argument was actually passed by the caller, or if the argument type was the correct one: it will read whatever the expected argument location currently contains, and interpret it as a value of the expected type. Correct functionality depends entirely on the conventions between the caller and the

called program. The programmer must follow these conventions, making sure to pass all the arguments that are actually needed in each call.

In the `printf()` family of functions, the convention is that each format specifier takes an additional argument. For example, in

```
printf("a is %d and b is %d\n", a, b);
```

the first "`%d`" will read the first argument (`a`) after the format string, interpret it as an integer, and print its decimal value; the second "`%d`" will read the next argument (`b`). On 32b systems, the first argument is on the stack, just below the pointer to the format string; the second argument is below the first one, and so on. On 64b systems the first 6 arguments (including the pointer to the format string) are passed in registers, and any additional arguments are pushed on the stack.

Now consider a call like this

```
printf("a is %d and b is %d\n", a);
```

where there are two "`%d`"s, but only one additional argument. This code will compile. At runtime, the `printf()` function will read and print the value of `a` correctly, but then it will also print whatever is stored under `a` on the stack (32b), or the current contents of the **`rdx`** register (64b).[1]

Finally, consider a statement like this

```
printf(buf);
```

where the contents of `buf` are controlled by the attacker. The programmer simply wanted to print a string, but `printf()` interprets every "`%`" character inside `buf` as a format specifier. Each one of these format specifiers needs a corresponding argument and `printf()` will read the registers or the memory locations where that argument should have been, *under the attacker's control* (the correct way to print a string is either `puts(buf)` or `printf("%s", buf)`).

3 Exploiting format string bugs

Now let us play the role of the attacker and assume that we can control a format string used by a victim program.

Probably the best way to think about what we can do, is to think of `printf()` as a new machine with its own programming language. The format string is the program and the instructions are normal characters and format specifiers. The `printf()` machine has its own instruction pointer, pointing to the next character/format specifier to "execute". This pointer moves only forward without jumps in either direction: there are no loops and no conditional branches.

The instructions update the machine state which includes its instruction pointer and:

[1] Modern compilers can be instructed to recognize "`printf`-like" functions and will issue a warning if the conventions are not followed.

1. an argument pointer, pointing to the argument to be used by the next format specifier;

2. an output counter, containing the number of characters that have been output so far.

The machine also produces output—the characters sent to the standard output. For example, any ordinary character, such as "a", can be seen as an instruction to print the character itself. As a side effect, the instruction pointer moves past the character in the string and the output counter is incremented by one, while the argument pointer doesn't change. As another example, a "`%d`" specifier reads the argument pointed to by the argument pointer and moves the argument pointer to the next position, interprets and outputs the argument as an integer, and increments the argument counter by the number of output characters; finally, the instruction pointer moves past the "`%d`" in the string. Surprisingly, the `printf()` machine can also *write to memory*: see the man page for the little-known "`%n`" format specifier. The argument to this specifier must be a pointer to an integer variable. `printf()` will execute it by writing the current output counter into the variable. For example, assume that `cnt1` and `cn2` are two **`int`** variables; then, the following statement

```
printf("AAAAA%nBBB%nCCCC", &cnt1, &cnt2);
```

will assign 5 to `cnt1` and 8 to `cnt2`.

3.1 Stack reads

The simplest way to exploit a format string vulnerability is to leak information from the stack of the process under attack. On 32b systems, a sequence of `%x` specifiers will cause `printf()` to print successive lines from the stack. On 64b systems, the first 5 `%lx` will print the contents of the **`rsi`**, **`rdx`**, **`rcx`**, **`r8`**, and **`r9`**, and any additional `%lx` will start printing successive stack lines. By studying the binary, or simply by observing the output, the attacker may be able to determine which of these lines contains the stack canary. On 32b systems the canary can be read with `%x`, but on 64b you need `%lx`, because `%x` will only read 4 bytes in both systems.

The only real difficulty for the attacker comes from space limitations in the controlled buffer, since the argument pointer is only moved forward by a format specifier. Any format specifier will move the argument pointer by at least one stack line (which is 4 bytes in 32b systems and 8 bytes in 64b systems), since arguments are always aligned to stack lines. Assume that the buffer size is s: the attacker can only move the argument pointer by $\lfloor s/2 \rfloor$ lines, which may not be enough to reach the canary's position.

3.2 Random access to arguments

Another little known fact is that the format string can also access its arguments in *random order* using the "`%`n`$`" syntax, which selects the nth argument

directly. For example,

```
printf("%4$d %1$d %3$d %2$d\n", 10, 20, 30, 40);
```

will print "`40 10 30 20`".

In some cases, this syntax can be used to easily overcome the space limitations that we have mentioned above. If we know that the canary is n stack-lines below the pointer to the format string, "`%`n`$x`" will print it directly on 32b systems, while "`%`$(n+5)$`$lx`" will do the same on 64b ones.

This was possible in old versions of glibc, or even in modern versions if some compile options were not been enabled (see `FORTIFY_SOURCE` below). Instead, according to the standard, random access and (the normal) sequential argument access are mutually exclusive (i.e., the same format string cannot contain both forms), and more importantly, once all the argument numbers have been collected, there can be no gaps left. This means a that a format string like "`%`n`$x`" with $n > 1$ is non-standard, since it references the nth argument without also referencing all the arguments from the 1st to the $(n-1)$th. We can understand why the standard imposes this no-gaps requirement: To jump to the nth argument, `printf()` must know how many stack lines (and registers) are occupied by the arguments up to the $(n-1)$th. However, arguments can occupy a variable number of stack lines, depending on their type. For example, **`long long`** occupies two lines on 32b systems, while **`long double`** takes three lines on 32b systems and 2 lines on 64b systems. To implement random access arguments, the `printf()` function should scan the format string a first time, without producing any output, to collect all the argument types. Then it should start the normal scan, using the types collected in the first scan to compute the correct stack line of each argument. For this algorithm to work, however, the first scan must eventually see all the arguments from the 1st to the highest referenced number. This is how musl libc works, for example.

We can see that, if the no-gaps rule is enforced, random access arguments cannot be used to overcome the space limitations in the buffer. When glibc allows this behaviour, though, it simply assumes that all non-referenced arguments occupy one stack-line each.

However, there may be limits on the maximum number of arguments, so you will usually not be able to use this feature to read memory very far down the stack, or especially at addresses lower than the top of the stack.

3.3 Arbitrary memory reads

The above limitation can be overcome if the attacker can control *both* the `printf()` program (i.e., the format string) and at least some of its arguments. This may be the case, for example, if the format string controlled by the attacker is itself on the stack and can be accessed by the argument pointer.

Suppose that there are o stack-lines between the line pointed to by first argument of `printf()` (included) and the first line of the copy of the format string (excluded). In 32b systems, arguments number 1 to o will read from these

o stack-lines, while argument number $o + 1$ will read from the first line of the format string. In 64b systems, arguments 1–5 will read from the usual registers, arguments 6 to $o + 5$ will read from the o stack-lines, and argument $o + 6$ will read from the first line of the format string. The attacker can therefore put both the instructions and their arguments in the same format string "program".

This is rather useless for instructions like "`%x`", but consider the "`%s`" instruction, instead. Normally, this prints a string, but when reinterpreted as in instruction for our `printf()` machine, it prints the contents of memory starting from the address specified by its argument and stopping at the first null byte. If the attacker can choose the address that the instruction will use, it is an arbitrary memory read instruction.

For example, suppose that o is 2 and the victim program is a 32b one. To read bytes from address `0x11223344` the attacker can prepare the string "`\x44\x33\x22\x11%c%c%s`". The purpose of the two "`%c`" instructions is to move the argument pointer until it points to the beginning of the format string, so that the "`%s`" instruction can take the `0x11223344` address as an argument. Note that we also need the buffer to be stack-line aligned, which may not always be the case. This just means that you may need some padding bytes at the beginning before writing the address.

A problem may arise if there are no null bytes to stop `printf()` before it reaches some unreadable addresses, which may cause the process to be terminated. We can easily overcome this limitation by using a "`%.`m`s`" instruction, which will always read (and print) at most m bytes.

Null bytes in the address, however, can be a problem, since the null byte is a halt instruction for `printf()`. For example, in the format string above a null byte in the address would stop the `printf()` before it could even see the first "`%c`" instruction. However, if null bytes are otherwise allowed in the format string, this is not really a problem: the address can be placed *after* the instructions. For example, suppose we want to read address `0x44002211`, the program is 32b and that o is 1, with the format string stack-line aligned. Then, we can send the string "`%c%c%c%s\x11\x22\x00\x44`". Note that we added an extra "`%c`" to move the argument pointer one step further. If random access is available, this is even easier: "`%3$s\x11\x22\x00\x44`". If null bytes are not allowed anywhere, but the address only contains null bytes in the most significant positions, the attacker can still succeed by placing the non-null bytes of the address at the very end of the string and exploiting any null bytes that might accidentally follow the string in memory.

3.4 Arbitrary memory writes

The ultimate power comes from the ability to overwrite arbitrary memory words with arbitrary values. This can be accomplished by using the "`%n`" instruction, taking the address from the format string itself, and by precisely controlling the output counter.

Controlling the output counter is less difficult than it may seem, since an instruction like "`%`m`c`" will always increment the output counter by exactly m.

If there are also other instructions in the format string, you must be careful to control the number of bytes that they output. This can be done by adding width specifiers to each one of them, but be aware of the exact semantics: "`%`m`s`" will always output *at least* m bytes, while "`%.`m`s`" will always output *at most* m bytes. If you want *exactly* m bytes, you need both: "`%`m`.`m`s`".

Another possible difficulty comes from the fact that, if you want to write a very large value (say, the address of a function), you may have to output an impractical or impossibly large number of bytes. This difficulty can be overcome by using the "`%hn`" instruction, which truncates the counter to a short (2 bytes), or even "`%hhn`", that truncates it to a char. If you use the latter instruction 4 times on consecutive addresses, for example, you can write any 32 bit value one byte at a time, always incrementing the output counter by a maximum of 255 bytes. Note that, if the LSB of the counter is c and you need a value $v < c$, you cannot *subtract* from the counter, but you can increment it by $256 - c + v$ bytes and the LSB will become v.

As an example, suppose that you want to write the value `0x44552233` and the LSB of the output counter starts at 32. You can send

```
"%36c%hhn%17c%hhn%205c%hhn%17c%hhn"
```

The first instruction sets the counter to $32 + 36 = 68 = (44)_{16}$ and the second instruction writes it to memory; the third instruction sets the counter to $68 + 17 = 85 = (55)_{16}$; the fourth instruction writes the new counter to memory; the fifth instruction sets the counter to $205 + 85 = 290 = (122)_{16}$ and the sixth instruction writes *its LSB*—i.e., $(22)_{16}$—to memory; finally, the seventh instruction sets the counter to $290+17 = 307 = (133)_{16}$ and the eight instruction writes the final $(33)_{16}$.

Of course, the above format string is incomplete, since we need to provide arguments for all of the "`%hhn`" instructions. Since we are moving the argument pointer sequentially, we also need to provide a dummy argument to each "`%`m`c`". For example, suppose that o is zero, the format string is stack line aligned, the system is 32b, and we want to write `0x44552233` to memory address `0x01020304`. We can complete the above format string by prefixint it with the following

```
"AAAA\x04\x03\x02\x01BBBB\x05\x03\x02\x01"
"CCCC\x06\x03\x02\x01DDDD\x07\x03\x02\x01"
```

The "`AAAA`", "`BBBB`", and so on, serve as dummy arguments for the `c` instructions and to re-align the next argument to the stack line. The other arguments are the addresses of all the bytes of the target memory location, starting from the least significant one.

Note that `printf()` will also process this part of the string as a program before reaching the part that will reuse this same string for the arguments. As a program, this part of the string only prints bytes, since it contains no format specifications. Howiever, it does increment the output counter, which

will end up being 32. For this reason we assumed an initial counter of 32 in the calculations above.

4 Mitigations

The `gcc` compiler and glibc library include a number of mitigations for this type of attack. The mitigations are enabled when the `_FORTIFY_SOURCE` macro is defined and the optimization level is at least one (`-O` or higher). The macro can be set to either 1 or 2, with the latter enabling stricter checks that may break some program. It is often the case that `_FORTIFY_SOURCE` has already been defined for you, so you only need to enable optimizations to include these mitigations in your programs.

This option enables several checks, both at compile time and at run time, that try to limit or prevent the effects of cetrain types of bugs. As far as format string bugs are concerned, these are the most relevant changes when `_FORTIFY_SOURCE` is set to 2:

- glibc will abort the process if a format string with random access arguments does not use all the arguments;
- glibc will abort the process if a format string containing a "`%n`" operator is read from writeable memory.

You can see how the most advanced uses of format string bugs, and in particular the arbitrary memory write exploits, are made much more difficult to exploit when these checks are in place.

Modern compilers also issue warnings when they see `printf()`-family functions being used in possibily unsecure ways. In `gcc` you can enable these warnings with the `-Wformat-security` compile option.