7. Format Strings

> The `a`'s at the beginning are just for alignment, the `%u`'s to skip bytes in the stack, the `%653300u` is to increment the # of bytes that have been "output", and the `%n` stores that value (whose LSBs have now flipped over to 0) to the location pointed to by the current "argument"—which just happens to point right after the `a`'s in this string. The bytes that replace the X's are the address where proftpd keeps the current user ID...
>
> T. Twillman, *Exploit for proftpd 1.2.0pre6*, Bugtraq mailing list, 1999

"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.

7.1 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

BUG
```
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).

Finally, consider a statement like this

BUG
```
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:

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

or, even better:

FIX
```
puts(buf);
```

7.2 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, see Figure 7.1. The format string (colored) 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 arguments are stored in "argument slots" numbered sequentially from 1. In the 32 bit `printf()` machine, each slot is 4 bytes and the first slot is the stack-line pointed by `esp` immediately before the `call` that jumps to `printf()`.

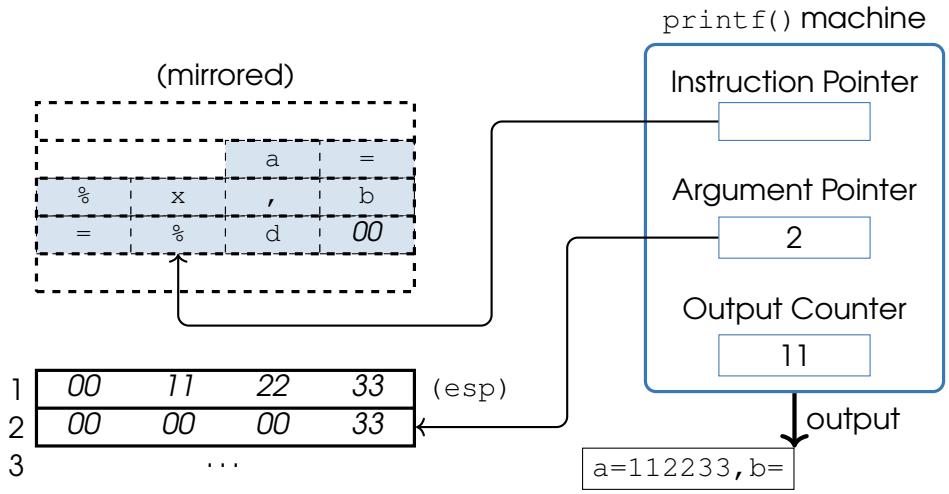


Figure 7.1 – The `printf()` machine (32 bit version)

R Our right-to-left convention in the representation of memory is useful when displaying addresses, but it is annoying when displaying strings, which come out reversed. As a compromise, we sometimes show parts of memory in a "mirrored" fashion, in left-to-right order. We use a dashed pattern for parts of memory displayed in this way and we add "(mirrored)" on top. In Figure 7.1, the part of memory that contains the format string is mirrored.

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

1. an argument pointer, containing the slot number of the next argument;
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 referenced by the argument pointer and moves the argument pointer to the next slot, 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. In Figure 7.1, the machine has already executed `a=%x,b=` and output 11 characters; instruction `%x` has consumed the argument stored in slot 1 and now the argument pointer points to slot 2; the next instruction, `%d`, will read the `0x00000033` argument stored in slot 2, convert it to base ten and output the corresponding characters.

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`.

The 64 bit `printf()` machine (Figure 7.2) is very similar, but the argument slots span 8 bytes and the first 5 slots are the `rsi`, `rdx`, `rcx`, `r8`, and `r9` registers; slot 6 is the stack-line pointed to by `rsp` immediately before the `call printf` instruction. Figure 7.3 shows the intermediate steps of the machine using the same program as in Figure 7.2. Each snapshot shows the state after the execution of a single instruction. Note, in Figure 7.3d, how the execution of "`%x`" advances the instruction pointer

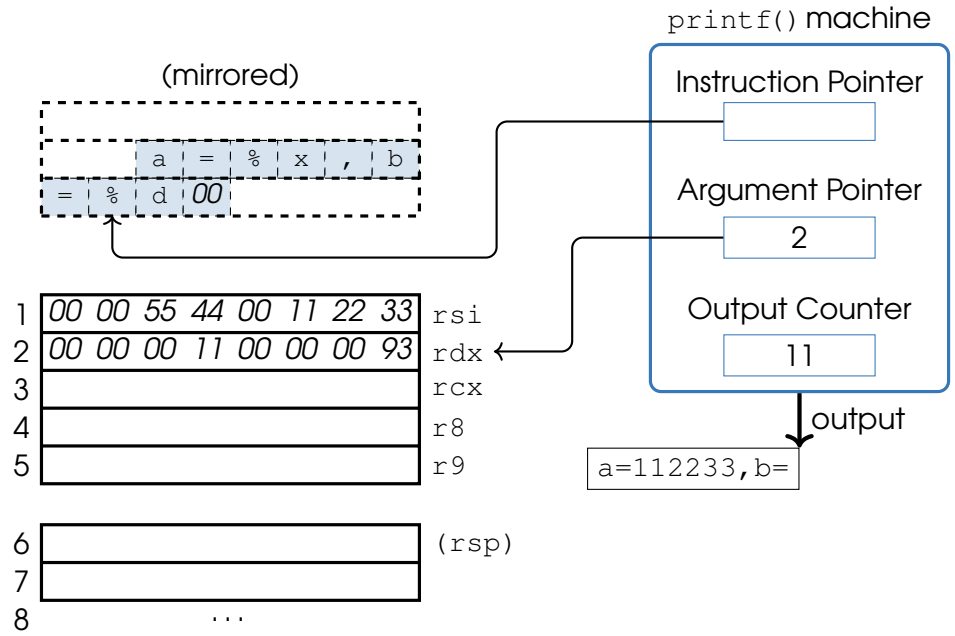


Figure 7.2 – The `printf()` machine (64 bit version)

of two bytes, advances the argument pointer to position 2, outputs 6 characters and adds 6 to the output counter. The characters output by instruction "`%x`" are those that correspond to the hexadecimal string representation of the binary integer stored in argument slot 1, i.e., in the `rsi` register. Since integers are represented on 4 bytes, the highest 4 bytes of `rsi` (`0x0005544`) are simply ignored. Figure 7.2 shows the state immediately after the one Figure 7.3f, where "=" has also been processed.

Exercise 7.1 Draw the state that follows the one in Figure 7.2. ■

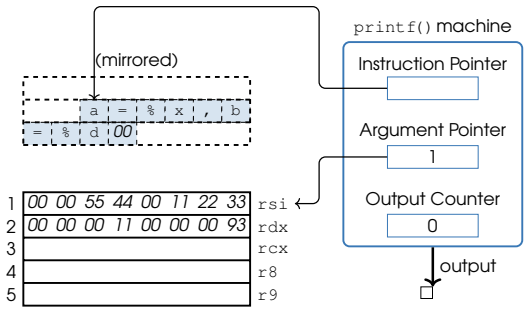
7.2.1 Stack reads

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.

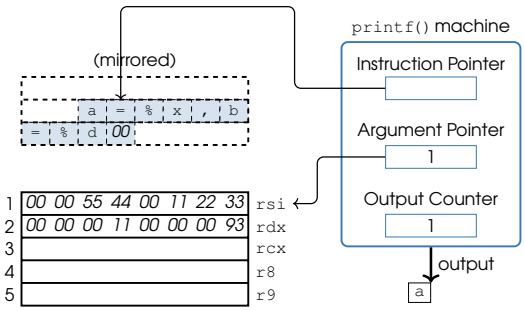
Exercise 7.2 — canary0. Steal the canary and then the flag from challenge *canary0*. ■

7.2.2 Random access to arguments

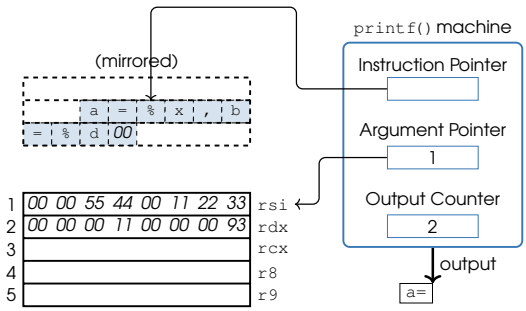
The only real difficulty in the attack of Section 7.2.1 comes from space limitations in the controlled buffer, since the argument pointer is only moved forward by format specifiers, and each format specifier requires space in the format string. Since any format specifier will move the argument pointer by at least one stack line (which is 4 bytes in 32b systems and 8 bytes in 64b systems) the attacker can use a format specifier which is as small as possible: any of `%d`, `%x`, `%c`, ... will do, so the attacker needs to



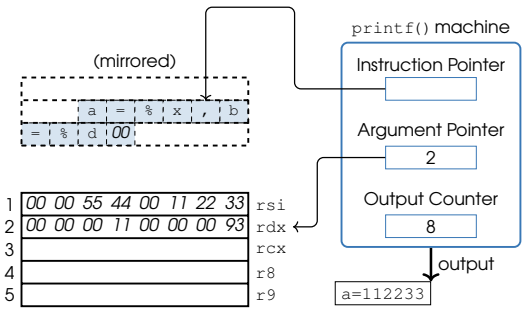
(a) Start



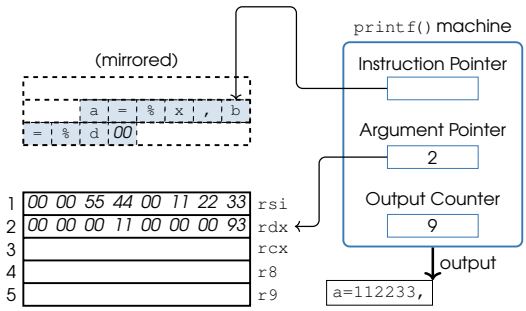
(b) Executed "a"



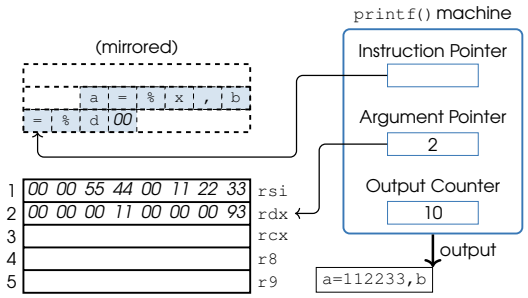
(c) Executed "="



(d) Executed "%x"



(e) Executed ","



(f) Executed "b"

Figure 7.3 – Example evolution of the 64bit `printf()` machine

use at least two bytes of the buffer for each stack line that she needs to skip: if 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.

However, there is another little known fact about format string: arguments can be accessed 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 stack top, "`%`n`$x`" will print it directly on 32b systems, while "`%`$(n+5)$`$lx`" will do the same on 64b ones.

This technique, however, relies on some implementation quirks of the C library. It was available in old versions of glibc, and in modern versions only if some compile options are not enabled (see `FORTIFY_SOURCE` in Section 7.4). According to the C 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.

R When mixing random-access and sequential-access specifiers in GNU libc, remember that the random-access specifiers don't move the argument pointer.

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 behavior, though, it simply assumes that all non-referenced arguments occupy one stack-line each.

Even when the technique is available, there may be limits on the maximum number of arguments, so the attacker 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.

7.2.3 Arbitrary memory reads

The above limitations 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 top line immediately before the call of `printf()` (included) and the first line that contains 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 (see Figure 7.4). 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

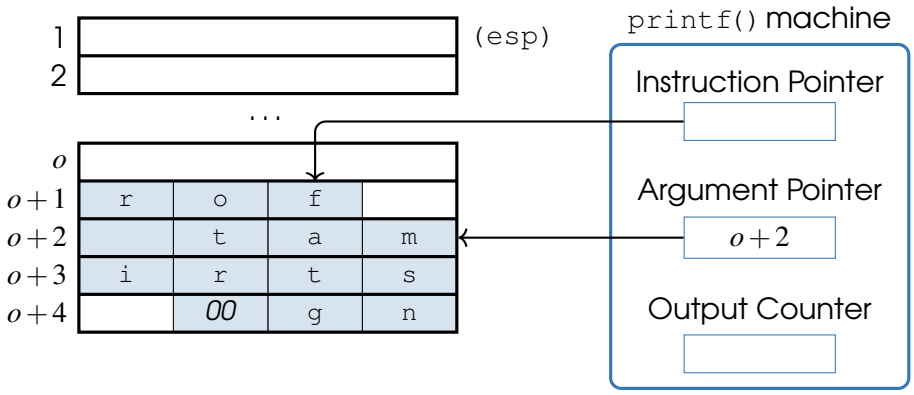


Figure 7.4 – Argument pointer inside the format string

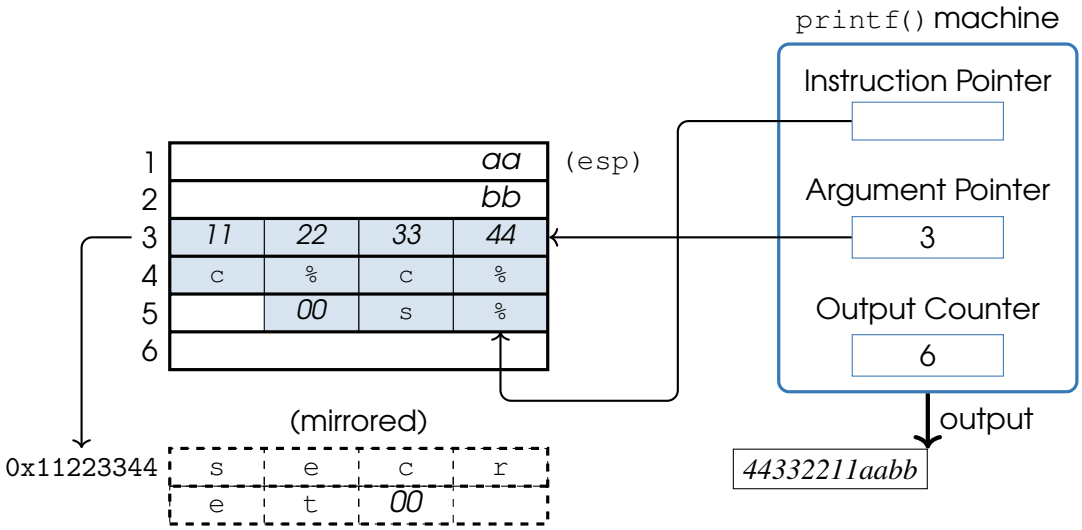


Figure 7.5 – Format string that reads from memory, at address `0x11223344`

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, the victim program is a 32b one, and the buffer is stack-aligned. To read bytes from address `0x11223344` the attacker can prepare the string

```
"\x44\x33\x22\x11%c%c%s"
```

Figure 7.5 shows the `printf()` machine loaded with this format string. A secret value is stored at address `0x11223344`. The format string starts with the address of the secret, so that it overlaps argument slot number 3. The purpose of the two "`%c`" instructions is to move the argument pointer until it points to slot 3, so that the "`%s`" instruction can take the `0x11223344` address as an argument.

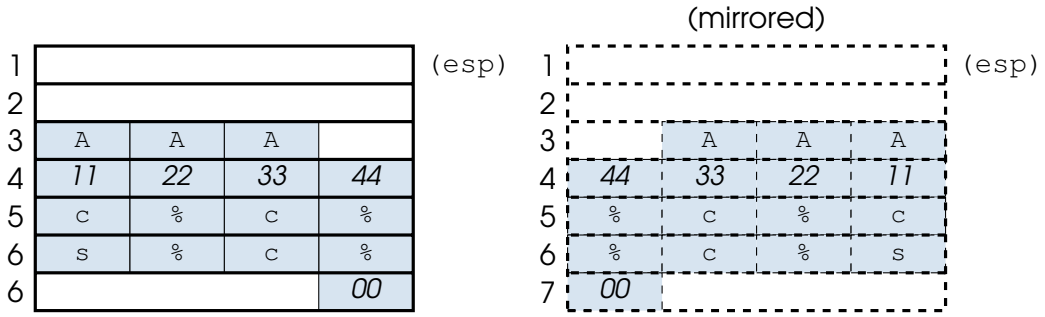


Figure 7.6 – Misaligned format string

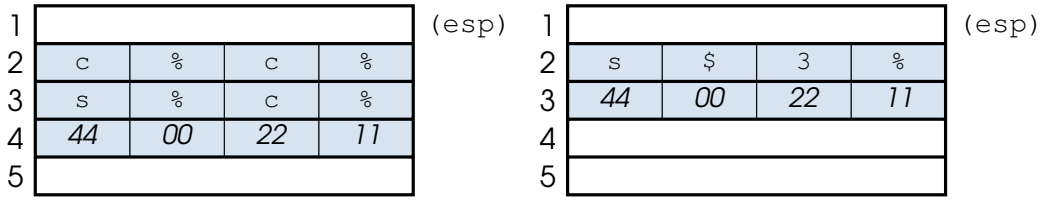


Figure 7.7 – Format strings with embedded null bytes

In the Figure, the instruction pointer has already reached the "`%s`" instruction. The machine has output 6 bytes so far: the four bytes of the address and two random bytes output by the "`%c`" instructions. In the next step the machine will print the secret.

If the buffer is not stack-line aligned you may need some padding bytes at the beginning before writing the address. For example, the format string in Figure 7.6 starts at byte 1 of stack line 3, so we have added three garbage characters to properly align the address. Now the the address is in argument slot 4 instead of 3, so we also added a third "`%c`" before the "`%s`".

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`" (Figure 7.7 on the left). 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`" (Figure 7.7 on the right). 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.

Exercise 7.3 — format1. Steal the `secret` from challenge *format1*. ■

7.2.4 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 `0x33225544` 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 $32 + 36 = 68 =$ `0x44` and the second instruction writes it to memory; the third instruction sets the counter to $68 + 17 = 85 =$ `0x55`; the fourth instruction writes the new counter to memory; the fifth instruction sets the counter to $205 + 85 = 290 =$ `0x0122` and the sixth instruction writes *its LSB*—i.e., `0x22`—to memory; finally, the seventh instruction sets the counter to $290 + 17 = 307 =$ `0x0133` and the eight instruction writes the final `0x33`.

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 `0x33225544` to memory address `0x01020304`. We can complete the above format string by prefixing 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.

Figure 7.8 shows the `printf()` machine loaded with this program. Note that `printf()` will also process the initial part of the string as a program before reaching the part that will reuse this same string for the arguments. Interpreted as a program, this part of the string only prints bytes, since it contains no format specifications. However, 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. In the Figure, the machine has already executed the part of the program that overwrites the three least significant bytes of the word at `0x01020304` and is about to execute the "`%17c`" command. This command will consume argument 7 and output 16 spaces and a "`D`"; the output counter will become 307 (`0x0133`), the argument pointer

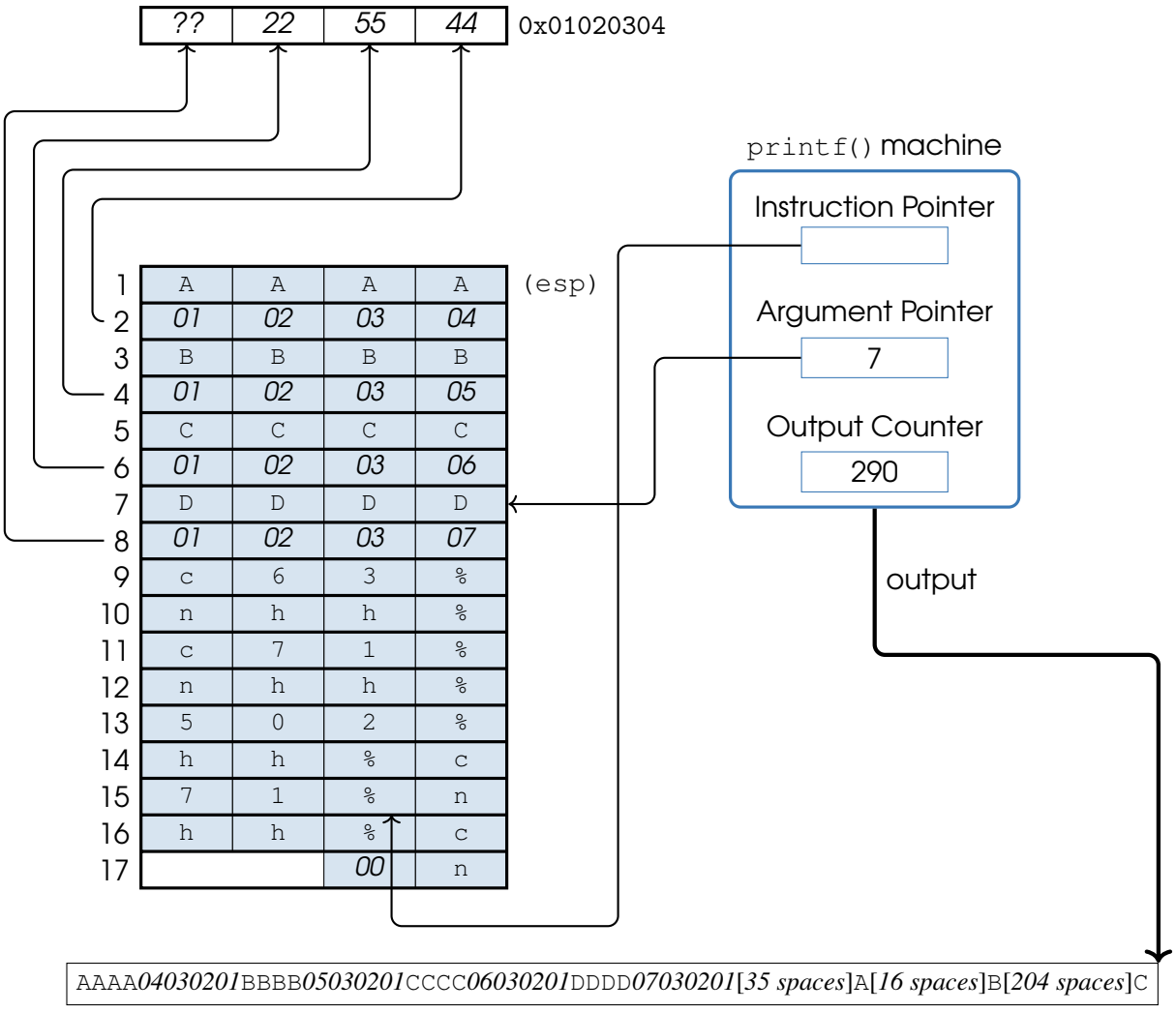


Figure 7.8 – Format string that overwrites memory (write 0x33225544 at address 0x01020304)

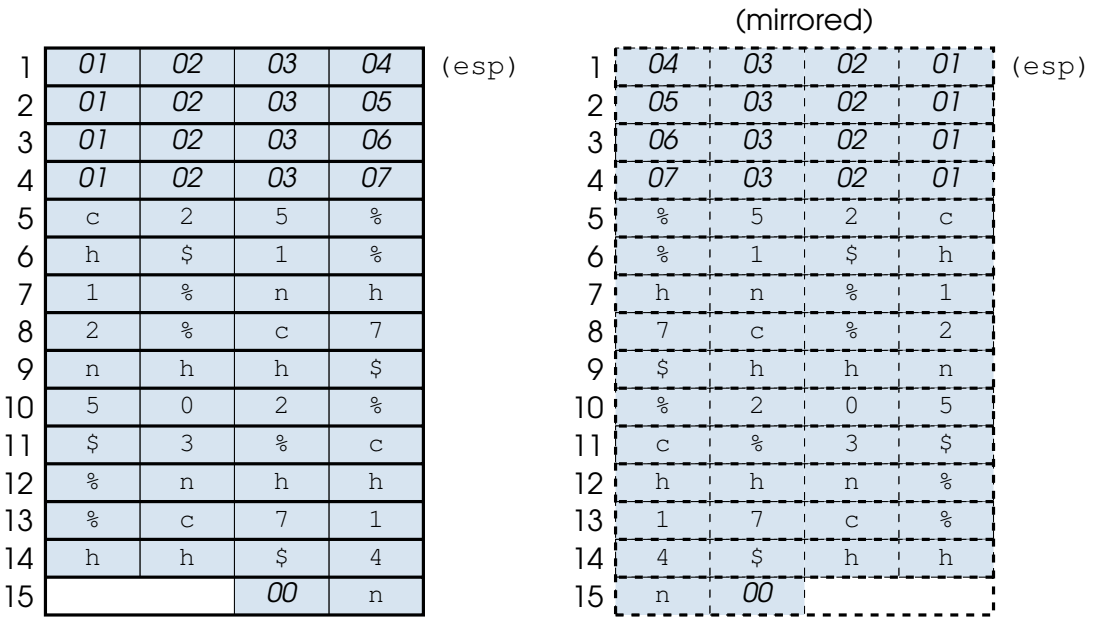


Figure 7.9 – Write 0x33225544 at address 0x01020304 with random access arguments

will move to slot 8 and the instruction pointer will move to the final "%hhn" command, which will write 0x33 in the most significant byte of the target word.

Random access arguments (Section 7.2.2) can slightly simplify the creation of such format strings, since we don't have to provide dummy arguments for the "%*m*c" specifiers. The resulting string is shown on the right of Figure 7.9. Note that the first command in Figure 7.9 is "%52c" instead of "%36c" as in Figure 7.8, since the initial part of the string is now 16 bytes instead of 32, so we need to output 16 more bytes to set the output counter to 0x44.

Exercise 7.4 — format2. Let the server send you the flag in challenge *format2*. ■

Exercise 7.5 — format3. Challenge *format3* is similar to Ex. 7.4, but now you need to be more precise. ■

Exercise 7.6 — canary1. Skip the canary and overwrite the return address directly in challenge *canary1*. ■

7.3 Prevention: Compiler warnings

The gcc compiler can issue warnings when it sees printf()-family functions being used in possibly insecure ways. There are may warning options devoted to this topic, all documented in the gcc manpage (search for Wformat) and most likely already enabled by default in your distribution. We will examine just a couple of them.

The -Wformat option lets the compiler parse the calls to printf()-like functions to check that the optional arguments match the format specifiers in the format string. For example, the first buggy call in Section 7.1 produces the following warning (edited):

```
warning: format '%d' expects a matching 'int' argument
```

This is a break in the abstraction barrier between the compiler and the library, but it is for a good purpose. The `gcc` compiler generalizes this check a bit, since any function can be considered `printf()`-like by annotating it with

```
__attribute__((format(printf, n, m)))
```

where n is the argument position of the format string, and m is the argument position of the first argument that must be checked. For example, the standard `printf()` function can be annotated with $n = 1$ and $m = 2$, while the `fprintf()` function, which takes a `FILE*` before the format string, can be annotated with $n = 2$ and $m = 3$. What cannot be generalized, however, is the parsing of the format string: here the compiler must make assumptions about the semantics of the operators. This is where the barrier is broken, since `gcc` assumes the semantics of the GNU C library, which includes the standard ones with several additions. If you are using a different C library implementation, some checks may not make sense. Another limitation is that the compiler can only check format strings whose value can be inferred at compile time.

The `-Wformat-security` flag lets the compiler issue a warning whenever a `printf()`-like function is used "insecurely". The definition of "insecure" is subject to change, but it currently includes calls like the second buggy one in Section 7.1. That call produces the following warning (edited):

```
warning: format not a string literal and no format arguments
```

Note how the warning logic tries to detect an unsafe usage while still allowing safe ones: non-literal format strings don't cause warnings if they are followed by arguments, since this usage may indicate a deliberate programming technique (the format string is built at runtime, perhaps to change the format based on user preferences). On the other hand, a call like `printf(buf)` makes no possible sense, and the warning is issued.

7.4 Mitigation: FORTIFY_SOURCE

The `gcc` compiler and glibc library include a number of mitigations for security attacks that are enabled when the `_FORTIFY_SOURCE` macro is defined and the optimization level is at least one (`-O` or higher). The patch that implements these mitigations was originally submitted by Red Hat.

The macro can be set to either 0 (i.e., disabled), 1, 2 or 3 (the latter since `gcc` 12). Higher values enable stricter checks that may break some program, or hurt performance. 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 certain types of bugs. As far as format string bugs are concerned, the most relevant changes are applied when `_FORTIFY_SOURCE` is set to 2. In this case, calls to `printf()` are redirected to `__printf_chk()` (and similarly for the other `printf()`-like functions) which will do the following:

- it will abort the process if a format string with random access arguments does not use all the arguments (see Section 7.2.2);
- it will abort the process if a format string containing a "`%n`" operator is read from writable memory.

The first check limits the range of argument slots that can be reached by a forged format string, and the second one tries to prevent arbitrary memory write exploits. Note how the second check doesn't try to ban `%n` altogether (it has legitimate uses), but it essentially limits its usage to constant format strings in the original program. Note that the first check may only break programs that don't conform

to the standard, but the second one is more controversial, since it disallows the legitimate runtime construction of format strings containing `%n`. However, the use of this operator is sufficiently rare that the restriction is considered reasonable.

R The `checksec` utility from the `pwntools` library detects `FORTIFY_SOURCE` by looking for any imported function whose name ends in `_chk`.