18-330 Exploit Notes

Note: This is provided as a resource and is not meant to include all material from lectures or recitations.

Note 1. Buffer overflows and format-string attacks are both instances of a *channeling vulnerability*, a recurring theme in this course. In many computer systems, there is supposed to be a separation between a *control* channel and a *data* channel. The control channel tells the program how to interpret the corresponding data. If the two channels are not properly separated, then an attacker can leverage influence over the data to exert control over the program! In a buffer overflow, the stack mixes data (e.g., values of local variables) with control (return addresses). In a format-string vulnerability, the attacker gains control of the format string (the control channel), which enables abuse of the data channel (the other arguments).

1 Buffer Overflows

Buffer overflows are covered in 213, so for additional background please consult those lectures, as well as Chapter 3 of *Computer Systems: A Programmers Perspective – Volume 2* for more details.

2 Format-String Attacks

2.1 Format Strings and Variadic Functions

Format-string functions are pervasive in programming languages. The canonical example is `printf(char* fmt, ...)`, but there are many more, including `fprintf`, `sprintf`, `syslog`, `setproctitle`, as well as custom error and other reporting functions. These are all examples of *variadic functions.* Unlike most functions, variadic functions do not have a fixed arity; i.e., they can take a variable number of arguments. The number and types of the arguments are dictated by a format argument (e.g., `fmt` above).

At the assembly level, non-variadic functions are relatively easy to handle. The compiler knows the number and types of arguments, so it can emit instructions to put them into the proper locations (e.g., on x64 with the System V calling convention, the compiler puts the first 6 arguments into registers, and then pushes the rest, if any, onto the stack), and it emits instructions for the callee to access the arguments either directly in the registers or via a frame/stack pointer.

For variadic functions, the compiler emits code that will *dynamically* walk through the registers and stack at *run time*, based on the value in the format string. In other words, the format string passed to a variadic function controls the way in which the callee finds and interprets its arguments.

On x64 with the System V calling convention, this means that a variadic function like `printf` looks for a pointer to its format string argument in `rdi`. It then starts parsing that string. Simplifying slightly, each time it finds a `%` followed by a format argument (e.g., `%s` or `%d`), it looks for the corresponding value in the "next" slot for arguments, as dictated by the calling convention. Recall that the order of arguments on x64 System V is: `rdi, rsi, rdx, rcx, r8, r9`. If there are more than six arguments, then the additional arguments are placed on the stack; i.e., the seventh function argument is located at `rsp + 8` (where `rsp` is the value when we started executing the variadic function), placing it just above the return address the callee will use to return to its caller. Likewise, the eigth argument is at `rsp + 16`, the ninth at `rsp + 24`, and so on.

When `printf` analyzes its format string argument (passed in `rdi`), when it first encounters a format argument like `%d`, it will look for the corresponding value in `rsi`. For the second format argument, it will look in `rdx`, and so on, moving first through the registers, and then through the stack slots discussed above. Note that the format argument also tells `printf` how to interpret the value it finds. If `rsi` holds the 64-bit value `0x0000000012345678`, then if the first format argument is `%x`, then `0x0000000012345678` will be printed as a hexidecimal number, but if the first format argument is `%s`, then `0x0000000012345678` will be treated as a pointer to a string, so `printf` will attempt to read characters starting at the memory address `0x0000000012345678` and print those until it encounters a NULL terminator.

Example 1. Suppose we have the following code snippet:

```
char s1[] = ”hello”;
char s2[] = ”world”;
printf(”%s %s %u”, s1, s2, 42);
```

When `printf` begins executing, it will look in `rdi` for its first argument. In `rdi` will be a pointer to the format string `"%s %s %u"`. Based on parsing that string, `printf` will treat `rsi` as a pointer to a string (which in this case will be the value of `s1`), `rdx` as a pointer to a string (which in this case will be the value of `s2`), and `rcx` as an unsigned integer. Hence, it will print “hello world 42”.

In the subsections below, we will look at how an attacker who can control the format string passed to a variadic function can abuse that control to read and/or write to arbitrary memory.

2.2 Viewing the Stack

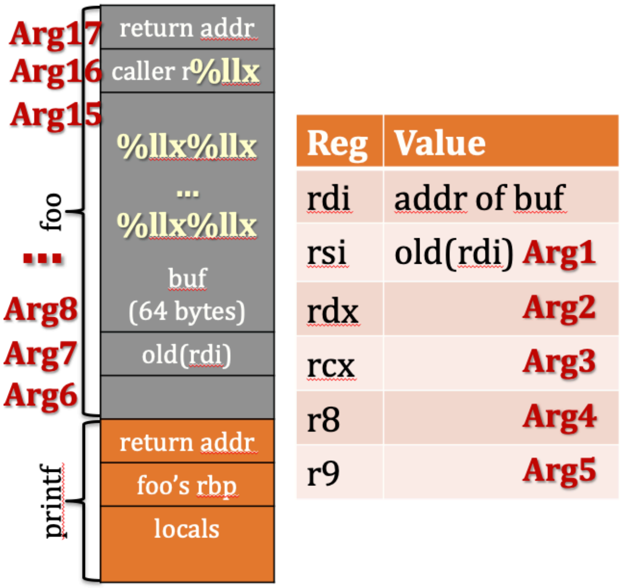
At its simplest, a variadic function can be abused to view a portion of memory, with the simplest portion being the stack. Illicitly reading memory is typically the first step in a more complicated attack (e.g., it can be a useful way to bypass stack canaries, which we discuss later).

Consider the following code snippet:

```
1. int foo(char *fmt) {
2.    char buf[64];
3.    memset(buf, 0, 64);
4.    strcpy(buf, fmt);
5.    printf(buf);
6. }
```

In this function, `buf` is a local variable, and hence, the compiler will emit instructions that will allocate 64 bytes on the stack for it. Above the `buf` on the stack, we will find the caller's value of `rbp` (saved during the prologue of `foo`) and above that, the return address for `foo`'s caller (pushed there by the `call` to `foo`).

On Line 3, we zero out the buffer, and then on Line 4, we copy the format string passed to `foo` into the buffer. Finally, on Line 5, we call `printf`; notice that the programmer incorrectly uses `buf` as the



Reg	Value
rdi	addr of buf
rsi	old(rdi) Arg1
rdx	Arg2
rcx	Arg3
r8	Arg4
r9	Arg5

Figure 1: Registers and stack diagram showing how an attacker can view the stack.

first argument to `printf`, meaning it will be used as the format string that dictates how to interpret additional arguments. If the programmer intended to print `buf` as a string, it would have been safer to call `printf("%s", buf)`. Unfortunately, in benign cases (i.e., when `fmt` does contain a normal string), the behavior with the buggy code above will be the same as the safe version, making it harder to detect vulnerabilities like this.

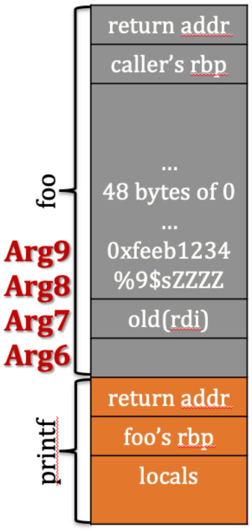
Suppose the attacker wishes to learn some values on the stack and can cause `fmt` to point at the string: `"%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx"`, i.e., 17 copies of `"%llx"`. When `foo` executes, this string will be copied into `buf` (on Line 4).[1] When `printf` is called (on Line 5), it will interpret the string above as a format string. Hence, it will be expecting 17 64-bit integer values as arguments. As discussed earlier, `printf` has no way of knowing how many arguments were really passed, so it does not know that `foo` only passed along one “real” argument. As a result, `printf` will first print the 5 values it finds in registers (i.e., in `rsi, rdx, rcx, r8, r9`), and then it will print the remaining 12 values from the stack (see Figure 1). It will expect the first of those values to be just above the return address in `printf`'s stack frame. It will expect the next one to be 8 bytes above there, and so on. For our particular compiler, the 12th stack value (the 17th argument) printed will be `foo`'s return address.

Obviously, this example can be generalized to view other values of interest on the stack.

2.3 Viewing a Specific Address

Suppose we want to view not just a stack address, but a very specific address in memory (e.g., `0xfeeb1234`), perhaps because we know an important secret key is stored there. The important observation is that an adversary can use its control over the data channel to put the desired address into memory, and then use its

[1] As a side effect, this will overwrite the lower bytes of the caller's `rbp` that was saved on the stack (since the string above is 68 bytes long, plus 1 byte for the NULL terminator), which may cause the program to crash later.



Reg	Value
rdi	addr of buf
rsi	old(rdi) Arg1
rdx	old(rdi) Arg2
rcx	Arg3
r8	Arg4
r9	Arg5

Figure 2: Registers and stack diagram showing how an attacker can view a specific address.

control over the control channel (the format string) to cause `printf` (or other variadic function) to interpret that data as a memory address to be printed.

As an example, using the same code snippet for `foo` from Section 2.2, suppose the attacker arranges for `fmt` to point at the string `"%9$sZZZZ\x34\x12\xeb\xfe"`. The `strcpy` on Line 4 will place this string in the `buf` array on the stack. On the next line, `printf` takes `buf` as its first (and only intended) argument, which means that `printf` will interpret it as a format string. The first format specifier is `%9$s`, which is a little known format type that says to print the 9th argument as a string (the 's' indicates the string type). Of course, `printf` does not know that it was only called with one argument, so it will blithely go looking for its 9th argument following the algorithm outlined in Section 2.1. We know that only the first five non-format-string arguments to `printf` will fit into registers, so the rest must be on the stack. As shown in Figure 2, in this instance, the attacker's careful choice of format string results in the value `0xfeeb1234` sitting on the stack at just the right slot; i.e., the slot where `printf` will look for its 9th argument. Because `printf` was told to treat that argument as a string, it will interpret `0xfeeb1234` as an address and start printing out the data stored at that address until it hits a NULL terminator.

Note 2. If we want to use the address `0xfeeb1234`, why does the format string use the reverse set of bytes, i.e., `"x34\x12\xeb\xfe"`? Remember that `strcpy` (conceptually) copies the source string to the destination string one byte at a time. Hence, in memory, we will find the byte `0x34` at the lowest memory address, the byte `0x12` at that address plus 1, and so on. If we read those four bytes from memory and interpret them as a 32-bit value, then on a little-endian machine, the integer value is `0xfeeb1234`.

Note 3. You may wonder why we included the `"ZZZZ"` in the format string. Notice that each stack "slot" is 8 bytes wide (on x64). The `"%9$s"` portion of the attacker's format string is only 4 bytes. Hence, without the `"ZZZZ"`, the stack would contain `"%9$s\x34\x12\xeb\xfe"` in the stack slot where `printf` looks for its 8th argument and zeroes in the slot for its 9th argument, so the attack would fail.

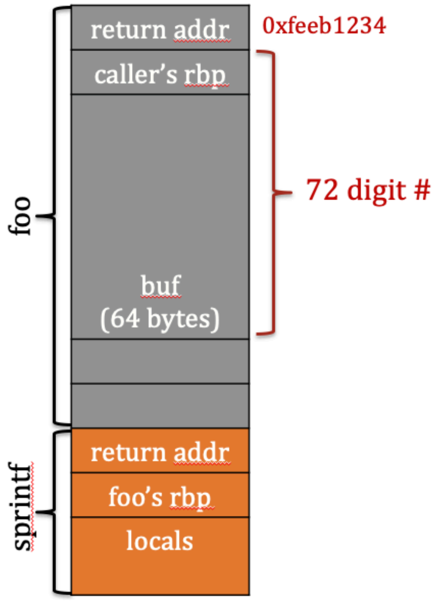


Figure 3: Stack diagram showing how an attacker can overwrite the return address by hijacking a call to sprintf.

2.4 Buffer Overflow by Format String

In the sections above, we focused on ways of reading information from memory. However, variadic functions can also be abused to write to memory as well. One "brute force" way to do so is via a variadic function like `sprintf`, which computes a formatted string and stores it in a caller-supplied buffer.

Consider the following code snippet:

```
int foo(char *fmt) {
    char buf[64];
    memset(buf, 0, 64);
    sprintf(buf, fmt);
}
```

Suppose the adversary can arrange for `fmt` to point at the string `"%72u\x34\x12\xeb\xfe"`. The initial format specifier `%72u` tells printf to treat the first argument as an unsigned integer, but to format the integer such that it takes up 72 characters (i.e., to pad it with enough spaces such that the spaces plus the characters representing the integer sum to 72). This kind of format specifier can be useful when printing tables of data where the numbers need to align even if some are longer or shorter than average.

When `sprintf` processes its format string, `%72u` tells it to look for its first argument (this time in `rdx`, since rdi holds the pointer to buf, and rsi holds the pointer to fmt) and print 72 characters to represent it. The remainder of the format string tells sprintf to print the eight hex digits (i.e., four bytes) corresponding to 0xfeeb1234 (see Figure 3). Hence, these four bytes will overwrite the lower four bytes of the return address stored on the stack, giving the attacker control over where foo will return to.

2.5 Writing to a Specific Address

Now suppose that we want to *write* to a *specific* address in memory (e.g., our favorite target 0xfeeb1234), perhaps because we know the program we're attacking will eventually use the value at that memory location for a control transfer (e.g., as a function pointer). One way to do this is to abuse the printf specifier %n, which tells printf (and others in that family) to print the number of characters printed thus far in processing the format string. The intended usage is for code like this:

```
int i;
printf("abcde%n\n", (int *) &i);
printf("i = %d\n", i);
```

which will output:

```
abcde
i = 5
```

because at the point where the first call to printf reaches the "%n" specifier in its format string, it has printed 5 characters (namely "abcde"), so 5 is stored in i.

We can use this to write the 4-byte value 0x80402010 to address 0xfeeb1234. Conceptually, we want to arrange for the victim program to make the following series of four printf calls:

```
printf("%16u%n",  330, 0xfeeb1234);
printf("%32u%n",  330, 0xfeeb1234 + 1);
printf("%64u%n",  330, 0xfeeb1234 + 2);
printf("%128u%n", 330, 0xfeeb1234 + 3);
```

In these calls, the value 330 doesn't matter and can really be any value. Looking at the first printf, the important part is that thanks to the width specifier in "%16u", by the time we reach the "%n" format specifier, printf will have printed 16 characters, so it will write 0x00000010 to 0xfeeb1234. Of course, that's not the full value we wanted to write, so that's why we need more calls to printf. In the second call, thanks to the width specifier in %32u, by the time we reach the %n format specifier, printf will have printed 32 characters, so it will write 0x00000020 to 0xfeeb1234 + 1. Note the offset by one. This means that if we look at memory starting from 0xfeeb1234, we'll actually read 0x00002010. Each subsequent printf fills in another byte of the desired value until we finally have 0x80402010. Note that as a side effect, we overwrite the three bytes following 0xfeeb1234 + 3 with zeroes. Let's hope nothing important was there!

Of course, we don't actually need four separate calls to printf. As an exercise to the reader, we suggest you think about how the effects described above could be caused via a single call to printf (hint: you may need to pass a lot more arguments!).