# 3. Memory Safety Vulnerabilities

# 3. Memory Safety Vulnerabilities

## 3.1. Buffer overflow vulnerabilities

We'll start our discussion of vulnerabilities with one of the most common types of errors — *buffer overflow* (also called *buffer overrun*) vulnerabilities. Buffer overflow vulnerabilities are a particular risk in C, and since C is an especially widely used systems programming language, you might not be surprised to hear that buffer overflows are one of the most pervasive kind of implementation flaws around. However, buffer overflows are not unique to C, as C++ and Objective-C both suffer from these vulnerabilities as well.

C is a low-level language, meaning that the programmer is always exposed to the bare machine, one of the reasons why C is such a popular systems language. Furthermore, C is also a very old language, meaning that there are several legacy systems, which are old codebases written in C that are still maintained and updated. A particular weakness that we will discuss is the absence of automatic bounds-checking for array or pointer accesses. For example, if the programmer declares an array `char buffer[4]`, C will not automatically throw an error if the programmer tries to access `buffer[5]`. It is the programmer's responsibility to carefully check that every memory access is in bounds. This can get difficult as your code gets more and more complicated (e.g. for loops, user inputs, multi-threaded programs).

It is through this absence of automatic bounds-checking that buffer overflows take advantage of. A buffer overflow bug is one where the programmer fails to perform adequate bounds checks, triggering an out-of-bounds memory access that writes beyond the bounds of some memory region. Attackers can use these out-of-bounds memory accesses to corrupt the program's intended behavior.

Let us start with a simple example.

```c
char buf[8];
void vulnerable() {
    gets(buf);
}
```

In this example, `gets()` reads as many bytes of input as the user supplies (through standard input), and stores them into `buf[]`. If the input contains more than 8 bytes of data, then `gets()` will write past the end of `buf`, overwriting some other part of memory. This is a bug.

Note that `char buf[8]` is defined outside of the function, so it is located in the static part of memory. Also note that each row of the diagram represents 4 bytes, so `char buf[8]` takes up 2 rows.

`gets(buf)` writes user input from lower addresses to higher addresses, starting at `buf`, and since there is no bounds checking, the attacker can overwrite parts of memory at addresses higher than `buf`.

To illustrate some of the dangers that this bug can cause, let's slightly modify the example:

```c
char buf[8];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

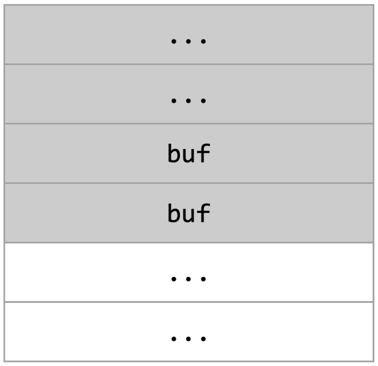


Figure 1: Two words of memory for buf overwritten and two more words of memory above it overwritten

Note that both `char buf[8]` and `authenticated` are defined outside of the function, so they are both located in the static part of memory. In C, static memory is filled in the order that variables are defined, so `authenticated` is at a higher address in memory than `buf` (since static memory grows upward and `buf` was defined first, `buf` is at a lower memory address).

Imagine that elsewhere in the code, there is a login routine that sets the `authenticated` flag only if the user proves knowledge of the password. Unfortunately, the `authenticated` flag is stored in memory right after `buf`. Note that we use "after" here to mean "at a higher memory address".

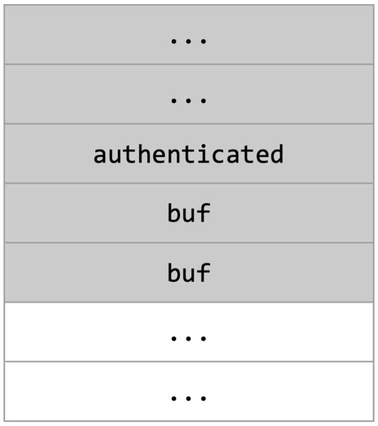


Figure 2: Two words of memory for buf overwritten and an authenticated above it overwritten

If the attacker can write 9 bytes of data to `buf` (with the 9th byte set to a non-zero value), then this will set the `authenticated` flag to true, and the attacker will be able to gain access.

The program above allows that to happen, because the `gets` function does no bounds-checking; it will write as much data to `buf` as is supplied to it by the user. In other words, the code above is *vulnerable*: an attacker who can control the input to the program can bypass the password checks.

Now consider another variation:

```c
char buf[8];
int (*fnptr)();
void vulnerable() {
    gets(buf);
}
```

`fnptr` is a *function pointer*. In memory, this is a 4-byte value that stores the address of a function. In other words, calling fnptr will cause the program to dereference the pointer and start executing instructions at that address.

Like authenticated in the previous example, fnptr is stored directly above buf in memory.

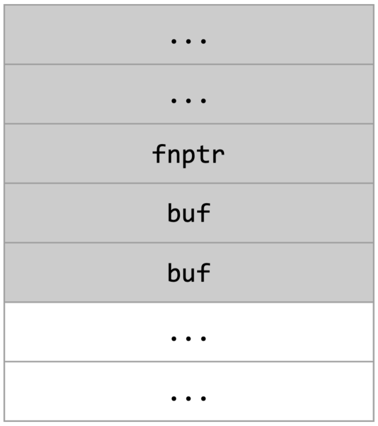


Figure 3: Two words of memory for buf overwritten and a function pointer above it overwritten

Suppose the function pointer fnptr is called elsewhere in the program (not shown). This enables a more serious attack: the attacker can overwrite fnptr with any address of their choosing, redirecting program execution to some other memory location.

Notice that in this attack, the attacker can choose to overwrite fnptr with any address of their choosing—so, for instance, they can choose to overwrite fnptr with an address where some malicious machine instructions are stored. This is a *malicious code injection* attack.

Of course, many variations on this attack are possible: the attacker could store malicious code anywhere in memory and redirect execution to that address.

Malicious code injection attacks allow an attacker to seize control of the program. At the conclusion of the attack, the program is still running, but now it is executing code chosen by the attacker, rather than the original code.

For instance, consider a web server that receives requests from clients across the network and processes them. If the web server contains a buffer overrun in the code that processes such requests, a malicious client would be able to seize control of the web server process. If the web server is running as root, once the attacker seizes control, the attacker can do anything that root can do; for instance, the attacker can leave a backdoor that allows them to log in as root later. At that point, the system has been “*owned*”[1].

The attacks illustrated above are only possible when the code satisfies certain special conditions: the buffer that can be overflowed must be followed in memory by some security-critical data (e.g., a function pointer, or a flag that has a critical influence on the subsequent flow of execution of the program). Because these conditions occur only rarely in practice, attackers have developed more effective methods of malicious code injection.

## 3.2. Stack smashing

One powerful method for exploiting buffer overrun vulnerabilities takes advantage of the way local variables are laid out on the stack.

*Stack smashing* attacks exploit the x86 function call convention. See Chapter 2 for a refresher on how x86 function calls work.

Suppose the code looks like this:

[1] You sometimes see variants on this like pwned, 0wned, ownzored, etc.

```
void vulnerable() {
    char buf[8];
    gets(buf);
}
```

When `vulnerable()` is called, a stack frame is pushed onto the stack. The stack will look something like this:

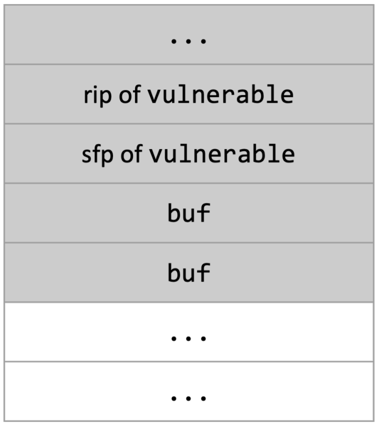


Figure 4: Two words of memory for buf overwritten and the rip and sfp above it overwritten

If the input is too long, the code will write past the end of `buf`, overwrite the sfp, and overwrite the rip. This is a *stack smashing* attack.

Note that even though we are on the stack, which "grows down," our input writes from lower addresses to higher addresses. The stack only grows down when we call a new function and need to allocate additional memory. When we call `gets`, user input is still written from lower addresses to higher addresses, just like before.

Stack smashing can be used for malicious code injection. First, the attacker arranges to inject a malicious code sequence somewhere in the program's address space, at a known address (perhaps using techniques previously mentioned). Let's suppose some malicious code exists at address `0xDEADBEEF`.

Next, the attacker provides a carefully-chosen input: `AAAAAAAAAAAA\xef\xbe\xad\xde`.

The first part of this input is a garbage byte `A` repeated many times. Since the `gets` call writes our user input starting at `buf`, we first need to overwrite all 8 bytes of `buf` with garbage. Furthermore, we don't care about the value in the sfp, so we need to overwrite the 4 bytes of the sfp with garbage. In total, we need $8 + 4 = 12$ garbage bytes at the beginning of our input.

After writing 12 garbage bytes, our next input bytes will overwrite the rip. Recall that the rip contains the address of the next instruction that will be executed after this function returns. If we overwrite the rip with some other address, then when the function returns, it will start executing instructions at that address! This is very similar to the example in the previous section, where we overwrote the function pointer with the address of malicious code.

Since malicious code exists at address `0xDEADBEEF`, the second part of our input, which overwrites the rip, is the address `0xDEADBEEF`. Note that since x86 is little-endian, we must input the bytes in reverse order: the byte `0xEF` is entered first, and the byte `0xDE` is entered last.

Now, when the `vulnerable()` function returns, the program will start executing instructions at the address in rip. Since we overwrote the rip with the address `0xDEADBEEF`, the program will start executing the malicious instructions at that address. This effectively transfers control of the program over to the attacker's malicious code.

Suppose the malicious code didn't already exist in memory, and we have to inject it ourselves during the

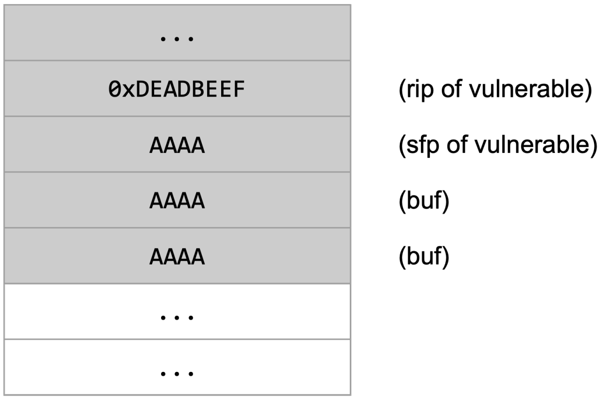


Figure 5: Two words of memory for buf and the sfp overwritten with 0xAAAA and the rip overwritten with 0xDEADBEEF

stack smashing attack. Sometimes we call this malicious code *shellcode*, because the malicious code is often written to spawn an interactive shell that lets the attacker perform arbitrary actions.

Now suppose the shellcode we want to inject is 8 bytes long. How might we place these bytes in memory? Our new input might look like this:

```
[shellcode] + [4 bytes of garbage] + [address of buf]
```

The first part of the input places our 8-byte shellcode at the start of the buffer.

At this point, we've entered 8 bytes, so we've filled up all of `buf`. Our next input will overwrite the sfp, but we want to overwrite the rip. As before, we will need to write some garbage bytes to overwrite the sfp so that we can overwrite the rip afterwards. We need 4 bytes of garbage to overwrite the sfp.

Finally, we overwrite the rip with the address of shellcode, as before. However, this time, the shellcode is located in the buffer, so we overwrite the rip with the address of `buf`. When the function returns, it will start executing instructions at `buf`, which causes the shellcode to execute.

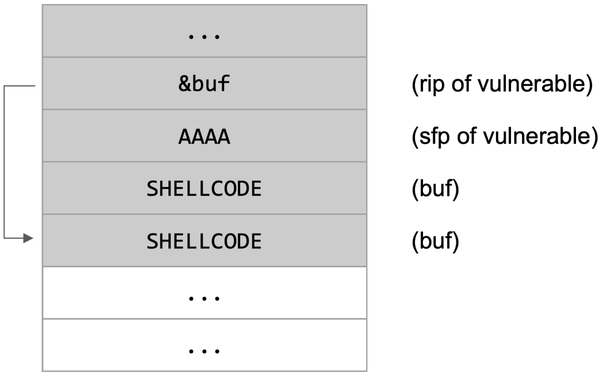


Figure 6: buf overwritten with shellcode, the sfp overwritten with 0xAAAA, and the rip overwritten with the address of buf

Now suppose our shellcode is 100 bytes long. If we try our input from before, the shellcode won't fit in the 12 bytes between the buffer and the rip. It turns out we can still craft an input to exploit the program:

```
[12 bytes of garbage] + [address of rip + 4] + [shellcode]
```

In this input, we place the shellcode directly above the rip in memory. The rip is 4 bytes long, so the address of the start of shellcode is 4 bytes greater than the address of the rip. When the function returns, it will start executing instructions 4 bytes above the address of the rip, where we've placed our shellcode.

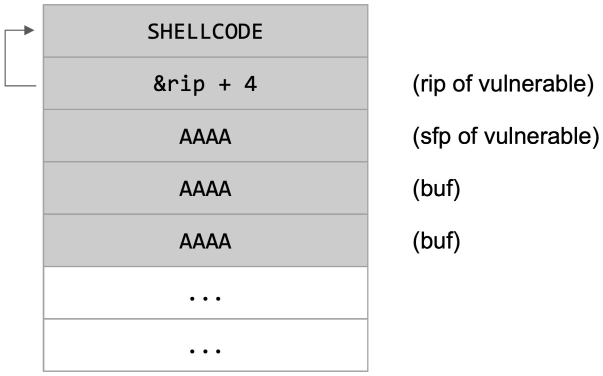


Figure 7: Two words of buf and the sfp overwritten with 0xAAAA, the rip overwritten with the address of rip + 4, and the shellcode overwritten above it

The discussion above has barely scratched the surface of techniques for exploiting buffer overrun bugs. Stack smashing dates back to at least the late 1980s, when the Morris Worm exploited a buffer overflow vulnerability to infect thousands of computers. Buffer overflows gained wider attention in 1998 with the publication of "Smashing the Stack for Fun and Profit" by Aleph One.

Modern methods are considerably more sophisticated and powerful. These attacks may seem esoteric, but attackers have become highly skilled at exploiting them. Indeed, you can find tutorials on the web explaining how to deal with complications such as:

- The malicious code is stored at an unknown location.
- The buffer is stored on the heap instead of on the stack.
- The characters that can be written to the buffer are limited (e.g., to only lowercase letters). Imagine writing a malicious sequence of instructions, where every byte in the machine code has to be in the range 0x61 to 0x7A ('a' to 'z'). Yes, it's been done.
- There is no way to introduce *any* malicious code into the program's address space.

Buffer overrun attacks may appear mysterious or complex or hard to exploit, but in reality, they are none of the above. Attackers exploit these bugs all the time. For example, the *Code Red* worm compromised 369,000 machines by exploiting a buffer overflow bug in the IIS web server. In the past, many security researchers have underestimated the opportunities for obscure and sophisticated attacks, only to later discover that the ability of attackers to find clever ways to exploit these bugs exceeded their imaginations. Attacks once thought to be esoteric to worry about are now considered easy and routinely mounted by attackers.

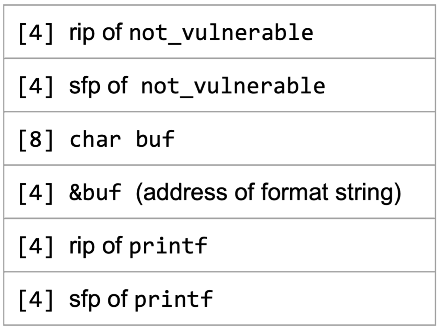
The bottom line is this: *If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.*

## 3.3. Format string vulnerabilities

Let's begin this section by walking through a normal printf call. Suppose we had the following piece of code:

```c
void not_vulnerable() {
    char buf[8];
    if (fgets(buf, sizeof(buf), stdin) == NULL)
        return;
    printf(buf);
}
```

The stack diagram for this function would look something like this:



[4] rip of not_vulnerable

[4] sfp of not_vulnerable

[8] char buf

[4] &buf (address of format string)

[4] rip of printf

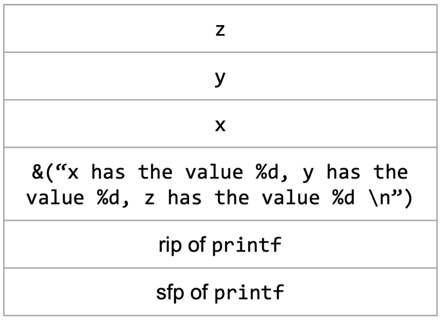
[4] sfp of printf

Figure 8: Initial non vulnerable code with printf

This is commonly another way we choose to represent a stack diagram, where each separate value or variable takes up one row and we dictate how many bytes it takes up in brackets to the left. For instance, the `[8] char buf` means that the `buf` character array takes up 8 bytes.

When the `printf()` function executes, it looks for a format string modifier denoted by a "%" in its first argument located 4 bytes above the RIP of `printf()`. If it finds the modifier, it then looks 8 bytes above the RIP for the "actual" argument (i.e. what the format modifier will be acting upon).

The behavior of the `printf()` function is generally controlled by the format modifier(s) that are passed into the function. The `printf()` function retrieves the parameters that are requested by the format string from the stack. Take, for example, the following line of code: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y, z);` The stack frame for this line of code would look like:



z

y

x

&("x has the value %d, y has the value %d, z has the value %d \n")

rip of printf

sfp of printf

Figure 9: Not vulnerable printf statement

Remember that arguments to a function are pushed onto the stack in reverse order, which is why the address of the format string is at a lower address compared to the values of `x`, `y`, and `z`.

`printf()`'s internal pointer points to the location on the stack 8 bytes above the RIP of `printf()` due to the existence of at least one format string modifier. This internal pointer tells the function where to find the actual arguments that will be modified and eventually printed out.

A logical question you might be asking yourself might be, "Well, all this is well and good when everything works fine, but what happens when there is a mismatch in the number of format string modifiers in the first argument and number of additional arguments?" In other words, suppose our printf statement instead looked like this: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y);` Pay close attention to the fact that the format string asks for 3 arguments by having three `%d` modifiers, but we only pass in 2 arguments (i.e. `x` and `y`).

Surely the C compiler is smart enough to catch such a mistake, you might be thinking. Well, unfortunately you would be wrong. `printf()` is defined as a function with a variable number of arguments; what this means is that as long as `printf()` receives at least one argument, everything looks fine to the compiler! In order to actually spot the mismatch, the compiler would have to understand how the `printf()` function actually works and what format string modifiers are – however, compilers aren't that sophisticated and most of them simply do not perform this kind of analysis.

Ok, well, if the C compiler doesn't catch this type of error, what about the `printf()` function itself? `printf()` simply fetches arguments from the stack according to the number of format modifiers that are present. In cases of a mismatch, it will fetch some data from the stack that does not belong to the function call.

Take the same mismatched `printf()` example we had before: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y);` The `printf()` function's internal pointer will start off 8 bytes above the RIP (since it realizes that there is at least one format modifier present). Thus, the `printf()` function takes the value 8 bytes above the RIP and prints out whatever is located there; in other words, the first `%d` consumes the value located 8 bytes above the RIP of `printf()`. Once this happens, the `printf()` function locates the next format string modifier (the second `%d`), and moves its internal pointer 4 bytes up (so now, the internal pointer is pointing 12 bytes above the RIP of `printf()`), before printing out the value located there. Finally, the `printf()` function will locate the third format string modifier, and again move its internal pointer 4 bytes up (the internal pointer is now pointing 16 bytes above the RIP of `printf()`). However, we never actually passed in a third argument to the `printf()` function, so the value located 16 bytes above the RIP of `printf()` has nothing to do with the `printf()` function at all and is instead some value left over from the previous stack frame. Since the `printf()` function does not know this, however, it looks 16 bytes above the RIP of `printf()` and prints out the value located there.

Similar to how the `%d` format modifier simply makes the `printf()` function print the value located at the expected address, various format string modifiers have different uses. Here are a couple of examples that might be useful:

- %s → Treat the argument as an address and print the string at that address up until the first null byte
- %n → Treat the argument as an address and write the number of characters that have been printed so far to that address
- %c → Treat the argument as a value and print it out as a character
- %x → Look at the stack and read the first variable after the format string
- %[b]u → Print out [b] bytes starting from the argument

The bottom line: *If your program has a format string vulnerability, assume that the attacker can learn any value stored in memory and can take control of your program.*

## 3.4. Integer conversion vulnerabilities

What's wrong with this code?

```
char buf[8];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 8) {
        error("length too large: bad dog, no cookie for you!");
        return;
    }
    memcpy(buf, p, len);
}
```

Here's a hint. The function definition for `memcpy()` is:

```
void *memcpy(void *dest, const void *src, size_t n);
```

And the definition of `size_t` is:

```
typedef unsigned int size_t;
```

Do you see the bug now? If the attacker provides a negative value for `len`, the `if` statement won't notice anything wrong, and `memcpy()` will be executed with a negative third argument. C will cast this negative value to an `unsigned int` and it will become a very large positive integer. Thus `memcpy()` will copy a huge amount of memory into `buf`, overflowing the buffer.

Note that the C compiler won't warn about the type mismatch between `signed int` and `unsigned int`; it silently inserts an implicit cast. This kind of bug can be hard to spot. The above example is particularly nasty, because on the surface it appears that the programmer has applied the correct bounds checks, but they are flawed.

Here is another example. What's wrong with this code?

```
void vulnerable() {
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

This code seems to avoid buffer overflow problems (indeed, it allocates 5 more bytes than necessary). But, there is a subtle problem: `len+5` can wrap around if `len` is too large. For instance, if `len = 0xFFFFFFFF`, then the value of `len+5` is `4` (on 32-bit platforms). In this case, the code allocates a 4-byte buffer and then writes a lot more than 4 bytes into it: a classic buffer overflow. You have to know the semantics of your programming language very well to avoid all the pitfalls.

## 3.5. Off-by-one vulnerabilities

Off-by-one errors are very common in programming: for example, you might accidentally use `<=` instead of `<`, or you might accidentally start a loop at `i=0` instead of `i=1`. As it turns out, even an off-by-one error can lead to dangerous memory safety vulnerabilities.

Consider a buffer whose bounds checks are off by one. This means we can write `n+1` bytes into a buffer of size `n`, overflowing the byte immediately after the buffer (but no more than that).

The following two diagrams are inspired by Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. They show how overwriting a single byte lets you start executing instructions at an arbitrary address in memory.

**Step 1**: This is what normal execution during a function looks like. Consider reviewing the x86 section of the notes if you'd like a refresher. The stack has the rip (saved eip), sfp (saved ebp), and the local variable `buff`. The esp register points to the bottom of the stack. The ebp register points to the sfp at the top of the stack. The sfp (saved ebp) points to the ebp of the previous function, which is higher up in memory. The rip (saved eip) points to somewhere in the code section.

**Step 2**: We overwrite all of `buff`, plus the byte immediately after `buff`, which is the least significant byte of the sfp directly above `buff`. (Remember that x86 is little-endian, so the least significant byte is stored at the lowest address in memory. For example, if the sfp is `0x12345678`, we'd be overwriting the byte `0x78`.) We can change the last byte of sfp so that the sfp points to somewhere inside `buff`. The SFP label becomes FSP here to indicate that it is now a forged sfp with the last byte changed.

Eventually, after your function finishes executing, it returns. Recall from the x86 section of these notes that when a function returns, it executes the following 3 instructions:

`mov %ebp, %esp`: Change the esp register to point to wherever ebp is currently pointing.

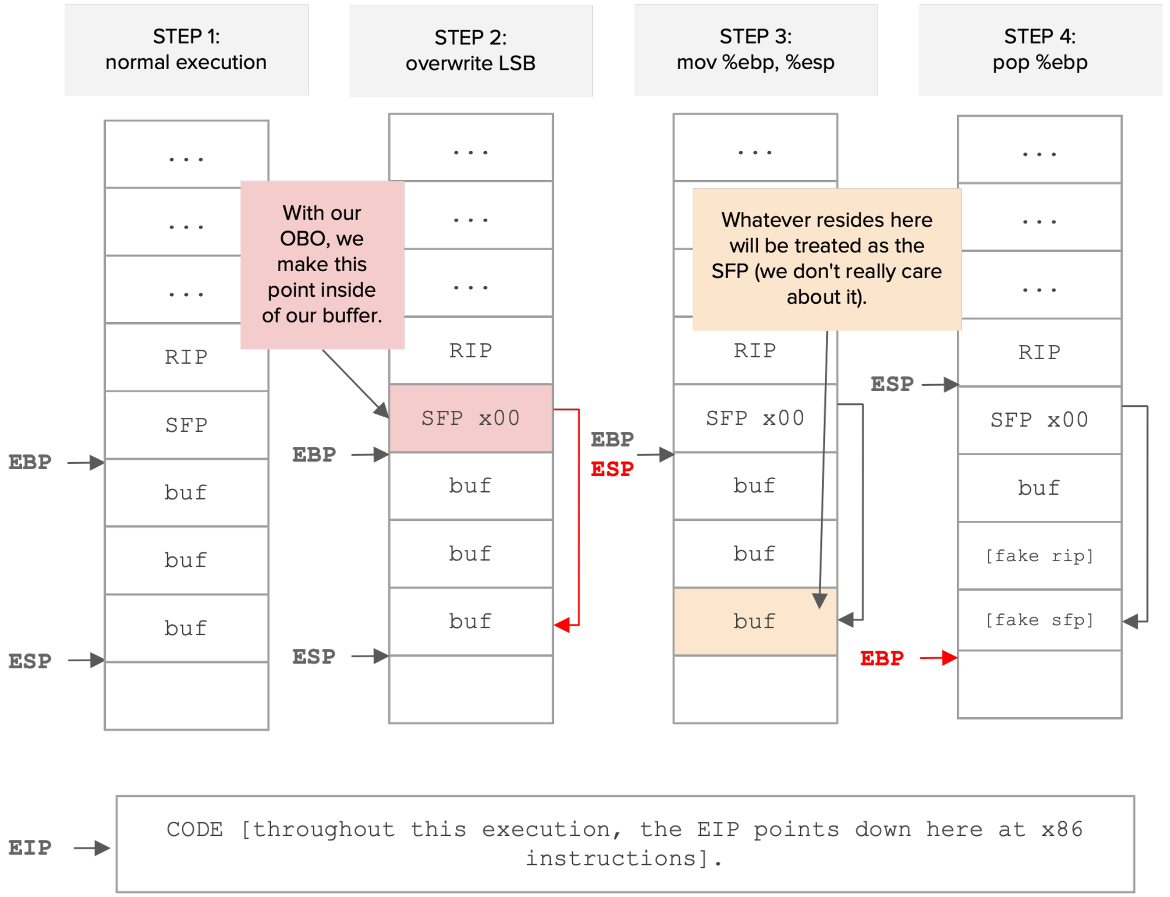

STEP 1:
normal execution
STEP 2:
overwrite LSB
STEP 3:
mov %ebp, %esp
STEP 4:
pop %ebp
...
RIP
SFP
buf
EBP
ESP
With our OBO, we make this point inside of our buffer.
SFP x00
Whatever resides here will be treated as the SFP (we don't really care about it).
[fake rip]
[fake sfp]
EIP
CODE [throughout this execution, the EIP points down here at x86 instructions].

Figure 10: Stack diagrams showing the exploitation of an off-by-one vulnerability for the first return

`pop %ebp`: Take the next value on the stack (where esp is currently pointing, since esp always points to the bottom of the stack), and place it in the ebp register. Move esp up by 4 to delete this value off the stack.

`pop %eip`: Take the next value on the stack and place it in the eip register. Move esp up by 4 to "delete" this value off the stack.

In normal execution, `mov %ebp, %esp` causes esp to point to sfp (recall that ebp always points to sfp during function execution). `pop %ebp` places the next value on the stack (sfp) inside the ebp register (in other words, you're restoring the saved ebp back into ebp). `pop %eip` places the next value on the stack (rip, just above sfp) inside the eip register (in other words, you're restoring the saved eip back into eip).

So now let's see what happens if you execute these same 3 instructions when sfp incorrectly points in the buffer.

**Step 3**: `mov %ebp, %esp`: esp now points where ebp is pointing, which is the forged sfp.

**Step 4**: `pop %ebp`: Take the next value on the stack, the forged sfp, and place it in the ebp register. Now ebp is pointing inside the buffer.

**Step 5**: `pop %eip`: Take the next value on the stack, the rip, and place it in the eip register. Since we didn't maliciously change the rip, the old eip is correctly restored.

After step 5, nothing has changed, except that the ebp now points inside the buffer. This makes sense: we only changed the sfp (saved ebp), so when ebp is restored, it will point to where the forged sfp was pointing (inside the buffer).

The key insight for this exploit is that one function return is not enough. However, eventually, if a second function return happens, it will allow us to start executing instructions at an arbitrary location. Let's walk through the same 3 instructions again, but this time with ebp incorrectly pointing in the buffer.

**Step 6**: `mov %ebp, %esp`: esp now points where ebp is pointing, which is inside the buffer. At this point in normal execution, both ebp and esp think that they are pointing at the sfp.

**Step 7**: `pop %ebp`: Take the next value on the stack (which the program thinks is the sfp, but is actually some attacker-controlled value inside the buffer), and place it in the ebp register. The question mark here says that even though the attacker controls what gets placed in the ebp register, we don't care what the value actually is.

**Step 8**: `pop %eip`: Take the next value on the stack (which the program thinks is the rip, but is actually some attacker-controlled value inside the buffer), and place it in the eip register. This is where you place the address of shellcode, since you control the values in `buff`, and the program is taking an address from `buff` and jumping there to execute instructions.

In step 8, note that there is an offset of 4 from where the forged sfp points and where you should place the address of shellcode. This is because the forged sfp points to a place the program eventually tries to interpret as the sfp, but we care about the place that the program eventually tries to interpret as the rip (which is 4 bytes higher).

Also, note that it is not enough to place the shellcode 4 bytes above where the forged sfp is pointing. You need to put the address of shellcode there, since the program will interpret that part of memory as the rip.

## 3.6. Other memory safety vulnerabilities

Buffer overflows, format string vulnerabilities, and the other examples above are examples of *memory safety* bugs: cases where an attacker can read or write beyond the valid range of memory regions. Other examples of memory safety violations include using a dangling pointer (a pointer into a memory region that has been freed and is no longer valid) and double-free bugs (where a dynamically allocated object is explicitly freed multiple times).

"Use after free" bugs, where an object or structure in memory is deallocated (freed) but still used, are particularly attractive targets for exploitation. Exploiting these vulnerabilities generally involve the attacker

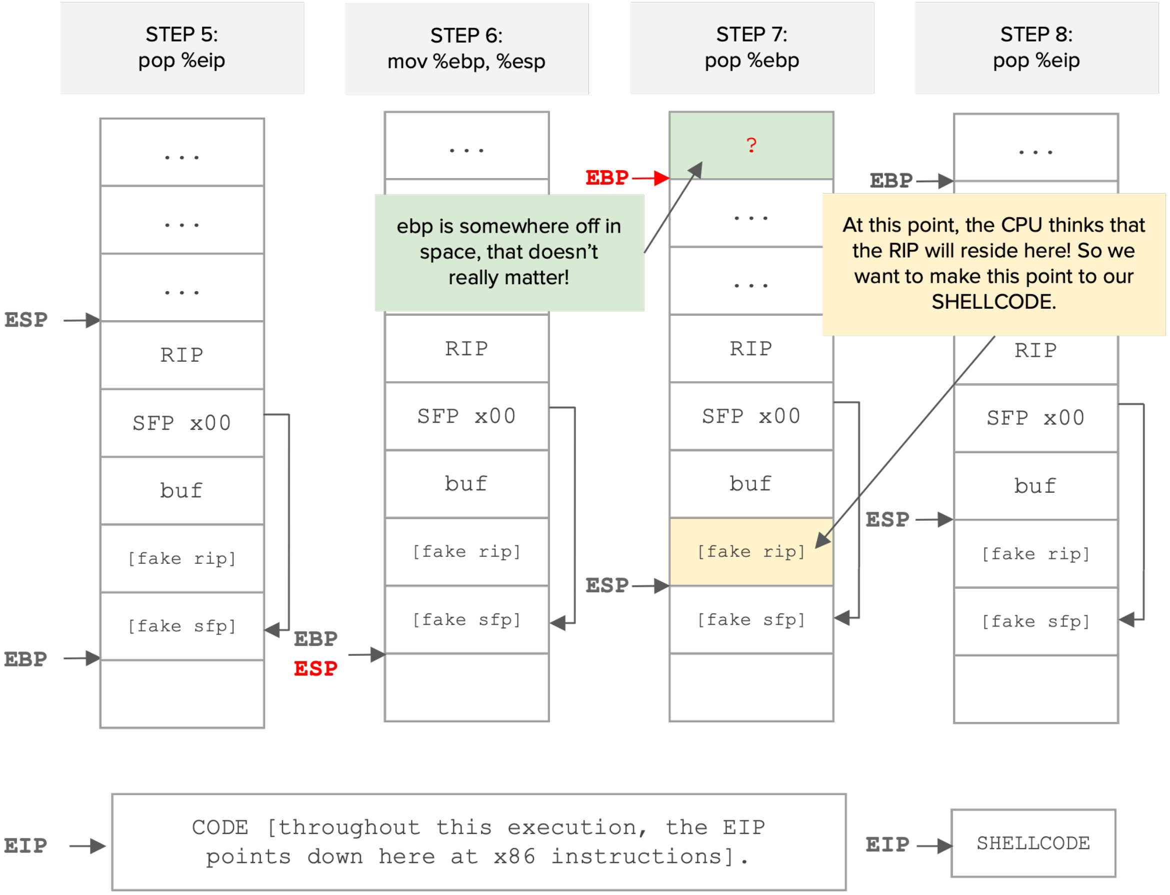

STEP 5:
pop %eip
STEP 6:
mov %ebp, %esp
STEP 7:
pop %ebp
STEP 8:
pop %eip
...
RIP
SFP x00
buf
[fake rip]
[fake sfp]
ESP
EBP
?
ebp is somewhere off in space, that doesn't really matter!
At this point, the CPU thinks that the RIP will reside here! So we want to make this point to our SHELLCODE.
EIP
CODE [throughout this execution, the EIP points down here at x86 instructions].
SHELLCODE

Figure 11: Stack diagrams showing the exploitation of an off-by-one vulnerability for the second return

triggering the creation of two separate objects that, because of the use-after-free on the first object, actually share the same memory. The attacker can now use the second object to manipulate the interpretation of the first object.

C++ vtable pointers are a classic example of a *heap overflow*. In C++, the programmer can declare an object on the heap. Storing an object requires storing a *vtable pointer*, a pointer to an array of pointers. Each pointer in the array contains the address of one of that object's methods. The object's instance variables are stored directly above the vtable pointer.

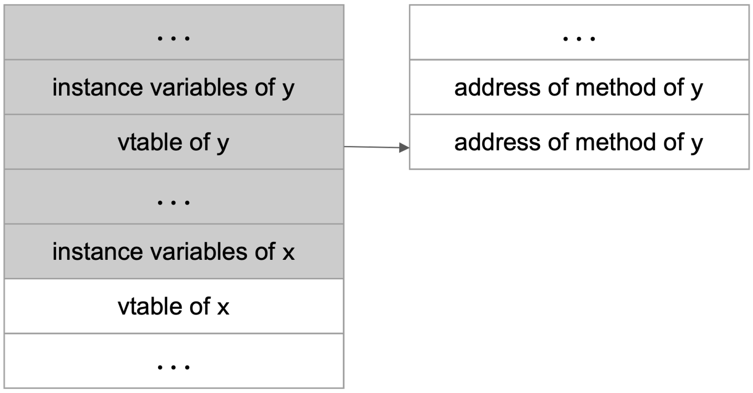


Figure 12: A diagram of a C++ object allocated in the heap, with its instance variables above its vtable

If the programmer fails to check bounds correctly, the attacker can overflow one of the instance variables of object `x`. If there is another object above `x` in memory, like object `y` in this diagram, then the attacker can overwrite that object's vtable pointer.

The attacker can overwrite the vtable pointer with the address of another attacker-controlled buffer somewhere in memory. In this buffer, the attacker can write the address of some malicious code. Now, when the program calls a method on object `y`, it will try to look up the address of the method's code in `y`'s vtable. However, `y`'s vtable pointer has been overwritten to point to attacker-controlled memory, and the attacker has written the address of some malicious code at that memory. This causes the program to start executing the attacker's malicious code.

This method of injection is very similar to stack smashing, where the attacker overwrites the rip to point to some malicious code. However, overwriting C++ vtables requires overwriting a pointer to a pointer.