Memory Safety — CS 161 Computer Security — Textbook

An online version is available at https://textbook.cs161.org.

Textbook by David Wagner, Nicholas Weaver, Peyrin Kao,
Fuzail Shakir, Andrew Law, and Nicholas Ngai

Additional contributions by Noura Alomar, Sheqi Zhang, and Shomil Jain

Last update: August 26, 2021

Contact for corrections: cs161-staff@berkeley.edu

In this section, we will be looking at software security—problems associated with the software implementation. You may have a perfect design, a perfect specification, perfect algorithms, but still have implementation vulnerabilities. In fact, after configuration errors, implementation errors are probably the largest single class of security errors exploited in practice.

In particular, we will look at a particularly prevalent class of software flaws, those that concern *memory safety*. Memory safety refers to ensuring that attackers cannot read or write to memory locations other than those intended by the programmer.

Because many security-critical applications have been written in C, and because C is not a memory-safe language, we will focus on memory safety vulnerabilities and defenses in C.

2 x86 Assembly and Call Stack

This section reviews some relevant concepts from CS 61C and introduces x86 assembly, which is different from the RISC-V assembly taught in 61C.

2.1 Number representation

At the lowest level, computers store memory as individual bits, where each bit is either 0 or 1. There are several units of measurement that we use for collections of bits:

- 1 *nibble* = 4 bits
- 1 *byte* = 8 bits
- 1 *word* = 32 bits (on 32-bit architectures)

A "word" is the size of a pointer, which depends on your CPU architecture. Real-world 64-bit architectures often include stronger defenses against memory safety exploits, so for ease of instruction, this class uses 32-bit architectures unless otherwise stated.

For example, the string `1000100010001000` has 16 bits, or 4 nibbles, or 2 bytes.

Sometimes we use hexadecimal as a shorthand for writing out long strings of bits. In hexadecimal shorthand, a nibble can be written as a single hexadecimal digit. The chart below shows conversions between nibbles written in binary and hexadecimal.

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

To distinguish between binary and hexadecimal strings, we put `0b` before binary strings and `0x` before hexadecimal strings.

Sanity check: Convert the binary string `0b1100000101100001` into hexadecimal.[1]

2.2 Compiler, Assembler, Linker, Loader (CALL)

Recall from 61C that there are four main steps to running a C program.

1. The *compiler* translates your C code into assembly instructions. 61C uses the RISC-V instruction set, but in 161, we use x86, which is more commonly seen in the real world.
2. The *assembler* translates the assembly instructions from the compiler into machine code (raw bits). You might remember using the RISC-V green sheet to translate assembly instructions into raw bits in 61C. This is what the assembler does.
3. The *linker* resolves dependencies on external libraries. After the linker is finished linking external libraries, it outputs a binary executable of the program that you can run (you can mostly ignore the linker for the purposes of 161).
4. When the user runs the executable, the *loader* sets up an address space in memory and runs the machine code instructions in the executable.

2.3 C memory layout

At runtime, the operating system gives the program an address space to store any state necessary for program execution. You can think of the address space as a large, contiguous chunk of memory. Each *byte* of memory has a unique address.

The size of the address space depends on your operating system and CPU architecture. In a 32-bit system, memory addresses are 32 bits long, which means the address space has 2^{32} bytes of memory. In a 64-bit system, memory addresses are 64 bits long. (Sanity check: how big is the address space in this system?[2]) In this class, unless otherwise stated we'll be using

[1] Answer: Using the table to look up each sequence of 4 bits, we get `0xC161`.
[2] Answer: 2^{64} bytes.

32-bit systems.

We can draw the memory layout as one long array with 2^{32} elements, where each element is one byte. The leftmost element has address `0x00000000`, and the rightmost element has address `0xFFFFFFFF`.[3]

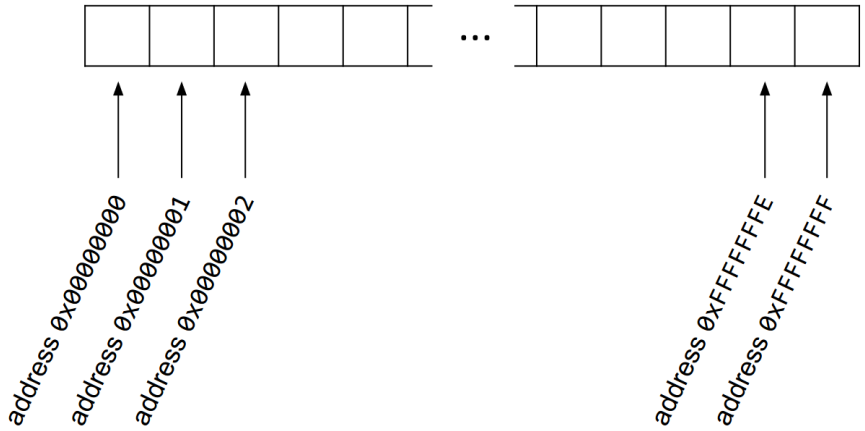
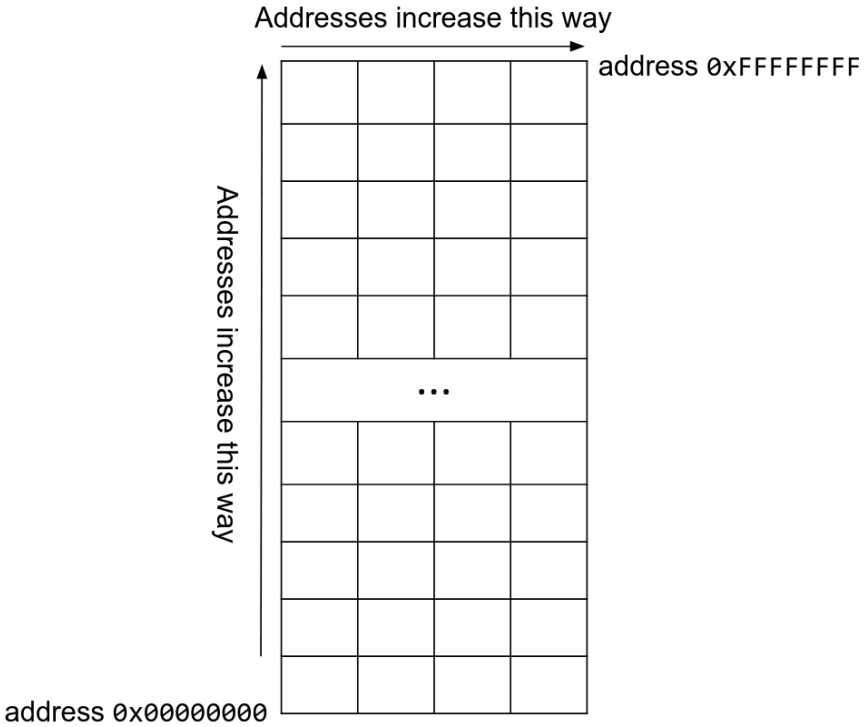


Figure 1: C memory layout drawn as an array. Each box fits one byte of data.

However, this is hard to read, so we usually draw memory as a grid of bytes. In the grid, the bottom-left element has address `0x00000000`, and the top-right element has address `0xFFFFFFFF`. Addresses increase as you move from left to right and from bottom to top.



[3]In reality your program may not have all this memory, but the operating system gives the program the illusion that it has access to all this memory. Refer to the virtual memory unit in CS 61C or take CS 162 to learn more.

Figure 2: C memory layout drawn as a grid. Each box fits one byte of data.

Although we can draw memory as a grid with annotations and labels, remember that the program only sees a huge array of raw bytes. It is up to the programmer and the compiler to manipulate this chunk of raw bytes to create objects like variables, pointers, arrays, and structs.

When a program is being run, the address space is divided into four sections. From lowest address to highest address, they are:

- The *code* section contains the executable instructions of the program (i.e. the code itself). Recall that the assembler and linker output raw bytes that can be interpreted as machine code. These bytes are stored in the code section.
- The *static* section contains constants and static variables that never change during program execution, and are usually allocated when the program is started.
- The *heap* stores dynamically allocated data. When you call `malloc` in C, memory is allocated on the heap and given to you for use until you call `free`. The heap starts at lower addresses and "grows up" to higher addresses as more memory is allocated.
- The *stack* stores local variables and other information associated with function calls. The stack starts at higher addresses and "grows down" as more functions are called.

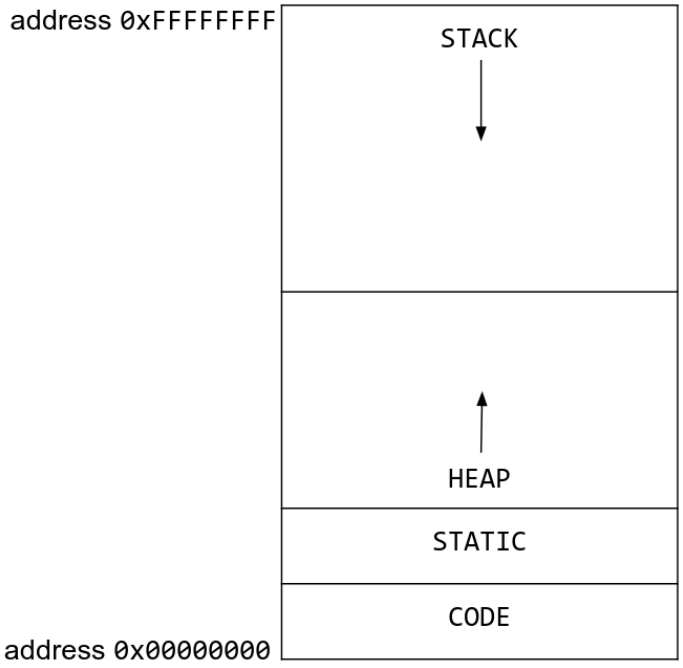


Figure 3: The four sections of the C address space.

2.4 Little-endian words

x86 is a *little-endian* system. This means that when storing a word in memory, the least significant byte is stored at the lowest address, and the most significant byte is stored at the

highest address. For example, here we are storing the word `0x44332211` in memory:

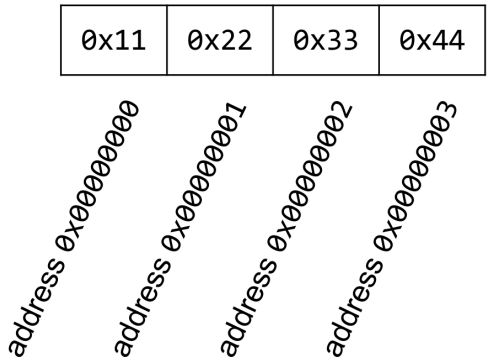


Figure 4: Storing the word `0x44332211` in little-endian.

Note that the least significant byte `0x11` is stored at the lowest address, and the most significant byte `0x44` is stored at the highest address.

Because we work with words so often, sometimes we will write words on the memory diagram instead of individual bytes. Each word is 4 bytes, so each row of the diagram has exactly one word.

Using words on the diagram lets us abstract away little-endianness when working with memory diagrams. However, it's important to remember that the bytes are actually being stored in little-endian format.

2.5 Registers

In addition to the 2^{32} bytes of memory in the address space, there are also *registers*, which store memory directly on the CPU. Each register can store one word (4 bytes). Unlike memory, registers do not have addresses. Instead, we refer to registers using names. There are three special x86 registers that are relevant for these notes:

- *eip* is the *instruction pointer*, and it stores the address of the machine instruction currently being executed. In RISC-V, this register is called the PC (program counter).
- *ebp* is the *base pointer*, and it stores the address of the top of the current stack frame. In RISC systems, this register is called the *FP* (frame pointer)[4].
- *esp* is the *stack pointer*, and it stores the address of the bottom of the current stack frame. In RISC-V, this register is called `SP` (stack pointer).

Note that the top of the current stack frame is the highest address associated with the current stack frame, and the bottom of the stack frame is the lowest address associated with the current stack frame.

If you're curious, the e in the register abbreviations stands for "extended" and indicates that we are using a 32-bit system (extended from the original 16-bit systems).

[4]RISC systems often omit this register because it is not necessary with the RISC stack design. For example, in RISC-V, `FP` is sometimes renamed `s0` and used as a general-purpose register

Since the values in these three registers are usually addresses, sometimes we will say that a register *points* somewhere in memory. This means that the address stored in the register is the address of that location in memory. For example, if we say eip is pointing to `0xDEADBEEF`, this means that the eip register is storing the value `0xDEADBEEF`, which can be interpreted as an address to refer to a location in memory.

Sanity check: Which section of C memory (code, static, heap, stack) do each of these registers usually point to?[5]

2.6 Stack: Pushing and popping

Sometimes we want to remember a value by saving it on the stack. There are two steps to adding a value on the stack. First, we have to allocate additional space on the stack by decrementing the esp. Then, we store the value in the newly allocated space. The x86 `push` instruction does both of these steps to add a value to the stack.

Figure 5: `push %ebx` decrements esp by 4 and stores the value in the ebx register on the stack.

We may also want to remove values from the stack. The x86 `pop` instruction increments esp to remove the next value on the stack. It also takes the value that was just popped and copies the value into a register.

Note that when we pop a value off the stack, the value is not wiped away from memory. However, we increment esp so that the popped value is now below esp. The esp register points to the bottom of the stack, so the popped value below esp is now in undefined memory.

[5] Answer: eip points to the code section, where instructions are stored. ebp and esp point to the stack section.

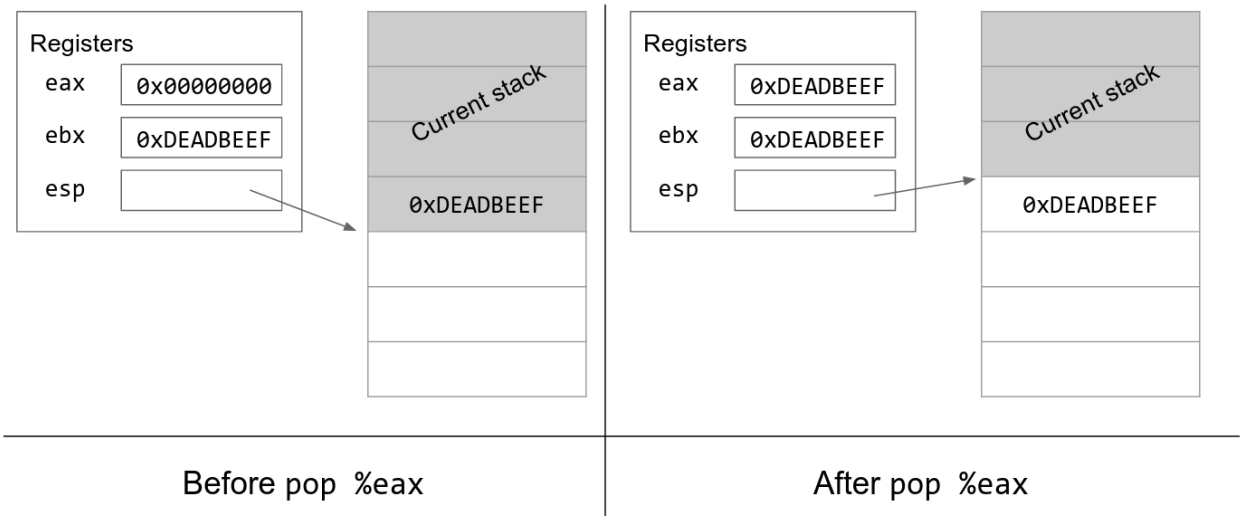


Figure 6: `pop %eax` stores the lowest value on the stack in the eax register and increments esp by 4.

(eax and ebx are general-purpose registers in x86. We use them here as an example of pushing and popping from the stack, but you don't need to know anything else about these registers.)

2.7 x86 calling convention

This class uses AT&T x86 syntax (since that is what GDB uses). This means that the destination register comes last; note that this is in contrast with RISC-V assembly, where the destination register comes first. Suppose our assembly instruction was `addl $0x8, %ebx`; here, the opcode is `addl`, the source is `$0x8`, and the destination is `%ebx`, so in pseudocode this can be read as `EBX = EBX + 0x8`.

References to registers are preceded with a percent sign, so if we wanted to reference `eax`, we would do so as `%eax`. Immediates are preceded with a dollar sign (i.e. $1, $0x4, etc.). Furthermore, memory references use parenthesis and can have immediate offsets; for example, `12(%esp)` dereferences memory 12 bytes above the address contained in ESP. If parentheses are used without an immediate offset, the offset can be thought of as an implicit 0.

Suppose our assembly instruction was `xorl 4(%esi), %eax`; here, the opcode is `xorl`, the source is `4(%esi)`, and the destination is `%eax`. As such, in pseudocode, this can be written as `EAX = EAX ^ *(ESI + 4)`. Since this is a memory reference, we are dereferencing the value 4 bytes above the address stored in ESI.

2.8 x86 function calls

When a function is called, the stack allocates extra space to store local variables and other information relevant to that function. Recall that the stack grows down, so this extra space will be at lower addresses in memory. Once the function returns, the space on the stack is freed up for future function calls. This section explains the steps of a function call in x86.

Recall that in a function call, the *caller* calls the *callee*. Program execution starts in the caller, moves to the callee as a result of the function call, and then returns to the caller after the function call completes.

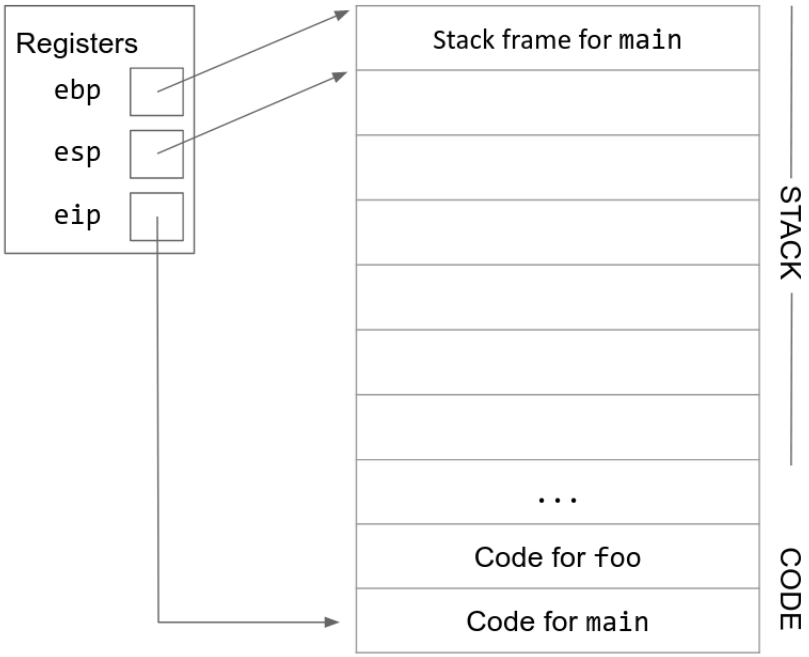
When we call a function in x86, we need to update the values in all three registers we've discussed:

- eip, the instruction pointer, is currently pointing at the instructions of the caller. It needs to be changed to point to the instructions of the callee.
- ebp and esp currently point to the top and bottom of the caller stack frame, respectively. Both registers need to be updated to point to the top and bottom of a new stack frame for the callee.

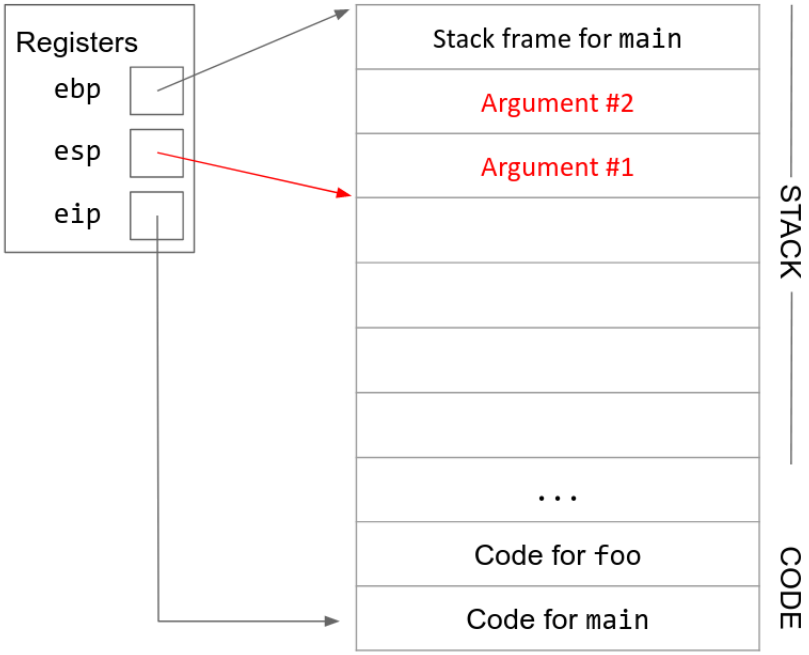
When the function returns, we want to restore the old values in the registers so that we can go back to executing the caller. *When we update the value of a register, we need to save its old value on the stack so we can restore the old value after the function returns.*

There are 11 steps to calling an x86 function and returning. In this example, `main` is the caller function and `foo` is the callee function. In other words, `main` calls the `foo` function.

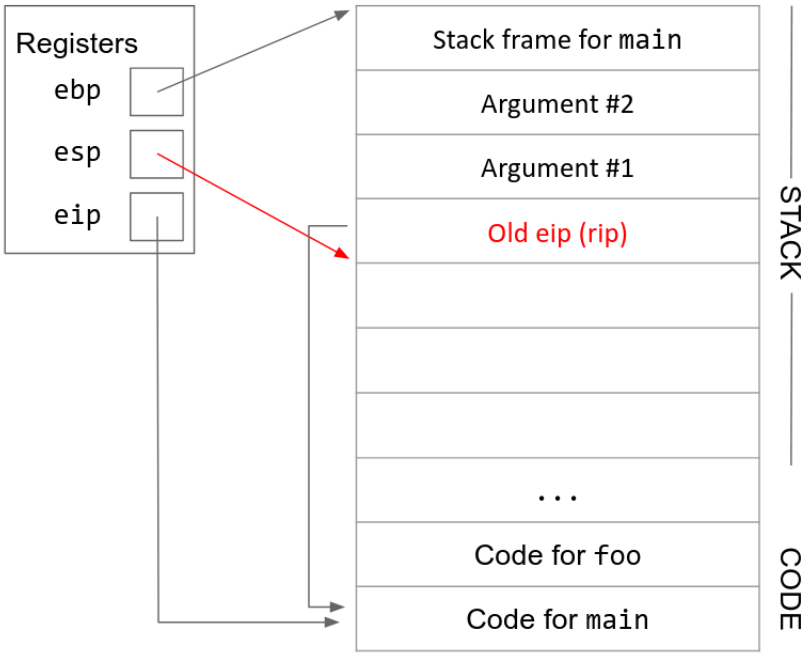
Here is the stack before the function is called. ebp and esp point to the top and bottom of the caller stack frame.



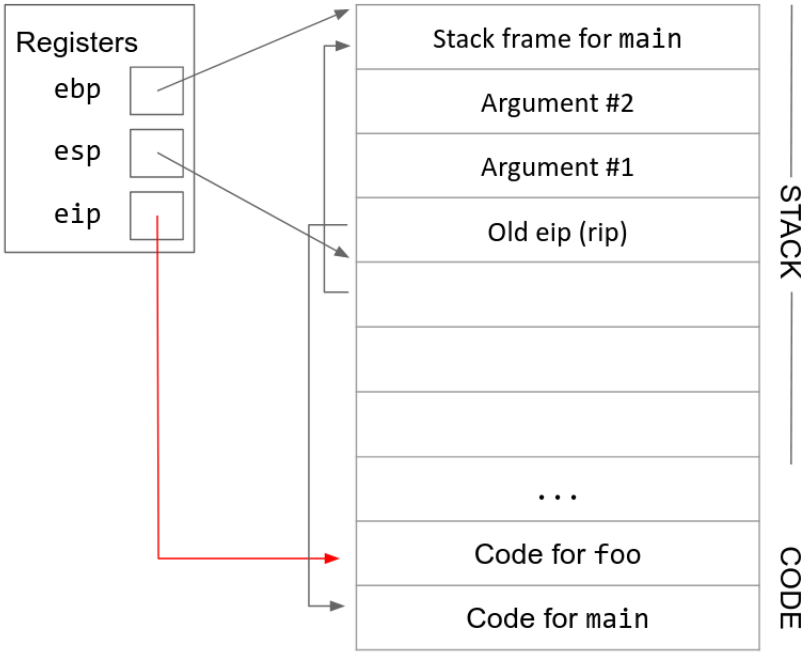
1. Push arguments onto the stack. RISC-V passes arguments by storing them in registers, but x86 passes arguments by pushing them onto the stack. Note that esp is decremented as we push arguments onto the stack. Arguments are pushed onto the stack in reverse order.



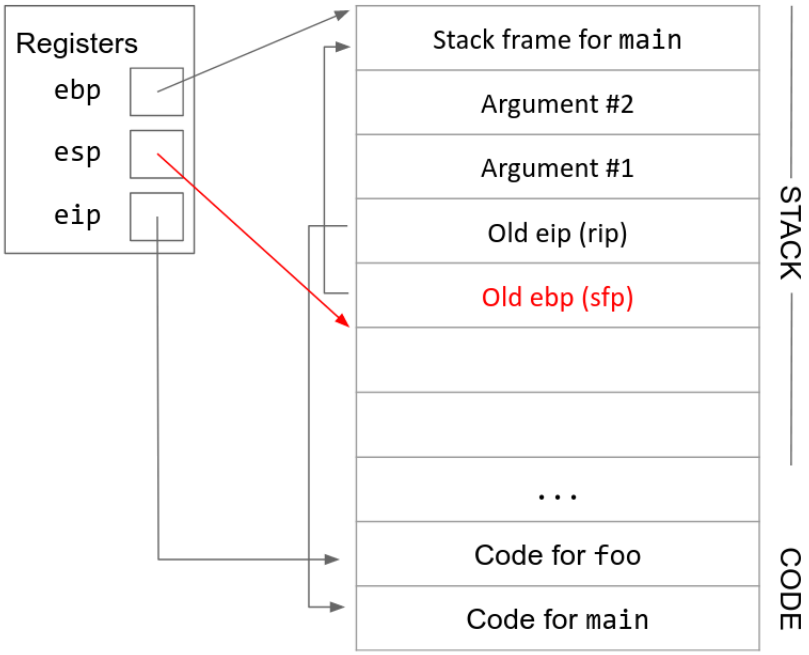
2. Push the old eip (rip) on the stack. We are about to change the value in the eip register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the *old eip* or the *rip* (return instruction pointer).[6]



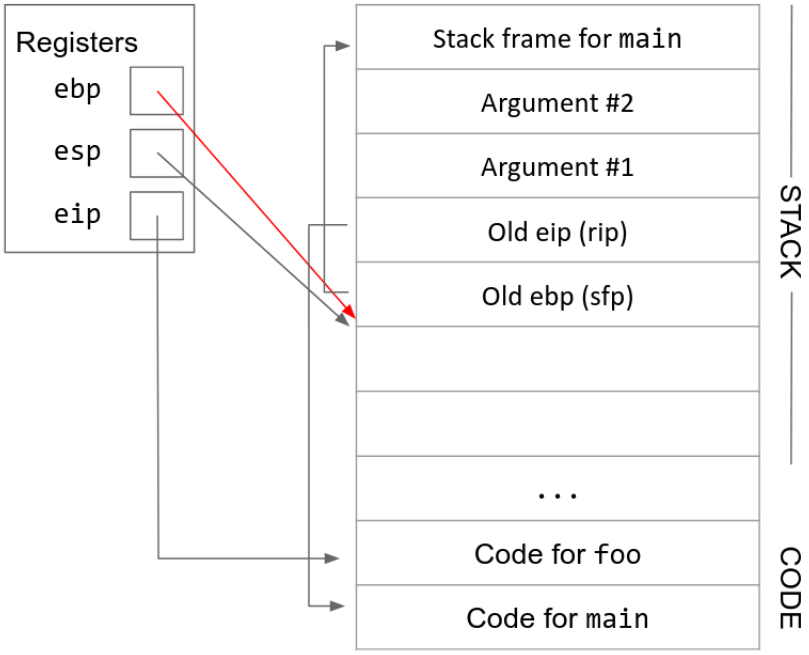
3. Move eip. Now that we've saved the old value of eip, we can safely change eip to point to the instructions for the callee function.



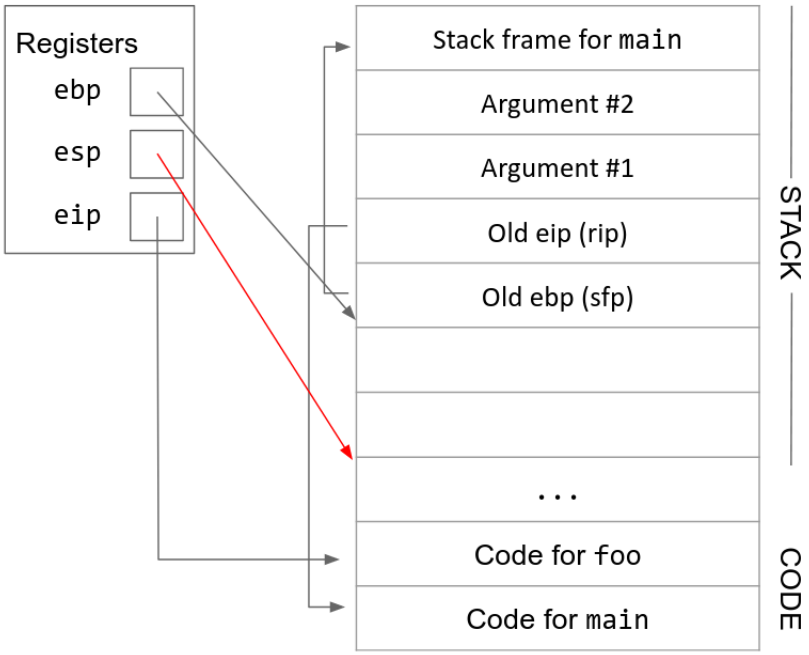
4. Push the old ebp (sfp) on the stack. We are about to change the value in the ebp register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the *old ebp* or the *sfp* (saved frame pointer). Note that esp has been decremented because we pushed a new value on the stack.



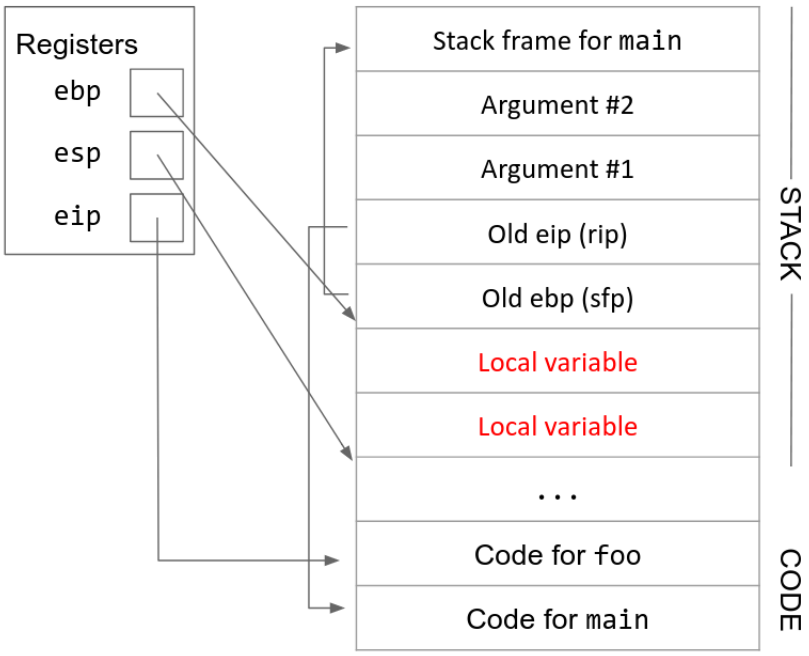
5. Move ebp down. Now that we've saved the old value of ebp, we can safely change ebp to point to the top of the new stack frame. The top of the new stack frame is where esp is currently pointing, since we are about to allocate new space below esp for the new stack frame.



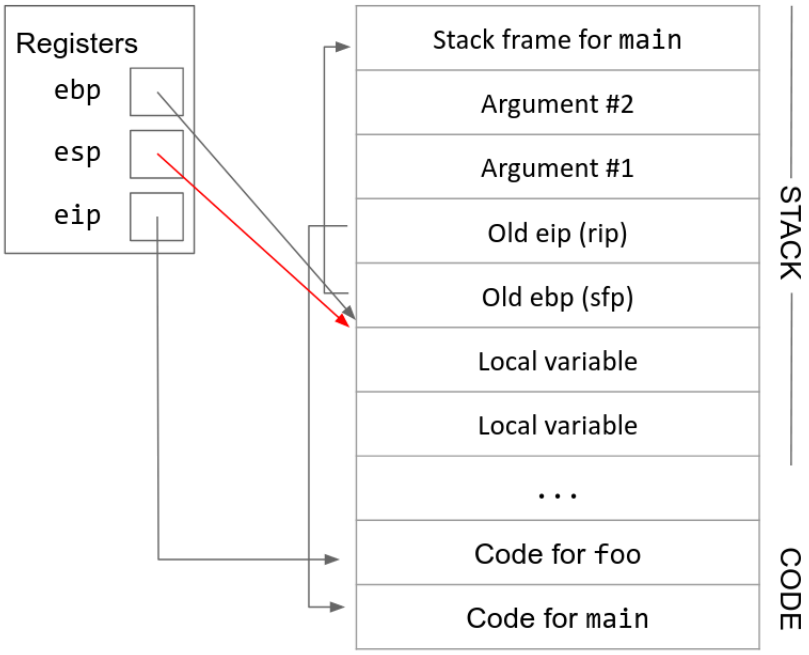
6. Move esp down. Now we can allocate new space for the new stack frame by decrementing esp. The compiler looks at the complexity of the function to determine how far esp should be decremented. For example, a function with only a few local variables doesn't require too much space on the stack, so esp will only be decremented by a few bytes. On the other hand, if a function declares a large array as a local variable, esp will need to be decremented by a lot to fit the array on the stack.



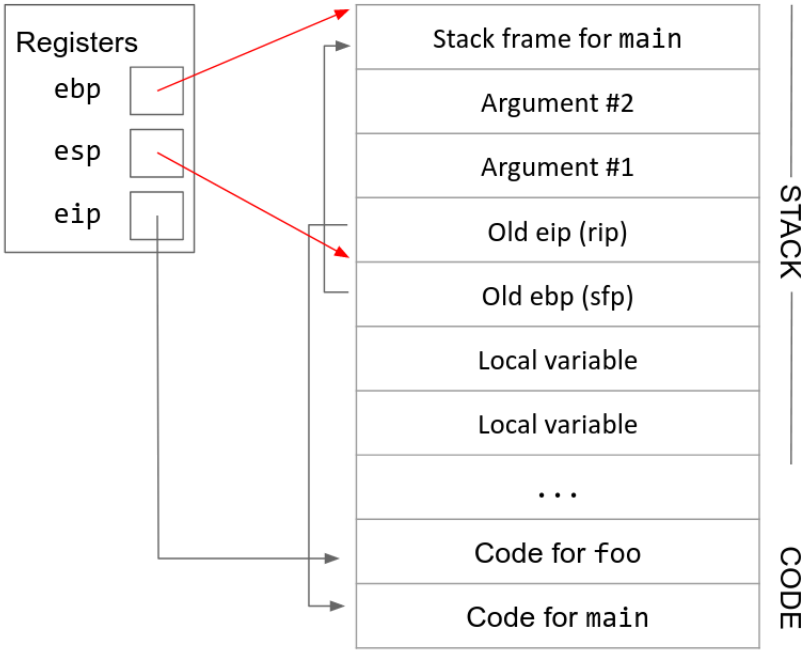
7. Execute the function. Local variables and any other necessary data can now be saved in the new stack frame. Additionally, since ebp is always pointing at the top of the stack frame, we can use it as a point of reference to find other variables on the stack. For example, the arguments will be located starting at the address stored in ebp, plus 8.



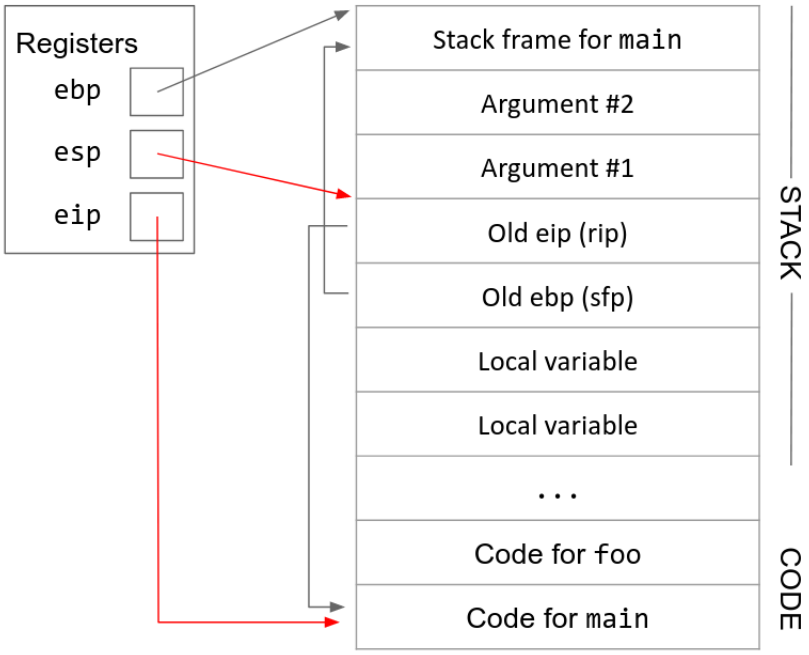
8. Move esp up. Once the function is ready to return, we increment esp to point to the top of the stack frame (ebp). This effectively erases the stack frame, since the stack frame is now located below esp. (Anything on the stack below esp is undefined.)



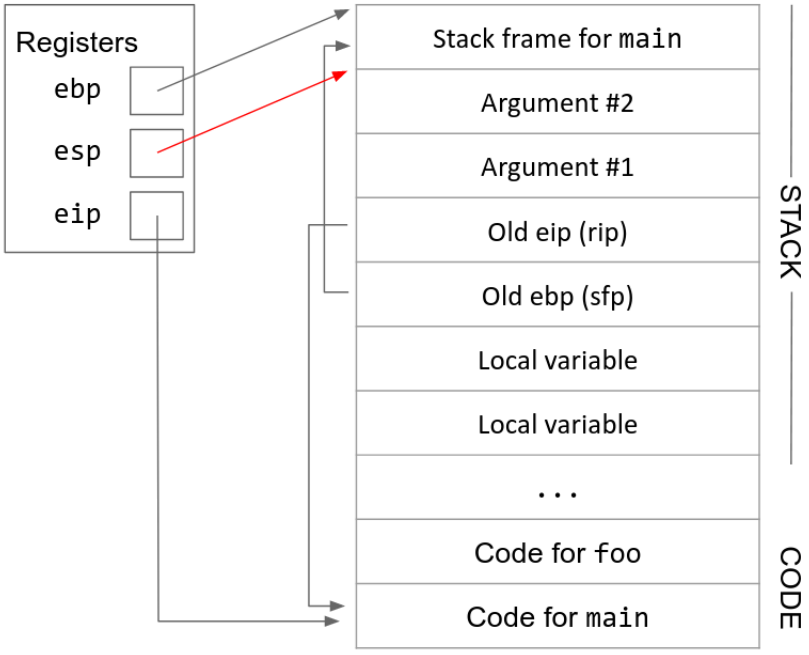
9. Restore the old ebp (sfp). The next value on the stack is the sfp, the old value of ebp before we started executing the function. We pop the sfp off the stack and store it back into the ebp register. This returns ebp to its old value before the function was called.



10. Restore the old eip (rip). The next value on the stack is the rip, the old value of eip before we started executing the function. We pop the rip off the stack and store it back into the eip register. This returns eip to its old value before the function was called.[7]



11. Remove arguments from the stack. Since the function call is over, we don't need to store the arguments anymore. We can remove them by incrementing esp (recall that anything on the stack below esp is undefined).



You might notice that we saved the old values of eip and ebp during the function call, but not the old value of esp. A nice consequence of this function call design is that esp will automatically move to the bottom of the stack as we push values onto the stack and automatically return to its old position as we remove values from the stack. As a result, there is no need to save the old value of esp during the function call.

2.9 x86 function call in assembly

Consider the following C code:

```
int main(void) {
    foo(1, 2);
}

void foo(int a, int b) {
    int bar[4];
}
```

The compiler would turn the `foo` function call into the following assembly instructions:

```
main:
    // Step 1. Push arguments on the stack in reverse order
    push $2
    push $1

    // Steps 2-3. Save old eip (rip) on the stack and change eip
    call foo

    // Execution changes to foo now. After returning from foo:

    // Step 11: Remove arguments from stack
    add $8, %esp

foo:
    // Step 4. Push old ebp (sfp) on the stack
    push %ebp

    // Step 5. Move ebp down to esp
    mov %esp, %ebp

    // Step 6. Move esp down
    sub $16, %esp

    // Step 7. Execute the function (omitted here)

    // Step 8. Move esp
    mov %ebp, %esp

    // Step 9. Restore old ebp (sfp)
    pop %ebp

    // Step 10. Restore old eip (rip)
    pop %eip
```

Note that steps 1-3 happen in the caller function (`main`). Step 3 is changing the eip to point to the callee function (`foo`). Once the eip is changed, program execution is now in `foo`, where steps 4-10 take place. Step 10 is changing the eip to point back to the caller function (`main`). Once the eip is changed back, program execution is now in `main`, where step 11 takes place.

The `call` instruction in steps 2-3 pushes the old eip (rip) onto the stack and then changes eip to point to the instructions for the `foo` function.

In step 6, esp is moved down by 16 bytes. The number 16 is determined by the compiler depending on the function being called. In this case, the compiler decides 16 bytes are required to fit the local variable and any other data needed for the function to execute.

This class uses AT&T x86 syntax, which means in the `mov` instruction, the source is the first argument, and the destination is the second argument. For example, step 5, `mov %esp, %ebp` says to take the value in esp and put it in ebp.[8]

Since function calls are so common, assembly programmers sometimes use shorthand to write function returns. The two instructions in steps 8 and 9 are sometimes abbreviated as the `leave` instruction, and the instruction in step 10 is sometimes abbreviated as the `ret` instruction. This lets x86 programmers simply write "`leave ret`" after each function.

Steps 4-6 are sometimes called the *function prologue*, since they must appear at the start of the assembly code of any C function. Similarly, steps 8-10 are sometimes called the *function epilogue*.

[8]Note that if you are searching for x86 resources online, you may run into Intel syntax, where the source and destination are reversed. Percent signs `%` usually mean you're reading AT&T syntax.

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.

```
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.

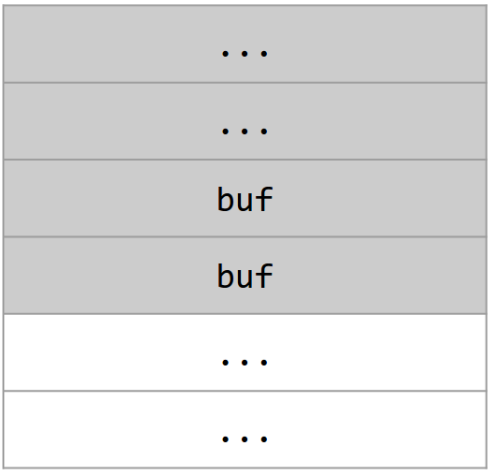


Figure 7: Memory layout of the vulnerable code snippet. The attacker can overwrite the shaded parts of memory.

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:

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

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".

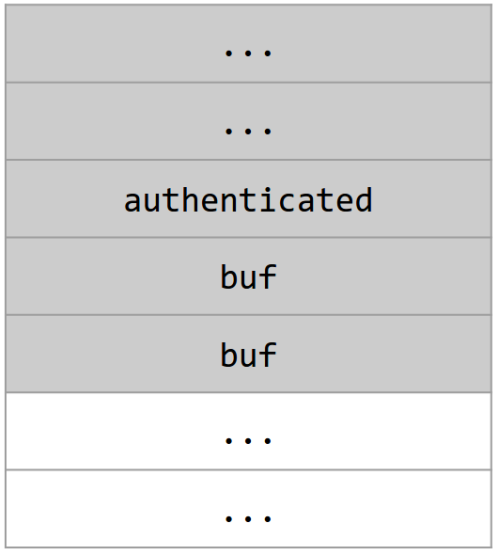


Figure 8: Memory layout of the vulnerable code snippet. The attacker can overwrite the shaded parts of memory.

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:

```
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.

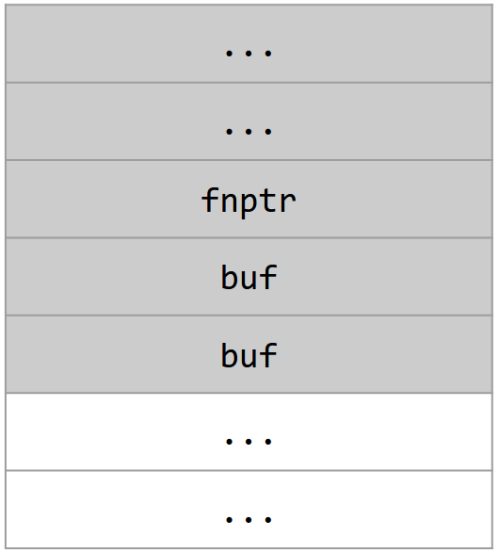


Figure 9: Memory layout of the vulnerable code snippet. The attacker can overwrite the shaded parts of memory.

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*"[9].

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.

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

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:

```
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:

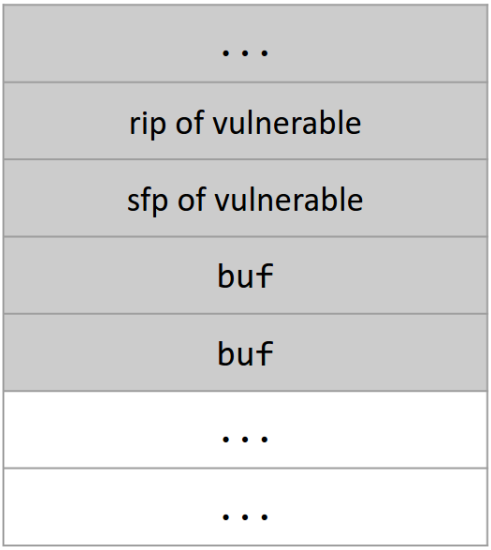


Figure 10: Memory layout of the vulnerable code snippet. The attacker can overwrite the shaded parts of memory.

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.

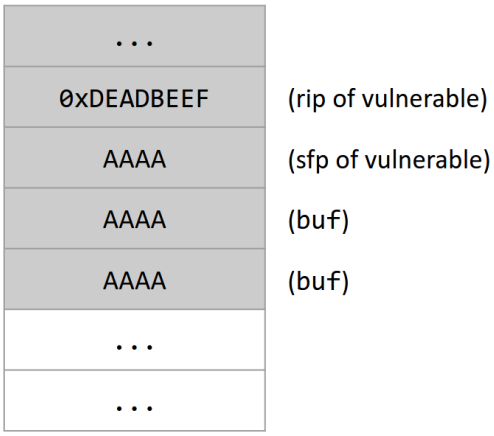


Figure 11: Memory layout after attacker input is entered.

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 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.

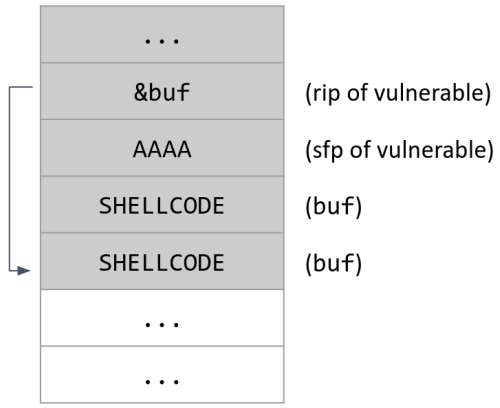


Figure 12: Memory layout after attacker input is entered.

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.

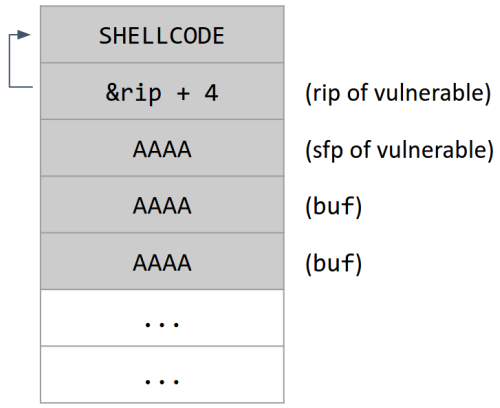


Figure 13: Memory layout after attacker input is entered.

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 look next at another type of vulnerability:

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

Do you see the bug? The last line should be `printf("%s", buf)`. If `buf` contains any `%` characters, `printf()` will look for non-existent arguments, and may crash or core-dump the program trying to chase missing pointers. But things can get much worse than that.

If the attacker can see what is printed, the attacker can mount several attacks:

- The attacker can learn the contents of the function's stack frame. (Supplying the string `"%x:%x"` reveals the first two words of stack memory.)

- The attacker can also learn the contents of any other part of memory, as well. (Supplying the string `"%s"` treats the next word of stack memory as an address, and prints the string found at that address. Supplying the string `"%x:%s"` treats the next word of stack memory as an address, the word after that as an address, and prints what is found at that string. To read the contents of memory starting at a particular address, the attacker can find a nearby place on the stack where that address is stored, and then supply just enough `%x`'s to walk to this place followed by a `%s`. Many clever tricks are possible, and the details are not terribly important for our purposes.) Thus, an attacker can exploit a format string vulnerability to learn passwords, cryptographic keys, or other secrets stored in the victim's address space.
- The attacker can write any value to any address in the victim's memory. (Use `%n` and many tricks; the details are beyond the scope of this writeup.) You might want to ponder how this could be used for malicious code injection.

Let's look at some more examples of format string vulnerabilities:

- `printf("100% done!")` will use the `%d` to print 4 bytes on the stack, 8 bytes above the RIP of `printf`
- `printf("100% stopped!")` will use the `%s` to print the bytes pointed to by the address located 8 bytes above the RIP of `printf` up until the first NULL byte.

The bottom line: *If your program has a format string bug, assume that the attacker can learn all secrets stored in memory, and assume that the attacker 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);
    ...
}
```

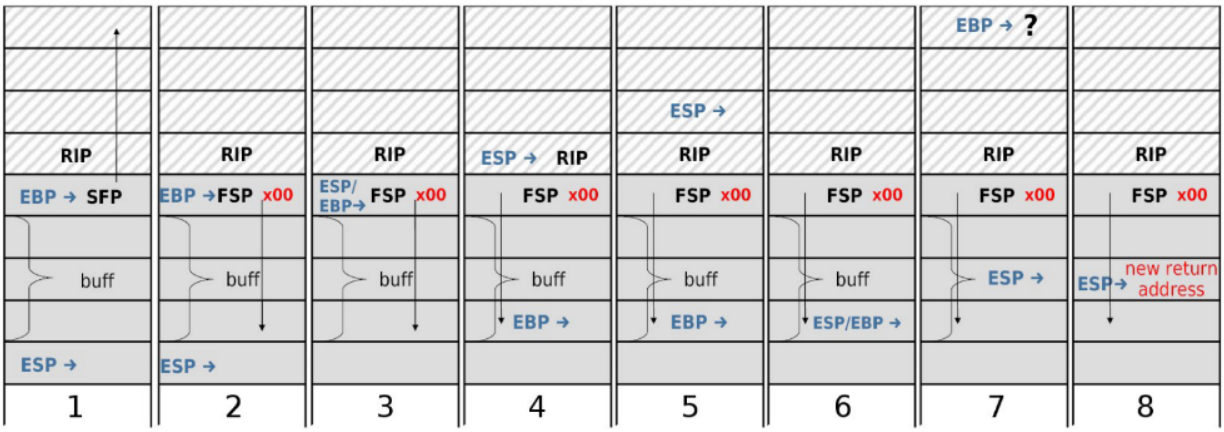
This code seems to avoid buffer overflow problems (indeed, it allocates 5 more bytes than necessary). But, there is a subtle problem: `len+5` can wrap around if `len` is too large. For instance, if `len = 0xFFFFFFFF`, then the value of `len+5` is `4` (on 32-bit platforms). In this case, the code allocates 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).

This following diagram is from Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. It shows 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.

`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 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.

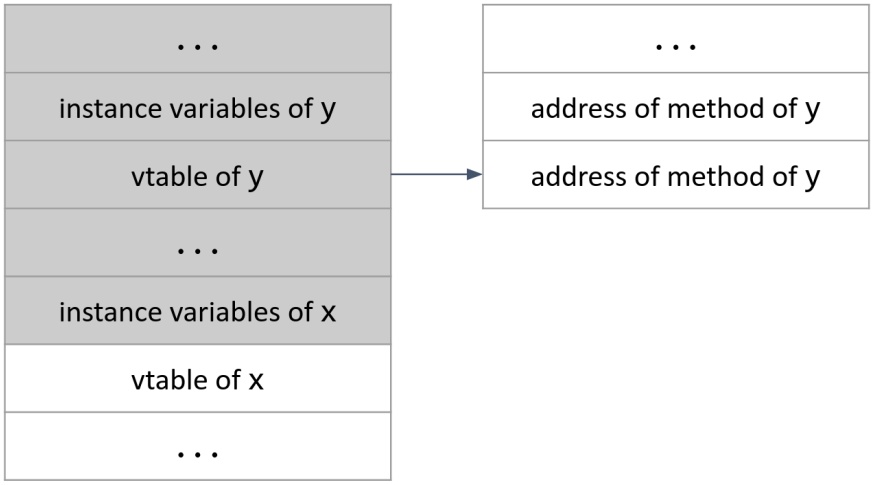


Figure 14: Memory layout of objects on the heap in C++. The attacker can overwrite the shaded parts of memory.

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.

4 Mitigating Memory-Safety Vulnerabilities

4.1 Use a memory-safe language

Some modern languages are designed to be intrinsically memory-safe, no matter what the programmer does. Java, Python, Go, Rust, Swift, and many other programming languages include a combination of compile-time and runtime checks that prevent memory errors from occurring. Using a memory safe language is the *only* way to stop 100% of memory safety vulnerabilities. In an ideal world, everyone would program in memory-safe languages and buffer overflow vulnerabilities would no longer exist. However, because of legacy code and perceived[10] performance concerns, memory-unsafe languages such as C are still prevalent today.

4.2 Writing memory-safe code

One way to ensure memory safety is to carefully reason about memory accesses in your code, by defining pre-conditions and post-conditions for every function you write and using invariants to prove that these conditions are satisfied. Although it is a good skill to have, this process is painstakingly tedious and rarely used in practice, so it is no longer in scope for this class. If you'd like to learn more, see this lecture from David Wagner: video, slides.

Another example of defending against memory safety vulnerabilities is writing memory-safe code through defensive programming and using safe libraries. Defensive programming is very similar to defining pre and post conditions for every written function, wherein you always add checks in your code just in case something could go wrong. For example, you would always check that a pointer is not null before dereferencing it, even if you are sure that the pointer is always going to be valid. However, as mentioned earlier, this relies a lot on programmer discipline and is very tedious to properly implement. As such, a more common method is to use safe libraries, which, in turn, use functions that check bounds so you don't have to. For example, using `fgets` instead of `gets`, `strncpy` or `strlcpy` instead of `strcpy`, and `snprintf` instead of `sprintf`, are all steps towards making your code slightly more safe.

4.3 Building secure software

Yet another way to defend your code is to use tools to analyze and patch insecure code. Utilizing run-time checks that do automatic bound-checking, for example is an excellent way to help your code stay safe. If your check fails, you can direct it towards a controlled crash, ensuring that the attacker does not succeed. Hiring someone to look over your code for memory safety errors, though expensive, can prove to be extremely beneficial as well. You can also probe your own system for vulnerabilities, by subjecting your code to thorough tests. Fuzz testing, or testing with random inputs, testing corner cases, and using tools like Valgrind (to detect memory leaks), are all excellent ways to help test your code. Though

[10] The one real performance advantage C has over a garbage collected language like Go is a far more deterministic behavior for memory allocation. But with languages like Rust, which are safe but not garbage collected, this is no longer an advantage for C.

it is pretty difficult to know whether you have tested your code "enough" to deem it safe, there are several code-coverage tools that can help you out.

4.4 Exploit mitigations

Sometimes you might be forced to program in a memory-unsafe language, and you cannot reason about every memory access in your code. For example, you might be asked to update an existing C codebase that is so large that you cannot go through and reason about every memory access. In these situations, a good strategy is to compile and run code with *code hardening defenses* to make common exploits more difficult.

Code hardening defenses are *mitigations*: they try to make common exploits harder and cause exploits to crash instead of succeeding, but they are not foolproof. The only way to prevent *all* memory safety exploits is to use a memory-safe language. Instead, these mitigations are best thought of as defense-in-depth: they cannot prevent all attacks, but by including many different defenses in your code, you can prevent more attacks. Over the years, there has been a back-and-forth arms race between security researchers developing new defenses and attackers developing new ways to subvert those defenses.

The rest of this section goes into more detail about some commonly-used code hardening defenses, and techniques for subverting those defenses. In many cases, using multiple mitigations produces a synergistic effect: one mitigation on its own can be bypassed, but a combination of multiple mitigations forces an attacker to discover multiple vulnerabilities in the target program.

4.5 Mitigation: Non-executable pages

Many common buffer overflow exploits involve the attacker writing some machine code into memory, and then redirecting the program to execute that injected code. For example, one of the stack smashing attacks in the previous section (`[shellcode] + [4 bytes of garbage] + [address of buf]`) involves the attacker writing machine code into memory and overwriting the rip to cause the program to execute that code.

One way to defend against this category of attacks is to make some portions of memory *non-executable*. What this means is that the computer should not interpret any data in these regions as CPU instructions. You can also think of it as not allowing the eip to ever contain the address of a non-executable part of memory.

Modern systems separate memory into *pages* in order to support virtual memory (see 61C or 162 to learn more). To defend against memory safety exploits, each page of memory is set to either be *writable or executable, but not both*. If the user can write to a page in memory, then that page of memory cannot be interpreted as machine instructions. If the program can execute a page of memory as machine instructions, then the user cannot write to that page.

This defense stops the stack smashing attack in the previous section where the attacker wrote machine code into memory. Because the attacker wrote machine code to a page in memory,

that page cannot be executed as machine instructions, so the attack no longer works.

This defense has several names in practice, including W^X (Write XOR Execute), DEP (Data Execution Prevention), and the NX bit (no-execute bit).

4.6 Subverting non-executable pages: Return into libc

Non-executable pages do not stop an attacker from executing existing code in memory. Most C programs import libraries with thousands or even million lines of instructions. All of these instructions are marked as executable (and non-writable), since the programmer may want to call these functions legitimately.

An attacker can exploit this by overwriting the rip with the address of a C library function. For example, the `execv` function lets the attacker start executing the instructions of some other executable.

Some of these library functions may take arguments. For example, `execv` takes a string with the filename of the program to execute. Recall that in x86, arguments are passed on the stack. This means that an attacker can carefully place the desired arguments to the library function in the right place on the stack, so that when the library function starts to execute, it will look on the stack for arguments and find the malicious argument placed there by the attacker. The argument is not being run as code, so non-executable pages will not stop this attack.

4.7 Subverting non-executable pages: Return-oriented programming

We can take this idea of returning to already-loaded code and extend it further to now execute arbitrary code. Return-oriented programming is a technique that overwrites a chain of return addresses starting at the RIP in order to execute a series of "ROP gadgets" which are equivalent to the desired malicious code. Essentially, we are constructing a custom shellcode using pieces of code that already exist in memory. Instead of executing an existing function, like we did in "Return to libc", with ROP you can execute your own code by simply executing different pieces of different code. For example, imagine we want to add 4 to the value currently in the EDX register as part of a larger program. In loaded memory, we have the following functions:

```
foo:
    ...
    0x4005a1 <foo+33> mov %edx, %eax
    0x4005a3 <foo+35> leave
    0x4005a4 <foo+36> ret
    ...
bar:
    ...
    0x400604 <bar+20> add $0x4, %eax
    0x400608 <bar+24> pop %ebx
    0x40060a <bar+26> leave
```

```
0x40060b <bar+27> ret
```

To emulate the `add $0x4, %edx` instruction, we could move the value in EDX to EAX using the gadget in `foo` and then add 4 to EAX using the gadget in EDX! If we set the first return address to `0x004005a1` and the second return address to `0x00400604`, we produce the desired result. Each time we jump to ROP gadget, we eventually execute the 'ret' instruction and then pop the next return address off the stack, jumping to the next gadget. We just have to keep track that our desired value is now in a different register, and because we execute a `pop %ebx` instruction in 'bar' before we return, we also have to remember that the value in EBX has been updated after executing these gadgets—but these are all behaviors that we can account for using standard compiler techniques. In fact, so-called "ROP compilers" exist to take an existing vulnerable program and a desired execution flow and generate a series of return addresses.

The general strategy for executing ROPs is to write a chain of return addresses at the RIP to achieve the behavior that we want. Each return address should point to a gadget, which is a small set of assembly instructions that already exist in memory and usually end in a `ret` instruction (note that gadgets are not functions, they don't need to start with a prologue or end with an epilogue!). The gadget then executes its instructions and ends with a `ret` instruction, which tells the code to jump to the next address on the stack, thus allowing us to jump to the next gadget!

If the code base is big enough, meaning that the code imports enough libraries, there are usually enough gadgets in memory for you to be able to run any shellcode that you want. In fact, ROP compilers exist on the Internet that will automatically generate an ROP chain for you based on a target binary and desired malicious code! ROP has become so common that non-executable pages are no longer a huge issue for attackers nowadays; while having writable and executable pages makes an attacker's life easier, not a lot of effort has to be put in to subvert this defense mechanism.

4.8 Mitigation: Stack canaries

In the old days, miners would protect themselves against toxic gas buildup in the mine by bringing a caged canary into the mine. These particularly noisy birds are also sensitive to toxic gas. If toxic gas builds up in the mine, the canary dies first, which gives the miners a warning sign that the air is toxic and they should evacuate immediately. The canary in the coal mine is a sacrificial animal: the miners don't expect it to survive, but its death acts as a warning to save the lives of the miners.

We can use this same idea to prevent against buffer overflow attacks. When we call a function, the compiler places a known dummy value, the *stack canary*, on the stack. This canary value is not used by the function at all, so it should stay unchanged throughout the duration of the function. When the function returns, the compiler checks that the canary value has not been changed. If the canary value has changed, then just like the canary in the mine dying, this is evidence that something bad has happened, and the program will crash before any further damage is done.

Like the canary in the coal mine, the stack canary is a sacrifical value: it has no purpose in the function execution and nothing bad happens if it is changed, but the canary changing acts as a warning that someone may be trying to exploit our program. This warning lets us safely crash the program instead of allowing the exploit to succeed.

The stack canary uses the fact that many common stack smashing attacks involve overflowing a local variable to overwrite the saved registers (sfp and rip) directly above. These attacks often write to *consecutive, increasing* addresses in memory, without any gaps. In other words, if the attacker starts writing at a buffer and wants to overwrite the rip, they must overwrite everything in between the buffer and the rip.

The stack canary is placed directly above the local variables and directly below the saved registers (sfp and rip):

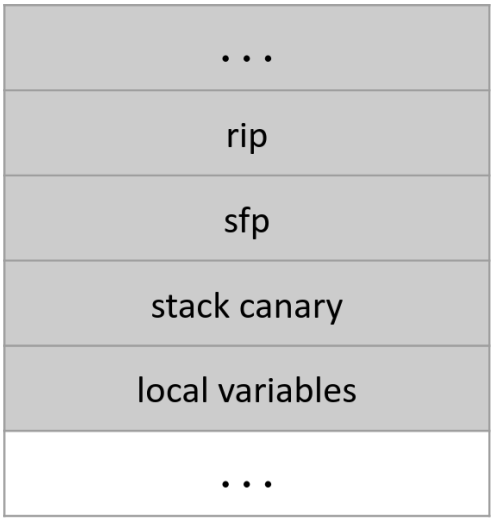


Figure 15: The location of the stack canary on the stack.

Suppose an attacker wants to overflow a local variable to overwrite the rip on the stack, and the vulnerability only allows the attacker to write to consecutive, increasing addresses in memory. Then the attacker must overwrite the stack canary before overwriting the rip, since the rip is located above the buffer in the stack.

Before the function returns and starts executing instructions at the rip, the compiler will check whether the canary value is unchanged. If the attacker has attempted to overwrite the rip, they will have also changed the canary value. The program will conclude that something bad is happening and crash before the attacker can take control. Note that the stack canary detects an attack before the function returns.

The stack canary is a random value generated at *runtime*. The canary is 1 word long, so it is 32 bits long in 32-bit architectures. In Project 1, the canary is 32 completely random bits. However, in reality, stack canaries are usually guaranteed to contain a null byte (usually as the first byte). This lets the canary defend against string-based memory safety exploits, such as vulnerable calls to `strcpy` that read or write values from the stack until they encounter a null byte. The null byte in the canary stops the `strcpy` call before it can copy past the

canary and affect the rip.

The canary value changes each time the program is run. If the canary was the same value each time the program was run, then the attacker could run the program once, write down the canary value, then run the program again and overwrite the canary with the correct value. Within a single run of the program, the canary value is usually the same for each function on the stack.

Modern compilers automatically add stack canary checking when compiling C code. The performance overhead from checking stack canaries is negligible, and they defend against many of the most common exploits, so there is really no reason not to include stack canaries when programming in a memory-unsafe language.

4.9 Subverting stack canaries

Stack canaries make buffer overflow attacks harder for an attacker, but they do not defend programs against all buffer overflow attacks. There are many exploits that the stack canary cannot detect:

- Stack canaries can't defend against attacks outside of the stack. For example, stack canaries do nothing to protect vulnerable heap memory.
- Stack canaries don't stop an attacker from overwriting other local variables. Consider the `authenticated` example from the previous section. An attacker overflowing a buffer to overwrite the `authenticated` variable never actually changes the canary value.
- Some exploits can write to non-consecutive parts of memory. For example, format string vulnerabilities let an attacker write directly to the rip without having to overwrite everything between a local variable and the rip. This lets the attacker write "around" the canary and overwrite the rip without changing the value of the canary.

Additionally, there are several techniques for defeating the stack canary. These usually involve the attacker modifying their exploit to overwrite the canary with its original value. When the program returns, it will see that the canary is unchanged, and the program won't detect the exploit.

Guess the canary: On a 32-bit architecture, the stack canary usually only has 24 bits of entropy (randomness), because one of the four bytes is always a null byte. If the attacker runs the program with an exploit, there is a roughly 1 in 2^{24} chance that the the value the attacker is overwriting the canary with matches the actual canary value. Although the probability of success is low on one try, the attacker can simply run the program 2^{24} times and successfully exploit the program at least once with high probability.

Depending on the setting, it may be easy or hard to run a program and inject an exploit 2^{24} times. If each try takes 1 second, the attacker would need to try for over 100 days before they succeed. If the program is configured to take exponentially longer to run each time the attacker crashes it, the attacker might never be able to try enough times to succeed. However,

if the attacker can try thousands of times per second, then the attacker will probably succeed in just a few hours.

On a 64-bit architecture, the stack canary has 56 bits of randomness, so it is significantly harder to guess the canary value. Even at 1,000 tries per second, an attacker would need over 2 million years on average to guess the canary!

Leak the canary: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. An attacker could use this vulnerability to leak the value of the canary, write it down, and then inject an exploit that overwrites the canary with its leaked value. All of this can happen within a single run of the program, so the canary value doesn't change on program restart.

4.10 Mitigation: Pointer authentication

As we saw earlier, stack canaries help detect if an attacker has modified the rip or sfp pointers by storing a secret value on the stack and checking if the secret value has been modified. As it turns out, we can generalize this idea of using secrets on the stack to detect when an attacker modifies *any* pointer on the stack.

Pointer authentication takes advantage of the fact that in a 64-bit architecture, many bits of the address are unused. A 64-bit address space can support 2^{64} bytes, or 16 exabytes of memory, but we are a long way off from having a machine with this much memory. A modern CPU might support a 16 terabyte address space, which means 44 bits are needed to address all of memory. This still leaves 20 unused bits in every address and pointer.

Consider using these unused bits to store a secret like the stack canary. Any time we need to store an address on the stack, the CPU first replaces the 20 unused bits with some secret value before pushing the value on the stack. When the CPU reads an address off the stack, it will check that the secret value is unchanged. If the secret is unchanged, then the CPU replaces the secret with the original unused bits and uses the address normally. However, if the secret has been changed, this is a warning sign that the attacker has overwritten the address! The CPU notices this and safely crashes the program.

As an example, suppose the rip of a function in a 64-bit system is `0x00000123456789000`. The address space for this architecture is 44 bits, which means the top 20 bits (5 bytes) are always 0 for every address. Instead of pushing this address directly on the stack, the CPU will first replace the 5 unused bytes with a secret value. For example, if the secret value is `0xABCDE`, then the address pushed on the stack is `0xABCDE123456789000`.

This address (with the secret value inserted) is invalid, and dereferencing it will cause the program to crash. When the function returns and the program needs to start executing instructions at the rip, the CPU will read this address from the stack and check that the secret `0xABCDE` is unchanged. If the secret is correct, then the CPU replaces the secret with the original unused bits to make the address valid again. Now the CPU can start executing instructions at the original rip `0x00000123456789000`.

Now, an attacker trying to overwrite the rip would need to know the secret in order to overwrite the rip with the address of some attacker shellcode. If the attacker overwrites the secret with an incorrect value, the CPU will detect this and crash the program.

We can strengthen this defense even further. Since it is the CPU's job to add and check the secret, we can ask the CPU to use a different secret for every pointer stored on the stack. However, we don't want to store all these secrets on the CPU, so we'll use some special math to help us generate secure secrets on the fly.

Consider a special function $f(\mathsf{KEY}, \mathsf{ADDRESS})$. The function f takes a secret key KEY and an address ADDRESS, and outputs a secret value by performing some operation on these two inputs. This function is deterministic, which means if we supply the same key and address twice, it will output the same secret value twice. This function is also secure: an attacker who doesn't know the value of KEY cannot output secret values of their own.[11]

Now, instead of using the same secret value for every address, we can generate a different secret value for each address we store in memory. Every time an address needs to be stored in memory, the CPU runs f with the secret key and the address to generate a unique secret value. Every time an address from memory needs to be dereferenced, the CPU runs f again with the secret key and the address to re-generate the secret value, and checks that the generated value matches the value in memory. The CPU only has to remember the secret key, because all the secret values can be re-generated by running f again with the key and the address.

Using a different secret value for every address makes this defense extremely strong. An attacker who can write to random parts of memory can defeat the stack canary, but cannot easily defeat pointer authentication: they could try to leave the secret value untouched, but because they've changed the address, the old secret value will no longer check out. The CPU will run f on the attacker-generated address, and the output will be different from the old secret value (which was generated by running f on the original address). The attacker also cannot generate the correct secret value for their malicious address, because they don't know what the secret key is. Finally, an attacker could try to leak some addresses and secret values from memory, but knowing the secret values doesn't help the attacker generate a valid secret value for their chosen malicious address.

With pointer authentication enabled, an attacker is never able to overwrite pointers on the stack (including the rip) without generating the corresponding secret for the attacker's malicious address. Without knowing the key, the attacker is forced to guess the correct secret value for their address. For a 20-bit secret, the attacker has a 1 in 2^{20} chance of success.

Another way to subvert pointer authentication is to find a separate vulnerability in the program that allows the attacker to trick the program into creating a validated pointer. The attacker could also try to discover the secret key stored in the CPU, or find a way to subvert the function f used to generate the secret values.

[11] This function is called a MAC (message authentication code), and we will study it in more detail in the cryptography unit.

4.11 Mitigation: Address Space Layout Randomization (ASLR)

Recall the stack smashing attacks from the previous section, where we overwrote the rip with the address of some malicious code in memory. This required knowing the exact address of the start of the malicious code. ASLR is a mitigation that tries to make predicting addresses in memory more difficult.

Although we showed that C memory is traditionally arranged with the code section starting at the lowest address and the stack section starting at the highest address, nothing is stopping us from shifting or rearranging the memory layout. With ASLR, each time the program is run, the beginning of each section of memory is randomly chosen. Also, if the program imports libraries, we can also randomize the starting addresses of each library's source code.

ASLR causes the absolute addresses of variables, saved registers (sfp and rip), and code instructions to be different each time the program is run. This means the attacker can no longer overwrite some part of memory (such as the rip) with a constant address. Instead, the attacker has to guess the address of their malicious instructions. Since ASLR can shuffle all four segments of memory, theoretically, certain attacks can be mitigated. By randomizing the stack, the attacker cannot place shellcode on the stack without knowing the address of the stack. By randomizing the heap, the attacker, similarly, cannot place shellcode on the heap without knowing the address of the heap. Finally, by randomizing the code, the attacker cannot construct an ROP chain or a return-to-libc attack without knowing the address of the code.

There are some constraints to randomizing the sections of memory. For example, segments usually need to start at a page boundary. In other words, the starting address of each section of memory needs to be a multiple of the page size (typically 4096 bytes in a 32-bit architecture).

Modern systems can usually implement ASLR with minimal overhead because they dynamically link libraries at runtime, which requires each segment of memory to be relocatable.

4.12 Subverting ASLR

The two main ways to subvert ASLR are similar to the main ways to subvert the stack canary: guess the address, or leak the address.

Guess the address: Because of the constraints on address randomization, a 32-bit system will sometimes only have around 16 bits of entropy for address randomization. In other words, the attacker can guess the correct address with a 1 in 2^{16} probability, or the attacker can try the exploit 2^{16} times and expect to succeed at least once. This is less of a problem on 64-bit systems, which have more entropy available for address randomization.

Like guessing the stack canary, the feasibility of guessing addresses in ASLR depends on the attack setting. For example, if each try takes 1 second, then the attacker can make 2^{16} attempts in less than a day. However, if each try after a crash takes exponentially longer, 2^{16} attempts may become infeasible.

Leak the address: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. The stack often stores absolute addresses, such as pointers and saved registers (sfp and rip). If the attacker can leak an absolute address, they may be able to determine the absolute address of other parts of memory relative to the absolute address they leaked.

Note that ASLR randomizes absolute addresses by changing the start of sections of memory, but it does not randomize the *relative* addresses of variables. For example, even if ASLR is enabled, the rip will still be 4 bytes above the sfp in a function stack frame. This means that an attacker who leaks the absolute address of the sfp could deduce the address of the rip (and possibly other values on the stack).

4.13 Combining Mitigations

We can use multiple mitigations together to force the attacker to find multiple vulnerabilities to exploit the program; this is a process known as *synergistic protection*, where one mitigation helps strengthen another mitigation. For example, combining ASLR and non-executable pages results in an attacker not being able to write their own shellcode, because of non-executable pages, and not being able to use existing code in memory, because they don't know the addresses of that code (ASLR). Thus, to defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities. First, they need to find a way to leak memory and reveal the address location (to defeat ASLR). Next, they need to find a way to write to memory and write an ROP chain (to defeat non-executable pages).