Another Linux Kernel Bug Surfaces, Allowing Root Access

Author:
Tara Seals

September 28, 2018
/ 2:11 pm

October 7, 2019

Kernel privilege escalation bug actively exploited in Android devices

Bradley Barth

Low Level Software Security

Computer Security and Privacy (CS642)

Earlence Fernandes

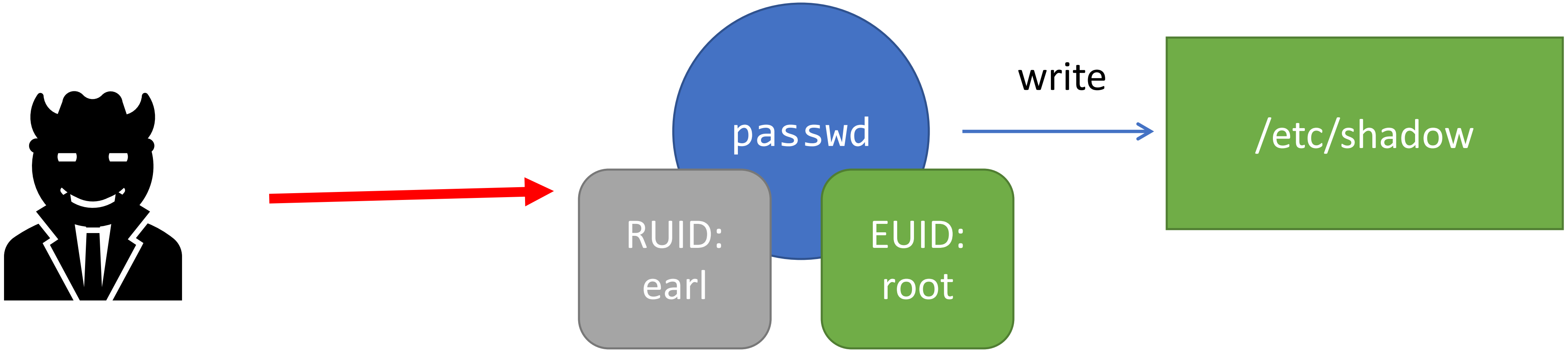
earlence@cs.wisc.edu

* Slides borrowed from Chatterjee, Davidson, Ristenpart

Processes are the front line of system security

- Control a process and you get the privileges of its UID
- So how do you control a process?
  - Send specially formed input to process

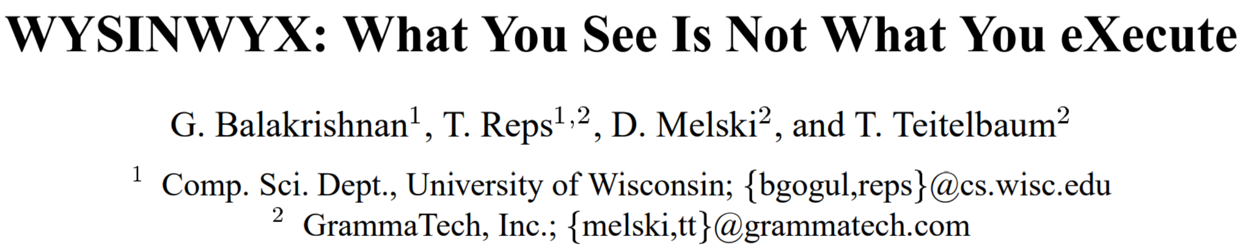
```
-rwsr-xr-x root root … passwd*
```



Roadmap

- Today
  - Enough x86 to understand (some) process vulnerabilities
    - ISA
    - Process memory layout, call stack
  - Buffer overflow attack
    - How such attacks occur

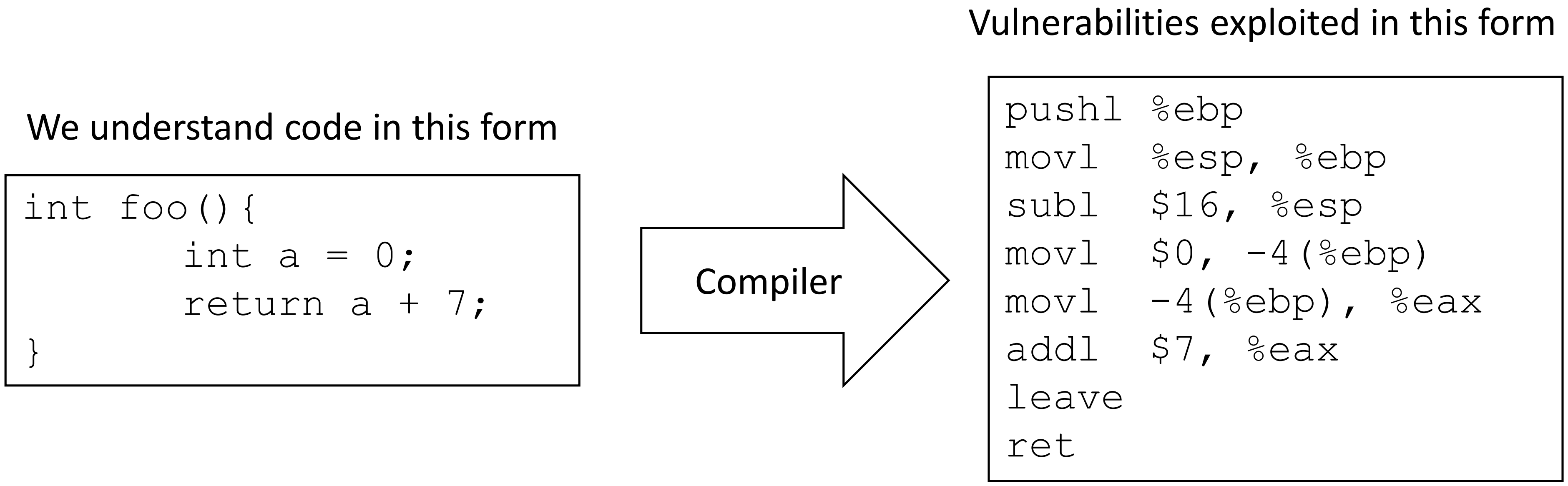
Why do we need to look at assembly?



WYSINWYX: What You See Is Not What You eXecute

G. Balakrishnan[1], T. Reps[1,2], D. Melski[2], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,tt}@grammatech.com



We understand code in this form

```
int foo(){
        int a = 0;
        return a + 7;
}
```

Vulnerabilities exploited in this form

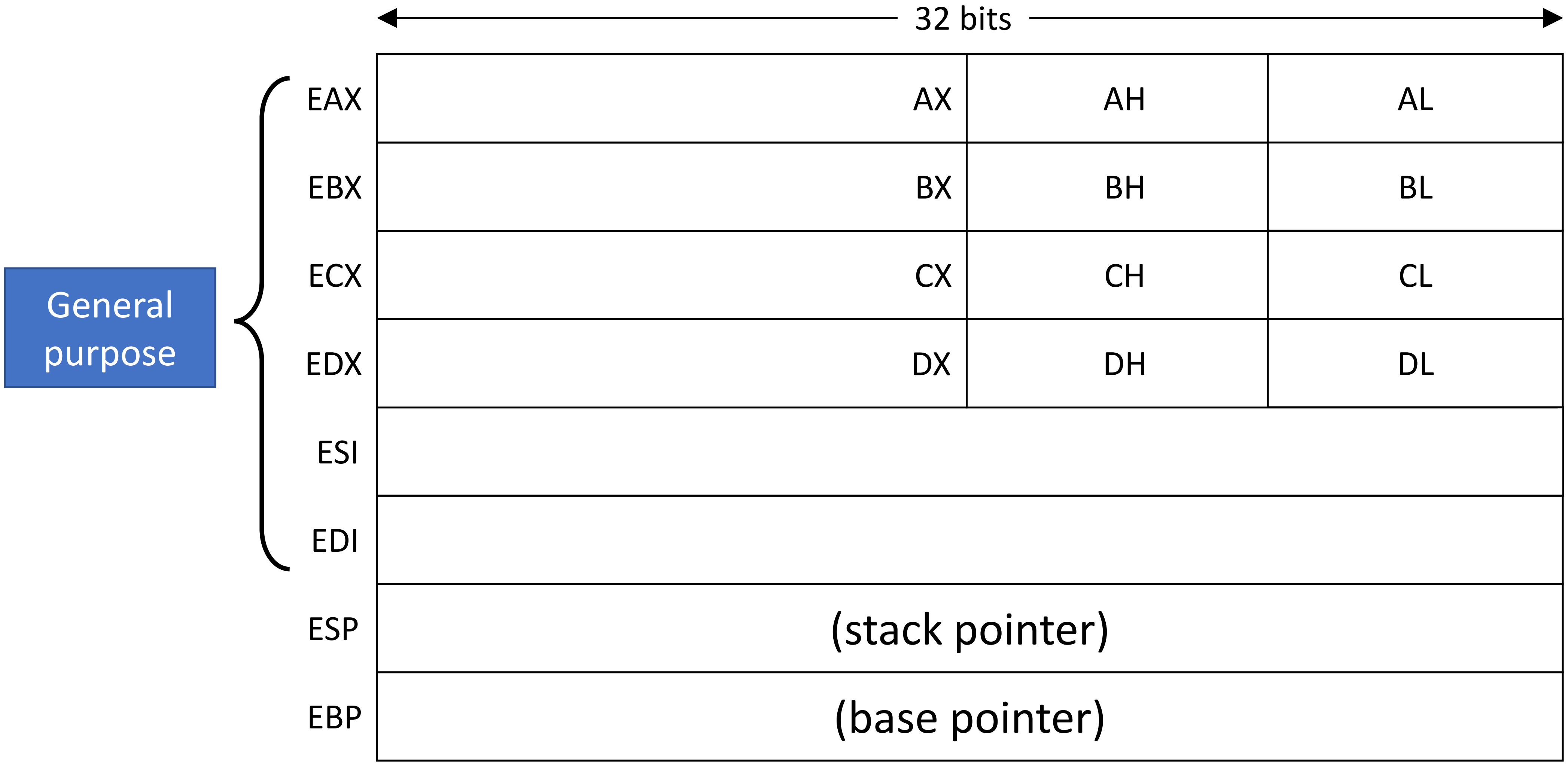
```
pushl %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  $0, -4(%ebp)
movl  -4(%ebp), %eax
addl  $7, %eax
leave
ret
```

X86: The De Facto Standard

- Extremely popular for desktop computers
- Alternatives
  - ARM: popular on mobile
  - MIPS: very simple
  - Itanium: ahead of its time
- CISC (complex instruction set computing)
  - Over 100 distinct opcodes in the set
- Register poor
  - Only 8 registers of 32-bits, only 6 are general-purpose
- Variable-length instructions
- Built of many backwards-compatible revisions
  - Many security problems preventable… in hindsight

Let’s Dive in To X86!

Registers

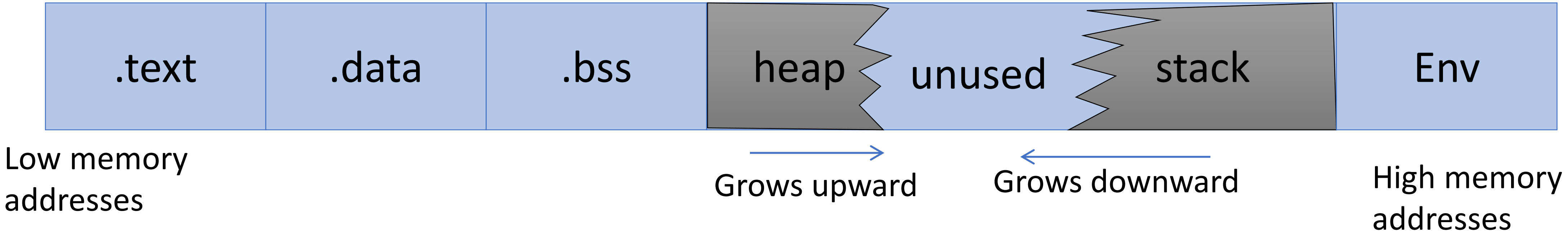


32 bits

General purpose

EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI			
EDI			
ESP	(stack pointer)		
EBP	(base pointer)		

Process memory layout



.text

- Machine code of executable

.data

- Global initialized variables

.bss

- Below Stack Section
  global uninitialized variables

heap

- Dynamic variables

stack

- Local variables
- Function call data

Env

- Environment variables
- Program arguments

Reminder: These are conventions

- Dictated by compiler
- Only instruction support by processor
  - Almost no structural notion of memory safety
    - Use of uninitialized memory
    - Use of freed memory
    - Memory leaks
- So how are they actually implemented?

Instruction Syntax

Examples:

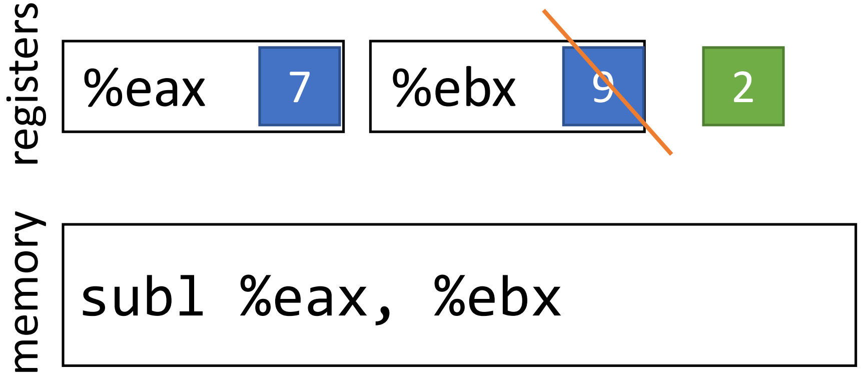
```
subl $16, %ebx

movl (%eax), %ebx
```

```
opcode src, dst
```

- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )

Register Instructions: `sub`

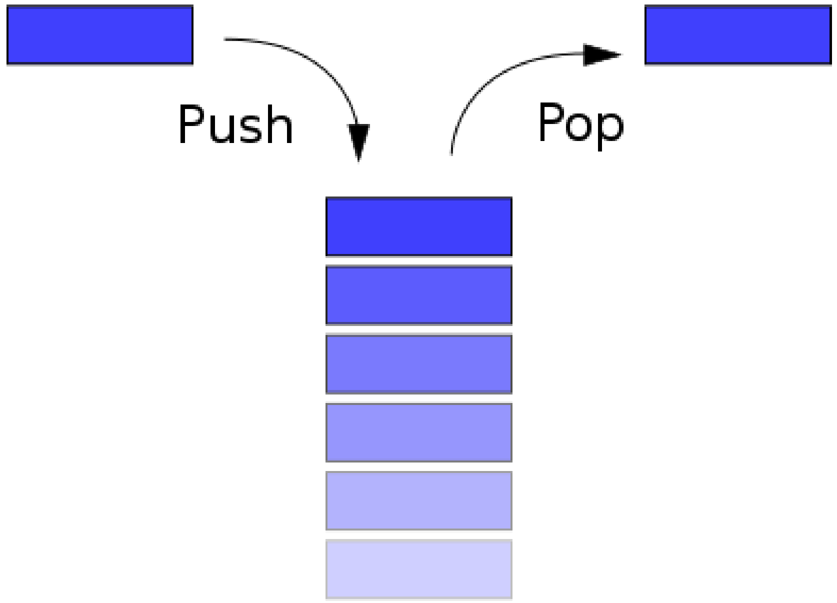


```
subl %eax, %ebx
```

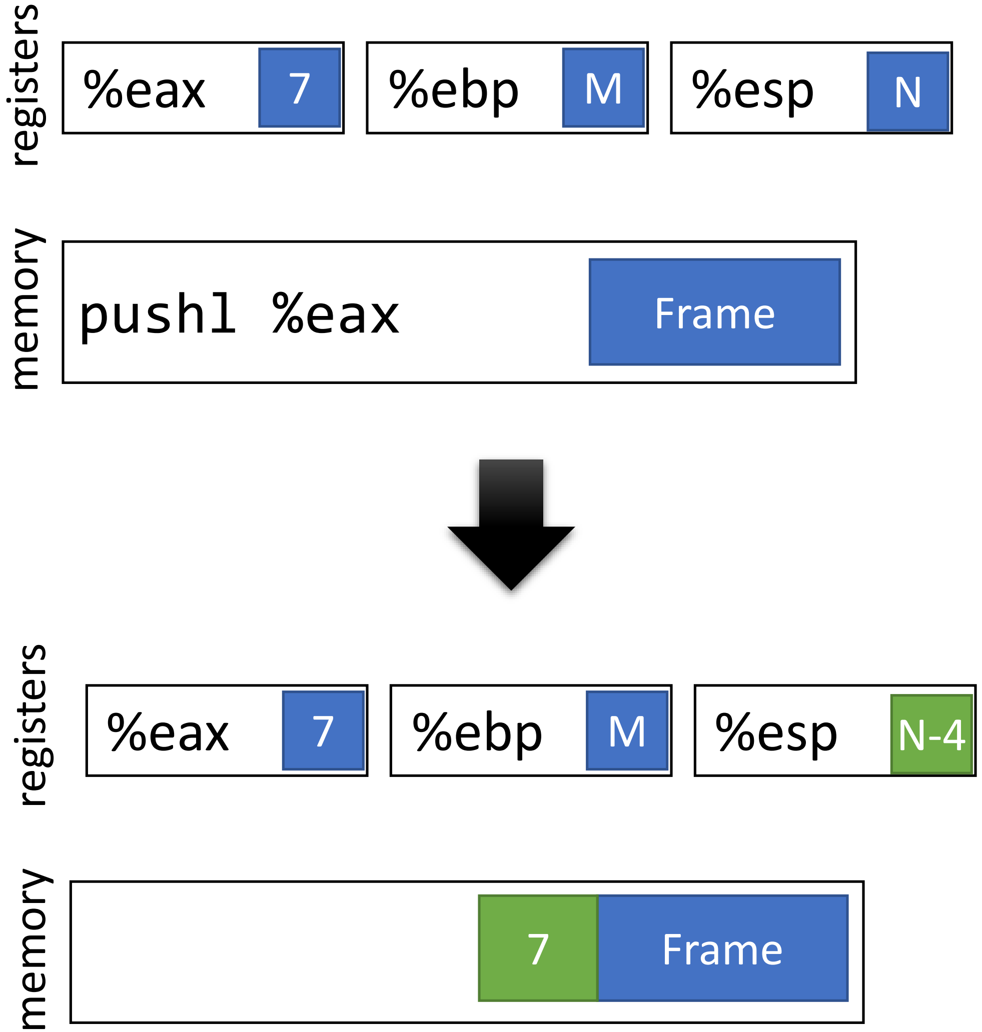
- Subtract from a register value

The Stack

- Local storage
  - Good place to keep data that doesn’t fit into registers
- Grows from high addresses towards low addresses



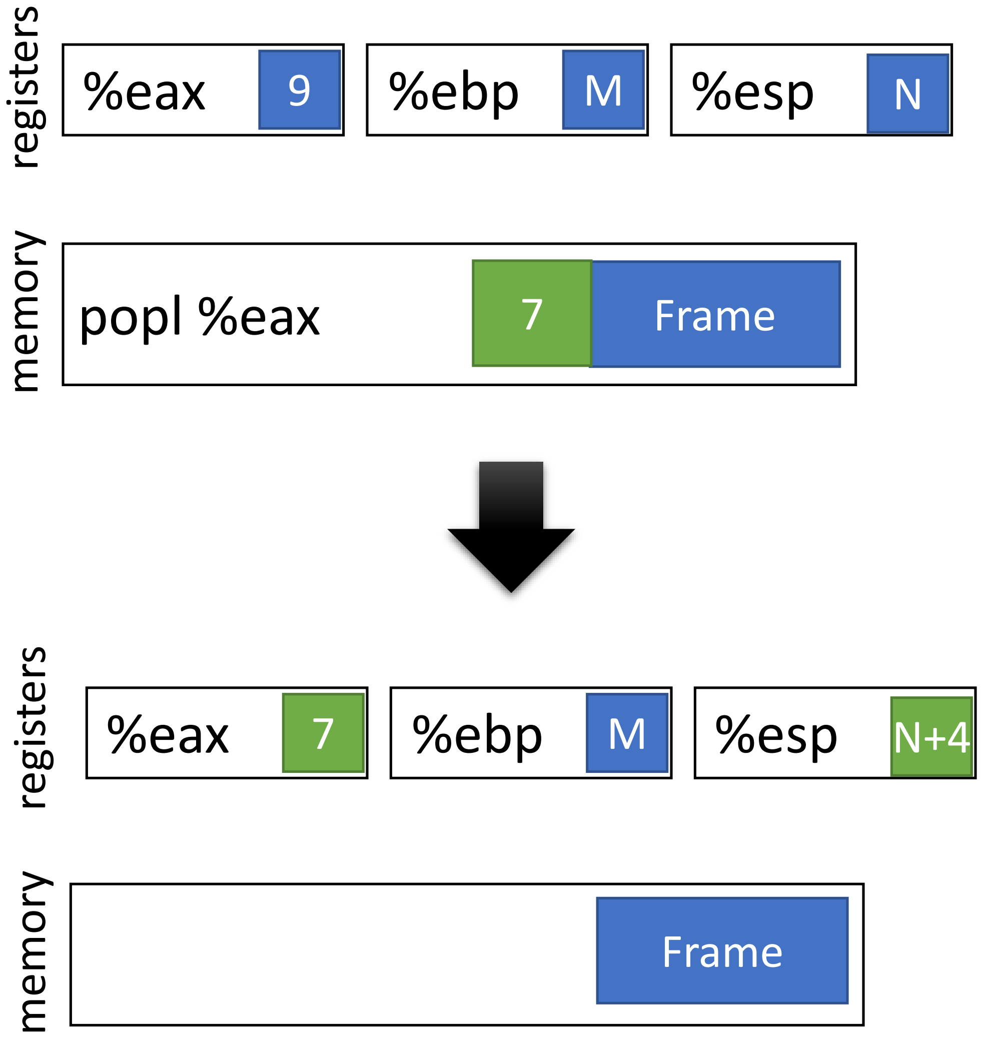
Frame Instructions: **push**



- Put a value on the stack
  - Pull from register
  - Value goes to %esp
  - Subtract from %esp
- Example:

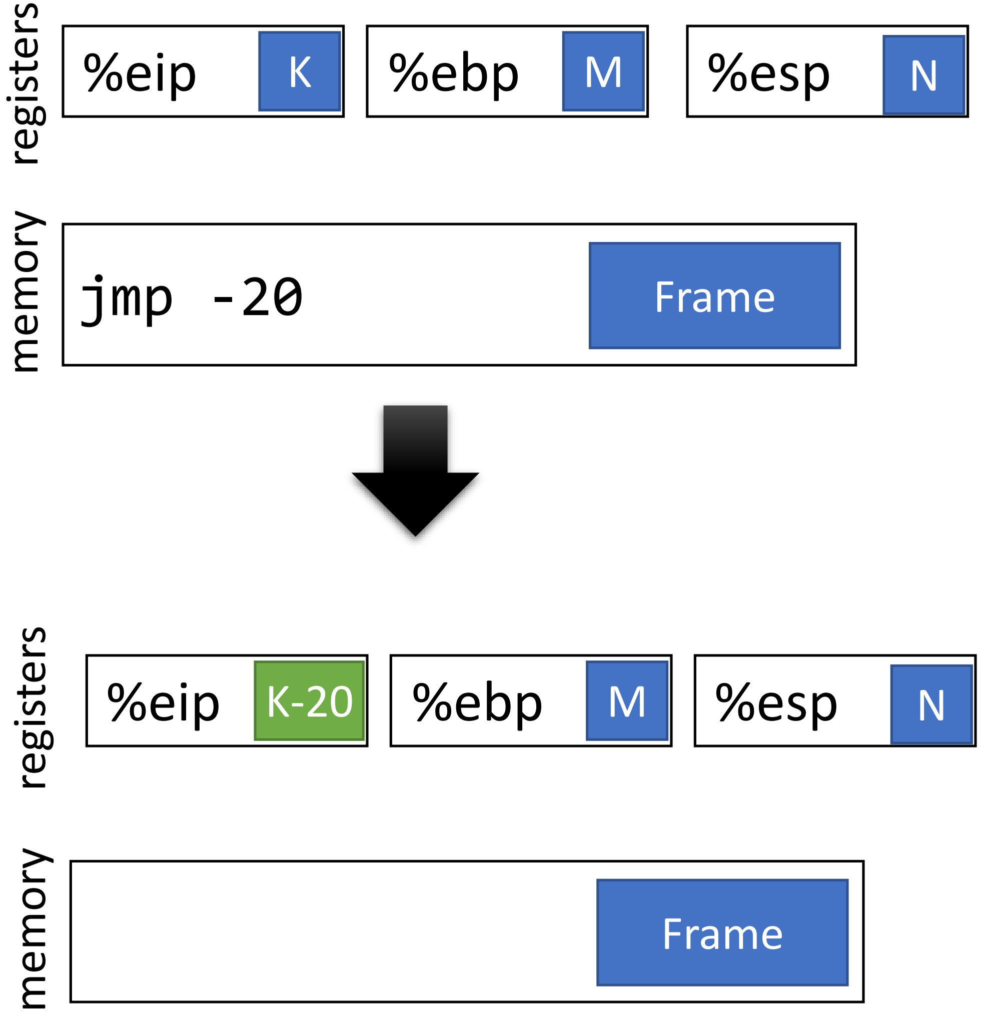
`pushl %eax`

Frame Instructions: **pop**



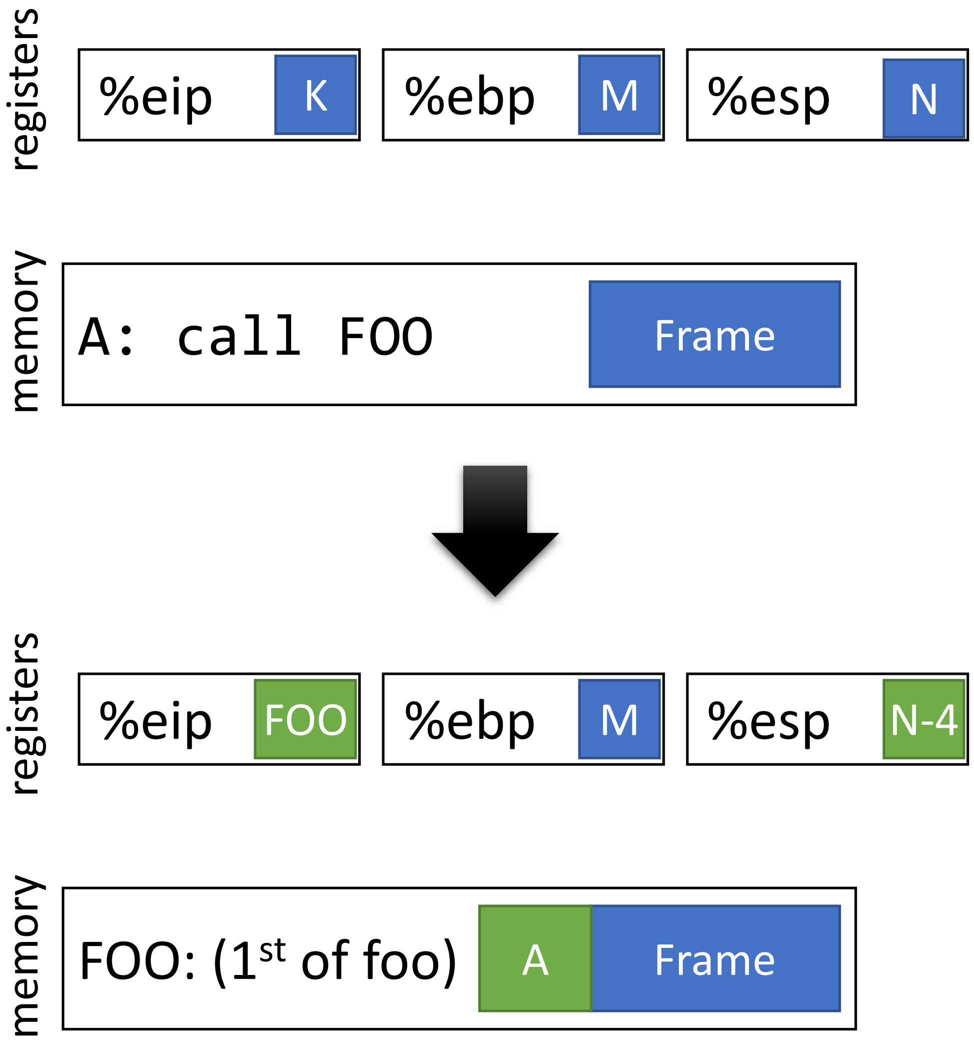
- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp

Control flow instructions: `jmp`



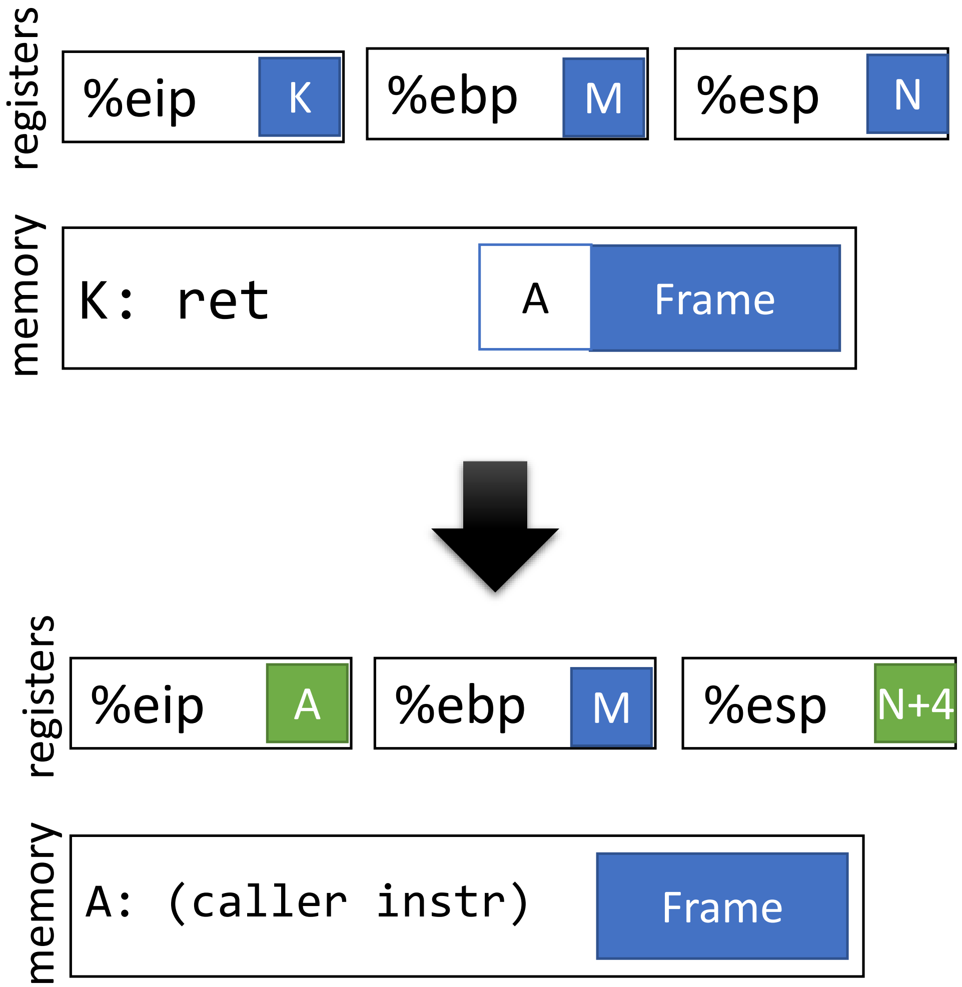
- `%eip` points to the currently executing instruction (in the <u>text section</u>)
- Has unconditional and conditional forms
- Uses relative addressing

Control flow instructions: `call`



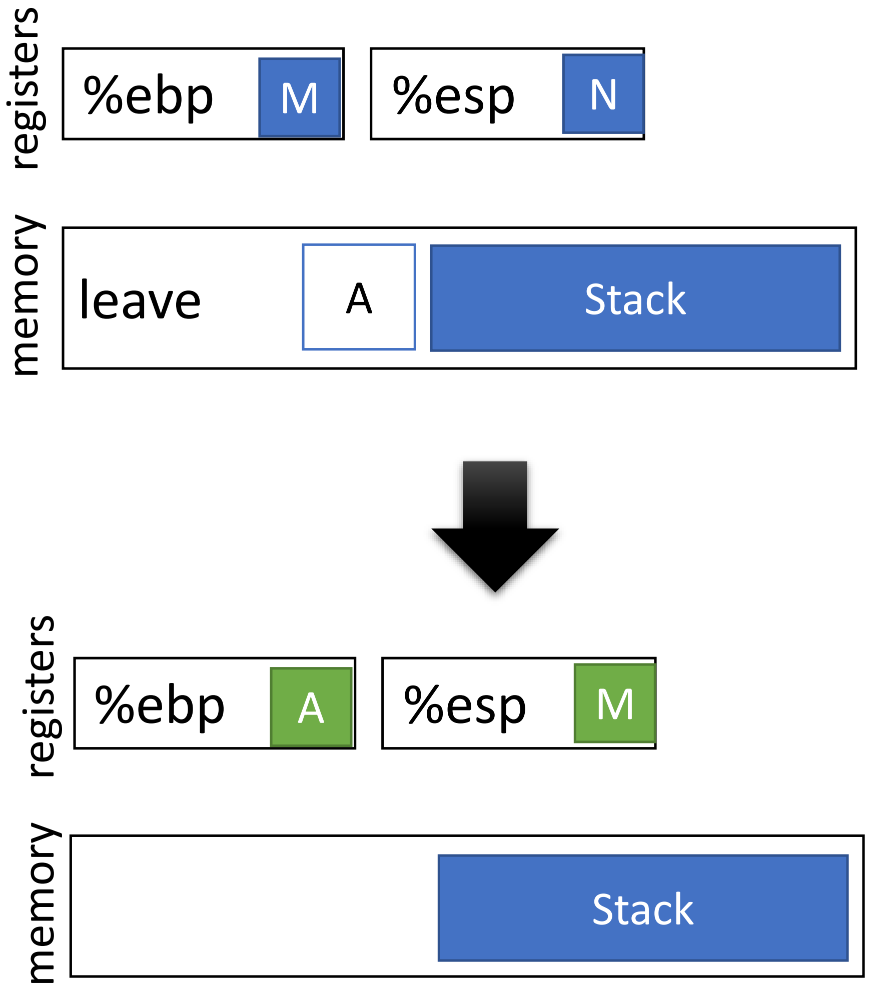
- Saves the current instruction pointer to the stack
- Jumps to the argument value

Control flow instructions: **`ret`**



- Pops the stack into the instruction pointer

Stack instructions: `leave` (and `enter`)

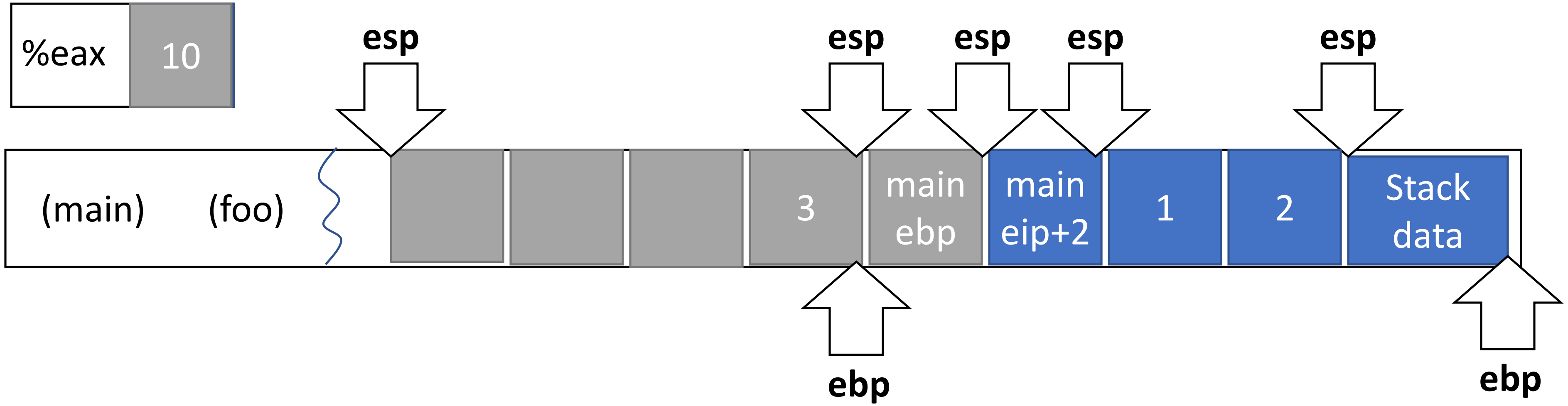


- Equivalent to

```
movl %ebp, %esp
popl %ebp
```

- copy EBP to ESP and then restore the old EBP from the stack

Implementing a function call



```
main:
        …
        subl    $8, %esp
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    foo
        addl    $8, %esp
        …
```

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $3, -4(%ebp)
        movl    8(%ebp), %eax
        addl    $9, %eax
        leave
        ret
```

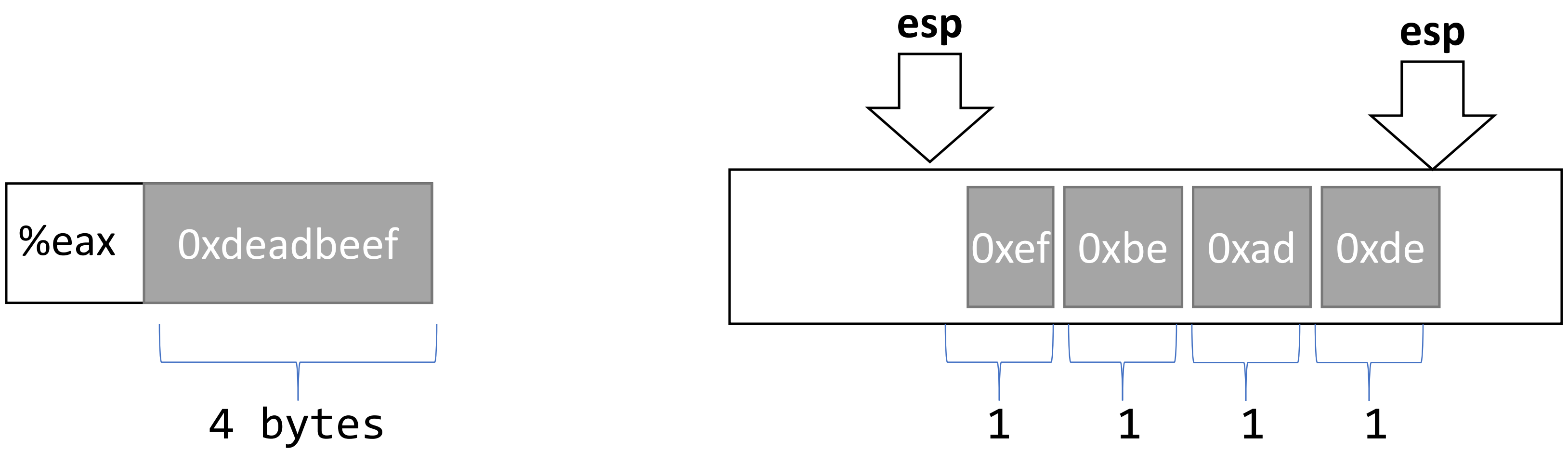
Function Calls: High level points

- Locals are organized into stack frames
  - Callees exist at lower address than the caller
- On call:
  - Save `%eip` so you can restore control
  - Save `%ebp` so you can restore data
- Implementation details are largely by convention
  - Somewhat codified by hardware

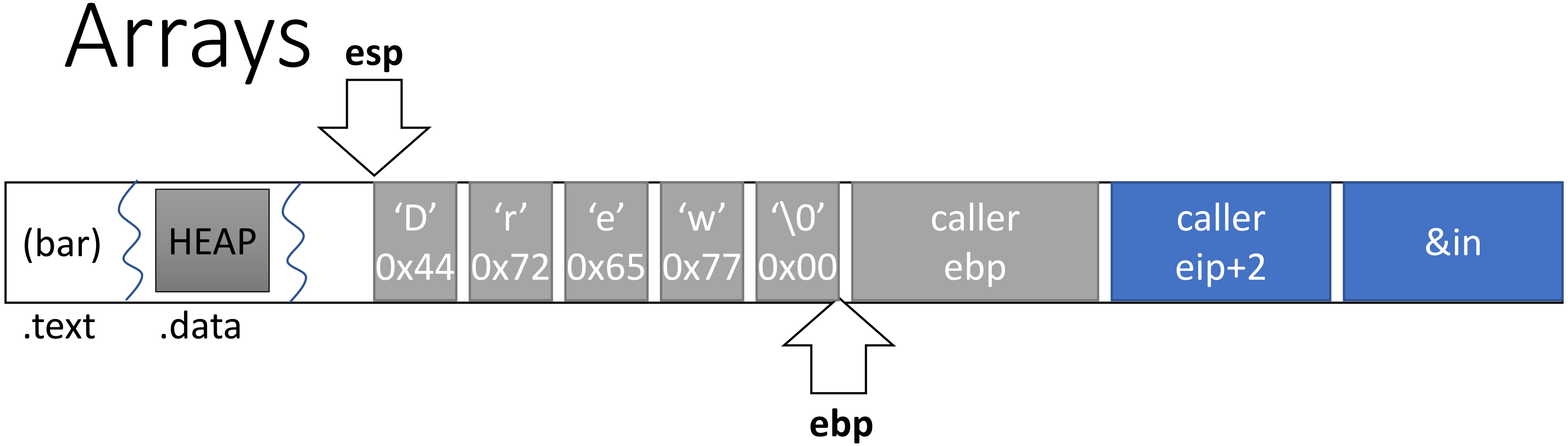
Data types / Endianness

- x86 is a little-endian architecture

```
pushl %eax
```



Arrays



```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
bar:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $5, %esp
  movl   8(%ebp), %eax
  movl   %eax, 4(%esp)
  leal   -5(%ebp), %eax
  movl   %eax, (%esp)
  call   strcpy
  leave
  ret
```

Tools: GCC

```
gcc –O0 –S program.c –o program.S –m32
```

Generate Assembly Code

```
gcc –O0 –g program.c –o program –m32
```

Generate Debugging Information

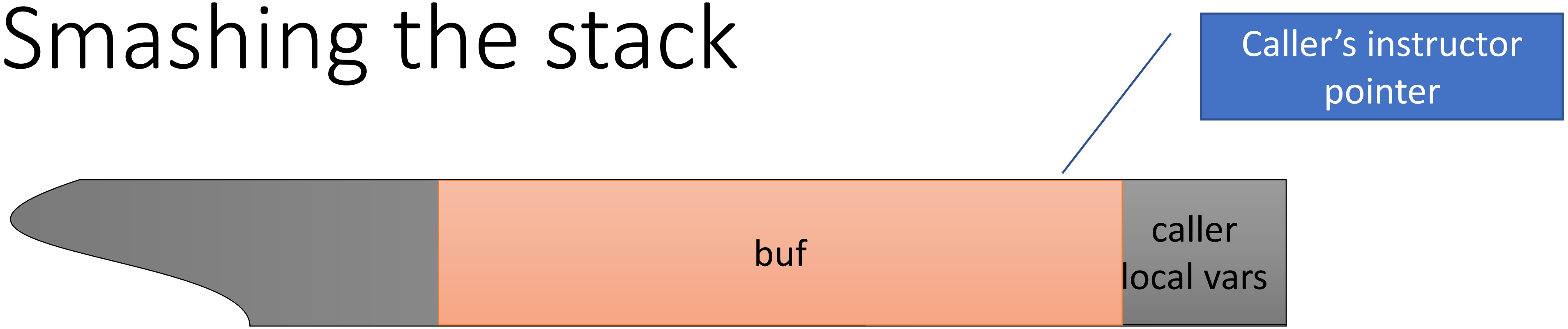
Tools: GDB

```
gdb program
(gdb) run
(gdb) list   /* show the high-level code */
(gdb) decompile foo
(gdb) disas foo  /* show assembly of foo */
(gdb) disas main
(gdb) quit
```

x86 Summary

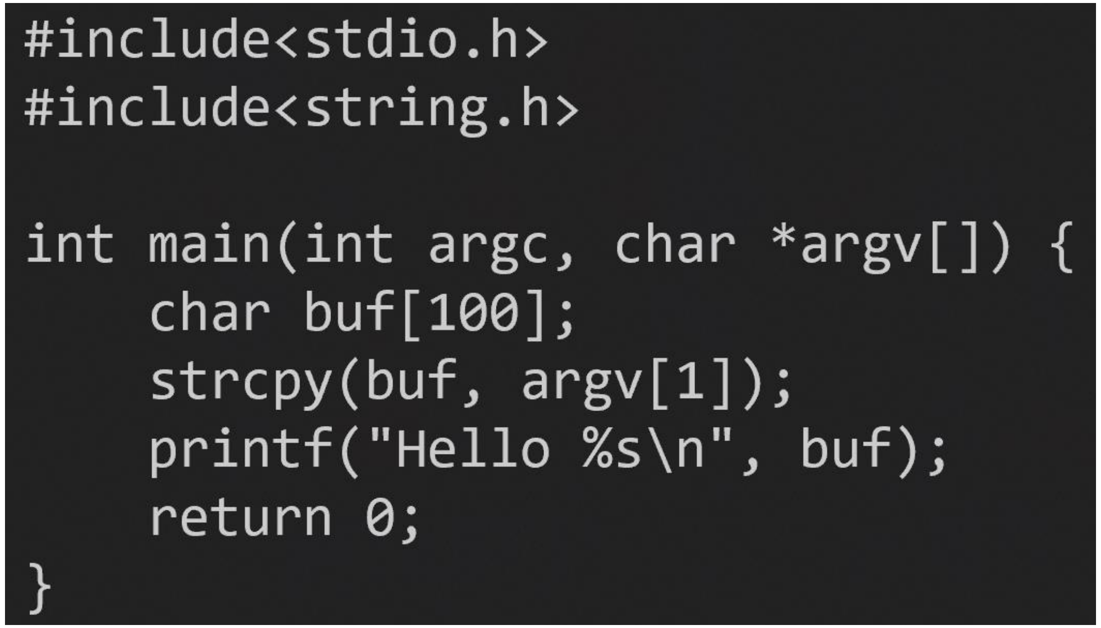
- Basics of x86
    - Process layout
    - ISA details
    - Most of the instructions that you’ll need
- Introduced the concept of a buffer overflow
- Some tools to play around with x86 assembly

Smashing the stack



Low memory addresses

High memory addresses



```
#include<stdio.h>
#include<string.h>

int main(int argc, char *argv[]) {
    char buf[100];
    strcpy(buf, argv[1]);
    printf("Hello %s\n", buf);
    return 0;
}
```

If `argv[1]` has more than 100 bytes…