Host Vulnerabilities

UT LAW379M

SPRING 2021

LECTURE NOTES

Brief Overview to Execution

Today: ***very brief*** overview of ***Control Flow Hijacking***

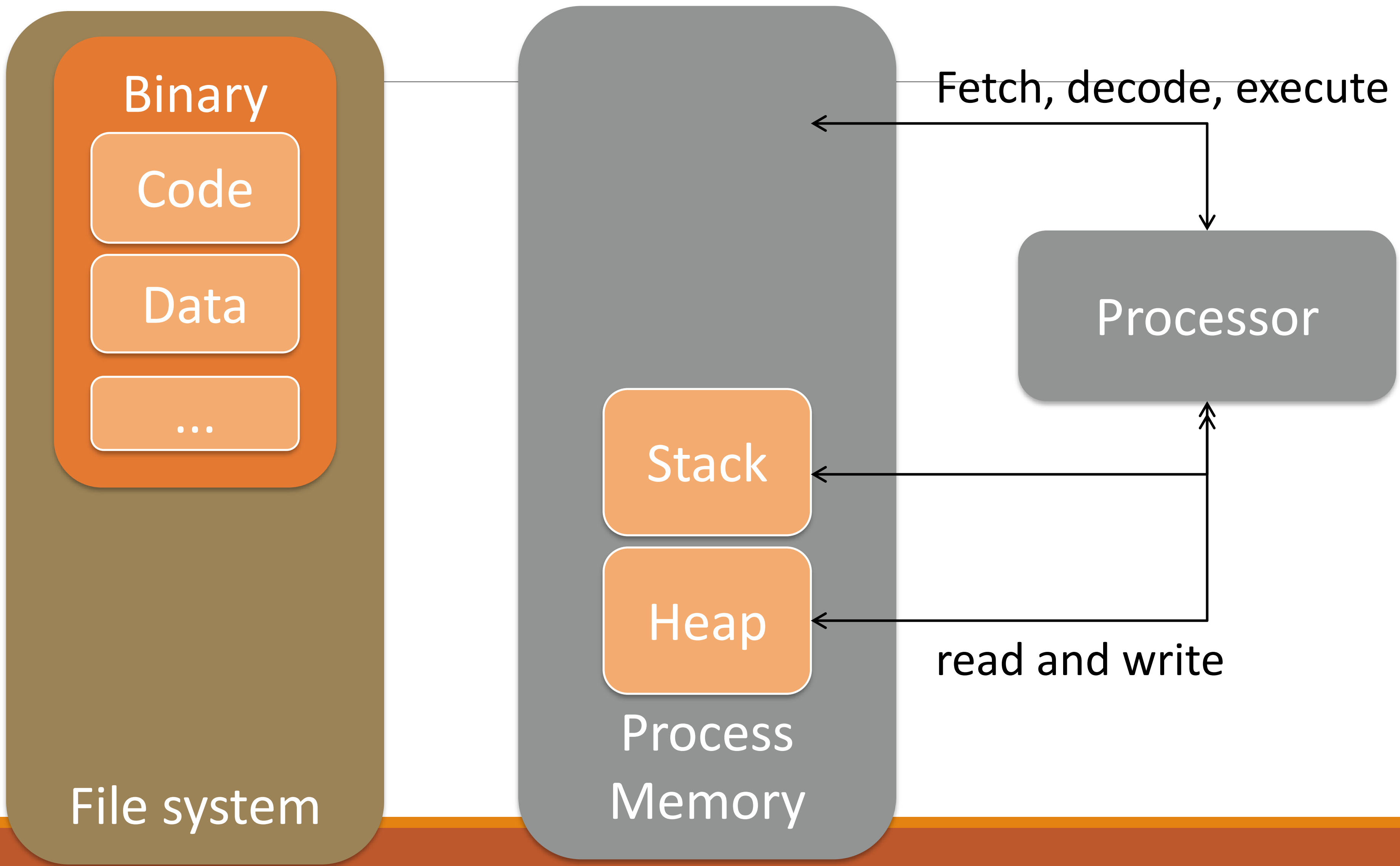
- There are other types of vulnerabilities (e.g misconfigured)
- Control Flow Hijacking is probably the hardest to grasp

Critical Concepts:

- The "normal" flow of control for authorized instructions
- Inputs that change the flow to unauthorized instructions

ATTRIBUTION: Derived from slides by Dave Brumley, CMU

Basic Execution



Seth’s Notes

Stack

- For temporary static variables
- Function call/return data
- Linear
- Generally, tightly managed

Heap

- Global variables and dynamic variables
- Hierarchical, “free floating”
- Fragmented, not tightly managed

Seth’s Notes

Assembly “function calls” don’t really exist

- Rather, jump to new location (“function”)
- Save context of old location
- Load context for new location
- Include information for “returning”

Seth’s Notes

There are multiple ways to do this

“Calling Conventions”

Caller Cleanup – caller cleans stack

Callee Cleanup – called function cleans stack

Other convention variations:

- Order that function data is loaded onto stack
- Whether some data is put into registers instead

Seth's Notes

Visualizing caller v callee cleanup

stdcall (callee)

```
    push arg1
    push arg2
    push arg3
    call proc

proc:
    pop  r1   ; the return address
    pop  r2
    pop  r2
    pop  r2
    push r1
    ret
```

cdecl (caller)

```
    push arg1
    push arg2
    push arg3
    call proc
    pop  r2
    pop  r2
    pop  r2

proc:
    ret
```

EBP and ESP

EBP

- Stack Base Pointer
- Where the stack was when the routine started

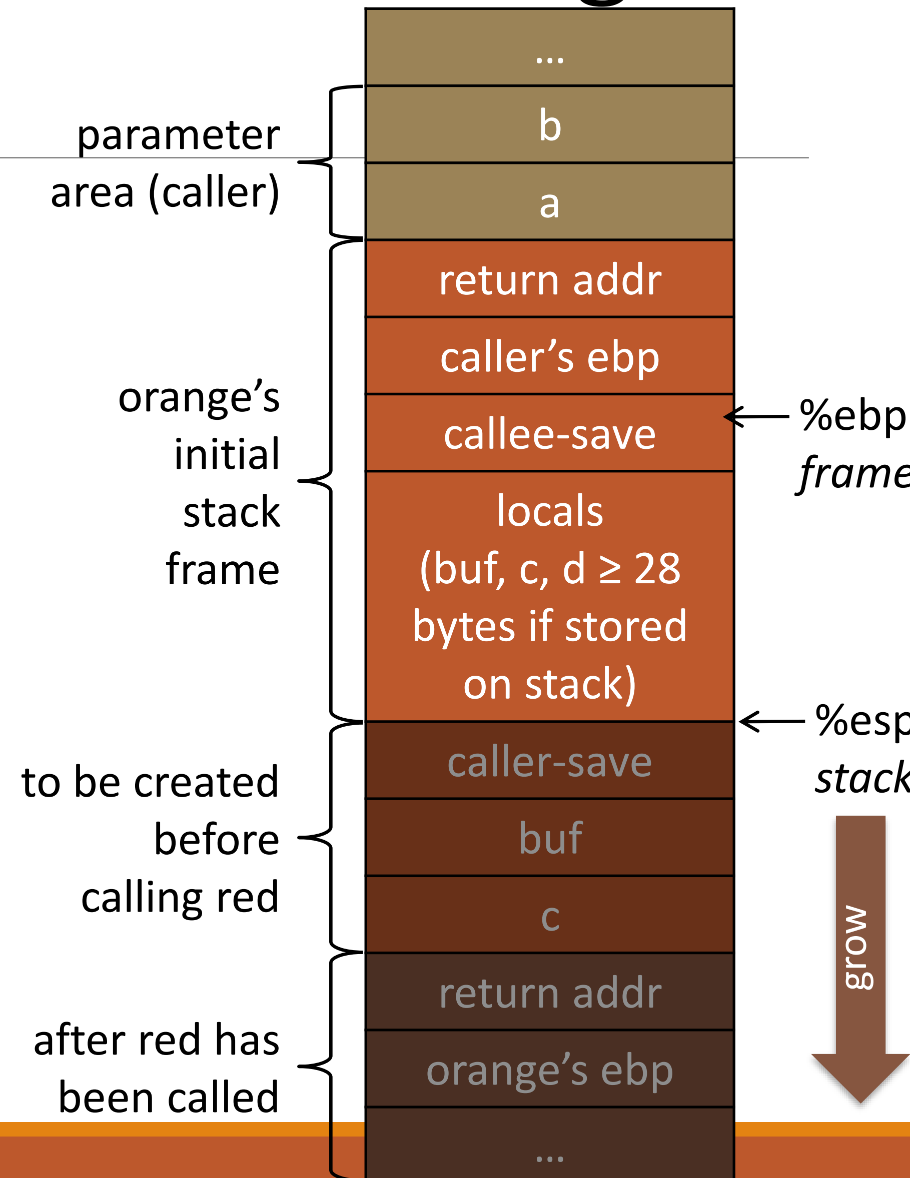
ESP

- Stack Pointer
- Top of the current stack

EBP is a previous function’s saved ESP

cdecl – 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;
}
```



What are Buffer Overflows?

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

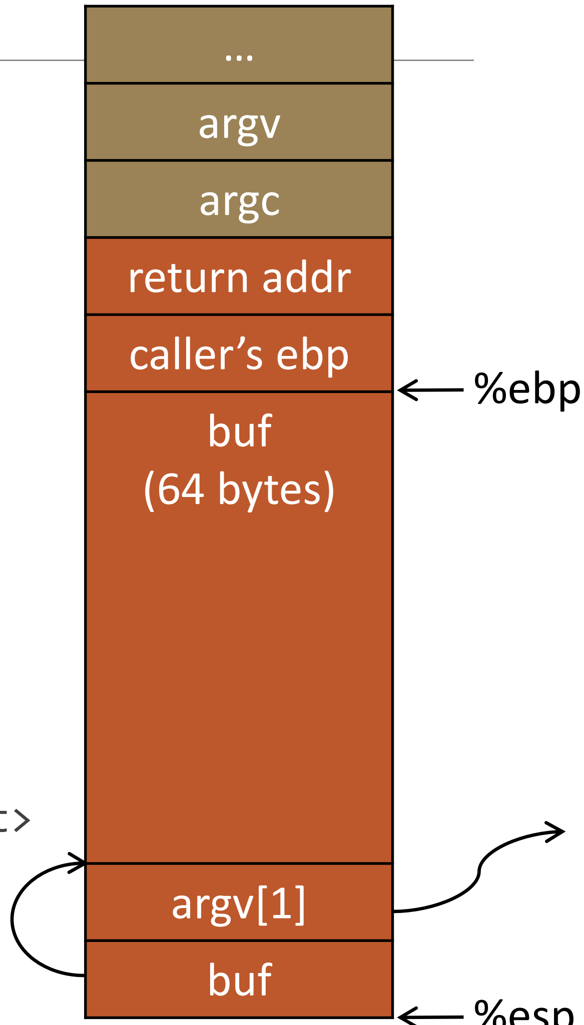
C does not check that writes are in-bound

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]);
}

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
```

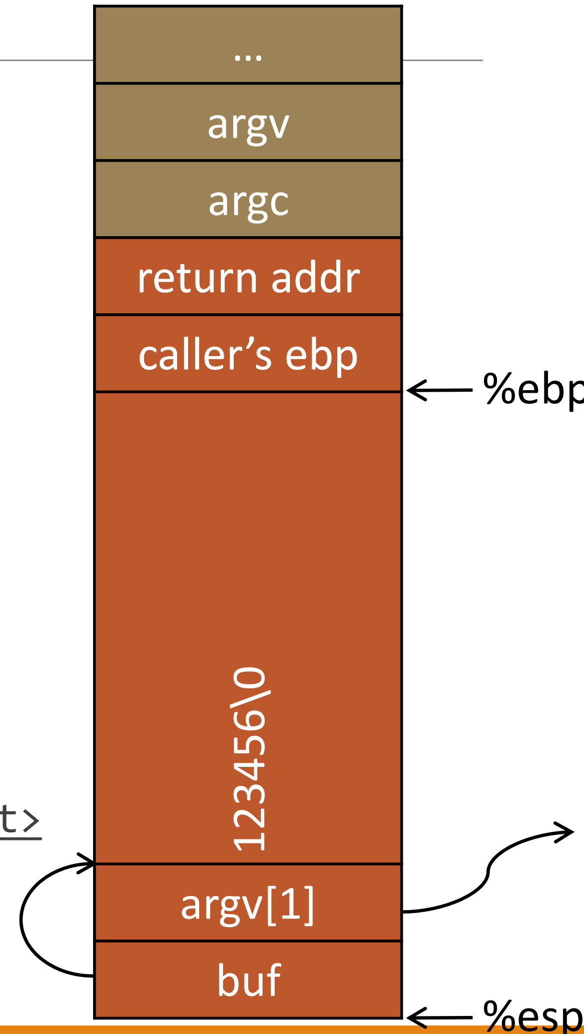


11

"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
```

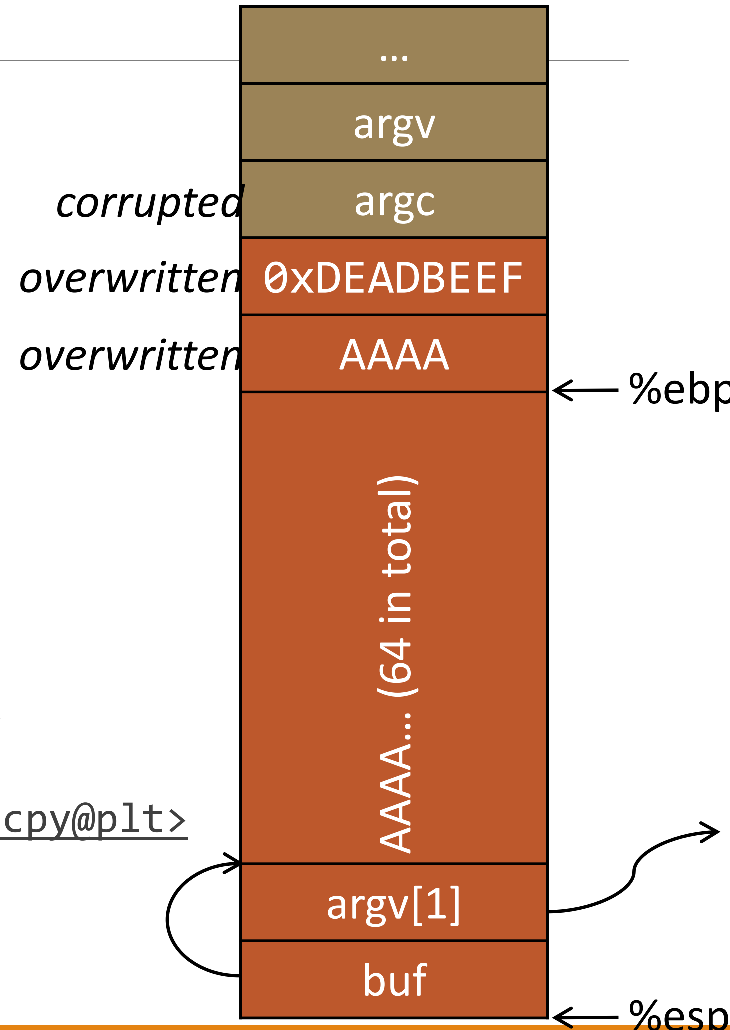


12

“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
```

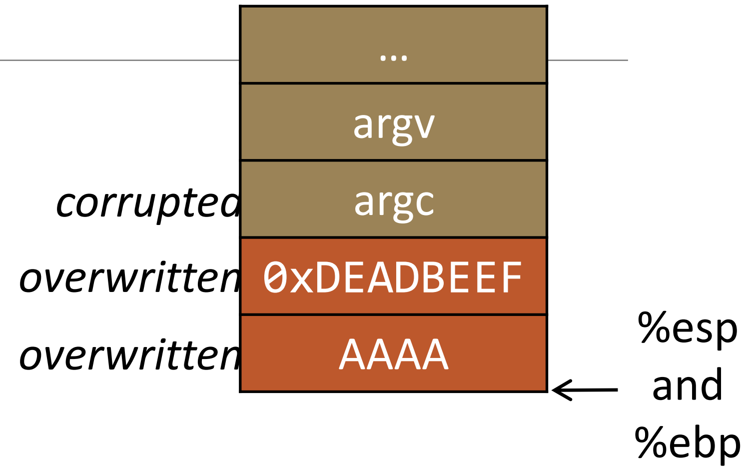


13

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
```



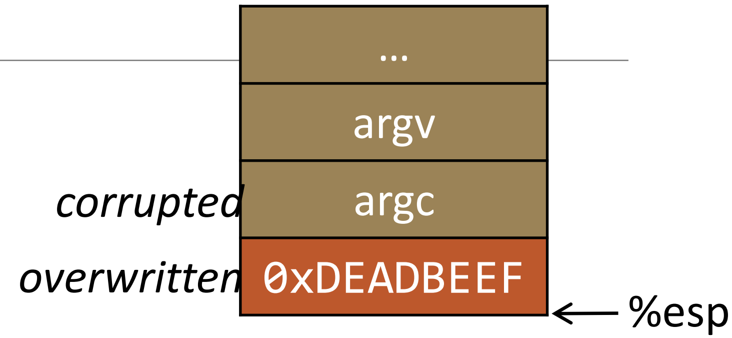

```
leave
1. mov %ebp,%esp
2. pop %ebp
```

← %esp

14

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
```


15

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)

16

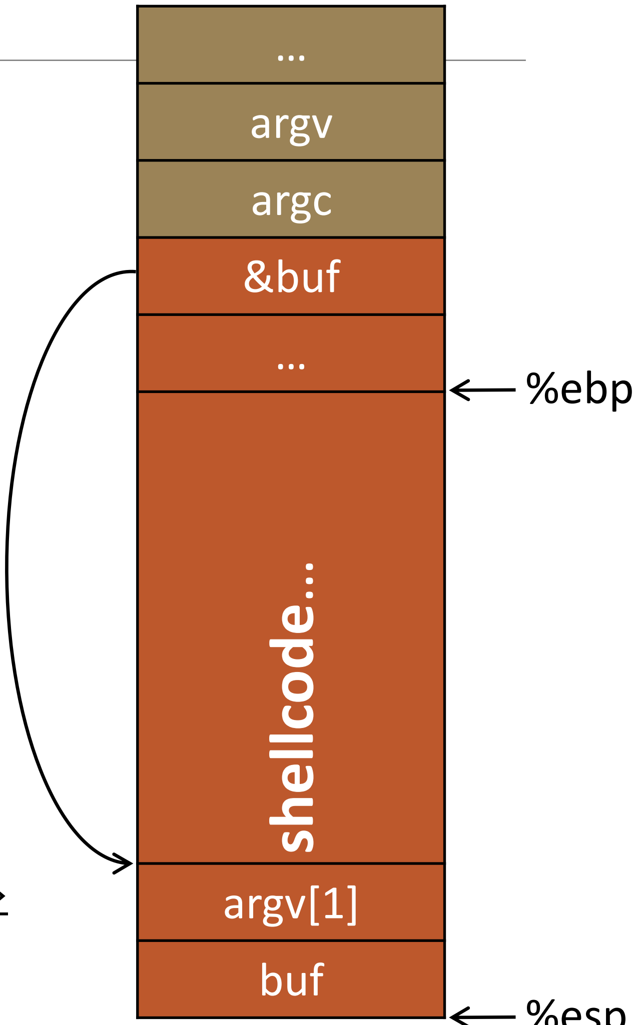
Shellcode

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

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

or search online

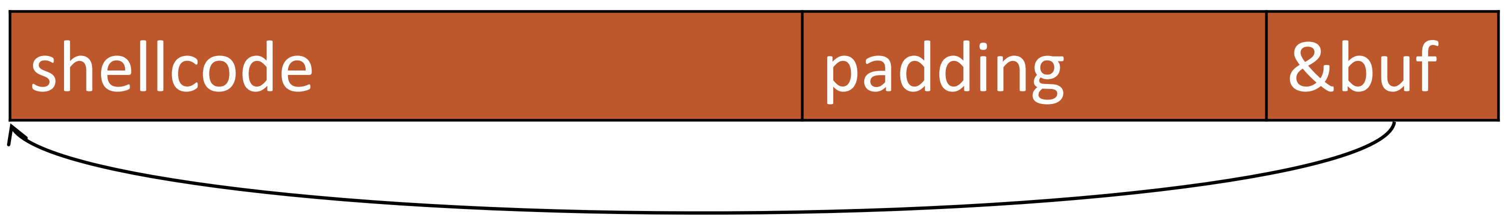
```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



Recap

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

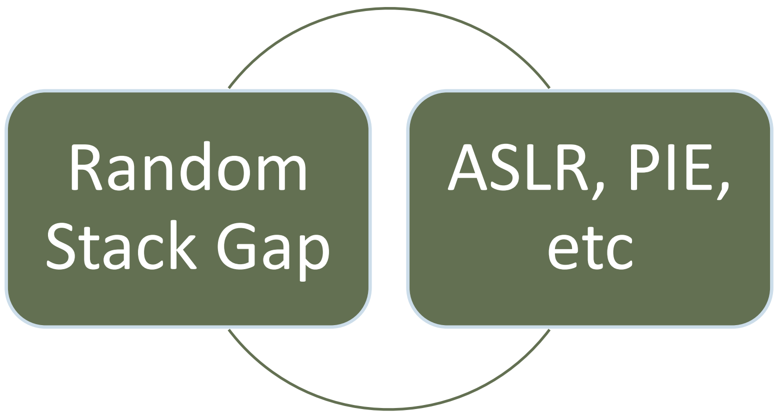
1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size



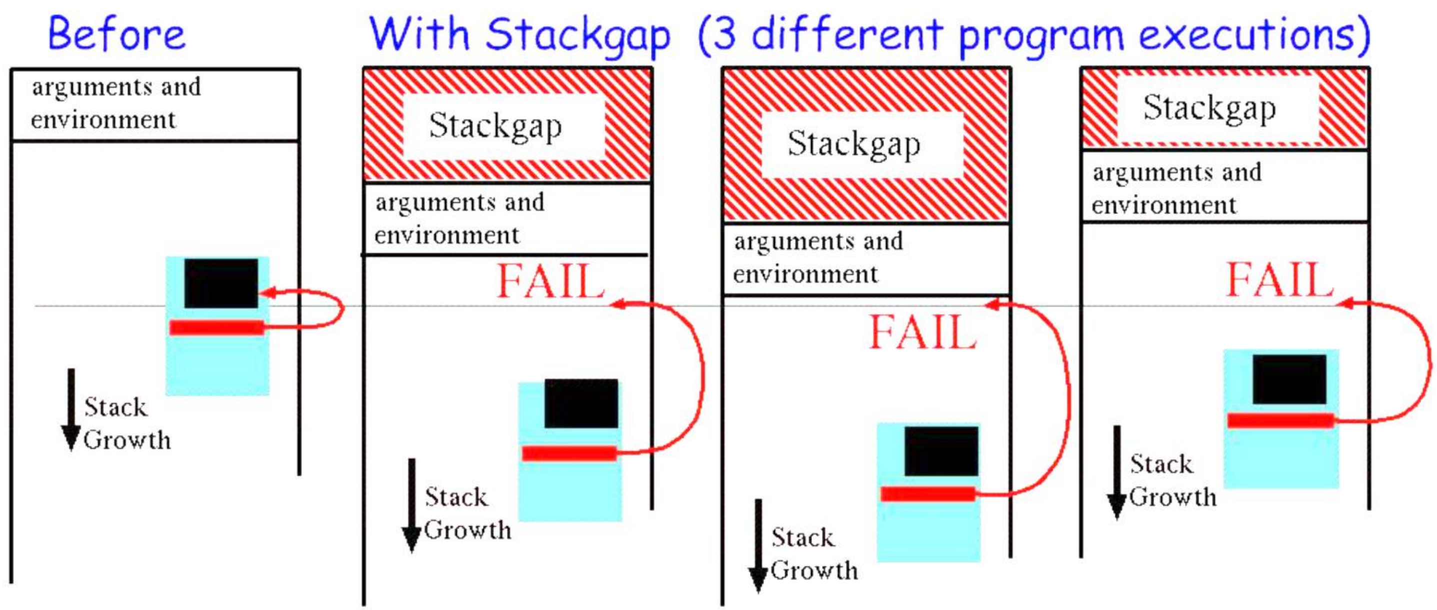
computation + ***control***

- Make it harder to control a subverted flow
- Make taking control of the flow innocuous
- Make it harder to get control of the flow

Disrupting Exploitative Operations



Random Stack Gap



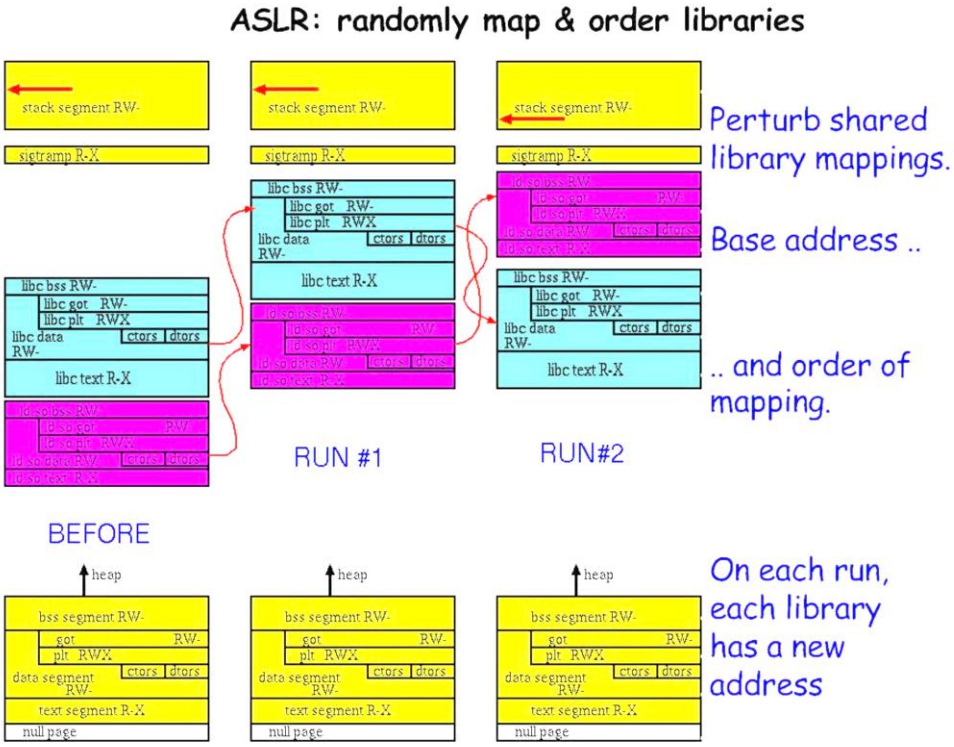
ASLR

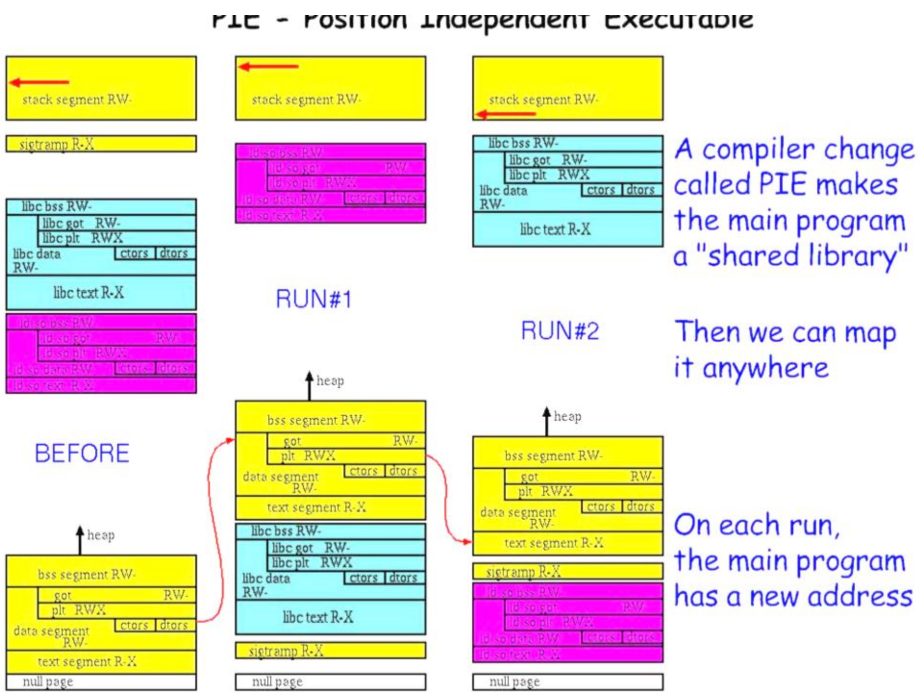
Address Space Layout Randomization

Subversion usually needs to know memory layout

General goal: make layout unpredictable

Start With Libraries





Add Executables

Finally, Dynamic Allocations

mmap

malloc

Limitations of ASLR

1. **Boot-time based randomization**
2. **Unsupported executables/libraries, low-entropy.**
3. **ASLR does not *trap* the attack**
4. **ASLR does not alert in a case of an attack**
5. **ASLR does not *provide information* about an attack**
6. **ASLR is being bypassed by exploits daily**

Posted by **MORDECHAI GURI, PH.D.** on December 17, 2015

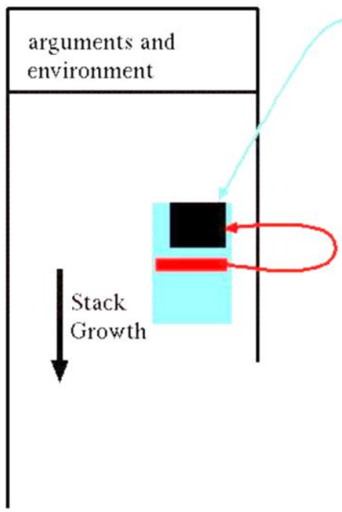
Making Violations Less Dangerous

W^X Permissions

rodata

W|x Permissions

Many bugs are exploitable because the address space has memory that is both writeable and executable (permissions = W | X)



this location has to be executable for the exploit to work

We could make the stack non-executable...

Hmmmm... how about a generic policy for the whole address space:

A page may be either writeable or executable, but not both (unless the program specifically requests)

We call this policy W ^ X (W xor X)

Executable Stacks

This is what static executables
used to look like in memory.

What is this?

The stack has a piece of executable
called the "signal trampoline"

First problem: The stack is
executable

5.6 Returning from a signal handler

When the program was interrupted by a signal, its status (including all integer and floating point registers) was saved, to be restored just before execution continues at the point of interruption.

This means that the return from the signal handler is more complicated than an arbitrary procedure return - the saved state must be restored.

To this end, the kernel arranges that the return from the signal handler causes a jump to a short code sequence (sometimes called *trampoline*) that executes a `sigreturn()` system call. This system call takes care of everything.

In the old days the trampoline lived on the stack, but nowadays (since 2.5.69) we have a trampoline in the vsyscall page, so that this trampoline no longer is an obstacle in case one wants a non-executable stack.

Linux Trampoline?

Linux Trampoline!!!

No-execute stacks [edit]

Some implementations of trampolines cause a loss of no-execute stacks (NX stack). In the GNU Compiler Collection (GCC) in particular, a nested function builds a trampoline on the stack at runtime, and then calls the nested function through the data on stack. The trampoline requires the stack to be executable.

No execute stacks and nested functions are mutually exclusive under GCC. If a nested function is used in the development of a program, then the NX stack is silently lost. GCC offers the `-Wtrampoline` warning to alert of the condition.

Software engineered using secure development lifecycle often do not allow the use of nested functions due to the loss of NX stacks.[11]

.wikipedia.org/wiki/Trampoline_(computing)#No-execute_stacks

The .rodata Segment

W^X Transition: The .rodata segment

Readonly strings and pointers were stored in the .text segment: X | R

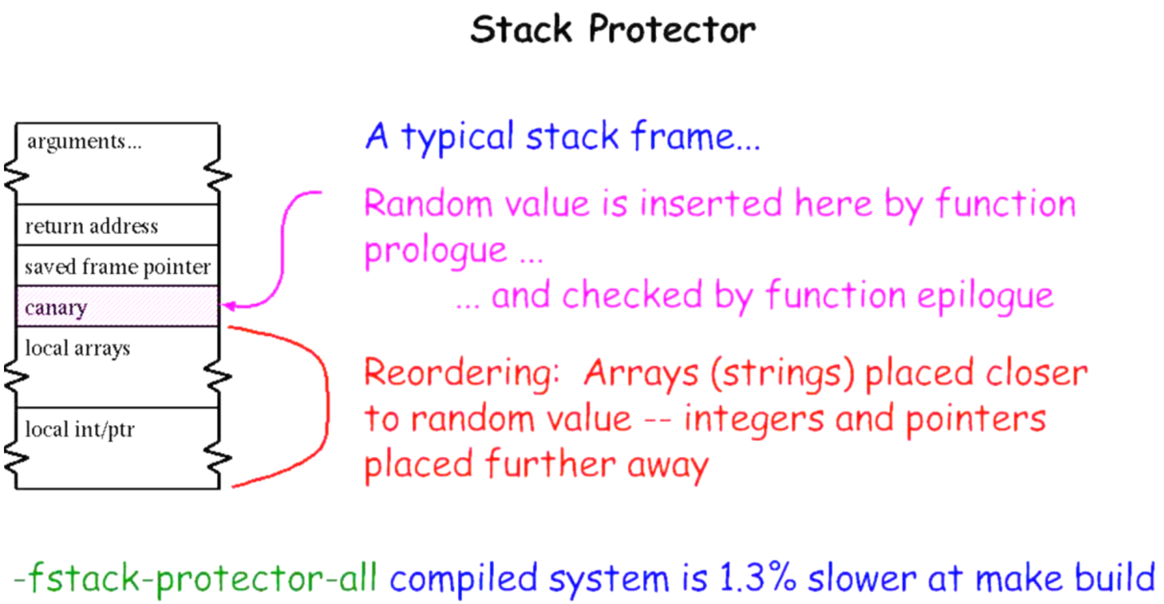
Meaning const data could be executed (could be code an attacker could use as ROP payload)

Solution: start using the ELF .rodata segment

These objects are now only R, lost their X permission

Greater policy: "minimal set of permissions"

Finally, Blocking Exploits

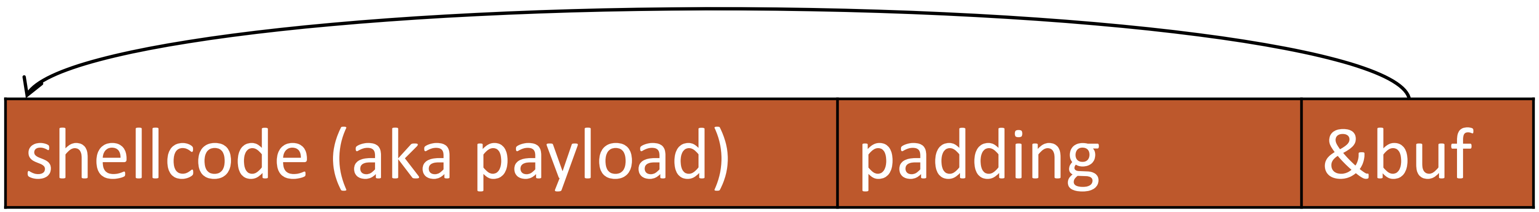


return-oriented programming

David Brumley
Carnegie Mellon University

Credit: Some slides from Ed Schwartz

Control Flow Hijack: Always control + computation

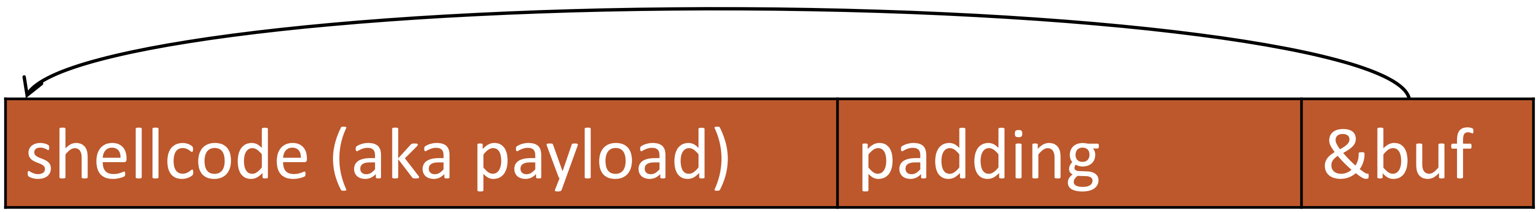


computation + ***control***

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

Previously: Executable code as input

Control Flow Hijack: Always control + computation



computation + ***control***

Today: Return Oriented Programming
Execution without injecting code

ROP Overview

Idea:

We forge shell code out of existing application logic gadgets

Requirements:

vulnerability + gadgets + some *unrandomized* code
(we need to know the addresses of gadgets)

Technically, *PREDICTABLE*

Motivation: Return-to-libc Attack

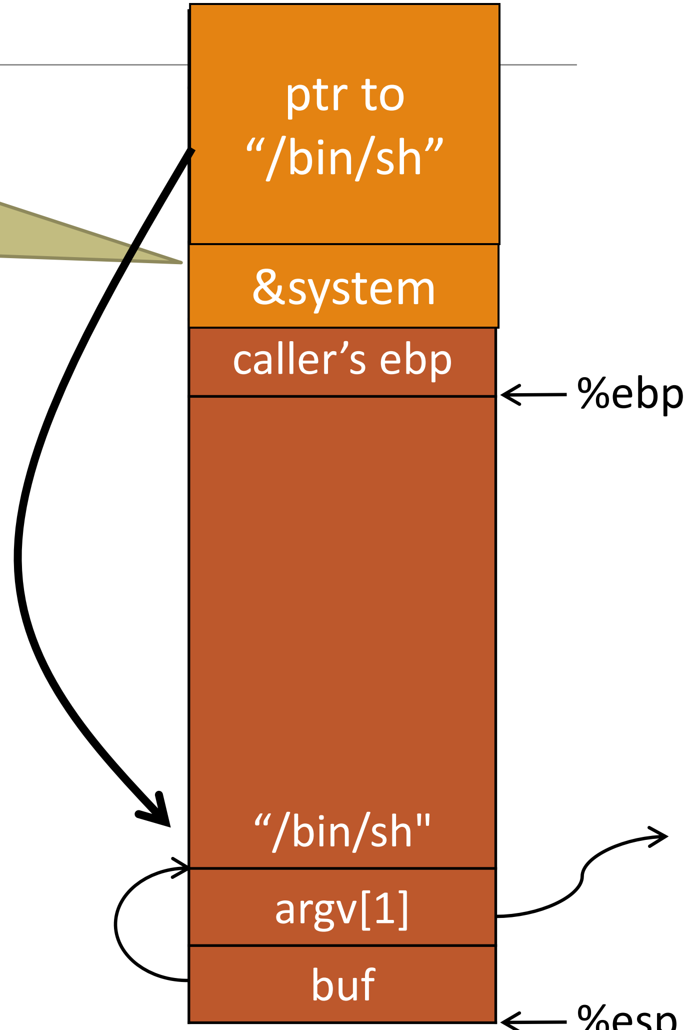
ret transfers control to `system`, which finds arguments on stack

Overwrite return address with *address* of libc function

setup fake return address and argument(s)

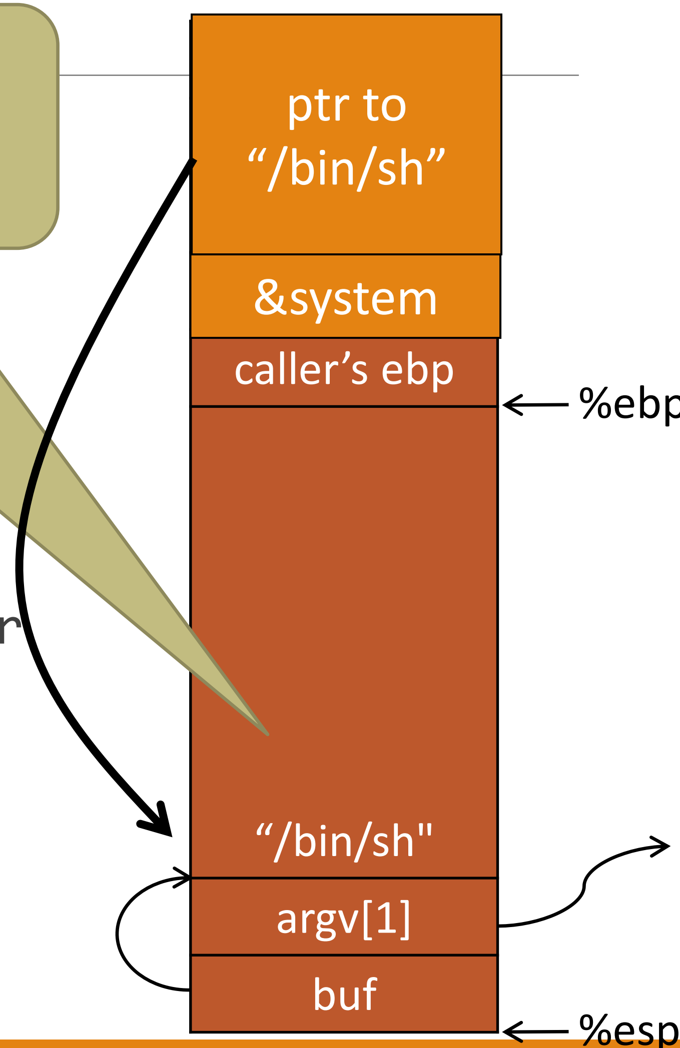
`ret` will "call" libc function

No injected code!



Question

Randomized!



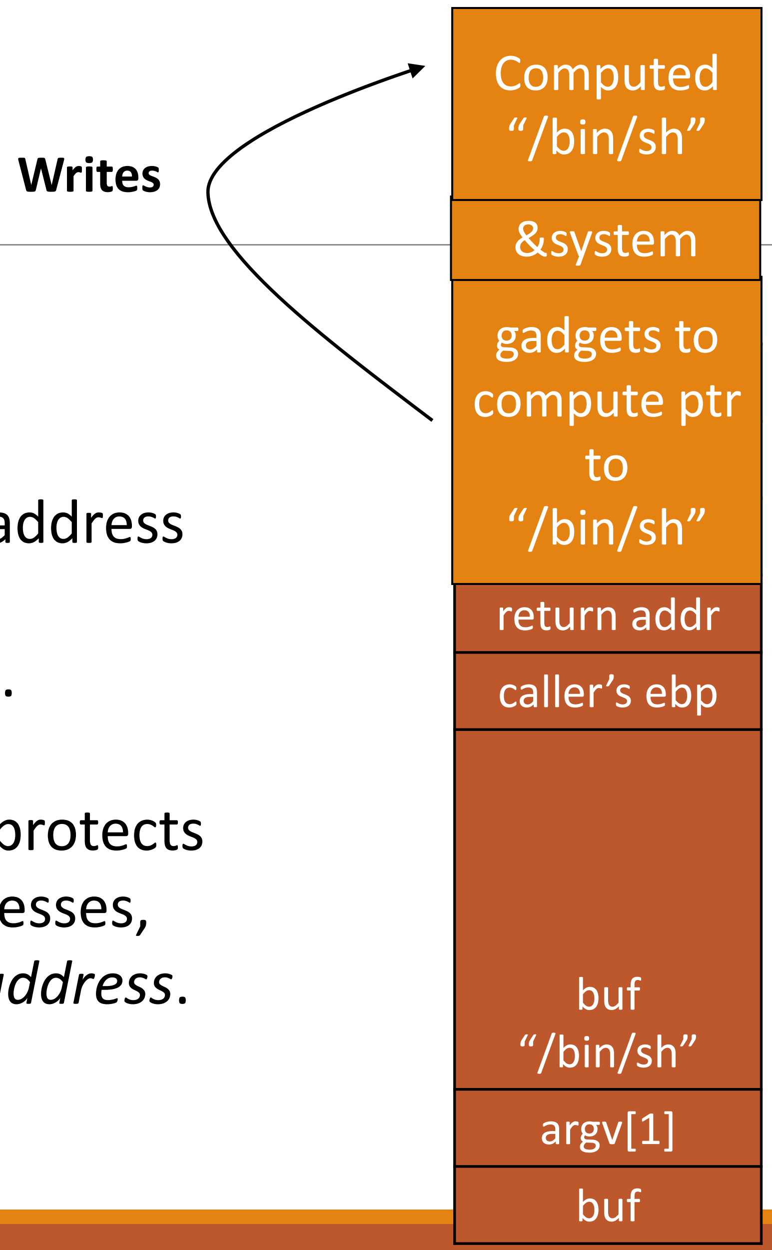
With ASLR, we cannot forge a correct value for `ptr` since ASLR will randomize addresses.

What can we do?

Idea!

Get a copy of ESP to calculate address of
“/bin/sh” on randomized stack.

This works because ASLR only protects against knowing *absolute* addresses, while we will find it’s *relative address*.



Return Chaining

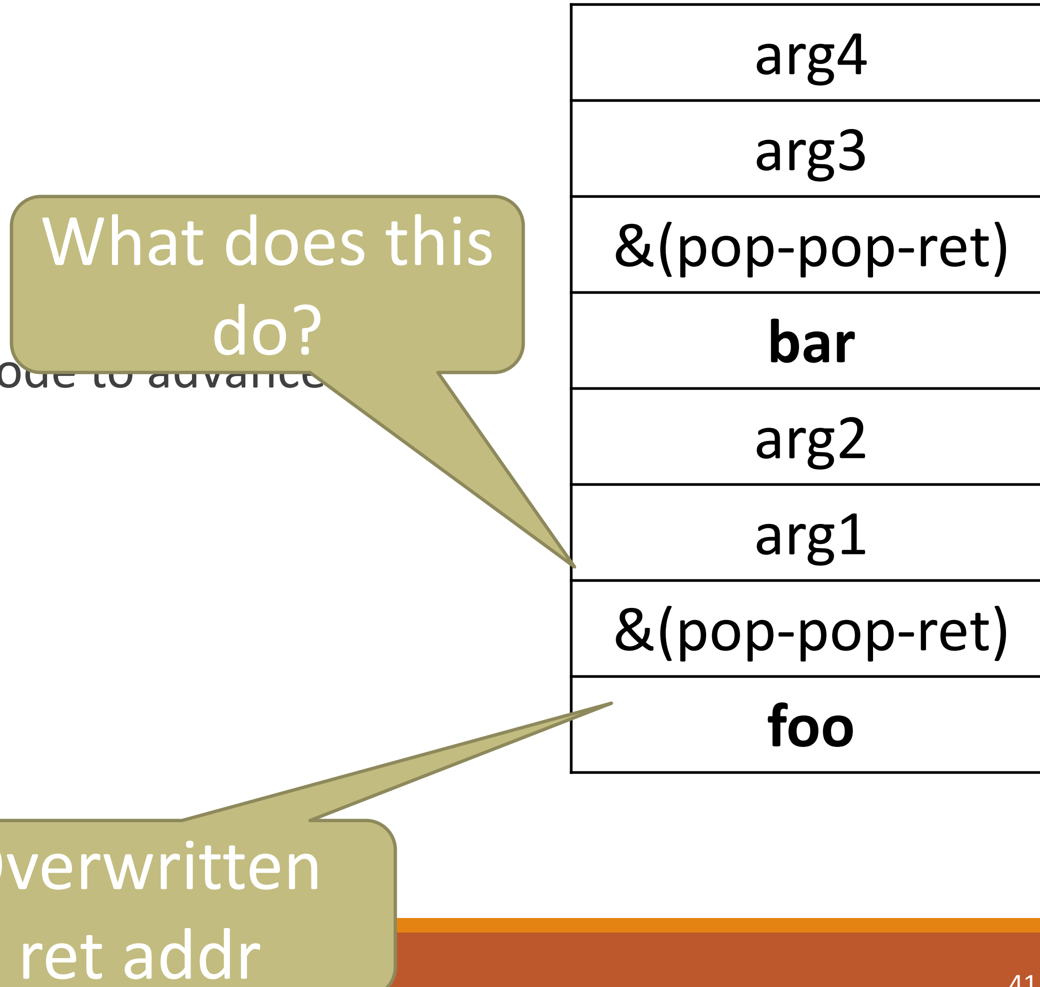
Suppose we want to call 2 functions in our exploit:

foo(arg1, arg2)
bar(arg3, arg4)

Stack unwinds up

First function returns into code to advance stack pointer

- e.g., pop; pop; ret



Return Chaining

When **foo** is executing, &pop-pop-ret is at the saved EIP slot.

When **foo** returns, it executes pop-pop-ret to clear up arg1 (pop), arg2 (pop), and transfer control to **bar** (ret)

arg4
arg3
&(pop-pop-ret)
bar
arg2
arg1
&(pop-pop-ret)
foo

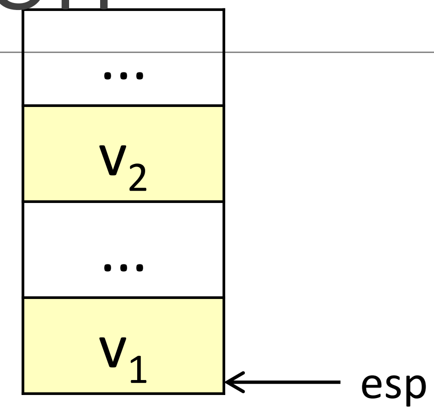
There are many *semantically equivalent* ways to achieve the same net shellcode effect

Let’s practice thinking in gadgets

An example operation

`Mem[v2] = v1`

Desired Logic



Stack

a_1: `mov eax, [esp] ; eax has v1`

a_2: `mov ebx, [esp+8] ; ebx has v2`

a_3: `mov [ebx], eax ; Mem[v2] = eax`

Implementation 1

implementing with gadgets

`Mem[v2] = v1`

Desired Logic

Suppose a_5 and a_3 on stack

Stack	
a_5	
v_2	
a_3	
v_1	← esp

Stack

eax	v_1
ebx	
eip	a_1

```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

`Mem[v2] = v1`

Desired Logic

Stack	
a_5	
v_2	
a_3	
	← esp
v_1	

Stack

eax	v_1
ebx	
eip	a_3

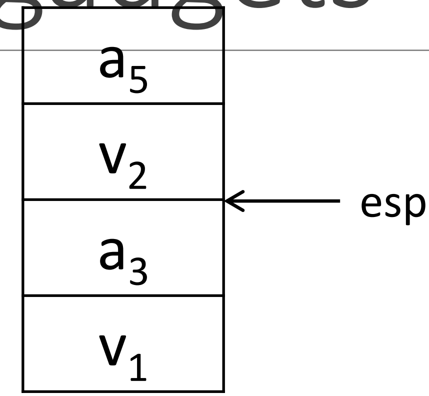
a_1: pop eax
a_2: ret
a_3: pop ebx
a_4: ret
a_5: mov [ebx], eax

Implementation 2

implementing with gadgets

Mem[v2] = v1

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_3

a_1: pop eax
a_2: ret
a_3: pop ebx
a_4: ret
a_5: mov [ebx], eax

Implementation 2

implementing with gadgets

`Mem[v2] = v1`

Desired Logic

a_5	
v_2	← esp
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

implementing with gadgets

`Mem[v2] = v1`

Desired Logic

a_5	← esp
v_2	
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_5

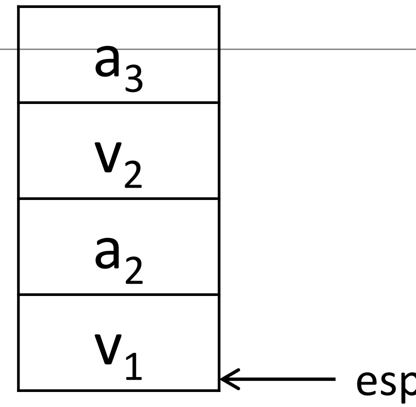
```
a1: pop eax;     } Gadget 1
a2: ret
a3: pop ebx;     } Gadget 2
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Equivalence

`Mem[v2] = v1`

Desired Logic



Stack

semantically equivalent

"Gadgets"

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Implementation 2

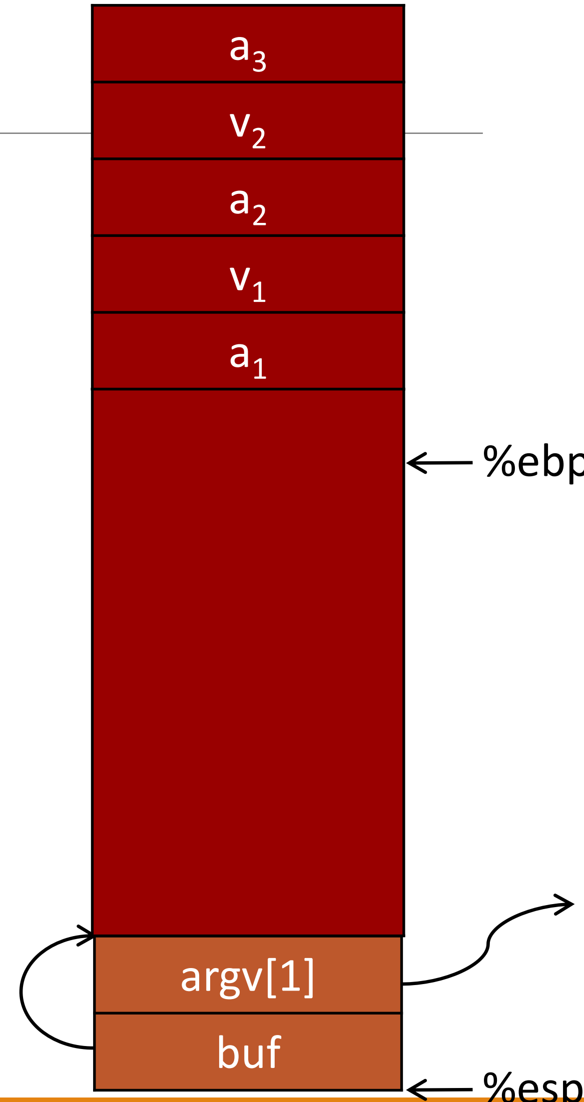
Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Desired store executed!



Gadgets

A gadget is a set of instructions for carrying out a semantic action

- mov, add, etc.

Gadgets typically have a number of instructions

- One instruction = native instruction set
- More instructions = synthesize <- ROP

Gadgets in ROP generally (but not always) end in return

Image by Dino Dai Zovi

RO(P?) Programming

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

Attacker Oriented Programming?

Behavior isn't a program

We should be able to perfectly detect bad behavior, right?

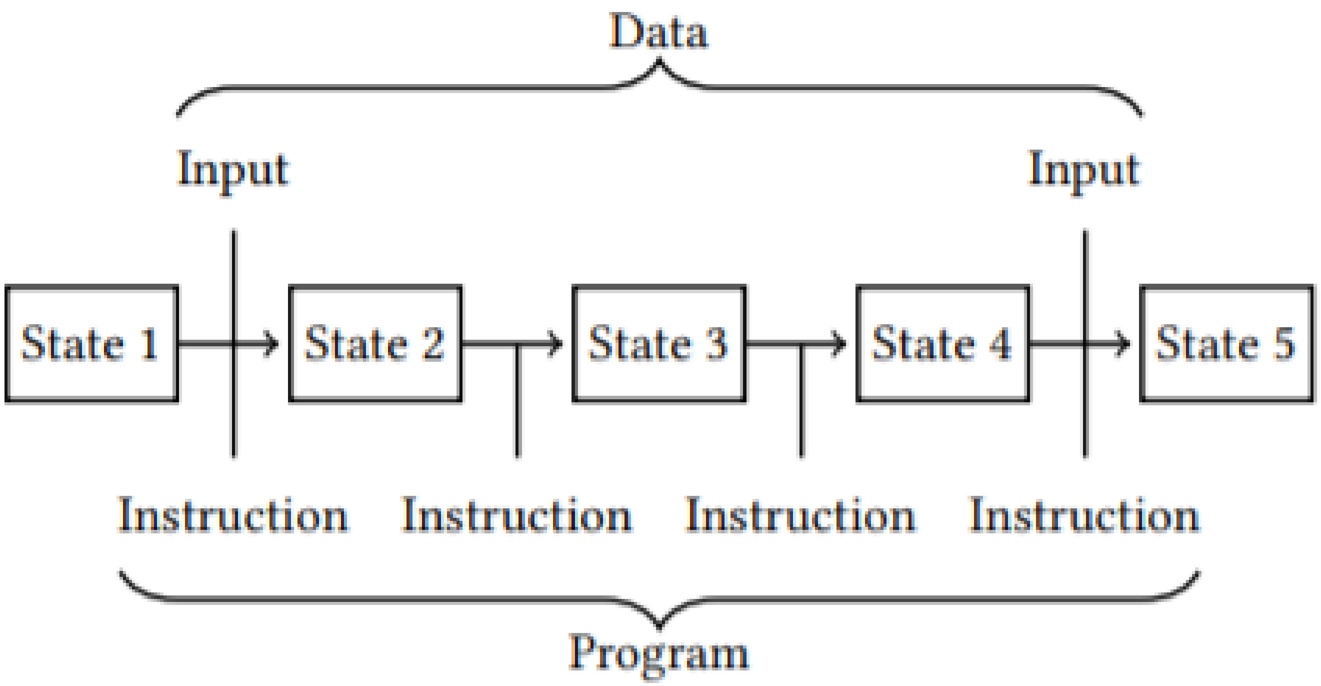
“Weird Machines”

“Weird machines, exploitability, and provable unexploitability”

Written by Thomas Dullien

Explains that users interacting with a program ***is a program***

What is a Program?



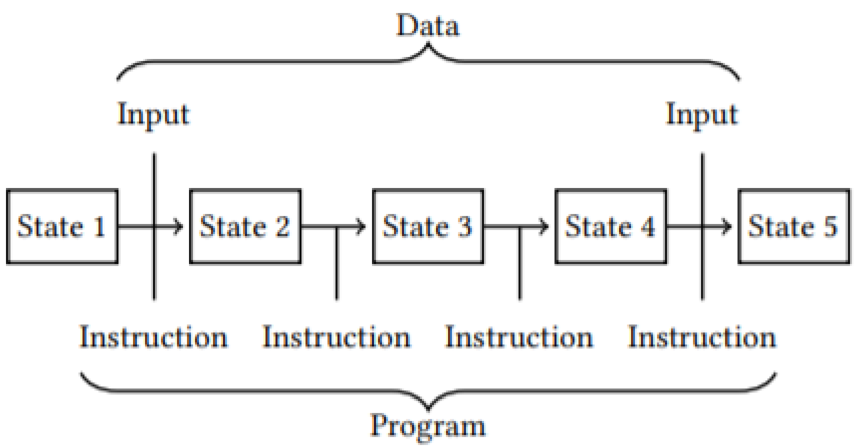
*From Dullien's Paper

State Machine View

View a "Program" as a state machine

Program starts in state S_0

Based on instruction, advances to state S_i

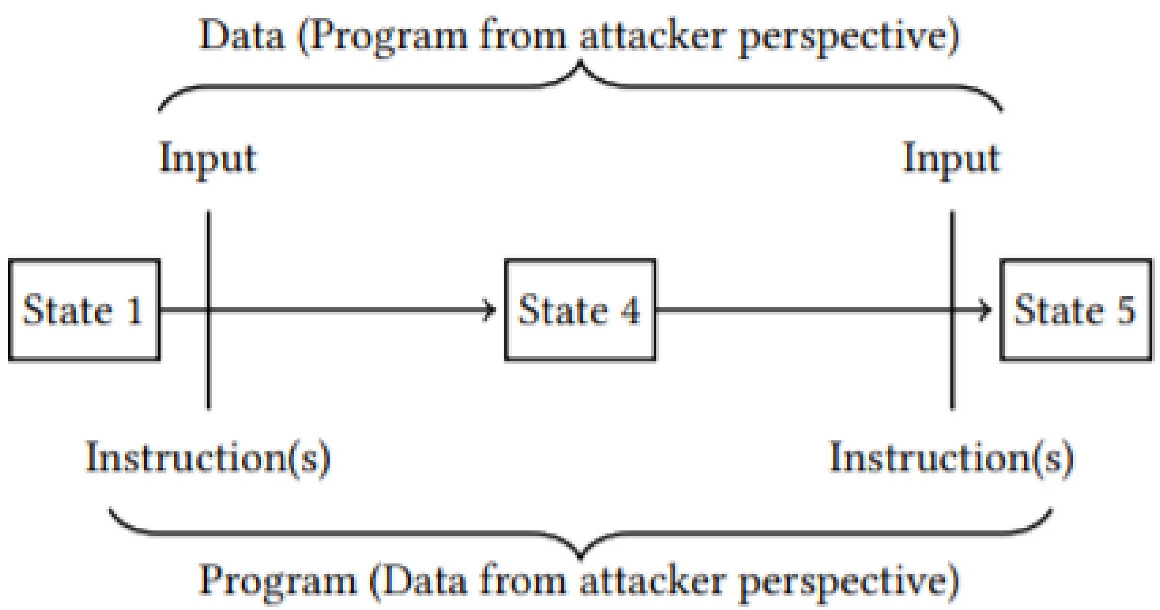


States and User Interactions

Program is in some State. Call it S_0

User interacts with the program

Program advances to state S_1



What is a “User”?

Do we literally mean a flesh-and-blood human?

Really, “user” is just whatever provides the input

This can, of course, just be another process

Thus, two processes interacting ***IS A PROGRAM***

Therefore, determining if “behavior” is good is undecidable