

CSE 127 Computer Security

Stefan Savage,
Fall 2023, Lecture 6

Control Flow Vulnerabilities:
ROP and CFI

PA 2 comes out today

- Buffer overflows!
 - You get to do all the stuff we talked about!
- Two quick details I want to emphasize:
 - The Gradescope Autograder **will not test your submission**. When it gives you a “score” it is simply how many of the required files you provided (i.e., 8.0/8.0 does not mean you aced the PA, just that you provided 8 parts)
 - **Cookies!** One of the first things you will do is generate a cookie file containing the usernames of your group members (max 2). This file is super important! Each assignment is slightly different based on the cookie file so if you mess up the cookie file or change it your assignment won't pass.
 - Corollary: if you try to change the makeup of your group, you'll need to start over... so don't do that.

Recall from last class

- Recall: Data Execution Prevention (DEP/W^X)
 - Prevent attacker input (which is data) from being interpreted as code by marking data pages as non-executable
 - One exception: applications like browsers that explicitly mark some of their data as executable to Just-In-Time compilation (JIT)
 - Win!
- Is there another way for an attacker to execute arbitrary code even without the ability to inject it into the victim process?

Code Reuse Attacks

- Use the code that's already there
- What code is already there?
 - Program + shared libraries (including libc)

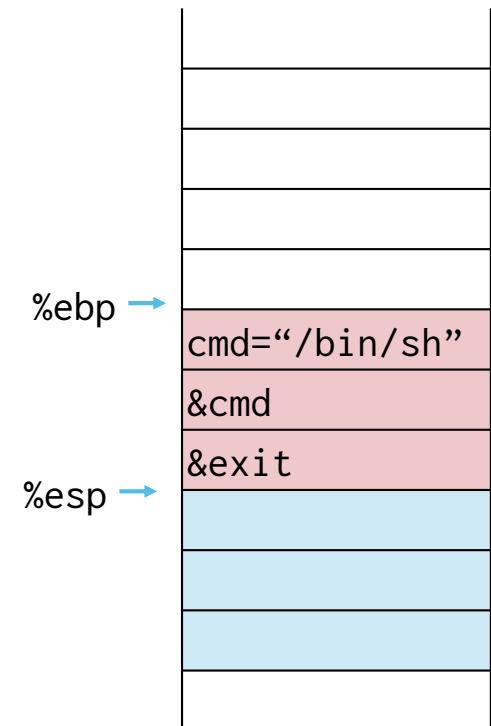


Return-To-Libc

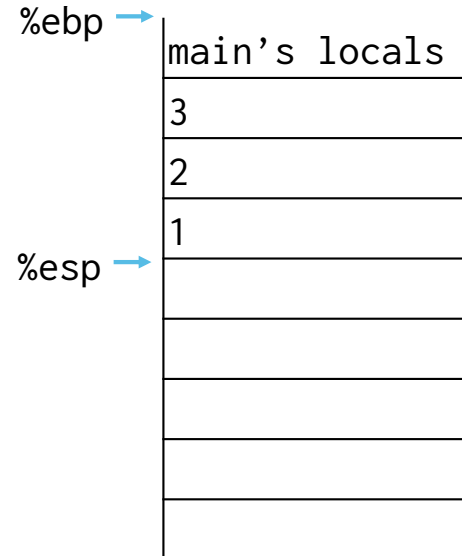
- What can we find in libc?
 - *“The system() library function uses fork(2) to create a child process that executes the shell command specified in command using execl(3) as follows:*
*execl("/bin/sh", "sh", "-c", command, (char *) 0);”*
 - Need to find the address of:
 - system()
 - String “/bin/sh”
 - Overwrite the return address to point to start of system()
 - Place address of “/bin/sh” on the stack so that system() uses it as the argument
 - To be clean, you also want to push exit() on the stack so it will shut down gracefully

Return-To-Libc

- What we want to get to
 - Transfer control to address of `system()` in `libc`
 - Setup stack frame to look like a normal call to `system()`
 - `int system(const char *command);`
 - `&exit()` system call is in the slot where the return address would be
 - `&cmd` is the argument
 - It points to the string `"/bin/sh"` stored further down the stack

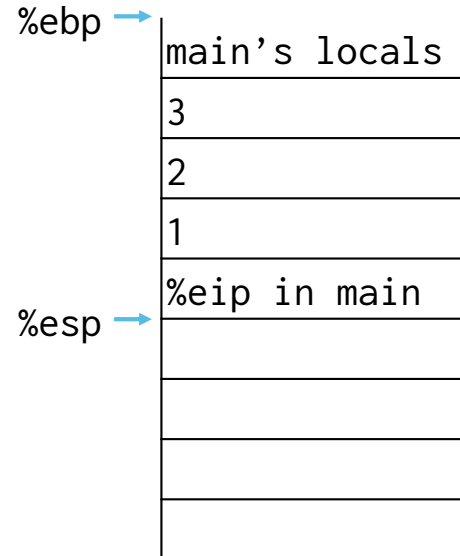


Review: calling and returning



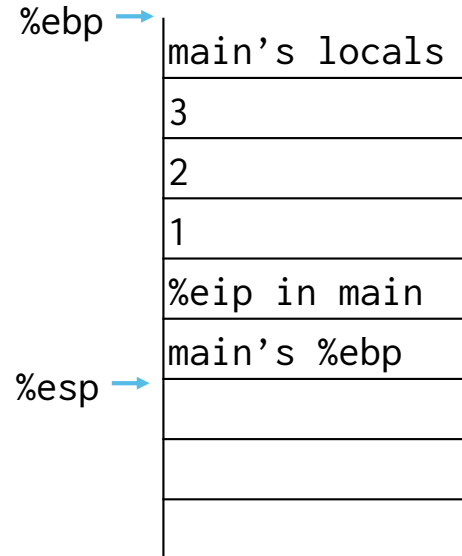
```
main()  
-> foo(1,2,3)  
—> bar(4)
```

Review: calling and returning



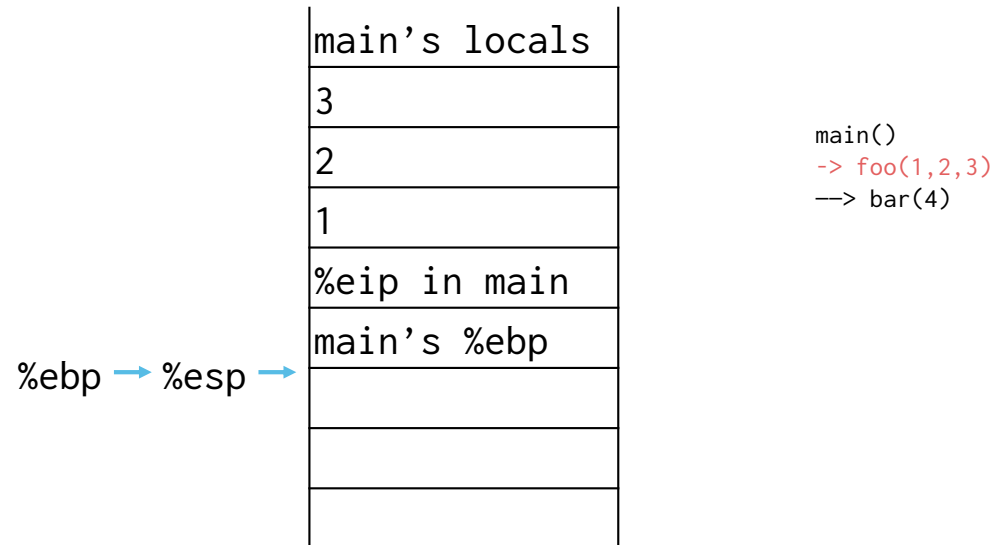
```
main()  
-> foo(1,2,3)  
--> bar(4)
```

Review: calling and returning

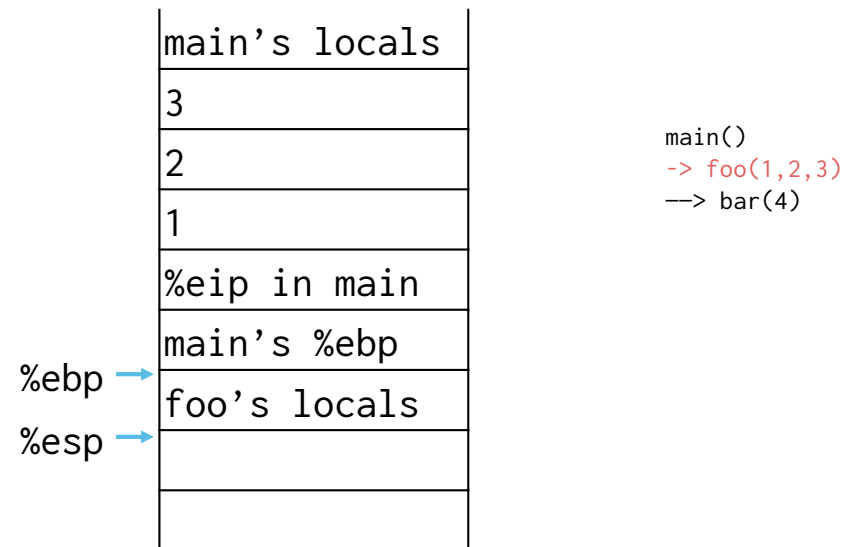


```
main()  
-> foo(1,2,3)  
—> bar(4)
```

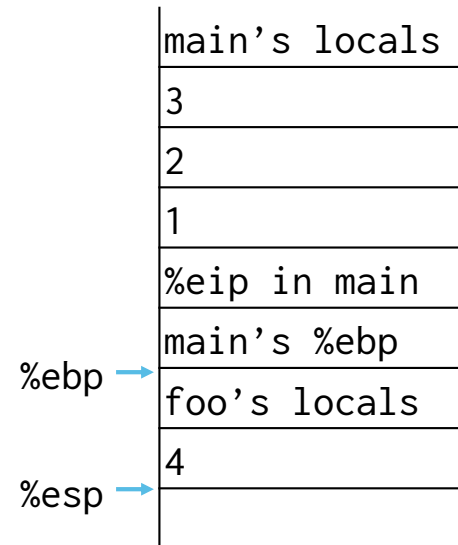
Review: calling and returning



Review: calling and returning

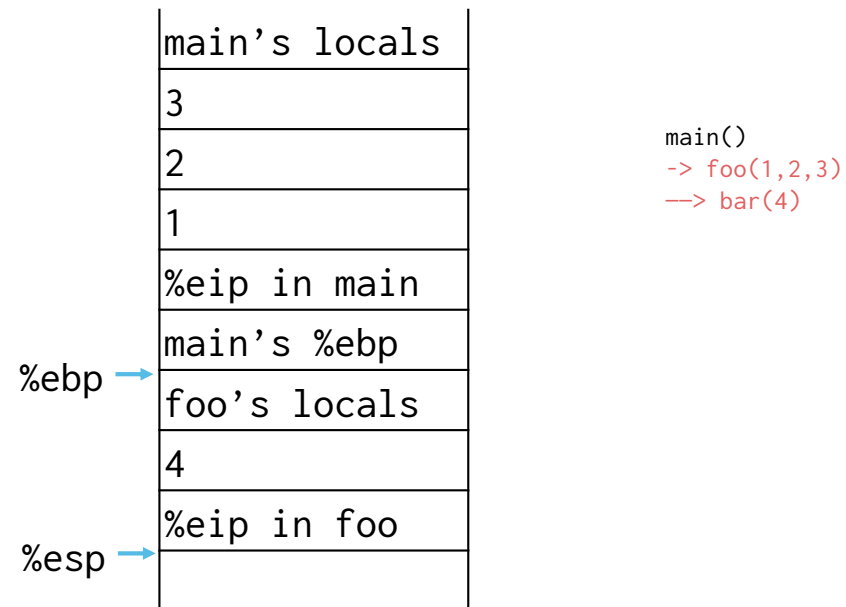


Review: calling and returning

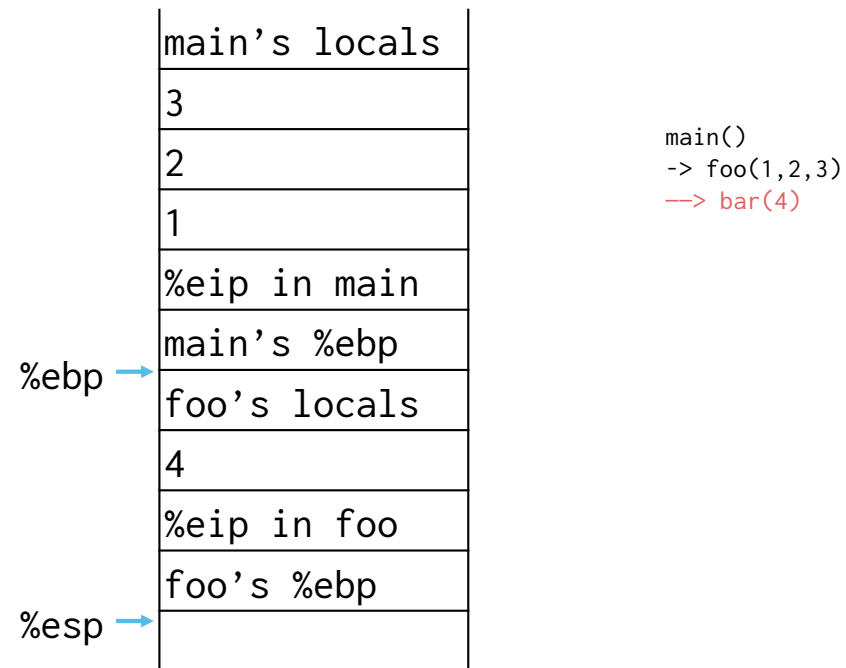


```
main()  
-> foo(1,2,3)  
—> bar(4)
```

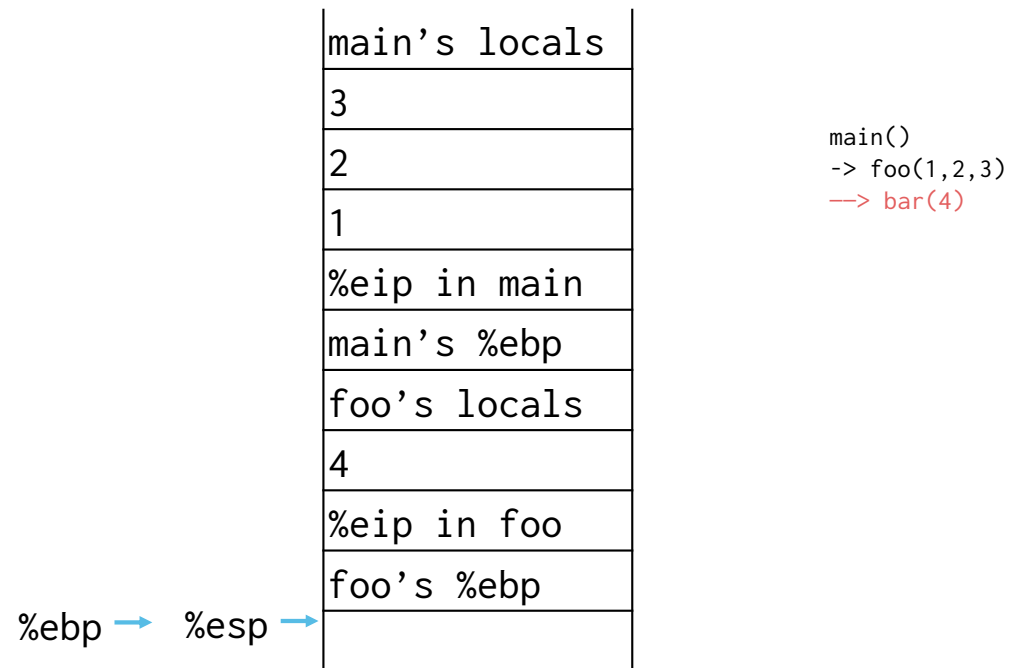
Review: calling and returning



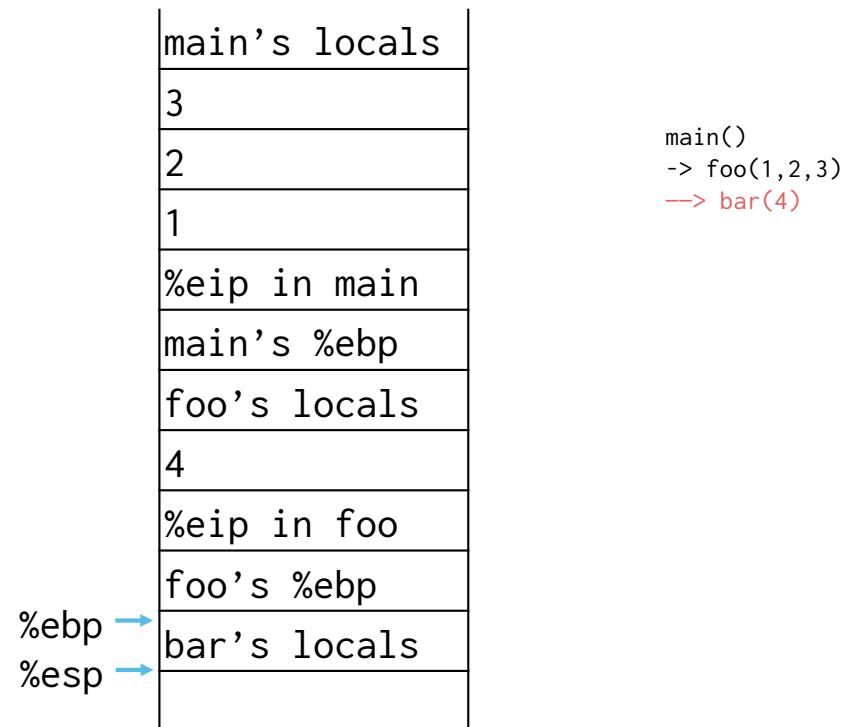
Review: calling and returning



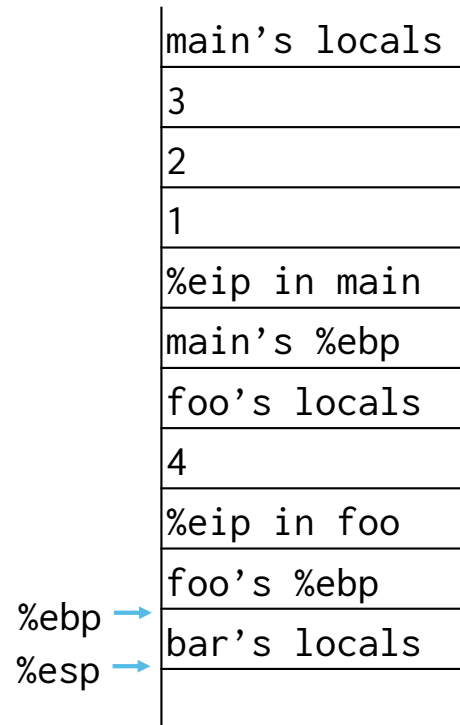
Review: calling and returning



Review: calling and returning



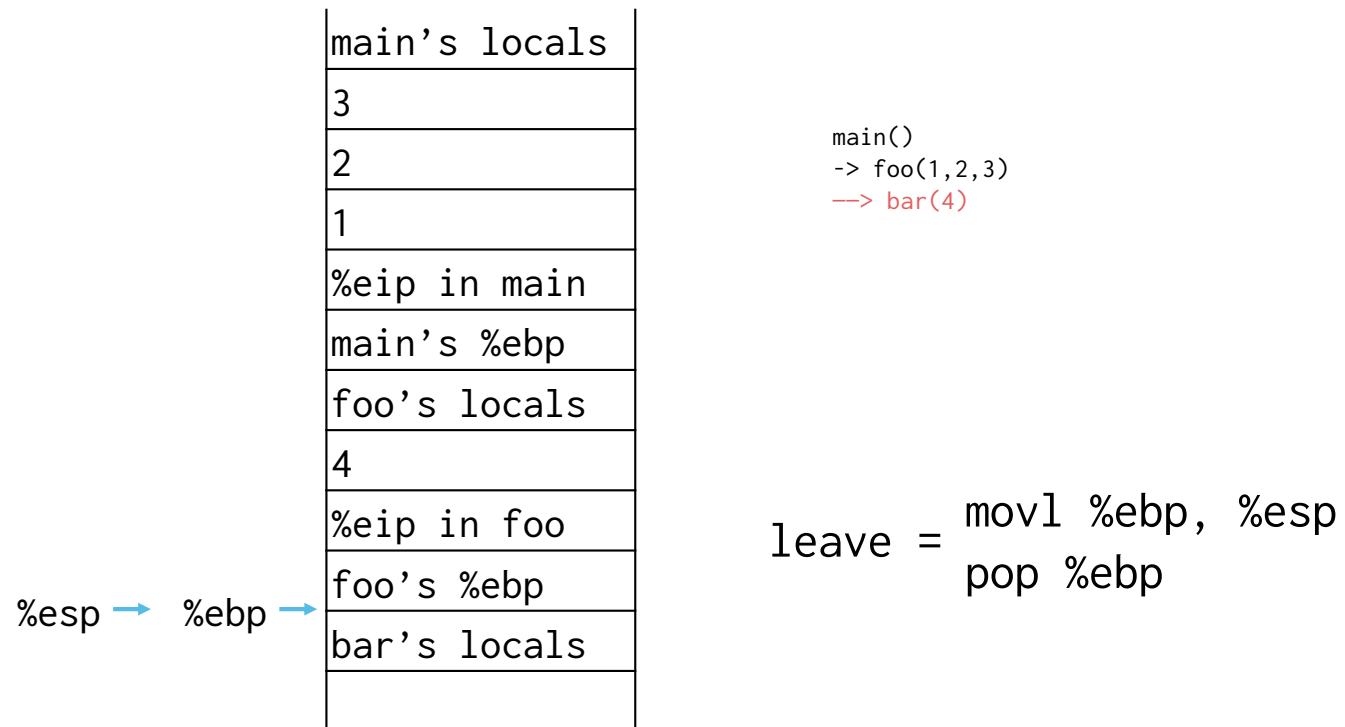
Review: calling and returning



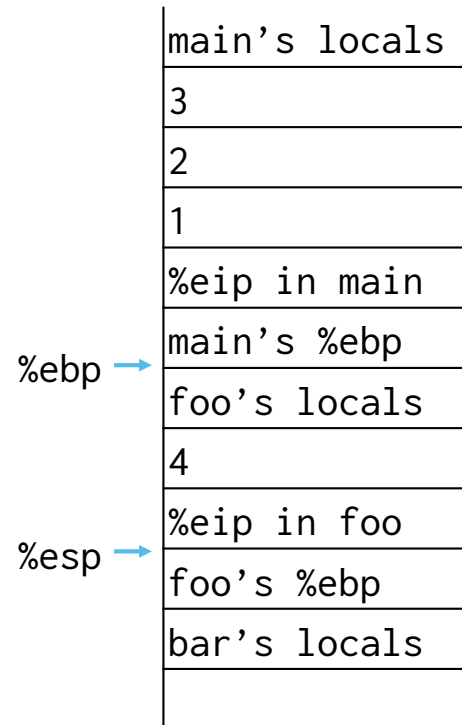
```
main()  
-> foo(1,2,3)  
→ bar(4)
```

```
leave = movl %ebp, %esp  
       pop %ebp
```

Review: calling and returning



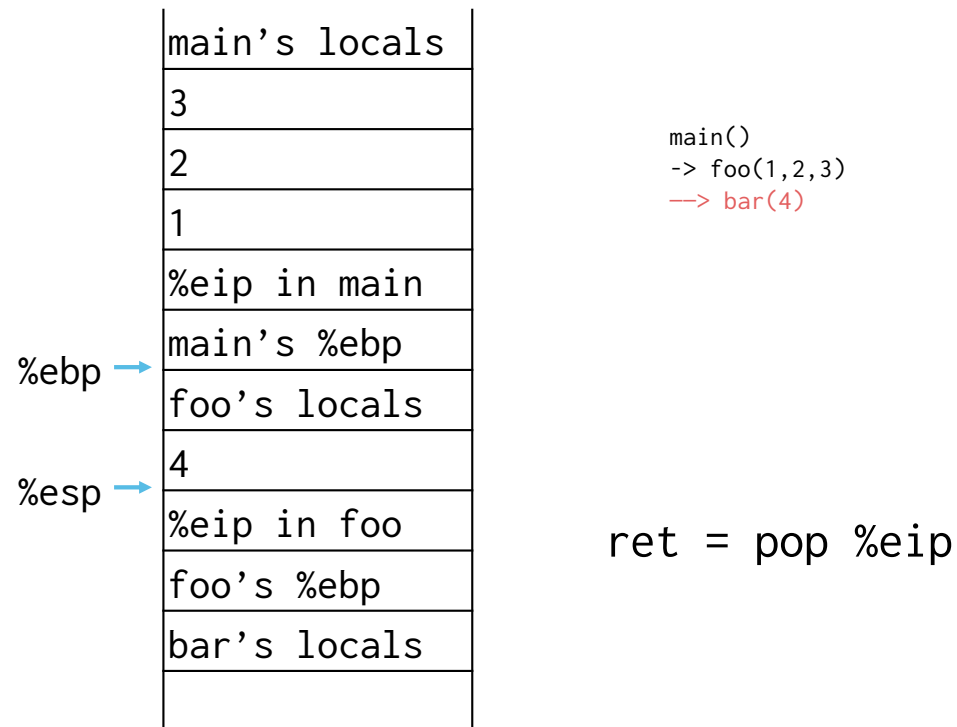
Review: calling and returning



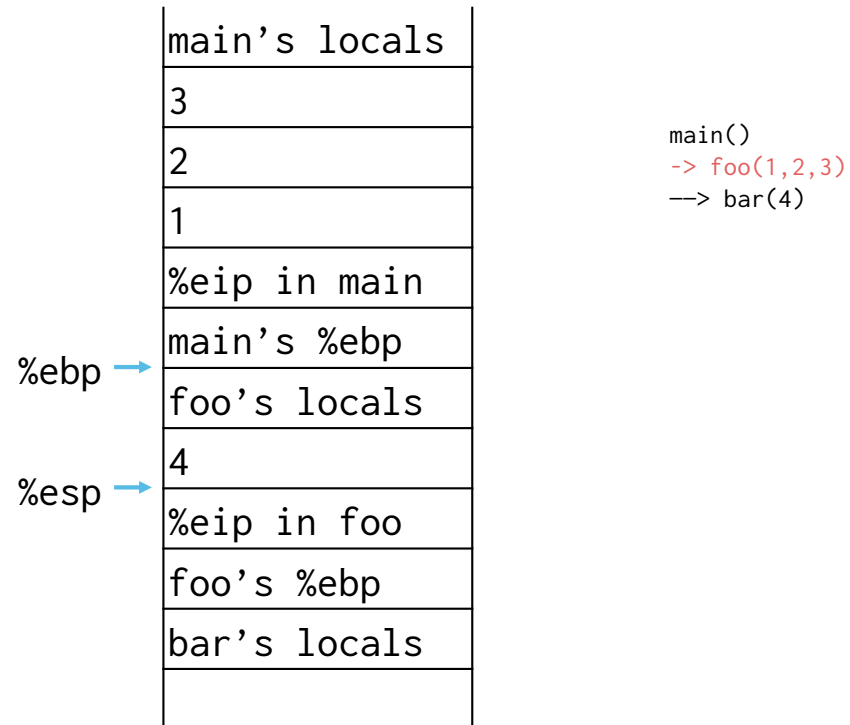
```
main()  
-> foo(1,2,3)  
→ bar(4)
```

```
leave = movl %ebp, %esp  
       pop %ebp
```

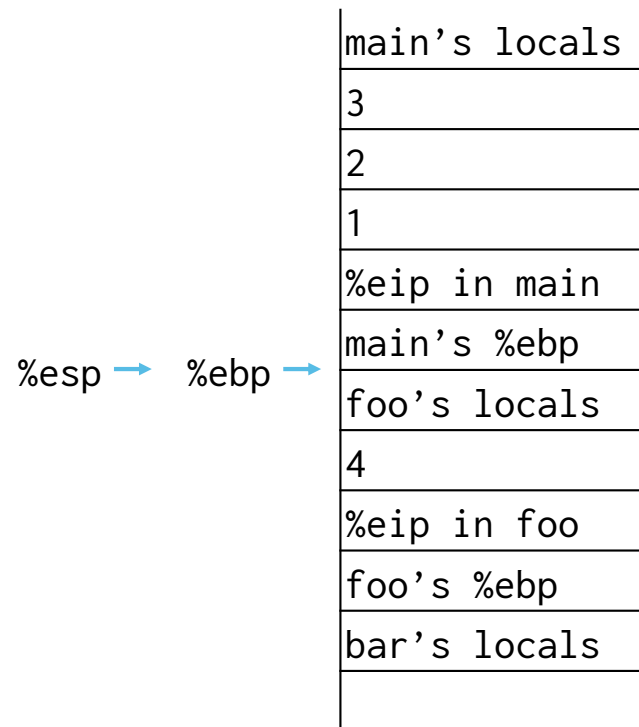
Review: calling and returning



Review: calling and returning



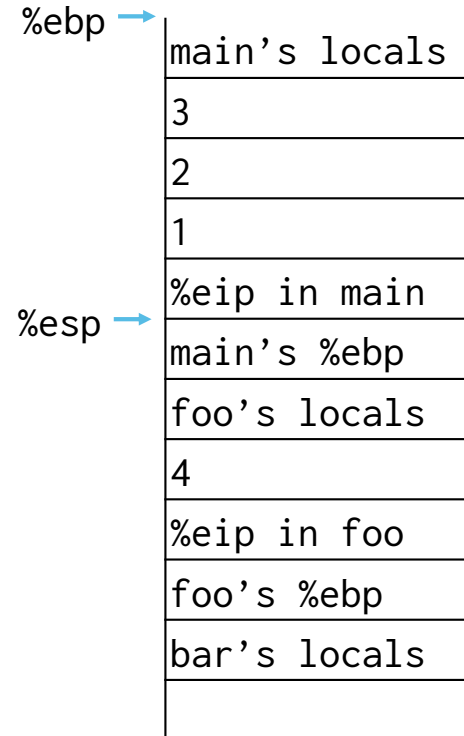
Review: calling and returning



```
main()  
-> foo(1,2,3)  
—> bar(4)
```

```
leave = movl %ebp, %esp  
       pop %ebp
```

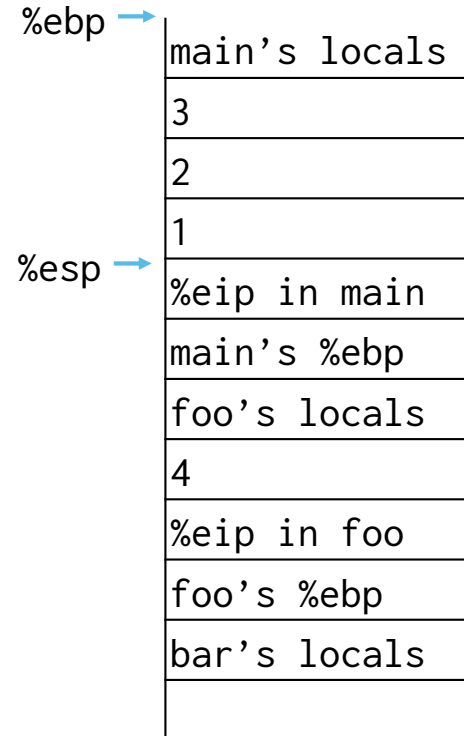
Review: calling and returning



```
main()  
-> foo(1,2,3)  
—> bar(4)
```

```
leave = movl %ebp, %esp  
       pop %ebp
```

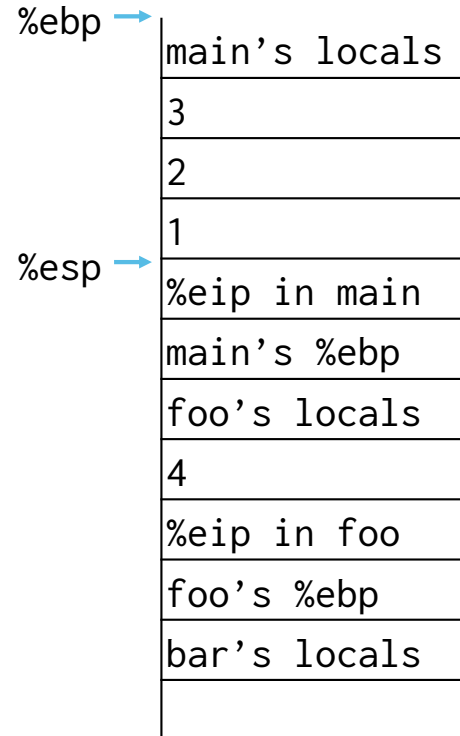
Review: calling and returning



```
main()  
-> foo(1,2,3)  
--> bar(4)
```

```
ret = pop %eip
```

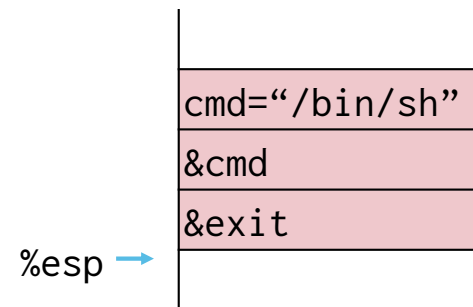
Review: calling and returning



```
main()  
-> foo(1,2,3)  
—> bar(4)
```

Suppose `bar()` had a stack overflow

- Our goal: call `system("/bin/sh")`
- Remember: Need to set up stack frame that looks like a legit call to `system`:



- But we're not going to use the `call` instruction to jump to `system`; we're going to use `ret`

Hijacking control flow

	main's locals	
	3	
	2	
	1	
	%eip in main	
	main's %ebp	cmd="/bin/sh"
	foo's locals	&cmd
	4	&exit
	%eip in foo	&system
%ebp →	foo's %ebp	
%esp →	bar's locals	

About to call leave

Hijacking control flow

	main's locals	
	3	
	2	
	1	
	%eip in main	
%ebp →	main's %ebp	cmd="/bin/sh"
	foo's locals	&cmd
%esp →	4	&exit
	%eip in foo	&system
	foo's %ebp	
	bar's locals	

Executing system()
gets arguments from
stack

Return To Libc

- Many different variants
- What else can attackers do by calling available functions with parameters of their choosing?
 - Move shellcode to unprotected memory
 - Change permissions on stack pages (`mprotect()`)
 - Etc.



Brendan Dolan-Gavitt
@moyix

Another CTF trick: if you need a string for `system()` that will get you a shell, consider the humble "ed". It supports running shell commands (!), and b/c of English past tense is often available as a suffix of some existing string in the binary, e.g.: "File transfer complet**ed**"

Return Oriented Programming (ROP)

- What if we cannot find just the right function? Or need more complex computation?

The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Return-Oriented Programming

is A lot like a ransom
note, BUT instead of cutting
out letters from magazines,
YOU ARE cutting out
instructions from text
segments

ret Steve Checkoway
ret Dino Dai Zovi

Return Oriented Programming (ROP)

- What happens if we jump *almost* to the end of some function?
 - We execute the last few instructions, and then?
 - Then we return. But where?
 - To the return address on the stack. But we overwrote the stack with our own data so **we control this address**
 - Let's choose to return to another *tail of an existing function*
 - Rinse and repeat

Return Oriented Programming (ROP)

- ROP core idea: make complex shellcode out of **existing application code**
- Stitching together arbitrary programs out of code “gadgets” already present in the target binary
 - **ROP Gadgets**: sets of code sequences ending in ret instruction.
 - Ret commonly added by compiler (at end of function)
 - But **also** (on x86) any sequence in executable memory ending in `0xC3` (`ret`).
 - x86 has variable-length instructions
 - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences

Aside: how Intel variable length instructions help ROP

- X86 instructions are variable length, yet can begin on any byte address

- Example:

```
81 c4 88 00 00 00 add $0x00000088, %esp
5f                pop %edi
5d                pop %ebp
c3                ret

00 5f 5d ad db    addb %bl, 93 (%edi)
c3                ret
```

- Result: more "function tails" to choose from

```
$otool -t /bin/ls |grep c3
0000000100000f70      39 48 38 7f 07 b8 ff ff ff ff 7d 02 5d c3 48 83
0000000100000fc0      00 00 7d 02 5d c3 48 83 c6 68 49 83 c0 68 48 89
0000000100001010      c3 48 83 c7 68 48 83 c6 68 5d e9 6b 35 00 00 55
0000000100001050      b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 49 83 c0
00000001000010a0      7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 34
00000001000010e0      48 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001120      7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 58 34
0000000100001150      b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 48 83 c1
00000001000011a0      7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 33
00000001000011e0      58 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001870      c0 09 c8 8a 0d ab 3c 00 00 89 c3 81 cb 80 00 00
0000000100001b70      5d d4 89 de e8 57 29 00 00 48 89 c3 48 85 db 0f
0000000100001c30      03 39 00 00 01 e9 52 01 00 00 0f b7 c0 83 f8 0d
0000000100001dd0      c3 48 8d 35 91 2d 00 00 eb 07 48 8d 35 c0 2d 00
0000000100001e20      36 0f b7 56 58 83 fa 07 75 02 5d c3 44 0f b7 c9
0000000100001ec0      00 48 8d 3d e2 2c 00 00 e8 21 26 00 00 48 89 c3
0000000100001f70      34 48 83 c3 02 80 f9 3a 75 19 80 7b fe 3a 75 13
0000000100001fa0      c3 84 c9 75 d0 44 89 b5 78 fb ff ff 45 89 e6 80
00000001000023b0      fb ff ff 74 5c 8b 78 74 e8 ef 20 00 00 48 89 c3
00000001000023e0      00 00 48 89 c3 48 85 db 0f 84 9a 04 00 00 48 89
0000000100002520      66 18 4d 8b 7e 20 41 8b 5e 30 48 63 c3 48 8d 34
0000000100002560      20 49 63 4e 30 41 89 04 8f 41 8b 5e 30 ff c3 41
0000000100002870      38 05 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 48
0000000100002970      c3 48 8d 3d 9e 22 00 00 48 8d 35 a1 22 00 00 48
0000000100002a30      ed 48 83 c3 68 48 89 df e8 4f 0b 00 00 89 c3 45
0000000100002a90      0f b7 7c 24 04 e8 28 0b 00 00 01 c3 89 d8 48 83
0000000100002aa0      c4 08 5b 41 5c 41 5d 41 5e 41 5f 5d c3 55 48 89
0000000100002c30      4f 28 48 8b 46 08 eb 0f 85 c0 45 8b 4f 38 48 8b
0000000100003200      00 48 83 c3 18 48 81 fb a8 01 00 00 75 84 bb 10
0000000100003260      45 89 fd 4c 8d bd b0 f7 ff ff 48 83 c3 18 48 83
00000001000032e0      5b 41 5c 41 5d 41 5e 41 5f 5d c3 48 8d 35 c5 18
0000000100003350      48 83 c4 08 5b 5d c3 48 8d 3d c6 1b 00 00 31 c0
00000001000034a0      c4 70 5b 41 5e 5d c3 e8 a0 0f 00 00 55 48 89 e5
0000000100003550      00 89 d8 48 83 c4 08 5b 5d c3 66 90 7e ff ff ff
00000001000035f0      00 00 00 5d c3 81 c1 00 60 00 00 81 e1 00 f0 00
00000001000036a0      75 06 48 83 c3 10 eb 69 48 8d 7b 68 e8 f7 0e 00
0000000100003740      5e 41 5f 5d c3 55 48 89 e5 41 57 41 56 41 55 41
0000000100003930      5c f0 ff ff 89 c3 48 8d 05 1b 1d 00 00 8b 08 85
0000000100003970      7c 04 85 c9 75 40 41 89 d4 89 c3 48 8d 05 b6 1c
0000000100003990      45 f8 e8 c3 0b 00 00 42 8d 04 2b 23 45 c8 44 89
00000001000039f0      83 c4 38 5b 41 5c 41 5d 41 5e 41 5f 5d c3 31 ff
0000000100003ac0      00 00 48 89 c3 8a 04 1a 88 45 d6 48 83 ca 01 48
0000000100003b00      f8 80 f9 30 75 36 83 c3 d0 41 89 1f 66 bb 01 00
0000000100003b40      9f 80 f9 07 77 08 83 c3 9f 41 89 1f eb 4e 89 c1
0000000100003b50      80 c1 bf 80 f9 07 77 12 83 c3 bf 41 89 1f 48 8b
0000000100003bd0      41 5d 41 5e 41 5f 5d c3 55 48 89 e5 41 56 53 41
0000000100003c30      c6 08 00 00 89 c7 44 89 f6 5b 41 5e 5d e9 e2 08
0000000100003c60      31 c0 48 83 c4 10 5d c3 55 48 89 e5 e8 e9 08 00
0000000100003c70      00 31 c0 5d c3 55 48 89 e5 41 56 53 89 f8 48 8d
0000000100003d10      5e 5d e9 b5 08 00 00 5b 41 5e 5d c3 55 48 89 e5
0000000100003e40      ff ff 4c 89 e6 4c 89 f9 e8 f5 06 00 00 48 89 c3
0000000100003e90      98 00 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 e8
```

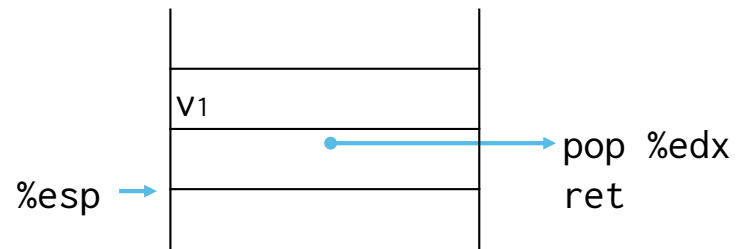
Return Oriented Programming (ROP)

- ROP idea: make shellcode out of **existing application code**.
- Stitching together arbitrary programs out of code gadgets already present in the target binary
 - **ROP Gadgets**: sets of code sequences ending in ret instruction.
 - Ret commonly added by compiler (at end of function)
 - But **also** (on x86) any sequence in executable memory ending in `0xC3` (ret).
 - x86 has variable-length instructions
 - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences
- Overwrite saved return address on stack to point to first gadget, the following word to point to second gadget, etc
- **Stack pointer is the new instruction pointer** in this crazy world

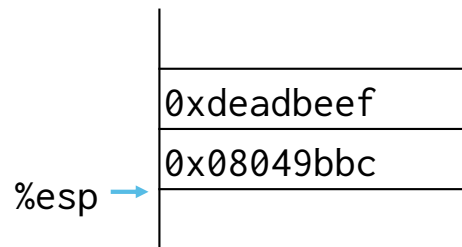
Return Oriented Programming

- *"Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake **arbitrary computation**."*
 - *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)* by Hovav Shacham
 - <https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf>
- Turing-complete computation.
 - Load and Store gadgets
 - Arithmetic and Logic gadgets
 - Control Flow gadgets

What does this gadget do?



relevant stack:



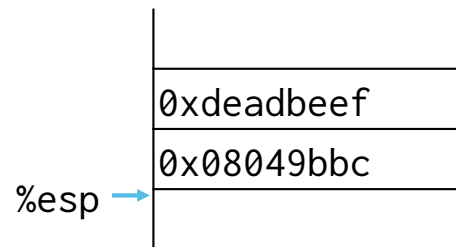
relevant register(s):

%edx = 0x00000000

relevant code:

```
%eip → 0x08049b62: nop
        0x08049b63: ret
        ...
        0x08049bbc: pop %edx
        0x08049bbd: ret
```

relevant stack:



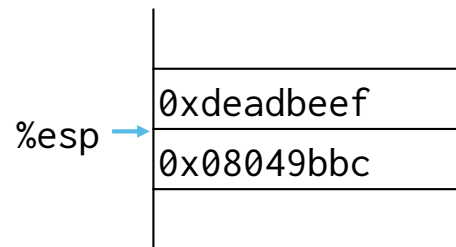
relevant register(s):

%edx = 0x00000000

relevant code:

```
0x08049b62: nop
%eip → 0x08049b63: ret
...
0x08049bbc: pop %edx
0x08049bbd: ret
```

relevant stack:



relevant register(s):

%edx = 0x00000000

relevant code:

0x08049b62: nop

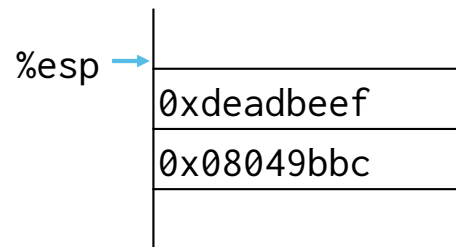
0x08049b63: ret

...

%eip → 0x08049bbc: pop %edx

0x08049bbd: ret

relevant stack:



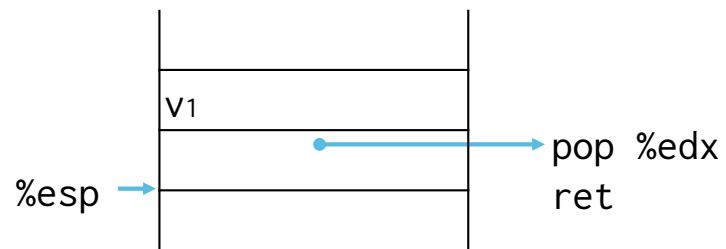
relevant register(s):

%edx = 0xdeadbeef

relevant code:

```
0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
%eip → 0x08049bbd: ret
```

What does this gadget do?

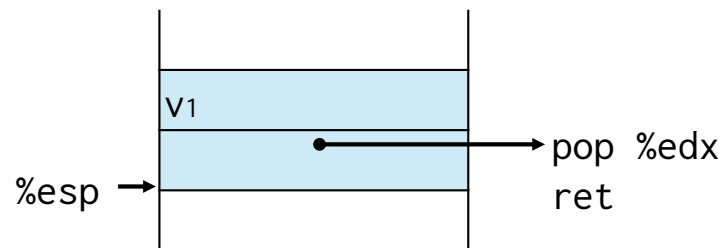


`%edx = v1`

`mov v1, %edx`

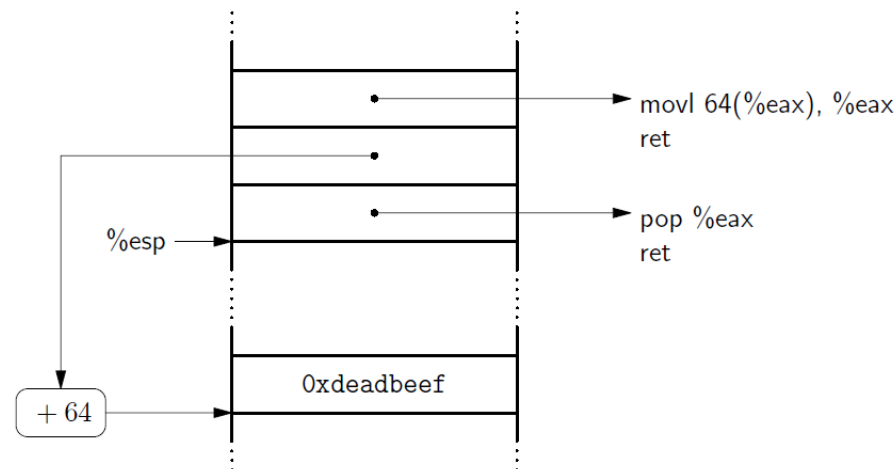
How do you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program
- E.g., if shellcode needs to write a value to `%edx`, use the previous gadget



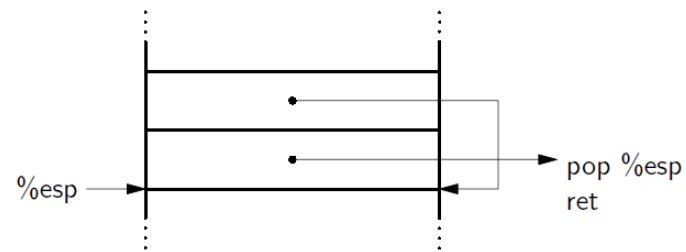
Return Oriented Programming

- Gadget for loading a value from memory
 - A bit more complex...
 - Attacker sets up stack so **address of value to be loaded** is on stack
 - Technically, 64 bytes less than addr
 - Return to gadget that pops that address into %eax
 - Return to gadget that loads the value based on address in %eax



Return Oriented Programming

- Control Flow Gadgets
 - Stack pointer is effectively the new instruction pointer (program counter)
 - To “jump” just pop a new value into %esp
 - Conditional jumps are more involved but still possible



Return Oriented Programming

- Stack pointer acts as instruction pointer
- Manually stitching gadgets together gets tricky
 - Automation to the rescue!
 - Gadget finder tools, ROP chain compilers, etc.
 - All of this has been quasi-automated
 - UCSD built first ROP compiler as grad class project in 2009
 - As of 2013 CIA had a working ROP compiler toolchain (Vault7 leaks)
 - Lots of compilers since for lots of architectures
- Note for most real attacks don't need this level of automation
 - Just do enough to get capability you need
- Also: not even really about "returns"...
other variants target other kinds of deterministic control flow (e.g. jump tables)
- Well, this sucks.... how to defend against this?

Control Flow Integrity

Control Flow Integrity

- Given a new attack technique, we must develop a new countermeasure.
 - Such is the cycle of life
- ***Control Flow Integrity (CFI)***
 - Motivation: in almost all attacks we've seen attacker is overwriting jump targets (e.g., return address on stack, function pointers on heap, etc.)
 - What if we ensured that rets, calls, etc... could only go to ***known good*** targets

Control Flow Integrity

- Focus is on protecting indirect transfer of control flow instructions.
- **Direct** control flow transfer:
 - Advancing to next sequential instruction
 - Jumping to (or calling a function at) an address hard-coded in the instruction
 - These are static in code, so assume attacker can't control them (if they can overwrite code segment its game over anyway)
- **Indirect** control flow transfer
 - Jumping to (or calling a function at) an address in register or memory
 - **Forward path**: indirect calls and branches (e.g., a function you are calling)
 - **Reverse path**: return addresses on the stack (returning from a called function)

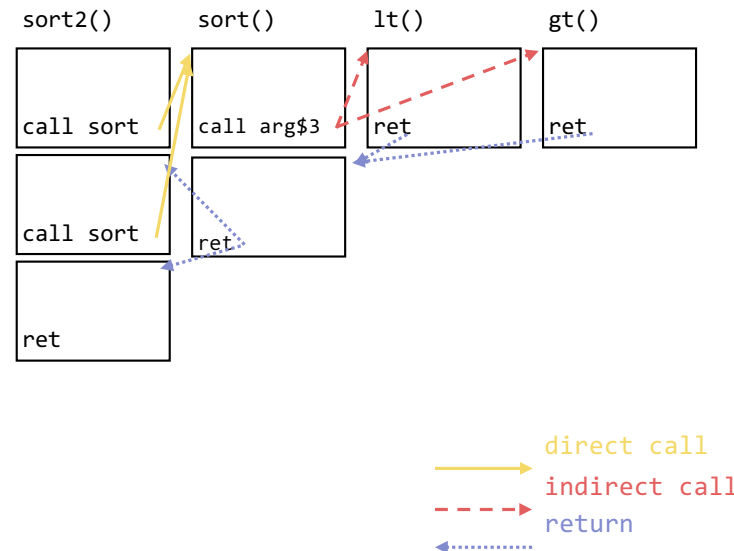
What's a legitimate target?

Look at the program control-flow graph (CFG)!

```
void sort2(int a[],int b[], int len {  
    sort(a, len, lt);  
    sort(b, len, gt);  
}
```

```
bool lt(int x, int y) {  
    return x < y;  
}
```

```
bool gt(int x, int y) {  
    return x > y;  
}
```



Control Flow Integrity

- Basic Design:
 - Restrict all control transfers to the control flow graph
 - Assign **labels** to all indirect jumps and their targets
 - Before taking an indirect jump, **validate** that target label matches jump site
 - **Like stack canaries, but for control flow targets**

CFI: One implementation of Labels

Original code

Opcode bytes	Source Instructions	Destination Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04 mov eax, [esp+4] ; dst

Instrumented code

B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

Fine grained CFI (Abadi et al.)

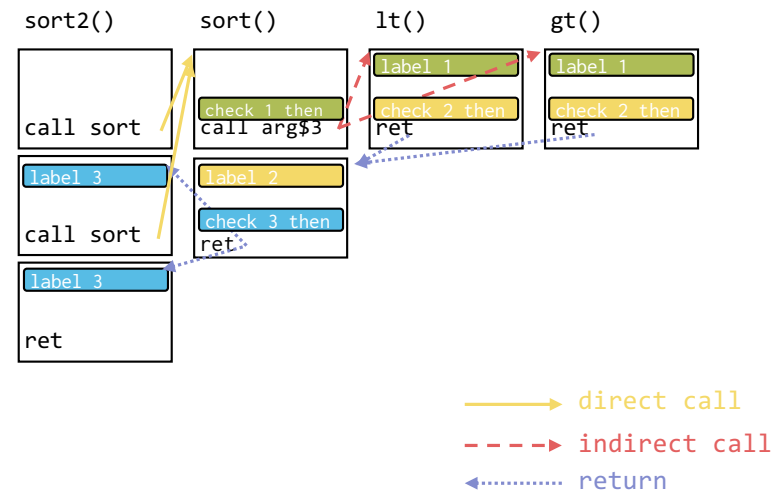
- Statically compute CFG
- Dynamically ensure program never deviates
 - Assign label to *each target* of **indirect** control transfer
 - Instrument *indirect control transfers* to compare label of destination with the expected label to ensure it's valid

Fine grained CFI (Abadi et al.)

```
void sort2(int a[],int b[], int len {  
    sort(a, len, lt);  
    sort(b, len, gt);  
}
```

```
bool lt(int x, int y) {  
    return x < y;  
}
```

```
bool gt(int x, int y) {  
    return x > y;  
}
```

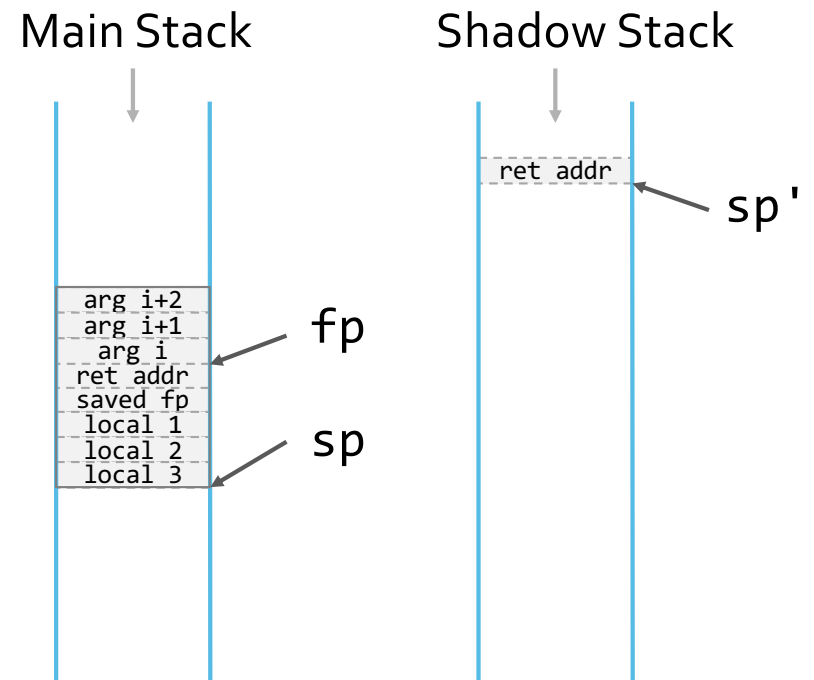


Fine grained CFI (Abadi et al.)

- Statically compute CFG
- Dynamically ensure program never deviates
 - Assign label to each target of indirect transfer
 - Instrument indirect transfers to compare label of destination with the expected label to ensure it's valid
- Note, returns are aliased in this example
 - Can return to multiple places, even if not “correct”
- Can do a bit better with a “shadow stack” (part of original CFI proposal)
 - Second, protected **stack just for control data**, to ensure that you are returning to function you called from

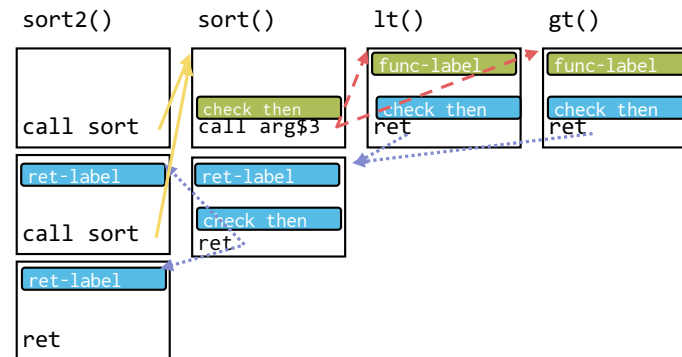
Shadow Stack

- Shadow Stack
 - On function entry, save a [shadow] copy of function call control flow data (return addresses and frame pointers) into another location
 - On function exit, compare the version on the stack to the shadow copy
- Requires compiler support
- Requires hardware support to be fast
 - Intel CET Shadow Stack starting with “Tiger Lake” processors
- Not yet widely deployed (just entered Linux release this year)



Coarse-grained CFI (e.g., bin-CFI)

- Tradeoff speed for precision
 - Identify if control transfer is *clearly wrong*, but not that it is right
 - i.e., can only transfer control to a legitimate destination in the CFG, but don't check which one
 - **Only two labels, no shadow stack**
- Label for destination of indirect calls (forward)
 - Make sure that every indirect call lands on a function entry
- Label for destination of rets and indirect jumps (reverse)
 - Make sure every indirect jump lands at start of a basic block



CFI Tradeoffs and Bypass Opportunities

- Overhead
 - Additional computation is needed before every free branch instruction.
 - Additional code size is needed before every free branch instruction and at each location (the label).
 - Hardware support needed for fine-grained CFI to have viable performance
- Scope
 - Data is not protected
 - CFI does not protect against interpreters
 - Needs reliable DEP (if you can change code all bets are off)
- Precision
 - If you don't validate **all** data-dependent control transfers, can still create gadgets
 - E.g., can still call `system()` with coarse grained CFI
 - Lots of potential gadgets on return path (you really need a Shadow Stack)
 - Performance/Precise tradeoff creates holes
 - *Out of Control: Overcoming Control-Flow Integrity*
 - *Stitching the Gadgets: On the Ineffectiveness of Coarse Grained CFI*
 - *Aside: C++ is a huge huge problem due to virtual functions – you can return to lots of different places – and that's enough*
- Some version of forward-path CFI is used today on both Apple iOS and Google Android

Summary

- Code reuse attacks bypass DEP by using existing code
 - E.g., Return to libc
- Return-oriented Programming
 - Generalizes idea. You can synthesize arbitrary malicious computation out benign code, by stitching it together with control flow
 - The stack is a convenient place to do this stitching
- Control-flow integrity
 - Promise to provide general-purpose protection against control flow vulnerabilities
 - Gets part of the way there at significant cost, but still bypassable due to limited precision

Next Lecture...

System Security

Guest Lecturer: Earlence Fernandes!