Return-to-libc and ROP Attacks

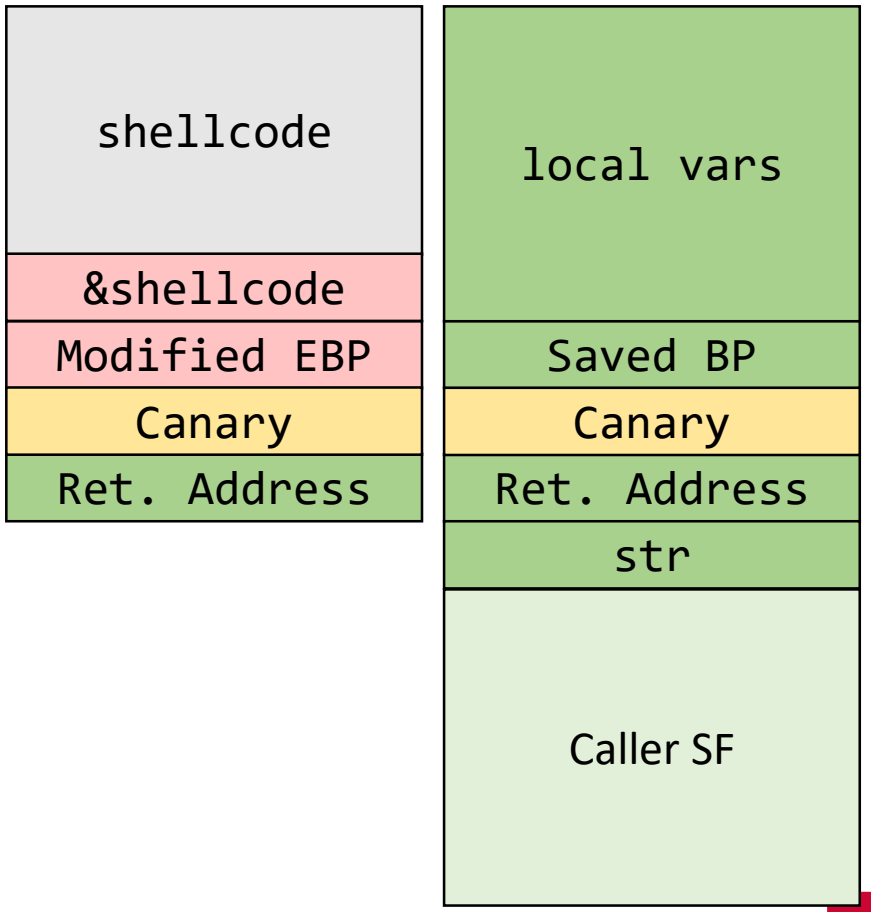
Instructor: Khaled Diab

Previous Lecture

- StackGuard, Shadow Stack ← We learned how to defeat these two 
- NOEXEC (W^X) ← Today, how we can defeat W^X. 
- ASLR

The Mistakes of StackGuard and Shadow Stack

- The attacker can modify local variables
  - Ones that are used in authentication
  - Function pointers
- The attacker can modify EBP
  - Frame pointer overwrite attack
  - EBP points to a fake frame inside the buffer
  - More details
- Assumes only the stack can be attacked!



NOEXEC (W^X)

- W^X → No single region is both **writable** and **executable**!
- Deployed in major OS
  - Linux
  - Windows
  - …
- Hardware Support
  - Intel: XD bit
  - AMD: NX bit
  - …

The Mistake of W^X

- Injecting code is the **only** way to hijack the control flow
- Makes sense… but is that true?

Today's Lecture

- To study attacks that do **not** rely on code injection
- The attacker controls the program flow by directing it to a different:
  - ***Function inside the program*** → Similar to Buffer overflow attack
  - ***Function inside libc*** → Return-to-libc Attack
  - ***Sequence of instructions*** → Return-oriented programming (ROP)

Function Re-use Attack

```
void bad() {
     system("/bin/sh");
}

int fn(char* str) {
     char* buffer[48];
     strcpy(buffer, str);
     return 1;
}
```

```
$ gcc jmp_to_fn.c -o jmp_to_fn
-fno-stack-protector -m32
```

Check if the stack is not executable…

```
$ readelf –l jmp_to_fn
Elf file type is EXEC (Executable file)
Entry point 0x80483f0
There are 9 program headers, starting at offset 52
…
GNU_STACK      0x000000 0x00000000 0x00000000
0x00000 0x00000 RW  0x10
…
```

Function Re-use Attack

- Checking bad address

```
$ objdump -d jmp_to_fn | grep bad
080484eb <bad>:
```

- Use it as the return address:

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000030  90 90 90 90 90 90 90 90  90 90 90 90 eb 84 04 08  |................|
```

Function Re-use Limitations

- The vulnerable program doesn’t have this function!

`libc`

- A library for C standard
- Implementing many functions:
  - String manipulation
  - IO
  - Memory
  - …
- We use it almost in every program!
  - `<std*.h>`
  - Check your program using `ldd`

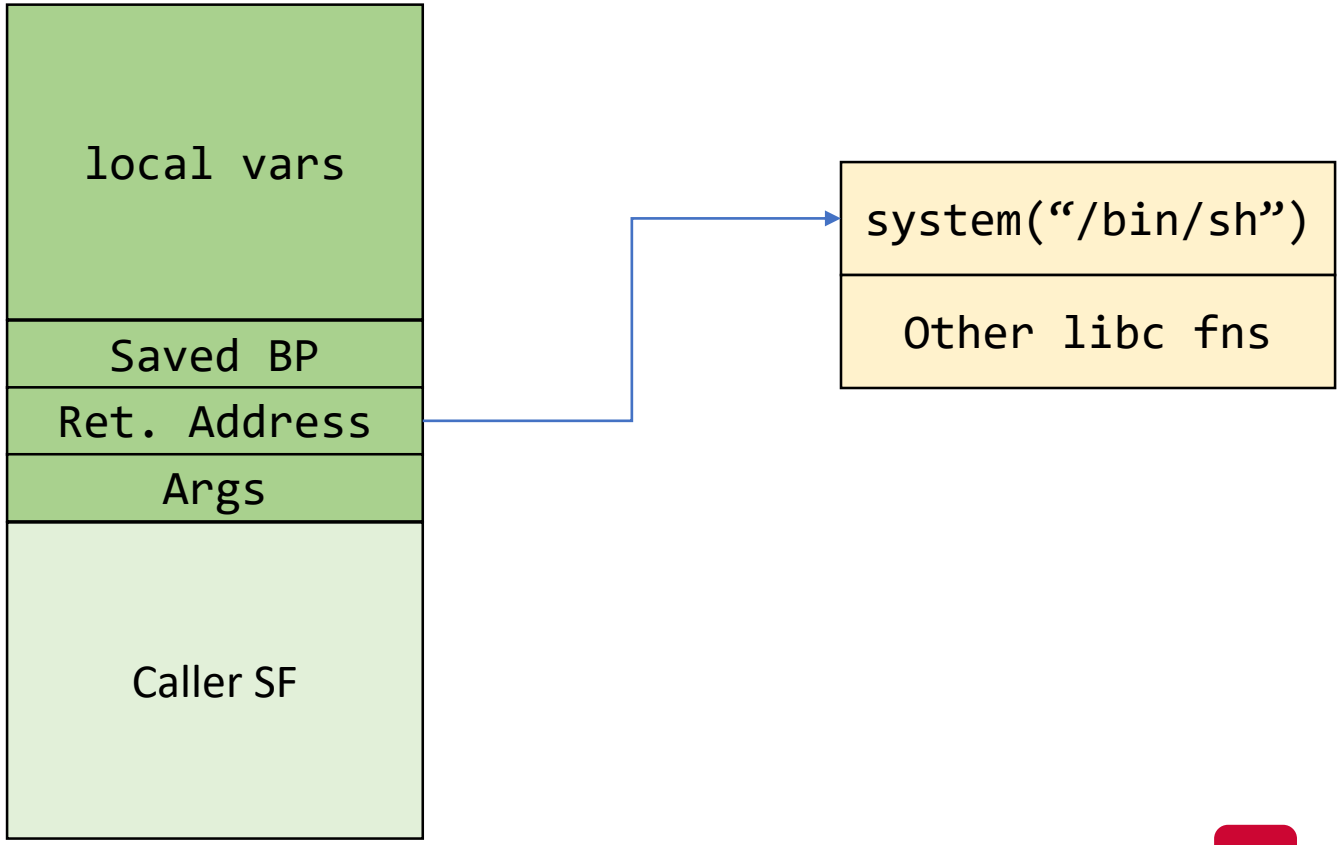
Return-to-libc [Solar Designer ‘97]

- Overwrite the return address to an address of a function in `libc`
  - Instead of relying on the program functions!

```
int fn(char* str) {
    char* buffer[48];
    strcpy(buffer, str);
    return 1;
}
```

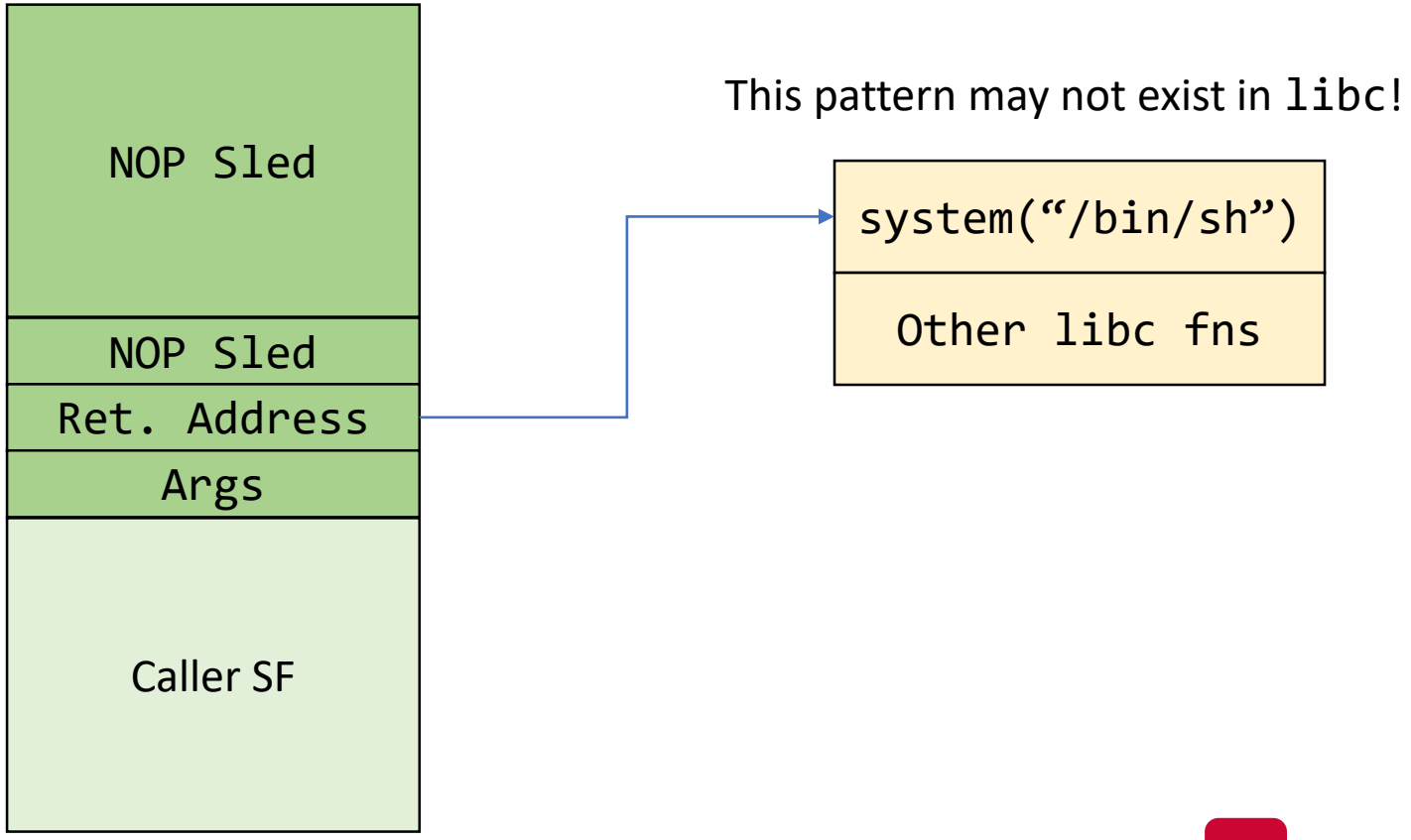
Return-to-libc

- Overwrite the return address to an address of a function in `libc`
  - Instead of relying on the program functions!



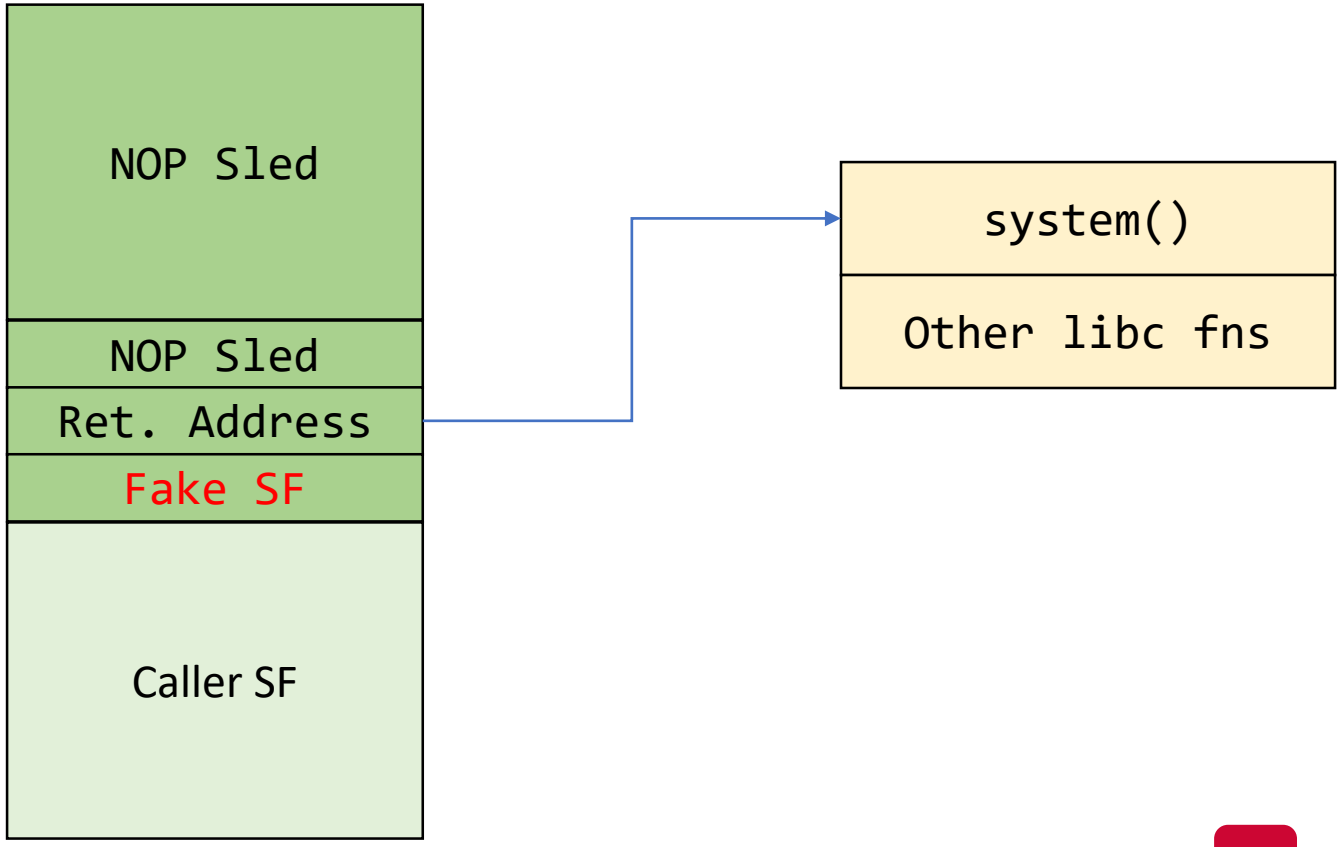
Return-to-libc: First Attempt

- Can we find the pattern `system("/bin/sh")`?
  - The attacker may not be lucky!



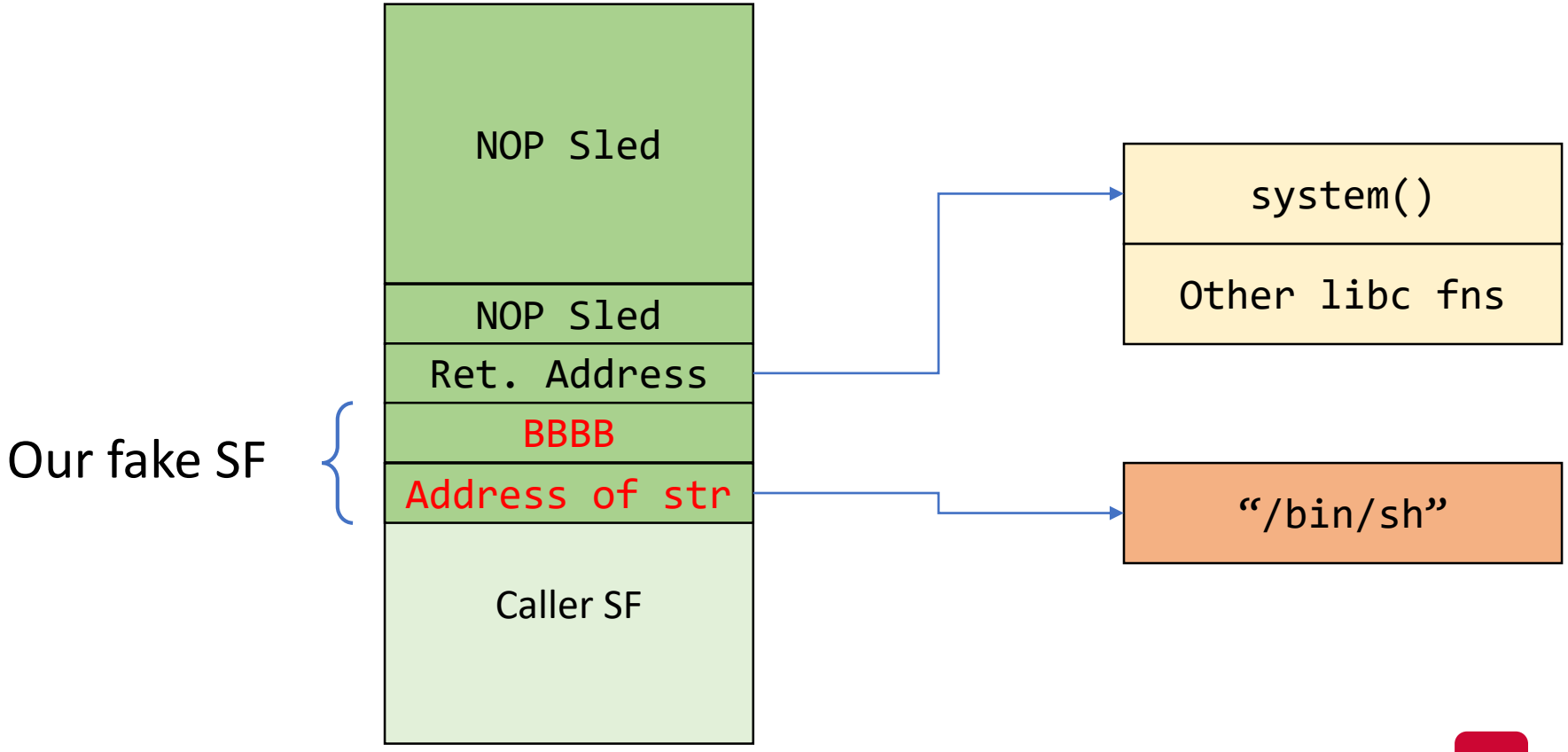
Return-to-libc: Second Attempt

- We need to construct a Fake SF for our attack!
- How would it look?



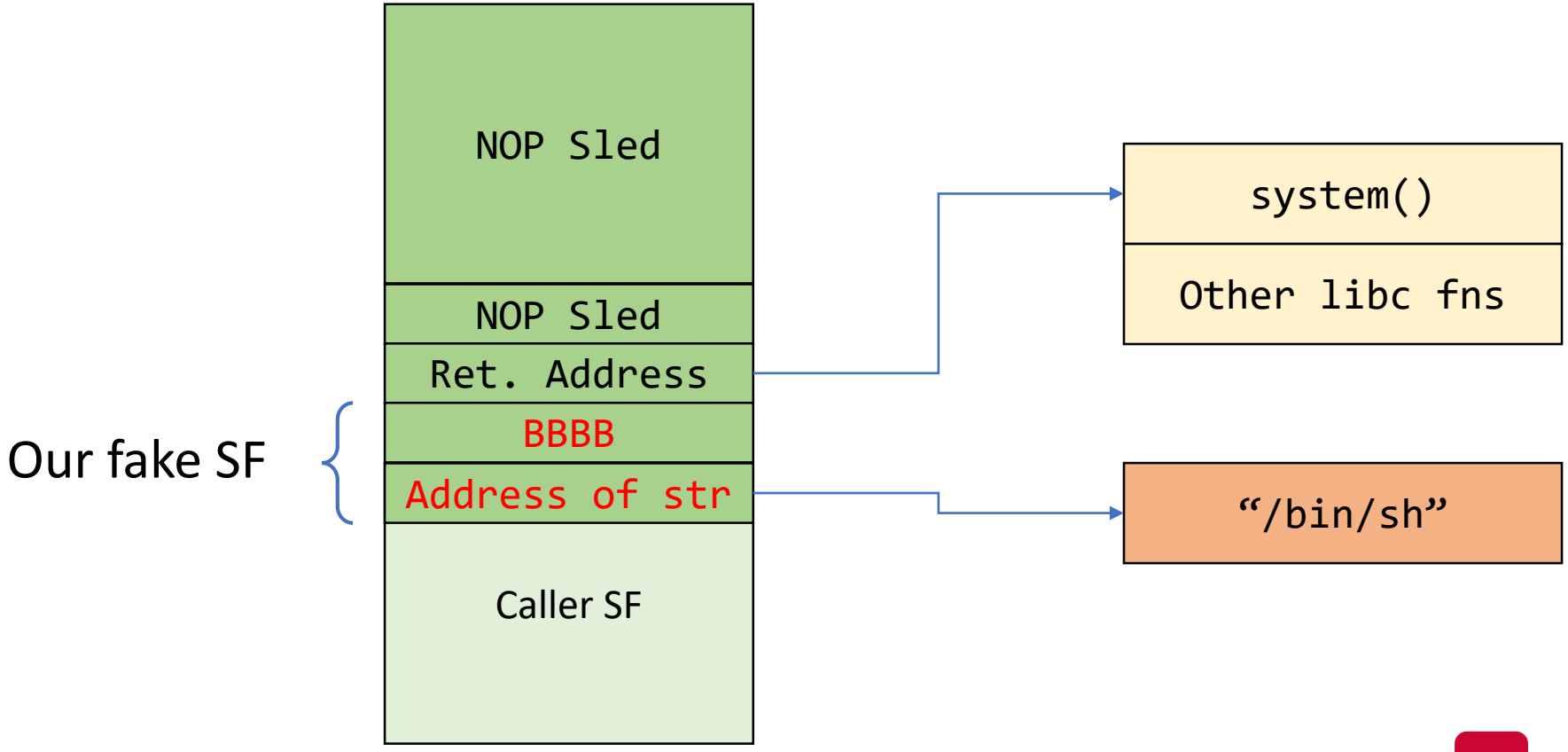
Return-to-libc: Second Attempt

- We need to construct a Fake SF for our attack!
- How would it look?



Return-to-libc: Second Attempt

- How can we find the string address “bin/sh”?
- Keep it in an env. var!



Return-to-libc: Steps

- Store “/bin/sh” in an env. variable
  - `export SHELL=“/bin/sh”`
- Find the address of system
- Find the address of the env. variable

Address of `system`

- Use `gdb` (after running the program and break at `main`)

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0
<__libc_system>
```

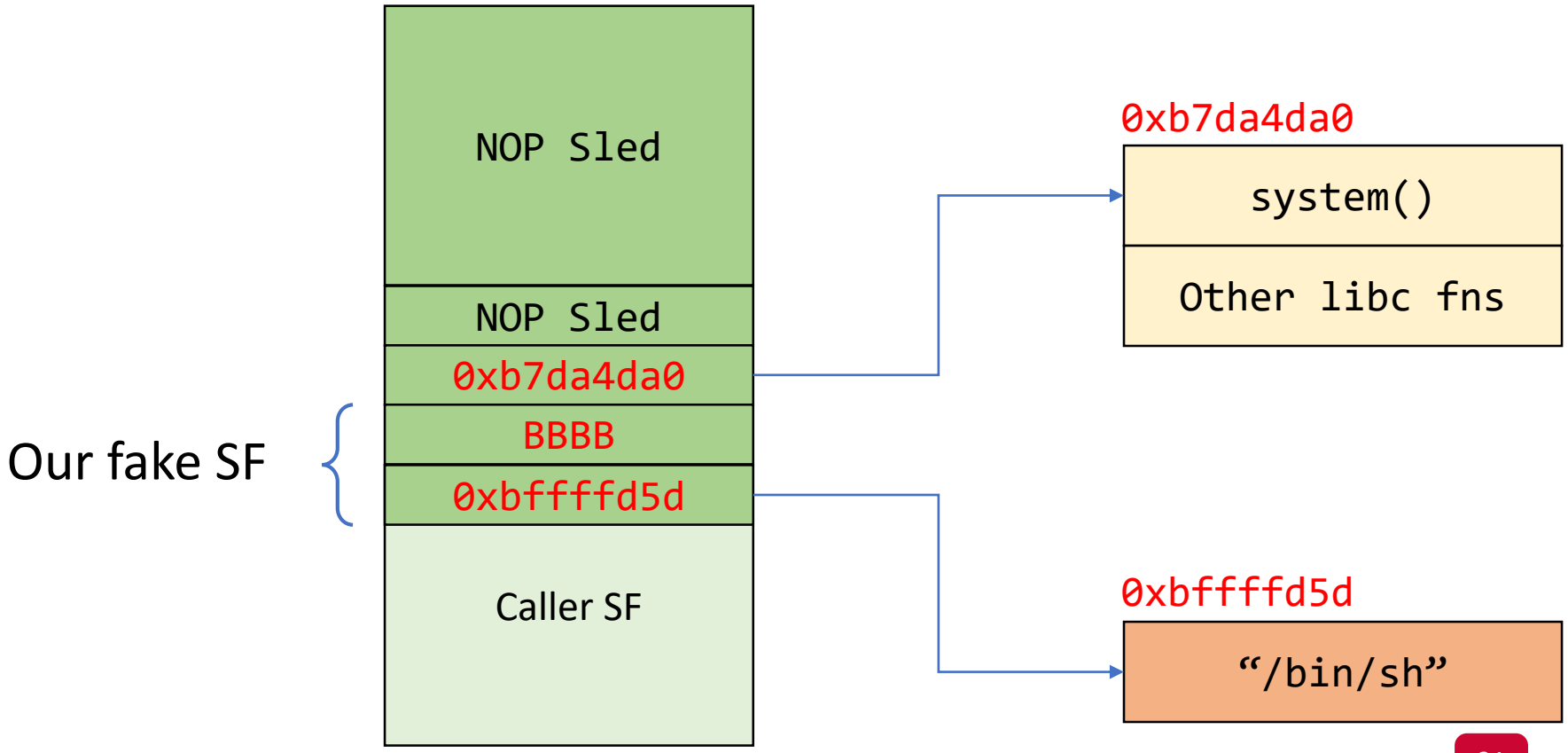
Address of "/bin/sh"

- Use gdb (after running the program and break at main)
- Print few strings from the stack

```
gdb-peda$ x/300s $esp
0xbffffd57:SHELL=/bin/sh
```

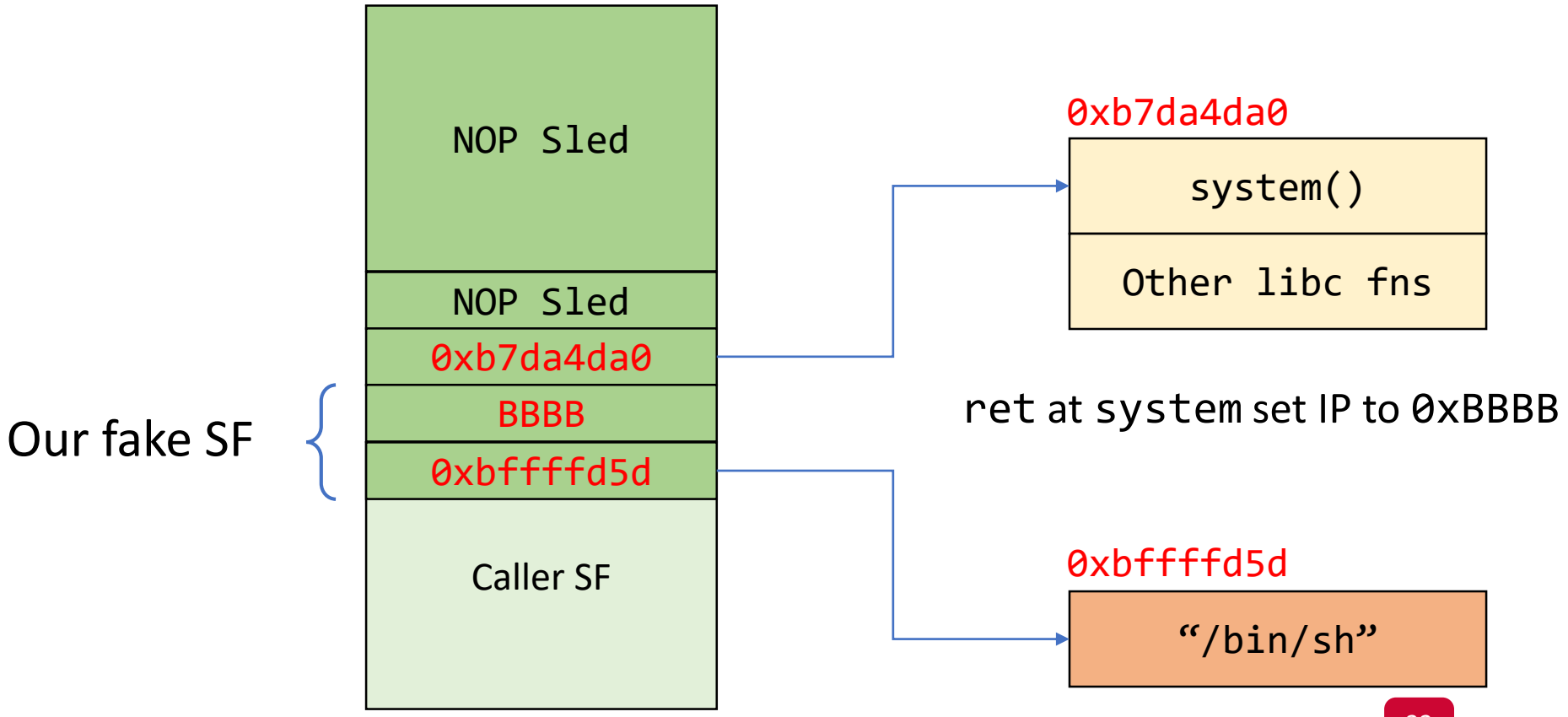
```
Address of the string = 0xbffffd57 + 6
                      = 0xbffffd5d
```

Return-to-libc: Our Stack



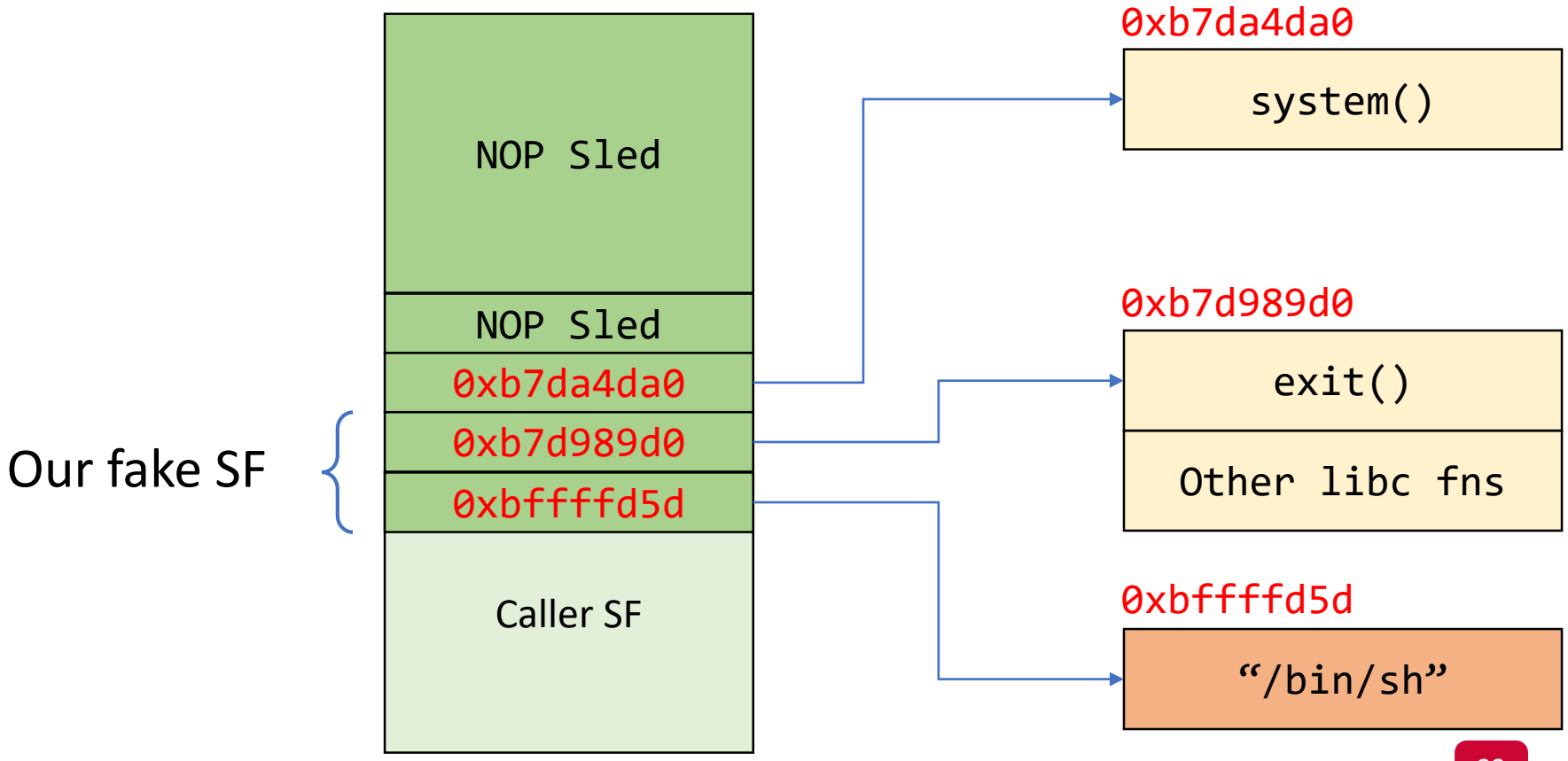
Return-to-libc: Our Stack

- SIGSEGV on exit…
- How can we fix this issue?



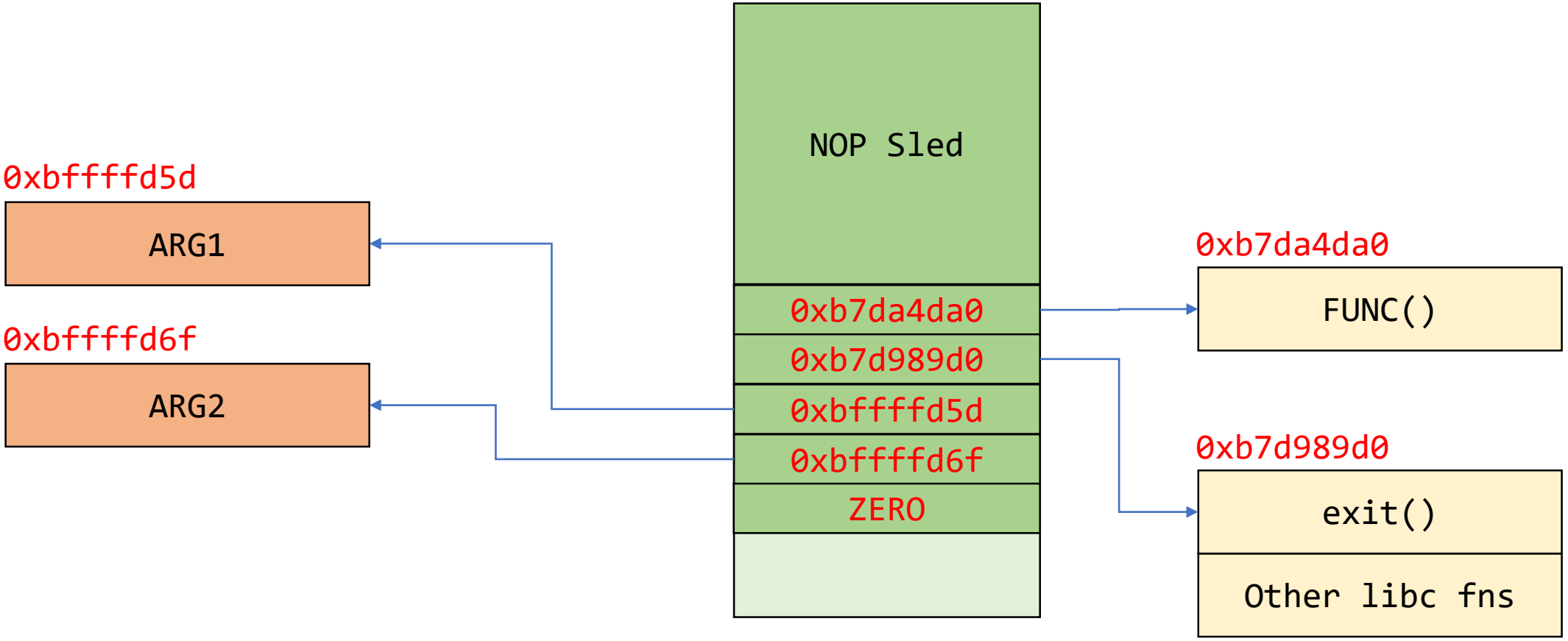
Return-to-libc: Our Stack

- The return address of `system` need to point to `exit`



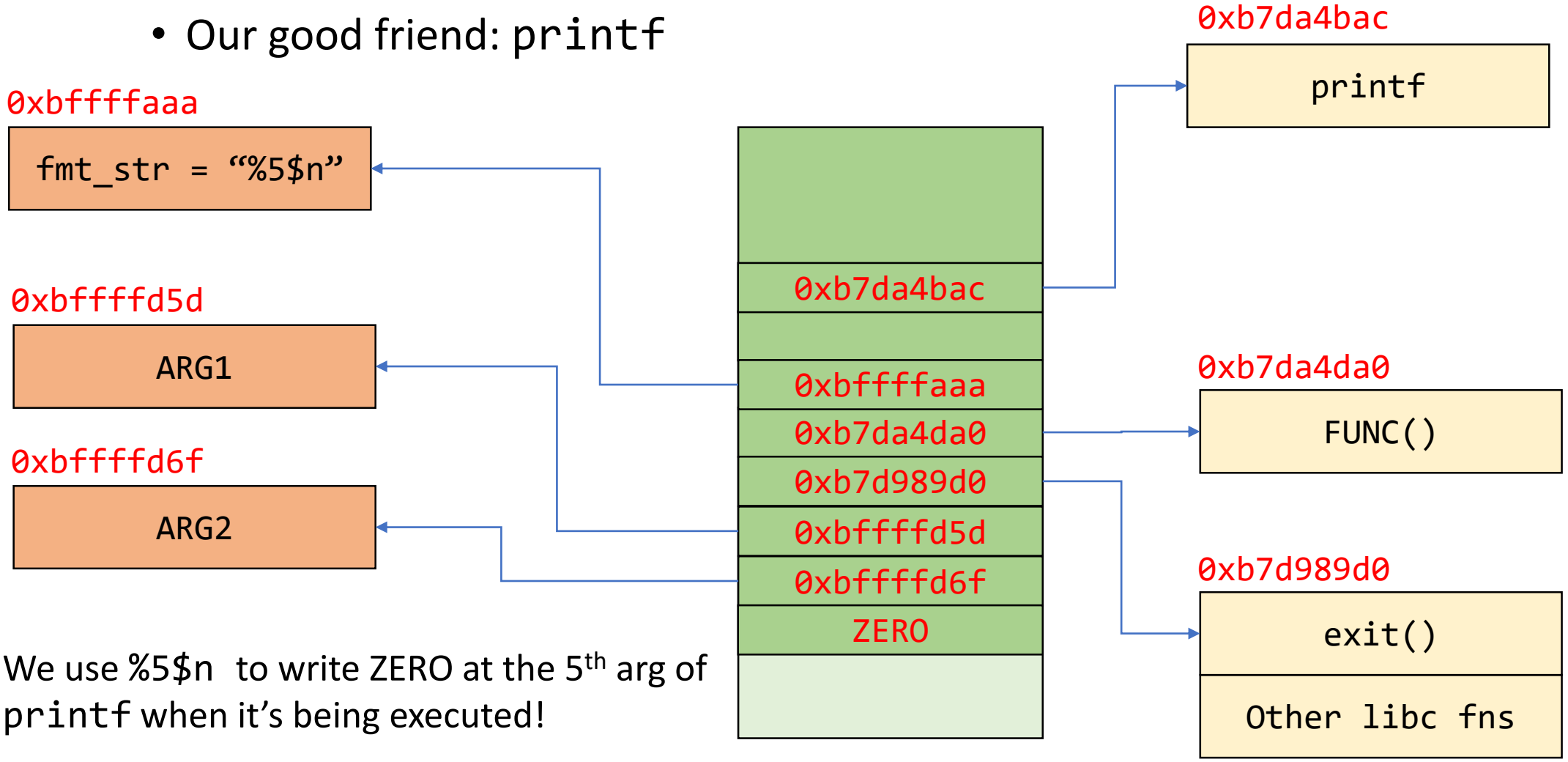
Return-to-libc: Injecting NULL Bytes

- Assume we want to call a function FUNC that takes three arguments
  - The third argument is NULL
  - How can we do it?



Return-to-libc: Injecting NULL Bytes

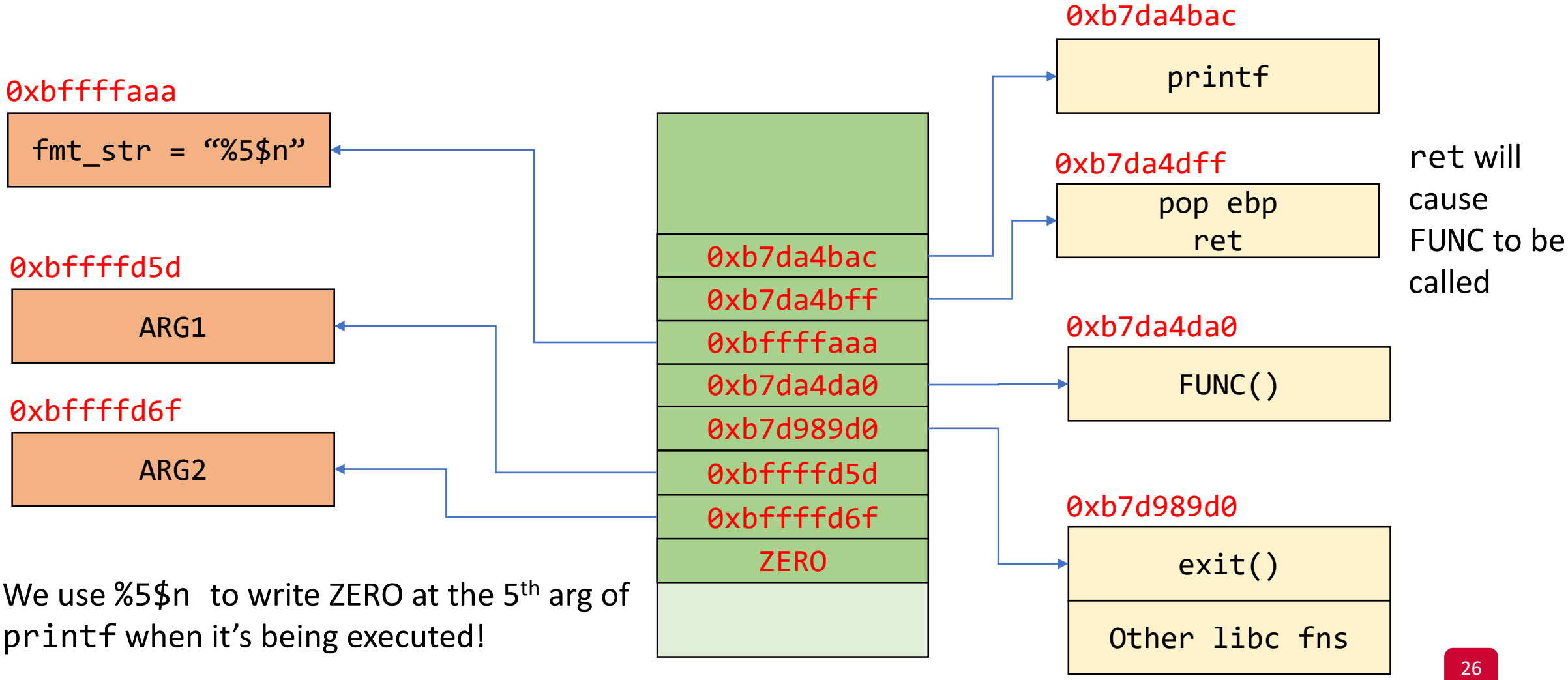
- How can we write a specific value to a specific address on the stack?
  - Our good friend: `printf`



We use %5$n to write ZERO at the 5th arg of `printf` when it's being executed!

Return-to-libc: Injecting NULL Bytes

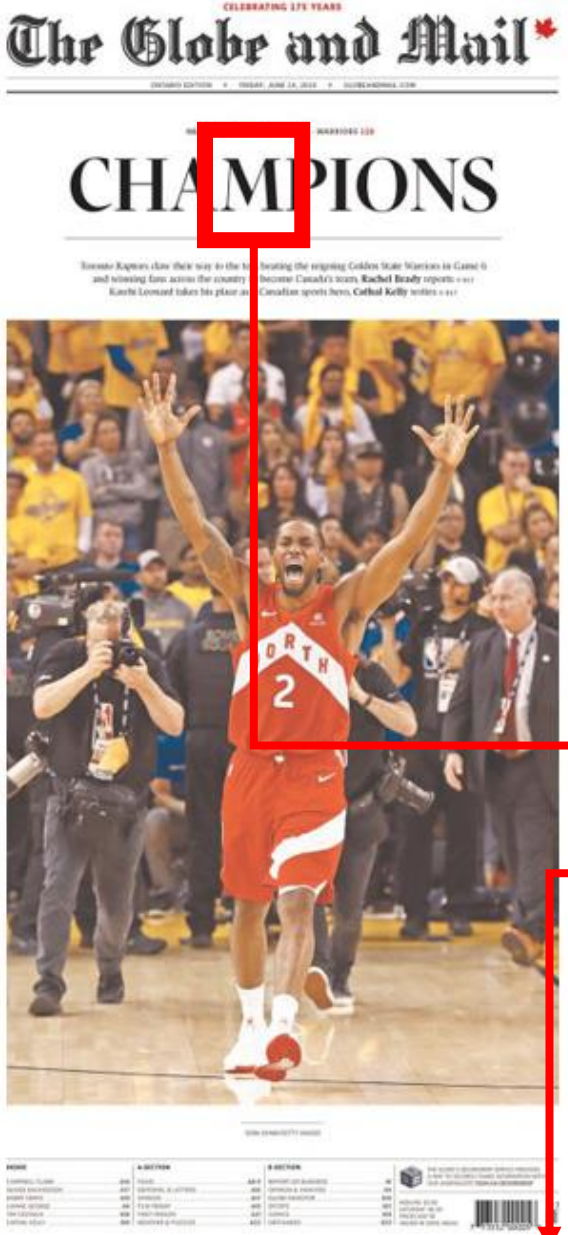
- What is the return address after `printf`?



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

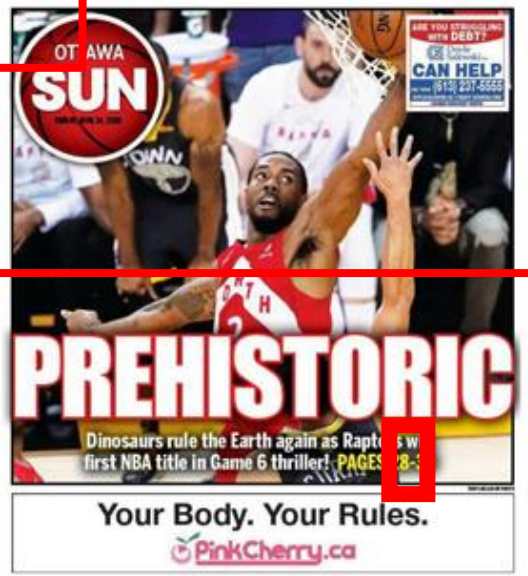
Return-oriented Programming (ROP)


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
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
HOMER BUG BITES OTTAVINO


OTTAWA SUN
PREHISTORIC
Dinosaurs rule the Earth again as Raptors win first NBA title in Game 6 thriller!
Your Body. Your Rules.
PinkCherry.ca


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

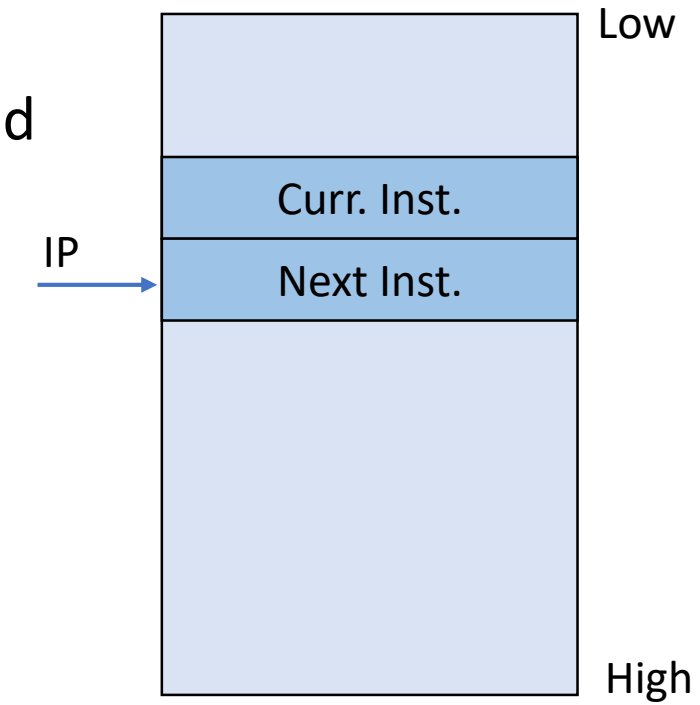
C MP T 4 7 9 9 8 0

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

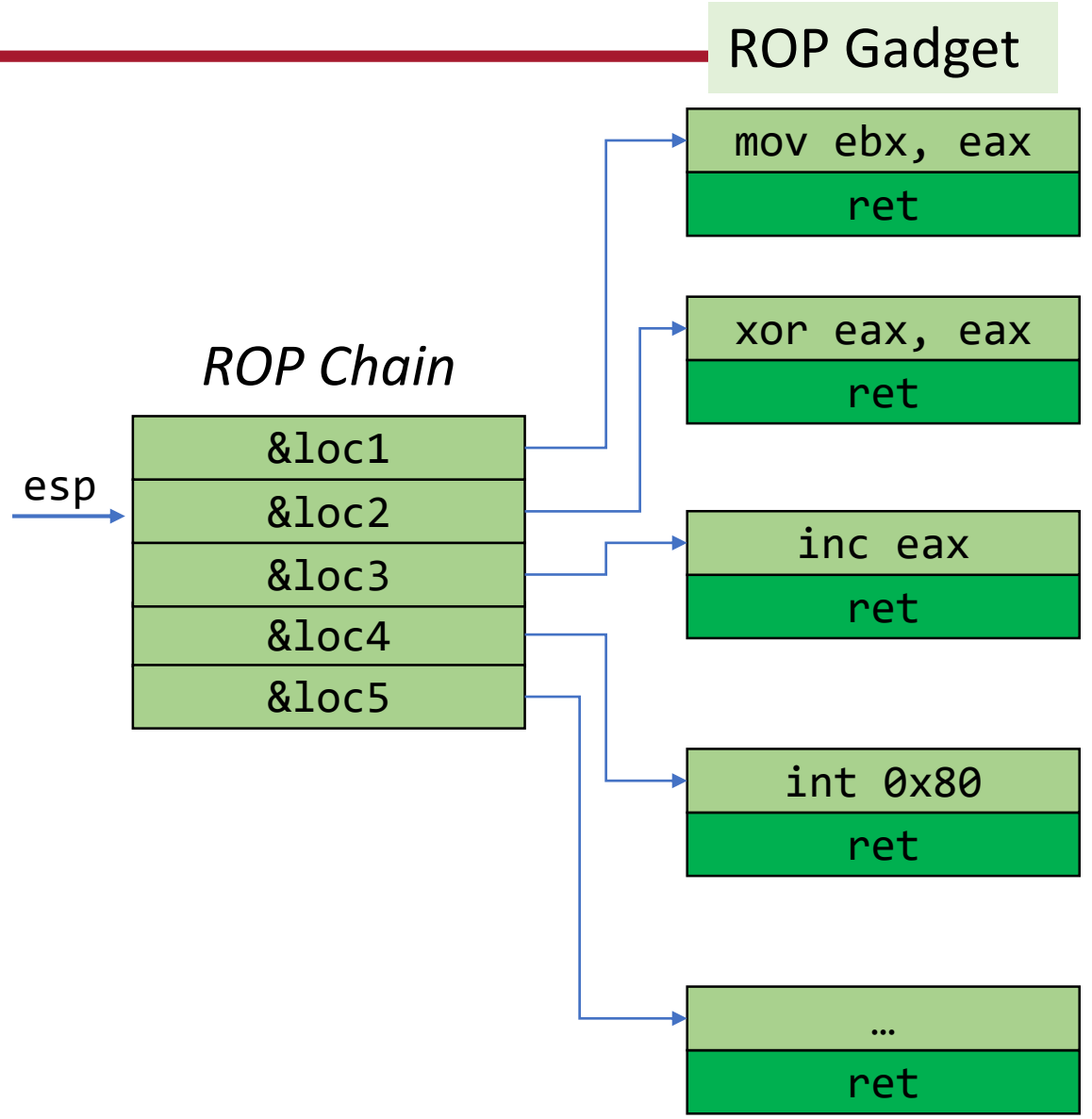
- A special register called IP:
  - Points to the next instruction to be fetched and executed
- Automatically incremented
- If we change IP → we change the program flow!



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
- An 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
```

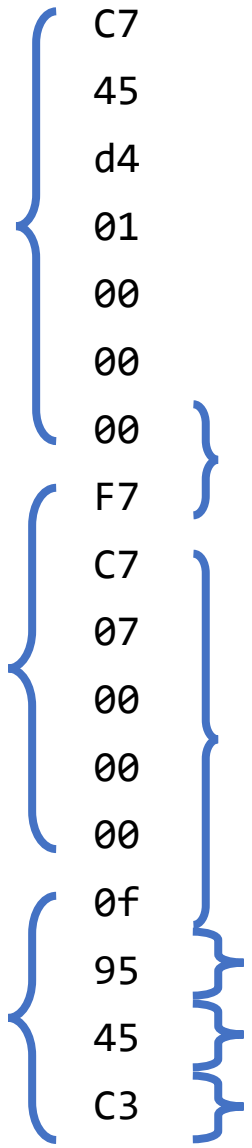
- They are *unintended*
  - The assembler/compiler didn't mean to put them this way

Unintended ROP Gadgets: Example

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

```
test edi, 0x00000007
```

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



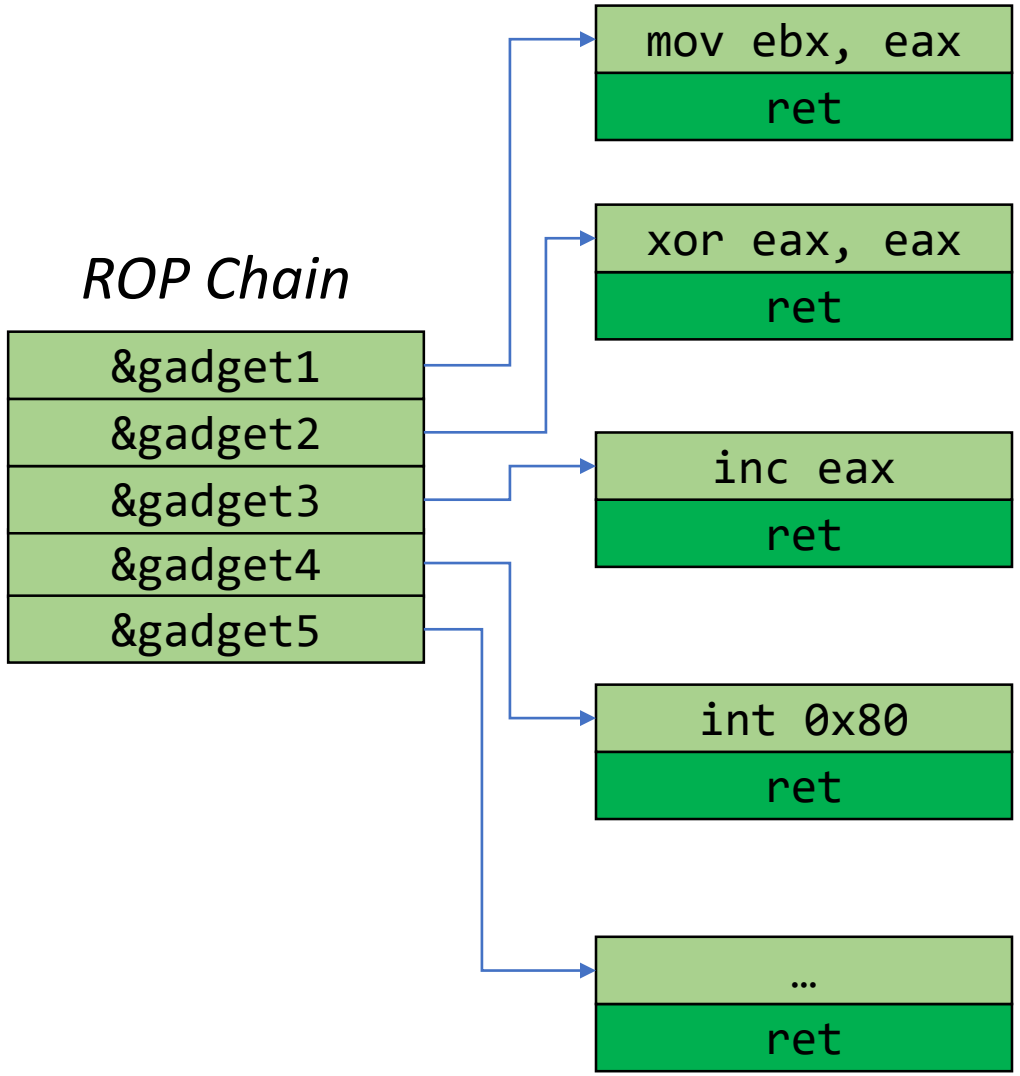
A new Gagdet!

```
add bh, dh

mov edi, 0x0f000000

xchg eax, ebx
inc ebp
ret
```

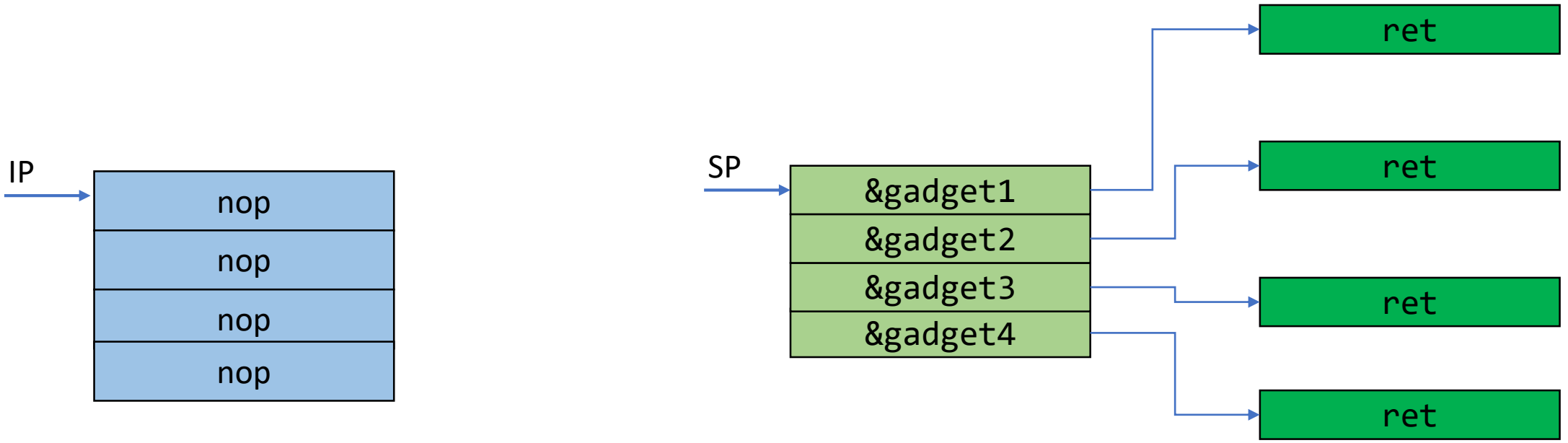
Start the Attack



Start the Attack

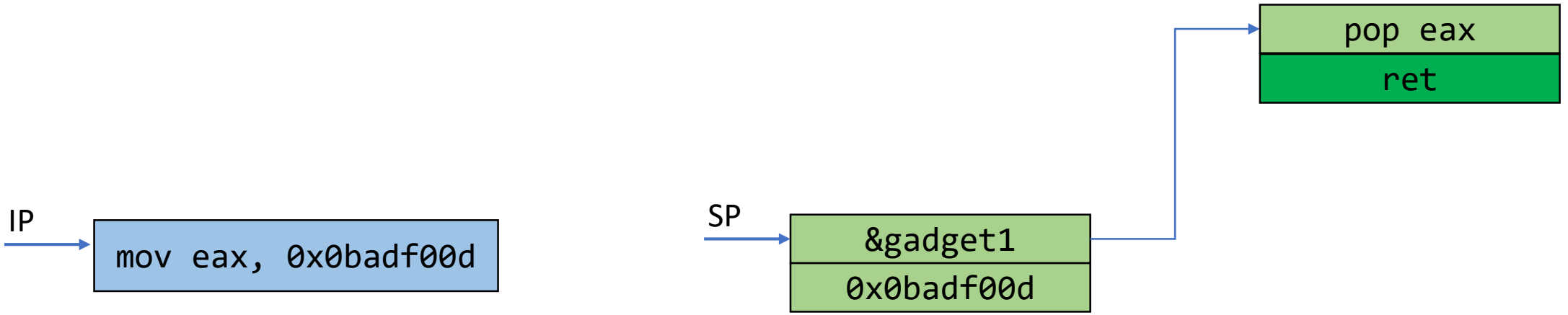
- We need to control `esp`
- Rewrite the Stack:
  - How?
- Move the Stack
  - Recall: the Frame Pointer overwrite attack!

Example 1: NOP

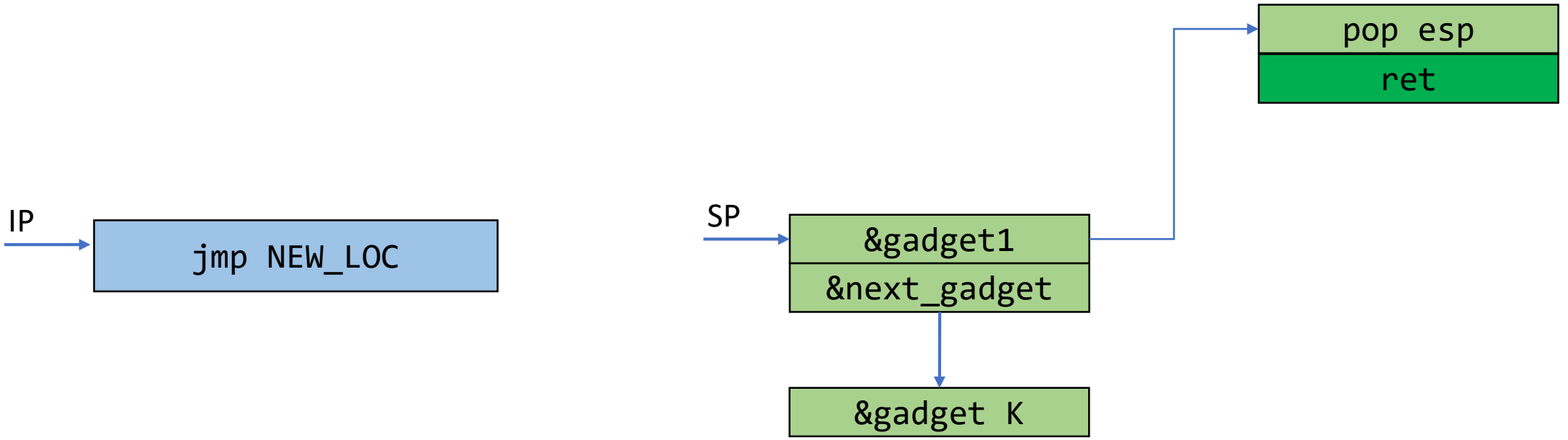


When the first inst. is being executed, SP points to the next 4 bytes.

Example 2: Load a Constant

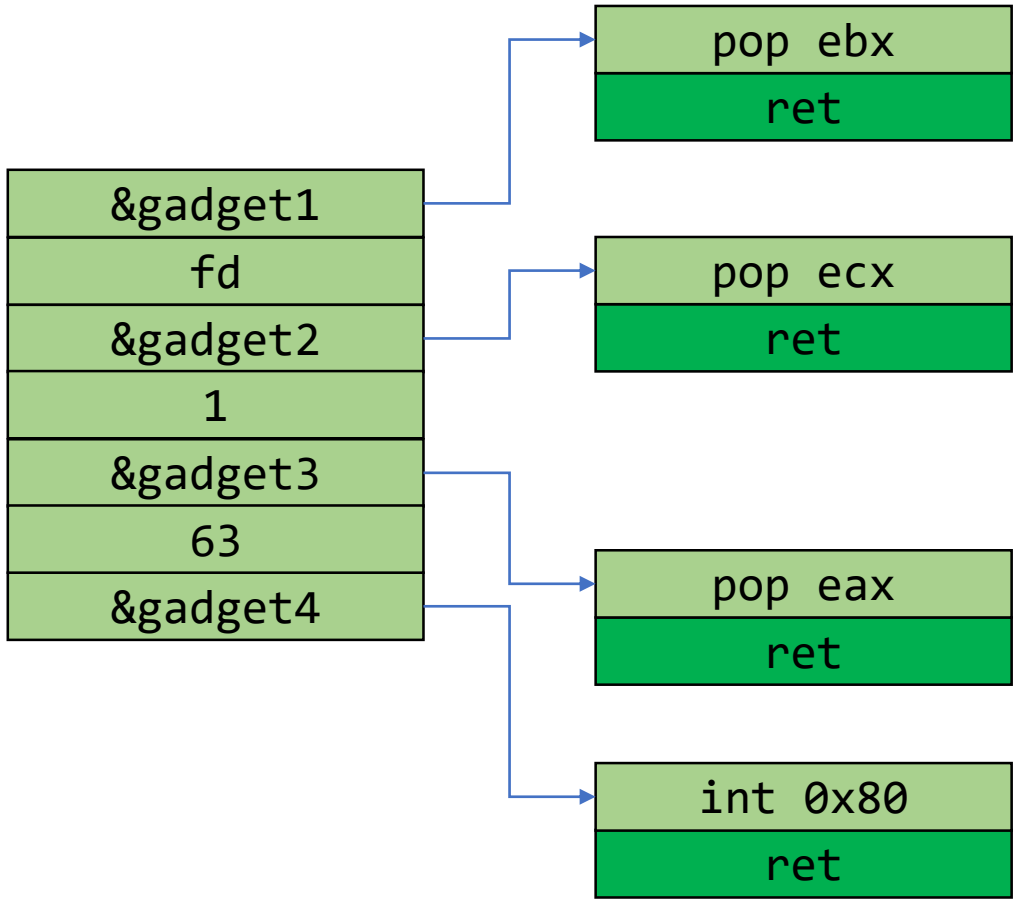


Example 3: Control Flow



ROP Chain: Example

- `A syscall: dup2`
- To duplicate the stdout



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 `c3` 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
```

Automated Gadget Hunting

- ROPGadget…

```
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 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?

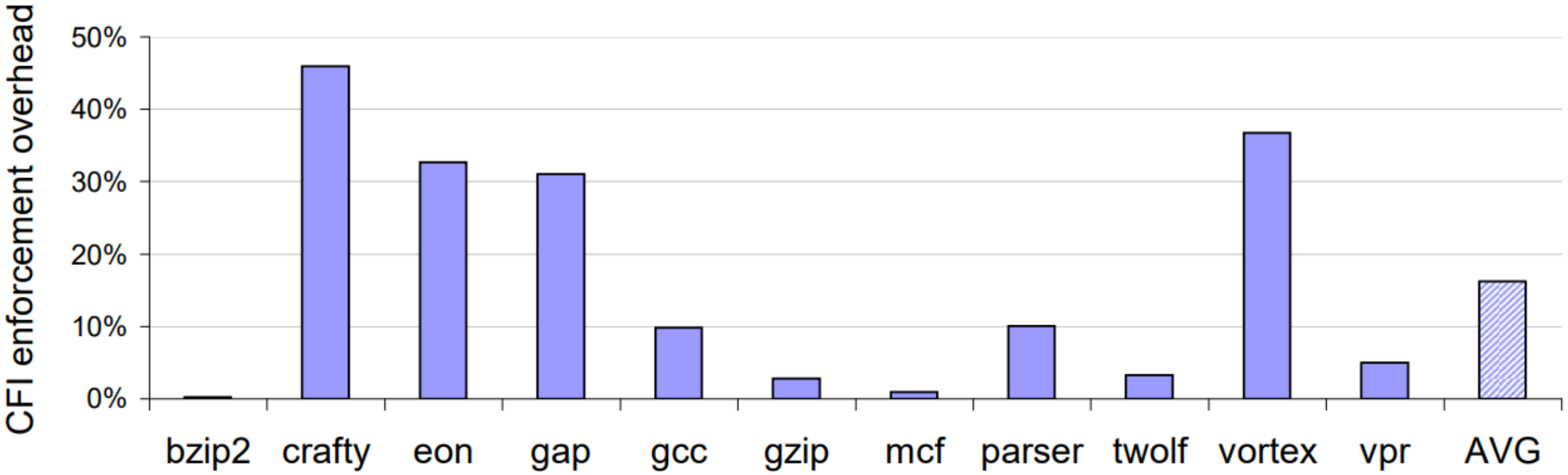
- No
  - x86, x86_64, Mips, Mips64, ARM, ARM64, SPARC, PowerPC, PowerPC64

ROP Defenses

- Control Flow Integrity (CFI)
- At compile time → Build a control-flow graph (CFG)
  - Reflects developer code
  - Lists all possible call targets
  - Can be done by static-code analysis, profiling etc.
- 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.

Next lecture

- OS Security