Binary Exploitation

yuawn

About

- yuawn
- Pwn
- Balsn / DoubleSigma

_yuawn

Outline

- ROP
- ret2plt
- ret2libc
  - Information leak
- Stack Pivoting
- ROP 萬解 - ret2csu

ROP

Return Oriented Programming Attack

ROP gadgets

ROP gadget

- 片段可執行的 code
  - 結尾是 ret instruction
  - call, jmp ... 等任何可以繼續控制流程的方式
- How to find the gadgets?
  - https://github.com/JonathanSalwan/ROPgadget
  - https://github.com/sashs/Ropper
  - 手動找

ROP

- NX - 見招拆招
- 在既有的執行區域 (code segment) 尋找 gadgets，運用這些 gadgets 疊成一長串的 return address chain (ROP chain)。
- 透過許多片段執行行為的 gadget，來串出任意代碼執行，藉此繞過 NX 保護機制的限制。
- Function return 時，會拿走第一個 return address，此時 rsp 指在第二個 return address，並跳至第一個 return address 執行，接著會執行到 ret instruction，而第二個 return address 也在開始時放置好，做到繼續 control flow，如此反覆來達到 Return Oriented Programming 型式的攻擊。

ROP

- Overflow
- NX - ~~Shellcode~~

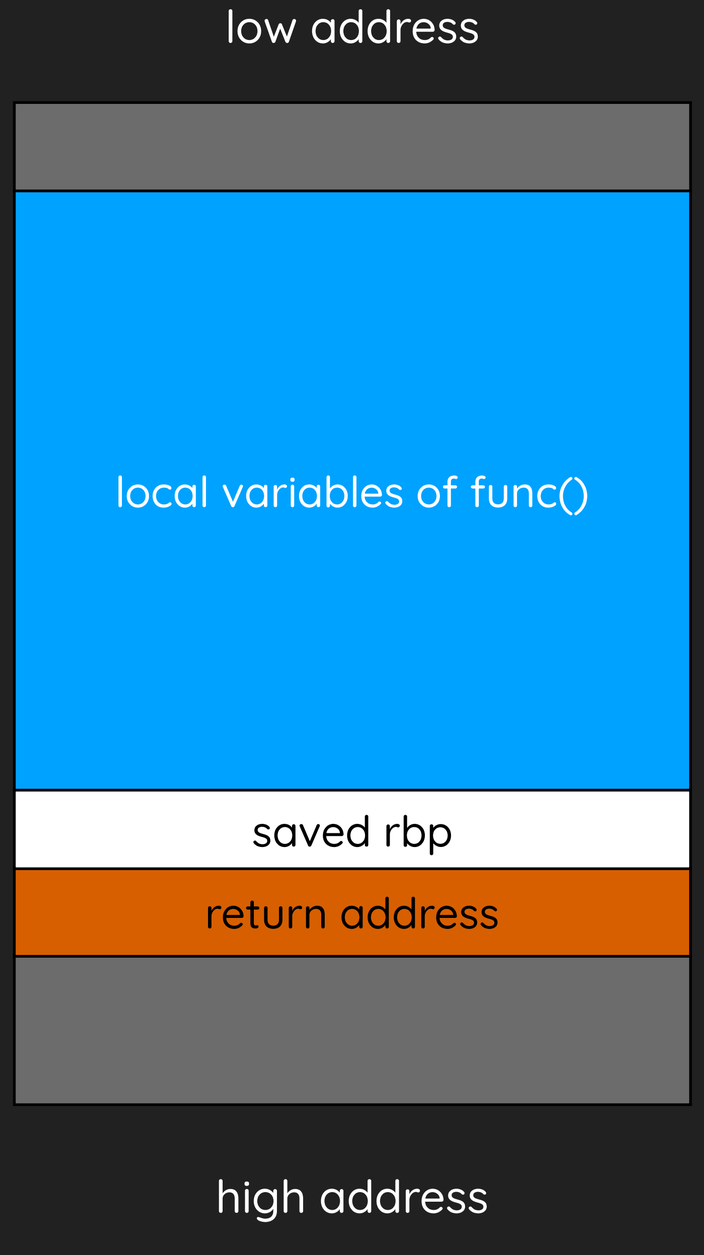
```
mov rax, 100
mov rbx, 66
add rax, rbx
```

```
        ...
Gadget 2:
        mov rbx, 66
        ret

        ...

Gadget 3:
        add rax, rbx
        ret
        ...
main:
        ...

        leave
        ret
        ...
Gadget 1:
        mov rax, 100
        ret
        ...
```

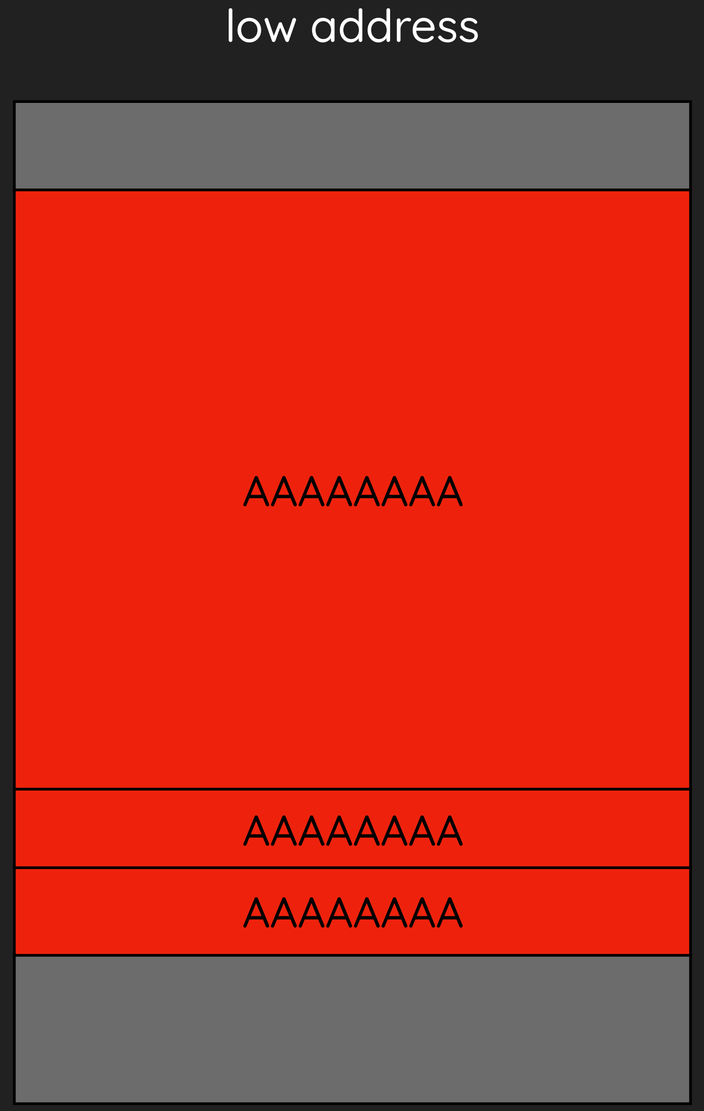


```
        ...
Gadget 2:
        mov rbx, 66
        ret

        ...

Gadget 3:
        add rax, rbx
        ret
        ...
main:
        ...

        leave
        ret
        ...
Gadget 1:
        mov rax, 100
        ret
        ...
```

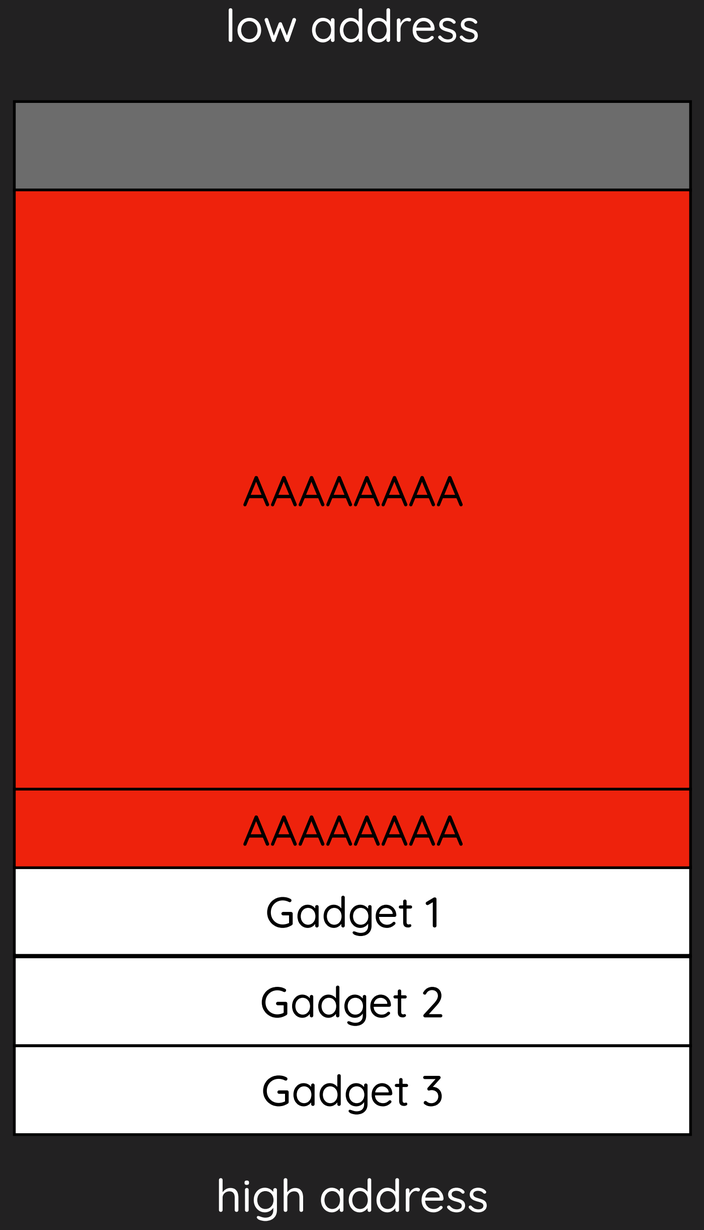


```
        ...
Gadget 2:
        mov rbx, 66
        ret

        ...

Gadget 3:
        add rax, rbx
        ret
        ...
main:
        ...

        leave
        ret
        ...
Gadget 1:
        mov rax, 100
        ret
        ...
```

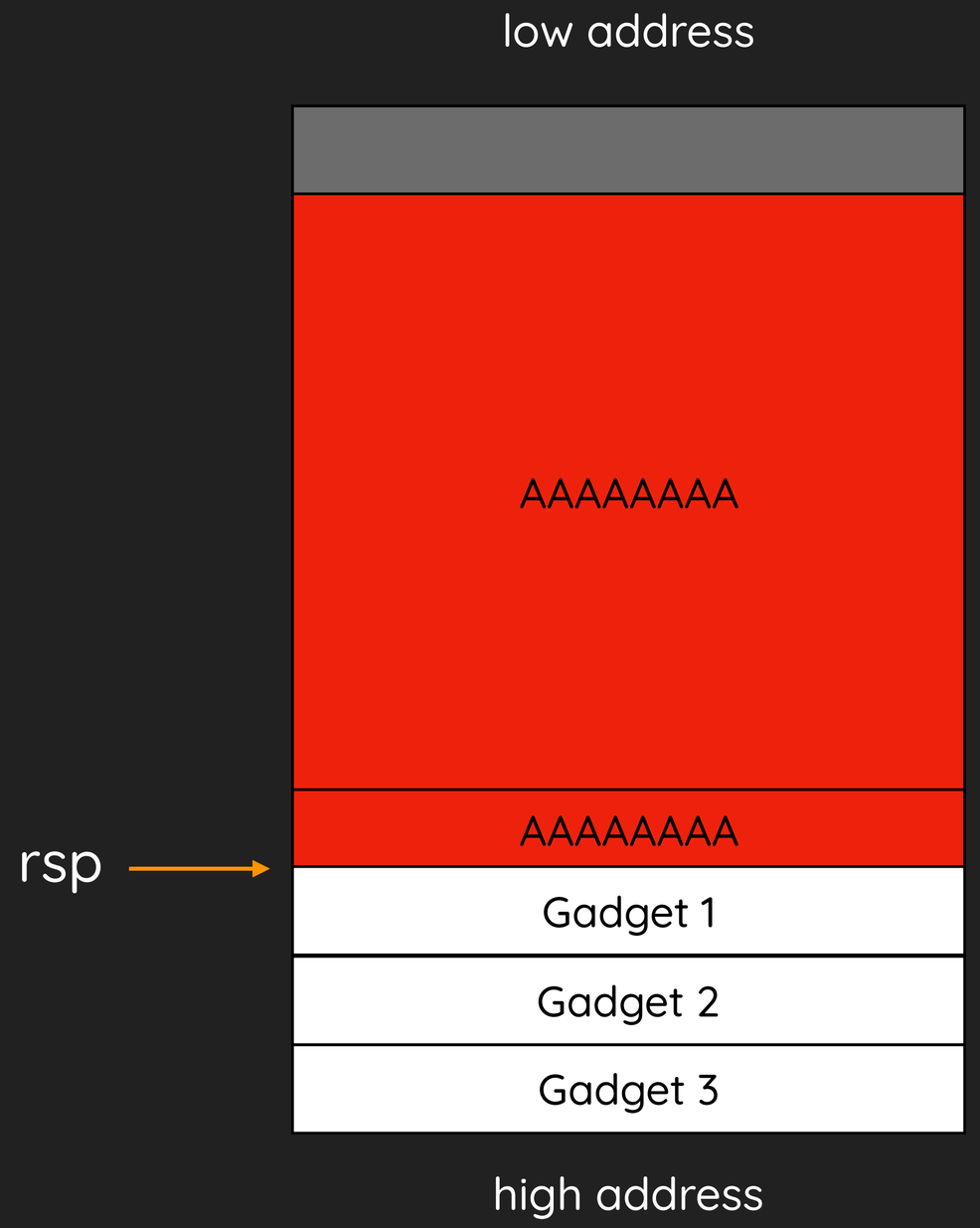


```
        ...
Gadget 2:
        mov rbx, 66
        ret

        ...

Gadget 3:
        add rax, rbx
        ret
        ...
main:
        ...

        leave
rip →   ret
        ...
Gadget 1:
        mov rax, 100
        ret
        ...
```

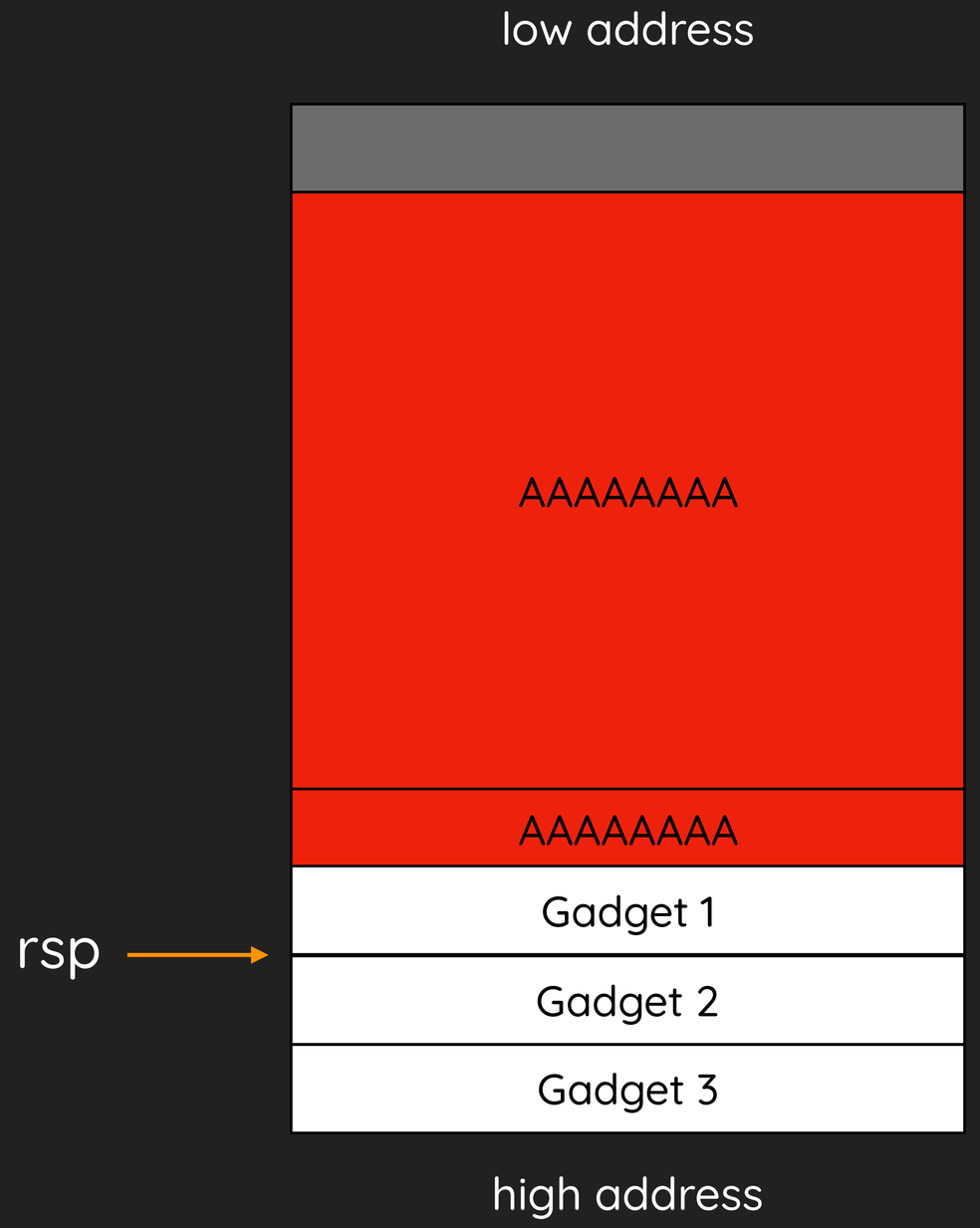


```
            ...
Gadget 2:
            mov rbx, 66
            ret

            ...

Gadget 3:
            add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
rip  →      mov rax, 100
            ret
            ...
```



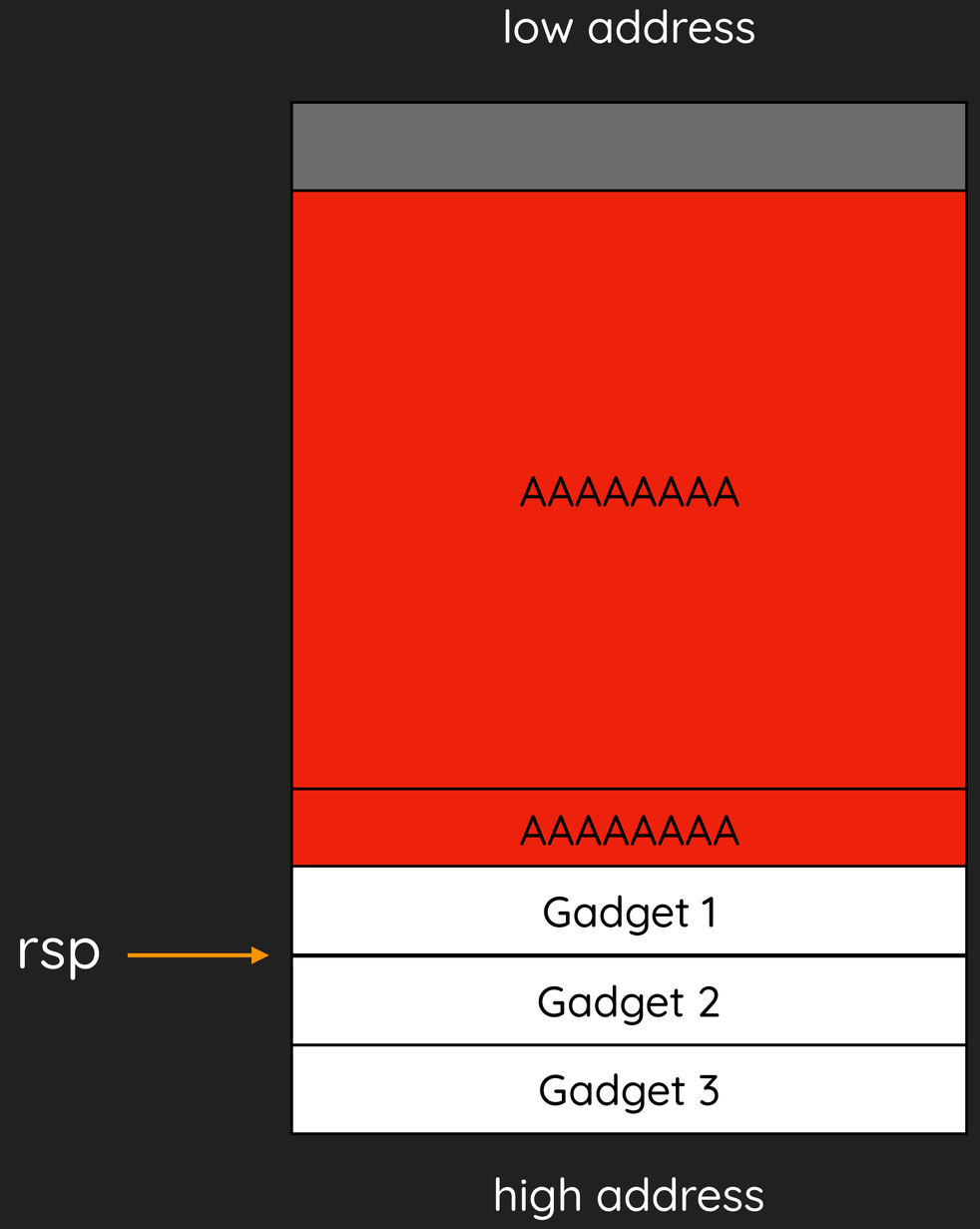
```
            ...
Gadget 2:
            mov rbx, 66
            ret

            ...

Gadget 3:
            add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
rip  →      ret
            ...
```

rax = 100



```
            ...
Gadget 2:
            mov rbx, 66
            ret

            ...

Gadget 3:
            add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
rip  →      ret
            ...
```

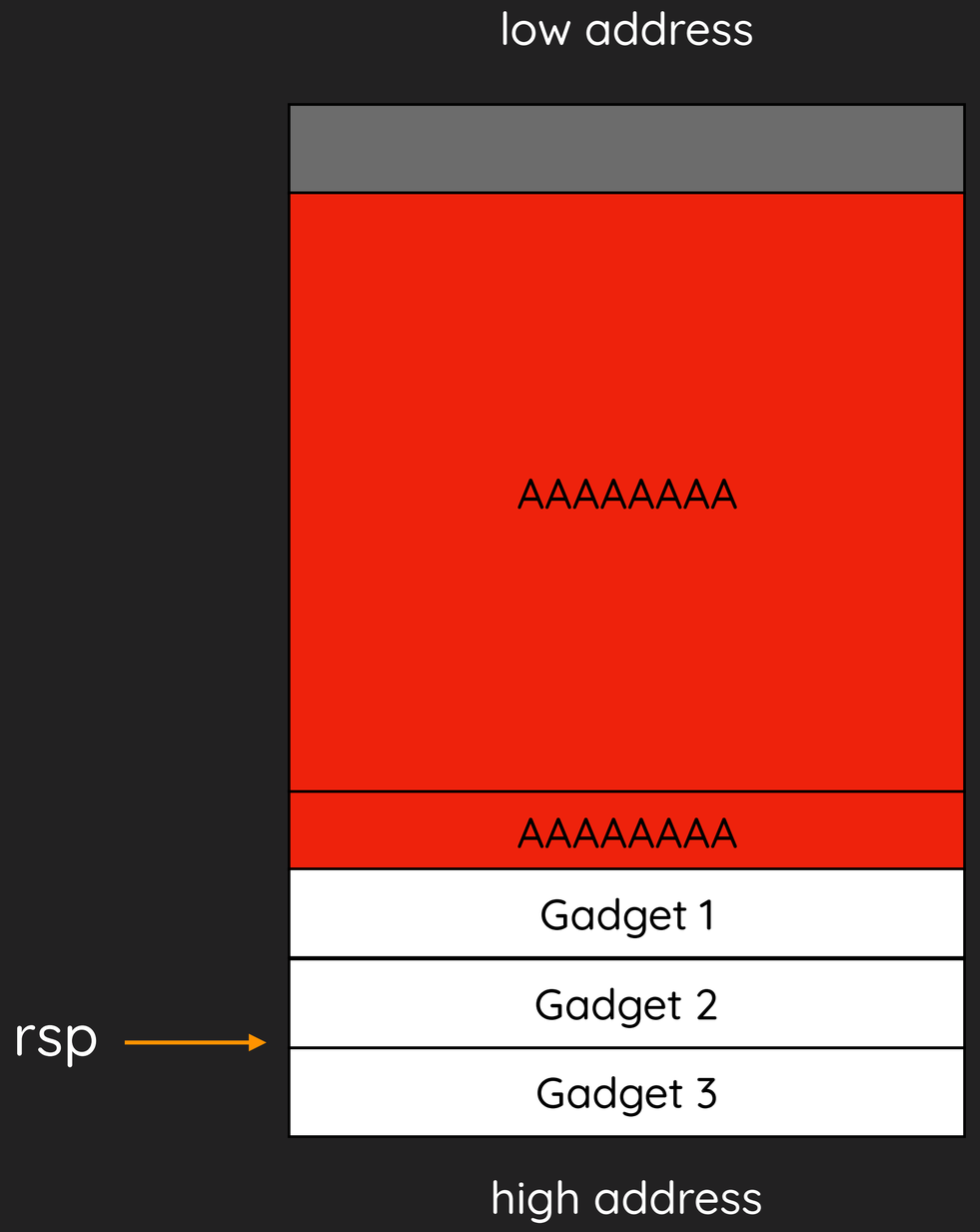
Control return address again.

```
            ...
Gadget 2:
rip  →      mov rbx, 66
            ret

            ...

Gadget 3:
            add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
            ret
            ...
```



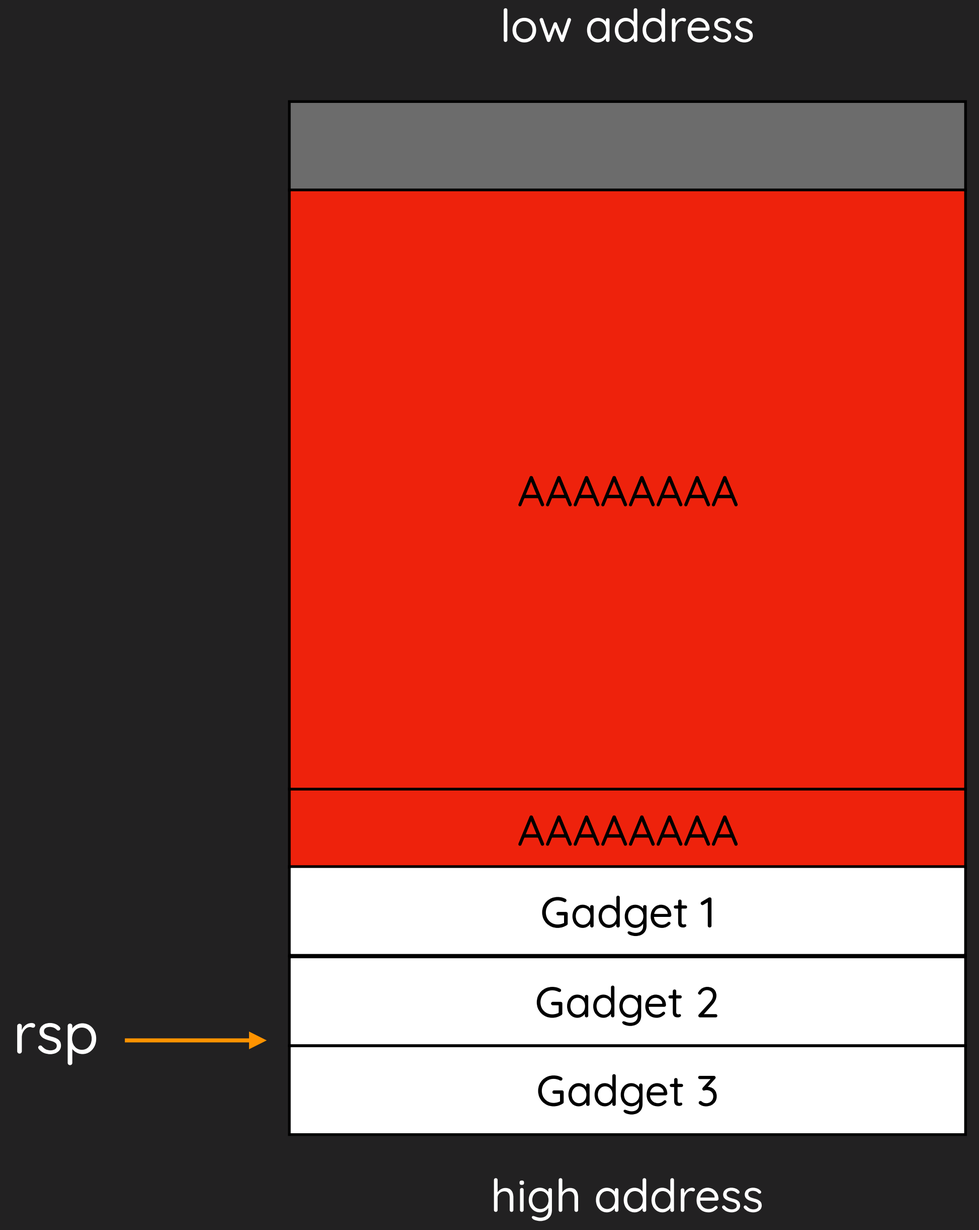
```
            ...
Gadget 2:
            mov rbx, 66
rip  →      ret

            ...

Gadget 3:
            add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
            ret
            ...
```

rbx = 66

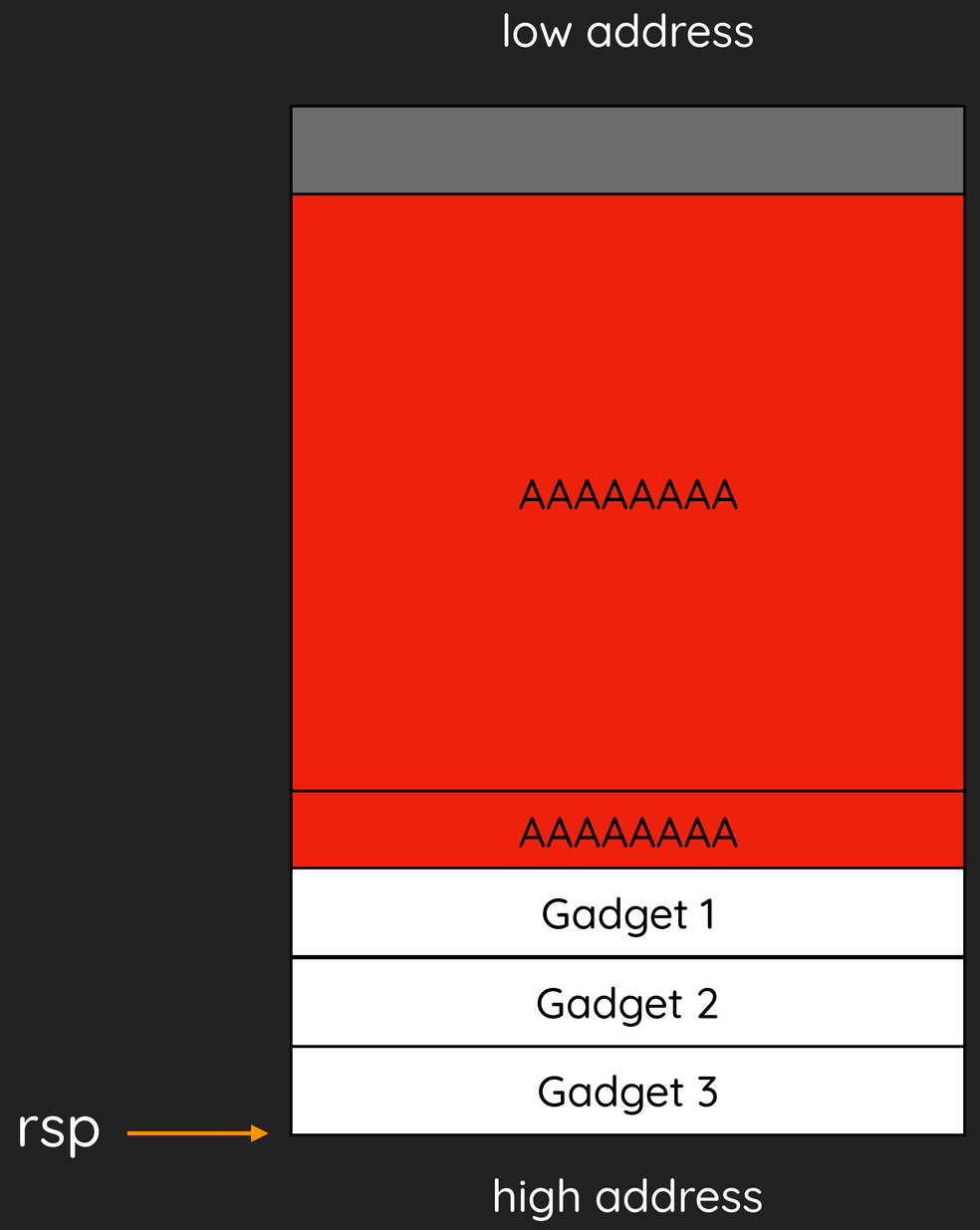


```
            ...
Gadget 2:
            mov rbx, 66
            ret

            ...

Gadget 3:
rip  →      add rax, rbx
            ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
            ret
            ...
```



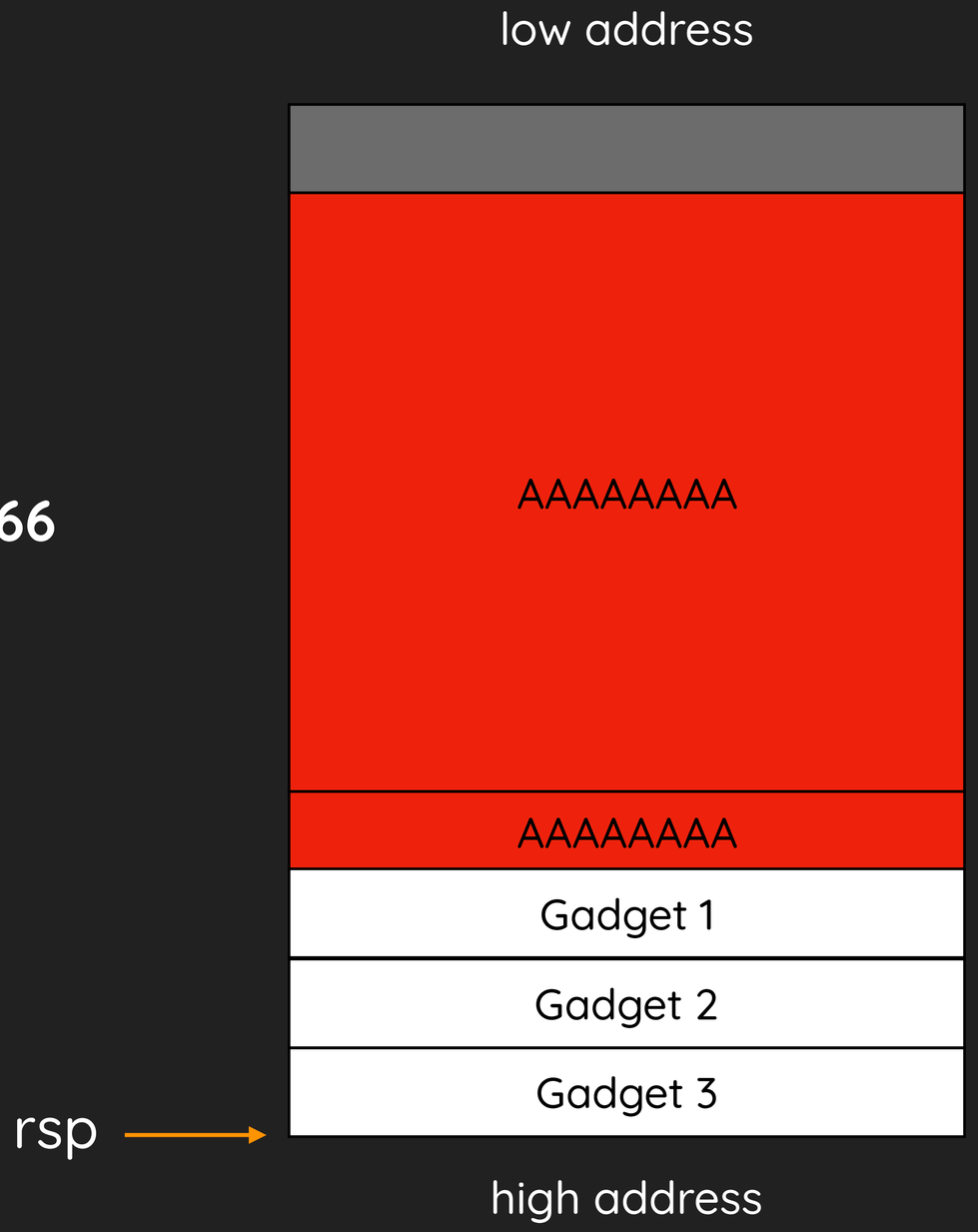
```
            ...
Gadget 2:
            mov rbx, 66
            ret

            ...

Gadget 3:
            add rax, rbx
rip  →      ret
            ...
main:
            ...

            leave
            ret
            ...
Gadget 1:
            mov rax, 100
            ret
            ...
```

rax = 100 + 66 = 166



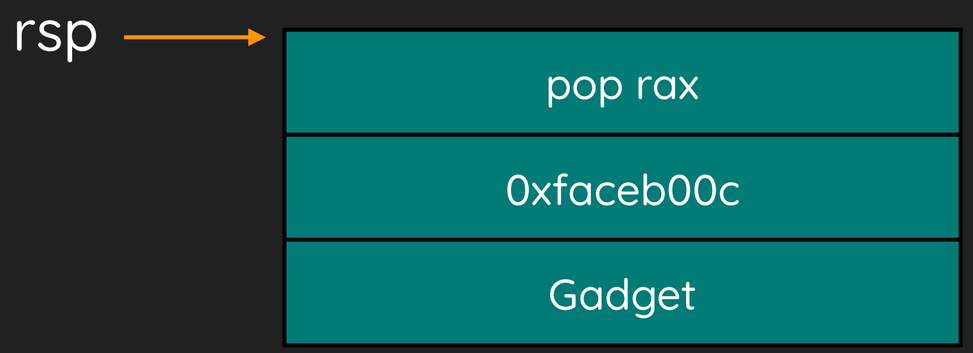
Control Register

ROP

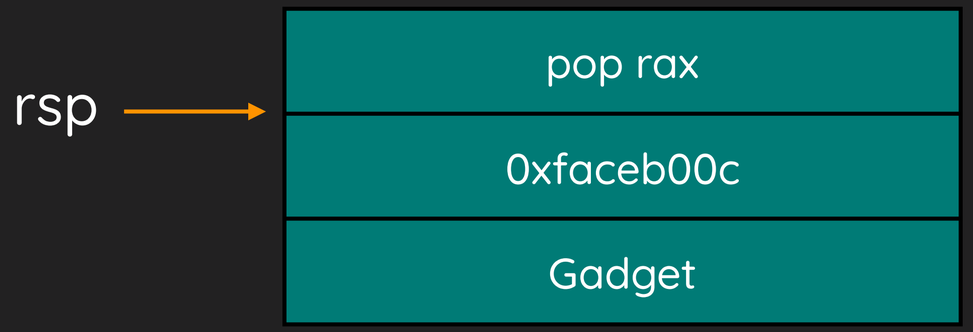
- Control Register
- Gadget - pop <reg>; ret

Return

pop rax
ret



rip → **pop rax**
ret

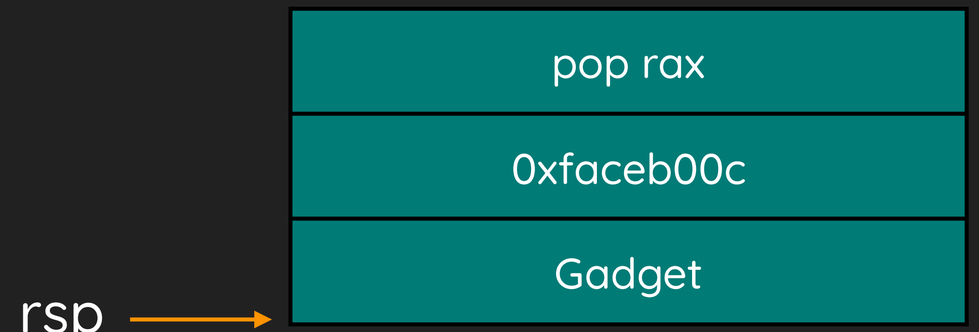


rax = 0xfaceb00c

繼續 ROP

pop rax
ret

rip → **Gadget**



ROP

- 常常不會那麼剛好都會有我們想要的 gadget 並且是 ret 結尾。
  - 想辦法組合 gadget，達成同樣的目的
- 例如想控制 rax，沒有 pop rax; ret gadget 也高機率不會剛剛好有 mov rax, <希望的值>：
  - 假設存在，pop rdi; ret 與 mov rax, rdi; ret，組合出控制 rax 值的 payload
  - 或是找到 xor rax, rax; ret 與 inc rax; ret，先將 rax 清零在一直加一至想要的值
  - 等等

ROP

- 存在 overflow，或任何成功 control rip 的前提下：
  - NX 關的情況下，我們可以撰寫執行 execve("/bin/sh" , 0 , 0) 的 shellcode 並嘗試控制 rip 跳至 shellcode。
  - 但 NX 開啟時，則可以透過 ROP 的方式，堆疊出執行 execve("/bin/sh" , 0 , 0) 行為的 ROP Chain。

ROP

Shellocode

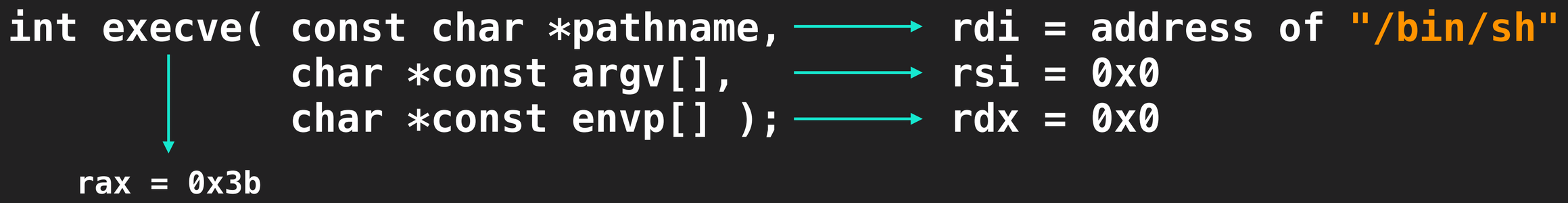
```
int execve( const char *pathname,  ──→  rdi = address of "/bin/sh"
            char *const argv[],    ──→  rsi = 0x0
            char *const envp[] );  ──→  rdx = 0x0
```



```
rax = 0x3b
```

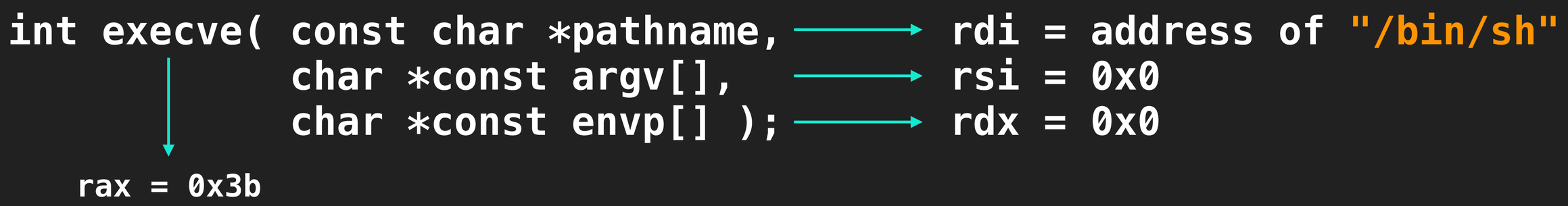
ROP

NX - ~~Shellocode~~

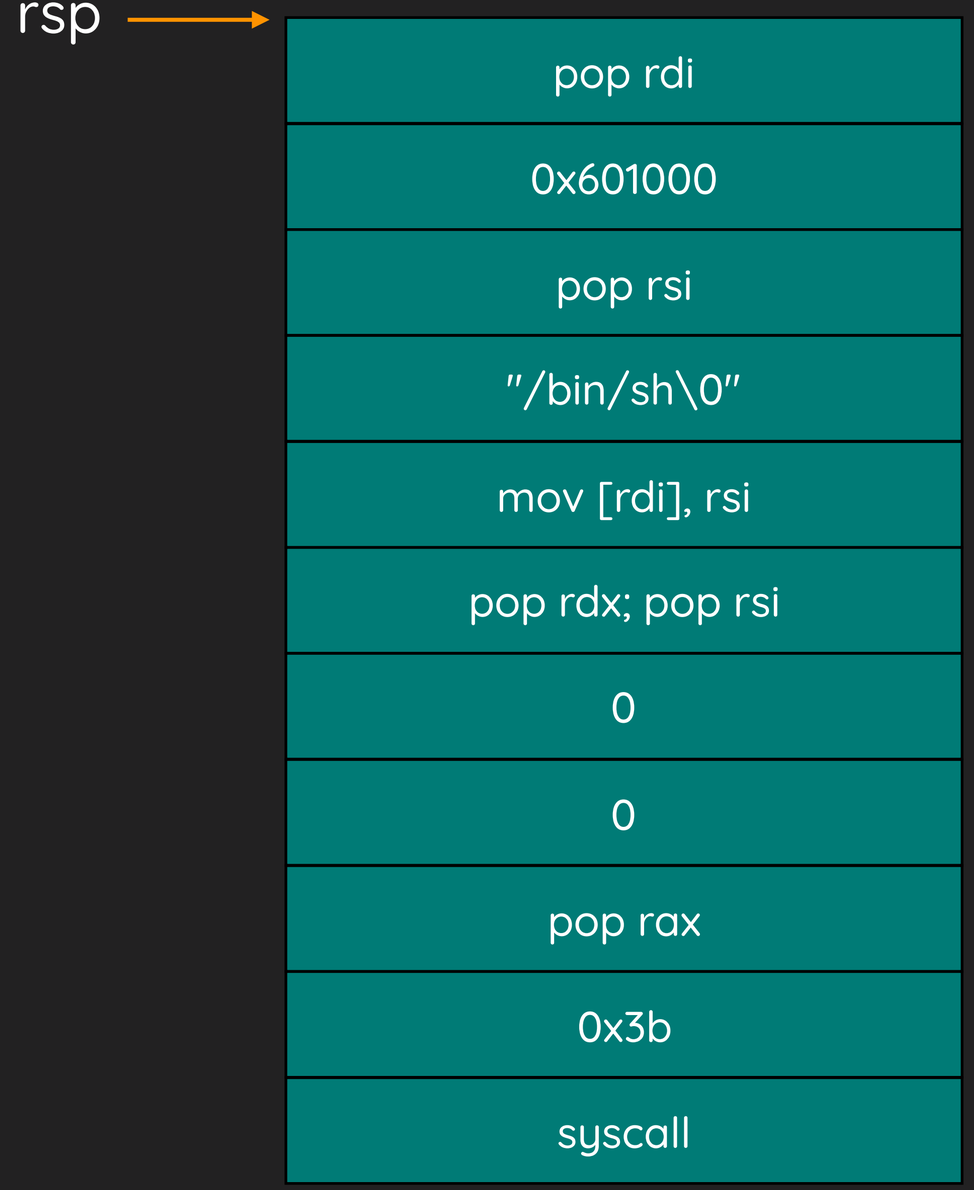


ROP

ROP!

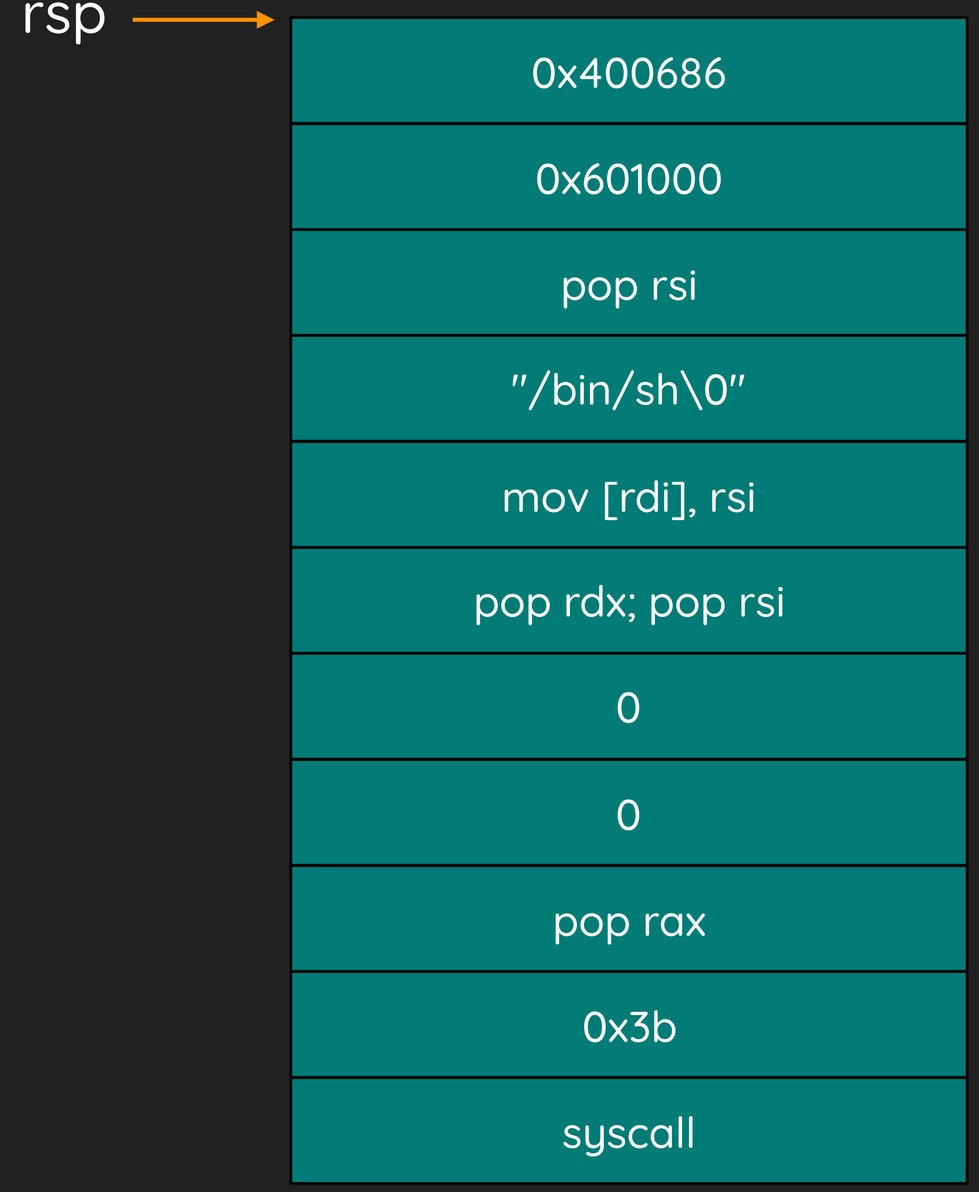

int execve(const char *pathname,
char *const argv[],
char *const envp[]);
rdi = address of "/bin/sh"
rsi = 0x0
rdx = 0x0
rax = 0x3b

ROP



rax =
rbx =
rcx =
rdx =
rdi =
rsi =

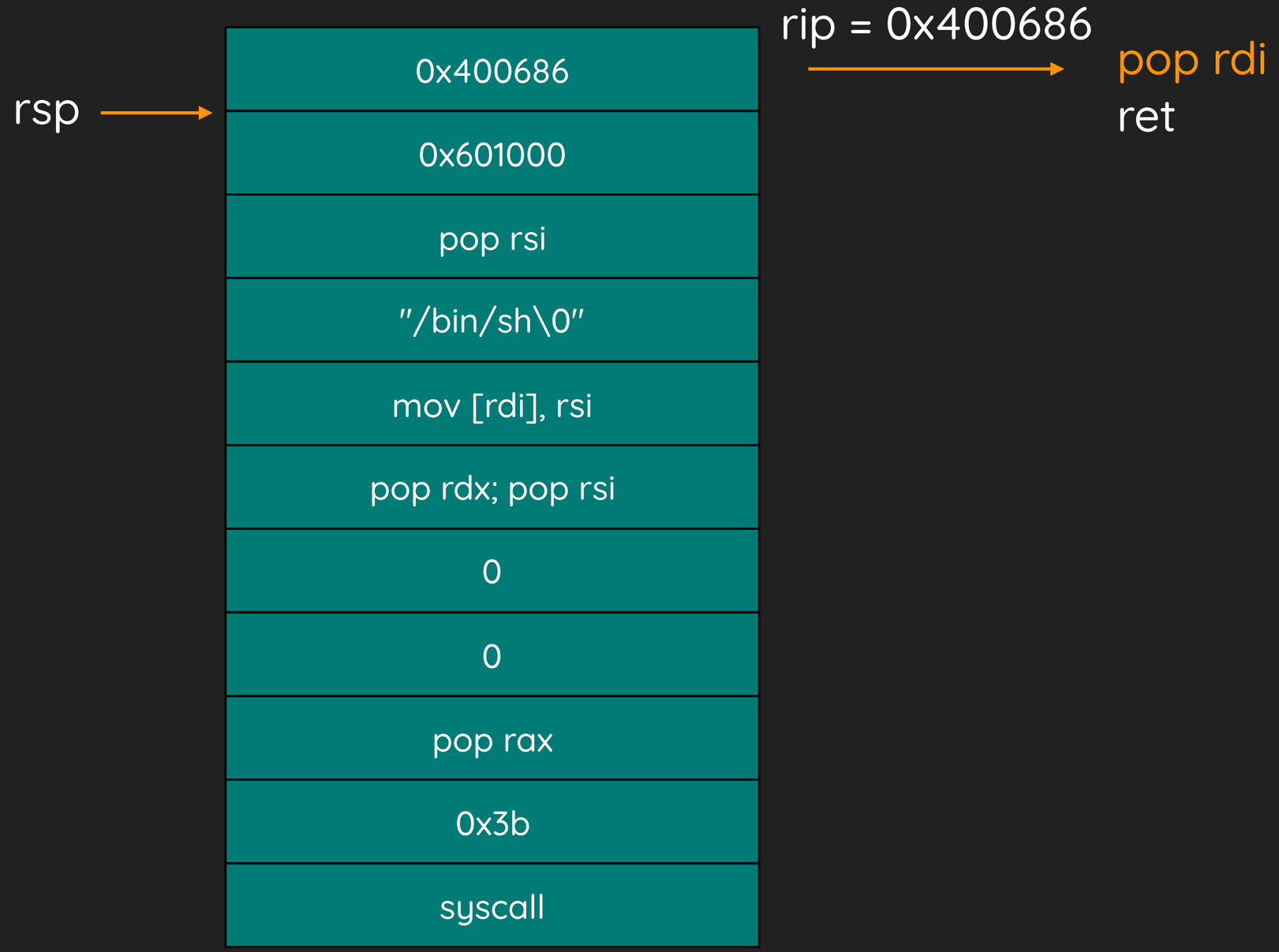
ROP



假如 pop rdi; ret 此 gadget 位於 0x400686

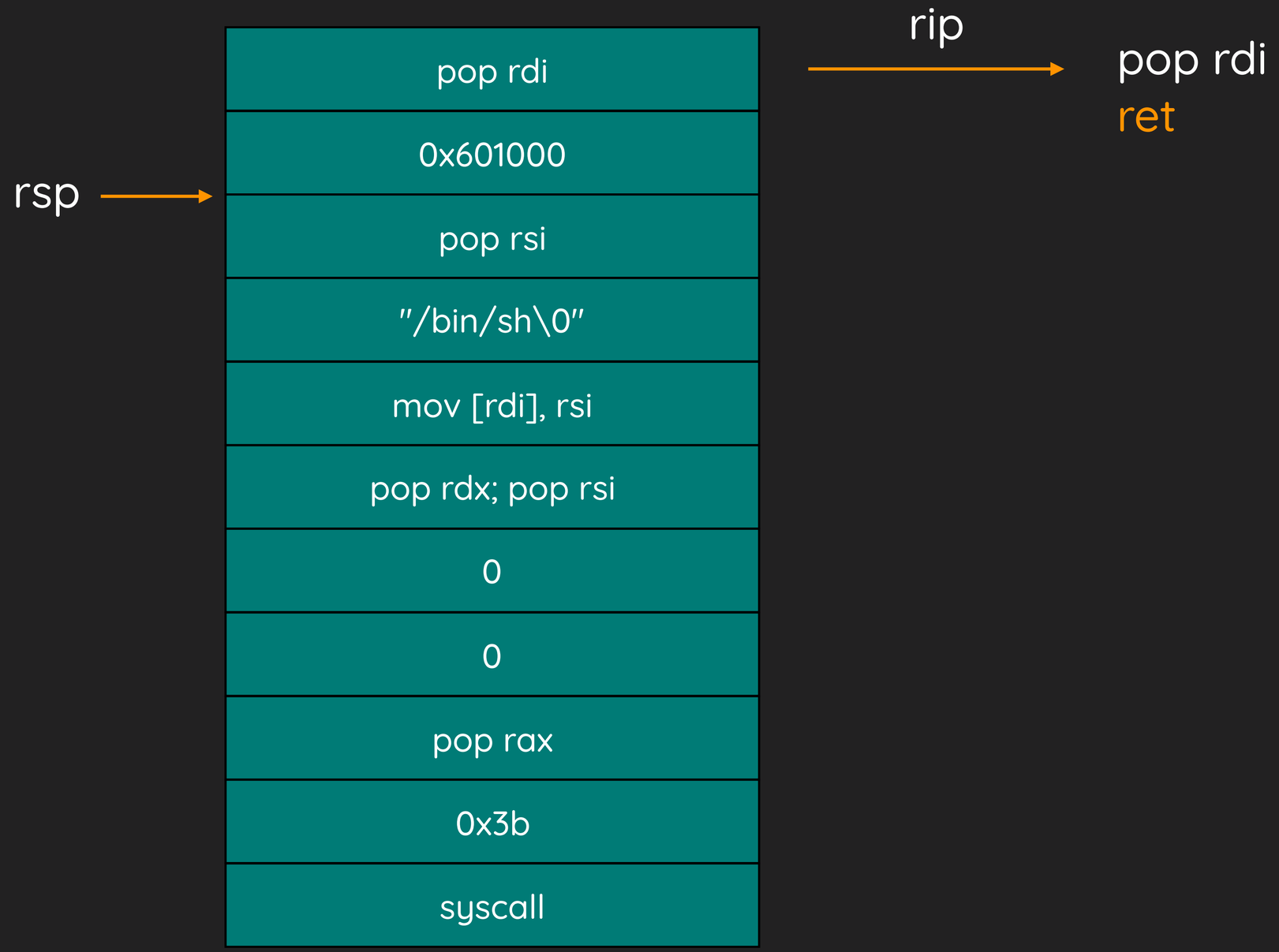
rax =
rbx =
rcx =
rdx =
rdi =
rsi =

ROP



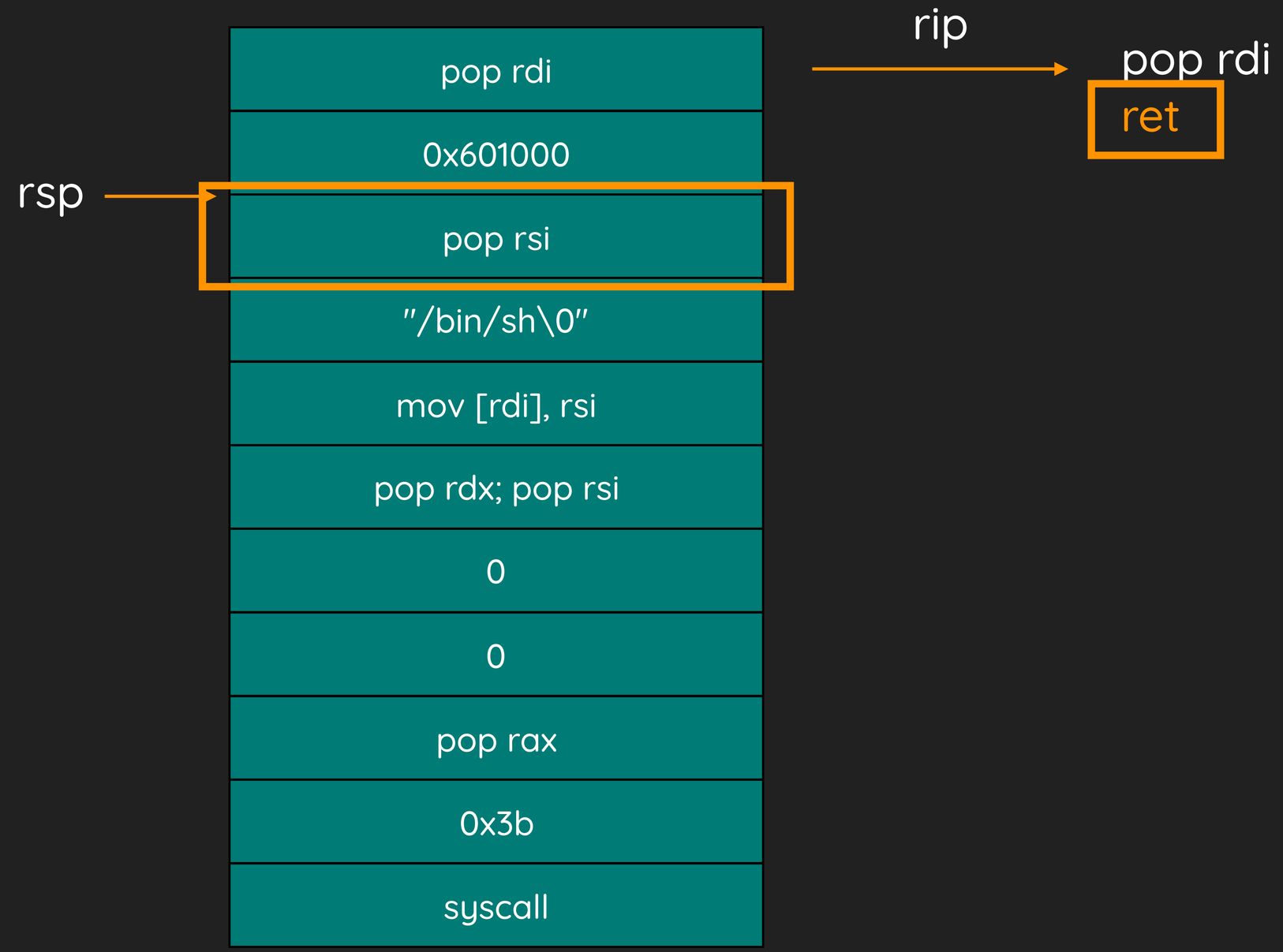
rax =
rbx =
rcx =
rdx =
rdi =
rsi =

ROP



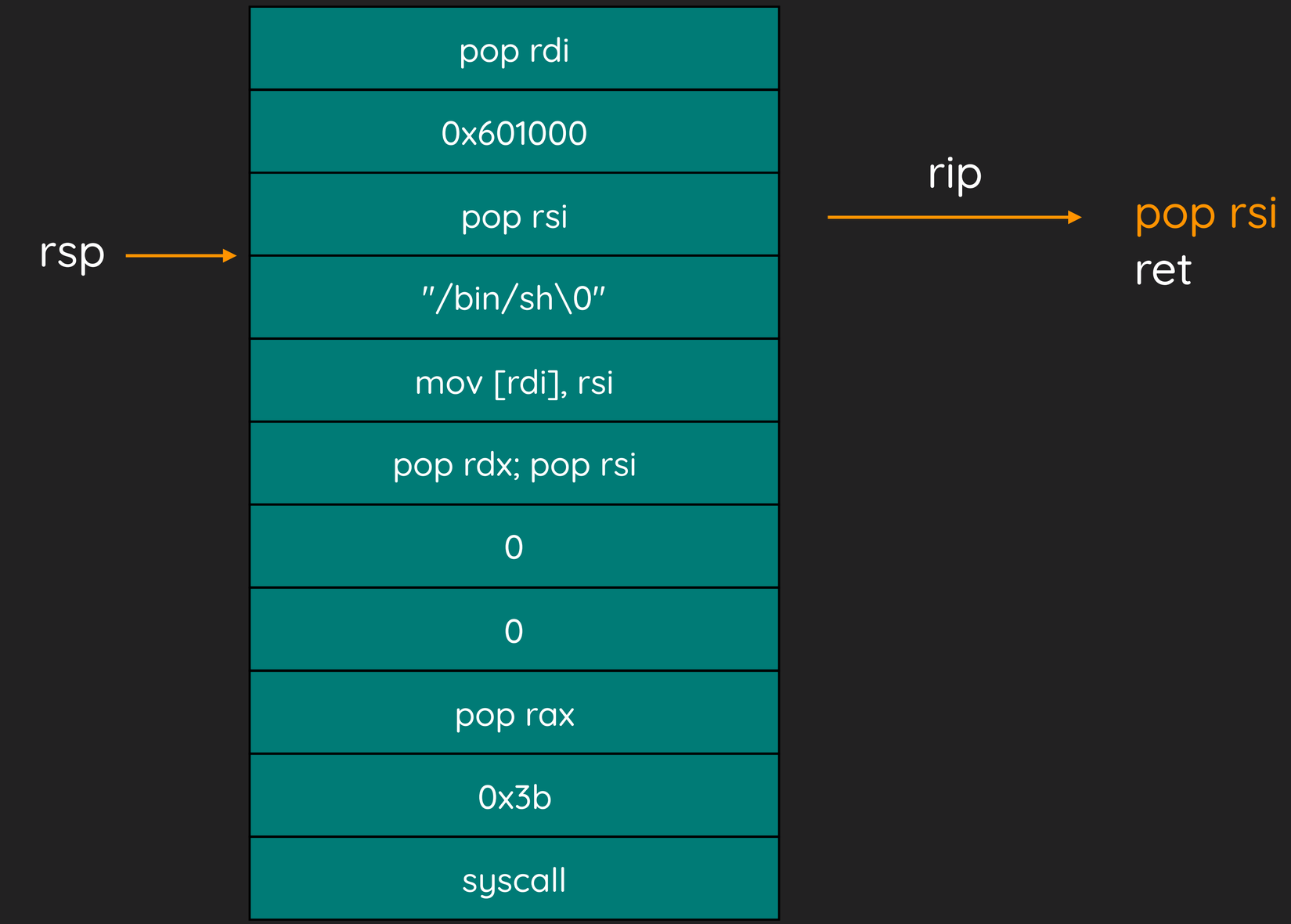

```
rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi =
```

ROP



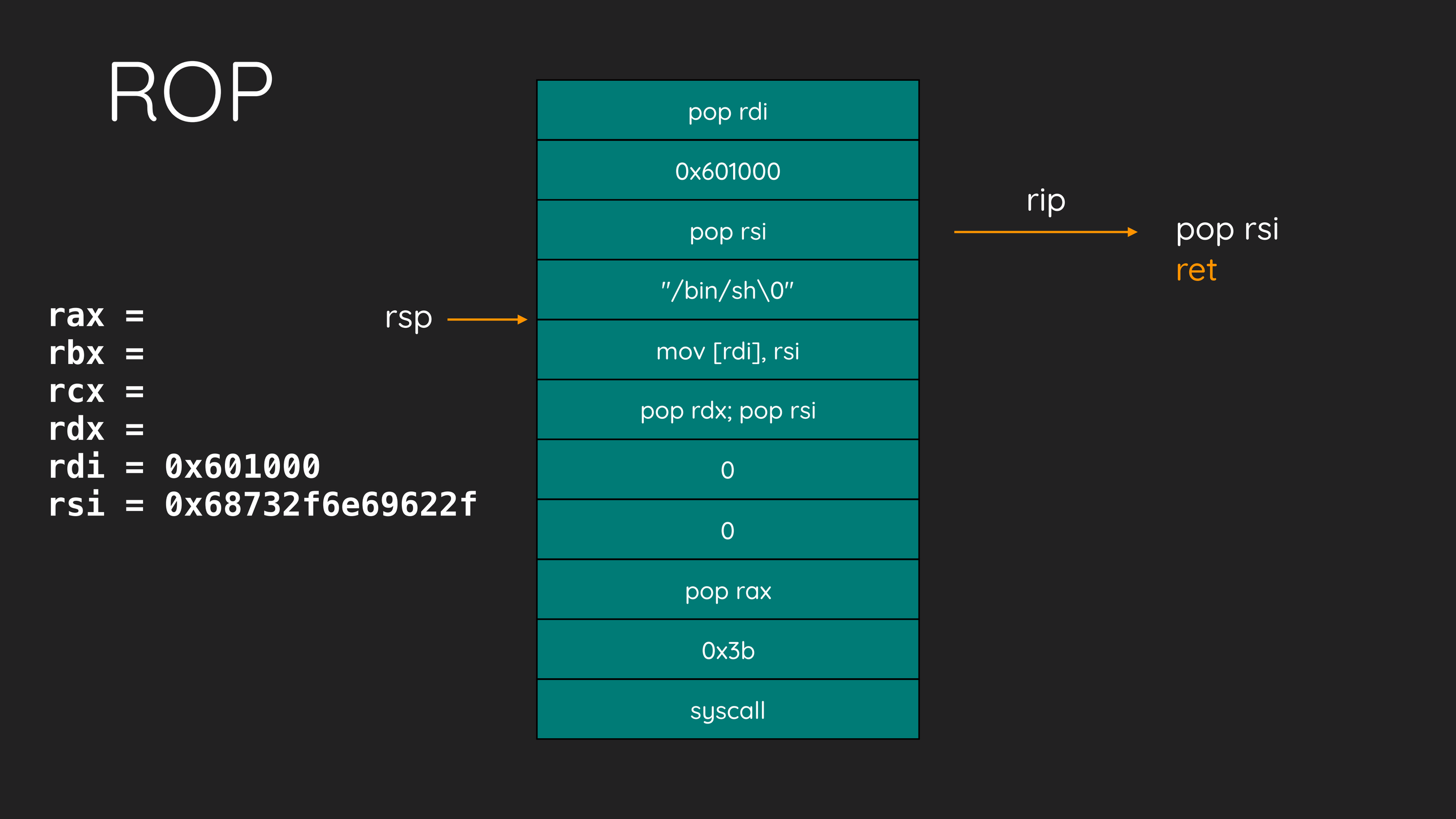

```
rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi =
```

ROP



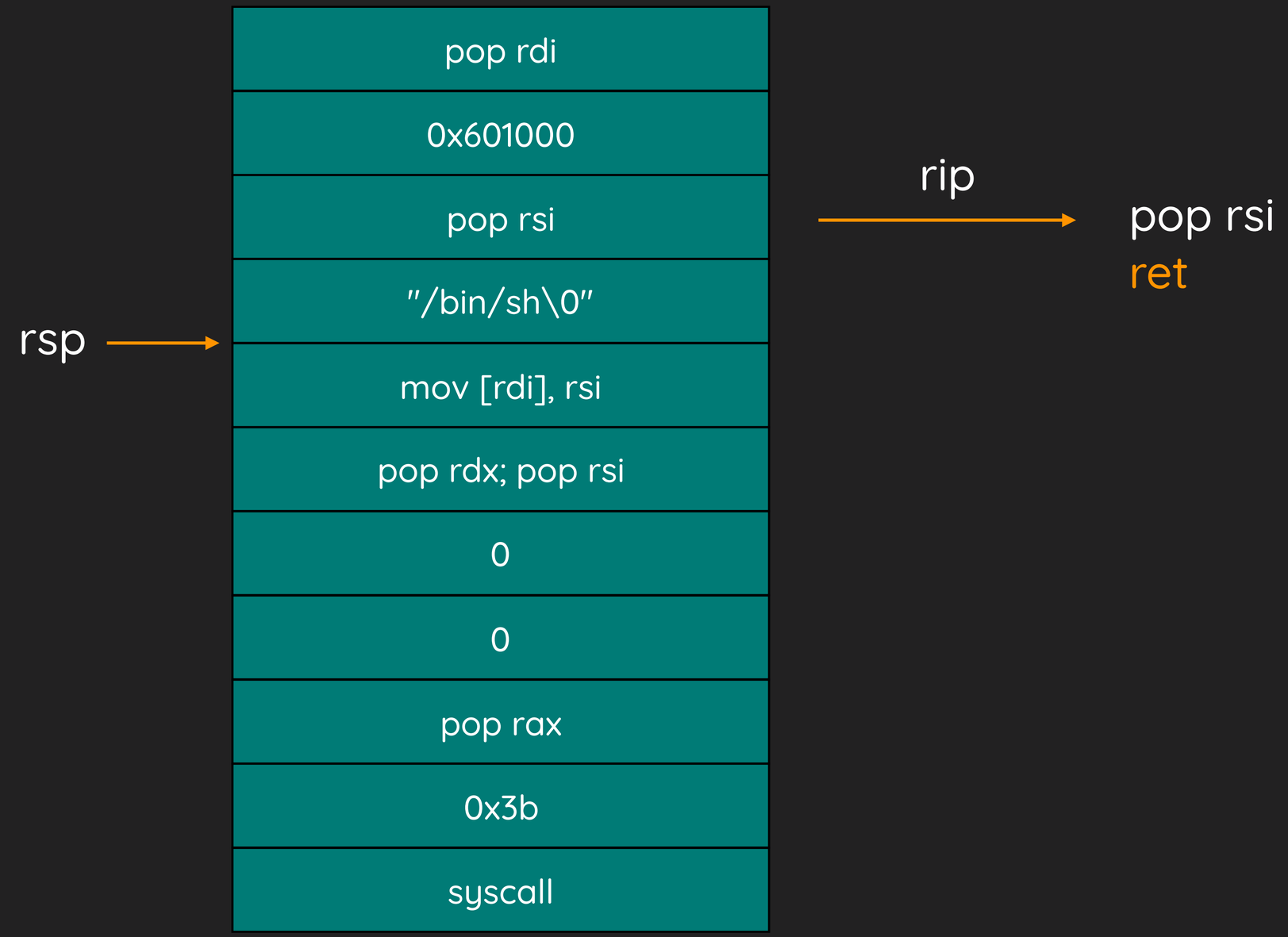
rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi =

ROP


rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi = 0x68732f6e69622f
rsp
pop rdi
0x601000
pop rsi
"/bin/sh\0"
mov [rdi], rsi
pop rdx; pop rsi
0
0
pop rax
0x3b
syscall
rip
pop rsi
ret

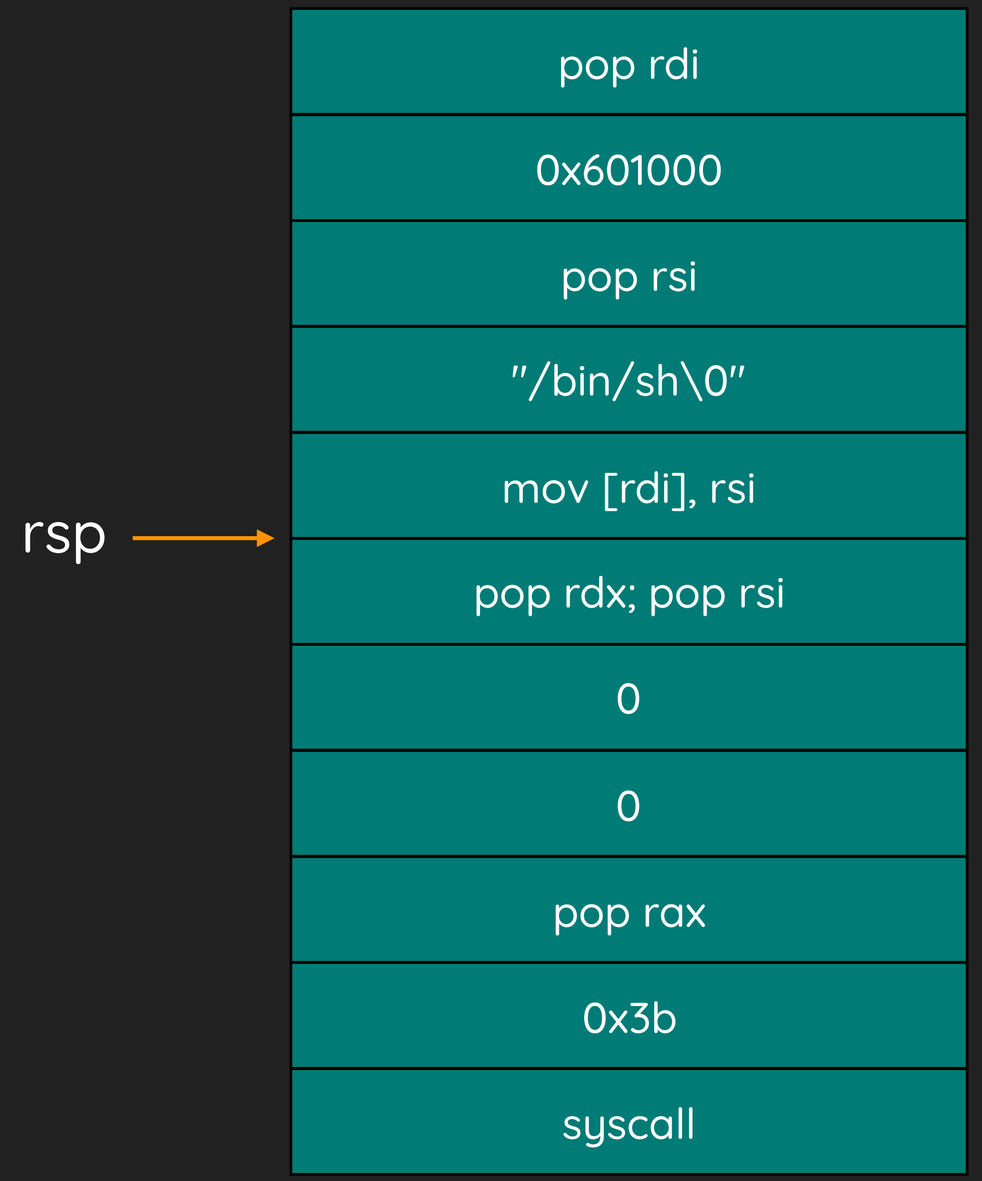
ROP

```
rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi = "/bin/sh"
```



ROP

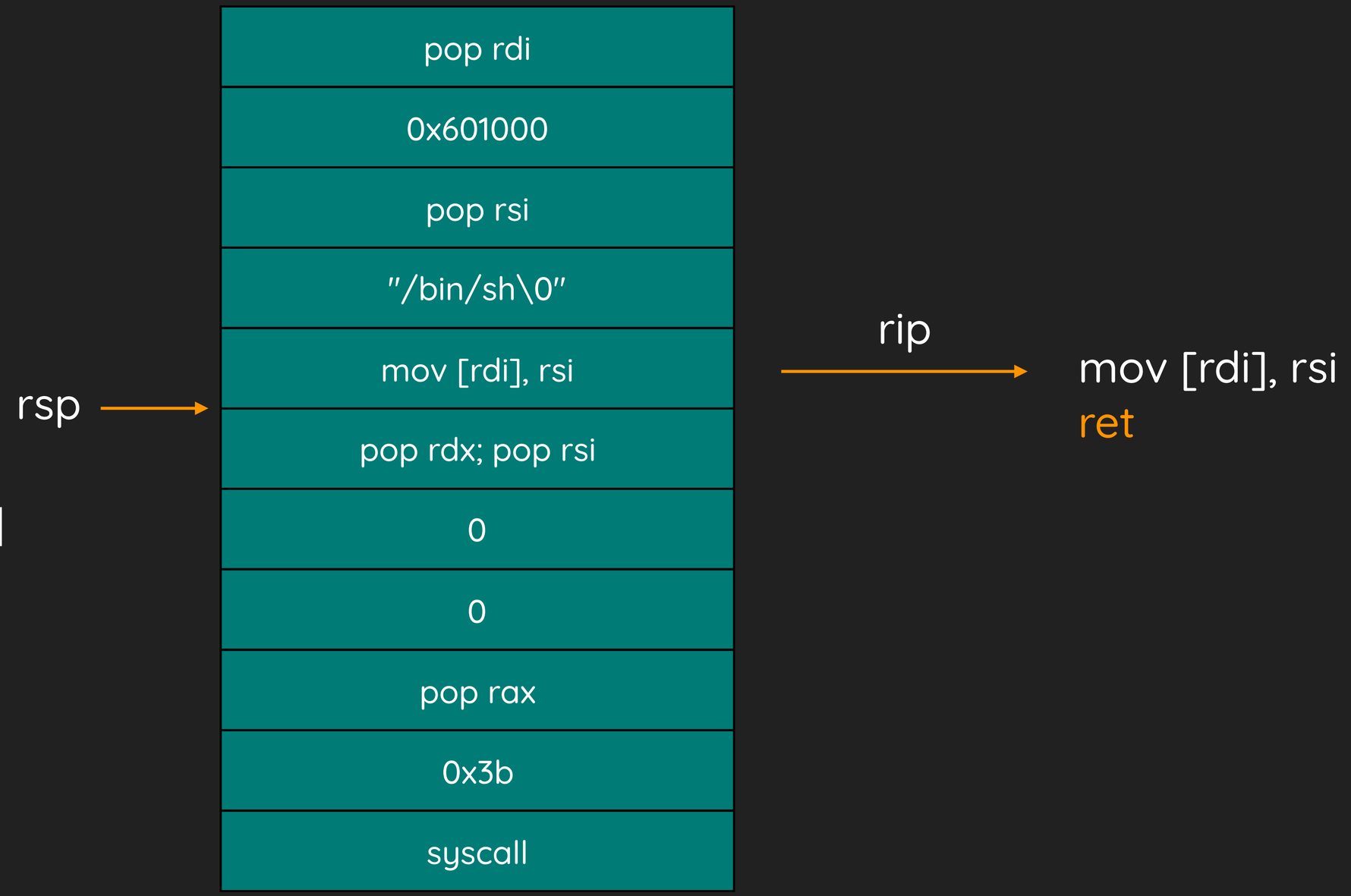
```
rax =
rbx =
rcx =
rdx =
rdi = 0x601000
rsi = "/bin/sh"
```



將 "/bin/sh" 字串
存到 0x601000 的位置

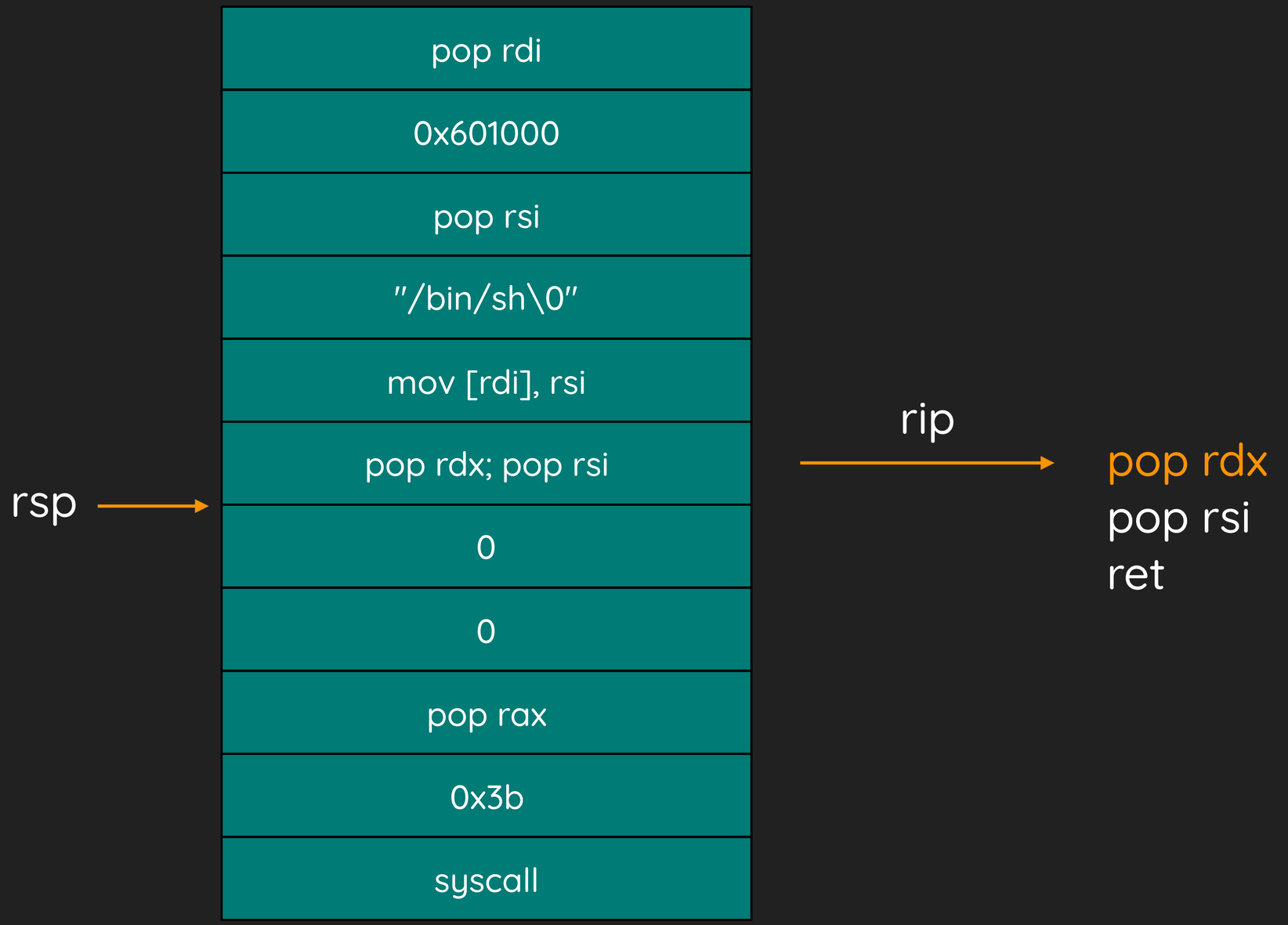
ROP

```
rax =
rbx =
rcx =
rdx =
rdi = ["/bin/sh"]
rsi = "/bin/sh"
```



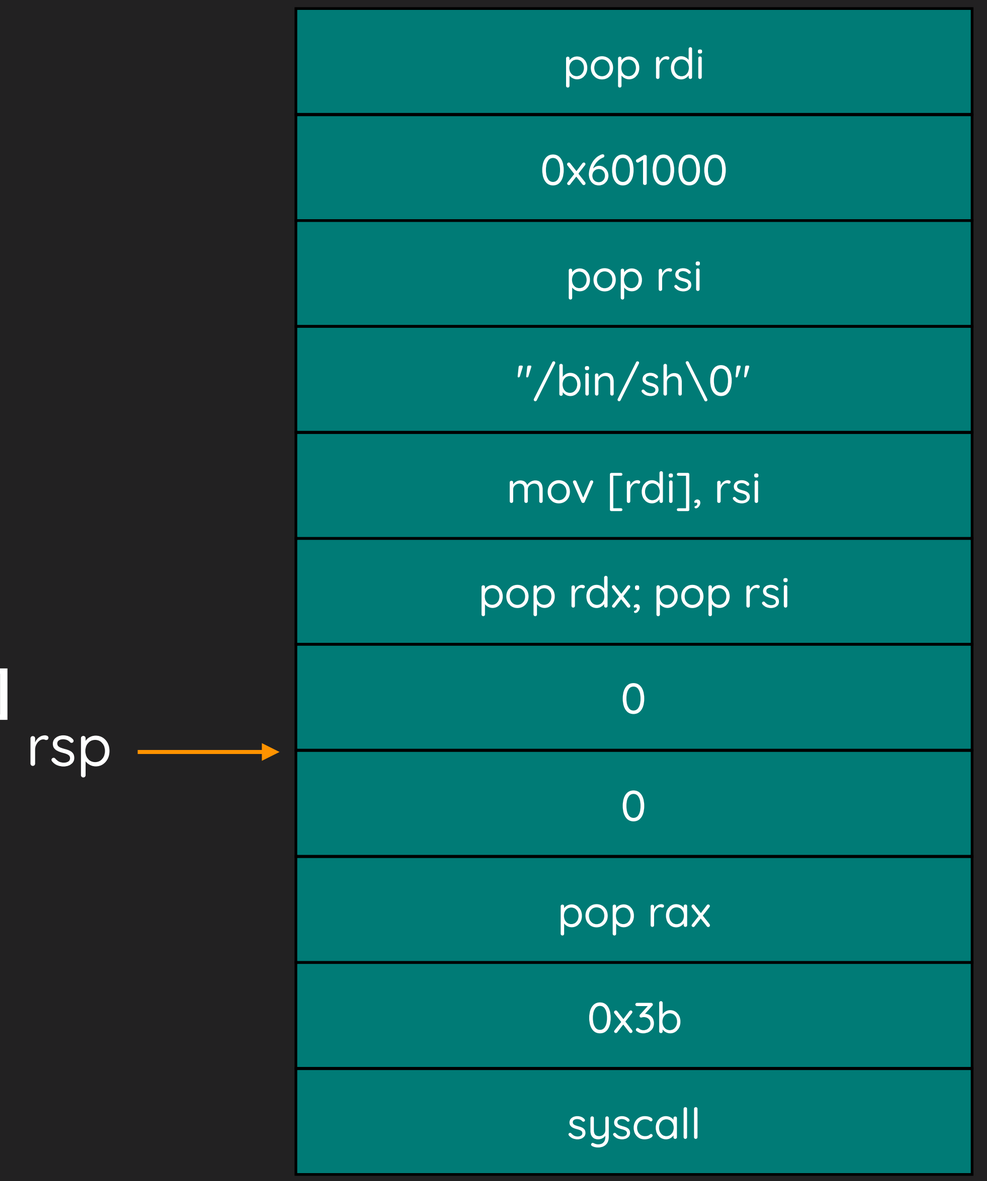
ROP

```
rax =
rbx =
rcx =
rdx =
rdi = ["/bin/sh"]
rsi = "/bin/sh"
```



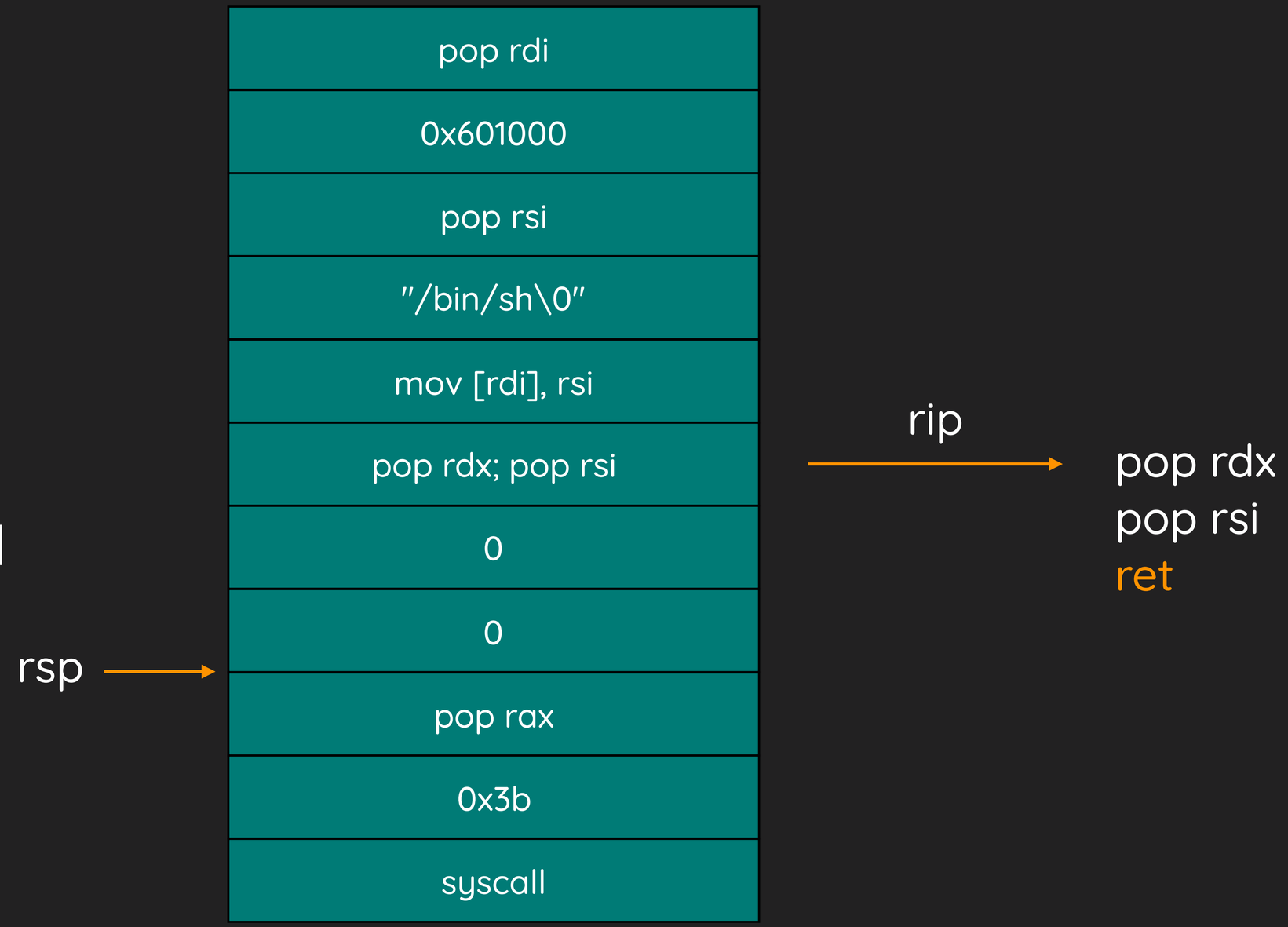
ROP

```
rax =
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = "/bin/sh"
```



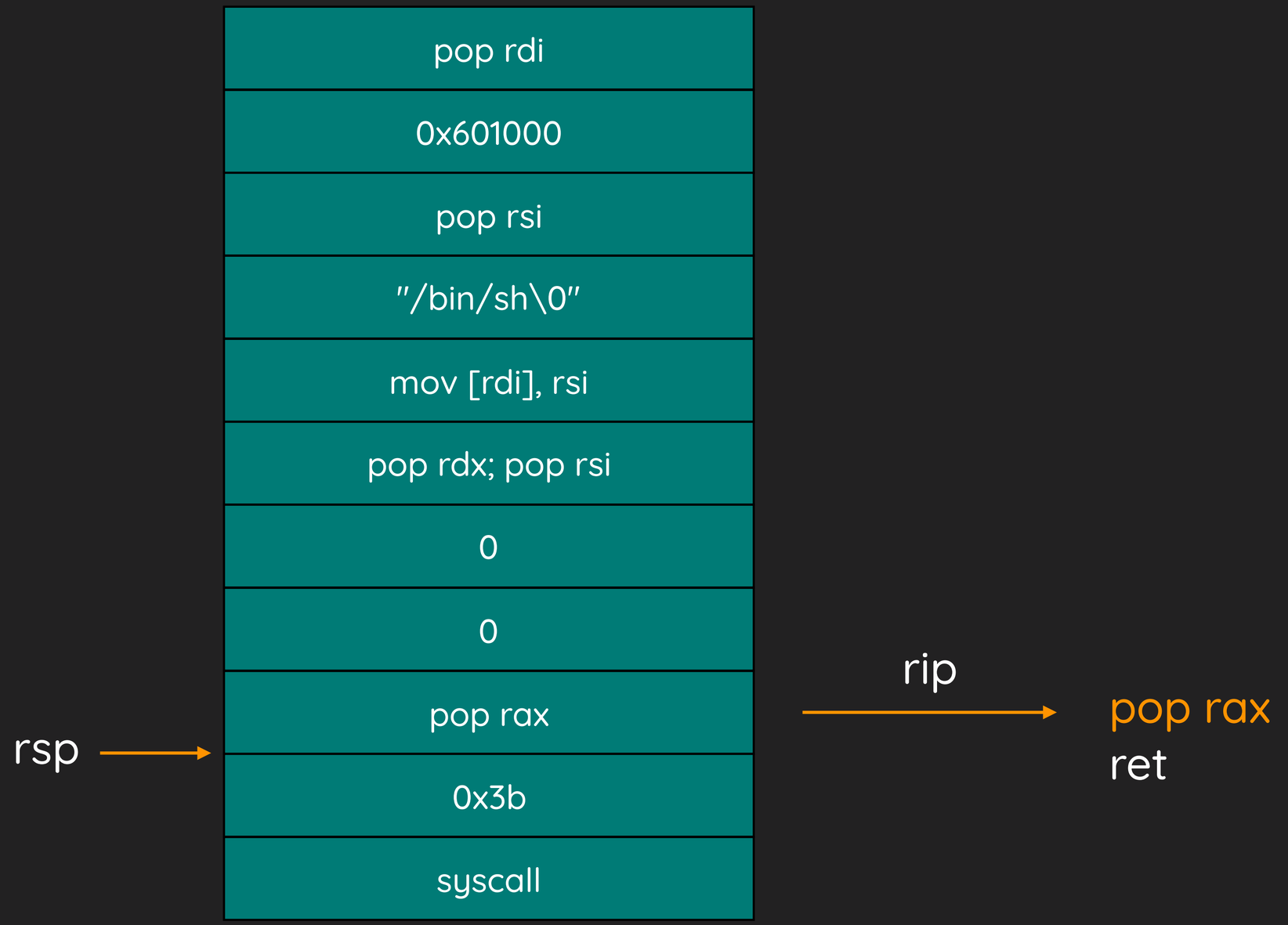
ROP

```
rax =
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
```



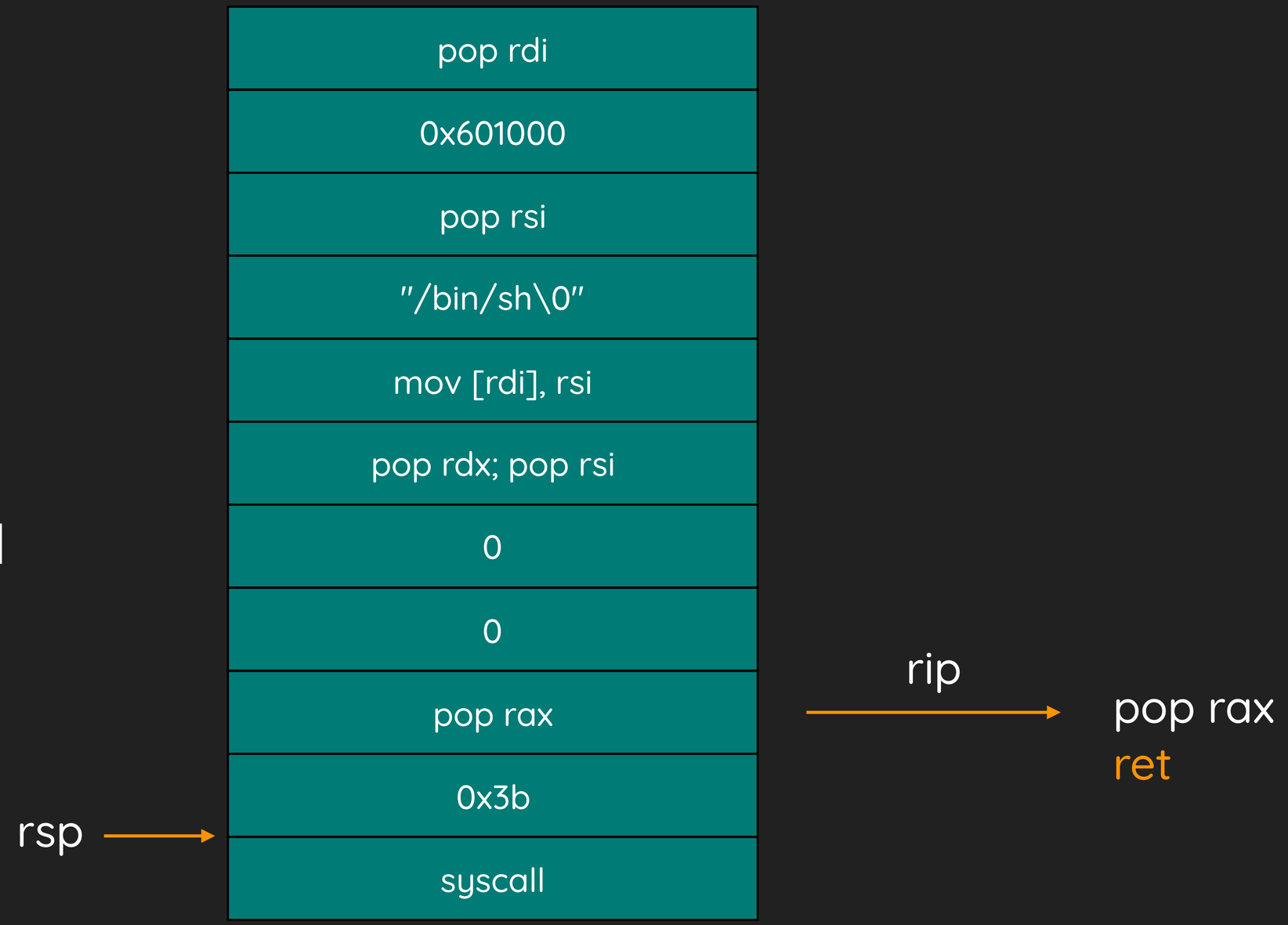
ROP

```
rax =
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
```



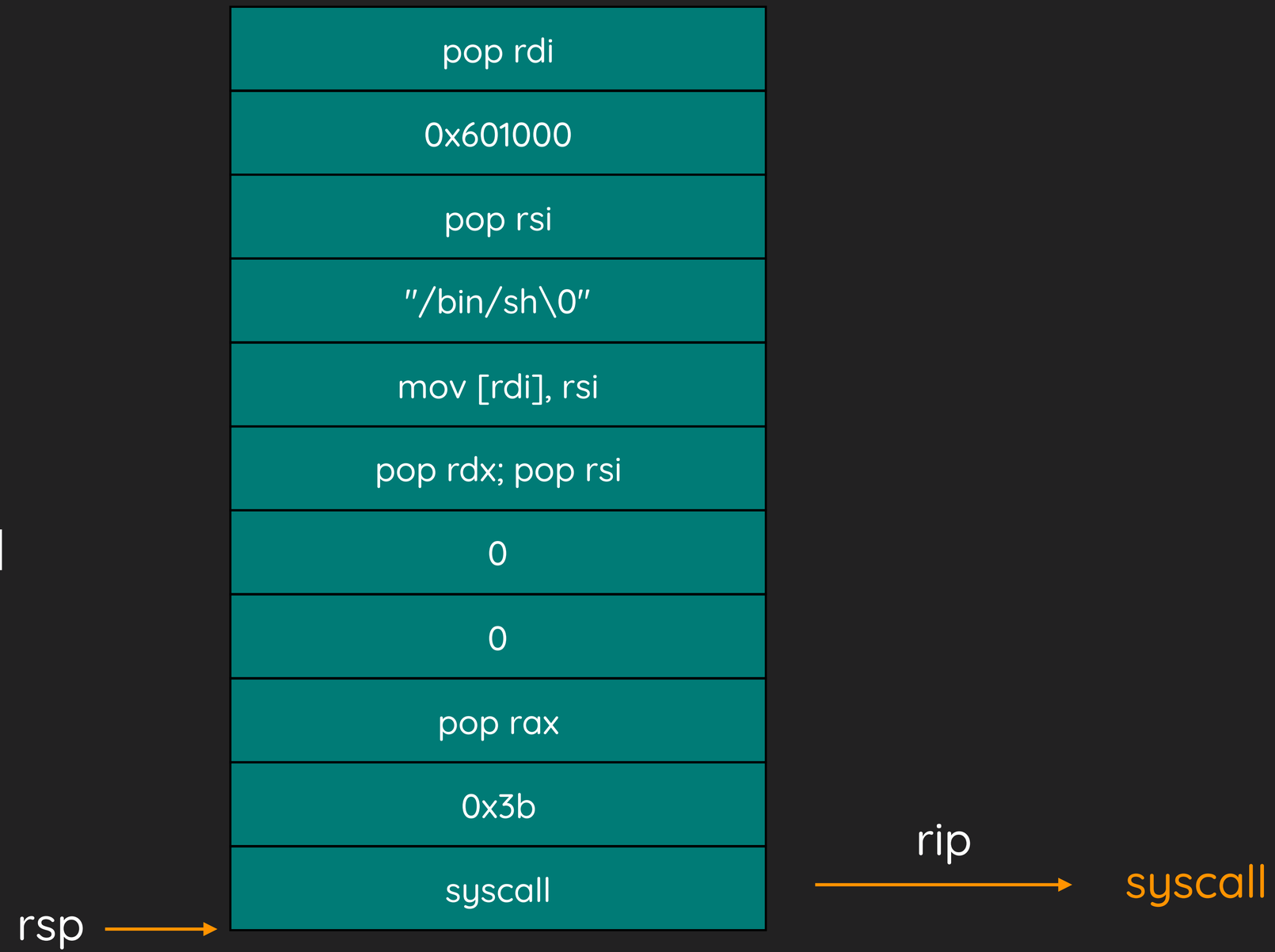
ROP

```
rax = 0x3b
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
```



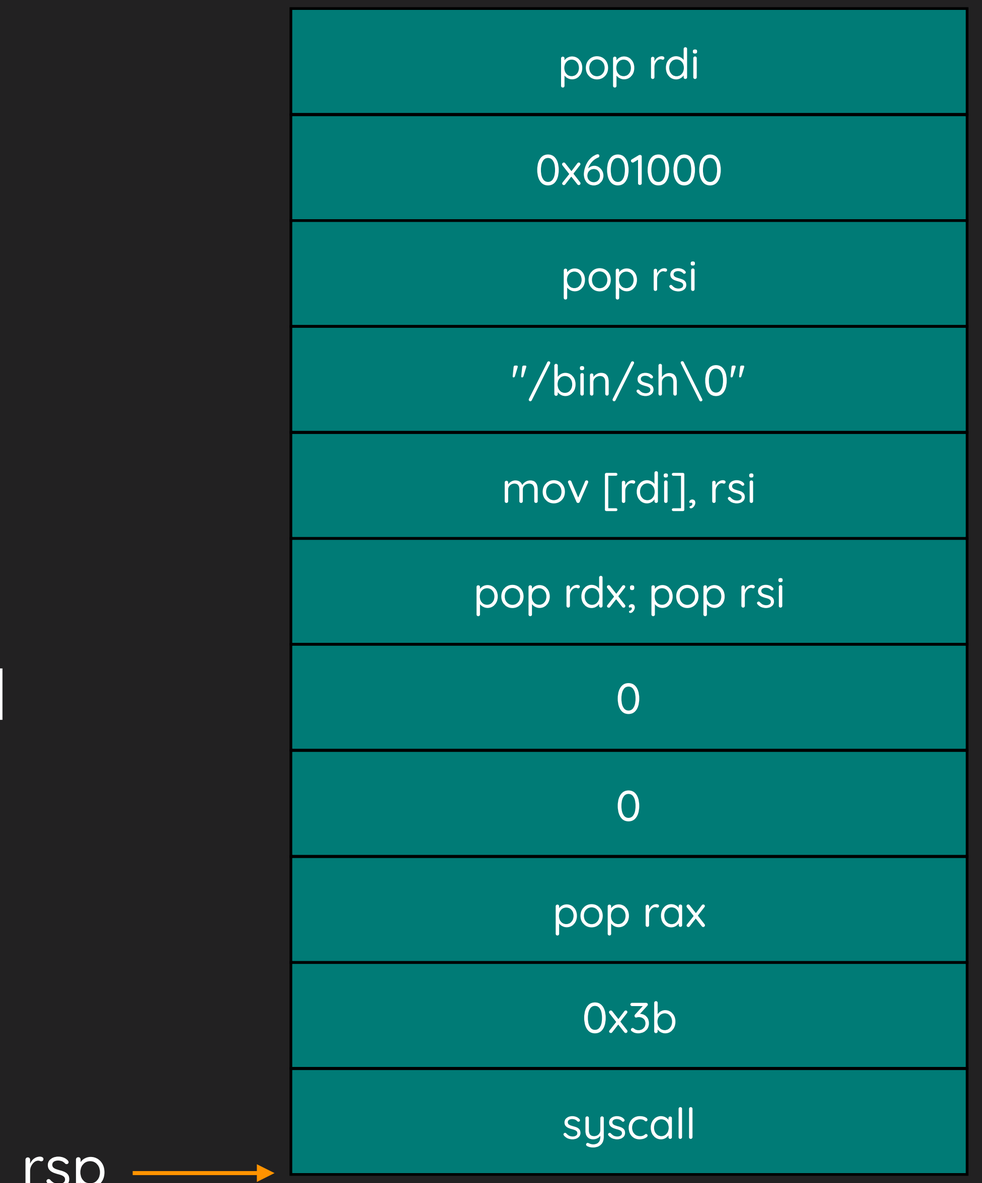
ROP

```
rax = 0x3b
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
```



ROP

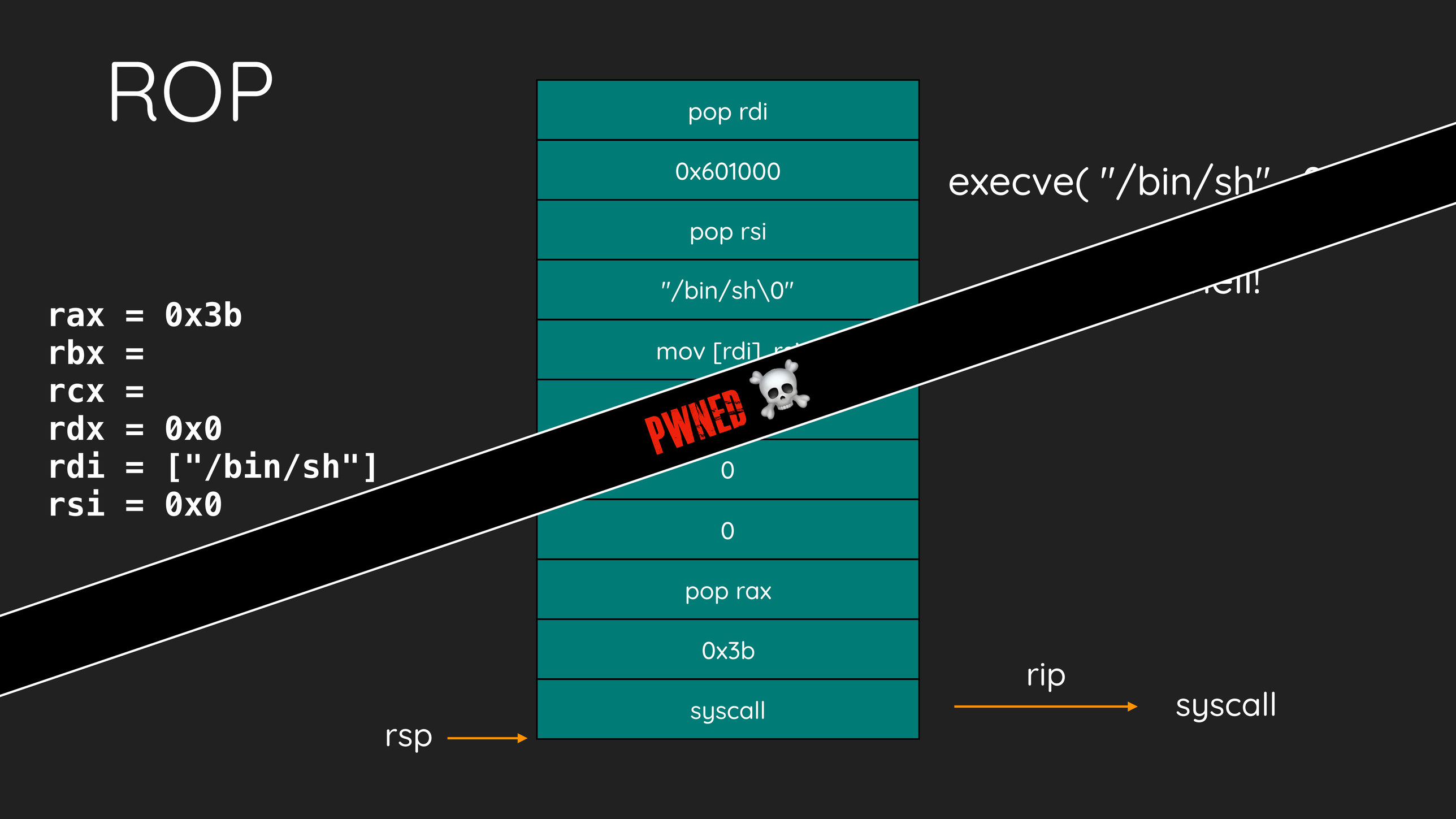
```
rax = 0x3b
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
```



execve("/bin/sh" , 0 , 0)

Get Shell!

ROP


rax = 0x3b
rbx =
rcx =
rdx = 0x0
rdi = ["/bin/sh"]
rsi = 0x0
pop rdi
0x601000
pop rsi
"/bin/sh\0"
mov [rdi]
0
0
pop rax
0x3b
syscall
PWNED
execve("/bin/sh"
rsp
rip
syscall

Demo

ret2plt

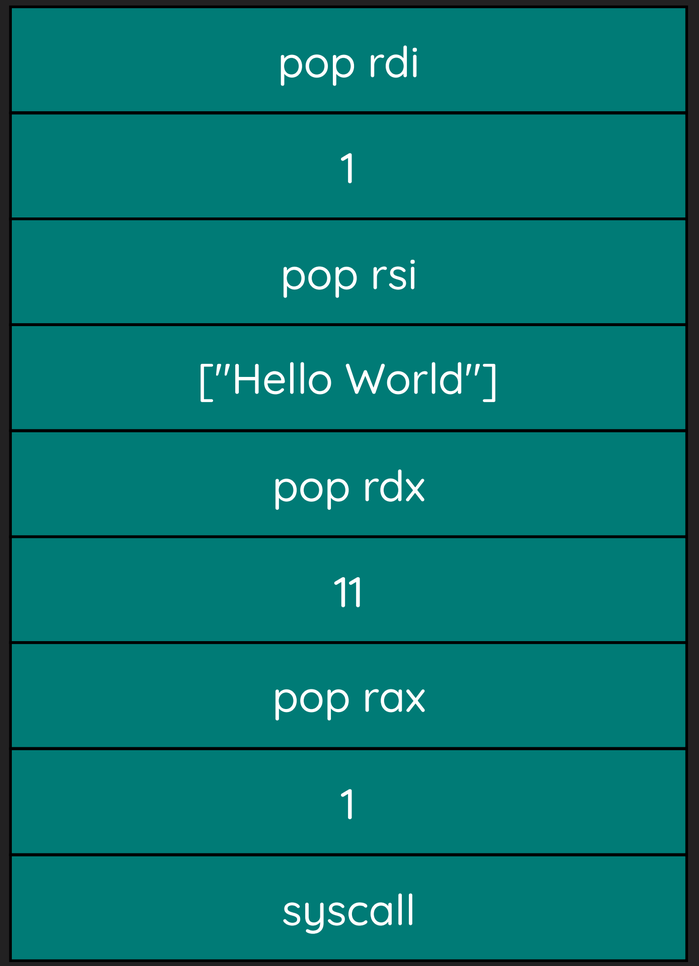
return to .plt

ret2plt

- 如果想要串的行為本身就有 function 在 binary 中，例如，binary 本身有 puts()，我們可以直接用 ROP 放好參數直接使用它，就不用用 ROP 去堆全部的行為。
- 在 PIE 關的情況下，即使不知道 library function address (因 ASLR)，也可以透過 return 到 .plt 上來使用這個 function，這個做法即稱為 ret2plt。

ret2plt

- write(1 , "Hello World" , 12)

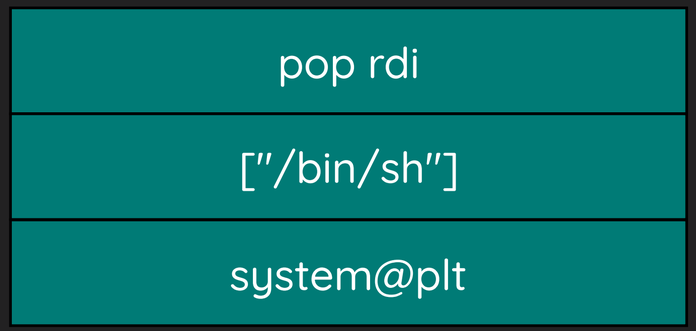


ret2plt

- puts("Hello World")

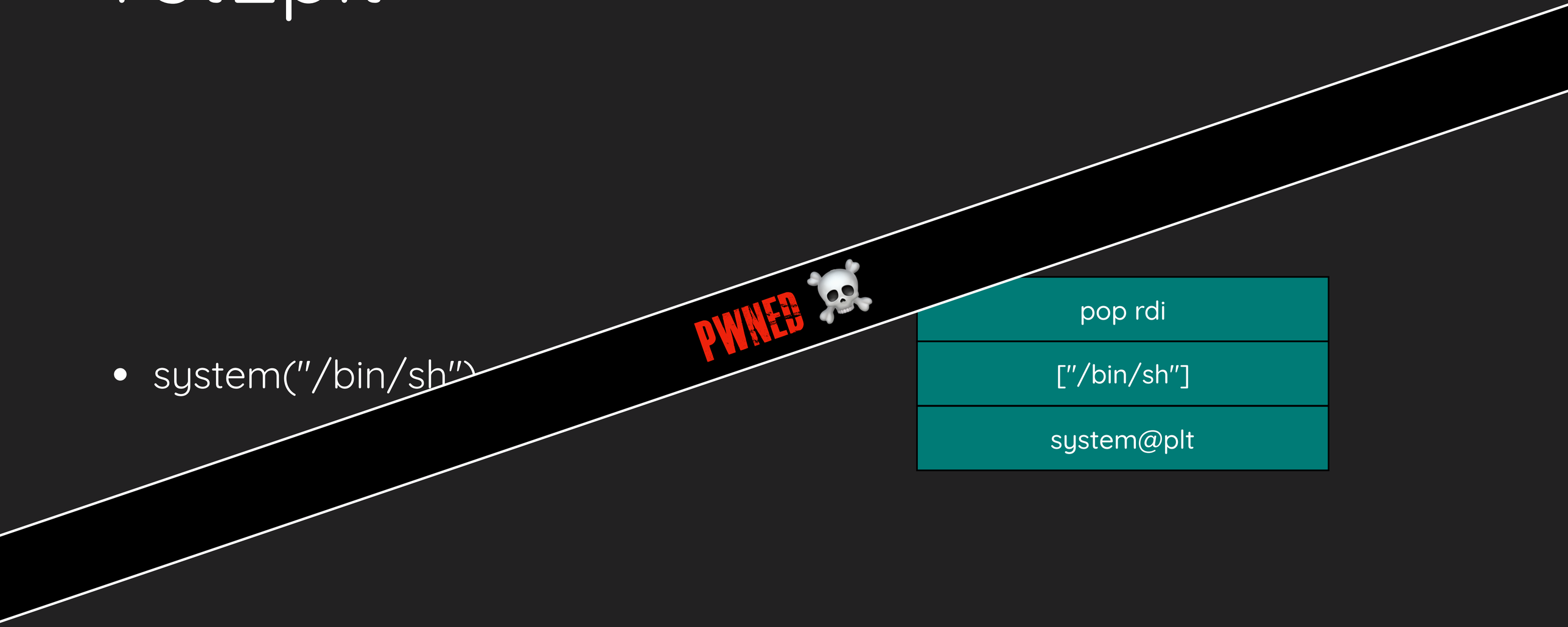
ret2plt

- system("/bin/sh")



ret2plt

- system("/bin/sh")



pop rdi
["/bin/sh"]
system@plt

Demo

ret2libc

return to libc

ret2libc

- Return to libc
- 倘若能得知 library 被 map 到的隨機起始地址 (base address)，則可以計算出 libc 中 function 的位置，便能調用 library 中的函式。

ret2libc

- 關鍵為 bypass ASLR，找出 libc 的隨機 base
  - 透過 information leak 漏洞，洩漏 memory 上的內容，獲取屬於 libc segment 的 address
  - 此 address 會是隨機的 base address 加上一固定位移植 offset (不同版本的 libc offset 不同)

ret2libc

- leaked_address - offset = base_address
- 例如，洩漏出 printf@got 中的內容為 0x7fd0f9e57e80，而已知 libc 版本為 2.27，可以透過靜態分析 (readelf -s 等) 得知 printf function 在 library 中的 offset 為 0x64e80，扣掉 offset 後求得此次執行 libc 的隨機 base address 是 0x7fd0f9df3000 = 0x7fd0f9e57e80 - 0x64e80
- 有 base 後就可以透過加上 offset 的方式得知其他 function 的 address，來 call 它，結合 ROP 等等。
  - system() = libc_base_address + system_offset
    = 0x7fd0f9df3000 + 0x4f440
    = 0x7fd0f9e42440

Demo

Demo

Information leak

Stack Pivoting

stack migration

Stack pivoting

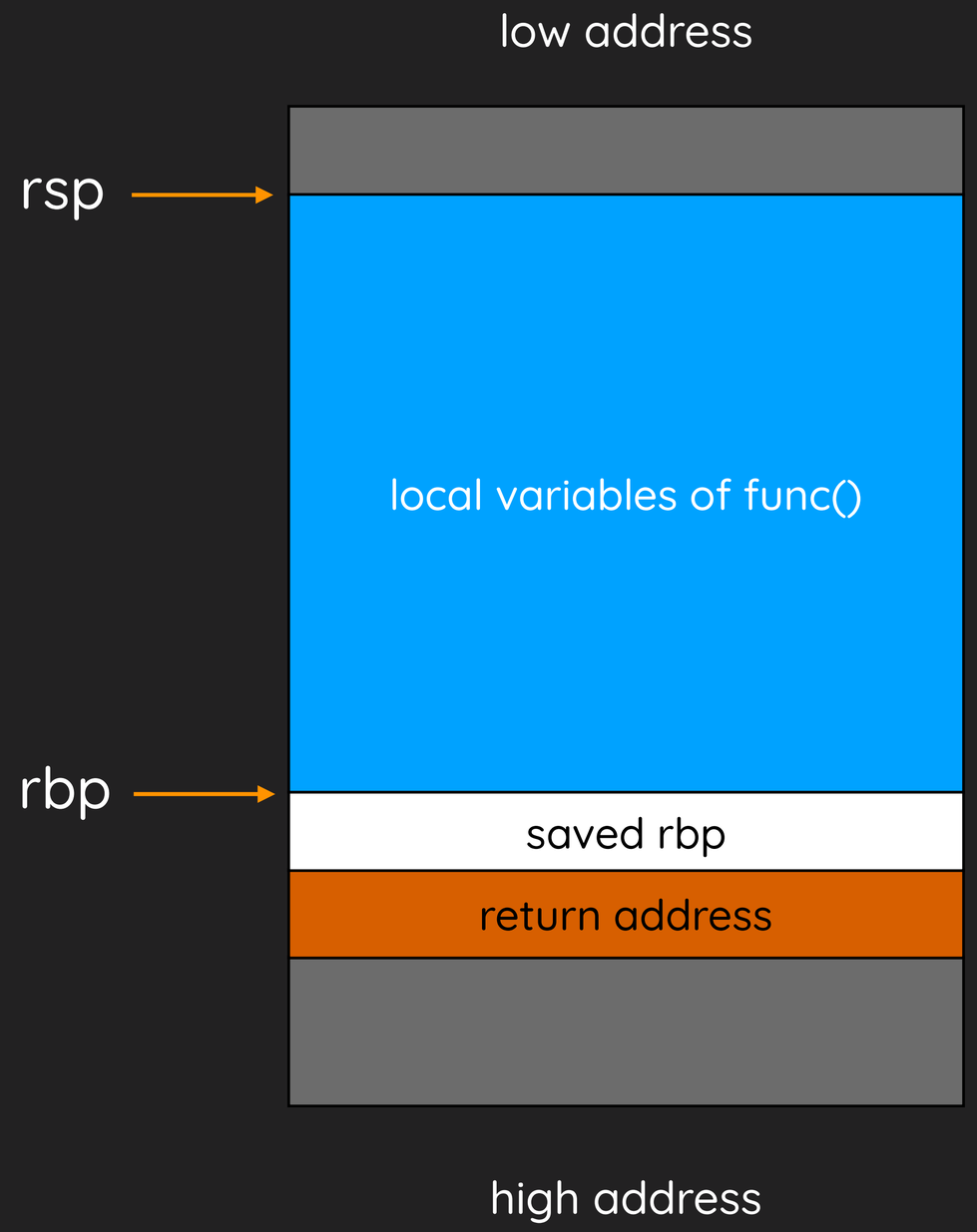
- ROP 需要很多的空間來放置長串的 ROP Chain，有時候並不會有那麼多的發揮空間，可能很短甚至只能控第一次 rip (function pointer 等等)，沒有足夠的空間存放 ROP payload。

- 倘若找到其他地方有足夠的空間放置 ROP payload，此時可以透過運用較少的 gadget 數，來將 stack 搬移至 ROP payload 的位置，再進行 ROP，此作法即為 stack pivoting 或 stack migration。

Stack pivoting

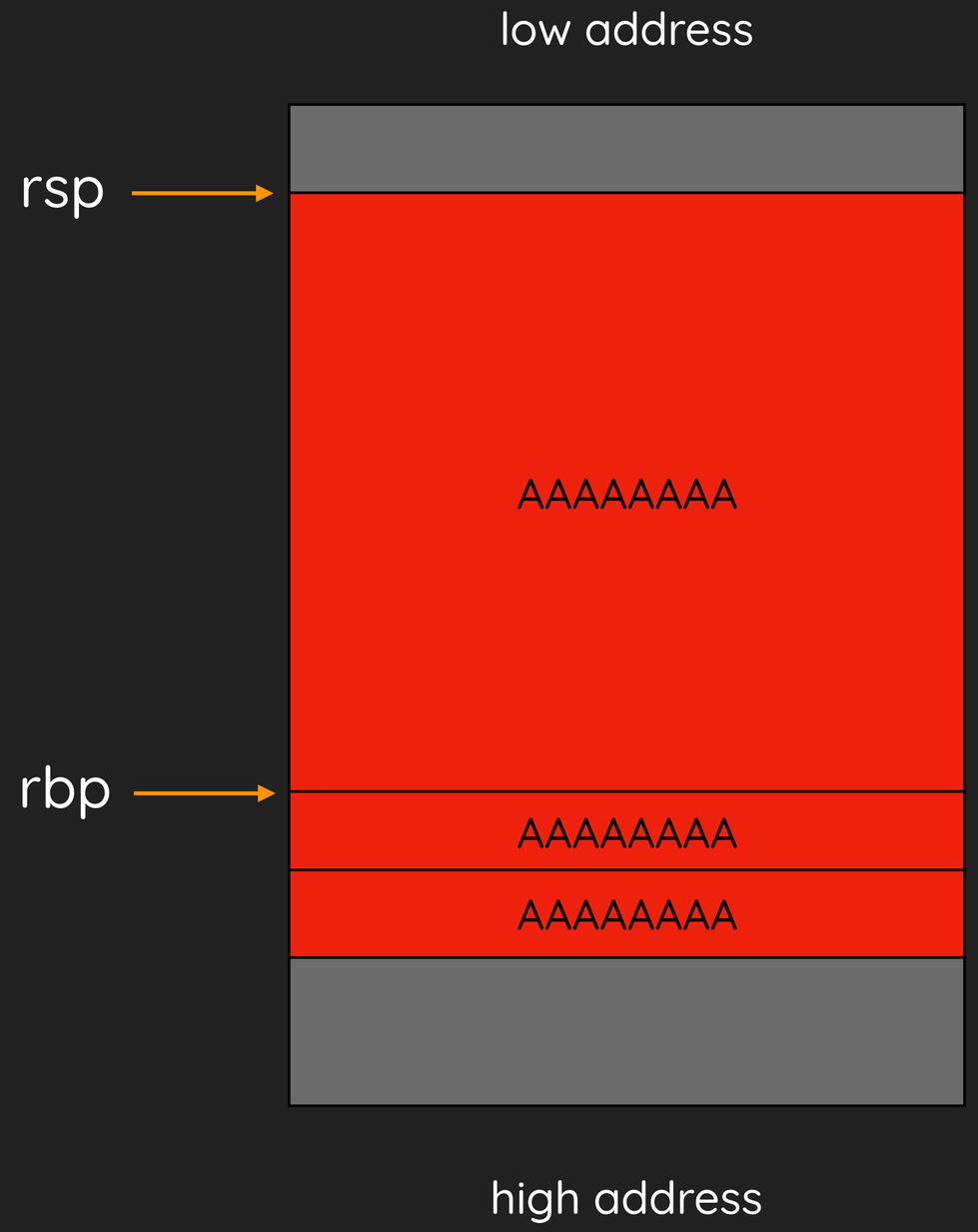
- 很多作法
  - leave; ret
    - overflow 時將 rbp 填成 ROP Chain 的 address - 8，
    - return address 填 leave; ret gadget
  - pop rsp; ret
  - 手動找針對各種當下情況的 gadget

Stack pivoting



Stack pivoting

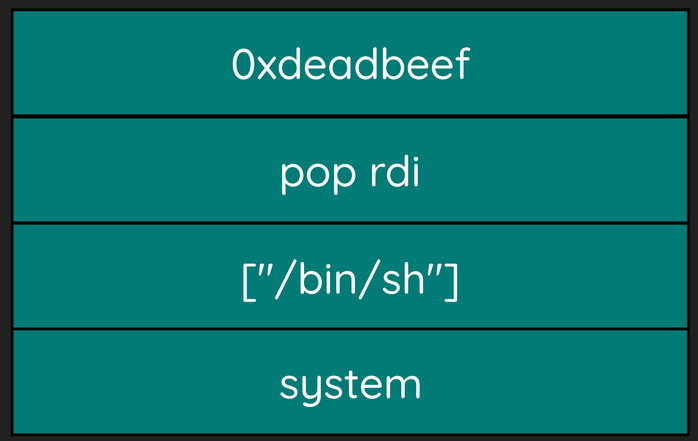
trigger 程式漏洞，僅能 overflow 16 bytes
恰只能剛好蓋到 return address

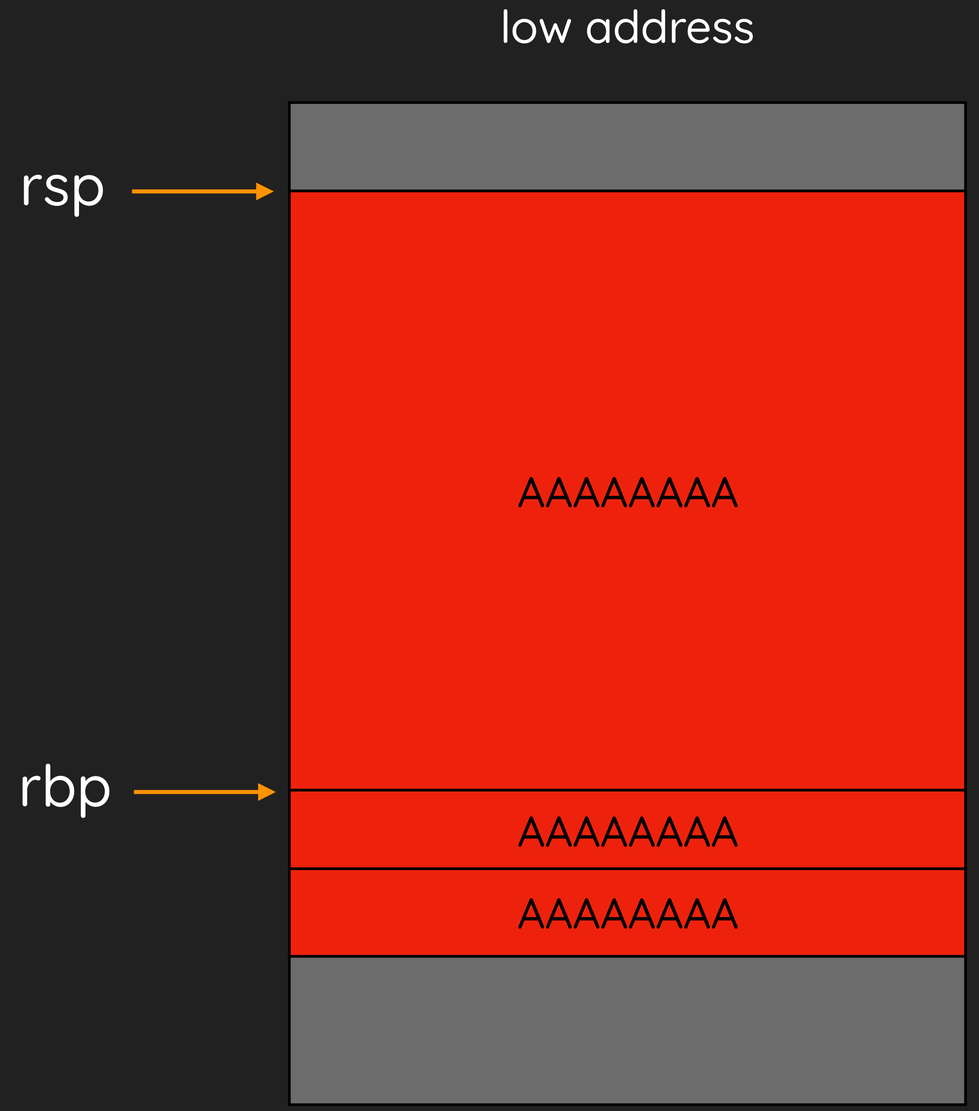


Stack pivoting

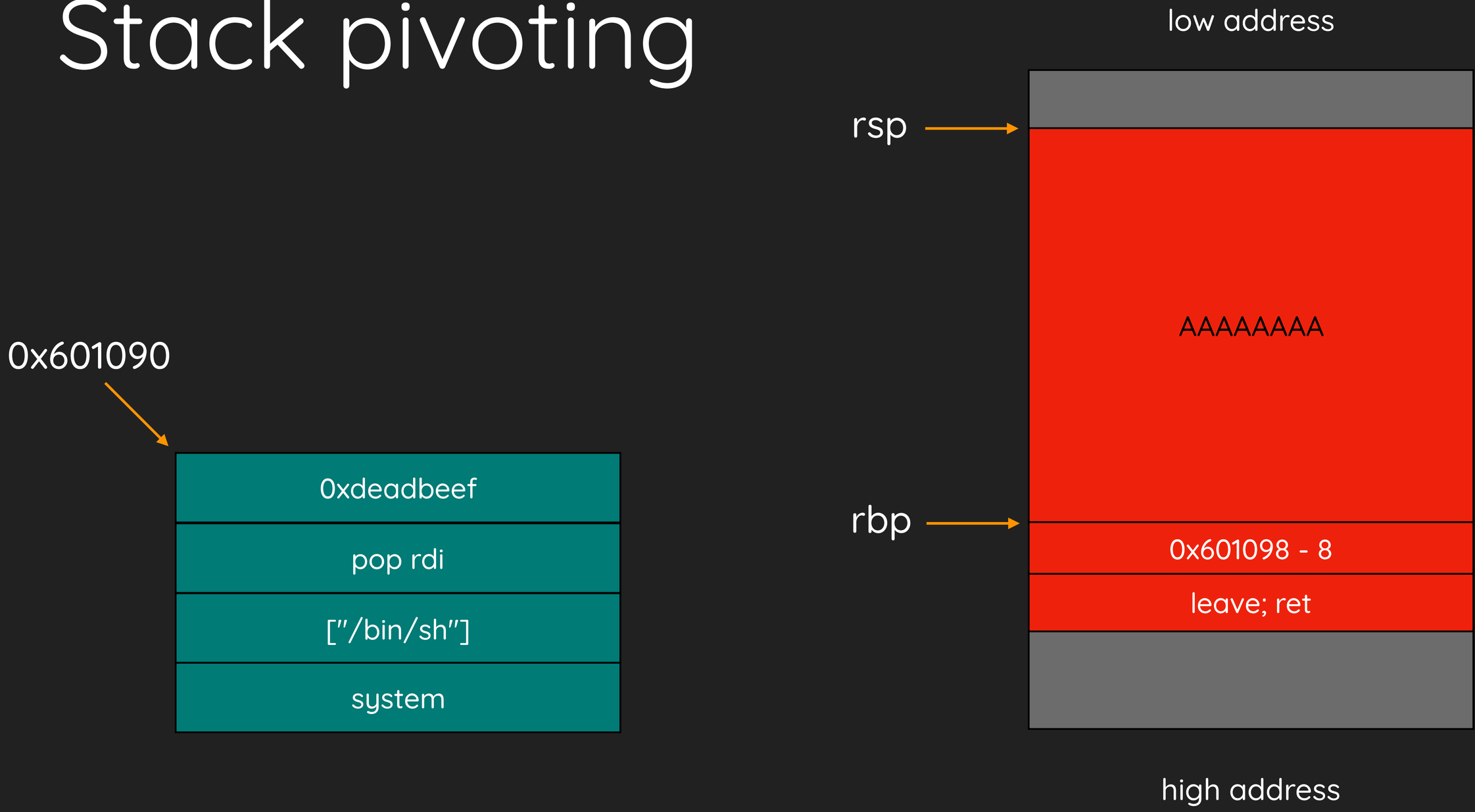
找到他處可以足夠放置 ROP payload 的空間

0x601090

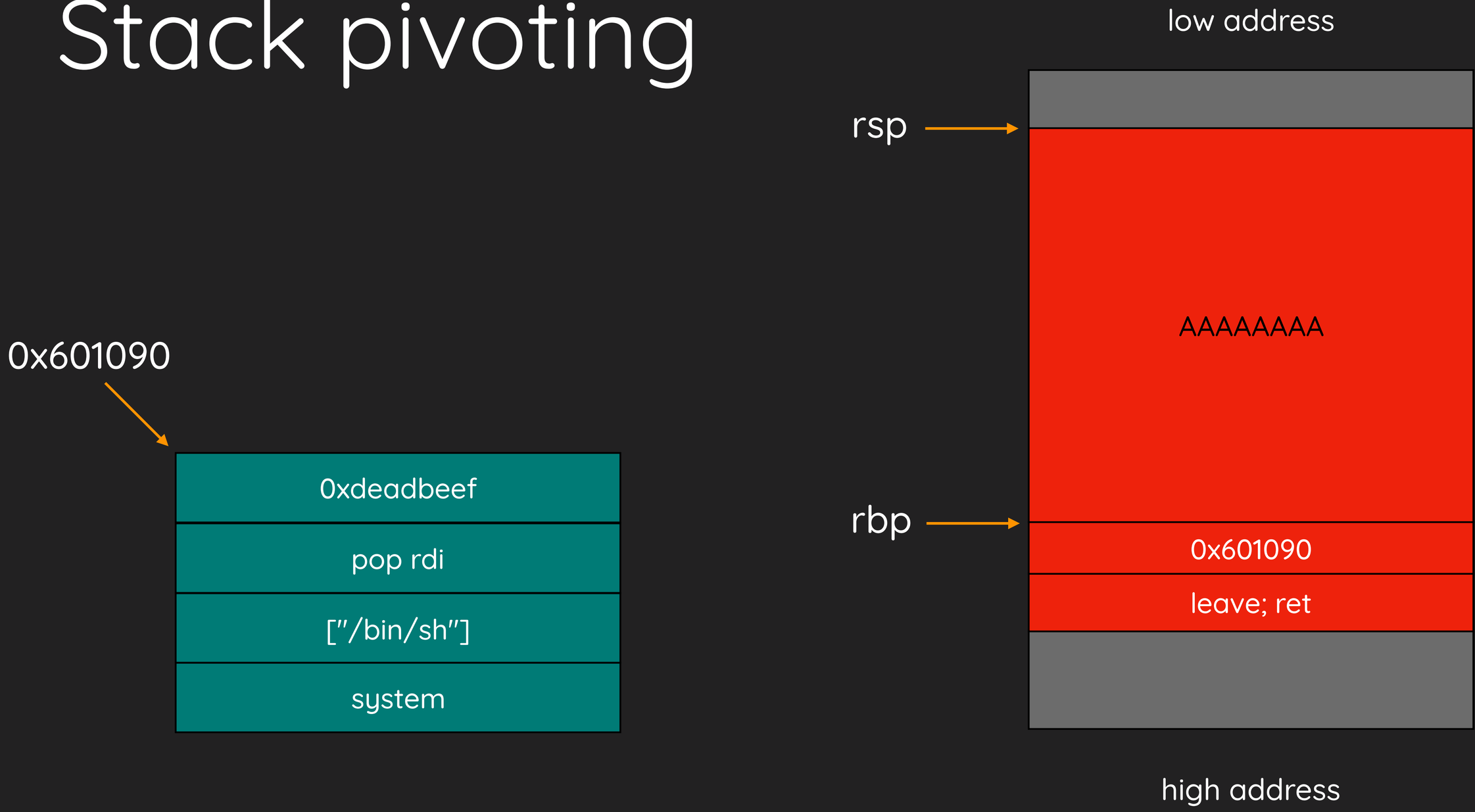




Stack pivoting



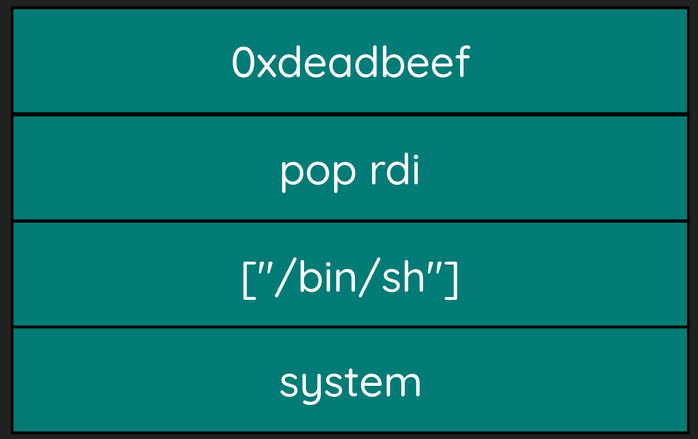
Stack pivoting

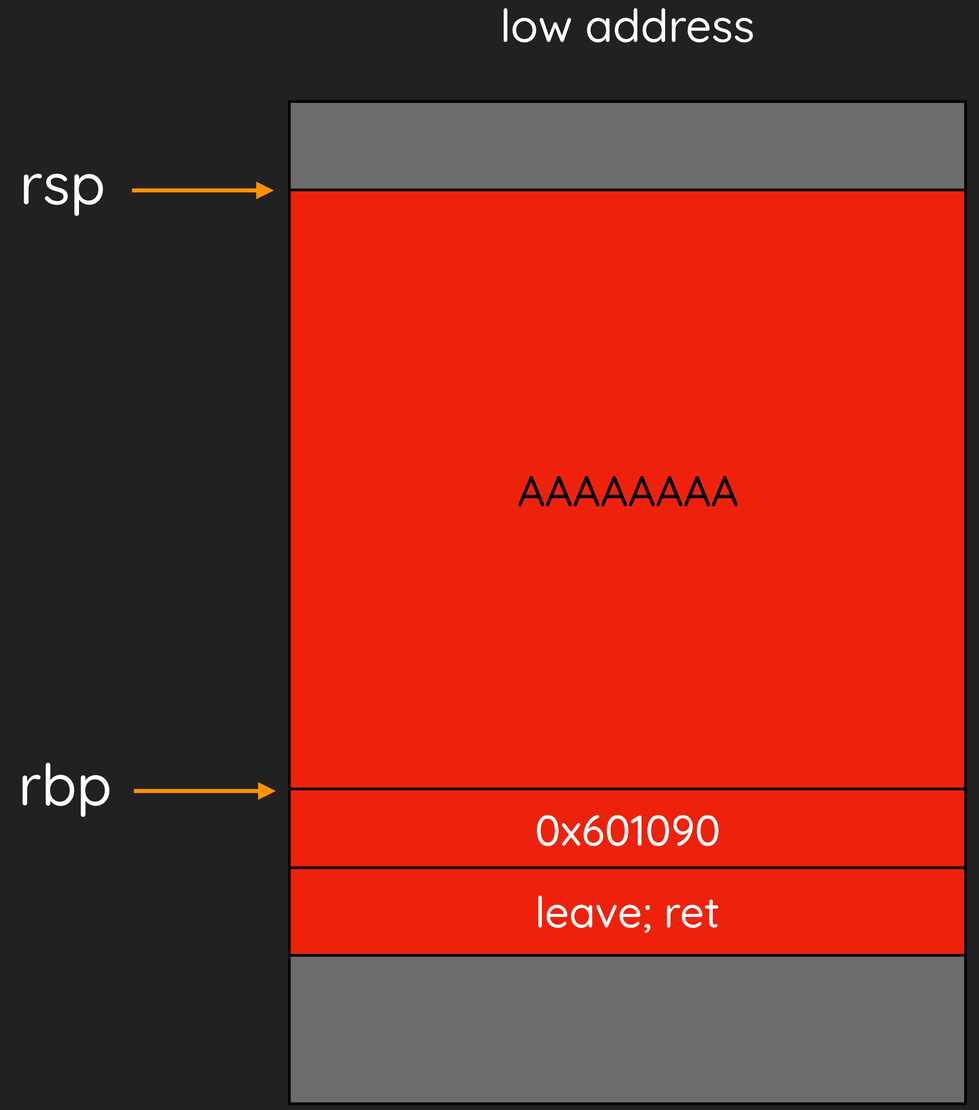


Stack pivoting

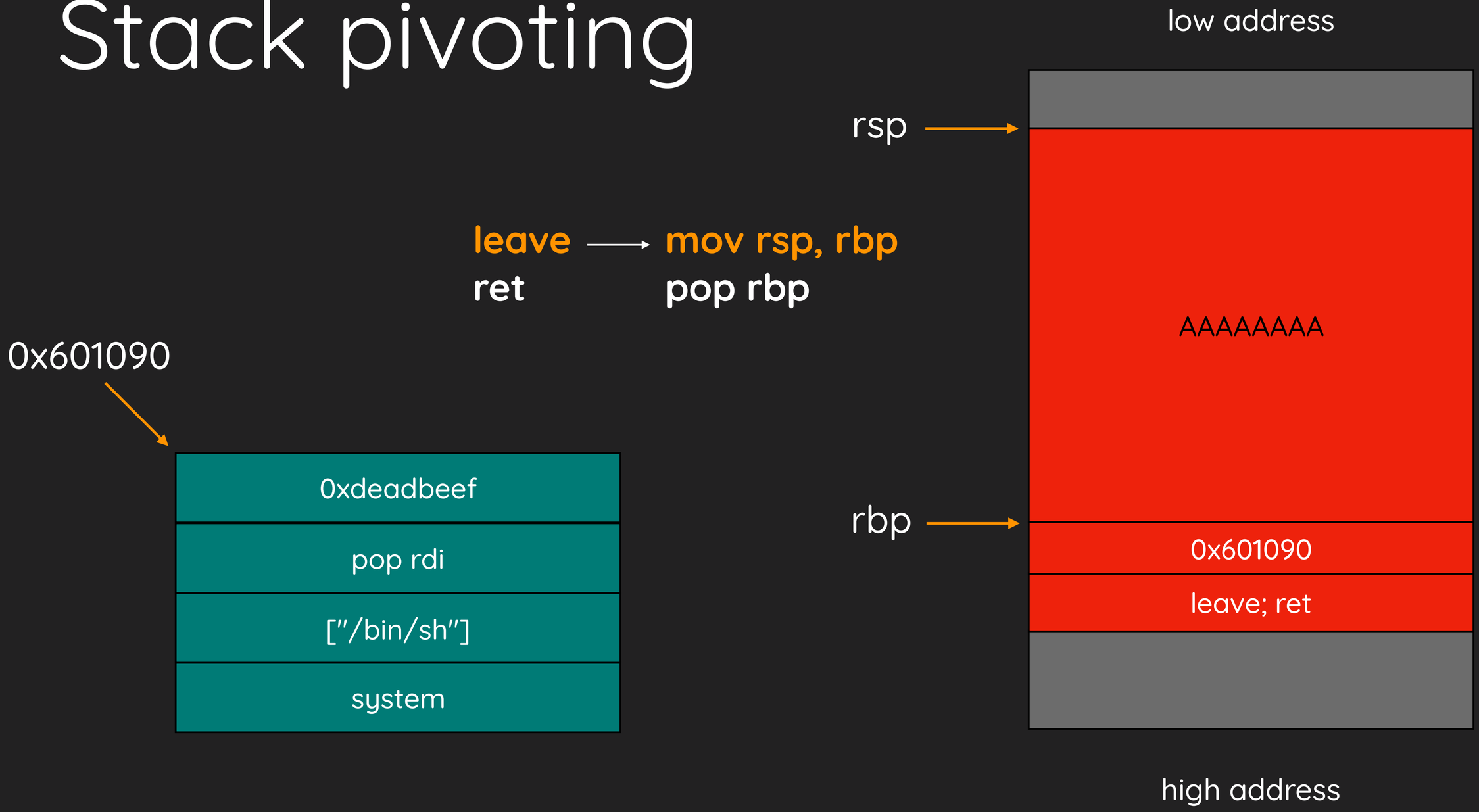
leave
ret

0x601090

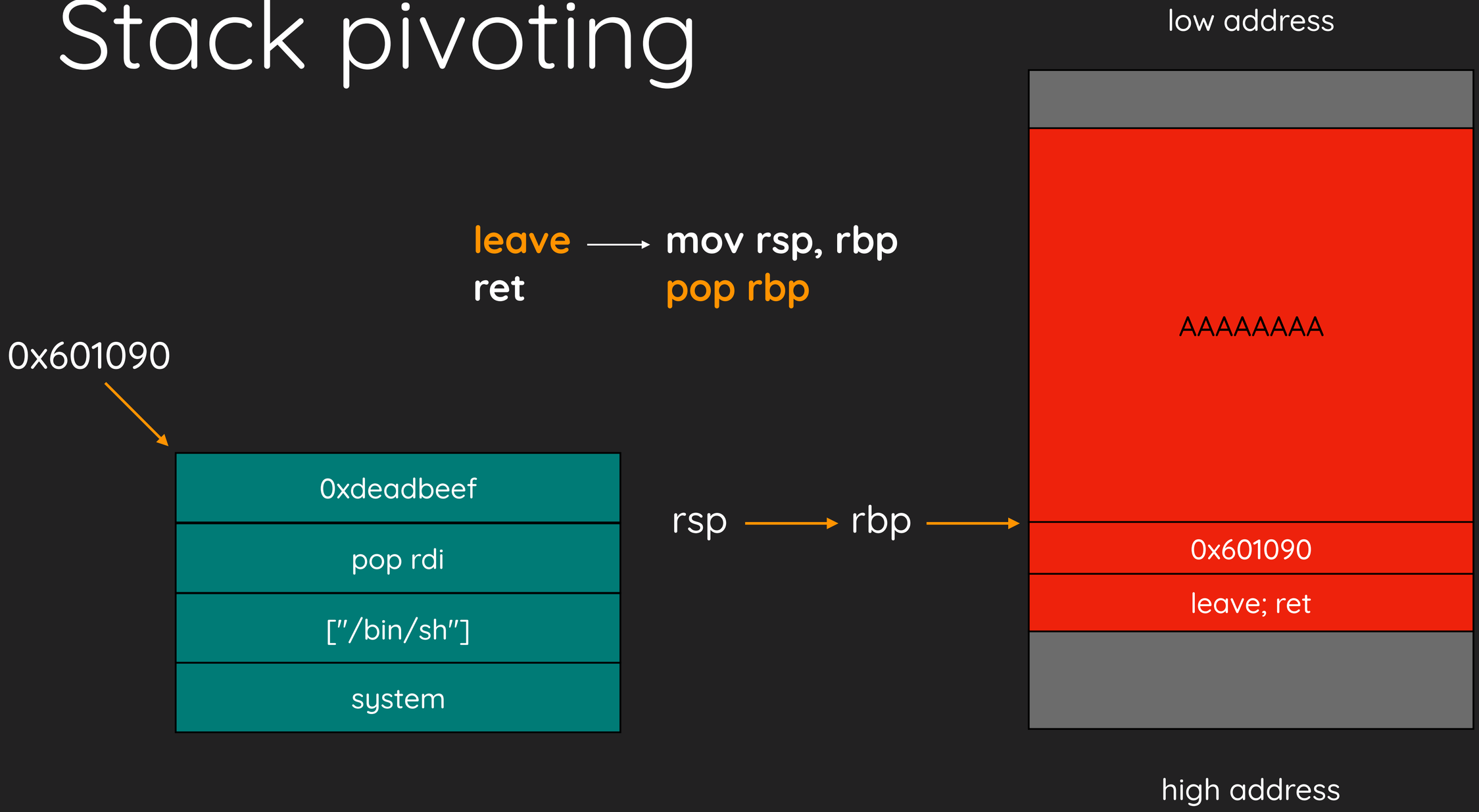




Stack pivoting

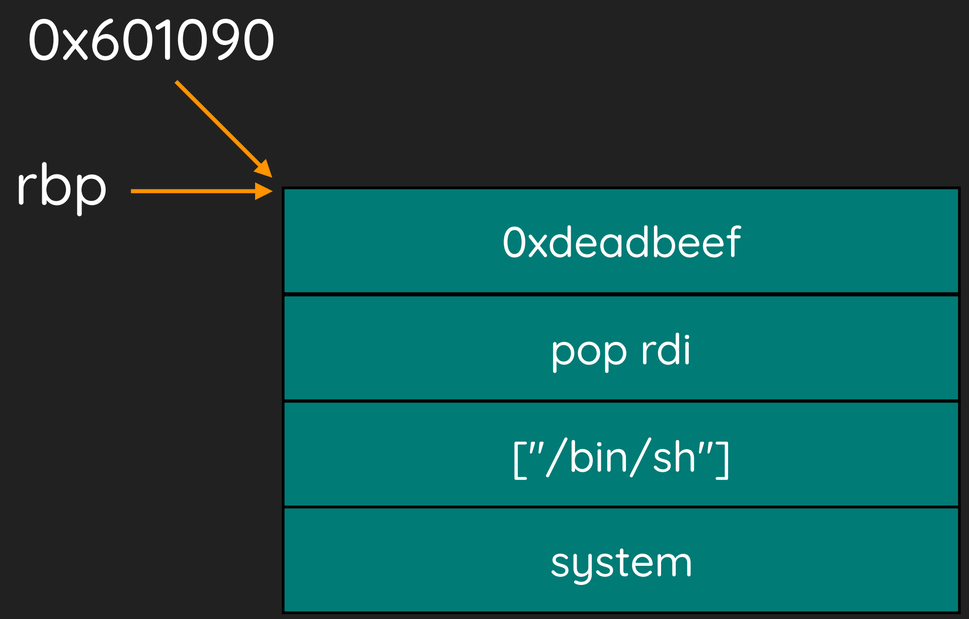


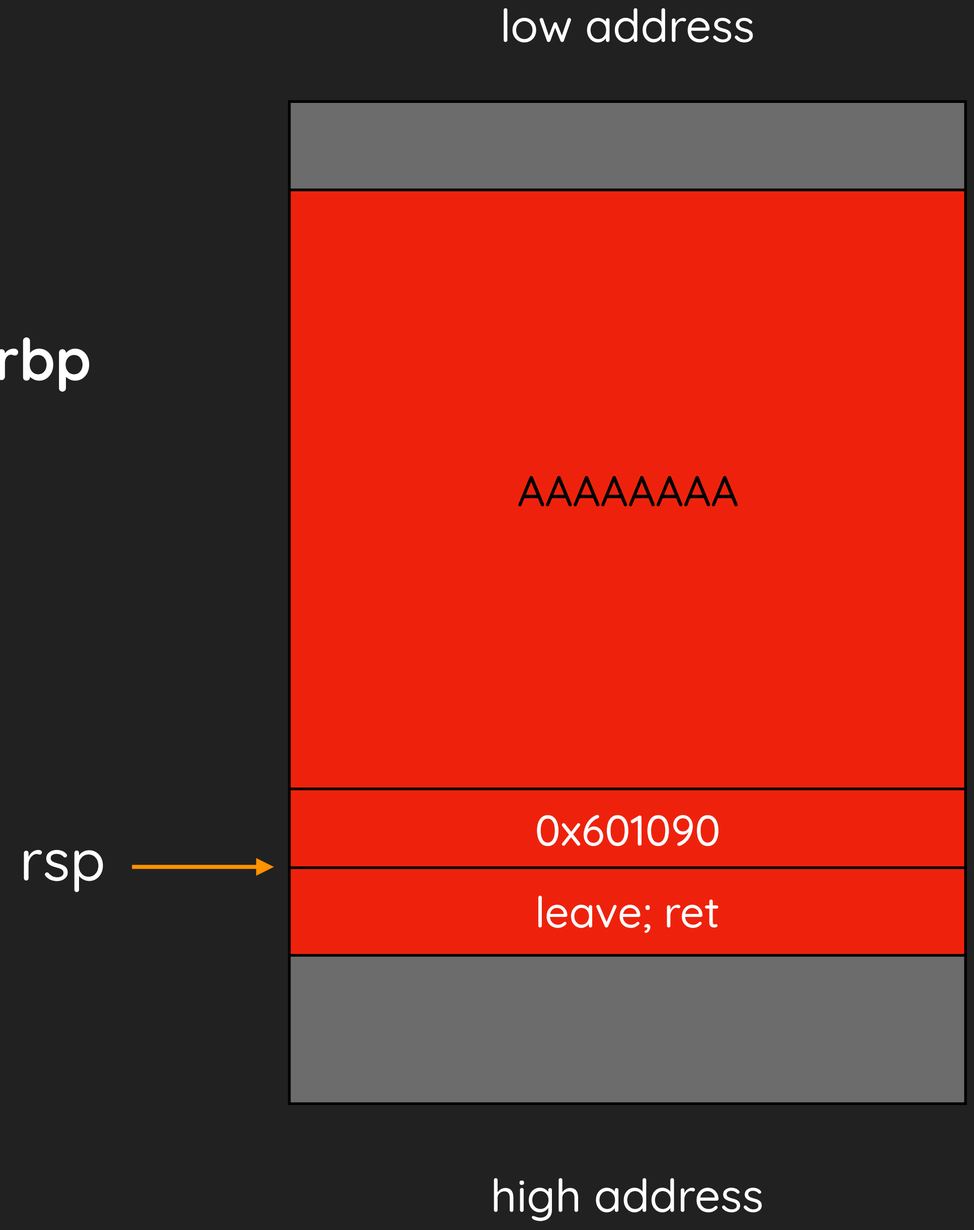
Stack pivoting



Stack pivoting

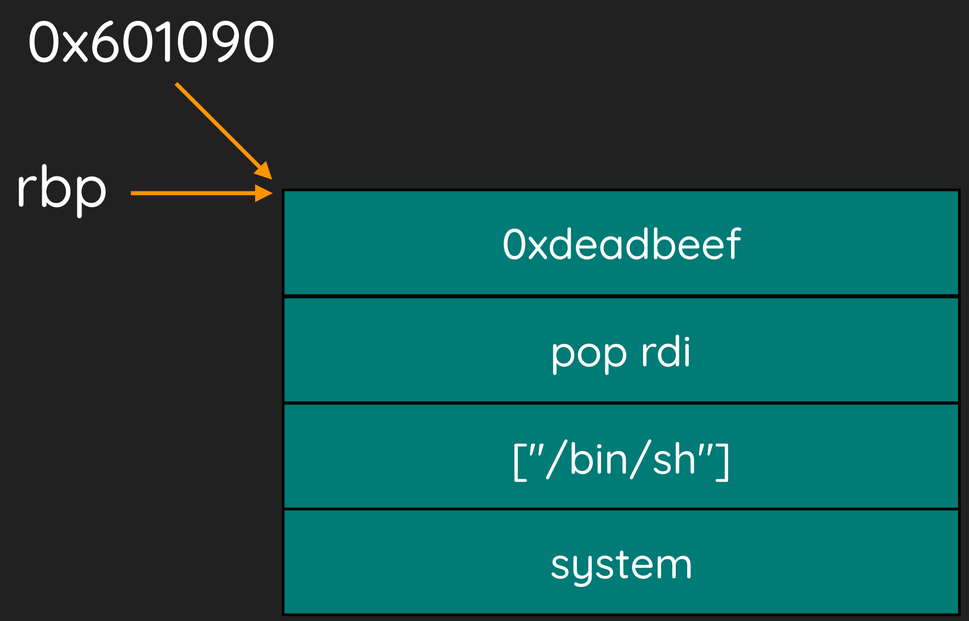
leave ⟶ **mov rsp, rbp**
ret **pop rbp**

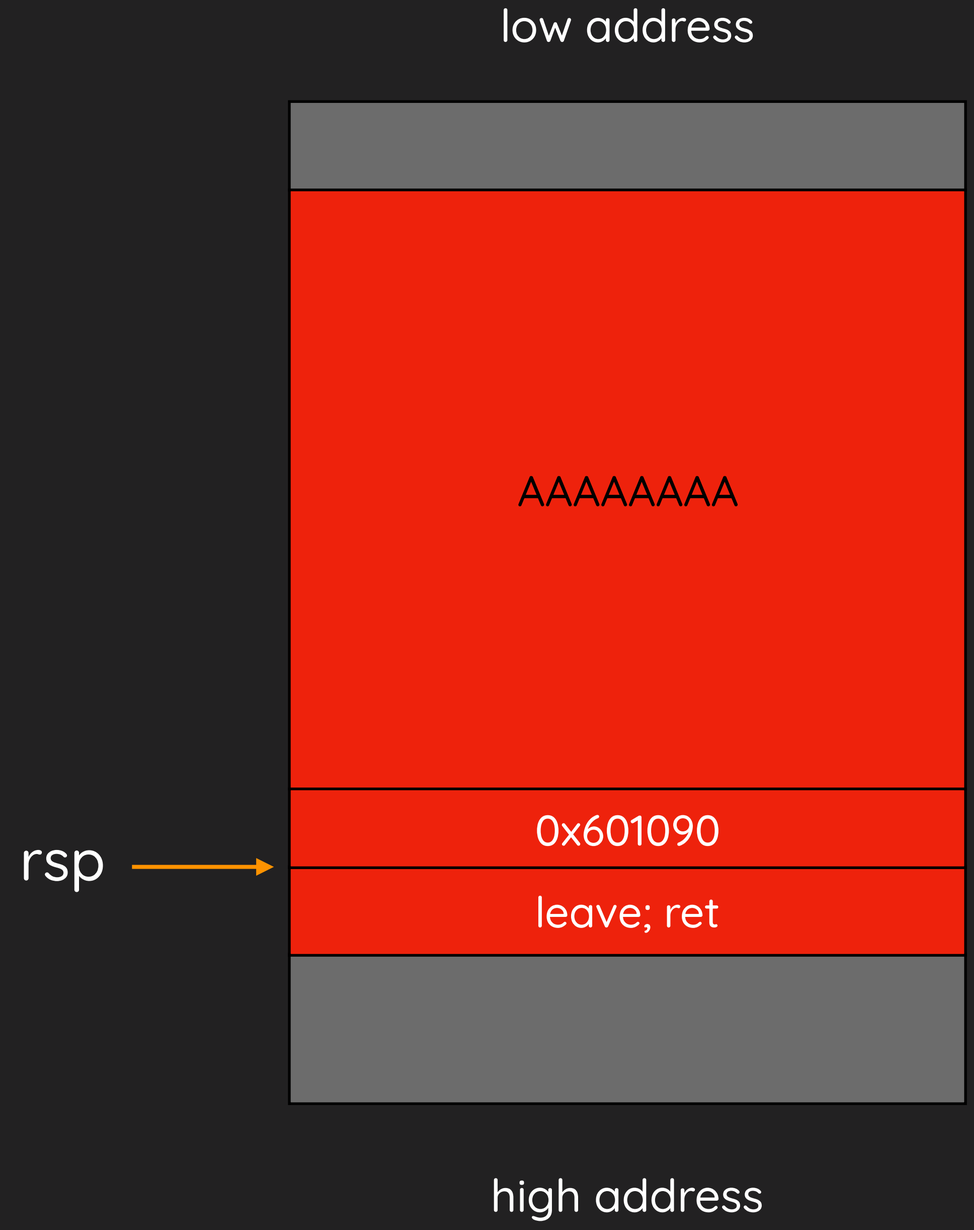




Stack pivoting

leave
ret



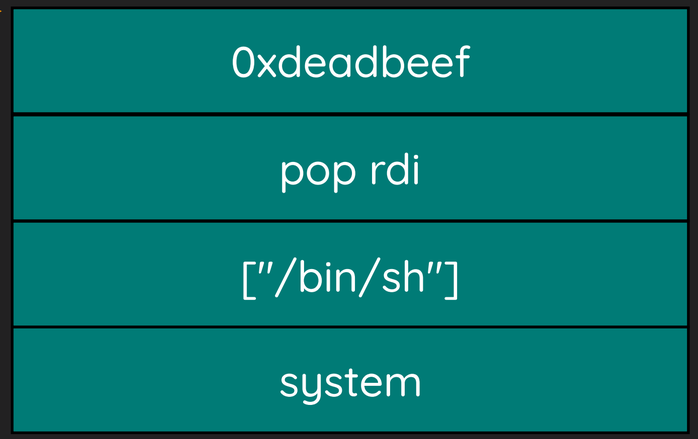


Stack pivoting

leave
ret

0x601090

rbp



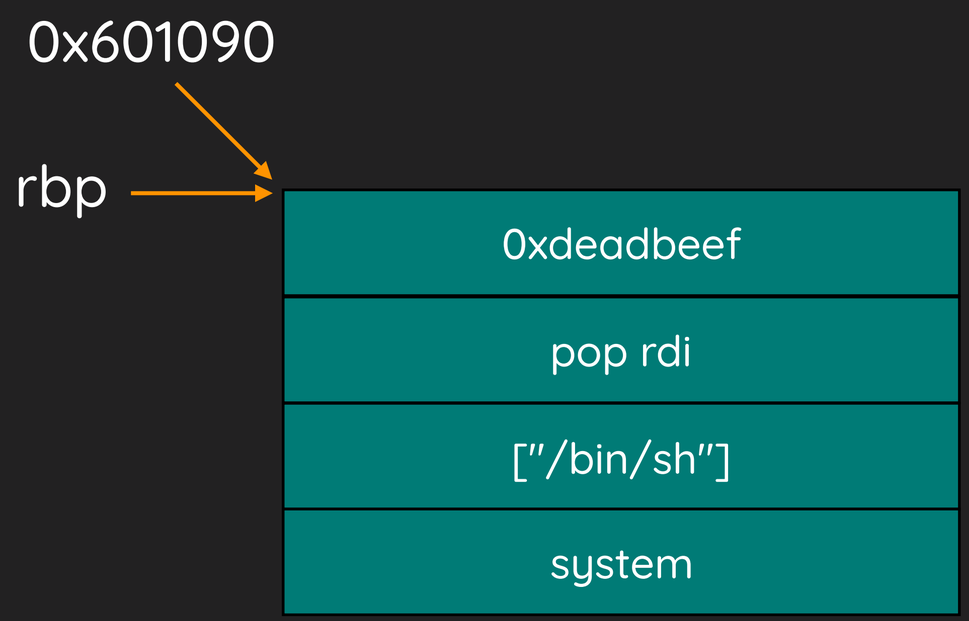
low address

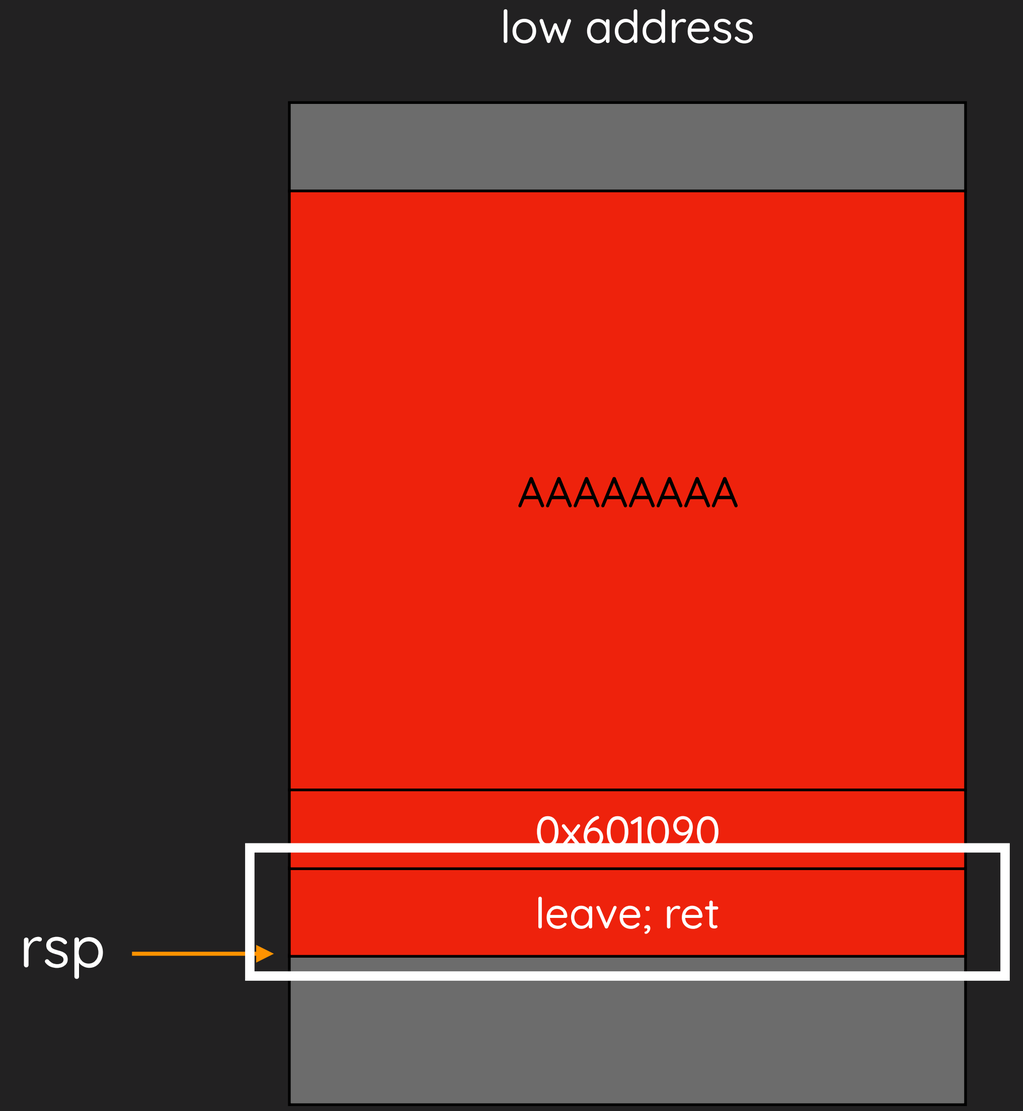
rsp

high address

Stack pivoting

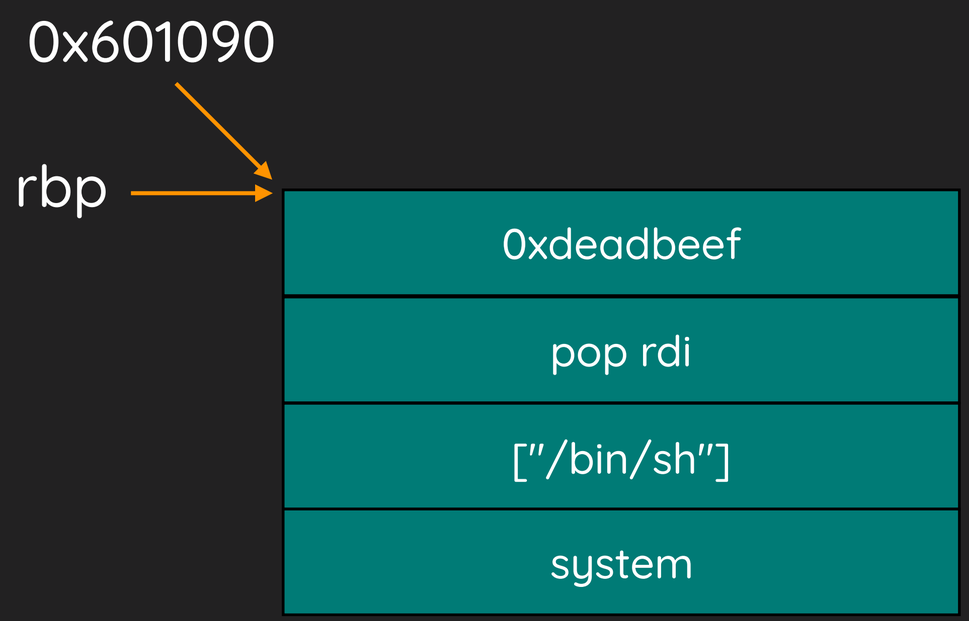
leave
ret





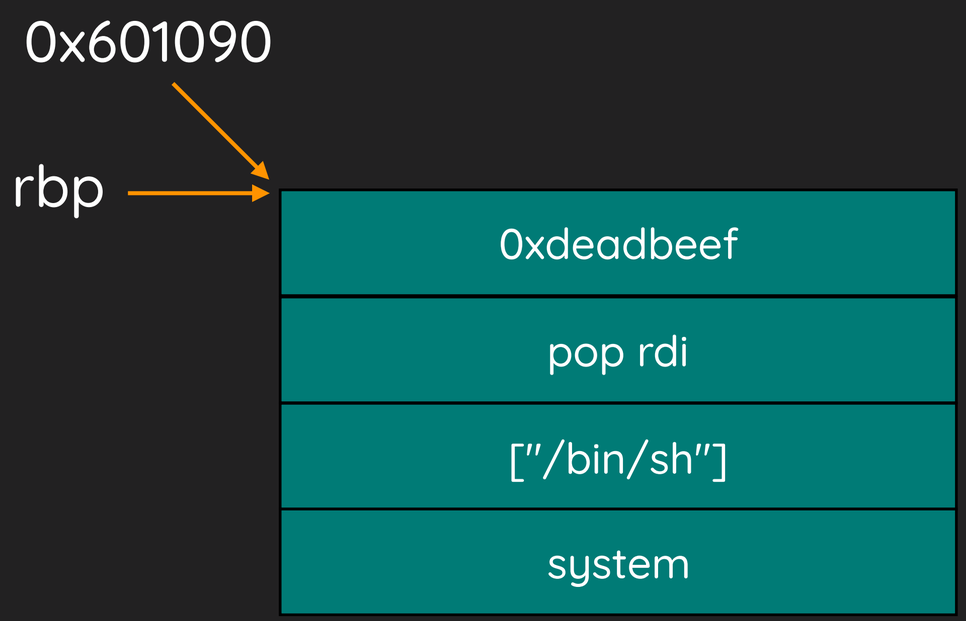
Stack pivoting

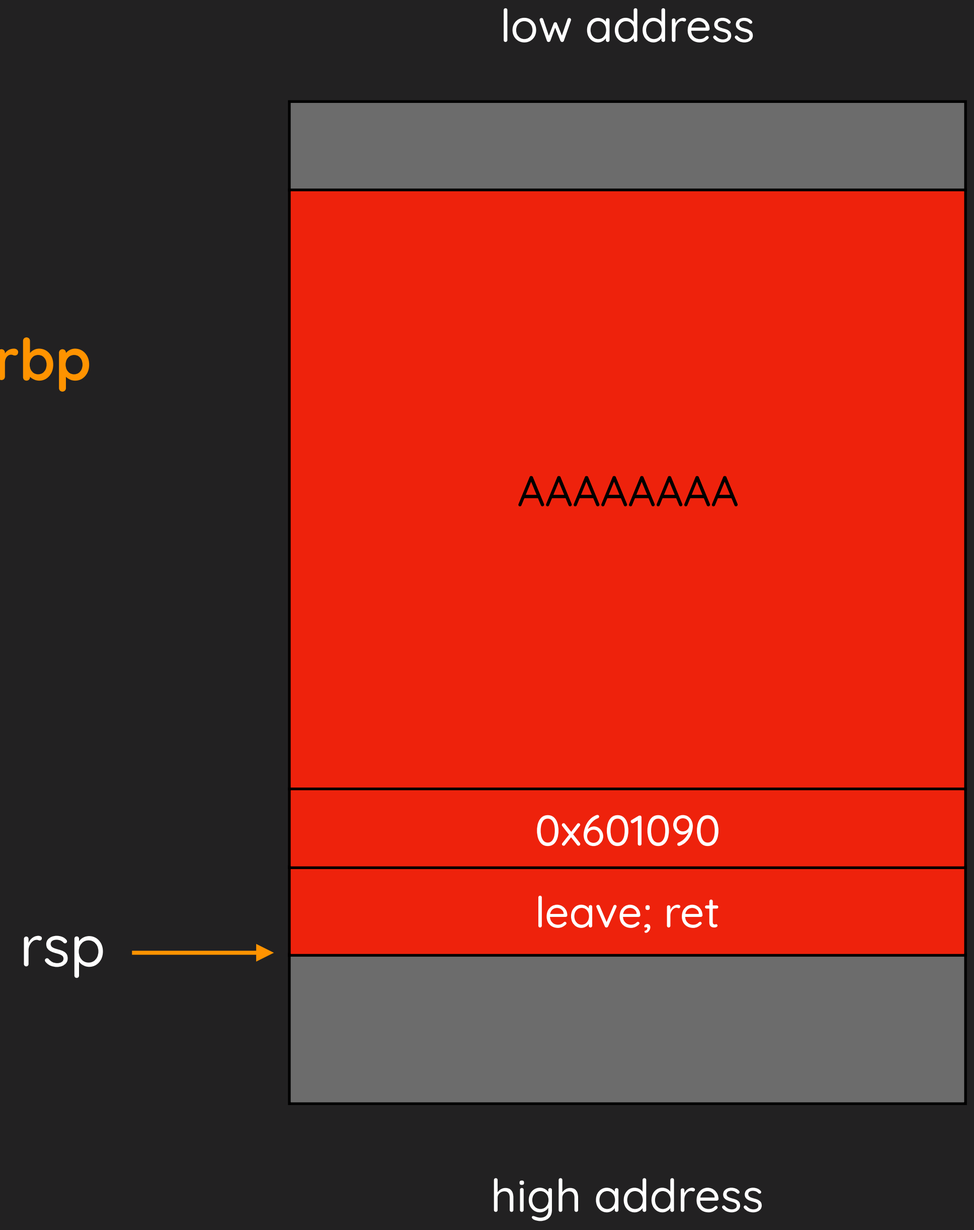
leave
ret



Stack pivoting

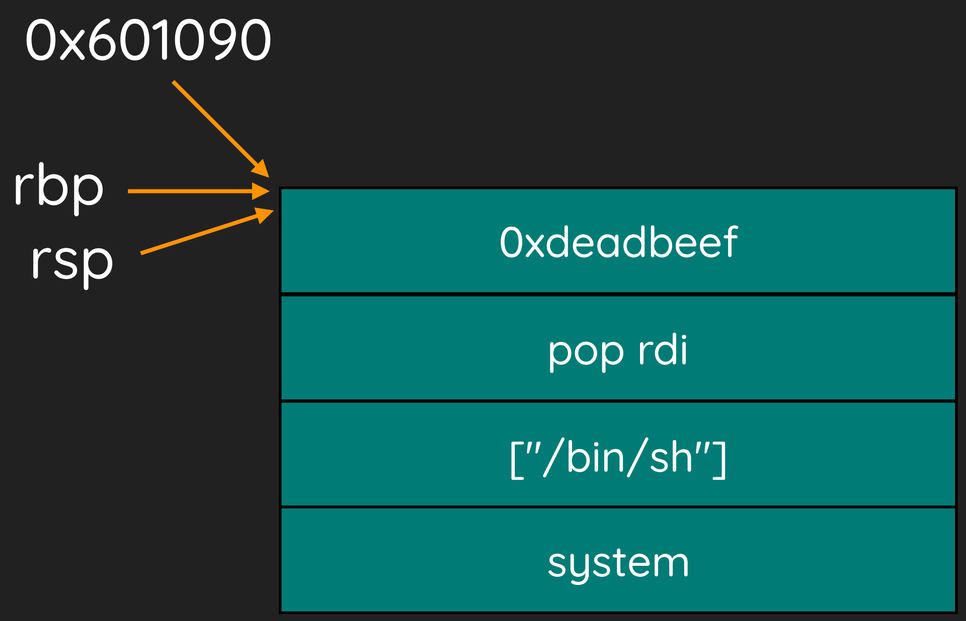
leave → **mov rsp, rbp**
ret **pop rbp**

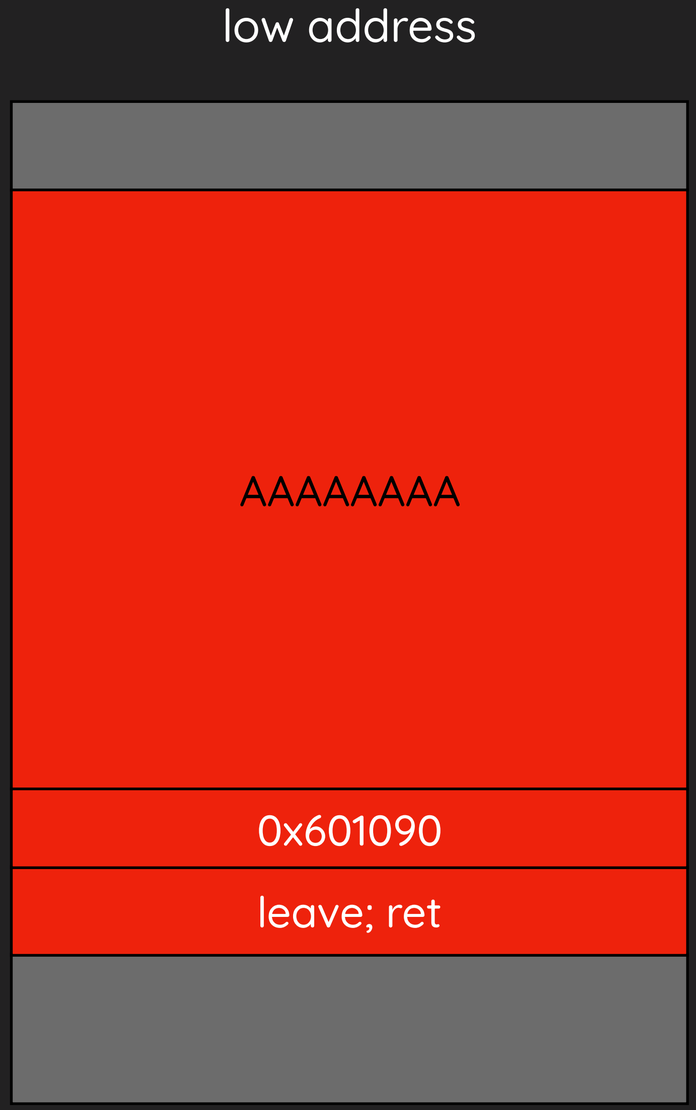




Stack pivoting

leave ⟶ **mov rsp, rbp**
ret **pop rbp**





Stack pivoting

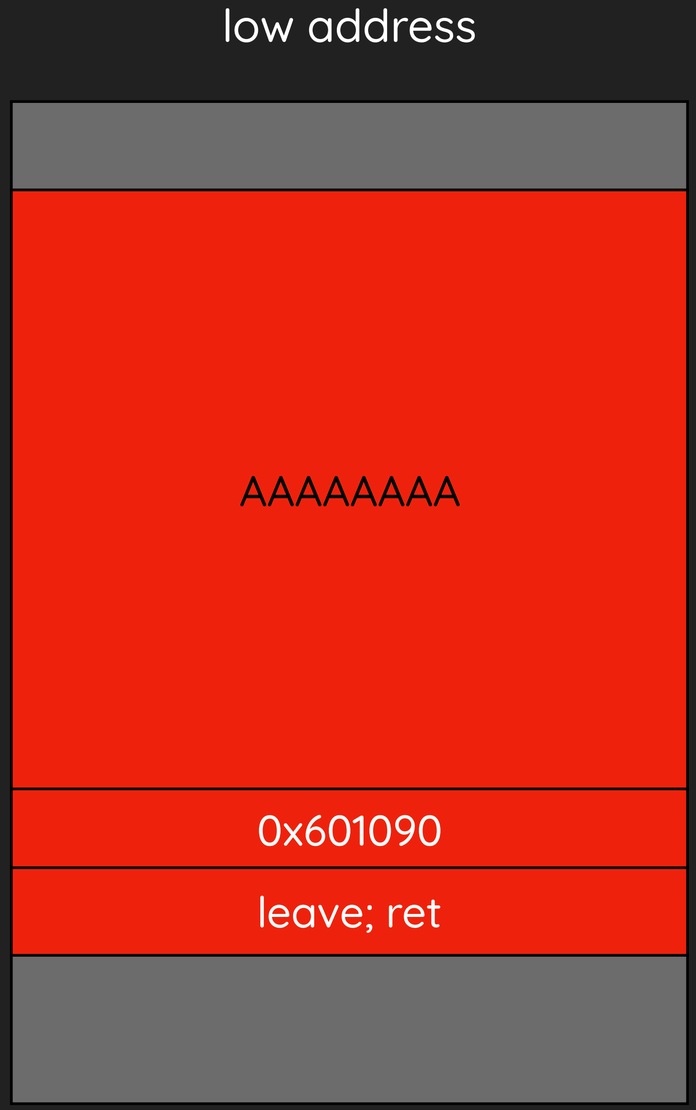
rbp → 0xdeadbeef

leave → **mov rsp, rbp**
ret **pop rbp**

0x601090

rsp →

0xdeadbeef
pop rdi
["/bin/sh"]
system



Stack pivoting

rbp → 0xdeadbeef

leave
ret

0x601090

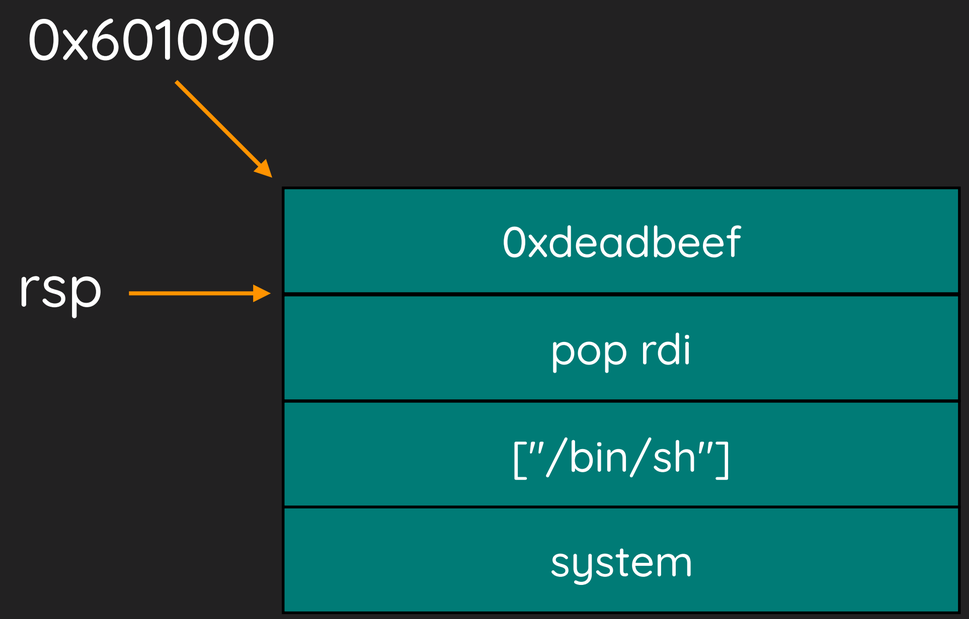
rsp →

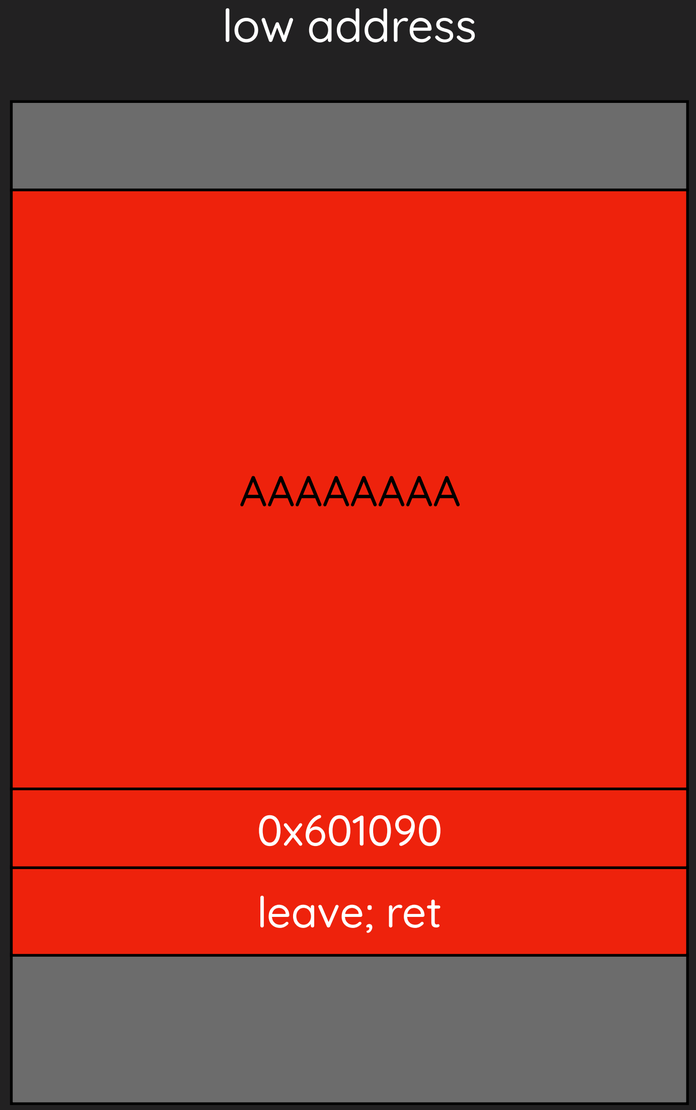
0xdeadbeef
pop rdi
["/bin/sh"]
system

Stack pivoting

rbp → 0xdeadbeef

ROP!!!!



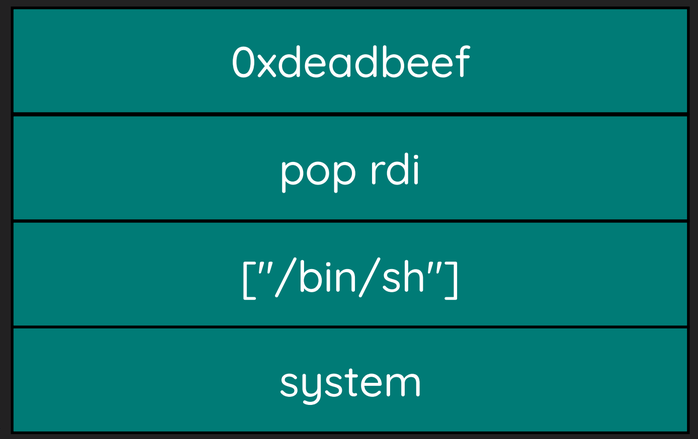


Stack pivoting

rbp → 0xdeadbeef

system("/bin/sh")

0x601090

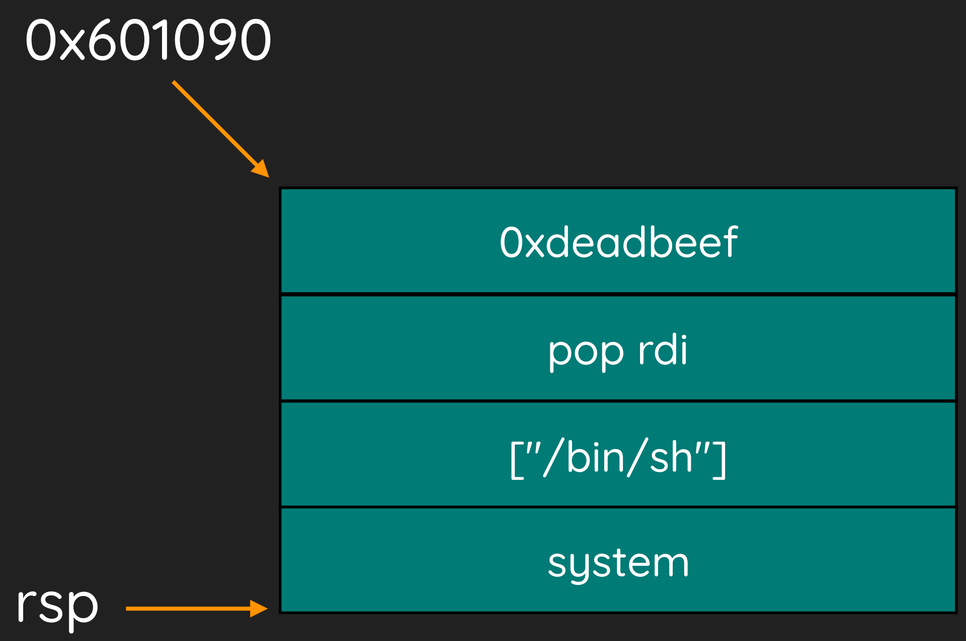


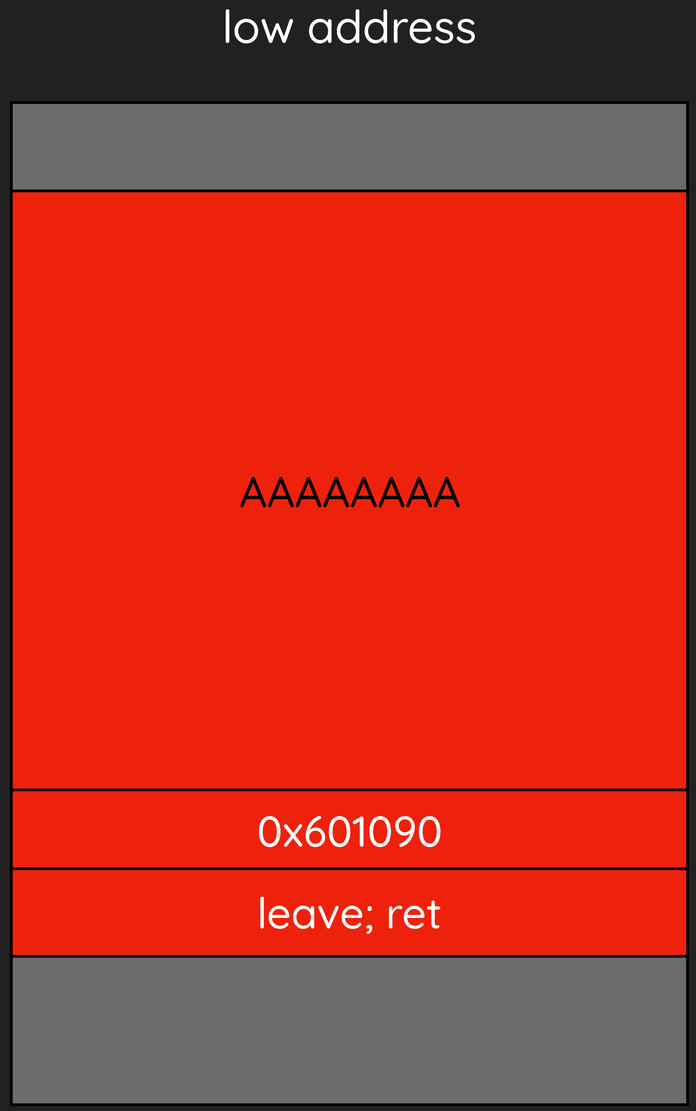
rsp →

Stack pivoting

rbp → 0xdeadbeef

Get shell!





Stack pivoting

rbp → 0xdeadbeef

low address

AAAAAAAA

0x601090

leave; ret

high address

Get shell!

PWNED

0x601090

0xdeadbeef

["/bin/sh"]

system

rsp

Demo

ret2csu

return to __libc_csu_init

ret2csu

- 位於 __libc_csu_init 函式中，為 compiler 編進去的 function。
- 尾部有一片段 code，很適合拿來控制 register 放置參數，以及 control flow。

ret2csu

- __libc_csu_init

```
00000000004006d0 <__libc_csu_init>:
  4006d0:    41 57                 push   r15
  4006d2:    41 56                 push   r14
  4006d4:    49 89 d7              mov    r15,rdx
  4006d7:    41 55                 push   r13
  4006d9:    41 54                 push   r12
  4006db:    4c 8d 25 2e 07 20 00  lea    r12,[rip+0x20072e]
  4006e2:    55                    push   rbp
  4006e3:    48 8d 2d 2e 07 20 00  lea    rbp,[rip+0x20072e]
  4006ea:    53                    push   rbx
  4006eb:    41 89 fd              mov    r13d,edi
  4006ee:    49 89 f6              mov    r14,rsi
  4006f1:    4c 29 e5              sub    rbp,r12
  4006f4:    48 83 ec 08           sub    rsp,0x8
  4006f8:    48 c1 fd 03           sar    rbp,0x3
  4006fc:    e8 f7 fd ff ff        call   4004f8 <_init>
  400701:    48 85 ed              test   rbp,rbp
  400704:    74 20                 je     400726 <__libc_csu_init+0x56>
  400706:    31 db                 xor    ebx,ebx
  400708:    0f 1f 84 00 00 00 00  nop    DWORD PTR [rax+rax*1+0x0]
  40070f:    00
  400710:    4c 89 fa              mov    rdx,r15
  400713:    4c 89 f6              mov    rsi,r14
  400716:    44 89 ef              mov    edi,r13d
  400719:    41 ff 14 dc           call   QWORD PTR [r12+rbx*8]
  40071d:    48 83 c3 01           add    rbx,0x1
  400721:    48 39 dd              cmp    rbp,rbx
  400724:    75 ea                 jne    400710 <__libc_csu_init+0x40>
  400726:    48 83 c4 08           add    rsp,0x8
  40072a:    5b                    pop    rbx
  40072b:    5d                    pop    rbp
  40072c:    41 5c                 pop    r12
  40072e:    41 5d                 pop    r13
  400730:    41 5e                 pop    r14
  400732:    41 5f                 pop    r15
  400734:    c3                    ret
```

ret2csu

- gadget

```
00000000004006d0 <__libc_csu_init>:
  4006d0:    41 57                 push   r15
  4006d2:    41 56                 push   r14
  4006d4:    49 89 d7              mov    r15,rdx
  4006d7:    41 55                 push   r13
  4006d9:    41 54                 push   r12
  4006db:    4c 8d 25 2e 07 20 00  lea    r12,[rip+0x20072e]
  4006e2:    55                    push   rbp
  4006e3:    48 8d 2d 2e 07 20 00  lea    rbp,[rip+0x20072e]
  4006ea:    53                    push   rbx
  4006eb:    41 89 fd              mov    r13d,edi
  4006ee:    49 89 f6              mov    r14,rsi
  4006f1:    4c 29 e5              sub    rbp,r12
  4006f4:    48 83 ec 08           sub    rsp,0x8
  4006f8:    48 c1 fd 03           sar    rbp,0x3
  4006fc:    e8 f7 fd ff ff        call   4004f8 <_init>
  400701:    48 85 ed              test   rbp,rbp
  400704:    74 20                 je     400726 <__libc_csu_init+0x56>
  400706:    31 db                 xor    ebx,ebx
  400708:    0f 1f 84 00 00 00 00  nop    DWORD PTR [rax+rax*1+0x0]
  40070f:    00
  400710:    4c 89 fa              mov    rdx,r15
  400713:    4c 89 f6              mov    rsi,r14
  400716:    44 89 ef              mov    edi,r13d
  400719:    41 ff 14 dc           call   QWORD PTR [r12+rbx*8]
  40071d:    48 83 c3 01           add    rbx,0x1
  400721:    48 39 dd              cmp    rbp,rbx
  400724:    75 ea                 jne    400710 <__libc_csu_init+0x40>
  400726:    48 83 c4 08           add    rsp,0x8
  40072a:    5b                    pop    rbx
  40072b:    5d                    pop    rbp
  40072c:    41 5c                 pop    r12
  40072e:    41 5d                 pop    r13
  400730:    41 5e                 pop    r14
  400732:    41 5f                 pop    r15
  400734:    c3                    ret
```

ret2csu

```
400710: 4c 89 fa              mov    rdx,r15
400713: 4c 89 f6              mov    rsi,r14
400716: 44 89 ef              mov    edi,r13d
400719: 41 ff 14 dc           call   QWORD PTR [r12+rbx*8]
40071d: 48 83 c3 01           add    rbx,0x1
400721: 48 39 dd              cmp    rbp,rbx
400724: 75 ea                 jne    400710 <__libc_csu_init+0x40>
400726: 48 83 c4 08           add    rsp,0x8
40072a: 5b                    pop    rbx
40072b: 5d                    pop    rbp
40072c: 41 5c                 pop    r12
40072e: 41 5d                 pop    r13
400730: 41 5e                 pop    r14
400732: 41 5f                 pop    r15
400734: c3                    ret
```

ret2csu

- 透過控制 rbp rbx r12 r13 r14 r15 registers 的值，跳至 gadget 開頭，r13 r14 r15，分別放置前三個參數 rdi rsi rdx，此部分解決了很少找到 pop rdx gadget，ROP 很難控制第三個參數的問題。

- 控制 r12 rbx 來指定任意記憶體位置 call [r12+rbx*8]。

- 將 rbx 設為 0，將 rbp 設為 1，在 call 完後使 rbx == rbp == 1，jne 不會 take，而繼續執行後面的連續 pop register，如此可重複使用，達到任意 ROP。

ret2csu

```
400710: 4c 89 fa          mov    rdx,r15
400713: 4c 89 f6          mov    rsi,r14
400716: 44 89 ef          mov    edi,r13d
400719: 41 ff 14 dc       call   QWORD PTR [r12+rbx*8]
40071d: 48 83 c3 01       add    rbx,0x1
400721: 48 39 dd          cmp    rbp,rbx
400724: 75 ea             jne    400710 <__libc_csu_init+0x40>
400726: 48 83 c4 08       add    rsp,0x8
40072a: 5b                pop    rbx
40072b: 5d                pop    rbp
40072c: 41 5c             pop    r12
40072e: 41 5d             pop    r13
400730: 41 5e             pop    r14
400732: 41 5f             pop    r15
400734: c3                ret
```

Casino++

HW

HW - Casino++

- Same source code.
- NX enabled
- Just pwn it again!

Thanks!

 _yuawn

 yuawn