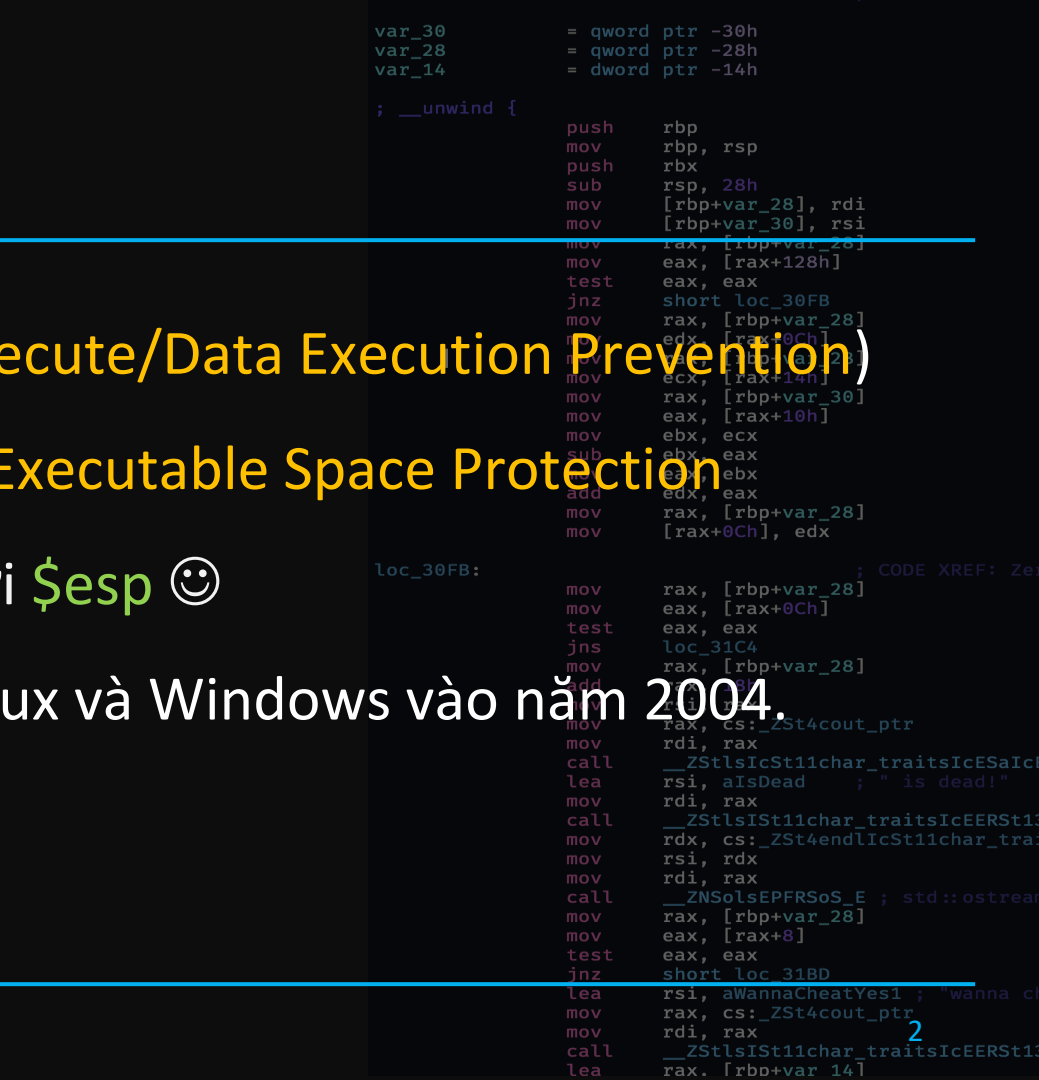
DEP & ROP


var_30 = qword ptr -30h
var_28 = qword ptr -28h
var_14 = dword ptr -14h
; __unwind {
push rbp
mov rbp, rsp
push rbx
sub rsp, 28h
mov [rbp+var_28], rdi
mov [rbp+var_30], rsi
mov rax, [rbp+var_28]
mov eax, [rax+128h]
test eax, eax
jnz short loc_30FB
mov rax, [rbp+var_28]
mov edx, [rax+0Ch]
mov rax, [rbp+var_28]
mov ecx, [rax+14h]
mov rax, [rbp+var_30]
mov eax, [rax+10h]
mov ebx, ecx
sub ebx, eax
mov eax, ebx
add edx, eax
mov rax, [rbp+var_28]
mov [rax+0Ch], edx
loc_30FB:
mov eax, [rax+0Ch]
test eax, eax
jns loc_31C4
mov rax, [rbp+var_28]
add rax, 18h
mov rsi, rax
mov rax, cs:_ZSt4cout_ptr
mov rdi, rax
lea rsi, aIsDead ; " is dead!"
mov rdi, rax
mov rdx, cs:_ZSt4endlIcSt11char_traits
mov rsi, rdx
mov rdi, rax
call __ZNSolsEPFRSoS_E ; std::ostrea
mov rax, [rbp+var_28]
mov eax, [rax+8]
test eax, eax
jnz short loc_31BD
lea rsi, aWannaCheatYes1 ; "wanna c
mov rax, cs:_ZSt4cout_ptr
mov rdi, rax

NX/DEP

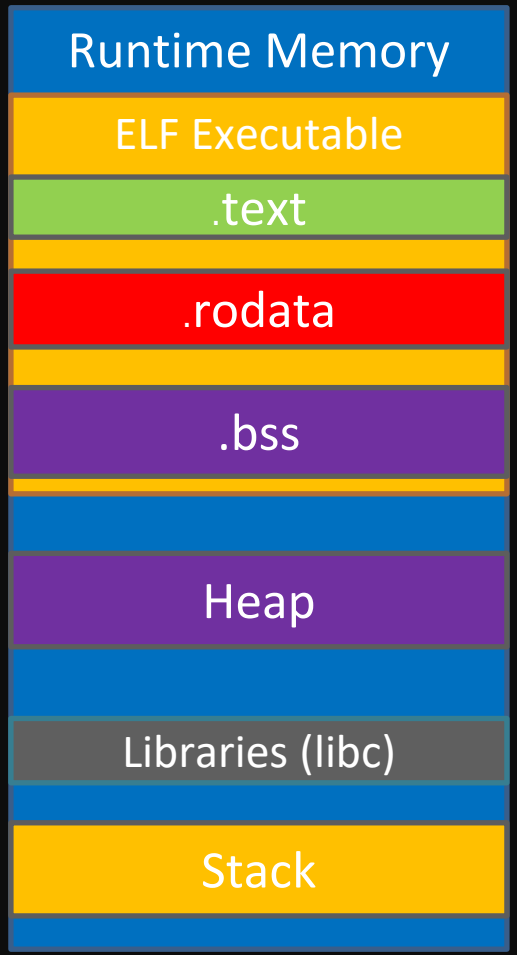
- Giới thiệu NX/DEP (No-Execute/Data Execution Prevention)
  - Còn được biết đến là Executable Space Protection
    - Đừng nhầm lẫn với $esp ☺
- Xuất hiện lần đầu trên Linux và Windows vào năm 2004.



DEP/NX/W^X

- Stack/Heap/data được map với No eXecute (chặn quyền thực thi)
- aka DEP (Data Execution Prevention)
- aka W^X (Write xor eXecute)
  - Phân vùng bộ nhớ chỉ được đánh dấu là writable HOẶC executable (không có cả hai).
  - Việc thực thi phân vùng bộ nhớ không được đánh dấu executable sẽ dẫn đến chương trình gặp segmentation fault.
- Không còn shellcode 😔

Runtime Memory without DEP



R-X

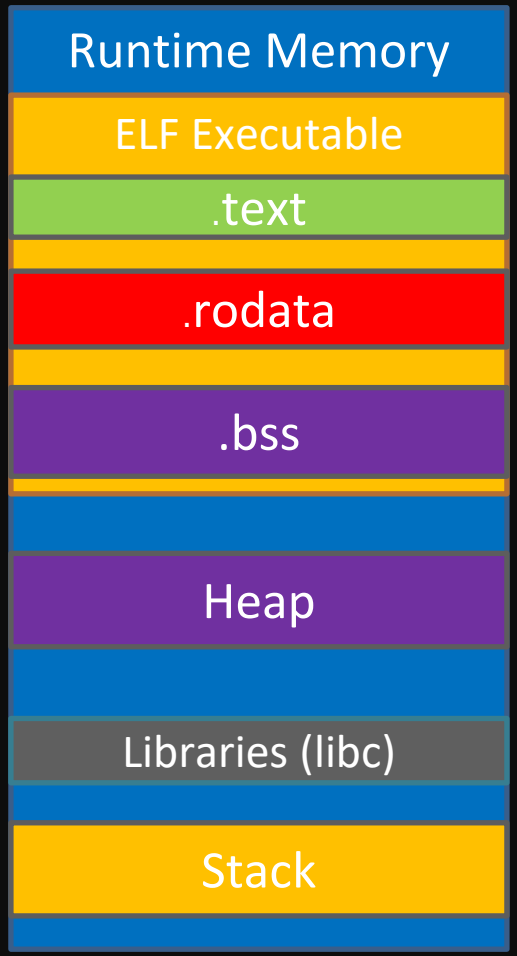
R-X

RWX

RWX

RWX

Runtime Memory without DEP



R-X

R--

RW-

RW-

RW-

Return Oriented Programming (ROP)

- Code reuse
  - Không thể sử dụng code riêng vì không có buffer nào thực thi được
  - Nhưng chúng ta vẫn có thể sử dụng những đoạn mã có sẵn của chương trình:
    - … điều gì sẽ xảy ra nếu chúng ta có thể kết hợp các đoạn code có sẵn để làm cho chương trình thực thi những gì ta muốn ?
    - Những đoạn code đó được gọi là gadgets

Return Oriented Programming (ROP)

- ROP gadgets
  - Những đoạn code thường kết thức với return, syscall, call / jump
  - Nếu chúng ta kiểm soát được $rsp – stack pointer, chúng ta có thể kết hợp những gadgets với nhau tạo thành chuỗi được gọi là ROP gadgets.
  - Mỗi lần “return” , chương trình sẽ thực thi gadget tiếp theo ngay sau đó.

```
pop rax
ret
```

```
syscall
```

```
pop rbx
pop rdx
ret
```

```
xchg rsp,
rax
ret
```

```
add rax, 0x4
ret
```

Return Oriented Programming (ROP)

Setup các thanh ghi gọi execve(“/bin/sh”,0,0) trên x86_64:

rax: 0x3b (sys_execve)

rdi: “/bin/sh”, char* pathname

rsi: 0, char const* argv[] (kept NULL)

rdx: 0, char *const envp[] (kept NULL)

syscall

Return Oriented Programming (ROP)

Trạng thái:

rax: ?

rdi: ?

rsi: ?

rdx: ?

syscall

Return Oriented Programming (ROP)

Trạng thái:

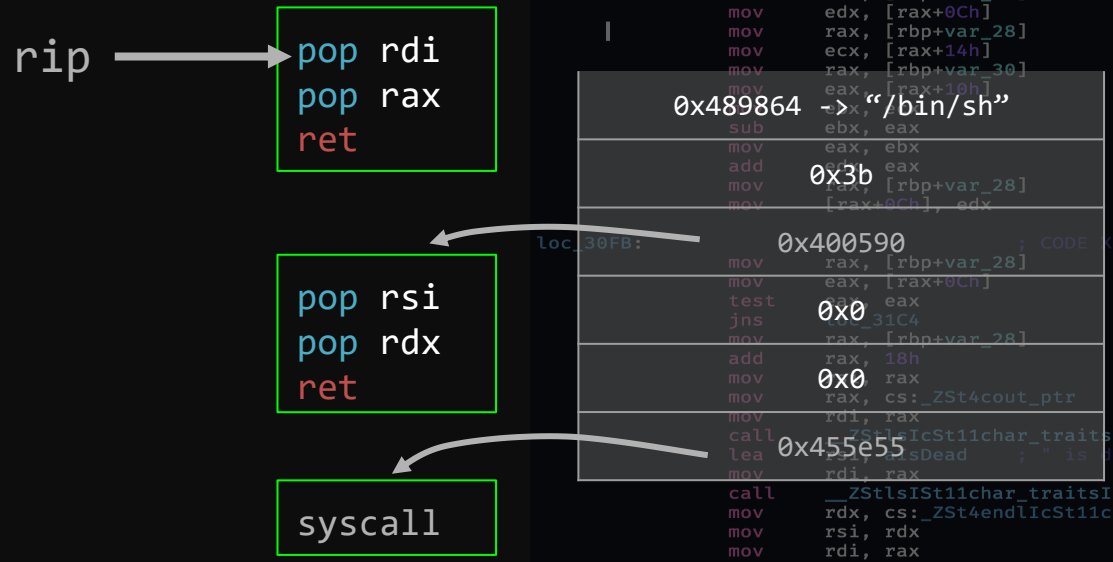
rax: ?

rdi: ?

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Trạng thái:

rax: ?

rdi: 0x489864 “/bin/sh”

rsi: ?

rdx: ?

syscall

Return Oriented Programming (ROP)

Trạng thái:

rax: 0x3b

rdi: 0x489864 “/bin/sh”

rsi: ?

rdx: ?

syscall

Return Oriented Programming (ROP)

Trạng thái:

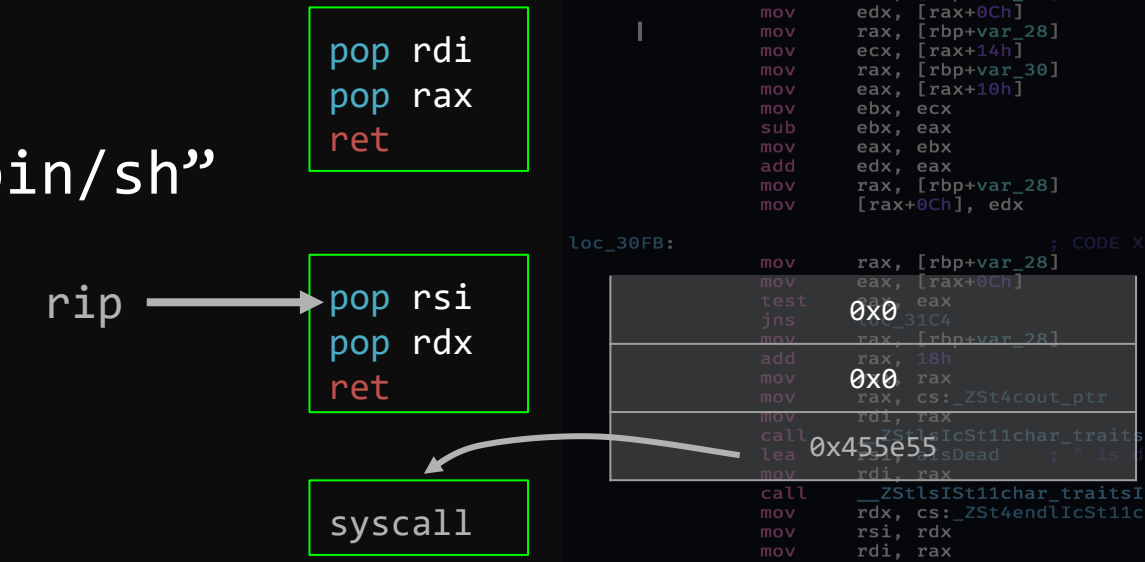
rax: 0x3b

rdi: 0x489864 “/bin/sh”

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Trạng thái:

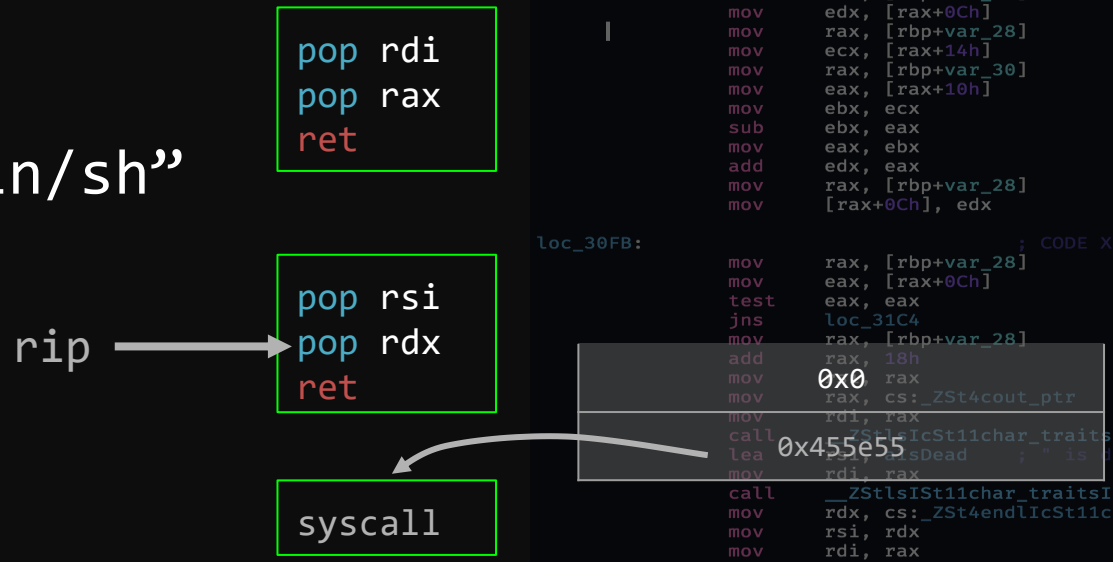
rax: 0x3b

rdi: 0x489864 “/bin/sh”

rsi: 0x0

rdx: ?

syscall



Return Oriented Programming (ROP)

Trạng thái:

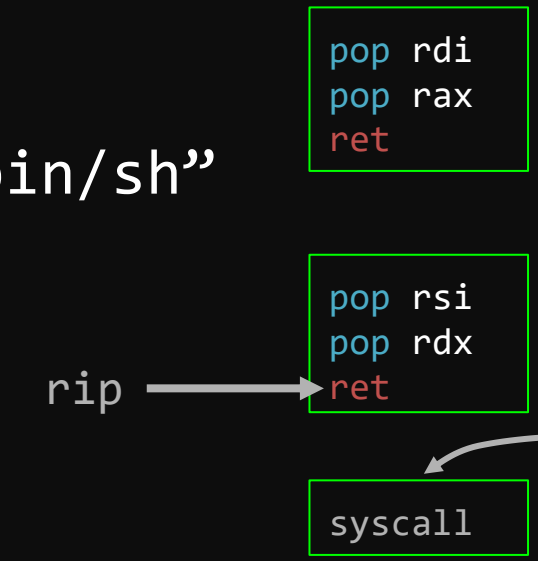
rax: 0x3b

rdi: 0x489864 “/bin/sh”

rsi: 0x0

rdx: 0x0

syscall



Return Oriented Programming (ROP)

Trạng thái:

rax: 0x3b

rdi: 0x489864 “/bin/sh”

rsi: 0x0

rdx: 0x0

syscall

Shell! :D

```
pop rdi
pop rax
ret
```

```
pop rsi
pop rdx
ret
```

rip → `syscall`

ROP

Công cụ tìm gadgets:

- ROPgadget --binary [file] (pip install ropgadget)
- rp++ (https://github.com/0vercl0k/rp)

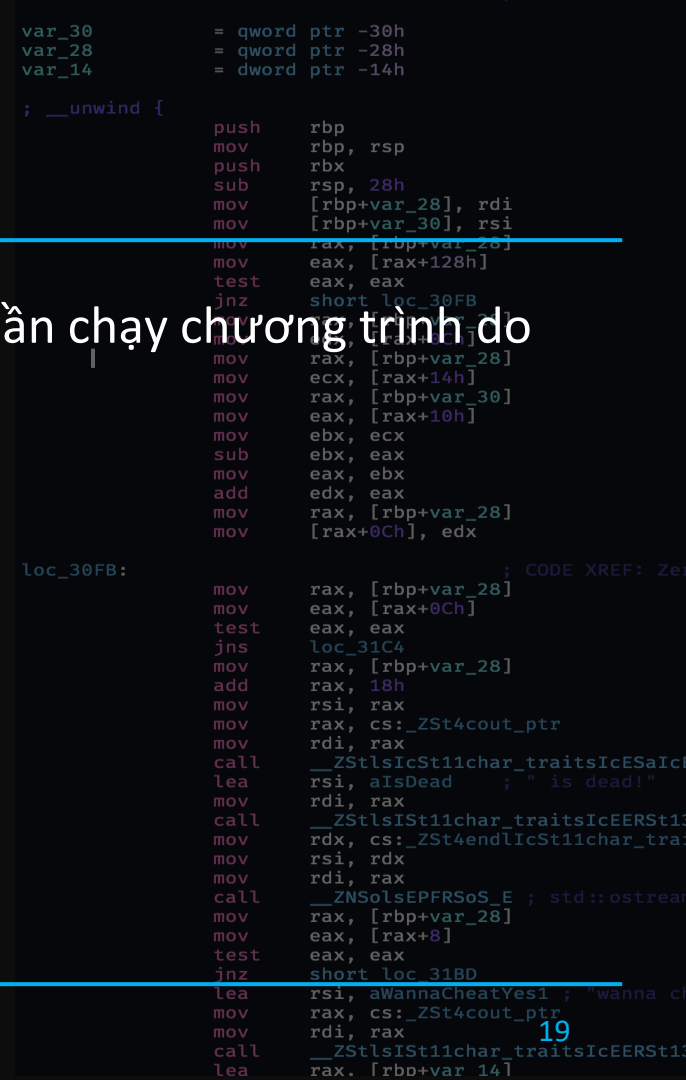
Better ROP

- Làm thế nào nếu như chúng ta không có đủ gadgets cần thiết (Vd: no `syscall`):
- Với những dynamically linked binaries, thư viện C chuẩn (libc) sẽ nằm ở đâu đó trên memory …
- ret2libc: sử dụng các hàm trong libc, như system()

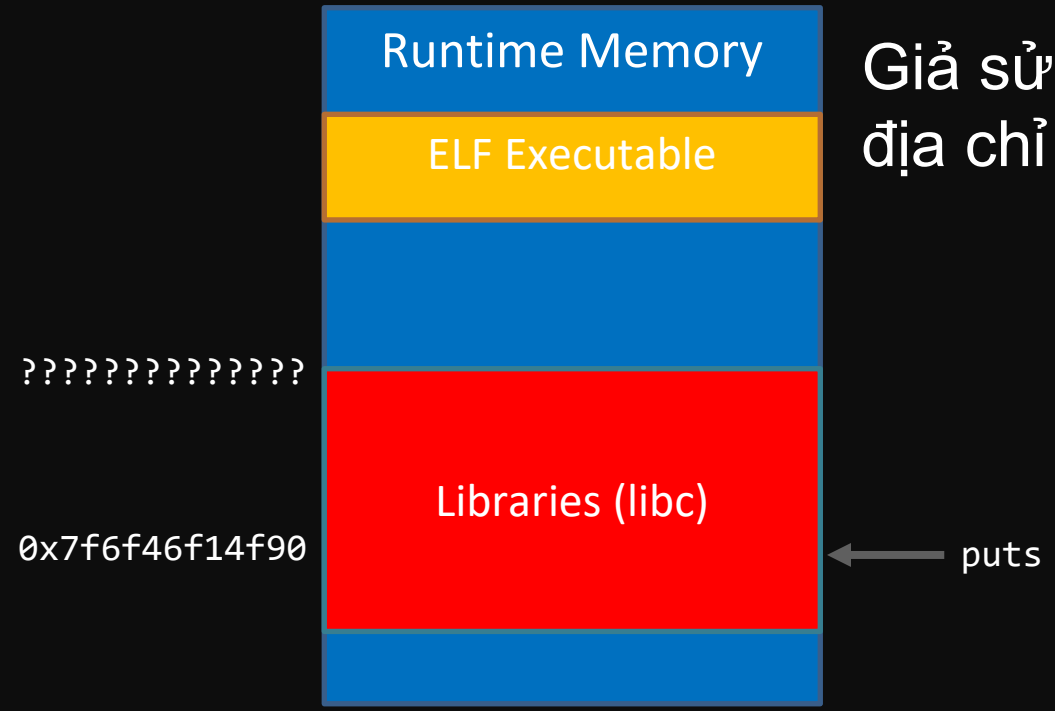
Bypass ASLR

- Thông thường địa chỉ libc sẽ thay đổi trong mỗi lần chạy chương trình do cơ chế ASLR.

- ASLR - Address Space Layout Randomization

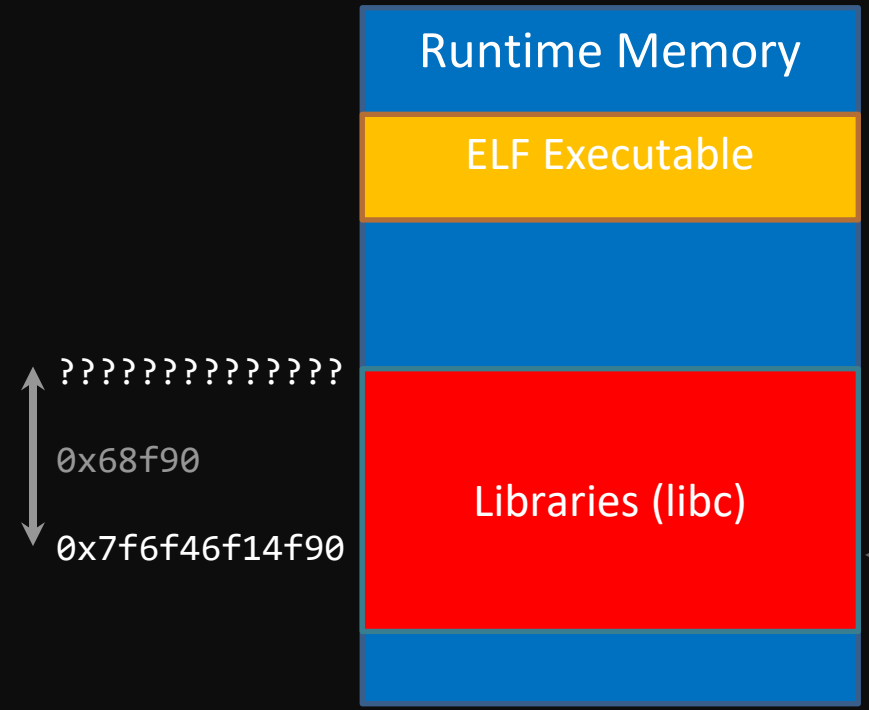


Bypass ASLR



Giả sử attacker đã leak được địa chỉ của một hàm trong libc.

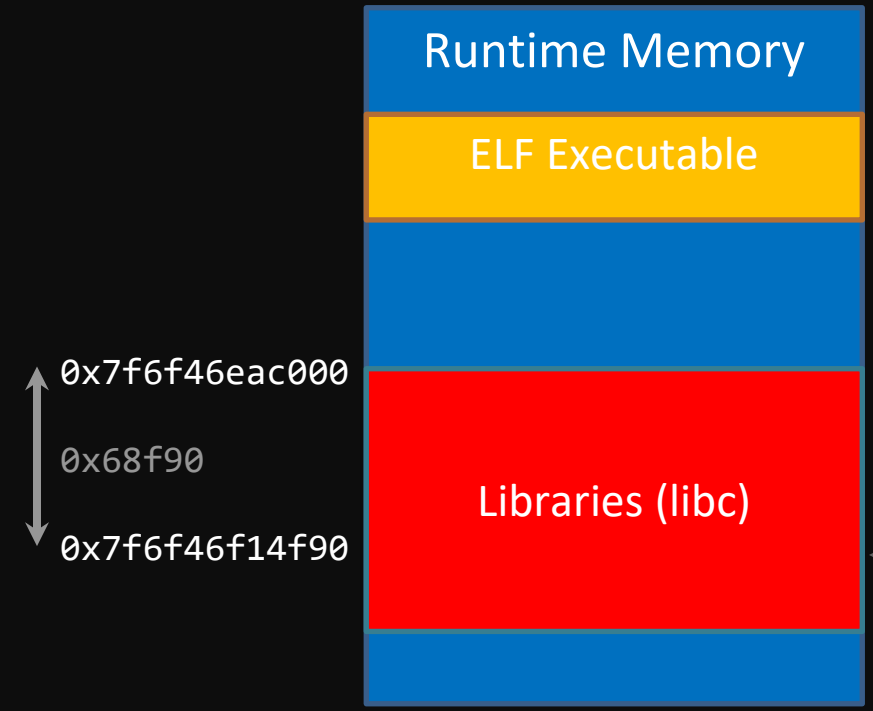
Bypass ASLR



Giả sử attacker đã leak được địa chỉ của một hàm trong libc.

Với tất cả các hàm, địa chỉ libc đều có offset so với địa chỉ base là không đổi.

Bypass ASLR

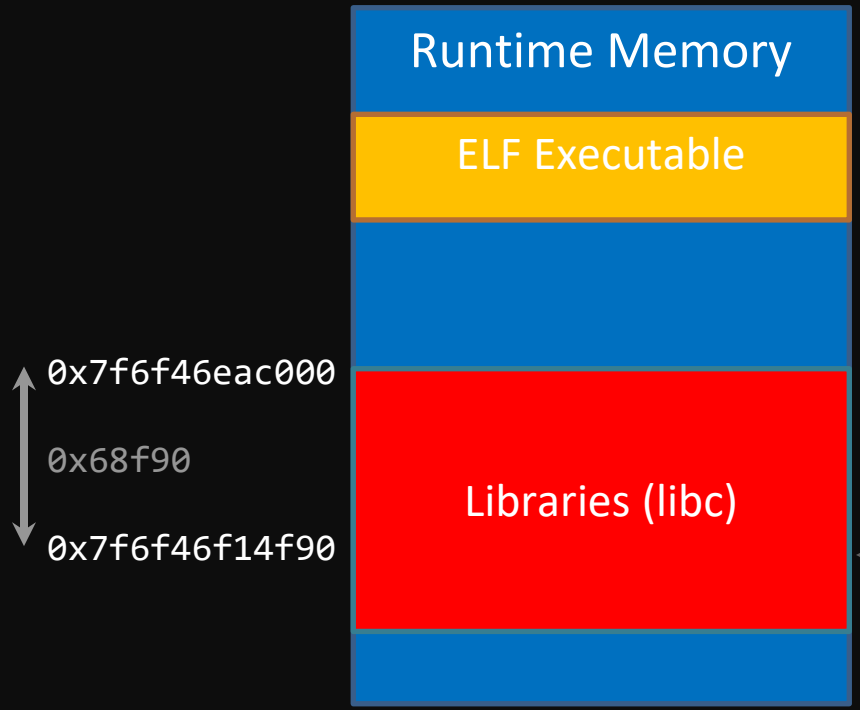


Giả sử attacker đã leak được địa chỉ của một hàm trong libc.

Tất cả các hàm, địa chỉ libc đều có offset so với địa chỉ base là không đổi.

Đơn giản ta chỉ cần cộng/trừ là có được địa chỉ base

Bypass ASLR



Khi có được địa chỉ base, việc tìm địa chỉ các hàm libc (vd: system) rất đơn giản, chỉ cần cộng offset.

Bypass ASLR

Địa chỉ puts

libc base

offset của system

```
pwndbg> p puts
$3 = {int (const char *)} 0x7ffff7e06ed0 <__GI__IO_puts>
pwndbg> vmmap libc
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x7ffff7d83000     0x7ffff7d86000 rw-p     3000      0 [anon_7ffff7d83]
    0x7ffff7d86000     0x7ffff7dae000 r--p    28000      0 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7dae000     0x7ffff7f43000 r-xp   195000  28000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7f43000     0x7ffff7f9b000 r--p    58000 1bd000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7f9b000     0x7ffff7f9f000 r--p     4000 214000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7f9f000     0x7ffff7fa1000 rw-p     2000 218000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7dae000     0x7ffff7f43000 r-xp   195000  28000 /usr/lib/x86_64-linux-gnu/libc.so.6
pwndbg> p system
$4 = {int (const char *)} 0x7ffff7dd6d60 <__libc_system>
pwndbg> p/x 0x7ffff7dd6d60-0x7ffff7d86000
$5 = 0x50d60
pwndbg> A
```

Even better ROP

Làm thế nào để leak địa chỉ?

Dynamic Linking

Làm thế nào để binary sử dụng các thư viện (libc, libcrypto...)

GOT - Global Offset Table

- Đơn giản chỉ là mảng các function pointer.

```
0x601018 --> 0x7f21c0e88190 (<printf>:  sub   rsp,0xd8)
0x601020 --> 0x7f21c0ea1660 (<gets>:    push  r12)
```

Dynamic Linking

GOT - Global Offset Table

- Mảng function table

PLT - Procedure Linkage Table

- Gọi *got[function_offset]()

```
0x400480 <gets@plt+0>: jmp QWORD PTR [rip+0x200b9a] # 0x601020

---GOT----
0x601018 --> 0x7f21c0e88190 (<printf>:       sub     rsp,0xd8)
0x601020 --> 0x7f21c0ea1660 (<gets>: push    r12)
```

Even better ROP

Liệu chúng ta có thể tận dụng GOT và PLT để leak libc ?

Hint: puts(puts@got); main();

ROPchain is so damm big

- Nếu như ROPchain của chúng ta quá lớn nhưng stack lại không đủ để chứa? :

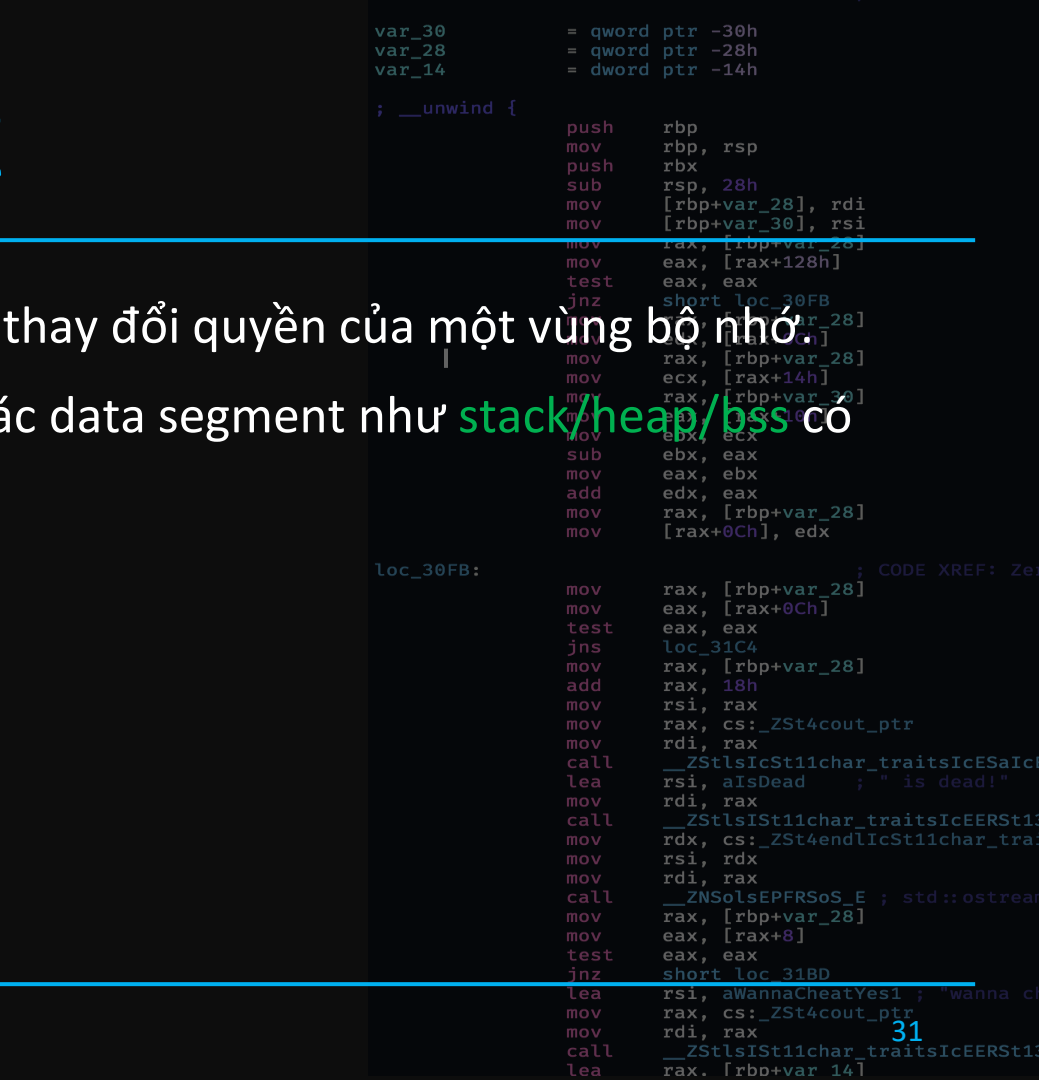
1. Ghi đè $rsp đi nơi khác (Fake stack ?)
2. Disable DEP và thực thi shellcode ?

Stack pivot

- Chúng ta có thể ghi đè $rsp (stack pointer) đi nơi khác, khi đó chương trình sẽ hiểu địa chỉ $rsp trỏ tới là stack.
- Gadget hữu ích:
  - pop rsp : rất hiếm khi thấy. Nhưng nếu thấy, thì bạn là một người rất may mắn ☺
  - xchg <reg>, rsp : hoán đổi giá trị của <reg> và rsp với nhau
  - leave : hoạt động như mov rsp, rbp ; pop rbp

Ret 2 mprotect

- Linux có hàm mprotect giúp ta thay đổi quyền của một vùng bộ nhớ.
- Ta có thể thay đổi quyền của các data segment như stack/heap/bss có quyền write lẫn execute.



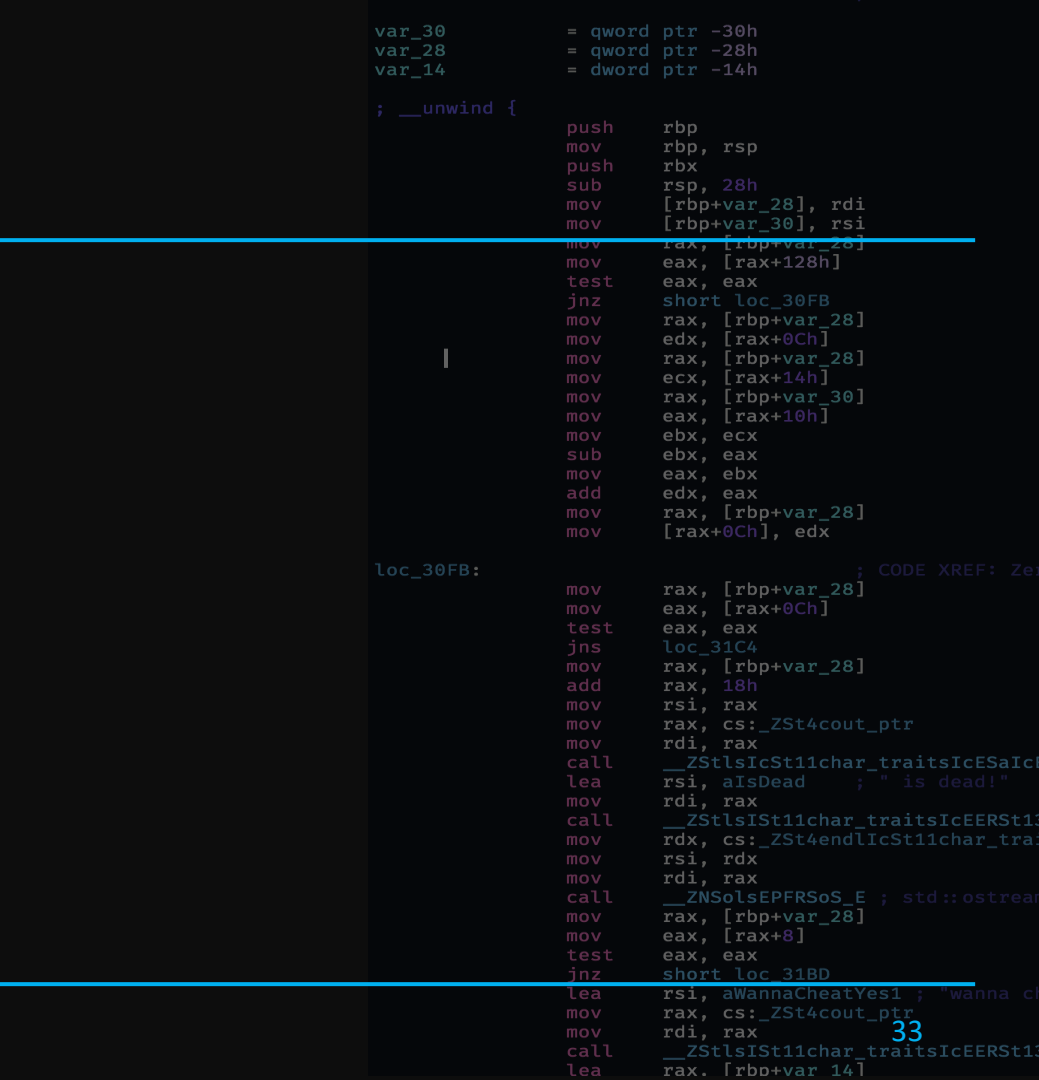
Ret 2 mprotect

- Đơn giản chỉ cần gọi mprotect(address,size,7) (https://man7.org/linux/man-pages/man2/mprotect.2.html)
- Kĩ thuật ret2shellcode giờ có thể áp dụng.

Practice

Wargames:

- microcorruption.com
- pwnable.kr
- pwnable.tw
- ropemporium.com



Reference

- https://github.com/RPISEC/MBE
- https://ir0nstone.gitbook.io/notes/types/stack/stack-pivoting