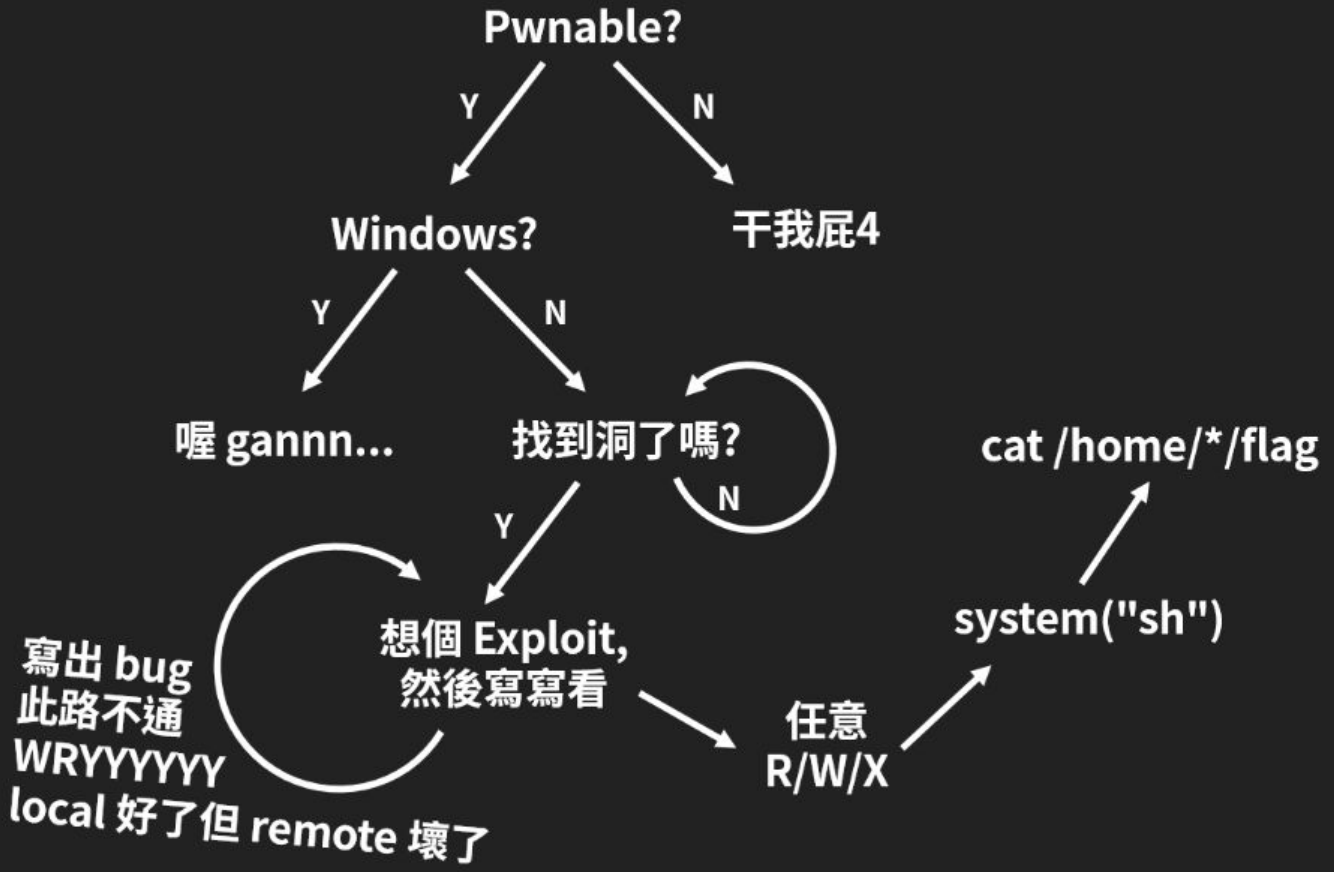
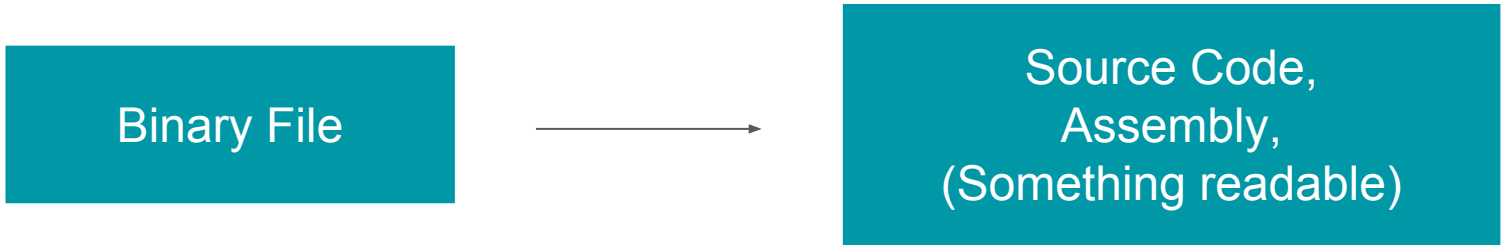
Basic Tools & Concept

kevin47

About me


Pwnable?
Y
N
Windows?
干我屁4
Y
N
喔 gannn...
找到洞了嗎?
N
Y
cat /home/*/flag
system("sh")
想個 Exploit,
然後寫寫看
任意
R/W/X
寫出 bug
此路不通
WRYYYYYY
local 好了但 remote 壞了

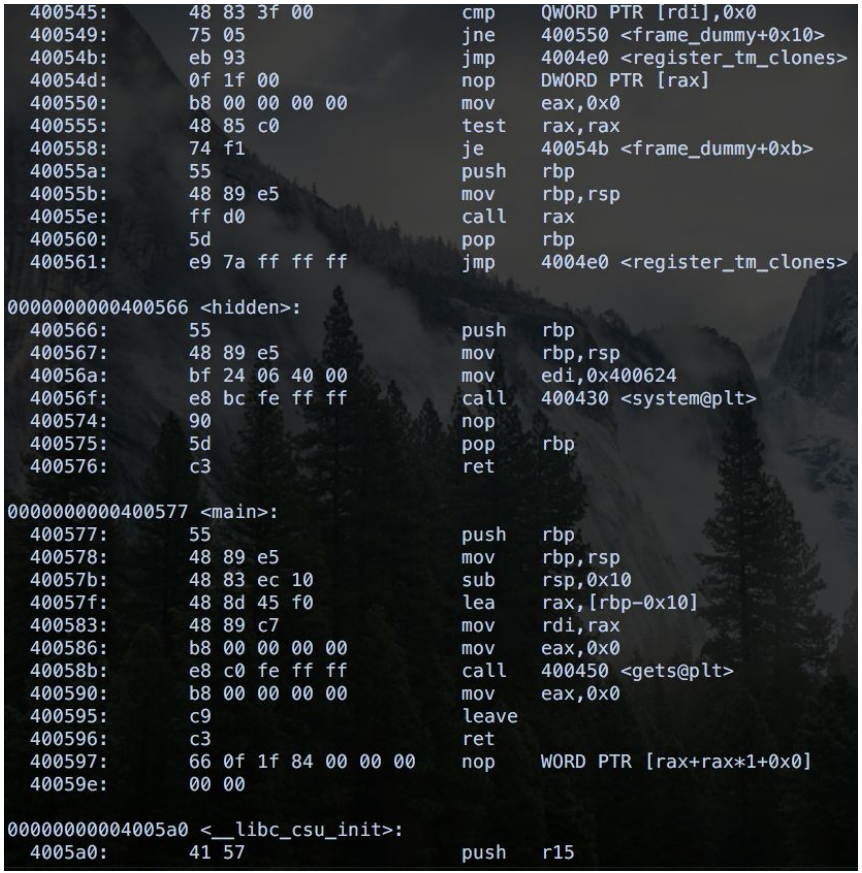
Reverse Engineering



- 正常使□者只會有執□檔，沒有該程式的原始碼
- 透過逆向□程分析程式來找出程式的漏洞或是修改程式

Reverse Engineering

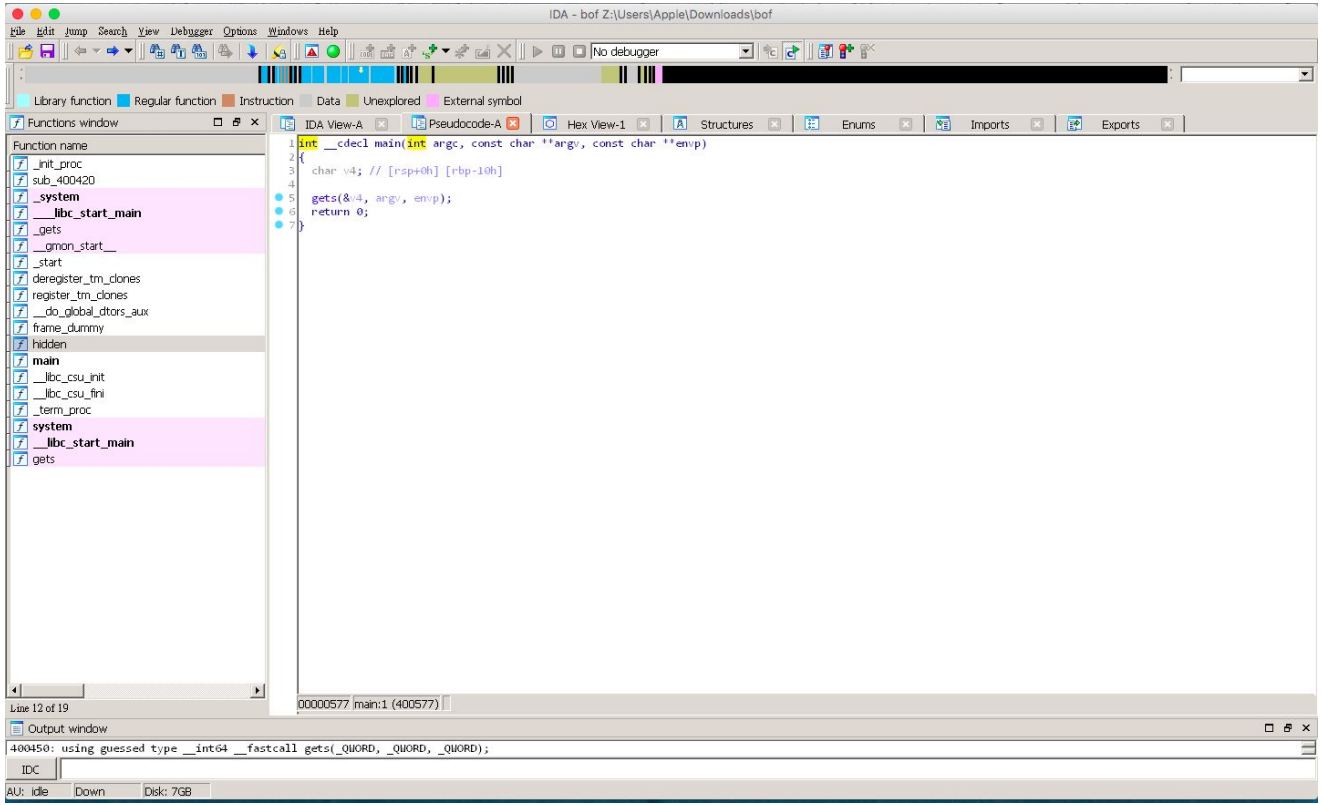
- Static Analysis
  - Analysis program without running
    - objdump
    - strings
- Dynamic Analysis
  - Analysis program with running
    - ltrace, strace
    - gdb



Exploitation (Pwn)

- 利□漏洞來達成攻擊者□的
- □般來說主要□的在於取得程式控制權
  - 本地提權
  - Remote Code Execution

IDA pro


IDA - bof Z:\Users\Apple\Downloads\bof
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
Functions window
Function name
_init_proc
sub_400420
_system
___libc_start_main
_gets
__gmon_start__
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
hidden
main
__libc_csu_init
__libc_csu_fini
_term_proc
system
__libc_start_main
gets
Line 12 of 19
IDA View-A
Pseudocode-A
Hex View-1
Structures
Enums
Imports
Exports
int __cdecl main(int argc, const char **argv, const char **envp)
{
char v4; // [rsp+0h] [rbp-10h]
gets(&v4, argv, envp);
return 0;
}
00000577 main:1 (400577)
Output window
400450: using guessed type __int64 __fastcall gets(_QWORD, _QWORD, _QWORD);
IDC
AU: idle
Down
Disk: 7GB

GDB

- Basic Commands:
  - run - 執□程式
  - disas *<function>* - 反組譯某個函數
  - break **<address>* - 在某個位置下斷點
  - delete *<break point id>* - 刪除第幾個斷點
  - info breakpoint - 查看所有斷點
  - info registers - 查看□前暫存器狀態
  - Info proc map - 查看目前的 memory map

GDB

- Basic Commands:
  - x/wx *<address>* - 查看某個位置的內容 (4 byte)
    - w 可替換成 b/h/g，分別是取 1, 2, 8 bytes
    - / 後可以放入要列幾個位置出來，e.g. x/40gx
    - x 可以換成 u/d/s/i，改變數值顯□□式
    - u - unsigned int，d - □進位，s - 字串，i - 指令

GDB

- Basic Commands:
  - si - 下□□指令，跟進 function call
  - ni - 下□□指令，不跟進 function call
  - backtrace - 顯□上層的 Stack Frame 資訊
  - continue - 繼續執□
  - record - 開始記錄，有記錄時可以反相執行
    - rsi - 上一行指令，同 si
    - rni - 上一行指令，同 ni
    - rc - 反相繼續執行，同 continue

GDB

- Basic Commands:
  - set *<reg>*=*<value>* - 把某個 register 改成數值　　ex: set $rbx=0x1234
  - set **<address>*=*<value>* - 把某個地址填入數 值 (4 byte)　　一次填4byte
  - * 可換成 {char} {short} {long} 來改變填入的長度
  - e.g. set {long}0x400000=1 - 把 0x400000 填入 0x00000001

GDB

- Basic Commands:
  - 如果執□檔包含 debug symbol 的時候
    - list - 可以列出程式碼
    - b - 可以直接對□數下斷點
    - info local - 可以列出區域變數
    - print *<var name>* - 可列出變數

GDB

- Basic Commands:
  - attach *<pid>* - 附加到某個正在運□的 process
    - 可以配合 ncat 來 debug exploit
    - $ ncat -vc *<elfpath>* -kl 127.0.0.1 *<port>* - □ ncat 把程式掛在某個 port 中

GDB

- TUI Commands:
  - layout src|asm|reg - 進入 TUI
  - Ctrl+x+a - 離開/進入TUI
  - focus cmd|src|asm　　swapping focus on cmd/src/asm window
    - focus 在 asm|src 的時候上下鍵是滑動 asm|src
    - focus 在 cmd 的時候上下鍵是上一個 /下一個 cmd

GDB-PEDA

- Python Exploit Development Assistance for GDB
- https://github.com/longld/peda
- https://github.com/scwuaptx/Pwngdb

Pwntools

- Functions:
  - remote(*<host>*, *<port>*) - 開連線到 *<host>* 的 *<port>*
    - r = remote('localhost', 7122)
  - interactive() - 切換到互動模式
    - r.interactive()
  - context.arch = *<arch>*
    - *<arch>* is one of: 'aarch64', 'alpha', 'amd64', 'arm', 'avr', 'cris', 'i386', 'ia64', 'm68k', 'mips', 'mips64', 'msp430', 'powerpc', 'powerpc64', 's390', 'sparc', 'sparc64', 'thumb', 'vax'

x64 x86

Pwntools

- Functions:
  - recv(<*N*>) - 從連線接收 <*N*> 個 bytes
    - r.recv(8)
  - recvuntil(<*str*>, [drop=True, False]) - 從連線接收所有東西，直到 <*str*> 出現為止
    - drop=True - 把 <*str*> 丟掉，drop=False - 連 <*str*> 一起接收，預設是這個
    - x = r.recvuntil('whatever', drop=True)
      - 假設連線送 "You can do whatever you want"
      - x == "You can do "

Pwntools

- Functions:
  - sendline(<*str*>) - 把 <*str*> + “\n” 送到連線
    - r.sendline(‘this is my input’)
  - sendlineafter(<*str1*>, <*str2*>)
    - 等同於先 r.recvuntil(<*str1*>) 再 r.sendline(<*str2*>)
    - r.sendlineafter(‘Your name: ’, ‘Ovuvuevuevue Enyetuenwuevue Ugbemugbem Osas’)

Pwntools

- Functions:
  - p32(*<32 bit integer>*) - Packs an integer to little-endian
    - p32(1234)
  - u32(*<4 bytes string>*) - Unpacks a little-endian string to integer
    - u32(“\xa0\x00\x01\x00”)
  - p64(*<64 bit integer>*) - 跟 p32 一樣，但是是 64 bit
  - u64(*<8 bytes string>*) - 跟 u32 一樣，但是是 8 bytes
  - flat(*<iterable>, ...*) - 把參數裡的東西全部都 p32/p64（看 context.arch 決定）
    - `flat([[0x1234, 0x4321]*3, 6,7], 7122)`

Pwntools

- Functions:
  - ELF(*<binary path>*) - 載入一個 binary
    - e = ELF(‘./bof’)
    - e = ELF(‘./libc.so.6’)
      - e.symbols[‘system’]
      - e.search(‘/bin/sh’)

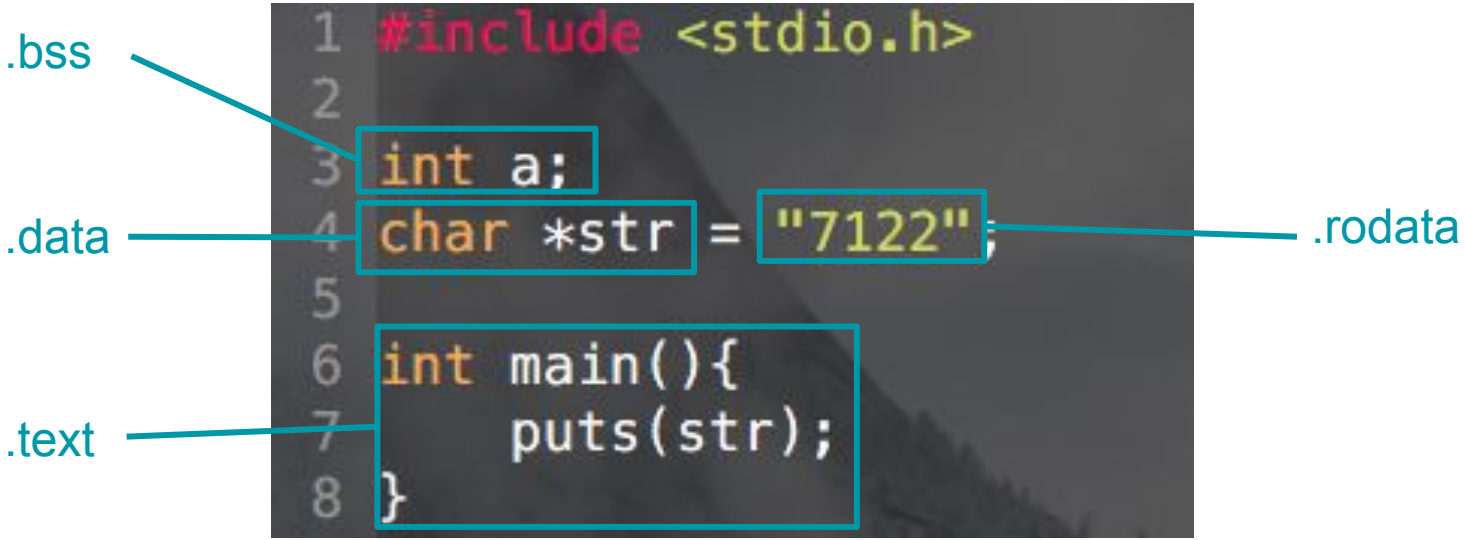
LAB 1-1

Section

在□般情況下程式碼會分成 text、data 以及 bss 等 section，並不會將 code 跟 data 混在□起

- .text
  - 存放 code 的 section
- .data
  - 存放有初始值的全域變數
- .bss
  - 存放沒有初始值的全域變數
- .rodata
  - 存放唯讀資料的 section

Section


.bss
.data
.text
.rodata
1 #include <stdio.h>
2
3 int a;
4 char *str = "7122";
5
6 int main(){
7 puts(str);
8 }

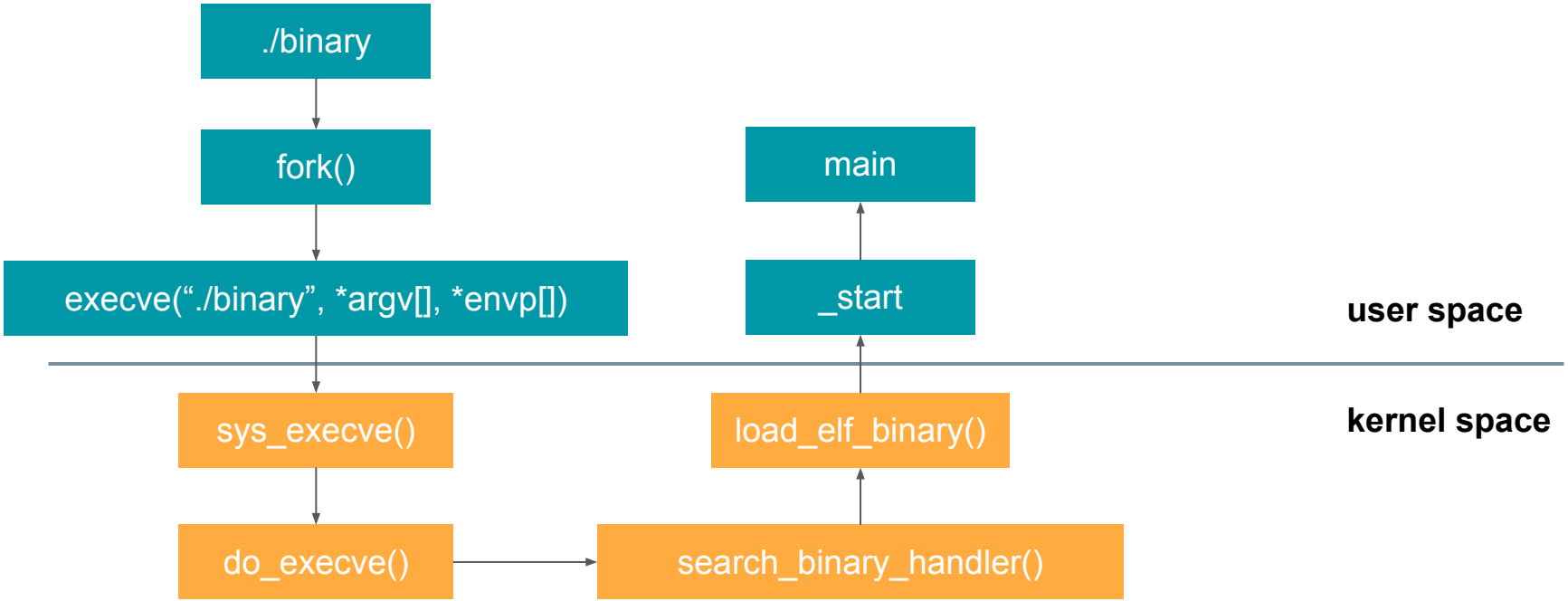
Segment

在程式執□時期才會有的概念，基本上會根據讀寫 執□權限及特性 來分為數個 segment

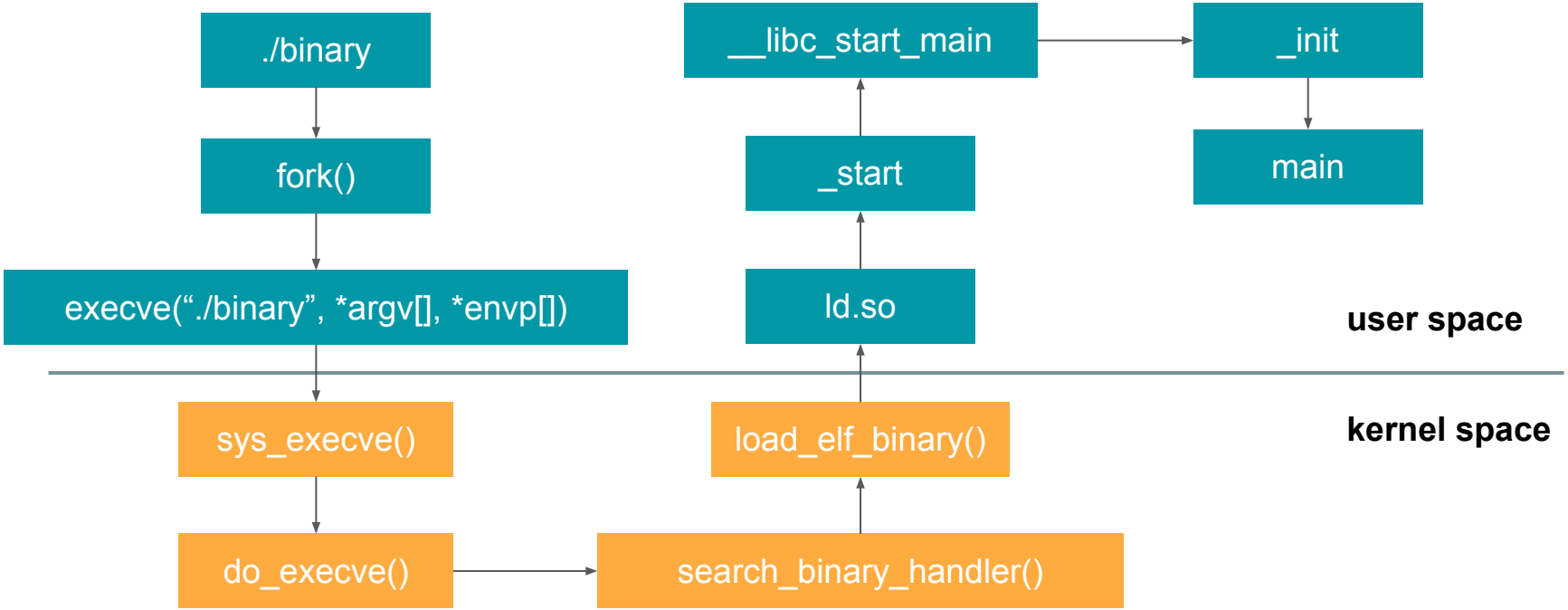
□般來說可分為 rodata、data、code、stack、heap 等 segment

- data: rw-
- code: r-x
- stack: rw-
- heap: rw-

Execution Flow (Static Linking)



Execution Flow (Dynamic Linking)



x64 assembly

- Registers
  - RAX RBX RCX RDX RSI RDI- 64 bit
  - EAX EBX ECX EDX ESI EDI - 32 bit
  - AX BX CX DX SI DI - 16 bit

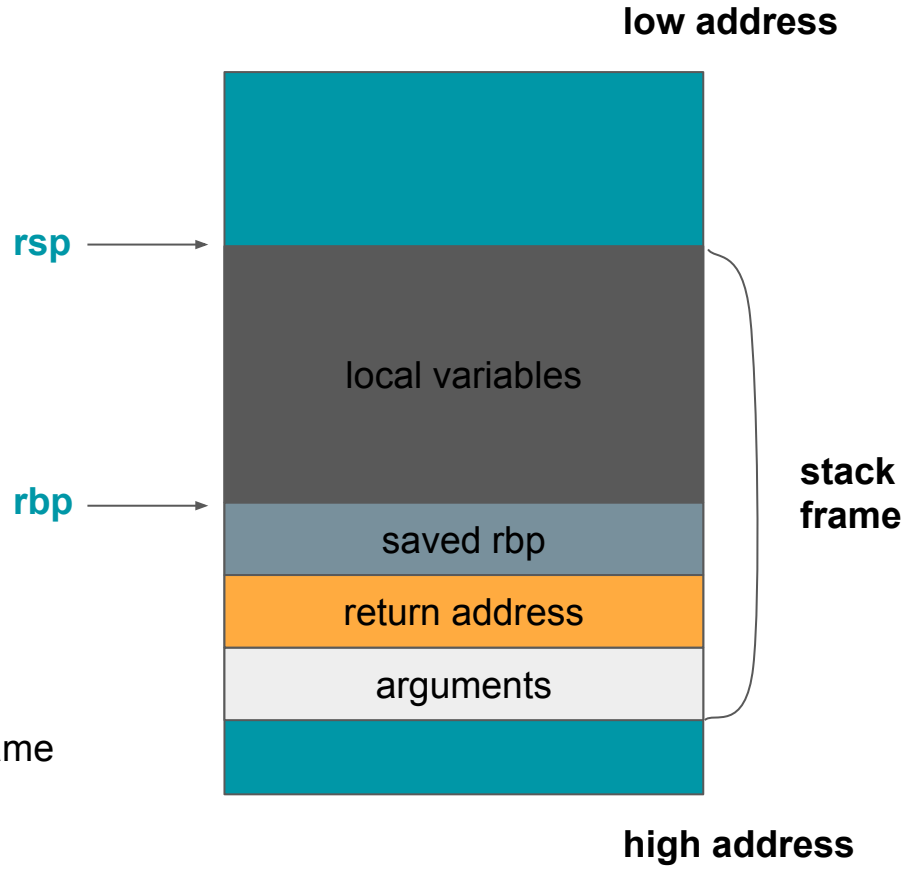
x64 assembly

- Registers
  - r8 r9 r10 r11 r12 r13 r14 r15 - 64 bit
  - r8d r9d r10d … - 32 bit
  - r8w r9w r10w … -16 bit
  - r8b r9b r10b … - 8 bit

x64 assembly

- Registers
  - Stack Pointer Register
    - RSP (64 bit) - 指向 stack 頂端
  - Base Pointer Register
    - RBP (64 bit) - 指向 stack 底端
  - Program Counter Register
    - RIP (64 bit) - 指向目前執行位置

RSP 到 function 參數範圍稱為該 function 的 Stack Frame



x64 assembly

- Syntax
  - AT&T (很醜，不要用)

```
0x400827 <main+17>:  mov    %rsi,-0x250(%rbp)
0x40082e <main+24>:  mov    %fs:0x28,%rax
0x400837 <main+33>:  mov    %rax,-0x8(%rbp)
```

  - **Intel (用他)**
    - 在 ~/.gdbinit 中加上 set disassembly-flavor intel

```
0x400827 <main+17>:  mov    QWORD PTR [rbp-0x250],rsi
0x40082e <main+24>:  mov    rax,QWORD PTR fs:0x28
0x400837 <main+33>:  mov    QWORD PTR [rbp-0x8],rax
```

x64 assembly

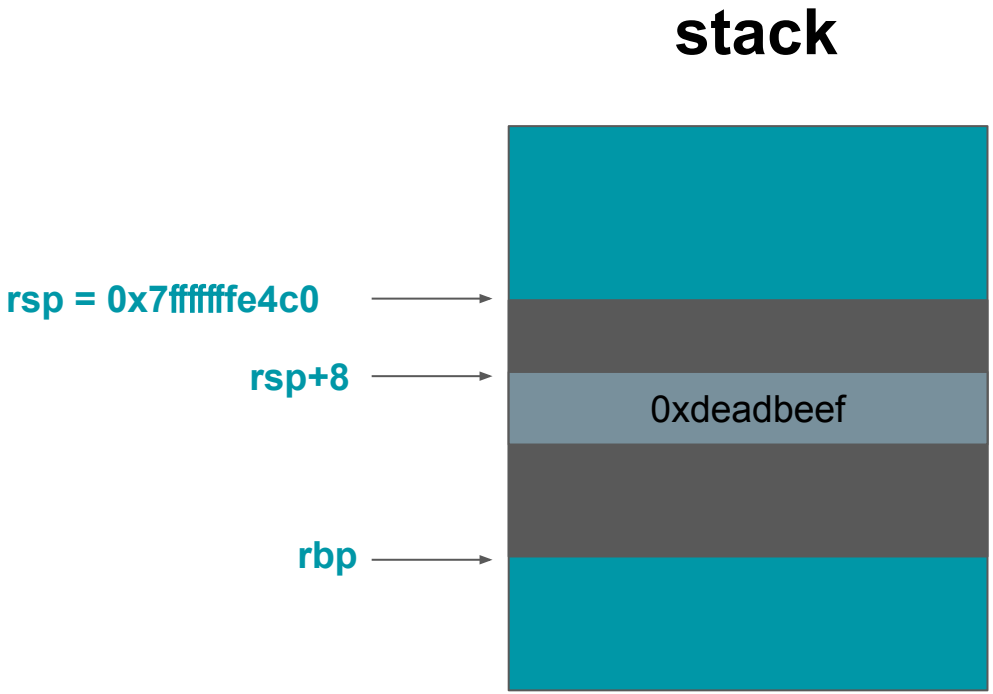
- Basic instructions
  - mov
  - lea
  - add/sub
  - and/or/xor
  - push/pop
  - jmp/call/ret

x64 assembly

- mov (move)
  - mov imm/reg/mem value to reg/mem
  - mov A, B (A = B)
  - A 和 B 的大小要一樣
    - mov rdi, rsi
    - ~~mov rdi, si~~ (X)
    - mov rdi, 0xdeadbeef

x64 assembly

- lea (load effective address)
- lea v.s. mov
  - lea rax, [rsp+8]
    - rax = 0x7fffffffe4c8
  - mov rax, [rsp+8]
    - rax = 0xdeadbeef



x64 assembly

- add/sub/or/xor/and
  - add/sub/or/xor/and reg,imm/reg
  - add/sub/or/xor/and A,B
  - A 與 B 的 size □樣要相等
    - add rbp,0x48
    - sub rax,rbx

x64 assembly

- push/pop
  - push/pop reg/mem
    - push rax = sub rsp, 8; mov [rsp], rax;
    - pop rdi = mov rdi, [rsp]; add rsp, 8;

x64 assembly

- jmp/call/ret
  - jmp 跳到程式某處
    - jmp A = mov rip, A
  - call 儲存本將執行的下一行指令，再跳到程式某處
    - call A = push next_rip; jmp A
  - ret 反回儲存位置
    - ret = pop rip

x64 assembly

- nop (no operation)
  - 不做任何事，常常拿來 patch
    - 例如程式呼叫了我們不想要的某個函式 A，就可以把 call A 改成 nop
  - 一個 byte，opcode = 0x90

x64 assembly

- leave 還原上一個 stack frame
  - mov rsp, rbp
  - pop rbp

x64 assembly

- Calling Convention
  - 參數用 register 傳，register 用完則放在 stack 裡
  - 使用的 register 順序為 rdi, rsi, rdx, rcx, r8, r9　　*傳參數的順序
  - 從第 7 個參數開始放 stack 裡

x64 assembly

- Calling Convention
  - Function prologue
    - compiler 在 function **開頭**加的指令，主要在保存 rbp 和分配區域變數所需空間

```
push rbp
mov rbp, rsp
sub rsp, 0x50
```

x64 assembly

- Calling Convention
  - Function epilogue
    - compiler 在 function **結尾**加的指令，主要在利用保存的 rbp 恢復 call function 前的 stack 狀態

```
leave
ret
```

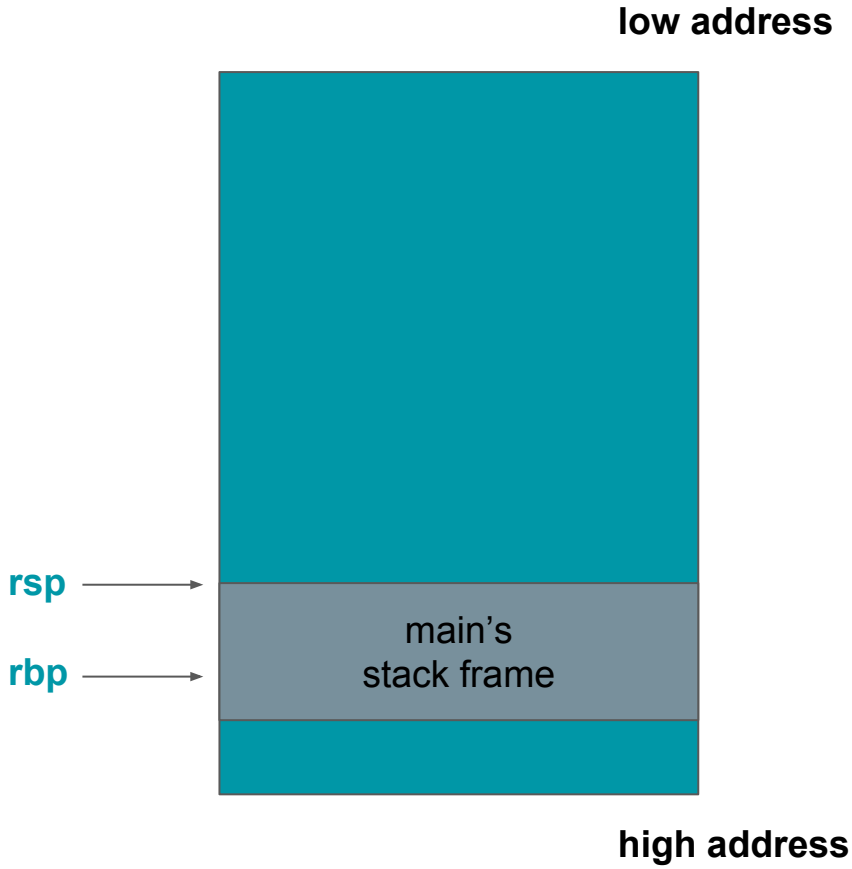
x64 assembly

Putting all together

```
void a(int b, int c){
     char d[0x50];
     ...
     return;
}

int main(){
     a(1234, 7122);
}
```

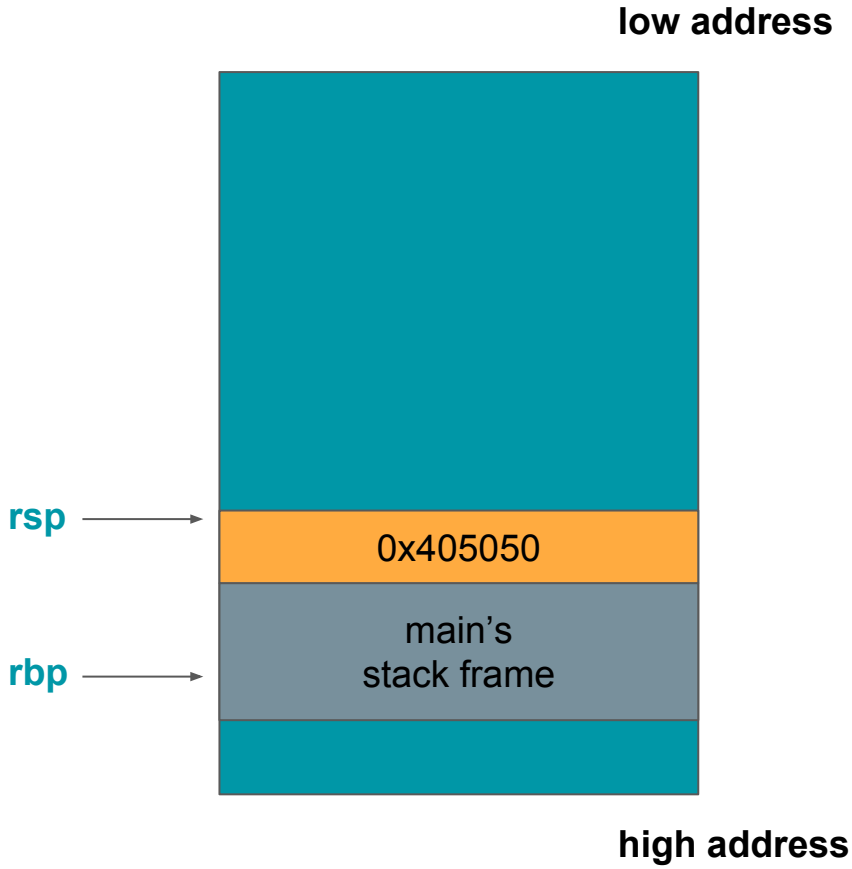
```
a:
     push rbp
     mov rbp, rsp
     sub rsp, 0x50
     ...
     leave
     ret

main:
     push rbp
     mov rbp, rsp
     mov rdi, 1234
     mov rsi, 7122
     call a
     leave
     ret
```

```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave
        ret
```

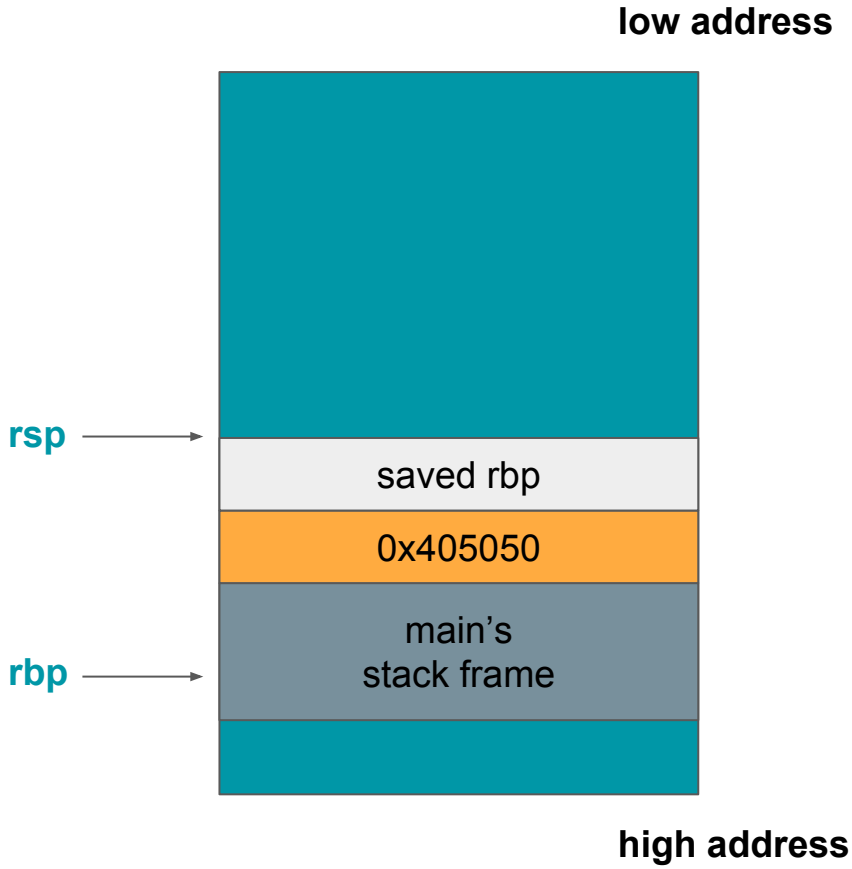


```
a:
      push rbp
      mov rbp, rsp
      sub rsp, 0x50
      ...
      leave
      ret

main:
      push rbp
      mov rbp, rsp
      mov rdi, 1234
      mov rsi, 7122
      call a
      leave (assume addr=0x405050)
      ret
```

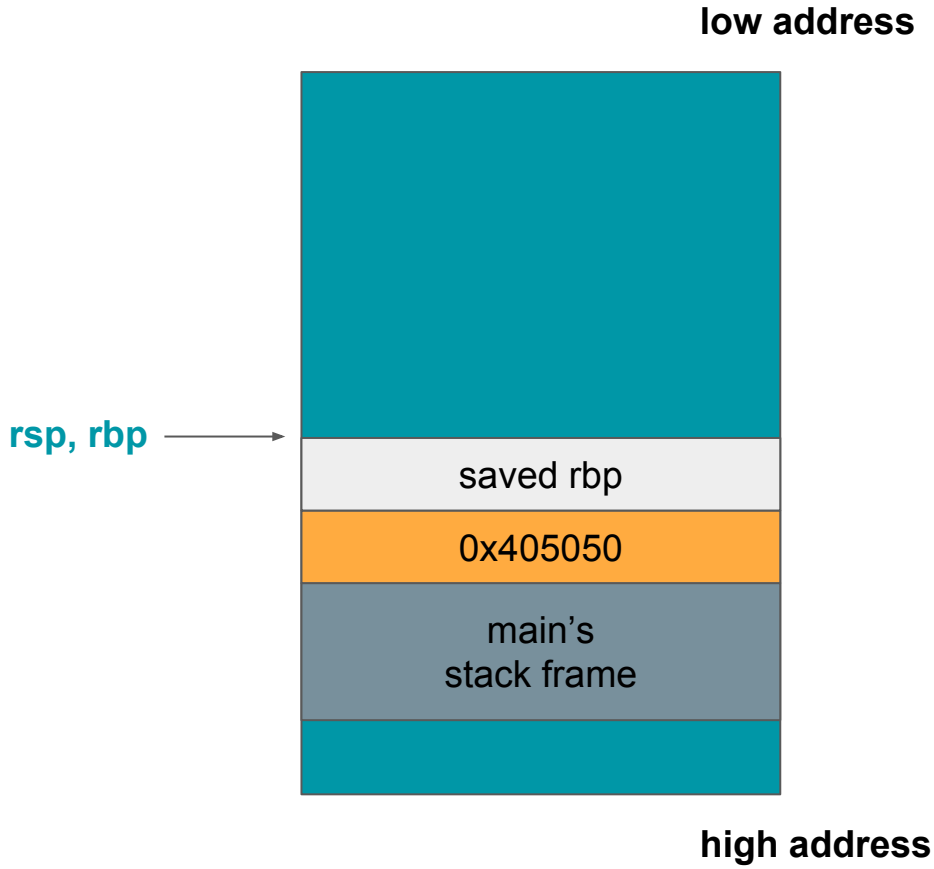
call A =
push next_rip
jmp A



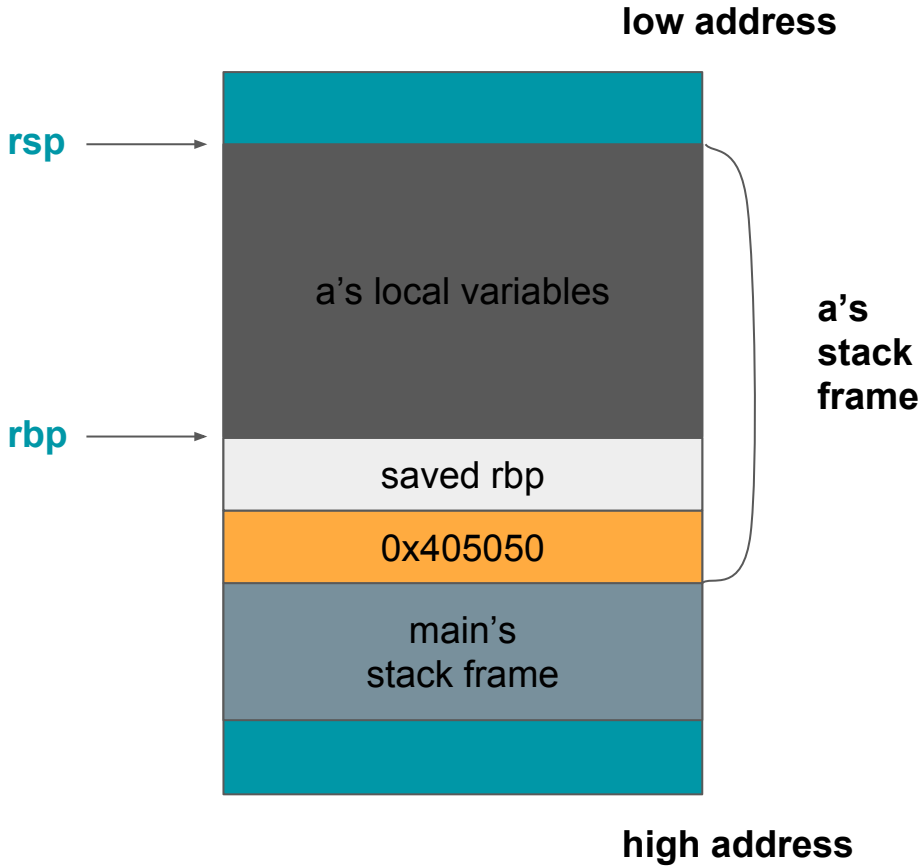
```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```



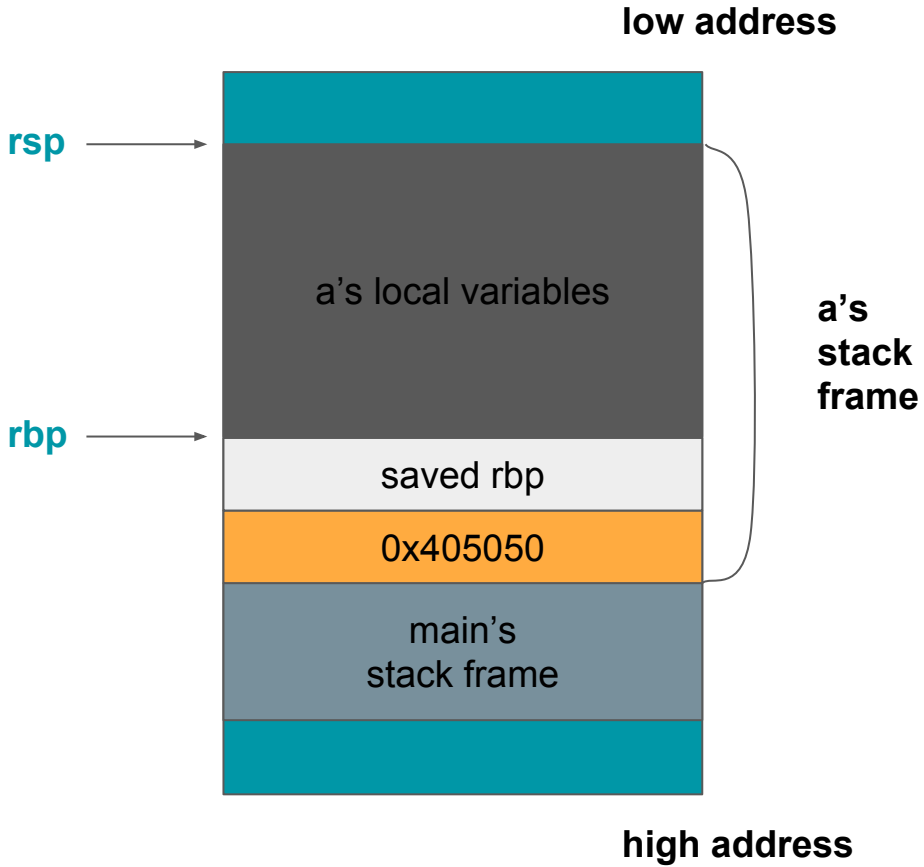
```
a:
      push rbp
      mov rbp, rsp
      sub rsp, 0x50
      ...
      leave
      ret

main:
      push rbp
      mov rbp, rsp
      mov rdi, 1234
      mov rsi, 7122
      call a
      leave (assume addr=0x405050)
      ret
```



```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```



```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```

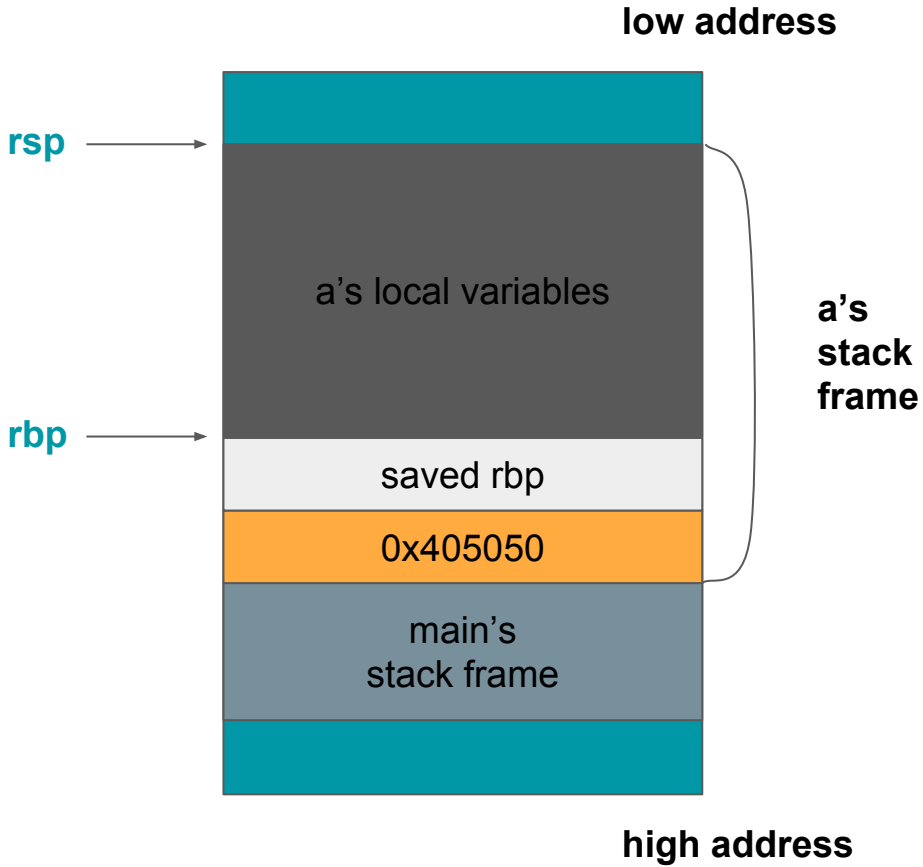


```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```

leave =

mov rsp, rbp

pop rbp

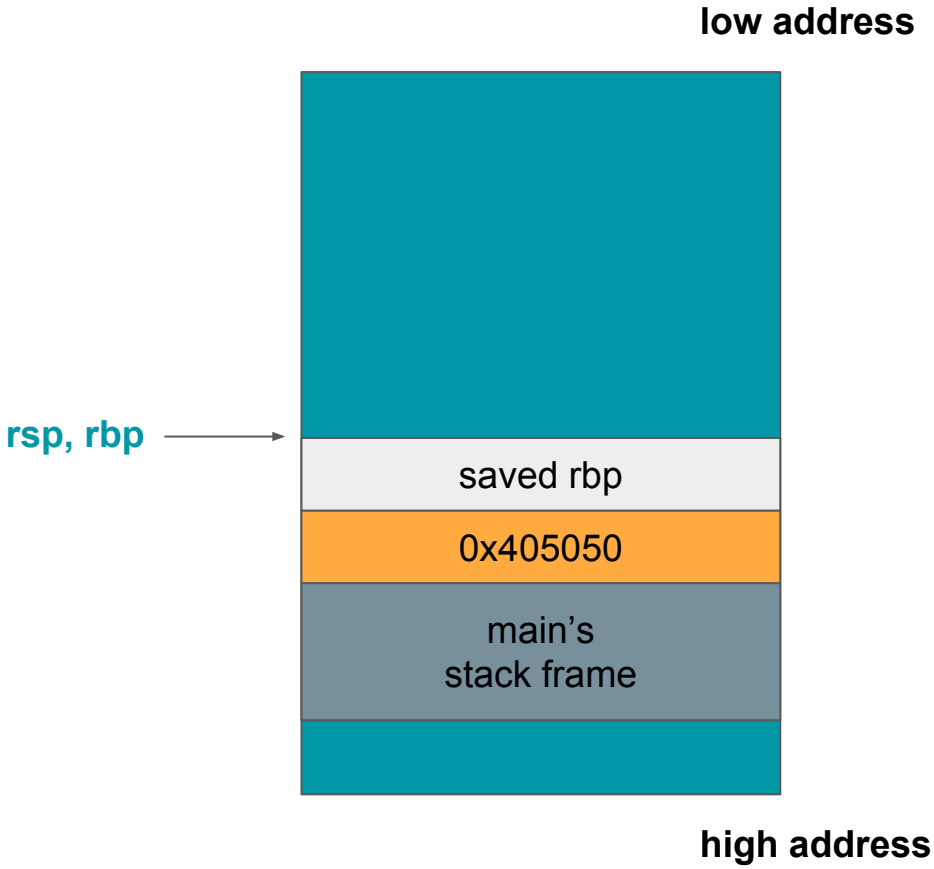


```
a:
    push rbp
    mov rbp, rsp
    sub rsp, 0x50
    ...
    leave
    ret

main:
    push rbp
    mov rbp, rsp
    mov rdi, 1234
    mov rsi, 7122
    call a
    leave (assume addr=0x405050)
    ret
```

leave =

mov rsp, rbp

pop rbp

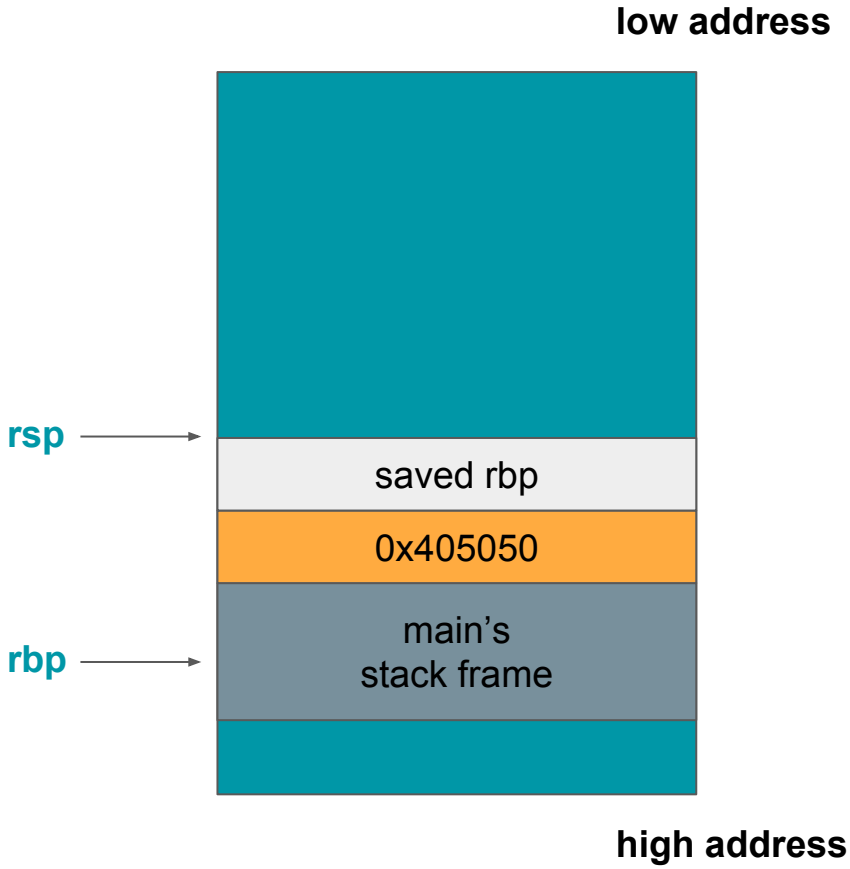


```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```

leave =

mov rsp, rbp

pop rbp

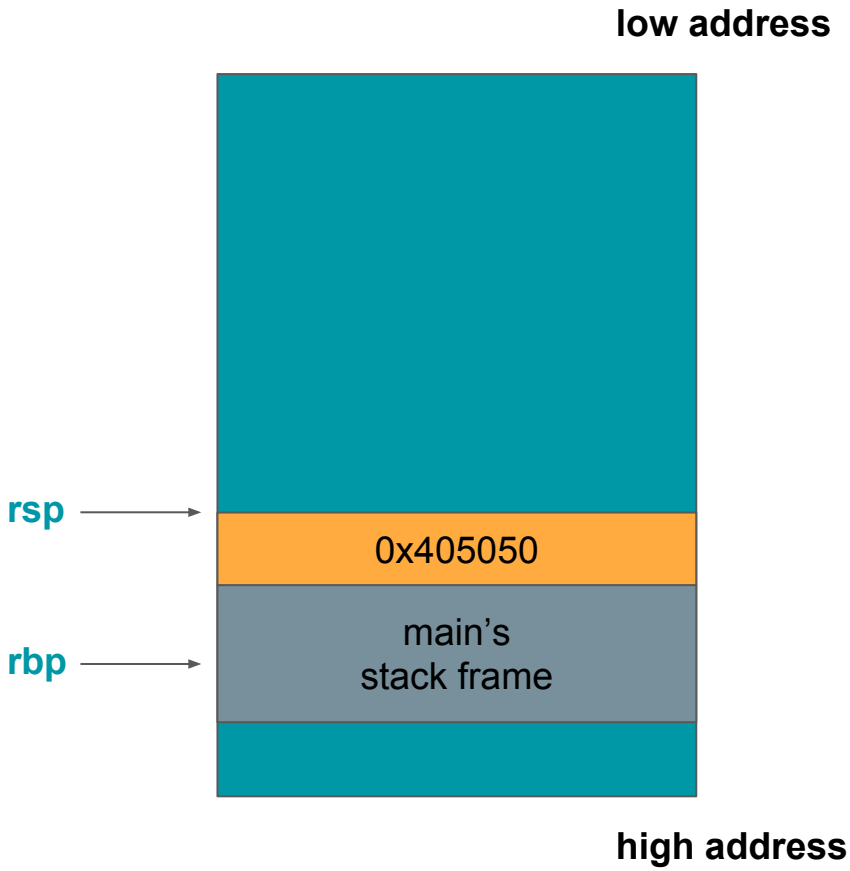


```
a:
        push rbp
        mov rbp, rsp
        sub rsp, 0x50
        ...
        leave
        ret

main:
        push rbp
        mov rbp, rsp
        mov rdi, 1234
        mov rsi, 7122
        call a
        leave (assume addr=0x405050)
        ret
```

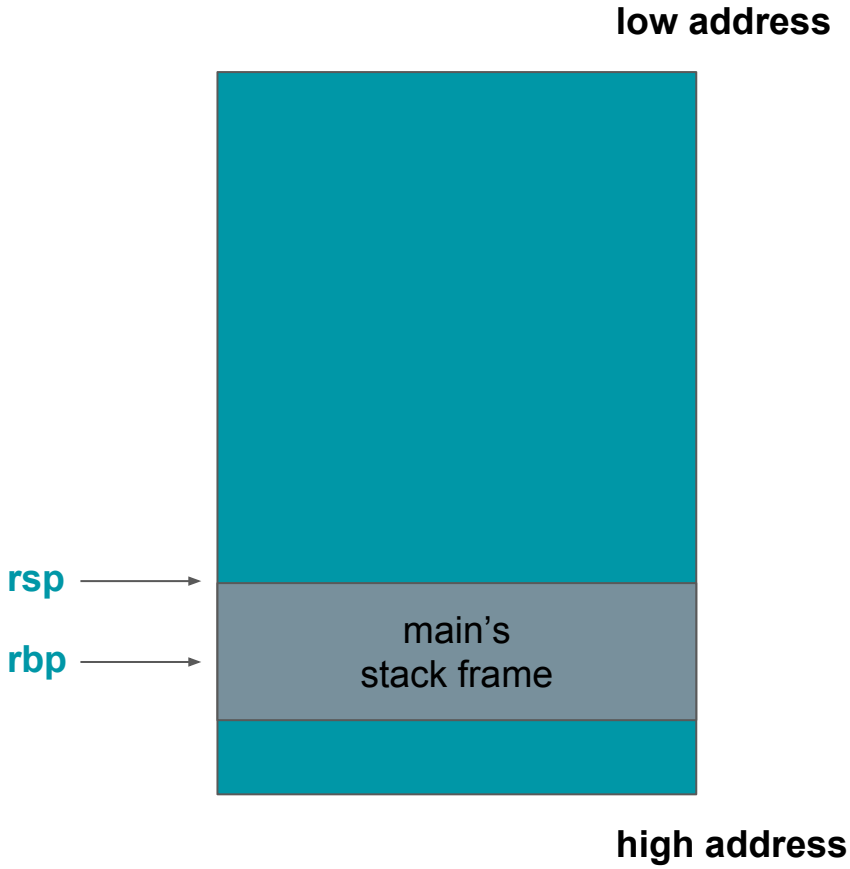
leave =

mov rsp, rbp

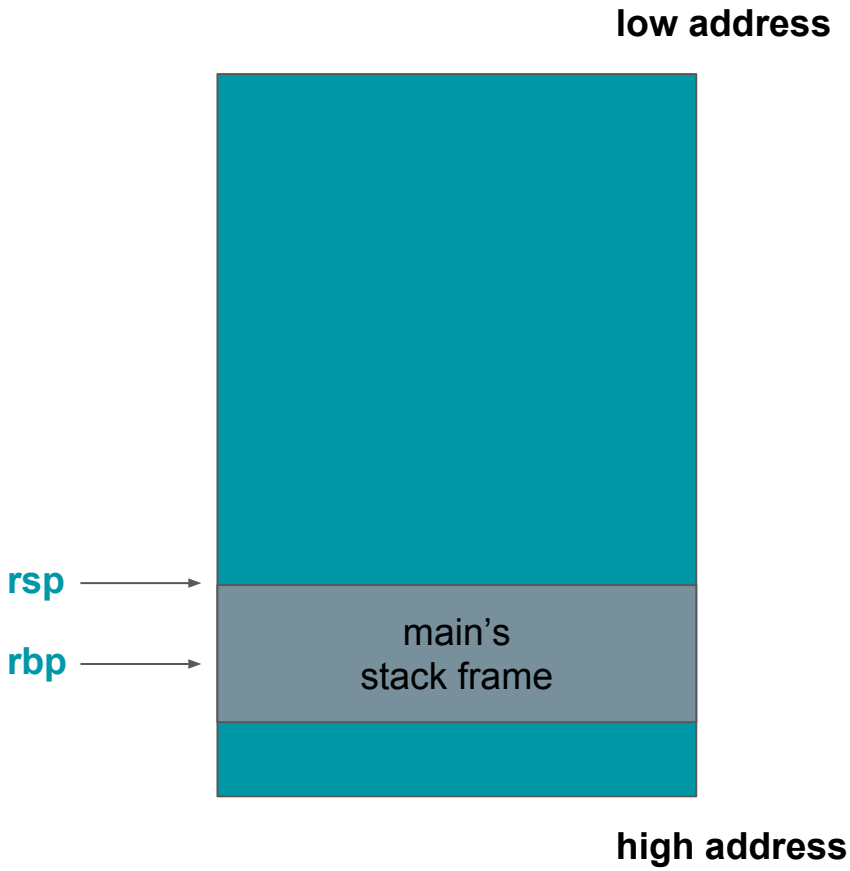
pop rbp



```
a:
      push rbp
      mov rbp, rsp
      sub rsp, 0x50
      ...
      leave
      ret          set $rip = $stack.top

main:
      push rbp
      mov rbp, rsp
      mov rdi, 1234
      mov rsi, 7122
      call a
      leave (assume addr=0x405050)
      ret
```



```
a:
      push rbp
      mov rbp, rsp
      sub rsp, 0x50
      ...
      leave
      ret

main:
      push rbp
      mov rbp, rsp
      mov rdi, 1234
      mov rsi, 7122
      call a
      leave (assume addr=0x405050)
      ret
```



x64 assembly

- System call
  - Instruction: syscall
  - rax: syscall number
  - arguments: rdi, rsi, rdx, rcx, r8, r9
  - return value: rax
  - http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

```
Assume buf = 0x602000
read(0, buf, 0x100):
xor rdi, rdi
mov rsi, 0x602000
mov rdx, 0x100
mov rax, 0
syscall
```

x64 assembly

- How to compile
  - Assembly code in hello.s
  - `nasm -felf64 hello.s -o hello.o`
  - `ld -m elf_x86_64 hello.o -o hello`

x64 assembly

- Shellcode
  - 顧名思義，攻擊者主要注入程式碼後的 □的為拿到 shell，故稱 shellcode
  - 由□系列的 machine code 組成，最後□的可做任何攻擊者想做的事
- 產□ shellcode
  - `objcopy -O binary hello.bin shellcode.bin`
  - `xxd -i shellcode.bin`

x64 assembly

- 用 pwntools 產生 shellcode
  - asm() - assembly
  - disasm() - disassembly

30 mins

LAB 1-2

Lazy binding

- Dynamic linking 的程式在執□過程中，有些 library 的函式可能到結束都不會執□到
- ELF 採取 Lazy binding 的機制，在第□次 call library 函式時，才會去尋找函式真正的位置進□ binding

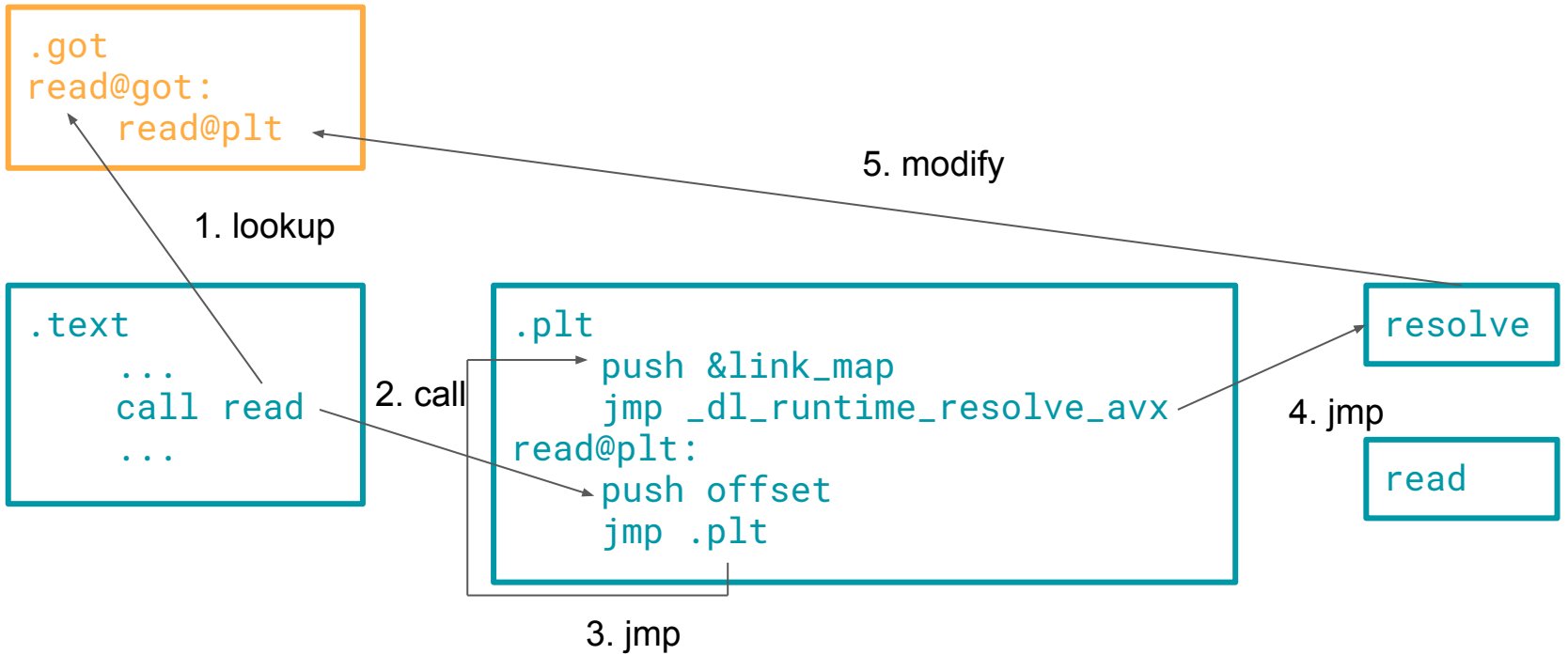
GOT (Global Offset Table)

- library 的位置再載入後才決定，因此無法在 compile 後，就知道 library 中的 function 在哪，該跳去哪
- GOT 為□個函式指標陣列，儲存其他 library 中，function 的位置，但因 lazy binding 的機制，並不會□開始就把正確的位置填上，□是填上□段 plt 位置的 code

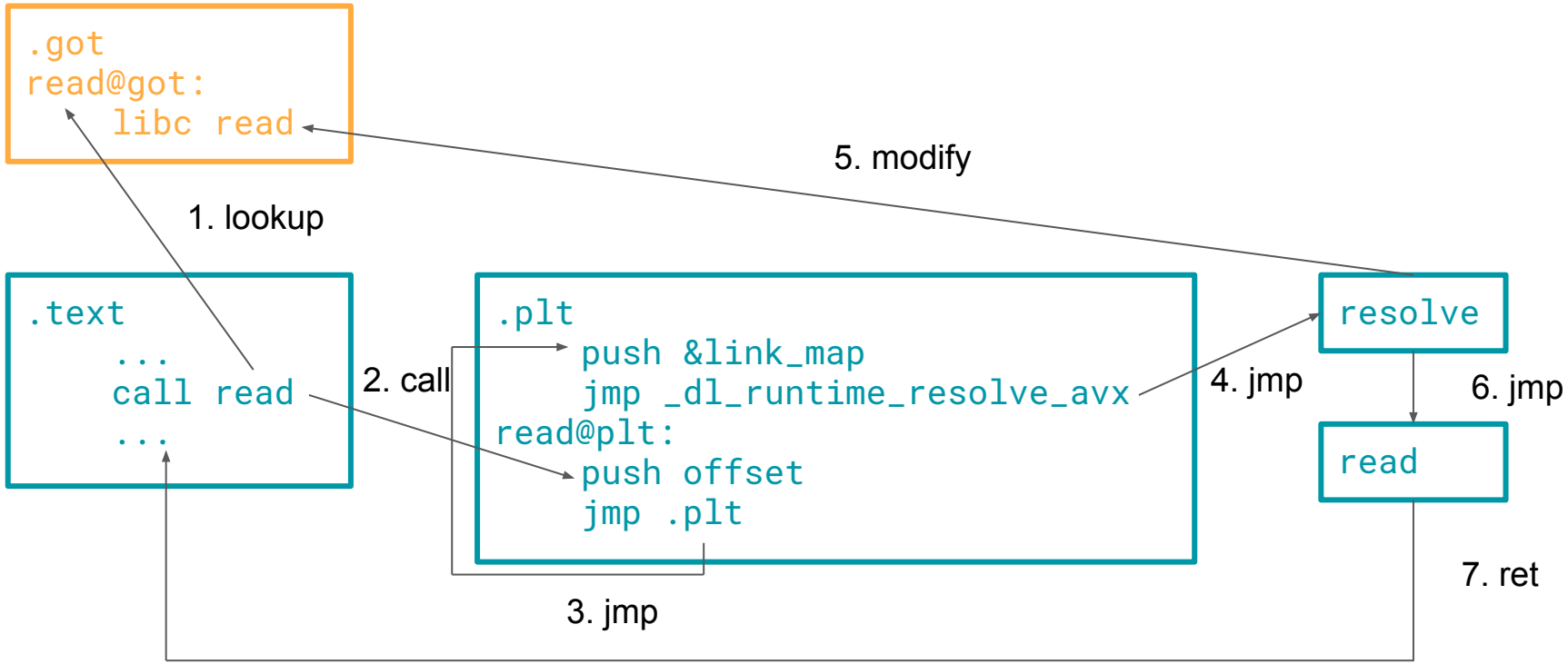
每個function自己的名字

- 當第一次執□到 library 的 function 時，會跳到 plt 去，plt 會去呼叫 _dl_fixup()，才會真正去尋找 function，最後再把 GOT 中的位置填上真正 function 的位置，這樣之後再 call 到這個 function 就有 offset 直接跳到 function 裡

GOT (Global Offset Table)



GOT (Global Offset Table)

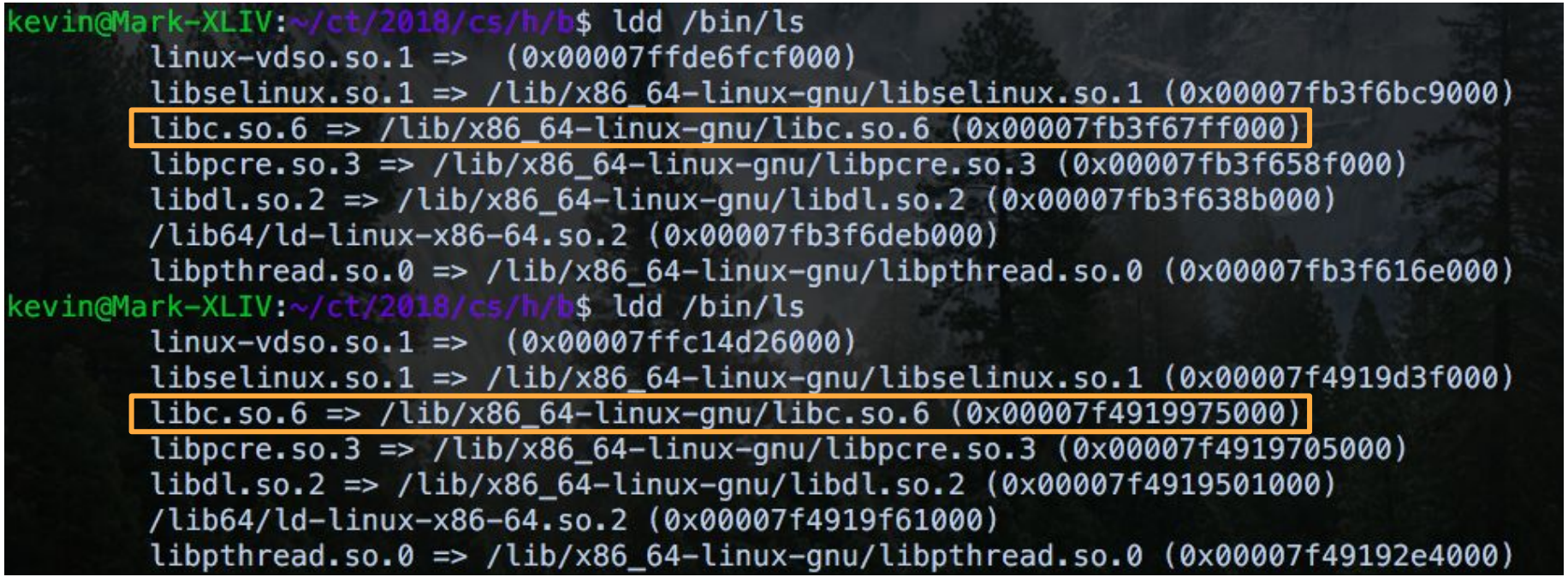


ASLR

- 記憶體位置隨機變化
- 每次執□程式時，stack 、heap、library 位置都不□樣
- 查看是否有開啟 ASLR
  - cat /proc/sys/kernel/randomize_va_space
  - 0 - disable
  - 1 - enable stack
  - 2 - enable all

ASLR

ldd = 看loaded libs



checksec 看binary的保護

- RELRO (RELocation Read Only)
- Canary
- NX (No eXecute)
- PIE (Position Independent Executable)

Text

```
kevin@Mark-XLIV:~$ checksec ctf/2018/csie2018/hw0/bof/bof
[*] '/home/kevin/ctf/2018/csie2018/hw0/bof/bof'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

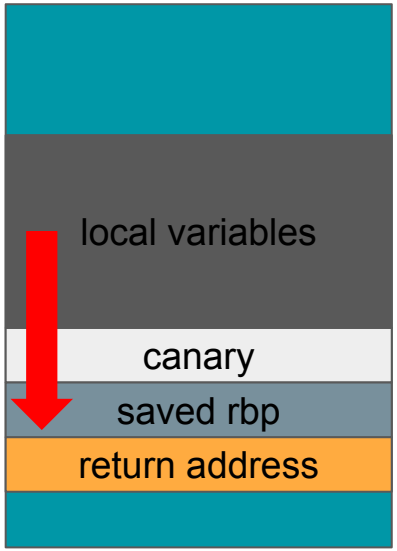
checksec

- RELRO (RELocation Read Only) 犧牲效能換安全性，lazy binding become useless
  - No/Partial/Full
    - No RELRO - link map 和 GOT 都可寫
    - Partial RELRO - link map 不可寫，GOT 可寫
    - Full RELRO - link map 和 GOT 都不可寫
      - 會在 load time 時將全部 function resolve 完畢
      - No lazy binding

checksec

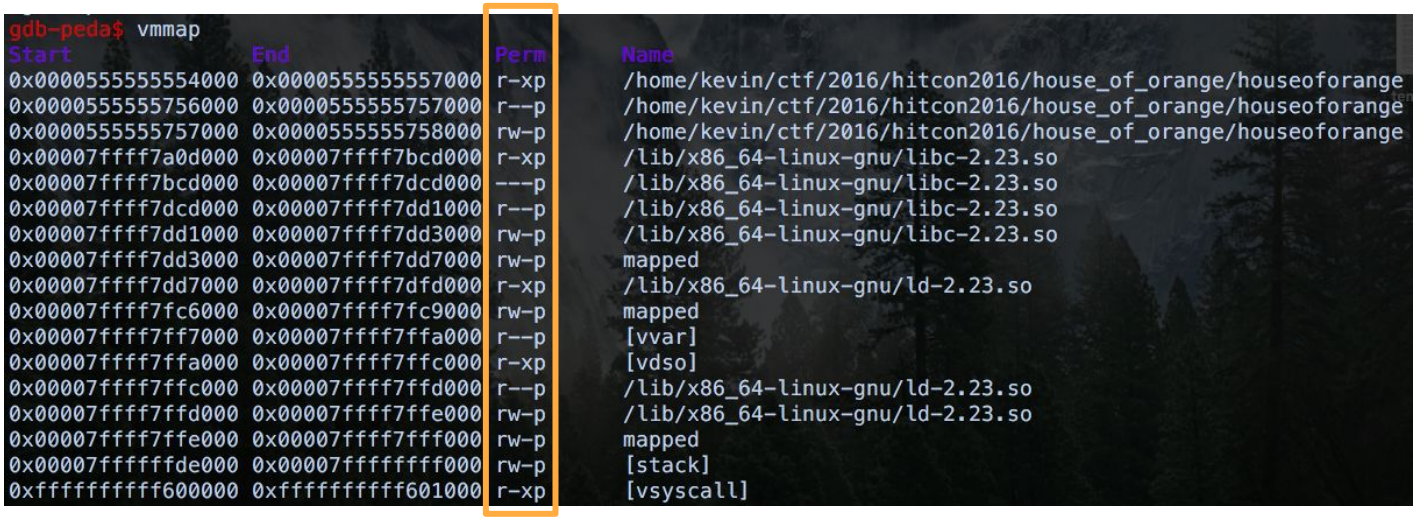
- Canary 有點像隔板的概念? random 7 bytes+lsb一定是0x00 => %s讀到0x00就會中斷，所以可以擋住 (stack gard)
  - 在 rbp 之前塞一個 random 值，在 ret 之前檢查那個 random 值有沒有被改變，有的話代表有 overflow，我們就讓程式 abort

```
kevin@Mark-XLIV:~/ct/2018/cs/h/b$ ./a.out
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```



checksec

- NX (No eXecute)
  - 又稱 DEP (Data Execution Prevention)
  - 可寫的不可執行，可執行的不可寫　不執行data



checksec

- PIE (Position Independent Executable) 讓你不知道要跳到哪
  - 一般預設沒開啟的情況下程式的 data 段及 code 段會位置是固定的
  - 但開啟之後 data 及 code 也會跟著 ASLR

.text永遠都是0x400000

one_gadget

- ROP 時可以跳一次就 get shell，不用自己慢慢堆 execve(“/bin/sh”, 0, 0)

有時沒辦法自己堆shellcode,但libc裡面常常會有這些好東西

```
kevin@Mark-XLIV:~$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

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

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

LD_PRELOAD 環境要注意

- 當本機的 libc 跟題目給的 libc 不一樣的時候，要用題目的 libc 跑程式
- `LD_PRELOAD=./libc.so.6 ./bof`

alarm -> isnan

防掛機的 -> 會影響debug，乾脆把它改掉，比較好debug

- 很多題目會有 alarm，動態逆向的時候很煩
- 直接用文字編輯器打開 binary，把所有的 alarm 改成 isnan
- Preeny

直接開binary, search alarm, 把它改成isnan
(isnan的字串長度跟alarm一樣,所以不會掛掉)

  - https://github.com/zardus/preeny
  - `LD_PRELOAD=~/preeny/x86_64-linux-gnu/dealarm.so`

Thanks for Listening