Buffer Overflow

nae @ NCTUCSC & BambooFox

前言

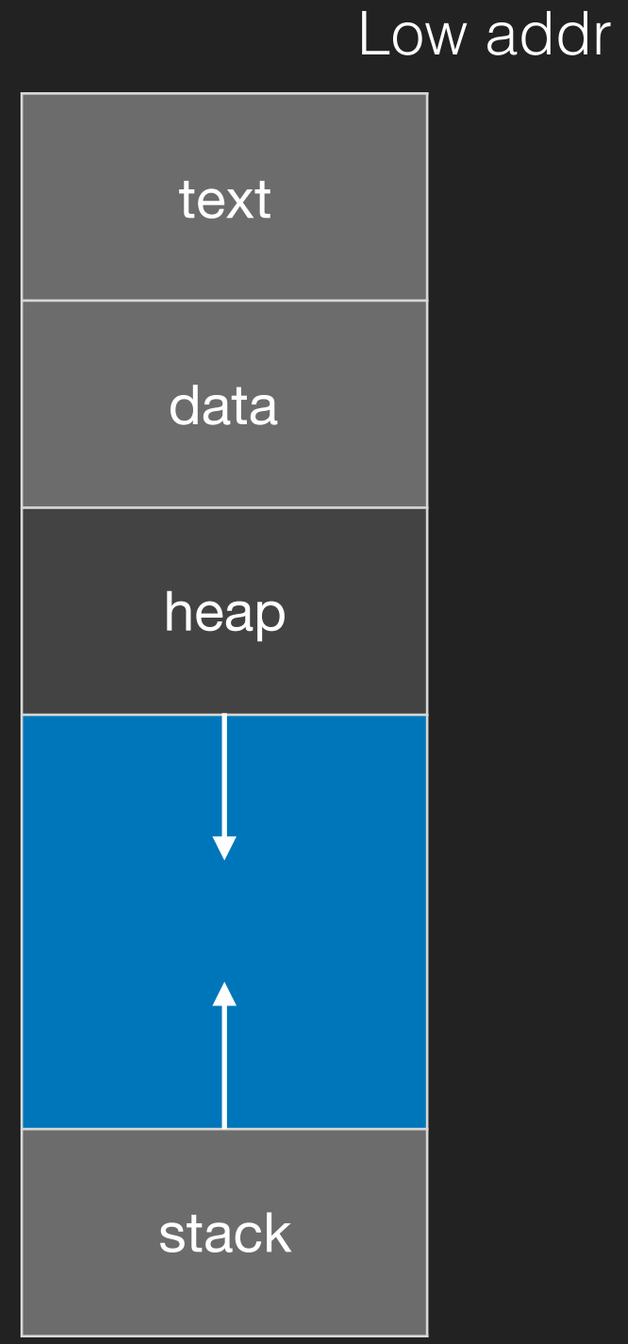
- 環境
  - gcc-4.8
  - Glibc - 2.24
  - Kernel - 4.8.0-59
  - 64-bit OS
- 內容會以 32 位元為主，較好說明與理解

Outline

- Background Knowledge
  - Stack
  - Function Calling Convention
- Buffer Overflow
- Exploit
  - Shellcode
  - Return to text
  - Return to libc
  - Bypass stack guard

What’s stack?

- OS kernel 會將程式 map 到記憶體上
- text
  - 程式指令
- data
  - 程式資料
- heap
  - 動態配置的記憶體
- stack
  - 區域變數、參數、base pointer、return address



Stack

- Stack: LIFO (Last In First Out)

Function Call & Stack

```c
#include <stdio.h>
#include <stdlib.h>

void Func(int a, int b){
    a = 10;
    printf("%d\n", a);
}

int main(){
    int x;
    Func(1, 2);
    return 0;
}
```

1. Call Function

2. Back to main to continue execution

Function Call & Stack

- Function 跑完後，程式需要回到原本呼叫 function 的下一行去執行，而程式就是利用 stack 來記錄下一行指令

Code

```
0x8048456 <main+22>: add    eax,0x1baa
0x804845b <main+27>: sub    esp,0x8
0x804845e <main+30>: push   0x2
0x8048460 <main+32>: push   0x1
=> 0x8048462 <main+34>: call   0x804840b <Func>
0x8048467 <main+39>: add    esp,0x10
0x804846a <main+42>: mov    eax,0x0
0x804846f <main+47>: mov    ecx,DWORD PTR [ebp-0x4]
0x8048472 <main+50>: leave
```

Back to here to continue

Jump to 0x0804840b to execute Func

Function Call & Stack

- Caller Part
- ESP: Stack Pointer
- void Func(int a, int b)
- …

push b

push a

call Func

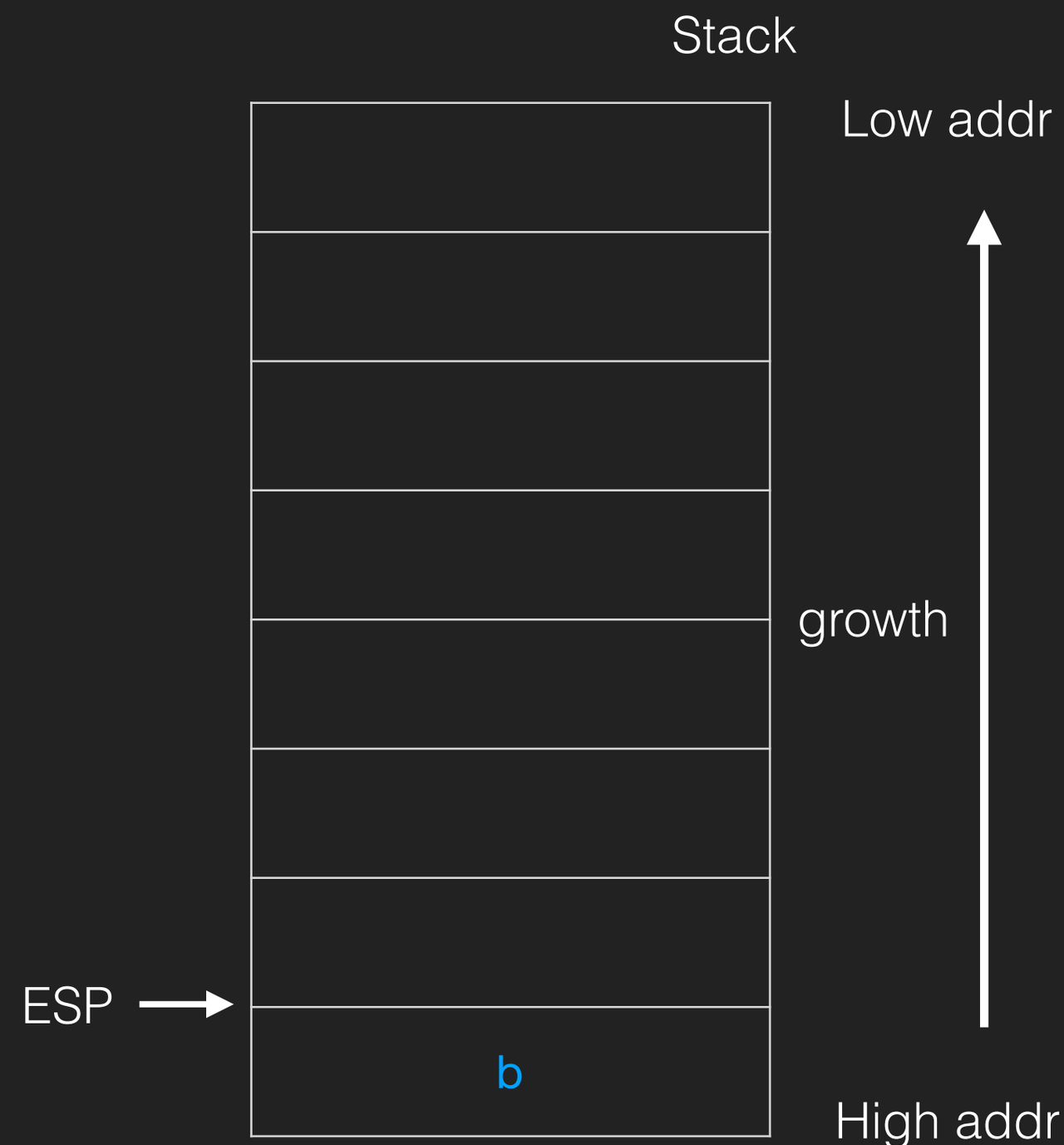
Function Call & Stack

- Caller Part
- ESP: Stack Pointer
- void Func(int a, int b)
- ...

push b

push a

call Func



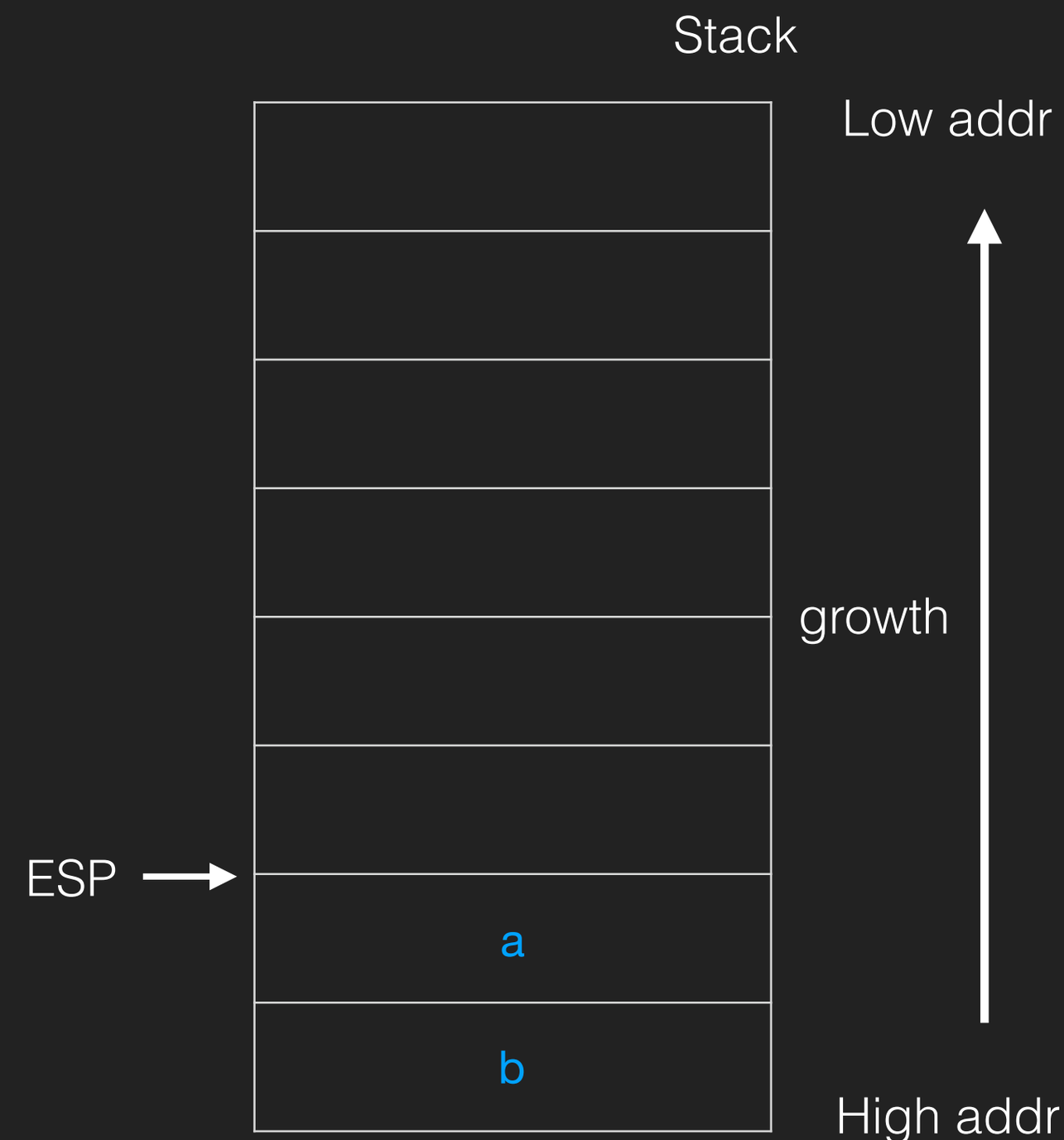
Function Call & Stack

- Caller Part
- ESP: Stack Pointer
- void Func(int a, int b)
- …

push b

push a

call Func



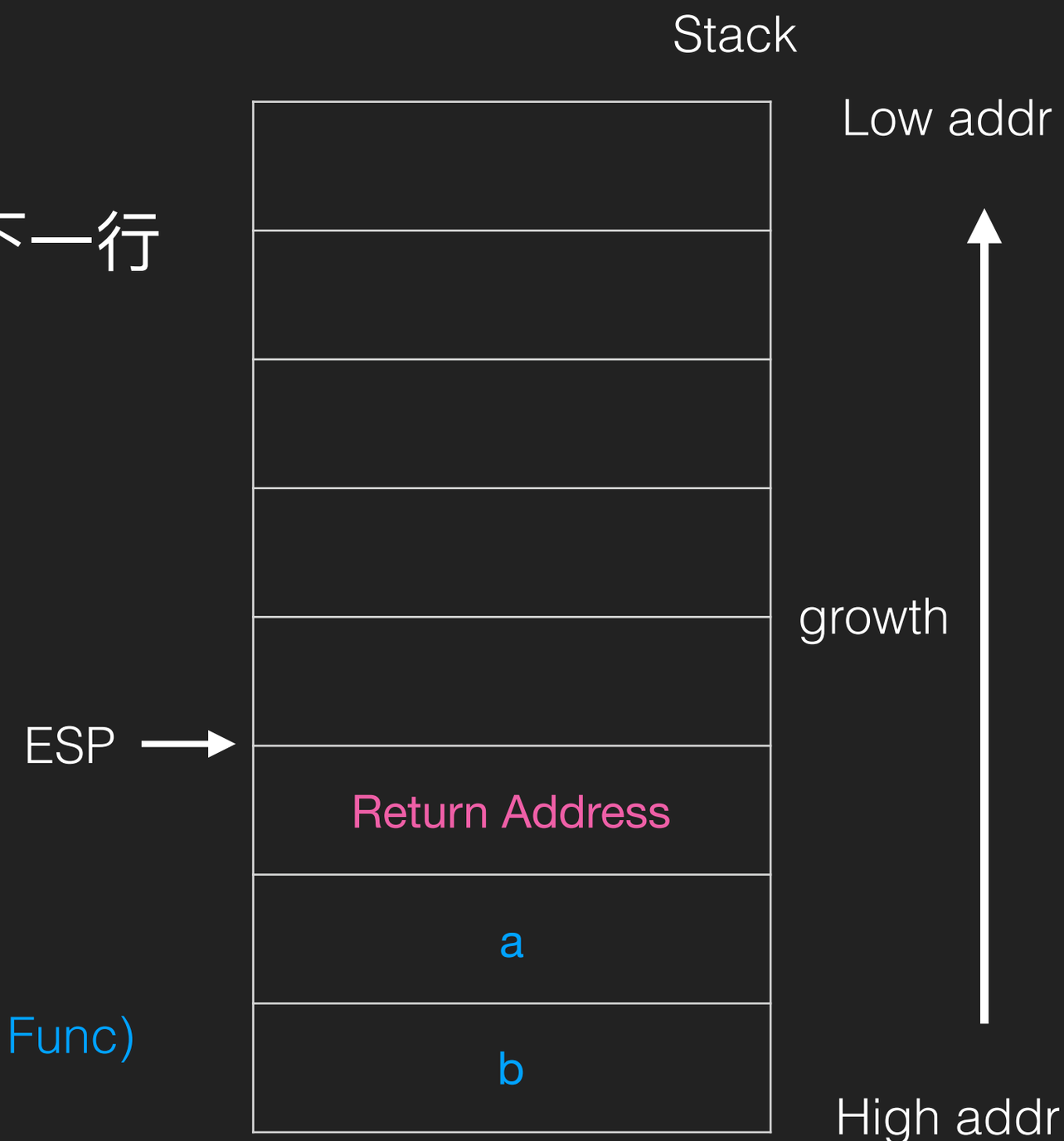
Function Call & Stack

- Caller Part
- Return Address: call Func 下一行指令的位址
- void Func(int a, int b)
- …

push b

push a

call Func (push ret addr; jmp Func)



Function Call & Stack

- Callee Part

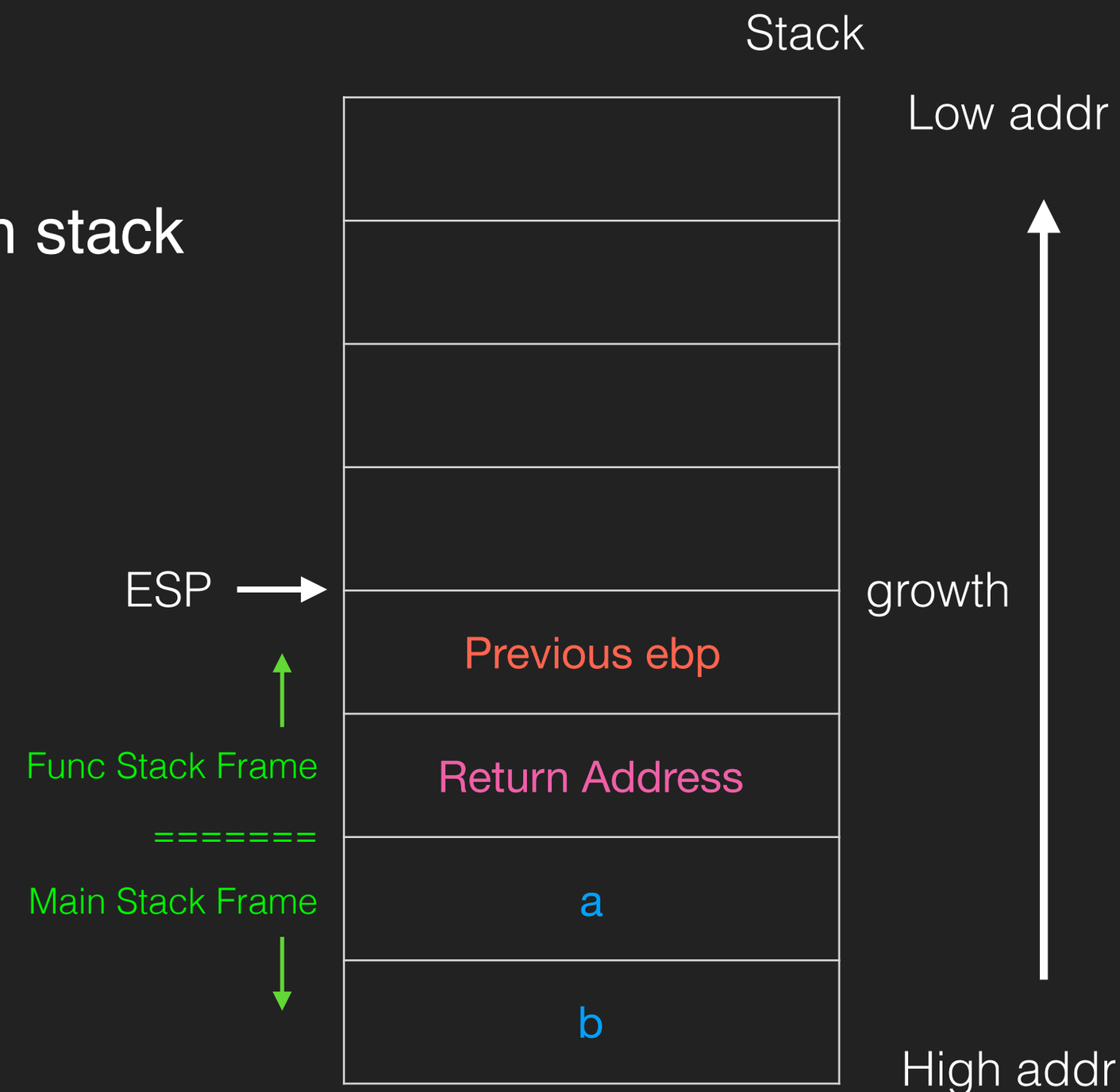
- 此時的 EBP 記錄著 main stack frame base 的位址

- void Func(int a, int b)
  char buf[12];

push ebp

mov ebp, esp

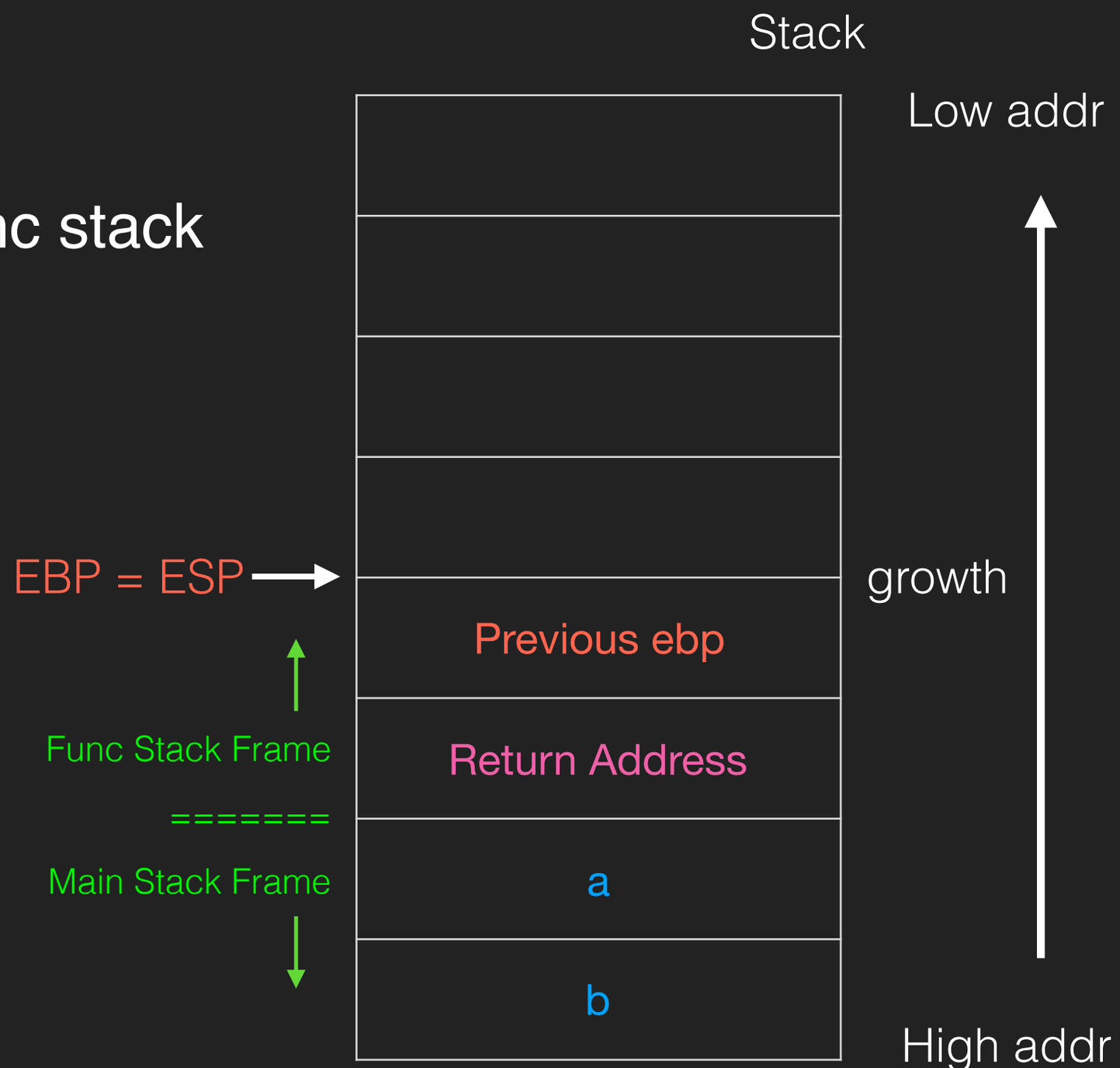
sub esp, 0xc



Function Call & Stack

- Callee Part
- 此時的 EBP 記錄著 Func stack frame base 的位址
- void Func(int a, int b)
  char buf[12];

```
push ebp

mov ebp, esp

sub esp, 0xc
```



Function Call & Stack

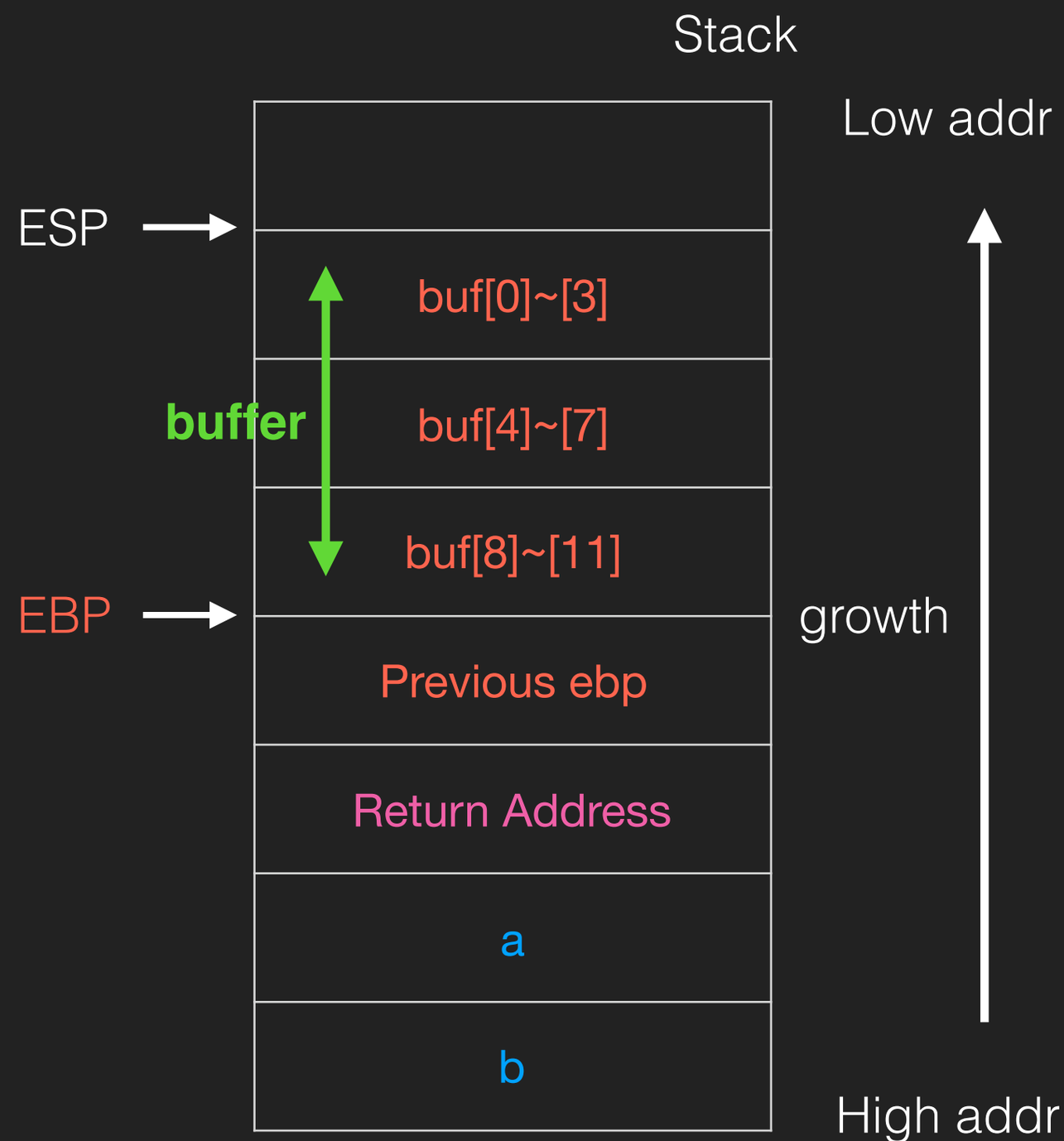
- Callee Part

- void Func(int a, int b)
  char buf[12];

push ebp

mov ebp, esp

sub esp, 0xc



Function Call & Stack

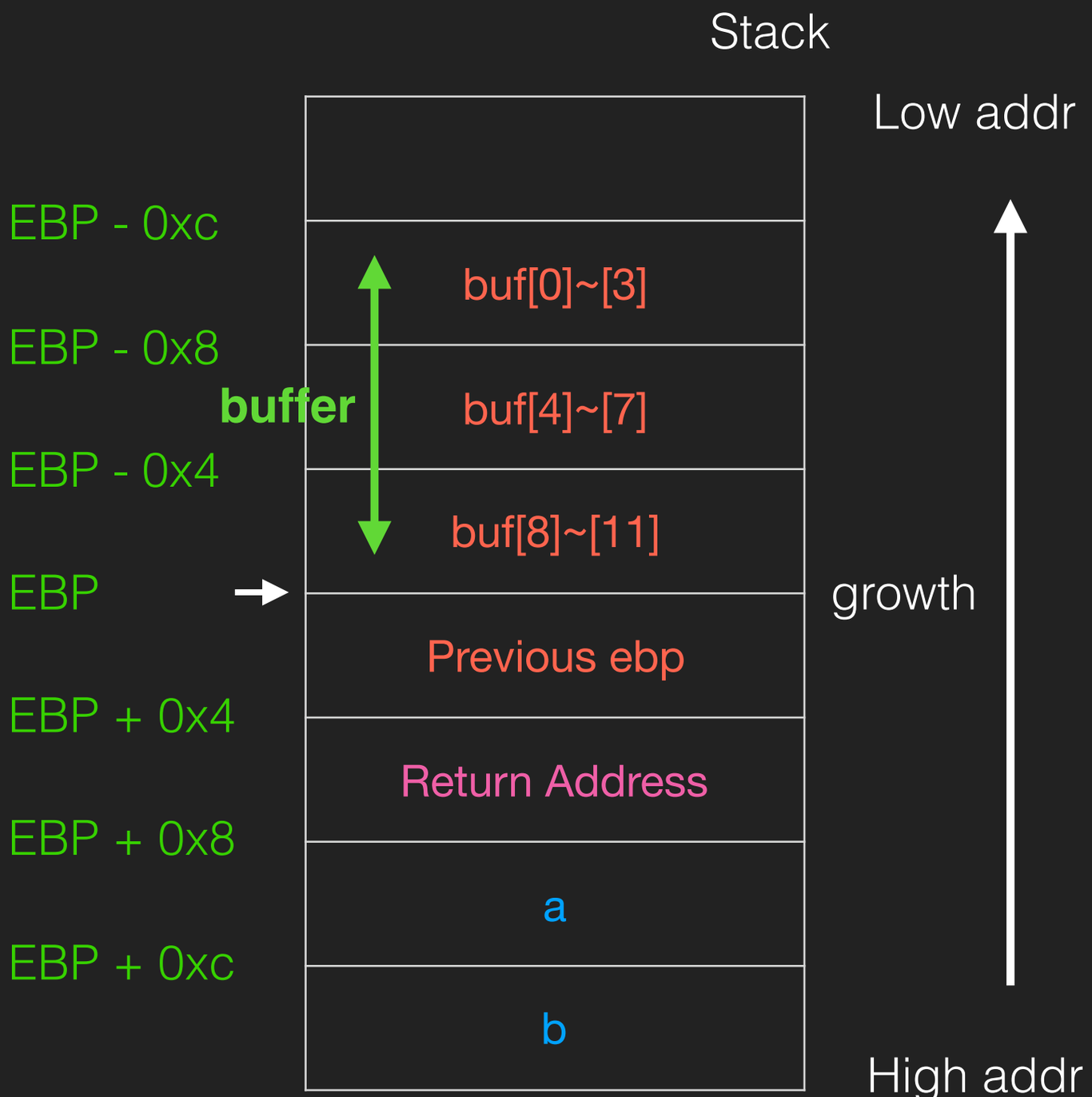
- void Func(int a, int b)
  char buf[12];

push ebp

mov ebp, esp

sub esp, 0xc

…



Buffer Overflow

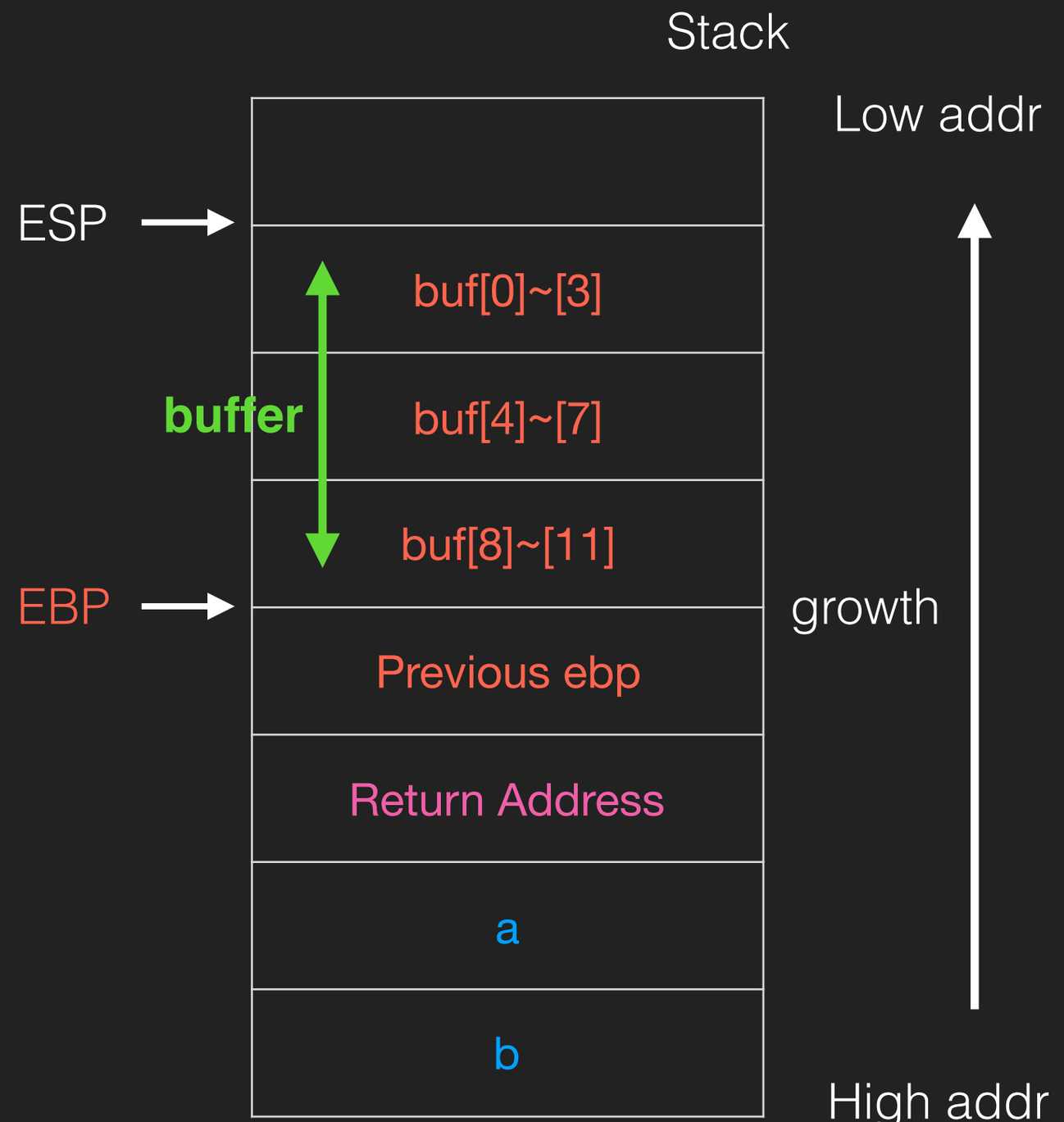
- 程式本身沒有正確檢查輸入的大小，如果輸入的大小比 buffer 還要大就會蓋到其他變數影響程式的執行 (控制變數 or 控制程式流程)

Buffer Overflow

- Unsafe Function
  - ~~gets~~ -> fgets
  - ~~scanf~~ -> never use scanf(“%s”)
  - ~~strcpy~~ -> strncpy
  - …
- Buffer Overflow 其實有很多種，根據不同的 memory 位址有不同的稱呼
  - stack overflow
  - heap overflow

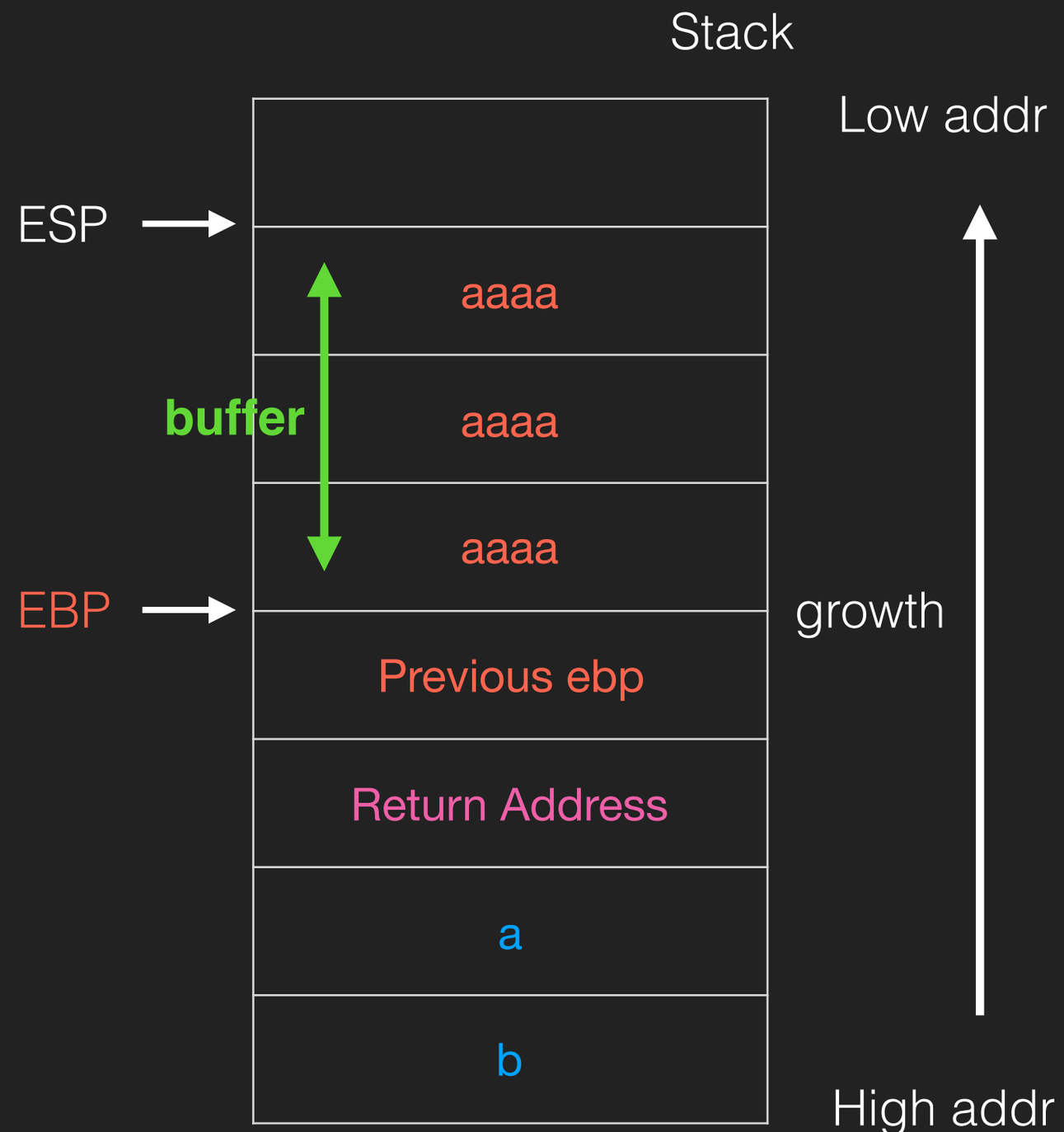
Stack Overflow

- void Func(int a, int b)
  char buf[12];
  gets(buf);



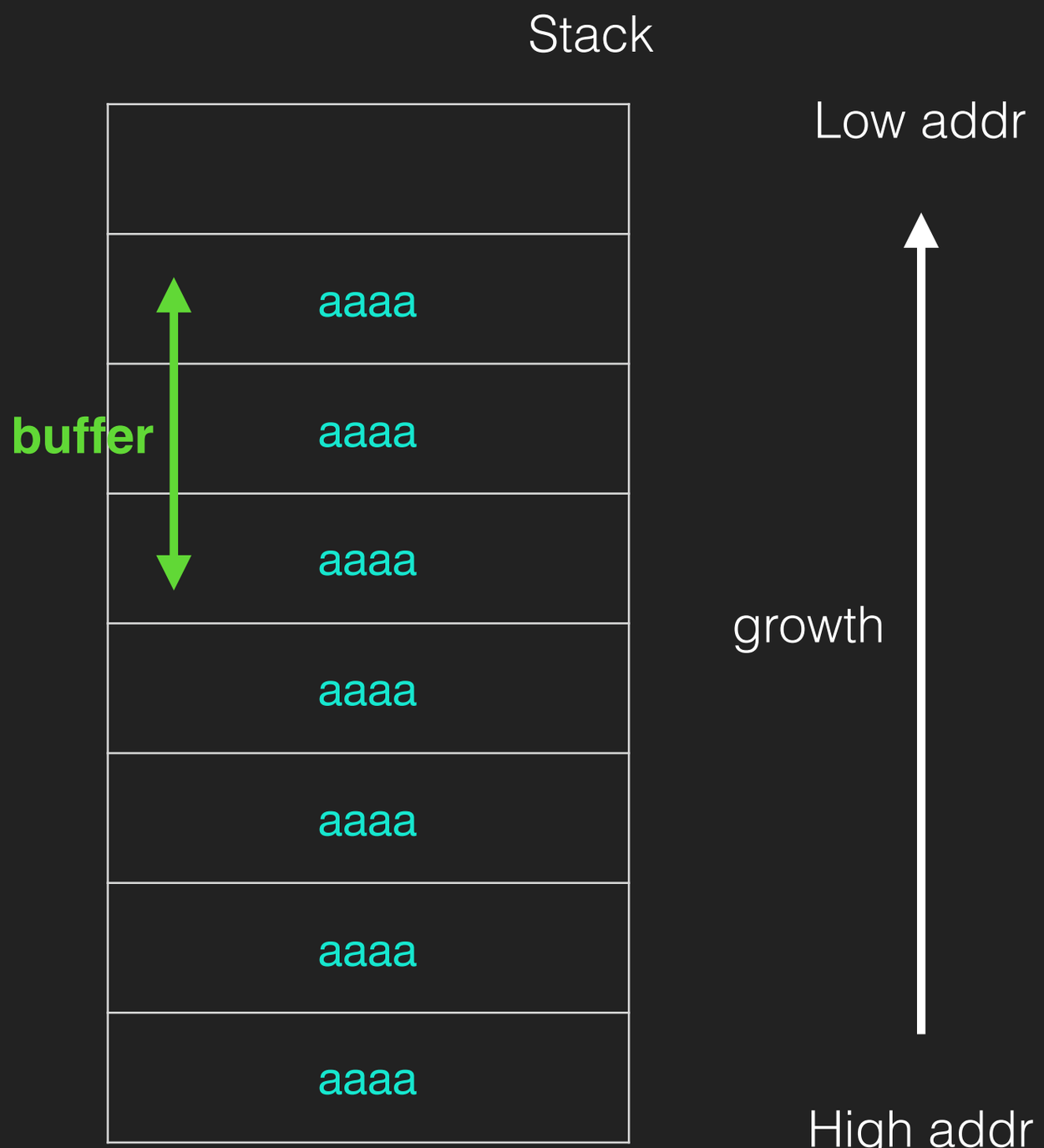
Stack Overflow

- aaaaaaaaaaaaa(a * 12)

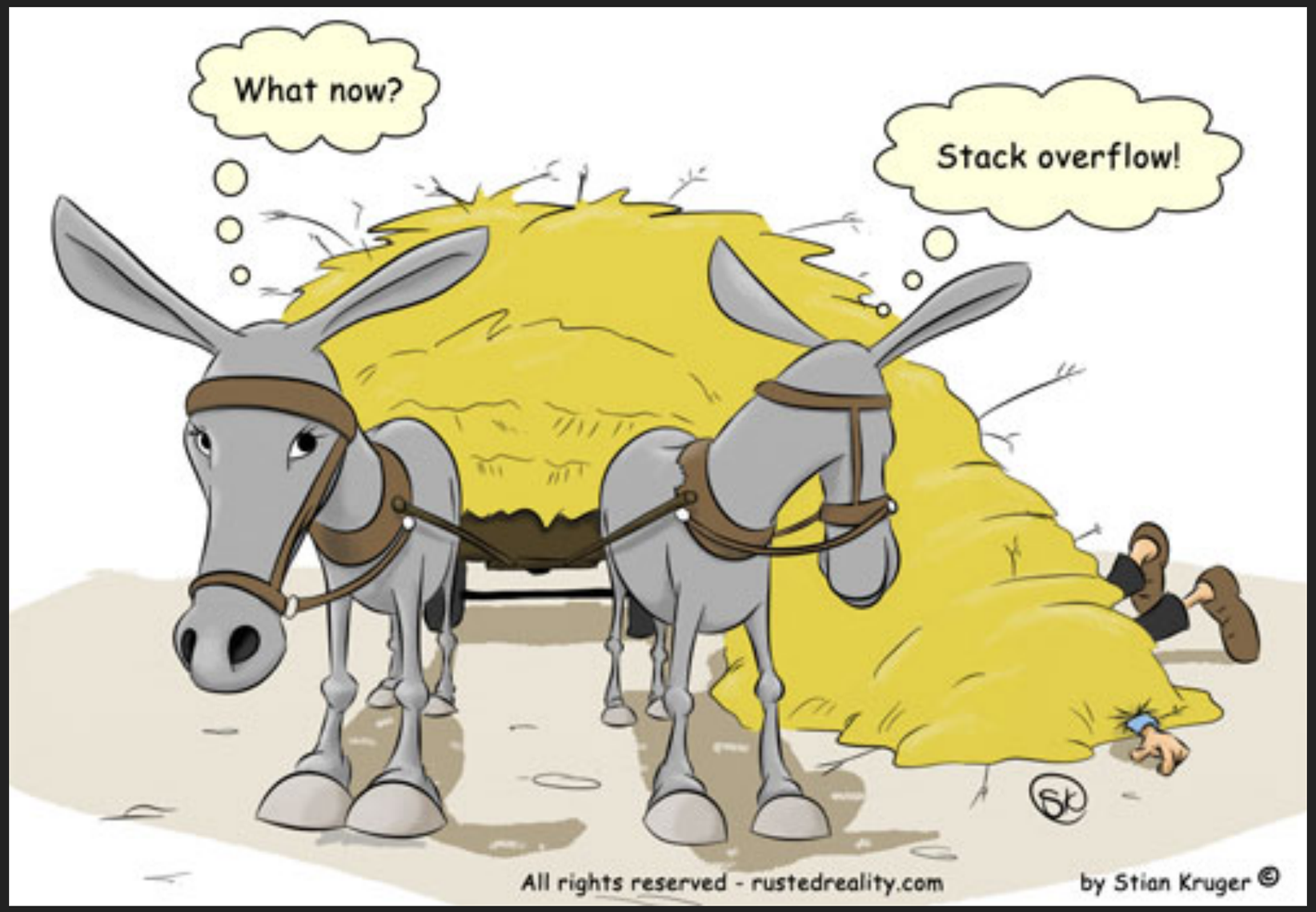


Stack Overflow

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa



Stack Overflow



https://www.linkedin.com/pulse/buffer-overflow-exploits-protection-mechanisms-roman-postanciuc

Stack Overflow

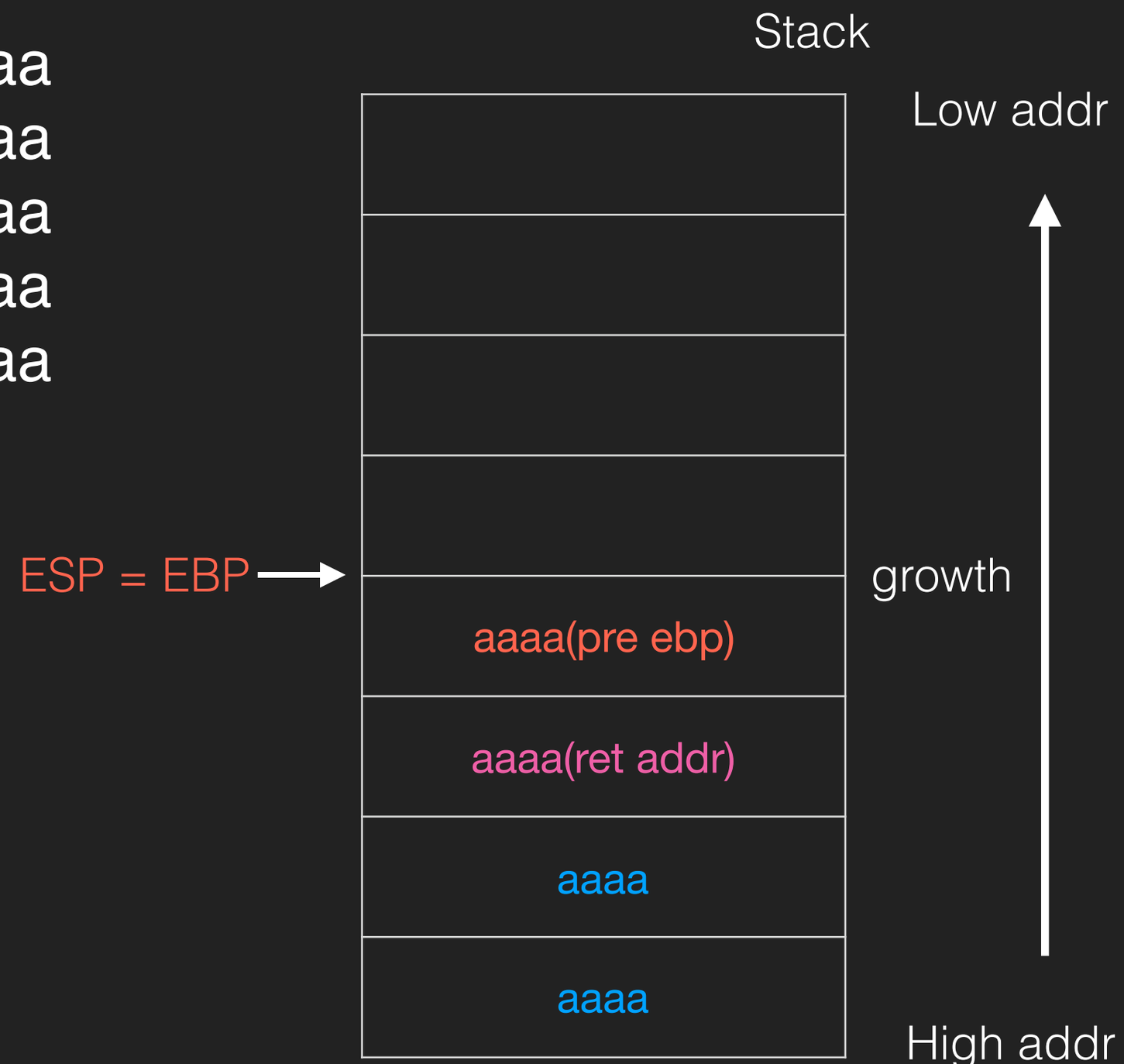
- aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  (a*100)

…

mov esp, ebp

pop ebp

ret



Stack Overflow

- aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  (a*100)

…

mov esp, ebp

pop ebp ; ebp == 0x61616161

ret

Stack Overflow

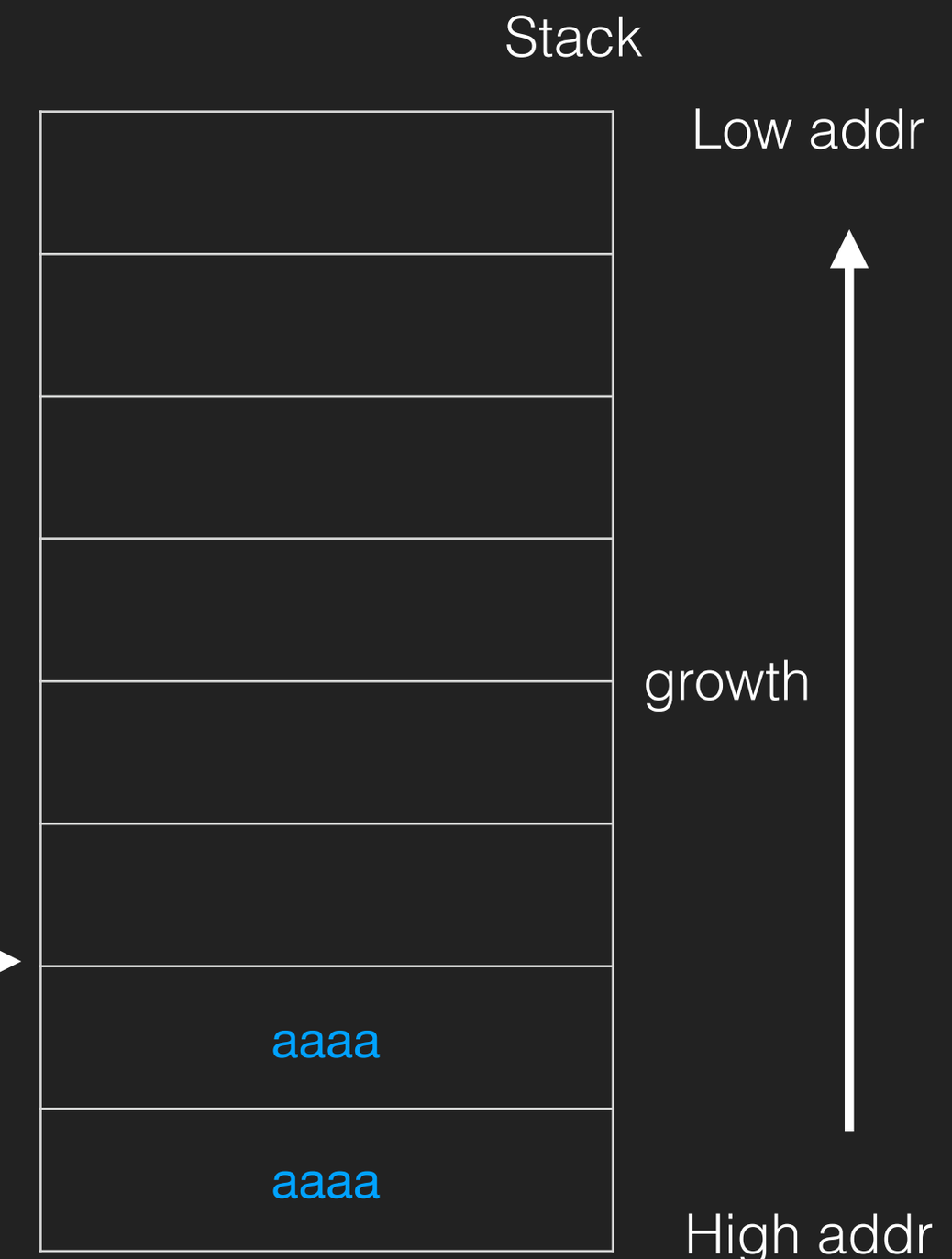
- aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  aaaaaaaaaaaaaaaaaaaaaa
  (a*100)

…

mov esp, ebp

pop ebp ; ebp == 0x61616161

ret (pop eip) ; eip == 0x61616161



Stack Overflow

- 程式會到 0x61616161 的位址拿指令執行

```
(gdb) r
Starting program: /home/naetw/Desktop/demo/bof
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
```

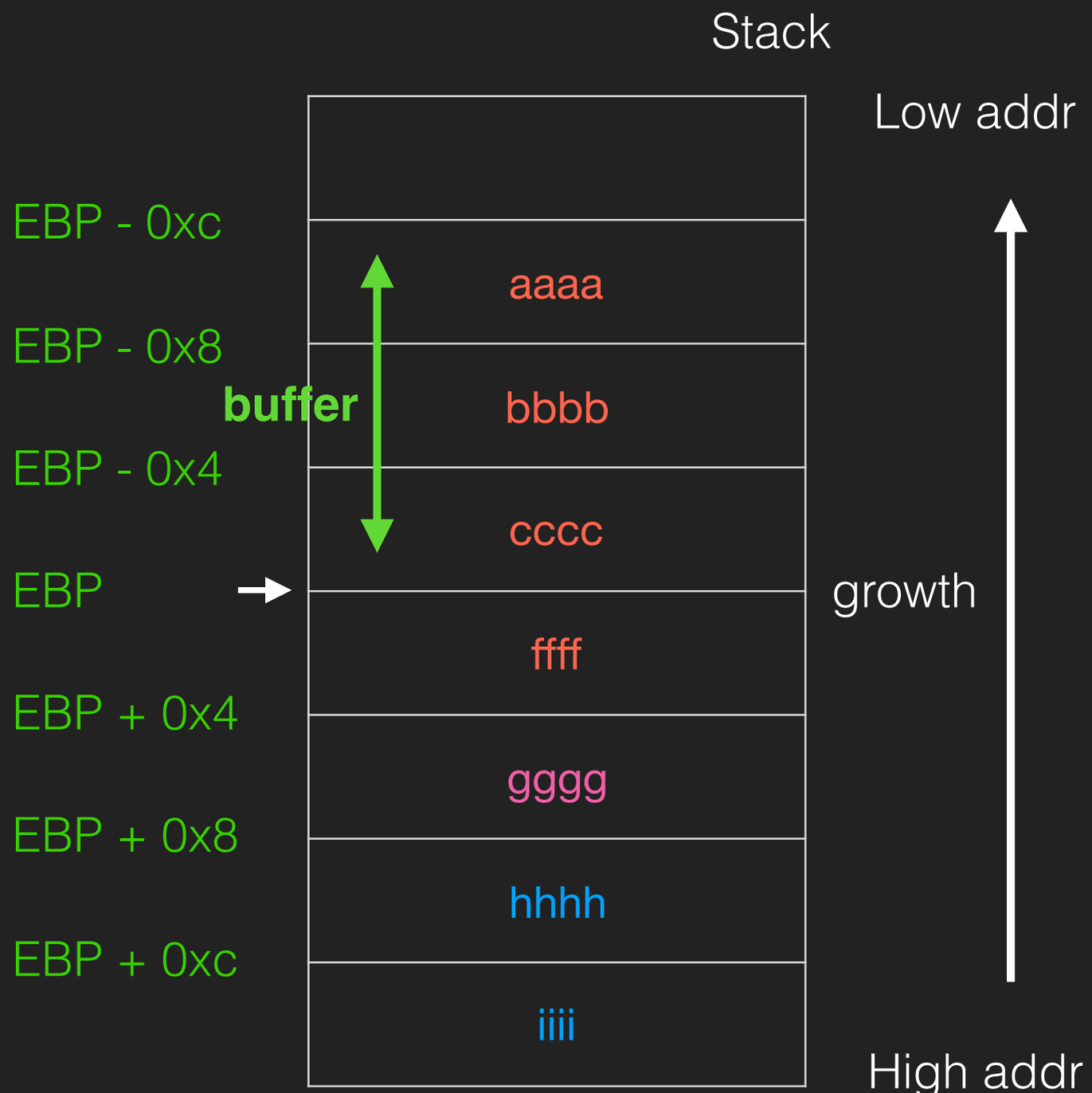
Stack Overflow

- 計算 offset

```
(gdb) r
Starting program: /home/naetw/Desktop/demo/bof
aaaabbbbccccddddeeeeffffggggghhhhiiii

Program received signal SIGSEGV, Segmentation fault.
0x67676767 in ?? ()
```

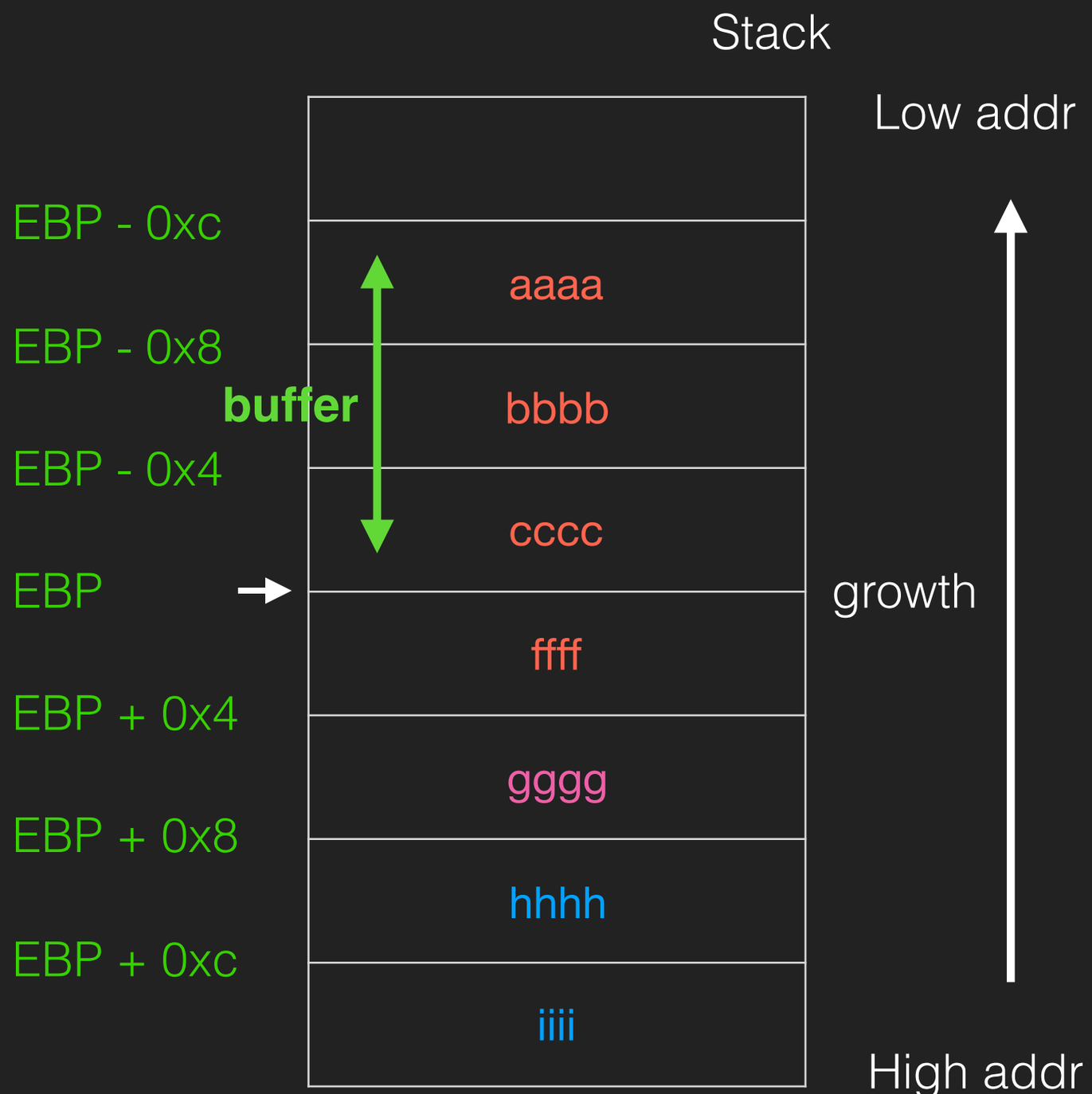
Stack Overflow

- 計算 offset
  - 中間少了 d & e
    - stack alignment



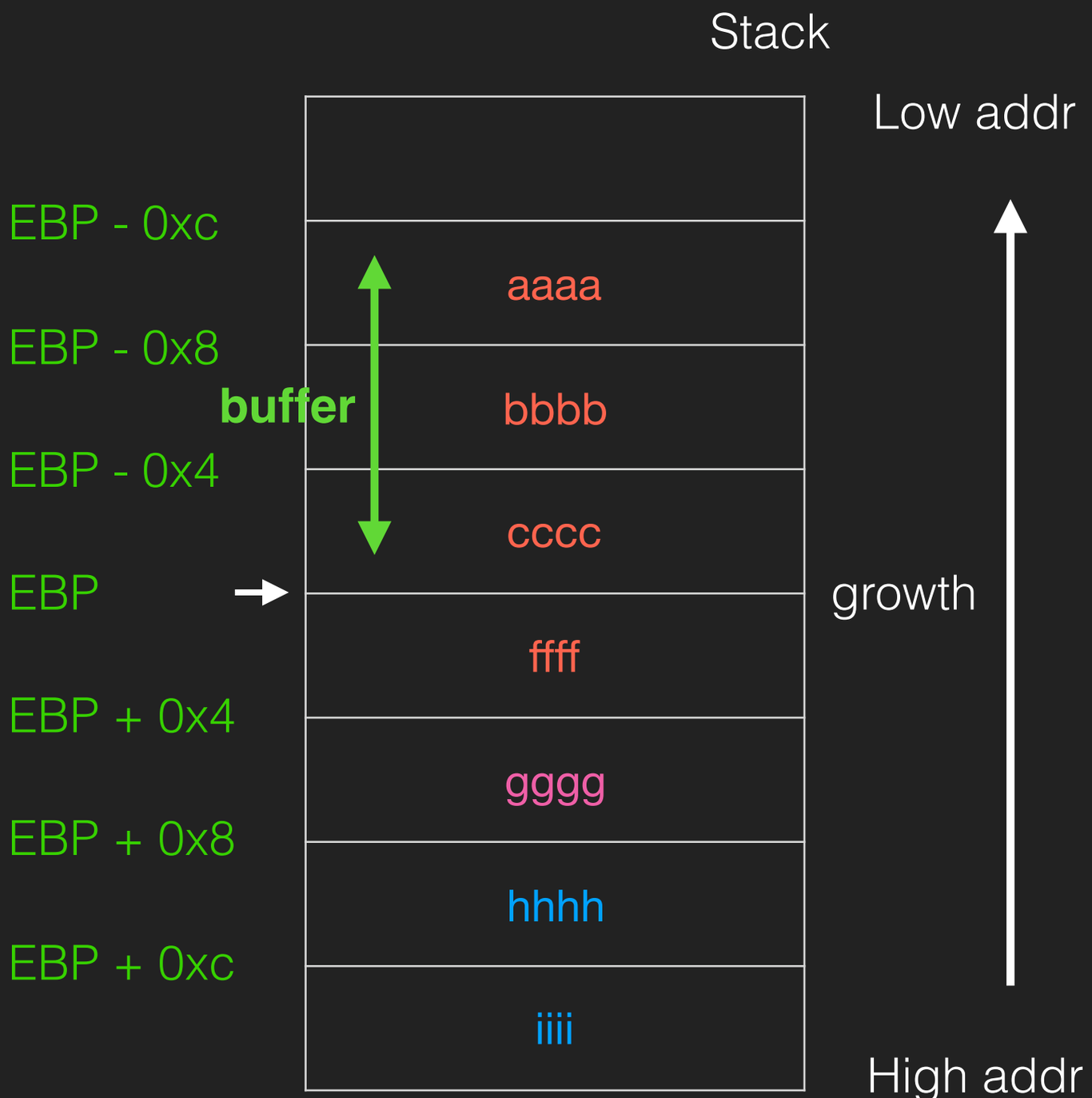
Stack Overflow

- 計算 offset
  - ~~通靈~~
  - 用 gdb
  - cyclic in pwntools



Stack Overflow

- 蓋到 return address 所需的 offset
  - 12 (buf)
  - 8 (alignment)
  - 4 (ebp)
  - 12 + 8 + 4 = 24 bytes



Exploit

- Shellcode
- Return to text
- Return to libc
- Bypass stack guard

Exploit

- Shellcode
  - 其實就是一段 machine code，而 machine code 就是 CPU 可以直接解讀的資料
  - 把 input data 當成 code 來跑
  - 範例

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31\xd2\xb0\x0b\xcd\x80

Exploit

- Shellcode
  - 這段 shellcode 翻成 assembly 就會是

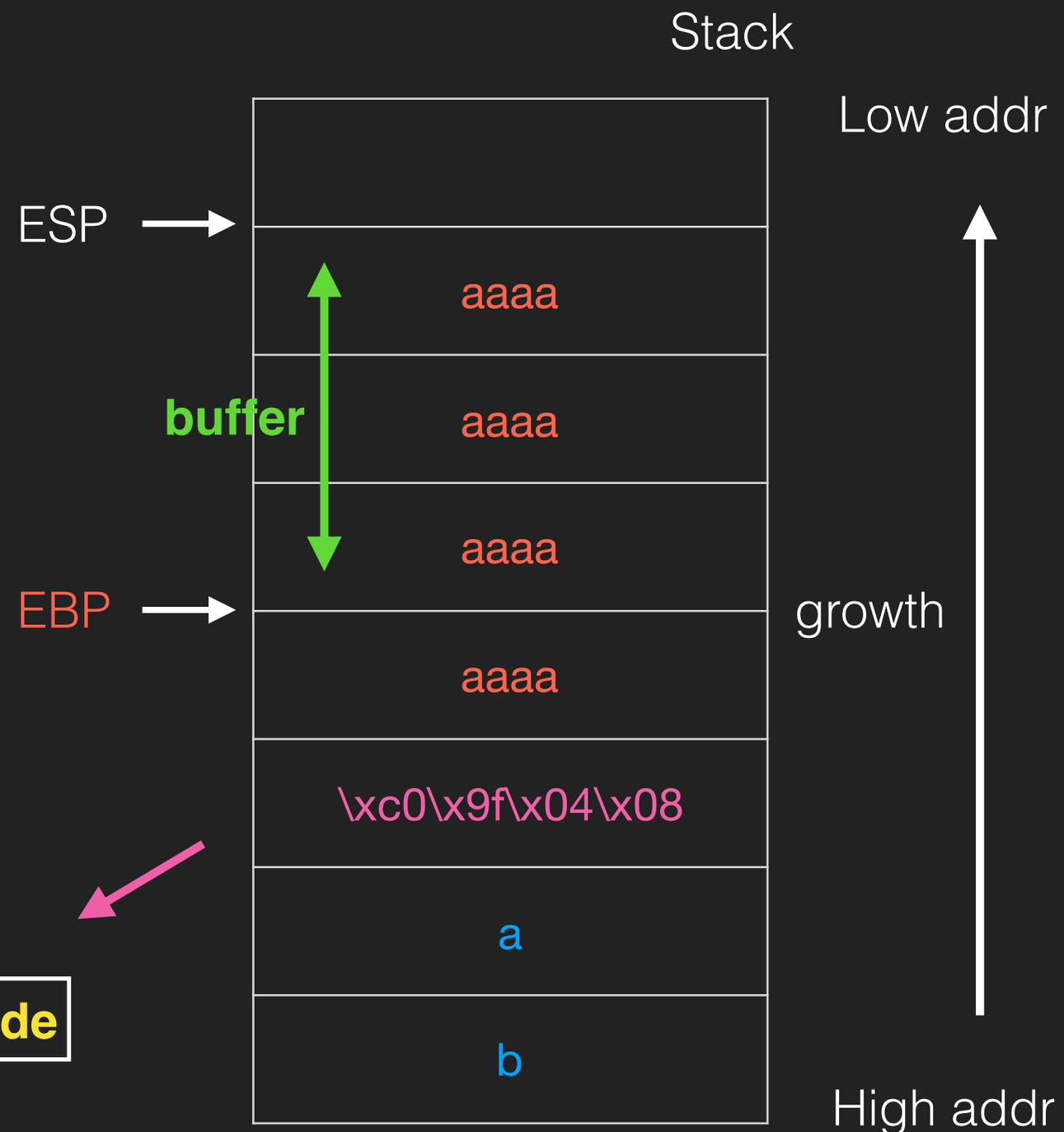
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31\xd2\xb0\x0b\xcd\x80

```
1 xor eax, eax
2 push eax
3 push 0x68732f2f
4 push 0x6e69622f
5 mov ebx, esp
6 xor ecx, ecx
7 xor edx, edx
8 mov al, 0xb
9 int 0x80
```

```
execve("/bin/sh", NULL, NULL);
```

Exploit

- Shellcode
  - 找一段 buffer 把 shellcode 寫上去
  - 利用 buffer overflow 把 return address 覆寫，讓程式跳到 shellcode 上去執行



Protection

- DEP (Data Execution Prevention)
- ASLR (Address Space Layout Randomization)
- Stack Guard (Stack Canary)
- RELRO (Relocation Read Only)
- PIE (Position Independent Executable)

Protection

- DEP (Data Execution Prevention)
  - 放在 **data** buffer 上的 shellcode 沒有執行的權限
    - ❌ stack
    - ❌ heap
    - ❌ .data
    - ❌ .bss

Protection

- DEP (Data Execution Prevention)
  - DEP 開啟/關閉
    - gcc -z execstack
      - execstack --set-execstack
      - execstack --clear-execstack
- 檢查 Program Header - GNU_STACK
  - `readelf --program-headers ${binary}`

Protection

- ASLR (Address Space Layout Randomization)
  - System dependent
    - Linux - `cat /proc/sys/kernel/randomize_va_space`
      - 預設是 2
        - 0 - 關掉 ASLR
        - 1 - stack, shared library, mmap()
        - 2 - 除了 1 的保護之外還加上 brk() 所管理的記憶體位址

Protection

- Stack Guard (Stack Canary)
  - 一個亂數，在 function call 時放進 stack
    - 在 return address & ebp 被 push 進 stack 後，再放上去
  - 在 return 前檢查 canary 是不是正確的

Protection

- Stack Guard (Stack Canary)
  - 開啟/關閉
    - gcc -fstack-protector / gcc -fno-stack-protector
  - 檢查 function symbol
    - `readelf -s ${binary} | grep __stack_chk_fail`

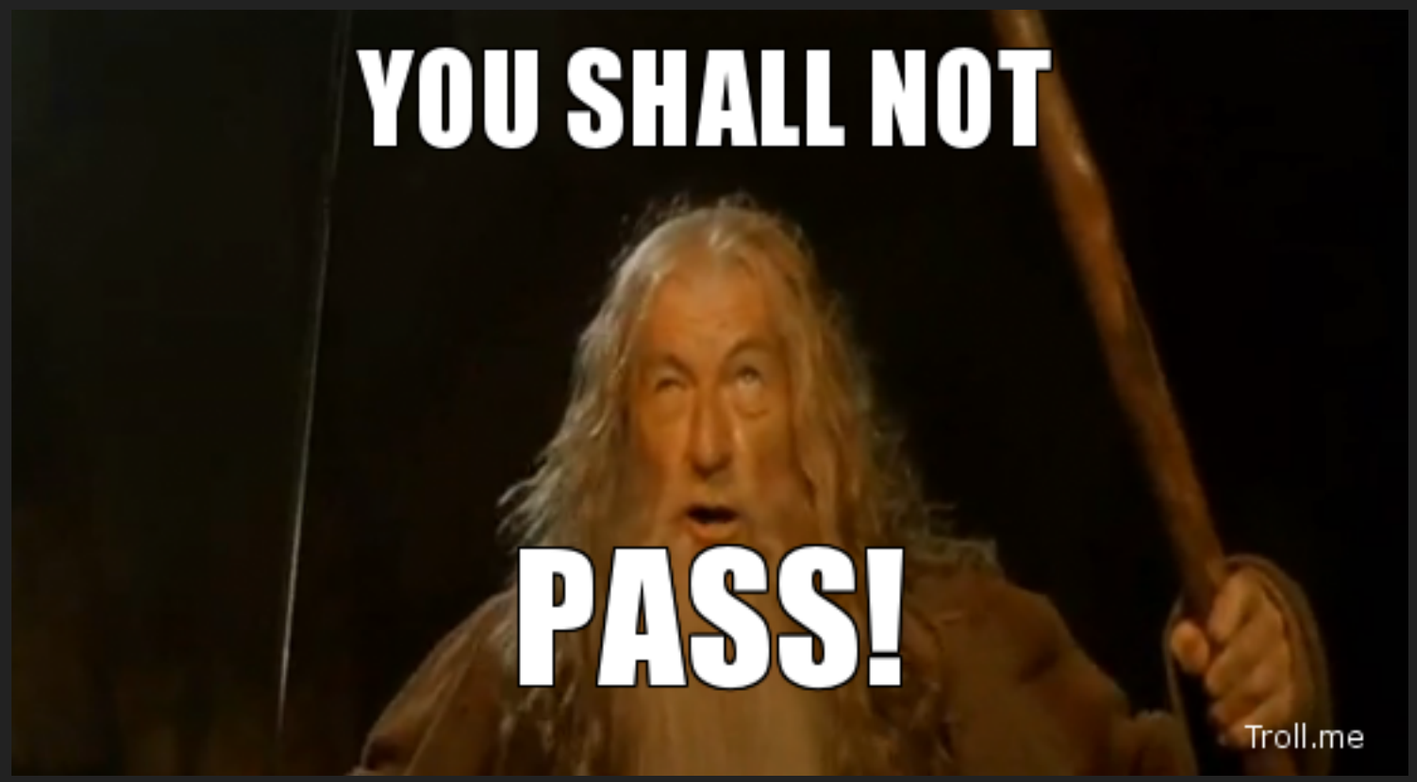
Protection

- Stack Guard (Stack Canary)

```
» ./bof2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./bof2 terminated
[1]    14244 abort (core dumped)  ./bof2
----------------------------------------------------------------
```

Protection

- Stack Guard (Stack Canary)



http://www.troll.me/2011/12/16/you-shall-not-pass-gandalf/you-shall-not-pass-16/

Protection

- Stack Guard (Stack Canary)

```
0804846d <Func>:
 804846d:       55                      push   ebp
 804846e:       89 e5                   mov    ebp,esp
 8048470:       83 ec 14                sub    esp,0x14
 8048473:       65 a1 14 00 00 00       mov    eax,gs:0x14
 8048479:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804847c:       31 c0                   xor    eax,eax
 804847e:       8d 45 f0                lea    eax,[ebp-0x10]
 8048481:       89 04 24                mov    DWORD PTR [esp],eax
 8048484:       e8 a7 fe ff ff          call   8048330 <gets@plt>
 8048489:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 804848c:       65 33 05 14 00 00 00    xor    eax,DWORD PTR gs:0x14
 8048493:       74 05                   je     804849a <Func+0x2d>
 8048495:       e8 a6 fe ff ff          call   8048340 <__stack_chk_fail@plt>
 804849a:       c9                      leave
 804849b:       c3                      ret
```

Protection

- RELRO (Relocation Read Only)
  - Disable
    - .got / .got.plt 可寫
  - Partial
    - .got.plt 可寫
  - Full
    - .got / .got.plt 不可寫
    - 在 load time 時就把所有 library function 解析完

Protection

- RELRO (Relocation Read Only)
  - Default 是 Partial
  - 開啟 Full / 關閉
    - gcc -Wl,-z,relro,-z,now / gcc -Wl,-z,norelro
- 檢查 dynamic tag
  - DT_BIND_NOW - Full
  - GNU_RELRO - Partial

Protection

- PIE (Position Independent Executable)
  - 預設不會開啟 (GCC 6 以前)
  - 開啟的話，text & data 段也會是受 ASLR 影響，造成撰寫 exploit 時更困難

Protection

- PIE (Position Independent Executable)
  - 開啟 / 關閉
    - gcc -fPIC -pie / gcc -no-pie
  - 檢查 ELF header
    - DYN (shared object file)

```
readelf -h ${binary} | grep Type
```

Protection

- 檢查保護機制
  - 多種檢查一次滿足
    - checksec in pwntools

```
» checksec bof2
[*] '/home/naetw/Desktop/demo/bof2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Exploit

- Shellcode
- Return to text
- Return to libc
- Bypass stack guard

Exploit

- 有了 DEP，shellcode 沒辦法執行，但是沒關係...

Exploit

- Return to text
  - 利用程式本身的 code (一定可執行)
  - 沒有 PIE 的保護下，code 的位置是固定的

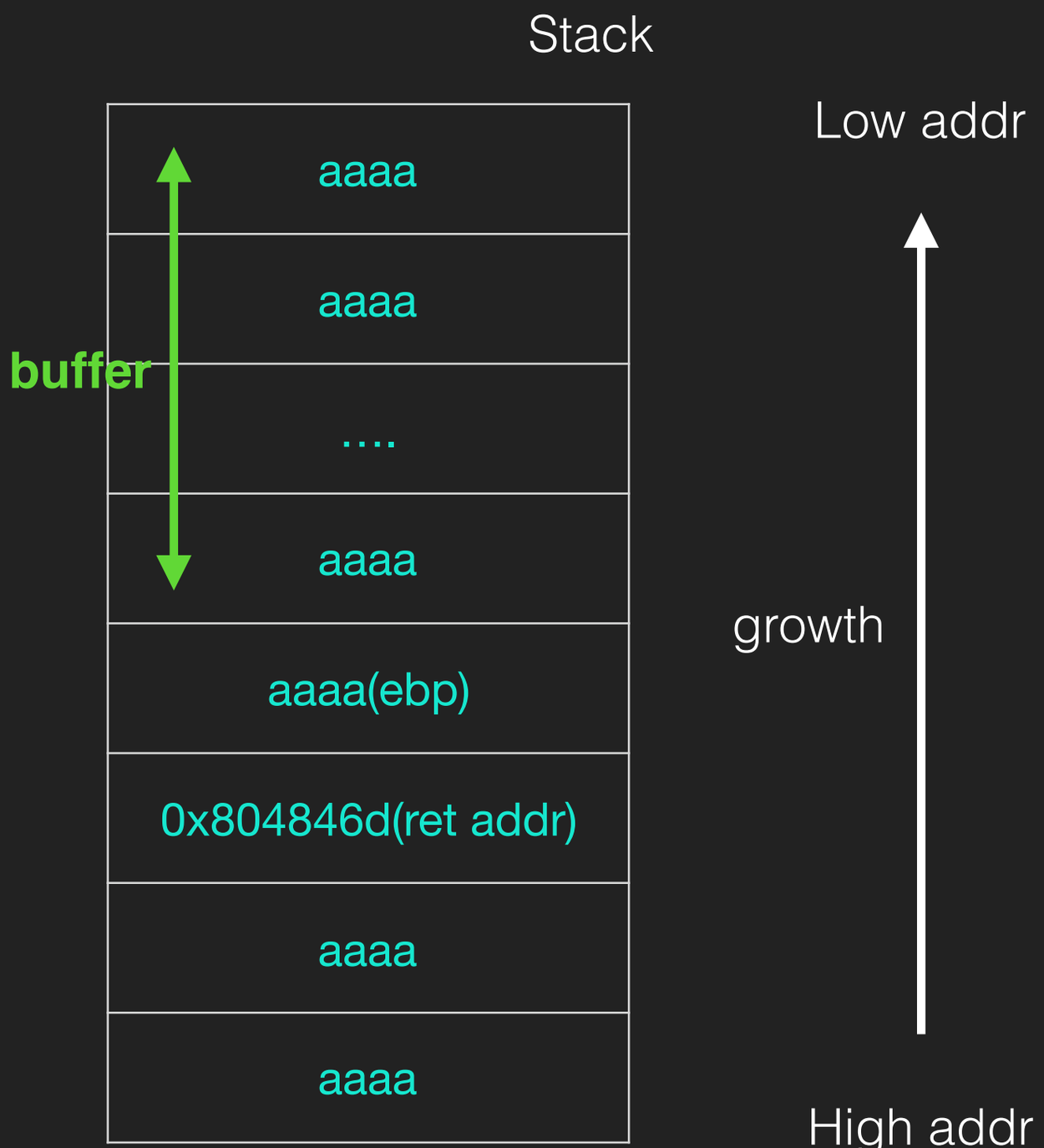
```
08048481 <smash>:
 8048481:       55                          push   ebp
 8048482:       89 e5                       mov    ebp,esp
 8048484:       83 ec 2c                    sub    esp,0x2c
 8048487:       8d 45 d8                    lea    eax,[ebp-0x28]
 804848a:       89 04 24                    mov    DWORD PTR [esp],eax
 804848d:       e8 8e fe ff ff              call   8048320 <gets@plt>
 8048492:       c9                          leave
 8048493:       c3                          ret
```

Exploit

- Return to text
  - Demo

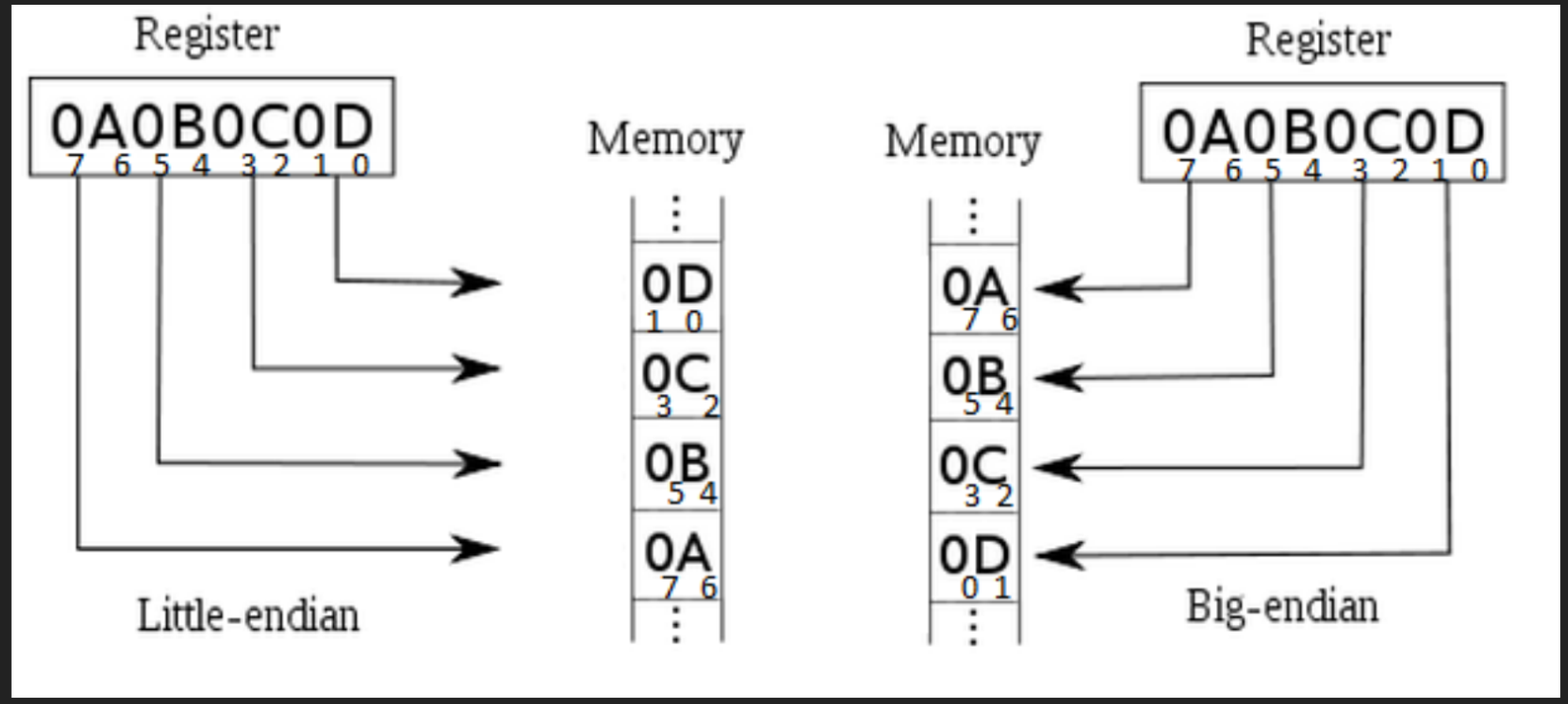
```c
#include <stdio.h>
#include <stdlib.h>

void neveruse() {
    system("/bin/sh");
}

void smash() {
    char buf[40];
    gets(buf);
}

int main() {
    puts("Let's start smashing the stack!!");
    smash();
}
```

Exploit


Stack
aaaa
aaaa
buffer
....
aaaa
aaaa(ebp)
0x804846d(ret addr)
aaaa
aaaa
Low addr
growth
High addr

Exploit

- Return to text
  - Endianness (位元組順序)



https://upload.wikimedia.org/wikipedia/en/7/77/Big-little_endian.png

Exploit

- Return to text
  - Endianness (位元組順序)
  - 在 x86 的架構下，是 little endian

```
(gdb) ni
ABCD
0x08048492 in smash ()
(gdb) x/4wx $eax
0xffffceb4:     0x44434241      0x00000000      0x000007d4      0xfbad2a84
```

High ← Low

Low → High

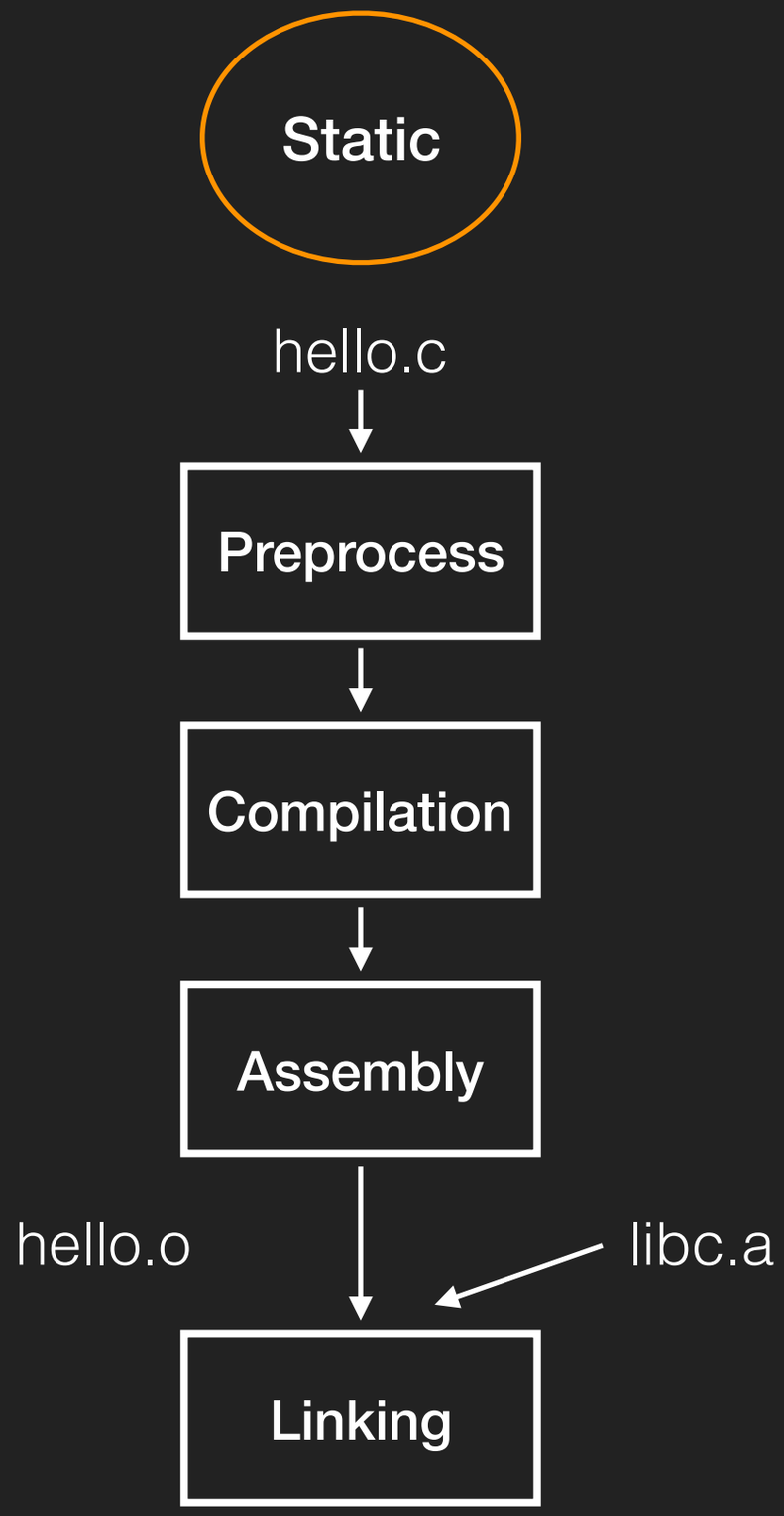
Exploit

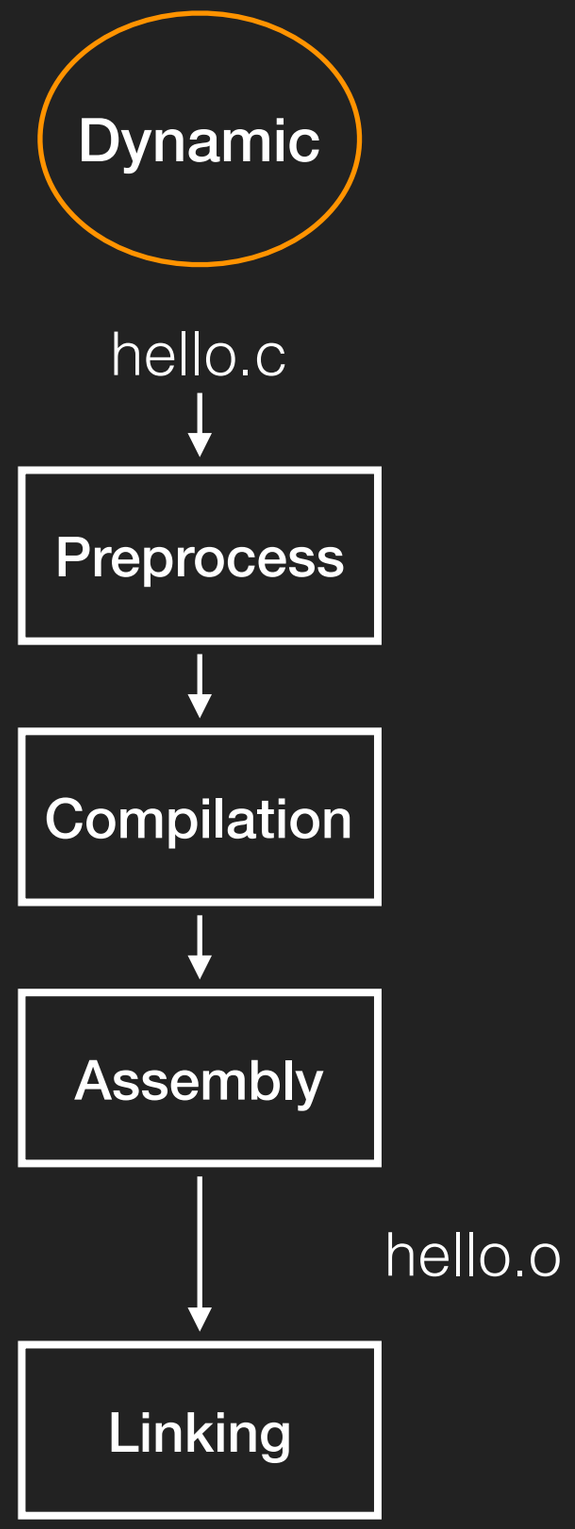
- Return to text
    - 在大多數情況下 binary 內有的 code 無法完成整個 exploit
- Return to libc + ROP

Exploit

- Shellcode
- Return to text
- Return to libc
  - Linking
  - Lazy Binding
- Bypass stack guard

Linking


Static
hello.c
Preprocess
Compilation
Assembly
hello.o
libc.a
Linking


Dynamic
hello.c
Preprocess
Compilation
Assembly
hello.o
Linking

Linking

- Linking
  - 靜態連結 Static Linking
    - 一開始就會直接把所有 library function 的 machine code 放在 ELF 中
  - 動態連結 Dynamic Linking
    - 把 library function 標記為一個動態連結的符號
    - shared object 跟著執行檔一起載入記憶體

Linking

- Dynamic Linking

```
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/naetw/Desktop/demo/lazy
0x00600000         0x00601000         r--p      /home/naetw/Desktop/demo/lazy
0x00601000         0x00602000         rw-p      /home/naetw/Desktop/demo/lazy
0x00007ffff7a10000 0x00007ffff7bce000 r-xp      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7bce000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfc000 r-xp      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7fe3000 0x00007ffff7fe5000 rw-p      mapped
0x00007ffff7ff5000 0x00007ffff7ff8000 rw-p      mapped
0x00007ffff7ff8000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

Lazy Binding

- 在一支程式裡，某些 library function 可能結束前都不會呼叫到

```c
#include <stdio.h>
#include <stdlib.h>

void edgeman() {
    char chr;
    printf("Could you be my friend?");
    scanf("%c%*c", &chr);
    printf("Thx!");
}

int main() {
    printf("Hello World!");
    return 0;
}
```

here

Lazy Binding

- 因此，dynamic linking 的 ELF 使用了 lazy binding 這個機制，第一次呼叫 library function 的時候才會去解析出真正的位址

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

0x400450: jmp Foo@got

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

<Foo@got>: 0x400456

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

0x400450: jmp Foo@got

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

<Foo@got>: 0x400456

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

0x400450: jmp Foo@got

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

<Foo@got>: 0x400456

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

0x400450: jmp Foo@got

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

<Foo@got>: 0x400456

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

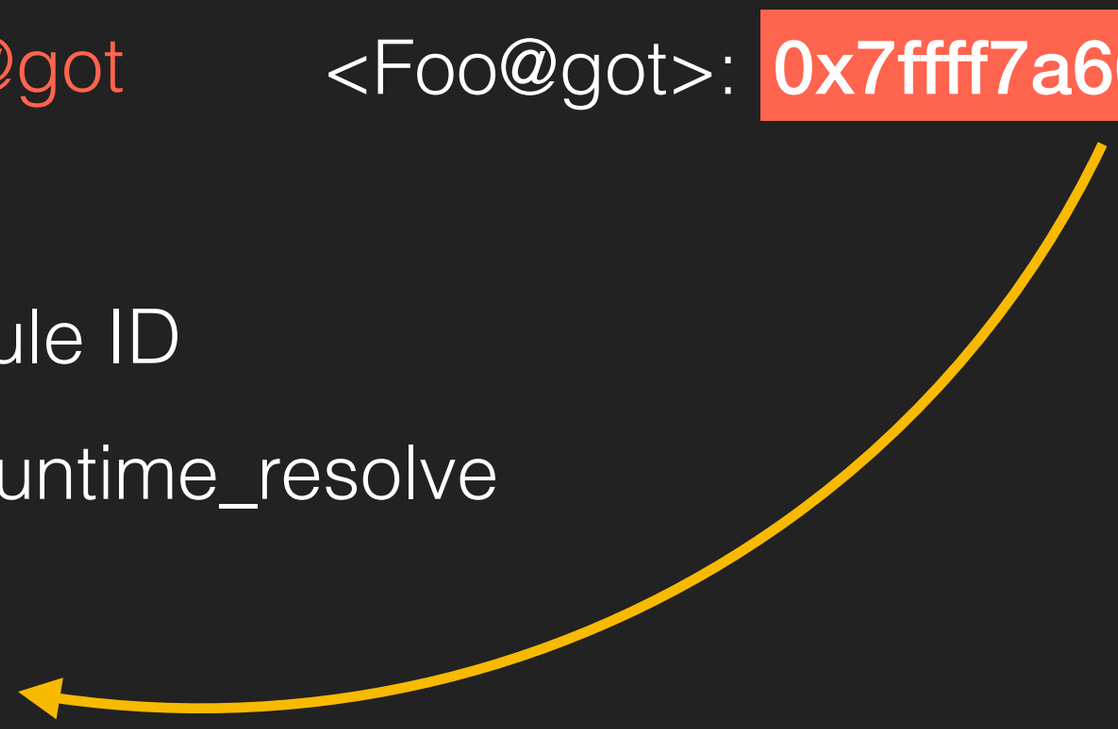
0x400450: jmp Foo@got <Foo@got>: **0x7ffff7a66666**

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

0x7ffff7a66666 <Foo>: …



Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作

<Foo@plt>:

0x400450: jmp Foo@got <Foo@got>: **0x7ffff7a66666**

0x400456: push n

0x40045b: push module ID

0x400461: jmp _dl_runtime_resolve

0x7ffff7a66666 <Foo>: …

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作
  - n
    - Foo 在 .rel.plt 的索引
  - module ID
    - 引用的 library name

```
<Foo@plt>:
0x400450: jmp       Foo@got
0x400456: push      n
0x40045b: push      module ID
0x400461: jmp       _dl_runtime_resolve
```

Lazy Binding

- ELF 使用 PLT (Procedure Linkage Table) 的方式來實作
  - Reuse code

```
<PLT0>:
0x400440: push      module ID
0x400446: jmp       _dl_runtime_resolve

<Foo@plt>:
0x400450: jmp       Foo@got
0x400456: push      n
0x40045b: jmp       PLT0
```

Lazy Binding

- 詳情請洽 Angelboy - Execution p.17

Exploit

- Return to libc
  - 除了 binary 本身的 code，shared library 的 code 也可以利用
    - system
    - execve
    - …

Exploit

- Return to libc
  - 因為 ASLR 的緣故，libc 每次載入位址不固定，需要搭配 information leak
  - 拿到某個在 libc 裡的位址便可以算出此次的 base address
  - 之後就可以加上想要的 function offset 隨意使用任一 library function
    - 在同一個 libc 裡，function offset 是固定的

Exploit

- Return to libc
  - 尋找 libc address
    - GOT
    - stack
      - e.g., return address of main
    - heap
      - e.g., main_arena

Exploit

- Return to libc
  - 尋找 library function offset

```
readelf -s /path/of/libc | grep ${function_name}
```

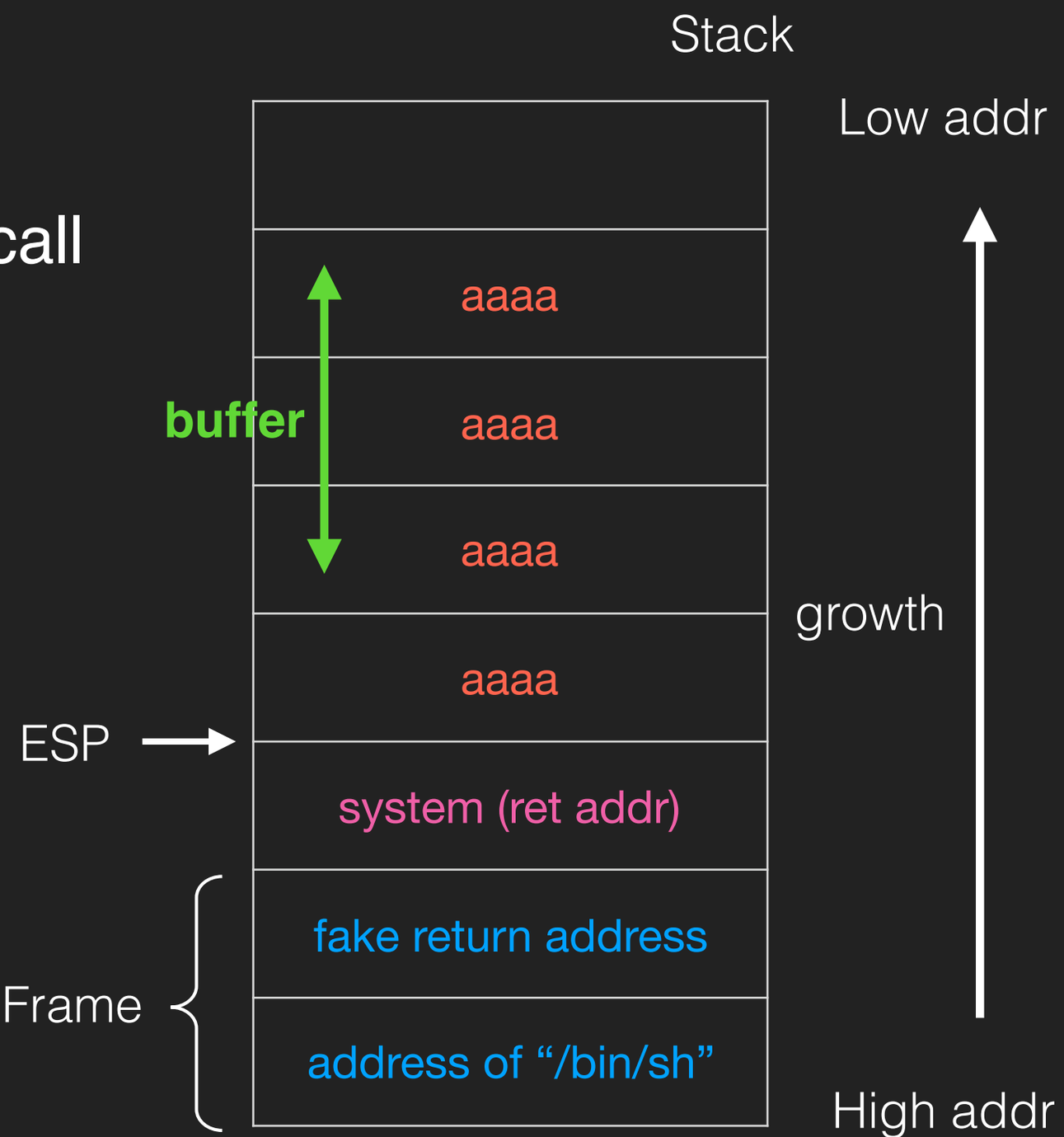
```
» readelf -s libc_64.so.6 | grep system
   225: 0000000000137c20    70 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.2.5
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1351: 0000000000045390    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
```

Exploit

- Return to libc
  - 獲得 libc base 之後
    - 在 stack 上偽造 function call
- GOT hijack
- …

Exploit

- Return to libc
  - 在 stack 上偽造 function call
    - 限制
      - overflow
      - 任意寫



Exploit

- Return to libc
  - GOT hijack
    - 限制
    - 任意寫
  - gets(buf) -> system(buf)

<gets@plt>:

0x400450: jmp gets@got

0x400456: push n

0x40045b: jmp PLT0

<gets@got>: 0x7ffff7a6dead

0x7ffff7a6dead <system>: …

Exploit

- Shellcode
- Return to text
- Return to libc
  - Linking
  - Lazy Binding
- Bypass stack guard

Exploit

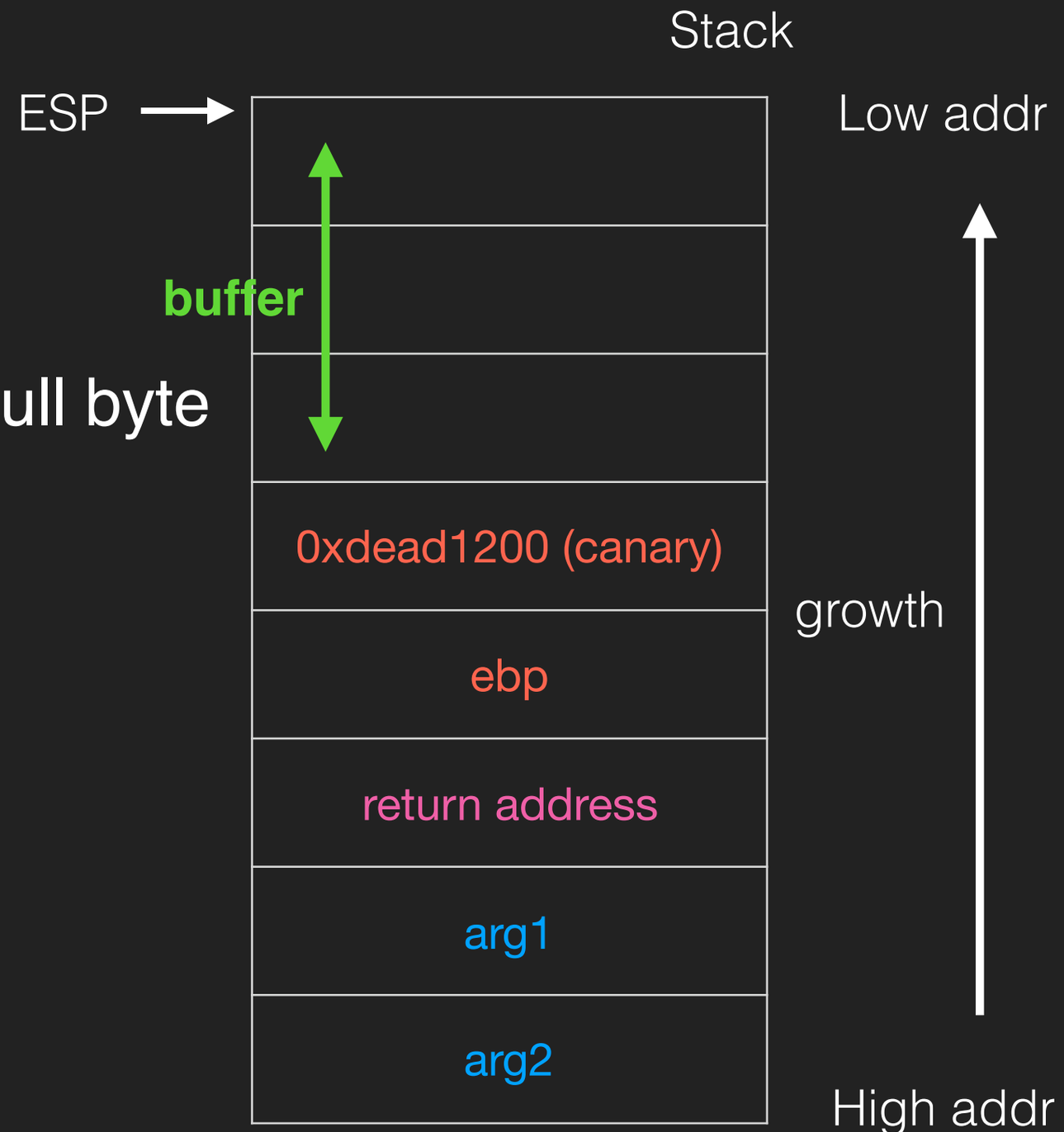
- Bypass stack guard
  - leak canary
  - GOT hijack
- ...

Exploit

- Bypass stack guard
  - leak canary
    - 某些 function 在讀完輸入之後不會放上 null byte，可以利用這點去做 information leak (canary, libc, stack)
      - read()
      - strcpy()
      - strncpy()

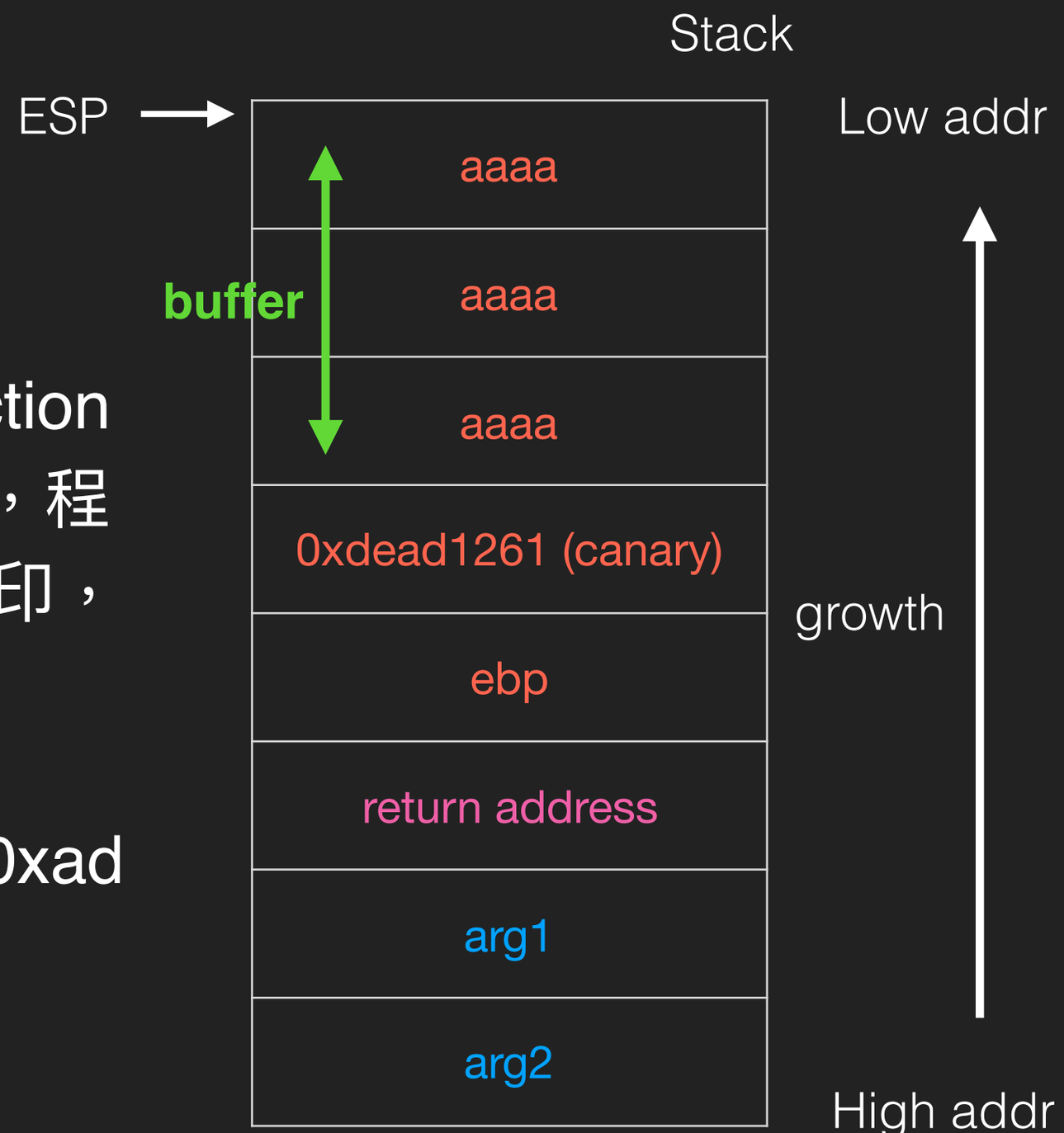
Exploit

- Bypass stack guard
  - leak canary
    - canary 的尾端一定是 null byte



Exploit

- Bypass stack guard
  - leak canary
    - 這時如果有個輸出 function (e.g., printf(“%s”, buf))，程式便會從 buf 開頭一路印，直到遇到 null byte
  - a*12 + 0x61 + 0x12 + 0xad + 0xde + …



Exploit

- Bypass stack guard
  - GOT hijack
    - 限制
      - 任意寫
      - Partial / No RELRO

Exploit

- Bypass stack guard
  - GOT hijack (cont.)
    - 將 __stack_chk_fail 的 GOT entry 內容改掉，這時可以根據修改的內容決定不同的利用方式
- NOP
  - 直接 overflow，疊 ROP
- one_gadget
- system
  - 偽造 function call (不需要疊 return address)
- …

Reference

- http://l4ys.tw/ROP_bamboofox.pdf
- AngelBoy
- 程式設計師的自我修養
- https://goo.gl/wKFGfn
- https://en.wikipedia.org/wiki/Executable_space_protection#Windows
- https://en.wikipedia.org/wiki/Address_space_layout_randomization

Contact

- Mail: wangzhelee@gmail.com