

Buffer Overflow

nae @ NCTUCSC & BambooFox

Preface

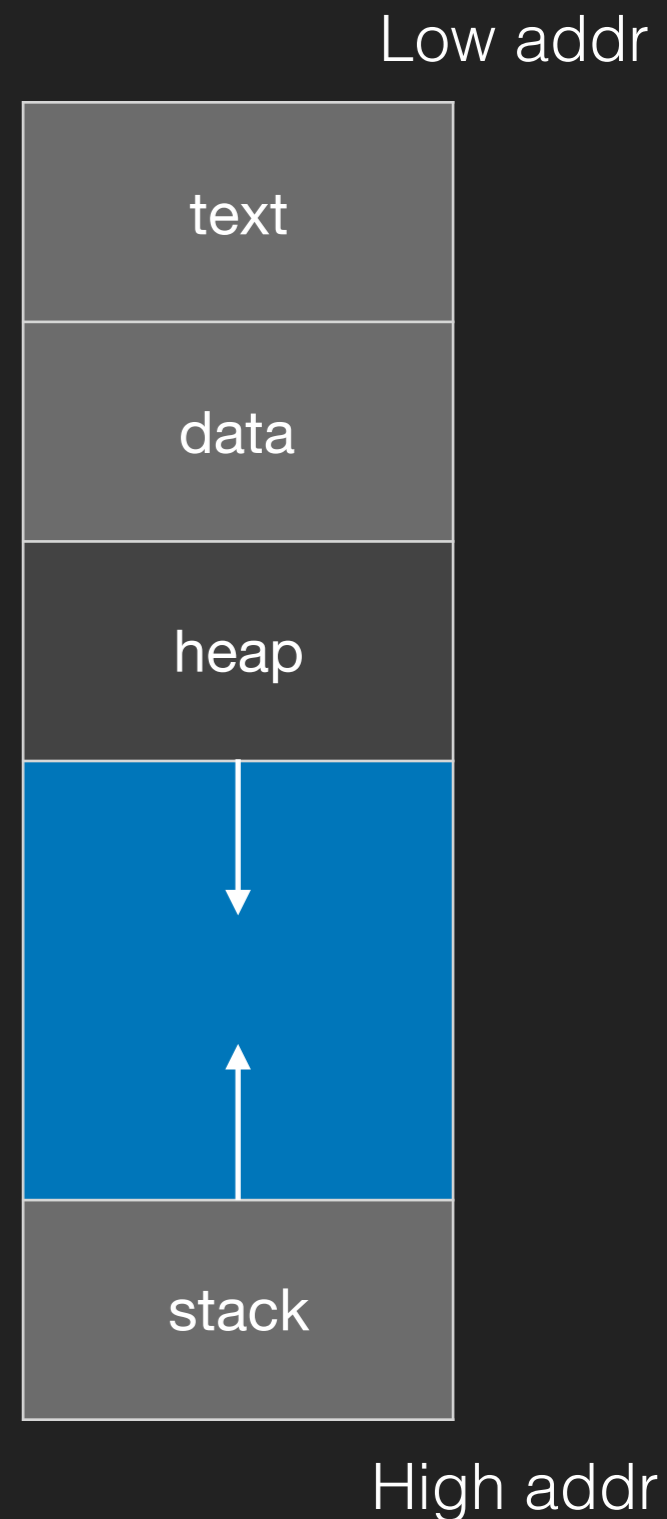
- Environment
 - gcc-4.8
 - Glibc - 2.24
 - Kernel - 4.8.0-59
 - 64-bit OS
- The following content will be based on 32-bit for better explanation and understanding.

Outline

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

What's stack?

- OS kernel maps the program to the memory
- text
 - program code
- data
 - program data
- heap
 - dynamic memory space
- stack
 - local variable, arguments, base pointer, return address



Stack Frame

- Stack: LIFO (Last In First Out)



Function Call & Stack

2. Back to main to
continue execution

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void Func(int a, int b){
5     a = 10;
6     printf("%d\n", a);
7 }
8
9 int main(){
10     int x;
11     Func(1, 2);
12     return 0;
13 }
```

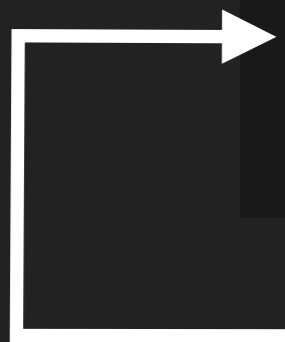
1. Call Function

Function Call & Stack

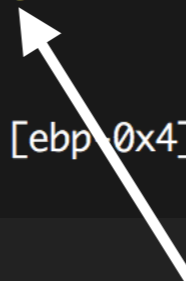
- After executing a function, the program needs to go back to the next instruction which is right after the function call and it will use **stack** to store the address of that instruction.

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

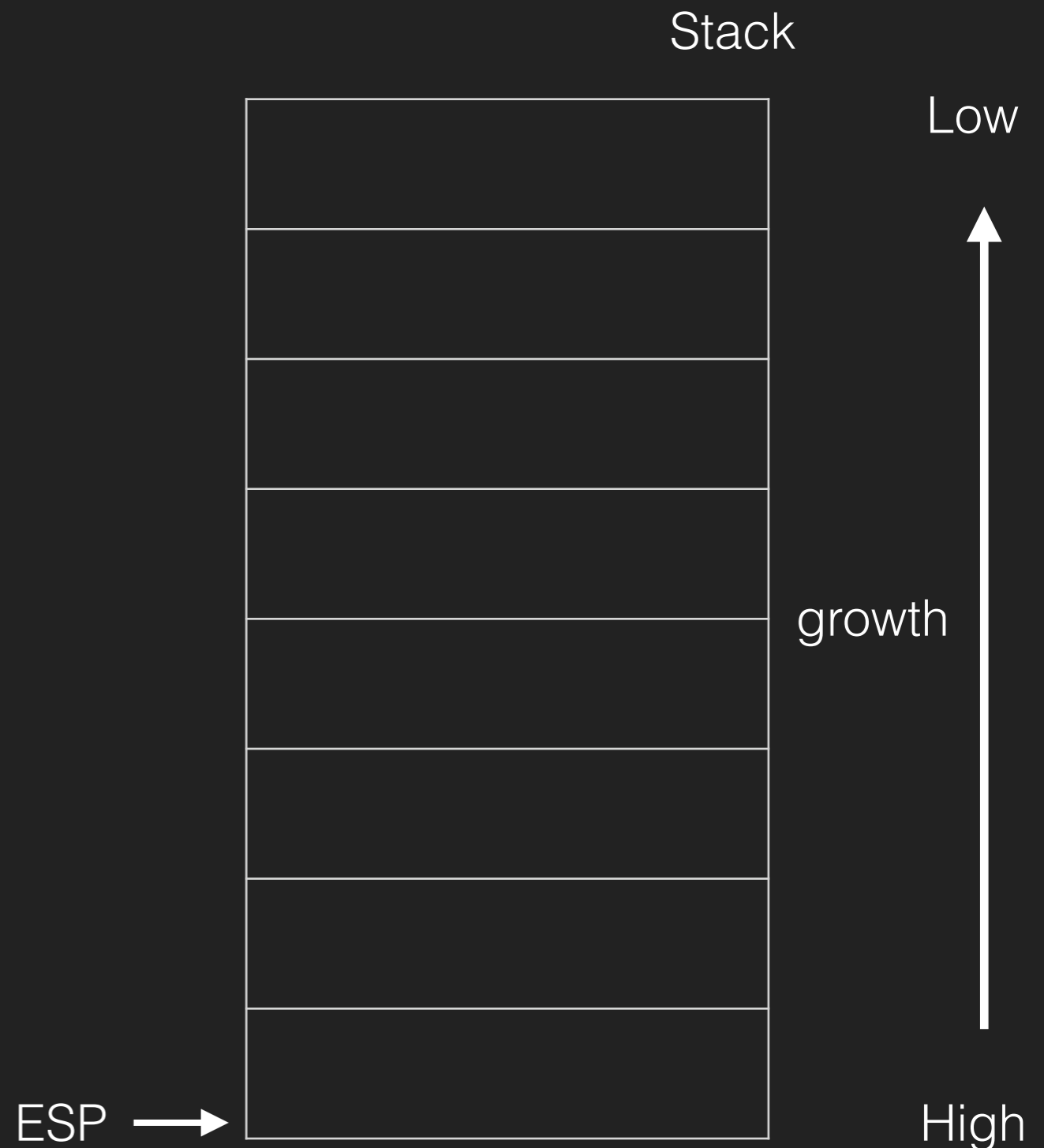
Function Call & Stack

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

push b

push a

call Func



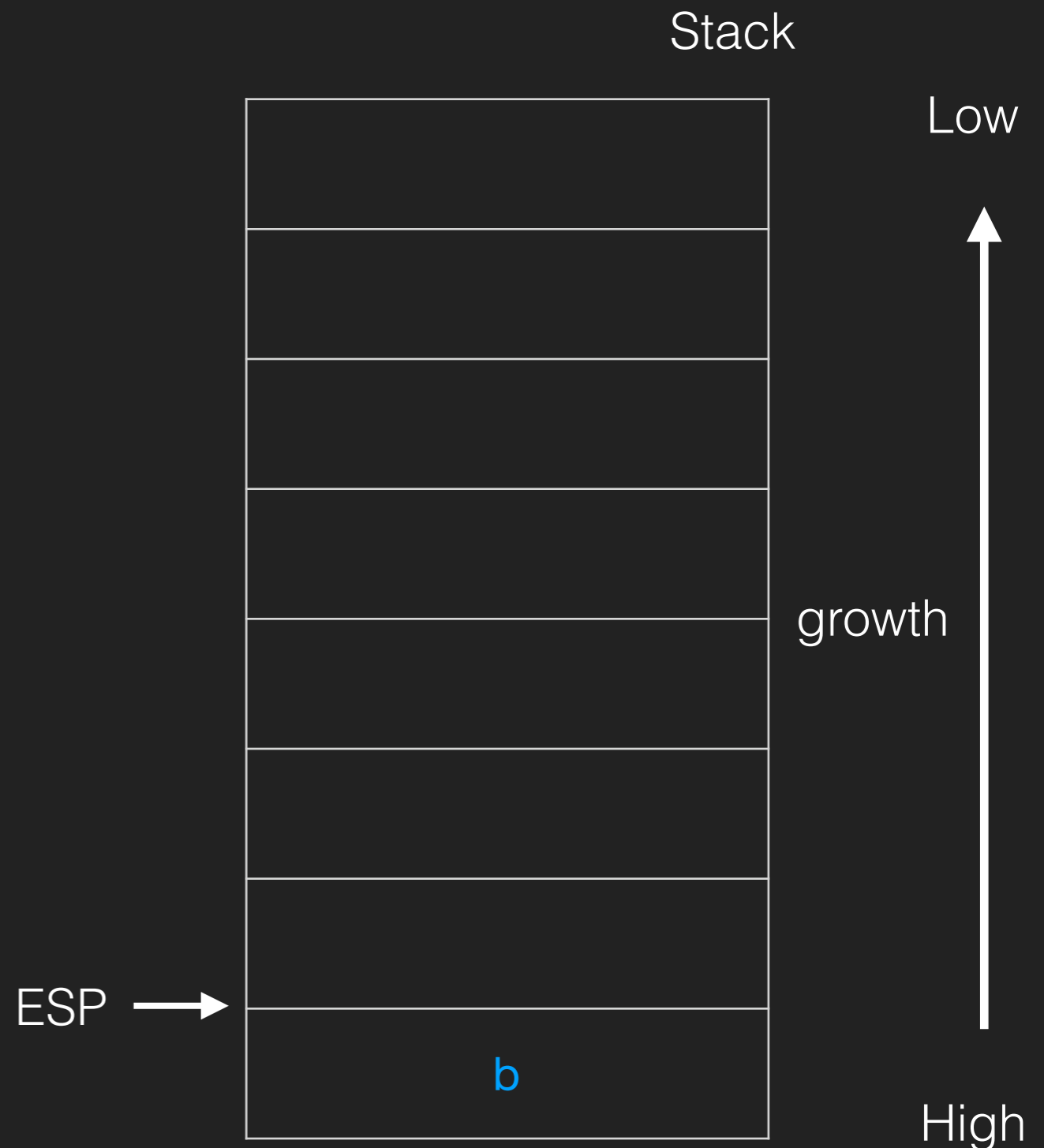
Function Call & Stack

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

push b

push a

call Func



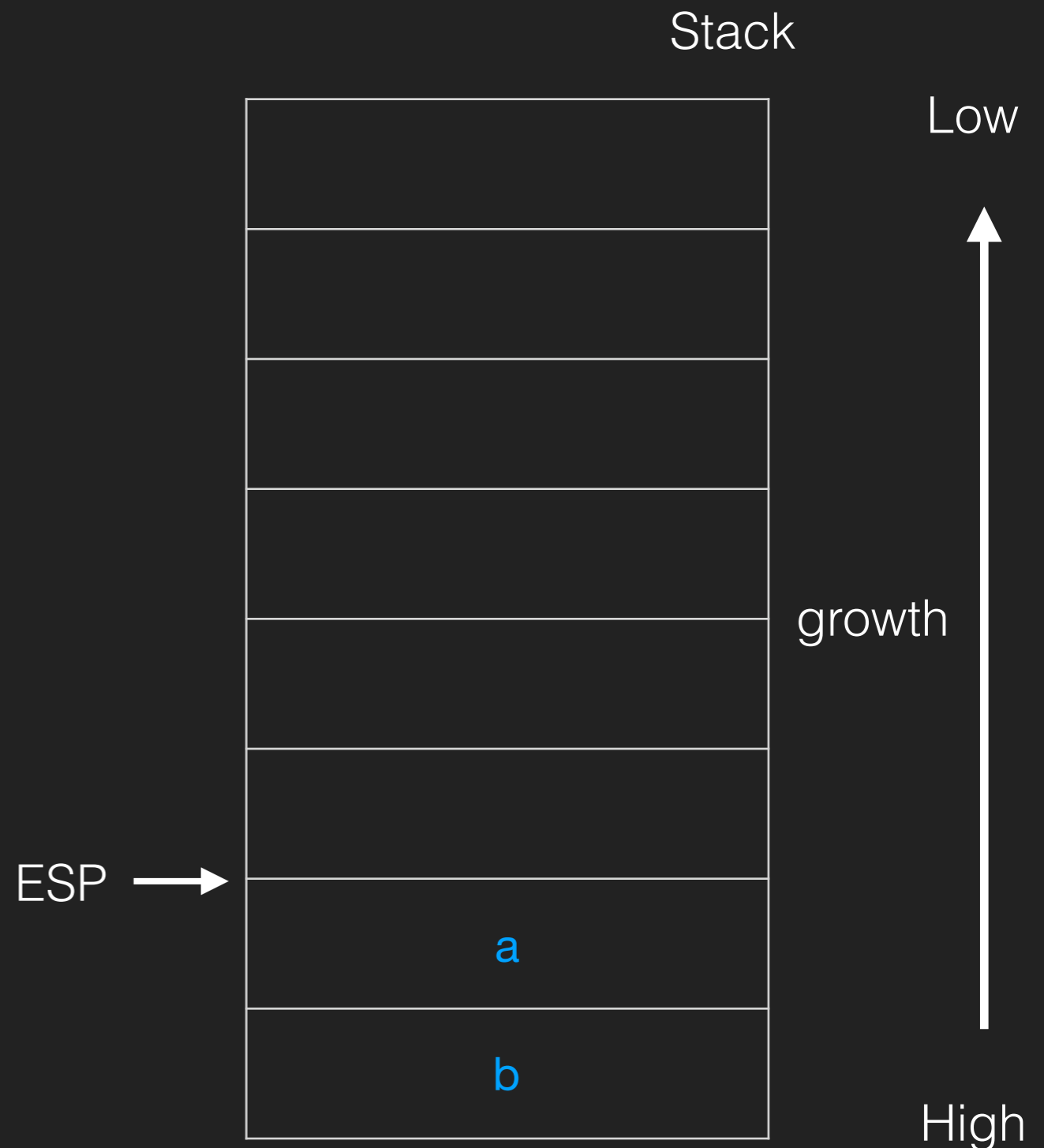
Function Call & Stack

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

push b

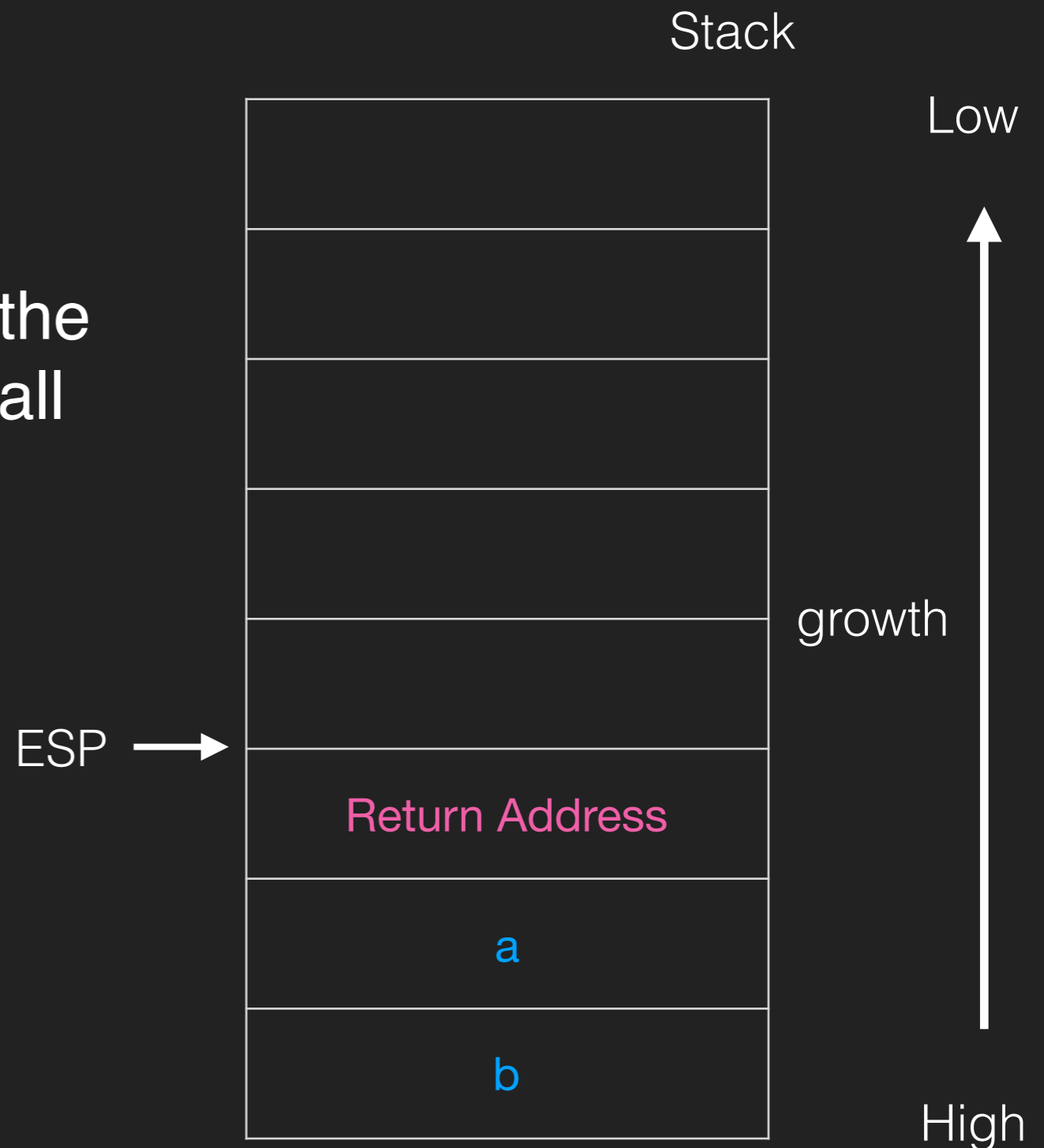
push a

call Func



Function Call & Stack

- **Caller Part**
- Return Address: address of the instruction next to function call
- `void Func(int a, int b)`
- ...
 - push b
 - push a
 - call Func



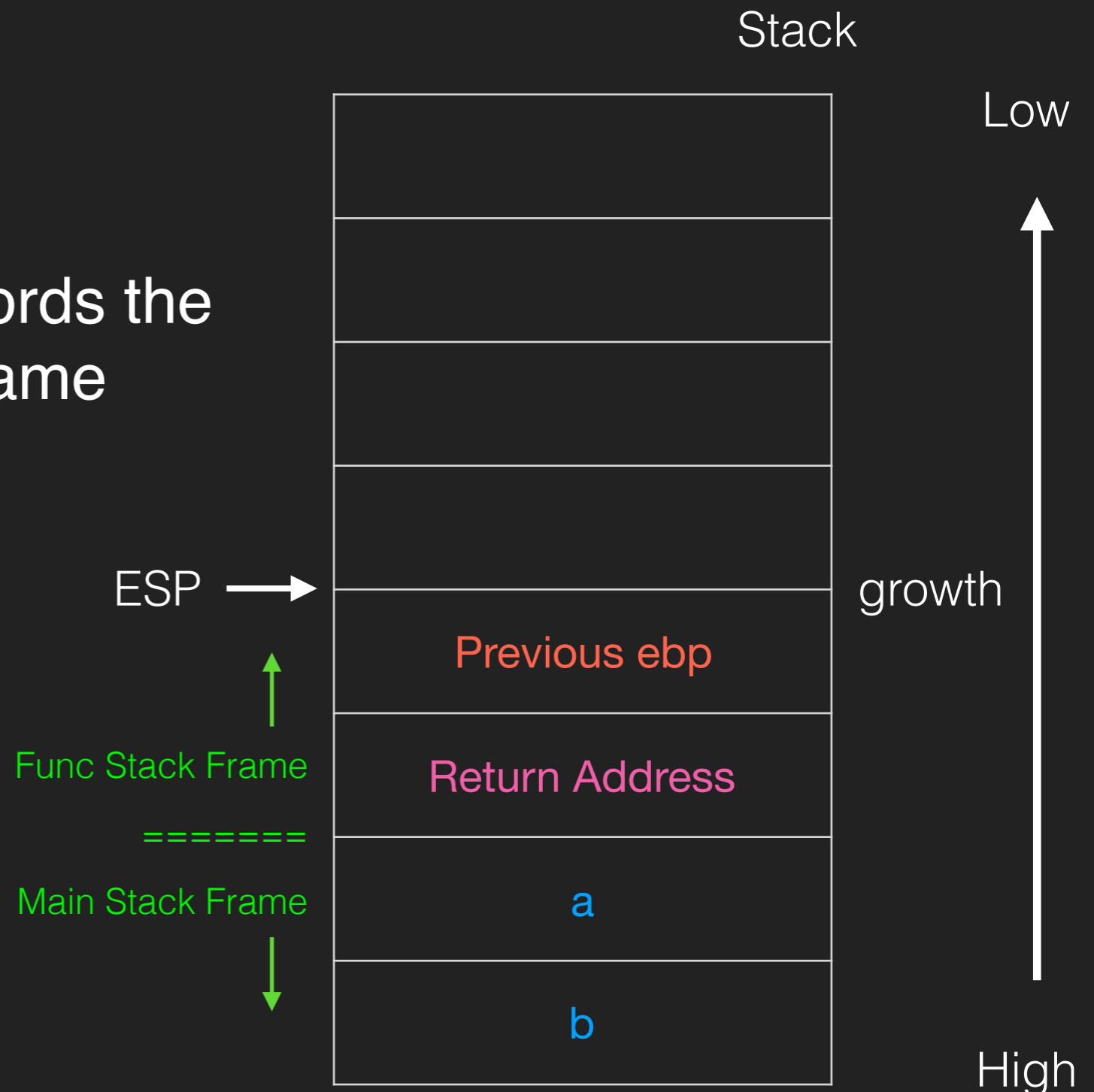
Function Call & Stack

- Callee Part
- At this moment EBP records the address of **main** stack frame
- `void Func(int a, int b)`
`char buf[12];`

`push ebp`

`mov ebp, esp`

`sub esp, 0xc`



Function Call & Stack

- Callee Part
- At this moment EBP records the address of **Func** stack frame
- `void Func(int a, int b)`
`char buf[12];`

`push ebp`

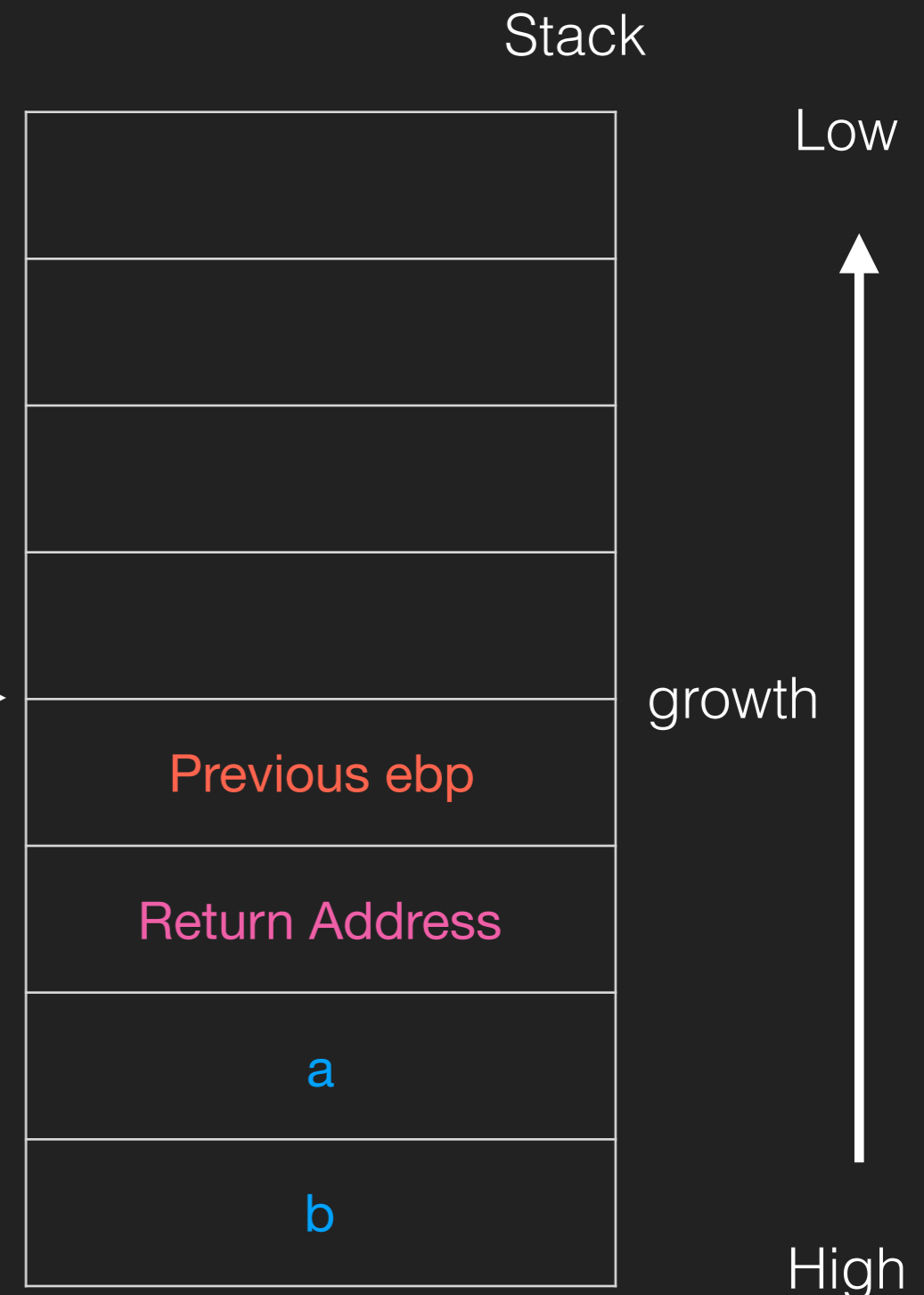
`mov ebp, esp`

`sub esp, 0xc`

EBP = ESP →

↑
Func Stack Frame

=====
Main Stack Frame
↓



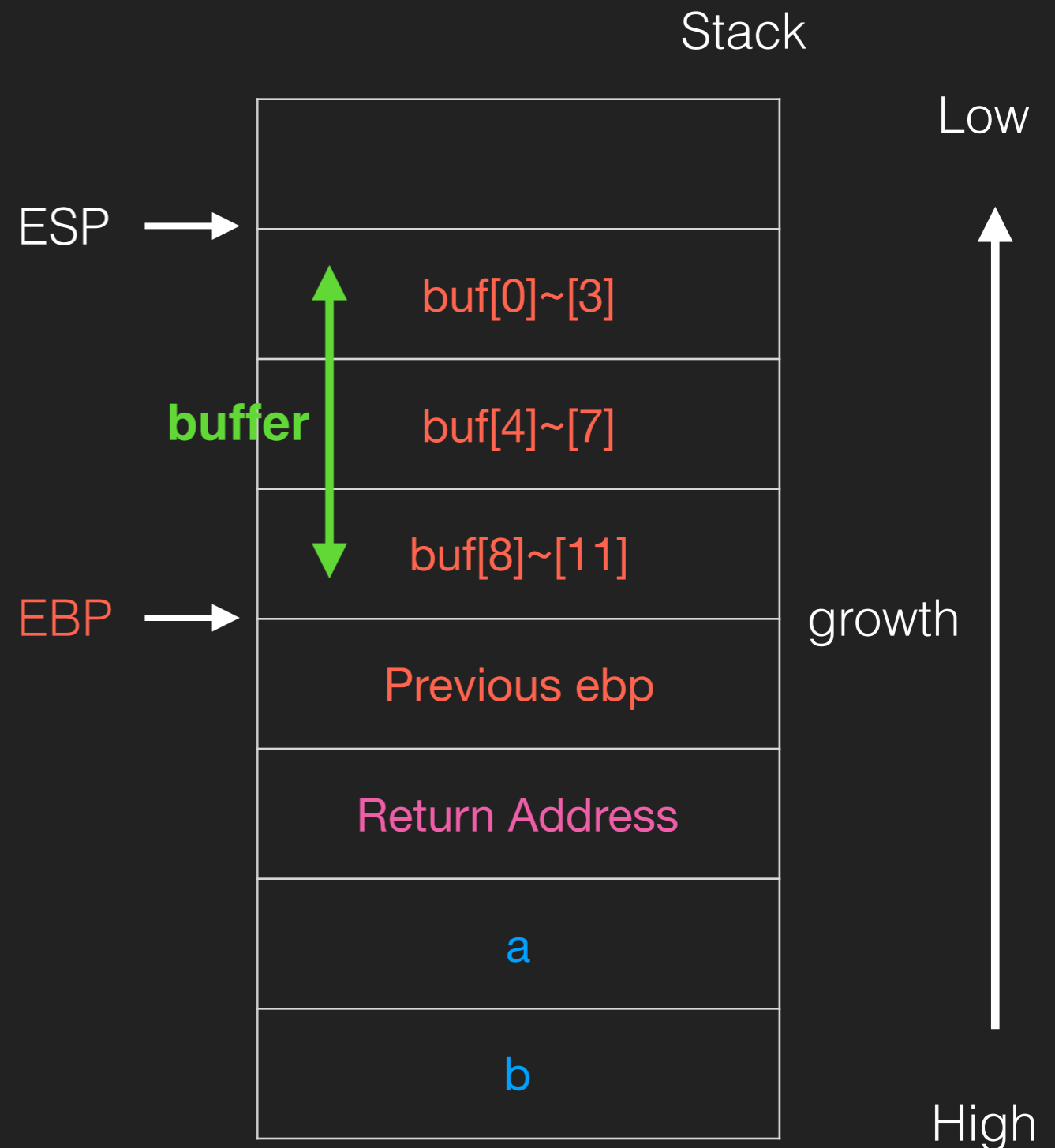
Function Call & Stack

- 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`

...

`EBP - 0xc`

`EBP - 0x8`

`EBP - 0x4`

`EBP`

`EBP + 0x4`

`EBP + 0x8`

`EBP + 0xc`

buffer



Buffer Overflow

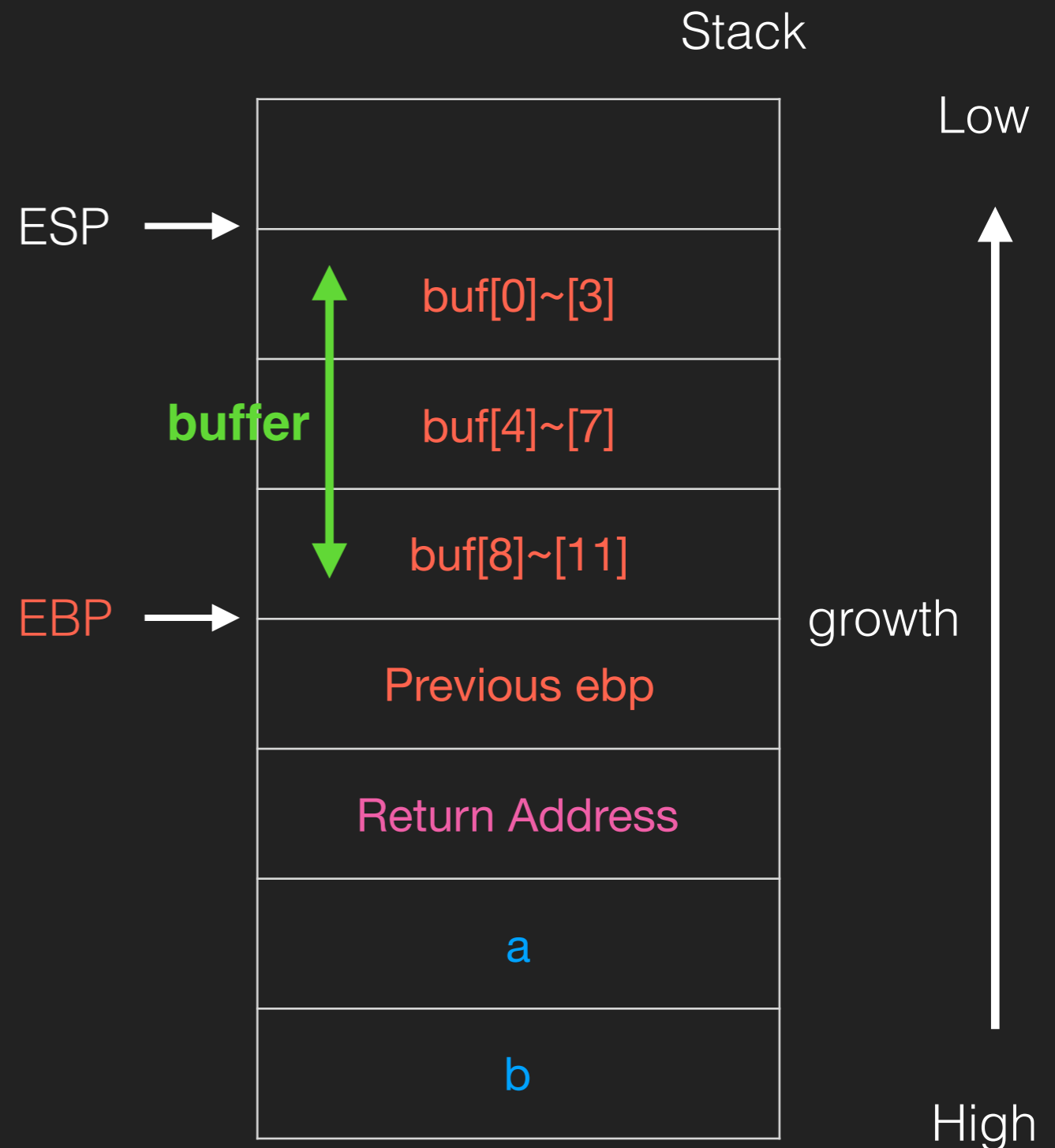
- If the program doesn't check input size, then it would be possible for the user to enter input that is larger than buffer size. Once that happens, the overflow part will override other variables and may effect the behavior of program.(e.g., variable value, control flow)

Buffer Overflow

- Unsafe Function
 - `gets` -> `fgets`
 - `scanf` -> never use `scanf("%s")`
 - `strcpy` -> `strncpy`
 - ...
- Buffer Overflow can be classified according to memory space
 - stack overflow
 - heap overflow

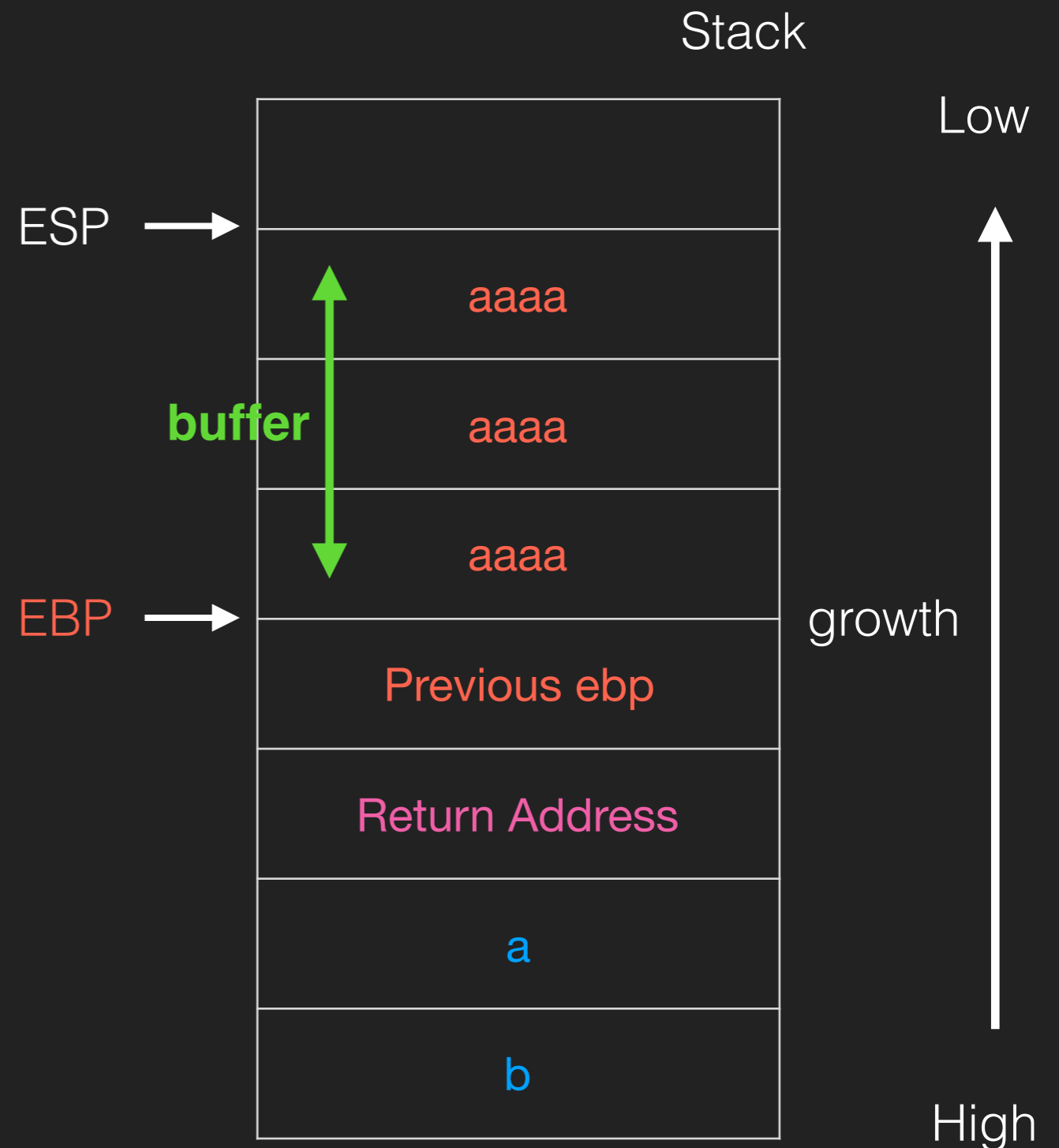
Stack Overflow

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



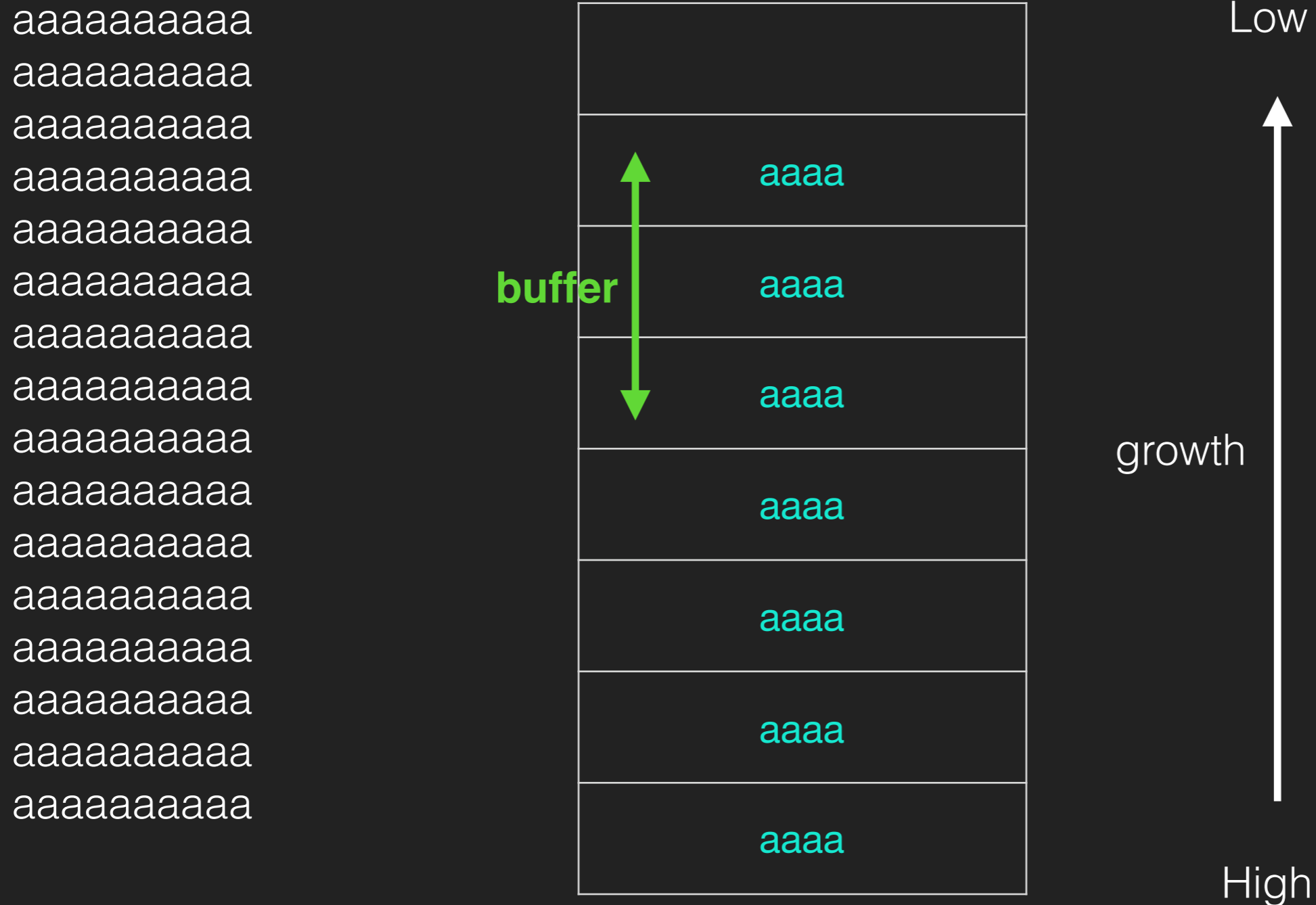
# Stack Overflow

- aaaaaaaaaaaaaa(a \* 12)

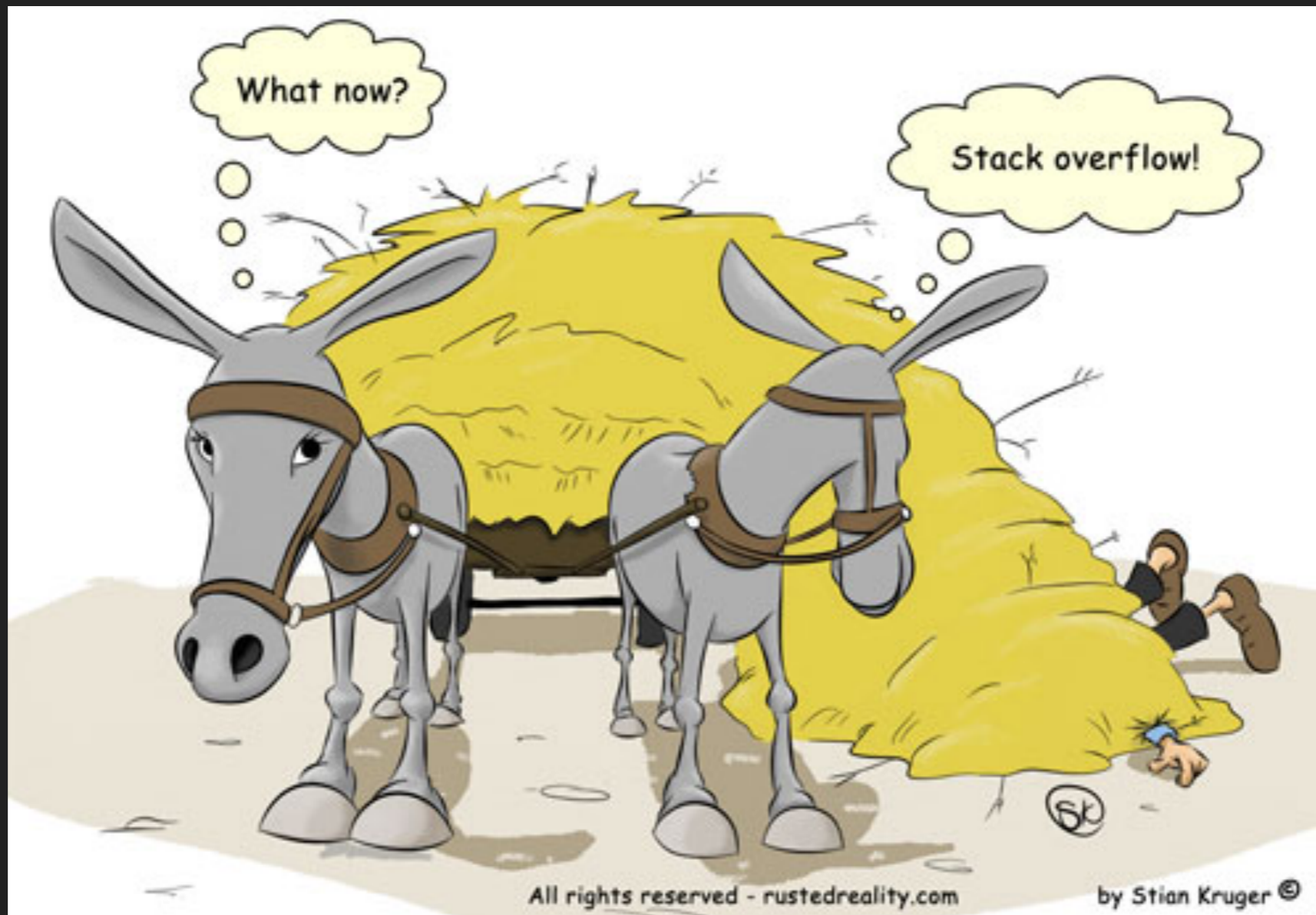


# Stack Overflow

# Stack



# Stack Overflow



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

# Stack Overflow

- aaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaa  
(a\*100)

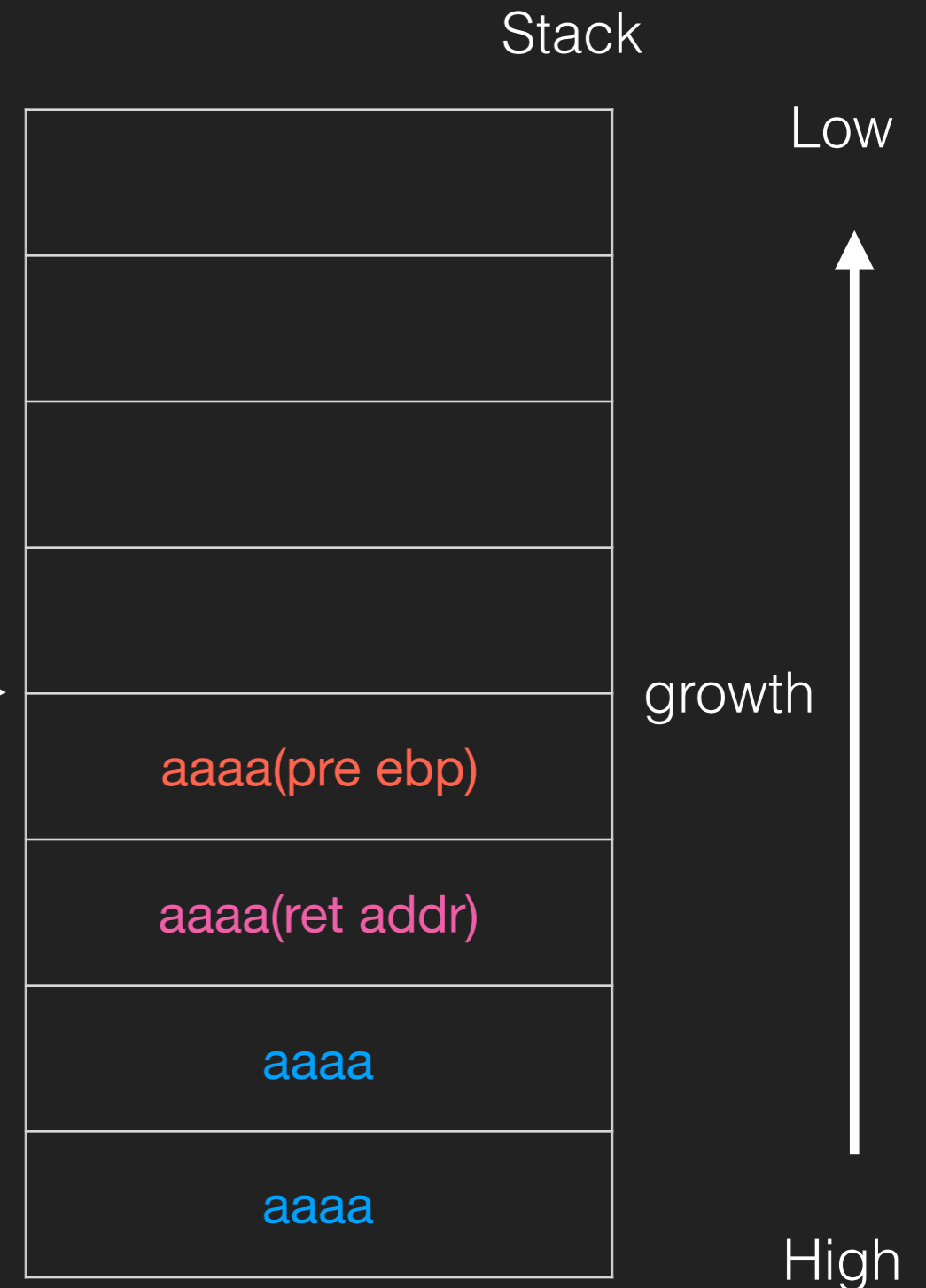
...

mov esp, ebp

pop ebp

ret

ESP = EBP →



# Stack Overflow

- aaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
(a\*100)

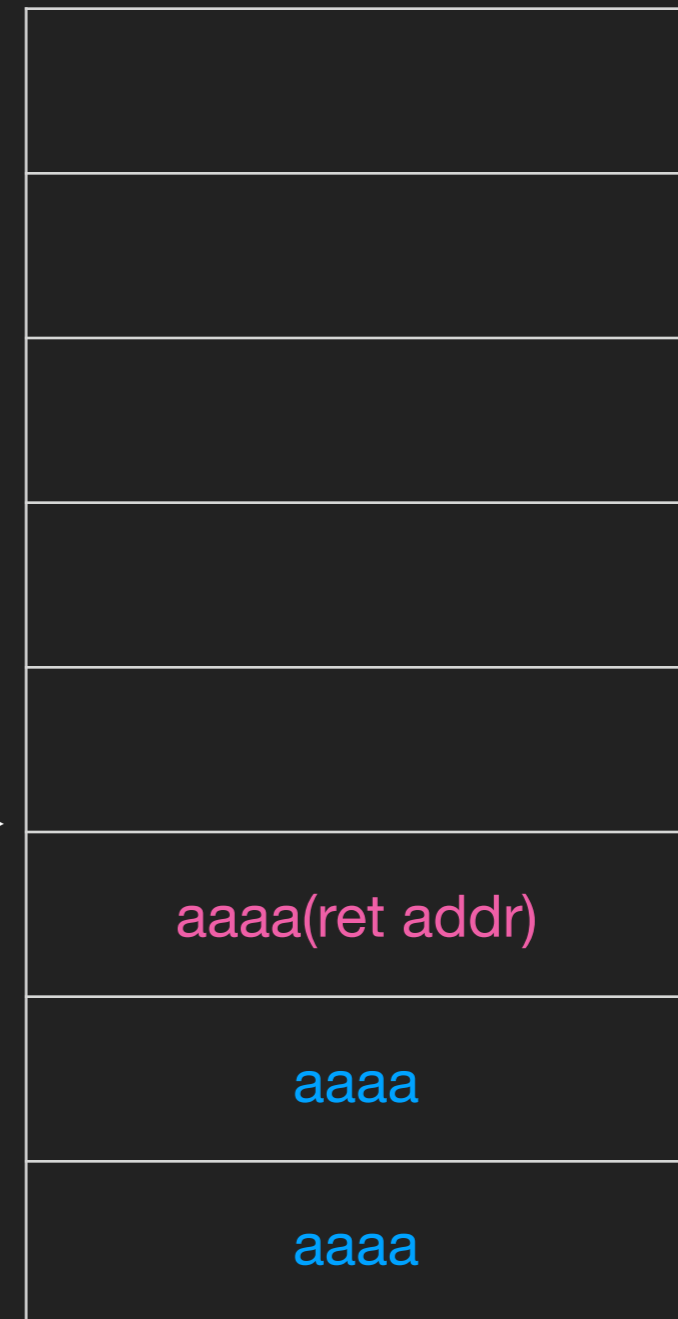
...

mov esp, ebp

pop ebp ; ebp == 0x61616161

ret

ESP →



# Stack Overflow

- aaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaa  
(a\*100)

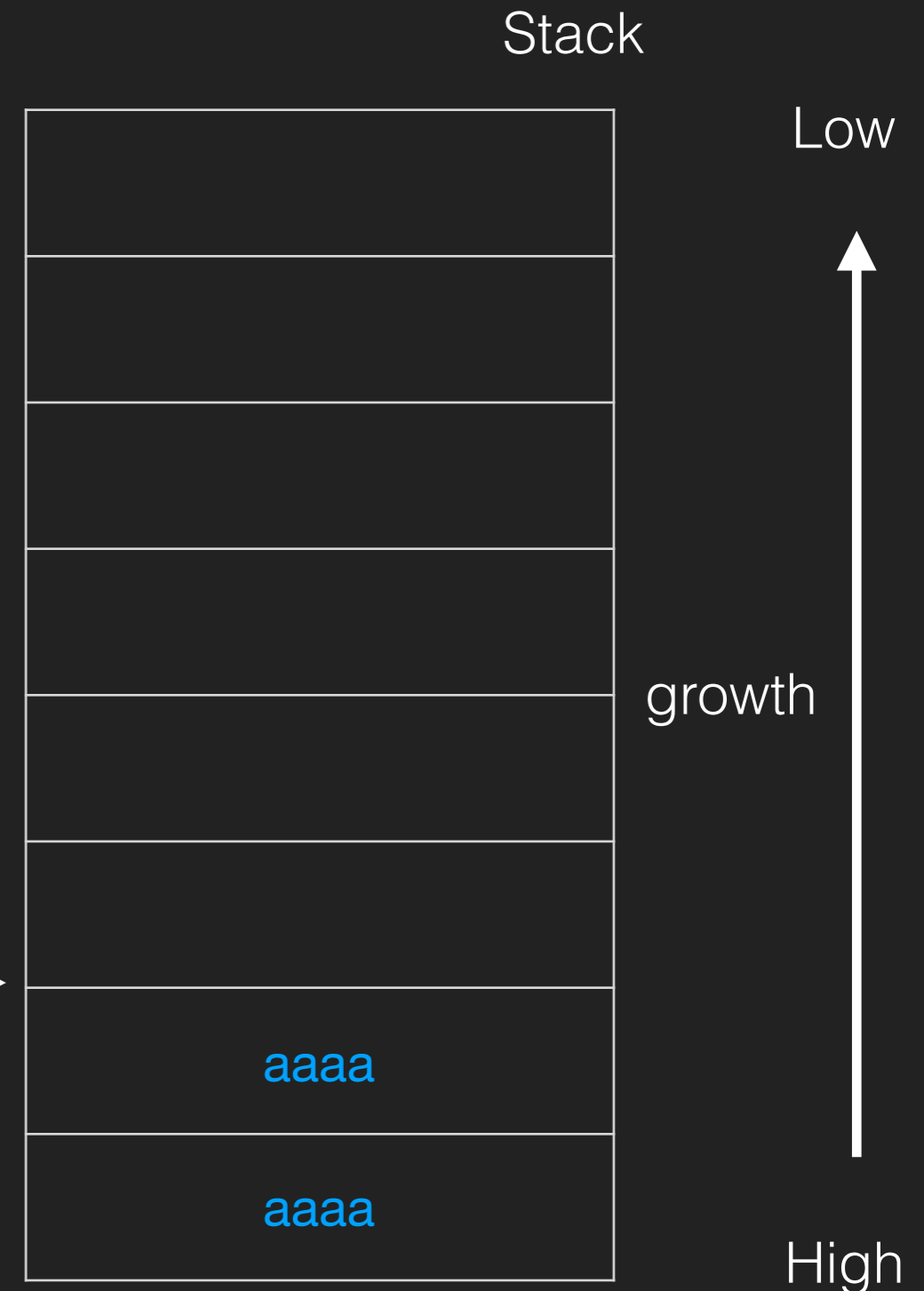
...

```
mov esp, ebp
```

```
pop ebp ; ebp == 0x61616161
```

```
ret (pop eip) ; eip == 0x61616161
```

ESP →



# Stack Overflow

- Program will try to get instruction from 0x61616161 which is an invalid address.

```
(gdb) r
Starting program: /home/naetw/Desktop/demo/bof
aa
Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
```

# Stack Overflow

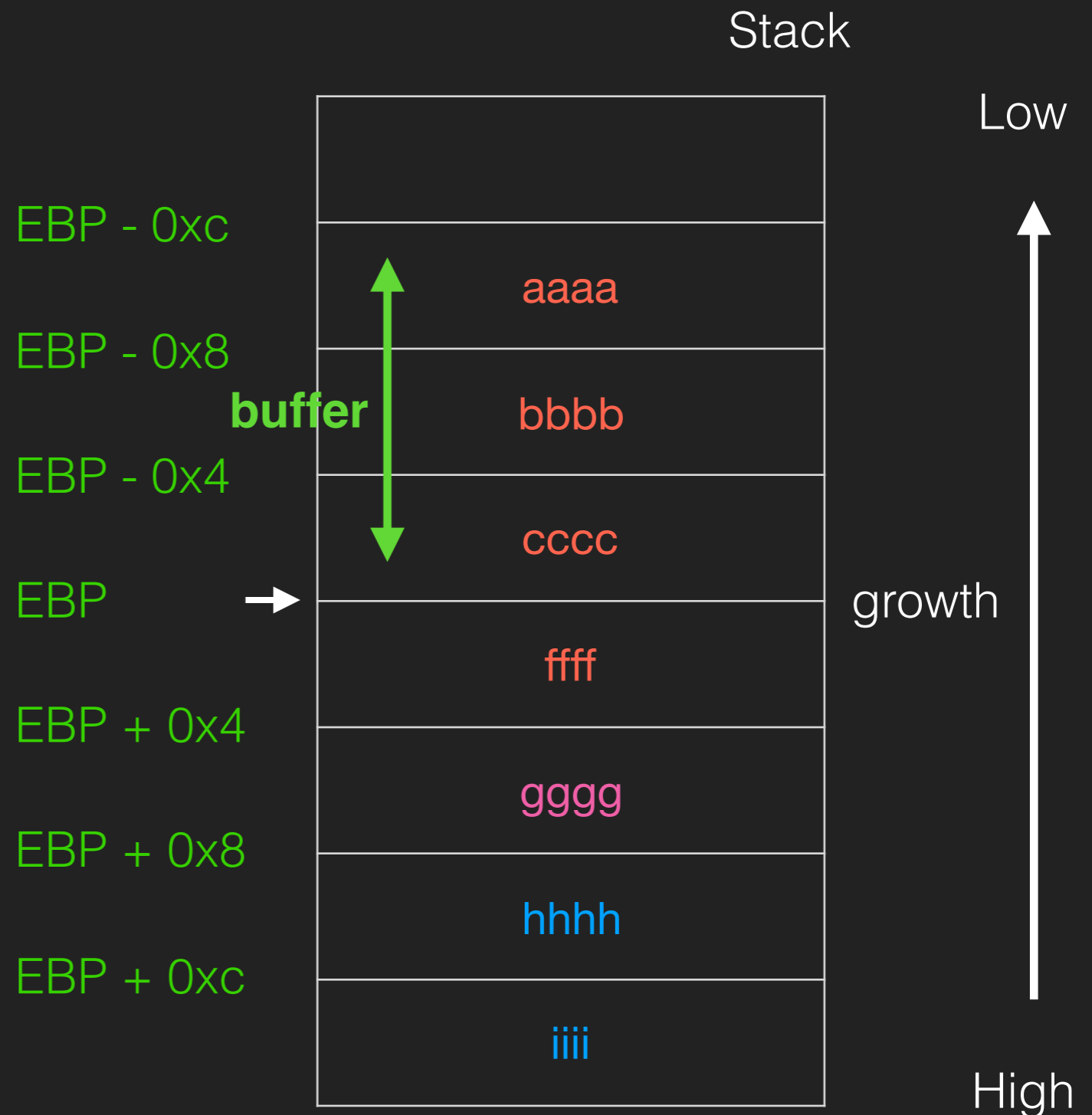
- Calculate the offset

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

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

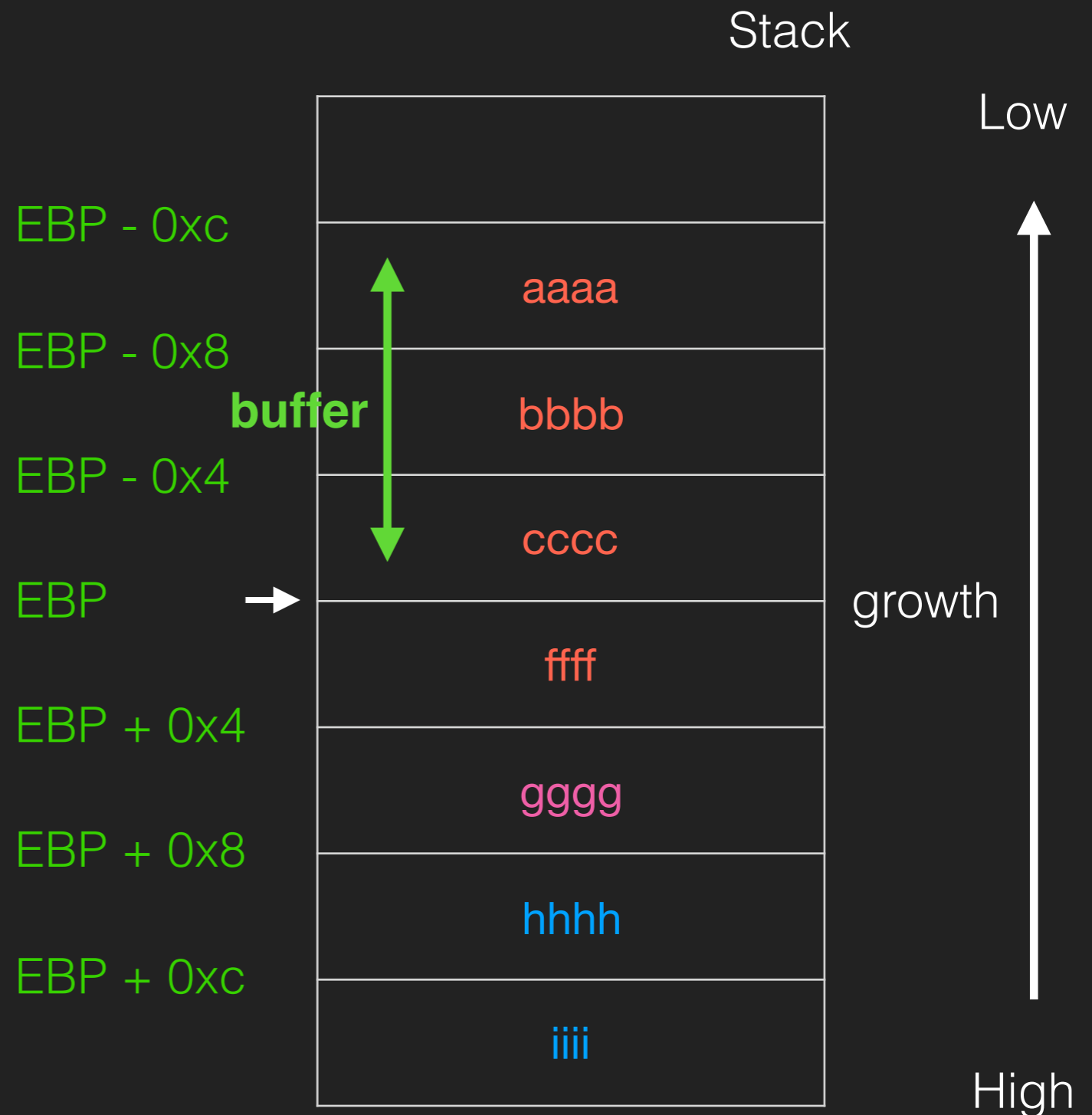
# Stack Overflow

- Calculate the offset
- Where is dddd & eeee
  - stack alignment



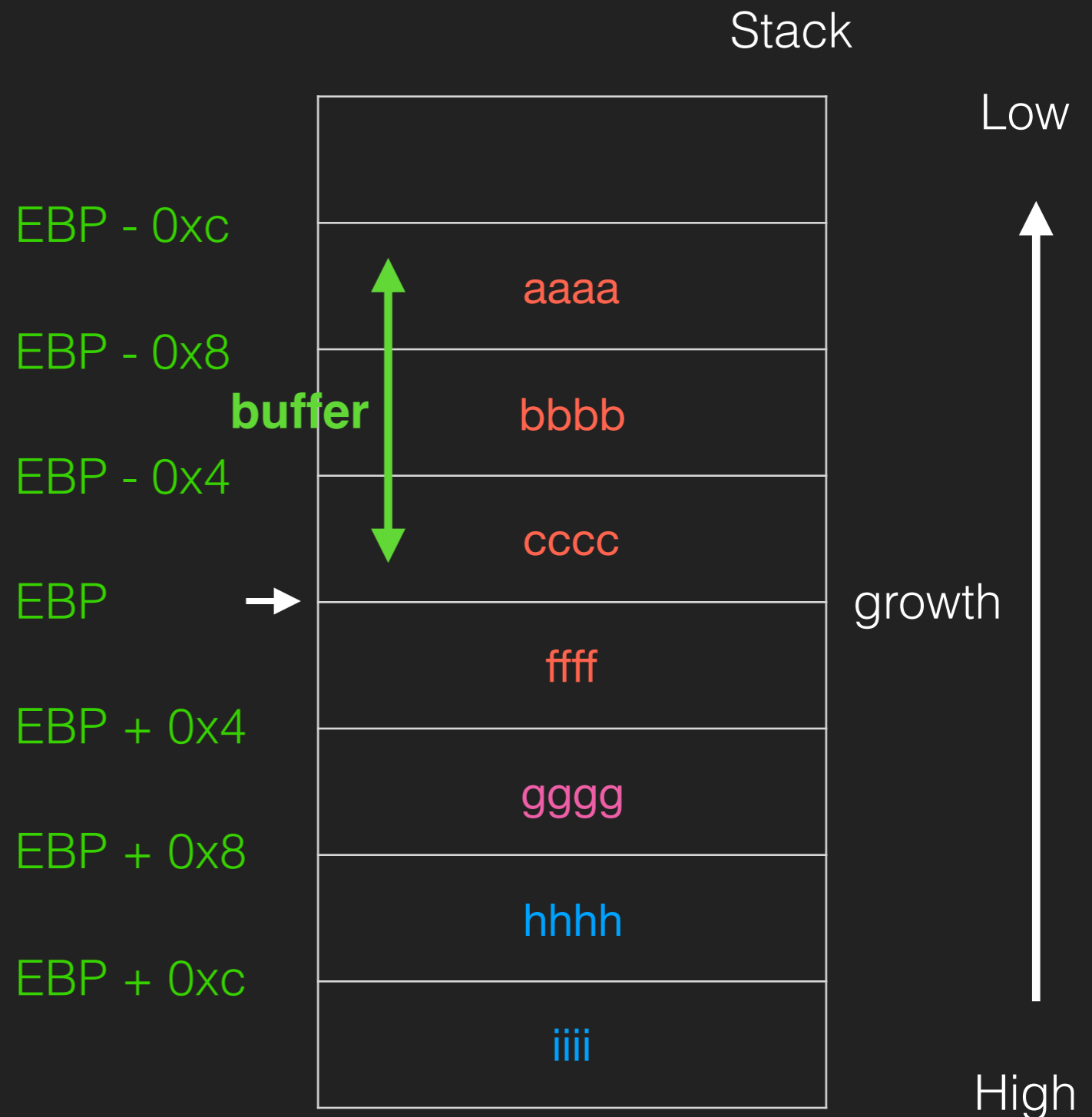
# Stack Overflow

- How to calculate the offset
  - gdb
  - cyclic in pwntools



# Stack Overflow

- Total bytes needed to pad from buf to return address
  - 12 (buf)
  - 8 (alignment)
  - 4 (ebp)
  - $12 + 8 + 4 = 24$  bytes



# Exploit

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

# Exploit

- Shellcode
  - Code written in machine language (can be execute directly)
  - Execution on input data
  - Example

`\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
  - Converting the shellcode to assembly will look like this.

\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
```

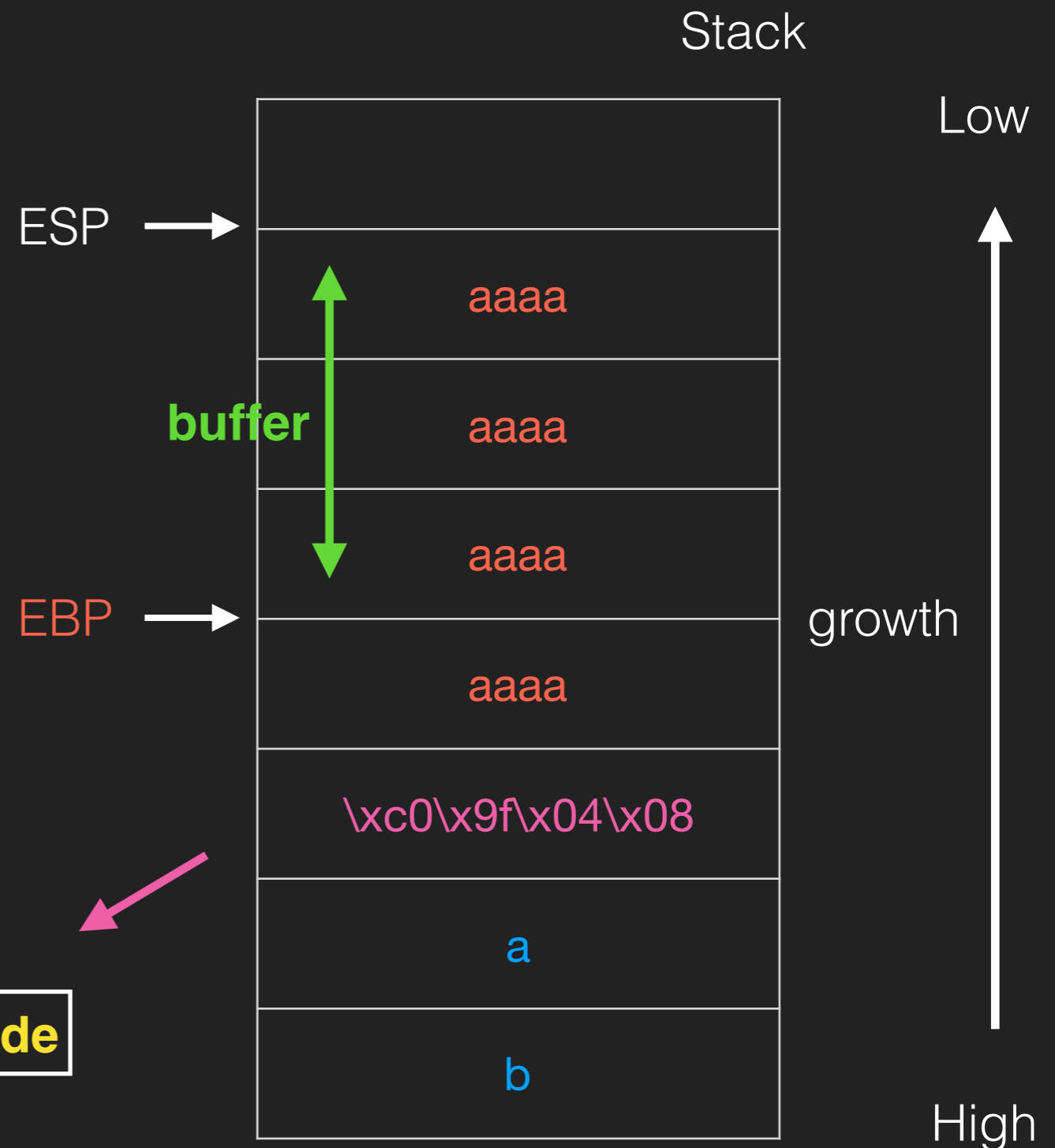
```
execve("/bin/sh", NULL, NULL);
```

# Exploit

- Shellcode
  - Write shellcode into any buffer.
  - Use buffer overflow to overwrite the return address, then it will return to the shellcode and execute.

0x08049fc0






**shellcode**



# 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)
    - Shellcode on **data** buffer can not be executed.
    -  stack
    -  heap
    -  .data
    -  .bss
- 

# Protection

- DEP (Data Execution Prevention)
  - DEP On / Off
    - `gcc -z execstack`
    - `execstack --set-execstack`
    - `execstack --clear-execstack`
  - Check Program Header - GNU\_STACK
    - `readelf --program-headers ${binary}`

# Protection

- ASLR (Address Space Layout Randomization)
  - System dependent
    - Linux - `cat /proc/sys/kernel/randomize_va_space`
  - Default is 2
    - 0 - Turn off ASLR
    - 1 - stack, shared library, mmap()
    - 2 - All the settings from mode 1 with the start of brk() area

# Protection

- Stack Guard (Stack Canary)
  - A random value will be pushed into stack after return address and ebp are pushed.
  - Check the canary before function returns.

# Protection

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

# Protection

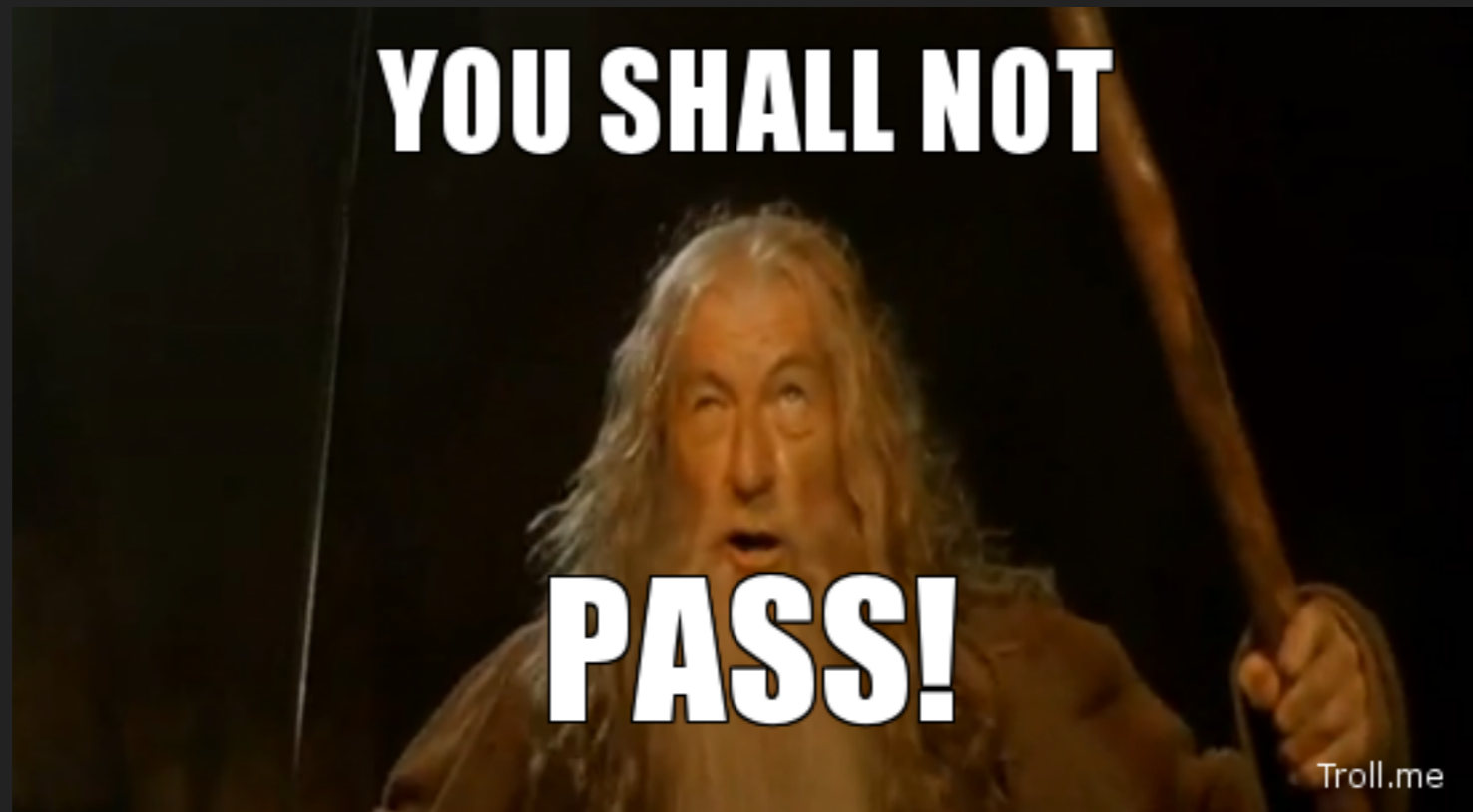
- Stack Guard (Stack Canary)

```
» ./bof2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** 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 are writable
  - Partial
    - .got.plt is writable
  - Full
    - .got / .got.plt are read-only
    - Every library function used in program is resolved during the loading time.

# Protection

- RELRO (Relocation Read Only)
  - Default is Partial
  - How to open full mode / disable mode
    - `gcc -Wl,-z,relro,-z,now` / `gcc -Wl,-z,norelro`
- Check dynamic tag
  - `DT_BIND_NOW` - Full
  - `GNU_RELRO` - Partial

# Protection

- PIE (Position Independent Executable)
  - Default is off. (before GCC 6)
  - Text and data section will be affected by ASLR. Thus, exploit would be more difficult when PIE is on.

# Protection

- PIE (Position Independent Executable)
  - On / Off
    - `gcc -fPIC -pie / gcc -no-pie`
- Check ELF header
  - DYN (shared object file)

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

# Protection

- Check protection
  - 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

- Return to text
- Use the code of program (Located at .text section, definitely executable)
- Without PIE, the address of code is fixed.

```
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
```

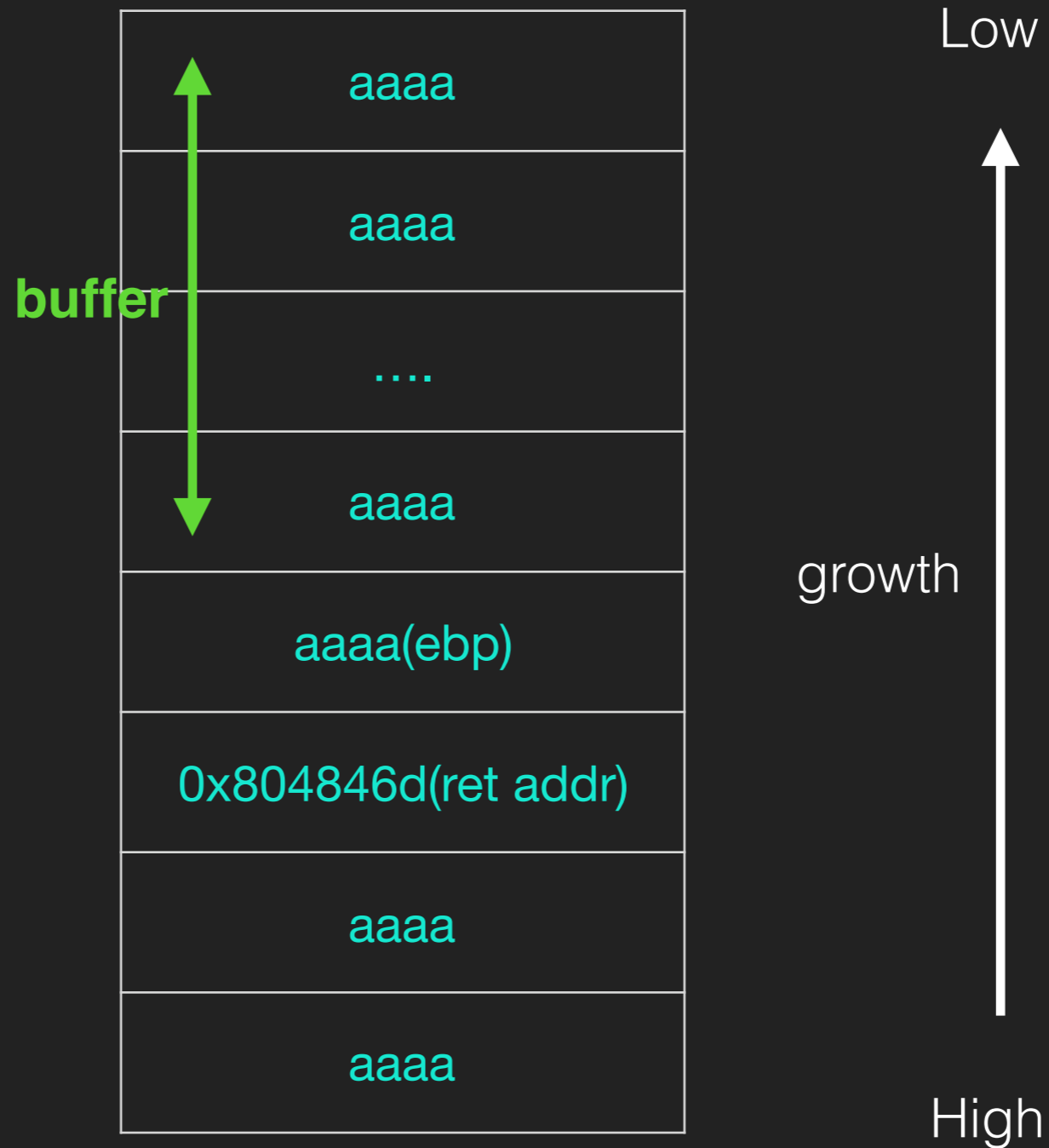
# Exploit

- Return to text
- Demo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void neveruse() {
5 system("/bin/sh");
6 }
7
8 void smash() {
9 char buf[40];
10 gets(buf);
11 }
12
13 int main() {
14 puts("Let's start smashing the stack!!");
15 smash();
16 }
```

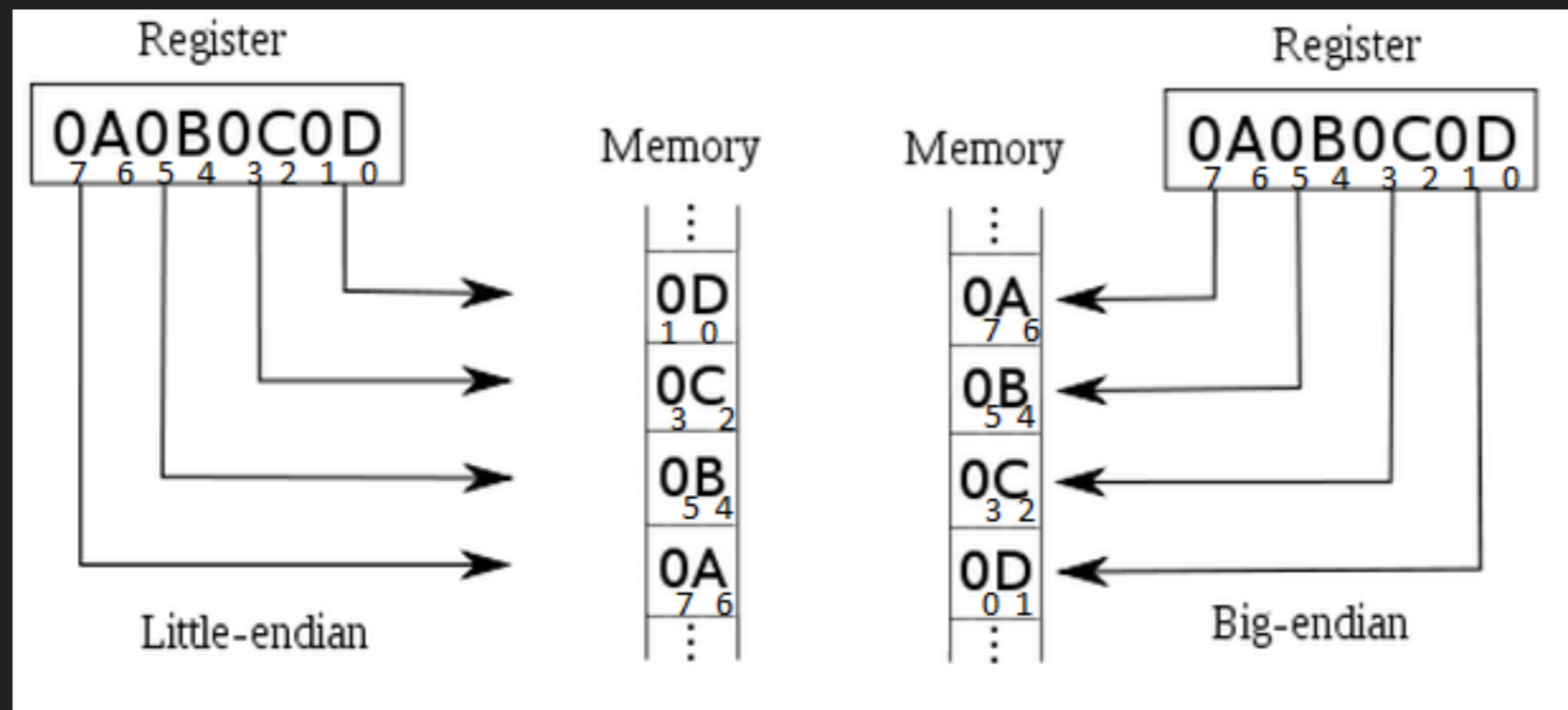
# Exploit

Stack



# Exploit

- Return to text
- Endianness



[https://upload.wikimedia.org/wikipedia/en/7/77/Big-little\\_endian.png](https://upload.wikimedia.org/wikipedia/en/7/77/Big-little_endian.png)

# Exploit

- Return to text
  - Endianness
    - x86 architecture use 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
  - In most case, we can not finish our exploit just with the code of program.
- 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
  - Static Linking
    - Put machine code of all library functions in ELF when linking.
  - Dynamic Linking
    - Mark library function with a symbol of dynamic linking
    - Shared object will be loaded into memory with ELF.

# 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                             |
| 0x00007ffffffffffde000 | 0x00007ffffffffff000 | rw-p | [stack]                            |
| 0xffffffffffff600000   | 0xffffffffffff601000 | r-xp | [vsyscall]                         |

# Lazy Binding

- Some library functions may not be called during the lifetime of a running program.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void edgeman() {
5 char chr;
6 printf("Could you be my friend?");
7 scanf("%c%c", &chr);
8 printf("Thx!");
9 }
10
11 int main() {
12 printf("Hello World!");
13 return 0;
14 }
```

here

# Lazy Binding

- The ELF with dynamic linking uses lazy binding. That means the program will not resolve the real function address until the very first call of that function.

# Lazy Binding

- ELF uses PLT (Procedure Linkage Table) to implement lazy binding.

<Foo@plt>:

0x400450: jmp        Foo@got

0x400456: push        n

0x40045b: push        module ID

0x400461: jmp        \_dl\_runtime\_resolve

<Foo@got>: 0x400456



# Lazy Binding

<Foo@plt>:

0x400450: jmp

Foo@got

<Foo@got>: 0x400456

0x400456: push

n

0x40045b: push

module ID

0x400461: jmp

\_dl\_runtime\_resolve



# Lazy Binding

<Foo@plt>:

0x400450: jmp        Foo@got

<Foo@got>: 0x400456

0x400456: push     n

0x40045b: push     module ID

0x400461: jmp        \_dl\_runtime\_resolve



# Lazy Binding

<Foo@plt>:

0x400450: jmp        Foo@got

<Foo@got>: 0x400456

0x400456: push        n

0x40045b: push        module ID

0x400461: jmp        \_dl\_runtime\_resolve



# Lazy Binding

<Foo@plt>:

0x400450: jmp        Foo@got

0x400456: push        n

0x40045b: push        module ID

0x400461: jmp        \_dl\_runtime\_resolve

<Foo@got>: 0x400456



# Lazy Binding

<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

<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

- n
  - index of Foo in .rel.plt
- module ID
  - name of library

<Foo@plt>:

0x400450: jmp        Foo@got

0x400456: push        n

0x40045b: push        module ID

0x400461: jmp        \_dl\_runtime\_resolve

# Lazy Binding

- 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

- See Angelboy - Execution p.17 for more details.

# Exploit

- Return to libc
  - In addition to the code in binary, we can use code in shared library.
    - system
    - execve
    - ...

# Exploit

- Return to libc
  - The base address of libc will be random each time the program executes due to ASLR. We need to leak the information we need.
    - Once we get the address of somewhere in libc, we can have the base address by calculating the offset.
  - Then, we can use any function in libc with function offset.
    - In the same libc, each function offset is fixed.

# Exploit

- Return to libc
  - Find libc address
    - GOT
    - stack
      - e.g., return address of main
    - heap
      - e.g., main\_arena

# Exploit

- Return to libc
- Find library function offset

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

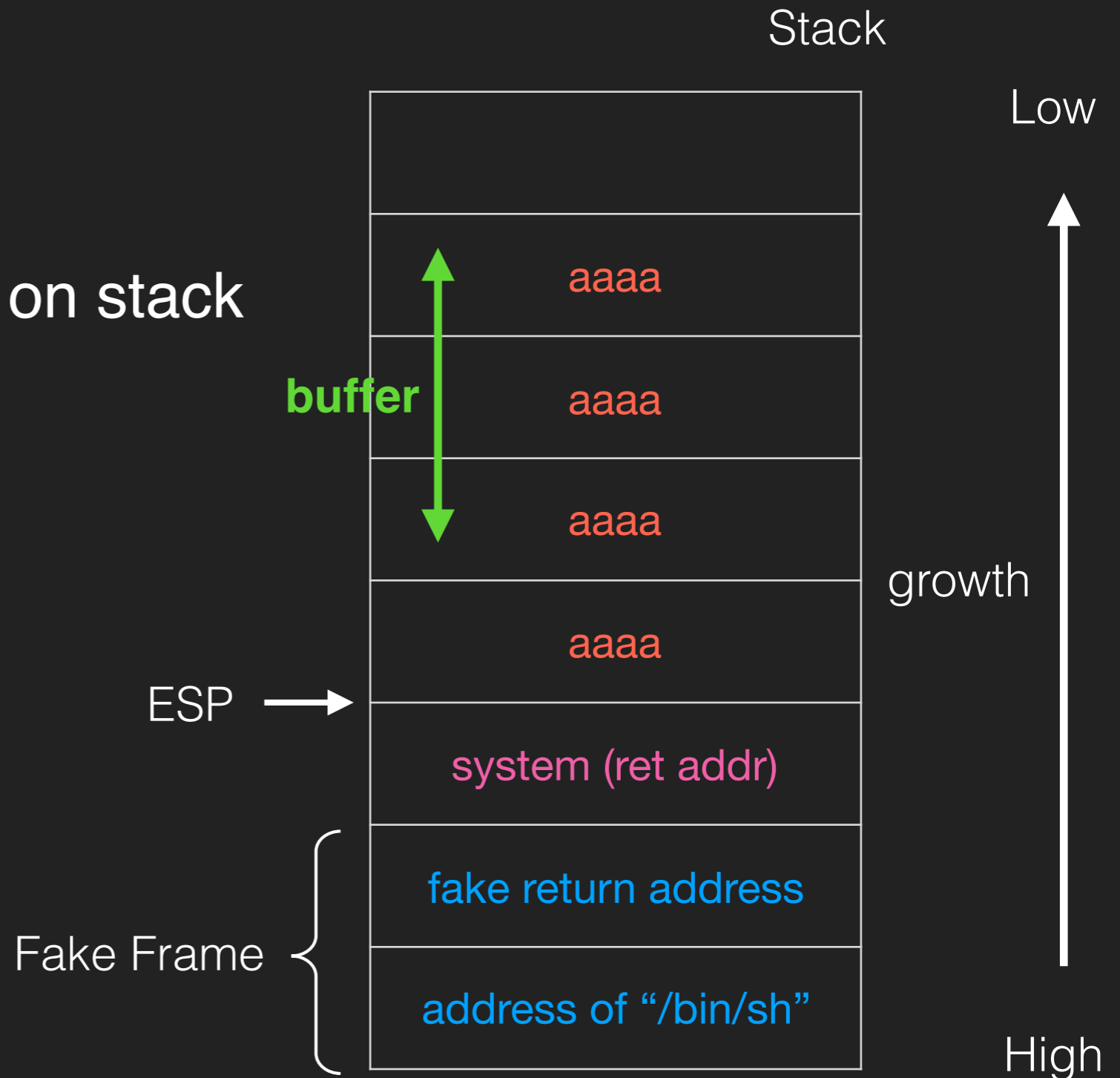
```
» readelf -s libc_64.so.6 | grep system
 225: 00000000000137c20 70 FUNC GLOBAL DEFAULT 13 svcerr_systemerr@@GLIBC_2.2.5
 584: 00000000000045390 45 FUNC GLOBAL DEFAULT 13 __libc_system@@GLIBC_PRIVATE
1351: 00000000000045390 45 FUNC WEAK DEFAULT 13 system@@GLIBC_2.2.5
```

# Exploit

- Return to libc
  - After getting the libc base address
    - Forge a function call on stack
    - GOT hijack
    - ...

# Exploit

- Return to libc
- Forge a function call on stack
  - Constraints
    - overflow
    - Arbitrary write



# Exploit

- Return to libc
- GOT hijack
  - Constraints
    - Arbitrary write
- `gets(buf) -> system(buf)`

<gets@plt>:

0x400450: jmp gets@got

0x400456: push n

0x40045b: jmp PLT0

<gets@got>:

0x7fff7a6dead

0x7fff7a6dead <system>: ...



# Exploit

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

# Exploit

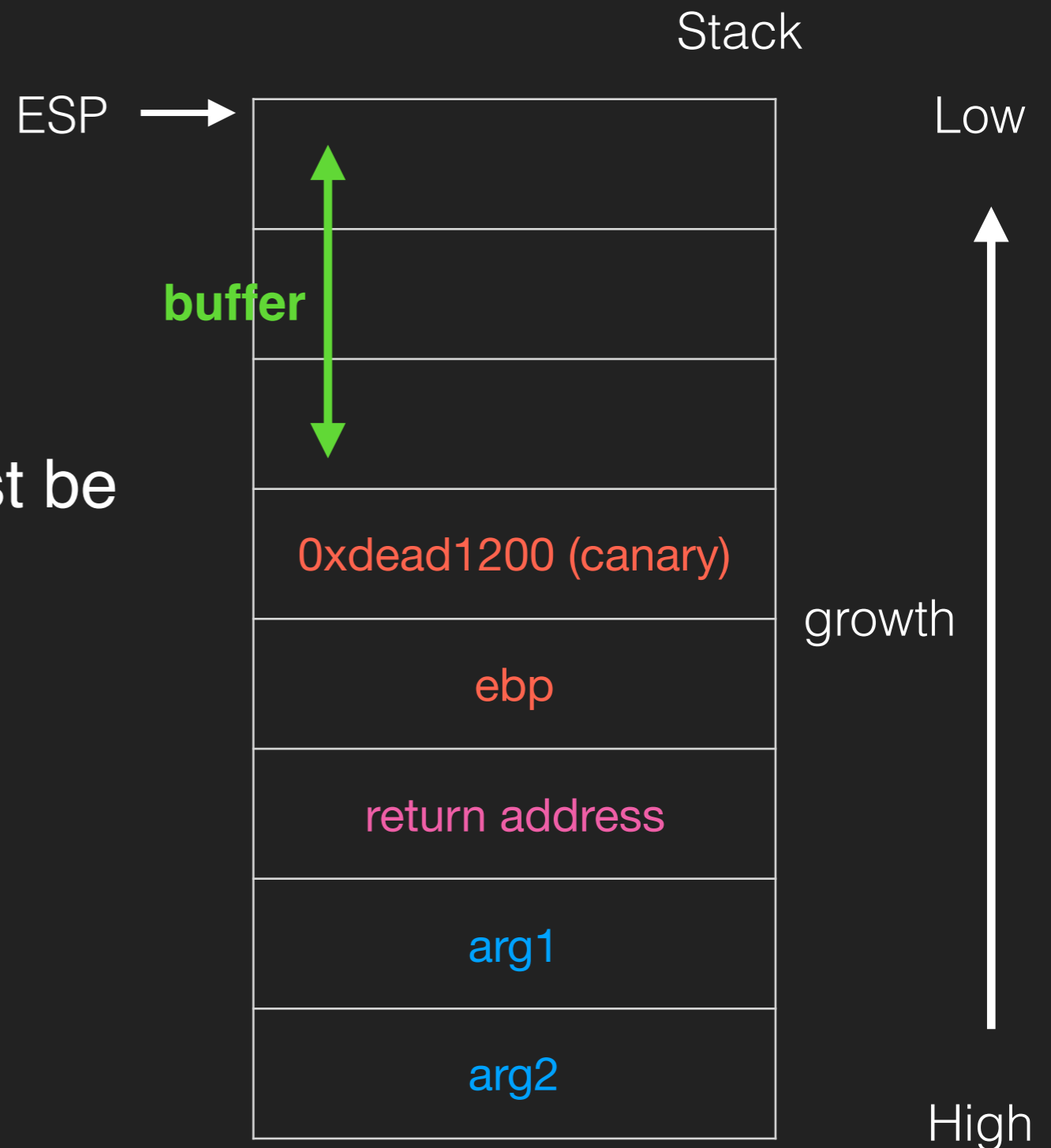
- Bypass stack guard
  - leak canary
  - GOT hijack
  - ...

# Exploit

- Bypass stack guard
  - leak canary
    - Some function doesn't put null byte at the end of buffer after getting the user's input. We can take advantage of this to leak canary. Of course, stack or libc may be leaked, too.
      - read()
      - strcpy()
      - strncpy()

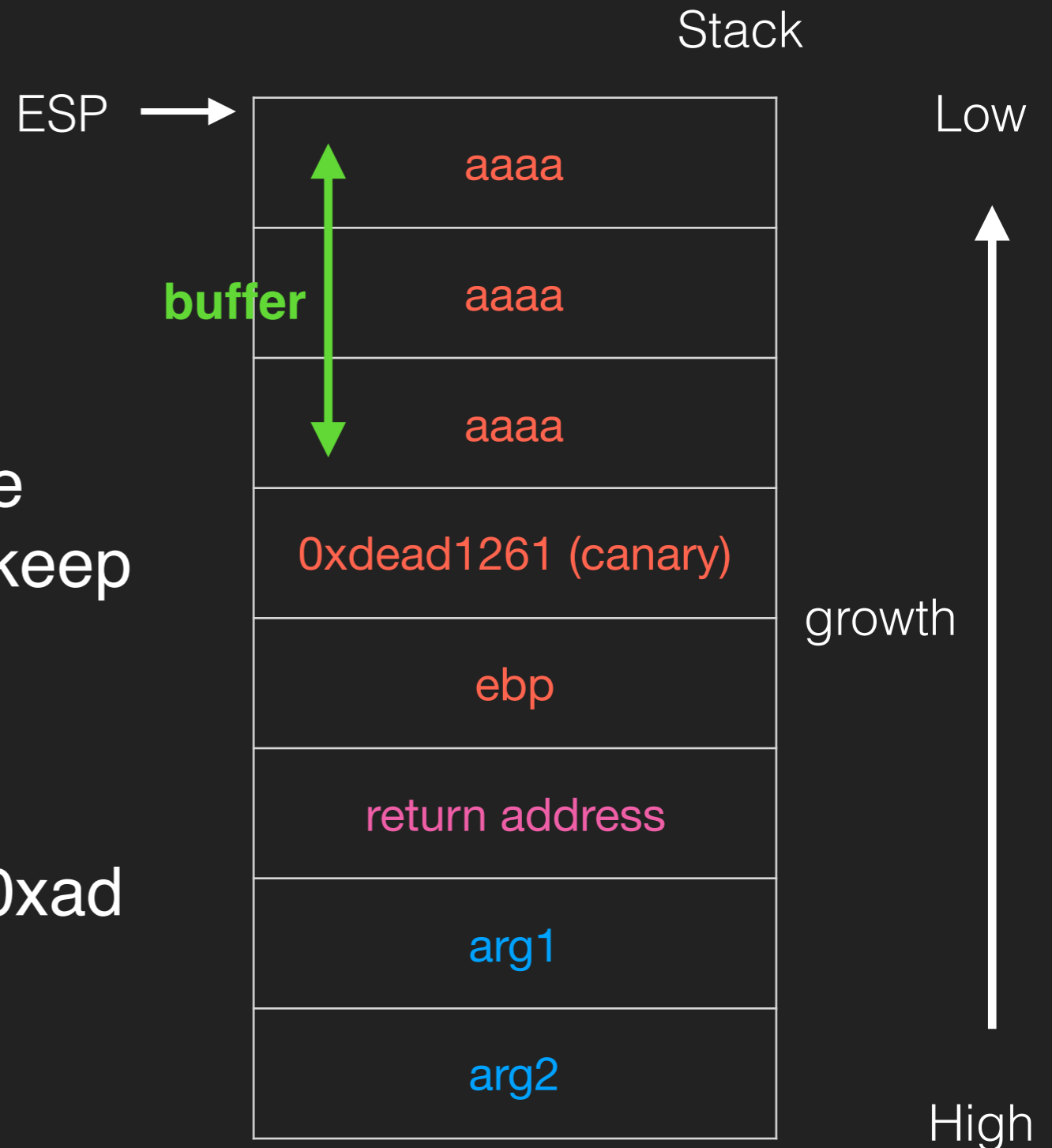
# Exploit

- Bypass stack guard
- leak canary
  - The end of canary must be null byte.



# Exploit

- Bypass stack guard
- leak canary
  - If there is a function like `printf("%s", buf)`, it will keep outputting data until reaching a null byte.
- $a*12 + 0x61 + 0x12 + 0xad + 0xde + \dots$



# Exploit

- Bypass stack guard
  - GOT hijack
    - Constraints
      - Write arbitrarily
      - Partial / No RELRO

# Exploit

- Bypass stack guard
  - GOT hijack (cont.)
    - Overwrite the GOT entry of `__stack_chk_fail`. According to different data we write, we have different ways to exploit
      - NOP
        - Overflow directly, then do ROP
      - `one_gadget`
      - `system`
        - Forge a function call (no need of return address)
      - ...

# Reference

- [http://l4ys.tw/ROP\\_bamboofox.pdf](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/Executable_space_protection#Windows)
- [https://en.wikipedia.org/wiki/Address\\_space\\_layout\\_randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization)

# Contact

- Mail: wangzhelee@gmail.com