Pwn😆

yuawn

Outline

- fmt - Format String attack
- fmt - Read & Write
- Advanced fmt
- ROP stack migration
- UaF

FMT

format string attack

fmt

- Format string attack
- Read - information leak:
  - PIE, stack heap libc ASLR, canary
- Write😮 - almost write every where
- Powerful vulnerability

fmt 通常可以write everywhere
Moreover, even though there is PIE or ASLR protection, writing everywhere usually allows us to leak information.
That is to say, we can know where we want to write.

fmt

- int printf (const char * format, ...);
- int fprintf (FILE * stream, const char * format, ...);
- int vprintf (const char * format, va_list arg);
- int vfprintf (FILE * stream, const char * format, va_list arg);
- int vdprintf (int fd, const char *format, va_list ap);
- …

fmt

- ✓ fmt 可控
  - Constrains
  - fmt 長度很短
  - 過濾

```
printf( buf );
```

fmt - read

- int printf (const char * format, ...);
- rdi, rsi, rdx, rcx, r8, r9, STACK😍

參數傳遞

利用stack上面的殘留值來推算heap, stack …的ASLR

- rdi -> fmt
- %d,%p,%s,%c,%x -> read data on the stack
- information leak

fmt - read

```
printf( buf );
```

- input:"aaaaaaaa"
  output:"aaaaaaaa"

- input:"%p.%p.%p"
  output:"0x7fbdb0e858b0.0x1.0x7fbdb0e858b0"🤔

可以選擇要第幾個，這裡就是選第三個，不用把前面都leak出來，較方便

- input:"%3$p" 👍🏿
  output:"0x7fbdb0e858b0"

echo😶

```c
char buf[0x100];

while( scanf( "%s" , buf ) != EOF ){
    printf( buf );
    printf( "\n" );
}
```

```
yuawn@ubuntu18:~/csie/demo/echo$ ./echo
aaaaaaaaaaaa
aaaaaaaaaaaa
123
123
hello:D
hello:D
%p.%p.%p.%p.%p.%p
0x7f77d429e8b0.0x1.0x7f77d429e8b0.0x559306d43846.0x7f77d44c4500.0x70252e70252e7025
```

DEMO

fmt - write

%n

- 將已輸出之字元數目當成interger值(4 bytes)，寫入address。

fmt - write

```c
int a = 0;

printf( "a = %d\n" , a );
printf( "123abc%n\n" , &a );
```

```
yuawn@ubuntu18:~/csie/demo/n$ ./fmt_write
a = 0
123abc
a = 6
```

fmt - write

- printf("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaa%n" , &ptr) 😨

- printf("%100c%n" , char , &ptr) 👍🏿

  用空白填滿100個

fmt - write

- "%100c%n" -> 寫入 4 bytes \x64\x00\x00\x00
- "%100c%hn" -> 寫入 2 bytes \x64\x00
- "%100c%hhn" -> 寫入 1 bytes \x64

DEMO

fmt - exploit

- If format string is stored on the stack:
  - 將address放置於payload，得知address位於第n個參數，用%n$去refenrence它。
- %n$p -> leak libc, PIE, Heap, Stack: bypass ASLR
- %n$n -> write value
- read & write everywhere

Lab 5-1

complex

fmt

- 當format string不在stack上，如位於global .bss
- 無法利用fmt payload來放address

- 尋找stack上有利殘留值

fmt

- pointer , address
  - libc text
  - stack

- 通常找到的address固定，只能往同個地方寫，但希望能拆開寫。
- 希望找到一個stack address，其指向之地方又是stack address。
- 利用第一個address來改寫第二個address，用第二個address來建構真的要的address。

第一個固定不動
利用第一個來改動第二個
如此可以利用第二個來調整寫入的位置　　＝二段跳
把真正想寫的東西慢慢寫入

✓ RBP Chain

prologue　　mov rsp, rbp
pop rbp

舊的rbp跟新的rbp都是指在stack上

```
0x7fffffffe340: 0x00007fffffffe4e8      0x00005555555547c3
0x7fffffffe350: 0x0000000000000000      0x0000000000000000
0x7fffffffe360: 0x0000000000000000      0x0000000000000000
0x7fffffffe370: 0x0000000000000000      0x0000000000000000
0x7fffffffe380: 0x0000000000000000      0x0000000000000000
0x7fffffffe390: 0x0000000000000000      0x0000000000000000
0x7fffffffe3a0: 0x00007fffffffe408      0xfd8f63f73c448e00
rbp 0x7fffffffe3b0: 0x00007fffffffe3e0      0x0000555555554804
0x7fffffffe3c0: 0x00007ffff7de59a0      0x0000000000000001
0x7fffffffe3d0: 0x0000000000000002      0x0000000000000003
舊rbp 0x7fffffffe3e0: 0x00007fffffffe3f0      0x0000555555554817
0x7fffffffe3f0: 0x0000555555554850      0x00007ffff7a05b97
0x7fffffffe400: 0x0000000000000001      0x00007fffffffe4d8
0x7fffffffe410: 0x0000000100008000      0x0000555555554807
0x7fffffffe420: 0x0000000000000000      0xde2e546a25d3c125
0x7fffffffe430: 0x00005555555546a0      0x00007fffffffe4d0
0x7fffffffe440: 0x0000000000000000      0x0000000000000000
0x7fffffffe450: 0x8b7b013f7d73c125      0x8b7b118003edc125
0x7fffffffe460: 0x00007fff00000000      0x0000000000000000
0x7fffffffe470: 0x0000000000000000      0x00007ffff7de5733
0x7fffffffe480: 0x00007ffff7dcb638      0x0000000018670cb8
0x7fffffffe490: 0x0000000000000000      0x0000000000000000
0x7fffffffe4a0: 0x0000000000000000      0x00005555555546a0
0x7fffffffe4b0: 0x00007fffffffe4d0      0x00005555555546ca
0x7fffffffe4c0: 0x00007fffffffe4c8      0x000000000000001c
0x7fffffffe4d0: 0x0000000000000001      0x00007fffffffe71f
0x7fffffffe4e0: 0x0000000000000000      0x00007fffffffe740
0x7fffffffe4f0: 0x00007fffffffe753      0x00007fffffffed3f
0x7fffffffe500: 0x00007fffffffed73      0x00007fffffffed95
0x7fffffffe510: 0x00007fffffffeda4      0x00007fffffffedb5
0x7fffffffe520: 0x00007fffffffedd2      0x00007fffffffede4
0x7fffffffe530: 0x00007fffffffedef      0x00007fffffffee0f
```

0xdeadeef

用rbp內存的位置寫入舊rbp
來調整要寫入的位置
只需改最小的一個byte即可（穩定）
f0對應到白匡
f2對應到籃筐
f4對應到黃匡

```
0x7fffffffe340: 0x00007fffffffe4e8    0x00005555555547c3
0x7fffffffe350: 0x0000000000000000    0x0000000000000000
0x7fffffffe360: 0x0000000000000000    0x0000000000000000
0x7fffffffe370: 0x0000000000000000    0x0000000000000000
0x7fffffffe380: 0x0000000000000000    0x0000000000000000
0x7fffffffe390: 0x0000000000000000    0x0000000000000000
0x7fffffffe3a0: 0x00007fffffffe408    0xfd8f63f73c448e00
0x7fffffffe3b0: 0x00007fffffffe3e0    0x0000555555554804
0x7fffffffe3c0: 0x00007ffff7de59a0    0x0000000000000001
0x7fffffffe3d0: 0x0000000000000002    0x0000000000000003
0x7fffffffe3e0: 0x00007fffffffe3f0    0x0000555555554817
0x7fffffffe3f0: 0x0000555555554850    0x00007ffff7a05b97
0x7fffffffe400: 0x0000000000000001    0x00007fffffffe4d8
0x7fffffffe410: 0x0000000100008000    0x0000555555554807
0x7fffffffe420: 0x0000000000000000    0xde2e546a25d3c125
0x7fffffffe430: 0x00005555555546a0    0x00007fffffffe4d0
0x7fffffffe440: 0x0000000000000000    0x0000000000000000
0x7fffffffe450: 0x8b7b013f7d73c125    0x8b7b118003edc125
0x7fffffffe460: 0x00007fff00000000    0x0000000000000000
0x7fffffffe470: 0x0000000000000000    0x00007ffff7de5733
0x7fffffffe480: 0x00007ffff7dcb638    0x0000000018670cb8
0x7fffffffe490: 0x0000000000000000    0x0000000000000000
0x7fffffffe4a0: 0x0000000000000000    0x00005555555546a0
0x7fffffffe4b0: 0x00007fffffffe4d0    0x00005555555546ca
0x7fffffffe4c0: 0x00007fffffffe4c8    0x000000000000001c
0x7fffffffe4d0: 0x0000000000000001    0x00007fffffffe71f
0x7fffffffe4e0: 0x0000000000000000    0x00007fffffffe740
0x7fffffffe4f0: 0x00007fffffffe753    0x00007fffffffed3f
0x7fffffffe500: 0x00007fffffffed73    0x00007fffffffed95
0x7fffffffe510: 0x00007fffffffeda4    0x00007fffffffedb5
0x7fffffffe520: 0x00007fffffffedd2    0x00007fffffffede4
0x7fffffffe530: 0x00007fffffffedef    0x00007fffffffee0f
```

Lab 5-2

stack migration

- ROP chain
- x64 address -> 8 bytes　　ROP問題：payload會太長…
- long return address sled
- Not enough memory to prepare ROP chain😢

stack migration

lab5-3: 只能overflow到return address

- leave ret gadget

在ret 塞入leave ret的gadget
->會再跑一次 leave ret
= mov rsp, rbp
pop rbp
= 把stack搬到你指定的位置

注意：
假設真正的ROP payload在a
那塞leave ret gadget的值要是a-0x8，才會剛好

- mov rsp, rbp
  pop rbp

- Overflow fake rbp -> leave ret
  rbp -> fake rbp
  ret -> leave ret gadget

- leave ret again:
  mov rsp, rbp -> 將stack搬至fake rbp
  pop rbp
  ret -> 跳到新的rop chain

stack migration

- 將stack搬到可儲存ROP payload的地方。
- 在有限的情況下，利用stack migration讓ROP持續活下去💪。

Lab 5-3

UAF

- Use After Free
- free(ptr)
- ptr未清空 -> Dangling pointer
- Object struct -> 解析方式不同
錯誤的reference存取方式
- function pointer -> control rip

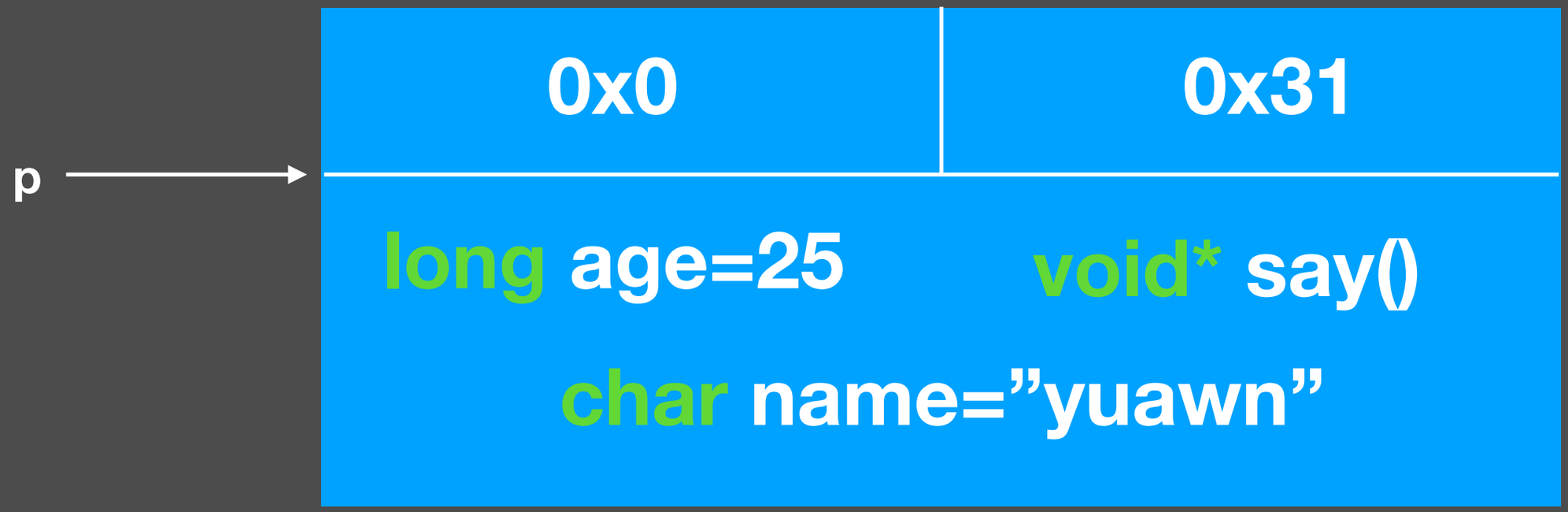
```
struct human{
    long age;
    void (*say)();
    char name[0x10]
};

struct str{
    long len;
    char data[0x18]
};
```

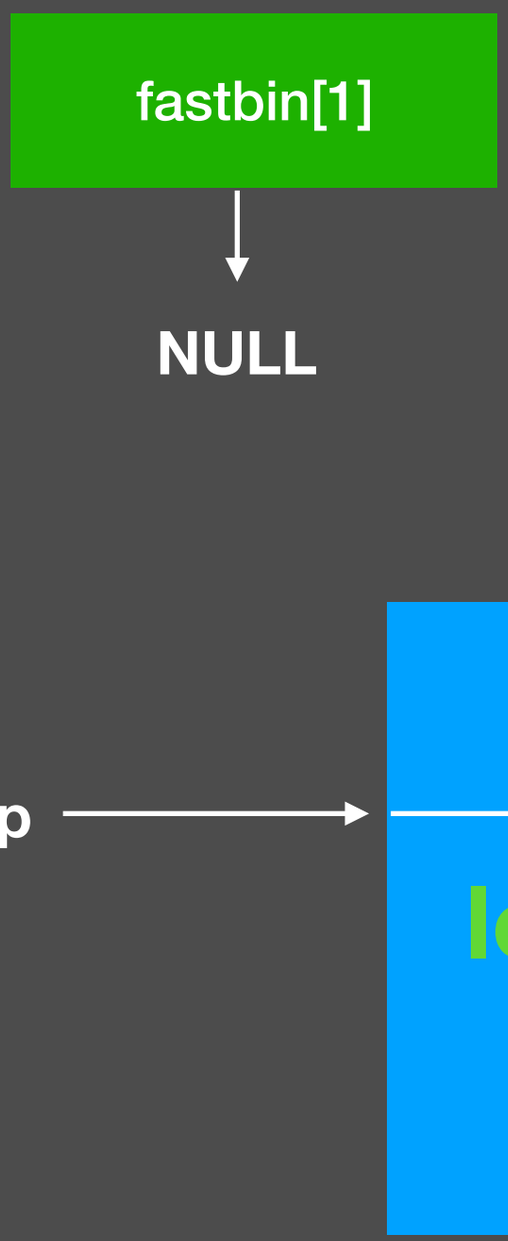
UAF

```
void say_hello(){
    puts( "Hello!" );
}
```

```
struct human *p = (struct human*)malloc( sizeof( struct human ) );
p->age = 25;
p->say = say_hello;
strncpy( p->name , "yuawn" , 5 );
printf( "name:%s\nage:%ld\n" , p->name , p->age );
p->say();
```



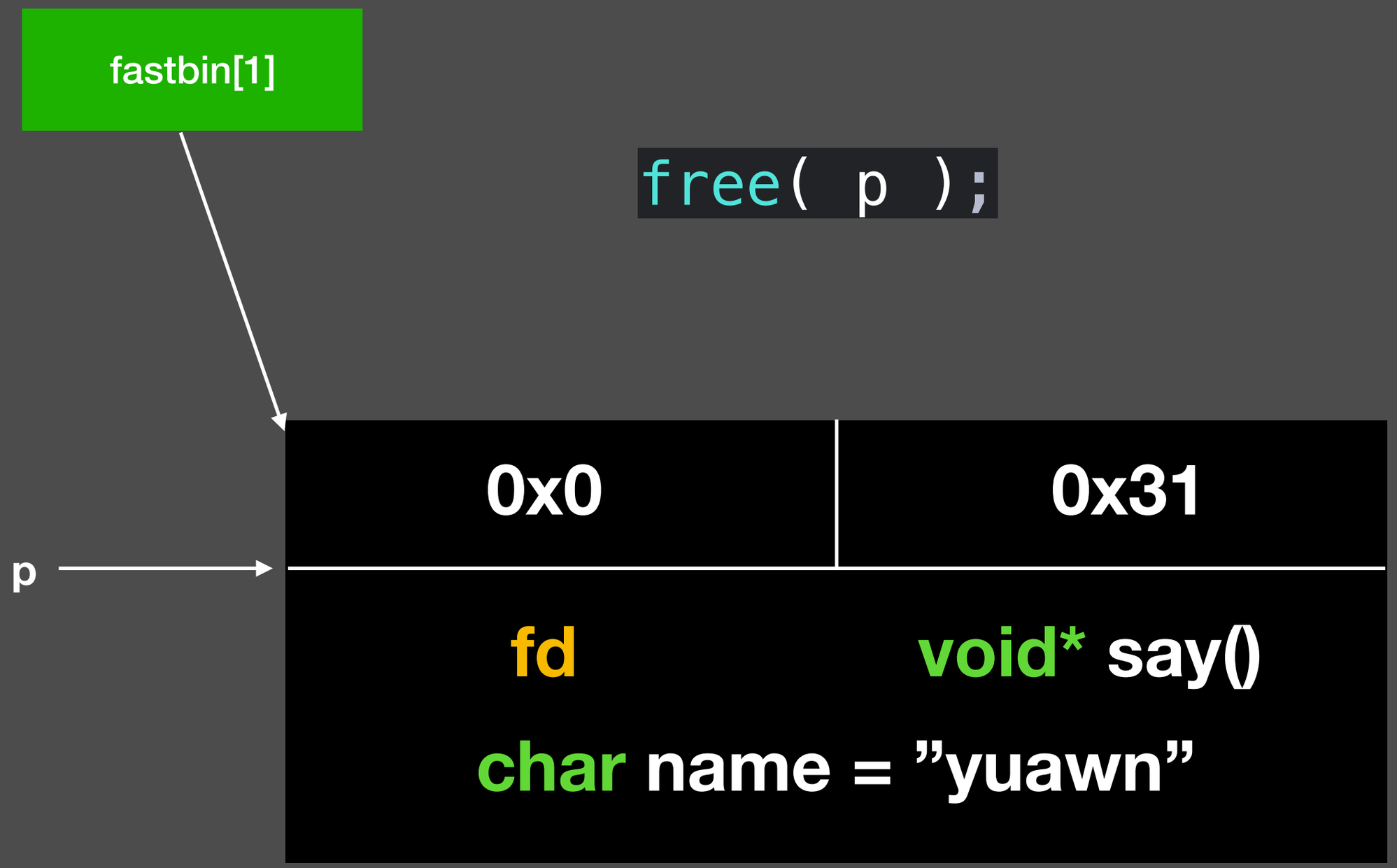
UAF



UAF

fastbin[1]

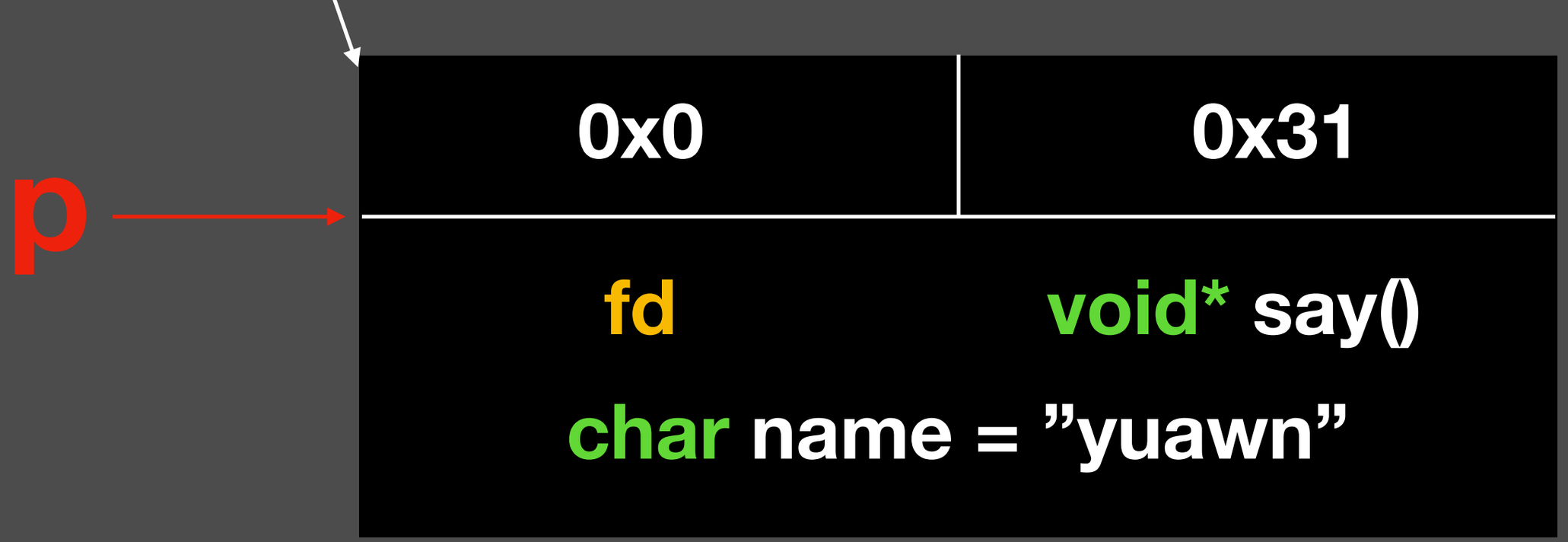
```
free( p );
```



UAF

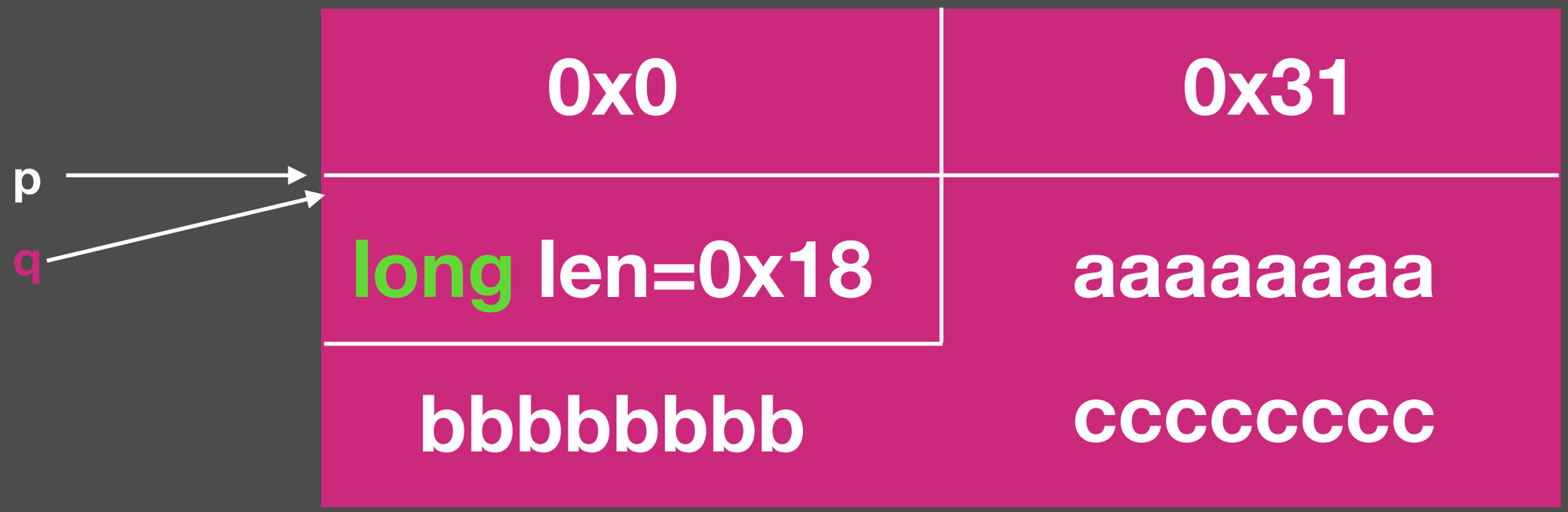
fastbin[1]

Dangling pointer

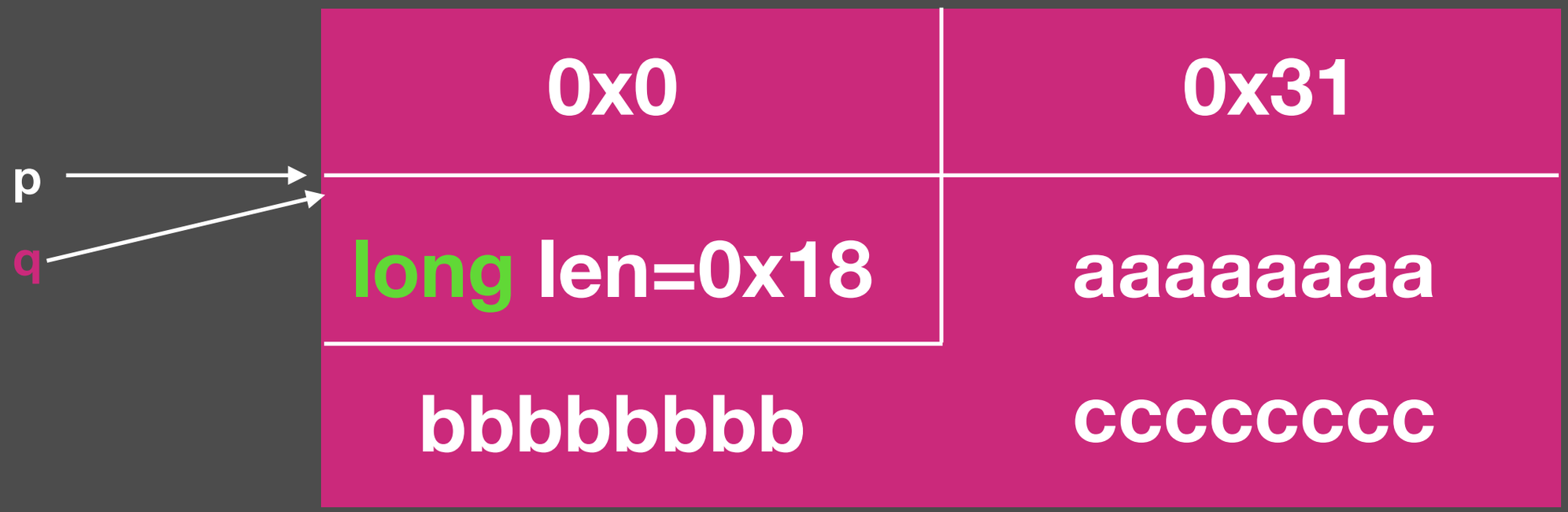


UAF

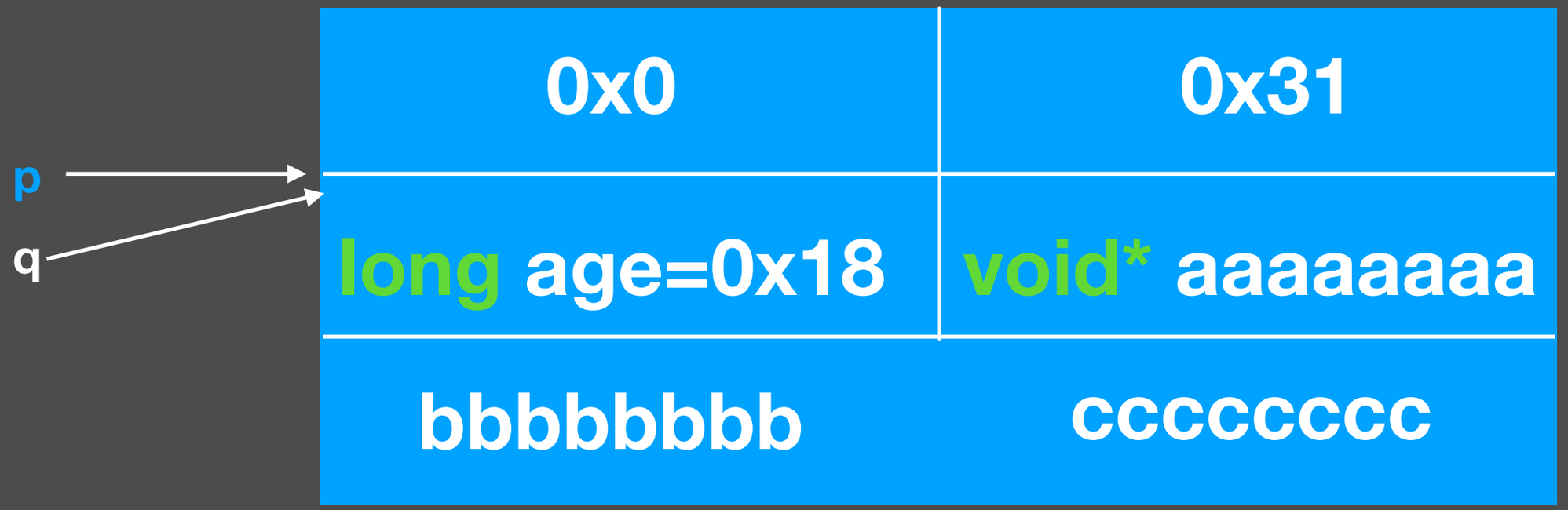
```
struct str *q = (struct str*)malloc( sizeof( struct str ) );
char buf[0x18] = “aaaaaaaaabbbbbbbbccccccccc";
q->len = strlen( buf );
strncpy( q->len , buf , sizeof( buf ) );
```



UAF

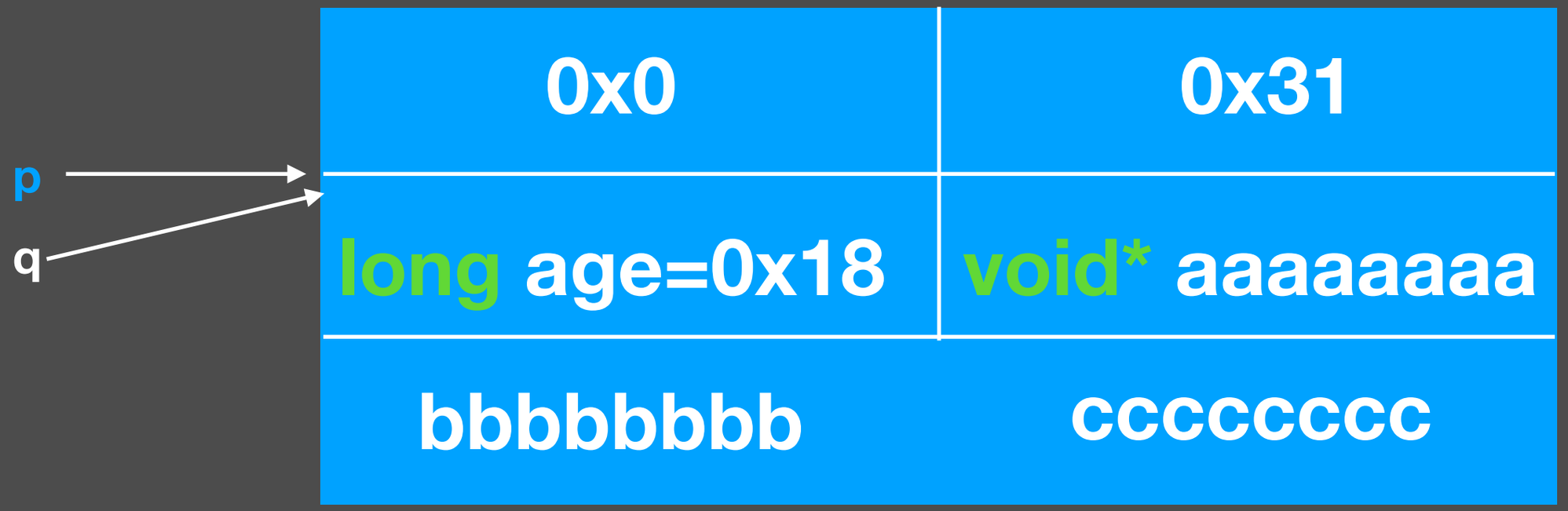


UAF



UAF

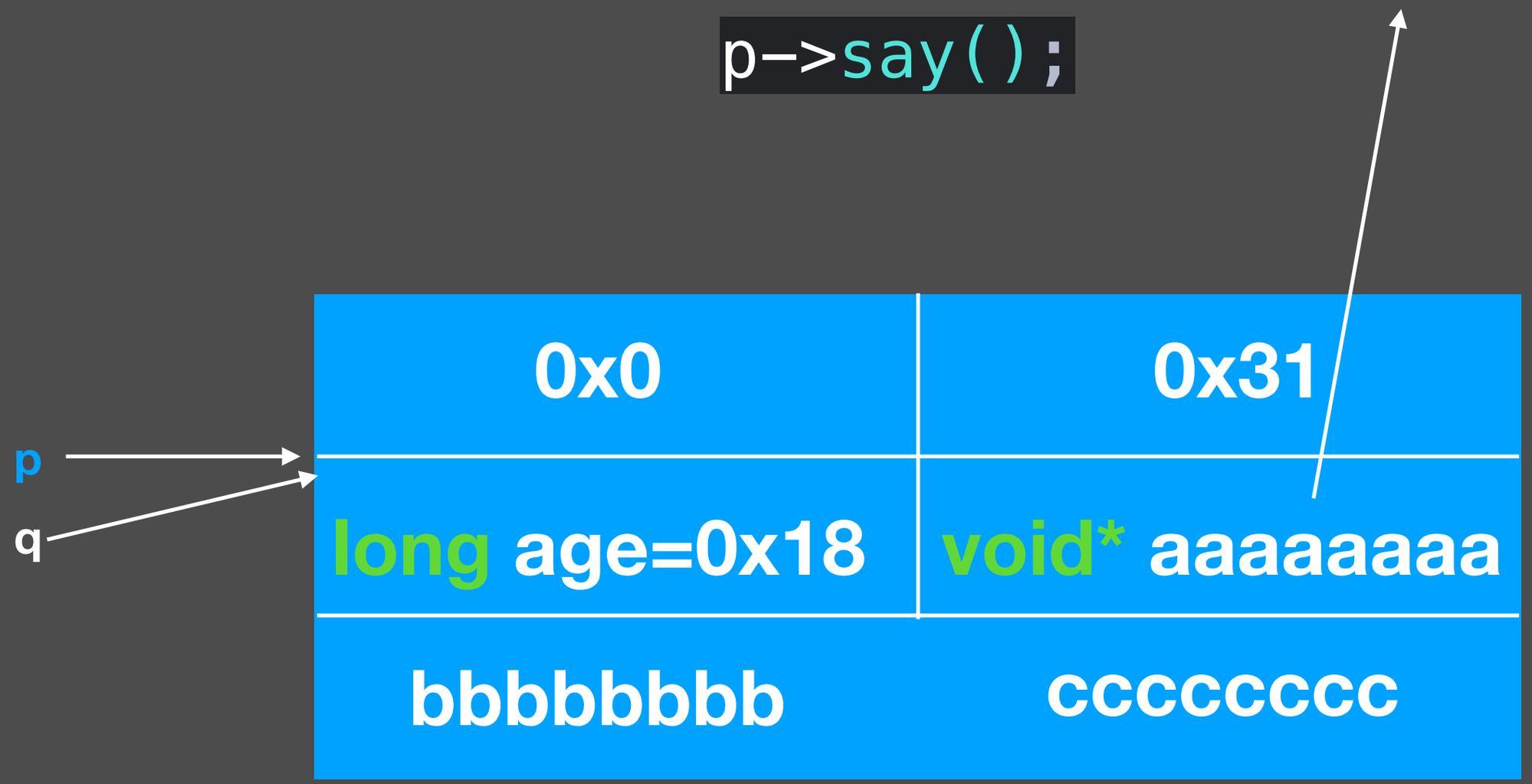
```
p->say();
```



UAF

rip = 0x6161616161616161

```
p->say();
```



Thanks for attention!

(๑˃ㅂ˂)و✧