Tcache 机制分析

> 如有错误以及图片不清晰等问题请提交 issue，谢谢 ~
>
> 源路径：[https://github.com/HATTER-LONG/glibc_Learnging]

Tcache 机制

Tcache

- tcache 机制

在 Glibc 的 2.26 中 新增了 Tcache 机制，这是 ptmalloc2 的 chunk freed 缓存机制，glibc 中默认是开启的，在进行实际 malloc 的时候会先尝试使用 tcache 进行分配，分配内存源码如下：

```
void *
__libc_malloc (size_t bytes)
{
    .....

#if USE_TCACHE
  /* int_free also calls request2size, be careful to not pad twice.  */
  size_t tbytes;
  if (!checked_request2size (bytes, &tbytes)) // tbytes 为 bytes 请求的 转换后得到的 chunk 的 size
    {
      __set_errno (ENOMEM);
      return NULL;
    }
  size_t tc_idx = csize2tidx (tbytes); // 根据大小 tbytes ， 找到 tcache->entries 索引

  MAYBE_INIT_TCACHE ();

  DIAG_PUSH_NEEDS_COMMENT;
  if (tc_idx < mp_.tcache_bins
      && tcache
      && tcache->counts[tc_idx] > 0) // 如果 tcache->entries[tc_idx] 有 chunk ，就返回
    {
      return tcache_get (tc_idx); // 调用 tcache_get 拿到 chunk 然后返回
    }
  DIAG_POP_NEEDS_COMMENT;
#endif

    .....

}
```

Tcache 之计算 Chunk

- 参考文章--C 标准库函数 宏定义浅析
- 参考文章--堆溢出学习笔记 (linux)

tcache 也是使用类似 bins 的方式管理 tcache 内存，首先是通过申请的内存大小计算所需的 chunk 大小，及 `checked_request2size` 接口：

```


/* MALLOC_ALIGNMENT is the minimum alignment for malloc'ed chunks.  It
   must be a power of two at least 2 * SIZE_SZ, even on machines for
   which smaller alignments would suffice. It may be defined as larger
   than this though. Note however that code and data structures are
   optimized for the case of 8-byte alignment.  */
#define MALLOC_ALIGNMENT (2 * SIZE_SZ < __alignof__ (long double) \
                          ? __alignof__ (long double) : 2 * SIZE_SZ)

/* The corresponding bit mask value.  */
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))

/* The smallest size we can malloc is an aligned minimal chunk */
#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))

#ifndef INTERNAL_SIZE_T
# define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size.  */
#define SIZE_SZ (sizeof (INTERNAL_SIZE_T))

#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

/* Check if REQ overflows when padded and aligned and if the resulting value
   is less than PTRDIFF_T.  Returns TRUE and the requested size or MINSIZE in
   case the value is less than MINSIZE on SZ or false if any of the previous
   check fail.  */
static inline bool
checked_request2size (size_t req, size_t *sz) __nonnull (1)
{
  if (__glibc_unlikely (req > PTRDIFF_MAX))
    return false;
  *sz = request2size (req);
  return true;
}
```

> 注意 `__nonnull` 表示入参不应为 null，如果显示传入了 null，编译期会报警告。

首先判断请求大小是否溢出，调用这次的主角 `request2size` 计算需要分配的 chunk 大小，计算的方式为用户申请的长度（req) + Chunk 头（SIZE_SZ），一般 SIZE_SZ 为 size_t 及 32 位 4 字节、64 位 8 字节。

注意 MIN_CHUNK_SIZE 计算方式是 `malloc_chunk` 结构体偏移到 fd_nextsize 成员的大小，这里涉及到 chunk 共享的知识点：

> 当一个 chunk 为空闲时，至少要有 prev_size、size、fd 和 bk 四个参数，因此 MINSIZE 就代表了这四个参数需要占用的内存大小；而当一个 chunk 被使用时，prev_size 可能会被前一个 chunk 用来存储，而当前 chunk 也可以使用下一个 chunk 的 prev_size，互相抵消；fd 和 bk 也会被当作内存存储数据，因此当 chunk 被使用时，只剩下了 size 参数需要设置，request2size 中的 SIZE_SZ 就是 INTERNAL_SIZE_T 类型的大小，因此至少需要 req + SIZE_SZ 的内存大小。MALLOC_ALIGN_MASK 用来对齐，至此 request2size 就计算出了所需的 chunk 的大小。

- 以 64 位系统举例：
  - 申请 24 字节堆块为例，24 + chunk 头 = 32 (100000)，接下来加上 MALLOC_ALIGN_MASK 01111 (101111) 最后 MALLOC_ALIGN_MASK 取反 10000，按位与的结果就是 100000 了，即 32。
  - 申请 25 字节堆块，25 + chunk 头 = 33 (100001)，加上 MALLOC_ALIGN_MASK 01111 (110000) 接下来按位与上 ~MALLOC_ALIGN_MASK (10000) 结果为 110000 及 48。

Tcache 之计算索引

- 参考文章--glibc2.26--tcache

当计算出 `chunk` 需要的大小后，接下来便是要找到对应的 chunk 索引了。这是通过 `csize2tidx` 宏来实现的

```
/* When "x" is from chunksize().  */
# define csize2tidx(x) (((x) - MINSIZE + MALLOC_ALIGNMENT - 1) / MALLOC_ALIGNMENT)
```

chunk 大小减去一个 chunk 头大小，在整除 MALLOC_ALIGNMENT(32 位：8, 64 位：16)，当前看起来是以某种固定大小进行递增的数组，类似 ptmalloc bins 管理内存一样以递增的方式对应不同的 bins。

- 想要具体了解怎么进行的查找需要先了解几个方面：
  1. Tcache 如何管理内存的？
  2. Tcache 如何进行初始化？
  3. Tcache 如何进行插入获取操作？

接下来会继续按照 malloc 源码进行分析当了解了以上问题，再来讨论计算索引的方式。

Tcache 之初始化

按照源码来看下一步会进行 Tcache 初始化：

```
# define MAYBE_INIT_TCACHE() \
  if (__glibc_unlikely (tcache == NULL)) \
    tcache_init();

void *
__libc_malloc (size_t bytes)
{
    .....
    MAYBE_INIT_TCACHE ();
    ....

}
```

- 解析 `MAYBE_INIT_TCACHE` 初始化前，先了解几个 tcache 内存管理的结构体。Tcache 管理 chunk 的方法：在 chunk freed（释放）时，会将 chunk 链接到一个单链表上，结构如下：

```
/* We overlay this structure on the user-data portion of a chunk when
the chunk is stored in the per-thread cache.  */
typedef struct tcache_entry
{
    struct tcache_entry *next;
    /* This field exists to detect double frees.  */
    struct tcache_perthread_struct *key;  // 每个线程都会有一个
} tcache_entry;

/* There is one of these for each thread, which contains the
per-thread cache (hence "tcache_perthread_struct").  Keeping
overall size low is mildly important.  Note that COUNTS and ENTRIES
are redundant (we could have just counted the linked list each
time), this is for performance reasons.
*/
typedef struct tcache_perthread_struct
{
    uint16_t counts[TCACHE_MAX_BINS];
    tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

- 接下来介绍 tcache 初始化，其结构也是使用分配区，每个线程都有一个独立的分配区，通过 __thread 进行声明。

```
// __thread 标识线程独立存储，各个线程互不影响，这也是每个线程都会有一个 tcache_perthread_struct 的原因，在第一次调用时初始化
static __thread tcache_perthread_struct *tcache = NULL;

/*
arena_get() 获取一个 arena 并锁定相应的互斥锁。
首先，尝试该线程最后成功锁定的一个。 （这是常见的情况，并使用宏处理以提高速度。）
然后，循环一次遍历 arenas 的循环链表。如果没有 arena 随时可用，创建一个新的。
在后一种情况下，“大小”只是关于在新的 arena 上立即需要多少内存。
 */

#define arena_get(ptr, size) do { \
      ptr = thread_arena;      \
      arena_lock (ptr, size);      \
  } while (0)

........

// tcache 初始化
static void
tcache_init(void)
{
mstate ar_ptr;
void *victim = 0;
const size_t bytes = sizeof (tcache_perthread_struct);

if (tcache_shutting_down)
```

```
    return;
// 获得分配区
arena_get (ar_ptr, bytes);

// 分配 tcache_perthread_struct 内存
victim = _int_malloc (ar_ptr, bytes); // _int_malloc 接口便是 ptmalloc 分配内存的具体实现了，会在后边章节具体解析
if (!victim && ar_ptr != NULL) //分配失败会进行重试
    {
    ar_ptr = arena_get_retry (ar_ptr, bytes);
    victim = _int_malloc (ar_ptr, bytes);
    }

if (ar_ptr != NULL)
    __libc_lock_unlock (ar_ptr->mutex);

/* In a low memory situation, we may not be able to allocate memory
    - in which case, we just keep trying later.  However, we
    typically do this very early, so either there is sufficient
    memory, or there isn't enough memory to do non-trivial
    allocations anyway.  */
if (victim)
    {
    tcache = (tcache_perthread_struct *) victim;
    memset (tcache, 0, sizeof (tcache_perthread_struct));
    }

}
```

Tcache 之参数限制

```
void *
__libc_malloc (size_t bytes)
{
    .....
    DIAG_PUSH_NEEDS_COMMENT; // 推送诊断信息
    if (tc_idx < mp_.tcache_bins
        && tcache
        && tcache->counts[tc_idx] > 0)
        {
        return tcache_get (tc_idx);
        }
    DIAG_POP_NEEDS_COMMENT;
    ....

}
```

首先回答上文中没有说明的一个问题，单链表如何管理长度的。可以看出是 mp_ 的成员进行限制，在进行存取时都会进行判断：

> 每个分配区是 struct malloc_state 的一个实例，ptmalloc 使用 malloc_state 来管理分配区，而参数管理使用 struct malloc_par, 全局拥有一个唯一的 malloc_par 实例。

```
/* There is only one instance of the malloc parameters.  */
static struct malloc_par mp_ =
{
  .top_pad = DEFAULT_TOP_PAD,
  .n_mmaps_max = DEFAULT_MMAP_MAX,
  .mmap_threshold = DEFAULT_MMAP_THRESHOLD,
  .trim_threshold = DEFAULT_TRIM_THRESHOLD,
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
  .arena_test = NARENAS_FROM_NCORES (1)
#if USE_TCACHE
  ,
  .tcache_count = TCACHE_FILL_COUNT, // 每个 tcache bin 可以容纳的 chunk 数量
  .tcache_bins = TCACHE_MAX_BINS,  // tcache bins 的数量。
  .tcache_max_bytes = tidx2usize (TCACHE_MAX_BINS-1), // 最大的 tcache bins 容量大小，计算 tidx2usize 宏，32 位下是 512，64 位下是 1024
  .tcache_unsorted_limit = 0 /* No limit.  */
#endif
};
/*
static struct malloc_par mp_ = {
    .top_pad = 131072,
    .n_mmaps_max = (65536),
    .mmap_threshold = (128 * 1024),
    .trim_threshold = (128 * 1024),
    .arena_test = ((1) * (sizeof(long) == 4 ? 2 : 8)),
    .tcache_count = 7,
    .tcache_bins = 64,
    .tcache_max_bytes =
        (((size_t)64 - 1) * (2 * (sizeof(size_t)) < __alignof(long double)
                             ? __alignof(long double)
                             : 2 * (sizeof(size_t))) +
         (unsigned long)((
             ((__builtin_offsetof(struct malloc_chunk, fd_nextsize)) +
              ((2 * (sizeof(size_t)) < __alignof(long double)
                    ? __alignof(long double)
                    : 2 * (sizeof(size_t))) -
               1)) &
             ~((2 * (sizeof(size_t)) < __alignof(long double)
                    ? __alignof(long double)
                    : 2 * (sizeof(size_t))) -
               1))) -
         (sizeof(size_t))),
    .tcache_unsorted_limit = 0}
*/
// 还有一些其他的参数都是在初始化过程中会进行赋值
struct malloc_par
{
  /* Tunable parameters */

  INTERNAL_SIZE_T arena_max;

  /* Memory map support */
  int n_mmaps;
  int max_n_mmaps;
  /* the mmap_threshold is dynamic, until the user sets
     it manually, at which point we need to disable any
     dynamic behavior. */
  int no_dyn_threshold;
```

```
  /* Statistics */
  INTERNAL_SIZE_T mmapped_mem;
  INTERNAL_SIZE_T max_mmapped_mem;

  /* First address handed out by MORECORE/sbrk.  */
  char *sbrk_base;

};
```

了解完结构体可以清楚的知道，每条线程都有自己的分配区，每个 tcache 分配区共有 64 条单链表 bins（tcache_bins），每条单链表最多有 7 个节点（tcache_count），每条 tcache 的 chunk 的大小在 32 位系统上是以 8 Bytes 递增，最大 chunk 为 512。在 64 位系统上是以 16 Bytes 递增，最大 chunk 为 1024（tcache_max_bytes）。

同时也明确了 csize2tidx 查找的就是对应大小的 bins，然后查找其上以单链表形式组织的 chunk 结构内存。

Tcache 之获取与存入

tcache 当通过条件判断后，确保对应链表存在节点后会通过 `tcache_get` 接口获取 chunk 中 user mem 内存。get 与 put 接口源码如下：

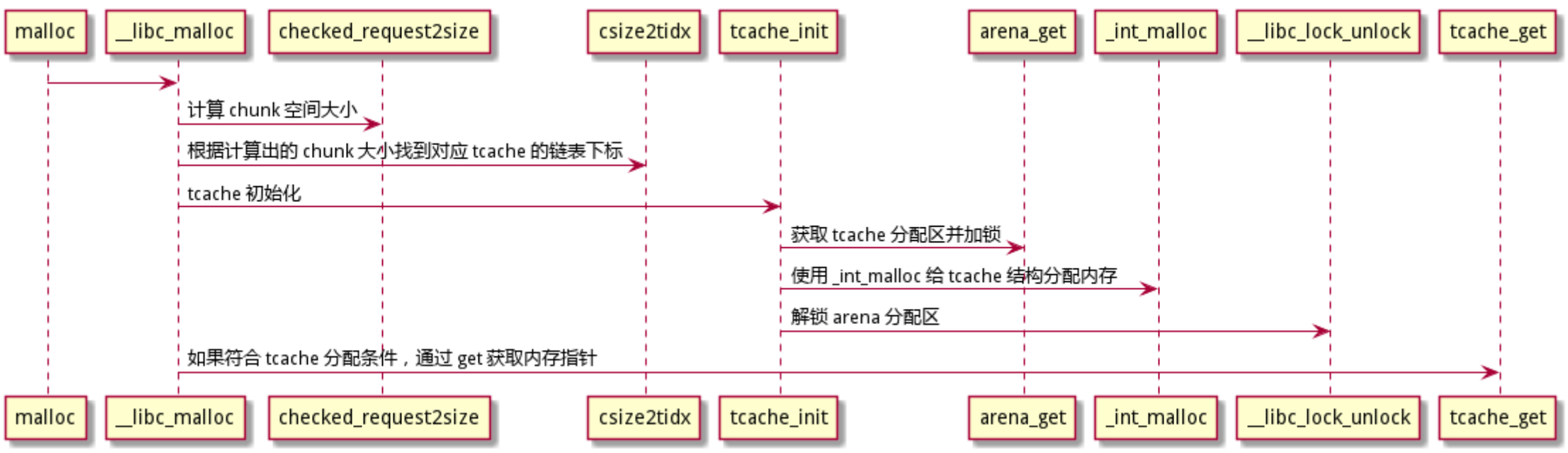
```
struct malloc_chunk;
typedef struct malloc_chunk* mchunkptr;

#define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

```
/* Caller must ensure that we know tc_idx is valid and there's room
   for more chunks.  */
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);

  /* Mark this chunk as "in the tcache" so the test in _int_free will
     detect a double free.  */
  e->key = tcache;

  e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]); // PROTECT_PTR 作用见下文
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}

/* 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)
{
  tcache_entry *e = tcache->entries[tc_idx];
  if (__glibc_unlikely (!aligned_OK (e)))
    malloc_printerr ("malloc(): unaligned tcache chunk detected");
  tcache->entries[tc_idx] = REVEAL_PTR (e->next);
  --(tcache->counts[tc_idx]);
  e->key = NULL;
  return (void *) e;
}
```

- 先分析 get 接口，传入对应链表的标号，取出链表头部的内存节点，count 计数 -1，返回内存指针。
- put 借口传入 chunk，通过 chunk2mem 计算出 chunk 中 user mem 地址指针，存入链表中。

总结

至此，我们分析了 tcache 中管理 chunk 内存的方式，以及一些申请内存的细节。



- 疑问：为什么 chunk2mem 中偏移量为 2×SIZE_SZ 与申请的内存容量之增加了 SIZE_SZ 对不上。
  - 因为在申请时不需要考虑 pre_size，因为每个 chunk 都是可以复用下一个 chunk 的头 pre_size 内存。而 chunk2mem 偏移则是考虑当前 chunk 结构。
- 疑问：tcache 链表中 chunk 节点是怎么来的？
  - 这个问题简单解释就是 free 时加入的，细节请关注后续 free 解析文档。

实践

> 实例代码：tcache.cpp

1. 判断是否支持 tcache，参考 pwndbg 实现：

```
test : gdb — Konsole
新建标签页(N)  左右拆分视图  上下拆分视图  配置键盘快捷键(H)  截屏(P)  复制(C)  粘贴  查找(F)  复制(C)  粘贴  查找(F)  清除回滚
pwndbg> p &mp_
$40 = (malloc_par *) 0x7ffff7ed0280 <mp_>
pwndbg> p mp_
$41 = {
  trim_threshold = 131072,
  top_pad = 131072,
  mmap_threshold = 131072,
  arena_test = 8,
  arena_max = 0,
  n_mmaps = 0,
  n_mmaps_max = 65536,
  max_n_mmaps = 0,
  no_dyn_threshold = 0,
  mmapped_mem = 0,
  max_mmapped_mem = 0,
  sbrk_base = 0x555555559000 "",
  tcache_bins = 64,
  tcache_max_bytes = 1032,
  tcache_count = 7,
  tcache_unsorted_limit = 0
}
pwndbg> p (tcache_perthread_struct)*(0x555555559000 + 0x10)
$42 = {
  counts = {7 <repeats 64 times>},
  entries = {0x55555556b380, 0x55555556b4e0, 0x55555556b6c0, 0x55555556b920, 0x55555556bc00, 0x55555556bf60, 0x55555556c340, 0x55555556c7a0, 0x55555556cc80, 0x55555556d1e0, 0x55555556d7c0,
0x55555556de20, 0x55555556e500, 0x55555556ec60, 0x55555556f440, 0x55555556fca0, 0x555555570580, 0x555555570ee0, 0x5555555718c0, 0x555555572320, 0x555555572e00, 0x555555573960, 0x55555557454
0, 0x5555555751a0, 0x555555575e80, 0x555555576be0, 0x5555555779c0, 0x555555578820, 0x555555579700, 0x55555557a660, 0x55555557b640, 0x55555557c6a0, 0x55555557d780, 0x55555557e8e0, 0x55555557
fac0, 0x555555580d20, 0x555555582000, 0x555555583360, 0x555555584740, 0x555555585ba0, 0x555555587080, 0x5555555885e0, 0x555555589bc0, 0x55555558b220, 0x55555558c900, 0x55555558e060, 0x55555
558f840, 0x5555555910a0, 0x555555592980, 0x5555555942e0, 0x555555595cc0, 0x555555597720, 0x555555599200, 0x55555559ad60, 0x55555559c940, 0x55555559e5a0, 0x5555555a0280, 0x5555555a1fe0, 0x55
55555a3dc0, 0x5555555a5c20, 0x5555555a7b00, 0x5555555a9a60, 0x5555555aba40, 0x5555555adaa0}
}
pwndbg> tcache
{
  counts = {7 <repeats 64 times>},
  entries = {0x55555556b380, 0x55555556b4e0, 0x55555556b6c0, 0x55555556b920, 0x55555556bc00, 0x55555556bf60, 0x55555556c340, 0x55555556c7a0, 0x55555556cc80, 0x55555556d1e0, 0x55555556d7c0,
0x55555556de20, 0x55555556e500, 0x55555556ec60, 0x55555556f440, 0x55555556fca0, 0x555555570580, 0x555555570ee0, 0x5555555718c0, 0x555555572320, 0x555555572e00, 0x555555573960, 0x55555557454
0, 0x5555555751a0, 0x555555575e80, 0x555555576be0, 0x5555555779c0, 0x555555578820, 0x555555579700, 0x55555557a660, 0x55555557b640, 0x55555557c6a0, 0x55555557d780, 0x55555557e8e0, 0x55555557
fac0, 0x555555580d20, 0x555555582000, 0x555555583360, 0x555555584740, 0x555555585ba0, 0x555555587080, 0x5555555885e0, 0x555555589bc0, 0x55555558b220, 0x55555558c900, 0x55555558e060, 0x55555
558f840, 0x5555555910a0, 0x555555592980, 0x5555555942e0, 0x555555595cc0, 0x555555597720, 0x555555599200, 0x55555559ad60, 0x55555559c940, 0x55555559e5a0, 0x5555555a0280, 0x5555555a1fe0, 0x55
55555a3dc0, 0x5555555a5c20, 0x5555555a7b00, 0x5555555a9a60, 0x5555555aba40, 0x5555555adaa0}
}
pwndbg>
```

> 为什么 tcache 要 `sbrk_base + 0x10`，由于 tcache 在 malloc 初始化过程中在最前边，也是第一个通过 _int_malloc 调用 brk 向系统申请内存以及更新 `mp_` 中的 `sbrk_base` 值， 同时由于 _int_malloc 中的内存以 chunk 结构进行申请管理，在返回时会调用 chunk2mem 跳过 chunk 头部，也就是 2 * size_t = 16 字节的大小，因此需要 + 0x10。

2. 获取 tcache 链表：

```
test : gdb — Konsole
新建标签页(N)  左右拆分视图  上下拆分视图  配置键盘快捷键(H)  截屏(P)  复制(C)  粘贴  查找(F)  复制(C)  粘贴  查找(F)  清除回滚
pwndbg> p (tcache_perthread_struct)*(0x555555559000 + 0x10)
$24 = {
  counts = {7 <repeats 64 times>},
  entries = {0x55555556b380, 0x55555556b4e0, 0x55555556b6c0, 0x55555556b920, 0x55555556bc00, 0x55555556bf60, 0x55555556c340, 0x55555556c7a0, 0x55555556cc80, 0x55555556d1e0, 0x55555556d7c0,
0x55555556de20, 0x55555556e500, 0x55555556ec60, 0x55555556f440, 0x55555556fca0, 0x555555570580, 0x555555570ee0, 0x5555555718c0, 0x555555572320, 0x555555572e00, 0x555555573960, 0x55555557454
0, 0x5555555751a0, 0x555555575e80, 0x555555576be0, 0x5555555779c0, 0x555555578820, 0x555555579700, 0x55555557a660, 0x55555557b640, 0x55555557c6a0, 0x55555557d780, 0x55555557e8e0, 0x55555557
fac0, 0x555555580d20, 0x555555582000, 0x555555583360, 0x555555584740, 0x555555585ba0, 0x555555587080, 0x5555555885e0, 0x555555589bc0, 0x55555558b220, 0x55555558c900, 0x55555558e060, 0x55555
558f840, 0x5555555910a0, 0x555555592980, 0x5555555942e0, 0x555555595cc0, 0x555555597720, 0x555555599200, 0x55555559ad60, 0x55555559c940, 0x55555559e5a0, 0x5555555a0280, 0x5555555a1fe0, 0x55
55555a3dc0, 0x5555555a5c20, 0x5555555a7b00, 0x5555555a9a60, 0x5555555aba40, 0x5555555adaa0}
}
pwndbg> p ((tcache_entry)(*(0x55555556b380)))
$25 = {
  next = 0x55500003e60b,
  key = 0x555555559010
}
pwndbg> x/64 0x55555556b380 - 0x30
0x55555556b350: 0x0000000000000000      0x0000000000000021
0x55555556b360: 0x000055500003e62b      0x0000555555559010
0x55555556b370: 0x0000000000000000      0x0000000000000021
0x55555556b380: 0x000055500003e60b      0x0000555555559010
0x55555556b390: 0x0000000000000000      0x0000000000000021
0x55555556b3a0: 0x000000055555556b      0x0000000000000000
0x55555556b3b0: 0x0000000000000000      0x0000000000000031
0x55555556b3c0: 0x000000055555556b      0x0000555555559010
0x55555556b3d0: 0x0000000000000000      0x0000000000000000
0x55555556b3e0: 0x0000000000000000      0x0000000000000031
0x55555556b3f0: 0x000055500003e6ab      0x0000555555559010
0x55555556b400: 0x0000000000000000      0x0000000000000000
0x55555556b410: 0x0000000000000000      0x0000000000000031
0x55555556b420: 0x000055500003e69b      0x0000555555559010
0x55555556b430: 0x0000000000000000      0x0000000000000000
0x55555556b440: 0x0000000000000000      0x0000000000000031
0x55555556b450: 0x000055500003e14b      0x0000555555559010
0x55555556b460: 0x0000000000000000      0x0000000000000000
0x55555556b470: 0x0000000000000000      0x0000000000000031
0x55555556b480: 0x000055500003e13b      0x0000555555559010
0x55555556b490: 0x0000000000000000      0x0000000000000000
0x55555556b4a0: 0x0000000000000000      0x0000000000000031
0x55555556b4b0: 0x000055500003e1eb      0x0000555555559010
0x55555556b4c0: 0x0000000000000000      0x0000000000000000
0x55555556b4d0: 0x0000000000000000      0x0000000000000031
0x55555556b4e0: 0x000055500003e1db      0x0000555555559010
0x55555556b4f0: 0x0000000000000000      0x0000000000000000
0x55555556b500: 0x0000000000000000      0x0000000000000031
0x55555556b510: 0x000000055555556b      0x0000000000000000
0x55555556b520: 0x0000000000000000      0x0000000000000000
0x55555556b530: 0x0000000000000000      0x0000000000000041
0x55555556b540: 0x000000055555556b      0x0000555555559010
pwndbg> p ((tcache_entry)(*(0x55555556b380)))
```

- 可以看出，其中 next 是一个异常值，其目的是为了安全，不过可以通过直接查看内存地址偏移来找到其他 tcache bins chunk 地址，或者通过 glibc 中的算法进行恢复：

```
/* Safe-Linking:
  Use randomness from ASLR (mmap_base) to protect single-linked lists
  of Fast-Bins and TCache.  That is, mask the "next" pointers of the
  lists' chunks, and also perform allocation alignment checks on them.
  This mechanism reduces the risk of pointer hijacking, as was done with
  Safe-Unlinking in the double-linked lists of Small-Bins.
  It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
  larger pages provide less entropy, although the pointer mangling
  still works.  */
/*
安全链接：
  使用来自 ASLR (mmap_base) 的随机性来保护单链表 Fast-Bins 和 TCache。
  也就是说，屏蔽“下一个”指针列表的块，并对它们执行分配对齐检查。
  这种机制降低了指针劫持的风险，就像在 Small-Bins 双链表中的安全解除链接。
  它假定最小页面大小为 4096 字节（12 位）。系统与较大的页面提供较少的熵，尽管指针重整仍然有效。
*/
```

```
#define PROTECT_PTR(pos, ptr) \
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```