glibc2.31 -- free 源码注释

发表于 2020-10-16 16:01 阅读次数：340 评论次数：0

BINARY EXPLOIT GLIBC GLIBC BINARY EXPLOIT

1 __lib_free

```
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  // 检查 malloc.h 中有没有定义 __free_hook 函数
  // 定义的话则执行它
  void (*hook) (void *, const void *)
```

```
  = atomic_forced_read (__free_hook);
if (__builtin_expect (hook != NULL, 0))
  {
    (*hook)(mem, RETURN_ADDRESS (0));
    return;
  }

if (mem == 0)                              /* free(0) has no effect */
  return;

// 传进来的指针 mem 是指向 data 字段的
// 只要减去 size 和 prev_size 字段的大小就能得到 chunk 的地址（不清楚的话可以去看一下在使用的
p = mem2chunk (mem);

// 检查 chunk 是不是通过 mmap 分配的
if (chunk_is_mmapped (p))                       /* release mmapped memory. */
  {
    /* See if the dynamic brk/mmap threshold needs adjusting.
       Dumped fake mmapped chunks do not affect the threshold.  */
    if (!mp_.no_dyn_threshold
        && chunksize_nomask (p) > mp_.mmap_threshold
        && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
        && !DUMPED_MAIN_ARENA_CHUNK (p))
      {
        mp_.mmap_threshold = chunksize (p);
        mp_.trim_threshold = 2 * mp_.mmap_threshold;
        LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                    mp_.mmap_threshold, mp_.trim_threshold);
      }
    munmap_chunk (p);
    return;
```

```
    }

  // 如果 chunk 不是通过 mmap 分配的（能执行到这里说明没有陷入上面的那个 if）

  // 检查 tcache 有没有初始化，没有初始化就先进行初始化
  MAYBE_INIT_TCACHE ();

  // 找到 chunk 所处的 arena
  ar_ptr = arena_for_chunk (p);

  // 执行 free 操作
  _int_free (ar_ptr, p, 0);
}
```

2 _int_free

```
static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr *fb;             /* associated fastbin */
  mchunkptr nextchunk;         /* next contiguous chunk */
  INTERNAL_SIZE_T nextsize;    /* its size */
  int nextinuse;               /* true if nextchunk is used */
  INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
  mchunkptr bck;               /* misc temp for linking */
  mchunkptr fwd;               /* misc temp for linking */

  // 取出 chunk 的 size 字段，屏蔽掉 size 字段的低 3 比特（标志位）就是 chunk 真正的 size
  size = chunksize (p);
```

```
  /* Little security check which won't hurt performance: the
     allocator never wrapps around at the end of the address space.
     Therefore we can exclude some size values which might appear
     here by accident or by "design" from some intruder.  */
  if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
      || __builtin_expect (misaligned_chunk (p), 0))
    malloc_printerr ("free(): invalid pointer");
  /* We know that each chunk is at least MINSIZE bytes in size or a
     multiple of MALLOC_ALIGNMENT.  */
  if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
    malloc_printerr ("free(): invalid size");

  check_inuse_chunk(av, p);

#if USE_TCACHE
// 开启 tcache 的话
  {
    // 获取 free 的 chunk 对应于 tcache 的那一条链表的下标
    size_t tc_idx = csize2tidx (size);
    // 如果 tcache 已经初始化，并且 这个 chunk 符合 tcache 的 chunk 的大小
    // （tcache_bins 等于 64，tcache 有 64 个链表对应 64 种大小的 chunk，如果，计算出来的 index
    if (tcache != NULL && tc_idx < mp_.tcache_bins)
      {
        /* Check to see if it's already in the tcache.  */
  // 把 p 转换成 tcache 节点（可以看到的是不同于 bins ， tcache 中的每一个 chunk 都是指向 mem 的，
        tcache_entry *e = (tcache_entry *) chunk2mem (p);

        /* This test succeeds on double free.  However, we don't 100%
           trust it (it also matches random payload data at a 1 in
           2^<size_t> chance), so verify it's not an unlikely
```

```
            coincidence before aborting.  */
          if (__glibc_unlikely (e->key == tcache))
            {
              tcache_entry *tmp;
              LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
              for (tmp = tcache->entries[tc_idx];
                   tmp;
                   tmp = tmp->next)
                if (tmp == e)
                  malloc_printerr ("free(): double free detected in tcache 2");
              /* If we get here, it was a coincidence.  We've wasted a
                 few cycles, but don't abort.  */
            }

          if (tcache->counts[tc_idx] < mp_.tcache_count)
            {
              tcache_put (p, tc_idx);
              return;
            }
        }
    }
#endif

  /*
    If eligible, place chunk on a fastbin so it can be found
    and used quickly in malloc.
  */

  // 如果 tcache 已经满了， p 的大小又符合 fastbin 的大小
  if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))
```

```
#if TRIM_FASTBINS
      /*
        If TRIM_FASTBINS set, don't place chunks
        bordering top into fastbins
      */
      // p 的下一个 chunk 不是 top chunk
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    // 检查 p 的大小是否合法
    // 2 * SIZE_SZ 是最小 chunk 的大小，chunk 小于这个数的说明，这个 chunk 被破坏了
    if (__builtin_expect (chunksize_nomask (chunk_at_offset (p, size))
                          <= 2 * SIZE_SZ, 0)
    //  检查 p 的 size 是不是大于 system_mem（ chunk 的大小不可能大于 system_mem）
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                             >= av->system_mem, 0))
      {
        bool fail = true;
        /* We might not have a lock at this point and concurrent modifications
           of system_mem might result in a false positive.  Redo the test after
           getting the lock.  */
  // 如果当前分配区没有上锁
        if (!have_lock)
          {
      // 给分配区上锁
            __libc_lock_lock (av->mutex);

      // 再次检查 p 的大小
            fail = (chunksize_nomask (chunk_at_offset (p, size)) <= 2 * SIZE_SZ
                    || chunksize (chunk_at_offset (p, size)) >= av->system_mem);
```

```
      // 解锁分配区
            __libc_lock_unlock (av->mutex);
          }

        if (fail)
          malloc_printerr ("free(): invalid next size (fast)");
      }

    // 使用 perturb_byte 填充 chunk 的 mem 字段
    free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

    // 先设定分配区的 have_fastchunks 字段（表示分配区的 fastbin 不为空）
    atomic_store_relaxed (&av->have_fastchunks, true);

    // 获取 p 对应的 fastbin 的索引
    unsigned int idx = fastbin_index(size);

    // 获取对应 p 的 fastbin 的链表的第一个 chunk（通过索引 idx）
    fb = &fastbin (av, idx);

    /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
    mchunkptr old = *fb, old2;

    // 如果是单线程的话
    if (SINGLE_THREAD_P)
      {
        /* Check that the top of the bin is not the record we are going to
           add (i.e., double free).  */
// old 是存的 fb，fb 是 fastbin 链表里面的第一个 chunk
// 如果 old 和释放的 p 相等，那么说明 p 之前已经 free 过了
// 触发 double free 的提示
```

```
      if (__builtin_expect (old == p, 0))
        malloc_printerr ("double free or corruption (fasttop)");

// 把 p 链到 fastbin
// fastbin 只会用到 fd 字段
// fd 字段会指向同一条 bin 里面的下一个 chunk
// 只要把 p 的 fd 设置成现在的 fastbin 的第一个 chunk
      p->fd = old;
// 再把 fastbin 的第一个 chunk 设置成 p，就完成了插入
      *fb = p;
    }

  // 如果不是单线程（其实区别就是在多线程时操作分配区要上锁，防止竞争条件）
  else
    do
      {
        /* Check that the top of the bin is not the record we are going to
           add (i.e., double free).  */
        if (__builtin_expect (old == p, 0))
          malloc_printerr ("double free or corruption (fasttop)");
        p->fd = old2 = old;
      }
    while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2))
           != old2);

  /* Check that size of fastbin chunk at the top is the same as
     size of the chunk that we are adding.  We can dereference OLD
     only if we have the lock, otherwise it might have already been
     allocated again.  */
  if (have_lock && old != NULL
      && __builtin_expect (fastbin_index (chunksize (old)) != idx, 0))
```

```
    malloc_printerr ("invalid fastbin entry (free)");
}

/*
  Consolidate other non-mmapped chunks as they arrive.
*/

// 如果 p 不属于 fastbin
// 并且不是通过 mmap 分配来的
else if (!chunk_is_mmapped(p)) {

  /* If we're single-threaded, don't lock the arena.  */
  // 如果是单线程，就不要给 分配区 上锁
  if (SINGLE_THREAD_P)
    have_lock = true;

  if (!have_lock)
    __libc_lock_lock (av->mutex);

  // 获取物理内存上（说物理内存也许不严谨）的下一个 chunk
  nextchunk = chunk_at_offset(p, size);

  /* Lightweight tests: check whether the block is already the
     top block.  */
  // 如果 p 是 top chunk
  if (__glibc_unlikely (p == av->top))
    malloc_printerr ("double free or corruption (top)");
  /* Or whether the next chunk is beyond the boundaries of the arena.  */
  if (__builtin_expect (contiguous (av)
                        && (char *) nextchunk
                        >= ((char *) av->top + chunksize(av->top)), 0))
```

```
    malloc_printerr ("double free or corruption (out)");
/* Or whether the block is actually not marked used.  */
// 检查要释放的 chunk 是否在是在使用（这里是通过检查 nextchunk 的 inuse 位）
if (__glibc_unlikely (!prev_inuse(nextchunk)))
  malloc_printerr ("double free or corruption (!prev)");

// 获取 下一个 chunk 的大小
nextsize = chunksize(nextchunk);

// 日常检查大小
if (__builtin_expect (chunksize_nomask (nextchunk) <= 2 * SIZE_SZ, 0)
    || __builtin_expect (nextsize >= av->system_mem, 0))
  malloc_printerr ("free(): invalid next size (normal)");

free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

/* consolidate backward */
// 检查物理内存上的上一个 chunk 是否在是在使用（通过检查 p 的 inuse）
// 如果上一个 chunk 是 释放 状态，就会触发合并
// 两个相邻的 chunk 如果都处于 释放 状态，则会触发合并
// （当然，这里所谓的 释放 指的是对应 chunk 的下一个 chunk 的 inuse 标志位为 0）
// 如果这两个 chunk 都是属于 fastbin 的话就不会触发合并（如果你从 _int_free 的第 0 行看到这里
// 因为 fastbin 在释放 chunk 后不会取消 inuse 标志位
if (!prev_inuse(p)) {
  // 取出 上一个 chunk 的大小
  prevsize = prev_size (p);
  // 把两个 chunk 的大小相加得到合并后的 chunk 的大小
  size += prevsize;
  // 通过把 p 的地址减去上一个 chunk 的大小得到新 chunk 的地址
  // 注意，这里为什么是 -((long)prevsize)， 因为 p 是位于高地址，所以。。。
  p = chunk_at_offset(p, -((long) prevsize));
```

```
      if (__glibc_unlikely (chunksize(p) != prevsize))
        malloc_printerr ("corrupted size vs. prev_size while consolidating");
      // 对 p 进行 unlink 操作
      unlink_chunk (av, p);
    }

    // 如果下一个 chunk 不是 top chunk 的话
    if (nextchunk != av->top) {
      /* get and clear inuse bit */
      // nextinuse 是标志 nextchunk 是否处于 释放 状态的标志
      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

      /* consolidate forward */
      // 如果 nextchunk 也是 释放 状态的话，还会再次触发 unlink，把 p 和 nextchunk 合并
      if (!nextinuse) {
        unlink_chunk (av, nextchunk);
        size += nextsize;
      } else
  // 把 nextchunk 的 inuse 位置 0
        clear_inuse_bit_at_offset(nextchunk, 0);

      /*
        Place the chunk in unsorted chunk list. Chunks are
        not placed into regular bins until after they have
        been given one chance to be used in malloc.
      */

      // 获取 unsortedbin 的链表的链表头
      bck = unsorted_chunks(av);
      // 获取 bck 的下一个 chunk
      fwd = bck->fd;
```

```
  if (__glibc_unlikely (fwd->bk != bck))
    malloc_printerr ("free(): corrupted unsorted chunks");
  // 把 p 插入 unsortedbin
  p->fd = fwd;
  p->bk = bck;

  // 如果 p 不是属于 smallbin（属于 largebin）
  if (!in_smallbin_range(size))
    {
// 初始化 largebin 独有的两个字段
      p->fd_nextsize = NULL;
      p->bk_nextsize = NULL;
    }
  bck->fd = p;
  fwd->bk = p;

  set_head(p, size | PREV_INUSE);
  set_foot(p, size);

  check_free_chunk(av, p);
}

/*
  If the chunk borders the current high end of memory,
  consolidate into top
*/

else {
  // 把 p 的 size 加上 nextchunk 的 size 得到合并后的 chunk 的 size
  size += nextsize;
  set_head(p, size | PREV_INUSE);
```

```
      av->top = p;
      check_chunk(av, p);
    }

    /*
      If freeing a large space, consolidate possibly-surrounding
      chunks. Then, if the total unused topmost memory exceeds trim
      threshold, ask malloc_trim to reduce top.

      Unless max_fast is 0, we don't know if there are fastbins
      bordering top, so we cannot tell for sure whether threshold
      has been reached unless fastbins are consolidated.  But we
      don't want to consolidate on each free.  As a compromise,
      consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
      is reached.
    */

    if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
      if (atomic_load_relaxed (&av->have_fastchunks))
        malloc_consolidate(av); // 合并回收所有 fastbin

      if (av == &main_arena) {
#ifndef MORECORE_CANNOT_TRIM
        if ((unsigned long)(chunksize(av->top)) >=
            (unsigned long)(mp_.trim_threshold))
          systrim(mp_.top_pad, av);
#endif
      } else {
        /* Always try heap_trim(), even if the top chunk is not
           large, because the corresponding heap might go away.  */
        heap_info *heap = heap_for_ptr(top(av));
```

```
        assert(heap->ar_ptr == av);
        heap_trim(heap, mp_.top_pad);
      }
    }

    if (!have_lock)
      __libc_lock_unlock (av->mutex);
  }
  /*
    If the chunk was allocated via mmap, release via munmap().
  */

  else {
    // 如果 chunk 是通过 mmap 分配的，则使用 munmap 来释放
    munmap_chunk (p);
  }
}
```

unlink_chunk

```
/* Take a chunk off a bin list.  */
static void
unlink_chunk (mstate av, mchunkptr p)
{
  // 检查 bin 是否被破坏
  // 正常情况下 chunk p 的下一个 chunk 的 prev_size 肯定还是等于 chunk p 的 size
  if (chunksize (p) != prev_size (next_chunk (p)))
    malloc_printerr ("corrupted size vs. prev_size");
```

```
  // 把 fd，bk 取出来
  mchunkptr fd = p->fd;
  mchunkptr bk = p->bk;

  // 绕一遍链表，确定 chunk 的 fd和bk 没有被破坏
  if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
    malloc_printerr ("corrupted double-linked list");

  /*
        bk                              p                               fd
+------------------+<--+  +------>+------------------+<--+   +----> +------------------+
|    prev_size     |   |  |       |    prev_size     |   |   |      |    prev_size     |
+------------------+   |  |       +------------------+   |   |      +------------------+
|      size        |   |  |       |      size        |   |   |      |      size        |
+------------------+   |  |       +------------------+   |   |      +------------------+
|       fd         +------+       |       fd         +-------+      |       fd         |
+------------------+   |          +------------------+   |          +------------------+
|       bk         |   +----------+       bk         |   +----------+       bk         |
+------------------+              +------------------+              +------------------+
|                  |              |                  |              |                  |
|      mem         |              |      mem         |              |      mem         |
|                  |              |                  |              |                  |
|                  |              |                  |              |                  |
+------------------+              +------------------+              +------------------+
   */
  // 把下一个 chunk 的 bk 字段设置成 p 的 bk
  /*
             +---------------------------------------+
             |                                       |
        bk   v                         p            |                fd
+------------+-----+<--+  +------>+------------------+  |    +----> +------------------+
```

```
|    prev_size     |   |  |       |    prev_size     |  |    |      |    prev_size     |
+------------------+   |  |       +------------------+  |    |      +------------------+
|      size        |   |  |       |      size        |  |    |      |      size        |
+------------------+   |  |       +------------------+  |    |      +------------------+
|       fd         +------+       |       fd         +-------+      |       fd         |
+------------------+   |          +------------------+  |           +------------------+
|       bk         |   +----------+       bk         |  +-----------+       bk         |
+------------------+              +------------------+              +------------------+
|                  |              |                  |              |                  |
|      mem         |              |      mem         |              |      mem         |
|                  |              |                  |              |                  |
|                  |              |                  |              |                  |
+------------------+              +------------------+              +------------------+
*/
  fd->bk = bk;
  // 把上一个 chunk 的 fd 字段设置成 p 的 fd
  /*
                           +-----------------------------------------+
                           |                                         |
                +----------------------------------------+           |
                |          |                             |           v
        bk      v          |           p                 |           fd
+------------+-----+<--+   |   +------------------+  |   +----> +------------------+
|    prev_size     |   |   |   |    prev_size     |  |   |      |    prev_size     |
+------------------+   |   |   +------------------+  |   |      +------------------+
|      size        |   |   |   |      size        |  |   |      |      size        |
+------------------+   |   |   +------------------+  |   |      +------------------+
|       fd         +---------+   |       fd         +-------+      |       fd         |
+------------------+   |       +------------------+  |          +------------------+
|       bk         |   +----------+       bk         |  +-----------+       bk         |
+------------------+              +------------------+              +------------------+
```

```
|                  |              |                  |              |                  |
|      mem         |              |      mem         |              |      mem         |
|                  |              |                  |              |                  |
|                  |              |                  |              |                  |
+------------------+              +------------------+              +------------------+
   */
  bk->fd = fd;


  // 如果 p 不是位于 smallbin，并且 fd_nextsize 不是 NULL （ fd_nextsize 不是 NULL 说明 p 位于 l
  if (!in_smallbin_range (chunksize_nomask (p)) && p->fd_nextsize != NULL)
    {
      // 日常检查 bin 链表有没有被破坏
      if (p->fd_nextsize->bk_nextsize != p
          || p->bk_nextsize->fd_nextsize != p)
        malloc_printerr ("corrupted double-linked list (not small)");

      // 其实我没有想明白这个 if 有什么用（理论上来说应该永远都不会陷入这个 if 吧）
      if (fd->fd_nextsize == NULL)
        {
          if (p->fd_nextsize == p)
            fd->fd_nextsize = fd->bk_nextsize = fd;
          else
            {
              fd->fd_nextsize = p->fd_nextsize;
              fd->bk_nextsize = p->bk_nextsize;
              p->fd_nextsize->bk_nextsize = fd;
              p->bk_nextsize->fd_nextsize = fd;
            }
        }
      else
```

```
            {
        // 对 largebin 进行的 unlink 操作
        // 因为 largebin 会用到这两个字段，形成另外的双链表
              p->fd_nextsize->bk_nextsize = p->bk_nextsize;
              p->bk_nextsize->fd_nextsize = p->fd_nextsize;
            }
        }
    }
```

__EOF__

分类: binary exploit , glibc

标签: glibc , binary exploit

« 上一篇： ECB Byte at Time
» 下一篇： how2heap -- glibc 2.23 -- fastbin_dup_consolidate.c 讲解

posted @ 2020-10-16 16:01 scriptk1d 阅读(340) 评论(0) 编辑 收藏 举报

· 2021财年亚马逊净销售额为4698亿美元 同比增长22%
· NASA“毅力号”团队发文称该火星车已逃离鹅卵石的“炼狱”
· Google Doodle涂鸦庆祝2022年北京冬奥会开幕
· Flutter 2.10发布：为构建Windows应用提供稳定支持
» 更多新闻...