MENU

glibc-malloc-unlink

发表于 2020-05-22 22:34 阅读次数：159 评论次数：0

GLIBC GLIBC

unlink 在 _int_free 中的调用是这样的：unlink(av, p, bck, fwd);

unlink 的原理（其实就是链表的操作）

1. 先把传入的块 P 的 fd 和 bk 指针存到参数的 FD ， BK，这两个参数其实是 chunk pointor。
2. 检查 自己的 metadata 有没有损坏，因为 FD 和 BK 在第一步的时候已经被指向 P 的 前一个 chunk 和后一个 chunk，所以 `FD -> bk ; BK->fd;` 必定指向 P，要不然就说明 P 损坏。
3. 第二步的检查如果通过，就直接把 FD 和 BK 连起来，让 FD 的 fd 指向 BK，让 BK 的 fd 指向 FD，这样就能把 P 拿走，这个是 smallbin 范围里面的 chunk 的操作。

4. 在 largebin 中，chunk 还会有 `fd_nextsize` ， `bk_nextsize` 字段，其实检查起来和第二步很相似，后面的操作也是差不多

关于 unlink 的参数

```
mchunkptr top_chunk = top (ar_ptr), p, bck, fwd;
```

传入的 P 是要 unlink 的块，BK 和 FD 是 malloc_chunk 指针

top 宏：

```
#define top(ar_ptr) ((ar_ptr)->top)
```

ar_ptr 是一个指向 malloc_state 结构体的指针：

```
struct malloc_state
{
  /* Serialize access.  */
  mutex_t mutex;

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;

  /* The remainder from the most recent split of a small request */
```

```
  mchunkptr last_remainder;

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;

  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;

  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

触发 unlink 的条件是：当前块的 inuse 位不为 1（也就是当前块的物理位置的前一个块是 free 的，当然位于 fastbin 里面的块除外，因为 fastbin 在 free 时不会把下一块的 inuse bit 置零，fastbin 在一般情况下面不会发生 unlink ）

```
if (!prev_inuse (p)) /* consolidate backward */
{
        p = prev_chunk (p);
        unlink (ar_ptr, p, bck, fwd);
}
```

prev_inuse 宏：

```
#define PREV_INUSE 0x1
/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->size & PREV_INUSE)
```

---

unlink 宏，完整的源码：

```
#define unlink(AV, P, BK, FD) {                                            \
    FD = P->fd; \
    BK = P->bk;                                                              \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                    \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                                                   \
        FD->bk = BK;                                                         \
        BK->fd = FD;                                                         \
        if (!in_smallbin_range (P->size)                                     \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {               \
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)       \
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
              malloc_printerr (check_action,                                 \
                               "corrupted double-linked list (not small)",    \
                               P, AV);                                       \
            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 {                                                  \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;           \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;           \
              }                                                         \
          }                                                             \
      }                                                                 \
  }
```

首先把 P 的下一个块和上一个块分别保存到 FD 和 BK（指针）

```
FD = P->fd;
BK = P->bk;
```

检查块是不是已经损坏：

```
FD->bk != P || BK->fd != P, 0
```

__builtin_expect 是 gcc 的内置函数用来分支预测优化 具体参见 (https://www.cnblogs.com/LubinLew/p/GCC-__builtin_expect.html)， 这里不多说。

glibc 2.23 主要是

```
FD->bk != P
BK->fd != P
```

意思就是：

下一个块的 上一个块 不是它自己的话就说明 chunk 被破坏

上一个块的 下一个块 不是它自己的话就说明 chunk 被破坏

，可以防止有些 heap expolit

检查通过了就

把下一个块的上一个块变成 当前块的 上一个块

把上一个块的下一个块变成 当前块的 下一个块

这样就能把 bin 中的前后两块链接，把当前块从 bin 中取下来

```
FD->bk = BK;
BK->fd = FD;
```

---

in_smallbin_range 宏是检查 chunk 是不是位于 smallbin 里面:

```
#define MALLOC_ALIGNMENT       (2 *SIZE_SZ)
#define NSMALLBINS         64
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)
#define in_smallbin_range(sz)  \
  ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)
```

MALLOC_ALIGNMENT 是用来进行块对齐的，最小 chunk 就是 2×SIZE_SZ。

SMALLBIN_CORRECTION 判断 bin 是不是被破坏， 因为不会有小于 2×SIZE_SZ 的 chunk。

MIN_LARGE_SIZE 是 largebin 的最小 size 在 32 bit 系统上面是 512 Byte， 在 62 bit 系统上面是 1024 Byte。

这样看的话 smallbin 是第一类 bin （当然从访问顺序上讲前面还有 unsortedbin ）只要小于 largebin 的最小 size 就说明这个 bin 是位于 smallbin 中（也许说的不够严谨），这就是 in_smallbin_range 的逻辑

---

smallbin 的 size 计算公式

```
Chunk_size=2 * SIZE_SZ * index
```

SIZE_SZ = 4 Byte(32 bit), 8 Byte(64 bit)

smallbin 共 62 个 bin （没有 0 号 bin），每个 bin 的大小以 SIZE_SZ 为公差的等差数列

由上面的公式可以知道

在 32 bit 系统下面范围是 8 - 504（Byte）

在 64 bit 系统下面范围是 16 - 1008（Byte）

言归正传，回到 unlink

如果 bin 是不是位于 smallbin 里面的话（ !in_smallbin_range (P->size) ）也就是位于 largebin 中，smallbin 的话是没有 fd_nextsize 和 bk_nextsize 的，所以 unlink 的话直接操作 fd 和 bk 指针就好了，但是 要是是 largebin 要 unlink 的话还要操作 bk_nextsize 和 fd_nextsize

```
P->fd_nextsize->bk_nextsize != P
P->bk_nextsize->fd_nextsize != P
```

日常检查 P 指向下一块又指向上一个块是不是它自己， P 指向上一块又指向下一个块是不是它自己

这样检查的目的就是 bin 是不是被恶意篡改了 bk_nextsize， fd_nextsize。当发生在 heap overflow（堆溢出）的时候，物理相邻的上一个块可以溢出，覆盖到当前块的 metadata，这个可能会造成任意地址读写。

后面再多说两句，上面提到一点，现在我大概连起来讲一下关于 largebin：

largebin 的 size 在 32 bit 系统下面是大于等于 512 Byte，64 bit 系统下面是大于等于 1024 Byte

在 largebin 中还会把 bin 再分类一次

引用 华庭的 ptmalloc 分析中的话：

> 在 SIZE_SZ 为 4B 的平台上,大于等于 512B 的空闲 chunk,或者,在 SIZE_SZ 为 8B 的平台上,
> 大小大于等于 1024B 的空闲 chunk,
> 由 sorted bins 管理。
> Large bins 一共包括 63 个 bin,

> 每个 bin 中的 chunk 大小不是一个固定公差的等差数列,而是分成 6 组 bin,每组 bin 是一个固定公差的等差数列,每组的 bin 数量依次为 32、16、8、4、2、1,公差依次为 64B、512B、4096B、32768B、262144B 等。

第一个组的 bin 的计算公式:

```
Chunk_size=512 + 64 * index
```

第二个组的 bin 的计算公式:

```
Chunk_size=512 + 64 * 32 + 512 * index
```

第三个组的 bin 的计算公式:

```
Chunk_size=512 + 64 * 32 + 512  * 16 + 4096 * index
```

第四个组的 bin 的计算公式 :

```
Chunk_size=512 + 64 * 32 + 512  * 16 + 4096 *  8 + 32768 * index
```

第五个组的 bin 的计算公式:

```
Chunk_size=512 + 64 * 32 + 512  * 16 + 4096 *  8 + 32768  * 4 + 262144 + index
```

第六个组的 bin 的计算公式:

```
Chunk_size=512 + 64 * 32 + 512  * 16 + 4096 *  8  + 32768 *  4 +  262144 *  2  +  2097152 * inde
```

注:这篇文章参考了 华庭的 ptmalloc 分析

__EOF__

MENU

分类: glibc

标签: glibc

好文要顶 关注我 收藏该文

scriptk1d
关注 - 2
粉丝 - 6
+加关注

0 0

« 上一篇： mmap 从 glibc 到 kernel 的实现
» 下一篇： 一道通过密文明文求解 IV 的密码学题目（crack AES-CBC IV）

posted @ 2020-05-22 22:34 scriptk1d 阅读(159) 评论(0) 编辑 收藏 举报

编辑推荐：
· Three.js 实现2022冬奥主题3D趣味页面
· 技术部如何做复盘——“年终盘点一对一”之创业失败的工程师
· 三探循环依赖 → 记一次线上偶现的循环依赖问题
· 优化.NET 应用程序 CPU 和内存的11 个实践
· 记一次 .NET 某智能交通后台服务 CPU爆高分析

最新新闻：
· 小红书自我反种草，成下一个被遗忘的贴吧？
· 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...

MENU