好好说话之Large Bin Attack

large bin attack这种方法本质上并不难，只是有点绕而已。他和上一篇unsorted bin attack有点类似，都是chunk挂进bin 链表的时候通过完成链表结构连接时发生的问题，只不过large chunk还需要考虑fd_nextsize和bk_nextsize这两个结构。接下来就剩Tcache attack和House系列啦！终于要熬出头了😂
PS:HHKB真香！！！

编写不易，如果能够帮助到你，希望能够点赞收藏加关注哦Thanks♪(·ω·)/

> 往期回顾：
> 好好说话之Unsorted Bin Attack
> 好好说话之Fastbin Attack（4）：Arbitrary Alloc
> （补题）2015 9447 CTF : Search Engine
> 好好说话之Fastbin Attack（3）：Alloc to Stack
> 好好说话之Fastbin Attack（2）：House Of Spirit
> 好好说话之Fastbin Attack（1）：Fastbin Double Free
> 好好说话之Use After Free
> 好好说话之unlink
> …

Large Bin Attack

从标题就可以看出这种攻击手法和Largebin有关，分配largebin有关的chunk，需要经过fast bin、unsort bin、small bin的分配，所以在学习large bin attack之前需要搞清楚fastbin和unsortbin分配的流程

Large Bin

large bin中一共包括63个bin，每个bin中的chunk大小不一致，而是出于一定区间范围内。此外这63个bin被分成了6组，每组bin中的chunk之间的公差一致

Large chunk的微观结构

大于512（1024）字节的chunk称之为large chunk，large bin就是用于管理这些large chunk的

被释放进Large Bin中的chunk，除了以前经常见到的prev_size、size、fd、bk之外，还具有 `fd_nextsize` 和 `bk_nextsize`：

- fd_nextsize，bk_nextsize：只有chunk可先的时候才使用，不过用于较大的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。这样可以避免在寻找合适chunk时挨个遍历

Large Bin的插入顺序

在index相同的情况下：

1、按照大小，从大到小排序（小的链接large bin块）
2、如果大小相同，按照free的时间排序
3、多个大小相同的堆块，只有首堆块的fd_nextsize和bk_nextsize会指向其他堆块，后面的堆块的fd_nextsize和bk_nextsize均为0
4、size最大的chunk的bk_nextsize指向最小的chunk，size最小的chunk的fd_nextsize指向最大的chunk

原理

这里我们选用how2heap中的large bin attack这个例子来解释，这里我做了一些简单的修改，把一些英文段落删掉了，只留下程序执行代码：

```
1  // gcc -g -no-pie hollk.c -o hollk
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7
8      unsigned long stack_var1 = 0;
```

```
9     unsigned long stack_var2 = 0;
10
```

简单的说明一下这个例子的执行流程：首先定义了两个变量stack_var1和stack_var2，并且都赋值为0。接下来打印出两个变量的地址stack_var1_addr和stack_var2_addr以及两个变量中的值。接下来分别申请size为0x330、0x410、0x410三个大堆块p1、p2、p3，以及三个size为0x20的小堆块。然后释放掉p1和p2，并申请了一个size为0xa0的堆块，继续释放p3.接着直接修改p2的size、fd、bk、fd_nextsize、bk_nextsize。接着又申请了一个size为0xa0大小的堆块。再次打印stack_var1、stack_var2的地址和值

查看stack_var1和stack_var2的地址及值

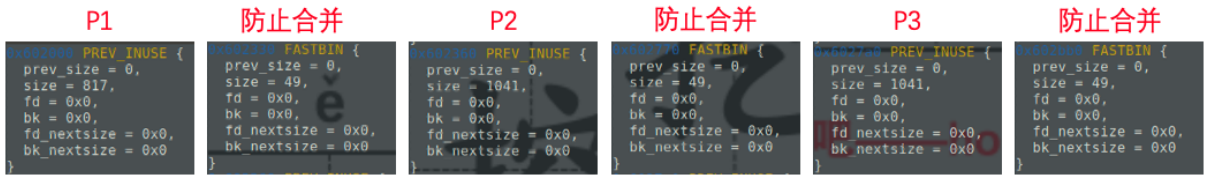
由于我们已经在编译阶段使用-g参数，所以可以直接使用gdb在程序中下行断点。首先在第14行下断点，我们看一下打印出来的stack_var1和stack_var2的地址：

```
stack_var1 (0x7fffffffdf28): 0
stack_var2 (0x7fffffffdf30): 0
```

可以看到stack_var1_addr为 `0x7fffffffdf28` ， stack_var2_addr为 `0x7fffffffdf30` ，两个变量中的值均为 `0`

查看已创建的chunk

接下来在第21行下断点，使程序完成对三个大堆块以及三个小堆块的创建：



可以看到六个chunk已经创建完毕了，P1头指针为 `0x602000` 、P2头指针为 `0x602360` 、P3头指针为 `0x6027a0` ，其中三个size为0x30的chunk是为了放置P1、P2、P3在释放的时候被top_chunk合并，并不是主要的执行流程。

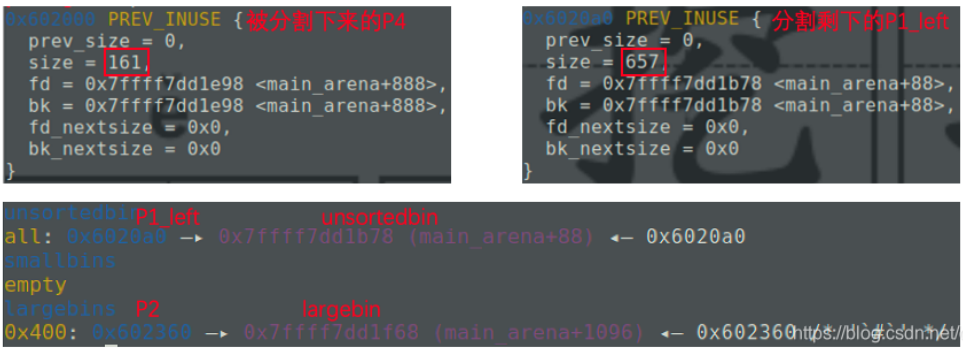
释放P1和P2

接下来我们将断点下载第24行，使程序释放p1和p2，这里需要注意的是释放顺序，先释放的是p1，后释放的是p2：

由于P1的size为0x330，P2的size为0x410，两个chunk的size均超过了fast chunk的最大值，所以在释放P1、P2的时候，两个chunk均进入unsortbin链表中。这里还可以细分，由于P1的size小于0x3F0，所以 `P1最终应该归属为small bin中` 。P2大于0x3F0，所以 `P2最终应该归属为large bin中`

分割P1满足P4的请求

接下来我们将断点在第26行，使程序执行 `void* p4 = malloc(0x90);` 这段代码， 这一步需要注意了！很关键！



`void* p4 = malloc(0x90);` 这段代码其实背后做了很多的事情：

- 从unsorted bin中拿出最后一个chunk(P1)

- 把这个chunk(P1)放进small bin中，并标记这个small bin中有空闲的chunk
- 从unsorted bin中拿出最后一个chunk(P2)（P1被拿走之后P2就作为最后一个chunk了）
- 把这个chunk(P2)放进large bin中，并标记这个large bin有空先的chunk
- 现在unsorted bin中为空，从small bin中的P1中分割出一个小chunk，满足请求的P4，并把剩下的chunk(0x330 - 0xa0后记 `P1_left`)放回unsorted bin中

释放P3

接下来我们将断点端在第28行，使程序释放P3：

```
unsortedP3n         P1_left
all: 0x6027a0 —▸ 0x6020a0 —▸ 0x7ffff7dd1b78 (main_arena+88) ◂— 0x6027a0
smallbins
empty
largebins   P2
0x400: 0x602360 —▸ 0x7ffff7dd1f68 (main_arena+1096) ◂— 0x602360 /* '`#`' */
```

由于P3的size也是 大于`0x3F0` 的，所以首先会被挂进 `unsorted bin` 中进行过度

修改P2结构内容

接下来我们将断点下在第34行，使程序完成对P2内部结构数据的修改，这里附上修改前后的对比图：

未修改前P2

```
pwndbg> x/6gx 0x602360
0x602360:       0x0000000000000000 prev_size          0x0000000000000411 size
0x602370:       0x00007ffff7dd1f68 fd                 0x00007ffff7dd1f68 bk
0x602380:       0x0000000000602360 fd_nextsize        0x0000000000602360 bk_nextsize
```

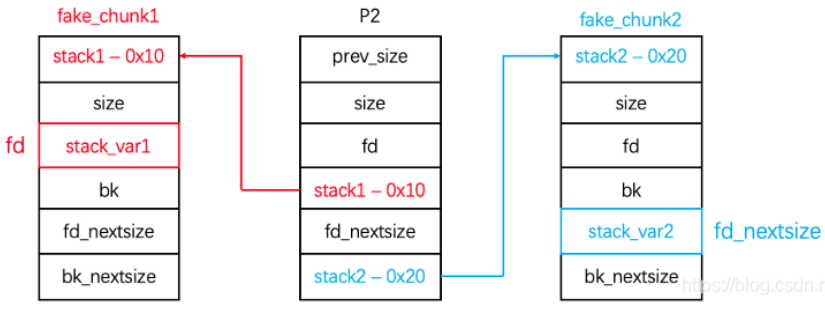
修改后P2

```
pwndbg> x/6gx 0x602360
0x602360:       0x0000000000000000 prev_size          0x00000000000003f1 改为0x3f1
0x602370:       0x0000000000000000 改为0x0            0x00007fffffffdf18 改为stack_var1-0x10
0x602380:       0x0000000000000000 改为0x0            0x00007fffffffdf10 改为stack_var2-0x20
```

https://blog.csdn.net/qq_41202237

可以看到，有五处内容被修改：

- size部分由原来的0x411修改成 `0x3f1` (重点※※※※※)
- fd部分置空（不超过一个地址位长度的数据都可以）
- bk由0x7ffff7dd1f68修改成了 `stack_var1_addr - 0x10(0x7fffffffdf18)`
- fd_nextsize置空（不超过一个地址位长度的数据都可以）
- bk_nextsize修改成 `stack_var2_addr - 0x20(0x7fffffffdf10)`



这里需要注意的是一个chunk的bk指向的是它的后一个被释放chunk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的头指针：

- 也就是说当前P2的bk指向的是一个以 `stack_var1_addr - 0x10` 为头指针的chunk，这里记做fake_chunk1，那么就意味着 `stack_var1_addr` 是作为这个fake_chunk1的 `fd` 指针。那么此时 `P2 --> bk --> fd` 就是stack_var1_addr
- P2的fd_nextsize指向的是一个以 `stack_var2_addr` 为头指针的chunk，这里记做fake_chunk2，那么就意味着 `stack_var2_addr` 是作为这个fake_chunk2的 `fd_nextsize` 指针。那么此时 `P2 --> bk_nextsize --> fd_nextsize` 就是stack_var2_addr

P3挂进large bin 的过程

接下来我们在第36行下断点，使程序执行 `malloc(0x90);` 完成申请size为0xa0的chunk。这一步也很关键，与第一次分割chunk的过程一致，首先从unsorted bin中拿出最后一个chunk(`P1_left` size = 0x290)，并放入small bin中标记该序列的small bin有空闲chunk。再从unsorted bin中拿出最后一个chunk(`P3` size = 0x410)，P3的size是 大于`0x3f0` 的，所以理所应当应该向 `large bin` 中挂

制定P2和P3两个large chunk的fd_nextsize和bk_nextsize，修改stack_var2的内容

从unsorted bin中拿出P3的时候，首先会判断P3应该归属的bin的类型，这里根据size判断出是large bin。由于large chunk的数据结构是带有 `fd_nextsize` 和 `bk_nextsize` 的，且large bin中已经存在了P2这个块，所以首先需要进行比较两个large chunk的大小，并根据大小情况制定两个large chunk的 `fd_nextsize` 、 `bk_nextsize` 、 `fd` 、 `bk` 的指针。在2.23的glibc中的malloc.c文件中，比较的过程如下：

```
3565                while ((unsigned long) size < fwd->size)
3566                  {
3567                    fwd = fwd->fd_nextsize;
3568                    assert ((fwd->size & NON_MAIN_ARENA) == 0);
3569                  }
```

https://blog.csdn.net/qq_41202237

large bin中的chunk如果index相同的情况下，是按照由大到小的顺序排列的。也就是说idex相同的情况下size越小的chunk，越接近large bin。这段代码就是遍历 比较`P3_size < P2_size` 的过程，我们只看while循环中的条件即可，这里的条件是当前从unsorted bin中拿出的chunk的size是否小于large bin中前一个被释放chunk的size，如果小于，则执行while循环中的流程。※※※※※(重点)但由于 `P2的size被我们修改成了0x3f0` (重点)※※※※※，P3的size为0x410，P3_size > P2_size，所以不执行while循环中的代码，直接进入接下里的判断

```
3571                if ((unsigned long) size == (unsigned long) fwd->size)
3572                  /* Always insert in the second position.  */
3573                  fwd = fwd->fd;
```

前一个判断的是P3_size < P2_size的情况，那么接下来判断的就是 `P3_size == P2_size` 的情况，很显然也不是，所以这条if判断也不执行

```
3574                else
3575                  {
3576                    victim->fd_nextsize = fwd;
3577                    victim->bk_nextsize = fwd->bk_nextsize;
3578                    fwd->bk_nextsize = victim;
3579                    victim->bk_nextsize->fd_nextsize = victim;
3580                  }
3581                bck = fwd->bk;
```

victim是P3

fwd是P2

那么就只剩下一种情况了，就是 比较`P3_size > P2_size` 的情况下执行上图中的内容，else中执行的就是将P3插入large bin中并 制定P2和P3两个large chunk的fd_nextsize和bk_nextsize 的过程。这里我们先不着急解释代码，回顾一下前面修改P2结构内容的情况：

- `P2->bk->fd = stack_var1_addr` （P2的fd指向的堆块的fd指向的是stack_var1的地址）
- `P2->bk_nextsize->fd_nextsize = stack_var2_addr` （P2的bk_nextsize指向的堆块的fd_nextsize指向的是stack_var2的地址）

我们将上图的代码根据这个例子的情况翻译一下：

```
1    else
2        {
3            P3->fd_nextsize = P2;  //P3的fd_nextsize要修改成P2的头指针
4            P3->bk_nextsize = P2->bk_nextsize; //P3的bk_nextsize要修改成P2的bk_nextsize指向的地址
5            P2->bk_nextsize = P3;  //P2的bk_nextsize要修改成P3的头指针
6            P3->bk_nextsize->fd_nextsize = P3; //P3的bk_nextsize所指向的堆块的fd_nextsize要修改成P3的头指针
7        }
8    bck = P2->bk; //bck等于P2的bk
```

那么这里就像是做一个二元一次方程组一样，已知条件为：

- `P2->bk_nextsize->fd_nextsize = stack_var2_addr`
- `P3->bk_nextsize = P2->bk_nextsize`
- `P3->bk_nextsize->fd_nextsize = P3`

那么就可以导出结论： `stack_var2的值 = P3头指针` ，所以stack_var2变量中的内容就被修改成了P3的头指针

制定P2和P3两个large chunk的fd和bk，修改stack_var1的内容

在执行完对P3和P2的fd_nextsize和bk_nextsize的制定之后，还需要 `对两个large chunk的fd和bk` 进行制定：

```
3588        mark_bin (av, victim_index);
3589        victim->bk = bck;
3590        victim->fd = fwd;
3591        fwd->bk = victim;
3592        bck->fd = victim;
```

bck是P2的bk指针

victim是P3的头指针

我们将上图的代码根据这个例子的情况翻译一下：

```
mark_bin(av, victim_index);
P3->bk = p2->bk; //P3的bk指针要等于P2的bk指针
P3->fd = P2; //P3的fd指针要等于P2的头指针
P2->bk = P3; //P2的bk指针要等于P3的头指针
P2->bk->fd = P3; //P2的bk指针指向的堆块的fd指针要等于P3的头指针
```

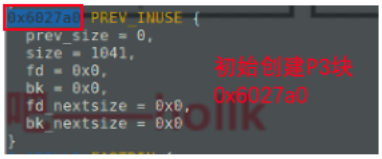
那么这依然还是一个二元一次方程组，已知条件为：

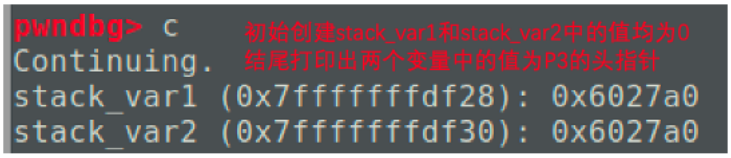
- `P2->bk->fd = stack_var1_addr`
- `P2->bk->fd = P3`

那么即可的出结论 `stack_var1的值 = P3的头指针` ，所以stack_var1的值在这个流程中被修改成了P3的头指针

查看修改结果

最后我们将直接运行程序至结束，再一次查看一下此时stack_var1和stack_var2中的值





可以看到此时stack_var1和stack_var2中的值已经被修改成了P3的头指针 `0x6027a0`

Large bin attack利用条件

- 可以修改一个large bin chunk的data
- 从unsorted bin中来的large bin chunk要紧跟在被构造过的chunk的后面

malloc.c中从unsorted bin中摘除chunk完整过程代码

```
          /* remove from unsorted list */
          unsorted_chunks (av)->bk = bck;
          bck->fd = unsorted_chunks (av);

          /* Take now instead of binning if exact fit */

          if (size == nb)
            {
              set_inuse_bit_at_offset (victim, size);
              if (av != &main_arena)
                victim->size |= NON_MAIN_ARENA;
```

我是hollk，交个朋友吧~
QQ名片

“相关推荐”对你有帮助么?
非常没帮助 没帮助 一般 有帮助 非常有帮助