好好说话之Tcache Attack（3）：tcache stashing unlink attack

tcache stashing unlink attack这种攻击利用有一个稍微绕的点，就是small bin中的空闲块挂进tcache bin这块。弯不大，仔细想想就好了

> 往期回顾：
> 好好说话之Tcache Attack（2）：tcache dup与tcache house of spirit
> 好好说话之Tcache Attack（1）：tcache基础与tcache poisoning
> 好好说话之Large Bin Attack
> 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
> …

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·)/

tcache stashing unlink attack

首先从名字就可以看出这种方法与unlink有关，这种攻击利用的是tcache bin中有剩余（数量小于TCACHE_MAX_BINS）时，同大小的small bin会放进tcache中，这种情况可以使用 `calloc` 分配同大小堆块触发，因为calloc分配堆块时不从tcache bin中选取。在获取到一个smallbin中的一个chunk后，如果tcache任由足够空闲位置，会将剩余的smallbin挂进tcache中，在这个过程中只对第一个bin进行了完整性检查，后面的堆块的检查缺失。当攻击者可以修改一个small bin的bk时，就可以实现在任意地址上写一个libc地址。构造得当的情况下也可以分配fake_chunk到任意地址

例题：how2heap中的tcache stashing unlink attack

例题源码如下，稍作改动，去掉了一些不影响执行流程的输出代码：

```
1  //gcc -g -no-pie hollk.c -o hollk
2  //patchelf --set-rpath 路径/2.27-3ubuntu1_amd64/ hollk
3  //patchelf --set-interpreter 路径/2.27-3ubuntu1_amd64/ld-linux-x86-64.so.2 hollk
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <assert.h>
7
8  int main(){
9      unsigned long stack_var[0x10] = {0};
10     unsigned long *chunk_lis[0x10] = {0};
11     unsigned long *target;
```

简单的描述一下这个程序的执行流程：首先创建了一个数组 `stack_var[0x10]` ，一个指针数组 `chunk_lis[0x10]` ，一个指针 `target` 。接下来调用setbuf()函数进行初始化。接着调用printf()函数打印stack_var、chunk_lis首地址及target的地址。接下来将 `stack_var[2]` 所在地址放在 `stack_var[3]` 中。接着循环创建 `8` 个size为 `0xa0` 大小的chunk，并将八个chunk的malloc指针依序放进chunk_lis[]中。然后根据chunk_lis[]中的堆块malloc指针循环释放 `6` 个已创建的chunk。接下来依序释放 `chunk_lis[1]、chunk_lis[0]、chunk_lis[2]` 中malloc指针指向的chunk。然后连续创建三个chunk，第一个size为 `0xb0` ，第二个size为 `0xa0` ，三个size为 `0xa0` 。接下来将 `chunk_lis\[2]\[1]` 位置中的内容修改成 `stack_var` 的起始地址，接着调用 `calloc()` 函数申请一个size为 `0xa0` 大小的chunk。最后申请一个size为 `0xa0` 大小的chunk，并将其malloc指针赋给target变量，并打印target

由于我们在编译的时候使用了 `-g` 参数，所以在使用gdb进行调试的时候可以在代码行下断点。首先我们在第 `19` 行下断点，查看一下打印出来的各个变量的地址：

```
stack_var addr is:0x7fffffffdde0
chunk_lis addr is:0x7fffffffde60
target addr is:0x7ffff7de01ef
```

可以看到stack_var的起始地址为 `0x7fffffffdde0` ，chunk_lis的起始地址为 `0x7fffffffde60` ，target的起始地址为 `0x7ffff7de01ef` 。这里建议拿个小本本记一下这三处地址，后面在查看内存变化的时候会反复查看，也可以现在就看一下三个地址内部的情况。接下来我们将断点下在第 `21` 行，使程序执行 `stack_var[3] = (unsigned long)(&stack_var[2]);` 这段代码：

```
pwndbg> x/10gx 0x7fffffffdde0
0x7fffffffdde0: 0x0000000000000000      0x0000000000000000
0x7fffffffddf0: 0x0000000000000000      0x00007fffffffddf0
0x7fffffffde00: 0x0000000000000000      0x0000000000000000
0x7fffffffde10: 0x0000000000000000      0x0000000000000000
```

这里我们看一下stack_var数组内部的情况，在stack_var[3]位置中的内容被修改成了stack_var[2]的地址，接下来我们将断点下在第 `29` 行，执行两个for循环：

```
pwndbg> bin
tcachebins
0xa0 [  6]: 0x602760 —▸ 0x6026c0 —▸ 0x602620 —▸ 0x602580 —▸ 0x6024e0 —▸ 0x602440 ◂— 0x0
```

这里强调一下，第二个for循环中起始释放的chunk的下标为 `3` ，所以释放是从第 `4` 个chunk开始的。上图是释放之后bin中的情况，由于我们使用patchelf将程序执行glibc的版本修改为 `2.27` ，所以存在tcache机制。被释放的chunk会进入tcache bin中，由于在第一个for循环中我们创建的都是size为 `0xa0` 大小的chunk，所以释放之后都会进入tcache bin中 `0xa0` 这条单项链表中。这里注意看链表中其实只有 `6` 个被释放块，但是tcache链表存放被释放块数量的最大值为 `7` ，所以此时tcache并不是满状态

接下来将断点下在第 `33` 行，依序释放chunk_lis[1]、chunk_lis[0]、chunk_lis[2]，我们再来看一下bin中的情况：

```
pwndbg> bin
tcachebins
0xa0 [  7]: 0x602300 —▸ 0x602760 —▸ 0x6026c0 —▸ 0x602620 —▸ 0x602580 —▸ 0x6024e0 —▸ 0x602440 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x602390 —▸ 0x602250 —▸ 0x7ffff7dcfca0 (main_arena+96) ◂— 0x602390
```


可以看到在释放chunk_lis[1]的时候 `chunk2` 作为最后一个进入tcache的chunk填满了整条链表，接下来再继续释放size为 `0xa0` 的堆块的话就不会在进入此条单向链表了。由于chunk_lis[0]、chunk_lis[2]中malloc指向的chunk的size都为 `0xa0` ，所以超过 `fastbin max size` ，所以会进入unsorted bin中，上图可以看到此时chunk1与chunk3已经进入了unsorted bin中。接下来我们将断点下在第 `34` 行，申请一块size为0xb0大小的chunk，我们在看一下bin中的情况：

```
pwndbg> bin
tcachebins
0xa0 [  7]: 0x602300 —▸ 0x602760 —▸ 0x6026c0 —▸ 0x602620 —▸ 0x602580 —▸ 0x6024e0 —▸ 0x602440 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0xa0: 0x602390 —▸ 0x602250 —▸ 0x7ffff7dcfd30 (main_arena+240) ◂— 0x602390
```


由于unsorted bin存取机制的原因，如果此时申请一个size为 `0xb0` 大小的chunk，unsorted bin中如果没有符合chunk size的空闲块（chunk3、chunk1的size小于0xb0），那么unsorted bin中的空闲块chunk3和chunk1会按照size落在small bin的 `0xa0` 链表中。接下来我们将断点下在第 `37` 行，完成两次申请size为 `0xa0` 大小的chunk

```
pwndbg> bin
tcachebins
0xa0 [  5]: 0x6026c0 —▸ 0x602620 —▸ 0x602580 —▸ 0x6024e0 —▸ 0x602440 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0xa0: 0x602390 —▸ 0x602250 —▸ 0x7ffff7dcfd30 (main_arena+240) ◂— 0x602390
```

这样一来由于tcache bin中又满足size为 0xa0 的空闲块，所以chunk2和chunk4就被重新启用了。那么此时bin中就形成了tcache bin中存在 5 个空闲块，small bin中存在 2 个空闲块的情况了，后面去讲为什么这样去部署。接下来我们将断点下在第 38 行，执行 chunk_lis[2][1] = (unsigned long)stack_var; 这条语句：

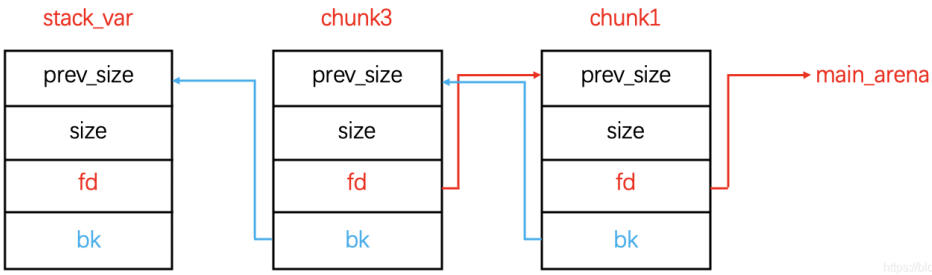
```
pwndbg> x/20gx 0x7fffffffde60 chunk_lis
0x7fffffffde60: 0x0000000000602260      0x0000000000602300
0x7fffffffde70: 0x00000000006023a0 chunk3 0x0000000000602440
0x7fffffffde80: 0x00000000006024e0      0x0000000000602580
0x7fffffffde90: 0x0000000000602620      0x00000000006026c0
0x7fffffffdea0: 0x0000000000602760      0x0000000000000000
```

```
pwndbg> x/20gx 0x0000000000602390 chunk3
0x602390:       0x0000000000000000      0x00000000000000a1
0x6023a0:       0x0000000000602250      0x00007fffffffdde0 bk=stack_var
0x6023b0:       0x0000000000000000      0x0000000000000000
```

chunk_lis[2][1] = (unsigned long)stack_var; 这条语句是这样执行的，首先chunk_lis[2]的位置就是存放chunk3的malloc指针的位置，那么chunk_lis[2][1]指的就是以chunk3头指针为起始位置，向后第二个地址位宽的位置，即chunk3的 bk 位置。chunk3_bk中的内容就被修改成了 stack_var 的头指针，这个时候我们可以看一下bin中的状况：

```
smallbins
0xa0 [corrupted]
FD: 0x602390 —▸ 0x602250 —▸ 0x7ffff7dcfd30 (main_arena+240) ◂— 0x602390
BK: 0x602250 —▸ 0x602390 —▸ 0x7fffffffdde0 —▸ 0x7fffffffddf0 ◂— 0x0
```

由于chunk3是unsorted bin中最后一个chunk，且chunk3的bk被修改成了stack_var的头指针，所以，stack_var会被认为是紧跟着chunk3之后释放的一个chunk：



那么接下来我们将断点下在第40行，调用calloc函数申请一个size为0xa0大小的chunk：

```
pwndbg> bin
tcachebins      stack_var       chunk3          chunk5          chunk6          chunk7          chunk8          chunk9
0xa0 [  7]: 0x7fffffffddf0 —▸ 0x6023a0 —▸ 0x6026c0 —▸ 0x602620 —▸ 0x602580 —▸ 0x6024e0 —▸ 0x602440 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0xa0 [corrupted]
FD: 0x602390 —▸ 0x6026c0 ◂— 0x0
BK: 0x7fffffffddf0 ◂— 0x0
```


这里说明一下为什么要使用 calloc 进行申请chunk，这是因为calloc在申请chunk的时候 不会从tcache bin中摘取空闲块 ，如果这里使用malloc的话就会直接从tcache bin中获得空闲块了。那么在calloc申请size为 0xa0 大小的chunk的时候就会直接从 small bin 中获取，那么由于small bin是 FIFO 先进先出机制，所以这里被重新启用的是 chunk1

这个时候就到了前面理论部分描述的内容了：在获取到一个 smallbin 中的一个 chunk 后会如果 tcache 仍有足够空闲位置（tcache中有两个位置，chunk3和stack_var刚好够落在这两个位置），剩下的 smallbin 从最后一个 stack_var (0x7fffffffddf0)开始顺着 bk 链接到 tcachebin 中，在这个过程中只对第一个 chunk3 进行了完整性检查，后面的stack_var的 检查缺失 。这样一来就造成上图的效果，stack_var就被挂进了tcache bin的链表中

接下来我们将断点下在第44行，最后申请一个size为 0xa0 大小的chunk，并将其malloc指针赋给target变量，并打印target：

```
pwndbg> c
Continuing.     stack_var
target now: 0x7fffffffddf0
```

由于stack_var处于tcache链表的最后一个，所以在申请size为0xa0大小的chunk的时候，stack_var就会被重新启用

例题后补~