好好说话之Tcache Attack（2）：tcache dup与tcache house of spirit

这篇文章介绍了两种tcache的利用方法，tcache dup和tcache house of spirit，两种方法都是用how2heap中的例题作为讲解。由于tcache attack这部分的内容比较多，所以分开几篇文章去写。例题后补，写完例题后可能会进行重新排版，内容不会少的！！！

> 往期回顾：
> 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 dup

这种利用方式和前面fastbin attack中的fastbin dup很像，tcache dup利用的是tcache_put()未做安全检查的缺陷，我们来回顾一下tcache_put()函数：

```
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  return (void *) e;
}
```

在具备tcache机制的情况下，申请释放内存的时候，_int_free()函数会调用tcache_put()函数，tcache_put()函数会按照size对应的idx将已释放块挂进tcache bins链表中。插入的过程也很简单，根据_int_free()函数传入的参数，将被释放块的malloc指针交给next成员变量。其中没有任何安全检查和保护机制，在大服务提高性能的同时，安全性几乎舍弃了大半

因为没有做任何的检查，所以我们可以对同一个chunk多次free，这就会造成cycliced list。我们在fastbin attack中经常用到

例题：how2heap 中的 tcache_dup

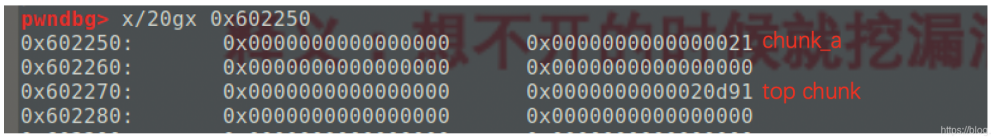
源码如下，做了一些小小的改动，将不重要的输出省略了，不影响正常执行：

```
//gcc -g -no-pie hollk.c -o hollk
//glibc_2_27:
//patchelf --set-rpath 路径/2.27-3ubuntu1_amd64/ hollk
//patchelf --set-interpreter 路径/2.27-3ubuntu1_amd64/ld-linux-x86-64.so.2 hollk
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
        int *a = malloc(8);
```

我们来简单的解读一下这个程序的执行流程：首先创建了一个size为 0x20 大小的chunk，并将chunk的malloc指针赋给了指针变量 a ，接下来连续释放了两次chunk_a。然后重新申请了两个size为 0x20 大小的chunk，并分别将两个chunk的malloc指针付给了指针变量 b 和指针变量 c 。最后打印出chunk_b和chunk_c的malloc指针

由于我们在编译的时候使用 -g 参数，所以在使用gdb进行动态调试的时候可以在对应代码行下断点，首先我们在第 13 行下断点，看一下已创建的chunk_a的地址在哪：

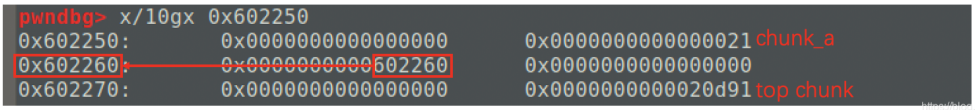


可以看到chunk_a的头指针为 0x602250 ， fd所在位置为 0x602260 ，接下来我们将断点下在第 14 行，看一下第一次释放chunk_a后bin中的情况：

由于我们利用patchelf将程序执行的glibc修改为2.27版本，所以是存在tcache机制的，所以第一次释放chunk_a之后会被挂在tcache bin中0x20这条单向链表中。接下来我们将断点下载第 15 行，再一次释放chunk_a：

bin中情况

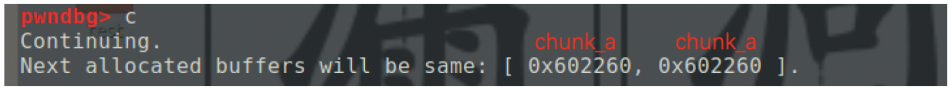
chunk_a内部情况



首先看上图的上半部分，再次释放chunk_a之后chunk_a的malloc指针 又一次被挂进了tcache bin中 ，我们也可以观察一下gdb给我们的提示，在0x20后有一个标识当前idx链表中chunk的数量，这里显示的是 2 ，tcache_put()函数并没有意识到chunk_a是一个已被释放的chunk，这就造成了 cycliced list ，chunk_a有指向了自己（~~是我S了我！~~ ）。我们再来看下半部分的chunk_a内部状况，可以看到chunk_a的 fd 此时指向的是 自身的malloc地址 。接下来我们将断点下在第 17 行，看一下重新申请一个0x20大小的chunk时的情况：

可以看到，由于tcache bin中有符合申请size要求的空闲块，所以在申请0x20大小的chunk时，tcache bin中0x20链表最后一个被释放的chunk_a理所应当的被重新启用。这个时候可以看一下gdb标识当前链表中的chunk数量的确从 2 变为了 1 ，但是chunk_a虽然被启用， 链表中依然还会存在chunk_a的malloc指针 ，这是由于 第一次被启用的是第二次被释放的chunk_a

接下来我们间断点下在第 20 行，完成第二次申请0x20的chunk，并打印出chunk_b和chunk_c的malloc指针：



可以看到打印出的chunk_b和chunk_c的malloc指针都是chunk_a的malloc指针，这是因为chunk_a被释放的两次，第一次申请的chunk_b启用了第二次释放的chunk_a，第二次申请的chunk_c启用了第一次被释放的chunk_a。

总体来说和前面的fastbin attack构造循环单项链表的方式差不多，只是对利用的机制不一样

tcache house of spirit

tcache house of spirit这种利用方式是由于tcache_put()函数检查不严格造成的，在释放的时候没有检查被释放的指针是否真的是堆块的malloc指针，如果我们构造一个size符合tcache bin size的fake_chunk，那么理论上讲其实可以将任意地址作为chunk进行释放。这里就直接采用wiki上面列出的例子进行讲解了：

how2heap中的tcache_house_of_spirit

源码如下，稍作改动，去掉了一些输出语句，不影响正常程序执行流程：

```
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  {
10         setbuf(stdout, NULL);
11
```

我们简单的说明一下这个程序的执行流程：首先使用setbuf()函数进行初始化，然后创建了一个堆块，这个堆块的作用其实是为了防止后面的chunk与top chunk合并 的。接下来定义了一个指针变量 a ，还定义了一个整型数组 fake_chunk[10] 。接下来打印了fake_chunk的起始地址，将 fake_chunk[1] 中的内容修改成了 0x40 。接下来将fake_chunk[2]所在地址赋给指针变量 a ，然后释放a。接着重新申请一个size为 0x40 大小的chunk，并将其malloc地址赋给指针变量 b ，最后打印出chunk_b的malloc地址

由于我们在编译阶段使用了 -g 参数，所以在使用gdb调试过程中可以在代码行下断点。首先我们在第 19 行下断点，看一下打印出来的fake_chunk[]的起始地址：

```
fake_chunk addr is 0x7fffffffdea0
```

可以看到fake_chunk[]的起始地址为 0x7fffffffdea0 ，接下来我们将断点下在第 21 行，执行 fake_chunks[1] = 0x40; 这段代码：

```
pwndbg> x/20gx 0x7fffffffdea0
0x7fffffffdea0: 0x000000000000000b      0x0000000000000040
0x7fffffffdeb0: 0x00007ffffffdf18       0x000000000f0b5ff
0x7fffffffdec0: 0x0000000000000001      0x00000000004007fd
0x7fffffffded0: 0x00007ffff7de59a0      0x0000000000000000
0x7fffffffdee0: 0x00000000004007b0      0x00000000004005e0
```

prev_size　size　fd

fake_chunks[1] = 0x40 这段代码修改了fake_chunk[1]中的内容，这其实是一个构造fake_chunk的过程，0x40就是这个fake_chunk的 size ，接下来我们将断点下在第 25 行，我们看一下在释放fake_chunk后bin中的状况：

```
pwndbg> bin
tcachebins      fake_chunk
0x40 [  1]: 0x7fffffffdeb0 ← 0x0
```

可以看到由于我们使用了patchelf将程序的glibc修改成了 2.27 版本，所以存在tcache机制，释放的chunk会被挂进tcache bin中。那么由于我们将 fake_chunk[2] 所在地址赋给了指针变量 a ，并且free(a)。由于tcache_put()函数没有做任何的安全检查，所以就将fake_chunk看作为一个正常的chunk给释放了 。所以我们可以在tcache bin的0x40这条单向链表中看到fake_chunk的malloc指针。接下来我们将断点下在第28行，重新申请一个size为0x40大小的chunk_b，并打印其malloc地址：

```
pwndbg> c
Continuing.      fake_chunk
malloc(0x30): 0x7fffffffdeb0
```

可以看到打印出的chunk_b的malloc地址就是fake_chunk的malloc地址。这是由于在tcache bin中的0x40链表里刚好有符合申请size要求的空闲块，这个空闲块就是fake_chunk，所以再次申请的时候fake_chunk作为链表中最后一个chunk，就被重新启用了

例题后补~