好好说话之Tcache Attack（1）：tcache基础与tcache poisoning

进入到了Tcache的部分，我还是觉得有必要多写一写基础的东西。以往的各种攻击手法都是假定没有tcache的，从练习 二进制漏洞挖掘的角度来看其实我们一直模拟的都是很老的环境，那么这样一来其实和真正的生产环境就产生了较大的差距，这导致了CTF-PWN就是CTF-PWN，除了比赛有用之外让人看不出实际的生产转化。以上均属个人想法，如果你觉得纯理论的东西没什么意思，完全可以跳过这前面部分直接看后面的例题

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
> …

Tcache overview（部分引用CTF-Wiki）

tcache是glibc 2.26(Ubuntu 17.10)之后引入的一种技术，其目的是为了提升堆管理的性能。我们都知道，一旦某个整体的应用添加了更加复杂的执行流程，那么就意味着整体执行的速度就会降低，那么为了弥补这一部分的欠缺，就不得不有所牺牲。所以虽然提升了整体的性能，但却舍弃了很多安全检查，这就意味着更多新的漏洞就伴随而来，也增添了很多利用方式

相关结构

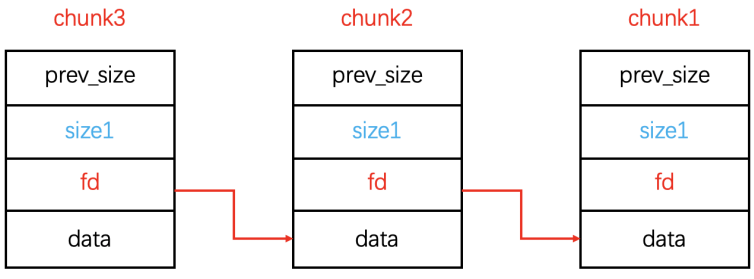
tcache引入了两个新的结构体： `tcache_entry` 和 `tcache_perthread_struct` 。增添的两个结构体其实与fastbin有些类似，但是也有一定的区别

tcache_entry

tcache_entry结构体如下：

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;
```

`tcache_entry` 用于链接空闲的chunk结构体，其中 `next` 指针指向下一个 大小相同 的chunk。



这里需要注意的是next指向chunk的 `data` 部分，这和fastbin有一些不同，fastbin的fd指向的是下一个chunk的头指针。tcache_entry会复用空闲chunk的data部分

tcache_perthread_struct

tcache_perthread_struct结构体如下

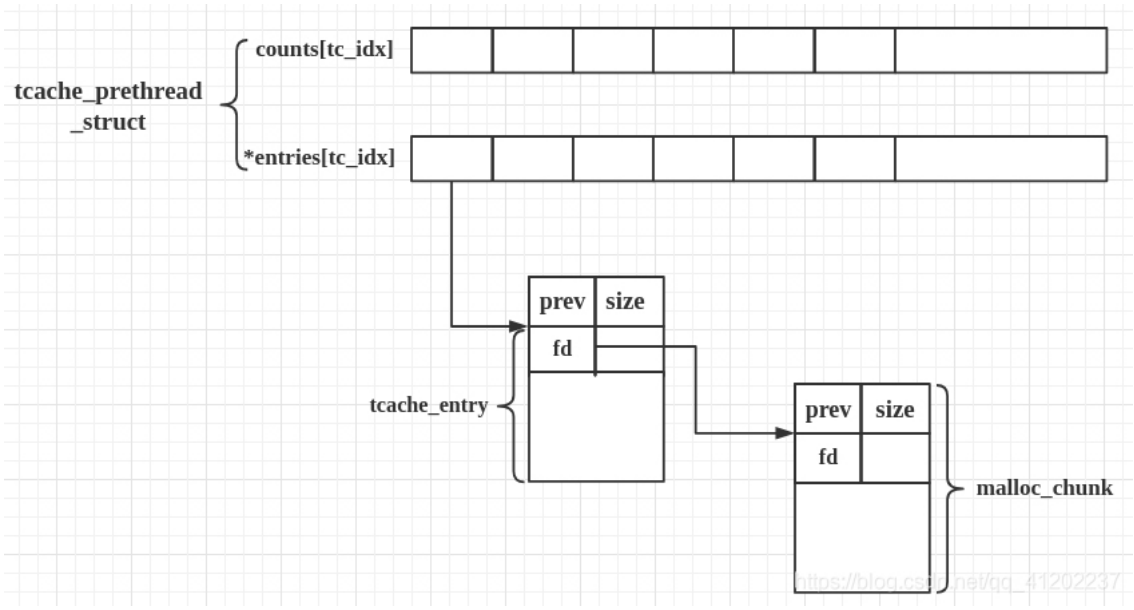
```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

```
6
7  # define TCACHE_MAX_BINS                64
8
9  static __thread tcache_perthread_struct *tcache = NULL;
```

tcache_perthread_struct是用来管理tcache链表的，这个结构体位于heap段的起始位置，size大小为0x251。每一个thread都会维护一个 `tcache_perthread_struct` 结构体，一共有 `TCACHE_MAX_BINS` 个计数器 `TCACHE_MAX_BINS` 项tcache_entry。其中：

- tcache_entry 用单向链表的方式链接了相同大小的处于空闲状态（free 后）的 chunk
- counts 记录了 tcache_entry 链上空闲 chunk 的数目，每条链上最多可以有 7 个 chunk

tcache_perthread_struct、tcache_entry和malloc_chunk三者的关系如下：



tcache_perthread_struct结构体的 `entries成员变量` 指向tcache_entry结构体的 `next成员变量地址` ，tcache_entry结构体的 `next成员变量` 指向空闲的malloc_chunk的 `malloc地址` 。可以形象的理解为tcache_perthread_struct结构体是大总管，tcache_entry结构体是经理，空闲chunk为员工

Tcache usage

tcache执行流程如下：

- 第一次malloc时，回显malloc一块内存用来存放 `tcache_perthread_struct` ，这块内存size一般为0x251
- 释放chunk时，如果chunk的size小于small bin size，在 `进入tcache之前` 会先放进fastbin或者unsorted bin中
- 在 `放入tcache后` ：
  - 先放到对应的tcache中，直到tcache被填满（7个）
  - `tcache被填满后` ，接下来再释放chunk，就会直接放进fastbin或者unsorted bin中
  - tcache中的chunk `不会发生合并` ，不取消inuse bit
- 重新申请chunk，并且申请的size符合tcache的范围，则先 `从tcache中取chunk` ，直到tcache为空
- tcache为空后，从bin中找
- tcache为空时，如果fastbin、small bin、unsorted bin中有size符合的chunk，会先把fastbin、small bin、unsorted bin中的chunk放到tcache中，直到填满，之后再从tcache中取

需要注意的是，在采用tcache的情况下，只要是bin中存在符合size大小的chunk，那么在重启之前都需要经过tcache一手。并且由于tcache为空时先从其他bin中导入到tcache，所以此时chunk在bin中和在tcache中的顺序会反过来

源码分析

内存申请

tcache初始化的部分在这里就不多说了，因为能利用的点很少。这里就直接从申请内存阶段开始讲解了，首先是申请内存的步骤：

```
  // 从 tcache list 中获取内存
  if (tc_idx < mp_.tcache_bins && tcache && tcache->entries[tc_idx] != NULL)
    {
      return tcache_get (tc_idx);
    }
  DIAG_POP_NEEDS_COMMENT;
#endif
    }
```

这部分的代码描述的是从tcache中取chunk的一系列步骤，首先是在tcache中有chunk的时候，if判断要取出的chunk的size是否满足idx的合法范围，在tcache->entries不为空时调用 `tcache_get()` 函数获取chunk。

tcache_get()函数

接下来看一下 `tcache_get()` 函数的代码：

```
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;
}

```

可以看到tcache_get()函数的执行流程很简单，从 `tcache->entries[tc_idx]` 获取一个chunk指针，并且 `tcache->counts` 减一，没有过多的安全检查或者保护

内存释放

我们看一下在有tcache的情况下 `_int_free()` 中的执行流程：

```
static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  ......
  ......
#if USE_TCACHE
  {
    size_t tc_idx = csize2tidx (size);
    if (tcache
        && tc_idx < mp_.tcache_bins // 64
        && tcache->counts[tc_idx] < mp_.tcache_count) // 7
```

可以看到首先判断 `tc_idx` 的合法性，判断 `tcache->counts[tc_idx]` 在7个以内时，进入 `tcache_put()` 函数，传递的一参为要释放的chunk指针，二参为chunk对应的size在tcache中的下标

tcache_put()函数

```
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  assert (tc_idx < TCACHE_MAX_BINS);
  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}
```

tcache_put()函数执行过程中把释放的chunk插入到了tcache->entries[tc_idx]链表的头部，整个插入的过程中也没有做任何的安全检查及保护，也没有将P标志位变为0

通过对tcache_get()和tcache_put()两个函数的分析，我们可以看到并没有很严格的进行安全检查，没有对溢出、复用、二次释放等攻击手段进行审计，所以tcache机制伴随而来的是更多的安全问题

PWN Tcache

tcache由于省略了很多安全保护机制，所以在pwn中的利用方式有很多，这篇文章我们首先介绍 `tcache poisoning` 这种利用方式

tcache poisoning

tcache poisoning主要的利用手段是覆盖tcache中的next成员变量，由于tcache_get()函数没有对next进行检查，所以理论上来讲如果我们将next中的地址进行替换，不需要伪造任何chunk结构即可实现malloc到任何地址

这里我们以how2heap中的tcache_poisoning作为例子讲解，稍作了一些改动，将不必要的输出语句省略掉了

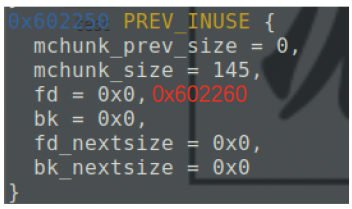
```
1  //gcc -g -no-pie hollk.c -o hollk
2  //glibc_2_27:
3  //patchelf --set-rpath 你的路径/2.27-3ubuntu1_amd64/ hollk
4  //patchelf --set-interpreter 你的路径/2.27-3ubuntu1_amd64/ld-linux-x86-64.so.2 hollk
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <stdint.h>
8  #include <assert.h>
9
10 int main()
11 {
```

简单的描述一下这个例子的执行流程，首先 `setbuf()` 函数进行初始化，然后定义了一个 `target` 变量。接下来申请了两个size为 `0x90` （128+16）的chunk，两个malloc指针分别给了指针变量a和指针变量b。接下来首先释放了chunk_a，又释放了chunk_b。然后修改指针数组 `b[idx]` 下标为 `0` 位置的内容为 `target变量的地址` 。随后重新申请了两个size为0x90大小的chunk，并将后申请的chunk的malloc指针赋给了指针变量 `c` 。最后打印出指针变量c

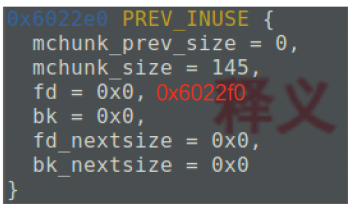
由于在编译阶段使用了 `-g` 参数，所以使用gdb进行动态调试的时候可以按照代码行下断点。首先我们在第 `18` 行下断点，看一下输出的target变量的地址：

可以看到打印出来的target_addr为 `0x7fffffffdee8` 。接下来我们将断点下在第 `21` 行，看一下申请的两个chunk的指针为多少：

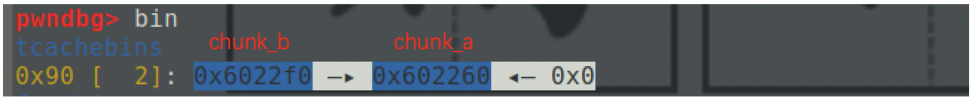
chunk_a



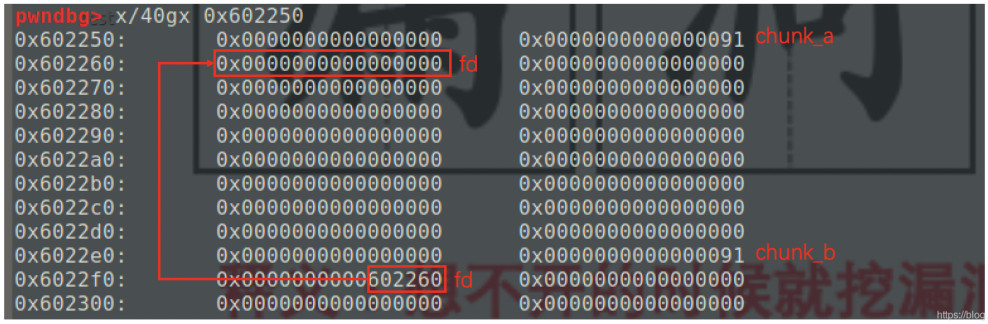
chunk_b



可以看到创建的两个chunk，其中chunk_a_addr = `0x602250` 、chunk_b_addr = `0x6022e0` 。由此可以推断chunk_a的fd位置的地址为 `0x602260` ，chunk_b的fd位置的地址为 `0x6022f0` 。接下来我们在第 `24` 行下断点，看一下释放chunk之后bin中的情况：



由于我们使用patchelf将glibc的版本调整到了2.27，所以是存在tcache机制的，由于chunk_a和chunk_b的size同为0x90，所以chunk_a和chunk_b都被刮进了tcache bin中0x90这条链表中。我们也可以看一下在释放后两个chunk内部的情况：

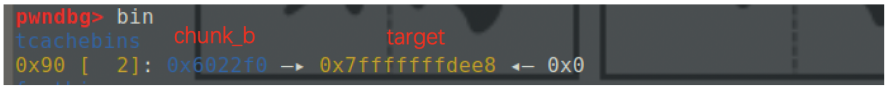


我们可以看到chunk_b的fd指向的其实是chunk_a的malloc指针，你再回头看一下前一个图，挂进tcache bin中的两个指针分别的chunk_a的malloc指针和chunk_b的malloc指针，是不是刚才忽略了，以为是头指针呢😂。接下来我们将断点下在第 26 行，使程序执行 `b[0] = (intptr_t)&target;` 这段代码：

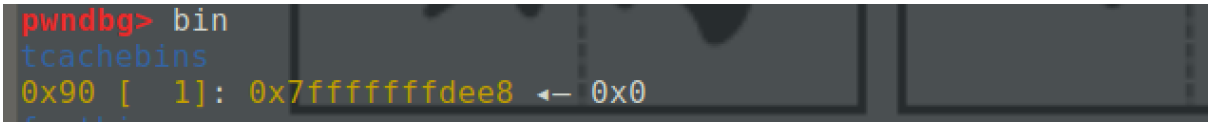
未修改前：fd指向chunk_a的data

修改后：fd指向target变量

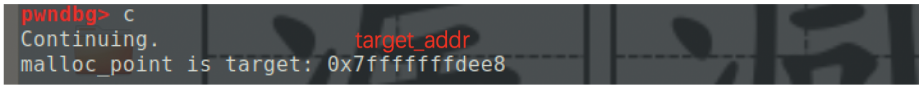
`b[0] = (intptr_t)&target;` 这段代码事实上就是修改了chunk_b的fd指针指向的地址，指针变量b承接的是chunk_b的malloc指针，所以b[0]的位置就是chunk_b的fd，这样一来chunk_b的fd从原来指向chunk_a，变成了指向target变量的地址，我们看一下修改后bin中的情况：



可以看到，随着chunk_b的fd被修改成target_addr，tcache bin链表中的成员也发生了改变，target就被认为是chunk_b前一个被释放的chunk。接下来我们将断点下在第 27 行，在申请一次0x90大小的chunk之后tcache bin中的情况怎么样：



可以看到target作为tcache bin中0x90这条链表中最后一个被释放块了，那么我们再将断点下在第 30 行，再次申请一个0x90大小的chunk，并打印出新申请的chunk的malloc指针：



可以看到被挂在tcache bin中的target被当做一个释放chunk重新启用了！

例题后补~