好好说话之Fastbin Attack（3）： Alloc to Stack

真的是好久好久都没更新了。。。。最近换工作再加上考CISP-PTE认证耽误了很长一段时间，这段时间打算把CISP-PTE的一些考点总结出来，如果你是做渗透工作或者ctf的web手，这个认证其实并不难。再来说说这篇文章吧，Alloc to Stack这种利用技巧其实和前面两篇区别不大，只不过就是伪造的chunk放在了栈上。这篇文章不会很长，因为wiki上没有适配的例子，自己也没找到。如果有做过这种技巧的例子的师傅，欢迎评论区留言～wiki上2015 9447 CTF : Search Engine这道题我会在下一篇文章中单独解析～

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
> …

Alloc to Stack

介绍

Alloc to Stack这种技巧和前两篇文章中的Fastbin Double Free与house of spirit技术差不多。这三种技巧的本质都在于fastbin链表的特性：当前chunk的fd指针指向下一个chunk

Alloc to Stack这种技术关键点在于劫持fastbin链表中chunk的fd指针，把fd指针指向我们想要分配的栈上，从而实现 控制栈中的一些关键数据，例如返回地址等

演示

在这个例子中，为了方便理解，我们讲fake_chunk置于栈中称为stack_chunk，同时劫持了fastbin链表中的chunk的fd值，通过把fd值指向stack_chunk就可以实现在栈中分配fastbin chunk

```
1  /*gcc -g test.c -o test*/
2  #include<stdio.h>
3
4  typedef struct _chunk
5  {
6      long long pre_size;
7      long long size;
8      long long fd;
9      long long bk;
10 } CHUNK,*PCHUNK;
11
```

我们来一起解读一下这个例子，首先自定义了一个结构体，long long型的，所以我们在你使用gcc编译的时候要编译成64位的。定义的结构体为了伪造嘛，所以和正常chunk的结构体一样，成员变量都是prev_size、size、fd、bk。接下来看main函数，首先创建了一个结构体指针stack_chunk。接着创建了两个long long类型的指针chunk1和chunk_a。将结构体 stack_chunk的size成员变量的值设置为0x21，创建一个0x10大小的chunk（实际堆块大小为0x20），并将chunk的malloc指针赋给chunk1指针。接下来释放了chunk1。重新在chunk1指针指向的位置赋予stack_chunk指针，接着 重新申请0x10的chunk，在将chunk的malloc指针赋给chunk_a

接下来使用gdb调试一下这个例子：gdb打开之后，因为我们在编译阶段使用了-g参数，所以首先在第22行下断点，然后执行，使用 `heap` 命令查看一下当前的堆块状态

```
pwndbg> heap
0x55555575a000 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x0,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x20fe1
}
```

```
pwndbg> x/10gx 0x55555575a000
0x55555575a000: 0x0000000000000000      0x0000000000000021
0x55555575a010: 0x0000000000000000      0x0000000000000000
0x55555575a020: 0x0000000000000000      0x0000000000020fe1
0x55555575a030: 0x0000000000000000      0x0000000000000000
0x55555575a040: 0x0000000000000000      0x0000000000000000
```

接下来将断点下在第24行，使程序完成释放，我们去看一下bin中的情况以及被释放chunk的内部状态：

```
pwndbg> bin
fastbins
0x20: 0x55555575a000 ◂— 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
```

```
pwndbg> x/20gx 0x55555575a000
0x55555575a000: 0x0000000000000000      0x0000000000000021
0x55555575a010: 0x0000000000000000      0x0000000000000000
0x55555575a020: 0x0000000000000000      0x0000000000020fe1
0x55555575a030: 0x0000000000000000      0x0000000000000000
```

可以看到在释放之后chunk精准的落在了fastbin中，由于chunk1前面并没有任何chunk被释放，所以chunk的fd位置为空，不指向任何chunk。但是这里出现了一个问题，在释放chunk1之后并没有将其malloc指针置空，这就造成了chunk1可以被重新修改的状况。接下来我们将断点下在第25行，完成对chunk1中fd的修改：

```
pwndbg> bin
fastbins                    Stack_chunk
0x20: 0x55555575a000 —▸ 0x7fffffffdf20 —▸ 0x555555554780 (__libc_csu_init) ◂— 0x6162d8d48550020 /* ' ' */
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
```

```
pwndbg> x/10gx 0x55555575a000   chunk1
0x55555575a000: 0x0000000000000000      0x0000000000000021
0x55555575a010: 0x00007fffffffdf20      0x0000000000000000
0x55555575a020: 0x0000000000000000      0x0000000000020fe1
0x55555575a030: 0x0000000000000000      0x0000000000000000
```

```
pwndbg> x/10gx 0x00007fffffffdf20   Stack_chunk
0x7fffffffdf20: 0x0000000000000000      0x0000000000000021
0x7fffffffdf30: 0x0000555555554780      0x000055555555545f0
0x7fffffffdf40: 0x00007fffffffe030      0x8e6b7bb411a29200
```

由于将chunk1中fd的值修改成了stack_chunk的结构体指针，那么在fastbin中看来，stack_chunk是在chunk1前面被释放的一个块，而stack_chunk其实是部署在栈上的一个伪造chunk：

```
pwndbg> stack
00:0000| rsp  0x7fffffffdf10 —▸ 0x55555575a010 —▸ 0x7fffffffdf20 —▸ 0x7fffffffdf4e ◂— 0x5555555547808e6b
01:0008|      0x7fffffffdf18 —▸ 0x5555555547cd (__libc_csu_init+77) ◂— 0x75dd394801c38348
02:0010| rdx  0x7fffffffdf20 —▸ 0x7fffffffdf4e ◂— 0x5555555547808e6b
03:0018|      0x7fffffffdf28 ◂— 0x21 /* '!' */
04:0020|      0x7fffffffdf30 —▸ 0x555555554780 (__libc_csu_init) ◂— 0x41d7894956415741
05:0028|      0x7fffffffdf38 —▸ 0x5555555545f0 (_start) ◂— 0x89485ed18949ed31
06:0030|      0x7fffffffdf40 —▸ 0x7fffffffe030 ◂— 0x1
07:0038|      0x7fffffffdf48 ◂— 0x8e6b7bb411a29200
```

那么这样一来堆管理器就会认为在fastbin0x20这条单向链表中存在两个0x20被释放的chunk。那么如果连续申请两块0x20大小的chunk，栈上伪造的stack_chunk就会被作为一个chunk来启用。我们将断点定到27行，完成两次申请，再去看一下bin中的状况：

```
pwndbg> bin
fastbins
0x20: 0x555555554780 (__libc_csu_init) ◂— 0x6162d8d48550020 /* ' ' */
0x30: 0x0
0x40: 0x0
0x50: 0x0
```

可以看到原来被挂进fastbin中的stack_chunk被启用调走了

总结

通过该技术我们可以把 fastbin chunk 分配到栈中，从而控制返回地址等关键数据。要实现这一点我们需要劫持 fastbin 中 chunk 的 fd 域，把它指到栈上，当然同时需要栈上存在有满足条件的 size 值