本文主要介绍Buddy System、Slab Allocator的实现机制以及现实中的一些漏洞利用方法，从攻击者角度加深对Linux内核内存管理机制的理解。

前言

伙伴系统

伙伴系统即Buddy System，是一种简单高效的内存分配策略。其主要思想是将大块内存按照一定策略去不断拆分（在到达最小的块之前），直至存在满足指定请求大小的最小块。其中块的大小由其相对根块的位置指定，通常称为order(阶)。一个最简单的拆分方式就是以2为指数进行拆分，例如定义最小块的大小为64K，order上限为4，则最大块的大小为：

64K * 2^4 = 1024K
最大块的order为4，最小块的order为0。对于请求大小为k的块，最小块为N，则其order值为align(k)/N。为什么叫buddy system呢？假设一个大块A所分解成的两个小块B和C，其中B和C就相互为彼此的 天使buddy。只有彼此的buddy才能够进行合并。

使用Buddy算法的的应用有很多，其中Linux内核就是一个，此外jemalloc也是使用Buddy技术的一个现代内存分配器。

维基百科中有一个很直观的例子：Buddy memory allocation[1]。Linux内核中的伙伴系统块大小为一页，通常是4096字节。最大的order一般是10，即MAX_ORDER为11。

```
/* Free memory management - zoned buddy allocator.  */
#ifndef CONFIG_FORCE_MAX_ZONEORDER
#define MAX_ORDER 11
#else
#define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
#endif
#define MAX_ORDER_NR_PAGES (1 << (MAX_ORDER - 1))
```

在Linux内核中，分配和释放较大内存都是直接通过伙伴系统，即：

```
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
```

order为0-10，指定从哪一阶伙伴中分配，order为n则分配2^n页大小的内存。操作系统中可以通过procfs查看伙伴系统的分配情况，如下：

```
$ cat /proc/buddyinfo
Node 0, zone      DMA      1      1      0      0      2      1      1      0      1      1      3
Node 0, zone    DMA32   3416   4852   3098   3205   3209   3029     33     22     15      7     52
Node 0, zone   Normal  29330 192053 148293  90568  33732   9018   2688    411    942    999   1852
Node 1, zone   Normal    107   1229  18644  76650  46053   8383   4398   5486   1751    497     84
```

此外，还有/proc/pagetypeinfo也可用于查看内存页的信息。

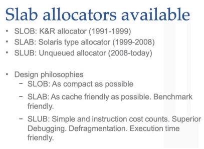
Slab分配器

上面说到，由于效率原因，伙伴系统中分配内存是以页为单位的，即使所分配的object大小为1byte，也需要分配一页，这样就导致了比较大的内存碎片。因此Linux引入了Slab分配器，加速对object的分配和释放速度，同时也减少碎片空间。

最初接触的时候心里通常有个大大的问号：Slab是什么？理解这个问题至关重要，经过翻阅多种资料和文章，可以大概这么回答：

1. Slab是一种缓存策略
2. Slab是一片缓冲区
3. Slab是一个或者多个连续的page

在内核代码中，我们能看到SLAB、SLOB、SLUB，其实都是兼容SLAB接口的具体分配器。目前默认使用的是SLUB，因此经常将其混称。按照引入Linux内核的时间划分如下：



Slab allocators available

- SLOB: K&R allocator (1991-1999)
- SLAB: Solaris type allocator (1999-2008)
- SLUB: Unqueued allocator (2008-today)

- Design philosophies
  - SLOB: As compact as possible
  - SLAB: As cache friendly as possible. Benchmark friendly.
  - SLUB: Simple and instruction cost counts. Superior Debugging. Defragmentation. Execution time friendly.

from：https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf

说句题外话，SLOB (Simple List Of Blocks) 可以看做是针对嵌入式设备优化的分配器，通常只需要几MB的内存。其采用了非常简单的first-fit算法来寻找合适的内存block。这种实现虽然去除了几乎所有的额外开销，但也因此会产生额外的内存碎片，因此一般只用于内存极度受限的场景。

数据结构

在本文中，我会尽量少粘贴大段的代码进行分析，但Slub分配器是比较依赖于实现而不是设计的，因此数据结构的介绍是难免的。

page

描述一个页的数据结构就是struct page。为了节约空间，page使用了大量的union结构，针对不同用处的页使用不同的字段。

Slab是一个或者多个连续页组成的内存空间，那么本质上指向一个Slab的数据结构不是别的，就是`struct page *`，对应Slab中的信息可以通过第一个page的某些字段描述。记住这点对后面的理解很重要。

kmem_cache

kmem_cache是Slab的主要管理结构，申请和释放对象都需要经过该结构操作，部分重要字段如下：

```
/*
 * Slab cache management.
 */
struct kmem_cache {
    struct kmem_cache_cpu __percpu *cpu_slab;
    /* Used for retriving partial slabs etc */
    unsigned long flags;
    unsigned long min_partial;
    int size;       /* The size of an object including meta data */
    int object_size;    /* The size of an object without meta data */
    int offset;     /* Free pointer offset. */
#ifdef CONFIG_SLUB_CPU_PARTIAL
    int cpu_partial;    /* Number of per cpu partial objects to keep around */
#endif
    ...
    struct kmem_cache_node *node[MAX_NUMNODES];
}
```

重点关注cpu_slab和node。

cpu_slab包含当前CPU的Slab。这是个__percpu的对象，什么意思呢？我的理解是内核为了加速当前CPU的访问，会对每个CPU保存一个变量，这样在当前CPU访问该变量时候就可以免去加锁的开销。在调试中发现该变量的值是个类似0x18940这样比较小的数，这个地址是没有映射的，访问percpu变量需要通过`raw_cpu_ptr`宏去获取实际的地址。

node数组中包括其他CPU的Slab。为什么叫做node？其实这是NUMA系统中的node概念。NUMA是为了多核优化而产生的架构，可以令某个CPU访问某段内存的速度更快。node的定义是"一段内存，其中每个字节到CPU的距离相等"，更加通俗的解释是："在同一物理总线上的内存+CPUs+IO+……"，更多细节可以参考NUMA FAQ[2]。

```
kmem_cache_cpu
cpu_slab是kmem_cache_cpu结构，如下：

 struct kmem_cache_cpu {
     void **freelist;    /* Pointer to next available object */
     unsigned long tid;  /* Globally unique transaction id */
     struct page *page;  /* The slab from which we are allocating */
 #ifdef CONFIG_SLUB_CPU_PARTIAL
     struct page *partial;   /* Partially allocated frozen slabs */
 #endif
 #ifdef CONFIG_SLUB_STATS
     unsigned stat[NR_SLUB_STAT_ITEMS];
 #endif
 };
freelist指向第一个空闲的对象(假设为x)，page指向x所在slab(的第一页)。这里的page有以下特点：

•objects = slab的对象数
•inuse = objects
•frozen = 1
•freelist = NULL
```

partial主要包含本地部分分配的slab。partial指向的page有以下特点：

```
•next指向下一个slab(page)
•freelist指向slab中第一个空闲object
•inuse = slab中已使用的object个数
•frozen = 1
```

其中第一个page的pbojects记录了partial objects数。

```
kmem_cache_node
struct kmem_cache_node {
    spinlock_t list_lock;
    ...
#ifdef CONFIG_SLUB
    unsigned long nr_partial;
    struct list_head partial;
    ..
#endif
};
```

这个数据结构根据配置的SL[OAU]B分配器而异，对于SLUB而言，使用的字段就只有两个，nr_partial和partial。其中partial是Linux内核中可插拔式通用双链表结构，使用内核中双链表的接口进行操作。nr_partial表示partial双链表中的元素个数，即slab的个数。

partial->next指向的page结构，用于该结构的page有如下特点：

```
•frozon = 0
•freelist指向slab中第一个空闲object
•inuse表示对应slab使用中的object个数
•通过lru字段索引链表中的下一个/前一个page
```

前三点没什么好说的，大家都差不多。需要关注的是第四点，这里不像cpu partial那样通过next指针连接页表，而是通过lru字段：

```
struct page {
...
      /*
       * Third double word block
       *
       * WARNING: bit 0 of the first word encode PageTail(). That means
       * the rest users of the storage space MUST NOT use the bit to
       * avoid collision and false-positive PageTail().
       */
      union {
          struct list_head lru;   /* Pageout list, eg. active_list
                       * protected by zone_lru_lock !
                       * Can be used as a generic list
                       * by the page owner.
                       */
     ...
```

分配和释放

对象的分配和释放涉及到几个指针，分别是：

```
•p1: 对象的虚拟地址(void *)
•p2: 对象地址所对应的page(struct page*)
•p3: 对象所属的slab(struct page*)
•p4: 对象所属的cache控制体(struct kmem_cache*)
```

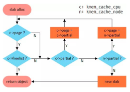
一个虚地址所对应的页首地址是是通过PAGE_MASK，因为页是对齐的，但需要注意页首地址并不是page指针所指向的地方。p1->p2的转换通过virt_to_page实现。

p2->p4可以通过page->slab_cache得到，这也是p1->p4函数virt_to_cache的操作。

分配

对象的分配，不考虑特殊情况的话(比如超过N页的对象直接通过伙伴系统分配)，一般流程如下：

1.kmem_cache_cpu->freelist不为空，直接出链返回；
2.kmem_cache_cpu->page->freelist不为空，则出链，更新cpu_slab->freelist，然后返回；
3.kmem_cache_cpu->partial不为空，取出第一个slab，更新cpu_slab的freelist和page，取出对象然后返回；
4.kmem_cache_node->partial不为空，取出第一个，类似3更新cpu_slab的freelist和page并返回；
5.上面都是空的，则通过伙伴系统分配新的slab，挂到kmem_cache_cpu中，然后goto 1；

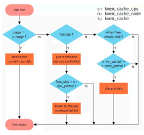


释放

对象的释放相对复杂，和释放之前对象所处的位置以及释放后cache情况有关。假设待释放的object地址为p1，p1对应的page为p2，p1对应的slab为p3，参考上面的几个指针定义，大致有以下路径：

1.p3就是当前CPU的kmem_cache_cpu->freelist所对应的slab，即p1位于当前cpu的kmem_cache_cpu->freelist所在的page中(p2 == cpu_slab->page)，此时可以直接释放到freelist上并返回；
2.p3位于当前CPU的kmem_cache_cpu->partial链表中，或者其他CPU的kmem_cache_cpu->freelist/partial中。此slab处于冻结状态，将p1链入p3->freelist中；
3.p3位于kmem_cache_node->partial链表中，此时释放分为两种情况：3.1 释放p1后，p3的状态为半满，此时正常将p1链入p3的freelist中。3.2 释放p1后，p3的状态为全空，此时除了将p1链入p3的freelist以外，还需要判断node中slab数是否超过规定值(node->nr_partial >= min_partial)，如果超过则需要将p3移出node->partial链表，并将p3释放给伙伴系统。
4.p3是一个全满的slab，不被任何kmem_cache管理。释放后p3变成一个半满的slab(更新freelist)，同时p3会被加入到当前CPU的kmem_cache_cpu.partial中。加入之前需要判断cpu partial中的空闲对象是否超过了规定值(partial.pobjects > cachep.cpu_partial)，并进行相应的处理：4.1 如果没超过，直接链入cpu partial即可 4.2 如果超过，则将cpu partial中所有slab解冻，将其中所有半满的slab交由node进行管理；将其中所有空的slab回收给伙伴系统；最后再将slab链入到partial中。

其中3的判断是为了避免node partial中存放了太多空slab；4的判断是为了避免cpu partial中存放太多空slab以及加快搜索速度。


slab free
c: kmem_cache_cpu
n: kmem_cache_node
s: kmem_cache
page == c->page ?
full slab ?
when free, empty slab ?
free to the current cpu slab
put it onto the per cpu partial list
n->nr_partial >= s->min_partial ?
free_objs > n->cpu_partial ?
discard slab
Move to the per node partial list
free object
Y
N

slab分配和释放的过程大致就是这样，上面的图主要来自窝窝的smcdef大神，其中还有一张大图可以配合观看理解：http://www.wowotech.net/content/uploadfile/201803/4a471520078976.png

调试

slab分配器是如此复杂，因此Linux内核中提供了很多调试措施。开启特定的调试编译选项后可以在object前后加上red zone检测越界，也可以检测slab的引用错误。

在用户态中，我们可以通过vfs进行调试，比如可以用slabinfo或者slabtop命令查看当前的slab分配情况，这些命令实际上是读取了/proc/slabinfo信息以及`/sys/kernel/slab/*`中的信息。

slabtop输出示例：

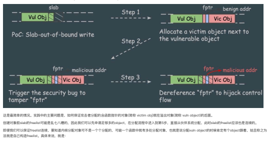
```
 Active / Total Objects (% used)    : 25864761 / 26174849 (98.8%)
 Active / Total Slabs (% used)      : 661098 / 661098 (100.0%)
 Active / Total Caches (% used)     : 93 / 158 (58.9%)
 Active / Total Size (% used)       : 4921033.80K / 5046143.93K (97.5%)
 Minimum / Average / Maximum Object : 0.01K / 0.19K / 295.08K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
18380583 18380583 100%    0.10K 471297       39   1885188K buffer_head
1785462 1711901   0%    0.19K  42511       42    340088K dentry
1705350 1629533   0%    1.06K  56845       30   1819040K ext4_inode_cache
589064 567016   0%    0.57K  21038       28    336608K radix_tree_node
530112 495734   0%    0.06K   8283       64     33132K kmalloc-64
475728 429025   0%    0.04K   4664      102     18656K ext4_extent_status
357632 355306   0%    0.06K   5588       64     22352K pid
258944 245861   0%    0.03K   2023      128      8092K kmalloc-32
247494 246414   0%    0.20K   6346       39     50768K vm_area_struct
231794 230768   0%    0.09K   5039       46     20156K anon_vma
174780 169836   0%    0.13K   5826       30     23304K kernfs_node_cache
164224 159505   0%    0.25K   5132       32     41056K filp
146688 143610   0%    0.02K    573      256      2292K kmalloc-16
120480 117291   0%    0.66K   2510       48     80320K proc_inode_cache
101376  97721   0%    0.01K    198      512       792K kmalloc-8
 86310  85793   0%    0.19K   2055       42     16440K cred_jar
 78122  75197   0%    0.59K   1474       53     47168K inode_cache
 48512  46984   0%    0.50K   1516       32     24256K kmalloc-512
```

直接从实现角度分析也许理解得不是很深刻，下面就来看几个实际的攻击案例，它们都巧妙地利用了上面提到的slab分配器的特性进行内存布局。

案例1：内核堆溢出漏洞利用

第一种类型是内核堆溢出漏洞。假如我们使用kmalloc分配了一个大小为30字节的对象，根据配置不同很可能会使用到名为kmalloc-32的kmem_cache去进行分配。因此，如果我们使用对象时写入超过32字节的数据，就可能产生堆溢出。

堆溢出的直接后果就是覆盖了slab中后面一个object块的数据，如果后面的object对象中被覆盖的部分包含函数指针，那么就有可能劫持内核控制流，跳转到任意地址，如下图所示：


slab
Vul Obj
PoC: Slab-out-of-bound write
Step 1
fptr
benign addr
Vul Obj
Vic Obj
Allocate a victim object next to the vulnerable object
Step 2
fptr
malicious addr
Vul Obj
Vic Obj
Trigger the security bug to tamper "fptr"
Step 3
fptr
malicious addr
Vul Obj
Vic Obj
Dereference "fptr" to hijack control flow

这是最简单的情况。实践中的主要问题是，如何保证攻击者分配的含函数指针的对象(简称 victim obj)就在溢出对象(简称 vuln object)的后面。

创建对象前slab的freelist可能是乱七八糟的，因此我们可以先申请足够多的object，在分配流程中进入到第5步，直接从伙伴系统分配，此时slab的freelist应该也是连续的。

即使我们可以保证freelist连续，要知道内核分配对象可不是一个个分配的，可能一个函数中就有多处分配对象，也就是说分配vuln object的时候肯定有个object跟着，姑且称之为xo。这时候如何利用呢？一个办法就是自己构造freelist，具体来说，就是：

1.依次分配3个object(同样的slab) A、B、C；此时freelist指向D的下一个object(D)；
2.我们希望freelist为A->C->B，因此需要依次释放B、C、A
3.这样，再次申请vuln object时其进入A，跟着的xo就进入到了C，我们的victim objcet就可以进入B，即在A的后面

这样，只要有合适的堆占位对象(如tty_struct)，就能稳定利用这类堆溢出漏洞了。