Demystifying Dirty COW: An Overview of the Vulnerability and its Consequences

Florian Nolte
School of Computation, Information and Technology
Technical University of Munich
Munich, Germany
florian.nolte@tum.de

***Abstract*—The vulnerability known as Dirty COW (designated CVE-2016-5195) enables attackers to escalate their local privileges. It is based on a flaw in Linux's memory management subsystem that can be abused through a race condition. This paper provides a complete overview of what Dirty COW is. It explores its origins and offers technical insights into how the vulnerability works. It highlights how it can be exploited, how it can be fixed and mitigated and also recognizes its history and impact. It will also discuss why vulnerabilities like Dirty COW are possible in the first place.**

***Index Terms*—Linux, memory management, Dirty COW, CVE-2016-5195, Container escape, GUP**

I. Introduction

Securing computer systems is of crucial concern. Even minor software errors can have enormous consequences. It can be a matter of a single character leading to a chain of bugs and vulnerabilities. As technology evolves, it becomes increasingly harder to keep track of all its intricacies.

As software becomes more complex, the attack surface increases accordingly. One vulnerability that has sprung out of the Linux kernel is called Dirty COW. It abuses a race condition[1] in Linux's memory management subsystem, enabling writing to otherwise write-protected pages. This has severe consequences, as attackers can use this vulnerability to escalate their privileges, potentially harming the system, its users, and their data.

II. Related Work

This paper provides a unique and complete look at the vulnerability known as Dirty COW. There is a write-up concerning the technical aspects of the original exploit [1]. Said blog post provides an in-depth analysis and mainly targets people knowledgeable about the Linux kernel.

The "official in-official" website https://dirtycow.ninja/ is a good entry point into the topic, especially if the goal is to ultimately exploit the vulnerability [2].

There is another write-up regarding a related bug, called *HugeDirtyCOW*, written by the researchers that have discovered the Dirty COW variant in the first place [3].

This paper does not exclusively focus on the inner workings of the Dirty COW exploit but also considers the overall context, history, and discusses why the exploit was possible in the first place.

[1]Race conditions occur when multiple actors modify the same resource simultaneously. This can result in unexpected effects (like data corruption).

III. Background

A. The "everything is a file" property of Linux

The *everything is a file* property of Linux is a fundamental concept that lets users and programmers treat various resources and devices like files. Those resources can be accessed using file descriptors. They must consequently support operations like `open`, `read`, `write`, `llseek`, and `release` [4].

B. Virtual Dynamic Shared Objects

The virtual Dynamic Shared Objects (vDSO) mechanism allows the kernel to expose certain frequently used kernel space functions to the user space. Processes can invoke those functions without switching into kernel space, which would otherwise take up precious time and resources [5].

In Linux, this is implemented using a small library automatically mapped into all user-space applications' address-space. The vDSO is called by *libc* itself, and thus application programmers usually do not notice that the mechanism is in place [5].

C. The setuid bit

In Linux, if the *setuid*-bit is set, an executable file is run with the privileges of the file's owner. If a file is owned by *root* (e.g., `/usr/bin/su` [6]), it will be executed with *root* privileges (no matter which user executes it) [7].

D. How shared library functions are called

To execute a function from a dynamically linked library like the C Standard Library, it must first be loaded into memory. This usually happens by invoking the `mmap()` system call internally (this is not noticed by the programmer), creating a virtual memory mapping from the *libc.so* file [8].

IV. Memory Management in Linux

A. Memory management

Each process has its own virtual address space. It is divided into pages, typically *4KB* in size. The processor (precisely: the *memory management unit*, a (typically) physical unit within

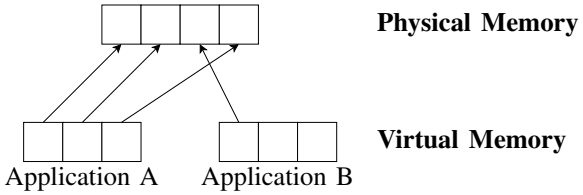

Figure 1. Representation of the virtual and physical address space

the processor) dynamically maps addresses from the virtual space to the actual physical computer memory using *page tables*. Each process has its own page table. A page table consists of many *page table entries* (PTE). The processor will notify the operating system of a *page fault* if a translation fails. Reasons for failure could be a mapping to an invalid address or the access control information in the *page table entry* might forbid a particular operation (writing to a page marked as read-only, for instance) [9].

This mechanism is displayed in Figure 1. Each square represents a *page*. An arrow indicates a mapping from a virtual memory page to a physical one. These mappings are stored in each process's *page table*. Note that not all virtual memory pages have to be mapped to a physical one: They could be swapped out into a file, for instance.

B. The copy-on-write mechanism

Copy-on-write (COW) is a mechanism that allows postponing the creation of the actual copy in favor of a virtual one. The actual copy will only be created once a process attempts to write into the virtual one. This approach decreases the resource usage necessary by omitting unnecessary data duplication.

Assume a process is duplicated using `fork()`[2]. The parent and the child process have their own virtual address space independent from each other. If one process writes into its memory, the other process's memory contents will not be affected. The kernel employs the COW mechanism to reduce the resources needed to duplicate the memory.

In practice, the original content of the memory will remain in the physical memory unchanged. The page table entries for all pages of **both** processes will be set to be read-only. This is shown in Figure 2. If one of the processes attempts to write into a page, a page fault will occur. The kernel will notice said page fault, copy the respective page, and change the page table entry to point to the new, cowed page instead, as shown in Figure 3. This entry will now allow writing [10].

C. The `mmap()` system call

That mechanism is used by various functions, like the `fork()`-function mentioned above [11] or some file systems, such as BTRFS [12] or ZFS [13]. The instance that will be particularly interesting for this paper is the `mmap()`-function. It allows a process to create a virtual memory mapping from

[2]The `fork()` system call was often used to realize parallel programming. The kernel duplicates the calling process's entire memory, essentially spawning a new (almost) identical one.

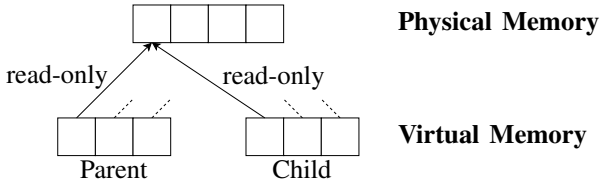

Figure 2. State of the memory after duplicating memory using COW

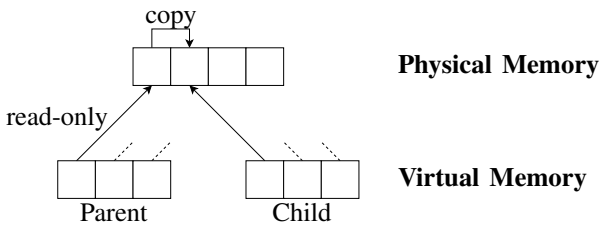

Figure 3. State of the memory after breaking COW

a *file descriptor*[3]. Using the `MAP_PRIVATE`-flag, a private mapping of a read-only file descriptor can be created using the COW mechanism. This private mapping can be written into. Changes to the private mapping will not affect the original [14]. This is shown in Figure 4. Note how `mprotect()` has to be called first. Otherwise, a segmentation fault[4] would occur since the copy-on-write mechanism respects the access flags of the original page.

D. Perspectives - User and kernel space

The user space is the protection layer where most programs will run. This is independent of the program's privilege: Even a root process runs in user space. To interact with the operating system, it will use its API by invoking system calls. These will be executed in kernel space.

If `fork()` is invoked, the kernel will not copy the process's memory immediately - as discussed previously. The user does **not** take notice of that mechanism. From the user's perspective, those pages exist immediately. They can be accessed and written to. Even system calls like `mprotect()` can change their access flags.

The kernel, however, does not create those pages immediately. It can be said that the kernel "lies" to the user by acting as if it has done something that it has not yet done. Luckily,

[3]A file descriptor is a handle for an opened file. They are managed by the operating system. Each process has its own set of file descriptors.

[4]If the kernel is unable to resolve a page fault (since the program did try to perform invalid memory access), it will throw a segmentation fault in the program.

```
int fd = open("file.txt", O_RDONLY);
char *data
  = mmap(NULL, 4096, PROT_READ, MAP_PRIVATE, fd, 0);
mprotect(data, 4096, PROT_READ | PROT_WRITE);
data[0] = 'A';
```

Figure 4. Using the `mmap()` function with the `MAP_PRIVATE` to create a copy-on-write mapping.

the kernel is able to keep up this illusion by breaking COW just in time (using the MMU to cause a write fault).

The kernel is not entirely truthful in regard to the access flags as well. As discussed earlier, the COW mechanism relies on the access flags to be set to read-only in order to trigger a page fault in the kernel. So why would the call to `mprotect()` in Figure 4 not interfere with that? The answer is that the flags set in `mprotect()` only change the access flags of the page table entry and are not guaranteed to be passed on to the physical one. The kernel is free to flag the physical page as it likes as long as it handles the resulting page faults transparently.

V. The attack

In short, the Dirty COW vulnerability allows an attacker to write to a read-only file descriptor mapping by exploiting a race condition between writing to and dismissing a private memory mapping.

The vulnerability is comparatively uncomplicated to exploit. First, a read-only-*file descriptor* must be opened onto the target file. That file shall then be loaded into a private memory mapping using the `mmap()`-function. Now, two threads have to be executed in a loop:

- **Thread** A**:** Make the kernel write to the private mapping (e.g., by calling `write()` on `/proc/self/mem`). The reason why the kernel has to do the writing will be explored in VI-A. Under normal circumstances, this should not affect the original read-only file descriptor.
- **Thread** B**:** Call `madvise()` with the `MADV_DONTNEED` parameter on the mapping. This will tell the kernel that the resource can be freed (a more detailed breakdown of the `madvise()` function can be found in XIII-B). Subsequent access (by Thread A in this case) might take longer, as the pages must be reloaded.

Expected behavior: The kernel will write into a cowed version. The page will be discarded by Thread B. The original file descriptor is left unchanged. When attempting another write, a new cowed page will be created. This happens over and over again.

Observed behavior: After a few executions, the data is not written into the private mapping but into the original file descriptor instead, bypassing the write protection.

VI. The technicalities

This section will explore how the attack works and why the observed behavior differs from the expected behavior. All information presented in this chapter is related to Linux kernel version *4.8* (the last vulnerable version before Dirty COW was patched).

A. Making the kernel write into the process's memory

A crucial step of the exploit is that the process does not write into its memory itself. It rather makes *someone else* write into its memory (here: the kernel). That's also where the crux of the exploit lies: The kernels write will not go through the MMU but use a software-based alternative instead. There are multiple ways to achieve that: For instance, one can `write()` into `/proc/self/mem` [5]. `proc` is a *pseudo-filesystem* that provides an interface to the kernel. `self` refers to the calling process and `mem` to the memory of the respective process [15] [1].

The following two sections of code (Figure 5 and 6) produce the same result: both set `foo[0]` to the value `0xABCDEF`.

```
int *foo = (int *)calloc(LEN, sizeof(int));
foo[0] = 0xABCDEF;
```

Figure 5. Writing into the memory directly

```
int *foo = (int *)calloc(LEN, sizeof(int));
int fd = open("/proc/self/mem", O_RDWR);
int number_to_be_written = 0xABCDEF;
lseek(fd, (uintptr_t)foo, SEEK_SET);
write(fd, &number_to_be_written, sizeof(int));
```

Figure 6. Making the kernel write into the process's memory

From a technical perspective, however, there is a difference: In Figure 5, the memory access is handled by the MMU directly (see IV-A), whereas in Figure 6, the access is handled by *some* kernel module that will be discussed in the next section.

B. Exploring how the kernel writes into memory

To write to another process's virtual memory (from the kernel's perspective), the `access_remote_vm()` function must be called.

As described in the previous section, writing to `/proc/self/mem` is one option. The kernel will invoke the pseudo-filesystem's write operation and eventually invoke `access_remove_vm()`, as shown in Figure 7.

That function function will then invoke the `__get_user_pages_locked()` function, which goes on to call `__get_user_pages()` [16] [17].

C. Pinning a user page in physical memory

There are a few challenges to overcome when the kernel attempts to write into a process's virtual memory:

- Ensuring the virtual memory page is ready (handling page faults).
- Locating the virtual page in physical memory.

[5]The `write()`-function is a *syscall* and will be executed in kernel space consequently.

```
[mem definition from proc_mem_operations]
mem_write() [fs/proc/base.c]
mem_rw() [fs/proc/base.c]
access_remote_vm() [mm/memory.c]
__access_remote_vm() [mm/memory.c]
```

Figure 7. Call stack for execution of `access_remote_vm()` by writing to */proc/self/mem*

```
static long __get_user_pages(/*[...]*/)
{
 // [...]
 do {
  unsigned int foll_flags = gup_flags;
  // [...]
retry:
  page = follow_page_mask(vma, start,
   foll_flags, &ctx);
  if (!page || PTR_ERR(page) == -EMLINK)
  {
   ret = faultin_page(vma, start, &foll_flags,
    PTR_ERR(page) == -EMLINK, locked);
   switch (ret) {
    case 0:
         goto retry;
    case -EBUSY:
    // [...]
   }
  }
   // [...]
 } while (nr_pages);
// [...]
}
```

Figure 8. Simplified version of the `__get_user_pages()`-function

- Pin it into the kernel's memory (making sure it won't be unloaded when it might still be needed)
- Ensure that the calling process has sufficient permissions to act on that page

Those tasks are handled by the `__get_user_pages()`-function [16]. A version of the function that is vastly broken down for this paper is shown in Figure 8. The `follow_page_mask()`-function attempts to get the page in question. If that page cannot be acquired, it will return `NULL`. There can be many reasons for the function to fail, e.g., accessing the page of a private mapping that has not yet been created due to the nature of the copy-on-write mechanism (technically, the reason is that the caller requests write access to a write-protected page). In that case, the `faultin_page()`-function will simulate the page fault and create the missing page. On success, 0 is returned. The function will jump back to the retry label and attempt to load the (now existing) page again [1] [16].

D. Executing the copy-on-write mechanism

The creation of the COW page is handled in the `faultin_page()`-function. Figure 9 shows a simplified version of that function. It essentially works like a wrapper around the `handle_mm_fault()`-function, which does the heavy lifting (this is the same function that would be executed after a page fault by the MMU as well)—the inner workings of `handle_mm_fault()` are not of concern here. One important thing to know is that, on breaking COW, the original page is not guaranteed to be writable. Internally, the `maybe_mkwrite()` function might decide that the cowed page should stay read-only [16]. It is not relevant whether that is the expected case or not. The critical aspect is that it *might* be write protected.

`faultin_page()` does some flag-conversion, calls `handle_mm_fault()`, examines the result and returns an according to value. Note how in the lower part of the function, the `FOLL_WRITE`-flag will be removed if the `VM_FAULT_WRITE`-bit is set. That bit indicates that COW was broken. According to a code comment, the `FOLL_WRITE`-flag can be removed in that case: *"We can thus safely do subsequent page lookups as if they were reads."* [16].

Since the flag is passed as a reference (precisely: a *pointer*), the `__get_user_pages()`-function's `foll_flags` will be changed as well, effectively tricking the function into thinking read-only access was requested.

Why is the `FOLL_WRITE`-flag removed in the first place?

Taking a second look at `__get_user_pages()` (Figure 8), the execution would find itself in a fairly similar situation in the first and second run of the loop. In both cases, the function will ask for write access to the page. The available page might be read-only. As mentioned earlier, it is not guaranteed that the cowed page allows being written to right away [6]. Thus, `follow_page_mask()` might return `NULL`, and COW will be broken by `faultin_page()` again and again. The function removes the `FOLL_WRITE`-flag as a precaution to break this loop.

Why it does not matter that the page is read-only

For one, The kernel has the privilege to overcome such restrictions; second, the page table entries only exist in the context of the respective process. This write, however, will be executed by the kernel. The kernel function called `__get_user_pages()` trusts that it will only return pages with sufficient permissions.

E. Where it all goes wrong

This mechanism assumes that if COW is broken in the first execution of the loop, the cowed page will still be there in the second one. There are no checks in place to verify that, however.

Assume an attacker truncates the cowed page after returning from `faultin_page()`. Now, `__get_user_pages()` will start another attempt to get the cowed page, but this time without asking for write access. Since multiple entities can read from the same page (and the function never remembers that the page was previously cowed), there is no reason not to return the original page (it would be a copy-on-read otherwise) [1].

From an outsider's perspective, both scenarios are indistinguishable: `access_remove_vm()` requests a page that may be written to and, in both cases, might get back a page with write protection. Only, in the exploited case, the page returned is not a copy but the original one.

`mem_rw()` writes the data into the returned page, and *some* synchronization mechanism will synchronize the written data from the file mapping back into the file system [1].

VII. Fixing Dirty COW

Looking back at what happened, we see that a core issue is that it cannot be effectively differentiated between:

[6] A page being write-protected does not necessarily mean that it cannot be written to. When attempting modification of such a page, a *page fault* will be invoked by the MMU and handled by the kernel. The kernel *might* decide to do *some* preprocessing and go through with the write.

```
static int faultin_page(
  struct task_struct *tsk,
  struct vm_area_struct *vma,
  unsigned long address,
  unsigned int *flags,
  int *nonblocking)
{
  unsigned int fault_flags = 0;

  // populating fault_flags [...]

  ret = handle_mm_fault(vma, address, fault_flags);

  // processing ret, potentially returning [...]

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

Figure 9. Simplified version of the `faultin_page()`-function

1) Asking for the original, read-only page.
2) Asking for the (potentially also read-only) cowed page after it was *presumably* cowed.

The implemented fix involves remembering that COW was broken in a previous loop cycle. The `FOLL_COW` flag was introduced to make that happen [18].

VIII. Mitigating Dirty COW

Once a fix has been published, the recommended method to protect against the bug is to apply the respective security patch by upgrading the kernel version (more on this in XII).

Alternatively, the vulnerability can be overcome using the *systemd* service. It allows the user to disable specific system calls, like `sys_madvise`, vital to the Dirty COW exploit [19]. However, whether `madvise()` is the only way to exploit the vulnerability is unknown. Also, disabling a system call can lead to unexpected behavior and system instability.

IX. Exploring various attack paths

There are different methods to exploit the Dirty COW vulnerability. In general, one can differentiate between how the exploit invokes the vulnerable function and how the override of the write protection can lead to a privilege escalation.

A. Making the kernel write into the process's memory

The essence of the vulnerability lies within the GUP function. There are various ways to invoke it:

- **Writing to `/proc/self/mem`:** The kernel invokes the function when a process writes into the `/proc/self/mem`-file as described in the previous sections.
- **Using `ptrace()`:** This function is often used by debuggers, as it provides the means to observe and control the execution of another process [7]. This also involves writing into its memory [20] [21]. Internally, `ptrace()` calls the `ptrace_readdata()`-function to read from a process's memory. This function invokes `access_process_vm()`, which ends up calling `__access_remote_vm()` [22] [17].

[7]Note that the `ptrace()`-function is a *syscall*. It will thus be executed in kernel space [20].

B. Abusing the vulnerability to gain user privileges

This section describes how a thread actor could potentially go from writing to a read-only file to escalating their user privileges:

1) Impersonating a user by overwriting a file with a set setuid-bit: As explained in III-C, a file with the *setuid* bit will be executed with the permissions of its owner. Some executables are owned by root, like `/usr/bin/passwd` or `/usr/bin/su`. For instance, overwriting these files with a malicious payload that opens a shell window can grant an attacker *root* access. Of course, this technique can also be used to impersonate non-root users, given that they own at least some readable and executable file with the *setuid* bit set.

2) Patching parts of libraries loaded into memory: Considering how library functions are loaded into the memory (see III-D), another attack path would be to use Dirty COW to write to the otherwise read-only memory mapping.

For instance, the address of the *getuid* function can be located in the processes memory using the `dlsym()` function [23]. When executing `/usr/bin/su` (which will always be run with *root* permissions as described in III-C), the `su` process will check the result of the `getuid()` call. If it returns a non-zero value, `su` will ask for *root* credentials. If, however, the function responds with *0*, `su` will spawn a superuser shell [24] [25].

Using the Dirty COW exploit, an attacker can overwrite a part of the `getuid()` function always to return 0, subsequently tricking the `su` executable into granting them *root* privileges [26].

3) Modifying shared memory: Dirty COW allows an attacker to overwrite shared memory that would be read-only. Specifically, this can be used to overwrite the shared memory segment used by some vDSO functions (see III-B). This will be further explored in X-B.

4) Writing into `/etc/passwd`: The `/etc/passwd` file contains information about user accounts. Everyone can read from that file. Historically, it had the encrypted passwords of the respective user accounts. This was later changed. Passwords are now replaced with an *x* and are stored in the `/etc/shadow` file instead. While discouraged due to security concerns, keeping an encrypted password in that file is still possible [27].

An attacker can use the Dirty COW vulnerability to append a new user entry into the `/etc/passwd` file with a password they know (or no password). That newly created user can have arbitrary permissions [28].

X. Container escape

One exciting aspect of Dirty COW is that, depending on the circumstances, it might even escape a virtualized container.

A. Linux kernel namespaces and containers

Before we can explore how Dirty COW escapes a container, we must first understand how containers work: In Linux, a so-called *namespace* allows a process to run in an isolated environment. Changes to resources are only visible to members of that same *namespace* but are invisible to processes outside of it [29].

There are different types of namespaces, each serving another goal. The *user namespaces*, for instance, isolates security-related identifiers and attributes (e.g., user and group IDs, the root directory, keys, and capabilities [30]). On the other hand, the *mount namespace* provides each container with an isolated view of the file system [31].

By combining different types of namespaces, one can create an isolated container to run a set of applications [32]. While seemingly isolated, those processes still run on the host system at the end of the day.

B. How to escape a namespace?

As described in III-B, every user space process can access a few functions stored in the vDSO shared memory. The addresses of those functions can be determined using the `getauxval()` function [5]. For obvious reasons, shared memory is read-only to all processes.

Using the `ptrace()` variant of the Dirty COW vulnerability, an attacker can overwrite one of the functions with a malicious payload [33]. Once the function is called, it will be executed in the context of the process that called it. If a process outside the Container eventually calls one of the compromised functions, the malicious shell code will be run, and the attacker has successfully escaped the Container.

C. How to escape Docker?

Docker is a platform that allows for easy deployment of applications inside containers. They offer a standardized environment and isolation from the host system. In Linux, these containers work using the *namespaces* mentioned above [34].

This means that, for one, an attacker can use the aforementioned approach to write to the vDSO memory and escape the Docker container [33]. Apart from that, there are other Docker-specific approaches to escape the container:

Using shared resources: An attacker can escape the Docker container if it uses shared resources. A directory on the host system could have been mounted as a read-only directory to the guest. An attacker could open a file descriptor on a file in the said directory and write to it using the exploit [35].

Using user namespaces: Docker allows the creation of so-called *user namespaces*. These are not to be confused with Linuxes' *namepsaces*. They give the *root*-User of the guest system the ability to access the host system as a regular user. If enabled, an attacker can first gain *root* privileges inside the guest system, use these to escape into the host system as a regular user and rerun the exploit to gain *root* privileges in the host system [35].

Note that this is not specific to Dirty COW; any root privilege escalation vulnerability might be exploited to accomplish that.

XI. HugeDirtyCOW

HugeDirtyCOW is a vulnerability quite similar to Dirty COW. It uses so-called *transparent huge pages* instead of regular ones. This section will briefly overview the vulnerability, parallels to Dirty COW, and background information about huge pages.

A. Huge Pages

Typically, pages are allocated in chunks of $4KB$. The more pages there are, the more time it takes to find them and the more space it takes to store their respective allocations. Hence, a mechanism for larger-than-normal pages was introduced. These are often referred to as *Huge pages* (Linux), *Super Pages* (BSD), or *Large Pages* (Windows) [36].

The size of huge pages is dependent on the respective architecture. *x86* CPUs, for instance, typically support $4k$, $2M$ and $1G$ page sizes. One way to use Huge Pages is by calling `mmap()` with the `MAP_HUGETLB`-flag [14] [37].

B. Transparent Huge Pages

Transparent Huge Pages (THP) run on top of regular huge pages and offer automatic promotion and demotion of huge pages. They can be used with `mmap()`s `MAP_ANONYMOUS`-flag, `tmpfs`, and `shmem` [38]. They can also be broken back into regular pages [3].

C. Dirty COW in Transparent Huge Pages

After Dirty COW was patched, a similar patch was applied to THPs as well [39]. This fix, however, turned out to be incomplete.

1) The original fix: A function called `can_follow_write_pmd()` was added. It follows the same logic as the Dirty COW fix employing the `FOLL_COW`-flag.

2) The bug: Unlike regular pages, Huge Pages can be marked dirty even without breaking COW by using the `touch_pmd()`-function (called by `follow_page_mask()`. Due to this logic, whenever a huge page is read using `get_user_pages()`. It will become dirty. This breaks the logic of the fix above and allows an attacker to access an otherwise read-only page [3].

3) Exploiting the vulnerability: Huge pages are much more limited than regular ones. A few mappings that can be targeted are:

- **`fork()`:** Anonymous mappings after *fork*ing a process. The parent could write into the child's process memory or vice versa [3].
- **Huge zero page**: Wen a read-fault on a huge page occurs, the so-called *huge zero page* will be returned. For optimization purposes, only one such page exists that will be mapped to all processes. That page is read-only but can be written into using the exploit [3].
- **Shared memory**: As file mappings into huge pages are supported by shared memory, an attacker could overcome their write protection using the HugeDirtyCOW exploit.

Fixing the fix: To mitigate the issue, the `touch_pmd()` will only make a page dirty if it is actually written into [3].

XII. History of Dirty COW

This section will discuss the history of Dirty COW. An overview is shown in Figure 10.

A. Race condition in `get_user_pages()`

In 2005, Linux Torvalds attempted to fix a race condition in the Linux kernel that, according to him, could have prevented the Dirty COW vulnerability [40] [18]. Unfortunately, that fix was later undone, as it caused compatibility issues with the S/390 platform (which was used by, e.g., IBM's mainframe) [41] [42].

B. Dirty COW becomes exploitable

With Linux kernel version 2.6.22 (released Jul 8, 2007), the Dirty COW vulnerability becomes exploitable for the first time [43] [2]. Almost prophetically, the commit message for that version read *"I'm sure somebody will report a "this doesn't compile, and I have a new root exploit" five minutes after release, but it still feels good ;)"* [44]

C. Discovery of CVE-2016-5195

On Oct 11, 2016, security researcher Phil Oester reported the vulnerability now known as Dirty COW. They found the vulnerability by analyzing a suspicious payload sent to a server. The fact that the payload was compiled using *gcc* version 4.8 (which was released on Mar 22, 2013) might indicate that the bug was already being actively exploited in the past [45] [46].

The occurrence was disclosed to the Linux kernel development community and later fixed. Unfortunately, it was forgotten to apply the same check to the logic handling huge memory pages at first. They were fixed a few months later [39].

D. Discovery of CVE-2017–1000405

Unfortunately, that fix was incomplete, which led to the Discovery of CVE-2017–1000405, dubbed HugeDirtyCOW, by the cyber security company *Bindecy* [3]. A patch was applied a few days later to the Linux kernel [47].

XIII. Linux versus Posix

POSIX (*Portable Operating System Interface*) is a standard that specifies the interface between the operating system and applications. It portably enables software to run on all POSIX-compliant systems [48]. Linux is primarily compliant with POSIX, but there are some exceptions [49].

A. `madvise` in POSIX

One difference, particularly relevant to the Dirty COW vulnerability, lies within the definitions of the `madvise` function. The `MADV_DONTNEED`-flag behaves differently than expected. POSIX describes the flag as follows: *"The application expects that it will not access the specified address range shortly."* [50]. It also specifies that the function *"shall not affect the semantics of access to memory"* [50]. In other words: No data will be lost. The only aspect that might change is the latency required for page access. A sample code displaying that behavior can be found in Figure 11. See how the memory can still be accessed after executing `POSIX_MADV_DONTNEED`, and no information is lost.

Date	Event
Aug 1, 2005	A race condition in `get_user_pages()` was discovered and subsequently fixed.
Aug 3, 2005	The previous fix was undone due to compatibility issues.
Jul 8, 2007	Dirty COW becomes exploitable.
Oct 11, 2016	CVE-2016-5195 "Dirty COW" was discovered.
Oct 18, 2016	A fix for Dirty COW was applied to the kernel.
Jan 25, 2017	The respective fix for huge pages was also applied.
Nov 22, 2017	HugeDirtyCOW (CVE-2017–1000405) was discovered.
Nov 27, 2017	A fix for HugeDirtyCOW was applied to the kernel.

Figure 10. Overview of the history of the Dirty COW vulnerability

```
char *some_text = "Lorem ipsum dolor sit\n";
int fd = memfd_create("my_fd1", 0);
write(fd, some_text, strlen(some_text));

char *map = mmap(NULL, strlen(some_text) + 1,
    PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

char *text = "<copied data>";
memcpy(map, text, strlen(text));

printf(map);
posix_madvise(map, strlen(text),
    POSIX_MADV_DONTNEED);
printf(map);
```

Output:

```
<copied data>olor sit
<copied data>olor sit
```

Figure 11. Sample code invoking the POSIX-version of `madvise`

B. `madvise` in Linux

Linux, however, implemented the function a little differently. The data in the copy will get discarded. The behavior is shown in Figure 12. A trace of the functions being invoked when calling `madvise()` with `MADV_DONTNEED` is presented by Figure 13. Instead of, e.g., flagging a page as a candidate for swapping, the kernel frees the page right away.

This behavior is (somewhat) acknowledged on the man page. It describes that *"[a]fter a successful MADV_DONTNEED operation, the semantics of memory access in the specified region are changed: subsequent accesses of pages in the range will succeed, but will result in either repopulating the memory contents [...] or zero-fill-on-demand pages"* [51].

```
char *some_text = "Lorem ipsum dolor sit\n";
int fd = memfd_create("my_fd2", 0);
write(fd, some_text, strlen(some_text));
char *map = mmap(NULL, strlen(some_text) + 1,
    PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

char *text = "<copied data>";
memcpy(map, text, strlen(text));

printf(map);
madvise(map, strlen(text), MADV_DONTNEED);
printf(map);
```

Output:

```
<copied data>olor sit
Lorem ipsum dolor sit
```

Figure 12. Sample code invoking the Linux-version of `madvise`

```
[madvise syscall is defined as do_madvise]
do_madvise() (mm/madvise.c)
madvise_vma_behavior() (mm/madvise.c)
madvise_dontneed_free() (mm/madvise.c)
madvise_dontneed_single_vma() (mm/madvise.c)
zap_page_range_single() (mm/memory.c)
unmap_single_vma() (mm/memory.c)
unmap_page_range() (mm/memory.c)
```

Figure 13. Call stack for execution of `madvise` with the `MADV_DONTNEED` parameter [17].

C. Impact on Dirty COW

This non-standard behavior makes Dirty COW possible in the first place. If Linux adhered to POSIX, the cowed pages would not have been removed, preventing the race condition in the first place [1].

There is some discussion regarding the nature and reasoning behind this non-standard behavior. Some speculate it could have come from the now retired operating system *Tru64*. Unlike Linux, *Tru64*'s man page is quite precise when it comes to the intentions of the `MADV_DONTNEED`-flag: *"MADV_DONTNEED tells the system to discard the contents of any pages currently allocated for the process"* [52] [53]. Unfortunately, we could not find official statements from the Linux community regarding that non-standard behavior.

XIV. The GUP Family

There are a few functions that are similar to the `get_user_pages()`-function. Those functions are commonly referred to as *the GUP family*. This is a non-exhaustive list of GUP functions: `get_user_pages_fast()`, `get_user_pages_locked()`, `get_user_pages_unlocked()`, and `get_user_pages_remote()`.

A. Problems with GUP

GUP functions aim to allow kernel code to work with user space virtual mappings. However, they are not always trivial to use, especially not safely. Tracking down bugs can be tedious, especially since the write-back of changes can be quite prone to errors [54]. A fundamental problem for the kernel is to keep track of references made to pages using GUP functions [55].

Also, `get_user_pages()` operates within the kernel's address space. However, the improper use of `get_user_pages()` can lead to issues such as data corruption or kernel crashes. For instance, there was a bug that could lead to a page pointer being taken over by another process. This eventually became known as CVE-2020-29374 (see XV-B) [56].

XV. Similar vulnerabilities

Determining similar vulnerabilities is challenging, as Dirty COW is hard to classify.

A. Classifying Dirty COW

There are a few categories the bug will fit into:

- **Race Condition:** Dirty COW would not work if the `get_user_pages()` function would not run concurrently.
- **Memory Management Bug:** Ultimately, the bug is caused by how Linux handles virtual memory pages.
- **Privilege Escalation Vulnerability:** Dirty COW allows attackers to escalate their privileges. Writing to a read-only page can be considered a privilege escalation. Additionally, we were able to show different methods to gain actual *root* privileges (see IX).
- **Write-back cache issue:** Even though this mechanism was not discussed here, the Dirty COW vulnerability allows the introduction of unintended changes into the write-back cache of a `mmap` file-based mappings.

B. Comparison to Similar Vulnerabilites

Some notable examples of exploits similar to Dirty COW are:

- **DirtyPipe**: Both vulnerabilities exploit how the kernel handles memory access. DirtyPipe manipulates pipe buffers to exploit a race condition in writing to a read-only file [57].
- **MacDirtyCow**: Exploits the way XNU handles the copy-on-write mechanism. This could lead to the execution of arbitrary code with kernel privileges. Several versions of iOS and iPadOS were vulnerable [58].
- **CVE-2020-29374**: Yet another issue with the `get_user_pages()` function that can cause unintended write access. [59]
- **CVE-2020-29368**: In the Transparent Huge Pages handler, an issue with the implementation of the copy-on-write mechanism can grant unintended write access due to a race condition in the `mapcount` check [60].
- **CVE-2022-3623**: Another bug in the implementation of the GUP functions that could lead to a race condition allowing remote code execution [61].

XVI. Incident Response

This section will explore the reactions from different communities to the newly found Dirty COW vulnerability.

A. Linux Kernel Development Community

The commit message was not particularly meaningful when the fix was introduced to Linux's kernel. Linus Torvalds explained that an "ancient bug" (referring to XII-A) was fixed without ever mentioning that this commit fixed a significant security flaw [18].

This course of action was not met without criticism. A point can be made that security through obscurity should be discouraged. Experts argue that the users of Linux should have been appropriately informed when the vulnerability was fixed so that they could update their system immediately. Instead, users had to rely on other sources to learn about the issue. This practice, however, is not new. Kernel developers argue that every new release fixes many security issues that haven't even been publicly noticed. Highlighting those vulnerabilities might draw attackers' attention, making exploitation more likely [62].

B. Software Vendors

Software vendors (in this case: distro developers) reacted fairly quickly. Ubuntu/Canonical released a fix on Oct 20, 2016 [63], openSUSE on Oct 21, 2016 [64], Red Hat on Oct 24, 2016 [65], and Fedora on Oct 23, 2016 [66] to name a few.

C. IT Community

With its catchy name, logo, website, and online shop, Dirty COW became an entirely branded bug [2]. Numerous IT-related magazines have published articles about vulnerability, mainly on a surface level.

D. Impact of Dirty COW in the real world

It is hard to estimate whether Dirty COW was abused to carry out attacks against organizations. We were unable to find any reports of incidents involving the bug.

However, the impact of Dirty COW on devices running the Android operation system should not be understated. Considering that Android is based on Linux, the vulnerability allows users to gain *root access* to their devices, offering greater control over the hard -and software [67].

XVII. Discussion

This exploit shows that even widespread open-source operating systems are not immune to critical faults. The fact that the bug was not discovered for almost ten years shows how imperative peer-reviewing code is. It points out how continuous testing specifically for race conditions cannot be understated.

As mentioned previously, two big factors allowed the vulnerability to exist in the first place:

1) Linux not conforming to POSIX.
2) The state of the memory management system.

While we cannot comment on Linuxes compliance to POSIX, it is essential to acknowledge the challenges of Memory Management. That system has to support many mechanisms like demand paging, copy-on-write, swapping, (file) mappings, page caching, etc. These should work in software (GUP) and on hardware components (MMU), as well as different platforms and architectures. On top of that, it has to work concurrently and efficiently as well. Implementing all of those in a single system is a considerable challenge. Bugs in memory management are, therefore, not surprising. Most of them have minimal impact. Some of them might corrupt data. Others even pose a security risk.

XVIII. Conclusion

Dirty COW is a critical vulnerability that allows attackers to escalate their privileges. We have explored how the kernel handles memory access and how a race condition could eventually trick it into writing to a range of memory it was not supposed to. We have looked at different attack vectors and ways to gain root privileges.

There aren't many ways to effectively secure a system from the bug nor are there mechanisms for detecting them. If that wasn't bad enough, even containers could not secure a system from the vulnerability.

Lastly, we have discussed a few factors that have led to the vulnerability being there in the first place. A point can be made that it could have been prevented in the first place by sticking to the POSIX standard. But hindsight is always easier than foresight.

Despite the quick response from the Linux team, the Dirty COW exploit reminds us that severe vulnerabilities can still go unnoticed for a long time. One can only imagine what we have not found yet.

References

[1] "Dirty cow and why lying is bad even if you are the linux kernel," https://chao-tic.github.io/blog/2017/05/24/dirty-cow, (Accessed on 06/02/2023).

[2] "Dirty cow (cve-2016-5195)," https://dirtycow.ninja/, (Accessed on 06/08/2023).

[3] ""huge dirty cow" (cve-2017–1000405) | by eylon ben yaakov | bindecy | medium," https://medium.com/bindecy/huge-dirty-cow-cve-2017-1000405-110eca132de0, (Accessed on 06/07/2023).

[4] L. Torvalds, "linux/fs/proc/base.c - linux kernel 4.8," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/proc/base.c?h=v4.8.

[5] *VDSO(7) Linux Programmer's Manual*, https://man7.org/linux/man-pages/man7/vdso.7.html, 08 2021.

[6] *su(1) — Linux manual page*, https://man7.org/linux/man-pages/man1/su.1.html, 12 2022.

[7] *setuid(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/setuid.2.html, 03 2021.

[8] "Linkers and dynamic linking," https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/lecture.php?topic=linkers, (Accessed on 05/29/2023).

[9] "Memory management," https://tldp.org/LDP/tlk/mm/memory.html, (Accessed on 05/28/2023).

[10] "get_user_pages() and cow, 2022 edition [lwn.net]," https://lwn.net/Articles/895439/, (Accessed on 06/12/2023).

[11] *fork(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/fork.2.html, 03 2021.

[12] "Btrfs - archwiki," https://wiki.archlinux.org/title/btrfs, (Accessed on 05/28/2023).

[13] "Oracle solaris zfs-administrationshandbuch," https://docs.oracle.com/cd/E19253-01/820-2313/zfsover-2/index.html, (Accessed on 05/28/2023).

[14] *mmap(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/mmap.2.html, 03 2021.

[15] *proc(5) — Linux manual page*, https://man7.org/linux/man-pages/man5/proc.5.html, 08 2021.

[16] L. Torvalds, *linux/mm/gup.c — Linux kernel 4.8*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/mm/gup.c?h=v4.8.

[17] ——, *linux/mm/memory.c — Linux kernel 4.8*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/mm/memory.c?h=v4.8.

[18] ——, *Linux kernel commit 19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619.

[19] "Mitigating dirtyc0w with systemd - linux.com," https://www.linux.com/news/mitigating-dirtyc0w-systemd/, (Accessed on 06/08/2023).

[20] *ptrace(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/ptrace.2.html, 03 2021.

[21] "Github - firefart/dirtycow: Dirty cow exploit - cve-2016-5195," https://github.com/firefart/dirtycow, (Accessed on 06/03/2023).

[22] L. Torvalds, *linux/kernel/ptrace.c — Linux kernel 4.8*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/ptrace.c?h=v4.8.

[23] *dlsym(3) — Linux manual page*, https://man7.org/linux/man-pages/man3/dlsym.3.htmll, 03 2021.

[24] *getuid(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/getuid.2.html, 03 2021.

[25] "util-linux/su-common.c at master · util-linux/util-linux · github," https://github.com/util-linux/util-linux/blob/master/login-utils/su-common.c, (Accessed on 05/29/2023).

[26] "dirtycow-mem.c · github," https://gist.github.com/scumjr/17d91f20f73157c722ba2aea702985d2, (Accessed on 05/29/2023).

[27] *passwd(5) — Linux manual page*, https://man7.org/linux/man-pages/man5/passwd.5.html, 03 2021.

[28] "dirtycow/dirty.c at master · firefart/dirtycow · github," https://github.com/FireFart/dirtycow/blob/master/dirty.c, (Accessed on 05/29/2023).

[29] *namespaces(7) — Linux manual page*, https://man7.org/linux/man-pages/man7/namespaces.7.html, 08 2021.

[30] *user_namespaces(7) — Linux manual page*, https://man7.org/linux/man-pages/man7/user_namespaces.7.html, 08 2021.

[31] *mount_namespaces(7) — Linux manual page*, https://man7.org/linux/man-pages/man7/mount_namespaces.7.html, 08 2021.

[32] "Building a linux container by hand using namespaces | enable sysadmin," https://www.redhat.com/sysadmin/building-container-namespaces, (Accessed on 05/29/2023).

[33] "Dirty cow - (cve-2016-5195) - docker container escape," https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/, (Accessed on 05/28/2023).

[34] "What is docker?" https://www.ibm.com/topics/docker, (Accessed on 05/28/2023).

[35] S. Dulce, "Dirty cow vulnerability: Impact on containers," https://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers, (Accessed on 05/28/2023).

[36] "Hugepages - debian wiki," https://wiki.debian.org/Hugepages, (Accessed on 06/07/2023).

[37] "kernel.org/doc/documentation/vm/hugetlbpage.txt," https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, (Accessed on 06/07/2023).

[38] "Transparent hugepage support — the linux kernel documentation," https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html, (Accessed on 06/07/2023).

[39] L. Torvalds, *Linux kernel commit 8310d48b125d19fcd9521d83b8293e63eb1646aa*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8310d48b125d19fcd9521d83b8293e63eb1646aa.

[40] ——, *Linux kernel commit 4ceb5db9757aaeadcf8fbbf97d76bd42aa4df0d6*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4ceb5db9757aaeadcf8fbbf97d76bd42aa4df0d6.

[41] "Ibm system/390 - wikipedia," https://en.wikipedia.org/wiki/IBM_System/390, (Accessed on 06/08/2023).

[42] L. Torvalds, *Linux kernel commit f33ea7f404e592e4563b12101b7a4d17da6558d7*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f33ea7f404e592e4563b12101b7a4d17da6558d7.

[43] "Linux kernel version history - wikipedia," https://en.wikipedia.org/wiki/Linux_kernel_version_history, (Accessed on 06/08/2023).

[44] L. Torvalds, *Linux kernel commit 7dcca30a32aadb0520417521b0c44f42d09fe05c*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7dcca30a32aadb0520417521b0c44f42d09fe05c.

[45] "Gcc releases - gnu project," https://gcc.gnu.org/releases.html, (Accessed on 06/08/2023).

[46] "Linux users urged to protect against 'dirty cow' security flaw | v3," https://web.archive.org/web/20171123180501/https://www.v3.co.uk/v3-uk/news/2474845/linux-users-urged-to-protect-against-dirty-cow-security-flaw, (Accessed on 06/08/2023).

[47] L. Torvalds, *Linux kernel commit a8f97366452ed491d13cf1e44241bc0b5740b1f0*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a8f97366452ed491d13cf1e44241bc0b5740b1f0.

[48] "Posix - wikipedia," https://en.wikipedia.org/wiki/POSIX, (Accessed on 06/03/2023).

[49] D. Locke, "Posix and linux application compatibility design rules," 01 2005.

[50] *posix_madvise(3) — Linux manual page*, https://man7.org/linux/man-pages/man3/posix_madvise.3.html, 03 2021.

[51] *madvise(2) — Linux manual page*, https://man7.org/linux/man-pages/man2/madvise.2.html, 03 2021.

[52] B. Cantrill, "A crime against common sense," http://dtrace.org/blogs/bmc/2018/02/03/talks/, (Accessed on 06/09/2023).

[53] "nmadvise - tru64," https://nixdoc.net/man-pages/Tru64/man3/nmadvise.3.html, (Accessed on 06/11/2023).

[54] "The trouble with get_user_pages() [lwn.net]," https://lwn.net/Articles/753027/, (Accessed on 06/11/2023).

[55] "get_user_pages() continued [lwn.net]," https://lwn.net/Articles/753272/, (Accessed on 06/11/2023).

[56] "Patching until the cows come home (part 1) [lwn.net]," https://lwn.net/Articles/849638/, (Accessed on 06/12/2023).

[57] "The dirty pipe vulnerability — the dirty pipe vulnerability documentation," https://dirtypipe.cm4all.com/, (Accessed on 06/06/2023).

[58] "Cve - cve-2022-46689," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-46689, (Accessed on 06/10/2023).

[59] "Nvd - cve-2020-29374," https://nvd.nist.gov/vuln/detail/CVE-2020-29374, (Accessed on 06/10/2023).

[60] "Nvd - cve-2020-29368," https://nvd.nist.gov/vuln/detail/cve-2020-29368, (Accessed on 06/10/2023).

[61] "Nvd - cve-2022-3623," https://nvd.nist.gov/vuln/detail/CVE-2022-3623, (Accessed on 06/11/2023).

[62] "Dirty cow and clean commit messages [lwn.net]," https://lwn.net/Articles/704231/, (Accessed on 06/08/2023).

[63] "Usn-3104-1: Linux kernel vulnerability | ubuntu security notices | ubuntu," https://ubuntu.com/security/notices/USN-3104-1, (Accessed on 06/08/2023).

[64] "[security-announce] suse-su-2016:2585-1: important: Security update for the linux kernel - opensuse security announce - opensuse mailing lists," https://lists.opensuse.org/archives/list/security-announce@lists.opensuse.org/message/UDUEZFTCIXU7YDYUQLIGKFQLDELNXQRZ/, (Accessed on 06/08/2023).

[65] "Rhsa-2016:2098 - security advisory - red hat customer portal," https://access.redhat.com/errata/RHSA-2016:2098.html, (Accessed on 06/08/2023).

[66] "[security] fedora 23 update: kernel-4.7.9-100.fc23 - package-announce - fedora mailing-lists," https://lists.fedoraproject.org/archives/list/package-announce@lists.fedoraproject.org/message/W3APRVDVPDBXLH4DC5UKZVCR742MJIM3/, (Accessed on 06/08/2023).

[67] "Rooting android with a dirty cow," https://book-of-gehn.github.io/articles/2021/08/22/Rooting-Android-with-a-Dirty-COW.html, (Accessed on 06/09/2023).