The Dirty Cow vulnerability

CVE-2016-5195

Sara Spagnoletto	345990808
Coral Belo	209089325

1. Introduction

In 2016, a new bug was reported concerning the code of the Linux Kernel. Red Hat, a company developing commercial Enterprise Linux distribution, described it as:

"A race condition was found in the way the Linux kernel's memory subsystem handled the copy-on-write breakage of private read-only memory mappings. An unprivileged, local user could use this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system."

This design flaw in the Linux Kernel allows the gaining of elevated access to resources that are normally reserved to root users. Therefore, an attacker could carry an exploit of type LPE. Specifically, it allows for **Vertical Privilege Escalation**, where a lower privilege user or application accesses functions or content reserved for higher privilege users or applications.

The attack gets its name from the Dirty Bit and the copy-on-write mechanisms of the Linux kernel, i.e. Dirty Cow. It was discovered by Phil Oester, and reported by all major Linux distributions: Red Hat, Debian, Ubuntu, and SUSE. The vulnerability has existed in the Linux kernel since version 2.6.22 released in September 2007. It has been patched in Linux kernel versions 4.8.3, 4.7.9, 4.4.26, and newer.

To further continue with the reading and understanding of the exploit, it is crucial to grasp two main concepts: race conditions, and the copy-on-write mechanism.

Race conditions: A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly. Race conditions are considered a common issue for **multithreaded** applications.

Copy-on-write mechanism (Linux): Copy-on-write (sometimes referred to as "COW") is an optimization strategy used in computer programming. The fundamental idea is that if multiple callers ask for initially indistinguishable resources, you can give them pointers to the same resource. This function can be maintained until a caller tries to modify its "copy" of the resource, at which point a true private copy is created to prevent the changes from becoming visible to everyone else. All of this happens transparently to the callers. The primary advantage is that if a caller never makes any modifications, no private copy need ever be created.

As we will see later in the on depth explanation, this attack relies on the fact it is possible to create a race condition in the way the Linux kernel's memory subsystem handles the copy-on-write of private read-only memory mappings.

2. How to execute the attack

The community of the DirtyCow exploit created a Github repository showcasing various attacks that can be implemented by taking advantage of the previously mentioned bug.
On the Github page are listed multiple PoCs (proof of concepts) on how to run the attack.
Some examples include:

- Allows users to write on files meant to be read-only.
- Gives the user root by overwriting `/usr/bin/passwd` or a SUID binary.
- Allows users to write on files meant to be read-only (android).
- Gives the user root by injecting shellcode into a SUID file.

In this report, we will focus on and research the first PoC: “allow users to write on files meant to be read-only”. The PoC provides a `C` file named `dirtyc0w.c`.
The program receives two args; the root file to write onto, and text to write into the root file. It can be compiled and run by:

```
gcc -lpthread dirtyc0w.c -o dirtyc0w
./dirtycow [root_file] [text_to_write]
```

Looking into and explain the exploit of `dirtyc0w.c`:

The `main` function

In the main function, we have:

```
87: f=open(argv[1], O_RDONLY);
88: fstat(f,&st);
89: name=argv[1];
```

In which the root file is opened as read-only.

Following comes a call to `nmap()`:

```
101: map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
```

The function `nmap` is used to create a new mapping in the virtual address space in the current process. Function’s parameters:

- `f` - file descriptor of the previously opened root file.
- `PROT_READ` - indicates this new memory area is read-only.
- `MAP_PRIVATE` - allows for a private copy-on-write (COW) mapping.

This last flag is crucial to the execution of the attack. This means the function won’t copy the whole file onto the memory, but initially just map it. This is useful when just reading from the file since it’s not wasting great amounts of RAM to load a copy. This also means it will potentially make a copy of the file only when writing is needed. As seen before, the copy-on-write mechanism means you copy the memory segment when writing. So even though the file is mapped as read-only, because of the private mapping, we can write a copy of it. In short, `nmap` will map the root file into memory, and we can either read the content of the file or write to a copy of it. The changes to the copy file should not be propagated to the real file.

Then, the main starts two different threads referring to two different functions. The threads are then joined as we must wait for the threads to finish.

```
106: pthread_create(&pth1,NULL,madviseThread,argv[1]);
107: pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
...
111: pthread_join(pth1,NULL);
112: pthread_join(pth2,NULL);
```

The two threads will run in parallel. Dirtycow is a race condition vulnerability, this means certain events have to occur in a specific order, that is fairly unlikely to happen under normal circumstances.

The `madviseThread` function

This function calls the syscall `madvise` in a loop one hundred million times.

```
45: c+=madvise(map,100,MADV_DONTNEED);
```

This syscall can be used for optimization reasons. You can provide the kernel with some information on how you intend to use a memory-mapped area as there are different techniques for how you handle caching. Here we tell the kernel that the memory area where we mapped the file to, or at least the first 100 bytes, is probably not needed anytime soon. The `MADV_DONTNEED` communicates to no expect to access the memory in the near future, so the kernel can free resources associated with it. Subsequent pages in this range will succeed but will result in the reloading of the memory content.

The `procselfmemThread` function

The function starts by opening the file `proc/self/mem`.

```
61: int f=open("/proc/self/mem", O_RDWR);
```

The `/proc` is called a pseudo filesystem, it doesn't really contain files in common sense but refers to something more general, something you can read and write to. In this case `/proc/self` refers to special “files” provided for the current process. Every process will have its own `proc/self`. Inside there, there is a file called mem, which represents the current process’s memory. You could read your own process's memory by reading from this file.
In the third section, we will explain more in-depth why are using the `proc/self/mem` to open and write.

Then the exploit writes to this file in a loop.

```
67: lseek(f,(uintptr_t) map,SEEK_SET);
68: c+=write(f,str,strlen(str));
```

First, it performs a seek, which moves the current cursors to the start of the file mapped into memory, and then writes the string we pass the program arguments to it.

Due to the previously mentioned copy-on-write mapping, **this will trigger a copy** of the memory so that we can write to it and see the changes.

The race condition

Both functions' operations are repeated simultaneously one hundred million times. If we were to do these operations isolated from each other, probably nothing particular would occur, and we would have the expected result. But, because there is a race condition somewhere, trying it over and over will create a specific edge case. It will trick the call to write to the actual underlying file instead of copy.

The problem relies on two crucial aspects. Firstly, we constantly call the `madvise` system call. Due to the flag used, we communicate the application no longer needs the pages.

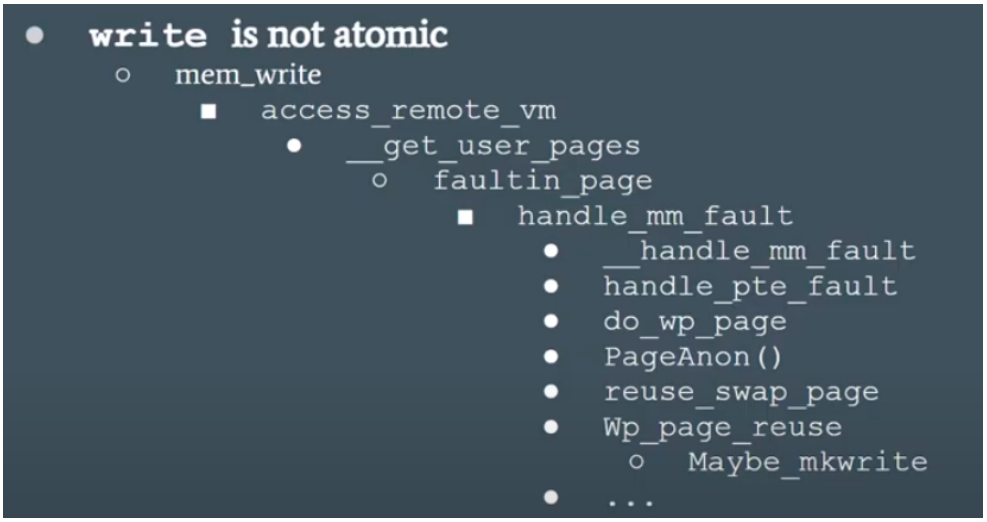
When reading and writing to disk the system caches or buffers the data. If the data is used for reading, it can simply leave it cached for further reads. Instead, when writing to the data, after writing it must tell the system the buffer it's "dirty". The system now must make sure the changes are propagated to the underlying physical memory. In this case, when writing to the copied `mmapped` memory, the memory page gets flagged dirty. This, with the addition of the `madvise` flag `MADV_DONTNEED`, will cause the throwing away of the memory from the cache. This is crucial for the exploit because this means whenever you try to write it next time, the copy of the memory will be tossed, so we need to reload a new copy from memory to write into.

Secondly, we constantly attempt to write to the mapped file. As mentioned above, since it's using the copy-on-write mechanism, and since no copy is saved on memory, it must create a copy. Creating this copy takes time, and here lies the race condition. When trying to write to the file in the `procselfmemThread` function, if the copy-on-write cycle is not complete yet, a race condition might occur which causes the writing to the actual root file.

3. The flaw in the Linux kernel

In this section, we will analyze the original flawed code that caused the race condition to occur.

First, we should note that a write operation is not atomic, it's actually made out of various follow-up functions, as shown in the picture below:



The file in which the "bug" lies is the `mm/gup.c` file, which was then patched to eliminate the exploit. The file belongs to Linux `mm`, which stands for memory management. The file name, `gup`, stands for `get_user_pages`.

Here we can find the `mm/gup.c` file in the latest commit version before it was patched.

In this file, we find the `_get_user_pages` function mentioned above, which is utilized when writing.

In the `_get_user_pages` function we notice there is a do loop, this suggests the function keeps trying until it gets a valid page. Then, inside the loop:

```
577: page = follow_page_mask(vma, start, foll_flags, &page_mask);
```

Here it tries to get the page, but since the requested access differs from the permissions of the page, it will return null. This will lead to the `fault_in_page` function.

```
578: if (!page) {
579:     int ret;
580:     ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);
...
```

In the `faultin_page` function checks the `foll_flags` parameter, which in this case suggests we don't have write permissions to the file, and it will resolve the error by handing us a copy of the requested page.

Then, it modifies the `FOLL_WRITE` flag (ie, the `flog_flags`) of the `linux/mm.h` file.

```
381: ret = handle_mm_fault(vma, address, fault_flags);
...
414:  if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
415:          *flags &= ~FOLL_WRITE;
```

Here we can find the `linux/mm.h` file in the latest commit version before it was patched. At this point, as you can see above, the flag is turned since we now have a cowed paged, and to prevent an infinite loop.

Now, going back to the `_get_user_pages` function, the `ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);` will succeed since we changed the flag.

But what if the `madvise` would run right now and drop the cowed page? This will cause another call to `faultin_page`, but this time with the `foll_flags` already deactivated. The `failtin_page` won't see the right flag, so it will return the original page, which maps to the original file on disk. At this point, the `_get_user_pages` will return the original file mapping on disk.

Another function to note when writing is the `access_remote_vm`. As seen before, we're opening the file using the `/proc/self/mem`. This is a pseudo-file generated by the kernel, which doesn't exist on disk, therefore the kernel must have direct access to our process's memory. This `access_remote_vm` allows the kernel to access our memory and generate the `/proc/self/mem` pseudo-file. By writing to `/proc/self/mem` we are piggybacking off the kernel's access to everything, even the read-only file on disk. This is how we write the dirty bits back to the file on disk.

4. The patching and conclusion

The vulnerability has been in the Linux kernel for a very long time. According to Linus Torvald, Linux's creator, the bug was attempted to be fixed once by him in 2005. In fact, you can find the attempt in the commit 4ceb5db9757a ("Fix get_user_pages() race for write access") of the Linux kernel. That was later undone due to problems on a different commit f33ea7f404e5 ("fix get_user_pages bug"), making into v3.9.

In addition, this problem was once considered a “theoretical” race condition due to constraints of older systems, but as the VM has gotten faster, today it’s simple to actually trigger.

Recalling from the previous section, the race condition was the `FOLL_WRITE` flag turning simultaneously to the drop of the cowed page, which caused the issue.

The fix introduces a new internal `FOLL_COW` flag to mark the "already did a COW" rather than play racy games with `FOLL_WRITE` that is very fundamental, and then use the `pte` dirty flag to validate that the `FOLL_COW` flag is still valid.

Here we can find the `mm/gup.c` file in the commit right after it was patched.
Here we can find the `linux/mm.h` file in the commit right after it was patched.

The patch adds a new flag `FOLL_COW` into the `linux/mm.h` file.

```
2220: #define FOLL_WRITE      0x01  /* check pte is writable */
```

Before the patch, as explained before, `FOLL_WRITE` was tossed out after the `VM_FAULT_WRITE` page fault. But after the patch, the new flag encodes the expectation that the next retry will encounter a dirty COWed page. If the cowed page is not there, a new file will be created instead of handing back the original copy. In the `mm/gup.c file`, this function was added:

```
67: static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
68: {
69:    return pte_write(pte) ||
70:           ((flags & FOLL_FORCE) && (flags & FOLL_COW) && pte_dirty(pte));
71: }
```

Then, in the `_get_user_page` function, in the do loop, we now check:

```
108:   if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
109:         pte_unmap_unlock(ptep, ptl);
110:         return NULL;
111:   }
```

Lastly, in the `faultin_page function`:

```
424:   if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
425:           *flags |= FOLL_COW;
```

In short, the fix maintains the expectation of the COWed page in the next round of retry, whereas the old version throws the write semantics and hopes that the COWed page is still there in the next retry.

The exploitation of this bug does not leave any trace of anything abnormal happening to the logs. But, according to Phil Oester, an exploit using this technique has been found in the wild from an HTTP packet capture.

Sources

- [Patch] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619
- [Community website] https://dirtycow.ninja/
- [Community github] https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails
- [Youtube explanation] https://www.youtube.com/watch?v=kEsshExn7aE&t=555s
- [Youtube explanation] https://www.youtube.com/watch?v=FKdZ0QElga8