The Dirty Truth About “Dirty COW” (CVE-2016-5195)

Posted December 7, 2016

0 Comments

The Dirty COW vulnerability (CVE-2016-5195) is a recent (and interesting) privilege escalation vulnerability in the Linux kernel. By exploiting this vulnerability, an ordinary, non-privileged user already on a machine can take complete control. Dirty COW works by taking advantage of a flaw in how the Linux kernel manages memory – more specifically, an optimization technique in how memory pages are utilized.

Going forward, Dirty COW stands to have more of an impact than something like AtomBombing (https://www.carbonblack.com/2016/11/15/largely-hyped-atombombing-new-variation-old-technique/) given the type of vulnerability and its presence across many different versions of the Linux kernel, going back quite far (2.6.22 circa mid 2007). This exploit is being seen in the wild; that is how it was discovered.

Since the Linux kernel is used across many different devices and systems – including embedded, mobile (Android), virtualization and cloud platforms (docker (https://www.docker.com/), AWS (https://aws.amazon.com/)), and IoT devices, the impact could be huge.

The good news is a fix has already been implemented (https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619). The bad news is it will be relatively easy to find machines that have not been patched, if they can even be (https://nakedsecurity.sophos.com/2016/04/21/29-of-android-devices-cant-be-patched-by-google/) patched (https://www.ietf.org/blog/2016/07/patching-the-internet-of-things-iot-software-update-workshop-2016/).

Before we dig a little deeper into Dirty COW, let’s take a look at some of the concepts in play.

COW (Copy on Write)

Copy on Write (https://en.wikipedia.org/wiki/Copy-on-write) is an optimization technique that allows virtual pages of memory in different processes, which have identical content to map to the same physical memory pages. Microsoft has a nice page with pictures demonstrating the technique here. (https://msdn.microsoft.com/en-us/library/windows/desktop/aa366785(v=vs.85).aspx)

As long as the processes only read from these memory pages, they are shared and the optimization is in effect. If, at any point, one of the processes writes to one of these shared pages, the data is copied to a new physical page and the virtual-to-physical-memory mappings are updated to reflect the changes. This technique is utilized in many different scenarios such as loaded dlls, forked processes, file system data blocks, etc.

Race Condition

Cookie Settings

A race condition is when the end result of an operation or sequence of operations is dependent on the order or timing of the operation(s). Race conditions usually have a timing window – the gap of time where the undesired/non-deterministic behavior can happen. You see this often when there are multiple, concurrently running threads accessing a shared resource with no locking.

A good example is using the increment/decrement operators (++ and — respectively, both pre and post flavors) in C/C++ to change a value. These operators usually map to three lower-level machine operations (load into register, increment value in register, and save contents of register back to memory).   You can make those operators atomic by wrapping them with a mutex, or using something else altogether, such as InterlockedIncrement() on Windows. If not protected, there will be these small windows where errors can occur.

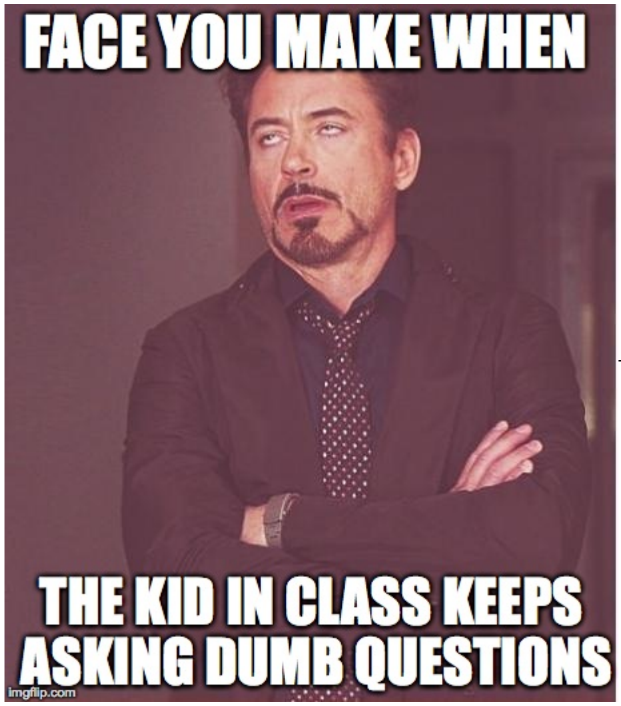
The size of this window for the most part will determine how easily it can be triggered. As an example, I once debugged a filesystem kernel driver issue that had a window of about 3µs and it took a long time for me to reproduce (and of course happened much more readily at customers' sites due to timing and load), but when it did repro the results were catastrophic – file corruption. The way the race condition in the Linux kernel is exploited by the dirtyc0w PoC (https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs) is by looping many times (100,000,000) to ensure it can trigger within what is most likely a very small time window.

The Vulnerability

So, with the technical details out of the way, what is Dirty COW? Dirty COW (CVE-2016-5195) is a recent privilege escalation vulnerability in the Linux kernel. In other words, an ordinary, non-privileged user already on a machine or in conjunction with another exploit to get execution on the box (perhaps Shellshock (https://en.wikipedia.org/wiki/Shellshock_(software_bug))?) can take complete control by exploiting this vulnerability. It works by taking advantage of a flaw in how the Linux kernel manages memory – more specifically an optimization technique in how memory pages are utilized.

Much to my chagrin, this particular vulnerability is following in the now common tradition of naming and branding higher profile security issues. The name derives from "COW" (Copy on Write, explained above) and "dirty," which could refer to the fact this feature is being manipulated in a malicious manner (how cheeky!) or referring to the slang term for modified/written to memory pages.

The Dirty COW "homepage (https://dirtycow.ninja/)" has some good information on the issue as well many examples of PoC exploit code. There's also a FAQ. Best question in the FAQ: "Is this an OpenSSL bug?"



(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/Carbon_Black_Dirty_COW_Linux_2.jpg)
Oh, and with Christmas right around the corner be sure to visit the online Dirty COW shop (http://www.zazzle.com/collections/dirty_cow_collection-119587962650451153) as well!

Exploiting the race condition

Looking at the very straightforward 'dirtyc0w' PoC (https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c) (which is claimed to be the method first seen in the wild), it takes as arguments a filename and a string to be written to the file. The file specified is opened read-only and then is mmap'ed into the current process read-only and with MAP_PRIVATE (http://man7.org/linux/man-pages/man2/mmap.2.html) specified to create a private COW mapping.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/DirtyCow_Carbon_Black_Linux_4.jpg)
It then kicks off two threads. One thread opens /proc/self/mem read-write and loops millions of times writing the data specified on the command-line to it at the offset of the mmap'ed file. /proc/self/mem is an entry in the procfs (https://en.wikipedia.org/wiki/Procfs) virtual file system which gives access to the pages of a current process's virtual memory.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/DirtyCow_Carbon_Black_Linux_5.jpg)
This writing to /proc/self/mem triggers COWs, but since these pages were marked as private, the memory pages written to will not be written back to the mapped file. However, the other thread loops, again millions of times, calling the madvise(2) system call on the same offset w/ the flag MADV_DONTNEED which tells the kernel to discard the pages that are mmap'ed.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/DirtyCow_Carbon_Black_Linux_6.jpg)
This is where the race condition lies. Continually writing to memory and, at the same time, telling the kernel you no longer need that memory exposes a race condition where the changes written to the private pages could be instead propagated to the underlying file!

The possibilities for exploitation are limited only by the imagination of the crafter. If an attacker can read a setuid root binary like /usr/bin/passwd or a writable only by root config file like /etc/passwd, they can pretty much modify the file at will as an ordinary user.

In my testing I successfully ran the PoC code on Centos 7, but not on a few 6.x flavors. From what I can tell this is due to /proc/self/mem not being writable on certain versions of RHEL/Centos. This, in my opinion, is a very GOOD thing. I'm not sure why it was made writable again in newer kernels. In addition, in my Centos 7.0 VM, the dirtyc0w PoC also locked up the box occasionally so it's not super stable. There are other PoC examples that use the ptrace(2) system call which ***will*** work on RHEL 5 and 6 (and Centos versions as well). In the event that patching the kernel is not possible or practical, there is a SystemTap script (https://bugzilla.redhat.com/show_bug.cgi?id=1384344) available which can be used to mitigate the issue. Additionally, someone has created an Ansible (https://www.ansible.com/) playbook (https://github.com/oleg-fiksel/ansible_CVE-2016-5195_check) to test systems for vulnerability.

Getting root

So, as a demonstration, I thought I would show how this PoC could be used, as is, to get root on a machine as a non-privileged user. The PoC is pretty limited in that it will write a single NULL terminated string due to the use of strlen, etc but could easily be augmented to write out a binary or more arbitrary length string payload.

The dirtyc0w exploit was a little unstable when attempting to write what seemed like more than a few bytes, so I tried out the pokemon (https://github.com/dirtycow/dirtycow.github.io/blob/master/pokemon.c) exploit which is pretty much the same code except uses ptrace(2) instead of writing to /proc/self/mem. My idea was to re-write the first line of /etc/passwd removing the field specifying there was a password for the root account. Here is /etc/passwd untouched on my Centos 7 VM.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/DirtyCow_Carbon_Black_Linux_7.jpg)Invoking the app with the string argument equal to root's entry in /etc/passwd but with the 'x' removed is shown below. This will change the line from "this account has a password and it is in /etc/shadow" to "this account has no password". This particular PoC has the very, um, unique feature of drawing an ASCII art cow with the string entered on the command line coming out of its rear end.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/2016/12/DirtyCow_Carbon_Black_Linux_8.jpg)
Ok, ran successfully… does it work? Yes! I was able to su without being prompted for a password. Not good.

(http://blogs.vmware.com/security/wp-content/blogs.dir/26/files/2020/05/2016/12/DirtyCow_Carbon_Black_Linux_9.jpg)
This does not spell the end of the world, but Dirty COW will be another option (and an easy one) for attackers to privesc on a Linux box. Combined with a remote execution exploit this would provide a nice one-two punch. The sheer number of vulnerable machines and their ability to be patched means it will probably be a useful exploit for a long time to come.

Comments

0 Comments have been added so far

vmware

VMware Security

VMware Enterprise Security Solutions (http://www.vmware.com/security.html)

VMware Security Response Center (https://www.vmware.com/security/vsrc.html)

VMware Security Certifications (https://www.vmware.com/security/certifications.html)

Security Hardening Guides (https://www.vmware.com/security/hardening-guides.html)

VMware Security Development Lifecycle (https://www.vmware.com/security/sdl.html)

Company Information

Leadership (http://vmware.com/company/leadership/)

Careers at VMware (http://vmware.com/company/careers/)

Acquisitions (http://vmware.com/company/acquisitions/)

Office Locations (http://vmware.com/company/office_locations/)

Contact VMware (http://vmware.com/company/contact/)

Investor Relations (http://ir.vmware.com/)

VMware Foundation (http://vmware.com/company/foundation.html)

Why Choose VMware? (http://vmware.com/why-choose-vmware/)

News & Events

Newsroom (http://vmware.com/company/news/)

Articles (http://vmware.com/company/news/articles/)

Events (http://vmware.com/events/)

Awards (http://vmware.com/company/news/media-resources/awards.html)

Media Resource Center (http://vmware.com/company/news/media-resources/)

Media & Contacts (http://vmware.com/company/news/media-contacts.html)

Community

Cookie Settings