CVE-2022-0847 - Dirty Pipe Vulnerability in Linux

Introduction

The Linux Dirty Pipe vulnerability, also known as CVE-2022-0847 is major a vulnerability first discovered near the end of February 2022 which affects Linux kernel versions 5.8 and above. The Dirty Pipe vulnerability is considered severe since there aren't workarounds (besides upgrading to a patched version of the Linux kernel) and it allows local privilege escalation allowing users to write to files they don't have write access to (such as /etc/passwd), which then allows them to run commands as root or drop to a root shell.

Before I go any further, if you are reading this post from a computer running Linux (or GNU/Linux), make sure you have updated your kernel to 5.16.11, 5.26.36 or 5.10.102 to make sure you're protected against this vulnerability!

This vulnerability was first discovered last month when investigating an older bug report regarding corrupted files. In particular, there were reports of corrupted CRC checksums at the end of zip files in a particular use case involving splices and pipes.

Some background information

There are a few background elements that I think are useful to understand this vulnerability and how it works: Pipes, pages, the page cache and the splice system call on Linux.una20

First: Pipes. Pipes in Linux are a unidirectional, inter-process communication method provided by the kernel. Pipes are commonly used to pass information between 2 applications. The output of the first application is saved to the pipe, which is then read by the second application. Pipes can be used both in the shell using '|' to redirect the output of one program to the input of the next (such as if you do cat foo | grep a | cut -c 2- | uniq | sort | tee bar), or within programs, which is more flexible and powerful. Under the hood, pipes are treated and behave just like files, which is part of the Unix philosophy. However, unlike files on secondary storage, the data in pipes is stored in a buffer (that works like a ring using pages) instead of on the disk, where data from the first application is added to the buffer, and the second application reads from it. In other words, it's a handy way to send data from one application to another, where one pushes data into the pipe, and the other pulls it from the pipe.

Next: Pages. As discussed in the earlier modules of the course, pages are blocks of virtual memory. Pages are generally the smallest unit of data used for virtual memory. These pages are what will contain files, data, code, so on, and can be loaded from the secondary disk (SSD/HDD), or from system memory. On Linux based systems, the page size is typically 4 KiB (4096 bytes). (You can check the page size for yourself by running 'getconf PAGESIZE')

To speed up I/O operations when a user is frequently working with certain pages, Linux maintains a page cache, which will essentially cache the commonly used items from secondary memory (SSD/HDD) onto primary RAM. This means that if you're continually loading a part of a file, instead of waiting for a hard drive each time, it can be cached and retrieved from the cache in memory. This cache is usually in unused parts of memory, and can be easily cleared if an application needs to actually use the memory. Pages in the page cache that are modified while in memory (such as by a write to the file) will mark the page as 'dirty', which have to be written back to the disk to save/persist those changes.

Finally, the splice() system call. Splice is a system call that moved data between files on the disk and pipes without having to load and go through the userspace. Generally, splice can work by just remapping pages, which makes it fast. This means that instead of passing the page around, it passes page references, without having to do the copy. You can splice a file to a pipe, then splice back from the pipe into another file without dropping to userspace, which makes this both fast and secure.

What are the causes?

The dirty pipe vulnerability takes advantage of a pipe and a splice call. This started as a bug that was introduced with a commit which refactored the pipe buffer code, changing how 'mergeable' checks were done.

As mentioned in the section above, all data, including the data temporarily stored in a pipe lives in a page, which can get cached into memory. The first write into a pipe allocates a 4 KiB page. Then, this page is used for writes until it fills up, at which point a new page is allocated. However, if you use the splice system call, the kernel first loads the data from the file (source) into the page cache, then, creates a 'pipe_buffer' struct (the destination pipe) which points to the page cache. This is a zero-copy, since it essentially just copies the reference to the page in the page cache. However, even if the page isn't full, it can't be appended to, since the page is owned by the page cache, not the pipe.

The checks to ensure that data isn't appended to page in the page cache have changed over time, but the latest revision with Linux 5.8 turned a bug into a critical issue. Since kernel 4.9, there has been a bug causing the flags member of the pipe_buffer struct to be missing, however, the new critical issue allows a user to inject PIPE_BUF_FLAG_CAN_MERGE into the page cache reference, which then allows the user to overwrite data in the page cache! Once the PIPE_BUF_FLAG_CAN_MERGE flag is injected into the reference, writing new specially crafted data into the pipe that was originally loaded with the splice causes the kernel to append this data to the end of the page in the page cache, instead of creating a new page.

Since the splice operation loaded the contents of a file into the pipe, by placing the pages for the file in the page cache, if we can write to the page cache, we can effectively write to the version of the file that will be used by the OS. This is critical, since you only need read-access in order to load the contents from a file into a pipe. However, as seen here, by setting the flag, we can write to the page in the page cache. This means that any user can write into files they would normally only have read access to. Essentially, the kernel is tricked into thinking this is a normal page for the pipe that is safe to write to, however, it's actually the contents of other files. Yikes.

Now, there is one aspect of this to consider. By using this vulnerability, you can write data into the page

cache, which is loaded with the contents of another file. However, since the kernel is unaware of this change, the page in the cache is not marked as dirty. This means that eventually, either on system restart, or if the kernel needs to reclaim the memory, the page in the cache will be dropped. This is why this bug was hard to spot in the wild, since in most cases, it would be gone by the time the system restarted. On the other hand, since the data in the page cache isn't written to the disk, and lasts for a considerable amount of time, this provides a stealthy attack surface that doesn't leave a trace on the disk.

Why is it important?

This vulnerability is critically important, since it can be exploited to achieve local privilege escalation. This vulnerability allows any unprivileged user to write to pages in the page-cache which correspond to files they only have read access to. Since the kernel will use the page cache to load a file if it is cached, this effectively allows the user to overwrite data into any file they have read access to, with some limitations.

This can be exploited to get root access (or a root shell) through several means. For example, through a few steps, one can simply clear the root password from /etc/passwd, add SSH keys, or change data used by root processes. The simplest is likely by disabling the password for the root user. While normal users can't access /etc/shadow, by removing the x in /etc/passwd, you can remove the lookup to /etc/shadow, and allow password-less login to the root user. For those who are curious, there is a proof-of-concept exploit available on the original dirty pipe vulnerability page that allows writes into read-only files. Once an attacker has root access on your system, there's little they can't do.

A system with this vulnerability can be exploited by attackers for any malicious purpose following a few steps. First, the user creates a pipe. This is a pipe which they must have write access to. Next, you fill the pipe with random or arbitrary data. The contents don't matter, it just needs to fill the pipe so that the PIPE_BUF_FLAG_CAN_MERGE which is later incorrectly checked will get set in all of the pages and entries used by the pipe. Next, you empty the pipe, by reading the data from it. This leaves the flag set in all instances of the pipe_buffer struct in the ring.

For step 4, you need to make a splice system call for the file you want to modify, opened with O_RDONLY (read only), and splice it into the pipe just up to the target offset. The target offset is the location in the file where you want to write your new data to and make your changes. For example, this could be the location of the root user password. Next, you write the data you want to replace in the file into the pipe. As discussed in the last step, instead of creating a new anonymous pipe_buffer struck with new pages, the new data added to the pipe will begin to replace the data in the page from the read-only file (which is in the page cache) because the PIPE_BUF_FLAG_CAN_MERGE flag is set. Recall that that flag would normally only be set if this were a normal pipe, but remained set.

This exploit was particular interesting (and terrifying) for myself since I had my secondary computer running an earlier build of Linux 5.16, which was vulnerable, and I was able to test both the small code by the original author, as well as their proof-of-concept exploit which both worked. While not trivial, it is a vulnerability that is relatively simple to exploit. While there was somewhat similar exploit named Dirty Cow from a few years ago, this is simpler to exploit.

There are however a few limitations to these exploits, although they are all simple to overcome once you gain root access or a root shell. First, you need to have read access to a file. Without read access, you won't be able to use the splice call. Next, the offset that you want to overwrite can't be on a page boundary (ie: can't be a multiple of 4096). This is because you need to splice at least 1 byte from the target file into the pipe. Next, you are limited to overwriting 1 page (4096 bytes). While this might seem short, since you can't change large files, it's more than enough to update important files such as /etc/passwd. Finally, you can't resize the file, so the data you write can't alter the file size. Again, once you have root access, the other limitations wouldn't matter for an attacker, since they can use their root access to change any other files they wanted.

Who is affected and in what ways?

Any device running Linux 5.8 or later (without the latest patches) are affected. This includes many servers, desktop computers and laptops, as well as many recent mobile devices running Android. In particular, systems running many major distributions from the past 2 years without the most recent updates are vulnerable. This includes Debian 11 (bullseye), Fedora 33 and up, Ubuntu 20.04 and up, as well as many others including Arch, Gentoo, OpenSUSE, as well as their derivatives. This constitutes a good portion of server and desktop systems running recent installations of GNU/Linux.

Although it is a local vulnerability, meaning that it doesn't add a new vector for an outside attacker, if an attacker has any access to a user on the system, be it physical or digital such as over SSH, Telnet or VNC, they can exploit this vulnerability to gain root access.

Equally important is the use of the Linux kernel in the Android mobile operating system. Within a few days of the discovery of the original vulnerability, the bug was reproduced on a Google Pixel 6, and reported to the Android Security Team. If the bug is reproduced, then it means that there will also be exploits targeting these devices. Of course, this impacts more devices than just the Pixel 6. For example, the Samsung Galaxy S22 series are all based on Android 12, running Linux kernel 5.10.43, which is vulnerable. Android versions prior to Android 11 used Linux kernel versions 4.4, 4.9, 4.14, 4.19 or 5.4, however, starting with Android 12, support for Linux 5.10 was added, with Android T (the experimental branch) also using 5.10. This means that many flagship phones are going to ship with a build of Linux 5.10. Given the poor record of Android manufacturers issuing software updates, this can end up affecting a good number of recent phones that were released.

How can similar problems be prevented in the future?

Given that this bug was introduced due to programmer error when refactoring a section of the code, it's harder to provide many meaningful suggestions. A solution that is frequently presented is to have a more stringent code-review process, with more developers or a stronger emphasis on security. However, patches going into the Linux kernel already undergo a thorough code-review from multiple developers. While there are also suggestions to avoid refactoring some code since it can introduce bugs such as this one, refactoring is also important in order to clean up the code and keep it consistent.

Some mistakes like this could be caught in part by using analysis tools. For example, the basis of this

vulnerability was originally a bug with uninitialized flags for the struct. This type of mistake such as uninitialized variables in structs could be caught if the team added corresponding checks. However, this of course comes with the downside that the developers might also get bogged down with tons of false positive reports. Some have also suggested switching to a safer language such as Rust, but in cases like this where it's very specific behavior in specific calls and specific checks, which don't necessarily break the safety of the language, it's debatable it would have caught a mistake such as this one.

We have also seen that having a simpler codebase makes auditing much simpler while also making it both faster and safer since it becomes easier to stop mistakes. For example, the Wireguard VPN protocol which was written to be a faster, safer VPN protocol was welcomed into the Linux kernel in that past years since it's so much simpler then competing protocols such as OpenVPN (which is based on the comparatively massive OpenSSL and TLS libraries). However, a kernel is much more sophisticated than VPN protocols, and given that Linux is a monolithic kernel, it is much harder to just have simpler code - although there are projects such as Redox OS which are both supposed to be simpler, using a microkernel architecture, as well as being written in the safer Rust language.

Overall, this is one of the aspects where open source shines. We can always have more developers looking at and through the code, and it allows users such as Max Kellermann (the original author of the vulnerability) to not only find bugs such as this one, but also patch them in a timely manner.

If such a problem happened, how can we mitigate it?

This vulnerability was reported professionally by the security researcher, who alerted the Linux kernel security team within a day of his discovery of the vulnerability, along with a patch. This was quickly sent to the Linux Kernel Mailing List the next day, where it was merged into the next release within a few days. Overall this problem fixed very quickly by the Linux developers, with patched versions of the kernel out within a week of the discovery of the original vulnerability. This is excellent, and it shows that security issues such as this one are taken very seriously by the developers who can work through these issues and patch them in an effective manner.

Once again, this is a good example of how the open source software community is very good at responding to issues like this once they are found with everyone working together to get it patched and ensure systems get updated. This vulnerability was fixed for the release of versions 5.16.11, 5.15.25, and 5.10.102. Of these, 5.16.11 is the latest stable release that distributions like Fedora and Arch use, with 5.15 and 5.10 being LTS releases.

Once the new versions of the kernel were released, all of the distributions mentioned in my section above have either upgraded their kernel to include the patch for this vulnerability, or, backported the patch for this vulnerability into their kernel versions. This is a good example of the author, kernel developer, as well as distribution maintainers working together to ensure security issues like this get patched.

As I mentioned in the start of this post, the mitigation for this vulnerability for both desktop and server systems is to update your kernel! Whether you're maintaining a server or your system, make sure your software is up-to-date! For servers in particular, make sure you're running a version of your distribution that is actively maintained, such as an LTS release to ensure patches like this one get backported to ensure your systems are safe.

Mobile users on the other hand will have to wait for updated versions of Android to be released for their devices, likely through security updates. We can hope that they will be as responsive as distribution maintainers are, and apply these patches and deliver new updates in a timely manner. Since this bug was introduced in Kernel 5.8 from a few years ago, and the most recent Android version to ship with a vulnerable kernel is Android 11, most devices that are vulnerable are relatively new, increasing the chance they will get patched.

Conclusion

I hope you enjoyed reading this and learned about the dirty pipe vulnerability on Linux, and how pipes, pages, the page cache and splice call work in general!

Biblography

- “CVE-2022-0847,” CVE, 03-Mar-2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847. [Accessed: 14-Mar-2022].
- D. Das, “What is the dirty pipe exploit in linux and how can you fix it?,” Make Use Of, 10-Mar-2022. [Online]. Available: https://www.makeuseof.com/what-is-the-dirty-pipe-exploit-in-linux-and-fix-it/. [Accessed: 14-Mar-2022].
- D. LeClair, “New dirty pipe linux vulnerability is the worst in years,” How-To Geek, 08-Mar-2022. [Online]. Available: https://www.howtogeek.com/790435/new-dirty-pipe-linux-vulnerability-is-the-worst-in-years/. [Accessed: 14-Mar-2022].
- M. Kellermann, “The dirty pipe vulnerability,” The Dirty Pipe Vulnerability - The Dirty Pipe Vulnerability documentation, 07-Mar-2022. [Online]. Available: https://dirtypipe.cm4all.com/. [Accessed: 14-Mar-2022].
- P. Arntz, “Linux ‘dirty pipe’ vulnerability gives unprivileged users root access,” Malwarebytes Labs, 11-Mar-2022. [Online]. Available: https://blog.malwarebytes.com/exploits-and-vulnerabilities/2022/03/linux-dirty-pipe-vulnerability-gives-unprivileged-users-root-access/. [Accessed: 14-Mar-2022].
- S. Hazarika, “Linux kernel bug dubbed 'dirty pipe' can lead to root access, affects Android devices as well,” XDA Developers, 09-Mar-2022. [Online]. Available: https://www.xda-developers.com/dirty-pipe-linux-vulnerability-root-android/. [Accessed: 14-Mar-2022].

Further reading

The original post disclosing this vulnerability has lots of other technical information, including how the vulnerability was discovered due to a problem with repeated corruption on web-server logs. https://dirtypipe.cm4all.com/

The commit that introduced this vulnerability is here:

https://github.com/torvalds/linux/commit/f6dd975583bd8ce088400648fd9819e4691c8958

For those of you that want to try the proof-of-concept exploit on your unpatched devices, the source code is available here: https://packetstormsecurity.com/files/166229/Dirty-Pipe-Linux-Privilege-Escalation.html

If you have any questions or comments, feel free to leave it below!