TRY HACK ME: Dirty Pipe CVE-2022-0847 Write-Up



**Task 1 Info Introduction and Deploy-**

In March 2022, a researcher named Max Kellerman publicly disclosed a Linux Kernel vulnerability (nicknamed "Dirty Pipe" for its similarities to the notorious "Dirty Cow" exploit affecting older versions of the kernel) that allowed attackers to arbitrarily overwrite files on the operating system. The vulnerability was responsibly disclosed in early 2022 and was publicly released in a blog post written by Max Kellerman soon after patches were made available.

Arbitrary file overwrites at the kernel level can be very easily leveraged to escalate privileges on the machine (i.e. to obtain administrator, or "root" privileges). This is a devastating vulnerability, made more so by its reach: any devices running a vulnerable version of the Linux kernel (including Android phones) are affected!

This room will provide an overview of the vulnerability, as well as give you an opportunity to exploit it for yourself in the vulnerable machine attached to this task. We will start by taking a look at the vulnerability and exploit at a high-level, before moving on to exploiting the vulnerability in subsequent sections of the room.

**Answer to the questions of this section-**

No Answer needed

**Task 2 Tutorial Exploit Background –**

Due to their low-level nature, any in-depth discussion of kernel vulnerabilities quickly becomes rather complicated. As such we will keep the information in this task relatively light, in the interests of keeping the information easy to digest. If you would like to read an in-depth explanation of the vulnerability (including a full code analysis), you are highly encouraged to check out Max Kellerman's original blog post - <https://dirtypipe.cm4all.com/>

Before continuing with this task, it is important to note that Dirty Pipe has been fixed in Linux kernel versions 5.16.11, 5.15.25 and 5.10.102, so if you use or manage any Linux (or Android) devices, make sure that they are running a kernel greater than one of these versions!

**Theory**

In order to understand how Dirty Pipe works, we must first look at how the Linux Kernel manages memory.

The smallest unit of memory controlled by the CPU is called a page — these are usually about 4Kib in modern systems. Of relevance to this room, pages are used when reading and writing files from the disk, although they have many other uses. The part of the kernel that manages pages is referred to as the page cache.

When a process attempts to open a file, the kernel loads it into pages which are then made available to the userland process. This userland access can be granted either by being copied into user-space memory, or by keeping the pages in kernel-space but making them accessible via system calls (effectively special functions which are used to interact with the kernel).

Okay, so what does this have to do with Dirty Pipe? The vulnerability arises because of how the kernel implements "pipes".

You will likely be familiar with the idea of "anonymous pipes" — these are featured in most shell scripting languages and allow you to pass data between two processes, usually with the pipe character (|). Pipes are characterised as having two ends — one for reading, and one for writing; you can see this for yourself in the way that anonymous pipes take the standard output of a process and write it into a pipe where it can be read by the standard input of the next program in the chain.

Back on topic: the Linux kernel provides a system call called "splice()", which is effectively a shortcut designed to speed up the process of pushing the contents of a file into a pipe. This optimisation is achieved by moving references to the pages storing the file contents, rather than moving the entirety of the data. In other words, splice() allows us to point a pipe at a page which is already loaded into memory, containing a section of a file originally opened by a process requesting read-only access. See where this is going?

By splicing a page into the pipe then writing our own arbitrary data to the pipe, we can overwrite the contents of the page!

It's not quite that simple, however; we are still missing one final piece of the puzzle. Usually when you write to a pipe after splicing a file, no changes actually happen on the disk. Changes are made to the page buffer (in memory), but nothing is permanently altered and the file itself remains unchanged. So, how do we force the kernel to write the contents of the relevant page(s) back to the disk?

This is the real crux of the vulnerability, and it can all be traced back to two commits in the Linux kernel:

A bug was introduced in Linux Kernel v4.9 (2016) which allowed pipes to be created with arbitrary flags. None of the flags available at the time were in any way dangerous, so this wasn't an issue, until...

Linux Kernel v5.8 (2020) added a new flag — PIPE\_BUF\_FLAG\_CAN\_MERGE . In simple terms, this flag tells the kernel that the changes written to the page pointed to by the pipe should be written directly back to the file that the page comes from.

**To summarise:** we have a flag that allows us to tell the kernel to write the page back to the disk as data comes into it, we have a bug that allows us to specify arbitrary flags for a pipe, and we have a system call that inadvertently allows us to point pipes at page buffers which were opened as read-only. What could possibly go wrong?

Put simply, the exploit first opens a target file with the read-only flag set — in order to do this, we must choose a file that we have permission to read. The exploit then prepares a pipe in a special way, forcing the addition of the PIPE\_BUF\_FLAG\_CAN\_MERGE flag. Next, it uses splice() to make the pipe point at the desired section of the target file. Finally, it writes whatever arbitrary data that the user has specified into the pipe, thereby overwriting the page buffer and — by merit of PIPE\_BUF\_FLAG\_CAN\_MERGE — overwriting the file on the disk.

**The Effects**

So, what does this all mean?

In short, it means that, with the right code, we can arbitrarily overwrite any file on the system, provided we can open it for reading. In other words: if our user has read access over the file (regardless of other permissions or mutability) then we can also write to it. Interestingly, this also applies to read-only file systems, or otherwise protected files which the kernel would usually stop us from writing to; by exploiting the kernel vulnerability and circumventing the "usual" write methods, we also bypass these protections.

**Remediations**

Fortunately, the remediation for this vulnerability is very simple: update your kernel.

Patched versions of the Linux Kernel have been released for supported major kernel versions — specifically, the vulnerability has been patched in Linux kernel versions 5.16.11, 5.15.25 and 5.10.102.

Ensure that you apply updates to all of your Linux devices (including any Android) as soon as security patches are released.

**Answer to the questions of this section-**

No Answer needed

**Task 3 Practical A Weaponised PoC –**

**Connecting to the target**

This lab can be accessed in one of two ways:

Using the in-browser machine on the right-hand side of the screen. This should have appeared automatically when you deployed the target.

Using SSH from a local attacking machine or the TryHackMe AttackBox. The credentials for this are:

**Username: tryhackme**

**Password: TryHackMe123!**

**i.e. ssh tryhackme@10.10.5.209**

Pick a method and access the target for yourself

**The Exploit**

In the previous task we looked at the background of the exploit and how it works. In this task we will be exploiting the vulnerability for ourselves!

A copy of Max Kellerman's original proof of concept exploit code (originally found in the disclosure blog post) can be found on the target machine at **/home/tryhackme/Exploit/PoC/poc.c**. Once compiled, this exploit gives us a lot of control over how we abuse the Dirty Pipe vulnerability. Specifically, it lets us specify the file we want to overwrite, the offset we would like to overwrite it at, and the content we would like to insert. We will acquire each of these things in the following paragraphs. An interactive video clip will be provided at the end of the task to assist with comprehension and debugging of the steps outlined in the following sections.

Bearing in mind that the exploit won't let us create files (we can only overwrite information in existing files), we first need to find a file our user can read, but that still allows us to elevate our privileges. The obvious easy choice in these conditions is **/etc/passwd**. Whilst password hashes are usually stored in the restricted-access **/etc/shadow** in modern Linux systems (as opposed to being stored traditionally in **/etc/passwd)**, most Linux variants do still check to see if account password hashes are given in **/etc/passwd**. This means that we can write a user with root permissions and a known password hash directly into the passwd file!

**Note:** If you are familiar with privilege escalation due to a writeable **/etc/passwd** file, skip to the next paragraph — otherwise click on the box below to read more about this technique.

Background Knowledge: The Passwd File (Click to read)

Let's generate a password hash and form a valid passwd entry before moving on. Pick a password then use the **openssl** command to create a SHA512Crypt hash of your chosen password:

**Generating a SHA512Crypt Hash**

**tryhackme@dirty-pipe:~$ openssl passwd -6 --salt THM "PASSWORD"**

**$6$THM$eRD0Ur0SZuwDLSwf9Lb2vyC2T6/PtQUA/B0Ssm6/jsiBtpSvc6QLjhFF0XNM8odgfkxMnC4oczGuvEomrVRfz0**

Finally, insert your username and hash into this passwd entry template: **USERNAME:HASH:0:0::/root:/bin/bash.**

Your entry should look something like this:

**muiri:$6$THM$eRD0Ur0SZuwDLSwf9Lb2vyC2T6/PtQUA/B0Ssm6/jsiBtpSvc6QLjhFF0XNM8odgfkxMnC4oczGuvEomrVRfz0:0:0::/root:/bin/bash**

Note: The password: **TryHackMe123!** has been used to generate the example hash above.

As we are overwriting existing entries in the password file, we also need to add a new line on at the end of our entry. This ensures that we avoid corrupting our entry with any remnants of the previous contents of the line.

Our final content should therefore look something like this (quotes included):**'muiri:$6$THM$eRD0Ur0SZuwDLSwf9Lb2vyC2T6/PtQUA/B0Ssm6/jsiBtpSvc6QLjhFF0XNM8odgfkxMnC4oczGuvEomrVRfz0:0:0::/root:/bin/bash'**

We have our file (**/etc/passwd**) and our content (the passwd entry) — all we need now is the offset. The offset is where in the file the exploit should begin writing at — in other words, which part of the file gets overwritten.

The vulnerability won't allow us to append to the file, so we are going to have to pick an account and overwrite it. Realistically speaking, given the length of our passwd entry (hash inclusive), this will probably actually overwrite several accounts. Looking through the passwd file, the **games** account stands out as being a good candidate for a little-used account which we can afford to nuke for a few minutes. We can use **grep** with the **-b** switch to find the offset of **games** from the start of the file:

**Finding the Offset**

**tryhackme@dirty-pipe:~$ grep -b "games" /etc/passwd**

**189:games:x:5:60:games:/usr/games:/usr/sbin/nologin**

The offset is revealed to be **189**, giving us the final piece of our puzzle.

We are finally ready to go!

The program can be compiled using the following commands:

**Compilation Command**

**tryhackme@dirty-pipe:~$ cd ~/Exploit/PoC**

**tryhackme@dirty-pipe:~/Exploit/PoC$ gcc poc.c -o exploit**

**tryhackme@dirty-pipe:~/Exploit/PoC$ ls**

**poc.c exploit**

This moves to the directory containing the exploit code, then compiles it with **gcc**.

Before we perform the exploit, it's very important that we backup the **/etc/passwd** file. This is a disruptive exploit which will cause damage to the system; with the passwd file backed up, we can easily revert the damage after the exploit has been completed.

Use **cp /etc/passwd /tmp/passwd** to copy the passwd file to **/tmp**, then execute the exploit!

Your command should look something like this:

**Executing the Exploit**

**tryhackme@dirty-pipe:~/Exploit/PoC$ ./exploit /etc/passwd 189 'USERNAME:HASH:0:0::/root:/bin/bash**

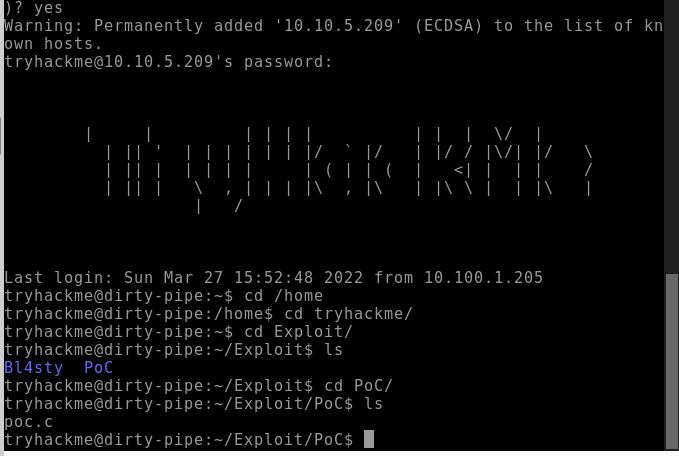
**> '**

The entire process as described in this task is demonstrated in the clip attached with this task in tryhackme:

**Answer the questions below**

**Steps to exploit vulnerability-**

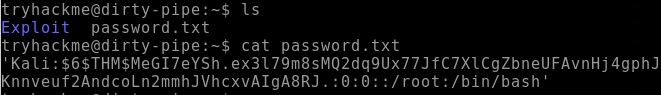
**1)** PoC of our exploit is available at cd /Exploit/PoC directory

****

**2)** first use openssl command to create a SHA512 crypt hash ( generate a password hash and form a valid passwd entry)

**C:\Users\shefali\Pictures\THM\3DP.jpg**

**3) now** save inserted username and created hash into this passwd entry template. (including quotes)

****

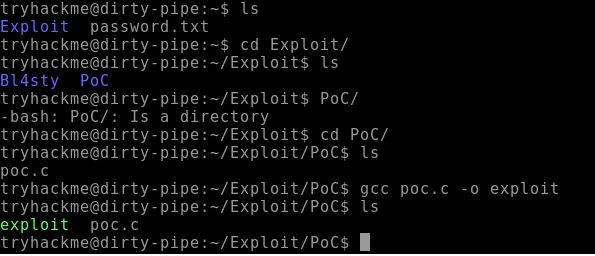
**4)** find the offset using below mentioned command so that out exploit can test writing arbitrary command to the identified offset

**C:\Users\shefali\Pictures\THM\5DP.jpg**

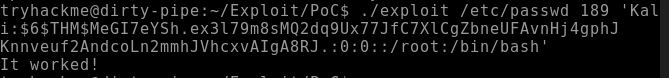
**5)**  Remember to copy /etc/passwd to /tmp this will help us in setting up backup again after cleaning up activities of our disruptive exploit.

**C:\Users\shefali\Pictures\THM\6DP.jpg**

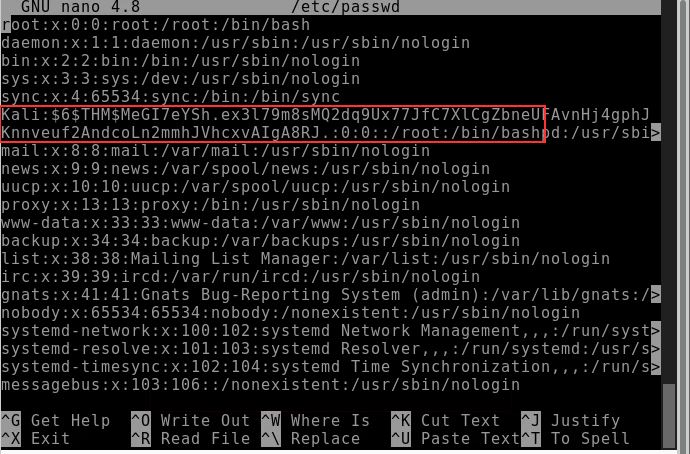
**6)**  compile poc.c exploit using gcc and output it to exploit file for execution

****

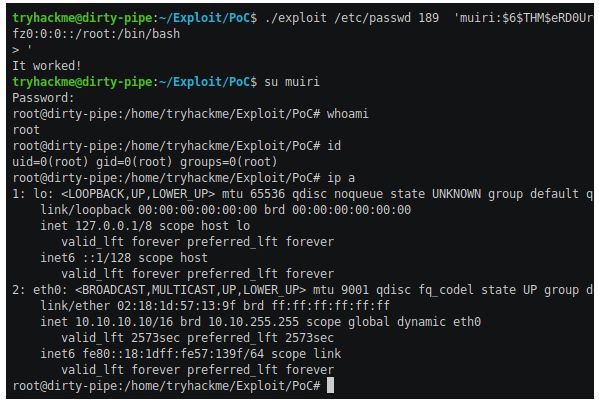
**7)** Below is the proof of that our exploit that it worked successfully

****

**8)** and our new username with hash got added to the list of /etc/passwd

****

**9) Below is how our complete execution steps look like**

****

**10)** now we can fetch our flag from/root/flag.txt after switching user into newly created account - su Kali, enter the password and capture the flag.

**Task 4 Practical Bonus Task A Second Exploit-**

n the previous task, we exploited the Dirty Pipe vulnerability using Max Kellerman's original proof of concept exploit code; however, other exploits have since been released. The original PoC allowed us to overwrite any file with arbitrary data at an offset of our choosing; however, other implementations have abused the arbitrary file write vulnerability in a variety of different ways.

To demonstrate this concept, a second exploit script has been added to the lab machine — this can be found on the target at /home/tryhackme/Exploit/Bl4sty/dirtypipez.c. As the directory structure suggests, this implementation was coded by @bl4sty, a security researcher who you may remember if you have already completed the "SudoVulns: Baron Samedit" room. The original exploit code can be downloaded from bl4sty's website here; however, as mentioned previously, a copy has already been added to the lab machine.

This exploit takes the arbitrary file write one stage further by abusing a rather special quality of the vulnerability. SUID Programs usually lose their SUID bit when you attempt to write to them; however, this does not happen with Dirty Pipe — in other words, we can write to any program that has permission to execute with higher privileges, without inadvertently destroying that extra permission (as would usually happen).

Bl4sty's exploit capitalises on this. Instead of overwriting a file like /etc/passwd, it overwrites a user-specified SUID binary (such as /bin/su), injecting shellcode into it which then gets executed with the permissions of the privileged user (i.e. root). Specifically, the exploit hijacks the chosen SUID binary and forces it to create a backdoor binary in /tmp which has the SUID bit and calls /bin/sh. It then restores the targeted SUID binary to full working order by re-adding the overwritten section, and uses the newly created backdoor to grant the attacker a shell as the privileged user.

**Exploitation**

Before continuing with this task, please ensure that you have exited your session as the root user. You should once again be executing commands in the context of the tryhackme user.

As the tryhackme user, compile the exploit using the same syntax as was given in the previous task, e.g.:

**Compilation Command**

**tryhackme@dirty-pipe:~/Exploit/Bl4sty$ gcc dirtypipez.c -o exploit**

**tryhackme@dirty-pipe:~/Exploit/Bl4sty$ ls**

**dirtypipez.c exploit**

With the exploit compiled, it should be run with a single argument specifying a target binary, owned by root and with the SUID bit set, for example: ./exploit /bin/su.

You should now once again have a root shell!

**Answer to the questions of this section-**

No Answer needed

That is all for this Write-up, hoping this will help you in solving the challenges of Dirty Pipe: CVE-2022-0847 room. Have Fun and Enjoy Hacking! Do visit other rooms and modules on TryHackMe for more learning.

-by Shefali Kumai