

Dangerous Device: Exploiting CVE-2026-31695 in the Linux Kernel via struct device

Alexander Popov

■ **positive technologies**



21.10.2026

Who Am I

- Alexander Popov
- Linux kernel developer since 2012
- Maintainer of some free software projects
- Principal Security Researcher and Head of

Open Source Program Office at  **positive technologies**

- Conference speaker:

Zer0Con, OffensiveCon, H2HC, Nullcon Goa, Linux Security Summit, Still Hacking Anyway, HITB, Positive Hack Days, ZeroNights, HighLoad++, Open Source Summit, TechLeadConf, OS Day, LinuxCon Japan, Linux Plumbers...

a13xp0p0v.github.io/conference_talks



I Will Tell You a Story

- ❶ **CVE-2026-31695** is complicated
- ❷ My exploit for it is complicated
(and beautiful)
- ❸ Describing them all at once would be difficult
- ❹ I will tell the story of this research
in chronological order 🕒
- ❺ **Let's go** 🏎️



Once Upon a Time

- My customized `syzkaller` fuzzer encountered an ordinary crash:

```
BUG: KASAN: slab-use-after-free in __pm_runtime_resume+0xe2/0xf0
Read of size 2 at addr ffff88810cfc46f8 by task pm/606
```

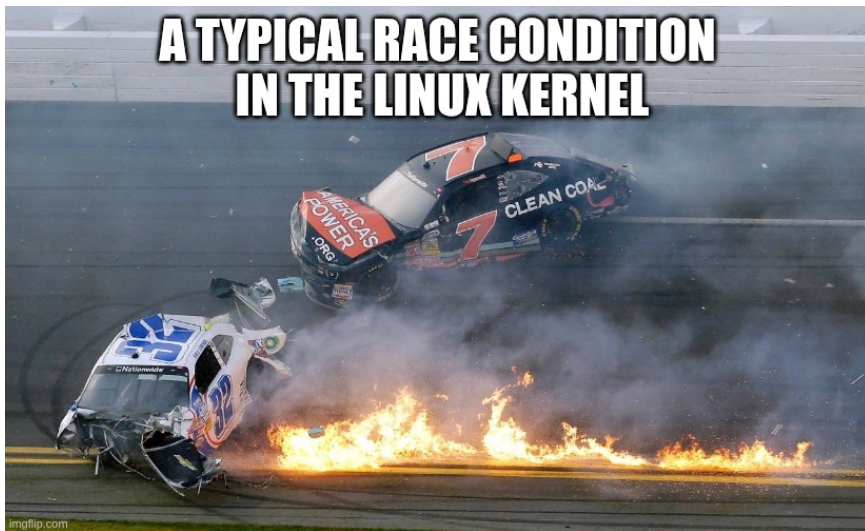
- A use-after-free (UAF) read of 2 bytes – **not very impressive**
- But I decided to spend some time on it

What? Again?

- A `syzlang` reproducer worked reliably on 24 threads:

```
# ./syz-execprog -executor=./syz-executor -arch=amd64 -sandbox=namespace \  
-procs=24 -threaded=1 -repeat=0 -cover=0 pm_log
```

- Decreasing the number of threads made the reproducer less reliable
- `syz-prog2c` failed to generate a working `C` reproducer
- So I realized that I found a weird race condition in the Linux kernel again 😓

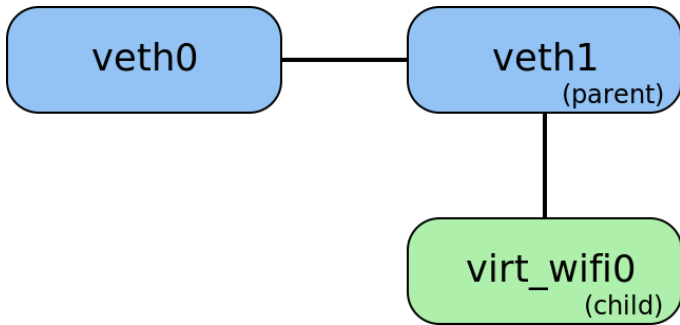


Was It a Vulnerability?

- I chose **Fedora Server** as a target of my offensive research
- I found the minimal kernel compile-time configuration required to trigger the crash:
 - `CONFIG_USER_NS=y`
 - `CONFIG_VETH=m`
 - `CONFIG_VIRT_WIFI=m`
- I managed to provoke my crash as an unprivileged user
- So it was a **vulnerability**
- I started to develop a working C reproducer manually (`syz-prog2c` failed)

Preparing Network Interfaces for Racing

- `unshare(CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWNS)`
- Create the following virtual network interfaces:



Organizing the Race

- Slowly, step by step, I managed to create an unstable `C` reproducer of the UAF
- It used only `2 pthreads`
- To provoke the memory corruption, I needed only `2 netlink`-messages:
 - The `RTM_DELLINK` routing message for the parent `veth1`
 - The `ETHTOOL_MSG_FEATURES_SET` command for the child `virt_wifi0`



Fuzzing Artifact

- The fuzzer generated the `netlink`-message for `ethtool` as the following binary blob:

```
* (uint64_t *) 0x200000000540 = 0;
* (uint32_t *) 0x200000000548 = 0;
* (uint64_t *) 0x200000000550 = 0x200000000500;
* (uint64_t *) 0x200000000500 = 0x200000000040;
memcpy((void *) 0x200000000040, "<\000\000\000", 4);
* (uint16_t *) 0x200000000044 = r[1];
memcpy((void *) 0x200000000046,
    "\x01\x00\x2d\xbd\x70\x00\xfb\xdb\xdf\x25\x0c\x00\x00\x00\x10\x00\x03"
    "\x80\x04\x00\x01\x00\x08\x00\x02\x00\x75\x00\x00\x00\x18\x00\x01\x80"
    "\x14\x00\x02\x00\x76\x69\x72\x74\x5f\x77\x69\x66\x69\x30\x00",
    49);
* (uint64_t *) 0x200000000508 = 0x3c;
* (uint64_t *) 0x200000000558 = 1;
* (uint64_t *) 0x200000000560 = 0;
* (uint64_t *) 0x200000000568 = 0;
* (uint32_t *) 0x200000000570 = 0x8800;
syscall(__NR_sendmsg, /*fd=*/r[0], /*msg=*/0x200000000540ul, /*f=MSG_PROBE|MSG_EOR*/ 0x90ul);
```

- Awful 

- I needed to reverse-engineer it to **understand** the kernel bug

Fighting Together With an LLM

- The Linux kernel has very poor documentation about the [netlink](#) interface for [ethtool](#)

Folk Wisdom

The best Linux kernel documentation is the source code



- So I tried to do reverse engineering **together with LLM**
- I used a local open-weight LLM ([Qwen3.5-397B-A17B-FP8](#) at that time)

Fighting ~~Together With~~ Against an LLM

- I just realized that:

The LLM is trying to justify the weird bytes generated by the fuzzer 🤪

- So my collaboration with the LLM looked like that:

Fighting Together With Against an LLM

- I just realized that:

The LLM is trying to justify the weird bytes generated by the fuzzer 🤪

- So my collaboration with the LLM looked like that:



I Developed the Reproducer and Understood the Bug

CPU 0

```
netlink_rcv_skb()
rtnetlink_rcv_msg()
rtnl_dellink()
rtnl_net_lock()
rtnl_lock()
rtnl_delete_link()
rtnl_net_unlock()
rtnl_unlock()
netdev_run_todo()
/* for virt_wifi0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth1 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);

dev = netdev_wait_allrefs_any(&list);
/* got veth1 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

dev = netdev_wait_allrefs_any(&list);
/* got veth0 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

/* wait for virt_wifi0 */
dev = netdev_wait_allrefs_any(&list);
```



prevents
returning
virt_wifi0

causes UAF

CPU 1

```
genl_rcv_msg()
genl_family_rcv_msg()
genl_family_rcv_msg_doit()
ethnl_set_features()
ethnl_parse_header_dev_get() /* for virt_wifi0 */
rtnl_lock()
netdev_lock_ops()
ethnl_ops_begin()
if (dev->dev.parent)
    pm_runtime_get_sync(dev->dev.parent) /* UAF */
```



My First Fixing Patch

```
--- a/net/ethtool/netlink.c
+++ b/net/ethtool/netlink.c
@@ -94,17 +94,16 @@ int ethnl_ops_begin(struct net_device *dev)
     if (!dev)
         return -ENODEV;

-    if (dev->dev.parent)
-        pm_runtime_get_sync(dev->dev.parent);
-
     netdev_ops_assert_locked(dev);

     if (!netif_device_present(dev) ||
         dev->reg_state >= NETREG_UNREGISTERING) {
-        ret = -ENODEV;
-        goto err;
+        return -ENODEV;
+    }

+    if (dev->dev.parent)
+        pm_runtime_get_sync(dev->dev.parent);
+
     if (dev->ethtool_ops->begin) {
         ret = dev->ethtool_ops->begin(dev);
         if (ret)
```



Responsible Disclosure

- 1 I reported the vulnerability and provided my patch to security@kernel.org

Responsible Disclosure

- ① I reported the vulnerability and provided my patch to security@kernel.org
- ② The maintainers recommended that I send the patch to the public LKML

Responsible Disclosure

- ❶ I reported the vulnerability and provided my patch to security@kernel.org
- ❷ The maintainers recommended that I send the patch to the public LKML
- ❸ I thought: **"Hm... OK, let's do the full disclosure"**

Responsible Disclosure

- ❶ I reported the vulnerability and provided my patch to security@kernel.org
- ❷ The maintainers recommended that I send the patch to the public LKML
- ❸ I thought: **"Hm... OK, let's do the full disclosure"**
- ❹ My patch provoked a discussion on LKML:
lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u

Responsible Disclosure

- ① I reported the vulnerability and provided my patch to security@kernel.org
- ② The maintainers recommended that I send the patch to the public LKML
- ③ I thought: **"Hm... OK, let's do the full disclosure"**
- ④ My patch provoked a discussion on LKML:
lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
- ⑤ Surprisingly, the reaction from some kernel maintainers was somewhat **harsh**
 - I don't name them, see the mailing list

Responsible Disclosure

- ❶ I reported the vulnerability and provided my patch to security@kernel.org
- ❷ The maintainers recommended that I send the patch to the public LKML
- ❸ I thought: **"Hm... OK, let's do the full disclosure"**
- ❹ My patch provoked a discussion on LKML:
lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
- ❺ Surprisingly, the reaction from some kernel maintainers was somewhat **harsh**
 - I don't name them, see the mailing list
- ❻ The maintainers decided **not to fix** this bug in [ethtool](#)

Responsible Disclosure

- ❶ I reported the vulnerability and provided my patch to security@kernel.org
- ❷ The maintainers recommended that I send the patch to the public LKML
- ❸ I thought: **"Hm... OK, let's do the full disclosure"**
- ❹ My patch provoked a discussion on LKML:
lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
- ❺ Surprisingly, the reaction from some kernel maintainers was somewhat **harsh**
 - I don't name them, see the mailing list
- ❻ The maintainers decided **not to fix** this bug in [ethtool](#)
- ❼ They asked me to prepare a patch for [virt_wifi](#) instead

My Second Fixing Patch

Currently we execute 'SET_NETDEV_DEV(dev, &priv->lowerdev->dev)' for the virt_wifi net devices. However, unregistering a virt_wifi device in netdev_run_todo() can happen together with the device referenced by SET_NETDEV_DEV().

...

Let's remove SET_NETDEV_DEV for virt_wifi...

--- a/drivers/net/wireless/virtual/virt_wifi.c

+++ b/drivers/net/wireless/virtual/virt_wifi.c

@@ -557,7 +557,6 @@ static int virt_wifi_newlink(struct net_device *dev,

eth_hw_addr_inherit(dev, priv->lowerdev);

netif_stacked_transfer_operstate(priv->lowerdev, dev);

- SET_NETDEV_DEV(dev, &priv->lowerdev->dev);

dev->ieee80211_ptr = kzalloc_obj(*dev->ieee80211_ptr);

if (!dev->ieee80211_ptr) {



- My second patch was applied to the Linux kernel mainline (commit [789b06f9f39c](#))
- This vulnerability had been present in the kernel code for nearly **5 years**
(introduced by commit [d43c65b05b84](#))
- The fix was backported to the following stable and longterm kernel branches:
 - [v6.19](#) (stable branch at that time)
 - [v6.18](#)
 - [v6.12](#)
 - [v6.6](#)
 - [v6.1](#)
 - [v5.15](#)
- I asked to assign a CVE identifier to this bug
- Greg Kroah-Hartman announced **CVE-2026-31695**





Now about exploiting CVE-2026-31695



CVE-2026-31695: Memory Corruption

- To see all memory corruptions, I enabled `kasan_multi_shot`
- `KASAN` showed **91 UAF accesses** in `pm_runtime_get_sync()` 😲
- UAF was performed on the `struct device` inside the freed `struct net_device`
- These kernel objects live in the `kmalloc-cg-4k` slab cache:

```
The buggy address belongs to the object at ffff8880275ac000
  which belongs to the cache kmalloc-cg-4k of size 4096
The buggy address is located 1888 bytes inside of
  freed 4096-byte region [ffff8880275ac000, ffff8880275ad000)
```

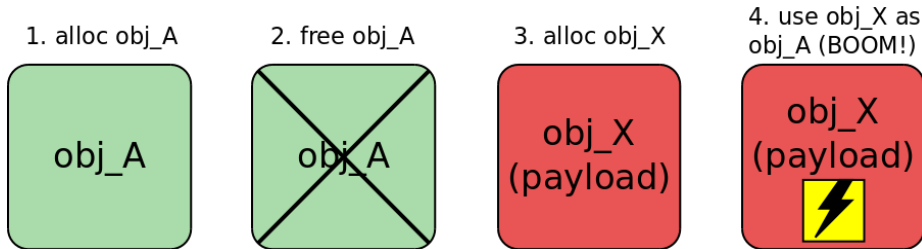
First Thoughts on Exploit Strategy

- Initially, the kernel **didn't** dereference any pointer from the freed memory
- `pm_runtime_get_sync()` performed reads and writes of only **1, 2, or 4 bytes** inside the freed `struct device`
- 👉 Dereferencing a pointer located in the freed `struct device` would provide AARW
 - I didn't find such an exploit primitive in `pm_runtime_get_sync()`, **alas!**
- 👍 The kernel didn't do anything causing a crash
 - It was my chance to develop a **100% stable** PoC exploit



Classical UAF Exploitation

- I planned to perform **kernel heap spraying + type confusion**



- I.e. I wanted to make the kernel allocate controlled data
at the place of the freed `struct device`

The Main Idea of CVE-2026-31695 Exploitation



What I Needed for CVE-2026-31695 Exploitation

- 1 Reliably hit the race condition between `RTM_DELLINK` and `ETHTOOL_MSG_FEATURES_SET`



What I Needed for CVE-2026-31695 Exploitation

- 1 Reliably hit the race condition between `RTM_DELLINK` and `ETHTOOL_MSG_FEATURES_SET`
- 2 Find out how to perform **kernel heap spraying** to overwrite the freed `struct device` with controlled data



What I Needed for CVE-2026-31695 Exploitation

- 1 Reliably hit the race condition between `RTM_DELLINK` and `ETHTOOL_MSG_FEATURES_SET`
- 2 Find out how to perform **kernel heap spraying** to overwrite the freed `struct device` with controlled data
- 3 Overwrite `struct device` **before** UAF in `pm_runtime_get_sync()`



What I Needed for CVE-2026-31695 Exploitation

- 1 Reliably hit the race condition between `RTM_DELLINK` and `ETHTOOL_MSG_FEATURES_SET`
- 2 Find out how to perform **kernel heap spraying** to overwrite the freed `struct device` with controlled data
- 3 Overwrite `struct device` **before** UAF in `pm_runtime_get_sync()`
- 4 Make `pm_runtime_get_sync()` do something **dangerous** when it works with my fake `struct device`



Entering the Obstacle Race...

CPU 0

```
netlink_rcv_skb()
rtnetlink_rcv_msg()
rtnl_dellink()
rtnl_net_lock()
rtnl_lock()
rtnl_delete_link()
rtnl_net_unlock()
rtnl_unlock()
netdev_run_todo()
/* for virt_wifi0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth1 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);

dev = netdev_wait_allrefs_any(&list); ← prevents returning virt_wifi0
/* got veth1 */
free_netdev(dev);
kobject_put(&dev->dev.kobj); ← causes UAF

dev = netdev_wait_allrefs_any(&list);
/* got veth0 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

/* wait for virt_wifi0 */
dev = netdev_wait_allrefs_any(&list);
```



CPU 1

```
genl_rcv_msg()
genl_family_rcv_msg()
genl_family_rcv_msg_doit()
ethnl_set_features()
ethnl_parse_header_dev_get() /* for virt_wifi0 */
rtnl_lock()
netdev_lock_ops()
ethnl_ops_begin()
if (dev->dev.parent)
    pm_runtime_get_sync(dev->dev.parent) /* UAF */
```



Entering the Obstacle Race!

CPU 0

CPU 1

```
netlink_rcv_skb()
rtnetlink_rcv_msg()
rtnl_dellink()
rtnl_net_lock()
rtnl_lock()
rtnl_delete_link()
rtnl_net_unlock()
rtnl_unlock()
netdev_run_todo()
/* for virt_wifi0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth1 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);

dev = netdev_wait_allrefs_any(&list);
/* got veth1 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

dev = netdev_wait_allrefs_any(&list);
/* got veth0 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

/* wait for virt_wifi0 */
dev = netdev_wait_allrefs_any(&list);
```



Obstacle #1

prevents
returning
virt_wifi0

causes UAF

Obstacle #3

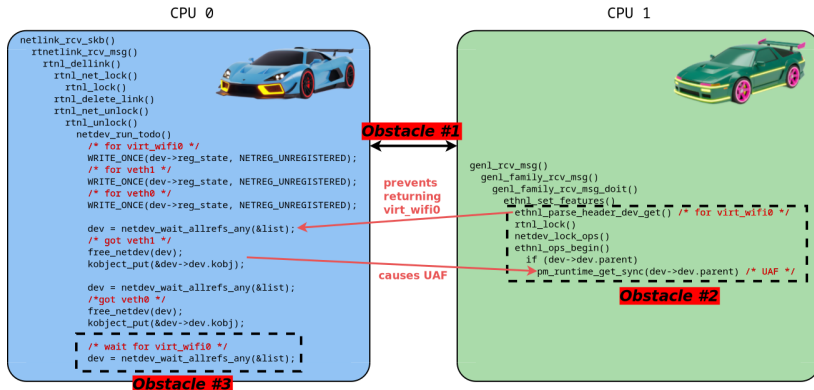
```
genl_rcv_msg()
genl_family_rcv_msg()
genl_family_rcv_msg_doit()
ethnl_set_features()
ethnl_parse_header_dev_get() /* for virt_wifi0 */
rtnl_lock()
netdev_lock_ops()
ethnl_ops_begin()
if (dev->dev.parent)
    pm_runtime_get_sync(dev->dev.parent) /* UAF */
```



Obstacle #2

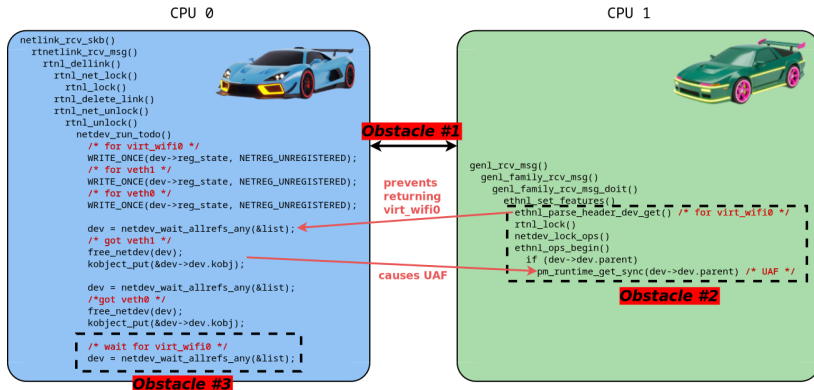
Obstacle Race

- ❶ Disabling **KASAN** and removing **printk()** debugging **broke the reproducer**



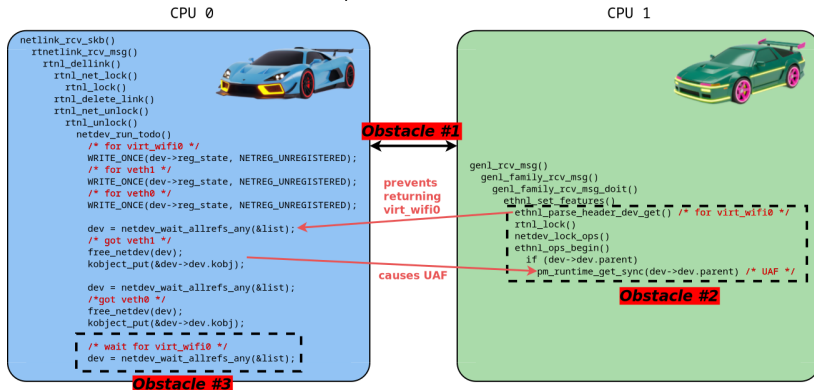
Obstacle Race

- 1 Disabling **KASAN** and removing **printk()** debugging **broke the reproducer**
- 2 The race window became **too small** for reallocating **struct device** (especially if I need a cross-cache attack to bypass the security hardening features in slab allocator)



Obstacle Race

- 1 Disabling **KASAN** and removing **printk()** debugging **broke the reproducer**
- 2 The race window became **too small** for reallocating **struct device** (especially if I need a cross-cache attack to bypass the security hardening features in slab allocator)
- 3 UAF in thread #1 happens **earlier** than thread #0 finishes sending **RTM_DELLINK** (I can't do **heap spraying in thread #0**)



Looked Closely at the Code

- I can **block the thread #1** in `rtnl_lock()`!

CPU 0

CPU 1

```
netlink_rcv_skb()
rtnetlink_rcv_msg()
rtnl_dellink()
rtnl_net_lock()
rtnl_lock()
rtnl_delete_link()
rtnl_net_unlock()
rtnl_unlock()
netdev_run_todo()
/* for virt_wifi0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth1 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);

dev = netdev_wait_allrefs_any(&list);
/* got veth1 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

dev = netdev_wait_allrefs_any(&list);
/* got veth0 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

/* wait for virt_wifi0 */
dev = netdev_wait_allrefs_any(&list);
```

Obstacle #3



Obstacle #1

prevents
returning
virt_wifi0

causes UAF

```
genl_rcv_msg()
genl_family_rcv_msg()
genl_family_rcv_msg_doit()
ethnl_set_features(dev->dev.parent) /* for virt_wifi0 */
ethnl_parse_ops(dev->dev.parent)
rtnl_lock()
netdev_lock_ops()
ethnl_ops_begin()
if (dev->dev.parent)
    pm_runtime_get_sync(dev->dev.parent) /* UAF */
```

Obstacle #2



Eureka: I Created 512 Threads #1, Which Call `rtnl_lock()`

Eureka: I Created 512 Threads #1, Which Call `rtnl_lock()`



100% Successful Obstacle Race

CPU 0

```
netlink_rcv_skb()
rtnetlink_rcv_msg()
rtnl_dellink()
rtnl_net_lock()
rtnl_lock()
rtnl_delete_link()
rtnl_net_unlock()
rtnl_unlock()
netdev_run_todo()
/* for virt_wifi0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth1 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);
/* for veth0 */
WRITE_ONCE(dev->reg_state, NETREG_UNREGISTERED);

dev = netdev_wait_allrefs_any(&list);
/* got veth1 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

dev = netdev_wait_allrefs_any(&list);
/* got veth0 */
free_netdev(dev);
kobject_put(&dev->dev.kobj);

/* wait for virt_wifi0 */
dev = netdev_wait_allrefs_any(&list);
```



Obstacle #1

prevents
returning
virt_wifi0

causes UAF

Obstacle #3

CPU 1

```
genl_rcv_msg()
genl_family_rcv_msg()
genl_family_rcv_msg_doit()
ethnl_set_features(dev, dev_get()) /* for virt_wifi0 */
rtnl_lock()
netdev_lock_ops()
ethnl_ops_begin()
if (dev->dev.parent)
    pm_runtime_get_sync(dev->dev.parent) /* UAF */
```



512 *

Obstacle #2

C'mon, Do Something Dangerous!

- I needed to make `pm_runtime_get_sync()` do **something dangerous** with `struct device`
- Deep inside, I found an opportunity for control flow hijacking

```
pm_runtime_get_sync()  
    __pm_runtime_resume()  
        rpm_resume()  
            callback = RPM_GET_CALLBACK(dev, runtime_resume)  
            rpm_callback(callback, dev)
```



- So I needed to construct a fake `struct device` that allows to reach this code



Slab Allocator Security Hardening

Fedora Server enables the security hardening features in slab allocator:

① `CONFIG_RANDOM_KMALLOC_CACHES`

- Creates multiple copies of slab caches
- Makes `kmalloc()` randomly pick one based on the code address
- That restricts the ability to perform **heap spraying**
(a **cross-cache attack** is needed to bypass it)

② `CONFIG_SLAB_BUCKETS`

- Creates a bunch of separate slab caches for kernel objects containing userspace data
- Moves the `msg_msg` kernel objects to separate slab caches
(I first performed heap spraying with `msg_msg` in my research
[Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel](#))

Wait, Where is Security Hardening?

- `struct net_device` lives in the `kmalloc-cg-4k` slab cache
- I was going to perform the cross-cache attack, but...
- Imagine **my surprise** to see the following output:

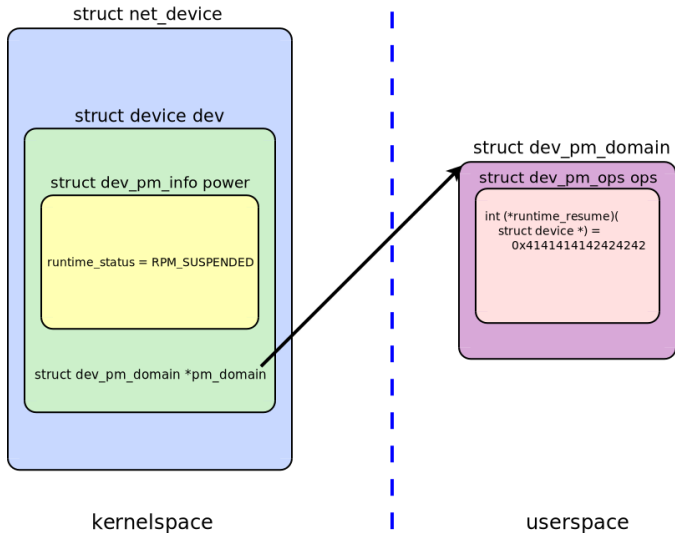
```
$ sudo grep kmalloc /proc/slabinfo |grep 4k
kmalloc-cg-4k      641      808  4096    8    8 : tunables    0    0    0 : slabdata   101    101    0
dma-kmalloc-4k     0         0  4096    8    8 : tunables    0    0    0 : slabdata    0     0    0
kmalloc-rcl-4k     0         0  4096    8    8 : tunables    0    0    0 : slabdata    0     0    0
kmalloc-rnd-15-4k  60        64  4096    8    8 : tunables    0    0    0 : slabdata    8     8    0
kmalloc-rnd-14-4k  28        32  4096    8    8 : tunables    0    0    0 : slabdata    4     4    0
kmalloc-rnd-13-4k  48        48  4096    8    8 : tunables    0    0    0 : slabdata    6     6    0
...
```

- `CONFIG_RANDOM_KMALLOC_CACHES` doesn't affect `kmalloc-cg-4k` 😲
- I performed **naive heap spraying** by creating network packets (`sk_buff.data` objects)

Defeating RANDOM_KMALLOC_CACHES by “Doing Nothing at All” ©



Reaching Control Flow Hijack: Contents of struct device



Reaching Control Flow Hijack in the Fedora Server Kernel



```
[!] let's go racing!
[ 37.213592] Oops: general protection fault, kernel NULL pointer dereference 0x0: 0000 [#1] SMP NOPTI
[ 37.214684] CPU: 1 UID: 1000 PID: 128840 Comm: pm Not tainted 6.19.10-300.fc44.x86_64 #1 PREEMPT(lazy)
[ 37.215775] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.17.0-10.fc44 06/10/2025
[ 37.216827] RIP: 0010:__pi__x86_indirect_thunk_r13+0xa/0x20
...
[ 37.219661] RSP: 0018:ffffc9000469f6f8 EFLAGS: 00010206
[ 37.220313] RAX: 0000000000000000 RBX: ffff888105fd55b0 RCX: 0000000000000017
[ 37.221138] RDX: 0000000000000000 RSI: ffff888105fd55b0 RDI: ffff888105fd55b0
[ 37.222060] RBP: 4141414142424242 R08: ffffffff84c03040 R09: 00000000bcca622b
[ 37.222985] R10: 0000000000000004 R11: 0000000000000001 R12: ffff888105fd5694
[ 37.223855] R13: 4141414142424242 R14: 0000000000000000 R15: 0000000000000000
[ 37.224715] FS: 00007fbc30aea6c0(0000) GS:ffff8881b73e6000(0000) knlGS:0000000000000000
[ 37.225683] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 37.226388] CR2: 0000000000489610 CR3: 0000000117acf000 CR4: 00000000000006f0
[ 37.227227] Call Trace:
[ 37.227526] <TASK>
[ 37.227790] ? __rpm_callback+0x57/0x210
...
```



Wake Up, Neo (an Unexpected Trouble)

- Then I implemented a basic **ret2usr attack**
(instead of calling dummy `0x4141414142424242`)
- Performed privilege escalation
- But after executing the `runtime_resume()` callback, the Linux kernel...
- ...Crashed in `rpm_resume()` trying to **wake up** threads in `device.power.wait_queue` 🙈

```
[ 19.170510] BUG: kernel NULL pointer dereference, address: 0000000000000000
...
[ 19.177543] RIP: 0010:__wake_up_common+0x49/0xd0
...
[ 19.192334] Call Trace:
[ 19.192697] <TASK>
[ 19.193037] __wake_up+0x36/0x60
[ 19.193562] rpm_resume+0x442/0x760
```

Don't Wake up, Neo

```
static int __wake_up_common(struct wait_queue_head *wq_head, unsigned int mode,
                           int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_entry_t *curr, *next;

    lockdep_assert_held(&wq_head->lock);

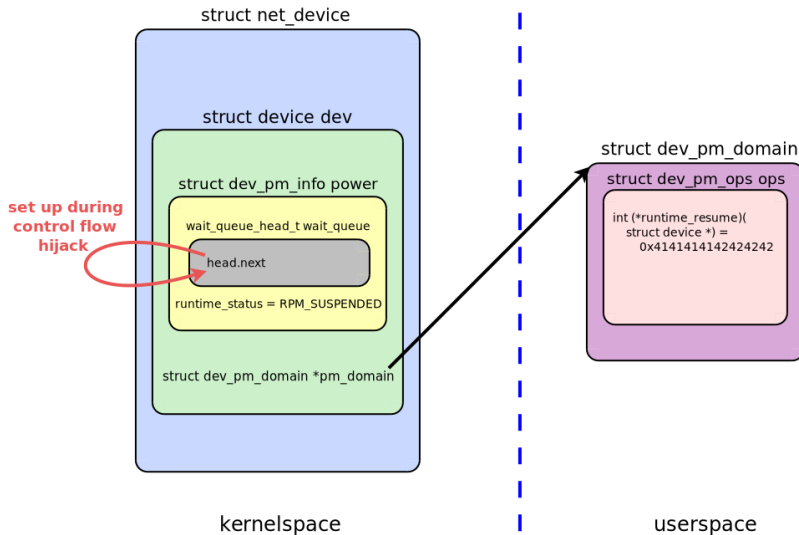
    curr = list_first_entry(&wq_head->head, wait_queue_entry_t, entry);

    if (&curr->entry == &wq_head->head)
        return nr_exclusive; /* <-- THIS IS MY CHANCE! */

    list_for_each_entry_safe_from(curr, next, &wq_head->head, entry) {
```

- I needed to fake an empty linked list in `device.power.wait_queue`
- I needed to make `device.power.wait_queue.head` contain the pointer to itself
- **But how could I achieve that without knowing its kernel address?** 🥵

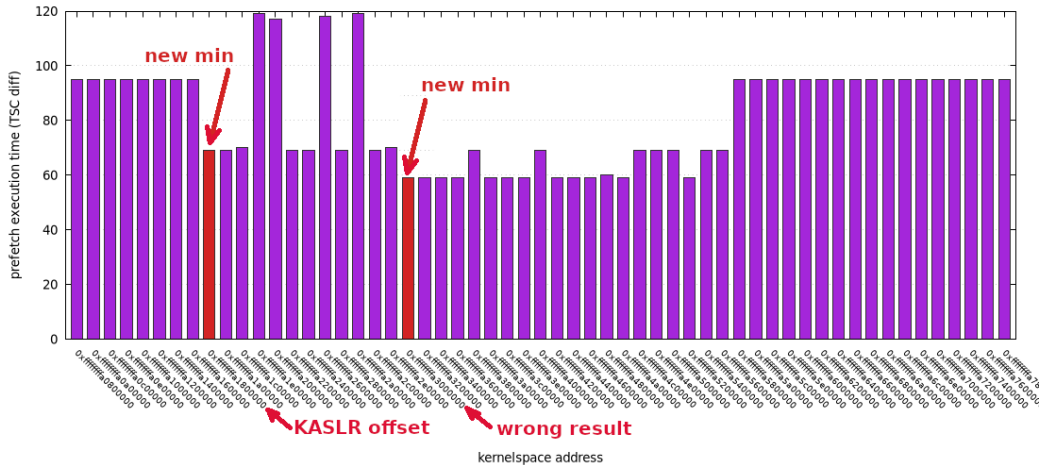
Eureka: I Can Do Everything During the Control Flow Hijack



Bypassing KASLR

- To bypass **KASLR**, I decided to use the **Prefetch Side-Channel Attack**
- The **kernelCTF** players usually copy the same KASLR bypassing code from each other
- But I decided to dig deep into it
- This attack was discovered by the team from **Graz University of Technology**
- **The main idea:** the `prefetch` instruction leaks timing information
 - on the exact translation level for every virtual address
- An unprivileged userspace process can detect the **execution time difference**
 - for `prefetch` on valid and invalid addresses in kernelspace
- On Intel CPUs, `prefetch` for a valid kernel address works **faster** than for an invalid one

The kernelCTF Implementation Fails on Intel Comet Lake CPU



A Fix for the kernelCTF Implementation of the Prefetch Attack



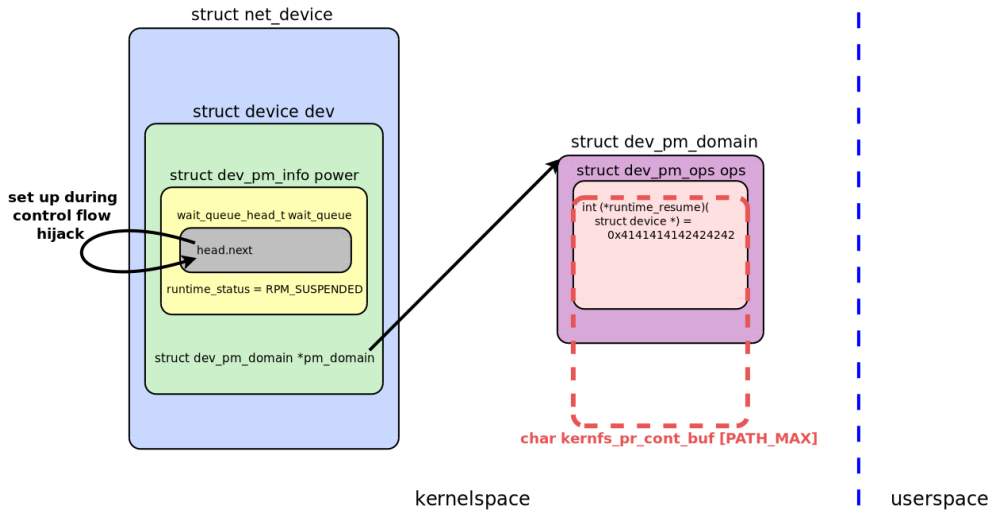
```
size_t minv = times[0];
size_t mini = 0;
for (unsigned long ti = 0; ti < ARRAY_LEN(times) - 1; ti++) {
    if (times[ti] < minv) {
        mini = ti;
+       if (times[ti] < minv * 90 / 100)
+           break; /* Stop on the first address having more than 10% faster access */
        minv = times[ti];
    }
}

bases[vote] = addrs[mini];
```



- I also start scanning at a lower address `0xffffffff80000000` instead of `0xffffffff81000000` (to make the algorithm work correctly with disabled KASLR)
- The `prefetch` attack became absolutely stable on `Intel Comet Lake CPU` – **I gained the KASLR secret**

Bypassing SMAP

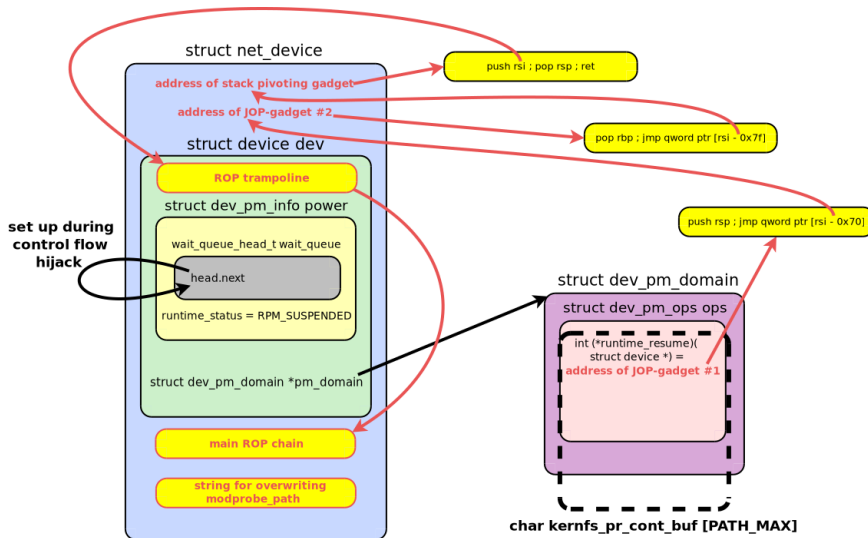


The Finals: ROP-Chain

- Building a ROP-chain was the final part of exploiting **CVE-2026-31695**
- I did that with special pleasure (**honestly, no kidding**)
- The ROP-chain had to perform the following tasks:
 - ① Disabling **SELinux**, which caused troubles for LPE
 - ② Overwriting `modprobe_path` to the LPE script path
 - ③ Creating an empty linked list in `device.power.wait_queue.head`
 - ④ Allowing the Linux kernel to continue the system call handling after my ROP chain
- The last task was **hard**
- During control flow hijack, only `RSP` contained a pointer from the kernel stack
- After the kernel stack pivoting, the `RSP` register would be **clobbered**



I Designed a ROP/JOP-Chain



Demo!



Possible Exploit Mitigations for CVE-2026-31695

❶ Disabling kernel module autoloading

- Not easy, see the details in my pet project **kernel-hardening-checker**
- That also prevents attacks that overwrite `modprobe_path` to gain LPE

❷ Clearing the kernel memory on freeing (`init_on_free`)

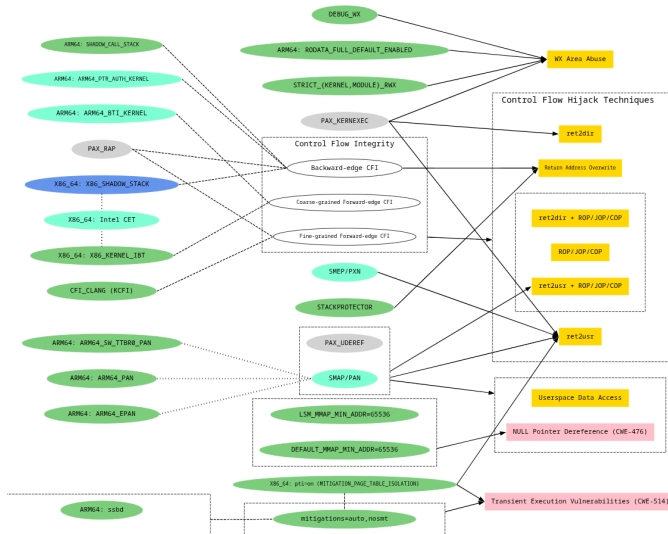
- Writes zeros to the `struct device` after `kfree()`
- Turns a silent UAF on `struct device` into a harmless kernel **WARNING**
- My exploit can still win the race and gain LPE 😎

❸ Page Table Isolation (PTI) requires some modifications in the Prefetch Side-Channel Attack

❹ Improvements in the slab allocator security hardening (I'm working on a patch)

❺ Control Flow Integrity (CFI) – see the details in my pet project **linux-kernel-defence-map**

Control Flow Hijacking and CFI in my Linux Kernel Defence Map



Finish Line

- I found, fixed in the upstream kernel, and exploited **CVE-2026-31695**
- I managed to bypass the security hardening mechanisms
and gain full control over **Fedora 44 Server**
- My exploit is very **stable**, despite the **elusive race condition**
- Did I use **LLMs** and **AI agents**? Only for reverse engineering of **netlink** blobs
- **But why?** I did this research myself **because it was a pleasure for me**
- My team experiments with LLMs and AI-agents in the Linux kernel security area
- **However, I want robots to wash the dishes and sort garbage,**
while I do creative work (not the opposite)



Thanks!

Enjoy the conference!

Contacts:

    [a13xp0p0v](#)

 alex.popov@linux.com

Blog: a13xp0p0v.tech



Channel: t.me/linkersec

