Опасное устройство: эксплуатация уязвимости CVE-2026-31695 в ядре Linux с помощью struct device

Александр Попов

positive technologies

ZERONIGHTS

30.09.2026

Кто я





- Александр Попов
- Разработчик ядра Linux с 2012 года
- Мейнтейнер ряда значимых открытых проектов
- Главный исследователь безопасности операционных систем

  и руководитель комитета по открытому коду в positive technologies
- Докладчик на конференциях:

  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.tech/conference_talks

Я расскажу историю

1. Уязвимость **CVE-2026-31695** сложная
2. Мой эксплойт для нее тоже сложный

   (и красивый)
3. Описывать их целиком чересчур сложно
4. Поэтому лучше я расскажу историю

   своего исследования

   в хронологическом порядке 🕑
5. **Поехали** 🏎️



Начало

- В один прекрасный день...
- Мой модифицированный `syzkaller` нашел обычный отказ ядра:

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

- Чтение 2 байт после освобождения – **пустяк, неинтересно**
- Но **я решил позаниматься** этим багом

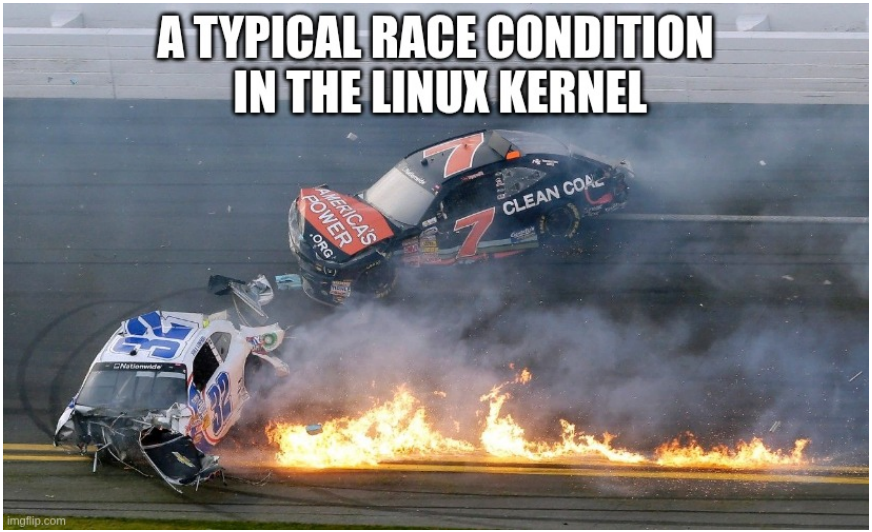
Что? Опять?

- Репродюсер на `syzlang` стабильно давал падение ядра только на 24 потоках:

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

- Уменьшение количества потоков ухудшало стабильность репродюсера
- Инструмент `syz-prog2c` не смог сгенерировать рабочий репродюсер на `C`
- Я сделал вывод, что снова поймал хитрое состояние гонки в ядре Linux 🥲

Гонки


A TYPICAL RACE CONDITION
IN THE LINUX KERNEL
AMERICAS POWER .ORG
CLEAN COA
imgflip.com

Уязвимость или просто ошибка?

- Я выбрал **Fedora Server** в качестве цели атакующего исследования
- Минимизация конфигурации ядра показала, что для UAF нужны:
  - `CONFIG_USER_NS=y`
  - `CONFIG_VETH=m`
  - `CONFIG_VIRT_WIFI=m`
- Запуск репродюсера от непривилегированного пользователя смог вызвать отказ ядра
- Значит, это **уязвимость, а не просто ошибка**!
- Я стал разрабатывать репродюсер на `C` вручную (`syz-prog2c` не справился)

Подготовка сетевых интерфейсов к гонке

- Создать новое пространство имен:

```
unshare(CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWNS);
```

- Назначить себе uid=0 и gid=0 в этом пространстве имен
- Создать сетевые интерфейсы veth0, veth1 и дочерний virt_wifi:

```
ret = netlink_add_veth(&nlmsg, sock, "veth0", "veth1");
if (ret) {
    printf("[-] netlink_add_veth\n");
    close(sock);
    return EXIT_FAILURE;
}
ret = netlink_add_linked(&nlmsg, sock, "virt_wifi", "virt_wifi0", "veth1");
close(sock);
if (ret) {
    printf("[-] netlink_add_linked\n");
    return EXIT_FAILURE;
}
```

Организация гонки

- Постепенно, шаг за шагом, я разработал рабочий репродюсер на 2 pthreads
- Для гонки нужна отправка всего 2 netlink-сообщений:
  - Сообщение маршрутизации RTM_DELLINK
  - Команда ETHTOOL_MSG_FEATURES_SET



Поток №0

- Работает на CPU 0
- Создает AF_NETLINK-сокет для протокола NETLINK_GENERIC
- Подготавливает сообщение RTM_DELLINK для удаления veth1:




```
ifindex = if_nametoindex("veth1");
if (ifindex == 0) {
    perror("[-] if_nametoindex");
    goto out;
}
memset(&ifinfo, 0, sizeof(ifinfo));
ifinfo.ifi_index = ifindex;
netlink_init(&nlmsg, RTM_DELLINK, 0, &ifinfo, sizeof(ifinfo));
```

- Синхронизируется с **потоком №1** с помощью pthread_barrier_wait()
- Выполняет netlink_send()

Поток №1

- Работает на `CPU 1`
- Создает `AF_NETLINK`-сокет для протокола `NETLINK_GENERIC`
- Определяет идентификатор `netlink`-семейства для `ethtool`
- Синхронизируется с **потоком №0** с помощью `pthread_barrier_wait()`
- Отправляет команду `ETHTOOL_MSG_FEATURES_SET` для дочернего `virt_wifi0`
- Вопрос в том, **как именно** формируется эта команда



Артефакт фаззинга

- Фаззер сгенерировал бинарный блоб `netlink`-сообщения для `ethtool`

```
*(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);
```

- Жуть 🤯
- Но мне нужно было **разобраться** в нем, чтобы **понять** ядерный баг

Работа с LLM

- Документация ядра Linux по `netlink`-интерфейсу для `ethtool` никуда не годится

Народная мудрость

Единственная нормальная документация по ядру Linux –

это его исходный код



- Поэтому я решил сначала попробовать **реверс-инжиниринг с LLM**
- Я взял локально развернутую `Qwen3.5-397B-A17B-FP8` с открытыми весами

~~Работа~~ Борьба с LLM

1. Я попросил `Qwen` написать код на `C`, создающий целевое `netlink`-сообщение
   - Получил нейрохалтуру, которая **не смогла воспроизвести баг**

~~Работа~~ Борьба с LLM

1. Я попросил `Qwen` написать код на `C`, создающий целевое `netlink`-сообщение
   - Получил нейрохалтуру, которая **не смогла воспроизвести баг**
2. Тогда я дал модели функции по работе с `netlink` из `syzkaller`
   - Сказал использовать их и не переизобретать
   - После многих попыток код от LLM стал более читаемым, но **баг не воспроизвел**

~~Работа~~ Борьба с LLM

1. Я попросил `Qwen` написать код на `C`, создающий целевое `netlink`-сообщение
   - Получил нейрохалтуру, которая **не смогла воспроизвести баг**
2. Тогда я дал модели функции по работе с `netlink` из `syzkaller`
   - Сказал использовать их и не переизобретать
   - После многих попыток код от LLM стал более читаемым, но **баг не воспроизвел**
3. Я взял `gdb` и сделал побайтовое сравнение `netlink`-блобов в памяти
   - Блобы от `syzkaller` и от LLM **отличались**, и я стал вручную устранять разницу
   - Для каждого изменения я проверял репродюсер

~~Работа~~ Борьба с LLM

1. Я попросил `Qwen` написать код на `C`, создающий целевое `netlink`-сообщение
   - Получил нейрохалтуру, которая **не смогла воспроизвести баг**
2. Тогда я дал модели функции по работе с `netlink` из `syzkaller`
   - Сказал использовать их и не переизобретать
   - После многих попыток код от LLM стал более читаемым, но **баг не воспроизвел**
3. Я взял `gdb` и сделал побайтовое сравнение `netlink`-блобов в памяти
   - Блобы от `syzkaller` и от LLM **отличались**, и я стал вручную устранять разницу
   - Для каждого изменения я проверял репродюсер
4. Кончилось тем, что я:
   - Изучил этот `netlink`-интерфейс в процессе мучений с LLM
   - Сам реализовал нужную команду `ETHTOOL_MSG_FEATURES_SET`

~~Работа~~ Борьба с LLM

- Я понял, что:
  - **LLM изо всех сил пытается логически обосновать случайные байты, которые сгенерировал фаззер** 🥴

~~Работа~~ Борьба с LLM

- Я понял, что:
  - **LLM изо всех сил пытается логически обосновать случайные байты, которые сгенерировал фаззер** 🥲
- Моя совместная работа с LLM выглядит так:

~~Работа~~ Борьба с LLM

- Я понял, что:
  - **LLM изо всех сил пытается логически обосновать случайные байты, которые сгенерировал фаззер** 😵
- Моя совместная работа с LLM выглядит так:



Написав репродюсер, я понял уязвимость

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);
```



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 */
```

prevents returning virt_wifi0

causes UAF

Как же исправить это состояние гонки?

- Поиск причины этого состояния гонки получился **тяжелым**
- Ошибка возникает на границе **4** подсистем ядра Linux:
  1. `netlink`
  2. `network device management`
  3. `ethtool`
  4. `power management`
- Мне было сложно определить местоположение источника ошибки
- Было неясно, по вине какой подсистемы происходит рассинхронизация
- Просто сообщить про баг мейнтейнерам ядра неинтересно – я хотел **его исправить**
- Хорошенько подумав, я разработал исправление для `ethtool`

Мой первый исправляющий патч

```
--- 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)
```



Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу security@kernel.org

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу security@kernel.org
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу security@kernel.org
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML
3. **“Ну что же, давайте бахнем full disclosure,”** – подумал я и отправил

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу `security@kernel.org`
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML
3. **“Ну что же, давайте бахнем full disclosure,”** – подумал я и отправил
4. В публичном LKML получилась живая дискуссия:
   lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу `security@kernel.org`
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML
3. **“Ну что же, давайте бахнем full disclosure,”** – подумал я и отправил
4. В публичном LKML получилась живая дискуссия:
   lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
5. К моему удивлению, некоторые мейнтейнеры **вели себя грубо**
   - Не пишу имен в слайдах, посмотрите по ссылке сами

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу `security@kernel.org`
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML
3. **“Ну что же, давайте бахнем full disclosure,”** – подумал я и отправил
4. В публичном LKML получилась живая дискуссия:
   lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
5. К моему удивлению, некоторые мейнтейнеры **вели себя грубо**
   - Не пишу имен в слайдах, посмотрите по ссылке сами
6. В итоге мейнтейнеры приняли решение **не исправлять баг в** `ethtool`

Процесс ответственного разглашения

1. Я сообщил об уязвимости и отправил патч по адресу `security@kernel.org`
2. Мейнтейнеры подумали и порекомендовали просто отправить патч в LKML
3. **“Ну что же, давайте бахнем full disclosure,”** – подумал я и отправил
4. В публичном LKML получилась живая дискуссия:
   lore.kernel.org/all/20260322075917.254874-1-alex.popov@linux.com/T/#u
5. К моему удивлению, некоторые мейнтейнеры **вели себя грубо**
   - Не пишу имен в слайдах, посмотрите по ссылке сами
6. В итоге мейнтейнеры приняли решение **не исправлять баг в** `ethtool`
7. Вместо этого меня попросили **сделать патч для** `virt_wifi`

Мой второй исправляющий патч

```
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) {
```



CVE-2026-31695

- Мой второй патч был принят в Linux kernel mainline (commit `789b06f9f39c`)
- Уязвимость существовала в ядре около 5 лет (появилась в commit `d43c65b05b84`)
- Поэтому патч был применен к ряду стабильных веток ядра:
  - `v6.19` (на тот момент stable)
  - `v6.18`
  - `v6.12`
  - `v6.6`
  - `v6.1`
  - `v5.15`
- Я запросил идентификатор CVE для этой уязвимости
- Greg Kroah-Hartman выдал для нее **CVE-2026-31695** 

CVE-2026-31695 с позиции атакующего



А теперь про мой эксплойт для CVE-2026-31695



Повреждение памяти в CVE-2026-31695

- Состояние гонки происходит между:
  1. `RTM_DELLINK` для родительского устройства `veth1`
  2. `ETHTOOL_MSG_FEATURES_SET` для дочернего устройства `virt_wifi0`
- Чтобы увидеть список всех UAF, я включил `kasan_multi_shot`
- `KASAN` показал **91 UAF-доступ** в `pm_runtime_get_sync()` 😮
- Ядро использует `struct device` после освобождения:

```
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)
```

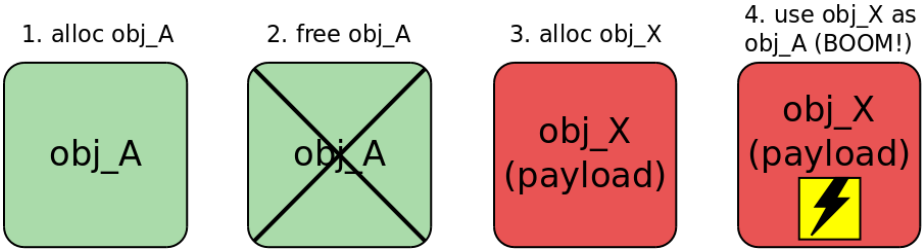
Какая перспектива по эксплуатации CVE-2026-31695?

- Среди всех **91 UAF НЕТ** разыменования указателей из `struct device`
- `pm_runtime_get_sync()` читает и пишет только **1, 2 или 4 байта** в `struct device`
- 👎 Без разыменования указателей из освобожденной памяти не сделать AARW
  - Значит эксплойт будет не такой простой
- 👍 Но это состояние гонки не приводит к отказу ядра
  - Это дает мне шанс сделать 100% стабильный эксплойт



Базовая идея эксплуатации UAF

- Я планировал использовать **kernel heap spraying + type confusion**



- Т.е. заставить ядро перезаписать освобожденную `struct device` контролируемыми данными до вызова `pm_runtime_get_sync()`

Основная идея моего эксплойта для CVE-2026-31695



Что мне нужно для эксплуатации CVE-2026-31695

1. Научиться **стабильно** вызывать состояние гонки между `RTM_DELLINK` и `ETHTOOL_MSG_FEATURES_SET`



Что мне нужно для эксплуатации CVE-2026-31695

1. Научиться **стабильно** вызывать состояние гонки между `RTM_DELLINK` и `ETHTOOL_MSG_FEATURES_SET`
2. Придумать **kernel heap spraying** для перезаписи освобожденной `struct device` нужными данными



Что мне нужно для эксплуатации CVE-2026-31695

1. Научиться **стабильно** вызывать состояние гонки между `RTM_DELLINK` и `ETHTOOL_MSG_FEATURES_SET`
2. Придумать **kernel heap spraying** для перезаписи освобожденной `struct device` нужными данными
3. Добиться перезаписи `struct device` **ДО** вызова `pm_runtime_get_sync()`



Что мне нужно для эксплуатации CVE-2026-31695

1. Научиться **стабильно** вызывать состояние гонки между `RTM_DELLINK` и `ETHTOOL_MSG_FEATURES_SET`
2. Придумать **kernel heap spraying** для перезаписи освобожденной `struct device` нужными данными
3. Добиться перезаписи `struct device` **ДО** вызова `pm_runtime_get_sync()`
4. Вынудить `pm_runtime_get_sync()` сделать **что-то опасное** с моей `struct device`



И тут обнаружились препятствия...

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);
```



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 */
```

prevents returning virt_wifi0

causes UAF

И тут обнаружились препятствия!

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 #3

Obstacle #1

prevents returning virt_wifi0

causes UAF

CPU 1

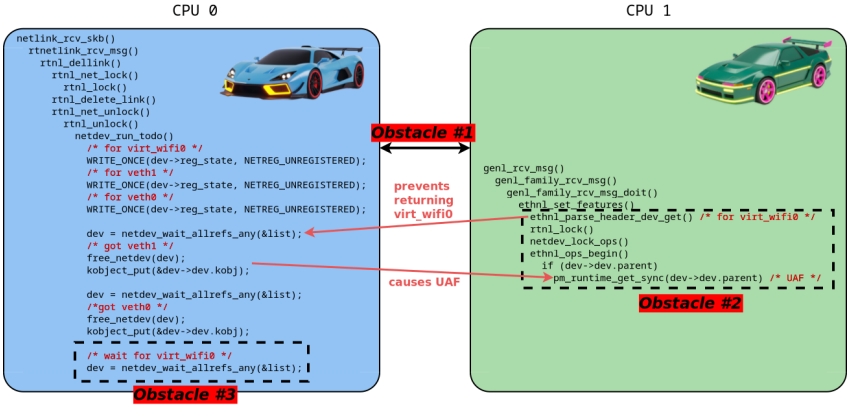


```
genl_rcv_msg()
  genl_family_rcv_msg()
    genl_family_rcv_msg_doit()
      ethnl_set_features()
        ethnl_parse_header_dev_get() /* for virt_wifi0 */
        rtnl_lock()
        netdev_lock_ops()
        ethnl_ops_begin()
          if (dev->dev.parent)
            pm_runtime_get_sync(dev->dev.parent)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

Obstacle #2

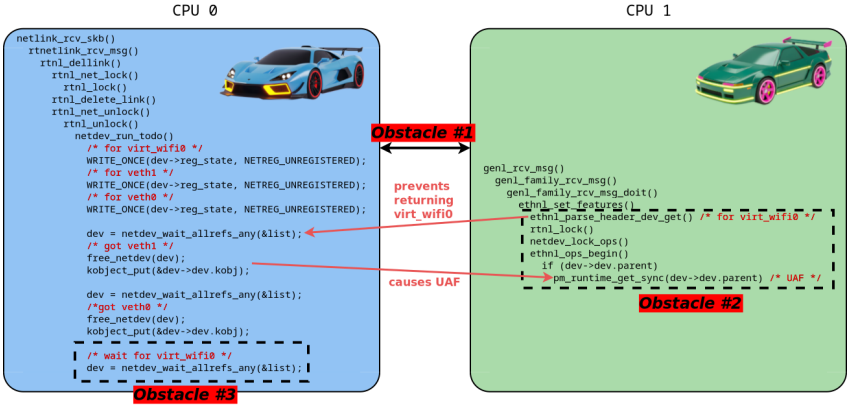
Гонки с препятствиями

1. После выключения KASAN и удаления printk() **гонка перестала воспроизводиться**



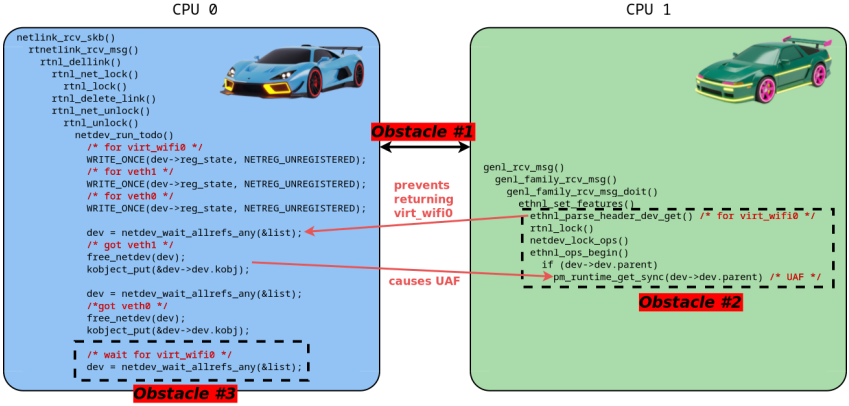
Гонки с препятствиями

1. После выключения KASAN и удаления printk() **гонка перестала воспроизводиться**
2. **Слишком мало времени** на heap spraying и реаллокацию struct device
   (тем более на кросс-кэш атаку, если включены средства защиты слэб-аллокатора)



Гонки с препятствиями

1. После выключения KASAN и удаления printk() **гонка перестала воспроизводиться**
2. **Слишком мало времени** на heap spraying и реаллокацию struct device
   (тем более на кросс-кэш атаку, если включены средства защиты слэб-аллокатора)
3. UAF в потоке №1 происходит до окончания обработки RTM_DELLINK
   (невозможно выполнять **heap spraying в потоке №0**)



Препятствие №1: диагностика

- Без `KASAN` и отладочного вывода временное окно гонки уменьшилось
- Я придумал забавный патч, позволяющий определить успех гонки
  - Записываю `dev->reg_state = 7` в функции `netdev_release()` в потоке №0
  - Проверяю `BUG_ON((to_net_dev(dev->dev.parent))->reg_state == 7)` в потоке №1
- **Хорошие новости:** патч почти не меняет временную диаграмму работы ядра Linux
- **Плохие новости:** состояние гонки никак не воспроизводится на двух потоках 😓
- `BUG_ON()` стреляет только при добавлении `mdelay(10)` в `ethnl_set_features()`

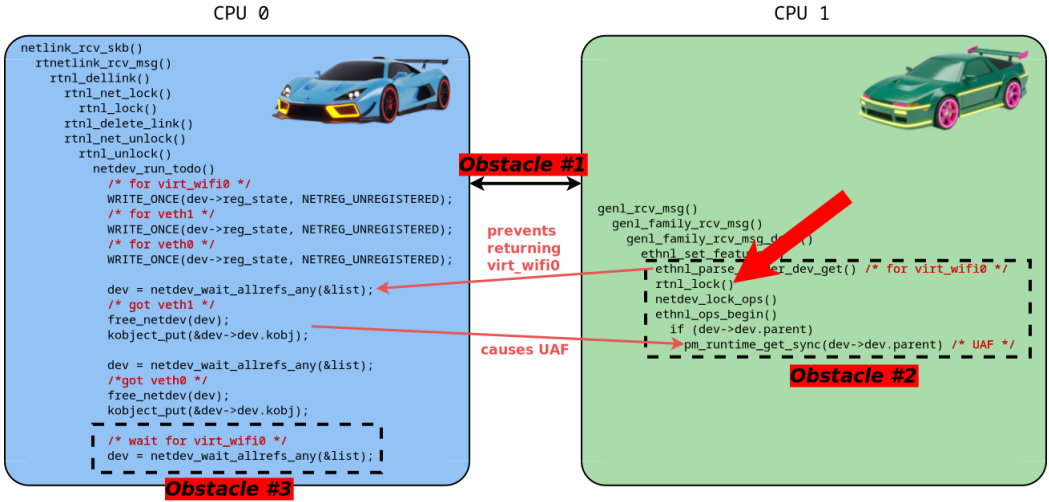
Препятствие №1: борьба за стабильность

- Я попробовал замедлить `ethnl_set_features()` (имитировать `mdelay(10)`)
- Использовал трюк **Янна Хорна** из его исследования Racing against the clock
- Ранее это отлично сработало в моей работе

  Kernel-hack-drill и новый эксплойт для CVE-2024-50264 в ядре Linux
- **В этот раз – бесполезно**: `BUG_ON()` все равно не стреляет 😓
- Дело в нестабильной временной диаграмме потока №0 (`RTM_DELLINK`)

Но тут я снова посмотрел на код

- Я могу добиться эффекта `mdelay()`, если **заставлю поток №1 ждать** в `rtnl_lock()`**!**

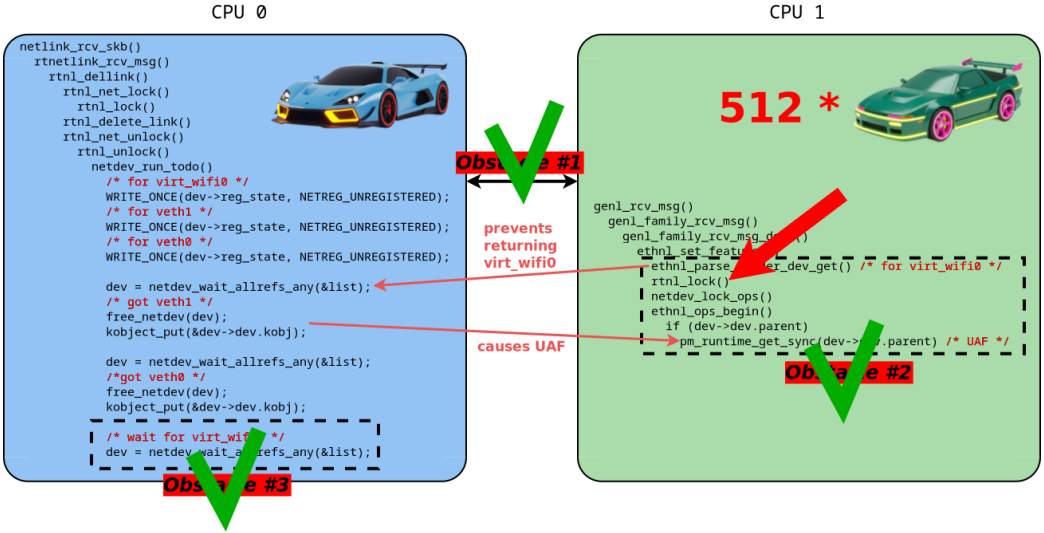


Эврика: я создал 512 потоков №1, конкурирующих в rtnl_lock()

Эврика: я создал 512 потоков №1, конкурирующих в rtnl_lock()



100% успешная гонка с препятствиями



Ну давай же, сделай что-то опасное!

- Теперь мне нужно было заставить ядерную функцию `pm_runtime_get_sync()` **сделать что-то опасное** из-за моей `struct device`
- Я прошагал отладчиком всю функцию (**91 UAF-доступ**)
- В глубине нашел примитив для перехвата потока управления




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

- Мне нужно было создать поддельную `struct device`, чтобы добраться туда

Защита слэб-аллокатора

В **Fedora Server** включены средства защиты ядерного слэб-аллокатора:

1. `CONFIG_RANDOM_KMALLOC_CACHES`
   - Создает несколько копий слэб-кэшей
   - При вызове `kmalloc()` ядро выбирает одну из них в зависимости от адреса вызова
   - Это противодействует технике **heap spraying** (для обхода нужна **кросс-кэш атака**)

2. `CONFIG_SLAB_BUCKETS`
   - Создает отдельные слэб-кэши для некоторых объектов с userspace-данными
   - Например слэб-кэши для объекта `msg_msg`

     (я изобрел heap spraying этим объектом при исследовании
     Сила четырех байтов: эксплуатация уязвимости CVE-2021-26708 в ядре Linux)

Стоп, а где защита?

- `struct net_device` живет в слэб-кэше `kmalloc-cg-4k`
- Я уже приготовился выполнять кросс-кэш атаку, но...
- Представьте **мое удивление**, когда я увидел вот это:

```
$ 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` не работает для кэша с моим объектом 😮

Дыра в CONFIG_RANDOM_KMALLOC_CACHES

- Я обнаружил, что могу атаковать UAF в ядре просто **наивным heap spraying**
- Я бы мог воспользоваться моим любимым `msg_msg`...
- Но `CONFIG_SLAB_BUCKETS` переселил его в отдельные слэб-кэши `msg_msg-*`
- Я взял широко известную альтернативу: данные сетевого пакета `sk_buff.data`
- Linux создает этот объект в функции `__alloc_skb()`
- Для этого нужно просто выполнить `send()` в `AF_UNIX`-сокет
- Если я отправляю сетевой пакет размером **3737** байтов,

  то ядро выделяет его в `kmalloc-cg-4k` – **это успех** 😉

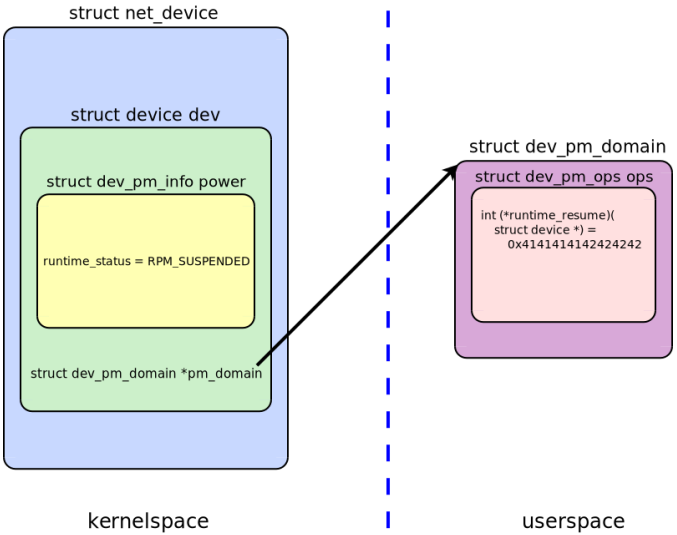
Дыра в CONFIG_RANDOM_KMALLOC_CACHES



Подделка struct device

- Для эксперимента я выключил **SMEP** и **SMAP**
- Я выяснил, что userspace-данные находятся в `sk_buff.data` без отступа (**удобно!**)
- Выполнил heap spraying **в потоке №2** и заполнил `struct device` нулями
- Шаг за шагом стал доводить содержимое `struct device`, чтобы добраться до **перехвата потока управления** в `rpm_resume()`

struct device для базового перехвата потока управления



Есть перехват потока управления в ядре Fedora Server



```
[!] 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
...
```



Внезапная проблема

- Я заменил `0x4141414142424242` на базовую **ret2usr-атаку**
- Повысил привилегии до root с помощью

  `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`
- После окончания `runtime_resume()` ядро Linux продолжило выполнение и...
- И затем ядро упало далее в функции `rpm_resume()` 

```
[   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
```

Проснись, Нео

- `rpm_resume()` пытается разбудить потоки в очереди `device.power.wait_queue`
- Это происходит после моего перехвата потока управления
- Я стал думать, как тут выкрутиться, придумал три варианта:

Проснись, Нео

- `rpm_resume()` пытается разбудить потоки в очереди `device.power.wait_queue`
- Это происходит после моего перехвата потока управления
- Я стал думать, как тут выкрутиться, придумал три варианта:

  1. Перезаписать `modprobe_path` и усыпить поток навсегда 
     - ★ Это некрасивое решение
     - ★ Оно к тому же ломает подсистему `netlink` (захвачен `rtnl_mutex` и др.)

Проснись, Нео

- `rpm_resume()` пытается разбудить потоки в очереди `device.power.wait_queue`
- Это происходит после моего перехвата потока управления
- Я стал думать, как тут выкрутиться, придумал три варианта:
  1. Перезаписать `modprobe_path` и усыпить поток навсегда 
     - ★ Это некрасивое решение
     - ★ Оно к тому же ломает подсистему `netlink` (захвачен `rtnl_mutex` и др.)
  2. Жестко выпрыгнуть в userspace после перехвата потока управления 
     - ★ Тоже некрасиво, особенно при включенном **PTI (Page Table Isolation)**

Проснись, Нео

- `rpm_resume()` пытается разбудить потоки в очереди `device.power.wait_queue`
- Это происходит после моего перехвата потока управления
- Я стал думать, как тут выкрутиться, придумал три варианта:

1. Перезаписать `modprobe_path` и усыпить поток навсегда 
   - ★ Это некрасивое решение
   - ★ Оно к тому же ломает подсистему `netlink` (захвачен `rtnl_mutex` и др.)
2. Жестко выпрыгнуть в userspace после перехвата потока управления 
   - ★ Тоже некрасиво, особенно при включенном **PTI (Page Table Isolation)**
3. Как-то модифицировать `struct device`, 
   чтобы ядро пережило `wake_up_all(&dev->power.wait_queue)`

Вот он, мой шанс

```
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);

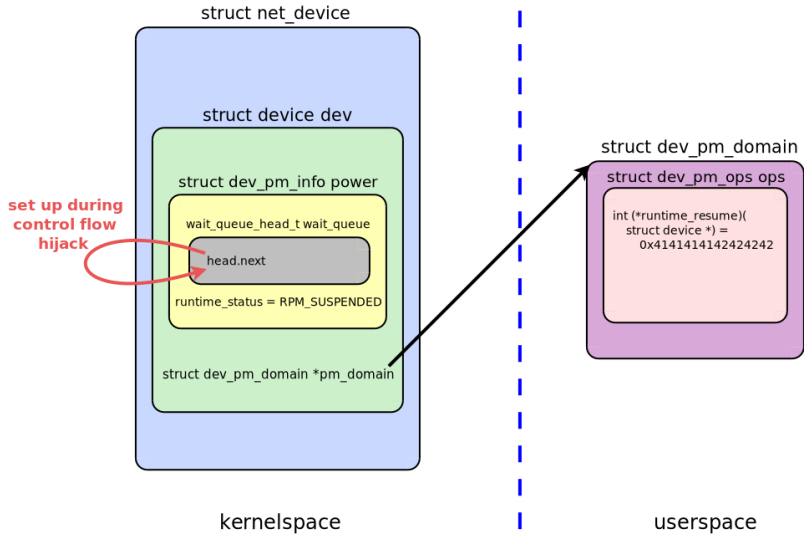
    curr = list_first_entry(&wq_head->head, wait_queue_entry_t, entry);

    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) {
```

- Мне нужно подделать пустой связный список `device.power.wait_queue`
- Для этого `device.power.wait_queue.head` должен указывать сам на себя
- **Но я же не могу знать заранее этот ядреный адрес!** 😓

Эврика: я могу вычислить адрес при перехвате потока управления



Битва с modprobe_path в Fedora Server

- Далее я заменил `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`

  на перезапись `modprobe_path`
- Это нужно для побега из контейнера (эксплойт работает в namespace)
- Попытался вызвать LPE-скрипт в `modprobe_path` с помощью `AF_ALG`-сокета

Битва с modprobe_path в Fedora Server

- Далее я заменил `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`

  на перезапись `modprobe_path`
- Это нужно для побега из контейнера (эксплойт работает в namespace)
- Попытался вызвать LPE-скрипт в `modprobe_path` с помощью `AF_ALG`-сокета
  - В Debian это работает, а в Fedora Server – **НЕТ**

Битва с modprobe_path в Fedora Server

- Далее я заменил `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`

  на перезапись `modprobe_path`
- Это нужно для побега из контейнера (эксплойт работает в namespace)
- Попытался вызвать LPE-скрипт в `modprobe_path` с помощью `AF_ALG`-сокета
  - В Debian это работает, а в Fedora Server – **НЕТ**
- Попытался аналогично переписать `/proc/sys/kernel/core_pattern`

Битва с modprobe_path в Fedora Server

- Далее я заменил `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`

  на перезапись `modprobe_path`
- Это нужно для побега из контейнера (эксплойт работает в namespace)
- Попытался вызвать LPE-скрипт в `modprobe_path` с помощью `AF_ALG`-сокета
  - В Debian это работает, а в Fedora Server – **НЕТ**
- Попытался аналогично переписать `/proc/sys/kernel/core_pattern`
  - В Debian это работает, а в Fedora Server – **НЕТ**

Битва с modprobe_path в Fedora Server

- Далее я заменил `commit_creds(prepare_kernel_cred(INIT_TASK_PTR))`

  на перезапись `modprobe_path`
- Это нужно для побега из контейнера (эксплойт работает в namespace)
- Попытался вызвать LPE-скрипт в `modprobe_path` с помощью `AF_ALG`-сокета
  - В Debian это работает, а в Fedora Server – **НЕТ**
- Попытался аналогично переписать `/proc/sys/kernel/core_pattern`
  - В Debian это работает, а в Fedora Server – **НЕТ**
- Мучился, сидел с отладчиком ядра, и тут заметил в логах 😮

  `SELinux is preventing kworker/u8:3 from read access on the lnk_file 3`

Эврика: при перехвате потока управления вырублю SELinux



Обход KASLR

- Для обхода **KASLR** я взял популярную **Prefetch Side-Channel Attack**
- Участники **kernelCTF** без конца копируют ее друг у друга
- А я решил разобраться в ней детально
- Изначальная идея – в статье команды из **Graz University of Technology**
- **Суть:** время работы непривилегированной инструкции `prefetch` зависит от трансляции виртуальных адресов
- Приложения могут наблюдать **разную длительность** `prefetch` для адресов ядра
- На **Intel CPU** `prefetch` для отображенных ядерных адресов работает **быстрее**, чем для неотображенных

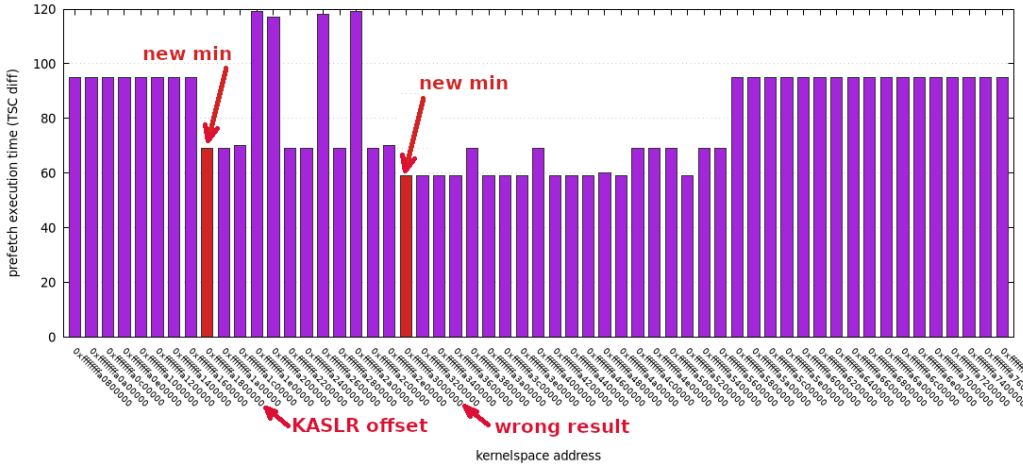
Реализация prefetch-атаки

- В решениях **kernelCTF** распространена такая реализация `prefetch`-атаки:
  1. **Замерить длительность** инструкции `prefetch` для последовательности адресов ядра, начиная с `0xffffffff81000000`
  2. **Сохранить адрес**, для которого CPU выполнил `prefetch` за минимальное время
  3. Повторить это несколько раз и **выбрать итоговый адрес** с помощью алгоритма большинства голосов Бойера — Мура
- Пример:
  https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-41009_lts_cos/exploit/lts-6.6.32/exploit.c

Ошибки реализации prefetch-атаки из kernelCTF

- Участники **kernelCTF** помимо кода `prefetch`-атаки копируют друг у друга ошибки
- Их реализация алгоритма Бойера — Мура **полностью сломана**
  - Она может сработать только в случае везения
  - Это исправляется одной строчкой
  - Подробности будут в подробной статье по этому исследованию
- `prefetch`-атаки из **kernelCTF** **работают нестабильно** на `Intel Comet Lake CPU` (10 поколение `Intel Core i7`)
  - Выдается **неверный адрес** начала кода ядра
  - Я стал выяснять причины

Почему prefetch-атака из kernelCTF нестабильна



Доработка prefetch-атаки из kernelCTF

```
    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];
```

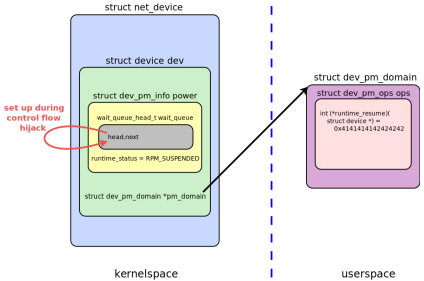
- А также я начинаю поиск раньше, с адреса `0xffffffff80000000`

  вместо `0xffffffff81000000` (на случай, если KASLR отключен)

- `prefetch`-атака становится 100% стабильной на `Intel Comet Lake CPU` – **я получил отступ KASLR**

Идеи по обходу SMEP и SMAP

- Зная отступ **KASLR**, я могу создать ROP-цепочку для обхода **SMEP**
- С обходом **SMAP** все сложнее
- Мне нужно было расположить ROP-цепочку в kernelspace по известному адресу
- Это возможно: `rpm_callback()` имеет адрес `struct device` как 1-й аргумент
- => Я могу положить ROP-цепочку в данные для heap spraying, и ее адрес взять из `RDI`
- Но мне еще нужно было расположить поддельный `dev_pm_domain` в kernelspace по известному адресу



iptables – полезная штука

- Для размещения `dev_pm_domain` я выбрал известную технику с `kernfs_pr_cont_buf`
- Она была опубликована у исследователя **Pumpkin**
- `kernfs_pr_cont_buf` – это глобальная переменная в `fs/kernfs/dir.c`
- Записать ее просто: `iptables -A OUTPUT -m cgroup --path "a5a5" -j LOG`
- После этого:

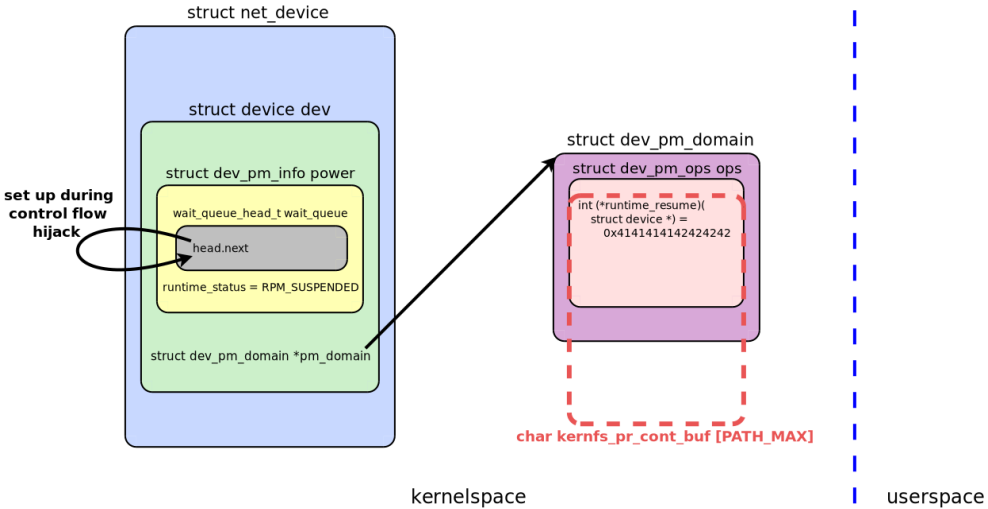
```
gef> p kernfs_pr_cont_buf
$3 = "a5a5", '\000' <repeats 4091 times>
```

- Но на Fedora Server **не загружаются нужные модули** ядра

  при выполнении этого действия в user namespace

- **Я выстрадал решение:** использовать `xtables-nft-multi` вместо `iptables`

Обход SMEP и SMAP

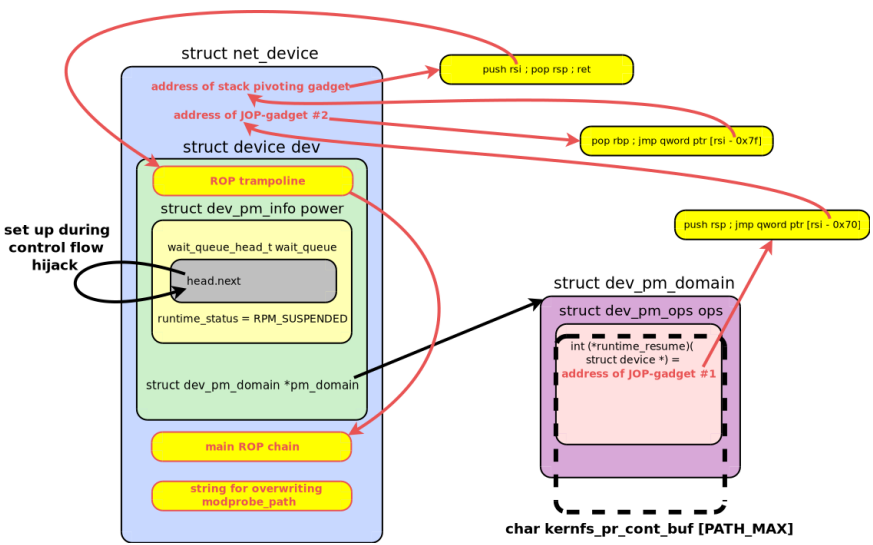


Финал: моя ROP-цепочка



- Разработка эксплойта для **CVE-2026-31695**
  завершалась созданием ROP-цепочки
- Это было **особое удовольствие** (честно, без шуток)
- Моя ROP-цепочка должна была:
  1. Вырубить **SELinux**
  2. Перезаписать `modprobe_path` на путь LPE-скрипта
  3. Создать пустой связный список в `device.power.wait_queue.head`
  4. Позволить ядру продолжить обработку системного вызова
- Последний пункт **сложный**: адрес из стека был в только в одном `RSP`
- При переключении стека (stack pivoting) **этот адрес был бы утрачен**

Я сконструировал ROP/JOP-цепочку



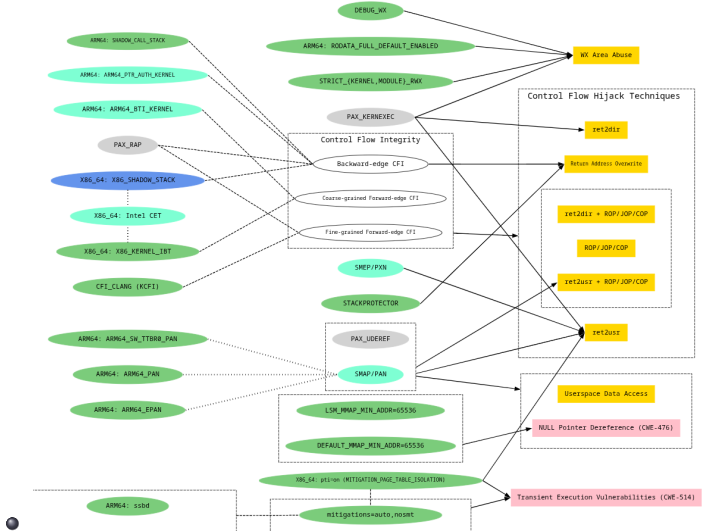
Live Demo!



Средства противодействия эксплуатации CVE-2026-31695

1. **Запрет автозагрузки модулей** ядра Linux
   - Подробности в моем проекте **kernel-hardening-checker**
   - Заодно это повлияет на технику перезаписи `modprobe_path`
2. **Очистка памяти после освобождения** (`init_on_free`)
   - Заполняет нулями `struct device` после `kfree()`
   - Вызывает несколько **WARNING** в ядре, но мой эксплойт все равно дает root!
   - Мой эксплойт выигрывает гонку с `init_on_free`
3. **Page Table Isolation (PTI)** вынудит доработать атаку на KASLR
4. **Улучшение средств защиты слэб-аллокатора** (я готовлю патч в upstream)
5. **Control Flow Integrity (CFI)** – см. мой проект **linux-kernel-defence-map**

Обзор CFI в linux-kernel-defence-map



В заключение

- Я обнаружил, исправил и проэксплуатировал состояние гонки **CVE-2026-31695**
- Мне удалось обойти средства защиты и захватить **Fedora 44 Server**
- Эксплойт получился **очень стабильным** (круто!)
- Использовал ли я **LLM** и **AI-агентов**? Только для реверса `netlink` (хаха)
- Почему? Потому что это исследование было **удовольствием для меня**



- Мой отдел исследует LLM и AI-агентов о области безопасности Linux (но это другая история)
- Я хочу, чтобы **роботы сортировали мусор**,

  а **я занимался творчеством** (а не наоборот)

Спасибо!

Успешной конференции!

Контакты:

a13xp0p0v

alex.popov@linux.com

Блог: a13xp0p0v.tech

Канал: t.me/linkersec

