CVE-2021-3609: CAN BCM local privilege escalation

This article is about a recent vulnerability in the Linux kernel labeled *CVE-2021-3609*. The issue was initially reported by *syzbot*. The vulnerable part of the kernel was the CAN BCM networking protocol in the CAN networking subsystem ranging from kernel version 2.6.25 to 5.13-rc6. In the following, I am going to cover the vulnerability and my exploitation approach for kernel version >= 5.4 which led to successful local privilege escalation to root.

Vulnerability

The vulnerability is a race condition which lets us free `struct bcm_op` and `struct bcm_sock` in `bcm_release()` while still being used in `bcm_rx_handler()`.

`struct bcm_op` is a structure which can be allocated by sending a message on a CAN BCM socket with the opcode `RX_SETUP`. It is used to setup either transmission or reception of CAN messages. In this particular case, we allocate an operation in `bcm_rx_setup()` to receive messages.

```
static int bcm_rx_setup(struct bcm_msg_head *msg_head, struct msghdr *msg,
                        int ifindex, struct sock *sk)
{
        ...

        /* check the given can_id */
        op = bcm_find_op(&bo->rx_ops, msg_head, ifindex);
        if (op) {
                /* update existing BCM operation */

                /* update struct members of op */

                /* Only an update -> do not call can_rx_register() */
                do_rx_register = 0;

        } else {
                /* insert new BCM operation for the given can_id */
                op = kzalloc(OPSIZ, GFP_KERNEL);

                /* initialization of op */

                do_rx_register = 1;                                              [1]

        }

        ...

        /* now we can register for can_ids, if we added a new bcm_op */
        if (do_rx_register) {
                if (ifindex) {
                        struct net_device *dev;

                        dev = dev_get_by_index(sock_net(sk), ifindex);
                        if (dev) {
                                err = can_rx_register(sock_net(sk), dev,        [2]
                                                      op->can_id,
                                                      REGMASK(op->can_id),
                                                      bcm_rx_handler, op,
                                                      "bcm", sk);

                                op->rx_reg_dev = dev;
                                dev_put(dev);
                        }
        ...
}
```

The excerpt above makes it clear that we have to specifically allocate a new `struct bcm_op` `[1]` in order to register a new CAN receiver. At `[2]`, we register such for our user-controlled network interface specified with `ifindex`. Notice that `bcm_rx_handler` is passed as an argument which means that this function will be called on message receival.

Now we have to send a CAN message from another CAN BCM socket which will be broadcasted to all sockets on this network interface. In total, we have **one socket for reception** (this is the one we are going to exploit) and **another one for transmission**. Because we registered the first socket with `RX_SETUP`, we can receive the incoming message. Interestingly enough, `TX_SETUP` for our sending socket is not required as we already specify the network interface in `connect()`.

At this point, we have a message incoming so `bcm_rx_handler()` is called. At the same time, we close the socket and `bcm_release()` is run in parallel to our receive handler.

```
static int bcm_release(struct socket *sock)
{
        ...

        /* remove bcm_ops, timer, rx_unregister(), etc. */

        unregister_netdevice_notifier(&bo->notifier);

        lock_sock(sk);                                                          [1]

        list_for_each_entry_safe(op, next, &bo->tx_ops, list)
                bcm_remove_op(op);

        list_for_each_entry_safe(op, next, &bo->rx_ops, list) {
                /*
                 * Don't care if we're bound or not (due to netdev problems)
                 * can_rx_unregister() is always a save thing to do here.
                 */
                if (op->ifindex) {
                        /*
                         * Only remove subscriptions that had not
                         * been removed due to NETDEV_UNREGISTER
                         * in bcm_notifier()
                         */
                        if (op->rx_reg_dev) {
                                struct net_device *dev;

                                dev = dev_get_by_index(net, op->ifindex);
                                if (dev) {
                                        bcm_rx_unreg(dev, op);
                                        dev_put(dev);
                                }
                        }
                }
        ...

                bcm_remove_op(op);                                              [2]
        }

        ...

        sock_orphan(sk);
        sock->sk = NULL;

        release_sock(sk);
        sock_put(sk);                                                           [3]

        return 0;
}
```

In `bcm_release()`, we take the socket lock `[1]`. One might ask themselves, *why do we have a race condition if we take a lock before accessing the socket?* It's because there is no similar locking in `bcm_rx_handler()` which would effectively hang `bcm_release()` to wait for `bcm_rx_handler()` to finish its work. Although, the patch for this bug does not take a lock `in bcm_rx_handler()`. Instead, we are under a so-called RCU read lock which is invoked in CAN receiver code before `bcm_rx_handler()`. For this reason, the patch adds a call to `synchronize_rcu()` right before `[2]` in order to wait for all RCU dependent operations to finish before completely closing the socket. I won't go into detail about how RCU works, but I'm leaving you a link at the bottom of this article.

Because there was no synchronizing feature prior the patch, we simply free `struct bcm_op` at `[2]` and decrease the refcount of the socket. Finally, `struct bcm_sock` will also be freed because refcount will reach 0.

Exploitation

So now we are still in `bcm_rx_handler()`, *but how do we want to exploit this?* After many trials, I've found it particularly hard to exploit any of the use-after-free's within `bcm_rx_handler()`. This is due to `bcm_rx_handler()` executing fast which means that it's tricky to overwrite `struct bcm_op` with heap spraying. In contrast to my previous *CAN ISOTP exploit*, it looks to me that there is no good opportunity to halt execution within `bcm_rx_handler()` and make it more reliable. Instead, I focus on another approach which I will explain in the following.

This particular code in `bcm_rx_setup()` turned out to be useful:

```
if (op->flags & SETTIMER) {

        /* set timer value */
        op->ival1 = msg_head->ival1;
        op->ival2 = msg_head->ival2;
        op->kt_ival1 = bcm_timeval_to_ktime(msg_head->ival1);
        op->kt_ival2 = bcm_timeval_to_ktime(msg_head->ival2);
        ...
}
```

When we allocate a new `struct bcm_op`, we can specify the flag `SETTIMER` and setup a timer. If the timer is started, `bcm_rx_timeout_handler()` will be called once the user-controlled time value `op->kt_ival1` has passed.

At the end of `bcm_rx_handler()`, we have a call to `bcm_rx_starttimer()` which will start this timer.

```
/*
 * bcm_rx_starttimer - enable timeout monitoring for CAN frame reception
 */
static void bcm_rx_starttimer(struct bcm_op *op)
```

```
{
        if (op->flags & RX_NO_AUTOTIMER)
                return;

        if (op->kt_ival1)
                hrtimer_start(&op->timer, op->kt_ival1, HRTIMER_MODE_REL_SOFT);
}
```

If we set a timer in `bcm_rx_setup()` , it will be started and run for `op->kt_ival1` which is controlled by the user. In my case, I have set the timer to expire after one second, so `bcm_rx_timeout_handler()` will be called one second after `hrtimer_start()` in `bcm_rx_starttimer()` . This allows me to have a sufficient time frame of one second in which I can perform a reliable heap spray.

For the heap spray, I use the already known technique with `setxattr()` and `userfaultfd()` which was described well by *Vitaly Nikolenko*. You can find a link to his article at the bottom.

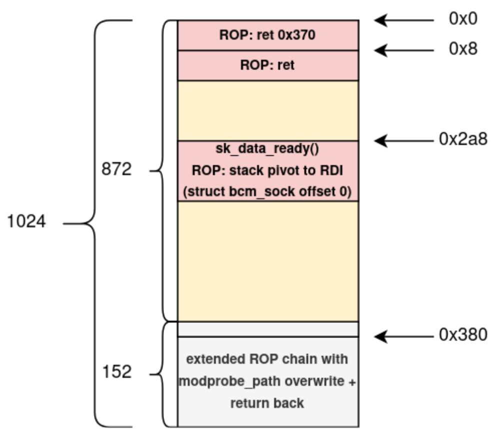
I didn't want to heap spray `struct bcm_op` because it is heavily used in `bcm_rx_handler()` where a reliable heap spray is hard. Instead, I hope that during the time span of running `bcm_rx_handler()` the freed `struct bcm_op` won't be overwritten until I start the timer in `bcm_rx_starttimer()` . This approach sort of works because `bcm_rx_handler()` runs fast so there is not much time in which the freed `struct bcm_op` could be overwritten.

Back to `bcm_rx_timeout_handler()` , `struct bcm_sock` has a few function pointers which I could overwrite with my heap spray. I decided to use the `sk_data_ready()` pointer which is called in the following call path:

```
bcm_rx_timeout_handler() -> bcm_send_to_user() -> sock_queue_rcv_skb() -> __sock_queue_rcv_skb()
-> sk->sk_data_ready(sk)
```

At this point, the `sk->sk_data_ready(sk)` pointer will be called and we end up with arbitrary kernel execution. Because the function is called with the parameter `sk (struct sock *)` , the address of our heap sprayed socket will be stored in the `RDI` register. This allows me to perform a stack pivot to the beginning of the socket structure and start executing ROP gadgets.

`struct bcm_sock` is 872 bytes big on my system which means that it is allocated in the generic `kmalloc-1024` SLAB cache. Because `struct bcm_sock` does not fill all 1024 bytes, I have 152 unused bytes (1024 - 872) which I can use to construct a ROP chain.



The extended ROP chain will overwrite a kernel address where `modprobe_path` is stored. I've already used this technique in my *CAN ISOTP exploit* (article available on my github) and it's explained well by *lkmidas* in his article. Check it out in the link at the bottom.

One problem I've stumbled upon during exploitation was that I couldn't jump to `do_task_dead()` to halt my hijacked kernel thread. Shortly after, I noticed what the issue was: `bcm_rx_timeout_handler()` is **executed by task swapper with PID 0**. I obviously can't kill task with PID 0, so I had to figure out another way to fixate the system after executing the ROP chain. Looking at the kernel panic logs which reveal registers, I noticed that the register `RBP` stored an address similar to `RSP` . Notice that I had to change `RSP` by performing a stack pivot to abandon the actual kernel stack for my own malicious one. The `RBP` register wasn't touched during execution of the ROP gadgets, so I could use it to move back to the old kernel stack. Even if `RBP` would change during the ROP execution, I could save the contents of `RBP` to another register and restore the kernel stack from this register instead.

So after executing ROP gadgets, I can basically reverse the stack pivot by moving `RBP` into `RSP` , then I pop one element off the stack and return back to `__sock_queue_rcv_skb()` . I also set `RAX` to `0` for a clean return without errors.

```
*rop++ = 0xffffffff81087bc3 + kaslr_offset; /* xor rax, rax ; ret */ /* return value */
*rop++ = 0xffffffff81087b0c + kaslr_offset; /* mov rsp, rbp ; pop rbp ; ret */
```

Finally, all is left is to execute `/tmp/dummy` which in turn runs `/tmp/x` with root privleges and the unprivileged user is added to `/etc/sudoers` without password. Local privilege escalation is done.

Getting the KASLR offset

In case we run on a system with KASLR enabled, we need to know the KASLR offset in order to return to valid kernel addresses in the ROP chain. On *Ubuntu 20.04.02 LTS*, I was able to retrieve a kernel text address from a warning in dmesg. If the target machine is 32-bit and KASLR is enabled, you could try *CVE-2021-34693* which is an infoleak of 4 bytes in `struct bcm_msg_head` . You can find a link to the PoC at the bottom.

Combining everything together

At this place, I covered all the steps which now have to be combined. The following sequence is used in my exploit:

- retrieve kernel text address for KASLR offset in dmesg
    - on 32-bit systems CVE-2021-34693 can be used

- setup user namespace
- setup vcan network interface
- open two CAN BCM sockets and connect each to the interface
- call `sendmsg()` on socket 1 with `RX_SETUP`, flag `SETTIMER` and time interval of one second to allocate `struct bcm_op`
- call `sendmsg()` on socket 2 to send message to socket 1

At the same time:

- `bcm_rx_handler()` is run on socket 1 in a softirq
    - `bcm_rx_starttimer()` starts the timer
- close socket 1 -> `bcm_release()` -> free `struct bcm_op` and `struct bcm_sock`
- heap spray `struct bcm_sock` with the malicious buffer
- `bcm_rx_timeout_handler()` is run after one second due to `bcm_rx_starttimer()`
- overwritten `sk->sk_data_ready(sk)` is called and we jump to the beginning of `struct bcm_sock`
- within `struct bcm_sock`, move to the end of the structure and start executing the extended ROP chain
- overwrite `modprobe_path` and return back to `__queue_sock_rcv_skb()`
- run `/tmp/dummy` so `/tmp/x` will be run by root -> unprivileged user is added to `/etc/sudoers` without password

Notice

Investigating into a syzbot report to find its root cause and prove exploitability was a great opportunity which taught me a couple of useful tricks. If you have any questions, send me an e-mail (nslusarek@gmx.net).

Also, I'm currently looking for job and internship opportunities in infosec in Germany/Europe. In case you are interested, please reach out to me via e-mail.

References

https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

https://duasynt.com/blog/linux-kernel-heap-spray

https://lkmidas.github.io/posts/20210223-linux-kernel-pwn-modprobe/

https://github.com/nrb547/kernel-exploitation/tree/main/cve-2021-34693