```
Date: Tue, 8 Jun 2021 08:53:42 +0800
From: Lin Horse <kylin.formalin@...il.com>
To: oss-security@...ts.openwall.com
Subject: CVE-2021-3573: UAF in hci_sock_bound_ioctl() function

Hello there,

Our team (BlockSec) found an UAF vulnerability in function
hci_sock_bound_ioctl(). It can allow attackers to corrupt kernel heaps
(kmalloc-8k to be specific) and adopt further exploitations.


=*=*=*=*=*=*=*=*=  BUG DETAILS  =*=*=*=*=*=*=*=*=

>>>>>>>> background knowledge <<<<<<<<

The hci_sock_bound_ioctl() function is in charge of five HCI commands.

/* Ioctls that require bound socket */
static int hci_sock_bound_ioctl(struct sock *sk, unsigned int cmd,
                unsigned long arg)
{
    struct hci_dev *hdev = hci_pi(sk)->hdev; // { 1 }

    if (!hdev)
        return -EBADFD;

    /* ..... */

    switch (cmd) {
    case HCISETRAW:
        ...

    case HCIGETCONNINFO:
        ...

    case HCIGETAUTHINFO:
        ...

    case HCIBLOCKADDR:
        ...

    case HCIUNBLOCKADDR:
        ...
    }

    return -ENOIOCTLCMD;
}

As you can see, the biggest difference between functions
hci_sock_bound_ioctl() and hci_sock_ioctl() is that the former one will
derive the hci_dev struct through hci_pi(sk)->hdev. (as code mark { 1 }
shows)

In other words, the bind() syscall needs to be called before the
hci_sock_bound_ioctl() to write this struct. The hdev is obtained through
hci_dev_get(), which based on the counter.

static int hci_sock_bind(struct socket *sock, struct sockaddr *addr,
             int addr_len)
{
    ...
    switch (haddr.hci_channel) {
    case HCI_CHANNEL_RAW:
        ...
            hdev = hci_dev_get(haddr.hci_dev); // { 2 }

        ...
        hci_pi(sk)->hdev = hdev;
        ...
    }
}

>>>>>>>> bug iteself <<<<<<<<

The bug itself is about the UAF of hdev and the root cause is the race
(again).

When the HCI device detaches from the kernel, the function
hci_unregister_dev() will be called. This function will call
hci_sock_dev_event(hdev, HCI_DEV_UNREG) to inform all sockets that this
device is going to be removed. The core logic is presented below.

void hci_sock_dev_event(struct hci_dev *hdev, int event)
{
...
    if (event == HCI_DEV_UNREG) {
        struct sock *sk;

        /* Detach sockets from device */
        read_lock(&hci_sk_list.lock);
        sk_for_each(sk, &hci_sk_list.head) {
            bh_lock_sock_nested(sk);
            if (hci_pi(sk)->hdev == hdev) {
                hci_pi(sk)->hdev = NULL;
                sk->sk_err = EPIPE;
                sk->sk_state = BT_OPEN;
                sk->sk_state_change(sk);

                hci_dev_put(hdev);
            }
            bh_unlock_sock(sk);
        }
        read_unlock(&hci_sk_list.lock);
    }
}

That is, the hci_sock_dev_event() function will release the hdev from the
bounded sockets, all at once.

Therefore, one question arises: Is there any possibility that the
hci_sock_dev_event() in detaching routine take places and release the hdev
while the hci_sock_bound_ioctl() is still working?

Unfortunately, the answer is YES. The hci_sock_dev_event() can release the
hdev and cause the UAF in function hci_sock_bound_ioctl(). This race can be
shown below.

hci_sock_bound_ioctl thread    |    hci_sock_dev_event thread
                               |
                               |
if (!hdev)                     |
    return -EBADFD;            |
                               |
                               |    hci_pi(sk)->hdev = NULL;
                               |    ...
                               |    hci_dev_put(hdev);
// UAF, for example            |
hci_dev_lock(hdev);            |
```

```
                                |
                                |
....
```

It is worth mentioning that the attacker can stably control and trigger this race with userfaultfd primitive, which will be discussed later.

=*=*=*=*=*=*=*=*= BUG EFFECTS =*=*=*=*=*=*=*=*=

There are four different types of functions will be called from the vulnerable hci_sock_bound_ioctl().

* hci_get_conn_info()
* hci_get_auth_info()
* hci_sock_blacklist_add()
* hci_sock_blacklist_del()

All these functions can have different effects when the UAF of hdev happens. For example, the hci_sock_blacklist_add() will allow the attacker to write arbitrary 6 bytes to any place if the released hdev->blacklist can be sprayed.

```
static int hci_sock_blacklist_add(struct hci_dev *hdev, void __user *arg)
{
    bdaddr_t bdaddr;
    int err;

    if (copy_from_user(&bdaddr, arg, sizeof(bdaddr)))
        return -EFAULT;

    hci_dev_lock(hdev);

    err = hci_bdaddr_list_add(&hdev->blacklist, &bdaddr, BDADDR_BREDR);
   // the user controlled bdaddr will be insert to list

    hci_dev_unlock(hdev);

    return err;
}
```

In a nutshell, the UAF of hdev can easily crash the kernel. It can also be the weapon of skillful hackers (with CAP_NET_ADMIN privilege). Below we provide the report from KASan.

```
[   12.663166]
==================================================================
[   12.664161] BUG: KASAN: use-after-free in mutex_lock+0xa9/0x130
[   12.664837] Write of size 8 at addr ffff88800c2ba010 by task exp/125
[   12.665551]
[   12.665731] CPU: 0 PID: 125 Comm: exp Not tainted 5.11.11+ #8
[   12.666378] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS
1.10.2-1ubuntu1 04/01/2014
[   12.667372] Call Trace:
[   12.667661]  dump_stack+0x1b9/0x22e
[   12.668068]  ? show_regs_print_info+0x12/0x12
[   12.668563]  ? log_buf_vmcoreinfo_setup+0x45d/0x45d
[   12.669114]  print_address_description+0x7b/0x3a0
[   12.669646]  __kasan_report+0x14e/0x200
[   12.670084]  ? mutex_lock+0xa9/0x130
[   12.670494]  kasan_report+0x47/0x60
[   12.670894]  check_memory_region+0x2e2/0x330
[   12.671379]  mutex_lock+0xa9/0x130
[   12.671777]  ? mutex_trylock+0xb0/0xb0
[   12.672206]  ? copy_user_generic_string+0x31/0x40
[   12.672742]  hci_get_auth_info+0xbb/0x2b0
[   12.673206]  ? hci_get_conn_info+0x630/0x630
[   12.673696]  ? release_sock+0x155/0x1b0
[   12.674140]  hci_sock_ioctl+0x749/0x900
[   12.674582]  ? hci_sock_getname+0x1d0/0x1d0
[   12.675060]  ? do_vfs_ioctl+0x892/0x1a50
[   12.675514]  ? selinux_file_ioctl+0xd41/0x1200
[   12.676036]  ? __ia32_compat_sys_ioctl+0xc00/0xc00
[   12.676599]  sock_do_ioctl+0xdc/0x310
[   12.677042]  ? sock_show_fdinfo+0xb0/0xb0
[   12.677523]  ? hci_sock_release+0x400/0x400
[   12.677991]  sock_ioctl+0x4a6/0x710
[   12.678386]  ? sock_poll+0x400/0x400
[   12.678816]  ? __sys_socket+0x1c2/0x350
[   12.679275]  ? security_file_ioctl+0xa3/0xc0
[   12.679818]  ? sock_poll+0x400/0x400
[   12.680219]  __se_sys_ioctl+0x101/0x170
[   12.680683]  do_syscall_64+0x33/0x40
[   12.681085]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[   12.681675] RIP: 0033:0x7f89a2384247
[   12.682077] Code: 00 00 90 48 8b 05 49 8c 0c 00 64 c7 00 26 00 00 00 48
c7 c0 ff ff ff ff c3 66 2e 0f 1f 84 00 00 00 00 00 b8 10 00 00 00 0f 05
<48> 3d 01 f0 ff ff 73 01 c3 48 8b 0d 19 8c 0c 00 f7 d8 64 89 01 48
[   12.684221] RSP: 002b:00007ffd49ce4538 EFLAGS: 00000246 ORIG_RAX:
0000000000000010
[   12.685111] RAX: ffffffffffffffda RBX: 00005623b3401c10 RCX:
00007f89a2384247
[   12.685918] RDX: 00007f89a249e000 RSI: 00000000800448d7 RDI:
0000000000000006
[   12.686752] RBP: 00007ffd49ce45c0 R08: 0000000000000001 R09:
00007f89a1a7c700
[   12.687561] R10: 0000000000000000 R11: 0000000000000246 R12:
00005623b3400d50
[   12.688367] R13: 00007ffd49ce46b0 R14: 0000000000000000 R15:
0000000000000000
[   12.689172]
[   12.689345] Allocated by task 125:
[   12.689758]  ____kasan_kmalloc+0xc6/0x100
[   12.690235]  kmem_cache_alloc_trace+0x124/0x200
[   12.690768]  hci_alloc_dev+0x4d/0x1ab0
[   12.691186]  hci_uart_tty_ioctl+0x3ba/0xa20
[   12.691686]  tty_ioctl+0x11ac/0x1b60
[   12.692121]  __se_sys_ioctl+0x101/0x170
[   12.692581]  do_syscall_64+0x33/0x40
[   12.693011]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[   12.693600]
[   12.693773] Freed by task 126:
[   12.694115]  kasan_set_track+0x3d/0x70
[   12.694593]  kasan_set_free_info+0x1f/0x40
[   12.695050]  ____kasan_slab_free+0x10e/0x140
[   12.695551]  kfree+0xeb/0x2d0
[   12.695920]  bt_host_release+0x18/0x20
[   12.696339]  device_release+0x9e/0x1d0
[   12.696791]  kobject_put+0x194/0x2b0
[   12.697188]  hci_uart_tty_close+0x1a7/0x220
[   12.697681]  tty_ldisc_hangup+0x4d7/0x6d0
[   12.698128]  __tty_hangup+0x6b2/0x970
[   12.698569]  tty_release+0x408/0x10e0
[   12.698979]  __fput+0x32f/0x7a0
[   12.699334]  task_work_run+0x15c/0x1e0
[   12.699819]  exit_to_user_mode_prepare+0xeb/0x110
[   12.700338]  syscall_exit_to_user_mode+0x20/0x40
[   12.700880]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[   12.701454]
[   12.701641] Last potentially related work creation:
[   12.702175]  kasan_save_stack+0x27/0x50
[   12.702633]  kasan_record_aux_stack+0xbd/0xe0
[   12.703116]  insert_work+0x4f/0x340
[   12.703536]  __queue_work+0x9cc/0xdb0
[   12.703975]  queue_work_on+0xd8/0x130
[   12.704387]  hci_recv_frame+0x182/0x1e0
[   12.704846]  h4_recv_buf+0x904/0xd40
[   12.705245]  h4_recv+0xf4/0x1b0
[   12.705628]  hci_uart_tty_receive+0x1be/0x380
[   12.706111]  tty_ldisc_receive_buf+0x130/0x170
```

```
[   12.706633]  tty_port_default_receive_buf+0x6a/0x90
[   12.707172]  flush_to_ldisc+0x2e8/0x510
[   12.707630]  process_one_work+0x6df/0xf80
[   12.708112]  worker_thread+0xac1/0x1340
[   12.708572]  kthread+0x2fc/0x320
[   12.708937]  ret_from_fork+0x22/0x30
[   12.709337]
[   12.709543] Second to last potentially related work creation:
[   12.710169]  kasan_save_stack+0x27/0x50
[   12.710627]  kasan_record_aux_stack+0xbd/0xe0
[   12.711111]  insert_work+0x4f/0x340
[   12.711532]  __queue_work+0x9cc/0xdb0
[   12.711983]  queue_work_on+0xd8/0x130
[   12.712392]  hci_event_packet+0x1bce1/0x23430
[   12.712908]  hci_rx_work+0x2a8/0x780
[   12.713308]  process_one_work+0x6df/0xf80
[   12.713783]  worker_thread+0xac1/0x1340
[   12.714212]  kthread+0x2fc/0x320
[   12.714610]  ret_from_fork+0x22/0x30
[   12.715012]
[   12.715186] The buggy address belongs to the object at ffff88800c2ba000
[   12.715186]  which belongs to the cache kmalloc-8k of size 8192
[   12.716629] The buggy address is located 16 bytes inside of
[   12.716629]  8192-byte region [ffff88800c2ba000, ffff88800c2bc000)
[   12.717930] The buggy address belongs to the page:
[   12.718485] page:(____ptrval____) refcount:1 mapcount:0
mapping:0000000000000000 index:0x0 pfn:0xc2b8
[   12.719519] head:(____ptrval____) order:3 compound_mapcount:0
compound_pincount:0
[   12.720364] flags: 0x100000000010200(slab|head)
[   12.720899] raw: 0100000000010200 ffffea0000346808 ffff888006c41270
ffff888006c4c2c0
[   12.721776] raw: 0000000000000000 0000000000010001 00000001ffffffff
0000000000000000
[   12.722676] page dumped because: kasan: bad access detected
[   12.723287]
[   12.723492] Memory state around the buggy address:
[   12.723961]  ffff88800c2b9f00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc
fc fc
[   12.724547]  ffff88800c2b9f80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc
fc fc
[   12.725133] >ffff88800c2ba000: fa fb fb fb fb fb fb fb fb fb fb fb fb fb
fb fb
[   12.725718]                          ^
[   12.726028]  ffff88800c2ba080: fb fb fb fb fb fb fb fb fb fb fb fb fb fb
fb fb
[   12.726612]  ffff88800c2ba100: fb fb fb fb fb fb fb fb fb fb fb fb fb fb
fb fb
[   12.727196]
==================================================================
[   12.727784] Disabling lock debugging due to kernel taint
```

=*=*=*=*=*=*=*=*= BUG REPRODUCE =*=*=*=*=*=*=*=*=

As above introduced, this race condition is highly controllable. This is because the four related functions all call copy_from_user() function after the check of hdev.

```
static int hci_sock_blacklist_add(struct hci_dev *hdev, void __user *arg)
{
    bdaddr_t bdaddr;
    int err;

    if (copy_from_user(&bdaddr, arg, sizeof(bdaddr)))
        return -EFAULT;

    ...
}

static int hci_sock_blacklist_del(struct hci_dev *hdev, void __user *arg)
{
    bdaddr_t bdaddr;
    int err;

    if (copy_from_user(&bdaddr, arg, sizeof(bdaddr)))
        return -EFAULT;

    ...
}

int hci_get_conn_info(struct hci_dev *hdev, void __user *arg)
{
    struct hci_conn_info_req req;
    struct hci_conn_info ci;
    struct hci_conn *conn;
    char __user *ptr = arg + sizeof(req);

    if (copy_from_user(&req, arg, sizeof(req)))
        return -EFAULT;

    ...
}

int hci_get_auth_info(struct hci_dev *hdev, void __user *arg)
{
    struct hci_auth_info_req req;
    struct hci_conn *conn;

    if (copy_from_user(&req, arg, sizeof(req)))
        return -EFAULT;

    ...
}
```

That is, we can adopt userfaultfd to stop these functions and then call the detach routine to release the hdev object. After the hdev is already freed, we handle the page fault from copy_from_user() can let these functions cause UAF. (attacker can further spray the heap during this window)

The provided POC code can be used to prove the feasibility.

=*=*=*=*=*=*=*=*= Bug FIX =*=*=*=*=*=*=*=*=

The adopted patch is presented at
https://git.kernel.org/pub/scm/linux/kernel/git/bluetooth/bluetooth.git/commit/?id=e305509e678b3a4af2b3cfd410f409f7cdaabb52

In short, this patch replaces the lock to the correct one for serialization requirements.

=*=*=*=*=*=*=*=*= Timeline =*=*=*=*=*=*=*=*=

2021-05-30: Bug reported to security@...nel.org and
linux-distros@...openwall.org
2021-05-31: Patch is adopted into Bluetooth tree
2021-06-01: CVE-2021-3573 is assigned

=*=*=*=*=*=*=*=*= Credt =*=*=*=*=*=*=*=*=
LinMa@...ckSec Team
syzkaller of course

Best Regards

Content of type "text/html**" skipped**

Download attachment "POC.zip**" of type "**application/zip**" (6574 bytes)**