Linux kernel (3.x-5.x) use-after-free in the XFRM subsystem

Author: Vitaly Nikolenko
Date: 19 Nov 2018

rev: 0.2

Summary

A kernel use-after-free vulnerability was identified in the XFRM netlink subsystem. After bisecting the vulnerability, we have identified that the issue was introduced in `e682adf021be` - "xfrm: Try to honor policy index if it's supplied by user" in late 2013.
This vulnerability leads to privilege escalation. There are several exploitation scenarios depending on the target kernel version/distribution. We demonstrate one of these scenarios for Ubuntu 18.04 (4.15.x) in the attached PoC. It is worth noting that support for unprivileged usernamespaces (specifically obtaining `CAP_NET_ADMIN` in the current namespace) is required to reach the vulnerable execution path. All non-EOL Ubuntu server (LTS) distributions (Trusty, Xenial, Bionic) allow unprivileged namespaces. We have not performed any further analysis on other distributions due to the initial scope. However, as long as unprivileged usernamespaces are allowed, any recent distribution is likely to be vulnerable.

Technical details

OOB access

The OOB access can be triggered by inserting a new policy with a specific user-defined index and a timer set. When inserting new policies via the `XFRM_MSG_NEWSA` request the following code path is executed:

```
static int xfrm_add_policy(struct sk_buff *skb, struct nlmsghdr *nlh,
                struct nlattr **attrs)
{
        struct net *net = sock_net(skb->sk);
        struct xfrm_userpolicy_info *p = nlmsg_data(nlh);
        struct xfrm_policy *xp;
        struct km_event c;
        int err;
        int excl;

        err = verify_newpolicy_info(p);                    [1]
        if (err)
                return err;
        err = verify_sec_ctx_len(attrs);
        if (err)
                return err;
```



```
        xp = xfrm_policy_construct(net, p, attrs, &err);
        if (!xp)
                return err;

        excl = nlh->nlmsg_type == XFRM_MSG_NEWPOLICY;
        err = xfrm_policy_insert(p->dir, xp, excl);        [2]
        xfrm_audit_policy_add(xp, err ? 0 : 1, true);
        ...
```

In [1], `verify_newpolicy_info` is responsible for checking the user-supplied netlink attributes (`struct xfrm_userpolicy_info`). If this check is successful, a new policy object is allocated and inserted into the global list of policies in [2].

```
static int verify_newpolicy_info(struct xfrm_userpolicy_info *p)
{
        ...
        ret = verify_policy_dir(p->dir);                          [3]
        if (ret)
                return ret;

        if (p->index && ((p->index & XFRM_POLICY_MAX) != p->dir)) [4]
                return -EINVAL;

        return 0;
}
```

The `verify_newpolicy_info` function shown above performs several parameter checks including the `p->dir` check in [3] and [4]. This function ensures that the provided direction is one of the predefined values (`XFRM_POLICY_IN` = 0, `XFRM_POLICY_OUT` = 1 or `XFRM_POLICY_FWD` = 2):

```
static int verify_policy_dir(u8 dir)
{
        switch (dir) {
        case XFRM_POLICY_IN:
        case XFRM_POLICY_OUT:
        case XFRM_POLICY_FWD:
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

The index value is then *aligned* with one of the policy directions (`XFRM_POLICY_MAX` is set to 3). The policy index and direction need to match to pass the check in [4]. For example, inserting a new policy with index 4 and direction 0 will pass all checks above.
The OOB access is triggered on the timer execution path assuming the policy was inserted with a timer set. When the timer expires, the following code path is executed:

```
static void xfrm_policy_timer(struct timer_list *t)
{
        struct xfrm_policy *xp = from_timer(xp, t, timer);
        unsigned long now = get_seconds();
        long next = LONG_MAX;
        int warn = 0;
```



```
        int dir;

        read_lock(&xp->lock);

        if (unlikely(xp->walk.dead))
                goto out;

        dir = xfrm_policy_id2dir(xp->index);          [5]
        ...
expired:
        read_unlock(&xp->lock);
        if (!xfrm_policy_delete(xp, dir))             [6]
                km_policy_expired(xp, dir, 1, 0);
        xfrm_pol_put(xp);
}
```

In [5], the direction is *recomputed* based on the user-provided index. However, this time instead of using `XFRM_POLICY_MAX`, `xfrm_policy_id2dir` function is used to obtain the policy direction:

```
static inline int xfrm_policy_id2dir(u32 index)
{
        return index & 7;
}
```

The index now becomes `4 & 7 = 4` and in [6], this index get passed to `__xfrm_policy_unlink` leading to OOB decrement in [7]:

```
static struct xfrm_policy *__xfrm_policy_unlink(struct xfrm_policy *pol,
                                                int dir)
{
        struct net *net = xp_net(pol);

        if (list_empty(&pol->walk.all))
                return NULL;

        /* Socket policies are not hashed. */
        if (!hlist_unhashed(&pol->bydst)) {
                hlist_del_rcu(&pol->bydst);
                hlist_del(&pol->byidx);
        }

        list_del_init(&pol->walk.all);
        net->xfrm.policy_count[dir]--;                [7]

        return pol;
}
```

However, this OOB access does not lead to any exploitation scenarios.

Use-after-free

The use-after-free 8-byte write primitive can be obtained by manipulating the aforementioned index checks using the following scenario:

1. The first policy object is inserted with index 0 (auto-generated by the subsystem), direction 0 and priority 0.
2. The second policy object is inserted with the user-defined index = 4, direction 0, priority 1 (> 0) and a timer set.

3. XFRM_SPD_IPV4_HTHRESH request is issued to trigger policy rehashing.
4. XFRM_FLUSH_POLICY request is issued freeing the first policy.
5. Once the timer expires on the second policy, UAF is triggered on the first policy that was freed in the previous step.

After steps 1 and 2, two policy objects are inserted into the same list (direction 0). The XFRM_SPD_IPV4_HTHRESH request (step 3) executes the following function *re-inserting* existing policies in reverse order into the bydst list:

```
static void xfrm_hash_rebuild(struct work_struct *work)
{
...
        /* re-insert all policies by order of creation */
         list_for_each_entry_reverse(policy, &net->xfrm.policy_all, walk.all) {
                 if (policy->walk.dead ||
                     xfrm_policy_id2dir(policy->index) >= XFRM_POLICY_MAX) {      [8]
                         /* skip socket policies */
                         continue;
                 }
                 newpos = NULL;
                 chain = policy_hash_bysel(net, &policy->selector,
                                           policy->family,
                                           xfrm_policy_id2dir(policy->index));
                 hlist_for_each_entry(pol, chain, bydst) {
                         if (policy->priority >= pol->priority)
                                 newpos = &pol->bydst;
                         else
                                 break;
                 }
                 if (newpos)
                         hlist_add_behind(&policy->bydst, newpos);
                 else
                         hlist_add_head(&policy->bydst, chain);
         }
```

In [8], the first policy gets re-inserted into the list. However, the second policy with index 4 (`4 & 7 = 4` is now checked against `XFRM_POLICY_MAX = 3` causing this policy to be *skipped* and not reinserted into the bydst policy list. At this point, two policies become *disjoint* but the second policy still points to the first one as shown in the following Figure.

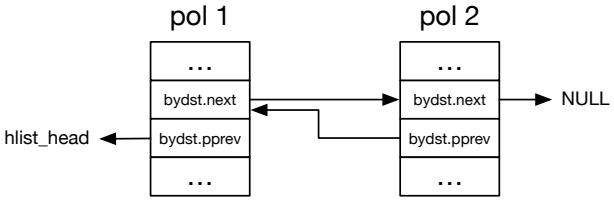

figure
Slabs layout for kmalloc-1024.

In step 4, the request to flush policies frees the first policy in [9], leaving the second policy object in its own disjoint state:

```
int xfrm_policy_flush(struct net *net, u8 type, bool task_valid)
{
```

```
        ...
        for (dir = 0; dir < XFRM_POLICY_MAX; dir++) {
                struct xfrm_policy *pol;
                int i;

        again1:
                hlist_for_each_entry(pol,
                                     &net->xfrm.policy_inexact[dir], bydst) {
                        if (pol->type != type)
                                continue;
                        __xfrm_policy_unlink(pol, dir);
                        spin_unlock_bh(&net->xfrm.xfrm_policy_lock);
                        cnt++;

                        xfrm_audit_policy_delete(pol, 1, task_valid);

                        xfrm_policy_kill(pol);                      [9]
        ...
```

When the preset timer expires on the second policy, the following execution path calls the unlink operation on the second policy leading to UAF write in [10]:

```
static struct xfrm_policy *__xfrm_policy_unlink(struct xfrm_policy *pol,
                                                int dir)
{
        struct net *net = xp_net(pol);

        if (list_empty(&pol->walk.all))
                return NULL;

        /* Socket policies are not hashed. */
        if (!hlist_unhashed(&pol->bydst)) {
                hlist_del_rcu(&pol->bydst);       [10]
                hlist_del(&pol->byidx);
        }

        list_del_init(&pol->walk.all);
        net->xfrm.policy_count[dir]--;

        return pol;
}
```

`hlist_del_rcu` then executes `__hlist_del` on the `bydst` list pointer in the second policy object:

```
static inline void __hlist_del(struct hlist_node *n)
{
        struct hlist_node *next = n->next;
        struct hlist_node **pprev = n->pprev;

        WRITE_ONCE(*pprev, next);                 [11]
        if (next)
                next->pprev = pprev;
}
```

The `pprev` pointer in the second policy object still references the freed first policy. Hence, the `next` pointer in the freed object gets overwritten with 0 (8-byte write) in [11].

Conclusion

This report presents technical analysis of the 0-day UAF 8-byte write (in the XFRM subsystem) leading to privilege escalation. It affects all recent distributions allowing unprivileged namespaces.
The provided PoC was tested on Ubuntu 18.04 Server. However, the exploitation technique is generic and should work on all vulnerable distributions / kernels without any modifications.

