The Net Filter Facility

The Linux net filter is a framework in the kernel that allows modules to observe and modify packets as they pass through the protocol stack. Kernel services or modules register custom hooks/filters by identifying both

- protocol family (e.g., PF_INET) and by
- the point in packet processing (e.g., NF_IP_LOCAL_IN) at which the filter is to be invoked.

The facility is currently available for IPv4, IPv6 and DECnet but could be extended to other protocol families. Each protocol family can provide several processing points in the stack where a packet of that protocol can be passed to a filter.

These points are referred to as hook points or hook types. Hence, when registering a custom hook, the protocol family and the protocol specific hook type must be specified.

Only kernel components or installable modules can directly register hooks --- never application code.
Hook management structures

A statically allocated array of list headers defined in net/netfilter/core.c is the root of the hook chains for each protocol and hook types. NF_MAX_HOOKS, the maximum types of hooks a protocol can support has been defined as 8 in include/linux/netfilter_ipv4.h.

52/* In this code, we can be waiting indefinitely for userspace to
53 * service a packet if a hook returns NF_QUEUE. We could keep a count
54 * of skbuffs queued for userspace, and not deregister a hook unless
55 * this is zero, but that sucks. Now, we simply check when the
56 * packets come back: if the hook is gone, the packet is discarded. */

57 struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS];

47 /* Largest hook number + 1 */
48 #define NF_MAX_HOOKS 8

There are 5 hook types defined for the IPV4 protocol. It is common for a packet to be processed by multiple hooks.

41 /* IP Hooks */
42 /* After promisc drops, checksum checks. */
43 #define NF_IP_PRE_ROUTING 0
44 /* If the packet is destined for this box. */
45 #define NF_IP_LOCAL_IN 1
46 /* If the packet is destined for another interface. */
47 #define NF_IP_FORWARD 2
48 /* Packets coming from a local process. */
49 #define NF_IP_LOCAL_OUT 3
50 /* Packets about to hit the wire. */
51 #define NF_IP_POST_ROUTING 4

52 #define NF_IP_NUMHOOKS 5
Hook function prototype

A hook handler has the following prototype.

```c
53 typedef unsigned int nf_hookfn(unsigned int hooknum,
54                                struct sk_buff **skb,
55                                const struct net_device *in,
56                                const struct net_device *out,
57                                int (*okfn)(struct sk_buff *));
```

Note that the prototype is a *typedef*. This C construct will allow a declaration of the form:

```c
nf_hookfn *hook;
```

to actually declare a pointer to a function having the parameters shown above, even though the declaration looks more like a pointer to a structure.

Here is an example, taken from a firewall. Even though a hook is passed a pointer to the “okfn” a typical hook will never invoke it, because that would prevent other hooks from inspecting the packet. The OK function is normally invoked by the netfilter code after all hooks have processed the packet.

```c
423 static unsigned int fw_output(
424 unsigned int hook,
425 struct sk_buff **pskb,
426 const struct net_device *in,
427 const struct net_device *out,
428 int (*okfn)(struct sk_buff *))
```
Defining and registering a netfilter hook

Each custom hook is defined using the following `nf_hook_ops` structure. This structure is passed to the `nf_register_hook` function.

```c
59 struct nf_hook_ops
60 {
61   struct list_head list;
62
63   /* User fills in from here down. */
64   nf_hookfn *hook;
65   struct module *owner;
66   int pf;
67   int hooknum;
68   /* Hooks are ordered in ascending priority. */
69   int priority;
70  };
```

Structure elements are used as follows:

- **list:** links all hooks of a common PROTO and hook type into the hook chain
- **pf:** protocol family (PF_INET) of the filter.
- **hooknum:** the protocol specific hook type (NF_IP_FORWARD) identifier.
- **priority:** determines order of the hook in the list.
- **hook:** A pointer to the hook function. It's prototype was shown previously
Registering a firewall hook

The firewall whose handler was shown earlier registered its handler in this way. The hook element of the structure should contain the address of the filter function.

```c
488 static struct nf_hook_ops postroute_ops = {
489          .hook           = fw_output,
490 #if (LINUX_VERSION_CODE >= 0x020500)
491          .owner          = THIS_MODULE,
492 #endif
493          .pf             = PF_INET,
494          .hooknum        = NF_IP_POST_ROUTING,
495          .priority       = NF_IP_PRI_FILTER,
496 }; 
532    rc = nf_register_hook(&postroute_ops);
533    if (rc < 0)
534    { 
535        printk("Register postroute failed \n");
536        return(-1);
537    }
```
Hook priorities

During registration hooks are inserted on the chain in priority order. Low numbers mean higher priority and packets are passed to high priority hooks before low priority hooks.

```c
54 enum nf_ip_hook_priorities {
55    NF_IP_PRI_FIRST = INT_MIN,
56    NF_IP_PRI_CONNTRACK_DEFRAG = -400,
57    NF_IP_PRI_RAW = -300,
58    NF_IP_PRI_SELINUX_FIRST = -225,
59    NF_IP_PRI_CONNTRACK = -200,
60    NF_IP_PRI_BRIDGE_SABOTAGE_FORWARD = -175,
61    NF_IP_PRI_MANGLE = -150,
62    NF_IP_PRI_NAT_DST = -100,
63    NF_IP_PRI_BRIDGE_SABOTAGE_LOCAL_OUT = -50,
64    NF_IP_PRI_FILTER = 0,
65    NF_IP_PRI_NAT_SRC = 100,
66    NF_IP_PRI_SELINUX_LAST = 225,
67    NF_IP_PRI_CONNTRACK_HELPER = INT_MAX - 2,
68    NF_IP_PRI_NAT_SEQ_ADJUST = INT_MAX - 1,
69    NF_IP_PRI_CONNTRACK_CONFIRM = INT_MAX,
70    NF_IP_PRI_LAST = INT_MAX,
71 };```

6
Registering a hook

The `nf_register_hook()` function defined in `net/core/netfilter.c` adds the `nf_hook_ops` structure that defines a custom hook to the appropriate list based on the protocol family and filter type. Since the list is ordered by ascending priority values, invocation order is lowest numerical value first.

```c
60 int nf_register_hook(struct nf_hook_ops *reg) 61 { 62 struct list_head *i; 63 64 spin_lock_bh(&nf_hook_lock); 65 list_for_each(i, &nf_hooks[reg->pf][reg->hooknum]) { 66     if (reg->priority < (((struct nf_hook_ops *)i)->priority) 67         break; 68 } 69 list_add_rcu(&reg->list, i->prev); 70 spin_unlock_bh(&nf_hook_lock); 71 synchronize_net(); 72 return 0; 73}
```
IP Packet transmission through the netfilter

From `ip_push_pending_frames()`, the IP packet is pushed to the netfilter layer using the `NF_HOOK` macro defined in `include/linux/netfilter.h`. Parameters passed include the output device to be used and the final "OK" output function to be invoked on successful verdict from all the hooks in the list. The hook type is `NF_IP_LOCAL_OUT`. The input device is set to NULL, since the packet originated on the local host.

```c
err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, 
    NULL, rt->u.dst.dev, dst_output);
```

211/* Activate hook; either okfn or kfree_skb called, unless a hook
212 returns NF_STOLEN (in which case, it's up to the hook to deal with
213 the consequences).
214
215 Returns -ERRNO if packet dropped. Zero means queued, stolen or
216 accepted. <---- No it doesn't
217*/
218
219/* RR: 
220 > I don't want nf_hook to return anything because people might forget
221 > about async and trust the return value to mean "packet was ok".
222
223 AK:
224 Just document it clearly, then you can expect some sense from kernel
225 coders :)
226*/
227
228/* This is gross, but inline doesn't cut it for avoiding the function
229 call in fast path: gcc doesn't inline (needs value tracking?). --RR */
230
231/* HX: It's slightly less gross now. */
232```
Hook macros

The macro translates to a call to the `nf_hook_thresh()` function.

- If a value of 1 is returned, the packet is passed to the OK function.
- The return code from the OK function is then returned to the caller of NF_HOOK.

```c
#define NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, thresh)                        
({int __ret;                         
  if ((__ret = nf_hook_thresh(pf, hook, &(skb), indev, outdev, okfn, thresh, 1)) == 1) 
    __ret = (okfn)(skb); 
  __ret;})
```

The NF_HOOK macro just invokes NF_HOOK_THRESH with the INT_MIN threshold parameter. This makes all hooks eligible to process the packet regardless of their priority.

```c
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) 
NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, INT_MIN)
```

The NF_HOOK_COND macro passes a cond argument through to `nf_hook_thresh()`. The NF_HOOK_COND macro also unconditionally sets thresh to INT_MIN. This macro is used in IP multicast output.

```c
#define NF_HOOK_COND(pf, hook, skb, indev, outdev, okfn, cond) 
({int __ret;                                               
  if ((__ret=nf_hook_thresh(pf, hook, &(skb), indev, outdev, okfn, INT_MIN, cond)) == 1) 
    __ret = (okfn)(skb); 
  __ret;})
```
The nf_hook_thresh() function

This function invokes the nf_hook_slow() function if the netfilter debug option is defined or if there are hooks/filters set for the specific protocol family and hook type. If the cond argument is 0 or the hook chain is empty a 1 is returned indicating that the OK function should be invoked.

```c
188 static inline int nf_hook_thresh(int pf, unsigned int hook,
189                                  struct sk_buff **pskb,
190                                  struct net_device *indev,
191                                  struct net_device *outdev,
192                                  int (*okfn)(struct sk_buff *),
193                                  int thresh,
194                                  int cond)
195 {
196     if (!cond)
197         return 1;
198 #ifndef CONFIG_NETFILTER_DEBUG
199     if (list_empty(&nf_hooks[pf][hook]))
200         return 1;
201 #endif
202 return nf_hook_slow(pf, hook, pskb, indev,
203                          outdev, okfn, thresh);
204 }
```
Hook actions

The following actions and corresponding return codes may be taken by a hook.

15 /* Responses from hook functions. */
16 #define NF_DROP   0
17 #define NF_ACCEPT 1
18 #define NF_STOLEN 2
19 #define NF_QUEUE   3
20 #define NF_REPEAT  4
21 #define NF_STOP    5
22 #define NF_MAX_VERDICT NF_STOP
23
24 /* we overload the higher bits for encoding auxiliary data such as the queue number. Not nice, but better than additional function arguments. */
26 #define NF_VERDICT_MASK 0x0000ffff
27 #define NF_VERDICT_BITS 16
29 #define NF_VERDICT_QMASK 0xffff0000
30 #define NF_VERDICT_QBITS 16
32 #define NF_QUEUE_NR(x) (((x << NF_VERDICT_QBITS) & NF_VERDICT_QMASK) | NF_QUEUE)
Invoking the hook chain

When the `cond` parameter is non-zero and the hook list is non-empty, the `nf_hook_slow()` function is invoked. The `nf_hook_slow()` function is defined in `net/core/netfilter.c`, its task is to invoke each hook in the specified list, and based on the verdict from the hooks, the appropriate action is taken.

In summary the return values mean the following to the NF_HOOK macro:

- **Negative**: packet was dropped.
- **Zero**: packet was queued or stolen so nothing more to do.
- **One**: the packet was not dropped, queued, stolen. The NF_HOOK macro invokes the OK function directly.

```c
int nf_hook_slow(int pf, unsigned int hook, struct sk_buff **pskb, struct net_device *indev, struct net_device *outdev, int (*okfn)(struct sk_buff *), int hook_thresh) {
    struct list_head *elem;
    unsigned int verdict;
    int ret = 0;
    /* Returns 1 if okfn() needs to be executed by the caller, -EPERM for NF_DROP, 0 otherwise. */
    
    /* ...
    
    return ret;
    */
```
Processing hook chain

The actual processing of the hook chain is done in `nf_iterate` which returns the final verdict. Note that NF_STOP implies NF_ACCEPT. This presumably allows high priority hooks to impose their will and thus disabling the normal procedure in which any hook can demand NF_DROP.

```c
/* We may already have this, but read-locks nest anyway */
rcu_read_lock();
elem = &nf_hooks[pf][hook];

next_hook:
    verdict = nf_iterate(&nf_hooks[pf][hook], pskb, hook,
                        indev, outdev, &elem, okfn, hook_thresh);

    if (verdict == NF_ACCEPT || verdict == NF_STOP) {
        ret = 1;
        goto unlock;
    } else if (verdict == NF_DROP) {
        kfree_skb(*pskb);
        ret = -EPERM;
    } else if ((verdict & NF_VERDICT_MASK) == NF_QUEUE) {
        NFDEBUG("nf_hook: Verdict = QUEUE.\n";
        if (!nf_queue(pskb, elem, pf, hook, inddev, outdev, okfn,
                      verdict >> NF_VERDICT_BITS))
            goto next_hook;
    }
unlock:
rcu_read_unlock();
return ret;
```
Iterating through the hook chain

The `nf_iterate()` function is defined in `net/core/netfilter.c`

```c
116 unsigned int nf_iterate(struct list_head *head,
117         struct sk_buff **skb,
118         int hook,
119         const struct net_device *indev,
120         const struct net_device *outdev,
121         struct list_head **i,
122         int (*okfn)(struct sk_buff *),
123         int hook_thresh)
124 {

One iteration of this loop occurs for each hook called.

```c
127     /*
128     * The caller must not block between calls to this
129     * function because of risk of continuing from deleted
130     * element    */

131     list_for_each_continue_rcu(*i, head) {
132         struct nf_hook_ops *elem = (struct nf_hook_ops *)*i;
133

This would appear to skip over hooks whose values are numerically lower than the hook threshold. The motivation for doing this is unclear. When called via NF_HOOK, the threshold is INT_MIN.

```c
134     if (hook_thresh > elem->priority)  
135         continue;
136
```
Invoking an individual hook

The call to `elem->hook` passes the packet to a function such as `fw_output()` which returns its verdict on the packet.

```c
137    /* Optimization: we don't need to hold module
138       reference here, since function can't sleep. --RR */
139       verdict = elem->hook(hook, skb, indev, outdev, okfn);
```

The verdict returned by the hook function determines the action taken. An immediate return, possibly aborting the send, is made if the value returned is NF_QUEUE, NF_STOLEN, or NF_DROP. For values of NF_REPEAT or NF_ACCEPT the `list_for_each()` loop continues.

```c
140       if (verdict != NF_ACCEPT) {
141 #ifdef CONFIG_NETFILTER_DEBUG
142          if (unlikely((verdict & NF_VERDICT_MASK)
143                          > NF_MAX_VERDICT)) {
144             NFDEBUG("Evil return from %p(%u).\n",,
145              elem->hook, hook);
146             continue;
147          }
148 #endif
149       } if (verdict != NF_REPEAT)
150            return verdict;
151       *i = (*i)->prev;
152 } // end of verdict != NF_ACCEPT
153 } // end of list_for_each()
154 return NF_ACCEPT;
155 }
```
The \textit{dst\_output\_function}

In the \textit{udpsend} notes it was seen that if the packet is accepted for transmission by \textit{nf\_hook\_slow}, the \textit{okfn()}, \textit{dst\_output()}, is called. It simply passes control to the \textit{output} function associated with the \textit{dst} structure that is presently bound to the \textit{sk\_buff}.

225 static inline int dst_output(struct sk_buff *skb)  
226 {  
227 return skb->dst->output(skb);  
228 }
The *ip_output* function

The pointer `skb->dst` refers to the route cache element associated with this packet's source and destination. During routing, `rt->u.dst->output` was set to `ip_output()` which is defined in `net/ipv4/ip_output.c`.

*The ip_output() function*

- sets `skb->dev` to the device associated with the route's associated output device structure and
- sets the protocol type to `ETH_P_IP`.

This indicates that the value 0x8000 represents an IP packet even if the output device is not an ethernet device. This used to be done in `ip_finish_output`. This function used to explicitly invoke `ip_do_nat()`.

```c
277 int ip_output(struct sk_buff *skb)
278 {  
279    struct net_device *dev = skb->dst->dev;
280    IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);
281    skb->dev = dev;
282    skb->protocol = htons(ETH_P_IP);
283    return NF_HOOK_COND(PF_INET, NF_IP_POST_ROUTING, skb,
284                        NULL, dev, ip_finish_output,
285                        !(IPCB(skb)->flags & IPSKB_REROUTED));
286 }
```

Note that the above two lines are the *only things that occur* between the end of the `NF_LOCAL_OUTPUT` hook and the post routing hook. But its possible that significant transformations (e.g. NAT could have occurred within the `NF_LOCAL_OUTPUT` hook).
The `ip_finish_output()` function

The `ip_finish_output()` is responsible for invoking fragmentation if the packet is too long and gso is not supported.

```c
static inline int ip_finish_output(struct sk_buff *skb) { 
#if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM) 
    /* Policy lookup after SNAT yielded a new policy */ 
    if (skb->dst->xfrm != NULL) { 
        IPCB(skb)->flags |= IPSKB_REROUTED; 
        return dst_output(skb); 
    } 
#endif 
    if (skb->len > dst_mtu(skb->dst) && !skb_is_gso(skb)) 
        return ip_fragment(skb, ip_finish_output2); 
    else 
        return ip_finish_output2(skb); 
} 
```

The *ip_finish_output2()* function

The *ip_finish_output2()* function is also defined in net/ipv4/ip_output.c. It first tries to confirm that sufficient *headroom* exists for the MAC header. If not, it will ask *skb_realloc_headroom()* to reallocate the *kmalloc’d* component with sufficient headroom.

You should make real sure that your protocol *doesn’t* trigger the call to *skb_realloc_headroom()*

```c
163 static inline int ip_finish_output2(struct sk_buff *skb) {
164    struct dst_entry *dst = skb->dst;
165    struct hh_cache *hh = dst->hh;
166    struct net_device *dev = dst->dev;
167    int hh_len = LL_RESERVED_SPACE(dev);
168
169    /* Be paranoid, rather than too clever. */
170    if (unlikely(skb_headroom(skb) < hh_len &&
171        dev->hard_header)) {
172        struct sk_buff *skb2;
173
174        skb2 = skb_realloc_headroom(skb,
175                                  LL_RESERVED_SPACE(dev));
176        if (skb2 == NULL) {
177            kfree_skb(skb);
178            return -ENOMEM;
179        }
180
181        if (skb->sk)
182            skb_set_owner_w(skb2, skb->sk);
183        kfree_skb(skb);
184        skb = skb2;
185    }
```

The *skb_realloc_headroom()* succeeds *skb2*, originally a clone of *skb*, now points to a new *kmalloc’d* component. Hence it is necessary to charge the *sock’s* write buffer quota, and then free the original *sk_buff* and *kmalloc’d* area.

```c
179    if (skb->sk)
180        skb_set_owner_w(skb2, skb->sk);
181    kfree_skb(skb);
182    skb = skb2;
183}
```
Passing the packet to the dev layer

There are two mechanisms by which calls to the dev layer may be made. If the dst_entry has an hh_cache pointer, then the hh_cache entry must contain both the hardware header itself and a pointer to an output function at the device / link layer.

The hh_output() function is set to dev_queue_xmit() if the ARP cache element is in the NUD_REACHABLE state, but will point to neigh_resolve_output() when the entry becomes stale.

If there is no hh pointer in the dst_entry, the neighbor pointer that was established when the route cache entry was constructed will be used. This neighbor structure has an output function pointer which was set to neigh->ops->output. For ethernet devices, this function is neigh_resolve_output(). Otherwise (for a loopback, point to point, or virtual device) it set to invoke dev_queue_xmit() by the arp_constructor() function that is called when each neighbour structure was created.

185    if (hh) {
186       int hh_alen;
187
188       read_lock_bh(&hh->hh_lock);
189       hh_alen = HH_DATA_ALIGN(hh->hh_len);
190       memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
191       read_unlock_bh(&hh->hh_lock);
192
193       skb_push(skb, hh->hh_len);
194       return hh->hh_output(skb);
195
196   } else if (dst->neighbour)
197      return dst->neighbour->output(skb);
Failure of routing

If there is no hardware header structure and no neighbor structure available, then there is no way to send the packet and it must be dropped. The `net_ratelimit()` function is used to limit the number of printk's generated to not more than 1 every 5 seconds to avoid flooding the syslog in case something is badly amiss in the network setup.

```c
197    if (net_ratelimit())
198       printk(KERN_DEBUG
199          "ip_finish_output2: No header cache and no neighbour!\n");
200    kfree_skb(skb);
201    return -EINVAL;
201 }
```
Hardware header length macros

228 /* cached hardware hdr; allow for machine alignment */
229 #define HH_DATA_MOD 16

230 #define HH_DATA_OFF(__len) \n231 (HH_DATA_MOD - (((__len - 1) & (HH_DATA_MOD - 1)) + 1))

232 #define HH_DATA_ALIGN(__len) \n233 (((__len)+(HH_DATA_MOD-1))&~(HH_DATA_MOD - 1))
234 unsigned long hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / \n235 sizeof(long)];
236
237 /* Reserve HH_DATA_MOD byte aligned hard_header_len, but at \n238 least that much.
239 * Alternative is:
240 * dev->hard_header_len ? (dev->hard_header_len + \n241 * (HH_DATA_MOD - 1)) & ~(HH_DATA_MOD - 1) : 0
242 *
243 * We could use other align values, but we must maintain the
244 * relationship HH alignment <= LL alignment.
245 *
246 * LL_ALLOCATED_SPACE also takes into account the tailroom the
247 * device may need.
248 */
249 #define LL_RESERVED_SPACE(dev) \n250 (((dev)->hard_header_len + \n251 (dev)->needed_headroom)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
252 #define LL_RESERVED_SPACE_EXTRA(dev,extra) \n253 (((dev)->hard_header_len+(dev)->needed_headroom + \n254 (extra))&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
254
255 #define LL_ALLOCATED_SPACE(dev) \n256 (((dev)->hard_header_len+(dev)->needed_headroom + \n257 (dev)->needed_tailroom)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
254