IP Layer Input Packet Processing (continuation)

The *ip_rcv_finish()* function is defined in net/ipv4/ip_input.c and is indirectly called from *ip_rcv()* as an *okfn()* passed through the netfilter mechanism. Its primary missions are to call *ip_route_input()* which determines the next function to handle the *sk_buff* and to compile any IP header options into the *sk_buff's* control buffer.

```
309 static inline int ip_rcv_finish(struct sk_buff *skb)
310 {
311     struct net_device *dev = skb->dev;
312     struct iphdr *iph = skb->nh.iph;
```

The value of *skb->dst* will typically (always?) be NULL if the packet was received from the outside world. When this is the case, *ip_route_input()* is called to set *skb->dst* to a destination entry that describes the next course of action. The address of the next function to handle the *sk_buff* is selected from.

ip_forward()	Forward to destination not on this host.
ip_local_deliver()	Process and deliver packet to transport layer .
ip_error()	An error occurred somewhere.

```
314 /*  Initialise the virtual path cache for the
        packet. It describes how the packet travels
        inside Linux networking.
317 */
318     if (skb->dst == NULL) {
319         if (ip_route_input(skb, iph->daddr,
                               iph->saddr, iph->tos, dev))
320         goto drop;
321     }

323 #ifdef CONFIG_NET_CLS_ROUTE
324     if (skb->dst->tclassid) {
325         struct ip_rt_acct *st = ip_rt_acct +
                          256 * smp_processor_id();
326         u32 idx = skb->dst->tclassid;
327         st[idx&0xFF].o_packets++;
328         st[idx&0xFF].o_bytes+=skb->len;
329         st[(idx>>16)&0xFF].i_packets++;
330         st[(idx>>16)&0xFF].i_bytes+=skb->len;
331     }
332 #endif
```

Check for presence of IP options. The only ones that appear to impact processing here are strict and loose source routing.

```
334     if (iph->ihl > 5) {
335          struct ip_options *opt;
336
337   /*It looks as overkill, because not all
338     IP options require packet mangling.
339     But it is the easiest for now, especially taking into
        account thatcombination of IP options and running
        sniffer is extremely rare condition.
342                                          --ANK (980813)
343   */
```

The headroom of an skb is defined to be the difference between the data and head pointers.

```
885 static inline int skb_headroom(const struct sk_buff *skb)
886 {
887     return skb->data - skb->head;
888 }
```

```
345          if (skb_cow(skb, skb_headroom(skb)))
346               goto drop;
```

The skb_cow() function is defined in include/linux/skbuff.h. It ensures that the headroom of the sk_buff is at least 16 bytes. The sk_buff is reallocated if its headroom is inadequate of small or if it has a clone. Recall that dev_alloc_skb() used skb_reserve() to establish a 16 byte headroom when the packet was allocated. Thus for the ‘‘normal’’ case the value of delta will be 0 here.

```
1071 static inline int
1072 skb_cow(struct sk_buff *skb, unsigned int headroom)
1073 {
1074     int delta = (headroom > 16 ? headroom : 16)
                        - skb_headroom(skb);
1075
1076     if (delta < 0)
1077          delta = 0;
```

When the headroom is small or the sk_buff is cloned, reallocate the sk_buff with specified headroom size.

```
1079     if (delta || skb_cloned(skb))
1080          return pskb_expand_head(skb,
                    (delta+15) & ~15, 0, GFP_ATOMIC);
1081     return 0;
1082 }
```

The value of iph is re–initialized as skb_cow() may have reallocated the sk_buff header.

```
 347          iph = skb->nh.iph;
 348
 349          skb->ip_summed = 0;
```

The ip_options_compile() function compiles IP options into a somewhat structured representation that is described by struct inet_skb_parm and resides in the control buffer portion of the struct sk_buff. The control buffer (skb->cb) is a buffer of 48 bytes, into which private variables may be temporarily saved by any layer of networking stack.

```
 350          if (ip_options_compile(NULL, skb))
 351               goto inhdr_error;
```

The IPCB macro, defined in include/net/ip.h, casts a pointer to the control buffer to type struct inet_skb_parm

```
  58 #define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
```

The local opt pointer of struct ip_options type is set to point to the options that have been compiled into the control buffer. When source route options are present in the IP options, opt->srr is not NULL.

```
353          opt = &(IPCB(skb)->opt);
354          if (opt->srr) {
355               struct in_device *in_dev =
                              in_dev_get(dev);
```

The test for the presence of the in_dev structure is a bit odd. One would think its presence should be mandatory, but processing continues without it if it is not present.

```
356               if (in_dev) {
```

IN_DEV_SOURCE_ROUTE() is a macro that is defined in include/linux/inetdevice.h. It returns true if both IP and the input device were configured to allow strict? source routing. If source routing is not allowed, the packet must be dropped.

```
 42 #define IN_DEV_SOURCE_ROUTE(in_dev)
        (ipv4_devconf.accept_source_route &&
        (in_dev)->cnf.accept_source_route)
```

The term martians is commonly used in Linux to refer to unresolvable addresses. IN_DEV_LOG_MARTIANS() returns true if IP or the device were configured to log source and destination addresses of packets associated with failed source routes.

```
 45 #define IN_DEV_LOG_MARTIANS(in_dev)
        (ipv4_devconf.log_martians ||(in_dev)->cnf.log_martians)
```

```
357                    if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
358                         if(IN_DEV_LOG_MARTIANS(in_dev)
                                 && net_ratelimit())
359                              printk(KERN_INFO "source
                                      route option %u.%u.%u.%u
                                      -> %u.%u.%u.%u\n",
360                                   NIPQUAD(iph->saddr),
                                      NIPQUAD(iph->daddr));
361                         in_dev_put(in_dev);
362                         goto drop;
363                    }
364                    in_dev_put(in_dev);
365               }
```

Arrival here indicates that there are source routing options and that they are allowed to be processed.

```
366               if (ip_options_rcv_srr(skb))
367                    goto drop;
368          }
369     }
```

Processing of source routing options

The ip_options_rcv_srr() function, defined in net/ipv4/ip_options.c verifies that the option data is syntactically sensible, extracts the next hop address from the options, calls ip_route_input() to determine if it is reachable, and returns 0 on success.

```
566 int ip_options_rcv_srr(struct sk_buff *skb)
567 {
568     struct ip_options *opt = &(IPCB(skb)->opt);
569     int srrspace, srrptr;
570     u32 nexthop;
571     struct iphdr *iph = skb->nh.iph;
572     unsigned char * optptr = skb->nh.raw + opt->srr;
573     struct rtable *rt = (struct rtable*)skb->dst;
574     struct rtable *rt2;
575     int err;
```

If no source route option is specified, opt->srr is not set and success is returned. Since this was previously checked for non-zero, there must be another caller of this function out there somewhere!

```
577     if (!opt->srr)
578         return 0;
```

PACKET_HOST is a packet type defined in include/linux/if_packet.h. It is the default type that is assigned when an sk_buff is allocated. For packet types other than PACKET_HOST, we return -EINVAL.

```
 24 #define PACKET_HOST       0 /* To us                */
 25 #define PACKET_BROADCAST  1 /* To all               */
```

```
580     if (skb->pkt_type != PACKET_HOST)
581         return -EINVAL;
```

When the destination is gatewayed/direct route (i.e. not RT_LOCAL) and strict source routing needs to be enforced, this is an error because this host must own the current destination. An ICMP message is sent and -EINVAL is returned. If it is a loose source route, and the route type is RT_UNICAST then this host wasn't in the list and the packet is just forwarded normally.

```
582     if (rt->rt_type == RTN_UNICAST) {
583         if (!opt->is_strictroute)
584             return 0;
585         icmp_send(skb, ICMP_PARAMETERPROB, 0,
                htonl(16<<24));
586         return -EINVAL;
587     }
```

For route types other than RTN_LOCAL, we return -EINVAL.

```
588     if (rt->rt_type != RTN_LOCAL)
589         return -EINVAL;
```

Updating the next hop address

Arrival here implies that source routing is in effect and that we own the current destination address. In that case the destination address must be put back in the to list and the next element of the list made the destination.. unless of course the end of the route is us. The for loop is apparently handling the case in which a source route that contains multiple interfaces owned by us is specified!

```
591     for (srrptr=optptr[2], srrspace = optptr[1];
             srrptr <= srrspace; srrptr += 4) {
592          if (srrptr + 3 > srrspace) {
593               icmp_send(skb, ICMP_PARAMETERPROB, 0,
                       htonl((opt->srr+2)<<24));
594               return -EINVAL;
595          }
596          memcpy(&nexthop, &optptr[srrptr-1], 4);
598          rt = (struct rtable*)skb->dst;
599          skb->dst = NULL;
600          err = ip_route_input(skb, nexthop,
                            iph->saddr, iph->tos, skb->dev);
601          rt2 = (struct rtable*)skb->dst;
```

Route to next hop must be either RTN_UNICAST or RTN_LOCAL.

```
602          if (err || (rt2->rt_type != RTN_UNICAST &&
                  rt2->rt_type != RTN_LOCAL)) {
603               ip_rt_put(rt2);
604               skb->dst = &rt->u.dst;
605               return -EINVAL;
606          }
607          ip_rt_put(rt);
```

When route to next hop is of type RTN_UNICAST, we exit the loop. Note that "skb->dst" now points to routing cache entry with next hop address as destination.

```
608          if (rt2->rt_type != RTN_LOCAL)
609               break;
```

We reach here if next hop address is strangely our own address (since route type is RTN_LOCAL). In that case, we copy next hop address into destination address field of IP packet.

```
610          /* Superfast 8) loopback forward */
611          memcpy(&iph->daddr, &optptr[srrptr-1], 4);
612          opt->is_changed = 1;
613     }
```

```
614     if (srrptr <= srrspace) {
615          opt->srr_is_hit = 1;
616          opt->is_changed = 1;
617     }
618     return 0;
619 }
```

Back in ip_rcv_finish, after processing IP options, input function of destination entry is triggered. Recall that skb->dst was set by ip_route_input. Note that the input function is one of the three below:

ip_forward:	Forwarded to destination.
ip_local_deliver:	Process and deliver packet to transport layer.
ip_error:	An error occurred somewhere. Packet is passed to this function which might send an ICMP message.

```
371     return skb->dst->input(skb);
```

In case of any error, the sk_buff is discarded.

```
373 inhdr_error:
374     IP_INC_STATS_BH(IpInHdrErrors);
375 drop:
376     kfree_skb(skb);
377     return NET_RX_DROP;
378 }
```

Determining the next hop with *ip_route_input()*

The ip_route_input() function is defined in net/ipv4/route.c. If first tries to find a suitable destination structure in the route cache and if that fails it invokes ip_route_input_slow() to perform a FIB lookup.

```
1622 int ip_route_input(struct sk_buff *skb, u32 daddr, u32
             saddr, u8 tos, struct net_device *dev)
1624 {
1625     struct rtable * rth;
1626     unsigned        hash;
1627     int iif = dev->ifindex;
1628
1629     tos &= IPTOS_RT_MASK;
```

The rt_hash_code() function returns the hash code that is used as an index into the route cache.

```
1630     hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
```

The hash function is implemented by the inline function rt_hash_code(). The code is derived from the source and destination addresses, the input interface index and the type of service.

```
 203 static __inline__ unsigned rt_hash_code(u32 daddr,
                                           u32 saddr, u8 tos)
 204 {
 205     unsigned hash = ((daddr & 0xF0F0F0F0) >> 4) |
 206                     ((daddr & 0x0F0F0F0F) << 4);
 207     hash ^= saddr ^ tos;
 208     hash ^= (hash >> 16);
 209     return (hash ^ (hash >> 8)) & rt_hash_mask;
 210 }
```

The hash code returned by the above function is used by ip_route_input to identify the proper chain in the rt_hash_table structure. First the chain is locked, and then all elements are examined in a search for an entry having the required attributes. CONFIG_IP_ROUTE_FWMARK is an option to specify different routes for packets with different (netfilter) mark values.

```
1632     read_lock(&rt_hash_table[hash].lock);
1633     for (rth = rt_hash_table[hash].chain; rth; rth =
              rth->u.rt_next) {
1634          if (rth->key.dst == daddr &&
1635               rth->key.src == saddr &&
1636               rth->key.iif == iif &&
1637               rth->key.oif == 0 &&
1638 #ifdef CONFIG_IP_ROUTE_FWMARK
1639               rth->key.fwmark == skb->nfmark &&
1640 #endif
1641               rth->key.tos == tos) {
```

On finding a match, the time of last use for this entry is updated. The dst_hold() function simply increments the reference count (atomic_inc(&dst->__refcnt)) The distinction between __use and __refcnt is not clear at present.

```
1642               rth->u.dst.lastuse = jiffies;
1643               dst_hold(&rth->u.dst);
1644               rth->u.dst.__use++;
1645               rt_cache_stat[smp_processor_id()].in_hit++;
1646               read_unlock(&rt_hash_table[hash].lock);
```

Set skb->dst to this entry and return.

```
1647               skb->dst = (struct dst_entry*)rth;
1648               return 0;
1649          }
1650     }
```

Falling out of the loop means a route couldn't be found in the route cache.

```
1651     read_unlock(&rt_hash_table[hash].lock);
```

Reaching this point in ip_route_input() implies that a suitable routing element was not present in the route cache. If the destination is a multicast address, it is necessary to determine whether the interface on which this packet was received belongs to this multicast group. The comment below describes how multicast routing is complicated by broken or deficient multicast filters on many ethernet cards.

```
1653     /*   Multicast recognition logic is moved from route
                cache to here.  The problem was that too many
                Ethernet cards have broken/missing hardware
                multicast filters :-( As result the host on
                multicasting network acquires a lot of useless route
                cache entries, sort of SDR messages from all the
                world. Now we try to get rid of them. Really,
                provided software IP multicast filter is organized
                reasonably (at least, hashed), it does not result in
                a slowdown comparing with route cache reject
                entries. Note, that multicast routers are not
                affected, because route cache entry is  created
                eventually.
1663      */
1664     if (MULTICAST(daddr)) {
1665          struct in_device *in_dev;
1666
1667          read_lock(&inetdev_lock);
```

Each net_device that supports IP traffic must also an associate struct in_device. The __in_dev_get() function returns its address.

```
1668          if ((in_dev = __in_dev_get(dev)) != NULL) {
```

The ip_check_mc() function, defined in net/ipv4/igmp.c, returns true if the interface is a member of the multicast group identified by daddr.

```
1669                int our = ip_check_mc(in_dev, daddr);
```

The mc_list element of struct in_device points to a linked list of the ip_mc_list structures that describes the multicast groups of which the network interface is a member.[1]

```
761 int ip_check_mc(struct in_device *in_dev, u32 mc_addr)
762 {
763     struct ip_mc_list *im;
765     read_lock(&in_dev->lock);
766     for (im=in_dev->mc_list; im; im=im->next) {
767          if (im->multiaddr == mc_addr) {
768               read_unlock(&in_dev->lock);
769               return 1;
770          }
771     }
772     read_unlock(&in_dev->lock);
773     return 0;
774 }
```

1 http://www.tldp.org/HOWTO/Multicast-HOWTO-7.html

Back in ip_route_input(), if the the destination was a multicast address and the interface was a member of the associated group and several configuration constraints are met, then the packet is sent to ip_route_input_mc() for routing.CONFIG_IP_MROUTE is an option to allow routing of IP packets that have several destination addresses. IN_DEV_MFORWARD is a macro defined in include/linux/inetdevice.h.

```
  40 #define IN_DEV_MFORWARD(in_dev)(ipv4_devconf.mc_forwarding
                        && (in_dev)->cnf.mc_forwarding)
```

```
1670                if (our
1671 #ifdef CONFIG_IP_MROUTE
1672                        || (!LOCAL_MCAST(daddr)
                                && IN_DEV_MFORWARD(in_dev))
1673 #endif
1674                        ){
1675                                read_unlock(&inetdev_lock);
1676                                return ip_route_input_mc(skb,
                                                daddr, saddr,
1677                                                tos, dev, our);
1678                }
1679          }
1680          read_unlock(&inetdev_lock);
1681          return -EINVAL;
1682     }
```

Reaching this point implies the packet was routeable neither through the routing cache nor as a multicast. The ip_route_input_slow() function must be called to try to route via the FIB.

```
1683     return ip_route_input_slow(skb, daddr, saddr, tos, dev);
1684 }
```