IP Layer Input Packet Processing

The receive functions of the IP layer include:

- IP header validation;
- IP header option processing;
- Reassembly of fragmented packets;
- Routing of the input packet to the proper transport.

However, to accomplish this limited mission a surprisingly large amount of processing is performed. A significant amount of this processing involves managing the layout and ownership of the `sk_buff`. This the primary concern of the `ip_rcv()` function which is defined in `net/ipv4/ip_input.c`.

```c
370 /*
371 *    Main IP Receive routine.
372 */
373 int ip_rcv(struct sk_buff *skb, struct net_device *dev,
374             struct packet_type *pt, struct ne
375             {*/
376     struct iphdr *iph;
377     u32 len;
```
When the interface is in promiscuous mode, any `sk_buff`, not directed to this host, is discarded without any processing.

The packet type is set to `PACKET_OTHERHOST` by `eth_type_trans()` in MAC layer processing

- if the packet was unicast and
- the destination MAC address was not the same as the MAC address carried by the `struct net_device` representing the interface upon which the packet arrived.

Operating an interface in promiscuous mode on a `true broadcast medium` is the only legitimate cause of this situation.

```c
    /* When the interface is in promisc. mode, drop all the crap
     * that it receives, do not try to analyse it.
     */
    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;
```

These are the counters that SNMP uses.

```c
    IP_INC_STATS_BH(IpInReceives);
```
Dealing with shared *skbs*

The `skb_share_check()` function determines if the *sk_buff* is shared. If so, the *sk_buff* is cloned, the original is freed, and a pointer to the clone is returned. If it is not shared, a pointer to the original is returned. If the *sk_buff* is shared, but the attempt to clone it fails, NULL is returned.

Buffers will be shared at this point if multiple handlers for a specific packet type have been registered. The fact that the shared *skb*’s become unshared at such a low level in the stack calls its usefulness into question in this application.

```c
386   if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
387      IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
388      goto out;
389   }
390```

IP Header Validation

Continuing in ip_rcv, the pskb_may_pull() is called. It ensures that IP header is entirely present in kmalloc’d area. It moves (pulls) the IP header from unmapped page fragments into the kmalloc’d area if required. We are not aware of any device drivers that create this ugly situation, but rectifying it requires an unbelievably tedious 10 pages of code which will not be examined here.

```c
391    if (!pskb_may_pull(skb, sizeof(struct iphdr)))
392       goto inhdr_error;
```

After ensuring that IP header is properly resident in the kmalloc’d area, IP header validation is performed. Validation includes ensuring that:

- the length of the datagram is at least the 20 byte length of an IP header;
- the IP version is 4;
- the header checksum is satisfactory;
- the packet length reported in the IP header is consistent with the length reported in the struct skbuff.

Verify that header length and version number are satisfactory.

```c
394    iph = skb->nh.iph;
407    if (iph->ihl < 5 || iph->version != 4)
408       goto inhdr_error;
```
Any IP header options must also be pulled into the kmalloc'd area of the sk_buff. This call is a return visit to the 10 pages of torture previously referenced.

    410    if (!pskb_may_pull(skb, iph->ihl*4))
    411           goto inhdr_error;

The IP header checksum is verified by ip_fast_csum. The header pointer must be reloaded here because the pskb_may_pull() operation may have rebuilt the whole skb.

    413    iph = skb->nh.iph;
    414
    415    if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
    416           goto inhdr_error;

If the length reported in iph->tot_len is greater than that reported in skb->len, or if it is less than the length of IP header, then there is a definite problem.

    418    len = ntohs(iph->tot_len);
    419    if (skb->len < len || len < (iph->ihl*4))
    420           goto inhdr_error;

However, it is legal for skb->len to exceed iph->tot_len. When this occurs, skb->len is adjusted downward to become consistent with iph->tot_len.

/* Our transport medium may have padded the buffer out. Now we know it is IP we can trim to the true length of the frame. Note this now means skb->len holds ntohs(iph->tot_len). */

    426    if (pskb_trim_rcsum(skb, len)) {
        427         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
        428         goto drop;
        429    }
Delivery to the netfilter

Finally, the packet is passed to the netfilter facility. The `okfn, ip_rcv_finish`, is called if the netfilter finds the packet to be acceptable.

```
431     /* Remove any debris in the socket control block */
432     memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));
433
434     return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
435                        ip_rcv_finish);
```

In the event of any error, the `sk_buff` is discarded.

```
436  inhdr_error:
437      IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
438  drop:
439      kfree_skb(skb);
440  out:
441      return NET_RX_DROP;
442  }
443  
```
Routing of Input Packets

The \texttt{ip_rcv_finish()} function is defined in \texttt{net/ipv4/ip_input.c} and is indirectly called from \texttt{ip_rcv()} as an \texttt{okfn()} passed through the \texttt{netfilter} mechanism. Its primary missions are

\begin{itemize}
  \item to call \texttt{ip_route_input()} which determines the next function to handle the \texttt{sk_buff},
  \item to compile any IP header options into the \texttt{sk_buff}’s control buffer, and
  \item if source routing options exist to process them.
\end{itemize}

331 static inline int ip_rcv_finish(struct sk_buff *skb)
332 {
333    struct iphdr *iph = skb->nh.iph;
334
The value of \texttt{skb->dst} will typically (always?) be NULL if the packet was received from the outside world. When this is the case, \texttt{ip_route_input()} is called to set \texttt{skb->dst} to a destination entry that describes the next course of action. The address of the next function to handle the \texttt{sk_buff} will be stored in \texttt{skb->dst->input} and is selected from.

\begin{verbatim}
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.
\end{verbatim}

339    if (skb->dst == NULL) {
340       int err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, skb->dev);
341       if (unlikely(err)) {
342          if (err == -EHOSTUNREACH)
343             IP_INC_STATS_BH(IPSTATS_MIB_INADDRERRORS);
344            goto drop;
345        }
346   }
347   }

ifdef CONFIG_NET_CLS_ROUTE
    if (unlikely(skb->dst->tclassid)) {
        struct ip_rt_acct *st = ip_rt_acct +
            256*smp_processor_id();
        u32 idx = skb->dst->tclassid;
        st[idx&0xFF].o_packets++;
        st[idx&0xFF].o_bytes+=skb->len;
        st[(idx>>16)&0xFF].i_packets++;
        st[(idx>>16)&0xFF].i_bytes+=skb->len;
    }
#endif

Processing of header options

A check is made for the presence of IP options. The only ones that impact processing here are
strict and loose source routing. This used to be done inline in a big ugly mess.

if (iph->ihl > 5 && ip_rcv_options(skb))
    goto drop;
return dst_input(skb);

drop:
    kfree_skb(skb);
return NET_RX_DROP;
}
Delivery to the next handler

After processing IP options, the `input` function of the destination entry is called. Recall that `skb->dst->input` was set by `ip_route_input` to point to one of the three functions below:

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

Therefore, under normal conditions, this triggers an upcall to either the forwarding or local delivery function. After delivery completes the return eventually leads to the end of the `softirq`.

```c
229 /* Input packet from network to transport. */
230 static inline int dst_input(struct sk_buff *skb)
231 {
232    int err;
233    for (;;) {
234       err = skb->dst->input(skb);
235       if (likely(err == 0))
236          return err;
237       /* Oh, Jamal... Seems, I will not forgive you this mess. :-) */
238       if (unlikely(err != NET_XMIT_BYPASS))
239          return err;
240    }
241 }
```
Processing of header options

278     static inline int ip_rcv_options(struct sk_buff *skb)
279     {
280         struct ip_options *opt;
281         struct iphdr *iph;
282         struct net_device *dev = skb->dev;
283
284         /* It looks as overkill, because not all
285          IP options require packet mangling.
286          But it is the easiest for now, especially taking
287          into account that combination of IP options
288          and running sniffer is extremely rare condition.
289     --ANK (980813)
291         */

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 and reallocates the data portion if the data portion is shared. The headroom of an skb is defined to be the difference between the data and head pointers. The sk_buff is reallocated if its headroom is inadequate 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 nothing will happen here — unless the buffer has been cloned. In that case it is copied and “de” cloned.

292         if (skb_cow(skb, skb_headroom(skb))) {
293             IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
294             goto drop;
295         }

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

296         iph = skb->nh.iph;
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.

```c
299    if (ip_options_compile(NULL, skb)) {
300       IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
301       goto drop;
302    }
```

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

```c
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.

```
303 304    opt = &(IPCB(skb)->opt);
305    if (unlikely(opt->srr)) {
306       struct in_device *in_dev = in_dev_get(dev);
```

The `struct in_device` is a "subclass" of the `net_device` that carries IP specific information.
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. The `struct in_device` carries the IP dependent attributes of a `net_device`. One such attribute is whether or not source routed IP datagrams are permitted on the device. The global flag `ipv4_devconf.accept_source_route` also can be used to completely disable source routes.

```c
if (in_dev) {
    if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
        if (IN_DEV_LOG_MARTIANS(in_dev) &&
            net_ratelimit())
            printk(KERN_INFO "source route option 
            "%u.%u.%u.%u - > %u.%u.%u.%u\n",
            NIPQUAD(iph->saddr),
            NIPQUAD(iph->daddr));
        in_dev_put(in_dev);
        goto drop;
    }
    else
        in_dev_put(in_dev);
}
```

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

```c
if (ip_options_rcv_srr(skb))
    goto drop;
}
return 0;
```

```c
drop:
return -1;
```
**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 source routing. If source routing is not allowed, the packet must be dropped.

```c
#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.

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

```c
588 int ip_options_rcv_srr(struct sk_buff *skb)
589 {
  struct ip_options *opt = &IPCB(skb)->opt;
  int srrspace, srrptr;
  u32 nexthop;
  struct iphdr *iph = skb->nh.iph;
  unsigned char *optptr = skb->nh.raw + opt->srr;
  struct rtable *rt = (struct rtable*)skb->dst;
  struct rtable *rt2;
  int err;

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

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!

```c
599   if (!opt->srr)
600     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, source routing is not permitted.

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

If strict source routing is specified, it is an error for the destination to be a gatewayed route (i.e. not RT_LOCAL) 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.

```c
  if (skb->pkt_type != PACKET_HOST)
    return -EINVAL
```

```c
  if (rt->rt_type == RTN_UNICAST) {
    if (!opt->is_strictroute)
      return 0;
    icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl(16<<24));
    return -EINVAL;
  }
```

If the route wasn't RTN_UNICAST it has to be RTN_LOCAL

```c
  if (rt->rt_type != RTN_LOCAL)
    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 this host is the end of the route. The for loop is handling the case in which a source route that contains multiple interfaces owned by this system!

```c
for (srrptr=optptr[2], srrspace = optptr[1];
    srrptr <= srrspace; srrptr += 4) {

Ensure that option length covers where next hop address should be:

```c
if (srrptr + 3 > srrspace) {
    icmp_send(skb, ICMP_PARAMETERPROB, 0,
              htonl((opt->srr+2)<<24));
    return -EINVAL;
}
```

Extract next hop address from optoin list and attempt to resolve route to it:

```c
memcpy(&nexthop, &optptr[srrptr-1], 4);
rt = (struct rtable*)skb->dst;
skb->dst = NULL;
err = ip_route_input(skb, nexthop, iph->saddr,
                     iph->tos, skb->dev);
rt2 = (struct rtable*)skb->dst;
```

The route type for the next hop must be either RTN_UNICAST or RTN_LOCAL.

```c
if (err || (rt2->rt_type != RTN_UNICAST &&
          rt2->rt_type != RTN_LOCAL)) {
    ip_rt_put(rt2);
    skb->dst = &rt->u.dst;
    return -EINVAL;
}
```
When route to next hop is of type RTN_UNICAST, the packet is to be forwarded. Note that
"skb->dst" now points to routing cache entry with next hop address as destination.

```c
630       if (rt2->rt_type != RTN_LOCAL)
631          break;
```

If the next hop is this host (route type is RTN_LOCAL), it is just copied into destination address
field of the IP packet.

```c
632       /* Superfast 8) loopback forward */
633       memcpy(&iph->daddr, &optptr[srrptr-1], 4);
634       opt->is_changed = 1;
635    
636    if (srrptr <= srrspace) {
637       opt->srr_is_hit = 1;
638       opt->is_changed = 1;
639    }
640    return 0;
641 }
```