Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 *4.2*

Linux/net/ipv4/ip_input.c

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              The Internet Protocol (IP) module.
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Donald Becker, <becker@super.org>
 *              Alan Cox, <alan@lxorguk.ukuu.org.uk>
 *              Richard Underwood
 *              Stefan Becker, <stefanb@yello.ping.de>
 *              Jorge Cwik, <jorge@laser.satlink.net>
 *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
 *
 *
 * Fixes:
 *              Alan Cox        :       Commented a couple of minor bits of surplus code
 *              Alan Cox        :       Undefining IP_FORWARD doesn't include the code
 *                                      (just stops a compiler warning).
 *              Alan Cox        :       Frames with >=MAX_ROUTE record routes, strict routes or loose routes
 *                                      are junked rather than corrupting things.
 *              Alan Cox        :       Frames to bad broadcast subnets are dumped
 *                                      We used to process them non broadcast and
 *                                      boy could that cause havoc.
 *              Alan Cox        :       ip_forward sets the free flag on the
 *                                      new frame it queues. Still crap because
 *                                      it copies the frame but at least it
 *                                      doesn't eat memory too.
 *              Alan Cox        :       Generic queue code and memory fixes.
 *              Fred Van Kempen :       IP fragment support (borrowed from NET2E)
 *              Gerhard Koerting:       Forward fragmented frames correctly.
 *              Gerhard Koerting:       Fixes to my fix of the above 8-).
 *              Gerhard Koerting:       IP interface addressing fix.
 *              Linus Torvalds  :       More robustness checks
 *              Alan Cox        :       Even more checks: Still not as robust as it ought to be
 *              Alan Cox        :       Save IP header pointer for later
 *              Alan Cox        :       ip option setting
 *              Alan Cox        :       Use ip_tos/ip_ttl settings
 *              Alan Cox        :       Fragmentation bogosity removed
 *                                      (Thanks to Mark.Bush@prg.ox.ac.uk)
 *              Dmitry Gorodchanin :    Send of a raw packet crash fix.
 *              Alan Cox        :       Silly ip bug when an overlength
 *                                      fragment turns up. Now frees the
 *                                      queue.
 *              Linus Torvalds/ :       Memory leakage on fragmentation
 *              Alan Cox        :       handling.
 *              Gerhard Koerting:       Forwarding uses IP priority hints
 *              Teemu Rantanen  :       Fragment problems.
 *              Alan Cox        :       General cleanup, comments and reformat
 *              Alan Cox        :       SNMP statistics
 *              Alan Cox        :       BSD address rule semantics. Also see
 *                                      UDP as there is a nasty checksum issue
 *                                      if you do things the wrong way.
 *              Alan Cox        :       Always defrag, moved IP_FORWARD to the config.in file
 *              Alan Cox        :       IP options adjust sk->priority.
 *              Pedro Roque     :       Fix mtu/length error in ip_forward.
 *              Alan Cox        :       Avoid ip_chk_addr when possible.
 *      Richard Underwood       :       IP multicasting.
 *              Alan Cox        :       Cleaned up multicast handlers.
 *              Alan Cox        :       RAW sockets demultiplex in the BSD style.
 *              Gunther Mayer   :       Fix the SNMP reporting typo
 *              Alan Cox        :       Always in group 224.0.0.1
 *      Pauline Middelink       :       Fast ip_checksum update when forwarding
 *                                      Masquerading support.
 *              Alan Cox        :       Multicast loopback error for 224.0.0.1
 *              Alan Cox        :       IP_MULTICAST_LOOP option.
 *              Alan Cox        :       Use notifiers.
 *              Bjorn Ekwall    :       Removed ip_csum (from slhc.c too)
 *              Bjorn Ekwall    :       Moved ip_fast_csum to ip.h (inline!)
 *              Stefan Becker   :       Send out ICMP HOST REDIRECT
 *      Arnt Gulbrandsen        :       ip_build_xmit
 *              Alan Cox        :       Per socket routing cache
 *              Alan Cox        :       Fixed routing cache, added header cache.
 *              Alan Cox        :       Loopback didn't work right in original ip_build_xmit - fixed it.
 *              Alan Cox        :       Only send ICMP_REDIRECT if src/dest are the same net.
 *              Alan Cox        :       Incoming IP option handling.
 *              Alan Cox        :       Set saddr on raw output frames as per BSD.
 *              Alan Cox        :       Stopped broadcast source route explosions.
 *              Alan Cox        :       Can disable source routing
 *              Takeshi Sone    :       Masquerading didn't work.
 *      Dave Bonn,Alan Cox      :       Faster IP forwarding whenever possible.
 *              Alan Cox        :       Memory leaks, tramples, misc debugging.
 *              Alan Cox        :       Fixed multicast (by popular demand 8))
 *              Alan Cox        :       Fixed forwarding (by even more popular demand 8))
 *              Alan Cox        :       Fixed SNMP statistics [I think]
 *      Gerhard Koerting        :       IP fragmentation forwarding fix
 *              Alan Cox        :       Device lock against page fault.
 *              Alan Cox        :       IP_HDRINCL facility.
 *      Werner Almesberger      :       Zero fragment bug
 *              Alan Cox        :       RAW IP frame length bug
 *              Alan Cox        :       Outgoing firewall on build_xmit
 *              A.N.Kuznetsov   :       IP_OPTIONS support throughout the kernel
 *              Alan Cox        :       Multicast routing hooks
 *              Jos Vos         :       Do accounting *before* call_in_firewall
 *      Willy Konynenberg       :       Transparent proxying support
 *
 *
 *
 * To Fix:
 *              IP fragmentation wants rewriting cleanly. The RFC815 algorithm is much more efficient
 *              and could be made very efficient with the addition of some virtual memory hacks to permit
 *              the allocation of a buffer that can then be 'grown' by twiddling page tables.
 *              Output fragmentation wants updating along with the buffer management to use a single
 *              interleaved copy algorithm so that fragmenting has a one copy overhead. Actual packet
 *              output should probably do its own fragmentation at the UDP/RAW layer. TCP shouldn't cause
 *              fragmentation anyway.
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */

#define pr_fmt(fmt) "IPv4: " fmt

#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/slab.h>

#include <linux/net.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/inetdevice.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>

#include <net/snmp.h>
#include <net/ip.h>
#include <net/protocol.h>
```

```
#include <net/route.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/arp.h>
#include <net/icmp.h>
#include <net/raw.h>
#include <net/checksum.h>
#include <net/inet_ecn.h>
#include <linux/netfilter_ipv4.h>
#include <net/xfrm.h>
#include <linux/mroute.h>
#include <linux/netlink.h>

/*
 *      Process Router Attention IP option (RFC 2113)
 */
bool ip_call_ra_chain(struct sk_buff *skb)
{
        struct ip_ra_chain *ra;
        u8 protocol = ip_hdr(skb)->protocol;
        struct sock *last = NULL;
        struct net_device *dev = skb->dev;

        for (ra = rcu_dereference(ip_ra_chain); ra; ra = rcu_dereference(ra->next)) {
                struct sock *sk = ra->sk;

                /* If socket is bound to an interface, only report
                 * the packet if it came  from that interface.
                 */
                if (sk && inet_sk(sk)->inet_num == protocol &&
                    (!sk->sk_bound_dev_if ||
                     sk->sk_bound_dev_if == dev->ifindex) &&
                    net_eq(sock_net(sk), dev_net(dev))) {
                        if (ip_is_fragment(ip_hdr(skb))) {
                                if (ip_defrag(skb, IP_DEFRAG_CALL_RA_CHAIN))
                                        return true;
                        }
                        if (last) {
                                struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
                                if (skb2)
                                        raw_rcv(last, skb2);
                        }
                        last = sk;
                }
        }

        if (last) {
                raw_rcv(last, skb);
                return true;
        }
        return false;
}

static int ip_local_deliver_finish(struct sock *sk, struct sk_buff *skb)
{
        struct net *net = dev_net(skb->dev);

        __skb_pull(skb, skb_network_header_len(skb));

        rcu_read_lock();
        {
                int protocol = ip_hdr(skb)->protocol;
                const struct net_protocol *ipprot;
                int raw;

        resubmit:
                raw = raw_local_deliver(skb, protocol);

                ipprot = rcu_dereference(inet_protos[protocol]);
                if (ipprot) {
                        int ret;

                        if (!ipprot->no_policy) {
                                if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                                        kfree_skb(skb);
                                        goto out;
                                }
                                nf_reset(skb);
                        }
                        ret = ipprot->handler(skb);
                        if (ret < 0) {
                                protocol = -ret;
                                goto resubmit;
                        }
                        IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
                } else {
                        if (!raw) {
                                if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                                        IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
                                        icmp_send(skb, ICMP_DEST_UNREACH,
                                                  ICMP_PROT_UNREACH, 0);
                                }
                                kfree_skb(skb);
                        } else {
                                IP_INC_STATS_BH(net, IPSTATS_MIB_INDELIVERS);
                                consume_skb(skb);
                        }
                }
        }
 out:
        rcu_read_unlock();

        return 0;
}

/*
 *      Deliver IP Packets to the higher protocol layers.
 */
int ip_local_deliver(struct sk_buff *skb)
{
        /*
         *      Reassemble IP fragments.
         */

        if (ip_is_fragment(ip_hdr(skb))) {
                if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
                        return 0;
        }

        return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, NULL, skb,
                       skb->dev, NULL,
                       ip_local_deliver_finish);
}

static inline bool ip_rcv_options(struct sk_buff *skb)
{
        struct ip_options *opt;
        const struct iphdr *iph;
        struct net_device *dev = skb->dev;

        /* It looks as overkill, because not all
           IP options require packet mangling.
           But it is the easiest for now, especially taking
           into account that combination of IP options
           and running sniffer is extremely rare condition.
                                              --ANK (980813)
        */
        if (skb_cow(skb, skb_headroom(skb))) {
                IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
                goto drop;
        }

        iph = ip_hdr(skb);
        opt = &(IPCB(skb)->opt);
        opt->optlen = iph->ihl*4 - sizeof(struct iphdr);

        if (ip_options_compile(dev_net(dev), opt, skb)) {
                IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
                goto drop;
        }

        if (unlikely(opt->srr)) {
                struct in_device *in_dev = __in_dev_get_rcu(dev);

                if (in_dev) {
                        if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                                if (IN_DEV_LOG_MARTIANS(in_dev))
                                        net_info_ratelimited("source route option %pI4 -> %pI4\n",
                                                             &iph->saddr,
                                                             &iph->daddr);
                                goto drop;
                        }
                }

                if (ip_options_rcv_srr(skb))
                        goto drop;
        }

        return false;
drop:
        return true;
}

int sysctl_ip_early_demux __read_mostly = 1;
EXPORT_SYMBOL(sysctl_ip_early_demux);

static int ip_rcv_finish(struct sock *sk, struct sk_buff *skb)
{
        const struct iphdr *iph = ip_hdr(skb);
        struct rtable *rt;

        if (sysctl_ip_early_demux && !skb_dst(skb) && !skb->sk) {
                const struct net_protocol *ipprot;
                int protocol = iph->protocol;

                ipprot = rcu_dereference(inet_protos[protocol]);
                if (ipprot && ipprot->early_demux) {
                        ipprot->early_demux(skb);
                        /* must reload iph, skb->head might have changed */
                        iph = ip_hdr(skb);
                }
        }

        /*
         *      Initialise the virtual path cache for the packet. It describes
         *      how the packet travels inside Linux networking.
         */
        if (!skb_dst(skb)) {
                int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
                                               iph->tos, skb->dev);
                if (unlikely(err)) {
                        if (err == -EXDEV)
                                NET_INC_STATS_BH(dev_net(skb->dev),
                                                 LINUX_MIB_IPRPFILTER);
                        goto drop;
                }
        }

#ifdef CONFIG_IP_ROUTE_CLASSID
        if (unlikely(skb_dst(skb)->tclassid)) {
                struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);
                u32 idx = skb_dst(skb)->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

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

        rt = skb_rtable(skb);
        if (rt->rt_type == RTN_MULTICAST) {
                IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
                                skb->len);
        } else if (rt->rt_type == RTN_BROADCAST)
                IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
                                skb->len);

        return dst_input(skb);

drop:
        kfree_skb(skb);
        return NET_RX_DROP;
}

/*
 *      Main IP Receive routine.
 */
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
{
        const struct iphdr *iph;
        u32 len;

        /* 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;


        IP_UPD_PO_STATS_BH(dev_net(dev), IPSTATS_MIB_IN, skb->len);

        skb = skb_share_check(skb, GFP_ATOMIC);
        if (!skb) {
                IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
                goto out;
        }

        if (!pskb_may_pull(skb, sizeof(struct iphdr)))
                goto inhdr_error;

        iph = ip_hdr(skb);

        /*
         *      RFC1122: 3.2.1.2 MUST silently discard any IP frame that fails the checksum.
         *
         *      Is the datagram acceptable?
         *
         *      1.      Length at least the size of an ip header
         *      2.      Version of 4
         *      3.      Checksums correctly. [Speed optimisation for later, skip loopback checksums]
         *      4.      Doesn't have a bogus length
         */

        if (iph->ihl < 5 || iph->version != 4)
                goto inhdr_error;

        BUILD_BUG_ON(IPSTATS_MIB_ECT1PKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_ECT_1);
        BUILD_BUG_ON(IPSTATS_MIB_ECT0PKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_ECT_0);
        BUILD_BUG_ON(IPSTATS_MIB_CEPKTS != IPSTATS_MIB_NOECTPKTS + INET_ECN_CE);
        IP_ADD_STATS_BH(dev_net(dev),
                        IPSTATS_MIB_NOECTPKTS + (iph->tos & INET_ECN_MASK),
                        max_t(unsigned short, 1, skb_shinfo(skb)->gso_segs));

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

        iph = ip_hdr(skb);

        if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
                goto csum_error;

        len = ntohs(iph->tot_len);
        if (skb->len < len) {
                IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INTRUNCATEDPKTS);
                goto drop;
        } else if (len < (iph->ihl*4))
                goto inhdr_error;

        /* 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).
         */
        if (pskb_trim_rcsum(skb, len)) {
                IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INDISCARDS);
                goto drop;
        }

        skb->transport_header = skb->network_header + iph->ihl*4;

        /* Remove any debris in the socket control block */
        memset(IPCB(skb), 0, sizeof(struct inet_skb_parm));

        /* Must drop socket now because of tproxy. */
        skb_orphan(skb);

        return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, NULL, skb,
                       dev, NULL,
                       ip_rcv_finish);

csum_error:
        IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_CSUMERRORS);
inhdr_error:
        IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
drop:
        kfree_skb(skb);
out:
        return NET_RX_DROP;
}

```