# Linux Cross Reference

## Free Electrons

## Embedded Linux Experts



# Linux/net/ipv4/tcp_ipv4.c

```
 1 /*
 2  * INET         An implementation of the TCP/IP protocol suite for the LINUX
 3  *              operating system.  INET is implemented using the  BSD Socket
 4  *              interface as the means of communication with the user level.
 5  *
 6  *              Implementation of the Transmission Control Protocol(TCP).
 7  *
 8  *              IPv4 specific functions
 9  *
10  *
11  *              code split from:
12  *              linux/ipv4/tcp.c
13  *              linux/ipv4/tcp_input.c
14  *              linux/ipv4/tcp_output.c
15  *
16  *              See tcp.c for author information
17  *
18  *      This program is free software; you can redistribute it and/or
19  *      modify it under the terms of the GNU General Public License
20  *      as published by the Free Software Foundation; either version
21  *      2 of the License, or (at your option) any later version.
22  */
23
24 /*
25  * Changes:
26  *              David S. Miller :       New socket lookup architecture.
27  *                                      This code is dedicated to John Dyson.
28  *              David S. Miller :       Change semantics of established hash,
29  *                                      half is devoted to TIME_WAIT sockets
30  *                                      and the rest go in the other half.
31  *              Andi Kleen :            Add support for syncookies and fixed
32  *                                      some bugs: ip options weren't passed to
33  *                                      the TCP layer, missed a check for an
34  *                                      ACK bit.
35  *              Andi Kleen :            Implemented fast path mtu discovery.
36  *                                      Fixed many serious bugs in the
37  *                                      request_sock handling and moved
38  *                                      most of it into the af independent code.
39  *                                      Added tail drop and some other bugfixes.
40  *                                      Added new listen semantics.
41  *              Mike McLagan    :       Routing by source
42  *      Juan Jose Ciarlante:            ip_dynaddr bits
43  *              Andi Kleen:             various fixes.
44  *      Vitaly E. Lavrov        :       Transparent proxy revived after year
45  *                                      coma.
46  *      Andi Kleen              :       Fix new listen.
47  *      Andi Kleen              :       Fix accept error reporting.
48  *      YOSHIFUJI Hideaki @USAGI and:   Support IPV6_V6ONLY socket option, which
49  *      Alexey Kuznetsov                allow both IPv4 and IPv6 sockets to bind
50  *                                      a single port at the same time.
51  */
```

```
 52
 53 #define pr_fmt(fmt) "TCP: " fmt
 54
 55 #include <linux/bottom_half.h>
 56 #include <linux/types.h>
 57 #include <linux/fcntl.h>
 58 #include <linux/module.h>
 59 #include <linux/random.h>
 60 #include <linux/cache.h>
 61 #include <linux/jhash.h>
 62 #include <linux/init.h>
 63 #include <linux/times.h>
 64 #include <linux/slab.h>
 65
 66 #include <net/net_namespace.h>
 67 #include <net/icmp.h>
 68 #include <net/inet_hashtables.h>
 69 #include <net/tcp.h>
 70 #include <net/transp_v6.h>
 71 #include <net/ipv6.h>
 72 #include <net/inet_common.h>
 73 #include <net/timewait_sock.h>
 74 #include <net/xfrm.h>
 75 #include <net/netdma.h>
 76 #include <net/secure_seq.h>
 77 #include <net/tcp_memcontrol.h>
 78 #include <net/busy_poll.h>
 79
 80 #include <linux/inet.h>
 81 #include <linux/ipv6.h>
 82 #include <linux/stddef.h>
 83 #include <linux/proc_fs.h>
 84 #include <linux/seq_file.h>
 85
 86 #include <linux/crypto.h>
 87 #include <linux/scatterlist.h>
 88
 89 int sysctl_tcp_tw_reuse __read_mostly;
 90 int sysctl_tcp_low_latency __read_mostly;
 91 EXPORT_SYMBOL(sysctl_tcp_low_latency);
 92
 93
 94 #ifdef CONFIG_TCP_MD5SIG
 95 static int tcp_v4_md5_hash_hdr(char *md5_hash, const struct tcp_md5sig_key *key,
 96                                __be32 daddr, __be32 saddr, const struct tcphdr *th);
 97 #endif
 98
 99 struct inet_hashinfo tcp_hashinfo;
100 EXPORT_SYMBOL(tcp_hashinfo);
101
102 static  __u32 tcp_v4_init_sequence(const struct sk_buff *skb)
103 {
104         return secure_tcp_sequence_number(ip_hdr(skb)->daddr,
105                                           ip_hdr(skb)->saddr,
106                                           tcp_hdr(skb)->dest,
107                                           tcp_hdr(skb)->source);
108 }
109
110 int tcp_twsk_unique(struct sock *sk, struct sock *sktw, void *twp)
111 {
112         const struct tcp_timewait_sock *tcptw = tcp_twsk(sktw);
113         struct tcp_sock *tp = tcp_sk(sk);
114
115         /* With PAWS, it is safe from the viewpoint
116            of data integrity. Even without PAWS it is safe provided sequence
117            spaces do not overlap i.e. at data rates <= 80Mbit/sec.
118
119            Actually, the idea is close to VJ's one, only timestamp cache is
120            held not per host, but per port pair and TW bucket is used as state
121            holder.
122
123            If TW bucket has been already destroyed we fall back to VJ's scheme
```

```
124             and use initial timestamp retrieved from peer table.
125          */
126         if (tcptw->tw_ts_recent_stamp &&
127             (twp == NULL || (sysctl_tcp_tw_reuse &&
128                              get_seconds() - tcptw->tw_ts_recent_stamp > 1))) {
129                 tp->write_seq = tcptw->tw_snd_nxt + 65535 + 2;
130                 if (tp->write_seq == 0)
131                         tp->write_seq = 1;
132                 tp->rx_opt.ts_recent       = tcptw->tw_ts_recent;
133                 tp->rx_opt.ts_recent_stamp = tcptw->tw_ts_recent_stamp;
134                 sock_hold(sktw);
135                 return 1;
136         }
137
138         return 0;
139 }
140 EXPORT_SYMBOL_GPL(tcp_twsk_unique);
141
142 /* This will initiate an outgoing connection. */
143 int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
144 {
145         struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
146         struct inet_sock *inet = inet_sk(sk);
147         struct tcp_sock *tp = tcp_sk(sk);
148         __be16 orig_sport, orig_dport;
149         __be32 daddr, nexthop;
150         struct flowi4 *fl4;
151         struct rtable *rt;
152         int err;
153         struct ip_options_rcu *inet_opt;
154
155         if (addr_len < sizeof(struct sockaddr_in))
156                 return -EINVAL;
157
158         if (usin->sin_family != AF_INET)
159                 return -EAFNOSUPPORT;
160
161         nexthop = daddr = usin->sin_addr.s_addr;
162         inet_opt = rcu_dereference_protected(inet->inet_opt,
163                                              sock_owned_by_user(sk));
164         if (inet_opt && inet_opt->opt.srr) {
165                 if (!daddr)
166                         return -EINVAL;
167                 nexthop = inet_opt->opt.faddr;
168         }
169
170         orig_sport = inet->inet_sport;
171         orig_dport = usin->sin_port;
172         fl4 = &inet->cork.fl.u.ip4;
173         rt = ip_route_connect(fl4, nexthop, inet->inet_saddr,
174                               RT_CONN_FLAGS(sk), sk->sk_bound_dev_if,
175                               IPPROTO_TCP,
176                               orig_sport, orig_dport, sk);
177         if (IS_ERR(rt)) {
178                 err = PTR_ERR(rt);
179                 if (err == -ENETUNREACH)
180                         IP_INC_STATS(sock_net(sk), IPSTATS_MIB_OUTNOROUTES);
181                 return err;
182         }
183
184         if (rt->rt_flags & (RTCF_MULTICAST | RTCF_BROADCAST)) {
185                 ip_rt_put(rt);
186                 return -ENETUNREACH;
187         }
188
189         if (!inet_opt || !inet_opt->opt.srr)
190                 daddr = fl4->daddr;
191
192         if (!inet->inet_saddr)
193                 inet->inet_saddr = fl4->saddr;
194         inet->inet_rcv_saddr = inet->inet_saddr;
195
```

```
196         if (tp->rx_opt.ts_recent_stamp && inet->inet_daddr != daddr) {
197                 /* Reset inherited state */
198                 tp->rx_opt.ts_recent       = 0;
199                 tp->rx_opt.ts_recent_stamp = 0;
200                 if (likely(!tp->repair))
201                         tp->write_seq      = 0;
202         }
203
204         if (tcp_death_row.sysctl_tw_recycle &&
205             !tp->rx_opt.ts_recent_stamp && fl4->daddr == daddr)
206                 tcp_fetch_timewait_stamp(sk, &rt->dst);
207
208         inet->inet_dport = usin->sin_port;
209         inet->inet_daddr = daddr;
210
211         inet_set_txhash(sk);
212
213         inet_csk(sk)->icsk_ext_hdr_len = 0;
214         if (inet_opt)
215                 inet_csk(sk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
216
217         tp->rx_opt.mss_clamp = TCP_MSS_DEFAULT;
218
219         /* Socket identity is still unknown (sport may be zero).
220          * However we set state to SYN-SENT and not releasing socket
221          * lock select source port, enter ourselves into the hash tables and
222          * complete initialization after this.
223          */
224         tcp_set_state(sk, TCP_SYN_SENT);
225         err = inet_hash_connect(&tcp_death_row, sk);
226         if (err)
227                 goto failure;
228
229         rt = ip_route_newports(fl4, rt, orig_sport, orig_dport,
230                                inet->inet_sport, inet->inet_dport, sk);
231         if (IS_ERR(rt)) {
232                 err = PTR_ERR(rt);
233                 rt = NULL;
234                 goto failure;
235         }
236         /* OK, now commit destination to socket.  */
237         sk->sk_gso_type = SKB_GSO_TCPV4;
238         sk_setup_caps(sk, &rt->dst);
239
240         if (!tp->write_seq && likely(!tp->repair))
241                 tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,
242                                                            inet->inet_daddr,
243                                                            inet->inet_sport,
244                                                            usin->sin_port);
245
246         inet->inet_id = tp->write_seq ^ jiffies;
247
248         err = tcp_connect(sk);
249
250         rt = NULL;
251         if (err)
252                 goto failure;
253
254         return 0;
255
256 failure:
257         /*
258          * This unhashes the socket and releases the local port,
259          * if necessary.
260          */
261         tcp_set_state(sk, TCP_CLOSE);
262         ip_rt_put(rt);
263         sk->sk_route_caps = 0;
264         inet->inet_dport = 0;
265         return err;
266 }
267 EXPORT_SYMBOL(tcp_v4_connect);
```

```
268
269 /*
270  * This routine reacts to ICMP_FRAG_NEEDED mtu indications as defined in RFC1191.
271  * It can be called through tcp_release_cb() if socket was owned by user
272  * at the time tcp_v4_err() was called to handle ICMP message.
273  */
274 void tcp_v4_mtu_reduced(struct sock *sk)
275 {
276         struct dst_entry *dst;
277         struct inet_sock *inet = inet_sk(sk);
278         u32 mtu = tcp_sk(sk)->mtu_info;
279
280         dst = inet_csk_update_pmtu(sk, mtu);
281         if (!dst)
282                 return;
283
284         /* Something is about to be wrong... Remember soft error
285          * for the case, if this connection will not able to recover.
286          */
287         if (mtu < dst_mtu(dst) && ip_dont_fragment(sk, dst))
288                 sk->sk_err_soft = EMSGSIZE;
289
290         mtu = dst_mtu(dst);
291
292         if (inet->pmtudisc != IP_PMTUDISC_DONT &&
293             ip_sk_accept_pmtu(sk) &&
294             inet_csk(sk)->icsk_pmtu_cookie > mtu) {
295                 tcp_sync_mss(sk, mtu);
296
297                 /* Resend the TCP packet because it's
298                  * clear that the old packet has been
299                  * dropped. This is the new "fast" path mtu
300                  * discovery.
301                  */
302                 tcp_simple_retransmit(sk);
303         } /* else let the usual retransmit timer handle it */
304 }
305 EXPORT_SYMBOL(tcp_v4_mtu_reduced);
306
307 static void do_redirect(struct sk_buff *skb, struct sock *sk)
308 {
309         struct dst_entry *dst = __sk_dst_check(sk, 0);
310
311         if (dst)
312                 dst->ops->redirect(dst, sk, skb);
313 }
314
315 /*
316  * This routine is called by the ICMP module when it gets some
317  * sort of error condition.  If err < 0 then the socket should
318  * be closed and the error returned to the user.  If err > 0
319  * it's just the icmp type << 8 | icmp code.  After adjustment
320  * header points to the first 8 bytes of the tcp header.  We need
321  * to find the appropriate port.
322  *
323  * The locking strategy used here is very "optimistic". When
324  * someone else accesses the socket the ICMP is just dropped
325  * and for some paths there is no check at all.
326  * A more general error queue to queue errors for later handling
327  * is probably better.
328  *
329  */
330
331 void tcp_v4_err(struct sk_buff *icmp_skb, u32 info)
332 {
333         const struct iphdr *iph = (const struct iphdr *)icmp_skb->data;
334         struct tcphdr *th = (struct tcphdr *)(icmp_skb->data + (iph->ihl << 2));
335         struct inet_connection_sock *icsk;
336         struct tcp_sock *tp;
337         struct inet_sock *inet;
338         const int type = icmp_hdr(icmp_skb)->type;
339         const int code = icmp_hdr(icmp_skb)->code;
```

```
340         struct sock *sk;
341         struct sk_buff *skb;
342         struct request_sock *fastopen;
343         __u32 seq, snd_una;
344         __u32 remaining;
345         int err;
346         struct net *net = dev_net(icmp_skb->dev);
347
348         sk = inet_lookup(net, &tcp_hashinfo, iph->daddr, th->dest,
349                          iph->saddr, th->source, inet_iif(icmp_skb));
350         if (!sk) {
351                 ICMP_INC_STATS_BH(net, ICMP_MIB_INERRORS);
352                 return;
353         }
354         if (sk->sk_state == TCP_TIME_WAIT) {
355                 inet_twsk_put(inet_twsk(sk));
356                 return;
357         }
358
359         bh_lock_sock(sk);
360         /* If too many ICMPs get dropped on busy
361          * servers this needs to be solved differently.
362          * We do take care of PMTU discovery (RFC1191) special case :
363          * we can receive locally generated ICMP messages while socket is held.
364          */
365         if (sock_owned_by_user(sk)) {
366                 if (!(type == ICMP_DEST_UNREACH && code == ICMP_FRAG_NEEDED))
367                         NET_INC_STATS_BH(net, LINUX_MIB_LOCKDROPPEDICMPS);
368         }
369         if (sk->sk_state == TCP_CLOSE)
370                 goto out;
371
372         if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {
373                 NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);
374                 goto out;
375         }
376
377         icsk = inet_csk(sk);
378         tp = tcp_sk(sk);
379         seq = ntohl(th->seq);
380         /* XXX (TFO) - tp->snd_una should be ISN (tcp_create_openreq_child() */
381         fastopen = tp->fastopen_rsk;
382         snd_una = fastopen ? tcp_rsk(fastopen)->snt_isn : tp->snd_una;
383         if (sk->sk_state != TCP_LISTEN &&
384             !between(seq, snd_una, tp->snd_nxt)) {
385                 NET_INC_STATS_BH(net, LINUX_MIB_OUTOFWINDOWICMPS);
386                 goto out;
387         }
388
389         switch (type) {
390         case ICMP_REDIRECT:
391                 do_redirect(icmp_skb, sk);
392                 goto out;
393         case ICMP_SOURCE_QUENCH:
394                 /* Just silently ignore these. */
395                 goto out;
396         case ICMP_PARAMETERPROB:
397                 err = EPROTO;
398                 break;
399         case ICMP_DEST_UNREACH:
400                 if (code > NR_ICMP_UNREACH)
401                         goto out;
402
403                 if (code == ICMP_FRAG_NEEDED) { /* PMTU discovery (RFC1191) */
404                         /* We are not interested in TCP_LISTEN and open_requests
405                          * (SYN-ACKs send out by Linux are always <576bytes so
406                          * they should go through unfragmented).
407                          */
408                         if (sk->sk_state == TCP_LISTEN)
409                                 goto out;
410
411                         tp->mtu_info = info;
```

```
412                         if (!sock_owned_by_user(sk)) {
413                                 tcp_v4_mtu_reduced(sk);
414                         } else {
415                                 if (!test_and_set_bit(TCP_MTU_REDUCED_DEFERRED, &tp->tsq_flags))
416                                         sock_hold(sk);
417                         }
418                         goto out;
419                 }
420
421                 err = icmp_err_convert[code].errno;
422                 /* check if icmp_skb allows revert of backoff
423                  * (see draft-zimmermann-tcp-lcd) */
424                 if (code != ICMP_NET_UNREACH && code != ICMP_HOST_UNREACH)
425                         break;
426                 if (seq != tp->snd_una  || !icsk->icsk_retransmits ||
427                     !icsk->icsk_backoff || fastopen)
428                         break;
429
430                 if (sock_owned_by_user(sk))
431                         break;
432
433                 icsk->icsk_backoff--;
434                 inet_csk(sk)->icsk_rto = (tp->srtt_us ? __tcp_set_rto(tp) :
435                         TCP_TIMEOUT_INIT) << icsk->icsk_backoff;
436                 tcp_bound_rto(sk);
437
438                 skb = tcp_write_queue_head(sk);
439                 BUG_ON(!skb);
440
441                 remaining = icsk->icsk_rto - min(icsk->icsk_rto,
442                                 tcp_time_stamp - TCP_SKB_CB(skb)->when);
443
444                 if (remaining) {
445                         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
446                                                   remaining, TCP_RTO_MAX);
447                 } else {
448                         /* RTO revert clocked out retransmission.
449                          * Will retransmit now */
450                         tcp_retransmit_timer(sk);
451                 }
452
453                 break;
454         case ICMP_TIME_EXCEEDED:
455                 err = EHOSTUNREACH;
456                 break;
457         default:
458                 goto out;
459         }
460
461         switch (sk->sk_state) {
462                 struct request_sock *req, **prev;
463         case TCP_LISTEN:
464                 if (sock_owned_by_user(sk))
465                         goto out;
466
467                 req = inet_csk_search_req(sk, &prev, th->dest,
468                                           iph->daddr, iph->saddr);
469                 if (!req)
470                         goto out;
471
472                 /* ICMPs are not backlogged, hence we cannot get
473                    an established socket here.
474                  */
475                 WARN_ON(req->sk);
476
477                 if (seq != tcp_rsk(req)->snt_isn) {
478                         NET_INC_STATS_BH(net, LINUX_MIB_OUTOFWINDOWICMPS);
479                         goto out;
480                 }
481
482                 /*
483                  * Still in SYN_RECV, just remove it silently.
```

```
484                  * There is no good way to pass the error to the newly
485                  * created socket, and POSIX does not want network
486                  * errors returned from accept().
487                  */
488                 inet_csk_reqsk_queue_drop(sk, req, prev);
489                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
490                 goto out;
491
492         case TCP_SYN_SENT:
493         case TCP_SYN_RECV:
494                 /* Only in fast or simultaneous open. If a fast open socket is
495                  * is already accepted it is treated as a connected one below.
496                  */
497                 if (fastopen && fastopen->sk == NULL)
498                         break;
499
500                 if (!sock_owned_by_user(sk)) {
501                         sk->sk_err = err;
502
503                         sk->sk_error_report(sk);
504
505                         tcp_done(sk);
506                 } else {
507                         sk->sk_err_soft = err;
508                 }
509                 goto out;
510         }
511
512         /* If we've already connected we will keep trying
513          * until we time out, or the user gives up.
514          *
515          * rfc1122 4.2.3.9 allows to consider as hard errors
516          * only PROTO_UNREACH and PORT_UNREACH (well, FRAG_FAILED too,
517          * but it is obsoleted by pmtu discovery).
518          *
519          * Note, that in modern internet, where routing is unreliable
520          * and in each dark corner broken firewalls sit, sending random
521          * errors ordered by their masters even this two messages finally lose
522          * their original sense (even Linux sends invalid PORT_UNREACHs)
523          *
524          * Now we are in compliance with RFCs.
525          *                                                      --ANK (980905)
526          */
527
528         inet = inet_sk(sk);
529         if (!sock_owned_by_user(sk) && inet->recverr) {
530                 sk->sk_err = err;
531                 sk->sk_error_report(sk);
532         } else  { /* Only an error on timeout */
533                 sk->sk_err_soft = err;
534         }
535
536 out:
537         bh_unlock_sock(sk);
538         sock_put(sk);
539 }
540
541 void __tcp_v4_send_check(struct sk_buff *skb, __be32 saddr, __be32 daddr)
542 {
543         struct tcphdr *th = tcp_hdr(skb);
544
545         if (skb->ip_summed == CHECKSUM_PARTIAL) {
546                 th->check = ~tcp_v4_check(skb->len, saddr, daddr, 0);
547                 skb->csum_start = skb_transport_header(skb) - skb->head;
548                 skb->csum_offset = offsetof(struct tcphdr, check);
549         } else {
550                 th->check = tcp_v4_check(skb->len, saddr, daddr,
551                                          csum_partial(th,
552                                                       th->doff << 2,
553                                                       skb->csum));
554         }
555 }
```

```
556
557 /* This routine computes an IPv4 TCP checksum. */
558 void tcp_v4_send_check(struct sock *sk, struct sk_buff *skb)
559 {
560         const struct inet_sock *inet = inet_sk(sk);
561
562         __tcp_v4_send_check(skb, inet->inet_saddr, inet->inet_daddr);
563 }
564 EXPORT_SYMBOL(tcp_v4_send_check);
565
566 /*
567  *      This routine will send an RST to the other tcp.
568  *
569  *      Someone asks: why I NEVER use socket parameters (TOS, TTL etc.)
570  *                    for reset.
571  *      Answer: if a packet caused RST, it is not for a socket
572  *              existing in our system, if it is matched to a socket,
573  *              it is just duplicate segment or bug in other side's TCP.
574  *              So that we build reply only basing on parameters
575  *              arrived with segment.
576  *      Exception: precedence violation. We do not implement it in any case.
577  */
578
579 static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
580 {
581         const struct tcphdr *th = tcp_hdr(skb);
582         struct {
583                 struct tcphdr th;
584 #ifdef CONFIG_TCP_MD5SIG
585                 __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
586 #endif
587         } rep;
588         struct ip_reply_arg arg;
589 #ifdef CONFIG_TCP_MD5SIG
590         struct tcp_md5sig_key *key;
591         const __u8 *hash_location = NULL;
592         unsigned char newhash[16];
593         int genhash;
594         struct sock *sk1 = NULL;
595 #endif
596         struct net *net;
597
598         /* Never send a reset in response to a reset. */
599         if (th->rst)
600                 return;
601
602         if (skb_rtable(skb)->rt_type != RTN_LOCAL)
603                 return;
604
605         /* Swap the send and the receive. */
606         memset(&rep, 0, sizeof(rep));
607         rep.th.dest   = th->source;
608         rep.th.source = th->dest;
609         rep.th.doff   = sizeof(struct tcphdr) / 4;
610         rep.th.rst    = 1;
611
612         if (th->ack) {
613                 rep.th.seq = th->ack_seq;
614         } else {
615                 rep.th.ack = 1;
616                 rep.th.ack_seq = htonl(ntohl(th->seq) + th->syn + th->fin +
617                                        skb->len - (th->doff << 2));
618         }
619
620         memset(&arg, 0, sizeof(arg));
621         arg.iov[0].iov_base = (unsigned char *)&rep;
622         arg.iov[0].iov_len  = sizeof(rep.th);
623
624 #ifdef CONFIG_TCP_MD5SIG
625         hash_location = tcp_parse_md5sig_option(th);
626         if (!sk && hash_location) {
627                 /*
```

```
628                  * active side is lost. Try to find listening socket through
629                  * source port, and then find md5 key through listening socket.
630                  * we are not loose security here:
631                  * Incoming packet is checked with md5 hash with finding key,
632                  * no RST generated if md5 hash doesn't match.
633                  */
634                 sk1 = __inet_lookup_listener(dev_net(skb_dst(skb)->dev),
635                                              &tcp_hashinfo, ip_hdr(skb)->saddr,
636                                              th->source, ip_hdr(skb)->daddr,
637                                              ntohs(th->source), inet_iif(skb));
638                 /* don't send rst if it can't find key */
639                 if (!sk1)
640                         return;
641                 rcu_read_lock();
642                 key = tcp_md5_do_lookup(sk1, (union tcp_md5_addr *)
643                                         &ip_hdr(skb)->saddr, AF_INET);
644                 if (!key)
645                         goto release_sk1;
646
647                 genhash = tcp_v4_md5_hash_skb(newhash, key, NULL, NULL, skb);
648                 if (genhash || memcmp(hash_location, newhash, 16) != 0)
649                         goto release_sk1;
650         } else {
651                 key = sk ? tcp_md5_do_lookup(sk, (union tcp_md5_addr *)
652                                              &ip_hdr(skb)->saddr,
653                                              AF_INET) : NULL;
654         }
655
656         if (key) {
657                 rep.opt[0] = htonl((TCPOPT_NOP << 24) |
658                                    (TCPOPT_NOP << 16) |
659                                    (TCPOPT_MD5SIG << 8) |
660                                    TCPOLEN_MD5SIG);
661                 /* Update length and the length the header thinks exists */
662                 arg.iov[0].iov_len += TCPOLEN_MD5SIG_ALIGNED;
663                 rep.th.doff = arg.iov[0].iov_len / 4;
664
665                 tcp_v4_md5_hash_hdr((__u8 *) &rep.opt[1],
666                                     key, ip_hdr(skb)->saddr,
667                                     ip_hdr(skb)->daddr, &rep.th);
668         }
669 #endif
670         arg.csum = csum_tcpudp_nofold(ip_hdr(skb)->daddr,
671                                       ip_hdr(skb)->saddr, /* XXX */
672                                       arg.iov[0].iov_len, IPPROTO_TCP, 0);
673         arg.csumoffset = offsetof(struct tcphdr, check) / 2;
674         arg.flags = (sk && inet_sk(sk)->transparent) ? IP_REPLY_ARG_NOSRCCHECK : 0;
675         /* When socket is gone, all binding information is lost.
676          * routing might fail in this case. No choice here, if we choose to force
677          * input interface, we will misroute in case of asymmetric route.
678          */
679         if (sk)
680                 arg.bound_dev_if = sk->sk_bound_dev_if;
681
682         net = dev_net(skb_dst(skb)->dev);
683         arg.tos = ip_hdr(skb)->tos;
684         ip_send_unicast_reply(net, skb, ip_hdr(skb)->saddr,
685                               ip_hdr(skb)->daddr, &arg, arg.iov[0].iov_len);
686
687         TCP_INC_STATS_BH(net, TCP_MIB_OUTSEGS);
688         TCP_INC_STATS_BH(net, TCP_MIB_OUTRSTS);
689
690 #ifdef CONFIG_TCP_MD5SIG
691 release_sk1:
692         if (sk1) {
693                 rcu_read_unlock();
694                 sock_put(sk1);
695         }
696 #endif
697 }
698
699 /* The code following below sending ACKs in SYN-RECV and TIME-WAIT states
```

```
700    outside socket context is ugly, certainly. What can I do?
701  */
702
703 static void tcp_v4_send_ack(struct sk_buff *skb, u32 seq, u32 ack,
704                             u32 win, u32 tsval, u32 tsecr, int oif,
705                             struct tcp_md5sig_key *key,
706                             int reply_flags, u8 tos)
707 {
708         const struct tcphdr *th = tcp_hdr(skb);
709         struct {
710                 struct tcphdr th;
711                 __be32 opt[(TCPOLEN_TSTAMP_ALIGNED >> 2)
712 #ifdef CONFIG_TCP_MD5SIG
713                            + (TCPOLEN_MD5SIG_ALIGNED >> 2)
714 #endif
715                         ];
716         } rep;
717         struct ip_reply_arg arg;
718         struct net *net = dev_net(skb_dst(skb)->dev);
719
720         memset(&rep.th, 0, sizeof(struct tcphdr));
721         memset(&arg, 0, sizeof(arg));
722
723         arg.iov[0].iov_base = (unsigned char *)&rep;
724         arg.iov[0].iov_len  = sizeof(rep.th);
725         if (tsecr) {
726                 rep.opt[0] = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
727                                    (TCPOPT_TIMESTAMP << 8) |
728                                    TCPOLEN_TIMESTAMP);
729                 rep.opt[1] = htonl(tsval);
730                 rep.opt[2] = htonl(tsecr);
731                 arg.iov[0].iov_len += TCPOLEN_TSTAMP_ALIGNED;
732         }
733
734         /* Swap the send and the receive. */
735         rep.th.dest    = th->source;
736         rep.th.source  = th->dest;
737         rep.th.doff    = arg.iov[0].iov_len / 4;
738         rep.th.seq     = htonl(seq);
739         rep.th.ack_seq = htonl(ack);
740         rep.th.ack     = 1;
741         rep.th.window  = htons(win);
742
743 #ifdef CONFIG_TCP_MD5SIG
744         if (key) {
745                 int offset = (tsecr) ? 3 : 0;
746
747                 rep.opt[offset++] = htonl((TCPOPT_NOP << 24) |
748                                           (TCPOPT_NOP << 16) |
749                                           (TCPOPT_MD5SIG << 8) |
750                                           TCPOLEN_MD5SIG);
751                 arg.iov[0].iov_len += TCPOLEN_MD5SIG_ALIGNED;
752                 rep.th.doff = arg.iov[0].iov_len/4;
753
754                 tcp_v4_md5_hash_hdr((__u8 *) &rep.opt[offset],
755                                     key, ip_hdr(skb)->saddr,
756                                     ip_hdr(skb)->daddr, &rep.th);
757         }
758 #endif
759         arg.flags = reply_flags;
760         arg.csum = csum_tcpudp_nofold(ip_hdr(skb)->daddr,
761                                       ip_hdr(skb)->saddr, /* XXX */
762                                       arg.iov[0].iov_len, IPPROTO_TCP, 0);
763         arg.csumoffset = offsetof(struct tcphdr, check) / 2;
764         if (oif)
765                 arg.bound_dev_if = oif;
766         arg.tos = tos;
767         ip_send_unicast_reply(net, skb, ip_hdr(skb)->saddr,
768                               ip_hdr(skb)->daddr, &arg, arg.iov[0].iov_len);
769
770         TCP_INC_STATS_BH(net, TCP_MIB_OUTSEGS);
771 }
```

```
772
773 static void tcp_v4_timewait_ack(struct sock *sk, struct sk_buff *skb)
774 {
775         struct inet_timewait_sock *tw = inet_twsk(sk);
776         struct tcp_timewait_sock *tcptw = tcp_twsk(sk);
777
778         tcp_v4_send_ack(skb, tcptw->tw_snd_nxt, tcptw->tw_rcv_nxt,
779                         tcptw->tw_rcv_wnd >> tw->tw_rcv_wscale,
780                         tcp_time_stamp + tcptw->tw_ts_offset,
781                         tcptw->tw_ts_recent,
782                         tw->tw_bound_dev_if,
783                         tcp_twsk_md5_key(tcptw),
784                         tw->tw_transparent ? IP_REPLY_ARG_NOSRCCHECK : 0,
785                         tw->tw_tos
786                         );
787
788         inet_twsk_put(tw);
789 }
790
791 static void tcp_v4_reqsk_send_ack(struct sock *sk, struct sk_buff *skb,
792                                   struct request_sock *req)
793 {
794         /* sk->sk_state == TCP_LISTEN -> for regular TCP_SYN_RECV
795          * sk->sk_state == TCP_SYN_RECV -> for Fast Open.
796          */
797         tcp_v4_send_ack(skb, (sk->sk_state == TCP_LISTEN) ?
798                         tcp_rsk(req)->snt_isn + 1 : tcp_sk(sk)->snd_nxt,
799                         tcp_rsk(req)->rcv_nxt, req->rcv_wnd,
800                         tcp_time_stamp,
801                         req->ts_recent,
802                         0,
803                         tcp_md5_do_lookup(sk, (union tcp_md5_addr *)&ip_hdr(skb)->daddr,
804                                           AF_INET),
805                         inet_rsk(req)->no_srccheck ? IP_REPLY_ARG_NOSRCCHECK : 0,
806                         ip_hdr(skb)->tos);
807 }
808
809 /*
810  *      Send a SYN-ACK after having received a SYN.
811  *      This still operates on a request_sock only, not on a big
812  *      socket.
813  */
814 static int tcp_v4_send_synack(struct sock *sk, struct dst_entry *dst,
815                               struct flowi *fl,
816                               struct request_sock *req,
817                               u16 queue_mapping,
818                               struct tcp_fastopen_cookie *foc)
819 {
820         const struct inet_request_sock *ireq = inet_rsk(req);
821         struct flowi4 fl4;
822         int err = -1;
823         struct sk_buff *skb;
824
825         /* First, grab a route. */
826         if (!dst && (dst = inet_csk_route_req(sk, &fl4, req)) == NULL)
827                 return -1;
828
829         skb = tcp_make_synack(sk, dst, req, foc);
830
831         if (skb) {
832                 __tcp_v4_send_check(skb, ireq->ir_loc_addr, ireq->ir_rmt_addr);
833
834                 skb_set_queue_mapping(skb, queue_mapping);
835                 err = ip_build_and_send_pkt(skb, sk, ireq->ir_loc_addr,
836                                             ireq->ir_rmt_addr,
837                                             ireq->opt);
838                 err = net_xmit_eval(err);
839         }
840
841         return err;
842 }
843
```

```
844 /*
845  *      IPv4 request_sock destructor.
846  */
847 static void tcp_v4_reqsk_destructor(struct request_sock *req)
848 {
849         kfree(inet_rsk(req)->opt);
850 }
851
852 /*
853  * Return true if a syncookie should be sent
854  */
855 bool tcp_syn_flood_action(struct sock *sk,
856                           const struct sk_buff *skb,
857                           const char *proto)
858 {
859         const char *msg = "Dropping request";
860         bool want_cookie = false;
861         struct listen_sock *lopt;
862
863 #ifdef CONFIG_SYN_COOKIES
864         if (sysctl_tcp_syncookies) {
865                 msg = "Sending cookies";
866                 want_cookie = true;
867                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPREQQFULLDOCOOKIES);
868         } else
869 #endif
870                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPREQQFULLDROP);
871
872         lopt = inet_csk(sk)->icsk_accept_queue.listen_opt;
873         if (!lopt->synflood_warned && sysctl_tcp_syncookies != 2) {
874                 lopt->synflood_warned = 1;
875                 pr_info("%s: Possible SYN flooding on port %d. %s.  Check SNMP counters.\n",
876                         proto, ntohs(tcp_hdr(skb)->dest), msg);
877         }
878         return want_cookie;
879 }
880 EXPORT_SYMBOL(tcp_syn_flood_action);
881
882 /*
883  * Save and compile IPv4 options into the request_sock if needed.
884  */
885 static struct ip_options_rcu *tcp_v4_save_options(struct sk_buff *skb)
886 {
887         const struct ip_options *opt = &(IPCB(skb)->opt);
888         struct ip_options_rcu *dopt = NULL;
889
890         if (opt && opt->optlen) {
891                 int opt_size = sizeof(*dopt) + opt->optlen;
892
893                 dopt = kmalloc(opt_size, GFP_ATOMIC);
894                 if (dopt) {
895                         if (ip_options_echo(&dopt->opt, skb)) {
896                                 kfree(dopt);
897                                 dopt = NULL;
898                         }
899                 }
900         }
901         return dopt;
902 }
903
904 #ifdef CONFIG_TCP_MD5SIG
905 /*
906  * RFC2385 MD5 checksumming requires a mapping of
907  * IP address->MD5 Key.
908  * We need to maintain these in the sk structure.
909  */
910
911 /* Find the Key structure for an address.  */
912 struct tcp_md5sig_key *tcp_md5_do_lookup(struct sock *sk,
913                                          const union tcp_md5_addr *addr,
914                                          int family)
915 {
```

```
916         struct tcp_sock *tp = tcp_sk(sk);
917         struct tcp_md5sig_key *key;
918         unsigned int size = sizeof(struct in_addr);
919         struct tcp_md5sig_info *md5sig;
920
921         /* caller either holds rcu_read_lock() or socket lock */
922         md5sig = rcu_dereference_check(tp->md5sig_info,
923                                        sock_owned_by_user(sk) ||
924                                        lockdep_is_held(&sk->sk_lock.slock));
925         if (!md5sig)
926                 return NULL;
927 #if IS_ENABLED(CONFIG_IPV6)
928         if (family == AF_INET6)
929                 size = sizeof(struct in6_addr);
930 #endif
931         hlist_for_each_entry_rcu(key, &md5sig->head, node) {
932                 if (key->family != family)
933                         continue;
934                 if (!memcmp(&key->addr, addr, size))
935                         return key;
936         }
937         return NULL;
938 }
939 EXPORT_SYMBOL(tcp_md5_do_lookup);
940
941 struct tcp_md5sig_key *tcp_v4_md5_lookup(struct sock *sk,
942                                          struct sock *addr_sk)
943 {
944         union tcp_md5_addr *addr;
945
946         addr = (union tcp_md5_addr *)&inet_sk(addr_sk)->inet_daddr;
947         return tcp_md5_do_lookup(sk, addr, AF_INET);
948 }
949 EXPORT_SYMBOL(tcp_v4_md5_lookup);
950
951 static struct tcp_md5sig_key *tcp_v4_reqsk_md5_lookup(struct sock *sk,
952                                                       struct request_sock *req)
953 {
954         union tcp_md5_addr *addr;
955
956         addr = (union tcp_md5_addr *)&inet_rsk(req)->ir_rmt_addr;
957         return tcp_md5_do_lookup(sk, addr, AF_INET);
958 }
959
960 /* This can be called on a newly created socket, from other files */
961 int tcp_md5_do_add(struct sock *sk, const union tcp_md5_addr *addr,
962                    int family, const u8 *newkey, u8 newkeylen, gfp_t gfp)
963 {
964         /* Add Key to the list */
965         struct tcp_md5sig_key *key;
966         struct tcp_sock *tp = tcp_sk(sk);
967         struct tcp_md5sig_info *md5sig;
968
969         key = tcp_md5_do_lookup(sk, addr, family);
970         if (key) {
971                 /* Pre-existing entry - just update that one. */
972                 memcpy(key->key, newkey, newkeylen);
973                 key->keylen = newkeylen;
974                 return 0;
975         }
976
977         md5sig = rcu_dereference_protected(tp->md5sig_info,
978                                            sock_owned_by_user(sk));
979         if (!md5sig) {
980                 md5sig = kmalloc(sizeof(*md5sig), gfp);
981                 if (!md5sig)
982                         return -ENOMEM;
983
984                 sk_nocaps_add(sk, NETIF_F_GSO_MASK);
985                 INIT_HLIST_HEAD(&md5sig->head);
986                 rcu_assign_pointer(tp->md5sig_info, md5sig);
987         }
```

```
988
989         key = sock_kmalloc(sk, sizeof(*key), gfp);
990         if (!key)
991                 return -ENOMEM;
992         if (!tcp_alloc_md5sig_pool()) {
993                 sock_kfree_s(sk, key, sizeof(*key));
994                 return -ENOMEM;
995         }
996
997         memcpy(key->key, newkey, newkeylen);
998         key->keylen = newkeylen;
999         key->family = family;
1000         memcpy(&key->addr, addr,
1001                (family == AF_INET6) ? sizeof(struct in6_addr) :
1002                                       sizeof(struct in_addr));
1003         hlist_add_head_rcu(&key->node, &md5sig->head);
1004         return 0;
1005 }
1006 EXPORT_SYMBOL(tcp_md5_do_add);
1007
1008 int tcp_md5_do_del(struct sock *sk, const union tcp_md5_addr *addr, int family)
1009 {
1010         struct tcp_md5sig_key *key;
1011
1012         key = tcp_md5_do_lookup(sk, addr, family);
1013         if (!key)
1014                 return -ENOENT;
1015         hlist_del_rcu(&key->node);
1016         atomic_sub(sizeof(*key), &sk->sk_omem_alloc);
1017         kfree_rcu(key, rcu);
1018         return 0;
1019 }
1020 EXPORT_SYMBOL(tcp_md5_do_del);
1021
1022 static void tcp_clear_md5_list(struct sock *sk)
1023 {
1024         struct tcp_sock *tp = tcp_sk(sk);
1025         struct tcp_md5sig_key *key;
1026         struct hlist_node *n;
1027         struct tcp_md5sig_info *md5sig;
1028
1029         md5sig = rcu_dereference_protected(tp->md5sig_info, 1);
1030
1031         hlist_for_each_entry_safe(key, n, &md5sig->head, node) {
1032                 hlist_del_rcu(&key->node);
1033                 atomic_sub(sizeof(*key), &sk->sk_omem_alloc);
1034                 kfree_rcu(key, rcu);
1035         }
1036 }
1037
1038 static int tcp_v4_parse_md5_keys(struct sock *sk, char __user *optval,
1039                                  int optlen)
1040 {
1041         struct tcp_md5sig cmd;
1042         struct sockaddr_in *sin = (struct sockaddr_in *)&cmd.tcpm_addr;
1043
1044         if (optlen < sizeof(cmd))
1045                 return -EINVAL;
1046
1047         if (copy_from_user(&cmd, optval, sizeof(cmd)))
1048                 return -EFAULT;
1049
1050         if (sin->sin_family != AF_INET)
1051                 return -EINVAL;
1052
1053         if (!cmd.tcpm_keylen)
1054                 return tcp_md5_do_del(sk, (union tcp_md5_addr *)&sin->sin_addr.s_addr,
1055                                       AF_INET);
1056
1057         if (cmd.tcpm_keylen > TCP_MD5SIG_MAXKEYLEN)
1058                 return -EINVAL;
1059
```

```
1060         return tcp_md5_do_add(sk, (union tcp_md5_addr *)&sin->sin_addr.s_addr,
1061                               AF_INET, cmd.tcpm_key, cmd.tcpm_keylen,
1062                               GFP_KERNEL);
1063 }
1064
1065 static int tcp_v4_md5_hash_pseudoheader(struct tcp_md5sig_pool *hp,
1066                                         __be32 daddr, __be32 saddr, int nbytes)
1067 {
1068         struct tcp4_pseudohdr *bp;
1069         struct scatterlist sg;
1070
1071         bp = &hp->md5_blk.ip4;
1072
1073         /*
1074          * 1. the TCP pseudo-header (in the order: source IP address,
1075          * destination IP address, zero-padded protocol number, and
1076          * segment length)
1077          */
1078         bp->saddr = saddr;
1079         bp->daddr = daddr;
1080         bp->pad = 0;
1081         bp->protocol = IPPROTO_TCP;
1082         bp->len = cpu_to_be16(nbytes);
1083
1084         sg_init_one(&sg, bp, sizeof(*bp));
1085         return crypto_hash_update(&hp->md5_desc, &sg, sizeof(*bp));
1086 }
1087
1088 static int tcp_v4_md5_hash_hdr(char *md5_hash, const struct tcp_md5sig_key *key,
1089                                __be32 daddr, __be32 saddr, const struct tcphdr *th)
1090 {
1091         struct tcp_md5sig_pool *hp;
1092         struct hash_desc *desc;
1093
1094         hp = tcp_get_md5sig_pool();
1095         if (!hp)
1096                 goto clear_hash_noput;
1097         desc = &hp->md5_desc;
1098
1099         if (crypto_hash_init(desc))
1100                 goto clear_hash;
1101         if (tcp_v4_md5_hash_pseudoheader(hp, daddr, saddr, th->doff << 2))
1102                 goto clear_hash;
1103         if (tcp_md5_hash_header(hp, th))
1104                 goto clear_hash;
1105         if (tcp_md5_hash_key(hp, key))
1106                 goto clear_hash;
1107         if (crypto_hash_final(desc, md5_hash))
1108                 goto clear_hash;
1109
1110         tcp_put_md5sig_pool();
1111         return 0;
1112
1113 clear_hash:
1114         tcp_put_md5sig_pool();
1115 clear_hash_noput:
1116         memset(md5_hash, 0, 16);
1117         return 1;
1118 }
1119
1120 int tcp_v4_md5_hash_skb(char *md5_hash, struct tcp_md5sig_key *key,
1121                         const struct sock *sk, const struct request_sock *req,
1122                         const struct sk_buff *skb)
1123 {
1124         struct tcp_md5sig_pool *hp;
1125         struct hash_desc *desc;
1126         const struct tcphdr *th = tcp_hdr(skb);
1127         __be32 saddr, daddr;
1128
1129         if (sk) {
1130                 saddr = inet_sk(sk)->inet_saddr;
1131                 daddr = inet_sk(sk)->inet_daddr;
```

```
1132         } else if (req) {
1133                 saddr = inet_rsk(req)->ir_loc_addr;
1134                 daddr = inet_rsk(req)->ir_rmt_addr;
1135         } else {
1136                 const struct iphdr *iph = ip_hdr(skb);
1137                 saddr = iph->saddr;
1138                 daddr = iph->daddr;
1139         }
1140
1141         hp = tcp_get_md5sig_pool();
1142         if (!hp)
1143                 goto clear_hash_noput;
1144         desc = &hp->md5_desc;
1145
1146         if (crypto_hash_init(desc))
1147                 goto clear_hash;
1148
1149         if (tcp_v4_md5_hash_pseudoheader(hp, daddr, saddr, skb->len))
1150                 goto clear_hash;
1151         if (tcp_md5_hash_header(hp, th))
1152                 goto clear_hash;
1153         if (tcp_md5_hash_skb_data(hp, skb, th->doff << 2))
1154                 goto clear_hash;
1155         if (tcp_md5_hash_key(hp, key))
1156                 goto clear_hash;
1157         if (crypto_hash_final(desc, md5_hash))
1158                 goto clear_hash;
1159
1160         tcp_put_md5sig_pool();
1161         return 0;
1162
1163 clear_hash:
1164         tcp_put_md5sig_pool();
1165 clear_hash_noput:
1166         memset(md5_hash, 0, 16);
1167         return 1;
1168 }
1169 EXPORT_SYMBOL(tcp_v4_md5_hash_skb);
1170
1171 static bool __tcp_v4_inbound_md5_hash(struct sock *sk,
1172                                       const struct sk_buff *skb)
1173 {
1174         /*
1175          * This gets called for each TCP segment that arrives
1176          * so we want to be efficient.
1177          * We have 3 drop cases:
1178          * o No MD5 hash and one expected.
1179          * o MD5 hash and we're not expecting one.
1180          * o MD5 hash and its wrong.
1181          */
1182         const __u8 *hash_location = NULL;
1183         struct tcp_md5sig_key *hash_expected;
1184         const struct iphdr *iph = ip_hdr(skb);
1185         const struct tcphdr *th = tcp_hdr(skb);
1186         int genhash;
1187         unsigned char newhash[16];
1188
1189         hash_expected = tcp_md5_do_lookup(sk, (union tcp_md5_addr *)&iph->saddr,
1190                                           AF_INET);
1191         hash_location = tcp_parse_md5sig_option(th);
1192
1193         /* We've parsed the options - do we have a hash? */
1194         if (!hash_expected && !hash_location)
1195                 return false;
1196
1197         if (hash_expected && !hash_location) {
1198                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPMD5NOTFOUND);
1199                 return true;
1200         }
1201
1202         if (!hash_expected && hash_location) {
1203                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPMD5UNEXPECTED);
```

```
1204                return true;
1205        }
1206
1207        /* Okay, so this is hash_expected and hash_location -
1208         * so we need to calculate the checksum.
1209         */
1210        genhash = tcp_v4_md5_hash_skb(newhash,
1211                                      hash_expected,
1212                                      NULL, NULL, skb);
1213
1214        if (genhash || memcmp(hash_location, newhash, 16) != 0) {
1215                net_info_ratelimited("MD5 Hash failed for (%pI4, %d)->(%pI4, %d)%s\n",
1216                                     &iph->saddr, ntohs(th->source),
1217                                     &iph->daddr, ntohs(th->dest),
1218                                     genhash ? " tcp_v4_calc_md5_hash failed"
1219                                     : "");
1220                return true;
1221        }
1222        return false;
1223 }
1224
1225 static bool tcp_v4_inbound_md5_hash(struct sock *sk, const struct sk_buff *skb)
1226 {
1227        bool ret;
1228
1229        rcu_read_lock();
1230        ret = __tcp_v4_inbound_md5_hash(sk, skb);
1231        rcu_read_unlock();
1232
1233        return ret;
1234 }
1235
1236 #endif
1237
1238 static void tcp_v4_init_req(struct request_sock *req, struct sock *sk,
1239                             struct sk_buff *skb)
1240 {
1241        struct inet_request_sock *ireq = inet_rsk(req);
1242
1243        ireq->ir_loc_addr = ip_hdr(skb)->daddr;
1244        ireq->ir_rmt_addr = ip_hdr(skb)->saddr;
1245        ireq->no_srccheck = inet_sk(sk)->transparent;
1246        ireq->opt = tcp_v4_save_options(skb);
1247 }
1248
1249 static struct dst_entry *tcp_v4_route_req(struct sock *sk, struct flowi *fl,
1250                                           const struct request_sock *req,
1251                                           bool *strict)
1252 {
1253        struct dst_entry *dst = inet_csk_route_req(sk, &fl->u.ip4, req);
1254
1255        if (strict) {
1256                if (fl->u.ip4.daddr == inet_rsk(req)->ir_rmt_addr)
1257                        *strict = true;
1258                else
1259                        *strict = false;
1260        }
1261
1262        return dst;
1263 }
1264
1265 struct request_sock_ops tcp_request_sock_ops __read_mostly = {
1266        .family         =       PF_INET,
1267        .obj_size       =       sizeof(struct tcp_request_sock),
1268        .rtx_syn_ack    =       tcp_rtx_synack,
1269        .send_ack       =       tcp_v4_reqsk_send_ack,
1270        .destructor     =       tcp_v4_reqsk_destructor,
1271        .send_reset     =       tcp_v4_send_reset,
1272        .syn_ack_timeout =      tcp_syn_ack_timeout,
1273 };
1274
1275 static const struct tcp_request_sock_ops tcp_request_sock_ipv4_ops = {
```

```
1276         .mss_clamp      =       TCP_MSS_DEFAULT,
1277 #ifdef CONFIG_TCP_MD5SIG
1278         .md5_lookup     =       tcp_v4_reqsk_md5_lookup,
1279         .calc_md5_hash  =       tcp_v4_md5_hash_skb,
1280 #endif
1281         .init_req       =       tcp_v4_init_req,
1282 #ifdef CONFIG_SYN_COOKIES
1283         .cookie_init_seq =      cookie_v4_init_sequence,
1284 #endif
1285         .route_req      =       tcp_v4_route_req,
1286         .init_seq       =       tcp_v4_init_sequence,
1287         .send_synack    =       tcp_v4_send_synack,
1288         .queue_hash_add =       inet_csk_reqsk_queue_hash_add,
1289 };
1290
1291 int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
1292 {
1293         /* Never answer to SYNs send to broadcast or multicast */
1294         if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
1295                 goto drop;
1296
1297         return tcp_conn_request(&tcp_request_sock_ops,
1298                                 &tcp_request_sock_ipv4_ops, sk, skb);
1299
1300 drop:
1301         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
1302         return 0;
1303 }
1304 EXPORT_SYMBOL(tcp_v4_conn_request);
1305
1306
1307 /*
1308  * The three way handshake has completed - we got a valid synack -
1309  * now create the new socket.
1310  */
1311 struct sock *tcp_v4_syn_recv_sock(struct sock *sk, struct sk_buff *skb,
1312                                   struct request_sock *req,
1313                                   struct dst_entry *dst)
1314 {
1315         struct inet_request_sock *ireq;
1316         struct inet_sock *newinet;
1317         struct tcp_sock *newtp;
1318         struct sock *newsk;
1319 #ifdef CONFIG_TCP_MD5SIG
1320         struct tcp_md5sig_key *key;
1321 #endif
1322         struct ip_options_rcu *inet_opt;
1323
1324         if (sk_acceptq_is_full(sk))
1325                 goto exit_overflow;
1326
1327         newsk = tcp_create_openreq_child(sk, req, skb);
1328         if (!newsk)
1329                 goto exit_nonewsk;
1330
1331         newsk->sk_gso_type = SKB_GSO_TCPV4;
1332         inet_sk_rx_dst_set(newsk, skb);
1333
1334         newtp                 = tcp_sk(newsk);
1335         newinet               = inet_sk(newsk);
1336         ireq                  = inet_rsk(req);
1337         newinet->inet_daddr   = ireq->ir_rmt_addr;
1338         newinet->inet_rcv_saddr = ireq->ir_loc_addr;
1339         newinet->inet_saddr           = ireq->ir_loc_addr;
1340         inet_opt              = ireq->opt;
1341         rcu_assign_pointer(newinet->inet_opt, inet_opt);
1342         ireq->opt             = NULL;
1343         newinet->mc_index     = inet_iif(skb);
1344         newinet->mc_ttl       = ip_hdr(skb)->ttl;
1345         newinet->rcv_tos      = ip_hdr(skb)->tos;
1346         inet_csk(newsk)->icsk_ext_hdr_len = 0;
1347         inet_set_txhash(newsk);
```

```
1348         if (inet_opt)
1349                 inet_csk(newsk)->icsk_ext_hdr_len = inet_opt->opt.optlen;
1350         newinet->inet_id = newtp->write_seq ^ jiffies;
1351
1352         if (!dst) {
1353                 dst = inet_csk_route_child_sock(sk, newsk, req);
1354                 if (!dst)
1355                         goto put_and_exit;
1356         } else {
1357                 /* syncookie case : see end of cookie_v4_check() */
1358         }
1359         sk_setup_caps(newsk, dst);
1360
1361         tcp_sync_mss(newsk, dst_mtu(dst));
1362         newtp->advmss = dst_metric_advmss(dst);
1363         if (tcp_sk(sk)->rx_opt.user_mss &&
1364             tcp_sk(sk)->rx_opt.user_mss < newtp->advmss)
1365                 newtp->advmss = tcp_sk(sk)->rx_opt.user_mss;
1366
1367         tcp_initialize_rcv_mss(newsk);
1368
1369 #ifdef CONFIG_TCP_MD5SIG
1370         /* Copy over the MD5 key from the original socket */
1371         key = tcp_md5_do_lookup(sk, (union tcp_md5_addr *)&newinet->inet_daddr,
1372                                 AF_INET);
1373         if (key != NULL) {
1374                 /*
1375                  * We're using one, so create a matching key
1376                  * on the newsk structure. If we fail to get
1377                  * memory, then we end up not copying the key
1378                  * across. Shucks.
1379                  */
1380                 tcp_md5_do_add(newsk, (union tcp_md5_addr *)&newinet->inet_daddr,
1381                                AF_INET, key->key, key->keylen, GFP_ATOMIC);
1382                 sk_nocaps_add(newsk, NETIF_F_GSO_MASK);
1383         }
1384 #endif
1385
1386         if (__inet_inherit_port(sk, newsk) < 0)
1387                 goto put_and_exit;
1388         __inet_hash_nolisten(newsk, NULL);
1389
1390         return newsk;
1391
1392 exit_overflow:
1393         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
1394 exit_nonewsk:
1395         dst_release(dst);
1396 exit:
1397         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
1398         return NULL;
1399 put_and_exit:
1400         inet_csk_prepare_forced_close(newsk);
1401         tcp_done(newsk);
1402         goto exit;
1403 }
1404 EXPORT_SYMBOL(tcp_v4_syn_recv_sock);
1405
1406 static struct sock *tcp_v4_hnd_req(struct sock *sk, struct sk_buff *skb)
1407 {
1408         struct tcphdr *th = tcp_hdr(skb);
1409         const struct iphdr *iph = ip_hdr(skb);
1410         struct sock *nsk;
1411         struct request_sock **prev;
1412         /* Find possible connection requests. */
1413         struct request_sock *req = inet_csk_search_req(sk, &prev, th->source,
1414                                                        iph->saddr, iph->daddr);
1415         if (req)
1416                 return tcp_check_req(sk, skb, req, prev, false);
1417
1418         nsk = inet_lookup_established(sock_net(sk), &tcp_hashinfo, iph->saddr,
1419                         th->source, iph->daddr, th->dest, inet_iif(skb));
```

```
1420
1421         if (nsk) {
1422                 if (nsk->sk_state != TCP_TIME_WAIT) {
1423                         bh_lock_sock(nsk);
1424                         return nsk;
1425                 }
1426                 inet_twsk_put(inet_twsk(nsk));
1427                 return NULL;
1428         }
1429
1430 #ifdef CONFIG_SYN_COOKIES
1431         if (!th->syn)
1432                 sk = cookie_v4_check(sk, skb, &(IPCB(skb)->opt));
1433 #endif
1434         return sk;
1435 }
1436
1437 /* The socket must have it's spinlock held when we get
1438  * here.
1439  *
1440  * We have a potential double-lock case here, so even when
1441  * doing backlog processing we use the BH locking scheme.
1442  * This is because we cannot sleep with the original spinlock
1443  * held.
1444  */
1445 int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
1446 {
1447         struct sock *rsk;
1448
1449         if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
1450                 struct dst_entry *dst = sk->sk_rx_dst;
1451
1452                 sock_rps_save_rxhash(sk, skb);
1453                 if (dst) {
1454                         if (inet_sk(sk)->rx_dst_ifindex != skb->skb_iif ||
1455                             dst->ops->check(dst, 0) == NULL) {
1456                                 dst_release(dst);
1457                                 sk->sk_rx_dst = NULL;
1458                         }
1459                 }
1460                 tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len);
1461                 return 0;
1462         }
1463
1464         if (skb->len < tcp_hdrlen(skb) || tcp_checksum_complete(skb))
1465                 goto csum_err;
1466
1467         if (sk->sk_state == TCP_LISTEN) {
1468                 struct sock *nsk = tcp_v4_hnd_req(sk, skb);
1469                 if (!nsk)
1470                         goto discard;
1471
1472                 if (nsk != sk) {
1473                         sock_rps_save_rxhash(nsk, skb);
1474                         if (tcp_child_process(sk, nsk, skb)) {
1475                                 rsk = nsk;
1476                                 goto reset;
1477                         }
1478                         return 0;
1479                 }
1480         } else
1481                 sock_rps_save_rxhash(sk, skb);
1482
1483         if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
1484                 rsk = sk;
1485                 goto reset;
1486         }
1487         return 0;
1488
1489 reset:
1490         tcp_v4_send_reset(rsk, skb);
1491 discard:
```

```
1492         kfree_skb(skb);
1493         /* Be careful here. If this function gets more complicated and
1494          * gcc suffers from register pressure on the x86, sk (in %ebx)
1495          * might be destroyed here. This current version compiles correctly,
1496          * but you have been warned.
1497          */
1498         return 0;
1499
1500 csum_err:
1501         TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
1502         TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
1503         goto discard;
1504 }
1505 EXPORT_SYMBOL(tcp_v4_do_rcv);
1506
1507 void tcp_v4_early_demux(struct sk_buff *skb)
1508 {
1509         const struct iphdr *iph;
1510         const struct tcphdr *th;
1511         struct sock *sk;
1512
1513         if (skb->pkt_type != PACKET_HOST)
1514                 return;
1515
1516         if (!pskb_may_pull(skb, skb_transport_offset(skb) + sizeof(struct tcphdr)))
1517                 return;
1518
1519         iph = ip_hdr(skb);
1520         th = tcp_hdr(skb);
1521
1522         if (th->doff < sizeof(struct tcphdr) / 4)
1523                 return;
1524
1525         sk = __inet_lookup_established(dev_net(skb->dev), &tcp_hashinfo,
1526                                        iph->saddr, th->source,
1527                                        iph->daddr, ntohs(th->dest),
1528                                        skb->skb_iif);
1529         if (sk) {
1530                 skb->sk = sk;
1531                 skb->destructor = sock_edemux;
1532                 if (sk->sk_state != TCP_TIME_WAIT) {
1533                         struct dst_entry *dst = sk->sk_rx_dst;
1534
1535                         if (dst)
1536                                 dst = dst_check(dst, 0);
1537                         if (dst &&
1538                             inet_sk(sk)->rx_dst_ifindex == skb->skb_iif)
1539                                 skb_dst_set_noref(skb, dst);
1540                 }
1541         }
1542 }
1543
1544 /* Packet is added to VJ-style prequeue for processing in process
1545  * context, if a reader task is waiting. Apparently, this exciting
1546  * idea (VJ's mail "Re: query about TCP header on tcp-ip" of 07 Sep 93)
1547  * failed somewhere. Latency? Burstiness? Well, at least now we will
1548  * see, why it failed. 8)8)                               --ANK
1549  *
1550  */
1551 bool tcp_prequeue(struct sock *sk, struct sk_buff *skb)
1552 {
1553         struct tcp_sock *tp = tcp_sk(sk);
1554
1555         if (sysctl_tcp_low_latency || !tp->ucopy.task)
1556                 return false;
1557
1558         if (skb->len <= tcp_hdrlen(skb) &&
1559             skb_queue_len(&tp->ucopy.prequeue) == 0)
1560                 return false;
1561
1562         skb_dst_force(skb);
1563         __skb_queue_tail(&tp->ucopy.prequeue, skb);
```

```
1564         tp->ucopy.memory += skb->truesize;
1565         if (tp->ucopy.memory > sk->sk_rcvbuf) {
1566                 struct sk_buff *skb1;
1567
1568                 BUG_ON(sock_owned_by_user(sk));
1569
1570                 while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) != NULL) {
1571                         sk_backlog_rcv(sk, skb1);
1572                         NET_INC_STATS_BH(sock_net(sk),
1573                                          LINUX_MIB_TCPPREQUEUEDROPPED);
1574                 }
1575
1576                 tp->ucopy.memory = 0;
1577         } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
1578                 wake_up_interruptible_sync_poll(sk_sleep(sk),
1579                                                 POLLIN | POLLRDNORM | POLLRDBAND);
1580                 if (!inet_csk_ack_scheduled(sk))
1581                         inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
1582                                                   (3 * tcp_rto_min(sk)) / 4,
1583                                                   TCP_RTO_MAX);
1584         }
1585         return true;
1586 }
1587 EXPORT_SYMBOL(tcp_prequeue);
1588
1589 /*
1590  *      From tcp_input.c
1591  */
1592
1593 int tcp_v4_rcv(struct sk_buff *skb)
1594 {
1595         const struct iphdr *iph;
1596         const struct tcphdr *th;
1597         struct sock *sk;
1598         int ret;
1599         struct net *net = dev_net(skb->dev);
1600
1601         if (skb->pkt_type != PACKET_HOST)
1602                 goto discard_it;
1603
1604         /* Count it even if it's bad */
1605         TCP_INC_STATS_BH(net, TCP_MIB_INSEGS);
1606
1607         if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
1608                 goto discard_it;
1609
1610         th = tcp_hdr(skb);
1611
1612         if (th->doff < sizeof(struct tcphdr) / 4)
1613                 goto bad_packet;
1614         if (!pskb_may_pull(skb, th->doff * 4))
1615                 goto discard_it;
1616
1617         /* An explanation is required here, I think.
1618          * Packet length and doff are validated by header prediction,
1619          * provided case of th->doff==0 is eliminated.
1620          * So, we defer the checks. */
1621
1622         if (skb_checksum_init(skb, IPPROTO_TCP, inet_compute_pseudo))
1623                 goto csum_error;
1624
1625         th = tcp_hdr(skb);
1626         iph = ip_hdr(skb);
1627         TCP_SKB_CB(skb)->seq = ntohl(th->seq);
1628         TCP_SKB_CB(skb)->end_seq = (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
1629                                     skb->len - th->doff * 4);
1630         TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
1631         TCP_SKB_CB(skb)->when    = 0;
1632         TCP_SKB_CB(skb)->ip_dsfield = ipv4_get_dsfield(iph);
1633         TCP_SKB_CB(skb)->sacked  = 0;
1634
1635         sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
```

```
1636         if (!sk)
1637                 goto no_tcp_socket;
1638
1639 process:
1640         if (sk->sk_state == TCP_TIME_WAIT)
1641                 goto do_time_wait;
1642
1643         if (unlikely(iph->ttl < inet_sk(sk)->min_ttl)) {
1644                 NET_INC_STATS_BH(net, LINUX_MIB_TCPMINTTLDROP);
1645                 goto discard_and_relse;
1646         }
1647
1648         if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb))
1649                 goto discard_and_relse;
1650
1651 #ifdef CONFIG_TCP_MD5SIG
1652         /*
1653          * We really want to reject the packet as early as possible
1654          * if:
1655          *  o We're expecting an MD5'd packet and this is no MD5 tcp option
1656          *  o There is an MD5 option and we're not expecting one
1657          */
1658         if (tcp_v4_inbound_md5_hash(sk, skb))
1659                 goto discard_and_relse;
1660 #endif
1661
1662         nf_reset(skb);
1663
1664         if (sk_filter(sk, skb))
1665                 goto discard_and_relse;
1666
1667         sk_mark_napi_id(sk, skb);
1668         skb->dev = NULL;
1669
1670         bh_lock_sock_nested(sk);
1671         ret = 0;
1672         if (!sock_owned_by_user(sk)) {
1673 #ifdef CONFIG_NET_DMA
1674                 struct tcp_sock *tp = tcp_sk(sk);
1675                 if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
1676                         tp->ucopy.dma_chan = net_dma_find_channel();
1677                 if (tp->ucopy.dma_chan)
1678                         ret = tcp_v4_do_rcv(sk, skb);
1679                 else
1680 #endif
1681                 {
1682                         if (!tcp_prequeue(sk, skb))
1683                                 ret = tcp_v4_do_rcv(sk, skb);
1684                 }
1685         } else if (unlikely(sk_add_backlog(sk, skb,
1686                                            sk->sk_rcvbuf + sk->sk_sndbuf))) {
1687                 bh_unlock_sock(sk);
1688                 NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
1689                 goto discard_and_relse;
1690         }
1691         bh_unlock_sock(sk);
1692
1693         sock_put(sk);
1694
1695         return ret;
1696
1697 no_tcp_socket:
1698         if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
1699                 goto discard_it;
1700
1701         if (skb->len < (th->doff << 2) || tcp_checksum_complete(skb)) {
1702 csum_error:
1703                 TCP_INC_STATS_BH(net, TCP_MIB_CSUMERRORS);
1704 bad_packet:
1705                 TCP_INC_STATS_BH(net, TCP_MIB_INERRS);
1706         } else {
1707                 tcp_v4_send_reset(NULL, skb);
```

```
1708         }
1709
1710 discard_it:
1711         /* Discard frame. */
1712         kfree_skb(skb);
1713         return 0;
1714
1715 discard_and_relse:
1716         sock_put(sk);
1717         goto discard_it;
1718
1719 do_time_wait:
1720         if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
1721                 inet_twsk_put(inet_twsk(sk));
1722                 goto discard_it;
1723         }
1724
1725         if (skb->len < (th->doff << 2)) {
1726                 inet_twsk_put(inet_twsk(sk));
1727                 goto bad_packet;
1728         }
1729         if (tcp_checksum_complete(skb)) {
1730                 inet_twsk_put(inet_twsk(sk));
1731                 goto csum_error;
1732         }
1733         switch (tcp_timewait_state_process(inet_twsk(sk), skb, th)) {
1734         case TCP_TW_SYN: {
1735                 struct sock *sk2 = inet_lookup_listener(dev_net(skb->dev),
1736                                                         &tcp_hashinfo,
1737                                                         iph->saddr, th->source,
1738                                                         iph->daddr, th->dest,
1739                                                         inet_iif(skb));
1740                 if (sk2) {
1741                         inet_twsk_deschedule(inet_twsk(sk), &tcp_death_row);
1742                         inet_twsk_put(inet_twsk(sk));
1743                         sk = sk2;
1744                         goto process;
1745                 }
1746                 /* Fall through to ACK */
1747         }
1748         case TCP_TW_ACK:
1749                 tcp_v4_timewait_ack(sk, skb);
1750                 break;
1751         case TCP_TW_RST:
1752                 goto no_tcp_socket;
1753         case TCP_TW_SUCCESS:;
1754         }
1755         goto discard_it;
1756 }
1757
1758 static struct timewait_sock_ops tcp_timewait_sock_ops = {
1759         .twsk_obj_size  = sizeof(struct tcp_timewait_sock),
1760         .twsk_unique    = tcp_twsk_unique,
1761         .twsk_destructor= tcp_twsk_destructor,
1762 };
1763
1764 void inet_sk_rx_dst_set(struct sock *sk, const struct sk_buff *skb)
1765 {
1766         struct dst_entry *dst = skb_dst(skb);
1767
1768         dst_hold(dst);
1769         sk->sk_rx_dst = dst;
1770         inet_sk(sk)->rx_dst_ifindex = skb->skb_iif;
1771 }
1772 EXPORT_SYMBOL(inet_sk_rx_dst_set);
1773
1774 const struct inet_connection_sock_af_ops ipv4_specific = {
1775         .queue_xmit        = ip_queue_xmit,
1776         .send_check        = tcp_v4_send_check,
1777         .rebuild_header    = inet_sk_rebuild_header,
1778         .sk_rx_dst_set     = inet_sk_rx_dst_set,
1779         .conn_request      = tcp_v4_conn_request,
```

```
1780         .syn_recv_sock     = tcp_v4_syn_recv_sock,
1781         .net_header_len    = sizeof(struct iphdr),
1782         .setsockopt        = ip_setsockopt,
1783         .getsockopt        = ip_getsockopt,
1784         .addr2sockaddr     = inet_csk_addr2sockaddr,
1785         .sockaddr_len      = sizeof(struct sockaddr_in),
1786         .bind_conflict     = inet_csk_bind_conflict,
1787 #ifdef CONFIG_COMPAT
1788         .compat_setsockopt = compat_ip_setsockopt,
1789         .compat_getsockopt = compat_ip_getsockopt,
1790 #endif
1791         .mtu_reduced       = tcp_v4_mtu_reduced,
1792 };
1793 EXPORT_SYMBOL(ipv4_specific);
1794
1795 #ifdef CONFIG_TCP_MD5SIG
1796 static const struct tcp_sock_af_ops tcp_sock_ipv4_specific = {
1797         .md5_lookup             = tcp_v4_md5_lookup,
1798         .calc_md5_hash          = tcp_v4_md5_hash_skb,
1799         .md5_parse              = tcp_v4_parse_md5_keys,
1800 };
1801 #endif
1802
1803 /* NOTE: A lot of things set to zero explicitly by call to
1804  *       sk_alloc() so need not be done here.
1805  */
1806 static int tcp_v4_init_sock(struct sock *sk)
1807 {
1808         struct inet_connection_sock *icsk = inet_csk(sk);
1809
1810         tcp_init_sock(sk);
1811
1812         icsk->icsk_af_ops = &ipv4_specific;
1813
1814 #ifdef CONFIG_TCP_MD5SIG
1815         tcp_sk(sk)->af_specific = &tcp_sock_ipv4_specific;
1816 #endif
1817
1818         return 0;
1819 }
1820
1821 void tcp_v4_destroy_sock(struct sock *sk)
1822 {
1823         struct tcp_sock *tp = tcp_sk(sk);
1824
1825         tcp_clear_xmit_timers(sk);
1826
1827         tcp_cleanup_congestion_control(sk);
1828
1829         /* Cleanup up the write buffer. */
1830         tcp_write_queue_purge(sk);
1831
1832         /* Cleans up our, hopefully empty, out_of_order_queue. */
1833         __skb_queue_purge(&tp->out_of_order_queue);
1834
1835 #ifdef CONFIG_TCP_MD5SIG
1836         /* Clean up the MD5 key list, if any */
1837         if (tp->md5sig_info) {
1838                 tcp_clear_md5_list(sk);
1839                 kfree_rcu(tp->md5sig_info, rcu);
1840                 tp->md5sig_info = NULL;
1841         }
1842 #endif
1843
1844 #ifdef CONFIG_NET_DMA
1845         /* Cleans up our sk_async_wait_queue */
1846         __skb_queue_purge(&sk->sk_async_wait_queue);
1847 #endif
1848
1849         /* Clean prequeue, it must be empty really */
1850         __skb_queue_purge(&tp->ucopy.prequeue);
1851
```

```
1852         /* Clean up a referenced TCP bind bucket. */
1853         if (inet_csk(sk)->icsk_bind_hash)
1854                 inet_put_port(sk);
1855
1856         BUG_ON(tp->fastopen_rsk != NULL);
1857
1858         /* If socket is aborted during connect operation */
1859         tcp_free_fastopen_req(tp);
1860
1861         sk_sockets_allocated_dec(sk);
1862         sock_release_memcg(sk);
1863 }
1864 EXPORT_SYMBOL(tcp_v4_destroy_sock);
1865
1866 #ifdef CONFIG_PROC_FS
1867 /* Proc filesystem TCP sock list dumping. */
1868
1869 /*
1870  * Get next listener socket follow cur.  If cur is NULL, get first socket
1871  * starting from bucket given in st->bucket; when st->bucket is zero the
1872  * very first socket in the hash table is returned.
1873  */
1874 static void *listening_get_next(struct seq_file *seq, void *cur)
1875 {
1876         struct inet_connection_sock *icsk;
1877         struct hlist_nulls_node *node;
1878         struct sock *sk = cur;
1879         struct inet_listen_hashbucket *ilb;
1880         struct tcp_iter_state *st = seq->private;
1881         struct net *net = seq_file_net(seq);
1882
1883         if (!sk) {
1884                 ilb = &tcp_hashinfo.listening_hash[st->bucket];
1885                 spin_lock_bh(&ilb->lock);
1886                 sk = sk_nulls_head(&ilb->head);
1887                 st->offset = 0;
1888                 goto get_sk;
1889         }
1890         ilb = &tcp_hashinfo.listening_hash[st->bucket];
1891         ++st->num;
1892         ++st->offset;
1893
1894         if (st->state == TCP_SEQ_STATE_OPENREQ) {
1895                 struct request_sock *req = cur;
1896
1897                 icsk = inet_csk(st->syn_wait_sk);
1898                 req = req->dl_next;
1899                 while (1) {
1900                         while (req) {
1901                                 if (req->rsk_ops->family == st->family) {
1902                                         cur = req;
1903                                         goto out;
1904                                 }
1905                                 req = req->dl_next;
1906                         }
1907                         if (++st->sbucket >= icsk->icsk_accept_queue.listen_opt->nr_table_entries)
1908                                 break;
1909 get_req:
1910                         req = icsk->icsk_accept_queue.listen_opt->syn_table[st->sbucket];
1911                 }
1912                 sk        = sk_nulls_next(st->syn_wait_sk);
1913                 st->state = TCP_SEQ_STATE_LISTENING;
1914                 read_unlock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
1915         } else {
1916                 icsk = inet_csk(sk);
1917                 read_lock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
1918                 if (reqsk_queue_len(&icsk->icsk_accept_queue))
1919                         goto start_req;
1920                 read_unlock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
1921                 sk = sk_nulls_next(sk);
1922         }
1923 get_sk:
```

```
1924         sk_nulls_for_each_from(sk, node) {
1925                 if (!net_eq(sock_net(sk), net))
1926                         continue;
1927                 if (sk->sk_family == st->family) {
1928                         cur = sk;
1929                         goto out;
1930                 }
1931                 icsk = inet_csk(sk);
1932                 read_lock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
1933                 if (reqsk_queue_len(&icsk->icsk_accept_queue)) {
1934 start_req:
1935                         st->uid         = sock_i_uid(sk);
1936                         st->syn_wait_sk = sk;
1937                         st->state       = TCP_SEQ_STATE_OPENREQ;
1938                         st->sbucket     = 0;
1939                         goto get_req;
1940                 }
1941                 read_unlock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
1942         }
1943         spin_unlock_bh(&ilb->lock);
1944         st->offset = 0;
1945         if (++st->bucket < INET_LHTABLE_SIZE) {
1946                 ilb = &tcp_hashinfo.listening_hash[st->bucket];
1947                 spin_lock_bh(&ilb->lock);
1948                 sk = sk_nulls_head(&ilb->head);
1949                 goto get_sk;
1950         }
1951         cur = NULL;
1952 out:
1953         return cur;
1954 }
1955
1956 static void *listening_get_idx(struct seq_file *seq, loff_t *pos)
1957 {
1958         struct tcp_iter_state *st = seq->private;
1959         void *rc;
1960
1961         st->bucket = 0;
1962         st->offset = 0;
1963         rc = listening_get_next(seq, NULL);
1964
1965         while (rc && *pos) {
1966                 rc = listening_get_next(seq, rc);
1967                 --*pos;
1968         }
1969         return rc;
1970 }
1971
1972 static inline bool empty_bucket(const struct tcp_iter_state *st)
1973 {
1974         return hlist_nulls_empty(&tcp_hashinfo.ehash[st->bucket].chain);
1975 }
1976
1977 /*
1978  * Get first established socket starting from bucket given in st->bucket.
1979  * If st->bucket is zero, the very first socket in the hash is returned.
1980  */
1981 static void *established_get_first(struct seq_file *seq)
1982 {
1983         struct tcp_iter_state *st = seq->private;
1984         struct net *net = seq_file_net(seq);
1985         void *rc = NULL;
1986
1987         st->offset = 0;
1988         for (; st->bucket <= tcp_hashinfo.ehash_mask; ++st->bucket) {
1989                 struct sock *sk;
1990                 struct hlist_nulls_node *node;
1991                 spinlock_t *lock = inet_ehash_lockp(&tcp_hashinfo, st->bucket);
1992
1993                 /* Lockless fast path for the common case of empty buckets */
1994                 if (empty_bucket(st))
1995                         continue;
```

```
1996
1997                    spin_lock_bh(lock);
1998                    sk_nulls_for_each(sk, node, &tcp_hashinfo.ehash[st->bucket].chain) {
1999                            if (sk->sk_family != st->family ||
2000                                !net_eq(sock_net(sk), net)) {
2001                                    continue;
2002                            }
2003                            rc = sk;
2004                            goto out;
2005                    }
2006                    spin_unlock_bh(lock);
2007            }
2008    out:
2009            return rc;
2010    }
2011
2012    static void *established_get_next(struct seq_file *seq, void *cur)
2013    {
2014            struct sock *sk = cur;
2015            struct hlist_nulls_node *node;
2016            struct tcp_iter_state *st = seq->private;
2017            struct net *net = seq_file_net(seq);
2018
2019            ++st->num;
2020            ++st->offset;
2021
2022            sk = sk_nulls_next(sk);
2023
2024            sk_nulls_for_each_from(sk, node) {
2025                    if (sk->sk_family == st->family && net_eq(sock_net(sk), net))
2026                            return sk;
2027            }
2028
2029            spin_unlock_bh(inet_ehash_lockp(&tcp_hashinfo, st->bucket));
2030            ++st->bucket;
2031            return established_get_first(seq);
2032    }
2033
2034    static void *established_get_idx(struct seq_file *seq, loff_t pos)
2035    {
2036            struct tcp_iter_state *st = seq->private;
2037            void *rc;
2038
2039            st->bucket = 0;
2040            rc = established_get_first(seq);
2041
2042            while (rc && pos) {
2043                    rc = established_get_next(seq, rc);
2044                    --pos;
2045            }
2046            return rc;
2047    }
2048
2049    static void *tcp_get_idx(struct seq_file *seq, loff_t pos)
2050    {
2051            void *rc;
2052            struct tcp_iter_state *st = seq->private;
2053
2054            st->state = TCP_SEQ_STATE_LISTENING;
2055            rc        = listening_get_idx(seq, &pos);
2056
2057            if (!rc) {
2058                    st->state = TCP_SEQ_STATE_ESTABLISHED;
2059                    rc        = established_get_idx(seq, pos);
2060            }
2061
2062            return rc;
2063    }
2064
2065    static void *tcp_seek_last_pos(struct seq_file *seq)
2066    {
2067            struct tcp_iter_state *st = seq->private;
```

```
2068         int offset = st->offset;
2069         int orig_num = st->num;
2070         void *rc = NULL;
2071
2072         switch (st->state) {
2073         case TCP_SEQ_STATE_OPENREQ:
2074         case TCP_SEQ_STATE_LISTENING:
2075                 if (st->bucket >= INET_LHTABLE_SIZE)
2076                         break;
2077                 st->state = TCP_SEQ_STATE_LISTENING;
2078                 rc = listening_get_next(seq, NULL);
2079                 while (offset-- && rc)
2080                         rc = listening_get_next(seq, rc);
2081                 if (rc)
2082                         break;
2083                 st->bucket = 0;
2084                 st->state = TCP_SEQ_STATE_ESTABLISHED;
2085                 /* Fallthrough */
2086         case TCP_SEQ_STATE_ESTABLISHED:
2087                 if (st->bucket > tcp_hashinfo.ehash_mask)
2088                         break;
2089                 rc = established_get_first(seq);
2090                 while (offset-- && rc)
2091                         rc = established_get_next(seq, rc);
2092         }
2093
2094         st->num = orig_num;
2095
2096         return rc;
2097 }
2098
2099 static void *tcp_seq_start(struct seq_file *seq, loff_t *pos)
2100 {
2101         struct tcp_iter_state *st = seq->private;
2102         void *rc;
2103
2104         if (*pos && *pos == st->last_pos) {
2105                 rc = tcp_seek_last_pos(seq);
2106                 if (rc)
2107                         goto out;
2108         }
2109
2110         st->state = TCP_SEQ_STATE_LISTENING;
2111         st->num = 0;
2112         st->bucket = 0;
2113         st->offset = 0;
2114         rc = *pos ? tcp_get_idx(seq, *pos - 1) : SEQ_START_TOKEN;
2115
2116 out:
2117         st->last_pos = *pos;
2118         return rc;
2119 }
2120
2121 static void *tcp_seq_next(struct seq_file *seq, void *v, loff_t *pos)
2122 {
2123         struct tcp_iter_state *st = seq->private;
2124         void *rc = NULL;
2125
2126         if (v == SEQ_START_TOKEN) {
2127                 rc = tcp_get_idx(seq, 0);
2128                 goto out;
2129         }
2130
2131         switch (st->state) {
2132         case TCP_SEQ_STATE_OPENREQ:
2133         case TCP_SEQ_STATE_LISTENING:
2134                 rc = listening_get_next(seq, v);
2135                 if (!rc) {
2136                         st->state = TCP_SEQ_STATE_ESTABLISHED;
2137                         st->bucket = 0;
2138                         st->offset = 0;
2139                         rc        = established_get_first(seq);
```

```
2140                 }
2141                 break;
2142         case TCP_SEQ_STATE_ESTABLISHED:
2143                 rc = established_get_next(seq, v);
2144                 break;
2145         }
2146 out:
2147         ++*pos;
2148         st->last_pos = *pos;
2149         return rc;
2150 }
2151
2152 static void tcp_seq_stop(struct seq_file *seq, void *v)
2153 {
2154         struct tcp_iter_state *st = seq->private;
2155
2156         switch (st->state) {
2157         case TCP_SEQ_STATE_OPENREQ:
2158                 if (v) {
2159                         struct inet_connection_sock *icsk = inet_csk(st->syn_wait_sk);
2160                         read_unlock_bh(&icsk->icsk_accept_queue.syn_wait_lock);
2161                 }
2162         case TCP_SEQ_STATE_LISTENING:
2163                 if (v != SEQ_START_TOKEN)
2164                         spin_unlock_bh(&tcp_hashinfo.listening_hash[st->bucket].lock);
2165                 break;
2166         case TCP_SEQ_STATE_ESTABLISHED:
2167                 if (v)
2168                         spin_unlock_bh(inet_ehash_lockp(&tcp_hashinfo, st->bucket));
2169                 break;
2170         }
2171 }
2172
2173 int tcp_seq_open(struct inode *inode, struct file *file)
2174 {
2175         struct tcp_seq_afinfo *afinfo = PDE_DATA(inode);
2176         struct tcp_iter_state *s;
2177         int err;
2178
2179         err = seq_open_net(inode, file, &afinfo->seq_ops,
2180                           sizeof(struct tcp_iter_state));
2181         if (err < 0)
2182                 return err;
2183
2184         s = ((struct seq_file *)file->private_data)->private;
2185         s->family               = afinfo->family;
2186         s->last_pos             = 0;
2187         return 0;
2188 }
2189 EXPORT_SYMBOL(tcp_seq_open);
2190
2191 int tcp_proc_register(struct net *net, struct tcp_seq_afinfo *afinfo)
2192 {
2193         int rc = 0;
2194         struct proc_dir_entry *p;
2195
2196         afinfo->seq_ops.start           = tcp_seq_start;
2197         afinfo->seq_ops.next            = tcp_seq_next;
2198         afinfo->seq_ops.stop            = tcp_seq_stop;
2199
2200         p = proc_create_data(afinfo->name, S_IRUGO, net->proc_net,
2201                              afinfo->seq_fops, afinfo);
2202         if (!p)
2203                 rc = -ENOMEM;
2204         return rc;
2205 }
2206 EXPORT_SYMBOL(tcp_proc_register);
2207
2208 void tcp_proc_unregister(struct net *net, struct tcp_seq_afinfo *afinfo)
2209 {
2210         remove_proc_entry(afinfo->name, net->proc_net);
2211 }
```

```
2212 EXPORT_SYMBOL(tcp_proc_unregister);
2213
2214 static void get_openreq4(const struct sock *sk, const struct request_sock *req,
2215                          struct seq_file *f, int i, kuid_t uid)
2216 {
2217         const struct inet_request_sock *ireq = inet_rsk(req);
2218         long delta = req->expires - jiffies;
2219
2220         seq_printf(f, "%4d: %08X:%04X %08X:%04X"
2221                 " %02X %08X:%08X %02X:%08lX %08X %5u %8d %u %d %pK",
2222                 i,
2223                 ireq->ir_loc_addr,
2224                 ntohs(inet_sk(sk)->inet_sport),
2225                 ireq->ir_rmt_addr,
2226                 ntohs(ireq->ir_rmt_port),
2227                 TCP_SYN_RECV,
2228                 0, 0, /* could print option size, but that is af dependent. */
2229                 1,    /* timers active (only the expire timer) */
2230                 jiffies_delta_to_clock_t(delta),
2231                 req->num_timeout,
2232                 from_kuid_munged(seq_user_ns(f), uid),
2233                 0,  /* non standard timer */
2234                 0, /* open_requests have no inode */
2235                 atomic_read(&sk->sk_refcnt),
2236                 req);
2237 }
2238
2239 static void get_tcp4_sock(struct sock *sk, struct seq_file *f, int i)
2240 {
2241         int timer_active;
2242         unsigned long timer_expires;
2243         const struct tcp_sock *tp = tcp_sk(sk);
2244         const struct inet_connection_sock *icsk = inet_csk(sk);
2245         const struct inet_sock *inet = inet_sk(sk);
2246         struct fastopen_queue *fastopenq = icsk->icsk_accept_queue.fastopenq;
2247         __be32 dest = inet->inet_daddr;
2248         __be32 src = inet->inet_rcv_saddr;
2249         __u16 destp = ntohs(inet->inet_dport);
2250         __u16 srcp = ntohs(inet->inet_sport);
2251         int rx_queue;
2252
2253         if (icsk->icsk_pending == ICSK_TIME_RETRANS ||
2254             icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
2255             icsk->icsk_pending == ICSK_TIME_LOSS_PROBE) {
2256                 timer_active    = 1;
2257                 timer_expires   = icsk->icsk_timeout;
2258         } else if (icsk->icsk_pending == ICSK_TIME_PROBE0) {
2259                 timer_active    = 4;
2260                 timer_expires   = icsk->icsk_timeout;
2261         } else if (timer_pending(&sk->sk_timer)) {
2262                 timer_active    = 2;
2263                 timer_expires   = sk->sk_timer.expires;
2264         } else {
2265                 timer_active    = 0;
2266                 timer_expires = jiffies;
2267         }
2268
2269         if (sk->sk_state == TCP_LISTEN)
2270                 rx_queue = sk->sk_ack_backlog;
2271         else
2272                 /*
2273                  * because we dont lock socket, we might find a transient negative value
2274                  */
2275                 rx_queue = max_t(int, tp->rcv_nxt - tp->copied_seq, 0);
2276
2277         seq_printf(f, "%4d: %08X:%04X %08X:%04X %02X %08X:%08X %02X:%08lX "
2278                         "%08X %5u %8d %lu %d %pK %lu %lu %u %u %d",
2279                 i, src, srcp, dest, destp, sk->sk_state,
2280                 tp->write_seq - tp->snd_una,
2281                 rx_queue,
2282                 timer_active,
2283                 jiffies_delta_to_clock_t(timer_expires - jiffies),
```

```
2284                 icsk->icsk_retransmits,
2285                 from_kuid_munged(seq_user_ns(f), sock_i_uid(sk)),
2286                 icsk->icsk_probes_out,
2287                 sock_i_ino(sk),
2288                 atomic_read(&sk->sk_refcnt), sk,
2289                 jiffies_to_clock_t(icsk->icsk_rto),
2290                 jiffies_to_clock_t(icsk->icsk_ack.ato),
2291                 (icsk->icsk_ack.quick << 1) | icsk->icsk_ack.pingpong,
2292                 tp->snd_cwnd,
2293                 sk->sk_state == TCP_LISTEN ?
2294                     (fastopenq ? fastopenq->max_qlen : 0) :
2295                     (tcp_in_initial_slowstart(tp) ? -1 : tp->snd_ssthresh));
2296 }
2297
2298 static void get_timewait4_sock(const struct inet_timewait_sock *tw,
2299                                struct seq_file *f, int i)
2300 {
2301         __be32 dest, src;
2302         __u16 destp, srcp;
2303         s32 delta = tw->tw_ttd - inet_tw_time_stamp();
2304
2305         dest  = tw->tw_daddr;
2306         src   = tw->tw_rcv_saddr;
2307         destp = ntohs(tw->tw_dport);
2308         srcp  = ntohs(tw->tw_sport);
2309
2310         seq_printf(f, "%4d: %08X:%04X %08X:%04X"
2311                 " %02X %08X:%08X %02X:%08lX %08X %5d %8d %d %d %pK",
2312                 i, src, srcp, dest, destp, tw->tw_substate, 0, 0,
2313                 3, jiffies_delta_to_clock_t(delta), 0, 0, 0, 0,
2314                 atomic_read(&tw->tw_refcnt), tw);
2315 }
2316
2317 #define TMPSZ 150
2318
2319 static int tcp4_seq_show(struct seq_file *seq, void *v)
2320 {
2321         struct tcp_iter_state *st;
2322         struct sock *sk = v;
2323
2324         seq_setwidth(seq, TMPSZ - 1);
2325         if (v == SEQ_START_TOKEN) {
2326                 seq_puts(seq, "  sl  local_address rem_address   st tx_queue "
2327                            "rx_queue tr tm->when retrnsmt   uid  timeout "
2328                            "inode");
2329                 goto out;
2330         }
2331         st = seq->private;
2332
2333         switch (st->state) {
2334         case TCP_SEQ_STATE_LISTENING:
2335         case TCP_SEQ_STATE_ESTABLISHED:
2336                 if (sk->sk_state == TCP_TIME_WAIT)
2337                         get_timewait4_sock(v, seq, st->num);
2338                 else
2339                         get_tcp4_sock(v, seq, st->num);
2340                 break;
2341         case TCP_SEQ_STATE_OPENREQ:
2342                 get_openreq4(st->syn_wait_sk, v, seq, st->num, st->uid);
2343                 break;
2344         }
2345 out:
2346         seq_pad(seq, '\n');
2347         return 0;
2348 }
2349
2350 static const struct file_operations tcp_afinfo_seq_fops = {
2351         .owner   = THIS_MODULE,
2352         .open    = tcp_seq_open,
2353         .read    = seq_read,
2354         .llseek  = seq_lseek,
2355         .release = seq_release_net
```

```
2356 };
2357
2358 static struct tcp_seq_afinfo tcp4_seq_afinfo = {
2359         .name           = "tcp",
2360         .family         = AF_INET,
2361         .seq_fops       = &tcp_afinfo_seq_fops,
2362         .seq_ops        = {
2363                 .show           = tcp4_seq_show,
2364         },
2365 };
2366
2367 static int __net_init tcp4_proc_init_net(struct net *net)
2368 {
2369         return tcp_proc_register(net, &tcp4_seq_afinfo);
2370 }
2371
2372 static void __net_exit tcp4_proc_exit_net(struct net *net)
2373 {
2374         tcp_proc_unregister(net, &tcp4_seq_afinfo);
2375 }
2376
2377 static struct pernet_operations tcp4_net_ops = {
2378         .init = tcp4_proc_init_net,
2379         .exit = tcp4_proc_exit_net,
2380 };
2381
2382 int __init tcp4_proc_init(void)
2383 {
2384         return register_pernet_subsys(&tcp4_net_ops);
2385 }
2386
2387 void tcp4_proc_exit(void)
2388 {
2389         unregister_pernet_subsys(&tcp4_net_ops);
2390 }
2391 #endif /* CONFIG_PROC_FS */
2392
2393 struct proto tcp_prot = {
2394         .name                   = "TCP",
2395         .owner                  = THIS_MODULE,
2396         .close                  = tcp_close,
2397         .connect                = tcp_v4_connect,
2398         .disconnect             = tcp_disconnect,
2399         .accept                 = inet_csk_accept,
2400         .ioctl                  = tcp_ioctl,
2401         .init                   = tcp_v4_init_sock,
2402         .destroy                = tcp_v4_destroy_sock,
2403         .shutdown               = tcp_shutdown,
2404         .setsockopt             = tcp_setsockopt,
2405         .getsockopt             = tcp_getsockopt,
2406         .recvmsg                = tcp_recvmsg,
2407         .sendmsg                = tcp_sendmsg,
2408         .sendpage               = tcp_sendpage,
2409         .backlog_rcv            = tcp_v4_do_rcv,
2410         .release_cb             = tcp_release_cb,
2411         .hash                   = inet_hash,
2412         .unhash                 = inet_unhash,
2413         .get_port               = inet_csk_get_port,
2414         .enter_memory_pressure  = tcp_enter_memory_pressure,
2415         .stream_memory_free     = tcp_stream_memory_free,
2416         .sockets_allocated      = &tcp_sockets_allocated,
2417         .orphan_count           = &tcp_orphan_count,
2418         .memory_allocated       = &tcp_memory_allocated,
2419         .memory_pressure        = &tcp_memory_pressure,
2420         .sysctl_mem             = sysctl_tcp_mem,
2421         .sysctl_wmem            = sysctl_tcp_wmem,
2422         .sysctl_rmem            = sysctl_tcp_rmem,
2423         .max_header             = MAX_TCP_HEADER,
2424         .obj_size               = sizeof(struct tcp_sock),
2425         .slab_flags             = SLAB_DESTROY_BY_RCU,
2426         .twsk_prot              = &tcp_timewait_sock_ops,
2427         .rsk_prot               = &tcp_request_sock_ops,
```

```
2428         .h.hashinfo             = &tcp_hashinfo,
2429         .no_autobind            = true,
2430 #ifdef CONFIG_COMPAT
2431         .compat_setsockopt      = compat_tcp_setsockopt,
2432         .compat_getsockopt      = compat_tcp_getsockopt,
2433 #endif
2434 #ifdef CONFIG_MEMCG_KMEM
2435         .init_cgroup            = tcp_init_cgroup,
2436         .destroy_cgroup         = tcp_destroy_cgroup,
2437         .proto_cgroup           = tcp_proto_cgroup,
2438 #endif
2439 };
2440 EXPORT_SYMBOL(tcp_prot);
2441
2442 static int __net_init tcp_sk_init(struct net *net)
2443 {
2444         net->ipv4.sysctl_tcp_ecn = 2;
2445         return 0;
2446 }
2447
2448 static void __net_exit tcp_sk_exit(struct net *net)
2449 {
2450 }
2451
2452 static void __net_exit tcp_sk_exit_batch(struct list_head *net_exit_list)
2453 {
2454         inet_twsk_purge(&tcp_hashinfo, &tcp_death_row, AF_INET);
2455 }
2456
2457 static struct pernet_operations __net_initdata tcp_sk_ops = {
2458        .init       = tcp_sk_init,
2459        .exit       = tcp_sk_exit,
2460        .exit_batch = tcp_sk_exit_batch,
2461 };
2462
2463 void __init tcp_v4_init(void)
2464 {
2465         inet_hashinfo_init(&tcp_hashinfo);
2466         if (register_pernet_subsys(&tcp_sk_ops))
2467                 panic("Failed to create the TCP control socket.\n");
2468 }
2469
```