Linux Cross Reference

Free Electrons

Embedded Linux Experts

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

Version:
2.0.40 2.2.26 2.4.37 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 ***3.17***

Linux/net/ipv4/tcp_offload.c

```
 1 /*
 2  *      IPV4 GSO/GRO offload support
 3  *      Linux INET implementation
 4  *
 5  *      This program is free software; you can redistribute it and/or
 6  *      modify it under the terms of the GNU General Public License
 7  *      as published by the Free Software Foundation; either version
 8  *      2 of the License, or (at your option) any later version.
 9  *
10  *      TCPv4 GSO/GRO support
11  */
12
13 #include <linux/skbuff.h>
14 #include <net/tcp.h>
15 #include <net/protocol.h>
16
17 static void tcp_gso_tstamp(struct sk_buff *skb, unsigned int ts_seq,
18                            unsigned int seq, unsigned int mss)
19 {
20         while (skb) {
21                 if (before(ts_seq, seq + mss)) {
22                         skb_shinfo(skb)->tx_flags |= SKBTX_SW_TSTAMP;
23                         skb_shinfo(skb)->tskey = ts_seq;
24                         return;
25                 }
26
27                 skb = skb->next;
28                 seq += mss;
29         }
30 }
31
32 struct sk_buff *tcp_gso_segment(struct sk_buff *skb,
33                                 netdev_features_t features)
34 {
35         struct sk_buff *segs = ERR_PTR(-EINVAL);
36         unsigned int sum_truesize = 0;
37         struct tcphdr *th;
38         unsigned int thlen;
39         unsigned int seq;
40         __be32 delta;
41         unsigned int oldlen;
42         unsigned int mss;
43         struct sk_buff *gso_skb = skb;
```

```
44          __sum16 newcheck;
45          bool ooo_okay, copy_destructor;
46
47          if (!pskb_may_pull(skb, sizeof(*th)))
48                  goto out;
49
50          th = tcp_hdr(skb);
51          thlen = th->doff * 4;
52          if (thlen < sizeof(*th))
53                  goto out;
54
55          if (!pskb_may_pull(skb, thlen))
56                  goto out;
57
58          oldlen = (u16)~skb->len;
59          __skb_pull(skb, thlen);
60
61          mss = tcp_skb_mss(skb);
62          if (unlikely(skb->len <= mss))
63                  goto out;
64
65          if (skb_gso_ok(skb, features | NETIF_F_GSO_ROBUST)) {
66                  /* Packet is from an untrusted source, reset gso_segs. */
67                  int type = skb_shinfo(skb)->gso_type;
68
69                  if (unlikely(type &
70                               ~(SKB_GSO_TCPV4 |
71                                 SKB_GSO_DODGY |
72                                 SKB_GSO_TCP_ECN |
73                                 SKB_GSO_TCPV6 |
74                                 SKB_GSO_GRE |
75                                 SKB_GSO_GRE_CSUM |
76                                 SKB_GSO_IPIP |
77                                 SKB_GSO_SIT |
78                                 SKB_GSO_MPLS |
79                                 SKB_GSO_UDP_TUNNEL |
80                                 SKB_GSO_UDP_TUNNEL_CSUM |
81                                 0) ||
82                               !(type & (SKB_GSO_TCPV4 | SKB_GSO_TCPV6))))
83                          goto out;
84
85                  skb_shinfo(skb)->gso_segs = DIV_ROUND_UP(skb->len, mss);
86
87                  segs = NULL;
88                  goto out;
89          }
90
91          copy_destructor = gso_skb->destructor == tcp_wfree;
92          ooo_okay = gso_skb->ooo_okay;
93          /* All segments but the first should have ooo_okay cleared */
94          skb->ooo_okay = 0;
95
96          segs = skb_segment(skb, features);
97          if (IS_ERR(segs))
98                  goto out;
99
100         /* Only first segment might have ooo_okay set */
101         segs->ooo_okay = ooo_okay;
102
103         delta = htonl(oldlen + (thlen + mss));
104
105         skb = segs;
106         th = tcp_hdr(skb);
107         seq = ntohl(th->seq);
108
```

```
109         if (unlikely(skb_shinfo(gso_skb)->tx_flags & SKBTX_SW_TSTAMP))
110                 tcp_gso_tstamp(segs, skb_shinfo(gso_skb)->tskey, seq, mss);
111
112         newcheck = ~csum_fold((__force __wsum)((__force u32)th->check +
113                                                (__force u32)delta));
114
115         do {
116                 th->fin = th->psh = 0;
117                 th->check = newcheck;
118
119                 if (skb->ip_summed != CHECKSUM_PARTIAL)
120                         th->check = gso_make_checksum(skb, ~th->check);
121
122                 seq += mss;
123                 if (copy_destructor) {
124                         skb->destructor = gso_skb->destructor;
125                         skb->sk = gso_skb->sk;
126                         sum_truesize += skb->truesize;
127                 }
128                 skb = skb->next;
129                 th = tcp_hdr(skb);
130
131                 th->seq = htonl(seq);
132                 th->cwr = 0;
133         } while (skb->next);
134
135         /* Following permits TCP Small Queues to work well with GSO :
136          * The callback to TCP stack will be called at the time last frag
137          * is freed at TX completion, and not right now when gso_skb
138          * is freed by GSO engine
139          */
140         if (copy_destructor) {
141                 swap(gso_skb->sk, skb->sk);
142                 swap(gso_skb->destructor, skb->destructor);
143                 sum_truesize += skb->truesize;
144                 atomic_add(sum_truesize - gso_skb->truesize,
145                            &skb->sk->sk_wmem_alloc);
146         }
147
148         delta = htonl(oldlen + (skb_tail_pointer(skb) -
149                                 skb_transport_header(skb)) +
150                       skb->data_len);
151         th->check = ~csum_fold((__force __wsum)((__force u32)th->check +
152                                 (__force u32)delta));
153         if (skb->ip_summed != CHECKSUM_PARTIAL)
154                 th->check = gso_make_checksum(skb, ~th->check);
155 out:
156         return segs;
157 }
158
159 struct sk_buff **tcp_gro_receive(struct sk_buff **head, struct sk_buff *skb)
160 {
161         struct sk_buff **pp = NULL;
162         struct sk_buff *p;
163         struct tcphdr *th;
164         struct tcphdr *th2;
165         unsigned int len;
166         unsigned int thlen;
167         __be32 flags;
168         unsigned int mss = 1;
169         unsigned int hlen;
170         unsigned int off;
171         int flush = 1;
172         int i;
173
```

```
174         off = skb_gro_offset(skb);
175         hlen = off + sizeof(*th);
176         th = skb_gro_header_fast(skb, off);
177         if (skb_gro_header_hard(skb, hlen)) {
178                 th = skb_gro_header_slow(skb, hlen, off);
179                 if (unlikely(!th))
180                         goto out;
181         }
182
183         thlen = th->doff * 4;
184         if (thlen < sizeof(*th))
185                 goto out;
186
187         hlen = off + thlen;
188         if (skb_gro_header_hard(skb, hlen)) {
189                 th = skb_gro_header_slow(skb, hlen, off);
190                 if (unlikely(!th))
191                         goto out;
192         }
193
194         skb_gro_pull(skb, thlen);
195
196         len = skb_gro_len(skb);
197         flags = tcp_flag_word(th);
198
199         for (; (p = *head); head = &p->next) {
200                 if (!NAPI_GRO_CB(p)->same_flow)
201                         continue;
202
203                 th2 = tcp_hdr(p);
204
205                 if (*(u32 *)&th->source ^ *(u32 *)&th2->source) {
206                         NAPI_GRO_CB(p)->same_flow = 0;
207                         continue;
208                 }
209
210                 goto found;
211         }
212
213         goto out_check_final;
214
215 found:
216         /* Include the IP ID check below from the inner most IP hdr */
217         flush = NAPI_GRO_CB(p)->flush | NAPI_GRO_CB(p)->flush_id;
218         flush |= (__force int)(flags & TCP_FLAG_CWR);
219         flush |= (__force int)((flags ^ tcp_flag_word(th2)) &
220                   ~(TCP_FLAG_CWR | TCP_FLAG_FIN | TCP_FLAG_PSH));
221         flush |= (__force int)(th->ack_seq ^ th2->ack_seq);
222         for (i = sizeof(*th); i < thlen; i += 4)
223                 flush |= *(u32 *)((u8 *)th + i) ^
224                          *(u32 *)((u8 *)th2 + i);
225
226         mss = tcp_skb_mss(p);
227
228         flush |= (len - 1) >= mss;
229         flush |= (ntohl(th2->seq) + skb_gro_len(p)) ^ ntohl(th->seq);
230
231         if (flush || skb_gro_receive(head, skb)) {
232                 mss = 1;
233                 goto out_check_final;
234         }
235
236         p = *head;
237         th2 = tcp_hdr(p);
238         tcp_flag_word(th2) |= flags & (TCP_FLAG_FIN | TCP_FLAG_PSH);
```

```
239
240 out_check_final:
241         flush = len < mss;
242         flush |= (__force int)(flags & (TCP_FLAG_URG | TCP_FLAG_PSH |
243                                         TCP_FLAG_RST | TCP_FLAG_SYN |
244                                         TCP_FLAG_FIN));
245
246         if (p && (!NAPI_GRO_CB(skb)->same_flow || flush))
247                 pp = head;
248
249 out:
250         NAPI_GRO_CB(skb)->flush |= (flush != 0);
251
252         return pp;
253 }
254
255 int tcp_gro_complete(struct sk_buff *skb)
256 {
257         struct tcphdr *th = tcp_hdr(skb);
258
259         skb->csum_start = (unsigned char *)th - skb->head;
260         skb->csum_offset = offsetof(struct tcphdr, check);
261         skb->ip_summed = CHECKSUM_PARTIAL;
262
263         skb_shinfo(skb)->gso_segs = NAPI_GRO_CB(skb)->count;
264
265         if (th->cwr)
266                 skb_shinfo(skb)->gso_type |= SKB_GSO_TCP_ECN;
267
268         return 0;
269 }
270 EXPORT_SYMBOL(tcp_gro_complete);
271
272 static int tcp_v4_gso_send_check(struct sk_buff *skb)
273 {
274         const struct iphdr *iph;
275         struct tcphdr *th;
276
277         if (!pskb_may_pull(skb, sizeof(*th)))
278                 return -EINVAL;
279
280         iph = ip_hdr(skb);
281         th = tcp_hdr(skb);
282
283         th->check = 0;
284         skb->ip_summed = CHECKSUM_PARTIAL;
285         __tcp_v4_send_check(skb, iph->saddr, iph->daddr);
286         return 0;
287 }
288
289 static struct sk_buff **tcp4_gro_receive(struct sk_buff **head, struct sk_buff *skb)
290 {
291         /* Use the IP hdr immediately proceeding for this transport */
292         const struct iphdr *iph = skb_gro_network_header(skb);
293         __wsum wsum;
294
295         /* Don't bother verifying checksum if we're going to flush anyway. */
296         if (NAPI_GRO_CB(skb)->flush)
297                 goto skip_csum;
298
299         wsum = NAPI_GRO_CB(skb)->csum;
300
301         switch (skb->ip_summed) {
302         case CHECKSUM_NONE:
303                 wsum = skb_checksum(skb, skb_gro_offset(skb), skb_gro_len(skb),
```

```
                                        0);

                /* fall through */

        case CHECKSUM_COMPLETE:
                if (!tcp_v4_check(skb_gro_len(skb), iph->saddr, iph->daddr,
                                  wsum)) {
                        skb->ip_summed = CHECKSUM_UNNECESSARY;
                        break;
                }

                NAPI_GRO_CB(skb)->flush = 1;
                return NULL;
        }

skip_csum:
        return tcp_gro_receive(head, skb);
}

static int tcp4_gro_complete(struct sk_buff *skb, int thoff)
{
        const struct iphdr *iph = ip_hdr(skb);
        struct tcphdr *th = tcp_hdr(skb);

        th->check = ~tcp_v4_check(skb->len - thoff, iph->saddr,
                                  iph->daddr, 0);
        skb_shinfo(skb)->gso_type |= SKB_GSO_TCPV4;

        return tcp_gro_complete(skb);
}

static const struct net_offload tcpv4_offload = {
        .callbacks = {
                .gso_send_check =       tcp_v4_gso_send_check,
                .gso_segment    =       tcp_gso_segment,
                .gro_receive    =       tcp4_gro_receive,
                .gro_complete   =       tcp4_gro_complete,
        },
};

int __init tcpv4_offload_init(void)
{
        return inet_add_offload(&tcpv4_offload, IPPROTO_TCP);
}

```