Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_fastopen.c

```
 1 #include <linux/err.h>
 2 #include <linux/init.h>
 3 #include <linux/kernel.h>
 4 #include <linux/list.h>
 5 #include <linux/tcp.h>
 6 #include <linux/rcupdate.h>
 7 #include <linux/rculist.h>
 8 #include <net/inetpeer.h>
 9 #include <net/tcp.h>
10
11 int sysctl_tcp_fastopen __read_mostly = TFO_CLIENT_ENABLE;
12
13 struct tcp_fastopen_context __rcu *tcp_fastopen_ctx;
14
15 static DEFINE_SPINLOCK(tcp_fastopen_ctx_lock);
16
17 void tcp_fastopen_init_key_once(bool publish)
18 {
19         static u8 key[TCP_FASTOPEN_KEY_LENGTH];
20
21         /* tcp_fastopen_reset_cipher publishes the new context
22          * atomically, so we allow this race happening here.
23          *
24          * All call sites of tcp_fastopen_cookie_gen also check
25          * for a valid cookie, so this is an acceptable risk.
26          */
27         if (net_get_random_once(key, sizeof(key)) && publish)
28                 tcp_fastopen_reset_cipher(key, sizeof(key));
29 }
30
31 static void tcp_fastopen_ctx_free(struct rcu_head *head)
32 {
33         struct tcp_fastopen_context *ctx =
34             container_of(head, struct tcp_fastopen_context, rcu);
35         crypto_free_cipher(ctx->tfm);
36         kfree(ctx);
37 }
38
39 int tcp_fastopen_reset_cipher(void *key, unsigned int len)
40 {
41         int err;
42         struct tcp_fastopen_context *ctx, *octx;
43
```

```
 44         ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
 45         if (!ctx)
 46                 return -ENOMEM;
 47         ctx->tfm = crypto_alloc_cipher("aes", 0, 0);
 48
 49         if (IS_ERR(ctx->tfm)) {
 50                 err = PTR_ERR(ctx->tfm);
 51 error:          kfree(ctx);
 52                 pr_err("TCP: TFO aes cipher alloc error: %d\n", err);
 53                 return err;
 54         }
 55         err = crypto_cipher_setkey(ctx->tfm, key, len);
 56         if (err) {
 57                 pr_err("TCP: TFO cipher key error: %d\n", err);
 58                 crypto_free_cipher(ctx->tfm);
 59                 goto error;
 60         }
 61         memcpy(ctx->key, key, len);
 62
 63         spin_lock(&tcp_fastopen_ctx_lock);
 64
 65         octx = rcu_dereference_protected(tcp_fastopen_ctx,
 66                                          lockdep_is_held(&tcp_fastopen_ctx_lock));
 67         rcu_assign_pointer(tcp_fastopen_ctx, ctx);
 68         spin_unlock(&tcp_fastopen_ctx_lock);
 69
 70         if (octx)
 71                 call_rcu(&octx->rcu, tcp_fastopen_ctx_free);
 72         return err;
 73 }
 74
 75 static bool __tcp_fastopen_cookie_gen(const void *path,
 76                                       struct tcp_fastopen_cookie *foc)
 77 {
 78         struct tcp_fastopen_context *ctx;
 79         bool ok = false;
 80
 81         tcp_fastopen_init_key_once(true);
 82
 83         rcu_read_lock();
 84         ctx = rcu_dereference(tcp_fastopen_ctx);
 85         if (ctx) {
 86                 crypto_cipher_encrypt_one(ctx->tfm, foc->val, path);
 87                 foc->len = TCP_FASTOPEN_COOKIE_SIZE;
 88                 ok = true;
 89         }
 90         rcu_read_unlock();
 91         return ok;
 92 }
 93
 94 /* Generate the fastopen cookie by doing aes128 encryption on both
 95  * the source and destination addresses. Pad 0s for IPv4 or IPv4-mapped-IPv6
 96  * addresses. For the longer IPv6 addresses use CBC-MAC.
 97  *
 98  * XXX (TFO) - refactor when TCP_FASTOPEN_COOKIE_SIZE != AES_BLOCK_SIZE.
 99  */
100 static bool tcp_fastopen_cookie_gen(struct request_sock *req,
101                                     struct sk_buff *syn,
102                                     struct tcp_fastopen_cookie *foc)
103 {
104         if (req->rsk_ops->family == AF_INET) {
105                 const struct iphdr *iph = ip_hdr(syn);
106
107                 __be32 path[4] = { iph->saddr, iph->daddr, 0, 0 };
108                 return __tcp_fastopen_cookie_gen(path, foc);
```

```
109         }
110
111 #if IS_ENABLED(CONFIG_IPV6)
112         if (req->rsk_ops->family == AF_INET6) {
113                 const struct ipv6hdr *ip6h = ipv6_hdr(syn);
114                 struct tcp_fastopen_cookie tmp;
115
116                 if (__tcp_fastopen_cookie_gen(&ip6h->saddr, &tmp)) {
117                         struct in6_addr *buf = (struct in6_addr *) tmp.val;
118                         int i = 4;
119
120                         for (i = 0; i < 4; i++)
121                                 buf->s6_addr32[i] ^= ip6h->daddr.s6_addr32[i];
122                         return __tcp_fastopen_cookie_gen(buf, foc);
123                 }
124         }
125 #endif
126         return false;
127 }
128
129 static bool tcp_fastopen_create_child(struct sock *sk,
130                                       struct sk_buff *skb,
131                                       struct dst_entry *dst,
132                                       struct request_sock *req)
133 {
134         struct tcp_sock *tp;
135         struct request_sock_queue *queue = &inet_csk(sk)->icsk_accept_queue;
136         struct sock *child;
137
138         req->num_retrans = 0;
139         req->num_timeout = 0;
140         req->sk = NULL;
141
142         child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
143         if (child == NULL)
144                 return false;
145
146         spin_lock(&queue->fastopenq->lock);
147         queue->fastopenq->qlen++;
148         spin_unlock(&queue->fastopenq->lock);
149
150         /* Initialize the child socket. Have to fix some values to take
151          * into account the child is a Fast Open socket and is created
152          * only out of the bits carried in the SYN packet.
153          */
154         tp = tcp_sk(child);
155
156         tp->fastopen_rsk = req;
157         /* Do a hold on the listner sk so that if the listener is being
158          * closed, the child that has been accepted can live on and still
159          * access listen_lock.
160          */
161         sock_hold(sk);
162         tcp_rsk(req)->listener = sk;
163
164         /* RFC1323: The window in SYN & SYN/ACK segments is never
165          * scaled. So correct it appropriately.
166          */
167         tp->snd_wnd = ntohs(tcp_hdr(skb)->window);
168
169         /* Activate the retrans timer so that SYNACK can be retransmitted.
170          * The request socket is not added to the SYN table of the parent
171          * because it's been added to the accept queue directly.
172          */
173         inet_csk_reset_xmit_timer(child, ICSK_TIME_RETRANS,
```

```
174                                     TCP_TIMEOUT_INIT, TCP_RTO_MAX);
175
176         /* Add the child socket directly into the accept queue */
177         inet_csk_reqsk_queue_add(sk, req, child);
178
179         /* Now finish processing the fastopen child socket. */
180         inet_csk(child)->icsk_af_ops->rebuild_header(child);
181         tcp_init_congestion_control(child);
182         tcp_mtup_init(child);
183         tcp_init_metrics(child);
184         tcp_init_buffer_space(child);
185
186         /* Queue the data carried in the SYN packet. We need to first
187          * bump skb's refcnt because the caller will attempt to free it.
188          *
189          * XXX (TFO) - we honor a zero-payload TFO request for now,
190          * (any reason not to?) but no need to queue the skb since
191          * there is no data. How about SYN+FIN?
192          */
193         if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq + 1) {
194                 skb = skb_get(skb);
195                 skb_dst_drop(skb);
196                 __skb_pull(skb, tcp_hdr(skb)->doff * 4);
197                 skb_set_owner_r(skb, child);
198                 __skb_queue_tail(&child->sk_receive_queue, skb);
199                 tp->syn_data_acked = 1;
200         }
201         tcp_rsk(req)->rcv_nxt = tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
202         sk->sk_data_ready(sk);
203         bh_unlock_sock(child);
204         sock_put(child);
205         WARN_ON(req->sk == NULL);
206         return true;
207 }
208 EXPORT_SYMBOL(tcp_fastopen_create_child);
209
210 static bool tcp_fastopen_queue_check(struct sock *sk)
211 {
212         struct fastopen_queue *fastopenq;
213
214         /* Make sure the listener has enabled fastopen, and we don't
215          * exceed the max # of pending TFO requests allowed before trying
216          * to validating the cookie in order to avoid burning CPU cycles
217          * unnecessarily.
218          *
219          * XXX (TFO) - The implication of checking the max_qlen before
220          * processing a cookie request is that clients can't differentiate
221          * between qlen overflow causing Fast Open to be disabled
222          * temporarily vs a server not supporting Fast Open at all.
223          */
224         fastopenq = inet_csk(sk)->icsk_accept_queue.fastopenq;
225         if (fastopenq == NULL || fastopenq->max_qlen == 0)
226                 return false;
227
228         if (fastopenq->qlen >= fastopenq->max_qlen) {
229                 struct request_sock *req1;
230                 spin_lock(&fastopenq->lock);
231                 req1 = fastopenq->rskq_rst_head;
232                 if ((req1 == NULL) || time_after(req1->expires, jiffies)) {
233                         spin_unlock(&fastopenq->lock);
234                         NET_INC_STATS_BH(sock_net(sk),
235                                          LINUX_MIB_TCPFASTOPENLISTENOVERFLOW);
236                         return false;
237                 }
238                 fastopenq->rskq_rst_head = req1->dl_next;
```

```
239                 fastopenq->qlen--;
240                 spin_unlock(&fastopenq->lock);
241                 reqsk_free(req1);
242         }
243         return true;
244 }
245
246 /* Returns true if we should perform Fast Open on the SYN. The cookie (foc)
247  * may be updated and return the client in the SYN-ACK later. E.g., Fast Open
248  * cookie request (foc->len == 0).
249  */
250 bool tcp_try_fastopen(struct sock *sk, struct sk_buff *skb,
251                       struct request_sock *req,
252                       struct tcp_fastopen_cookie *foc,
253                       struct dst_entry *dst)
254 {
255         struct tcp_fastopen_cookie valid_foc = { .len = -1 };
256         bool syn_data = TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq + 1;
257
258         if (!((sysctl_tcp_fastopen & TFO_SERVER_ENABLE) &&
259               (syn_data || foc->len >= 0) &&
260               tcp_fastopen_queue_check(sk))) {
261                 foc->len = -1;
262                 return false;
263         }
264
265         if (syn_data && (sysctl_tcp_fastopen & TFO_SERVER_COOKIE_NOT_REQD))
266                 goto fastopen;
267
268         if (tcp_fastopen_cookie_gen(req, skb, &valid_foc) &&
269             foc->len == TCP_FASTOPEN_COOKIE_SIZE &&
270             foc->len == valid_foc.len &&
271             !memcmp(foc->val, valid_foc.val, foc->len)) {
272                 /* Cookie is valid. Create a (full) child socket to accept
273                  * the data in SYN before returning a SYN-ACK to ack the
274                  * data. If we fail to create the socket, fall back and
275                  * ack the ISN only but includes the same cookie.
276                  *
277                  * Note: Data-less SYN with valid cookie is allowed to send
278                  * data in SYN_RECV state.
279                  */
280 fastopen:
281                 if (tcp_fastopen_create_child(sk, skb, dst, req)) {
282                         foc->len = -1;
283                         NET_INC_STATS_BH(sock_net(sk),
284                                          LINUX_MIB_TCPFASTOPENPASSIVE);
285                         return true;
286                 }
287         }
288
289         NET_INC_STATS_BH(sock_net(sk), foc->len ?
290                          LINUX_MIB_TCPFASTOPENPASSIVEFAIL :
291                          LINUX_MIB_TCPFASTOPENCOOKIEREQD);
292         *foc = valid_foc;
293         return false;
294 }
295 EXPORT_SYMBOL(tcp_try_fastopen);
296
```