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_minisocks.c

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Implementation of the Transmission Control Protocol(TCP).
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Mark Evans, <evansmp@uhura.aston.ac.uk>
 *              Corey Minyard <wf-rch!minyard@relay.EU.net>
 *              Florian La Roche, <flla@stud.uni-sb.de>
 *              Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
 *              Linus Torvalds, <torvalds@cs.helsinki.fi>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *              Matthew Dillon, <dillon@apollo.west.oic.com>
 *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
 *              Jorge Cwik, <jorge@laser.satlink.net>
 */

#include <linux/mm.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/sysctl.h>
#include <linux/workqueue.h>
#include <net/tcp.h>
#include <net/inet_common.h>
#include <net/xfrm.h>

int sysctl_tcp_syncookies __read_mostly = 1;
EXPORT_SYMBOL(sysctl_tcp_syncookies);

int sysctl_tcp_abort_on_overflow __read_mostly;

struct inet_timewait_death_row tcp_death_row = {
        .sysctl_max_tw_buckets = NR_FILE * 2,
        .period         = TCP_TIMEWAIT_LEN / INET_TWDR_TWKILL_SLOTS,
        .death_lock     = __SPIN_LOCK_UNLOCKED(tcp_death_row.death_lock),
        .hashinfo       = &tcp_hashinfo,
        .tw_timer       = TIMER_INITIALIZER(inet_twdr_hangman, 0,
                                            (unsigned long)&tcp_death_row),
        .twkill_work    = __WORK_INITIALIZER(tcp_death_row.twkill_work,
                                             inet_twdr_twkill_work),
/* Short-time timewait calendar */

        .twcal_hand     = -1,
        .twcal_timer    = TIMER_INITIALIZER(inet_twdr_twcal_tick, 0,
                                            (unsigned long)&tcp_death_row),
};
EXPORT_SYMBOL_GPL(tcp_death_row);

static bool tcp_in_window(u32 seq, u32 end_seq, u32 s_win, u32 e_win)
{
        if (seq == s_win)
                return true;
        if (after(end_seq, s_win) && before(seq, e_win))
                return true;
        return seq == e_win && seq == end_seq;
```

```
}

/*
 * * Main purpose of TIME-WAIT state is to close connection gracefully,
 *   when one of ends sits in LAST-ACK or CLOSING retransmitting FIN
 *   (and, probably, tail of data) and one or more our ACKs are lost.
 * * What is TIME-WAIT timeout? It is associated with maximal packet
 *   lifetime in the internet, which results in wrong conclusion, that
 *   it is set to catch "old duplicate segments" wandering out of their path.
 *   It is not quite correct. This timeout is calculated so that it exceeds
 *   maximal retransmission timeout enough to allow to lose one (or more)
 *   segments sent by peer and our ACKs. This time may be calculated from RTO.
 * * When TIME-WAIT socket receives RST, it means that another end
 *   finally closed and we are allowed to kill TIME-WAIT too.
 * * Second purpose of TIME-WAIT is catching old duplicate segments.
 *   Well, certainly it is pure paranoia, but if we load TIME-WAIT
 *   with this semantics, we MUST NOT kill TIME-WAIT state with RSTs.
 * * If we invented some more clever way to catch duplicates
 *   (f.e. based on PAWS), we could truncate TIME-WAIT to several RTOs.
 *
 * The algorithm below is based on FORMAL INTERPRETATION of RFCs.
 * When you compare it to RFCs, please, read section SEGMENT ARRIVES
 * from the very beginning.
 *
 * NOTE. With recycling (and later with fin-wait-2) TW bucket
 * is _not_ stateless. It means, that strictly speaking we must
 * spinlock it. I do not want! Well, probability of misbehaviour
 * is ridiculously low and, seems, we could use some mb() tricks
 * to avoid misread sequence numbers, states etc.  --ANK
 *
 * We don't need to initialize tmp_out.sack_ok as we don't use the results
 */
enum tcp_tw_status
tcp_timewait_state_process(struct inet_timewait_sock *tw, struct sk_buff *skb,
                           const struct tcphdr *th)
{
        struct tcp_options_received tmp_opt;
        struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
        bool paws_reject = false;

        tmp_opt.saw_tstamp = 0;
        if (th->doff > (sizeof(*th) >> 2) && tcptw->tw_ts_recent_stamp) {
                tcp_parse_options(skb, &tmp_opt, 0, NULL);

                if (tmp_opt.saw_tstamp) {
                        tmp_opt.rcv_tsecr       -= tcptw->tw_ts_offset;
                        tmp_opt.ts_recent       = tcptw->tw_ts_recent;
                        tmp_opt.ts_recent_stamp = tcptw->tw_ts_recent_stamp;
                        paws_reject = tcp_paws_reject(&tmp_opt, th->rst);
                }
        }

        if (tw->tw_substate == TCP_FIN_WAIT2) {
                /* Just repeat all the checks of tcp_rcv_state_process() */

                /* Out of window, send ACK */
                if (paws_reject ||
                    !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
                                   tcptw->tw_rcv_nxt,
                                   tcptw->tw_rcv_nxt + tcptw->tw_rcv_wnd))
                        return TCP_TW_ACK;

                if (th->rst)
                        goto kill;

                if (th->syn && !before(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt))
                        goto kill_with_rst;

                /* Dup ACK? */
                if (!th->ack ||
                    !after(TCP_SKB_CB(skb)->end_seq, tcptw->tw_rcv_nxt) ||
                    TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq) {
                        inet_twsk_put(tw);
                        return TCP_TW_SUCCESS;
                }

                /* New data or FIN. If new data arrive after half-duplex close,
                 * reset.
                 */
                if (!th->fin ||
                    TCP_SKB_CB(skb)->end_seq != tcptw->tw_rcv_nxt + 1) {
kill_with_rst:
                        inet_twsk_deschedule(tw, &tcp_death_row);
                        inet_twsk_put(tw);
                        return TCP_TW_RST;
                }

                /* FIN arrived, enter true time-wait state. */
                tw->tw_substate   = TCP_TIME_WAIT;
                tcptw->tw_rcv_nxt = TCP_SKB_CB(skb)->end_seq;
                if (tmp_opt.saw_tstamp) {
                        tcptw->tw_ts_recent_stamp = get_seconds();
                        tcptw->tw_ts_recent       = tmp_opt.rcv_tsval;
                }

                if (tcp_death_row.sysctl_tw_recycle &&
                    tcptw->tw_ts_recent_stamp &&
                    tcp_tw_remember_stamp(tw))
                        inet_twsk_schedule(tw, &tcp_death_row, tw->tw_timeout,
                                           TCP_TIMEWAIT_LEN);
                else
                        inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                                           TCP_TIMEWAIT_LEN);
                return TCP_TW_ACK;
        }

        /*
         *      Now real TIME-WAIT state.
         *
         *      RFC 1122:
         *      "When a connection is [...] on TIME-WAIT state [...]
         *      [a TCP] MAY accept a new SYN from the remote TCP to
         *      reopen the connection directly, if it:
         *
         *      (1)  assigns its initial sequence number for the new
         *      connection to be larger than the largest sequence
         *      number it used on the previous connection incarnation,
         *      and
         *
         *      (2)  returns to TIME-WAIT state if the SYN turns out
         *      to be an old duplicate".
         */

        if (!paws_reject &&
            (TCP_SKB_CB(skb)->seq == tcptw->tw_rcv_nxt &&
             (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq || th->rst))) {
                /* In window segment, it may be only reset or bare ack. */

                if (th->rst) {
                        /* This is TIME_WAIT assassination, in two flavors.
                         * Oh well... nobody has a sufficient solution to this
                         * protocol bug yet.
                         */
                        if (sysctl_tcp_rfc1337 == 0) {
kill:
                                inet_twsk_deschedule(tw, &tcp_death_row);
                                inet_twsk_put(tw);
                                return TCP_TW_SUCCESS;
                        }
                }
                inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                                   TCP_TIMEWAIT_LEN);

                if (tmp_opt.saw_tstamp) {
                        tcptw->tw_ts_recent       = tmp_opt.rcv_tsval;
                        tcptw->tw_ts_recent_stamp = get_seconds();
                }

                inet_twsk_put(tw);
                return TCP_TW_SUCCESS;
        }

        /* Out of window segment.

           All the segments are ACKed immediately.

           The only exception is new SYN. We accept it, if it is
           not old duplicate and we are not in danger to be killed
           by delayed old duplicates. RFC check is that it has
           newer sequence number works at rates <40Mbit/sec.
           However, if paws works, it is reliable AND even more,
           we even may relax silly seq space cutoff.

           RED-PEN: we violate main RFC requirement, if this SYN will appear
           old duplicate (i.e. we receive RST in reply to SYN-ACK),
           we must return socket to time-wait state. It is not good,
           but not fatal yet.
         */

        if (th->syn && !th->rst && !th->ack && !paws_reject &&
            (after(TCP_SKB_CB(skb)->seq, tcptw->tw_rcv_nxt) ||
             (tmp_opt.saw_tstamp &&
              (s32)(tcptw->tw_ts_recent - tmp_opt.rcv_tsval) < 0))) {
                u32 isn = tcptw->tw_snd_nxt + 65535 + 2;
                if (isn == 0)
                        isn++;
                TCP_SKB_CB(skb)->when = isn;
                return TCP_TW_SYN;
        }

        if (paws_reject)
                NET_INC_STATS_BH(twsk_net(tw), LINUX_MIB_PAWSESTABREJECTED);

        if (!th->rst) {
                /* In this case we must reset the TIMEWAIT timer.
                 *
                 * If it is ACKless SYN it may be both old duplicate
                 * and new good SYN with random sequence number <rcv_nxt.
                 * Do not reschedule in the last case.
                 */
                if (paws_reject || th->ack)
                        inet_twsk_schedule(tw, &tcp_death_row, TCP_TIMEWAIT_LEN,
                                           TCP_TIMEWAIT_LEN);

                /* Send ACK. Note, we do not put the bucket,
                 * it will be released by caller.
                 */
                return TCP_TW_ACK;
        }
        inet_twsk_put(tw);
        return TCP_TW_SUCCESS;
}
EXPORT_SYMBOL(tcp_timewait_state_process);

/*
 * Move a socket to time-wait or dead fin-wait-2 state.
 */
void tcp_time_wait(struct sock *sk, int state, int timeo)
{
        struct inet_timewait_sock *tw = NULL;
        const struct inet_connection_sock *icsk = inet_csk(sk);
        const struct tcp_sock *tp = tcp_sk(sk);
        bool recycle_ok = false;

        if (tcp_death_row.sysctl_tw_recycle && tp->rx_opt.ts_recent_stamp)
                recycle_ok = tcp_remember_stamp(sk);

        if (tcp_death_row.tw_count < tcp_death_row.sysctl_max_tw_buckets)
                tw = inet_twsk_alloc(sk, state);

        if (tw != NULL) {
                struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
                const int rto = (icsk->icsk_rto << 2) - (icsk->icsk_rto >> 1);
                struct inet_sock *inet = inet_sk(sk);

                tw->tw_transparent      = inet->transparent;
                tw->tw_rcv_wscale       = tp->rx_opt.rcv_wscale;
                tcptw->tw_rcv_nxt       = tp->rcv_nxt;
                tcptw->tw_snd_nxt       = tp->snd_nxt;
                tcptw->tw_rcv_wnd       = tcp_receive_window(tp);
                tcptw->tw_ts_recent     = tp->rx_opt.ts_recent;
                tcptw->tw_ts_recent_stamp = tp->rx_opt.ts_recent_stamp;
                tcptw->tw_ts_offset     = tp->tsoffset;

#if IS_ENABLED(CONFIG_IPV6)
                if (tw->tw_family == PF_INET6) {
                        struct ipv6_pinfo *np = inet6_sk(sk);

                        tw->tw_v6_daddr = sk->sk_v6_daddr;
                        tw->tw_v6_rcv_saddr = sk->sk_v6_rcv_saddr;
                        tw->tw_tclass = np->tclass;
                        tw->tw_flowlabel = np->flow_label >> 12;
                        tw->tw_ipv6only = sk->sk_ipv6only;
                }
#endif

#ifdef CONFIG_TCP_MD5SIG
                /*
                 * The timewait bucket does not have the key DB from the
                 * sock structure. We just make a quick copy of the
                 * md5 key being used (if indeed we are using one)
                 * so the timewait ack generating code has the key.
                 */
                do {
                        struct tcp_md5sig_key *key;
                        tcptw->tw_md5_key = NULL;
                        key = tp->af_specific->md5_lookup(sk, sk);
                        if (key != NULL) {
                                tcptw->tw_md5_key = kmemdup(key, sizeof(*key), GFP_ATOMIC);
                                if (tcptw->tw_md5_key && !tcp_alloc_md5sig_pool())
                                        BUG();
                        }
                } while (0);
#endif

                /* Linkage updates. */
                __inet_twsk_hashdance(tw, sk, &tcp_hashinfo);

                /* Get the TIME_WAIT timeout firing. */
                if (timeo < rto)
                        timeo = rto;

                if (recycle_ok) {
                        tw->tw_timeout = rto;
                } else {
                        tw->tw_timeout = TCP_TIMEWAIT_LEN;
                        if (state == TCP_TIME_WAIT)
                                timeo = TCP_TIMEWAIT_LEN;
                }

                inet_twsk_schedule(tw, &tcp_death_row, timeo,
                                   TCP_TIMEWAIT_LEN);
                inet_twsk_put(tw);
        } else {
                /* Sorry, if we're out of memory, just CLOSE this
                 * socket up.  We've got bigger problems than
                 * non-graceful socket closings.
                 */
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPTIMEWAITOVERFLOW);
        }

        tcp_update_metrics(sk);
        tcp_done(sk);
}

void tcp_twsk_destructor(struct sock *sk)
{
#ifdef CONFIG_TCP_MD5SIG
        struct tcp_timewait_sock *twsk = tcp_twsk(sk);

        if (twsk->tw_md5_key)
                kfree_rcu(twsk->tw_md5_key, rcu);
#endif
}
EXPORT_SYMBOL_GPL(tcp_twsk_destructor);

void tcp_openreq_init_rwin(struct request_sock *req,
                           struct sock *sk, struct dst_entry *dst)
{
        struct inet_request_sock *ireq = inet_rsk(req);
        struct tcp_sock *tp = tcp_sk(sk);
        __u8 rcv_wscale;
        int mss = dst_metric_advmss(dst);

        if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < mss)
                mss = tp->rx_opt.user_mss;

        /* Set this up on the first call only */
        req->window_clamp = tp->window_clamp ? : dst_metric(dst, RTAX_WINDOW);

        /* limit the window selection if the user enforce a smaller rx buffer */
        if (sk->sk_userlocks & SOCK_RCVBUF_LOCK &&
            (req->window_clamp > tcp_full_space(sk) || req->window_clamp == 0))
                req->window_clamp = tcp_full_space(sk);

        /* tcp_full_space because it is guaranteed to be the first packet */
        tcp_select_initial_window(tcp_full_space(sk),
                mss - (ireq->tstamp_ok ? TCPOLEN_TSTAMP_ALIGNED : 0),
                &req->rcv_wnd,
                &req->window_clamp,
                ireq->wscale_ok,
                &rcv_wscale,
                dst_metric(dst, RTAX_INITRWND));
        ireq->rcv_wscale = rcv_wscale;
}
EXPORT_SYMBOL(tcp_openreq_init_rwin);

static inline void TCP_ECN_openreq_child(struct tcp_sock *tp,
                                         struct request_sock *req)
{
        tp->ecn_flags = inet_rsk(req)->ecn_ok ? TCP_ECN_OK : 0;
}

/* This is not only more efficient than what we used to do, it eliminates
 * a lot of code duplication between IPv4/IPv6 SYN recv processing. -DaveM
 *
 * Actually, we could lots of memory writes here. tp of listening
 * socket contains all necessary default parameters.
 */
struct sock *tcp_create_openreq_child(struct sock *sk, struct request_sock *req, struct sk_buff *skb)
{
        struct sock *newsk = inet_csk_clone_lock(sk, req, GFP_ATOMIC);

        if (newsk != NULL) {
                const struct inet_request_sock *ireq = inet_rsk(req);
                struct tcp_request_sock *treq = tcp_rsk(req);
                struct inet_connection_sock *newicsk = inet_csk(newsk);
                struct tcp_sock *newtp = tcp_sk(newsk);

                /* Now setup tcp_sock */
                newtp->pred_flags = 0;

                newtp->rcv_wup = newtp->copied_seq =
                newtp->rcv_nxt = treq->rcv_isn + 1;

                newtp->snd_sml = newtp->snd_una =
                newtp->snd_nxt = newtp->snd_up = treq->snt_isn + 1;

                tcp_prequeue_init(newtp);
                INIT_LIST_HEAD(&newtp->tsq_node);

                tcp_init_wl(newtp, treq->rcv_isn);

                newtp->srtt_us = 0;
                newtp->mdev_us = jiffies_to_usecs(TCP_TIMEOUT_INIT);
                newicsk->icsk_rto = TCP_TIMEOUT_INIT;

                newtp->packets_out = 0;
                newtp->retrans_out = 0;
                newtp->sacked_out = 0;
                newtp->fackets_out = 0;
                newtp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
                tcp_enable_early_retrans(newtp);
                newtp->tlp_high_seq = 0;
                newtp->lsndtime = treq->snt_synack;
                newtp->total_retrans = req->num_retrans;

                /* So many TCP implementations out there (incorrectly) count the
                 * initial SYN frame in their delayed-ACK and congestion control
                 * algorithms that we must have the following bandaid to talk
                 * efficiently to them.  -DaveM
                 */
                newtp->snd_cwnd = TCP_INIT_CWND;
                newtp->snd_cwnd_cnt = 0;

                if (newicsk->icsk_ca_ops != &tcp_init_congestion_ops &&
                    !try_module_get(newicsk->icsk_ca_ops->owner))
                        newicsk->icsk_ca_ops = &tcp_init_congestion_ops;

                tcp_set_ca_state(newsk, TCP_CA_Open);
                tcp_init_xmit_timers(newsk);
                __skb_queue_head_init(&newtp->out_of_order_queue);
                newtp->write_seq = newtp->pushed_seq = treq->snt_isn + 1;

                newtp->rx_opt.saw_tstamp = 0;

                newtp->rx_opt.dsack = 0;
                newtp->rx_opt.num_sacks = 0;

                newtp->urg_data = 0;

                if (sock_flag(newsk, SOCK_KEEPOPEN))
                        inet_csk_reset_keepalive_timer(newsk,
                                                       keepalive_time_when(newtp));

                newtp->rx_opt.tstamp_ok = ireq->tstamp_ok;
                if ((newtp->rx_opt.sack_ok = ireq->sack_ok) != 0) {
                        if (sysctl_tcp_fack)
                                tcp_enable_fack(newtp);
                }
                newtp->window_clamp = req->window_clamp;
                newtp->rcv_ssthresh = req->rcv_wnd;
                newtp->rcv_wnd = req->rcv_wnd;
                newtp->rx_opt.wscale_ok = ireq->wscale_ok;
                if (newtp->rx_opt.wscale_ok) {
                        newtp->rx_opt.snd_wscale = ireq->snd_wscale;
                        newtp->rx_opt.rcv_wscale = ireq->rcv_wscale;
                } else {
                        newtp->rx_opt.snd_wscale = newtp->rx_opt.rcv_wscale = 0;
                        newtp->window_clamp = min(newtp->window_clamp, 65535U);
                }
                newtp->snd_wnd = (ntohs(tcp_hdr(skb)->window) <<
                                  newtp->rx_opt.snd_wscale);
                newtp->max_window = newtp->snd_wnd;

                if (newtp->rx_opt.tstamp_ok) {
                        newtp->rx_opt.ts_recent = req->ts_recent;
                        newtp->rx_opt.ts_recent_stamp = get_seconds();
                        newtp->tcp_header_len = sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
                } else {
                        newtp->rx_opt.ts_recent_stamp = 0;
                        newtp->tcp_header_len = sizeof(struct tcphdr);
                }
                newtp->tsoffset = 0;
#ifdef CONFIG_TCP_MD5SIG
                newtp->md5sig_info = NULL;      /*XXX*/
                if (newtp->af_specific->md5_lookup(sk, newsk))
                        newtp->tcp_header_len += TCPOLEN_MD5SIG_ALIGNED;
#endif
                if (skb->len >= TCP_MSS_DEFAULT + newtp->tcp_header_len)
                        newicsk->icsk_ack.last_seg_size = skb->len - newtp->tcp_header_len;
                newtp->rx_opt.mss_clamp = req->mss;
                TCP_ECN_openreq_child(newtp, req);
                newtp->fastopen_rsk = NULL;
                newtp->syn_data_acked = 0;

                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_PASSIVEOPENS);
        }
        return newsk;
}
EXPORT_SYMBOL(tcp_create_openreq_child);

/*
 * Process an incoming packet for SYN_RECV sockets represented as a
 * request_sock. Normally sk is the listener socket but for TFO it
 * points to the child socket.
 *
 * XXX (TFO) - The current impl contains a special check for ack
 * validation and inside tcp_v4_reqsk_send_ack(). Can we do better?
 *
 * We don't need to initialize tmp_opt.sack_ok as we don't use the results
 */

struct sock *tcp_check_req(struct sock *sk, struct sk_buff *skb,
                           struct request_sock *req,
                           struct request_sock **prev,
                           bool fastopen)
{
        struct tcp_options_received tmp_opt;
        struct sock *child;
        const struct tcphdr *th = tcp_hdr(skb);
        __be32 flg = tcp_flag_word(th) & (TCP_FLAG_RST|TCP_FLAG_SYN|TCP_FLAG_ACK);
        bool paws_reject = false;

        BUG_ON(fastopen == (sk->sk_state == TCP_LISTEN));

        tmp_opt.saw_tstamp = 0;
        if (th->doff > (sizeof(struct tcphdr)>>2)) {
                tcp_parse_options(skb, &tmp_opt, 0, NULL);

                if (tmp_opt.saw_tstamp) {
                        tmp_opt.ts_recent = req->ts_recent;
                        /* We do not store true stamp, but it is not required,
                         * it can be estimated (approximately)
                         * from another data.
                         */
                        tmp_opt.ts_recent_stamp = get_seconds() - ((TCP_TIMEOUT_INIT/HZ)<<req->num_timeout);
                        paws_reject = tcp_paws_reject(&tmp_opt, th->rst);
                }
        }

        /* Check for pure retransmitted SYN. */
        if (TCP_SKB_CB(skb)->seq == tcp_rsk(req)->rcv_isn &&
            flg == TCP_FLAG_SYN &&
            !paws_reject) {
                /*
                 * RFC793 draws (Incorrectly! It was fixed in RFC1122)
                 * this case on figure 6 and figure 8, but formal
                 * protocol description says NOTHING.
                 * To be more exact, it says that we should send ACK,
                 * because this segment (at least, if it has no data)
                 * is out of window.
                 *
                 *  CONCLUSION: RFC793 (even with RFC1122) DOES NOT
                 *  describe SYN-RECV state. All the description
                 *  is wrong, we cannot believe to it and should
                 *  rely only on common sense and implementation
                 *  experience.
                 *
                 * Enforce "SYN-ACK" according to figure 8, figure 6
                 * of RFC793, fixed by RFC1122.
                 *
                 * Note that even if there is new data in the SYN packet
                 * they will be thrown away too.
                 *
                 * Reset timer after retransmitting SYNACK, similar to
                 * the idea of fast retransmit in recovery.
                 */
                if (!inet_rtx_syn_ack(sk, req))
                        req->expires = min(TCP_TIMEOUT_INIT << req->num_timeout,
                                           TCP_RTO_MAX) + jiffies;
                return NULL;
        }

        /* Further reproduces section "SEGMENT ARRIVES"
           for state SYN-RECEIVED of RFC793.
           It is broken, however, it does not work only
           when SYNs are crossed.

           You would think that SYN crossing is impossible here, since
           we should have a SYN_SENT socket (from connect()) on our end,
           but this is not true if the crossed SYNs were sent to both
           ends by a malicious third party.  We must defend against this,
           and to do that we first verify the ACK (as per RFC793, page
           36) and reset if it is invalid.  Is this a true full defense?
           To convince ourselves, let us consider a way in which the ACK
           test can still pass in this 'malicious crossed SYNs' case.
           Malicious sender sends identical SYNs (and thus identical sequence
           numbers) to both A and B:

                A: gets SYN, seq=7
                B: gets SYN, seq=7

           By our good fortune, both A and B select the same initial
           send sequence number of seven :-)

                A: sends SYN|ACK, seq=7, ack_seq=8
                B: sends SYN|ACK, seq=7, ack_seq=8

           So we are now A eating this SYN|ACK, ACK test passes.  So
           does sequence test, SYN is truncated, and thus we consider
           it a bare ACK.

           If icsk->icsk_accept_queue.rskq_defer_accept, we silently drop this
           bare ACK.  Otherwise, we create an established connection.  Both
           ends (listening sockets) accept the new incoming connection and try
           to talk to each other. 8-)

           Note: This case is both harmless, and rare.  Possibility is about the
           same as us discovering intelligent life on another plant tomorrow.

           But generally, we should (RFC lies!) to accept ACK
           from SYNACK both here and in tcp_rcv_state_process().
           tcp_rcv_state_process() does not, hence, we do not too.

           Note that the case is absolutely generic:
           we cannot optimize anything here without
           violating protocol. All the checks must be made
           before attempt to create socket.
         */

        /* RFC793 page 36: "If the connection is in any non-synchronized state ...
         *                  and the incoming segment acknowledges something not yet
         *                  sent (the segment carries an unacceptable ACK) ...
         *                  a reset is sent."
         *
         * Invalid ACK: reset will be sent by listening socket.
         * Note that the ACK validity check for a Fast Open socket is done
         * elsewhere and is checked directly against the child socket rather
         * than req because user data may have been sent out.
         */
        if ((flg & TCP_FLAG_ACK) && !fastopen &&
            (TCP_SKB_CB(skb)->ack_seq !=
             tcp_rsk(req)->snt_isn + 1))
                return sk;

        /* Also, it would be not so bad idea to check rcv_tsecr, which
         * is essentially ACK extension and too early or too late values
         * should cause reset in unsynchronized states.
         */

        /* RFC793: "first check sequence number". */

        if (paws_reject || !tcp_in_window(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
                                          tcp_rsk(req)->rcv_nxt, tcp_rsk(req)->rcv_nxt + req->rcv_wnd)) {
                /* Out of window: send ACK and drop. */
                if (!(flg & TCP_FLAG_RST))
                        req->rsk_ops->send_ack(sk, skb, req);
                if (paws_reject)
                        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSESTABREJECTED);
                return NULL;
        }

        /* In sequence, PAWS is OK. */

        if (tmp_opt.saw_tstamp && !after(TCP_SKB_CB(skb)->seq, tcp_rsk(req)->rcv_nxt))
                req->ts_recent = tmp_opt.rcv_tsval;

        if (TCP_SKB_CB(skb)->seq == tcp_rsk(req)->rcv_isn) {
                /* Truncate SYN, it is out of window starting
                   at tcp_rsk(req)->rcv_isn + 1. */
                flg &= ~TCP_FLAG_SYN;
        }

        /* RFC793: "second check the RST bit" and
         *         "fourth, check the SYN bit"
         */
        if (flg & (TCP_FLAG_RST|TCP_FLAG_SYN)) {
                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_ATTEMPTFAILS);
                goto embryonic_reset;
        }

        /* ACK sequence verified above, just make sure ACK is
         * set.  If ACK not set, just silently drop the packet.
         *
         * XXX (TFO) - if we ever allow "data after SYN", the
         * following check needs to be removed.
         */
        if (!(flg & TCP_FLAG_ACK))
                return NULL;

        /* For Fast Open no more processing is needed (sk is the
         * child socket).
         */
        if (fastopen)
                return sk;

        /* While TCP_DEFER_ACCEPT is active, drop bare ACK. */
        if (req->num_timeout < inet_csk(sk)->icsk_accept_queue.rskq_defer_accept &&
            TCP_SKB_CB(skb)->end_seq == tcp_rsk(req)->rcv_isn + 1) {
                inet_rsk(req)->acked = 1;
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPDEFERACCEPTDROP);
                return NULL;
        }

        /* OK, ACK is valid, create big socket and
         * feed this segment to it. It will repeat all
         * the tests. THIS SEGMENT MUST MOVE SOCKET TO
         * ESTABLISHED STATE. If it will be dropped after
         * socket is created, wait for troubles.
         */
        child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
        if (child == NULL)
                goto listen_overflow;

        inet_csk_reqsk_queue_unlink(sk, req, prev);
        inet_csk_reqsk_queue_removed(sk, req);

        inet_csk_reqsk_queue_add(sk, req, child);
        return child;

listen_overflow:
        if (!sysctl_tcp_abort_on_overflow) {
                inet_rsk(req)->acked = 1;
                return NULL;
        }

embryonic_reset:
        if (!(flg & TCP_FLAG_RST)) {
                /* Received a bad SYN pkt - for TFO We try not to reset
                 * the local connection unless it's really necessary to
                 * avoid becoming vulnerable to outside attack aiming at
                 * resetting legit local connections.
                 */
                req->rsk_ops->send_reset(sk, skb);
        } else if (fastopen) { /* received a valid RST pkt */
                reqsk_fastopen_remove(sk, req, true);
                tcp_reset(sk);
        }
        if (!fastopen) {
                inet_csk_reqsk_queue_drop(sk, req, prev);
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_EMBRYONICRSTS);
        }
        return NULL;
}
EXPORT_SYMBOL(tcp_check_req);

/*
 * Queue segment on the new socket if the new socket is active,
 * otherwise we just shortcircuit this and continue with
 * the new socket.
 *
 * For the vast majority of cases child->sk_state will be TCP_SYN_RECV
 * when entering. But other states are possible due to a race condition
 * where after __inet_lookup_established() fails but before the listener
 * locked is obtained, other packets cause the same connection to
 * be created.
 */

int tcp_child_process(struct sock *parent, struct sock *child,
                      struct sk_buff *skb)
{
        int ret = 0;
        int state = child->sk_state;

        if (!sock_owned_by_user(child)) {
                ret = tcp_rcv_state_process(child, skb, tcp_hdr(skb),
                                            skb->len);
                /* Wakeup parent, send SIGIO */
                if (state == TCP_SYN_RECV && child->sk_state != state)
                        parent->sk_data_ready(parent);
        } else {
                /* Alas, it is possible again, because we do lookup
                 * in main socket hash table and lock on listening
                 * socket does not protect us more.
                 */
                __sk_add_backlog(child, skb);
        }

        bh_unlock_sock(child);
        sock_put(child);
        return ret;
}
EXPORT_SYMBOL(tcp_child_process);
```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog