UDP Receive Processing

UDP receive processing

The `udp_rcv()` function is defined in `net/ipv4/udp.c`. It is invoked by the IP layer when it is determined that the protocol is 17 (UDP). Its mission is to verify the integrity of the UDP packet and to queue one or more copies for delivery to multicast and broadcast sockets and at most one copy to a one unicast socket. At this point the value of `skb->len` is the size of the TPDU and the `skb->nh.iph` and `skb->th.uh` pointers have been properly set up by the IP layer. Your `cop_rcv()` routine will be called in this way. Therefore, you may safely cast the `skb->h.uh` pointer to a `cop` header pointer as is done on line 1133 on the next page.

```
1117 int udp_rcv(struct sk_buff *skb)
1118 {
1119    struct sock *sk;
1120    struct udphdr *uh;
1121    unsigned short uelen;
1122    struct rtable *rt = (struct rtable*)skb->dst;
1123    u32 saddr = skb->nh.iph->saddr;
1124    u32 daddr = skb->nh.iph->daddr;
1125    int len = skb->len;

1127    /*
1128     * Validate the packet and the UDP length.
1129     */
1130    if (!pskb_may_pull(skb, sizeof(struct udphdr)))
1131        goto no_header;
```
Packet length verification

The *len* field specifies the actual distance in bytes between *skb-*\texttt{data} and *skb-*\texttt{tail}. At the point *skb-*\texttt{data} points to the UDP header. The *ulen* field is the length of the UDP header plus data as set by the sender. If

- *ulen* is more than the length actually received or
- *ulen* is not even long enough for a full UDP header,

then a short packet condition is raised. **You should include this code.**

```c
    uh = skb->h.uh;
    ulen = ntohs(uh->len);
    if (ulen > len || ulen < sizeof(*uh))
        goto short_packet;
```

On the other hand, the size specified in *ulen* may be less than the actual data received. In this case an attempt is made to trim the packet down to size. If that succeeds, the trimmed packet is accepted. The *sk_buff* is trimmed to the size specified in the UDP header by the *pskb_trim()* function. For a linear *sk_buff*, the *pskb_trim()* function will set *skb-*\texttt{tail} to *skb-*\texttt{data} + *ulen*.

This function returns 0 on success and *ENOMEM* on failure. Failure is possible only if the buffer is non-linear.

```c
    if (pskb_trim_rcsum(skb, ulen))
        goto short_packet;
```
Initial processing

The \textit{udp_checksum_init()} function initializes the UDP checksum by setting it to the checksum of the UDP pseudo header. This is the pseudo header described in Steven's book.

\begin{verbatim}
1143    udp_checksum_init(skb, uh, ulen, saddr, daddr);
1144
\end{verbatim}

The pointer to the route cache element, \textit{rt}, is carried in \textit{skb->dst} where it was set during the call to route input. If the route type is \texttt{RTCF_BROADCAST} or \texttt{RTCF_MULTICAST}, packet delivery is handled by the function \textit{udp_v4_mcast_deliver()}. It is not possible for a packet to be delivered to both broadcast and unicast sockets.

\begin{verbatim}
1145    if(rt->rt_flags & (RTCF_BROADCAST | RTCF_MULTICAST))
1146       return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
1147
\end{verbatim}
Delivery of unicast packets

Reaching this point back in `udp_rcv` implies that a UNICAST packet is being processed. The `udp_v4_lookup` function is called to identify the UDP socket that best corresponds to the given source address, source port, destination address, destination port and device index of the interface on which the packet arrived.

```
1148    sk = udp_v4_lookup(saddr, uh->source, daddr, 
                      uh->dest, skb->dev->ifindex);
1149```

Upon finding a valid socket for a received packet, `udp_queue_rcv_skb()` is called to enqueue the `sk_buff` in the receive queue of the socket. If there exists insufficient space in the buffer quota of the socket the packet may be discarded here. The `sock_put()` here releases the reference that was obtained in `udp_v4_lookup()`.

The resubmit facility is new in kernel 2.6. We may see how it works later on, but treat it with extreme caution. If you return an inappropriate value after freeing an `sk_buff`, the buffer you freed may be reallocated while the `dev` layer continues to believe it owns it. This leads to segfaults on unrelated TCP connections.

```
1150    if (sk != NULL) {
1151       int ret = udp_queue_rcv_skb(sk, skb);
1152       sock_put(sk);
1153 
1154       /* a return value > 0 means to resubmit the input, but
1155        * it it wants the return to be -protocol, or 0
1156        */
1157       if (ret > 0)
1158          return -ret;
1159       return 0;
1160    }
```
Handling of undeliverable packets

Reaching this point means the packet is undeliverable because no socket can be matched to it. The xfrm system is a complicated conglomeration of policy and security based routing decisions that was introduced with SEL in kernel 2.6. Its not clear why it needs to be called for a doomed packet. You should try calling nf_reset() and see what happens!

```c
1162    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
1163       goto drop;
```

The netfilter facility contains a connection tracking mechanism in kernel 2.6. This causes the packet to hold a reference to a connection identifier structure. The reference is dropped by nf_reset.

```c
1164    nf_reset(skb);
1165
```

If control reaches this point, a valid socket for packet received was not found. In this case, udp_checksum_complete is called to verify the checksum. If there is a checksum error, the correct action is to discard the packet without sending an ICMP error message.

```c
1166 /* No socket. Drop packet silently, if checksum is wrong */
1167    if (udp_checksum_complete(skb))
1168       goto csum_error;
1169
```

If the checksum is correct, then an ICMP port unreachable error message is sent and packet is discarded. Your protocol should send the ICMP error message (but not touch the snmp data).

```c
1170    UDP_INC_STATS_BH(UDP_MIB_NOPORTS);
1171    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
1172
```

```c
1173    /*
1174    * Hmm. We got an UDP packet to a port to which we
1175    * don't wanna listen. Ignore it.
1176    */
1177    kfree_skb(skb);
1178    return(0);
1179
```
In case of any other errors, packet is discarded.

1180 short_packet:
1181   LIMIT_NETDEBUG(KERN_DEBUG "UDP: short packet:
1182       From %u.%u.%u.%u %d/%d to %u.%u.%u.%u
1183       ntohs(uh->source),
1184       ulen,
1185       len,
1186       ntohs(uh->dest));
1188 no_header:
1189   UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1190   kfree_skb(skb);
1191   return(0);
1192
1193 csum_error:
1194   /*
1195   * RFC1122: OK. Discards the bad packet silently (as far as
1196   * the network is concerned, anyway) as per 4.1.3.4 (MUST).
1197   */
1198   LIMIT_NETDEBUG(KERN_DEBUG "UDP: bad checksum.
1199       From %d.%d.%d.%d to %d.%d.%d.%d ul
1200       ntohs(uh->source),
1201       NIPQUAD(daddr),
1202       ntohs(uh->dest),
1203       ulen);
1204 drop:
1205   UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1206   kfree_skb(skb);
1207   return(0);
1208 }
Resetting the connection tracking reference

```c
1460 static inline void nf_reset(struct sk_buff *skb) {
1461    nf_conntrack_put(skb->nfct);
1462    skb->nfct = NULL;
1463    if (defined(CONFIG_NF_CONNTRACK) ||
1464        defined(CONFIG_NF_CONNTRACK_MODULE))
1465        nf_conntrack_put_reasm(skb->nfct_reasm);
1466    skb->nfct_reasm = NULL;
1467 #endif
1468 #ifdef CONFIG_BRIDGE_NETFILTER
1469    nf_bridge_put(skb->nf_bridge);
1470    skb->nf_bridge = NULL;
1471 #endif
1472}
```

```c
1426 static inline void nf_conntrack_put(struct nf_conntrack *nfct) {
1427    if (nfct && atomic_dec_and_test(&nfct->use))
1428        nfct->destroy(nfct);
1430 }
```
The xfrm facility for policy and security based routing

701 static inline int xfrm4_policy_check(struct sock *sk, int dir, 
    struct sk_buff *skb)
702 {
    return xfrm_policy_check(sk, dir, skb, AF_INET);
704 }

691 static inline int xfrm_policy_check(struct sock *sk, int dir, 
    struct sk_buff *skb, unsigned short family)
692 {
693     if (sk && sk->sk_policy[XFRM_POLICY_IN])
694         return __xfrm_policy_check(sk, dir, skb, family);
695
696     return (!xfrm_policy_list[dir] && !skb->sp) ||
697             (skb->dst->flags & DST_NOPOLICY) ||
698             __xfrm_policy_check(sk, dir, skb, family);
699 }

1055 int __xfrm_policy_check(struct sock *sk, int dir, struct sk_buff 
1056                      *skb, unsigned short family)
1057 {
1058     struct xfrm_policy *pol;
1059     struct flowi fl;
1060     u8 fl_dir = policy_to_flow_dir(dir);
1061     u32 sk_sid;
1062
1063     if (xfrm_decode_session(skb, &fl, family) < 0)
1064         return 0;
1065     nf_nat_decode_session(skb, &fl, family);
1066
1067     sk_sid = security_sk_sid(sk, &fl, fl_dir);
1068
1069    /* First, check used SA against their selectors. */
1070     if (skb->sp) {
1071         int i;
1072
1073         for (i=skb->sp->len-1; i>=0; i--) {
1074             struct xfrm_state *x = skb->sp->xvec[i];
1075             if (!xfrm_selector_match(&x->sel, &fl, family))
1076                 return 0;
1077         }
1078     }
1079 8
pol = NULL;
if (sk && sk->sk_policy[dir])
    pol = xfrm_sk_policy_lookup(sk, dir, &fl, sk_sid);
if (!pol)
    pol = flow_cache_lookup(&fl, sk_sid, family, fl_dir,
                        xfrm_policy_lookup);
if (!pol)
    return !skb->sp || !secpath_has_tunnel(skb->sp, 0);
pol->curlft.use_time = (unsigned long)xtime.tv_sec;
if (pol->action == XFRM_POLICY_ALLOW) {
    struct sec_path *sp;
    static struct sec_path dummy;
    int i, k;
    if ((sp = skb->sp) == NULL)
        sp = &dummy;
    /* For each tunnel xfrm, find the first matching tmpl.
     * For each tmpl before that, find corresponding xfrm.
     * Order is _important_. Later we will implement
     * some barriers, but at the moment barriers
     * are implied between each two transformations.
     */
    for (i = pol->xfrm_nr-1, k = 0; i >= 0; i--){
        k = xfrm_policy_ok(pol->xfrm_vec+i, sp, k, family);
        if (k < 0)
            goto reject;
    }
    if (secpath_has_tunnel(sp, k))
        goto reject;
    xfrm_pol_put(pol);
    return 1;
}
reject:
    xfrm_pol_put(pol);
    return 0;
Multicast and broadcast delivery

The `udp_v4_mcast_deliver()` function is defined in `net/ipv4/udp.c`. For a multicast/broadcast destination addresses, the packet is delivered to each socket that wants to receive it.

```c
1050 /*
1051  * Multicasts and broadcasts go to each listener.
1052  *
1053  * Note: called only from the BH handler context,
1054  * so we don't need to lock the hashes.
1055 */
1056 static int udp_v4_mcast_deliver(struct sk_buff *skb,
1057              struct udphdr *uh,
1058              u32 saddr, u32 daddr)
1059 {
1060    struct sock *sk;
1061    int dif;
1062    read_lock(&udp_hash_lock);
1063    sk = sk_head(&udp_hash[ntohs(uh->dest) &
1064                   (UDP_HTABLE_SIZE - 1)]);
1065    dif = skb->dev->ifindex;
1066    sk = udp_v4_mcast_next(sk, uh->dest, daddr, uh->source,
1067                         saddr, dif);
```
The multicast delivery loop

In this loop `udp_v4_mcast_next` is iteratively called to retrieve the next `struct sock` that is bound to this port. As has been seen earlier, it is necessary to determine if there is a next sock before it can be determined whether the packet must be cloned before being queued for this sock. However, the approach here is somewhat cleaner than others that are used.

```c
 1066   if (sk) {
 1067       struct sock *sknext = NULL;
 1068
 1069       do {
 1070          struct sk_buff *skb1 = skb;
 1071
 1072          sknext = udp_v4_mcast_next(sk_next(sk),
 1073                                       uh->dest, daddr,
 1074                                       uh->source, saddr, dif);
 1075          if(sknext)
 1076             skb1 = skb_clone(skb, GFP_ATOMIC);
 1077          if(skb1) {
 1078             int ret = udp_queue_rcv_skb(sk, skb1);
 1079             if (ret > 0)
 1080                /* we should probably re-process instead
 1081                   * of dropping packets here. */
 1082                kfree_skb(skb1);
 1083          }}
 1084       sk = sknext;
 1085     } while(sknext);
```

If the first call to `udp_v4_mcast_next()` returned 0, then the packet is not deliverable.

```c
 1086   } else
 1087       kfree_skb(skb);
 1088       read_unlock(&udp_hash_lock);
 1089       return 0;
 1090 }
```
Mulitcast socket lookup

The `udp_v4_mcast_next()` function is defined in `net/ipv4/udp.c`. The next matched socket in the given hash bucket is returned by this function. The matching logic unconditionally requires that the destination port match the port to which the socket is bound.

If any of

- local IP address,
- remote IP address,
- remote port address,
- or bound interface

are not zero they must match too. The call to `ip_mc_sf_allow()` is yet another filter system.

```c
281 static inline struct sock *udp_v4_mcast_next(struct sock *sk,
282                                            u16 loc_port, u32 loc_addr,
283                                            u16 rmt_port, u32 rmt_addr,
284                                            int dif)
285 {
286    struct hlist_node *node;
287    struct sock *s = sk;
288    unsigned short hnum = ntohs(loc_port);
289    sk_for_each_from(s, node) {
290       struct inet_sock *inet = inet_sk(s);
291       struct inet_sock *inet = inet_sk(s);
292       if (inet->num != hnum ||
293            (inet->daddr && inet->daddr != rmt_addr) ||
294            (inet->dport != rmt_port && inet->dport) ||
295            (inet->rcv_saddr && inet->rcv_saddr != loc_addr) ||
296            ipv6_only_sock(s) ||
297            (s->sk_bound_dev_if && s->sk_bound_dev_if != dif))
298       continue;
299       if (!ip_mc_sf_allow(s, loc_addr, rmt_addr, dif))
300       continue;
301       goto found;
302   }
303   s = NULL;
304   found:
305   return s;
306 }
```
The multicast filter

```
2156 /*
2157  * check if a multicast source filter allows delivery
2158  * for a given <src,dst,intf>
2159 */
2160 int ip_mc_sf_allow(struct sock *sk, u32 loc_addr,
2161             u32 rmt_addr, int dif)
2162 {
2163    struct inet_sock *inet = inet_sk(sk);
2164    struct ip_mc_socklist *pmc;
2165    struct ip_sf_socklist *psl;
2166    int i;
2167    if (!MULTICAST(loc_addr))
2168        return 1;
2169    for (pmc=inet->mc_list; pmc; pmc=pmc->next) {
2170        if (pmc->multi.imr_multiaddr.s_addr == loc_addr &&
2171            pmc->multi.imr_ifindex == dif)
2172            break;
2173    }
2174    if (!pmc)
2175        return 1;
2176    psl = pmc->sflist;
2177    if (!psl)
2178        return pmc->sfmode == MCAST_EXCLUDE;
2179    for (i=0; i<psl->sl_count; i++) {
2180        if (psl->sl_addr[i] == rmt_addr)
2181            break;
2182    }
2183    if (pmc->sfmode == MCAST_INCLUDE && i >= psl->sl_count)
2184        return 0;
2185    if (pmc->sfmode == MCAST_EXCLUDE && i < psl->sl_count)
2186        return 0;
2187    return 1;
2188 }
```
Unicast socket lookup

The `udp_v4_lookup()` function is defined in `net/ipv4/udp.c`. After read locking the UDP hash table, it calls `udp_v4_lookup_longway()`. If a socket is found, `sock_hold()` is called to increment its reference count. You must obtain this reference in your `cop_lookup` function when the target socket is identified and remember to drop this reference after the packet is enqueued.

```c
268 static __inline__ struct sock *udp_v4_lookup(u32 saddr,
                 u16 sport,
                 u32 daddr, u16 dport, int dif)
269 {
    struct sock *sk;
272    read_lock(&udp_hash_lock);
274    sk = udp_v4_lookup_longway(saddr, sport, daddr, dport, dif);
275    if (sk)
276        sock_hold(sk);
277    read_unlock(&udp_hash_lock);
278    return sk;
279 }
280```

14
Identifying the destination `struct sock`.

The `udp_v4_lookup_longway()` function selects the socket which matches received packet's fields most closely with respect to the following criteria.

<table>
<thead>
<tr>
<th>Socket</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
</tr>
<tr>
<td>Source address</td>
<td>Destination address</td>
</tr>
<tr>
<td>Destination port</td>
<td>Source port</td>
</tr>
<tr>
<td>Destination address</td>
<td>Source address</td>
</tr>
<tr>
<td>Device (bound)</td>
<td>Device (received)</td>
</tr>
</tbody>
</table>

The low order bits of the destination port are used as an index to identify the correct hash chain. For each `struct sock` on the chain in which the `local port to which the socket is bound matches the destination port in the packet`, a goodness of matching "score" is computed based upon how many other attributes of the socket match attributes of the arriving packet.

If all fields are matched, that `struct sock` is immediately accepted. Otherwise, the `struct sock` that matches largest number of fields is returned. A mismatch with a specified field is an immediate disqualifier.

```c
223 static struct sock *udp_v4_lookup_longway(u32 saddr,
224       u16 sport,
225       u32 daddr, u16 dport, int dif)
226 {
227    struct sock *sk, *result = NULL;
228    struct hlist_node *node;
229    unsigned short hnum = ntohs(dport);
230    int badness = -1;
```
The scoring loop

Back in the good old days of 2.4 the score was incremented by 1 on each match and the comparison with PF_INET was absent. Then a perfect score was 4. Presumably these changes are made for IPV6 support.

```c
    sk_for_each(sk, node, &udp_hash[hnum &
                 (UDP_HTABLE_SIZE - 1)]) {
       struct inet_sock *inet = inet_sk(sk);

       if (inet->num == hnum && !ipv6_only_sock(sk)) {
           int score = (sk->sk_family == PF_INET ? 1 : 0);
           if (inet->rcv_saddr) {
               if (inet->rcv_saddr != daddr)
                   continue;
               score+=2;
           }
           if (inet->daddr) {
               if (inet->daddr != saddr)
                   continue;
               score+=2;
           }
           if (inet->dport) {
               if (inet->dport != sport)
                   continue;
               score+=2;
           }
           if (sk->sk_bound_dev_if) {
               if (sk->sk_bound_dev_if != dif)
                   continue;
               score+=2;
           }
           if(score == 9) {
               result = sk;
               break;
           } else if(score > badness) {
               result = sk;
               badness = score;
           }
       }
    }
return result;
```
Delivery to the UDP receive queue.

The `udp_queue_rcv_skb()` function is defined in `net/ipv4/udp.c`. Its mission is to add the `sk_buff` to the receive queue if there exists sufficient space in the buffer quota of the process.

```c
984 /* returns:
985  * -1: error
986  *  0: success
987  * >0: "udp encap" protocol resubmission
988  *
989  * Note that in the success and error cases, the skb is assumed to
990  * have either been requeued or freed.
991 */
992 static int udp_queue_rcv_skb(struct sock * sk,
993    struct sk_buff * skb)
994 {
995    struct udp_sock * up = udp_sk(sk);

This comment is now in the wrong place because new code has been inserted. The call to `xfrm4_policy_check()` allows the `xfrm` system to prevent delivery.

```c
996    /*
997    * Charge it to the socket, dropping if the queue is full.
998    */
999    if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb)) {
1000       kfree_skb(skb);
1001       return -1;
1002    }
1003    nf_reset(skb);
1004```
Processing encapsulation sockets

```c
if (up->encap_type) {
  /* This is an encapsulation socket, so let's see if this is
   * an encapsulated packet.
   * If it's a keepalive packet, then just eat it.
   * If it's an encapsulated packet, then pass it to the
   * IPsec xfrm input and return the response
   * appropriately. Otherwise, just fall through and
   * pass this up the UDP socket.
   */

  int ret;

  ret = udp_encap_rcv(sk, skb);
  if (ret == 0) {
    /* Eat the packet .. */
    kfree_skb(skb);
    return 0;
  }
  if (ret < 0) {
    /* process the ESP packet */
    ret = xfrm4_rcv_encap(skb, up->encap_type);
    UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
    return -ret;
  }
  /* FALLTHROUGH -- it's a UDP Packet */
}
```
Handling regular UDP packets

The remaining two actions to be performed are checksum computation and the actual delivery to the receive queue of the packet. The role of `sk_filter` is a bit nebulous. We shall see it again later on. Presumably the filter can modify the packet so that if checksumming is to be done, it must be done before the filter is run.

```c
1032    if (sk->sk_filter && skb->ip_summed != CHECKSUM_UNNECESSARY) {
1033       if (__udp_checksum_complete(skb)) {
1034          UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1035          kfree_skb(skb);
1036          return -1;
1037       }
1038       skb->ip_summed = CHECKSUM_UNNECESSARY;
1039    }
1040
```

The `sock_queue_rcv_skb()` function is called to queue this `sk_buff`. If adequate space does not exist, then it is necessary to discard the packet and to update the associated error counters. This is a useful function for your protocol to employ.

```c
1041    if (sock_queue_rcv_skb(sk,skb)<0) {
1042       UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1043       kfree_skb(skb);
1044       return -1;
1045    }
1046    UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
1047    return 0;
1048 }
```
Adding the `sk_buff` to the sock's receive queue

The `sock_queue_rcv_skb()` function is defined as an inline function in `/net/core.sock.c`. It is a generic function that can be used by all transports to attempt to queue received packets.

```c
236 int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb) {
237     int err = 0;
238     int skb_len;
239
240     /* Cast skb->rcvbuf to unsigned... It's pointless, but reduces ...
      number of warnings when compiling with -W --ANK */
241     if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
242         (unsigned)sk->sk_rcvbuf) {
243         err = -ENOMEM;
244         goto out;
245     }
246     /* It would be deadlock, if sock_queue_rcv_skb is used ...
       with socket lock! We assume that users of this ...
       function are lock free. */
248     if (err)
249         goto out;
```

Make sure that sufficient memory exists to queue this `sk_buff`. Capacities are specified in units of bytes.

- `sk->rmem_alloc`: Total data queued in receive queue.
- `skb->truesize`: Size of buffer.
- `sk->rcvbuf`: Size of receiver buffer. It was set by `sys_socket` to default max value of `(sysctl_rmem_default)`.

The call to `sk_filter()` runs any BPF (Berkeley packet filter) chains that have been attached to the socket. Possibly this is related to tcpdump and friends.
Waking processes sleeping on the socket

The `skb_set_owner_r()` function sets `skb->sk = sk`, establishing this sockets ownership of the `sk_buff` and also charges the `struct sock` for the buffer space.

```c
258    skb->dev = NULL;
259    skb_set_owner_r(skb, sk);
260
261    /* Cache the SKB length before we tack it onto the receive
262     * queue. Once it is added it no longer belongs to us and
263     * may be freed by other threads of control pulling packets
264     * from the queue.
265     */
266    skb_len = skb->len;
```

The `sk_buff` is appended to the receive queue of the socket.

```c
268    skb_queue_tail(&sk->sk_receive_queue, skb);
```

It is also necessary to wake up any processes that might be sleeping while waiting for a packet to arrive. The `sk->data_ready` pointer was set by `sock_init_data()` to address of function `sock_def_readable()`.

```c
270    if (!sock_flag(sk, SOCK_DEAD))
271        sk->sk_data_ready(sk, skb_len);
272 out:
273    return err;
274 }```
/**
 *  sk_filter - run a packet through a socket filter
 *  @sk: sock associated with &sk_buff
 *  @skb: buffer to filter
 *  @needlock: set to 1 if the sock is not locked by caller.
 *  *
 *  Run the filter code and then cut skb->data to correct size returned by
 *  sk_run_filter. If pkt_len is 0 we toss packet. If skb->len is smaller
 *  than pkt_len we keep whole skb->data. This is the socket level
 *  wrapper to sk_run_filter. It returns 0 if the packet should
 *  be accepted or -EPERM if the packet should be tossed.
 *  *
 */

static inline int sk_filter(struct sock *sk,
    struct sk_buff *skb, int needlock)
{
    int err;

    err = security_sock_rcv_skb(sk, skb);
    if (err)
        return err;

    if (sk->sk_filter) {
        struct sk_filter *filter;

        if (needlock)
            bh_lock_sock(sk);

        filter = sk->sk_filter;
        if (filter) {
            unsigned int pkt_len = sk_run_filter(skb, filter->insns,
                                                   filter->len);
            err = pkt_len ? pskb_trim(skb, pkt_len) : -EPERM;
        }

        if (needlock)
            bh_unlock_sock(sk);
    }

    return err;
}
/**
 * sk_run_filter - run a filter on a socket
 * @skb: buffer to run the filter on
 * @filter: filter to apply
 * @flen: length of filter
 *
 * Decode and apply filter instructions to the skb->data.
 * Return length to keep, 0 for none. skb is the data we are
 * filtering, filter is the array of filter instructions, and
 * len is the number of filter blocks in the array.
 */

unsigned int sk_run_filter(struct sk_buff *skb, struct sock_filter *filter, int flen)
{
    struct sock_filter *fentry; /* We walk down these */
    void *ptr;
    u32 A = 0;        /* Accumulator */
    u32 X = 0;        /* Index Register */
    u32 mem[BPF_MEMWORDS]; /* Scratch Memory Store */
    u32 tmp;
    int k;
    int pc;

    /*
     * Process array of filter instructions.
     */
    for (pc = 0; pc < flen; pc++) {
        fentry = &filter[pc];

        switch (fentry->code) {
        case BPF_ALU|BPF_ADD|BPF_X:
            A += X;
            continue;
        case BPF_ALU|BPF_ADD|BPF_K:
            A += fentry->k;
            continue;
        case BPF_ALU|BPF_SUB|BPF_X:
            A -= X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_K:
            A -= fentry->k;
            continue;
        }
    }
}
Awakening blocked readers

The `sock_def_readable()` function is responsible for awaking processes that may be waiting on data from the socket. The element `sk->sleep` is of type `wait_queue_head_t *`. The first test in the `if` statement is to see if it actually points to a `wait_queue_head_t`. Even if it does, the list might be empty. Thus the second test is performed.

```c
1420 static void sock_def_readable(struct sock *sk, int len)  
1421 {  
1422    read_lock(&sk->sk_callback_lock);  
1423    if (sk->sk_sleep && waitqueue_active(sk->sk_sleep))  
1424      wake_up_interruptible(sk->sk_sleep);  
1425    sk_wake_async(sk,1,POLL_IN);  
1426    read_unlock(&sk->sk_callback_lock);  
1427 }
```

The `sk_wake_async()` function is used to initiate the sending of a signal to a process that is using the asynchronous I/O facility. This does not apply the situation we are studying.
Accounting for the allocation of receive buffer space.

A device driver will not call `skb_set_owner_r()` because it does not know which `struct sock` will eventually own the `sk_buff`. However, when a received `sk_buff` is eventually assigned to a `struct sock`, `skb_set_owner_r()` will be called.

Interestingly, unlike `skb_set_owner_w()`, the `skb_set_owner_r()` function does not call `sock_hold()` even though it does hold a pointer to the `struct sock`. This seems to set up the possibility of an ugly race condition if a socket is closed about the time a packet is received.

```c
1102 static inline void skb_set_owner_r(struct sk_buff *skb, struct sock *sk) {
1103    skb->sk = sk;
1104    skb->destructor = sock_rfree;
1105    atomic_add(skb->truesize, &sk->sk_rmem_alloc);
1107 }

1021 void sock_rfree(struct sk_buff *skb) {
1022    struct sock *sk = skb->sk;
1023    atomic_sub(skb->truesize, &sk->sk_rmem_alloc);
1026 }
```