1. Arp probe

When a packet sends/forwards out, it has to find the peer/gateway MAC address.

Arp entry: key-----------output

Dev +dst ip dst mac

inline int ip\_finish\_output2(struct sk\_buff \*skb)

{

…….

# get the nexthop:daddr or gateway

nexthop = (\_\_force u32) rt\_nexthop(rt, ip\_hdr(skb)->daddr);

# find the arp entry with the dev and nexthop

neigh = \_\_ipv4\_neigh\_lookup\_noref(dev, nexthop);

/\*if none create one with status NUD\_NONE\*/

if (unlikely(!neigh))

neigh = \_\_neigh\_create(&arp\_tbl, &nexthop, dev, false);

if (!IS\_ERR(neigh)) {

/\*handle the packet with the arp entry status \*/

int res = dst\_neigh\_output(dst, neigh, skb);

}

}

static const struct neigh\_ops arp\_generic\_ops = {

.family = AF\_INET,

.solicit = arp\_solicit,

.error\_report = arp\_error\_report,

.output = neigh\_resolve\_output,

.connected\_output = neigh\_connected\_output,

};

int dst\_neigh\_output(struct dst\_entry \*dst, struct neighbour \*n, struct sk\_buff \*skb)

{

hh = &n->hh;

/\*if the status of arp entry is NUD\_REACHABLE or NUD\_PERMANENT and the dst mac addr is cache, send the packet padding with the mac header \*/

if ((n->nud\_state & NUD\_CONNECTED) && hh->hh\_len)

return neigh\_hh\_output(hh, skb);

else

/\*the status is other, then neigh\_resolve\_output \*/

return n->output(n, skb);

}

neigh\_resolve\_output---->neigh\_probe

static void neigh\_probe(struct neighbour \*neigh)

{

struct sk\_buff \*skb = skb\_peek(&neigh->arp\_queue);

/\* keep skb alive even if arp\_queue overflows \*/

if (skb)

skb = skb\_copy(skb, GFP\_ATOMIC);

write\_unlock(&neigh->lock);

neigh->ops->solicit(neigh, skb);

atomic\_inc(&neigh->probes);

kfree\_skb(skb);

}

void arp\_solicit(struct neighbour \*neigh, struct sk\_buff \*skb)

{

/\*get the dev\*/

struct net\_device \*dev = neigh->dev;

\_\_be32 target = \*(\_\_be32 \*)neigh->primary\_key;

in\_dev = \_\_in\_dev\_get\_rcu(dev);

\_\_be32 saddr = 0;

/\*src ip selection\*/

switch (IN\_DEV\_ARP\_ANNOUNCE(in\_dev)) {

default:

case 0:

/\* By default choose the src ip of the packet need to be sent if src ip is the local one \*/

if (skb && inet\_addr\_type(dev\_net(dev),

ip\_hdr(skb)->saddr) == RTN\_LOCAL)

saddr = ip\_hdr(skb)->saddr;

break;

case 1:

/\* choose the src ip of the packet need to be sent if src ip is the local one and must be an primary addr and in the same subnet with the target ip\*/

if (!skb)

break;

saddr = ip\_hdr(skb)->saddr;

if (inet\_addr\_type(dev\_net(dev), saddr) == RTN\_LOCAL) {

/\* saddr should be known to target \*/

if (inet\_addr\_onlink(in\_dev, target, saddr))

break;

}

saddr = 0;

break;

case 2:

break;

}

/\*if the src ip is not choosen

1. first chose one on the dev, the one must be a primary addr, in t and not a SCOPE\_HOST addr, prefer to choose the one has the same subnet with the target
2. choose a one on other devices , the one must be a primary addr and not a SCOPE\_HOST and SCOPE\_LINK address

\*/

if (!saddr)

saddr = inet\_select\_addr(dev, target, RT\_SCOPE\_LINK);

arp\_send(ARPOP\_REQUEST, ETH\_P\_ARP, target, dev, saddr,

dst\_hw, dev->dev\_addr, NULL);

}

1. Arp receive

Arp\_rcv----> arp\_process((struct sk\_buff \*skb)

{

/\*arp request\*/

if (arp->ar\_op == htons(ARPOP\_REQUEST) &&

/\*tip is in local and also did the reverse path filter\*/

ip\_route\_input\_noref(skb, tip, sip, 0, dev) == 0) {

rt = skb\_rtable(skb);

addr\_type = rt->rt\_type;

/\*find the route and must be a LOCAL one\*/

if (addr\_type == RTN\_LOCAL) {

int dont\_send;

/\*ignore check: /proc/sys/net/ipv4/conf/eth0/arp\_ignore\*/

dont\_send = arp\_ignore(in\_dev, sip, tip);

if (!dont\_send && IN\_DEV\_ARPFILTER(in\_dev))

/\*did the reverse path filter again. But for the L2

/proc/sys/net/ipv4/conf/eth0/arp\_filter\*/

dont\_send = arp\_filter(sip, tip, dev);

if (!dont\_send) {

/\*create a entry according to the request\*/

n = neigh\_event\_ns(&arp\_tbl, sha, &sip, dev);

if (n) {

/\*send replay\*/

arp\_send(ARPOP\_REPLY, ETH\_P\_ARP, sip,

dev, tip, sha, dev->dev\_addr,

sha);

neigh\_release(n);

}

goto out;

}

/\*update the arp tables for sip and dev, the entry must be exist for this sip and dev\*/

n = \_\_neigh\_lookup(&arp\_tbl, &sip, dev, 0);

/\* /proc/sys/net/ipv4/conf/eth0/arp\_accept\*/

if (IN\_DEV\_ARP\_ACCEPT(in\_dev)) {

/\* Unsolicited ARP is not accepted by default.

It is possible, that this option should be enabled for some

devices (strip is candidate)

\*/

/\*when arp\_accept enable unsolicited replay also update it if find the route to sip \*/

if (n == NULL &&

(arp->ar\_op == htons(ARPOP\_REPLY) ||

(arp->ar\_op == htons(ARPOP\_REQUEST) && tip == sip)) &&

inet\_addr\_type(net, sip) == RTN\_UNICAST)

n = \_\_neigh\_lookup(&arp\_tbl, &sip, dev, 1);

}

if (n) {

/\*replay one\*/

int state = NUD\_REACHABLE;

int override;

if (arp->ar\_op != htons(ARPOP\_REPLY) ||

skb->pkt\_type != PACKET\_HOST)

/\*for request one in arp\_accept\*/

state = NUD\_STALE;

neigh\_update(n, sha, state,

override ? NEIGH\_UPDATE\_F\_OVERRIDE : 0);

}

out:

consume\_skb(skb);

return 0;

}

static int arp\_ignore(struct in\_device \*in\_dev, \_\_be32 sip, \_\_be32 tip)

{

int scope;

switch (IN\_DEV\_ARP\_IGNORE(in\_dev)) {

case 0: /\* Reply, the tip is already validated \*/

return 0;

case 1: /\* Reply only if tip is configured on the incoming interface \*/

sip = 0;

scope = RT\_SCOPE\_HOST;

break;

case 2: /\*

\* Reply only if tip is configured on the incoming interface

\* and is in same subnet as sip

\*/

scope = RT\_SCOPE\_HOST;

break;

case 3: /\* Do not reply for scope host addresses \*/

sip = 0;

scope = RT\_SCOPE\_LINK;

break;

case 4: /\* Reserved \*/

case 5:

case 6:

case 7:

return 0;

case 8: /\* Do not reply \*/

return 1;

default:

return 0;

}

return !inet\_confirm\_addr(in\_dev, sip, tip, scope);

}

1. Arp status Map

Status of the arp entry:

#define NUD\_INCOMPLETE 0x01

#define NUD\_REACHABLE 0x02

#define NUD\_STALE 0x04

#define NUD\_DELAY 0x08

#define NUD\_PROBE 0x10

#define NUD\_FAILED 0x20

Timers and counters

# ls /proc/sys/net/ipv4/neigh/eth0/

anycast\_delay base\_reachable\_time\_ms locktime proxy\_qlen ucast\_solicit

app\_solicit delay\_first\_probe\_time mcast\_solicit retrans\_time unres\_qlen

base\_reachable\_time gc\_stale\_time proxy\_delay retrans\_time\_ms unres\_qlen\_bytes

