XFRM policy lookup, current state and future work

Florian Westphal
4096R/AD5FF600 fw@strlen.de
80A9 20C5 B203 E069 F586
AE9F 7091 A8D9 AD5F F600

Red Hat

March 2018

Intro

- Policy database controls how packets are handled
- policies exists for input,forward and output directions
- policies can be injected by external programs (IKE daemon, ip xfrm, ...)
- policies can be attached directly for a given socket via setsockopt
  - e.g. to allow IKE daemon to set exception for its sockets
- 3 different uapis: setsockopt, `PF_KEY` sockets, `netlink` based api

In earlier kernels ...

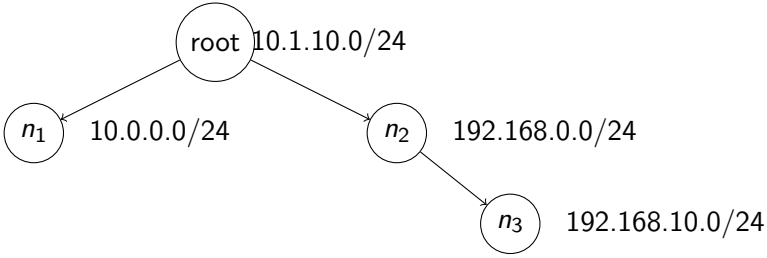
- One list for each of input,forward,output directions
- Ordered by priority
- "direct" policies are stored in a hash table
  - "direct": /32 or /128, i.e. match a specific host on BOTH saddr and daddr
  - keeps lists shorter in case many of those exist
  - later addition: can tune the hash table thresh: e.g. place /31, /30, /29 in hash table instead of lists
- "direct policies" DO NOT override others:
  - first perform a lookup in hash table
  - then in the appropriate inexact list
- Only other advantage: can abort inexact search early if lookup in hash table yielded a higher priority
- ...NOT "best" match, the highest priority policy "wins"

In current kernels ...

- direct hash table still exists, and consulted first
- inexact policies are stored in any of four search list classes that are stored inside a tree
- Lookup algorithm:
  1. locate the search tree (input, forward, output)
     - this also takes netns and ifid (xfrm device number) into account
     - removes need to search policies that would never match (e.g. tied to a different xfrm device)
  2. contains the "any list". This contains wildcard policies, i.e. `0.0.0.0/0 <-> 0.0.0.0/0`
  3. after this, rbtrees are searched

First Layer rbtrees

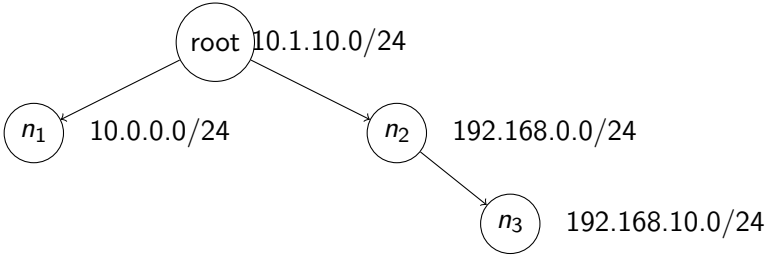
First tree: sorted by daddr, second tree sorted by saddr.



The saddr-tree contains list of policies that match given network/prefix but where destination is the any address.

Second Layer rbtrees

Every node of first layer daddr tree contains another tree root



... containing only policies that match BOTH daddr and saddr.
With lots of policies, there could be thousands of such trees

Or, a bit more complex ascii diagram...

```
* +- root_d: sorted by daddr:prefix
* |      |
* |   xfrm_pol_inexact_node
* |      +- root: sorted by saddr/prefix
* |      |          |
* |      |   xfrm_pol_inexact_node
* |      |          + root: unused
* |      |          + hhead: saddr:daddr policies
* |      |
* |      +- coarse policies and all any:daddr policies
* |
* +--root_s: sorted by saddr:prefix
*              |
*      xfrm_pol_inexact_node
*              + root: unused
*              + hhead: saddr:any policies
```

In current kernels ...

- the tree lookup doesn't do a policy lookup ...
- it figures out which lists need to be searched, returning up to four list head pointers:

```
struct xfrm_pol_inexact_candidates {
struct hlist_head *res[4];
};
```

1. `XFRM_POL_CAND_BOTH` contains only policies that match both ip source and destination of the packet.
2. `XFRM_POL_CAND_SADDR` contains only policies that match ip source of the packet. Destination is a wildcard.
3. `XFRM_POL_CAND_DADDR` same, except source is wildcard and daddr is fixed.
4. `XFRM_POL_CAND_ANY` both saddr and daddr are wildcards.

Linear search is then applied to all four lists to find the entry with the highest priority.

Pain points, aside from code complexity

assumptions:

- most policies have both a source and destination
- the source and destination networks do not overlap

When adding policies for `10.0.0.0/24`, `10.0.1.0/24`, `10.0.1.0/20` nodes have to be merged in kernel
No extra consideration for lookups by ports and the like.
Upside: Kernel internal details, can be changed at will (incl. revert).

Future work

The direct hash table is home-grown, predates rhashtable

- has custom code for rehashing and grow/shrink
- rhashtable handles this automatically
- direct hash table still needed:
  - consider 10.1.2.0/24 where we have one extra policy for each address (10.1.2.1,2,3...)
  - with rbtree its stored in one node (10.1.2.0/24), so list has 254+ entries
  - could embed a small hash table into the rbnode, but that has no advantage vs. direct hash table