Bridge filtering with nftables

Florian Westphal
Red Hat
fw@strlen.de

Abstract

The current Linux bridge/ebtables architecture has several shortcomings. Nftables, a framework to replace and unify the various address family specific packet filtering tools in the Linux kernel offers an opportunity to provide a more flexible approach to handling bridge filtering needs. After a brief summary of the ebtables and bridge filtering issues, this paper will show some of the advantages that nft bridge offers over ebtables and present two features that are currently being worked on in detail: stateful packet filtering and nfqueue (queue packets to userspace and let application decide fate of packet).

Current bridge netfilter state

ebtables

"The ebtables utility enables basic Ethernet frame filtering on a Linux bridge, logging, MAC NAT."[7]

It was forked from the iptables/ipv4 netfilter code base more than a decade ago.

Some of the matching capabilities include the ability to test on ip or ipv6 addresses, VLAN ids, the packet type as seen by the kernel (multicast, broadcast, "this host" or "other host") and the packet nfmark (sometimes also called fwmark).

Packet mangling features offered by ebtables includes stateless translation of MAC addresses and the ability to redirect frames to the local network stack, pretending they were addressed to the bridge MAC address.

netfilter hooks

Several hooks are placed in the bridge module to make packets available to the ebtables rules.

Those are:

- `NF_BR_PRE_ROUTING` first hook invoked, runs before forward database is consulted.
- `NF_BR_LOCAL_IN` invoked for packets destined for the machine where the bridge was configured on. This means that ipv4 or ipv6 packets will be passed up the stack and will be evaluated in the context of iptables (or ip6tables) later.
- `NF_BR_FORWARD`, called for frames that are bridged to a different port of the same logical bridge device. This is the first hook where the bridge output port is known.

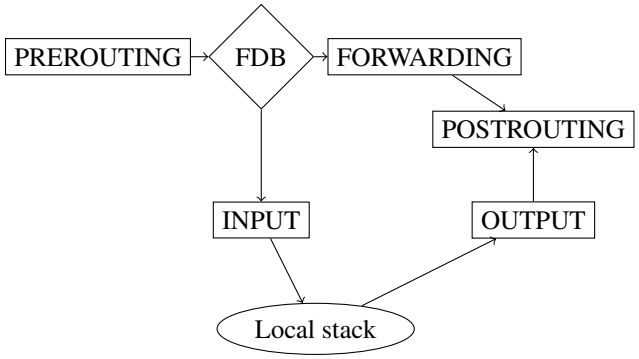

Figure 1: Bridge netfilter Hooks. Packets delivered to the local stack will be processed by the next protocol handler, e.g. the IP stack.

- `NF_BR_LOCAL_OUT` called for locally originating packets that will be transmitted via the bridge.
- `NF_BR_POST_ROUTING` called for all locally generated packets and all bridged packets.
- `NF_BR_BROUTING` not a hook – used by the magic ebtables "broute" table which can be used to have packets enter the local stack without even hitting the main bridge code.

Diagram of the above:

There is nothing fundamentally broken with the placement of these hooks. The only issue with them is that – like ipv4 and ipv6 family netfilter hooks – they are unique per namespace. This means that on a system with lots of bridge interfaces where only a small subsets of these need filtering rules all the other bridges also pay the cost of the ebtables traverser invocation.

Shortcomings and problems

The feature set provided by ebtables is limited to link-layer matching and ability to match ip and ipv6 addresses in ethernet frames.

The few network layer matches offered are duplicated code – ebtables cannot use xtables targets or modules offered by the ip(6)tables core from the ebtables rule set directly.

The call-iptables layer

To overcome some of these issues the kernel provides a bridge netfilter module, `br_netfilter.ko`, which implements a feature referenced here as "call-iptables" layer, named after the sysctl that controls this behaviour: `net.bridge.bridge-nf-call-iptables`.

When the module is loaded, it will register bridge netfilter hooks that then call the ipv4/ipv6 netfilter hooks from the bridge layer based on the ethernet type. In other words, bridged packets will pass though the `PRE`, `FORWARD` and `POSTROUTING` chains of the ip(6)tables rule set configured on the host.

For this to work bridge netfilter will perform rudimentary ip/ipv6 header sanity checks that would normally be done by the IP stack and then calls the ip/ipv6 netfilter hooks. This has a few advantages, especially from a feature set perspective – the feature set offered by the ip/ip6tables and netfilter infrastructure becomes available. This also includes support for connection tracking and Layer 3/Layer 4 NAT/PAT with the iptables `SNAT`, `DNAT` and `REDIRECT` targets in the iptables `nat` table.

The downside is that introduces several layering violations and other problems.

In `ebtables`, the in and out devices that can be matched with the ebtables `--in-interface` and `--out-interface` specifies the name of the bridge port. However, iptables has no notion of "bridges" – by the time a packet is fed to a netfilter hook the in and out interface is the name of the logical (bridge) interface, for example `br0`. `iptables` provides a match (`physdev`) solely for the purpose of examining the "bridge" details of a packet. Packets that were sent to an IP address configured on the bridge machine itself or are currently being bridged (passed to another interface) can thus be "queried" for the interface name of the bridge port they originally arrived on or are in the process of being sent out by.

This feature comes with additional cost to the kernel – even if a packet has not crossed a bridge interface, every clone or free operation on an `sk_buff` (the data structure used to represent network packets) needs to check for the presence of this bridge meta data.

Another problem is that the ipv4 and ipv6 netfilter may call into ip or ipv6 stack functions which make the perfectly reasonable assumption that they're called from the ipv4 or ipv6 stack. This is not true for `call-iptables` – therefore, for every hook invocation of the ip/ipv6 hooks the `skb->cb[]` (control buffer, contains meta data from the layer that currently owns the packet) needs to be saved and restored to inet or bridge state.

For packets that are directed to the bridge, with `call-iptables` mode enabled, we invoke ip `PRE_ROUTING` hook twice. First from the bridge layer as part of the `call-iptables` feature, and again from the normal IP stack.

To suppress the re-invocation, the bridge netfilter code has to register a "sabotage" hook that suppressed the re-invocation early on. Again, another small piece of overhead that is imposed on a bridge `call-iptables` setup.

More issues with current approach

When VLAN was introduced, ebtables gained the ability to match on the VLAN ID. But xtables cannot be used to filter on the VLAN id.

A router configuration or end host doesn't care about this; it can just use the device interface name ("`-i eth0.42`"). To work around this, the `call-iptables` layer has a `bridge-nf-filter-vlan-tagged` mode – if enabled, such skbs are pushed to ip(6)tables anyway. This "works" because the kernel keeps VLAN headers only as meta data, but it comes at a price – all VLANs must have distinct IP addresses. Otherwise, VLAN isolation breaks down because ip defragmentation and connection tracking has no means to tell packets from those VLANs apart.

There is more subtle interaction between some features of xtables and connection tracking when invoked via the `call-iptables` infrastructure. For instance, the kernel can crash quite soon if the `NFQUEUE` target is used from the xtables ruleset[1].

Lastly, NAT support, while functional, is problematic from an architectural standpoint. The bridge code needs to make calls into the ipv4 or ipv6 forwarding database to obtain the new destination (which also means that the bridge needs a routing table for this to work) and its possible that the bridge has to loop an skb through the ipv4 neighbour code because it has to look up a new destination MAC address if the IP address is changed by L3 NAT handling.

Current state of nftables bridge

What is nftables

"nftables is the project that aims to replace the existing ip,ip6,arp,ebtables framework."[4]

Aside from providing the new userspace tool "nft" it is also a new packet classification framework based on lessons learnt from ip and ip6tables. nftables was presented at Netfilter Workshop 2008 (Paris, France) and released in March 2009 by Patrick McHardy. Nftables is available since Linux 3.13 (January 2014).

nft not only provides a replacement for arp-, eb-, ip- and ip6tables, it also adds the new "inet" and "netdev" families. The *inet* family is a pseudo-family for combined ipv4/ipv6 rule sets.

family	old tool	nft table
arp	arptables	arp
bridge	ebtables	bridge
ipv4	iptables	ip
ipv6	ip6tables	ip6

Table 1: commands and nft table family names of the existing protocol families

The *netdev* family can be used to attach a ruleset to an interface. Those rule sets are evaluated early, even before an

[1] brief version: bridge clones skbs when flooding packets, but connection tracking assumes an skb in "new" conntrack state is only visible to current CPU

skb is handed to a bridge or ip stack. It can be used for arbitrary filtering of any protocol, including arp, ppp frames or any other Layer two network protocol that has a notion of a network device.

Unlike iptables, all protocol specific details reside in the userspace tool. The nftables kernel part contains an interpreter that provides a small pseudo-instruction set. These instructions are called expressions and provide a specialized task each.

This also means that all supported address families have access to the same feature set. There is no need for duplication as with the old ebtables, arptables and iptables tools.

If a user specifies an nftables rule that tests for a particular source ip address, the nft tool will use the payload expression to load the ip address into a register and then use the cmp instruction to compare that register with the ip address. The kernel has no understanding of the ip address itself, it just loads some data and then does a comparison.

Example: Given the rule

```
nft add rule bridge filter forward \
      ip saddr 10.0.0.0/8 accept
```

nft will send following instructions[2] to the kernel:

```
[ payload load 2b @ link+12 => reg1 ]
[ cmp eq reg 1 0x00000008 ]
[ payload load 4b @ network+12 => reg1 ]
[ bitwise reg 1 = (reg=1 & 0x00ffffff ) ]
[ cmp eq reg 1 0x0000000a ]
[ immediate reg 0 accept ]
```

The first two lines add a test for `ether type ip`. This is injected by nft behind the scenes to avoid false positive matches when the bridge processes e.g. an ipv6 packet. The next line loads the source ip address into a register, masks out the part of the netmask that we are not interested in, and then compares the result to the desired ip address. As the rule specifies the *bridge* family, the rule set will be attached to the `NF_BRIDGE` forward hook point.

Inserting the same rule as `rule ip filter` results in exactly the same code – except that the implicit ip dependency is not added since its not needed in the ip protocol family.

This also showcases another important bit about nftables – all the protocol details are in userspace, the kernel merely loads a number of bytes from a given offset into a register.

nftables supports three bases for offsets. They can be relative to the link layer header, the network header or the transport header.

Another major advantage over the old tools is the addition of sets and maps. In nft it is possible to jump directly to chains with a verdict map, for instance:

```
add rule bridge filter prerouting vmap \
  meta iifname { bport0 : jump vm_one, \
                 bport1 : jump vm_two, \
                 bport2 : drop }
```

Unlike the ebtables equivalent, which would involve three rules that are tested one after another, this results in only two instructions, regardless of the number of elements in the map:

```
[ meta load iifname => reg 1 ]
[ lookup reg 1 set map0 dreg 0 ]
```

First instruction loads the name of the bridge port into a register, second instruction sets the verdict register based on the lookup result in the anonymous set.

This is especially useful if each bridge port should have a specific filter policy.

In case there are many bridges or bridge ports on a system, but only a very small amount of bridge ports need filtering, it might be advisable to instead use the *netdev* family and attach the policy to the device instead.

Example: This adds the *mytable* table with an ingress hook to the device `eth0`.

```
table netdev mytable {
   chain myingress {
     type filter hook ingress \
             device eth0 priority 0;
  }
}
```

Then, the earlier example can be attached like this:

```
nft add rule netdev mytable myingress \
      ip saddr 10.0.0.0/8 accept
```

The advantage is that traffic on other interfaces does not even result in a call into the netfilter core anymore provided no other hooks are registered on the system.

nft bridge

The nft utility and the kernel already implements the full ebtables feature set with the exception of the "arpreply" target. Since the bridge family is only a subset of nftables all the other features like sets or verdict maps are also available.

- bridge filtering is usually done via both ebtables and iptables rulesets
- because thats the only way to get conntrack and iptables features
- corner cases will result in kernel panic
- another problem: netfilter hooks are per namespace, not per bridge
- 500 bridges and only one with real filtering rules: not possible
- ebtables == 'iptables from 2001' (e.g. rwlock in main traverser)

nfqueue for nft bridge

The nfqueue mechanism allows an ip or nftables ruleset to pass a packet to userspace which can inspect and even modify the packet. The userspace program must drop or re-inject the packet into the kernel[3].

While nfqueue is already available in nftables, the netfilter backend that performs the queuing to userspace only works for packets queued from the ip, ipv6 or inet families.

While the technical reasons for this limitation can easily resolved, several considerations need to be made beforehand.

[2] slightly trimmed for brevity

Queuing from application point of view

Userspace programs use a netlink[5] socket to bind a nfqueue via a 16bit identification number. Packets queued by the kernel can then be received via this socket encoded as nfnetlink messages. The kernel provides several meta data attributes in addition to the actual packet data.

Examples of attributes provided are:

- the family and hook that queued the packet
- in and outgoing interface index
- packet nfmark
- the packet payload
- link layer hardware address, if available

Since nfqueue is currently only supported with ipv4/ipv6 packets, the packet payload starts with the IP or ipv6 header and contains as many bytes as userspace requested. Userspace can also alter the packet as needed – the kernel will detect changes in the ip header and will re-lookup the route.

Considerations for bridge nfqueue

While we do not necessarily have to support all features that are currently possible with the ipv4/ipv6 families, we should make sure to not prevent later addition.

Some functionality that might be interesting or desirable to have at some point, is the ability to perform packet payload rewrites, including the ability to add or remove existing VLAN headers.

Because most typical mainstream Network Adapters perform VLAN header offloads, the VLAN header is not part of the payload area but rather kept as meta data in the kernel internal skbuff structure. This is true even for non-offload case, the kernel will do this VLAN header stripping in software if needed. This happens very early, before packet sockets (tcpdump) and also before all netfilter hooks including the netdev family ingress hook.

Last but not least it makes sense to attempt to make the new bridge backend generic enough to also support nfqueue for the netdev family.

So in summary:

- `skb->data` pointer is exactly after the L2 header (if any).
- VLAN headers are always stored as skb meta data

nfqueue: Implementation

One way to implement it would be to just re-use the existing payload netlink attribute. This however has several drawbacks: We would have to push the mac header so `skb->data` points to start of mac header, then pull it again after re-inject so stack finds it pointing to the expected place. Not doing this would mean the L2 header isn't accessible by userspace so we would not be able to implement VLAN header addition, for example.

Another problem is that we would need to undo the VLAN header stripping done by core stack. Finally, when using an approach that just re-used the payload attribute with starting point being the L2 header – it becomes next to impossible to support packet mangling in userspace. The kernel would have to parse the L2 header to see if a VLAN tag got added or removed and so on. In the ingress case this is even worse since the L2 header could be anything.

Considering all of this the most sensible choice is to add new attributes:

- attribute containing the L2 header, i.e. the data pointed to by `skb->mac_header`.
- attribute containing the VLAN header.

This would also allow VLAN header stripping or addition, for example by allowing userspace to submit a verdict message that also provides the L2 header attribute.

Since netlink provides size of attributes all the kernel needs to do is to possibly expand headroom, fix up any other skb offsets (network, transport headers) and then replace the old header. Something similar could be added for removing or changing VLAN information.

conntrack

The other much desired feature for bridges is stateful firewalling. This is also known as Connection tracking[6].

The Linux conntrack subsystem offers stateful tracking for several transport protocols, including TCP, UDP and SCTP on top of either ip or ipv6. Since conntrack hooks in the ip and ipv6 families only, it is not available on a bridge unless the `call-iptables` layer is used.

To allow native hooking, we should first consider some details.

Defragmentation

The prerequisite for meaningful stateful connection tracking is IP defragmentation, otherwise the kernel cannot match fragments to an existing connection.

For ip/ip6tables, the protocol defragmentation modules (`nf_defrag_ipv4`, `nf_defrag_ipv6`) hook at prerouting, any fragmented packet is queued for defragmentation. There is only an artificial dependency on the defrag module in order to ensure that loading a conntrack module the `modprobe` dependency resolution also loads the defrag module for us.

Unfortunately, defragmentation on Linux bridges has limitations, since defragmentation is only provided by the ipv4/ipv6 netfilter modules one needs to enable the `call-iptables` feature sysctl.

As the IP stack neither considers the bridge interface nor VLAN ids when merging fragments, the network separation provided by VLANs is removed – with overlapping addresses in different VLANs packets from distinct networks can be merged by fragment reassembly. This also happens when a host has multiple bridges configured that handle distinct L2 networks but share the same overlapping L3 address space.

Defragmentation – Bridge implementation

The are two possible approaches to implement defrag for bridge.

1. stub module that just registers a `PRE_ROUTING` bridge hook and then calls the ipv4 ipv6 defragmentation hooks based on `skb->protocol`
2. a fully functional module that implements defragmentation directly and also automatically considers MAC addresses, VLAN id and the arriving interface index to separate packets when defragmenting.

Either solution could be implemented implicitly – module load enables functionality – or explicitly via ruleset activation. In the former case, one would simply load a module, for example `modprobe nf_defrag_bridge`, in the latter case, one would need to add a rule for it, for instance `nft add rule bridge raw pre defrag` to enable defragmentation. Implicit activation, while simpler, doesn't have the same flexibility that is offered by rule based configuration. For instance, one could enable defragmentation only on a selected bridge port or for a limited set of hosts, provided that no connection tracking is used.

Connection Tracking on bridges has the same problem as defragmentation once IP address spaces in distinct networks overlap. We will discuss solutions in the next section.

Connection Tracking

The canonical solution for the "overlapping addresses" use case on routers (for example when using policy based routing to separate networks) is the use of connection tracking zones.

Simply put, one configures iptables rules to set a conntrack zone (a 16 bit number) in the raw table, before conntrack inspects the packet. This zone id serves as an additional key to make a distinction between otherwise identical flows. Example rule set that puts traffic on eth0 into a separate zone:

```
iptables -t raw -A PREROUTING \
        -i eth0 -j CT --zone 42
iptables -t raw -A OUTPUT \
        -o eth0 -j CT --zone 42
```

On a bridge we could introduce a similar mechanism, for instance something like

```
add rule bridge track pre ct zone set \
                vlan id map { \
                   1 : 1, 2 : 2, }
```

While it would be possible to add the VLAN id to the connection tracking tuples to make them distinct entities automatically this doesn't solve all scenarios. For example one might have to cope with overlapping addresses in different bridge interfaces, or the overlap might be between a bridged VLAN and a GRE tunnel endpoint on the same host.

Conclusion

It appears best if both a defrag and conntrack functionality for bridge netfilter is implemented as a dissection step that calls the desired ipv4 or ipv6 functionality based on the `skb->protocol` value and rely on ruleset based configuration to ensure that distinct networks with identical addresses are kept separate.

Defragmentation for bridge is thus straightforward. The (to be written) bridge conntrack would just directly call the bridge defrag functionality.

```
add rule bridge track prerouting \
 ip saddr . tcp dport \
  { 10.1.2.3 . 21, 10.1.2.1 . 2121 } \
  ct set helper ftp
```

Figure 2: Explicit conntrack activation via nftables rule set

Since connection tracking can alter packets and imposes additional processing overhead it appears preferable to restrict tracking to flows that need this and require explicit conntrack setup – an inverse logic to what can be done with the iptables "`-j CT --notrack` or the older "`-j NOTRACK`" targets. Also, in the ipv4 and ipv6 netfilter conntrack engine automatic helper assignment has been deprecated since 2012 for security reasons ([2]), so to use helpers like ftp or sip will soon require manual setup for ipv4 and ipv6 too.

An example of how this could look like with nftables is given in figure 2.

To request connection tracking when clients connect to the ftp server at the specified addresses.

For traffic in the reply direction and packets related to the flow in some way, ICMP messages for Path MTU discovery for example, we cannot use explicit configuration as we need to examine all packets traversing the bridge. Because we cannot tell whether a fragment matches an existing flow we also need to enforce defragmentation for all packets.

Conntrack: Implementation

IPv4 (and IPV6) conntrack uses several hook points:

- PREROUTING: conntrack input
- INPUT: helper, confirm
- OUTPUT: local conntrack
- POSTROUTING: helper, conntrack confirmation

In addition to these hooks, the IP conntrack modules depend on their respective defragmentation modules which will hook in `PREROUTING` and `OUTPUT` before conntrack can examine the packet.

A bridge conntrack doesn't need to consider INPUT and OUTPUT – the IP stack will already handle local packets. bridge conntrack needs to add an implicit hook for processing packets that may belong to or might be related to existing connections.

This results in following hooks in addition to explicit rule set based invocation:

1. PREROUTING: hook to pick up related and reply traffic
2. POSTROUTING: helper invocation (e.g. to create expectations)
3. POSTROUTING: conntrack confirmation (commit a new conntrack entry to main conntrack table)

The first hook would perform a lookup in the conntrack table.

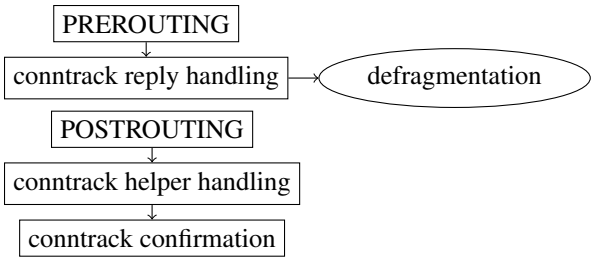

Figure 3: Hook placement for bridge connection tracking. Only Pre and Postrouting hooks are used. The conntrack reply hook also handles defragmentation.

If a match is found (packet is in original or reply direction of already existing connection) the conntrack can be associated with the packet and that packet can continue processing within the bridge.

If we don't find a matching connection, this hook would instead search for a matching expectation (packet is related to an existing connection – an ICMP error, FTP data connection, etc).

If the packet cannot be associated with an existing flow either via established or related matching, then the hook would – unlike the current ipv4 and ipv6 conntrack implementations – return without creating a new connection. Instead, it would be up to the user to configure the nftables bridge ruleset to decide if tracking should be performed or not. Therefore, the bridge conntrack would only pick up a new connection if the packet matches a "conntrack rule" similar to the example given in figure 2.

The second and third hooks would work exactly like their ipv4/ipv6 counterparts – it would be required to support creation of expectations e.g. for FTP DATA transfers or RTP streams announced by SIP messages. Conntrack confirmation is needed to commit a newly allocated conntrack entry to the main table. This happens as the last step so that we do not add conntracks to the main table if the packet is going to be dropped by a rule.

Figure 3 provides a summary of the new hooks that need to be registered with the nftables bridge family.

For local traffic there is one problem: We end up with invoking the POSTROUTING hooks twice, once from the IP or IPV6 hooks, once via the bridge hooks proposed here.

The confirm hook is unimportant – it is a no-op for virtually all cases. All it does is commit newly created conntracks into the global conntrack flow table.

But invoking the helper hook twice is a real issue. The most simple solution to this is to do the following:

1. add a 'bridged' marker bit to the conntrack (or the helper extension).
2. when assigning helper via bridge conntrack expression, set this bit
3. from the IP conntrack POSTROUTING hook, clear the bit again if it was set.
4. from the bridge conntrack POSTROUTING hooks, only invoke helper callback if that bit is set

The third step is needed to ensure that a packet arriving on a bridge interface that is then routed and later sent out via another bridge interface doesn't result in another hook invocation.

Alternatively, one could change the last step to clear `skb->nfct`. However, doing so also removes the ability to do stateful filtering at the bridge layer and limits features for traffic classification at the egress stage, so this solution should be avoided.

Summary

Development to add nfqueue for bridge is already being worked on by Stephane Bryant, see [1], work on bridge connection tracking has started as well, following a rule-based activation model.

References

[1] Bryant, S. 2016. netfilter: bridge: add nf_afinfo to enable queuing to userspace. patchwork. https://patchwork.ozlabs.org/patch/567977/.

[2] Leblond, E. 2012. Secure use of iptables and connection tracking helpers. blog. https://home.regit.org/netfilter-en/secure-use-of-helpers/.

[3] netfilter.org project. 2016a. iptables-extensions. netfilter.org web site. http://git.netfilter.org/iptables/tree/extensions/libxt_NFQUEUE.man?h=v1.6.0.

[4] netfilter.org project. 2016b. The netfilter.org nftables project. netfilter.org web site. http://nftables.org/projects/nftables/index.html.

[5] Pablo Neira Ayuso, Rafael M. Gasca, Laurent Lefèvre. 2010. Communicating between the kernel and user-space in Linux using Netlink sockets. *Software: Practice and Experience* 40(9).

[6] Pablo Neira Ayuso. 2006. Netfilter's Connection Tracking System. *In :LOGIN;, The USENIX magazine, Vol. 32, No. 3, pages 34-39*.

[7] Schuymer, B. D. 2015. ebtables. netfilter.org web site. http://ebtables.netfilter.org/documentation/what.html.

Author Biography

Florian Westphal is a contributor to the Linux kernel network stack, in particular netfilter. He is also a member of the netfilter core team.