Network Filters

This page provides an introduction to libvirt's network filters, their goals, concepts and XML format.

Goals and background

The goal of the network filtering XML is to enable administrators of virtualized system to configure and enforce network traffic filtering rules on virtual machines and manage the parameters of network traffic that virtual machines are allowed to send or receive. The network traffic filtering rules are applied on the host when a virtual machine is started. Since the filtering rules cannot be circumvented from within the virtual machine, it makes them mandatory from the point of view of a virtual machine user.

Network filtering support is available ***since 0.8.1 (Qemu, KVM)***

Concepts

The network traffic filtering subsystem enables configuration of network traffic filtering rules on individual network interfaces that are configured for certain types of network configurations. Supported network types are

- `network`
- `ethernet` -- must be used in bridging mode
- `bridge`
- `direct` -- only protocols mac, arp, ip and ipv6 can be filtered

The interface XML is used to reference a top-level filter. In the following example, the interface description references the filter `clean-traffic`.

```
...
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'/>
  </interface>
</devices>
...
```

Network filters are written in XML and may either contain references to other filters, contain rules for traffic filtering or can hold a combination of both. The above referenced filter `clean-traffic` is a filter that for example only contains references to other filters and no actual filtering rules. Since references to other filters can be used, a *tree* of filters can be built. The `clean-traffic` filter can be viewed using the command `virsh nwfilter-dumpxml clean-traffic`.

As previously mentioned, a single network filter can be referenced by multiple virtual machines. Since interfaces will typically have individual parameters associated with their respective traffic filtering rules, the rules described in a filter XML can be parameterized with variables. In this case, the variable name is used in the filter XML and the name and value are provided at the place where the filter is referenced. In the following example, the interface description has been extended with the parameter `IP` and a dotted IP address as value.

```
  ...
  <devices>
    <interface type='bridge'>
      <mac address='00:16:3e:5d:c7:9e'/>
      <filterref filter='clean-traffic'>
        <parameter name='IP' value='10.0.0.1'/>
      </filterref>
    </interface>
  </devices>
  ...
```

In this particular example, the `clean-traffic` network traffic filter will be instantiated with the IP address parameter 10.0.0.1 and enforce that the traffic from this interface will always be using 10.0.0.1 as the source IP address, which is one of the purposes of this particular filter.

Usage of variables in filters

Two variables names have so far been reserved for usage by the network traffic filtering subsystem: `MAC` and `IP`.

`MAC` is the MAC address of the network interface. A filtering rule that references this variable will automatically be instantiated with the MAC address of the interface. This works without the user having to explicitly provide the MAC parameter. Even though it is possible to specify the MAC parameter similar to the IP parameter above, it is discouraged since libvirt knows what MAC address an interface will be using.

The parameter `IP` represents the IP address that the operating system inside the virtual machine is expected to use on the given interface. The `IP` parameter is special in so far as the libvirt daemon will try to determine the IP address (and thus the IP parameter's value) that is being used on an interface if the parameter is not explicitly provided but referenced. For current limitations on IP address detection, consult the section on limitations on how to use this feature and what to expect when using it.

The following is the XML description of the network filer `no-arp-spoofing`. It serves as an example for a network filter XML referencing the `MAC` and `IP` parameters. This particular filter is referenced by the `clean-traffic` filter.

```
<filter name='no-arp-spoofing' chain='arp'>
  <uuid>f88f1932-debf-4aa1-9fbe-f10d3aa4bc95</uuid>
  <rule action='drop' direction='out' priority='300'>
    <mac match='no' srcmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='out' priority='350'>
    <arp match='no' arpsrcmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='out' priority='400'>
    <arp match='no' arpsrcipaddr='$IP'/>
  </rule>
  <rule action='drop' direction='in' priority='450'>
    <arp opcode='Reply'/>
    <arp match='no' arpdstmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='in' priority='500'>
    <arp match='no' arpdstipaddr='$IP'/>
  </rule>
  <rule action='accept' direction='inout' priority='600'>
    <arp opcode='Request'/>
  </rule>
  <rule action='accept' direction='inout' priority='650'>
    <arp opcode='Reply'/>
  </rule>
  <rule action='drop' direction='inout' priority='1000'/>
</filter>
```

Note that referenced variables are always prefixed with the $ (dollar) sign. The format of the value of a variable must be of the type expected by the filter attribute in the XML. In the above example, the `IP` parameter must hold a dotted IP address in decimal numbers format. Failure to provide the correct value type will result in the filter not being instantiatable and will prevent a virtual machine from starting or the interface from attaching when hotplugging is used. The types that are expected for each XML attribute are shown below.

Element and attribute overview

The root element required for all network filters is named `filter` with two possible attributes. The `name` attribute provides a unique name of the given filter. The `chain` attribute is optional but allows certain filters to be better organized for more efficient processing by the firewall subsystem of the underlying host. Currently the system only supports the chains `root`, `ipv4`, `ipv6`, `arp` and `rarp`.

References to other filers

Any filter may hold references to other filters. Individual filters may be referenced multiple times in a filter tree but references between filters must not introduce loops (directed acyclic graph).

The following shows the XML of the `clean-traffic` network filter referencing several other filters.

```
<filter name='clean-traffic'>
  <uuid>6ef53069-ba34-94a0-d33d-17751b9b8cb1</uuid>
  <filterref filter='no-mac-spoofing'/>
  <filterref filter='no-ip-spoofing'/>
  <filterref filter='allow-incoming-ipv4'/>
  <filterref filter='no-arp-spoofing'/>
  <filterref filter='no-other-l2-traffic'/>
  <filterref filter='qemu-announce-self'/>
</filter>
```

To reference another filter, the XML node `filterref` needs to be provided inside a `filter` node. This node must have the attribute `filter` whose value contains the name of the filter to be referenced.

New network filters can be defined at any time and may contain references to network filters that are not known to libvirt, yet. However, once a virtual machine is started or a network interface referencing a filter is to be hotplugged, all network filters in the filter tree must be available. Otherwise the virtual machine will not start or the network interface cannot be attached.

Filter rules

The following XML shows a simple example of a network traffic filter implementing a rule to drop traffic if the IP address (provided through the value of the variable IP) in an outgoing IP packet is not the expected one, thus preventing IP address spoofing by the VM.

```
<filter name='no-ip-spoofing' chain='ipv4'>
  <uuid>fce8ae33-e69e-83bf-262e-30786c1f8072</uuid>
  <rule action='drop' direction='out' priority='500'>
    <ip match='no' srcipaddr='$IP'/>
  </rule>
</filter>
```

A traffic filtering rule starts with the `rule` node. This node may contain up to three attributes

- action -- mandatory; must either be `drop` or `accept` if the evaluation of the filtering rule is supposed to drop or accept a packet
- direction -- mandatory; must either be `in`, `out` or `inout` if the rule is for incoming, outgoing or incoming-and-outgoing traffic
- priority -- optional; the priority of the rule controls the order in which the rule will be instantiated relative to other rules. Rules with lower value will be instantiated and therefore evaluated before rules with higher value. Valid values are in the range of 0 to 1000. If this attribute is not provided, the value 500 will automatically be assigned.

The above example indicates that the traffic of type `ip` will be asscociated with the chain 'ipv4' and the rule will have priority 500. If for example another filter is referenced whose traffic of type `ip` is also associated with the chain 'ipv4' then that filter's rules will be ordered relative to the priority 500 of the shown rule.

A rule may contain a single rule for filtering of traffic. The above example shows that traffic of type `ip` is to be filtered.

Supported protocols

The following sections enumerate the list of protocols that are supported by the network filtering subsystem. The type of traffic a rule is supposed to filter on is provided in the `rule` node as a nested node. Depending on the traffic type a rule is filtering, the attributes are different. The above example showed the single attribute `srcipaddr` that is valid inside the `ip` traffic filtering node. The following sections show what attributes are valid and what type of data they are expecting. The following datatypes are available:

- UINT8 : 8 bit integer; range 0-255
- UINT16: 16 bit integer; range 0-65535
- MAC_ADDR: MAC adrress in dotted decimal format, i.e., 00:11:22:33:44:55

- MAC_MASK: MAC address mask in MAC address format, i.e., FF:FF:FF:FC:00:00
- IP_ADDR: IP address in dotted decimal format, i.e., 10.1.2.3
- IP_MASK: IP address mask in either dotted decimal format (255.255.248.0) or CIDR mask (0-32)
- IPV6_ADDR: IPv6 address in numbers format, i.e., FFFF::1
- IPV6_MASK: IPv6 mask in numbers format (FFFF:FFFF:FC00::) or CIDR mask (0-128)
- STRING: A string

Every attribute except for those of type IP_MASK or IPV6_MASK can be negated using the `match` attribute with value `no`. Multiple negated attributes may be grouped together. The following XML fragment shows such an example using abstract attributes.

```
[...]
  <rule action='drop' direction='in'>
    <protocol match='no' attribute1='value1' attribute2='value2'/>
    <protocol attribute3='value3'/>
  </rule>
[...]
```

Rules perform a logical AND evaluation on all values of the given protocol attributes. Thus, if a single attribute's value does not match the one given in the rule, the whole rule will be skipped during evaluation. Therefore, in the above example incoming traffic will only be dropped if the protocol property attribute1 does not match value1 AND the protocol property attribute2 does not match value2 AND the protocol property attribute3 matches value3.

MAC (Ethernet)

Protocol ID: `mac`
Note: Rules of this type should go into the `root` chain.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
protocolid	UINT16 (0x600-0xffff), STRING	Layer 3 protocol ID

Valid Strings for `protocolid` are: arp, rarp, ipv4, ipv6

Example:

```
<mac match='no' srcmacaddr='$MAC'/>
```

ARP/RARP

Protocol ID: `arp` or `rarp`
Note: Rules of this type should either go into the `root` or `arp/rarp` chain.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
hwtype	UINT16	Hardware type
protocoltype	UINT16	Protocol type
opcode	UINT16, STRING	Opcode
arpsrcmacaddr	MAC_ADDR	Source MAC address in ARP/RARP packet
arpdstmacaddr	MAC_ADDR	Destination MAC address in ARP/RARP packet
arpsrcipaddr	IP_ADDR	Source IP address in ARP/RARP packet
arpdstipaddr	IP_ADDR	Destination IP address in ARP/RARP packet

Valid strings for the `Opcode` field are: Request, Reply, Request_Reverse, Reply_Reverse, DRARP_Request, DRARP_Reply, DRARP_Error, InARP_Request, ARP_NAK

IPv4

Protocol ID: `ip` Note: Rules of this type should either go into the `root` or `ipv4` chain.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
protocol	UINT8, STRING	Layer 4 protocol identifier
srcportstart	UINT16	Start of range of valid source ports; requires `protocol`

srcportend	UINT16	End of range of valid source ports; requires `protocol`
dstportstart	UINT16	Start of range of valid destination ports; requires `protocol`
dstportend	UINT16	End of range of valid destination ports; requires `protocol`

Valid strings for `protocol` are: tcp, udp, udplite, esp, ah, icmp, igmp, sctp

IPv6

Protocol ID: `ipv6` Note: Rules of this type should either go into the `root` or `ipv6` chain.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IPV6_ADDR	Source IPv6 address
srcipmask	IPV6_MASK	Mask applied to source IPv6 address
dstipaddr	IPV6_ADDR	Destination IPv6 address
dstipmask	IPV6_MASK	Mask applied to destination IPv6 address
protocol	UINT8	Layer 4 protocol identifier
srcportstart	UINT16	Start of range of valid source ports; requires `protocol`
srcportend	UINT16	End of range of valid source ports; requires `protocol`
dstportstart	UINT16	Start of range of valid destination ports; requires `protocol`
dstportend	UINT16	End of range of valid destination ports; requires `protocol`

Valid strings for `protocol` are: tcp, udp, udplite, esp, ah, icmpv6, sctp

TCP/UDP/SCTP

Protocol ID: `tcp`, `udp`, `sctp`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	Start of range of source IP address

srcipto	IP_ADDR	End of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
srcportstart	UINT16	Start of range of valid source ports
srcportend	UINT16	End of range of valid source ports
dstportstart	UINT16	Start of range of valid destination ports
dstportend	UINT16	End of range of valid destination ports

ICMP

Protocol ID: `icmp`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	Start of range of source IP address
srcipto	IP_ADDR	End of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
type	UINT16	ICMP type
code	UINT16	ICMP code

IGMP, ESP, AH, UDPLITE, 'ALL'

Protocol ID: `igmp`, `esp`, `ah`, `udplite`, `all`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address

dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	Start of range of source IP address
srcipto	IP_ADDR	End of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address

TCP/UDP/SCTP over IPV6

Protocol ID: `tcp-ipv6`, `udp-ipv6`, `sctp-ipv6`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IPV6_ADDR	Source IP address
srcipmask	IPV6_MASK	Mask applied to source IP address
dstipaddr	IPV6_ADDR	Destination IP address
dstipmask	IPV6_MASK	Mask applied to destination IP address
srcipfrom	IPV6_ADDR	Start of range of source IP address
srcipto	IPV6_ADDR	End of range of source IP address
dstipfrom	IPV6_ADDR	Start of range of destination IP address
dstipto	IPV6_ADDR	End of range of destination IP address
srcportstart	UINT16	Start of range of valid source ports
srcportend	UINT16	End of range of valid source ports
dstportstart	UINT16	Start of range of valid destination ports
dstportend	UINT16	End of range of valid destination ports

ICMPv6

Protocol ID: `icmpv6`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IPV6_ADDR	Source IPv6 address
srcipmask	IPV6_MASK	Mask applied to source IPv6 address
dstipaddr	IPV6_ADDR	Destination IPv6 address
dstipmask	IPV6_MASK	Mask applied to destination IPv6 address
srcipfrom	IPV6_ADDR	Start of range of source IP address
srcipto	IPV6_ADDR	End of range of source IP address
dstipfrom	IPV6_ADDR	Start of range of destination IP address
dstipto	IPV6_ADDR	End of range of destination IP address

type	UINT16	ICMPv6 type
code	UINT16	ICMPv6 code

IGMP, ESP, AH, UDPLITE, 'ALL' over IPv6

Protocol ID: `igmp-ipv6`, `esp-ipv6`, `ah-ipv6`, `udplite-ipv6`, `all-ipv6`
Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to `root`.

Attribute	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IPV6_ADDR	Source IPv6 address
srcipmask	IPV6_MASK	Mask applied to source IPv6 address
dstipaddr	IPV6_ADDR	Destination IPv6 address
dstipmask	IPV6_MASK	Mask applied to destination IPv6 address
srcipfrom	IPV6_ADDR	Start of range of source IP address
srcipto	IPV6_ADDR	End of range of source IP address
dstipfrom	IPV6_ADDR	Start of range of destination IP address
dstipto	IPV6_ADDR	End of range of destination IP address

Command line tools

The libvirt command line tool `virsh` has been extended with life-cycle support for network filters. All commands related to the network filtering subsystem start with the prefix `nwfilter`. The following commands are available:

- nwfilter-list : list UUIDs and names of all network filters
- nwfilter-define : define a new network filter or update an existing one
- nwfilter-undefine : delete a network filter given its name; it must not be currently in use
- nwfilter-dumpxml : display a network filter given its name
- nwfilter-edit : edit a network filter given its name

Example network filters

The following is a list of example network filters that are automatically installed with libvirt.

Name	Description
no-arp-spoofing	Prevent a VM from spoofing ARP traffic; this filter only allows ARP request and reply messages and enforces that those packets contain the MAC and IP addresses of the VM.
allow-dhcp	Allow a VM to request an IP address via DHCP (from any DHCP server)

allow-dhcp-server	Allow a VM to request an IP address from a specified DHCP server. The dotted decimal IP address of the DHCP server must be provided in a reference to this filter. The name of the variable must be *DHCPSERVER*.
no-ip-spoofing	Prevent a VM from sending of IP packets with a source IP address different from the one in the packet.
no-ip-multicast	Prevent a VM from sending IP multicast packets.
clean-traffic	Prevent MAC, IP and ARP spoofing. This filter references several other filters as building blocks.

Note that most of the above filters are only building blocks and require a combination with other filters to provide useful network traffic filtering. The most useful one in the above list is the *clean-traffic* filter. This filter itself can for example be combined with the *no-ip-multicast* filter to prevent virtual machines from sending IP multicast traffic on top of the prevention of packet spoofing.

Writing your own filters

Since libvirt only provides a couple of example networking filters, you may consider writing your own. When planning on doing so there are a couple of things you may need to know regarding the network filtering subsystem and how it works internally. Certainly you also have to know and understand the protocols very well that you want to be filtering on so that no further traffic than what you want can pass and that in fact the traffic you want to allow does pass.

The network filtering subsystem is currently only available on Linux hosts and only works for Qemu and KVM type of virtual machines. On Linux it builds upon the support for `ebtables`, `iptables` and `ip6tables` and makes use of their features. From the above list of supported protocols the following ones are implemented using `ebtables`:

- mac
- arp, rarp
- ip
- ipv6

All other protocols over IPv4 are supported using iptables, those over IPv6 are implemented using ip6tables.

On a Linux host, all traffic filtering instantiated by libvirt's network filter subsystem first passes through the filtering support implemented by ebtables and only then through iptables or ip6tables filters. If a filter tree has rules with the protocols `mac`, `arp`, `rarp`, `ip`, or `ipv6` ebtables rules will automatically be instantiated.
The role of the `chain` attribute in the network filter XML is that internally a new user-defined ebtables table is created that then for example receives all `arp` traffic coming from or going to a virtual machine, if the chain `arp` has been specified. Further, a rule is generated in an interface's `root` chain that directs all ipv4 traffic into the user-defined chain. Therefore, all ARP traffic rules should then be placed into filters specifying this chain. This type of branching into user-define

tables is only supported with filtering on the ebtables layer.
As an example, it is possible to filter on UDP traffic by source and destination ports using the `ip` protocol filter and specifying attributes for the protocol, source and destination IP addresses and ports of UDP packets that are to be accepted. This allows early filtering of UDP traffic with ebtables. However, once an IP or IPv6 packet, such as a UDP packet, has passed the ebtables layer and there is at least one rule in a filter tree that instantiates iptables or ip6tables rules, a rule to let the UDP packet pass will also be necessary to be provided for those filtering layers. This can be achieved with a rule containing an approriate `udp` or `udp-ipv6` traffic filtering node.

Example custom filter

As an example we want to now build a filter that fulfills the following list of requirements:

- prevents a VM's interface from MAC, IP and ARP spoofing
- opens only TCP ports 22 and 80 of a VM's interface
- allows the VM to send ping traffic from an interface but no let the VM be pinged on the interface

The requirement to prevent spoofing is fulfilled by the existing `clean-traffic` network filter, thus we will reference this filter from our custom filter.
To enable traffic for TCP ports 22 and 80 we will add 2 rules to enable this type of traffic. To allow the VM to send ping traffic we will add a rule for ICMP traffic. For simplicity reasons we allow general ICMP traffic to be initated from the VM, not just ICMP echo request and response messages. To then disallow all other traffic to reach or be initated by the VM we will then need to add a rule that drops all other traffic. Assuming our VM is called *test* and the interface we want to associate our filter with is called *eth0*, we name our filter *test-eth0*. The result of these considerations is the following network filter XML:

```
<filter name='test-eth0'>
  <!-- reference the clean traffic filter preventing
       MAC, IP and ARP spoofing. By not providing
       and IP address parameter libvirt will detect the
       IP address the VM is using. -->
  <filterref filter='clean-traffic'/>

  <!-- enable TCP ports 22 (ssh) and 80 (http) to be reachable -->
  <rule action='accept' direction='in'>
    <tcp dstportstart='22'/>
  </rule>

  <rule action='accept' direction='in'>
    <tcp dstportstart='80'/>
  </rule>

  <!-- enable general ICMP traffic to be initiated by the VM;
       this includes ping traffic -->
  <rule action='accept' direction='out'>
    <icmp/>
  </rule>

  <!-- drop all other traffic -->
  <rule action='drop' direction='inout'>
```

```
    <all/>
  </rule>

</filter>
```

Note that none of the rules in the above XML contain the IP address of the VM as either source or destination address, yet the filtering of the traffic works correctly. The reason is that the evaluation of the rules internally happens on a per-interface basis and the rules are evaluated based on the knowledge about which (tap) interface has sent or will receive the packet rather than what their source or destination IP address may be.

An XML fragment for a possible network interface description inside the domain XML of the `test` VM could then look like this:

```
   [...]
    <interface type='bridge'>
      <source bridge='mybridge'/>
      <filterref filter='test-eth0'/>
    </interface>
   [...]
```

To more strictly control the ICMP traffic and enforce that only ICMP echo requests can be sent from the VM and only ICMP echo responses be received by the VM, the above `ICMP` rule can be replaced with the following two rules:

```
  <!-- enable outgoing ICMP echo requests-->
  <rule action='accept' direction='out'>
    <icmp type='8'/>
  </rule>

  <!-- enable incoming ICMP echo replies-->
  <rule action='accept' direction='in'>
    <icmp type='0'/>
  </rule>
```

Limitations

The following sections list (current) limitations of the network filtering subsystem.

IP Address Detection

In case a network filter references the variable *IP* and no variable was defined in any higher layer references to the filter, IP address detection will automatically be started when the filter is to be instantiated (VM start, interface hotplug event). Only IPv4 addresses can be detected and only a single IP address legitimately in use by a VM on a single interface will be detected. In case a VM was to use multiple IP address on a single interface (IP aliasing), the IP addresses would have to be provided explicitly either in the network filter itself or as variables used in attributes' values. These variables must then be defined in a higher level reference to the filter and each assigned the value of the IP address that the VM is expected to be using. Different IP addresses in use by multiple interfaces of a VM (one IP address each) will be independently detected.

Once a VM's IP address has been detected, its IP network traffic may be locked to that address, if for example IP address spoofing is prevented by one of its filters. In that case the user of the VM will not be able to change the IP address on the interface inside the VM, which would be considered IP address spoofing.

In case a VM is resumed after suspension or migrated, IP address detection will be restarted.

VM Migration

VM migration is only supported if the whole filter tree that is referenced by a virtual machine's top level filter is also available on the target host. The network filter *clean-traffic* for example should be available on all libvirt installations of version 0.8.1 or later and thus enable migration of VMs that for example reference this filter. All other custom filters must be migrated using higher layer software. It is outside the scope of libvirt to ensure that referenced filters on the source system are equivalent to those on the target system and vice versa.

Migration must occurr between libvirt insallations of version 0.8.1 or later in order not to loose the network traffic filters associated with an interface.