



Start

- [Log](#)

About

- [Short description](#)
- [Pronunciation](#)
- [Who's behind eatables?](#)
- [Licence](#)
- [About the code](#)

Downloads

- [Latest release](#)
- [CVS repository](#)

Documentation

- [What is eatables?](#)
- [Main features](#)
- [What can it do?](#)
- [What can't it do?](#)
- [Documents](#)
- [Todo](#)
- [Links](#)

Examples

- [Simple examples](#)
- [Real-life examples](#)

Contact

- [Mailing lists](#)
- [Webmaster](#)

Simple examples:

Example filtering configuration:

For the explanation of the various commands I suggest reading the manual page.

```
eatables -P FORWARD DROP
eatables -A FORWARD -p IPv4 -j ACCEPT
eatables -A FORWARD -p ARP -j ACCEPT
eatables -A FORWARD -p LENGTH -j ACCEPT
eatables -A FORWARD --log-level info --log-ip --log-prefix EBFW
eatables -P INPUT DROP
eatables -A INPUT -p IPv4 -j ACCEPT
eatables -A INPUT -p ARP -j ACCEPT
eatables -A INPUT -p LENGTH -j ACCEPT
eatables -A INPUT --log-level info --log-ip --log-prefix EBFW
eatables -P OUTPUT DROP
eatables -A OUTPUT -p IPv4 -j ACCEPT
eatables -A OUTPUT -p ARP -j ACCEPT
eatables -A OUTPUT -p LENGTH -j ACCEPT
eatables -A OUTPUT --log-level info --log-ip --log-arp --log-prefix EBFW -j DROP
```

This is a basic filter configuration which will only let frames made by the protocols IP version 4 and ARP through. Also, the network has some old machines that use the protocol field of the Ethernet frame as a length field (they use the Ethernet 802.2 or 802.3 protocol). There was no reason not to let those machines through, more precisely: there was a reason to let them through ;-). So, those frames, with protocol LENGTH denoting that it's really a length field, are accepted. Of course one could filter on the MAC addresses of those old machines so no other machine can use the old Ethernet 802.2 or 802.3 protocol. All other frames get logged and dropped. This logging consists of the protocol number, the MAC addresses, the ip/arp info (if it's an IP/ARP packet of course) and the in and out interfaces.

Important note:

If you don't absolutely need to let those old machines (using the 802.2 or 803.2 Ethernet protocol) through the bridge, don't let them. Opening it up with the "eatables -A FORWARD -p LENGTH -j ACCEPT" actually breaches security if you're filtering IP bridge traffic with iptables: IP traffic passing the bridge using the 802.2 or 802.3 Ethernet protocol won't get filtered by iptables.

Example to associate IP addresses to MAC addresses:

```
eatables -A FORWARD -p IPv4 --ip-src 172.16.1.4 -s ! 00:11:22:33:44:55 -j DROP
```

This is an anti-spoofing filter rule. It says that the computer using IP address 172.16.1.4 has to be the one that uses ethernet card 00:11:22:33:44:55 to send this traffic.

Note: this can also be done using iptables. In iptables it would look like this:

```
iptables -A FORWARD -s 172.16.1.4 -m mac --mac-source ! 00:11:22:33:44:55 -j DROP
```

The difference is that the frame will be dropped earlier if the eatables rule is used, because eatables inspects the frame before iptables does. Also note the subtle difference in what is considered the default type for a source address: an IP address in iptables, a MAC address in eatables.

[Example NAT \(see the real-life examples page for a usage\):](#)

```
eatables -t nat -A PREROUTING -d 00:11:22:33:44:55 -i eth0 -j dnat --to-destination 54:44:33:22:11:00
```

This will make all frames destined to 00:11:22:33:44:55 that arrived on interface eth0 be transferred to 54:44:33:22:11:00 instead. Note that this does not care about protocols of higher layers. If the network layer is IP f. e. this will probably have not much use: device 54:44:33:22:11:00 will see that the destination IP address is not the same as its own IP address and will probably discard the packet (unless it's a router). If you want to use IP NAT, use iptables.

[A practical example:](#)

This situation was described by someone:

"For a wierd setup (kind of half a half bridge :-)) I would need a generic MAC-source based filter. I need to prevent ARPs and other Layer2 based packets (DEC diag. packets, netbios, etc.) from a specific MAC-source to cross the bridge, to prevent loops."

This is easily solved with eatables:

```
eatables -A FORWARD -s 00:11:22:33:44:55 -p IPV4 -j ACCEPT
eatables -A FORWARD -s 00:11:22:33:44:55 -j DROP
```

[Making a brouter:](#)

Here is an example setup for a brouter with the following situation: br0 with ports eth0 and eth1

```
ifconfig br0 0.0.0.0
```

```

ifconfig eth0 172.16.1.1 netmask 255.255.255.0
ifconfig eth1 172.16.2.1 netmask 255.255.255.0
ebtables -t broute -A BROUTING -p ipv4 -i eth0 --ip-dst 172.16.1.1 -j DROP
ebtables -t broute -A BROUTING -p ipv4 -i eth1 --ip-dst 172.16.2.1 -j DROP
ebtables -t broute -A BROUTING -p arp -i eth0 -d $MAC_OF_ETH0 -j DROP
ebtables -t broute -A BROUTING -p arp -i eth1 -d $MAC_OF_ETH1 -j DROP

```

As mentioned in the man pages, the DROP target in the BROUTING chain actually tells the network code to route the frame instead of bridging it.

The first two ebtables commands are easy to explain: they make sure the IP packets that must be routed enter the IP routing code through the eth0 (resp. eth1) device, not through the br0 device.

The last two commands are needed for a subtle reason. When the router sends an ARP request for, let's say 172.16.1.5, this request is sent through the eth0 device. Without the third ebtables rule, the ARP reply would arrive on the br0 device instead of the eth0 device, as far as the ARP code can tell. This reply is discarded. Using the third rule, the reply arrives on the eth0 device and the ARP code is happy. So the last 2 rules are needed to make the ARP code use the ARP replies. Without the third rule, the router will not send IP packets to 172.16.1.5 (unless it already knew the MAC address of 172.16.1.5 and therefore didn't send an ARP request in the first place).

Using the redirect target:

Here is a simple example that will make all IP traffic entering a (forwarding) bridge port be routed instead of bridged (suppose eth0 is a port of the bridge br0):

```

ebtables -t broute -A BROUTING -i eth0 -p ipv4 -j redirect --redirect-target DROP

```

As mentioned in the man pages, the DROP target in the BROUTING chain actually tells the network code to route the frame. The redirect target will trick the routing code to think the packet was originally destined for the box.

Using the following rule has a similar effect:

```

ebtables -t nat -A PREROUTING --logical-in br0 -p ipv4 -j redirect --redirect-target ACCEPT

```

The difference is that in the second case the IP code and routing code will think the IP packet entered through the br0 device. In the first case the IP code and routing code will think the IP packet entered through the eth0 device. It depends on the situation in which chain to use the redirect target.

Atomically load a complete table into the kernel:

Why do we want to be able to atomically load a complete table with any predefined contents into the kernel? Because then the data is given to the kernel in one step, saving a lot of context switching and kernel time. The

most obvious time to do this is when the box is booted. Here is a brief description how to do this. The examples will use the nat table, of course this works for any table.

The simplest situation is when the kernel table already contains the right data. We can then do the following: First we put the kernel's table into the file "nat_table":

```
ebtables --atomic nat_table -t nat --atomic-save
```

Then we (optionally) zero the counters of the rules in the file:

```
ebtables -t nat --atomic-file nat_table -Z
```

At bootup we use the following command to get everything into the kernel table at once:

```
ebtables --atomic-file nat_table -t nat --atomic-commit
```

We can also build up the complete table in the file. We can use the environment variable \$EBTABLES_ATOMIC_FILE. First we set the environment variable:

```
export EBTABLES_ATOMIC_FILE=nat_table
```

Then we initialize the file with the default table, which has empty chains and policy ACCEPT:

```
ebtables -t nat --atomic-init
```

We then add our rules, user defined chains, change policies:

```
ebtables -t nat -A PREROUTING -j DROP
```

We can check the contents of our table with:

```
ebtables -t nat -L --Lc --Ln
```

We then use the following command to get everything into the kernel table at once:

```
ebtables -t nat --atomic-commit
```

Don't forget to unset the environment variable:

```
unset EBTABLES_ATOMIC_FILE
```

Now all ebtables commands will execute onto the actual kernel table again, instead of on the file nat_table.

Filtering with eatables on interfaces not enslaved to a bridge:

From time to time people ask if it's possible to use eatables on a non-bridge interface. The answer is no. However, if you really need filtering on an interface and can't use a standard way of doing it, there is a solution. We're building up tension here... What follows will inform you what to do.

One use of the following is if you want to do basic Appletalk filtering. With basic is meant any Appletalk filtering that can be done with eatables.

The example below is for a computer that also uses the IP protocol. Obviously, if you only need to filter the IP stuff, just use iptables. The IP protocol is included in this, because it gives an idea of what configuring could be needed for the other protocol (e.g. Appletalk). If the computer does indeed use the IP protocol too, then the following IP stuff will need to be done.

Suppose your current setup consists of device eth0 with IP address 172.16.1.10.

The first three commands make sure eatables will see all traffic entering on eth0, which will be a bridge port of br0.

The other commands are purely IP related.

First make a bridge device:

```
brctl addbr br0
```

Then (perhaps) disable the spanning tree protocol on that bridge:

```
brctl stp br0 off
```

Then add the physical device eth0 to the logical bridge device:

```
brctl addif br0 eth0
```

give the IP address of eth0 to the bridge device and remove it from eth0:

```
ifconfig br0 172.16.1.10 netmask 255.255.255.0  
ifconfig eth0 0.0.0.0
```

The routing table must be corrected too, f.e.:

```
route del -net 172.16.1.0 netmask 255.255.255.0 dev eth0  
route add -net 172.16.1.0 netmask 255.255.255.0 dev br0  
route del default gateway $DEFAULT_GW dev eth0  
route add default gateway $DEFAULT_GW dev br0
```

So, now all IP traffic that originally went through eth0 will go through br0. Note that this is kind of a hack: using a bridge with only one enslaved device. However, now eatables will see all the traffic that passes eth0, because eth0 is now a port of the bridge device br0.

The other protocol used (f.e. Appletalk) will have to be configured to accept traffic from br0 (instead of eth0) and to transmit traffic to br0 (instead of eth0).

Alternatively, this can be used in conjunction with the router functionality. A real-life example for filtering Appletalk, which can be found in the real-life examples section, uses this approach. For performance reasons, it's actually better to use the router approach.

Using the mark match and target:

Say there are 3 types of traffic you want to mark. The best mark values are powers of 2, because these translate to setting one bit in the unsigned long mark value. So, as we have three types of traffic, we will use the mark values 1, 2 and 4.

How do we mark traffic? Simple, filter out the exact traffic you need and then use the mark target. Example: Mark, in the filter table's FORWARD chain, all IP traffic that entered through eth0 with the second mark value; and let later rules have the chance of seeing the frame/packet.

```
eatables -A FORWARD -p ipv4 -i eth0 -j mark --set-mark 2 --mark-target CONTINUE
```

Suppose we want to do something to all frames that are marked with the first mark value:

```
eatables -A FORWARD --mark 1/1
```

Suppose we want to do something to all frames that are marked with either the first, either the third mark value:

```
eatables -A FORWARD --mark /5
```

$1 + 4 = 5$. We only specified the mark mask, which results in taking the logical and of the mark value of the frame with the specified mark mask and comparing the result with 0. So, if the result is non-zero, which means that the mark value is either 1, either 4, either 5, the rule matches.

Note that iptables uses the same unsigned long value for its mark match and target. So this enables 'communication' between eatables and iptables.

Real-life examples:

These example setups were given by eatables users, and they are very much appreciated. If you have a configuration involving eatables which you would like to share, please write a little story about it so it can be put here.

- [Example 1](#): Linux brouting, MAC snat and MAC dnat all in one. By [Enrico Ansaloni](#), who got things working together with [Alessandro Eusebi](#).
- [Example 2](#): Filtering Appletalk on a box not intended to be a bridge. By [Ashok Aiyar](#).
- [Example 3](#): Transparent routing with Freeswan. By [Dominique Blas](#).

