Introduction

For more information about the various commands and options used in these examples, we refer to the manual page.

Feel free to supply additional examples and applications of ebtables into other network configurations. Your contributions are appreciated. You can mail the webmaster to get the additional content online.

When needing a network protocol for an example, we usually use the IP protocol, as it is most common. When appropriate, these examples can be reformulated using a different network protocol (like Appletalk). Unless otherwise stated, we assume a bridge setup consisting of one bridge device br0 with two bridge ports eth0 and eth1.

Basic examples

Choose one of the examples below

You can also view all examples on one page.

Basic filtering configuration:

```
ebtables -P FORWARD DROP
ebtables -A FORWARD -p IPv4 -j ACCEPT
ebtables -A FORWARD -p ARP -j ACCEPT
ebtables -A FORWARD -p LENGTH -j ACCEPT
ebtables -A FORWARD --log-level info --log-ip --log-prefix EBFW
ebtables -P INPUT DROP
ebtables -A INPUT -p IPv4 -j ACCEPT
ebtables -A INPUT -p ARP -j ACCEPT
ebtables -A INPUT -p LENGTH -j ACCEPT
ebtables -A INPUT --log-level info --log-ip --log-prefix EBFW
ebtables -P OUTPUT DROP
ebtables -A OUTPUT -p IPv4 -j ACCEPT
ebtables -A OUTPUT -p ARP -j ACCEPT
ebtables -A OUTPUT -p LENGTH -j ACCEPT
ebtables -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 `ebtables -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 (it's on the todo list).

Associate IP addresses to MAC addresses (anti-spoofing rules):

```
ebtables -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 ebtables rule is used, because ebtables 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 ebtables.

If you have many such rules, you can also use the `among` match to speed up the filtering.

```
ebtables -N MATCHING-MAC-IP-PAIR
ebtables -A FORWARD -p IPv4 --among-dst 00:11:22:33:44:55=172.16.1.4,00:11:33:44:22:55=172.16.1.5 \
-j MATCHING-MAC-IP-PAIR
```

We first make a new user-defined chain `MATCHING-MAC-IP-PAIR` and we send all traffic with matching MAC-IP source address pair to that chain, using the `among` match. The filtering in the `MATCHING-MAC-IP-PAIR` chain can then assume that the MAC-IP source address pairs are correct.

MAC NAT:

See the real-life examples section for an application.

```
ebtables -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. As this change of destination MAC address is done in the `PREROUTING` chain of the `nat` table, it is done before the bridge code makes the forwarding decision. The hosts with addresses 00:11:22:33:44:55 and 54:44:33:22:11:00 therefore don't need to be on the same side of the bridge. If the host with MAC address 54:44:33:22:11:00 is on the same side of the bridge as where the packet arrived, this packet won't be sent out again. You can therefore only use this if the host with the destination MAC address 54:44:33:22:11:00 is on another side of the bridge than the sender of the packet. Note that this MAC NAT does not care about protocols of higher layers. F.e. when the network layer is IP, the host with MAC

ADDRESS 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.

Only forward IPv4 for a specific MAC address:

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 ebtables:

```
ebtables -A FORWARD -s 00:11:22:33:44:55 -p IPV4 -j ACCEPT
ebtables -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
ebtables -t broute -A BROUTING -p arp -i eth0 --arp-ip-dst 172.16.1.1 -j DROP
ebtables -t broute -A BROUTING -p arp -i eth1 --arp-ip-dst 172.16.2.1 -j DROP
```

As mentioned in the man pages, the `DROP` target in the `BROUTING` chain actually broutes the frame. This means the bridge code won't touch the frame and it is sent up to the higher network layers. This results in the frame entering the box as if it

didn't arrive on a bridge port but on the device itself.
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 throught the br0 device. If you want the box to also route traffic with a MAC destination address different from the router's, you need to use the `redirect` target, which changes the MAC destination address to the bridge's MAC address (see the subsequent example).
The last four commands are needed to get ARP working. When the brouter sends an ARP request for, let's say 172.16.1.5, this request is sent through the eth0 or eth1 device (we assume there is no route using output device br0). Without the third ebtables rule, the ARP reply would arrive on the br0 device instead of the eth{0,1} device, as far as the ARP code can tell. This reply is then discarded by the ARP code. Using the third rule, the reply arrives on the eth0 device and the ARP code is happy. So the third and fourth rules are needed to make the ARP code use the ARP replies. Without the third rule, the brouter 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). The last two commands are needed so that the ARP requests for 172.16.1.1 and 172.16.2.1 are answered. You can use more restrictive matching on the ARP packets (e.g. only match on arp requests in the last two rules).

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 broutes the frame. The `redirect` target will trick the network 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. F.e., if your routing table only uses br0, then the redirect belongs in the PREROUTING chain.

Atomically load or update a table:

Why do we want to be able to atomically load or update a table? Because then the table data is given to the kernel in one step. This is sometimes desirable to prevent race conditions when adding multiple rules at once. The most obvious use case, however, is when the tables are initially populated with rules. Committing the table to the kernel at once saves a lot of context switching and kernel time, resulting in much faster configuration. 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-file 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 ebtables on interfaces not enslaved to a bridge:

If you really need filtering on an interface and can't use a standard way of doing it (i.e. there is no standard filtering tool for the protocol), there is a solution if you only need basic filtering.
We consider here the case where basic Appletalk filtering is needed. As there is no Appletalk filter mechanism for Linux, we need something else. 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 ebtables 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 ebtables 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 brouter 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 brouter approach, see the next example to find out why.

Speeding up traffic destinated to the bridge itself:

In some situations the bridge not only serves as a bridge box, but also talks to other hosts. Packets that arrive on a bridge port and that are destinated to the bridge box itself will by default enter the iptables `INPUT` chain with the logical bridge port as input device. These packets will be queued twice by the network code, the first time they are queued after they are received by the network device. The second time after the bridge code examined the destination MAC address and determined it was a locally destinated packet and therefore decided to pass the frame up to the higher protocol stack.

The way to let locally destinated packets be queued only once is by brouting them in the `BROUTING` chain of the `broute` table. Suppose br0 has an IP address and that br0's bridge ports do not have an IP address. Using the following rule

should make all locally directed traffic be queued only once:

```
ebtables -t broute -A BROUTING -d $MAC_OF_BR0 -p ipv4 -j redirect --redirect-target DROP
```

The replies from the bridge will be sent out through the br0 device (assuming your routing table is correct and sends all traffic through br0), so everything keeps working neatly, without the performance loss caused by the packet being queued twice.

The redirect target is needed because the MAC address of the bridge port is not necessarily equal to the MAC address of the bridge device. The packets destinated to the bridge box will have a destination MAC address equal to that of the bridge br0, so that destination address must be changed to that of the bridge port.

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.

```
ebtables -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:

```
ebtables -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:

```
ebtables -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 checking if the result is non-zero. 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 `MARK` target. So this enables communication between ebtables and iptables. Be sure the mark values used in iptables and those used in ebtables don't conflict with each other.

Using `arpreply` for arp requests and letting the arp request populate the arp cache:

The `arpreply` target can only be used in the `PREROUTING` chain of the `nat` table and its default target is `DROP`. A rule like below will therefore prevent an update of the arp cache of the bridge box:

```
ebtables -t nat -A PREROUTING -p arp --arp-opcode Request -j arpreply \
--arpreply-mac 10:11:12:13:14:15
```

This can be fixed by changing the target to `ACCEPT` or `CONTINUE`:

```
ebtables -t nat -A PREROUTING -p arp --arp-opcode Request -j arpreply \
--arpreply-mac 10:11:12:13:14:15 --arpreply-target ACCEPT
```

Changing the destination IP and MAC address to the respective broadcast addresses:

This is not possible in a standard way, but it is possible with some tricks. Suppose you want to direct traffic to 192.168.0.255, then this should work:

```
# suppose there is no route to 192.168.0.0 yet
route add -net 192.168.0.0 netmask 255.255.255.0 dev br0
ifconfig br0 0.0.0.0
arp -s 192.168.0.255 ff:ff:ff:ff:ff:ff
iptables -t nat -A PREROUTING -j DNAT --to-destination 192.168.0.255
```

The bridge device should not have an address in the range of 192.168.0.0/24, because if it does, the routing code won't decide to send the packet out through the bridge device.

Copying packets to userspace for logging:

The `ulog` watcher passes the packet to a userspace logging daemon using netlink multicast sockets. This differs from the log watcher in the sense that the complete packet is sent to userspace instead of a descriptive text and that netlink multicast sockets are used instead of the syslog. This watcher enables parsing of packets with userspace programs. Sending this information to userspace is simple, just use the `ulog` watcher. The physical bridge input and output ports are also included in the netlink messages.

For example, the following rule will send all to be forwarded packets to userspace programs listening on netlink group number 5 before dropping the packets:

```
ebtables -A FORWARD --ulog-nlgroup 5 -j DROP
```

To read the packets sent to userspace, a program needs to be written. Under `examples/ulog/` in the ebtables directory you can find a working example in C that looks for ICMP echo requests and replies. See `INSTALL` for compilation.

Gustavo J. A. M. Carneiro <gjc_at_inescporto.pt> has written some Python code to be able to look at the data using Python code. These are the files he released: ebtulogmodule_CC.c and ulog_GC.c.

Enable support for running the ebtables tool concurrently:

Updating the ebtables kernel tables is a two-phase process. First, the userspace program sends the new table to the kernel and receives the packet counters for the rules in the old table. In a second phase, the userspace program uses these counter values to determine the initial counter values of the new table, which is already active in the kernel. These values are sent to the kernel which adds these values to the kernel's counter values. Due to this two-phase process, it is possible to confuse the ebtables userspace tool when more than one instance is run concurrently. Note that even in a one-phase process it would be possible to confuse the tool.

The ebtables tool supports an option named `--concurrent`, which makes the ebtables tool first acquire a lock on a specific file before reading and updating the kernel tables. As long as you make sure all ebtables processes run with this option enabled, there is no problem in running ebtables processes concurrently. If your firewall scripts can run concurrently, make sure to enable this option.

An alternative would be to use the tool `flock` in your script.

When the ebtables process crashes unexpectedly while holding the file lock, subsequent execution of the program will fail to acquire the lock. In this case, you will need to explicitly delete the lock file: `/var/lib/ebtables/lock`.

Closing IP security holes with multiple networks:

When the bridge is configured to allow iptables or ip6tables to filter bridged traffic, care must be taken in order to prevent unforeseen security holes. The iptables chains are traversed for all IP packets on all bridges. If there is more than one bridge, you must make sure that packets forwarded by different bridges don't interfere in iptables rules. One simple way to avoid this, is by using different IP subnets for each bridged network. This is not always possible, of course. A similar concern arrises when you allow iptables to filter bridged IP traffic encapsulated in a vlan or pppoe header.

The multi-bridge scenario is especially a potential problem for connection tracking, since this doesn't consider the input and output interfaces. The vlan/pppoe scenario is also a potential problem for all other IP traffic, since iptables itself isn't aware of the vlan id or pppoe session id.

It is possible, however, to let iptables indirectly know these details by using the mark target of ebtables. In the example below, we use netfilter's connection tracking zone mechanism to separate connection tracking between the bridged vlan traffic with vlan id 1 and 5.

```
# set up the connection tracking zones
iptables -t raw -A PREROUTING -m mark --mark 1 -j CT --zone 1
iptables -t raw -A PREROUTING -m mark --mark 2 -j CT --zone 2
# mark packets according to the vlan id
ebtables -t nat -A PREROUTING -p 802_1Q --vlan-id 1 -j mark --mark-set 1
ebtables -t nat -A PREROUTING -p 802_1Q --vlan-id 5 -j mark --mark-set 2
```