Published on *Linux Journal* (http://www.linuxjournal.com)

Kernel Korner - Linux as an Ethernet Bridge

By *Jim Robinson*
Created *2005-05-26 02:00*

Have you ever been asked to secure a router over which you did not have administrative control? What about when you are on a network you don't own but want to secure the segment are you using? A request similar to this one is what brought me to the wonderful world of Bridge, the Linux Ethernet bridging project.

According to the Bridge Web site:

> Ethernet bridging is a way to connect networks together to form a larger network. The standard for bridging is ANSI/IEEE 802.1d. A bridge is a way to connect two separate network segments together in a protocol-independent way. Packets are forwarded based on Ethernet address, rather than IP address (like a router). Since forwarding is done at Layer 2, all protocols can go transparently through a bridge.

The code currently is maintained by Stephen Hemminger for both the Linux 2.4 and 2.6 kernels. Most modern distributions using the 2.6 series kernel have the bridging code built in. For the purposes of this article, we are using Fedora Core 3, which is built on the 2.6 kernel. If you're stuck with the 2.4 kernel, don't despair. Kernel patches are available on the Bridge site (see the on-line Resources), so you can play too.

The firewall component of the bridging firewall is achieved by using another related project called ebtables. The ebtables program is a filtering layer for a bridging firewall. The filtering connects into the Link Layer Ethernet frame field. In addition to filtering, you also may manipulate the Ethernet MAC addresses. The ebtables code also allows iptables rules to function in bridging mode, giving you both IP- and MAC-level filters for your firewall.

What Is a Bridge?

A bridge is a device that links two or more network segments that use the same network technologies. The topologies may differ, though, so you can go from fiber to copper, but the technologies must remain the same. In its most simple form, think of a Linux hub. Add as many ports to the box as you want, and they all become part of the single hub device. What comes in one port goes out all of the other ports in the hub fabric, unless you state otherwise in the rules. Once your hub is up, you can use iptables and ebtables to filter traffic as you would any other Linux forwarding system.

Getting Started

We start out simply by attempting to achieve connectivity between a simple two-NIC machine. When we are finished, this Linux box should act as a standard hub, passing traffic from one port to another as needed. When we plug one NIC in to our regular network jack and a laptop into the second NIC, we will be able to use the network from the laptop as if we were connected directly.

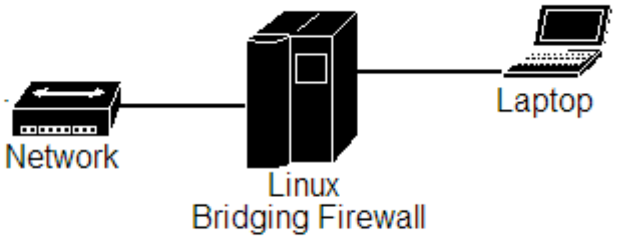

Figure 1. In this simple network, the Linux system acts like an Ethernet hub, passing all traffic.

We want this bridge to be transparent to any device plugged in to it. Interestingly enough, beyond the ability to connect remotely to the bridge to maintain it and check logs, there is no requirement to give the bridge an IP address. Of course, in today's connected world it makes sense to assign an IP address and we do so here.

I started with an old box that has been waiting for a project such as this. It's an AMD K6-450 with 256MB of RAM. It has a single 15GB IDE hard drive and a single 3Com 10/100MB Ethernet card. I also had a spare 3Com 10/100MB Ethernet card that works well with Linux, so it is added as the second interface. I am going to run only the bridge software, some simple firewall rules and perhaps Snort for intrusion detection. The traffic volumes are low and I don't expect massive amounts of Snort data, so 256MB of RAM should suffice. If you're going to be passing gigabit traffic and want to sniff live, ramp up the specs of the machine considerably.

Now install Fedora Core 3, selecting the extras you feel are needed. If you work in high-security environments, I recommend keeping your software options to the bare minimum. You always can grab extras later with YUM if you forget something. For now, simply get a working Linux install going and make sure that it finds your network cards. You need the kernel source and usual compile utilities to make the ebtables code, so add those in. Remember to stay secure and remove any software you don't need once you place the device into production. Once the install completes, reboot and log in as root.

Now you are ready to create a virtual network device. You can call it whatever you want; I went with br0—the first bridge device:

```
#> brctl addbr br0
```

Run `ifconfig`. Do you see your network interfaces (Listing 1)?

Listing 1. Before configuring the network, check that both Ethernet interfaces are up.

```
#> ifconfig

eth0      Link encap:Ethernet  HWaddr 00:CC:D0:99:EB:26
          inet6 addr: fe80::2b0:d0ff:fe99:eb26/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:86208855 errors:0 dropped:0 overruns:63 frame:0
          TX packets:77098217 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3871506445 (3692.1 Mb)  TX bytes:266311184 (253.9 Mb)
          Interrupt:5 Base address:0xec00
```

```
eth1      Link encap:Ethernet  HWaddr 00:CC:03:D8:3A:1A
          inet6 addr: fe80::201:3ff:fed8:3a1a/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:77087614 errors:0 dropped:0 overruns:0 frame:0
          TX packets:85110321 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:264995582 (252.7 Mb)  TX bytes:3672580334 (3502.4 Mb)
          Interrupt:9 Base address:0xec80
```

In Listing 1, you can see that we have two network cards with no IP addresses bound to them. If you have IP addresses assigned to the interface, remove them for simplicity's sake. On Fedora, edit the file /etc/sysconfig/networking-scripts/ifcfg-X, where X is the card identifier. On my system, the two interfaces are eth0 and eth1. Delete or comment out the lines that relate to the IP address. It is important to make sure the cards are on at boot time. Listing 2 shows a basic configuration that should work. Don't forget to reinitialize networking once you've completed the above, using `service network reload`.

Listing 2. Two Simple Config Files for Network Cards with No IP Addresses

```
/etc/sysconfig/networking-scripts/ifcfg-eth0:

DEVICE=eth0
ONBOOT=yes
BOOTPROTO=static


/etc/sysconfig/networking-scripts/ifcfg-eth1:

DEVICE=eth1
ONBOOT=yes
BOOTPROTO=static
```

Next, tell the system what devices belong to this group, as shown below. Also, give the command that actually initializes the virtual device, as shown in the last line:

```
#> brctl addif br0 eth0
#> brctl addif br0 eth1
#> ip link set br0 up
```

In its most basic form, your Linux box now is acting like a hub. For the keen ones, you can plug in the Ethernet adapters and begin to play. The box itself, however, currently is passing traffic blindly and does not have an IP address assigned to it. I like to be able to connect remotely to my devices after I install them, so I am going to add an IP address and some routing information to the virtual device br0.

To add an IP address to the bridge interface, issue:

```
#> ip addr add 10.1.1.18/16 brd + dev br0
```

I had to state both the subnet mask (/16) and which bridge device it should be assigned to. This becomes important if you have more than one virtual device on the machine. I have only the one, but the syntax requires it. If you named your bridge device something else, you need to state that explicitly here.

The last thing to do before you can play with your bridge remotely is to configure the routing:

```
#> route add default gw 10.1.1.1 dev br0
```

The usual routing rules and commands apply, and for all intents and purposes you can use the device (br0) as you would any other Linux network interface.

Testing

Now that we have everything in place, let's test it out. First, let's confirm that all of our configurations have taken hold:

```
#> brctl show
bridge name  bridge id          STP enabled  interfaces
br0          8000.0030843e5aa2  no           eth0
                                             eth1
```

As you can see above, we have a single bridge device called br0 that uses interfaces eth0 and eth1. This confirms that we should be in business.

Installation

Now it's time to do the physical setup. Connect one network card to your network switch as you would normally do for any other computer. You should see link lights on both ends of the link. Connect a desktop or laptop to the other interface on your Linux box using a crossover cable. Wait for the link lights to come on, count to ten and ping another node on your network from your desktop or laptop. You should be able to use the network on the other side of the Linux hub as if it was attached directly.

Surviving a Reboot

How you set up your install to survive a reboot is your choice. A simple way is to add all of the commands we have used to /etc/rc.local, which is processed at the end of startup. Enter the commands used above to this file, and your bridge is functional after startup.

Firewalling

As with any Linux install that passes or forwards traffic, you have the ability to filter the stream of information as it passes by. A bridging firewall is no different. There are many ways to create and maintain firewall configurations. Below, I explain how to use the most basic firewall type: deny all, pass some. We want to deny everything passing this firewall unless we specifically state that something is allowed.

This firewall configuration requires you to download and install the ebtables user-space tools available from the ebtables Web site (see Resources). At the time of this writing, the latest release was v2.0.6. Grab a copy of this from one of the many mirrors. Do the usual extract and install dance without the initial configure step:

```
#> tar -xzf ebtables-v2.0.6.tar.gz
#> cd ebtables-v2.0.6
#> make
#> install
```

If all goes well, you should have the ebtables command set at your fingertips. Test this by typing `ebtables` at the prompt; you should see something similar to this:

```
#> ebtables -V
ebtables v2.0.6 (November 2003)
```

Let's start by making sure iptables is set to accept. Remember we're on Fedora Core 3, so we simply can tell the service to quit, which does the same thing:

```
#> service iptables stop
#> chkconfig --level 35 iptables off
```

You can do something similar by issuing the flush command. List your available chains and then flush each of them in turn:

```
#> iptables -L
#> iptables -F INPUT
#> iptables -F OUTPUT
#> iptables -F FORWARD
#> iptables -F RH-Firewall-1-INPUT
```

Now we want to stop all traffic from all areas of our network from passing through the firewall. The following rules are specific to the network we're working with for this example; you need to amend the subnets or hosts to reflect your specific requirements:

```
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.2.0.0/16 -j DROP
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.7.0.0/16 -j DROP
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.4.0.0/16 -j DROP
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.5.0.0/16 -j DROP
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.6.0.0/16 -j DROP
/sbin/ebtables -A FORWARD -p IPv4 \
--ip-source 10.1.0.0/16 -j DROP
```

Those of you familiar with iptables should notice that the syntax above is similar. We tell the ebtables program that when FORWARDING using the IPv4 protocol to DROP any packets sourced from the 10.1.0.0/16 subnet. We then tell it to repeat for the rest of the subnets.

The next step is to allow the device behind the firewall itself. If you do not allow its IP address to pass through, nothing works. Also, if you assign an IP address to the firewall itself, don't forget to allow it as well:

```
/sbin/ebtables -I FORWARD 1 -p IPv4 \
--ip-source 10.1.1.5 -j ACCEPT
/sbin/ebtables -I FORWARD 1 -p IPv4 \
--ip-source 10.1.1.18 -j ACCEPT
```

Here, I add the devices on my network that are allowed to access my laptop:

```
/sbin/ebtables -I FORWARD 1 -p IPv4 \
--ip-source 10.1.10.30 -j ACCEPT
/sbin/ebtables -I FORWARD 1 -p IPv4 \
--ip-source 10.1.10.19 -j ACCEPT
/sbin/ebtables -I FORWARD 1 -p IPv4 \
--ip-source 10.1.10.87 -j ACCEPT
```

To test this, I simply go to a machine listed in the ACCEPT rules above and see if I can ping my laptop at 10.1.1.5. Now move to a node not listed above—no pings for you!

Real-World Implementation

Recently, I was called to a customer's site to secure a financial server. The request was simple: we need a firewall in front of this system but we cannot change its IP address. With two NICs and a Linux OS, I was able to have a working firewall up and running in a few minutes. Installation also was a breeze. I simply used a crossover cable that connected the firewall to the server and a regular cable from the other network card on the firewall to the network jack. That was it. No redesign was necessary of any part of the existing IP scheme; it truly was plug-and-play. Once a few rules were in place to drop all packets unless they were from the IP addresses and ports listed as acceptable, the project was completed.

One of the beautiful aspects of Linux is its ability to run many services on one system. Take the above example. I quickly firewalled a sensitive server, but that was not the end of the project. With all the extra time and money we saved using Linux, we were able to load Snort on the firewall. With a quick hack to the sniffer's config file—/etc/snort.conf in our case—we told Snort to listen to interface br0, and snort immediately began to do its stuff on the bridging interface.

This is where the true power of the bridging code can be felt. Ever had a segment of the network running slow but you don't know why? Next time, load a Linux box with Snort and any other sleuthing software you like and get the bridge up and running. Find your trusty crossover cable and head out to the site. Because the bridge acts like a hub, you simply can insert your Linux box at any point in the network. As long as you have the physical connections, you can drop your box in and begin to sniff live in a matter of seconds. The latest project we have been working on included transparent Squid cache servers that are truly transparent requiring zero reconfiguration to the IP scheme, clients or browsers. Simply insert the Squid box in front of the router and redirect all port 80 traffic to the box itself and you're done.

The ability of Linux to slide transparently into existing network infrastructure opens a world of new and improved services that the penguin can provide. With the ability to place dissimilar networking devices into one virtual entity, you can use a single device to firewall and monitor any aspect of your network. You're only limitation is the speed of your hardware and its number of available slots.

Acknowledgements

The author sends kudos to the guys and girls that wrote the Bridging code and for the authors of the ebtables command set for producing stable, usable tools and releasing them under the GPL.

Resources for this article: /article/8261 [1].

Jim Robinson is President of Linux Solutions Provider, Inc., a consulting company based in Macon, Georgia. He enjoys being a husband and a dad, playing guitar and, of course, using Linux.

Webmaster

Source URL: http://www.linuxjournal.com/article/8172

Links:
[1] http://www.linuxjournal.com/article/8261