Linux Bridge+Firewall Mini−HOWTO version 1.2.0

Table of Contents

Linux Bridge+Firewall Mini−HOWTO version 1.2.0

Peter Breuer (ptb@it.uc3m.es)

v, 19 December 1997

1.Introduction

You should look at the original Bridging mini−HOWTO by Chris Cole for a different perspective on this. He is chris@polymer.uakron.edu. The version of his HOWTO that I have based this document on (alternatively, ripped off) is 1.03 dated Aug 23 1996.

2.What and Why (and How?)

2.1 What

A bridge is an intelligent connecting wire betwen two network cards. A firewall is an intelligent insulator.

2.2 Why

You might want a bridge if you have several computers:

1. to save the price of a new hub when you just happen to have an extra ethernet card available.
2. to save the bother of learning how to do IP−forwarding and other tricks when you _have_ two cards in your computer.
3. to avoid maintenance work in the future when things change around!

``Several computers" might be as few as three if those are routing or bridging or just moving around the room from time to time! You also might want a bridge just for the fun of finding out what it does. 2 was what I wanted a bridge for.

If you are really interested in 1, you have to be one of the very few. Check the NET−2−HOWTO and the Serial−HOWTO for better tricks.

You want a firewall if

1. you are trying to protect your network from external accesses, or
2. you are trying to deny access to the world outside from your network.

Curiously, I needed 2 here too. Policy at my university presently is that we should not act as internet service providers to undergraduates.

2.3 How?

I started out bridging the network cards in a firewalling machine and ended up firewalling without having cut the bridge. It seems to work and is more flexible than either configuration alone. I can take down the firewall and keep bridging or take down the bridge when I want to be more circumspect.

I would guess that the bridge code lives just above the physical device layer and the firewalling code lives one layer higher up, so that the bridging and firewalling configurations effectively act as though they are running connected together ``in sequence'' and not ``in parallel'' (ouch!). Diagram:

```
    -> Bridge-in -> Firewall-in -> Kernel -> Firewall-out -> Bridge-out ->
```

There is no other way to explain how one machine can be a ``conductor'' and an ``insulator'' at the same time. There are a few caveats but I'll come to those later. Basically you must route packets that you want to firewall. Anyway, it all seems to work together nicely for me. Here is what you do ...

3.BRIDGING

3.1 Software

Get the bridge configuration utility from Alan Cox's home pages. This is the same reference as in Chris' document. I just didn't realize that it was an ftp and not an http URL ...

3.2 Prior Reading.

Read the Multiple Ethernet HOWTO for some advice on getting more than one network card recognized and configured.

Yet more details of the kind of boot magic that you may need are in the Boot Prompt HOWTO.

You may be able to get away without the NET−2 HOWTO. It is a good long read and you will have to pick from it the details you need.

3.3 Boot configuration

The reading material above will tell you that you need to prepare the kernel to recognize a second ethernet device at boot up by adding this to your **/etc/lilo.conf**, and then re−run **lilo**:

```
append = "ether=0,0,eth1"
```

Note the "eth1". "eth0" is the first card. "eth1" is the second card. You can always add the boot parameters in your response to the line that lilo offers you. This is for three cards:

```
linux ether=0,0,eth1 ether=0,0,eth2
```

I use **loadlin** to boot my kernel from DOS:

```
loadlin.exe c:\vmlinuz root=/dev/hda3 ro ether=0,0,eth1 ether=0,0,eth2
```

Note that this trick makes the kernel probe at bootup. That will not happen if you load the ethernet drivers as **modules** (for safety since the probe order can't be determined) so if you use modules you will have to add the appropriate IRQ and port parameters for the driver in your **/etc/conf.modules**. I have at least

```
alias eth0 3c509
alias eth1 de620
options 3c509 irq=5 io=0x210
options de620 irq=7 bnc=1
```

You can tell if you use modules by using ``ps −aux" to see if **kerneld** is running and checking that there are .o files in a subdirectory of your **/lib/modules** directory. You want the directory named with what uname −r tells you. If you have kerneld and/or you have a foo.o then edit **/etc/conf.modules** and read the man page for depmod carefully.

Note also that until recently (kernel 2.0.25) the **3c509** driver could not be used for more than one card if used as a module. I have seen a patch floating around that fixes the oversight. It may be in the kernel when you read this.

3.4 Kernel configuration

Recompile the kernel with bridging enabled.

```
        CONFIG_BRIDGE=y
```

I also compiled with firewalling and IP−forwarding and −masquerading and the rest enabled. Only if you want firewalling too ...

```
        CONFIG_FIREWALL=y
        CONFIG_NET_ALIAS=y
        CONFIG_INET=y
        CONFIG_IP_FORWARD=y
        CONFIG_IP_MULTICAST=y
        CONFIG_IP_FIREWALL=y
        CONFIG_IP_FIREWALL_VERBOSE=y
        CONFIG_IP_MASQUERADE=y
```

You don't need all of this. What you do need apart from this is the standard net configuration:

```
        CONFIG_NET=y
```

and I do not think you need worry about any of the other networking options. I have any options that I did not actually compile into the kernel available through kernel modules that I can add in later.

Install the new kernel in place, rerun lilo and reboot with the new kernel. Nothing should have changed at this point!

3.5 Network addresses

Chris says that a bridge should not have an IP address but that is not the setup to be described here.

You are going to want to use the machine for connecting to the net so you need an address and you need to make sure that you have the loopback device configured in the normal way so that your software can talk to the places they expect to be able to talk to. If loopback is down the name resolver or other net sevices might fail. See the NET−2−HOWTO, but your standard configuration should already have done this bit:

```
        ifconfig lo 127.0.0.1 route add −net 127.0.0.0
```

You will have to give addresses to your network cards. I altered the /etc/rc.d/rc.inet1 file in my slackware (3.x) to setup two cards and you should also essentially just look for your net configuration file and double or treble the number of instructions in it. Suppose that you already have an address at

```
        192.168.2.100
```

(that is in the private net reserved address space, but never mind − it won't hurt anybody if you use this address by mistake) then you probably already have a line like

```
        ifconfig eth0 192.168.2.100 netmask 255.255.255.0 metric 1
```

in your configuration. The first thing you are going to probably want to do is cut the address space reached by this card in half so that you can eventually bridge or firewall the two halves. So add a line which reduces the mask to address a smaller number of machines:

```
        ifconfig eth0 netmask 255.255.255.128
```

Try it too. That restricts the card to at most the address space between .0 and .127.

Now you can set your second card up in the other half of the local address space. Make sure that nobody already has the address. For symmetry I set it at 228=128+100 here. Any address will do so long as it is not in the other card's mask, and even then, well, maybe. Avoid special addresses like .0, .1, .128 etc. unless you really know what you are doing.

```
        ifconfig eth1 192.168.2.228 netmask 255.255.255.128 metric 1
```

That restricts the second card to addresses between .128 and .255.

3.6 Network routing

This is where I have to announce the caveats in the bridging + firewalling scheme: you cannot firewall packets which are not routed. No routes, no firewall. At least this appears to be true in the 2.0.30 and more recent kernels. The firewalling filters are closely involved with the ip−forwarding code.

That does not mean that you cannot bridge. You can bridge between two cards and firewall them from a third. You can have only two cards and firewall both of them against an outside IP such as a nearby router, provided that the router is routed by you to exactly one of the cards.

In other words, since I will be doing firewalling, so I want to precisely control the physical destination of some packets.

I have the small net of machines attached to a hub hanging off eth0, so I configure a net there:

```
        route add −net 192.168.2.128 netmask 255.255.255.128 dev eth0
```

The 128 would be 0 if I had a full class C network there. I don't, by definition, since I just halved the address space. The "dev eth0" is not necessary here because the cards address falls within the mask, but it may be necessary for you. One might need more than one card holding up this subnet (127 machines on one segment, oh yeah) but those cards would be being bridged under the same netmask so that they appear as one to the routing code.

On the other card I have a line going straight through to a big router that I trust.

```
                                             client 129
         __                                        |    __
client 1   \    .0                    .128         |   /   net 1
client 2 --- Hub − eth0 − Kernel − eth1 − Hub − Router --- net 2
client 3 __/       .100            .228         .2 |   \__ net 3
                                                   |
                                             client 254
```

I attach the address of the router to that card as a fixed ("static") route because it would otherwise fall within the first cards netmask and the kernel would be thinking wrongly about how to send packets to the big router. I will want to firewall these packets and that is another reason fow wanting to route them specifically.

```
        route add 192.168.2.2 dev eth1
```

I don't need it, since I don't have any more machines in that half of the address space, but I declare a net also on the second card. Separating my interfaces into two sets via routing will allow me to do very tight firewalling eventually , but you can get away with far less routing than this.

```
route add −net 192.168.2.128 netmask 255.255.255.128 dev eth1
```

I also need to send all non−local packets out to the world so I tell the kernel to send them to the big router

```
route add default gw 192.168.2.2
```

3.7 Card configuration

So much was standard networking setup, but we are bridging so we also have to listen on both (?) cards for packets that are not aimed at us. The following should go into the network configuration file.

```
ifconfig promisc eth0 ifconfig promisc eth1
```

The man page says allmulti=promisc, but it didn't work for me.

3.8 Additional routing

One thing that I noticed was that I had to put at least the second card into a mode where it would respond to the big router's questions about which machines I was hiding in my local net.

```
ifconfig arp eth1
```

For good measure I did this to the other card too.

```
ifconfig arp eth0.
```

3.9 Bridge Configuration

Put bridging enabling on and into your configuration file:

```
brcfg −enable
```

You should have been trying this out in real time all along, of course! The bridge configure will bring up some numbers. You can experiment with turning on and off the ports one at a time

```
brcfg −port 0 −disable/−enable
 brcfg −port 1 −disable/−enable
```

You get status reports anytime by just running

```
brcfg
```

without any parameters. You will see that the bridge listens,learns, and then does forwarding. (I don't understand why the code repeats the same hardware addresses for both my cards, but never mind .. Chris' howto say that is OK)

3.10 Try it out

If you are still up and running as things are, try out your configuration script for real by taking down both cards and then executing it:

```
ifconfig eth0 down ifconfig eth1 down /etc/rc.d/rc.inet1
```

With any luck the various subsystems (**nfs**, **ypbind**, etc.) won't notice. *Do not try this unless you are sitting at the keyboard!*

If you want to be more careful than this, you should take down as many daemons as possible beforehand, and unmount nfs directories. The worst that can happen is that you have to reboot in single−user mode (the "**single**" parameter to **lilo** or **loadlin**), and take out your changes before rebooting with things the way they were before you started.

3.11 Checks

Verify that there is different traffic on each interface:

```
tcpdump −i eth0
(in one window)

tcpdump −i eth1
(in another window)
```

You should get used to using **tcpdump** to look for things that should not be happening or that are happening and should not.

For instance look for packets that have gone through the bridge to the second card from the internal net. Here I am looking for packets from the machine with address .22:

```
tcpdump −i eth1 −e host 192.168.2.22
```

Then send a ping from the .22 host to the router. You should see the packet reported by tcpdump.

At this stage you should have a bridge ready that also has two network addreses. Test that you can ping them from outside and inside your local net, and that you can telnet and ftp around between inside and outside too.

4.FIREWALLING

4.1 Software and reading

You should read the Firewall−HOWTO.

That will tell you where to get **ipfwadm** if you don't already have it. There are other tools you can get but I made no progress until I tried **ipfwadm**. It is nice and low level! You can see exactly what it is doing.

4.2 Preliminary checks

You have compiled IP−forwarding and masquerading into the kernel so you will want to check that the firewall is in its default (accepting) state with

```
ipfwadm −I −l ipfwadm −O −l ipfwadm −F −l
```

That is respectively, "display the rules affecting the .." incoming or outgoing or forwarding (masquerading) ".. sides of the firewall". The "−l" means "list".

You might have compiled in accounting too:

```
ipfwadm −A −l
```

You should see that there are no rules defined and that the default is to accept every packet. You can get back to this working state anytime with

```
ipfwadm −I −f
ipfwadm −O −f
ipfwadm −F −f
```

The "−f" means "flush". You may need to use that.

4.3 Default rule

I want to cut the world off from my internal net and do nothing else, so I will want to give as a last (default) rule that the firewall should ignore any packets coming in from the internal net and directed to outside. I put all the rules (in this order) into **/etc/rc.d/rc.firewall** and execute it from **/etc/rc.d/rc.local** at bootup.

```
ipfwadm −I −a reject −S 192.168.2.0/255.255.255.128 −D 0.0.0.0/0.0.0.0
```

The "−S" is the source address/mask. The "−D" is the destination address/mask.

This format to is rather long−winded. **Ipfwadm** is intelligent about network names and some common abbreviations. Check the man pages.

It is possibly more convenient to put some or all of these rules on the outgoing half of the firewall by using "−O" instead of "−I", but I'll state the rules here all formulated for the incoming half.

4.4 Holes per address

Before that default rule, I have to place some rules that serve as exceptions to this general denial of external services to internal clients.

I want to treat the firewall machines address on the internal net specially. I will stop people logging in to the firewall machine unless they have special permission, but once they are there they should be allowed to talk to the world.

```
ipfwadm −I −i accept −S 192.168.2.100/255.255.255.255 \
 −D 0.0.0.0/0.0.0.0
```

I also want the internal clients to be able to talk to the firewalling machine. Maybe they can persuade it to let them get out!

```
ipfwadm −I −i accept −S 192.168.2.0/255.255.255.128 \
 −D 192.168.2.100/255.255.255.255
```

Check at this point that you can get in to the clients from outside the firewall via **telnet**, but that you cannot get out. That should mean that you can just about make first contact, but the clients cannot send you any prompts. You should be able to get all the way in if you use the firewall machine as a staging post. Try **rlogin** and **ping** too, with **tcpdump** running on one card or the other. You should be able to make sense of what you see.

4.5 Holes per protocol

I went on to relax the rules protocol by protocol. I want to allow pings from the outside to the inside to get an echo back, for instance, so I inserted the rule:

```
ipfwadm −I −i accept −P icmp −S 192.168.2.0/255.255.255.128 \
 −D 0.0.0.0/0.0.0.0
```

The "`−P icmp`" works the protocol−specific magic.

Until I get hold of an **ftp** proxy I am also allowing ftp calls out with port−specific relaxations. This targets ports 20 21 and 115 on outside machines.

```
ipfwadm −I −i accept −P tcp −S 192.168.2.0/255.255.255.128 \
 −D 0.0.0.0/0.0.0.0 20 21 115
```

I could not make **sendmail** between the local clients work without a nameserver. Rather than set up a nameserver right then on the firewall, I just lifted the firewall for tcp domain service queries precisely aimed at the nearest existing nameserver and put its address in the clients **/etc/resolv.conf** ("`nameserver 123.456.789.31`" on a separate line).

```
ipfwadm −I −i accept −P tcp −S 192.168.2.0/255.255.255.128 \
 −D 123.456.789.31/255.255.255.255 54
```

You can find which port number and protocol a service requires with **tcpdump**. Trigger the service with a an **ftp** or a **telnet** or whatever to or from the internal machine and then watch for it on the input and output ports of the firewall with **tcpdump**:

```
tcpdump −i eth1 −e host client04
```

for example. The **/etc/services** file is another important source of clues. To let **telnet** and **ftp** IN to the firewall from outside, you have to allow the local clients to call OUT on a specific port. I understand why this is necessary for **ftp** − it's the server that establishes the data stream in the end − but I am not sure why **telnet** also needs this.

```
ipfwadm −I −i accept −P tcp −S 192.168.2.0/255.255.255.128 ftp telnet \
 −D 0.0.0.0/0.0.0.0
```

There is a particular problem with some daemons that look up the hostname of the firewalling machine in order to decide what is their networking address. **Rpc.yppasswdd** is the one I had trouble with. It insists on broadcasting information that says it is outside the firewall (on the second card). That means the clients inside can't contact it.

Rather than start IP aliasing or change the daemon code, I mapped the name to the inside card address on the clients in their **/etc/hosts**.

4.6 Checks

You want to test that you can still **telnet**, **rlogin** and **ping** from the outside. From the inside you should be able to **ping** out. You should also be able to **telnet** to the firewall machine from the inside and the latter should be able to do anything.

That is it. At this point you probably want to learn about **rpc**/**Y**ellow **P**ages and the interaction with the password file. The firewalled network wants to run without its unprivileged users being able to log on to the firewall − and thus get out. Some other HOWTO!
