Starting a Linux Firewall from Scratch

Dinil Divakaran

Abstract

The first steps in getting started with iptables.

Building a firewall is something that easily can be done using a Linux machine. This article describes the basic steps involved in developing a firewall from scratch, using tools in Linux. It is intended for newbies interested in learning about (Linux) firewalls. More important, this article is for all new administrators who would like to dirty their hands and get a firewall up and running as soon as possible, but without missing the important concepts en route. My experience in working on a Linux-based firewall at the DON (Distributed and Optical Networking) lab, in the department of Computer Science and Engineering at the Indian Institute of Technology (IIT) Madras, is the most motivating factor behind writing this article.

In this article, we examine developing a firewall that will sit on the edge, separating your private network from the rest of the world; therefore, the firewall also will act as a gateway.

![Firewall Diagram]

Figure 1. Firewall Diagram

First of all, why do you need such a firewall? Most important, you need to restrict access to machines in
your network, a network that might consist of various servers. One of them might be a mail server, and another might be a DNS server, but only those particular services (provided by these servers) need to be accessed, not anything and everything on the network. Putting it simply, firewalls are used to protect a private network from the rest of the world—call it a public network (which is the Internet in most scenarios).

One less obvious reason for having a firewall is that it is necessary to block all unwanted traffic flowing into or through your network, which might otherwise throttle the bandwidth. Such traffic should ideally stop at the gate (gateway or firewall). One good example is when there are many subnetworks, such as at a college or university campus. One of the machines in such a subnetwork could become infected with a virus and might flood or broadcast ARP packets. Similarly, some Windows PCs from outside the private network might be broadcasting netbios (netbios-ns/netbios-dgm) packets, which are meaningless to your network and, therefore, should be blocked by the firewall.

But, some of the ARP packets might be legitimate requests for machines in your network (or subnet). If you block such legitimate ARP broadcast requests, no packet (good or bad) will reach your network, as machines outside the private network will not be able to obtain the Ethernet address corresponding to the IP address of the machine in your network. To solve this problem, you should configure your firewall to act as a proxy for ARP requests—that is, your firewall should reply to the ARP requests.

Now, let's get into the implementation details. Assume your private network is 192.168.9.0/24. Your firewall, which is also a gateway, must have two interfaces: one pointing to your network (eth0) and the other connecting to the public network (eth1).

First, configure the IPs for both interfaces. This can be done using the network configuration tool or with the ifconfig command. Ideally, it is better to use the system network configuration tool (system-config-network in Fedora Core 2–5) or edit the configuration files (at /etc/sysconfig/network-scripts in FC 2–5), so that the configurations are retained even when the network is started (as part of the boot process) or restarted (manually). You also can configure the IP by appending the ifconfig command at the end of /etc/rc.d/rc.local (as this file is executed at the end of the boot process). If you do this, however, ensure that these commands are executed when the network is restarted manually.

We use ifconfig to be distribution-independent (for lack of a better term).

There is no hard and fast rule on the IP addresses to be used for the interfaces, but generally, the last two IP addresses in the subnet are used for such purposes. Now, assign 192.168.9.253 to eth0 and 192.168.9.254 to eth1:

```bash
echo "Configuring eth0"
/sbin/ifconfig eth0 192.168.9.253 up

echo "Configuring eth1"
/sbin/ifconfig eth1 192.168.9.254 up
```

The most important function of a firewall that takes the role of a gateway is to forward packets. This is how we do it:

```bash
echo "Enabling IP forwarding"
echo "1" > /proc/sys/net/ipv4/ip_forward
```
Earlier, we said the firewall also should act as a proxy for ARP requests. This means the firewall will reply to the ARP requests querying for the Ethernet address of any machines in your network (192.168.9.0/24). Will the firewall send the MAC address of the machine for which the query was broadcasted (say 192.168.9.8)? No. Instead, it will send its own MAC address, and later, when it receives a packet for 192.168.9.8, it will forward the packet to 192.168.9.8 (of course, only if the rules allow the packet to pass through). Enabling proxy ARP is quite easy in new distributions:

```bash
echo "Enabling Proxy ARP"
```
```bash
echo "1" > /proc/sys/net/ipv4/conf/eth1/proxy_arp
```

Next, set up the routing entries in the firewall. The private network is reachable through eth0, although packets to the public network should go through eth1:

```bash
echo "Route to 192.168.9.0/24 is through eth0"
/sbin/route add -net 192.168.9.0/24 eth0
```
```bash
echo "The default gateway is eth1"
/sbin/route add default eth1
```

Similarly, you have to tell all machines in your network to use 192.168.9.253 as the default gateway (because you have to go through the gateway to access any machine outside your network). LAN machines can be accessed directly. Do the following on all machines (except the firewall, obviously) in your network:

```bash
echo "Add default route through the gateway"
/sbin/route add default gw 192.168.9.253 eth0
```
```bash
echo "192.168.9.0/24 is directly reachable"
/sbin/route add -net 192.168.9.0/24 eth0
```

Next comes the firewall rules—rules that protect a network. Rules are written using the `iptables` tool. This is a very useful tool, although a bit complex, with a detailed man page on the various options. The `iptables` Netfilter uses three different built-in chains: INPUT, FORWARD and OUTPUT. Packets traverse through the chains, and therefore, the rules are written for specific chains. With respect to your firewall, any packet destined to your firewall (192.168.9.253 or 192.168.9.254) goes to the INPUT chain. If the packet is meant to be forwarded (that is, it is not for your firewall, and there is a route in your firewall to the destination), it goes through the FORWARD chain. Any packet generated by your firewall will go out from the box through the OUTPUT chain. (This brief explanation is applicable to any Linux box.)

Although you would never want the firewall to forward every packet passing through it, you might want to test whether the functionality of the gateway is working with the above configuration. To do this, make the default policy of the FORWARD chain as ACCEPT (using the `-P` option)—that is, any packet going through the forward chain is accepted:

```bash
/sbin/iptables -P FORWARD ACCEPT
```

A ping request from any machine in the network 192.168.9.0/24 (save, the firewall) to any (live) machine outside the network will now return with the ICMP echo reply packet. If the external machine is not reachable, there may be some problem with the cable or network card, or you might have misconfigured something.

Now, let's build the “wall”. The easiest way of setting up a firewall is by rejecting (DROP) every kind of...
packet, and then writing rules to allow (ACCEPT) those packets that you want to see go through. So, let's make the default policy in each of the chains to drop packets. Before doing that, clear all the existing rules:

```
echo "Flush existing rules"
/sbin/iptables -F
```

```
echo "Set the default policy to drop packets"
/sbin/iptables -P INPUT DROP
/sbin/iptables -P OUTPUT DROP
/sbin/iptables -P FORWARD DROP
```

By now, you might have noticed that a rule basically specifies some conditions that the packet must possess. If these conditions are matched, the action specified in the rule is taken, or else the next rule in the chain is checked, and this continues until a rule is matched. If none of the rules in the chain is matched, the default action or policy (here, DROP) is taken.

Let's write our first rule—a rule to allow outgoing SSH packets from the private network:

```
echo "Allow outgoing SSH"
/sbin/iptables -A FORWARD -p TCP -i eth0 \
  -s 192.168.9.0/24 -d 0/0 --dport 22 -j ACCEPT
```

This rule is self-explanatory—well, almost. The option -A specifies the chain to which the rule is to be appended, and -p specifies the protocol (UDP, TCP, ICMP and so on). The option -i names the interface through which the packets will be received. Because the packets are coming from the 192.168.9.0/24 network (the -s specifies the source address) for outgoing SSH packets, it will come through eth0 of the firewall. The destination port (--dport) is 22 for SSH traffic. The destination address is indicated with the -d option, and 0/0 means any address. Finally, the action for such packets that are matched is ACCEPT (specified with the -j option), which means allow the matched packets to go through.

Now, we have written a rule to allow SSH traffic from 192.168.9.0/24 to go anywhere. But, will this work? Will you be able to do an SSH logon from your private network to a machine in the public network? Where have we allowed packets to come from the SSH server (in the public network) back to the client (in the private network)? The following rule achieves that:

```
/sbin/iptables -A FORWARD -p TCP -i eth1 -s 0/0 \ 
  --sport 22 -d 192.168.9.0/24 -j ACCEPT
```

This looks fine, but then we need to write such a rule for every service. Worse, the above rule does more than what is required. It allows any machine to connect to the private network using the source port 22. What we should do instead is append a rule that allows only those packets from the public network that are part of the SSH connections initiated by machines in the 192.168.9.0/24 network.

```
/sbin/iptables -A FORWARD -p TCP -i eth1 -s 0/0 \ 
  --sport 22 -d 192.168.9.0/24 -j ACCEPT
```

This looks fine, but then we need to write such a rule for every service. Worse, the above rule does more than what is required. It allows any machine to connect to the private network using the source port 22. What we should do instead is append a rule that allows only those packets from the public network that are part of the SSH connections initiated by machines in the 192.168.9.0/24 network.

If you would like to learn exactly how the connection tracking mechanism works. Now (after removing the above rule), to forward all those packets forming part of the ESTABLISHED connection, we write the following rule:

```
```

iptables maintains state information to do such connection tracking. The four states maintained are NEW, ESTABLISHED, RELATED and INVALID. We won't discuss these states in detail here. For the time being, keep in mind that state NEW indicates the packet is part of a new connection. When a response packet is seen in the reverse direction, the connection becomes ESTABLISHED. Note that this has nothing to do with the states in the TCP connection establishment process. An ICMP or UDP reply for the corresponding requests also will mark the connection as ESTABLISHED. Refer to http://iptables-tutorial.frozentux.net/iptables-tutorial.html#STATEMACHINE to learn exactly how the connection tracking mechanism works. Now (after removing the above rule), to forward all those packets forming part of the ESTABLISHED connection, we write the following rule:
This rule ensures that only packets part of an ESTABLISHED connection will be accepted; a new connection request to 192.168.9.0/24 will *not* be accepted. Ideally, to access any services (such as HTTP or FTP), we need to allow only NEW and ESTABLISHED connections to go out (NEW will allow the first packet, ESTABLISHED will allow all following packets of the same connection), and only ESTABLISHED connections to come into the private network. Similarly, if you have a DNS server in your network, which has to be permitted access (queried) from the outside, the following rule does that (assuming that 198.168.9.1 is the DNS server):

```
echo "Allowing incoming DNS requests"
/sbin/iptables -A FORWARD -p TCP -i eth1 \ 
   -d 198.168.9.1 --dport 53 -j ACCEPT
```

Note that the interface used here is eth1, as the packets from the public network will be received at eth1. (We have not used -s 0/0, as it is added by default.) Also, keep in mind that DNS lookup will succeed only because we already have appended the rule for allowing ESTABLISHED connections to the FORWARD list (yes, UDP traffic also has an associated ESTABLISHED state).

So far, we have blocked every protocol except SSH and DNS. It is a common practice for a new system administrator to block ICMP packets. This is not a good idea, as ICMP packets are useful for many purposes, such as for learning the routes between different interconnected networks in a large LAN, to see if a machine is up, for Path MTU discovery and so on. So, assuming we are sensible administrators, let's allow ICMP packets through the firewall:

```
echo "Allowing ICMP packets"
/sbin/iptables -A FORWARD -p ICMP -j ACCEPT
```

Earlier, we had blocked any packet to and from the firewall box (using INPUT and OUTPUT chains). For diagnostic purposes, we can allow ICMP packets through both chains—that is, allow ICMP packets to and from the firewall:

```
echo "Allowing ICMP packets to the firewall"
/sbin/iptables -A INPUT -p ICMP -j ACCEPT
```

```
echo "Allowing ICMP packets from the firewall"
/sbin/iptables -A OUTPUT -p ICMP -j ACCEPT
```

The ICMP packets also can be rate-limited (as a precaution against ICMP-based attacks):

```
echo "Limit ICMP requests to 5 per second"
/sbin/iptables -A FORWARD -p icmp --icmp-type echo-request -m limit --limit 5/s -j ACCEPT
```

We also might choose to ignore ping broadcasts—that is, ICMP packets to broadcast addresses, such as ping 192.168.9.255 (ICMP broadcast requests are used in Smurf attacks):

```
echo "Ignoring ICMP broadcast requests"
echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

All these rules (commands) will be lost once the system is rebooted; however, iptables has options for saving and restoring these rules. But, a better approach is to save the rules in a file (say, firewall.sh), give it executable permission and append the script name to the end of /etc/rc.d/rc.local. This way, you
always can edit and make modifications to the firewall script.